Texture Fetch Instruction

| | Cycle 0 | | | | Cycle 1 | | | | **Cycle 2** | | | | **Cycle 3** | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | 96 | RD | bit |
| 0 | OPCODE | 0 | 0 | 0 | DST_SEL_W | 1 | 1 | 10 | USE_COMP_LOD | 0 | 1 | 28 | unused | 5 | 2 | 14 | 1 | 0 | 5 |
| 1 | OPCODE | 1 | 0 | 1 | DST_SEL_W | 1 | 1 | 11 | USE_REG_LOD | 0 | 1 | 29 | unused | 6 | 2 | 15 | 1 | 0 | 6 |
| 2 | OPCODE | 2 | 0 | 2 | MAG_FILTER | 0 | 1 | 12 | USE_REG_LOD | 1 | 1 | 30 | OFFSET_X | 0 | 2 | 16 | 1 | 0 | 7 |
| 3 | OPCODE | 3 | 0 | 3 | MAG_FILTER | 1 | 1 | 13 | unused | - | 1 | 31 | OFFSET_X | 1 | 2 | 17 | 1 | 0 | 8 |
| 4 | OPCODE | 4 | 0 | 4 | MIN_FILTER | 0 | 1 | 14 | USE_REG_GRADIENTS | 0 | 2 | 0 | OFFSET_X | 2 | 2 | 18 | 1 | 0 | 9 |
| 5 | FETCH_VALID_ONLY | 0 | 0 | 19 | MIN_FILTER | 1 | 1 | 15 | SAMPLE_LOCATION | 0 | 2 | 1 | OFFSET_X | 3 | 2 | 19 | 1 | 0 | 10 |
| 6 | TX_COORD_DENORM | 0 | 0 | 25 | MIP_FILTER | 0 | 1 | 16 | LOD_BIAS | 0 | 2 | 2 | OFFSET_X | 4 | 2 | 20 | 1 | 0 | 11 |
| 7 | SAMPLE_LOCATION | 0 | 0 | 26 | MIP_FILTER | 1 | 1 | 17 | LOD_BIAS | 1 | 2 | 3 | OFFSET_Y | 0 | 2 | 21 | 1 | 0 | 12 |
| 8 | DST_SEL_X | 0 | 1 | 0 | ANISO_FILTER | 0 | 1 | 18 | LOD_BIAS | 2 | 2 | 4 | OFFSET_Y | 1 | 2 | 22 | 1 | 0 | 13 |
| 9 | DST_SEL_X | 1 | 1 | 1 | ANISO_FILTER | 1 | 1 | 19 | LOD_BIAS | 3 | 2 | 5 | OFFSET_Y | 2 | 2 | 23 | 1 | 0 | 14 |
| 10 | DST_SEL_X | 2 | 1 | 2 | ANISO_FILTER | 2 | 1 | 20 | LOD_BIAS | 4 | 2 | 6 | OFFSET_Y | 3 | 2 | 24 | 1 | 0 | 15 |
| 11 | DST_SEL_Y | 0 | 1 | 3 | ARBITRARY_FILTER | 0 | 1 | 21 | LOD_BIAS | 5 | 2 | 7 | OFFSET_Y | 4 | 2 | 25 | 1 | 0 | 16 |
| 12 | DST_SEL_Y | 1 | 1 | 4 | ARBITRARY_FILTER | 1 | 1 | 22 | LOD_BIAS | 6 | 2 | 8 | OFFSET_Z | 0 | 2 | 26 | 1 | 0 | 17 |
| 13 | DST_SEL_Y | 2 | 1 | 5 | ARBITRARY_FILTER | 2 | 1 | 23 | unused | 0 | 2 | 9 | OFFSET_Z | 1 | 2 | 27 | 1 | 0 | 18 |
| 14 | DST_SEL_Z | 0 | 1 | 6 | VOL_MAG_FILTER | 0 | 1 | 24 | unused | 1 | 2 | 10 | OFFSET_Z | 2 | 2 | 28 | 1 | 0 | 20 |
| 15 | DST_SEL_Z | 1 | 1 | 7 | VOL_MAG_FILTER | 1 | 1 | 25 | unused | 2 | 2 | 11 | OFFSET_Z | 3 | 2 | 29 | 1 | 0 | 21 |
| 16 | DST_SEL_Z | 2 | 1 | 8 | VOL_MIN_FILTER | 0 | 1 | 26 | unused | 3 | 2 | 12 | OFFSET_Z | 4 | 2 | 30 | 1 | 0 | 22 |
| 17 | DST_SEL_W | 0 | 1 | 9 | VOL_MIN_FILTER | 1 | 1 | 27 | unused | 4 | 2 | 13 | unused | - | unused | - | 1 | 0 | 23 |
| | | | | | | | | | | | | | | | | | 1 | 0 | 26 |
| | | | | | | | | | | | | | | | | | 1 | 0 | 27 |

Vertex Fetch Instruction

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | | 1 | 0 | 28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | 1 | 0 | 29 |
| 0 | OPCODE | 0 | 0 | 0 | DST_SEL_W | 1 | 1 | 10 | EXP_ADJUST_ALL | 4 | 1 | 28 | OFFSET_X | 6 | 2 | 14 | 1 | 0 | 30 |
| 1 | OPCODE | 1 | 0 | 1 | DST_SEL_W | 1 | 1 | 11 | EXP_ADJUST_ALL | 5 | 1 | 29 | OFFSET_X | 7 | 2 | 15 | 1 | 0 | 31 |
| 2 | OPCODE | 2 | 0 | 2 | FORMAT_COMP_ALL | 0 | 1 | 12 | unused | - | 1 | 30 | OFFSET_X | 8 | 2 | 16 | 1 | 1 | 31 |
| 3 | OPCODE | 3 | 0 | 3 | NUM_FORMAT_ALL | 0 | 1 | 13 | unused | - | 1 | 31 | OFFSET_X | 9 | 2 | 17 | 1 | 2 | 31 |
| 4 | OPCODE | 4 | 0 | 4 | SIGNED_RF_MODE_ALL | 0 | 1 | 14 | STRIDE | 0 | 2 | 0 | OFFSET_X | 10 | 2 | 18 | 70 | | |
| 5 | FETCH_VALID_ONLY | 0 | 0 | 19 | INDEX_ROUND | 0 | 1 | 15 | STRIDE | 1 | 2 | 1 | OFFSET_X | 11 | 2 | 19 | | | |
| 6 | CONST_INDEX_SEL | 0 | 0 | 25 | DATA_FORMAT | 0 | 1 | 16 | STRIDE | 2 | 2 | 2 | OFFSET_X | 12 | 2 | 20 | | | |
| 7 | CONST_INDEX_SEL | 1 | 0 | 26 | DATA_FORMAT | 1 | 1 | 17 | STRIDE | 3 | 2 | 3 | OFFSET_X | 13 | 2 | 21 | | | |
| 8 | DST_SEL_X | 0 | 1 | 0 | DATA_FORMAT | 2 | 1 | 18 | STRIDE | 4 | 2 | 4 | OFFSET_X | 14 | 2 | 22 | | | |
| 9 | DST_SEL_X | 1 | 1 | 1 | DATA_FORMAT | 3 | 1 | 19 | STRIDE | 5 | 2 | 5 | OFFSET_X | 15 | 2 | 23 | | | |
| 10 | DST_SEL_X | 2 | 1 | 2 | DATA_FORMAT | 4 | 1 | 20 | STRIDE | 6 | 2 | 6 | OFFSET_X | 16 | 2 | 24 | | | |
| 11 | DST_SEL_Y | 0 | 1 | 3 | DATA_FORMAT | 5 | 1 | 21 | STRIDE | 7 | 2 | 7 | OFFSET_X | 17 | 2 | 25 | | | |
| 12 | DST_SEL_Y | 1 | 1 | 4 | unused | - | 1 | 22 | OFFSET_X | 0 | 2 | 8 | OFFSET_X | 18 | 2 | 26 | | | |
| 13 | DST_SEL_Y | 2 | 1 | 5 | unused | - | 1 | 23 | OFFSET_X | 1 | 2 | 9 | OFFSET_X | 19 | 2 | 27 | | | |
| 14 | DST_SEL_Z | 0 | 1 | 6 | EXP_ADJUST_ALL | 0 | 1 | 24 | OFFSET_X | 2 | 2 | 10 | OFFSET_X | 20 | 2 | 28 | | | |
| 15 | DST_SEL_Z | 1 | 1 | 7 | EXP_ADJUST_ALL | 1 | 1 | 25 | OFFSET_X | 3 | 2 | 11 | OFFSET_X | 21 | 2 | 29 | | | |
| 16 | DST_SEL_Z | 2 | 1 | 8 | EXP_ADJUST_ALL | 2 | 1 | 26 | OFFSET_X | 4 | 2 | 12 | OFFSET_X | 22 | 2 | 30 | | | |
| 17 | DST_SEL_W | 0 | 1 | 9 | EXP_ADJUST_ALL | 3 | 1 | 27 | OFFSET_X | 5 | 2 | 13 | unused | - | unused | - | | | |

Texture Fetch Constant Fields

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit |
| 0 | unused | - | 3 | 0 | BASE_ADDRESS | 4 | 4 | 16 | NUM_FORMAT_ALL | 0 | 6 | 0 | LOD_BIAS | 4 | 7 | 16 |
| 1 | unused | - | 3 | 1 | BASE_ADDRESS | 5 | 4 | 17 | DST_SEL_X | 0 | 6 | 1 | LOD_BIAS | 5 | 7 | 17 |
| 2 | FORMAT_COMP_X | 0 | 3 | 2 | BASE_ADDRESS | 6 | 4 | 18 | DST_SEL_X | 1 | 6 | 2 | LOD_BIAS | 6 | 7 | 18 |
| 3 | FORMAT_COMP_X | 1 | 3 | 3 | BASE_ADDRESS | 7 | 4 | 19 | DST_SEL_X | 2 | 6 | 3 | LOD_BIAS | 7 | 7 | 19 |
| 4 | FORMAT_COMP_Y | 0 | 3 | 4 | BASE_ADDRESS | 8 | 4 | 20 | DST_SEL_Y | 0 | 6 | 4 | LOD_BIAS | 8 | 7 | 20 |
| 5 | FORMAT_COMP_Y | 1 | 3 | 5 | BASE_ADDRESS | 9 | 4 | 21 | DST_SEL_Y | 1 | 6 | 5 | LOD_BIAS | 9 | 7 | 21 |
| 6 | FORMAT_COMP_Z | 0 | 3 | 6 | BASE_ADDRESS | 10 | 4 | 22 | DST_SEL_Y | 2 | 6 | 6 | GRAD_EXP_ADJUST_H | 0 | 7 | 22 |
| 7 | FORMAT_COMP_Z | 1 | 3 | 7 | BASE_ADDRESS | 11 | 4 | 23 | DST_SEL_Z | 0 | 6 | 7 | GRAD_EXP_ADJUST_H | 1 | 7 | 23 |
| 8 | FORMAT_COMP_W | 0 | 3 | 8 | BASE_ADDRESS | 12 | 4 | 24 | DST_SEL_Z | 1 | 6 | 8 | GRAD_EXP_ADJUST_H | 2 | 7 | 24 |
| 9 | FORMAT_COMP_W | 1 | 3 | 9 | BASE_ADDRESS | 13 | 4 | 25 | DST_SEL_Z | 2 | 6 | 9 | GRAD_EXP_ADJUST_H | 3 | 7 | 25 |
| 10 | CLAMP_X | 0 | 3 | 10 | BASE_ADDRESS | 14 | 4 | 26 | DST_SEL_W | 0 | 6 | 10 | GRAD_EXP_ADJUST_H | 4 | 7 | 26 |
| 11 | CLAMP_X | 1 | 3 | 11 | BASE_ADDRESS | 15 | 4 | 27 | DST_SEL_W | 1 | 6 | 11 | GRAD_EXP_ADJUST_V | 0 | 7 | 27 |
| 12 | CLAMP_X | 2 | 3 | 12 | BASE_ADDRESS | 16 | 4 | 28 | DST_SEL_W | 2 | 6 | 12 | GRAD_EXP_ADJUST_V | 1 | 7 | 28 |
| 13 | CLAMP_Y | 0 | 3 | 13 | BASE_ADDRESS | 17 | 4 | 29 | EXP_ADJUST_ALL | 0 | 6 | 13 | GRAD_EXP_ADJUST_V | 2 | 7 | 29 |
| 14 | CLAMP_Y | 1 | 3 | 14 | BASE_ADDRESS | 18 | 4 | 30 | EXP_ADJUST_ALL | 1 | 6 | 14 | GRAD_EXP_ADJUST_V | 3 | 7 | 30 |
| 15 | CLAMP_Y | 2 | 3 | 15 | BASE_ADDRESS | 19 | 4 | 31 | EXP_ADJUST_ALL | 2 | 6 | 15 | GRAD_EXP_ADJUST_V | 4 | 7 | 31 |
| 16 | CLAMP_Z | 0 | 3 | 16 | SIZE | 0 | 5 | 0 | EXP_ADJUST_ALL | 3 | 6 | 16 | BORDER_COLOR | 0 | 8 | 0 |
| 17 | CLAMP_Z | 1 | 3 | 17 | SIZE | 1 | 5 | 1 | EXP_ADJUST_ALL | 4 | 6 | 17 | BORDER_COLOR | 1 | 8 | 1 |
| 18 | CLAMP_Z | 2 | 3 | 18 | SIZE | 2 | 5 | 2 | EXP_ADJUST_ALL | 5 | 6 | 18 | FORCE_BC_W_TO_MAX | 0 | 8 | 2 |
| 19 | SIGNED_RF_MODE_ALL | 0 | 3 | 19 | SIZE | 3 | 5 | 3 | MAG_FILTER | 0 | 6 | 19 | TRI_JUICE | 0 | 8 | 3 |
| 20 | DIM | 0 | 3 | 20 | SIZE | 4 | 5 | 4 | MAG_FILTER | 1 | 6 | 20 | TRI_JUICE | 1 | 8 | 4 |
| 21 | DIM | 1 | 3 | 21 | SIZE | 5 | 5 | 5 | MIN_FILTER | 0 | 6 | 21 | unused | - | 8 | 5 |
| 22 | PITCH | 0 | 3 | 22 | SIZE | 6 | 5 | 6 | MIN_FILTER | 1 | 6 | 22 | unused | - | 8 | 6 |
| 23 | PITCH | 1 | 3 | 23 | SIZE | 7 | 5 | 7 | MIP_FILTER | 0 | 6 | 23 | unused | - | 8 | 7 |
| 24 | PITCH | 2 | 3 | 24 | SIZE | 8 | 5 | 8 | MIP_FILTER | 1 | 6 | 24 | unused | - | 8 | 8 |
| 25 | PITCH | 3 | 3 | 25 | SIZE | 9 | 5 | 9 | ANISO_FILTER | 0 | 6 | 25 | unused | - | 8 | 9 |
| 26 | PITCH | 4 | 3 | 26 | SIZE | 10 | 5 | 10 | ANISO_FILTER | 1 | 6 | 26 | unused | - | 8 | 10 |
| 27 | PITCH | 5 | 3 | 27 | SIZE | 11 | 5 | 11 | ANISO_FILTER | 2 | 6 | 27 | MIP_PACKING | 0 | 8 | 11 |
| 28 | PITCH | 6 | 3 | 28 | SIZE | 12 | 5 | 12 | ARBITRARY_FILTER | 0 | 6 | 28 | MIP_ADDRESS | 0 | 8 | 12 |
| 29 | PITCH | 7 | 3 | 29 | SIZE | 13 | 5 | 13 | ARBITRARY_FILTER | 1 | 6 | 29 | MIP_ADDRESS | 1 | 8 | 13 |
| 30 | PITCH | 8 | 3 | 30 | SIZE | 14 | 5 | 14 | ARBITRARY_FILTER | 2 | 6 | 30 | MIP_ADDRESS | 2 | 8 | 14 |
| 31 | TILED | 0 | 3 | 31 | SIZE | 15 | 5 | 15 | BORDER_SIZE | 0 | 6 | 31 | MIP_ADDRESS | 3 | 8 | 15 |
| 32 | DATA_FORMAT | 0 | 4 | 0 | SIZE | 16 | 5 | 16 | VOL_MAG_FILTER | 0 | 7 | 0 | MIP_ADDRESS | 4 | 8 | 16 |
| 33 | DATA_FORMAT | 1 | 4 | 1 | SIZE | 17 | 5 | 17 | VOL_MIN_FILTER | 0 | 7 | 1 | MIP_ADDRESS | 5 | 8 | 17 |
| 34 | DATA_FORMAT | 2 | 4 | 2 | SIZE | 18 | 5 | 18 | MIN_MIP_LEVEL | 0 | 7 | 2 | MIP_ADDRESS | 6 | 8 | 18 |
| 35 | DATA_FORMAT | 3 | 4 | 3 | SIZE | 19 | 5 | 19 | MIN_MIP_LEVEL | 1 | 7 | 3 | MIP_ADDRESS | 7 | 8 | 19 |
| 36 | DATA_FORMAT | 4 | 4 | 4 | SIZE | 20 | 5 | 20 | MIN_MIP_LEVEL | 2 | 7 | 4 | MIP_ADDRESS | 8 | 8 | 20 |
| 37 | DATA_FORMAT | 5 | 4 | 5 | SIZE | 21 | 5 | 21 | MIN_MIP_LEVEL | 3 | 7 | 5 | MIP_ADDRESS | 9 | 8 | 21 |
| 38 | ENDIAN_SWAP | 0 | 4 | 6 | SIZE | 22 | 5 | 22 | MAX_MIP_LEVEL | 0 | 7 | 6 | MIP_ADDRESS | 10 | 8 | 22 |
| 39 | ENDIAN_SWAP | 1 | 4 | 7 | SIZE | 23 | 5 | 23 | MAX_MIP_LEVEL | 1 | 7 | 7 | MIP_ADDRESS | 11 | 8 | 23 |
| 40 | REQUEST_SIZE | 0 | 4 | 8 | SIZE | 24 | 5 | 24 | MAX_MIP_LEVEL | 2 | 7 | 8 | MIP_ADDRESS | 12 | 8 | 24 |
| 41 | REQUEST_LATENCY | 0 | 4 | 9 | SIZE | 25 | 5 | 25 | MAX_MIP_LEVEL | 3 | 7 | 9 | MIP_ADDRESS | 13 | 8 | 25 |
| 42 | unused | - | 4 | 10 | SIZE | 26 | 5 | 26 | MAG_ANISO_WALK | 0 | 7 | 10 | MIP_ADDRESS | 14 | 8 | 26 |
| 43 | NEAREST_CLAMP_POLICY | 0 | 4 | 11 | SIZE | 27 | 5 | 27 | MIN_ANISO_WALK | 0 | 7 | 11 | MIP_ADDRESS | 15 | 8 | 27 |
| 44 | BASE_ADDRESS | 0 | 4 | 12 | SIZE | 28 | 5 | 28 | LOD_BIAS | 0 | 7 | 12 | MIP_ADDRESS | 16 | 8 | 28 |
| 45 | BASE_ADDRESS | 1 | 4 | 13 | SIZE | 29 | 5 | 29 | LOD_BIAS | 1 | 7 | 13 | MIP_ADDRESS | 17 | 8 | 29 |
| 46 | BASE_ADDRESS | 2 | 4 | 14 | SIZE | 30 | 5 | 30 | LOD_BIAS | 2 | 7 | 14 | MIP_ADDRESS | 18 | 8 | 30 |
| 47 | BASE_ADDRESS | 3 | 4 | 15 | SIZE | 31 | 5 | 31 | LOD_BIAS | 3 | 7 | 15 | MIP_ADDRESS | 19 | 8 | 31 |

Vertex Fetch Constant Fields

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit |
| 0 | unused | - | 3 | 0 | SIZE | 14 | 4 | 16 | ENDIAN_SWAP | 0 | 6 | 0 | BASE_ADDRESS | 14 | 7 | 16 |
| 1 | unused | - | 3 | 1 | SIZE | 15 | 4 | 17 | ENDIAN_SWAP | 1 | 6 | 1 | BASE_ADDRESS | 15 | 7 | 17 |
| 2 | BASE_ADDRESS | 0 | 3 | 2 | SIZE | 16 | 4 | 18 | SIZE | 0 | 6 | 2 | BASE_ADDRESS | 16 | 7 | 18 |
| 3 | BASE_ADDRESS | 1 | 3 | 3 | SIZE | 17 | 4 | 19 | SIZE | 1 | 6 | 3 | BASE_ADDRESS | 17 | 7 | 19 |
| 4 | BASE_ADDRESS | 2 | 3 | 4 | SIZE | 18 | 4 | 20 | SIZE | 2 | 6 | 4 | BASE_ADDRESS | 18 | 7 | 20 |
| 5 | BASE_ADDRESS | 3 | 3 | 5 | SIZE | 19 | 4 | 21 | SIZE | 3 | 6 | 5 | BASE_ADDRESS | 19 | 7 | 21 |
| 6 | BASE_ADDRESS | 4 | 3 | 6 | SIZE | 20 | 4 | 22 | SIZE | 4 | 6 | 6 | BASE_ADDRESS | 20 | 7 | 22 |
| 7 | BASE_ADDRESS | 5 | 3 | 7 | SIZE | 21 | 4 | 23 | SIZE | 5 | 6 | 7 | BASE_ADDRESS | 21 | 7 | 23 |
| 8 | BASE_ADDRESS | 6 | 3 | 8 | SIZE | 22 | 4 | 24 | SIZE | 6 | 6 | 8 | BASE_ADDRESS | 22 | 7 | 24 |
| 9 | BASE_ADDRESS | 7 | 3 | 9 | SIZE | 23 | 4 | 25 | SIZE | 7 | 6 | 9 | BASE_ADDRESS | 23 | 7 | 25 |
| 10 | BASE_ADDRESS | 8 | 3 | 10 | CLAMP_X | 0 | 4 | 26 | SIZE | 8 | 6 | 10 | BASE_ADDRESS | 24 | 7 | 26 |
| 11 | BASE_ADDRESS | 9 | 3 | 11 | BORDER_COLOR | 0 | 4 | 27 | SIZE | 9 | 6 | 11 | BASE_ADDRESS | 25 | 7 | 27 |
| 12 | BASE_ADDRESS | 10 | 3 | 12 | REQUEST_SIZE | 0 | 4 | 28 | SIZE | 10 | 6 | 12 | BASE_ADDRESS | 26 | 7 | 28 |
| 13 | BASE_ADDRESS | 11 | 3 | 13 | REQUEST_SIZE | 1 | 4 | 29 | SIZE | 11 | 6 | 13 | BASE_ADDRESS | 27 | 7 | 29 |
| 14 | BASE_ADDRESS | 12 | 3 | 14 | CLAMP_DISABLE | 0 | 4 | 30 | SIZE | 12 | 6 | 14 | BASE_ADDRESS | 28 | 7 | 30 |
| 15 | BASE_ADDRESS | 13 | 3 | 15 | unused | - | 4 | 31 | SIZE | 13 | 6 | 15 | BASE_ADDRESS | 29 | 7 | 31 |
| 16 | BASE_ADDRESS | 14 | 3 | 16 | unused | - | 5 | 0 | SIZE | 14 | 6 | 16 | ENDIAN_SWAP | 0 | 8 | 0 |
| 17 | BASE_ADDRESS | 15 | 3 | 17 | unused | - | 5 | 1 | SIZE | 15 | 6 | 17 | ENDIAN_SWAP | 1 | 8 | 1 |
| 18 | BASE_ADDRESS | 16 | 3 | 18 | BASE_ADDRESS | 0 | 5 | 2 | SIZE | 16 | 6 | 18 | SIZE | 0 | 8 | 2 |
| 19 | BASE_ADDRESS | 17 | 3 | 19 | BASE_ADDRESS | 1 | 5 | 3 | SIZE | 17 | 6 | 19 | SIZE | 1 | 8 | 3 |
| 20 | BASE_ADDRESS | 18 | 3 | 20 | BASE_ADDRESS | 2 | 5 | 4 | SIZE | 18 | 6 | 20 | SIZE | 2 | 8 | 4 |
| 21 | BASE_ADDRESS | 19 | 3 | 21 | BASE_ADDRESS | 3 | 5 | 5 | SIZE | 19 | 6 | 21 | SIZE | 3 | 8 | 5 |
| 22 | BASE_ADDRESS | 20 | 3 | 22 | BASE_ADDRESS | 4 | 5 | 6 | SIZE | 20 | 6 | 22 | SIZE | 4 | 8 | 6 |
| 23 | BASE_ADDRESS | 21 | 3 | 23 | BASE_ADDRESS | 5 | 5 | 7 | SIZE | 21 | 6 | 23 | SIZE | 5 | 8 | 7 |
| 24 | BASE_ADDRESS | 22 | 3 | 24 | BASE_ADDRESS | 6 | 5 | 8 | SIZE | 22 | 6 | 24 | SIZE | 6 | 8 | 8 |
| 25 | BASE_ADDRESS | 23 | 3 | 25 | BASE_ADDRESS | 7 | 5 | 9 | SIZE | 23 | 6 | 25 | SIZE | 7 | 8 | 9 |
| 26 | BASE_ADDRESS | 24 | 3 | 26 | BASE_ADDRESS | 8 | 5 | 10 | CLAMP_X | 0 | 6 | 26 | SIZE | 8 | 8 | 10 |
| 27 | BASE_ADDRESS | 25 | 3 | 27 | BASE_ADDRESS | 9 | 5 | 11 | BORDER_COLOR | 0 | 6 | 27 | SIZE | 9 | 8 | 11 |
| 28 | BASE_ADDRESS | 26 | 3 | 28 | BASE_ADDRESS | 10 | 5 | 12 | REQUEST_SIZE | 0 | 6 | 28 | SIZE | 10 | 8 | 12 |
| 29 | BASE_ADDRESS | 27 | 3 | 29 | BASE_ADDRESS | 11 | 5 | 13 | REQUEST_SIZE | 1 | 6 | 29 | SIZE | 11 | 8 | 13 |
| 30 | BASE_ADDRESS | 28 | 3 | 30 | BASE_ADDRESS | 12 | 5 | 14 | CLAMP_DISABLE | 0 | 6 | 30 | SIZE | 12 | 8 | 14 |
| 31 | BASE_ADDRESS | 29 | 3 | 31 | BASE_ADDRESS | 13 | 5 | 15 | unused | - | 6 | 31 | SIZE | 13 | 8 | 15 |
| 32 | ENDIAN_SWAP | 0 | 4 | 0 | BASE_ADDRESS | 14 | 5 | 16 | unused | - | 7 | 0 | SIZE | 14 | 8 | 16 |
| 33 | ENDIAN_SWAP | 1 | 4 | 1 | BASE_ADDRESS | 15 | 5 | 17 | unused | - | 7 | 1 | SIZE | 15 | 8 | 17 |
| 34 | SIZE | 0 | 4 | 2 | BASE_ADDRESS | 16 | 5 | 18 | BASE_ADDRESS | 0 | 7 | 2 | SIZE | 16 | 8 | 18 |
| 35 | SIZE | 1 | 4 | 3 | BASE_ADDRESS | 17 | 5 | 19 | BASE_ADDRESS | 1 | 7 | 3 | SIZE | 17 | 8 | 19 |
| 36 | SIZE | 2 | 4 | 4 | BASE_ADDRESS | 18 | 5 | 20 | BASE_ADDRESS | 2 | 7 | 4 | SIZE | 18 | 8 | 20 |
| 37 | SIZE | 3 | 4 | 5 | BASE_ADDRESS | 19 | 5 | 21 | BASE_ADDRESS | 3 | 7 | 5 | SIZE | 19 | 8 | 21 |
| 38 | SIZE | 4 | 4 | 6 | BASE_ADDRESS | 20 | 5 | 22 | BASE_ADDRESS | 4 | 7 | 6 | SIZE | 20 | 8 | 22 |
| 39 | SIZE | 5 | 4 | 7 | BASE_ADDRESS | 21 | 5 | 23 | BASE_ADDRESS | 5 | 7 | 7 | SIZE | 21 | 8 | 23 |
| 40 | SIZE | 6 | 4 | 8 | BASE_ADDRESS | 22 | 5 | 24 | BASE_ADDRESS | 6 | 7 | 8 | SIZE | 22 | 8 | 24 |
| 41 | SIZE | 7 | 4 | 9 | BASE_ADDRESS | 23 | 5 | 25 | BASE_ADDRESS | 7 | 7 | 9 | SIZE | 23 | 8 | 25 |
| 42 | SIZE | 8 | 4 | 10 | BASE_ADDRESS | 24 | 5 | 26 | BASE_ADDRESS | 8 | 7 | 10 | CLAMP_X | 0 | 8 | 26 |
| 43 | SIZE | 9 | 4 | 11 | BASE_ADDRESS | 25 | 5 | 27 | BASE_ADDRESS | 9 | 7 | 11 | BORDER_COLOR | 0 | 8 | 27 |
| 44 | SIZE | 10 | 4 | 12 | BASE_ADDRESS | 26 | 5 | 28 | BASE_ADDRESS | 10 | 7 | 12 | REQUEST_SIZE | 0 | 8 | 28 |
| 45 | SIZE | 11 | 4 | 13 | BASE_ADDRESS | 27 | 5 | 29 | BASE_ADDRESS | 11 | 7 | 13 | REQUEST_SIZE | 1 | 8 | 29 |
| 46 | SIZE | 12 | 4 | 14 | BASE_ADDRESS | 28 | 5 | 30 | BASE_ADDRESS | 12 | 7 | 14 | CLAMP_DISABLE | 0 | 8 | 30 |
| 47 | SIZE | 13 | 4 | 15 | BASE_ADDRESS | 29 | 5 | 31 | BASE_ADDRESS | 13 | 7 | 15 | unused | - | 8 | 31 |

**SQ_TP_thread_id**

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit |
| 0 | SQ_TP_thread_id | 0 | - | - | SQ_TP_thread_id | 2 | - | - | SQ_TP_thread_id | 4 | - | - | SQ_TP_end_of_clause | 0 | - | - |
| 1 | SQ_TP_thread_id | 1 | - | - | SQ_TP_thread_id | 3 | - | - | SQ_TP_thread_id | 5 | - | - | unused | - | - | - |

SQ_TP_gpr_wr_addr

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit |
| 0 | SQ_TP_type | 0 | - | - | SQ_TP_gpr_wr_addr | 1 | - | - | SQ_TP_gpr_wr_addr | 3 | - | - | SQ_TP_gpr_wr_addr | 5 | - | - |
| 1 | SQ_TP_gpr_wr_addr | 0 | - | - | SQ_TP_gpr_wr_addr | 2 | - | - | SQ_TP_gpr_wr_addr | 4 | - | - | SQ_TP_gpr_wr_addr | 6 | - | - |

Texture Fetch Instruction

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | 96 | DWORD | bit |
| 0 | OPCODE | 0 | 0 | 0 | DST_SEL_W | 1 | 1 | 10 | USE_COMP_LOD | 0 | 1 | 28 | LOD_BIAS_V | 6 | 2 | 14 | 1 | 0 | 5 |
| 1 | OPCODE | 1 | 0 | 1 | DST_SEL_W | 1 | 1 | 11 | USE_REG_LOD | 0 | 1 | 29 | LOD_BIAS_V | 7 | 2 | 15 | 1 | 0 | 6 |
| 2 | OPCODE | 2 | 0 | 2 | MAG_FILTER | 0 | 1 | 12 | USE_REG_LOD | 1 | 1 | 30 | SAMPLE_SHIFT_X | 0 | 2 | 16 | 1 | 0 | 7 |
| 3 | OPCODE | 3 | 0 | 3 | MAG_FILTER | 1 | 1 | 13 | unused | - | 1 | 31 | SAMPLE_SHIFT_X | 1 | 2 | 17 | 1 | 0 | 8 |
| 4 | OPCODE | 4 | 0 | 4 | MIN_FILTER | 0 | 1 | 14 | LOD_BIAS_H | 0 | 2 | 0 | SAMPLE_SHIFT_X | 2 | 2 | 18 | 1 | 0 | 9 |
| 5 | FETCH_VALID_ONLY | 0 | 0 | 19 | MIN_FILTER | 1 | 1 | 15 | LOD_BIAS_H | 1 | 2 | 1 | SAMPLE_SHIFT_X | 3 | 2 | 19 | 1 | 0 | 10 |
| 6 | TX_COORD_NUM | 0 | 0 | 25 | MIP_FILTER | 0 | 1 | 16 | LOD_BIAS_H | 2 | 2 | 2 | SAMPLE_SHIFT_X | 4 | 2 | 20 | 1 | 0 | 11 |
| 7 | unused | - | 0 | 26 | MIP_FILTER | 1 | 1 | 17 | LOD_BIAS_H | 3 | 2 | 3 | SAMPLE_SHIFT_Y | 0 | 2 | 21 | 1 | 0 | 12 |
| 8 | DST_SEL_X | 0 | 1 | 0 | ANISO_FILTER | 0 | 1 | 18 | LOD_BIAS_H | 4 | 2 | 4 | SAMPLE_SHIFT_Y | 1 | 2 | 22 | 1 | 0 | 13 |
| 9 | DST_SEL_X | 1 | 1 | 1 | ANISO_FILTER | 1 | 1 | 19 | LOD_BIAS_H | 5 | 2 | 5 | SAMPLE_SHIFT_Y | 2 | 2 | 23 | 1 | 0 | 14 |
| 10 | DST_SEL_X | 2 | 1 | 2 | ANISO_FILTER | 2 | 1 | 20 | LOD_BIAS_H | 6 | 2 | 6 | SAMPLE_SHIFT_Y | 3 | 2 | 24 | 1 | 0 | 15 |
| 11 | DST_SEL_Y | 0 | 1 | 3 | ARBITRARY_FILTER | 0 | 1 | 21 | LOD_BIAS_H | 7 | 2 | 7 | SAMPLE_SHIFT_Y | 4 | 2 | 25 | 1 | 0 | 16 |
| 12 | DST_SEL_Y | 1 | 1 | 4 | ARBITRARY_FILTER | 1 | 1 | 22 | LOD_BIAS_V | 0 | 2 | 8 | SAMPLE_SHIFT_Z | 0 | 2 | 26 | 1 | 0 | 17 |
| 13 | DST_SEL_Y | 2 | 1 | 5 | ARBITRARY_FILTER | 2 | 1 | 23 | LOD_BIAS_V | 1 | 2 | 9 | SAMPLE_SHIFT_Z | 1 | 2 | 27 | 1 | 0 | 18 |
| 14 | DST_SEL_Z | 0 | 1 | 6 | VOL_MAG_FILTER | 0 | 1 | 24 | LOD_BIAS_V | 2 | 2 | 10 | SAMPLE_SHIFT_Z | 2 | 2 | 28 | 1 | 0 | 20 |
| 15 | DST_SEL_Z | 1 | 1 | 7 | VOL_MAG_FILTER | 1 | 1 | 25 | LOD_BIAS_V | 3 | 2 | 11 | SAMPLE_SHIFT_Z | 3 | 2 | 29 | 1 | 0 | 21 |
| 16 | DST_SEL_Z | 2 | 1 | 8 | VOL_MIN_FILTER | 0 | 1 | 26 | LOD_BIAS_V | 4 | 2 | 12 | SAMPLE_SHIFT_Z | 4 | 2 | 30 | 1 | 0 | 22 |
| 17 | DST_SEL_W | 0 | 1 | 9 | VOL_MIN_FILTER | 1 | 1 | 27 | LOD_BIAS_V | 5 | 2 | 13 | unused | - | unused | - | 1 | 0 | 23 |
| | | | | | | | | | | | | | | | | | 1 | 0 | 26 |
| | | | | | | | | | | | | | | | | | 1 | 0 | 27 |
| | | | | | | | | | | | | | | | | | 1 | 0 | 28 |
| | | | | | | | | | | | | | | | | | 1 | 0 | 29 |
| | | | | | | | | | | | | | | | | | 1 | 0 | 30 |
| | | | | | | | | | | | | | | | | | 1 | 0 | 31 |
| | | | | | | | | | | | | | | | | | 1 | 1 | 31 |
| | | | | | | | | | | | | | | | | | 1 | 2 | 31 |
| | | | | | | | | | | | | | | | | | 70 | | |

Vertex Fetch Instruction

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit |
| 0 | OPCODE | 0 | 0 | 0 | DST_SEL_W | 1 | 1 | 10 | EXP_ADJUST_ALL | 4 | 1 | 28 | unused | - | 2 | 14 |
| 1 | OPCODE | 1 | 0 | 1 | DST_SEL_W | 1 | 1 | 11 | EXP_ADJUST_ALL | 5 | 1 | 29 | unused | - | 2 | 15 |
| 2 | OPCODE | 2 | 0 | 2 | FORMAT_COMP_ALL | 0 | 1 | 12 | unused | - | 1 | 30 | OFFSET | 0 | 2 | 16 |
| 3 | OPCODE | 3 | 0 | 3 | NUM_FORMAT_ALL | 0 | 1 | 13 | unused | - | 1 | 31 | OFFSET | 1 | 2 | 17 |
| 4 | OPCODE | 4 | 0 | 4 | SIGNED_RF_MODE_ALL | 0 | 1 | 14 | STRIDE | 0 | 2 | 0 | OFFSET | 2 | 2 | 18 |
| 5 | unused | - | 0 | 19 | unused | - | 1 | 15 | STRIDE | 1 | 2 | 1 | OFFSET | 3 | 2 | 19 |
| 6 | CONST_INDEX_SEL | 0 | 0 | 25 | DATA_FORMAT | 0 | 1 | 16 | STRIDE | 2 | 2 | 2 | OFFSET | 4 | 2 | 20 |
| 7 | CONST_INDEX_SEL | 1 | 0 | 26 | DATA_FORMAT | 1 | 1 | 17 | STRIDE | 3 | 2 | 3 | OFFSET | 5 | 2 | 21 |
| 8 | DST_SEL_X | 0 | 1 | 0 | DATA_FORMAT | 2 | 1 | 18 | STRIDE | 4 | 2 | 4 | OFFSET | 6 | 2 | 22 |
| 9 | DST_SEL_X | 1 | 1 | 1 | DATA_FORMAT | 3 | 1 | 19 | STRIDE | 5 | 2 | 5 | OFFSET | 7 | 2 | 23 |
| 10 | DST_SEL_X | 2 | 1 | 2 | DATA_FORMAT | 4 | 1 | 20 | STRIDE | 6 | 2 | 6 | unused | - | 2 | 24 |
| 11 | DST_SEL_Y | 0 | 1 | 3 | DATA_FORMAT | 5 | 1 | 21 | STRIDE | 7 | 2 | 7 | unused | - | 2 | 25 |
| 12 | DST_SEL_Y | 1 | 1 | 4 | unused | - | 1 | 22 | unused | - | 2 | 8 | unused | - | 2 | 26 |
| 13 | DST_SEL_Y | 2 | 1 | 5 | unused | - | 1 | 23 | unused | - | 2 | 9 | unused | - | 2 | 27 |
| 14 | DST_SEL_Z | 0 | 1 | 6 | EXP_ADJUST_ALL | 0 | 1 | 24 | unused | - | 2 | 10 | unused | - | 2 | 28 |
| 15 | DST_SEL_Z | 1 | 1 | 7 | EXP_ADJUST_ALL | 1 | 1 | 25 | unused | - | 2 | 11 | unused | - | 2 | 29 |
| 16 | DST_SEL_Z | 2 | 1 | 8 | EXP_ADJUST_ALL | 2 | 1 | 26 | unused | - | 2 | 12 | unused | - | 2 | 30 |
| 17 | DST_SEL_W | 0 | 1 | 9 | EXP_ADJUST_ALL | 3 | 1 | 27 | unused | - | 2 | 13 | unused | - | unused | - |

Texture Fetch Constant Fields

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit |
| 0 | unused | - | 3 | 0 | BASE_ADDRESS | 4 | 4 | 16 | NUM_FORMAT_ALL | 0 | 6 | 0 | LOD_BIAS_H | 6 | 7 | 16 |
| 1 | unused | - | 3 | 1 | BASE_ADDRESS | 5 | 4 | 17 | DST_SEL_X | 0 | 6 | 1 | LOD_BIAS_H | 7 | 7 | 17 |
| 2 | FORMAT_COMP_X | 0 | 3 | 2 | BASE_ADDRESS | 6 | 4 | 18 | DST_SEL_X | 1 | 6 | 2 | LOD_BIAS_H | 8 | 7 | 18 |
| 3 | FORMAT_COMP_X | 1 | 3 | 3 | BASE_ADDRESS | 7 | 4 | 19 | DST_SEL_X | 2 | 6 | 3 | LOD_BIAS_H | 9 | 7 | 19 |
| 4 | FORMAT_COMP_Y | 0 | 3 | 4 | BASE_ADDRESS | 8 | 4 | 20 | DST_SEL_Y | 0 | 6 | 4 | LOD_BIAS_V | 0 | 7 | 20 |
| 5 | FORMAT_COMP_Y | 1 | 3 | 5 | BASE_ADDRESS | 9 | 4 | 21 | DST_SEL_Y | 1 | 6 | 5 | LOD_BIAS_V | 1 | 7 | 21 |
| 6 | FORMAT_COMP_Z | 0 | 3 | 6 | BASE_ADDRESS | 10 | 4 | 22 | DST_SEL_Y | 2 | 6 | 6 | LOD_BIAS_V | 2 | 7 | 22 |
| 7 | FORMAT_COMP_Z | 1 | 3 | 7 | BASE_ADDRESS | 11 | 4 | 23 | DST_SEL_Z | 0 | 6 | 7 | LOD_BIAS_V | 3 | 7 | 23 |
| 8 | FORMAT_COMP_W | 0 | 3 | 8 | BASE_ADDRESS | 12 | 4 | 24 | DST_SEL_Z | 1 | 6 | 8 | LOD_BIAS_V | 4 | 7 | 24 |
| 9 | FORMAT_COMP_W | 1 | 3 | 9 | BASE_ADDRESS | 13 | 4 | 25 | DST_SEL_Z | 2 | 6 | 9 | LOD_BIAS_V | 5 | 7 | 25 |
| 10 | CLAMP_X | 0 | 3 | 10 | BASE_ADDRESS | 14 | 4 | 26 | DST_SEL_W | 0 | 6 | 10 | LOD_BIAS_V | 6 | 7 | 26 |

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | CLAMP_X | 1 | 3 | 11 | BASE_ADDRESS | 15 | 4 | 27 | DST_SEL_W | 1 | 6 | 11 | LOD_BIAS_V | 7 | 7 | 27 |
| 12 | CLAMP_X | 2 | 3 | 12 | BASE_ADDRESS | 16 | 4 | 28 | DST_SEL_W | 2 | 6 | 12 | LOD_BIAS_V | 8 | 7 | 28 |
| 13 | CLAMP_Y | 0 | 3 | 13 | BASE_ADDRESS | 17 | 4 | 29 | EXP_ADJUST_ALL | 0 | 6 | 13 | LOD_BIAS_V | 9 | 7 | 29 |
| 14 | CLAMP_Y | 1 | 3 | 14 | BASE_ADDRESS | 18 | 4 | 30 | EXP_ADJUST_ALL | 1 | 6 | 14 | **DIM_3D** | **0** | 7 | 30 |
| 15 | CLAMP_Y | 2 | 3 | 15 | BASE_ADDRESS | 19 | 4 | 31 | EXP_ADJUST_ALL | 2 | 6 | 15 | unused | - | 7 | 31 |
| 16 | CLAMP_Z | 0 | 3 | 16 | SIZE | 0 | 5 | 0 | EXP_ADJUST_ALL | 3 | 6 | 16 | BORDER_COLOR | 0 | 8 | 0 |
| 17 | CLAMP_Z | 1 | 3 | 17 | SIZE | 1 | 5 | 1 | EXP_ADJUST_ALL | 4 | 6 | 17 | BORDER_COLOR | 1 | 8 | 1 |
| 18 | CLAMP_Z | 2 | 3 | 18 | SIZE | 2 | 5 | 2 | EXP_ADJUST_ALL | 5 | 6 | 18 | FORCE_BC_W_TO_MAX | 0 | 8 | 2 |
| 19 | SIGNED_RF_MODE_ALL | 0 | 3 | 19 | SIZE | 3 | 5 | 3 | MAG_FILTER | 0 | 6 | 19 | unused | - | 8 | 3 |
| 20 | DIM | 0 | 3 | 20 | SIZE | 4 | 5 | 4 | MAG_FILTER | 1 | 6 | 20 | unused | - | 8 | 4 |
| 21 | DIM | 1 | 3 | 21 | SIZE | 5 | 5 | 5 | MIN_FILTER | 0 | 6 | 21 | unused | - | 8 | 5 |
| 22 | PITCH | 0 | 3 | 22 | SIZE | 6 | 5 | 6 | MIN_FILTER | 1 | 6 | 22 | unused | - | 8 | 6 |
| 23 | PITCH | 1 | 3 | 23 | SIZE | 7 | 5 | 7 | MIP_FILTER | 0 | 6 | 23 | unused | - | 8 | 7 |
| 24 | PITCH | 2 | 3 | 24 | SIZE | 8 | 5 | 8 | MIP_FILTER | 1 | 6 | 24 | unused | - | 8 | 8 |
| 25 | PITCH | 3 | 3 | 25 | SIZE | 9 | 5 | 9 | ANISO_FILTER | 0 | 6 | 25 | unused | - | 8 | 9 |
| 26 | PITCH | 4 | 3 | 26 | SIZE | 10 | 5 | 10 | ANISO_FILTER | 1 | 6 | 26 | unused | - | 8 | 10 |
| 27 | PITCH | 5 | 3 | 27 | SIZE | 11 | 5 | 11 | ANISO_FILTER | 2 | 6 | 27 | unused | - | 8 | 11 |
| 28 | PITCH | 6 | 3 | 28 | SIZE | 12 | 5 | 12 | ARBITRARY_FILTER | 0 | 6 | 28 | MIP_ADDRESS | 0 | 8 | 12 |
| 29 | PITCH | 7 | 3 | 29 | SIZE | 13 | 5 | 13 | ARBITRARY_FILTER | 1 | 6 | 29 | MIP_ADDRESS | 1 | 8 | 13 |
| 30 | PITCH | 8 | 3 | 30 | SIZE | 14 | 5 | 14 | ARBITRARY_FILTER | 2 | 6 | 30 | MIP_ADDRESS | 2 | 8 | 14 |
| 31 | TILED | 0 | 3 | 31 | SIZE | 15 | 5 | 15 | **BORDER_SIZE** | **0** | 6 | 31 | MIP_ADDRESS | 3 | 8 | 15 |
| 32 | DATA_FORMAT | 0 | 4 | 0 | SIZE | 16 | 5 | 16 | VOL_MAG_FILTER | 0 | 7 | 0 | MIP_ADDRESS | 4 | 8 | 16 |
| 33 | DATA_FORMAT | 1 | 4 | 1 | SIZE | 17 | 5 | 17 | VOL_MIN_FILTER | 0 | 7 | 1 | MIP_ADDRESS | 5 | 8 | 17 |
| 34 | DATA_FORMAT | 2 | 4 | 2 | SIZE | 18 | 5 | 18 | MIN_MIP_LEVEL | 0 | 7 | 2 | MIP_ADDRESS | 6 | 8 | 18 |
| 35 | DATA_FORMAT | 3 | 4 | 3 | SIZE | 19 | 5 | 19 | MIN_MIP_LEVEL | 1 | 7 | 3 | MIP_ADDRESS | 7 | 8 | 19 |
| 36 | DATA_FORMAT | 4 | 4 | 4 | SIZE | 20 | 5 | 20 | MIN_MIP_LEVEL | 2 | 7 | 4 | MIP_ADDRESS | 8 | 8 | 20 |
| 37 | DATA_FORMAT | 5 | 4 | 5 | SIZE | 21 | 5 | 21 | MIN_MIP_LEVEL | 3 | 7 | 5 | MIP_ADDRESS | 9 | 8 | 21 |
| 38 | unused | - | 4 | 6 | SIZE | 22 | 5 | 22 | MAX_MIP_LEVEL | 0 | 7 | 6 | MIP_ADDRESS | 10 | 8 | 22 |
| 39 | unused | - | 4 | 7 | SIZE | 23 | 5 | 23 | MAX_MIP_LEVEL | 1 | 7 | 7 | MIP_ADDRESS | 11 | 8 | 23 |
| 40 | unused | - | 4 | 8 | SIZE | 24 | 5 | 24 | MAX_MIP_LEVEL | 2 | 7 | 8 | MIP_ADDRESS | 12 | 8 | 24 |
| 41 | unused | - | 4 | 9 | SIZE | 25 | 5 | 25 | MAX_MIP_LEVEL | 3 | 7 | 9 | MIP_ADDRESS | 13 | 8 | 25 |
| 42 | unused | - | 4 | 10 | SIZE | 26 | 5 | 26 | LOD_BIAS_H | 0 | 7 | 10 | MIP_ADDRESS | 14 | 8 | 26 |
| 43 | unused | - | 4 | 11 | SIZE | 27 | 5 | 27 | LOD_BIAS_H | 1 | 7 | 11 | MIP_ADDRESS | 15 | 8 | 27 |
| 44 | BASE_ADDRESS | 0 | 4 | 12 | SIZE | 28 | 5 | 28 | LOD_BIAS_H | 2 | 7 | 12 | MIP_ADDRESS | 16 | 8 | 28 |
| 45 | BASE_ADDRESS | 1 | 4 | 13 | SIZE | 29 | 5 | 29 | LOD_BIAS_H | 3 | 7 | 13 | MIP_ADDRESS | 17 | 8 | 29 |
| 46 | BASE_ADDRESS | 2 | 4 | 14 | ENDIAN_SWAP | 0 | 5 | 30 | LOD_BIAS_H | 4 | 7 | 14 | MIP_ADDRESS | 18 | 8 | 30 |
| 47 | BASE_ADDRESS | 3 | 4 | 15 | ENDIAN_SWAP | 1 | 5 | 31 | LOD_BIAS_H | 5 | 7 | 15 | MIP_ADDRESS | 19 | 8 | 31 |

Vertex Fetch Constant Fields

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit |
| 0 | unused | - | 3 | 0 | LIMIT_ADDRESS | 14 | 4 | 16 | ENDIAN_SWAP | 0 | 6 | 0 | BASE_ADDRESS | 14 | 7 | 16 |
| 1 | unused | - | 3 | 1 | LIMIT_ADDRESS | 15 | 4 | 17 | ENDIAN_SWAP | 1 | 6 | 1 | BASE_ADDRESS | 15 | 7 | 17 |
| 2 | BASE_ADDRESS | 0 | 3 | 2 | LIMIT_ADDRESS | 16 | 4 | 18 | LIMIT_ADDRESS | 0 | 6 | 2 | BASE_ADDRESS | 16 | 7 | 18 |
| 3 | BASE_ADDRESS | 1 | 3 | 3 | LIMIT_ADDRESS | 17 | 4 | 19 | LIMIT_ADDRESS | 1 | 6 | 3 | BASE_ADDRESS | 17 | 7 | 19 |
| 4 | BASE_ADDRESS | 2 | 3 | 4 | LIMIT_ADDRESS | 18 | 4 | 20 | LIMIT_ADDRESS | 2 | 6 | 4 | BASE_ADDRESS | 18 | 7 | 20 |
| 5 | BASE_ADDRESS | 3 | 3 | 5 | LIMIT_ADDRESS | 19 | 4 | 21 | LIMIT_ADDRESS | 3 | 6 | 5 | BASE_ADDRESS | 19 | 7 | 21 |
| 6 | BASE_ADDRESS | 4 | 3 | 6 | LIMIT_ADDRESS | 20 | 4 | 22 | LIMIT_ADDRESS | 4 | 6 | 6 | BASE_ADDRESS | 20 | 7 | 22 |
| 7 | BASE_ADDRESS | 5 | 3 | 7 | LIMIT_ADDRESS | 21 | 4 | 23 | LIMIT_ADDRESS | 5 | 6 | 7 | BASE_ADDRESS | 21 | 7 | 23 |
| 8 | BASE_ADDRESS | 6 | 3 | 8 | LIMIT_ADDRESS | 22 | 4 | 24 | LIMIT_ADDRESS | 6 | 6 | 8 | BASE_ADDRESS | 22 | 7 | 24 |
| 9 | BASE_ADDRESS | 7 | 3 | 9 | LIMIT_ADDRESS | 23 | 4 | 25 | LIMIT_ADDRESS | 7 | 6 | 9 | BASE_ADDRESS | 23 | 7 | 25 |
| 10 | BASE_ADDRESS | 8 | 3 | 10 | LIMIT_ADDRESS | 24 | 4 | 26 | LIMIT_ADDRESS | 8 | 6 | 10 | BASE_ADDRESS | 24 | 7 | 26 |
| 11 | BASE_ADDRESS | 9 | 3 | 11 | LIMIT_ADDRESS | 25 | 4 | 27 | LIMIT_ADDRESS | 9 | 6 | 11 | BASE_ADDRESS | 25 | 7 | 27 |
| 12 | BASE_ADDRESS | 10 | 3 | 12 | LIMIT_ADDRESS | 26 | 4 | 28 | LIMIT_ADDRESS | 10 | 6 | 12 | BASE_ADDRESS | 26 | 7 | 28 |
| 13 | BASE_ADDRESS | 11 | 3 | 13 | LIMIT_ADDRESS | 27 | 4 | 29 | LIMIT_ADDRESS | 11 | 6 | 13 | BASE_ADDRESS | 27 | 7 | 29 |
| 14 | BASE_ADDRESS | 12 | 3 | 14 | LIMIT_ADDRESS | 28 | 4 | 30 | LIMIT_ADDRESS | 12 | 6 | 14 | **LIMIT_ADDRESS** | 29 | **7** | 30 |
| 15 | BASE_ADDRESS | 13 | 3 | 15 | LIMIT_ADDRESS | 29 | 4 | 31 | LIMIT_ADDRESS | 13 | 6 | 15 | BASE_ADDRESS | 29 | 7 | 31 |
| 16 | BASE_ADDRESS | 14 | 3 | 16 | unused | - | 5 | 0 | LIMIT_ADDRESS | 14 | 6 | 16 | ENDIAN_SWAP | 0 | 8 | 0 |

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 17 | BASE_ADDRESS | 15 | 3 | 17 | unused | - | 5 | 1 | LIMIT_ADDRESS | 15 | 6 | 17 | ENDIAN_SWAP | 1 | 8 | 1 |
| 18 | BASE_ADDRESS | 16 | 3 | 18 | BASE_ADDRESS | 0 | 5 | 2 | LIMIT_ADDRESS | 16 | 6 | 18 | LIMIT_ADDRESS | 0 | 8 | 2 |
| 19 | BASE_ADDRESS | 17 | 3 | 19 | BASE_ADDRESS | 1 | 5 | 3 | LIMIT_ADDRESS | 17 | 6 | 19 | LIMIT_ADDRESS | 1 | 8 | 3 |
| 20 | BASE_ADDRESS | 18 | 3 | 20 | BASE_ADDRESS | 2 | 5 | 4 | LIMIT_ADDRESS | 18 | 6 | 20 | LIMIT_ADDRESS | 2 | 8 | 4 |
| 21 | BASE_ADDRESS | 19 | 3 | 21 | BASE_ADDRESS | 3 | 5 | 5 | LIMIT_ADDRESS | 19 | 6 | 21 | LIMIT_ADDRESS | 3 | 8 | 5 |
| 22 | BASE_ADDRESS | 20 | 3 | 22 | BASE_ADDRESS | 4 | 5 | 6 | LIMIT_ADDRESS | 20 | 6 | 22 | LIMIT_ADDRESS | 4 | 8 | 6 |
| 23 | BASE_ADDRESS | 21 | 3 | 23 | BASE_ADDRESS | 5 | 5 | 7 | LIMIT_ADDRESS | 21 | 6 | 23 | LIMIT_ADDRESS | 5 | 8 | 7 |
| 24 | BASE_ADDRESS | 22 | 3 | 24 | BASE_ADDRESS | 6 | 5 | 8 | LIMIT_ADDRESS | 22 | 6 | 24 | LIMIT_ADDRESS | 6 | 8 | 8 |
| 25 | BASE_ADDRESS | 23 | 3 | 25 | BASE_ADDRESS | 7 | 5 | 9 | LIMIT_ADDRESS | 23 | 6 | 25 | LIMIT_ADDRESS | 7 | 8 | 9 |
| 26 | BASE_ADDRESS | 24 | 3 | 26 | BASE_ADDRESS | 8 | 5 | 10 | LIMIT_ADDRESS | 24 | 6 | 26 | LIMIT_ADDRESS | 8 | 8 | 10 |
| 27 | BASE_ADDRESS | 25 | 3 | 27 | BASE_ADDRESS | 9 | 5 | 11 | LIMIT_ADDRESS | 25 | 6 | 27 | LIMIT_ADDRESS | 9 | 8 | 11 |
| 28 | BASE_ADDRESS | 26 | 3 | 28 | BASE_ADDRESS | 10 | 5 | 12 | LIMIT_ADDRESS | 26 | 6 | 28 | LIMIT_ADDRESS | 10 | 8 | 12 |
| 29 | BASE_ADDRESS | 27 | 3 | 29 | BASE_ADDRESS | 11 | 5 | 13 | LIMIT_ADDRESS | 27 | 6 | 29 | LIMIT_ADDRESS | 11 | 8 | 13 |
| 30 | BASE_ADDRESS | 28 | 3 | 30 | BASE_ADDRESS | 12 | 5 | 14 | LIMIT_ADDRESS | 28 | 6 | 30 | LIMIT_ADDRESS | 12 | 8 | 14 |
| 31 | BASE_ADDRESS | 29 | 3 | 31 | BASE_ADDRESS | 13 | 5 | 15 | **BASE_ADDRESS** | 28 | **6** | 31 | LIMIT_ADDRESS | 13 | 8 | 15 |
| 32 | ENDIAN_SWAP | 0 | 4 | 0 | BASE_ADDRESS | 14 | 5 | 16 | unused | - | 7 | 0 | LIMIT_ADDRESS | 14 | 8 | 16 |
| 33 | ENDIAN_SWAP | 1 | 4 | 1 | BASE_ADDRESS | 15 | 5 | 17 | unused | - | 7 | 1 | LIMIT_ADDRESS | 15 | 8 | 17 |
| 34 | LIMIT_ADDRESS | 0 | 4 | 2 | BASE_ADDRESS | 16 | 5 | 18 | BASE_ADDRESS | 0 | 7 | 2 | LIMIT_ADDRESS | 16 | 8 | 18 |
| 35 | LIMIT_ADDRESS | 1 | 4 | 3 | BASE_ADDRESS | 17 | 5 | 19 | BASE_ADDRESS | 1 | 7 | 3 | LIMIT_ADDRESS | 17 | 8 | 19 |
| 36 | LIMIT_ADDRESS | 2 | 4 | 4 | BASE_ADDRESS | 18 | 5 | 20 | BASE_ADDRESS | 2 | 7 | 4 | LIMIT_ADDRESS | 18 | 8 | 20 |
| 37 | LIMIT_ADDRESS | 3 | 4 | 5 | BASE_ADDRESS | 19 | 5 | 21 | BASE_ADDRESS | 3 | 7 | 5 | LIMIT_ADDRESS | 19 | 8 | 21 |
| 38 | LIMIT_ADDRESS | 4 | 4 | 6 | BASE_ADDRESS | 20 | 5 | 22 | BASE_ADDRESS | 4 | 7 | 6 | LIMIT_ADDRESS | 20 | 8 | 22 |
| 39 | LIMIT_ADDRESS | 5 | 4 | 7 | BASE_ADDRESS | 21 | 5 | 23 | BASE_ADDRESS | 5 | 7 | 7 | LIMIT_ADDRESS | 21 | 8 | 23 |
| 40 | LIMIT_ADDRESS | 6 | 4 | 8 | BASE_ADDRESS | 22 | 5 | 24 | BASE_ADDRESS | 6 | 7 | 8 | LIMIT_ADDRESS | 22 | 8 | 24 |
| 41 | LIMIT_ADDRESS | 7 | 4 | 9 | BASE_ADDRESS | 23 | 5 | 25 | BASE_ADDRESS | 7 | 7 | 9 | LIMIT_ADDRESS | 23 | 8 | 25 |
| 42 | LIMIT_ADDRESS | 8 | 4 | 10 | BASE_ADDRESS | 24 | 5 | 26 | BASE_ADDRESS | 8 | 7 | 10 | LIMIT_ADDRESS | 24 | 8 | 26 |
| 43 | LIMIT_ADDRESS | 9 | 4 | 11 | BASE_ADDRESS | 25 | 5 | 27 | BASE_ADDRESS | 9 | 7 | 11 | LIMIT_ADDRESS | 25 | 8 | 27 |
| 44 | LIMIT_ADDRESS | 10 | 4 | 12 | BASE_ADDRESS | 26 | 5 | 28 | BASE_ADDRESS | 10 | 7 | 12 | LIMIT_ADDRESS | 26 | 8 | 28 |
| 45 | LIMIT_ADDRESS | 11 | 4 | 13 | BASE_ADDRESS | 27 | 5 | 29 | BASE_ADDRESS | 11 | 7 | 13 | LIMIT_ADDRESS | 27 | 8 | 29 |
| 46 | LIMIT_ADDRESS | 12 | 4 | 14 | BASE_ADDRESS | 28 | 5 | 30 | BASE_ADDRESS | 12 | 7 | 14 | LIMIT_ADDRESS | 28 | 8 | 30 |
| 47 | LIMIT_ADDRESS | 13 | 4 | 15 | BASE_ADDRESS | 29 | 5 | 31 | BASE_ADDRESS | 13 | 7 | 15 | LIMIT_ADDRESS | 29 | 8 | 31 |

SQ_TP_clause

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit |
| 0 | SQ_TP_clause_num | 0 | - | - | SQ_TP_clause_num | 1 | - | - | SQ_TP_clause_num | 2 | - | - | SQ_TP_end_of_clause | 0 | - | - |

SQ_TP_gpr_wr_addr

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SQ_TP | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit | FIELD | bit | DWORD | bit |
| 0 | SQ_TP_type | 0 | - | - | SQ_TP_gpr_wr_addr | 1 | - | - | SQ_TP_gpr_wr_addr | 3 | - | - | SQ_TP_gpr_wr_addr | 5 | - | - |
| 1 | SQ_TP_gpr_wr_addr | 0 | - | - | SQ_TP_gpr_wr_addr | 2 | - | - | SQ_TP_gpr_wr_addr | 4 | - | - | SQ_TP_gpr_wr_addr | 6 | - | - |

**Vertex Fetch Constant Fields**

| | Cycle 0 | | | | Cycle 1 | | | | Cycle 2 | | | | Cycle 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **SQ_TP** | **FIELD** | **bit** | **DWORD** | **bit** | **FIELD** | **bit** | **DWORD** | **bit** | **FIELD** | **bit** | **DWORD** | **bit** | **FIELD** | **bit** | **DWORD** | **bit** |
| 0 | unused | - | 2 | 30 | unused | - | 2 | 0 | unused | - | 5 | 6 | ENDIAN_SWAP | 0 | 1 | 0 |
| 1 | unused | - | 2 | 31 | unused | - | 2 | 1 | unused | - | 5 | 7 | ENDIAN_SWAP | 1 | 1 | 1 |
| 2 | BASE_ADDRESS | 0 | 0 | 2 | unused | - | 2 | 2 | unused | - | 5 | 8 | LIMIT_ADDRESS | 0 | 1 | 2 |
| 3 | BASE_ADDRESS | 1 | 0 | 3 | unused | - | 2 | 3 | unused | - | 5 | 9 | LIMIT_ADDRESS | 1 | 1 | 3 |
| 4 | BASE_ADDRESS | 2 | 0 | 4 | unused | - | 2 | 4 | unused | - | 5 | 10 | LIMIT_ADDRESS | 2 | 1 | 4 |
| 5 | BASE_ADDRESS | 3 | 0 | 5 | unused | - | 2 | 5 | unused | - | 5 | 11 | LIMIT_ADDRESS | 3 | 1 | 5 |
| 6 | BASE_ADDRESS | 4 | 0 | 6 | unused | - | 2 | 6 | unused | - | 5 | 12 | LIMIT_ADDRESS | 4 | 1 | 6 |
| 7 | BASE_ADDRESS | 5 | 0 | 7 | unused | - | 2 | 7 | unused | - | 5 | 13 | LIMIT_ADDRESS | 5 | 1 | 7 |
| 8 | BASE_ADDRESS | 6 | 0 | 8 | unused | - | 2 | 8 | unused | - | 5 | 14 | LIMIT_ADDRESS | 6 | 1 | 8 |
| 9 | BASE_ADDRESS | 7 | 0 | 9 | unused | - | 2 | 9 | unused | - | 5 | 15 | LIMIT_ADDRESS | 7 | 1 | 9 |
| 10 | BASE_ADDRESS | 8 | 0 | 10 | unused | - | 2 | 10 | unused | - | 5 | 16 | LIMIT_ADDRESS | 8 | 1 | 10 |
| 11 | BASE_ADDRESS | 9 | 0 | 11 | unused | - | 2 | 11 | unused | - | 5 | 17 | LIMIT_ADDRESS | 9 | 1 | 11 |
| 12 | BASE_ADDRESS | 10 | 0 | 12 | unused | - | 2 | 12 | unused | - | 5 | 18 | LIMIT_ADDRESS | 10 | 1 | 12 |
| 13 | BASE_ADDRESS | 11 | 0 | 13 | unused | - | 2 | 13 | unused | - | 5 | 19 | LIMIT_ADDRESS | 11 | 1 | 13 |
| 14 | BASE_ADDRESS | 12 | 0 | 14 | unused | - | 2 | 14 | unused | - | 5 | 20 | LIMIT_ADDRESS | 12 | 1 | 14 |
| 15 | BASE_ADDRESS | 13 | 0 | 15 | unused | - | 2 | 15 | unused | - | 5 | 21 | LIMIT_ADDRESS | 13 | 1 | 15 |
| 16 | BASE_ADDRESS | 14 | 0 | 16 | unused | - | 2 | 16 | unused | - | 5 | 22 | LIMIT_ADDRESS | 14 | 1 | 16 |
| 17 | BASE_ADDRESS | 15 | 0 | 17 | unused | - | 2 | 17 | unused | - | 5 | 23 | LIMIT_ADDRESS | 15 | 1 | 17 |
| 0 | BASE_ADDRESS | 16 | 0 | 18 | unused | - | 2 | 18 | unused | - | 5 | 24 | LIMIT_ADDRESS | 16 | 1 | 18 |
| Fetch Insti | BASE_ADDRESS | 17 | 0 | 19 | unused | - | 2 | 19 | unused | - | 4 | 2 | LIMIT_ADDRESS | 17 | 1 | 19 |
| 0 | BASE_ADDRESS | 18 | 0 | 20 | unused | - | 2 | 20 | unused | - | 4 | 3 | LIMIT_ADDRESS | 18 | 1 | 20 |
| SQ_TP | BASE_ADDRESS | 19 | 0 | 21 | unused | - | 2 | 21 | unused | - | 4 | 4 | LIMIT_ADDRESS | 19 | 1 | 21 |
| 0 | BASE_ADDRESS | 20 | 0 | 22 | unused | - | 2 | 22 | unused | - | 4 | 5 | LIMIT_ADDRESS | 20 | 1 | 22 |
| 1 | BASE_ADDRESS | 21 | 0 | 23 | unused | - | 2 | 23 | unused | - | 4 | 6 | LIMIT_ADDRESS | 21 | 1 | 23 |
| 2 | BASE_ADDRESS | 22 | 0 | 24 | unused | - | 2 | 24 | unused | - | 4 | 7 | LIMIT_ADDRESS | 22 | 1 | 24 |
| 3 | BASE_ADDRESS | 23 | 0 | 25 | unused | - | 2 | 25 | unused | - | 4 | 8 | LIMIT_ADDRESS | 23 | 1 | 25 |
| 4 | BASE_ADDRESS | 24 | 0 | 26 | unused | - | 2 | 26 | unused | - | 4 | 9 | LIMIT_ADDRESS | 24 | 1 | 26 |
| 5 | BASE_ADDRESS | 25 | 0 | 27 | unused | - | 2 | 27 | unused | - | 4 | 10 | LIMIT_ADDRESS | 25 | 1 | 27 |
| 6 | BASE_ADDRESS | 26 | 0 | 28 | unused | - | 2 | 28 | unused | - | 4 | 11 | LIMIT_ADDRESS | 26 | 1 | 28 |
| 7 | BASE_ADDRESS | 27 | 0 | 29 | unused | - | 2 | 29 | unused | - | 4 | 12 | LIMIT_ADDRESS | 27 | 1 | 29 |
| 8 | BASE_ADDRESS | 28 | 0 | 30 | BASE_ADDRESS | 18 | 0 | 20 | unused | - | 4 | 13 | LIMIT_ADDRESS | 28 | 1 | 30 |
| 9 | BASE_ADDRESS | 29 | 0 | 31 | BASE_ADDRESS | 19 | 0 | 21 | unused | - | 4 | 14 | LIMIT_ADDRESS | 29 | 1 | 31 |
| 10 | unused | - | 5 | 0 | unused | - | 4 | 30 | unused | - | 4 | 15 | unused | - | 3 | 31 |
| 11 | unused | - | 5 | 1 | unused | - | 3 | 19 | unused | - | 4 | 16 | unused | - | 5 | 2 |
| 12 | unused | - | 3 | 0 | unused | - | 3 | 20 | unused | - | 4 | 17 | unused | - | 5 | 25 |
| 13 | unused | - | 3 | 1 | unused | - | 3 | 21 | unused | - | 4 | 18 | unused | - | 5 | 26 |
| 14 | unused | - | 3 | 2 | unused | - | 3 | 22 | unused | - | 4 | 19 | unused | - | 5 | 27 |
| 15 | unused | - | 3 | 3 | unused | - | 3 | 23 | unused | - | 4 | 20 | unused | - | 5 | 28 |
| 16 | unused | - | 3 | 4 | unused | - | 3 | 24 | unused | - | 4 | 21 | unused | - | 5 | 29 |
| 17 | unused | - | 3 | 5 | unused | - | 3 | 25 | unused | - | 4 | 22 | unused | - | 5 | 30 |
| 0 | unused | - | 3 | 6 | unused | - | 3 | 26 | unused | - | 4 | 23 | unused | - | 5 | 31 |
| etch Const | unused | - | 3 | 7 | unused | - | 3 | 27 | unused | - | 4 | 24 | unused | - | 3 | 13 |
| 0 | unused | - | 3 | 8 | unused | - | 3 | 28 | unused | - | 4 | 25 | unused | - | 3 | 14 |
| SQ_TP | unused | - | 3 | 9 | unused | - | 3 | 29 | unused | - | 4 | 26 | unused | - | 3 | 15 |
| 0 | unused | - | 3 | 10 | unused | - | 3 | 30 | unused | - | 4 | 27 | unused | - | 3 | 16 |
| 1 | unused | - | 3 | 11 | unused | - | 4 | 0 | unused | - | 4 | 28 | unused | - | 3 | 17 |
| 2 | unused | - | 3 | 12 | unused | - | 4 | 1 | unused | - | 4 | 29 | unused | - | 3 | 18 |

**Vertex Fetch Constant Fields (Straightforward Packing)**

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **1** | **2.65625** | | | **9** | **4.15625** | | | **1** | **5.65625** | | | **9** | **7.15625** | |
| **6** | **FORMAT_COMP_Z** | **0** | **2.6875** | **6** | **BASE_ADDRESS** | **10** | **4.1875** | **22** | **DST_SEL_Y** | **2** | **5.6875** | **6** | **AD_EXP_ADJUST_H** | **0** | **7.1875** | **22** |
| 7 | BASE_ADDRESS | 0 | 0 | 2 | LIMIT_ADDRESS | 15 | 1 | 17 | unused | - | 3 | 0 | unused | - | 4 | 15 |
| 8 | BASE_ADDRESS | 1 | 0 | 3 | LIMIT_ADDRESS | 16 | 1 | 18 | unused | - | 3 | 1 | unused | - | 4 | 16 |
| 9 | BASE_ADDRESS | 2 | 0 | 4 | LIMIT_ADDRESS | 17 | 1 | 19 | unused | - | 3 | 2 | unused | - | 4 | 17 |
| 10 | BASE_ADDRESS | 3 | 0 | 5 | LIMIT_ADDRESS | 18 | 1 | 20 | unused | - | 3 | 3 | unused | - | 4 | 18 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 11 | BASE_ADDRESS 4 | 0 6 | LIMIT_ADDRESS 19 | 1 21 | unused - | 3 4 | unused - | 4 19 |
| 12 | BASE_ADDRESS 5 | 0 7 | LIMIT_ADDRESS 20 | 1 22 | unused - | 3 5 | unused - | 4 20 |
| 13 | BASE_ADDRESS 6 | 0 8 | LIMIT_ADDRESS 21 | 1 23 | unused - | 3 6 | unused - | 4 21 |
| 14 | BASE_ADDRESS 7 | 0 9 | LIMIT_ADDRESS 22 | 1 24 | unused - | 3 7 | unused - | 4 22 |
| 15 | BASE_ADDRESS 8 | 0 10 | LIMIT_ADDRESS 23 | 1 25 | unused - | 3 8 | unused - | 4 23 |
| 16 | BASE_ADDRESS 9 | 0 11 | LIMIT_ADDRESS 24 | 1 26 | unused - | 3 9 | unused - | 4 24 |
| 17 | BASE_ADDRESS 10 | 0 12 | LIMIT_ADDRESS 25 | 1 27 | unused - | 3 10 | unused - | 4 25 |
| 18 | BASE_ADDRESS 11 | 0 13 | LIMIT_ADDRESS 26 | 1 28 | unused - | 3 11 | unused - | 4 26 |
| 19 | BASE_ADDRESS 12 | 0 14 | LIMIT_ADDRESS 27 | 1 29 | unused - | 3 12 | unused - | 4 27 |
| 20 | BASE_ADDRESS 13 | 0 15 | LIMIT_ADDRESS 28 | 1 30 | unused - | 3 13 | unused - | 4 28 |
| 21 | BASE_ADDRESS 14 | 0 16 | LIMIT_ADDRESS 29 | 1 31 | unused - | 3 14 | unused - | 4 29 |
| 22 | BASE_ADDRESS 15 | 0 17 | unused - | 2 0 | unused - | 3 15 | unused - | 4 30 |
| 23 | BASE_ADDRESS 16 | 0 18 | unused - | 2 1 | unused - | 3 16 | unused - | 5 0 |
| 24 | BASE_ADDRESS 17 | 0 19 | unused - | 2 2 | unused - | 3 17 | unused - | 5 1 |
| 25 | BASE_ADDRESS 18 | 0 20 | unused - | 2 3 | unused - | 3 18 | unused - | 5 2 |
| 26 | BASE_ADDRESS 19 | 0 21 | unused - | 2 4 | unused - | 3 19 | unused - | 5 6 |
| 27 | BASE_ADDRESS 20 | 0 22 | unused - | 2 5 | unused - | 3 20 | unused - | 5 7 |
| 28 | BASE_ADDRESS 21 | 0 23 | unused - | 2 6 | unused - | 3 21 | unused - | 5 8 |
| 29 | BASE_ADDRESS 22 | 0 24 | unused - | 2 7 | unused - | 3 22 | unused - | 5 9 |
| 30 | BASE_ADDRESS 23 | 0 25 | unused - | 2 8 | unused - | 3 23 | unused - | 5 10 |
| 31 | BASE_ADDRESS 24 | 0 26 | unused - | 2 9 | unused - | 3 24 | unused - | 5 11 |
| 32 | BASE_ADDRESS 25 | 0 27 | unused - | 2 10 | unused - | 3 25 | unused - | 5 12 |
| 33 | BASE_ADDRESS 26 | 0 28 | unused - | 2 11 | unused - | 3 26 | unused - | 5 13 |
| 34 | BASE_ADDRESS 27 | 0 29 | unused - | 2 12 | unused - | 3 27 | unused - | 5 14 |
| 35 | BASE_ADDRESS 28 | 0 30 | unused - | 2 13 | unused - | 3 28 | unused - | 5 15 |
| 36 | BASE_ADDRESS 29 | 0 31 | unused - | 2 14 | unused - | 3 29 | unused - | 5 16 |
| 37 | ENDIAN_SWAP 0 | 1 0 | unused - | 2 15 | unused - | 3 30 | unused - | 5 17 |
| 38 | ENDIAN_SWAP 1 | 1 1 | unused - | 2 16 | unused - | 3 31 | unused - | 5 18 |
| 39 | LIMIT_ADDRESS 0 | 1 2 | unused - | 2 17 | unused - | 4 0 | unused - | 5 19 |
| 40 | LIMIT_ADDRESS 1 | 1 3 | unused - | 2 18 | unused - | 4 1 | unused - | 5 20 |
| 41 | LIMIT_ADDRESS 2 | 1 4 | unused - | 2 19 | unused - | 4 2 | unused - | 5 21 |
| 42 | LIMIT_ADDRESS 3 | 1 5 | unused - | 2 20 | unused - | 4 3 | unused - | 5 22 |
| 43 | LIMIT_ADDRESS 4 | 1 6 | unused - | 2 21 | unused - | 4 4 | unused - | 5 23 |
| 44 | LIMIT_ADDRESS 5 | 1 7 | unused - | 2 22 | unused - | 4 5 | unused - | 5 24 |
| 45 | LIMIT_ADDRESS 6 | 1 8 | unused - | 2 23 | unused - | 4 6 | unused - | 5 25 |
| 46 | LIMIT_ADDRESS 7 | 1 9 | unused - | 2 24 | unused - | 4 7 | unused - | 5 26 |
| 47 | LIMIT_ADDRESS 8 | 1 10 | unused - | 2 25 | unused - | 4 8 | unused - | 5 27 |
| 0 | LIMIT_ADDRESS 9 | 1 11 | unused - | 2 26 | unused - | 4 9 | unused - | 5 28 |
| etch Consta | LIMIT_ADDRESS 10 | 1 12 | unused - | 2 27 | unused - | 4 10 | unused - | 5 29 |
| 0 | LIMIT_ADDRESS 11 | 1 13 | unused - | 2 28 | unused - | 4 11 | unused - | 5 30 |
| SQ_TP | LIMIT_ADDRESS 12 | 1 14 | unused - | 2 29 | unused - | 4 12 | unused - | 5 31 |
| 0 | LIMIT_ADDRESS 13 | 1 15 | unused - | 2 30 | unused - | 4 13 | unused - | unused - |
| 1 | LIMIT_ADDRESS 14 | 1 16 | unused - | 2 31 | unused - | 4 14 | unused - | unused - |

| | ORIGINATE DATE | EDIT DATE | DOCUMENT-REV. NUM. | PAGE |
|---|---|---|---|---|
| | 4 June, 2002 | [date \@ "d MMMM, | GEN-CXXXXX-REVA | 1 of 32 |

**Author:** Jay C. Wilkinson

| | |
|---|---|
| **Issue To:** | **Copy No:** |

# R400 RB Depth
# (RBD)

## ver 0.1

**Overview:** The R400 RB Depth computes stencil and depth tests on quads of pixels.

WARNING: Familiarity with "R400 Memory Format Specification" (Perforce //depot/r400/doc_lib/design/chip/memory/-R400_MemoryFormat.pdf) is **required**.

AUTOMATICALLY UPDATED FIELDS:

| | |
|---|---|
| **Document Location:** | C:\r400\doc_lib\design\blocks\rb\R400 RB Depth.doc |
| **Current Intranet Search Title** : | R400 RB Depth (RBD) |

APPROVALS

| Name/Dept | Signature/Date |
|---|---|
| | |
| | |
| | |

Remarks:

THIS DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

# Table Of Contents

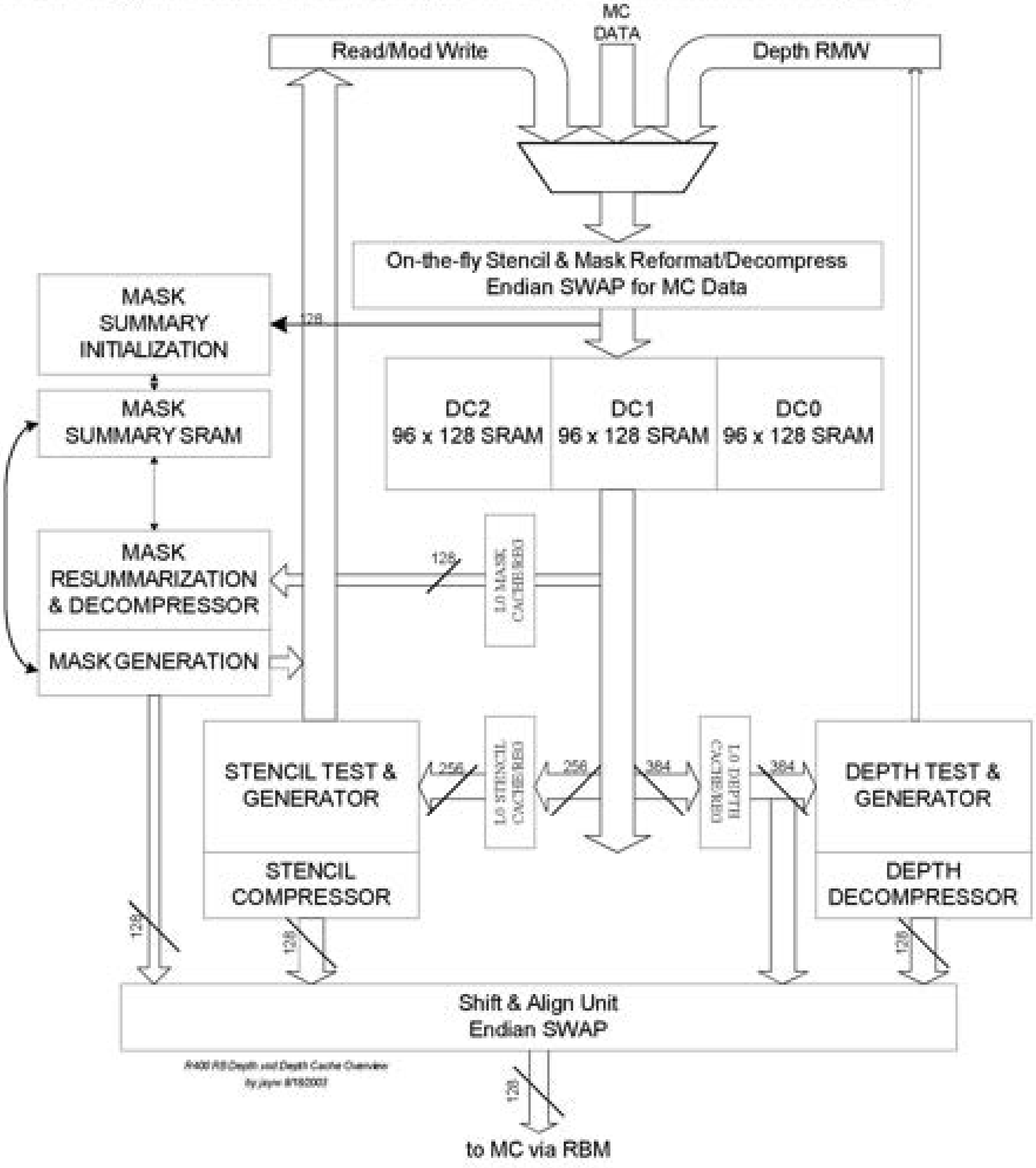
# Registers

## Revision Changes:

| | |
|---|---|
| **Rev 0.0 Jay C. Wilkinson**<br>Date: June 4, 2002 | Initial revision.<br>TBD: Basic DC diagram and RBM interface. |
| **Rev 0.1 Jay C. Wilkinson**<br>Date: July 17, 2002 | Added start of depth tile calculation and detailed stencil frame buffer organization descriptions. |

## Introduction

The Crayola depth logic block is one of four major subblocks of the Render Backend (RB). The depth logic is responsible for performing Stencil, Depth and early alpha testing and updating the depth and stencil data in the frame buffer. Quads surviving these tests are are sent along to the Render Backend Color subblock (RBC).



# 1. Architectural Requirements

## 2. Overview

The R400 has two sets of data to maintain, stencil and depth (often confusingly called 'Z'). The storage and processing of these two pieces on information had been simple and straight forward in previous designs. The R300 brought about the advent of 'compressed' depth and the R400 uses a similar scheme. The R400 also provides 2X stencil compression, a much larger tile size and more depth precision.

In the R400 frame buffer information is split between the pairs of RB&MC blocks. The R400 uses one, two or four RB&MC block pairs, dividing the frame buffer into a sophisticated (read that as complex) checkerboard pattern.

An additional dimension of complexity is that pixels/quads/tiles may be multi-sampled (**PA_SC_AA_CONFIG.-MSAA_ENABLE{ XE "PA_SC_AA_CONFIG:**MSAA_ENABLE" **}** and **PA_SC_AA_CONFIG.MSAA_NUM_SAMPLES{ XE "PA_SC_AA_CONFIG:**MSAA_NUM_SAMPLES" **}** > 0). The R400 supports 1, 2, 3, 4, 6 and 8X multi-sampling. The stencil and depth values are stored and processed on a per sample basis. This results in significant requirements in frame buffer size, memory bandwidth, internal storage and processing power.

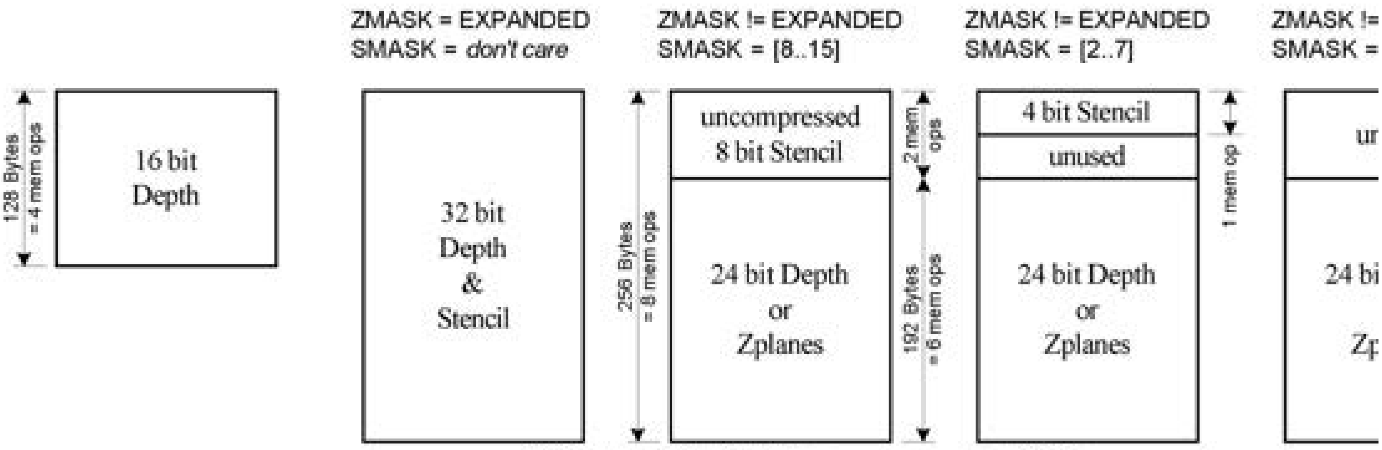
'Compressed' depth introduces a third set of data, a companion to 'compressed' depth. This third data set is depth Pmask data. Each sample within a pixel may use a sharable 'compressed' depth with another sample within the same tile. The Pmask identifies for every sample the index of its shared depth plane (Zplane).

In order to optimize memory read and write bandwidth the arrangement of the stencil, Pmask and depth data have several variations to pack them as tightly as possible within 32 byte blocks (the size of all memory transfers). The packing format is stored and maintained on a tile by tile basis by the Tile Logic subblock of the RB as SMASK and ZMASK.

When one multiplies all the variations of sizes, compressions and memory packing the result is quite dizzying. The result should be a very significant decrease in required bandwidth by the depth logic as compared to any previous graphics designs. In the future, when even more gate and buffer space is available, much more effective compression techniques may be used which should dramatically improve on the R400's performance especially in multisample rendering with stencils.

# 3. Stencil Formats

Stencil has a few variations in where and in what format it's stored in its part of the tile's depth surface. Below is an overview of the stencil and depth partitioning within a single tile's section of the depth surface. The case for a single sample tile's depth surface is shown below.



*R400 RB Depth vsd Tile Format Stencil Overview*    *by jayw 8/18/2003*

In the case of a multisampled tile each section's size is multiplied by the number of samples.

## 3.1 No Stencil Data (Frame Buffer only)

The R400 only supports stencil with 24 bit depth (**RB_DEPTH_INFO.DEPTH_FORMAT{** XE "**RB_DEPTH_INFO**:DEPTH_FORMAT" **}** is **DEPTH_24_8** or **DEPTH_24_8_FLOAT**); there is no stencil when the depth surface is 16 bit (**RB_DEPTH_INFO.DEPTH_FORMAT{** XE "**RB_DEPTH_INFO**:DEPTH_FORMAT" **}** is **DEPTH_16**).

The R400 supports stencil being disabled (**RB_DEPTHCONTROL.STENCIL_ENABLE = 0{** XE "**RB_DEPTHCONTROL**:STENCIL_ENABLE" **}**) in a context. As may be expected no stencil reads or writes will occur and the memory bandwidth requirements are, usually, lessened correspondingly. The extremely rare exception is when ZMASK is "**EXPANDED**", i.e. the 24 bit depth is interleaved with the 8 bit stencil. ZMASK is "**EXPANDED**" when ZMASK is disabled (**RB_DEPTH_INFO.ZMASK_ENABLE{** XE "**RB_DEPTH_INFO**:ZMASK_ENABLE" **}** = 0) or when the depth surface is being converted for software compatibility (**RB_TILECONTROL.FAST_DEPTH_EXPAND{** XE "**RB_TILECONTROL**:FAST_DEPTH_EXPAND" **}** = 1).

## 3.2 Uncompressed Stencil Data

Uncompressed stencils are 8 bit unsigned integers, one per sample. Except for when ZMASK is "**EXPANDED**", all the stencil data for an 8 x 8 pixel tile is stored contiguously at the start of a tile's depth surface. Therefore the sixty-four pixels times eight bits per stencil-sample requires 512 bits or 64 bytes of stencil storage per tile per sample. The result is a stencil area from 64 bytes (one sample per pixel) up to 512 bytes (8X multisampling) per tile. Stencil data is stored uncompressed when SMASK is disabled (**RB_DEPTH_INFO.SMASK_ENABLE{** XE "**RB_DEPTH_INFO**:SMASK_ENABLE" **}** = 0) or when the depth surface is being converted for software compatibility (**RB_TILECONTROL.FAST_DEPTH_EXPAND{** XE "**RB_TILECONTROL**:FAST_DEPTH_EXPAND" **}** = 1).

Stencil data inside the R400's depth cache is always stored as uncompressed even when stored as compressed in the Frame Buffer. The determination of whether to store Stencil compressed is made at the tile's stencil flush time.

*BOZO: Must store clear and compare with cache line to compress after the context is gone.*

## 3.3 Compressed Stencil Data (Frame Buffer only)

Compressed stencils are zero or four bit unsigned integers per stencil-sample; typically a >50% savings over uncompressed stencils. A depth surface may have stencil compressed if enabled (**RB_DEPTH_INFO.SMASK_ENABLE{** XE "**RB_DEPTH_INFO**:SMASK_ENABLE" **}** = 1). All of the stencil values within a tile are stored in the frame buffer with the same format; either 0-bit SMASK, compressed 4 bit or uncompressed 8 bit. Each eight by eight pixel tile is independently compressible, controlled by the tile's SMASK. The compressed stencil data is always stored contiguously on a per tile basis. Stencil compression is not supported when ZMASK is "**EXPANDED**".

Room for uncompressed eight bits per stencil-sample is always allocated in the frame buffer which becomes the fall back format when compression fails. When compression is enabled (**RB_DEPTHCONTROL.STENCIL_ENABLE = 1**{ XE "**RB_DEPTHCONTROL**:STENCIL_ENABLE" **}**) and (**RB_DEPTH_INFO.SMASK_ENABL**{ XE "**RB_DEPTH_INFO**:SMASK_ENABLE" **}E = 1**{ XE "**RB_DEPTH_INFO**:SMASK_ENABLE" **}**) and ZMASK != "**EXPANDED**" and the compression for the whole tile is successful ($0010_2 \le$ SMASK $\le 0111_2$), stencil data is stored as packed four bit stencil data in the frame buffer. When stencil compression is enabled but stencil compression has failed for any stencil-sample in the tile ($1001_2 \le$ SMASK $\le 1111_2$), the result is identical to having stencil compression disabled.

The R400 also supports "compressed" zero bit stencil values. If all the stencil values within a tile are equal to the stencil clear value (**RB_STENCIL_CLEAR.STENCIL_CLEAR**{ XE "**RB_STENCIL_CLEAR**:STENCIL_CLEAR" **}**) then SMASK is set to $0000_2$. If all the stencil values within a tile are equal to the stencil compare value (**RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**) then SMASK is set to $0001_2$. For either of these two cases the stencil values from the frame buffer are not read. This, of course, requires that stencils be enabled (**RB_DEPTHCONTROL.STENCIL_ENABLE = 1**{ XE "**RB_DEPTHCONTROL**:STENCIL_ENABLE" **}**) and that SMASK be enabled (**RB_DEPTH_INFO.SMASK_ENABLE**{ XE "**RB_DEPTH_INFO**:SMASK_ENABLE" **}** = 1).

Bit 0 of SMASK is used to denote when any stencils are equal to **RB_STENCIL_CLEAR.STENCIL_COMPAR**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}E**. Bit 1 of SMASK is used to denote when any stencils are less than **RB_STENCIL_CLEAR.STENCIL_COMPAR**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}E**. Bit 2 is used to denote when any stencils are greater than **RB_STENCIL_CLEAR.STENCIL_COMPAR**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}E**.

| SMASK | Num Bits | Description |
|---|---|---|
| $0000_2$ | 0 | Every stencil value in the tile is = **RB_STENCIL_CLEAR.STENCIL_CLEAR**{ XE "**RB_STENCIL_CLEAR**:STENCIL_CLEAR" **}**.{ XE "**RB_STENCIL_CLEAR**:STENCIL_CLEAR" **}** |
| $0001_2$ | 0 | Every stencil value in the tile is = **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $0010_2$ | 4 | Every stencil value in the tile is < **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $0011_2$ | 4 | Every stencil value in the tile is ≤ **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $0100_2$ | 4 | Every stencil value in the tile is > **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $0101_2$ | 4 | Every stencil value in the tile is ≥ **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $0110_2$ | 4 | Every stencil value in the tile is ≠ **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $0111_2$ | 4 | Some stencil values <, some >, some =. |
| $1000_2$ | n/a | *Reserved* |
| $1001_2$ | 8 | Every stencil value in the tile is = **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $1010_2$ | 8 | Every stencil value in the tile is < **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $1011_2$ | 8 | Every stencil value in the tile is ≤ **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $1100_2$ | 8 | Every stencil value in the tile is > **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $1101_2$ | 8 | Every stencil value in the tile is ≥ **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $1110_2$ | 8 | Every stencil value in the tile is ≠ **RB_STENCIL_CLEAR.STENCIL_COMPARE**{ XE "**RB_STENCIL_CLEAR**:STENCIL_COMPARE" **}**. |
| $1111_2$ | 8 | Some stencil values <, some >, some =. |

**Table 1: SMASK codes**

### 3.3.1 *Compressed Stencil format*

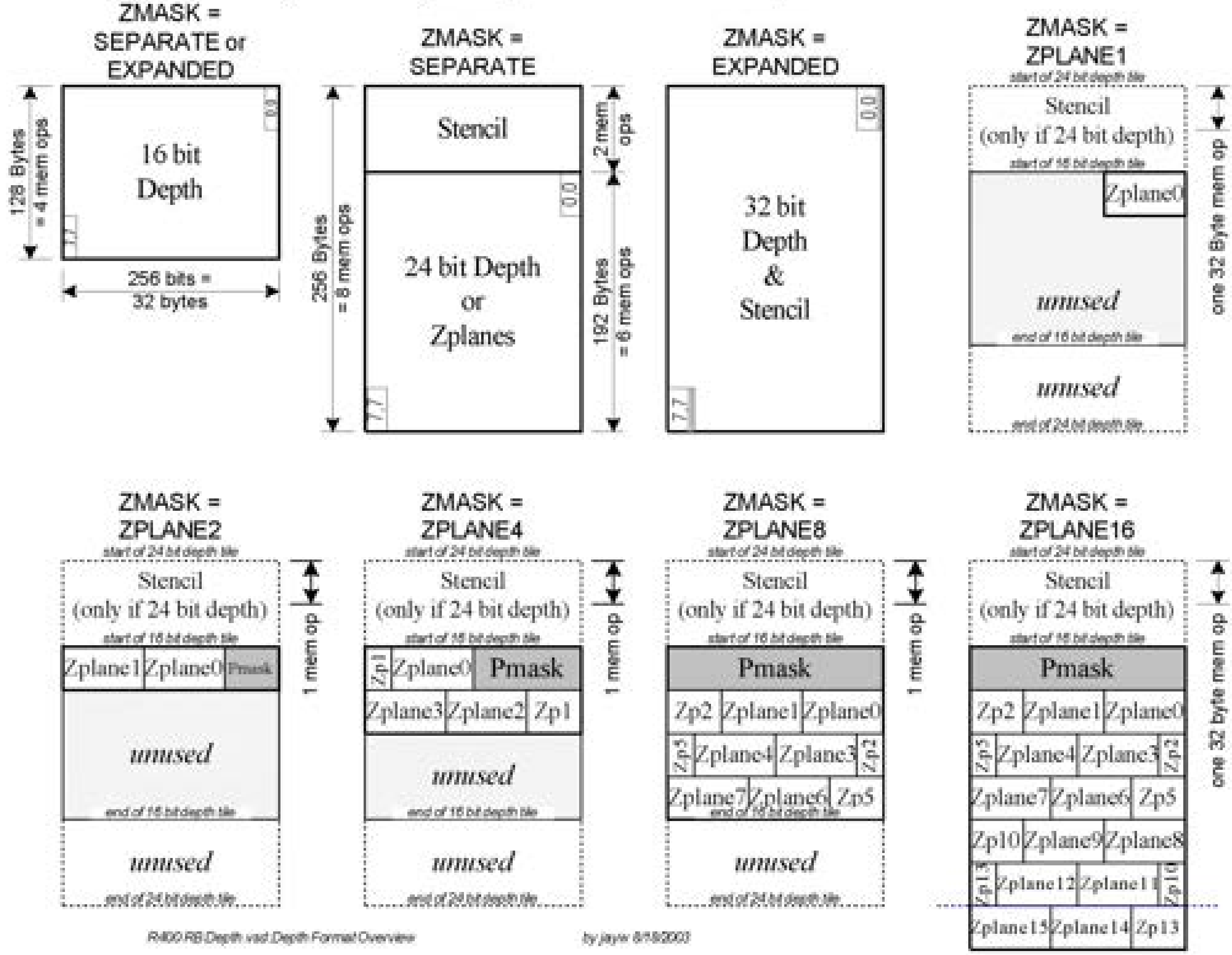
Stencil values are compressed by taking advantage of value locality. All of the stencil values within a tile have a high likelihood of having a limited range. Compression is possible with the introduction of an SMASK (Stencil MASK) per tile managed and cached in the RB's subblock Tile Logic (RBT). The format of a compressed stencil-sample is a four bit unsigned integer offset ([0…15]). The value of compressed stencil is calculated by adding the four bit compressed stencil value to the per context stencil min value (**RB_DEPTH_INFO.STENCIL_MIN{** XE **"RB_DEPTH_INFO:**STENCIL_MIN" **}**). Therefore the value of **RB_DEPTH_INFO.STENCIL_MI{** XE **"RB_DEPTH_INFO:**STENCIL_MIN" **}N** must not change after a surface has been initialized. NOTE: The value of **RB_DEPTH_INFO.STENCIL_MI{** XE **"RB_DEPTH_INFO:**STENCIL_MIN" **}N** must not exceed 240 (255-15) as the stencil comparisons are all 8 bit unsigned.

All of the stencil data for the tile is allocated and read into the depth cache when any compressible stencil data is required by the depth cache for a tile. This simplifies the steps required when stencil compression fails for any stencil in the tile. When stencil compression fails for **any** stencil value within a tile, **all** stencil data in the frame buffer must be written with their full eight bit values upon a tile depth cache flush. Stencil compression failure is detected during the depth cache's flushing of stencil data and the SMASK is updated. Stencil data inside the R400's depth cache is always stored as uncompressed.

When stencil is compressed to 0 bits ($0000_2 \leq$ SMASK $\leq 0001_2$) the stencil is not stored in the Frame Buffer but only in the tile's SMASK.

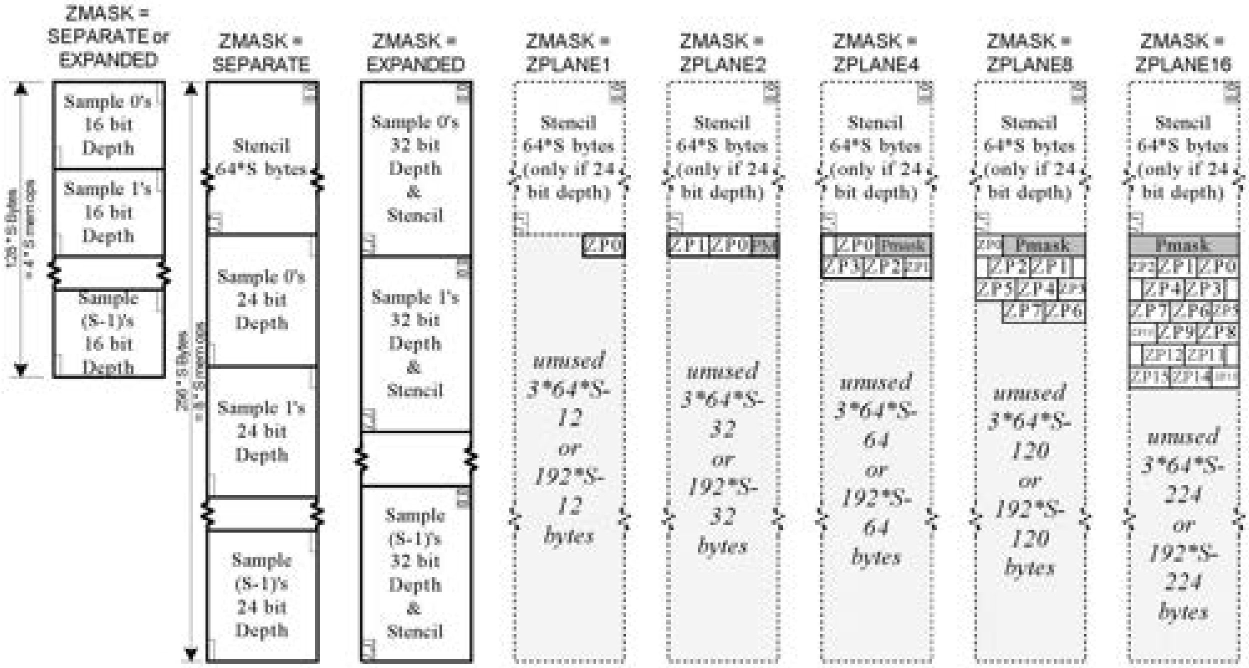
# 4. Depth Formats

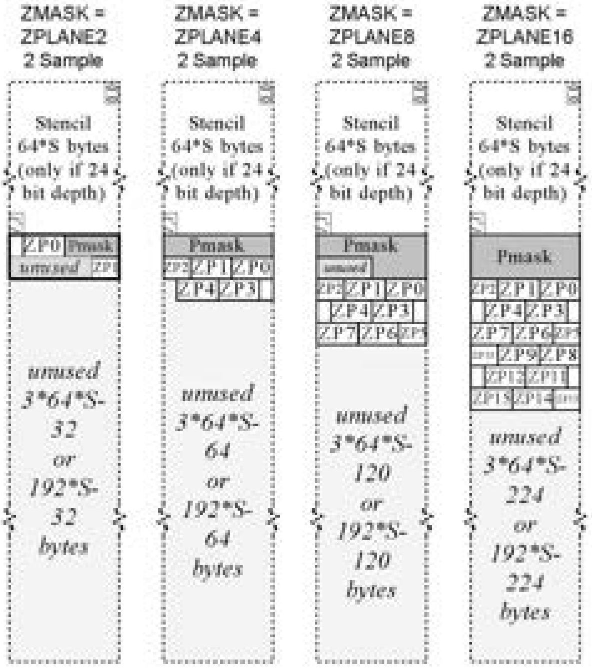
The Depth data for a tile has several variations in what Frame Buffer format it's stored in. Below in an overview of the stencil and depth partitioning within a single-sample tile's section of the depth surface.

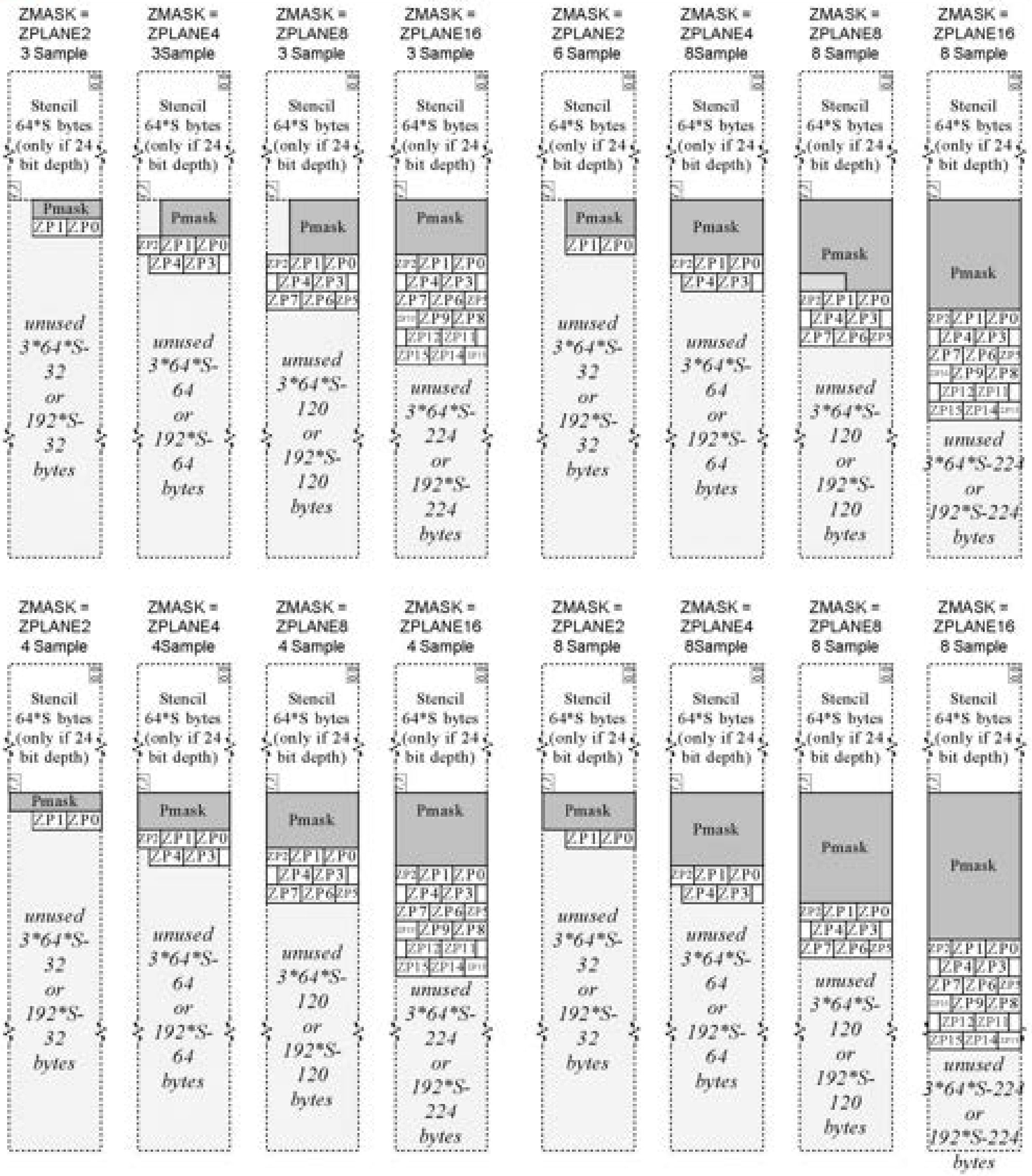


Note that the ZPLANE16 format does not fit within a single sample tile size, therefore the format ZPLANE16 is only used for multisample surfaces.

Multisample depth is stored as ‘S’ contiguous single-sample tile depth data sections per tile; where ‘S’ is the number of samples of the surface.







## 4.1 Depth 16-bit URF

Sixteen bit unsigned repeating fraction depth is the most basic and standard depth format. The value of the depth is: Depth = ((Depth$_{URF-16}$)/($2^{16}$-1)). The value range of 16 bit URF Depth is [0…1].

| 15 ... 0 |
|---|
| Depth, 16 bit URF |

R400 RB Depth.vsd Depth 16 URF
by jayw 8/18/2003

## 4.2 Depth, 24 bit URF

Twenty four bit unsigned repeating fraction depth is the normal 'high' precision depth format. The value of the depth is: Depth = $((\mathrm{Depth}_{URF\text{-}24})/(2^{24}-1))$. The value range of 24 bit URF Depth is [0…1].

R400 RB Depth vsd Depth, 24 bit URF
by jayw 8/18/2003

## 4.3 Depth, 24 bit Floating Point

The 24 bit floating point format is new and is designed to eliminate the need for a W buffer. (A W buffer stored the reciprocal of depth, giving it the characteristic of high precision for small depth values and low precision for high depth values. Switching to a floating point format meets the requirements of high precision for small depth values while sacrificing very little precision as compared with 24 bit URF.

The 24 bit floating point is based upon the IEEE 754-1985 32 bit floating point format. There is no sign and the range of valid depth values is [0…2). Values greater than 1.0 are invalid and neither generated nor supported, but the value of 1.0 is represented exactly; without resorting to repeating fraction 'trickery'. The mantissa is 20 bits with an implied leading integer one when the exponent is non-zero. The exponent is an unsigned integer with a bias of +15. If the exponent is zero then subnormalized numbers are encoded, just as in IEEE 754. The encodings for positive or negative infinities and NAN (Not A Number) are NOT supported.

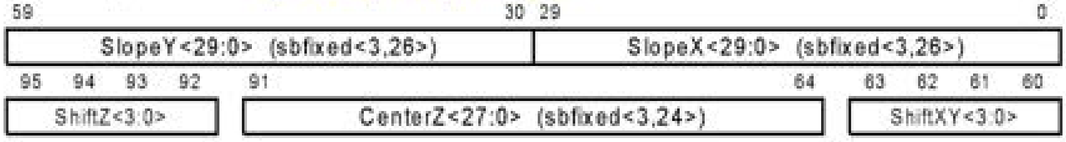
R400 RB Depth vsd Depth, 24 bit floating point
by jayw 8/18/2003

| Value | Value in Scientific Notation | 24-bit Floating Point | | Comment |
|---|---|---|---|---|
| | | Exp | Mantissa | |
| 1.00 | $+1.00000000000000000000_2 \times 2^{+0}$ | $1111_2$ | $00000000000000000000_2$ | Largest valid positive number |
| 0.75 | $+1.10000000000000000000_2 \times 2^{-1}$ | $1110_2$ | $10000000000000000000_2$ | |
| $2^{-1}$ | $+1.00000000000000000000_2 \times 2^{-1}$ | $1110_2$ | $00000000000000000000_2$ | |
| $2^{-13}$ | $+1.00000000000000000000_2 \times 2^{-13}$ | $0010_2$ | $00000000000000000000_2$ | |
| $2^{-14}$ | $+1.00000000000000000000_2 \times 2^{-14}$ | $0001_2$ | $00000000000000000000_2$ | Smallest normalized positive |
| $2^{-14}-2^{-34}$ | $+1.11111111111111111110_2 \times 2^{-15}$ | $0000_2$ | $11111111111111111111_2$ | Largest subnorm number |
| $2^{-15}$ | $+1.00000000000000000000_2 \times 2^{-15}$ | $0000_2$ | $10000000000000000000_2$ | |
| $2^{-16}$ | $+1.00000000000000000000_2 \times 2^{-16}$ | $0000_2$ | $01000000000000000000_2$ | |
| $2^{-33}$ | $+1.00000000000000000000_2 \times 2^{-33}$ | $0000_2$ | $00000000000000000010_2$ | |
| $2^{-34}$ | $+1.00000000000000000000_2 \times 2^{-34}$ | $0000_2$ | $00000000000000000001_2$ | Smallest positive subnorm |
| 0 | $+0.00000000000000000000_2 \times 2^{-15}$ | $0000_2$ | $00000000000000000000_2$ | ZERO |

**Table 2: Depth, 24-bit FP Format Examples**

## 4.4 Depth, 96 bit Zplane

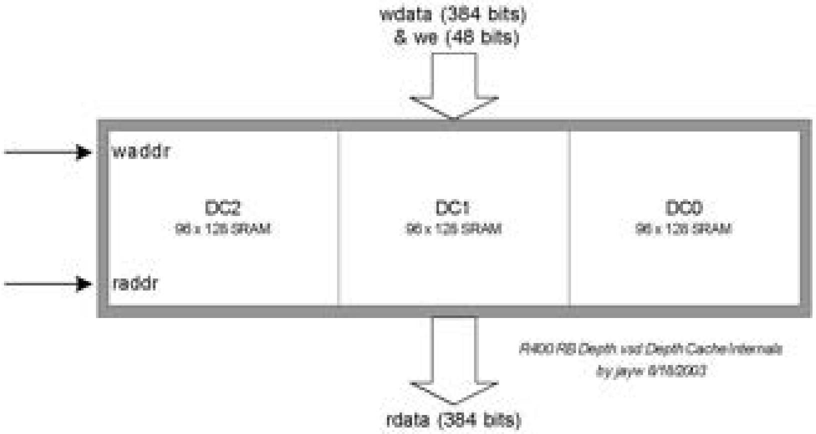
The definitive definition and description of Zplanes is in the document "//depot/r400/doc_lib/design/chip/memory/-R400_MemoryFormat.pdf" section "Zplane Depth Representation".



# 5. Depth Cache

The depth cache is constructed from three 96 x 128 SRAMs. 384 bits is a universally useful size for a depth and stencil cache '**word**'. Except for the LSB write and read address lines the write enable and the write and read address lines are common to all three SRAMs. The depth cache's read and write ports are the full 384 bits (48 Bytes) wide along with byte write enables.

The depth cache is logically divided into forty eight cache lines. The cache is fully associative using a 48 entry CAM. A cache line is 768 bits (96 Bytes). A single cache line may store a tile-sample's (1 Sample * 8x8 pixels) Pmask and stencil information, half a tile-sample's uncompressed depth or half a tile's compressed depth.



*R400 RB Depth.vsd Depth Cache Internals by jayw 8/18/2003*

Data to the RBM and from the RBM is run through an endian swap block. The functionality is defined by the context register **RB_DEPTH_INFO.DEPTH_ENDIAN{ XE "RB_DEPTH_INFO:DEPTH_ENDIAN" }**. Note: It's not clear that endian needs to be applied for any format other than 32 bit packed, as that's the only software usable format. *Even then it may not be needed. TBD.*

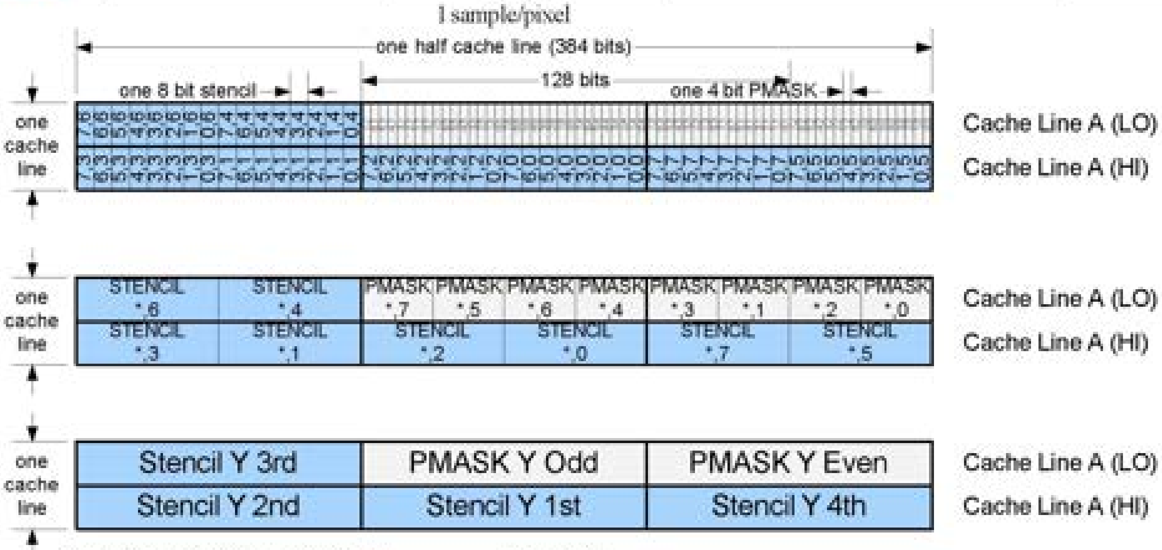
| DEPTH#_ENDIAN | Description |
|---|---|
| ENDIAN_NONE | 0xAABBCCDD -> 0xAABBCCDD |
| ENDIAN_8IN16 | 0xAABBCCDD -> 0xBBAADDCC |
| ENDIAN_8IN32 | 0xAABBCCDD -> 0xDDCCBBAA |
| ENDIAN_16IN32 | 0xAABBCCDD -> 0xCCDDAABB |

# 6. Depth Cache Formats

The data cache reads and writes four data cache words per cycle. The format of each data cache word in any cache line is identical.

## 6.1 Stencil & Pmask Depth Cache Format

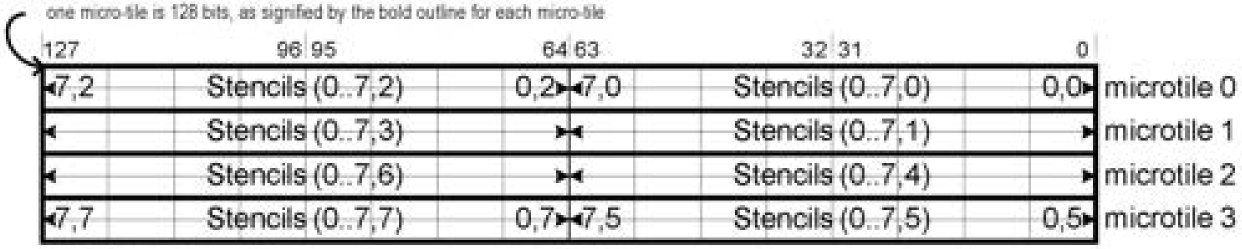
Stencil and Pmask data are always stored together in the same cache line. A depth cache line stores sixtyfour samples worth of Pmask and stencil data. Since PMASK may be stored compressed along with Zplanes all of the PMASK data for single sample and the Zplanes are stored in a separate cache line, all the PMASK data for a single sample tile must be writable into the depth cache within a single write operation. A single write operation into the depth cache is ½ cache line. For this reason the following cache line arrangement for the stencil and PMASK data was created:



*R400 RB Depth.vsd Depth Cache Line S&M 1sample* *by jayw 8/18/2003*

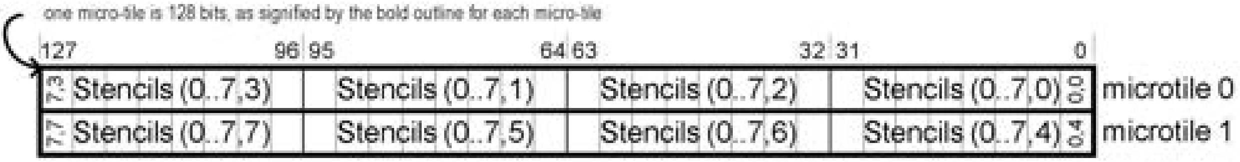
## 6.2 Single Sample Stencil & Pmask Frame Buffer Format

The uncompressed 8 bit stencil is stored just as 8 bit pixels are stored:



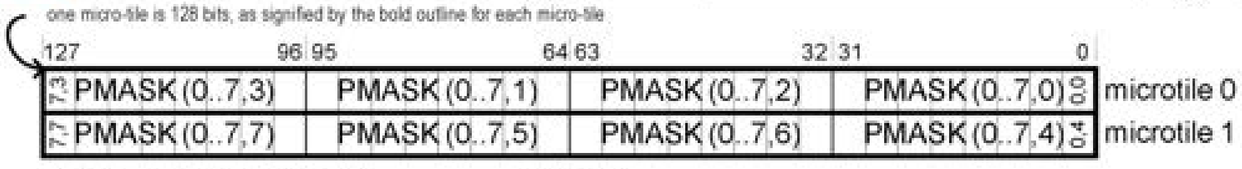
*R400 RB Depth.vsd 8 bit stencil FB format* *by jayw 8/18/2003*

The compressed 4 bit stencil is stored with the same order as 8 bit stencil, just that each stencil value is half size, as such:



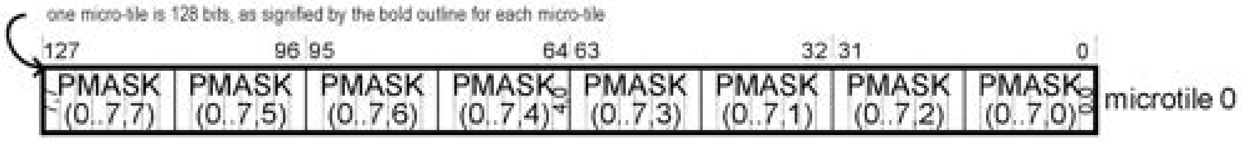
*R400 RB Depth.vsd 4 bit microtile FB format* *by jayw 8/18/2003*

The 4 bit and 3 bit PMASK data is stored with the same order and size as the compressed 4 bit stencil. The '3' bit PMASK values are always stored as 4 bit PMASK with a most significant 4th bit of zero.



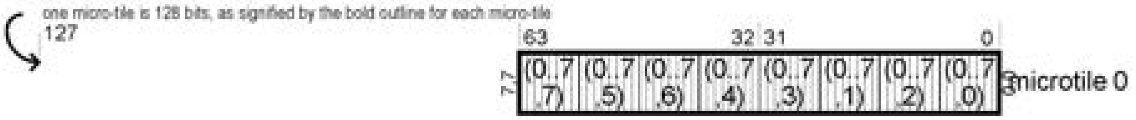
R400 RB Depth vsd 4 bit microtile PMASK FB Format by jayw 8/18/2003

The 2 bit PMASK data is stored with the same order as the compressed 4 bit stencil, but with just 2 bits per sample.



R400 RB Depth vsd 2 bit microtile PMASK FB Format by jayw 8/18/2003

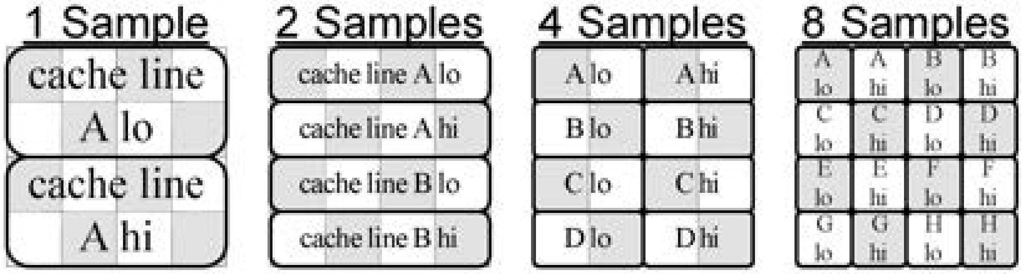
The 1 bit PMASK data is stored with the same order as the compressed 4 bit stencil, but with just 1 bit per sample.



R400 RB Depth vsd 1 bit microtile PMASK FB Format by jayw 8/18/2003

On a multisample surface all samples for pixel are contiguous in the frame buffer. Multisample tiles consume 'S' cache lines of stencil and Pmask. Effectively the data element size becomes 'S' bytes per pixel. For multisample surfaces of 1, 2, 4 and 8 samples/pixel, the resulting mapping is fairly simple.

Here is a representation of how a single tile is split into stencil depth cache line halves (1 depth cache line half = 1 depth cache read = 384 bits). I have checker boarded the quads within each tile for illustrative purposes. Note that every quad in the tile is contained within a single depth cache line half no matter whether the number of samples is 1, 2, 4 or 8. This allows stencil testing of an entire quad per cycle.

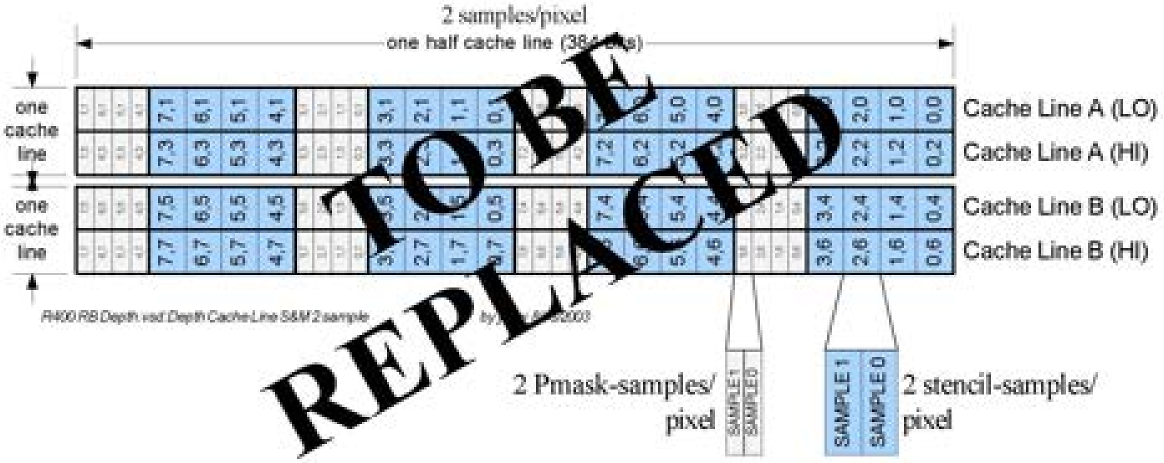


R400 RB Depth vsd Depth Cache Line S&M as Tiles by jayw 8/18/2003

| Num Samples | Tile Offset [8:0] | Cache Line Offset [4:0] | Cache Line Lo/Hi [5] | Cache Line Number [8:6] |
|---|---|---|---|---|
| 1 | $Y_2, Y_0, Y_1, X_2, X_1, X_0$ | $Y_0, Y_1, X_2, X_1, X_0$ | $Y_2$ | 0 |
| 2 | $Y_2, Y_1, Y_0, X_2, X_1, X_0$, 0 | $Y_0, X_2, X_1, X_0$, 0 | $Y_1$ | $Y_2$ |
| 3 | $\{Y_2, Y_1, X_2, X_1, X_0\}*3 >> 4$,<br>$Y_0$,<br>$\{Y_1, X_2, X_1, X_0\}*3$ & 0xF | $Y_0$,<br>$\{Y_1, X_2, X_1, X_0\}*3$ & 0xF | $(\{Y_2, Y_1, X_2, X_1, X_0\}*3 >> 4)$ & 1 | $\{Y_2, Y_1, X_2, X_1, X_0\}*3 >> 5$ |
| 4 | $Y_2, Y_1, X_2, Y_0, X_1, X_0$, 0, 0 | $Y_0, X_1, X_0$, 0, 0 | $X_2$ | $Y_2, Y_1$ |
| 6 | $\{Y_2, Y_1, X_2, X_1, X_0\}*3 >> 3$, | | $(\{Y_1, X_2, X_1, X_0\}*3 >>$ | $\{Y_2, Y_1, X_2, X_1, X_0\}*3$ |

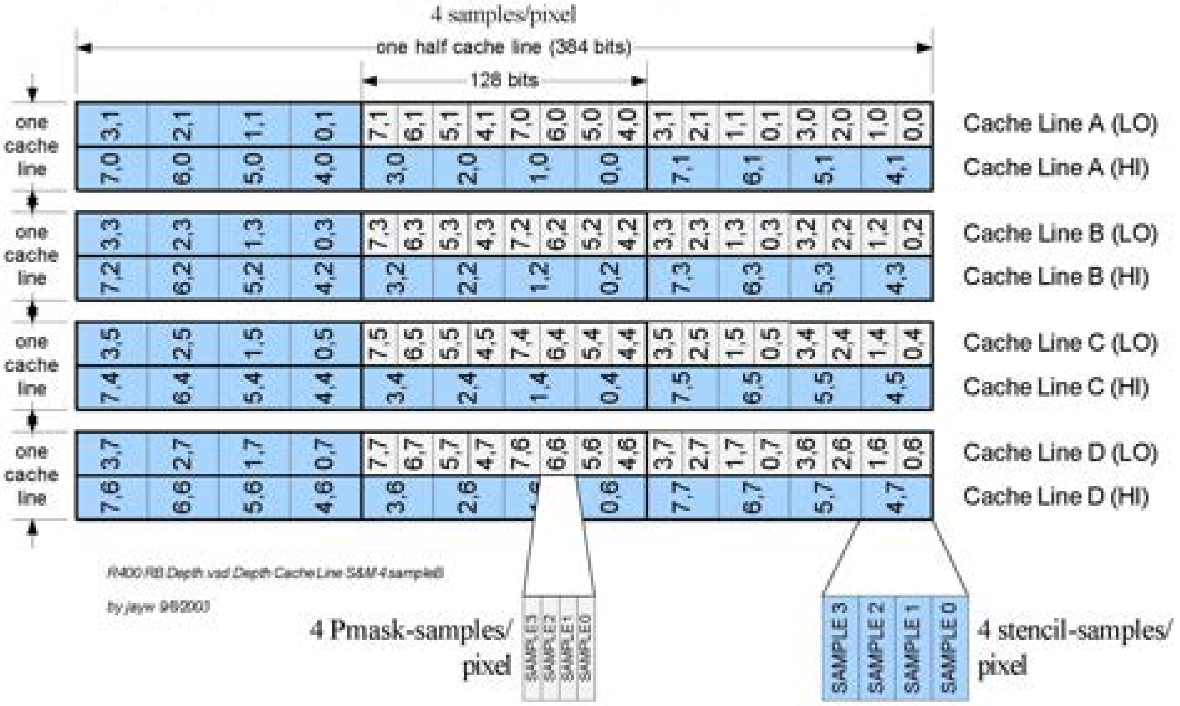
| | | | | |
|---|---|---|---|---|
| | $Y_0$,<br>$\{X_2, X_1, X_0\}$*3 & 0x7,<br>0 | $Y_0$,<br>$\{X_2, X_1, X_0\}$*3 & 0x7,<br>0 | 3) & 1 | >> 4 |
| 8 | $Y_2, Y_1, X_2, X_1, Y_0, X_0$, 0, 0, 0 | $Y_0, X_0$, 0, 0, 0 | $X_1$ | $Y_2, Y_1, X_2$ |

Here explicitly is the order of stencil information within a cache line. Please note that the stencil order is microtiled just as it is stored in memory. A 2 sample tile's stencil data takes 4 memory ops to read if uncompressed and 2 memory ops to read if compressed. Here is the case of 2 samples per pixel:



R400 RB Depth.vsd Depth CacheLine S&M 2 sample

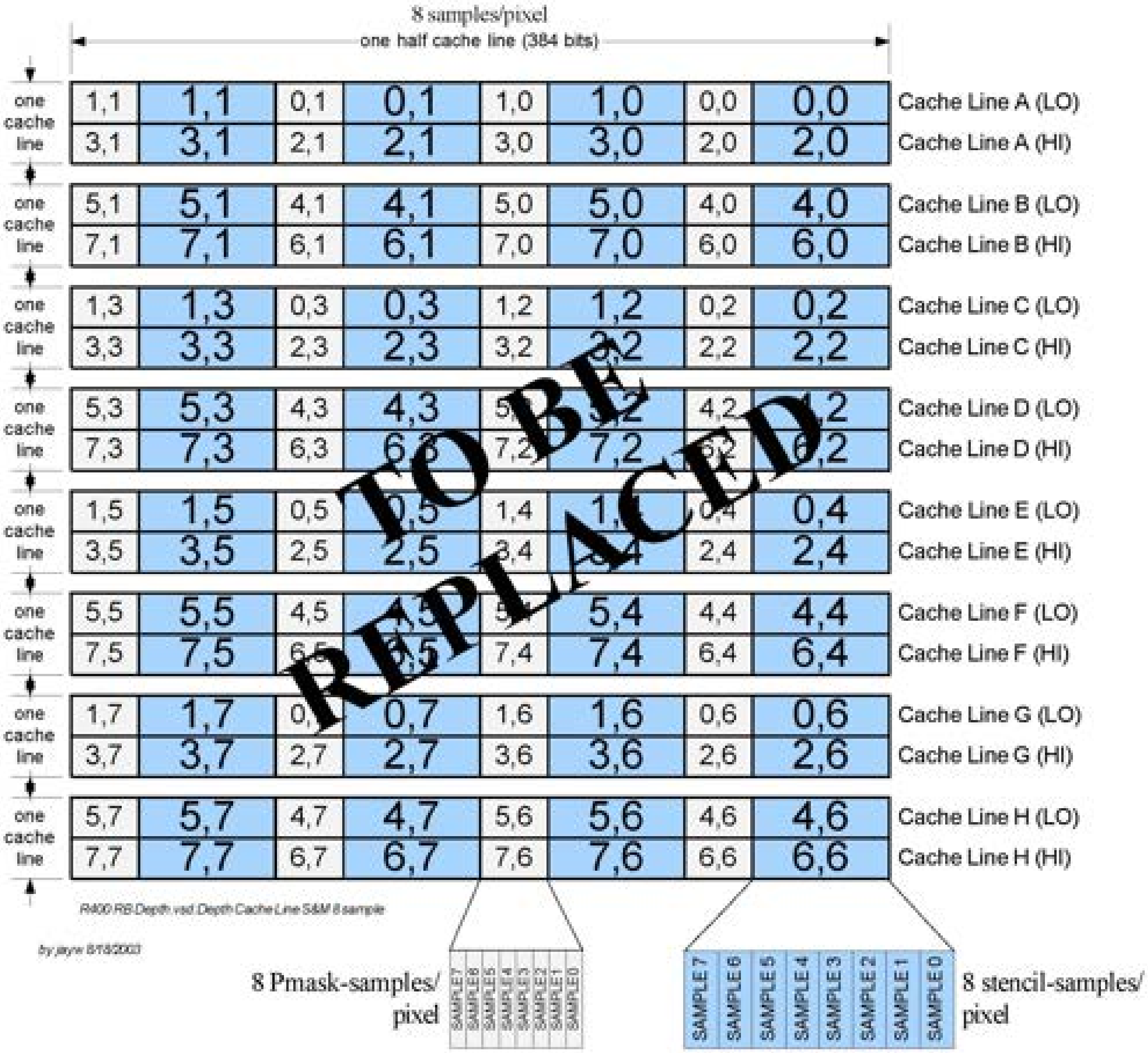
And 4 samples per pixel. Note: Two quads can be processed per cache read.



R400 RB Depth.vsd Depth CacheLine S&M 4sampleB

by jayw 9/8/2003

And finally 8 samples per pixel. Note: Only one 8 sample quad can processed per depth cache read.

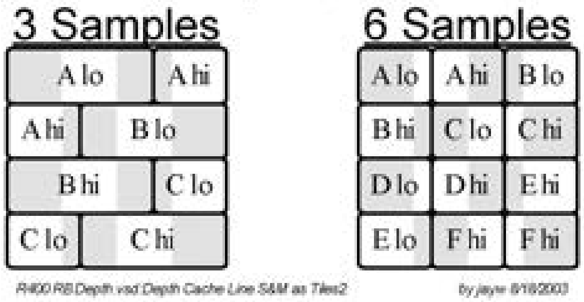


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| one cache line | 1,1 | 1,1 | 0,1 | 0,1 | 1,0 | 1,0 | 0,0 | 0,0 | Cache Line A (LO) |
| | 3,1 | 3,1 | 2,1 | 2,1 | 3,0 | 3,0 | 2,0 | 2,0 | Cache Line A (HI) |
| one cache line | 5,1 | 5,1 | 4,1 | 4,1 | 5,0 | 5,0 | 4,0 | 4,0 | Cache Line B (LO) |
| | 7,1 | 7,1 | 6,1 | 6,1 | 7,0 | 7,0 | 6,0 | 6,0 | Cache Line B (HI) |
| one cache line | 1,3 | 1,3 | 0,3 | 0,3 | 1,2 | 1,2 | 0,2 | 0,2 | Cache Line C (LO) |
| | 3,3 | 3,3 | 2,3 | 2,3 | 3,2 | 3,2 | 2,2 | 2,2 | Cache Line C (HI) |
| one cache line | 5,3 | 5,3 | 4,3 | 4,3 | 5,2 | 5,2 | 4,2 | 4,2 | Cache Line D (LO) |
| | 7,3 | 7,3 | 6,3 | 6,3 | 7,2 | 7,2 | 6,2 | 6,2 | Cache Line D (HI) |
| one cache line | 1,5 | 1,5 | 0,5 | 0,5 | 1,4 | 1,4 | 0,4 | 0,4 | Cache Line E (LO) |
| | 3,5 | 3,5 | 2,5 | 2,5 | 3,4 | 3,4 | 2,4 | 2,4 | Cache Line E (HI) |
| one cache line | 5,5 | 5,5 | 4,5 | 4,5 | 5,4 | 5,4 | 4,4 | 4,4 | Cache Line F (LO) |
| | 7,5 | 7,5 | 6,5 | 6,5 | 7,4 | 7,4 | 6,4 | 6,4 | Cache Line F (HI) |
| one cache line | 1,7 | 1,7 | 0,7 | 0,7 | 1,6 | 1,6 | 0,6 | 0,6 | Cache Line G (LO) |
| | 3,7 | 3,7 | 2,7 | 2,7 | 3,6 | 3,6 | 2,6 | 2,6 | Cache Line G (HI) |
| one cache line | 5,7 | 5,7 | 4,7 | 4,7 | 5,6 | 5,6 | 4,6 | 4,6 | Cache Line H (LO) |
| | 7,7 | 7,7 | 6,7 | 6,7 | 7,6 | 7,6 | 6,6 | 6,6 | Cache Line H (HI) |

R400 RB Depth.vsd:Depth Cache Line S&M 8 sample

by jayw 8/18/2003

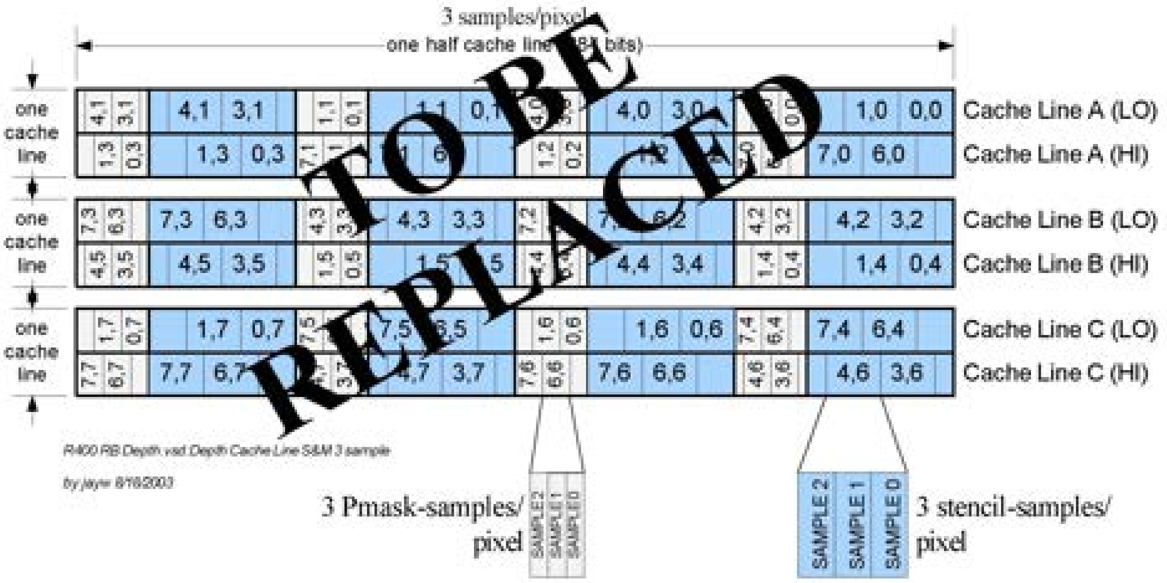
For multisample surfaces of 3 or 6 samples/pixel the arrangement causes some individual quads to span cache lines. It is for this case that the depth cache supports a split read; i.e. not all four rams receive the same address.



R400 RB Depth.vsd:Depth Cache Line S&M as Tiles2

by jayw 8/18/2003

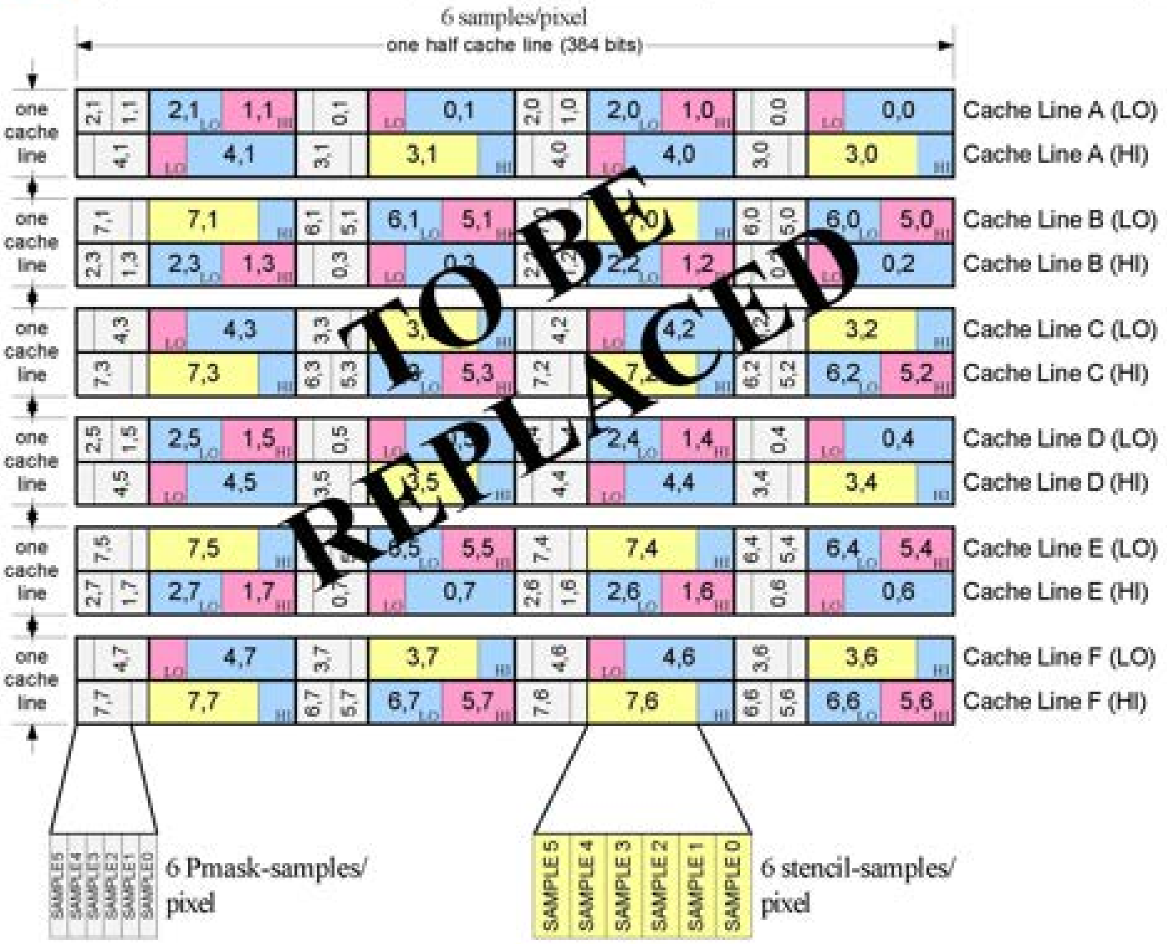
Here's the depth cache layout for 3 samples per pixel.



*R400 RB Depth vsd Depth Cache Line S&M 3 sample*

*by jayw 8/16/2003*

Here's 6 samples/pixel.



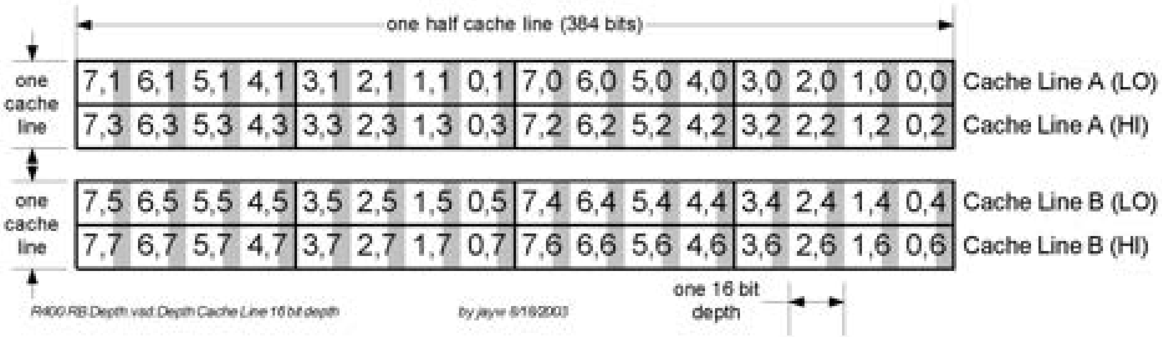
R400 RB Depth vad Depth Cache Line S&M 6 SampleB

by jayw 8/18/2003

## 6.3 16 bit Depth Cache Word Format

The 16 bit depth format is a pleasure, so easy to use. This format is always aligned in the frame buffer but is only valid with ZMASK formats "SEPARATE" and "EXPANDED". Cache lines of this format are stored in the frame buffer with unused padding to expand them to 24 bits. This makes reading and writing of 16 bit depths within the depth cache the same as for 24 bit depth values. A single cache line stores 32 16 bit depth values. Two cache lines store a full single sample tile's depth values.

R400 RB Depth.vsd Depth Cache Word 16 bit depth by jayw 8/18/2003

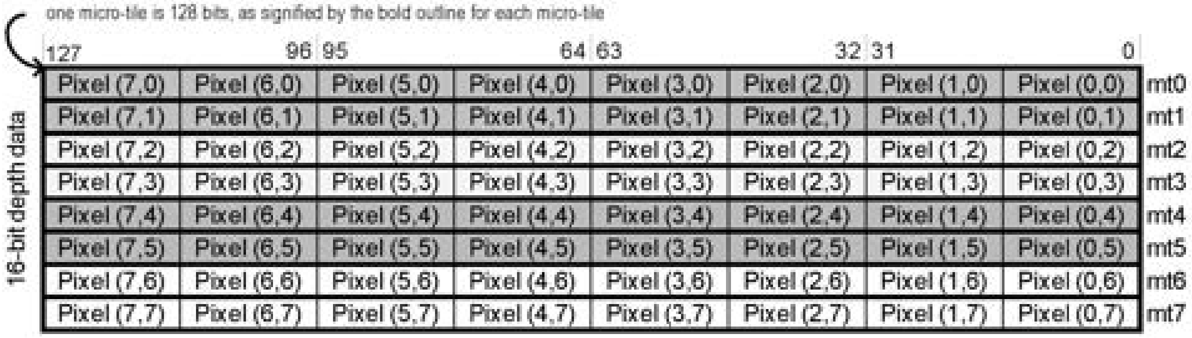


| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7,1 | 6,1 | 5,1 | 4,1 | 3,1 | 2,1 | 1,1 | 0,1 | 7,0 | 6,0 | 5,0 | 4,0 | 3,0 | 2,0 | 1,0 | 0,0 | Cache Line A (LO) |
| 7,3 | 6,3 | 5,3 | 4,3 | 3,3 | 2,3 | 1,3 | 0,3 | 7,2 | 6,2 | 5,2 | 4,2 | 3,2 | 2,2 | 1,2 | 0,2 | Cache Line A (HI) |
| 7,5 | 6,5 | 5,5 | 4,5 | 3,5 | 2,5 | 1,5 | 0,5 | 7,4 | 6,4 | 5,4 | 4,4 | 3,4 | 2,4 | 1,4 | 0,4 | Cache Line B (LO) |
| 7,7 | 6,7 | 5,7 | 4,7 | 3,7 | 2,7 | 1,7 | 0,7 | 7,6 | 6,6 | 5,6 | 4,6 | 3,6 | 2,6 | 1,6 | 0,6 | Cache Line B (HI) |

R400 RB Depth.vsd Depth Cache Line 16 bit depth by jayw 8/18/2003

The 16 bit depth surface has no stencil data and has the smallest depth surface size. The depth data is stored in microtiled order. Each 256 bit depth cache fill read fills just one half of a depth cache line as a single depth cache write. Multisample 16 bit depth surfaces store the single sample's depth values contiguously, and then followed by the next sample's depth values.

Total size is 128 Bytes per sample. This is the smallest depth surface.

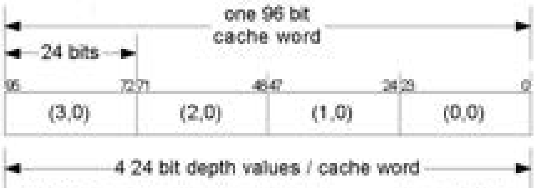


| 127 | 96 | 95 | 64 | 63 | 32 | 31 | 0 | |
|---|---|---|---|---|---|---|---|---|
| Pixel (7,0) | Pixel (6,0) | Pixel (5,0) | Pixel (4,0) | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) | mt0 |
| Pixel (7,1) | Pixel (6,1) | Pixel (5,1) | Pixel (4,1) | Pixel (3,1) | Pixel (2,1) | Pixel (1,1) | Pixel (0,1) | mt1 |
| Pixel (7,2) | Pixel (6,2) | Pixel (5,2) | Pixel (4,2) | Pixel (3,2) | Pixel (2,2) | Pixel (1,2) | Pixel (0,2) | mt2 |
| Pixel (7,3) | Pixel (6,3) | Pixel (5,3) | Pixel (4,3) | Pixel (3,3) | Pixel (2,3) | Pixel (1,3) | Pixel (0,3) | mt3 |
| Pixel (7,4) | Pixel (6,4) | Pixel (5,4) | Pixel (4,4) | Pixel (3,4) | Pixel (2,4) | Pixel (1,4) | Pixel (0,4) | mt4 |
| Pixel (7,5) | Pixel (6,5) | Pixel (5,5) | Pixel (4,5) | Pixel (3,5) | Pixel (2,5) | Pixel (1,5) | Pixel (0,5) | mt5 |
| Pixel (7,6) | Pixel (6,6) | Pixel (5,6) | Pixel (4,6) | Pixel (3,6) | Pixel (2,6) | Pixel (1,6) | Pixel (0,6) | mt6 |
| Pixel (7,7) | Pixel (6,7) | Pixel (5,7) | Pixel (4,7) | Pixel (3,7) | Pixel (2,7) | Pixel (1,7) | Pixel (0,7) | mt7 |

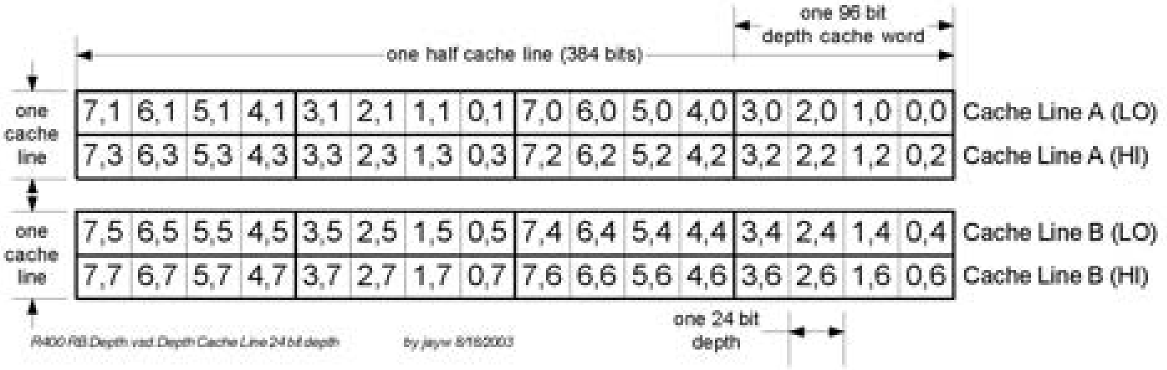
R400 RB Depth.vsd 16 bit FB format by jayw 8/18/2003

## 6.4 24 bit (URF or FP) Depth Cache Word Format

The 24 bit depth format fits well, but the filling from the MC and flushing to the MC involve complex shifting and possible merging with Pmask data. Internally it's a fairly clean arrangement. A data cache word stores four 24 bit depth values. A cache line stores 32 24 bit depth values. Two cache lines store a full single sample-tile's depth values. The 24 bit URF and 24 bit FP depth formats are indistinguishable in the depth cache.



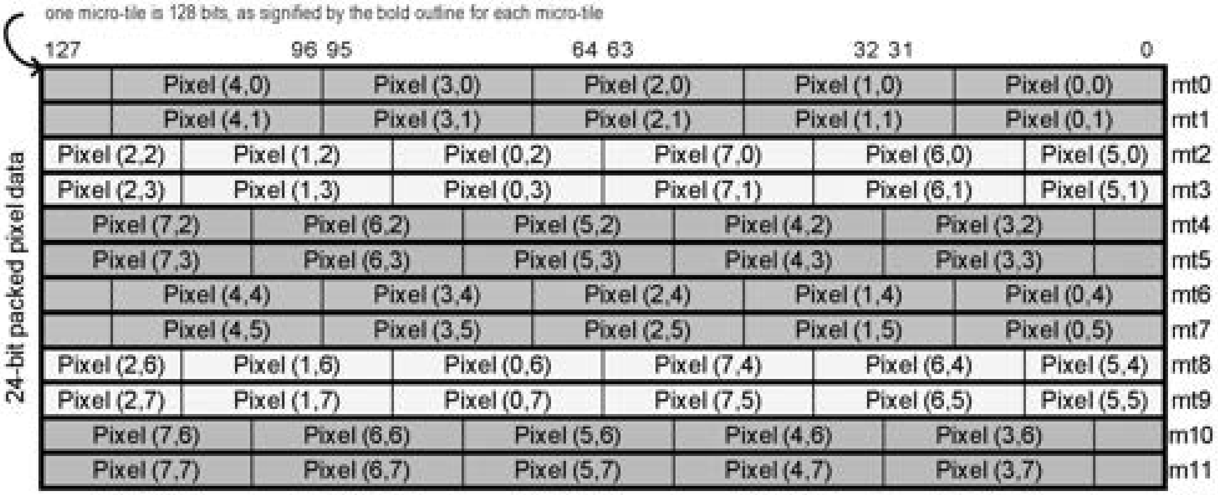
R400 RB Depth.vsd:Depth Cache Word 24 bit depth by jayw 8/16/2003



| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7,1 | 6,1 | 5,1 | 4,1 | 3,1 | 2,1 | 1,1 | 0,1 | 7,0 | 6,0 | 5,0 | 4,0 | 3,0 | 2,0 | 1,0 | 0,0 | Cache Line A (LO) |
| 7,3 | 6,3 | 5,3 | 4,3 | 3,3 | 2,3 | 1,3 | 0,3 | 7,2 | 6,2 | 5,2 | 4,2 | 3,2 | 2,2 | 1,2 | 0,2 | Cache Line A (HI) |
| 7,5 | 6,5 | 5,5 | 4,5 | 3,5 | 2,5 | 1,5 | 0,5 | 7,4 | 6,4 | 5,4 | 4,4 | 3,4 | 2,4 | 1,4 | 0,4 | Cache Line B (LO) |
| 7,7 | 6,7 | 5,7 | 4,7 | 3,7 | 2,7 | 1,7 | 0,7 | 7,6 | 6,6 | 5,6 | 4,6 | 3,6 | 2,6 | 1,6 | 0,6 | Cache Line B (HI) |

R400 RB Depth.vsd:Depth Cache Line 24 bit depth by jayw 8/16/2003

The arrangement in the Frame buffer is not so clean.



| | | | | | | |
|---|---|---|---|---|---|---|
| | Pixel (4,0) | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) | mt0 |
| | Pixel (4,1) | Pixel (3,1) | Pixel (2,1) | Pixel (1,1) | Pixel (0,1) | mt1 |
| Pixel (2,2) | Pixel (1,2) | Pixel (0,2) | Pixel (7,0) | Pixel (6,0) | Pixel (5,0) | mt2 |
| Pixel (2,3) | Pixel (1,3) | Pixel (0,3) | Pixel (7,1) | Pixel (6,1) | Pixel (5,1) | mt3 |
| Pixel (7,2) | Pixel (6,2) | Pixel (5,2) | Pixel (4,2) | Pixel (3,2) | | mt4 |
| Pixel (7,3) | Pixel (6,3) | Pixel (5,3) | Pixel (4,3) | Pixel (3,3) | | mt5 |
| | Pixel (4,4) | Pixel (3,4) | Pixel (2,4) | Pixel (1,4) | Pixel (0,4) | mt6 |
| | Pixel (4,5) | Pixel (3,5) | Pixel (2,5) | Pixel (1,5) | Pixel (0,5) | mt7 |
| Pixel (2,6) | Pixel (1,6) | Pixel (0,6) | Pixel (7,4) | Pixel (6,4) | Pixel (5,4) | mt8 |
| Pixel (2,7) | Pixel (1,7) | Pixel (0,7) | Pixel (7,5) | Pixel (6,5) | Pixel (5,5) | mt9 |
| Pixel (7,6) | Pixel (6,6) | Pixel (5,6) | Pixel (4,6) | Pixel (3,6) | | m10 |
| Pixel (7,7) | Pixel (6,7) | Pixel (5,7) | Pixel (4,7) | Pixel (3,7) | | m11 |

R400 RB Depth.vsd:24 bit microtile FB format colo by lseiler from R400_Memory_Tiles.vsd: FB24AllMicro

The 24 bit depth cache lines for a Tile are filled by sections. Each cache line requires three memory read operations. Memory reads return 5⅓ depth values per (128 bit) microtile. The three (256 bit) memory reads each ⅓ of a cache line. The 24 bit depth values are aligned at (0,0) and at (0,4). As shown below the cache lines are shown divided by three.

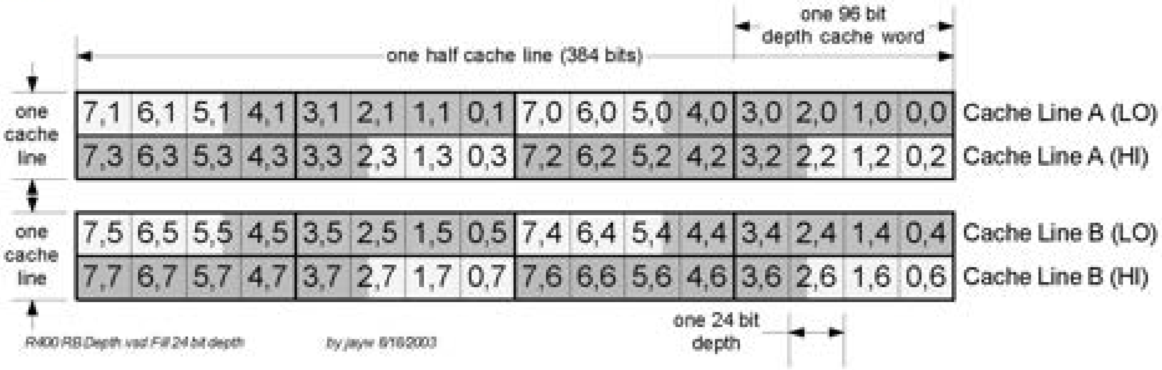
The depth data is read by order of X, Y.

The first read the X,Y range [(0,0)…(4,0) + one byte of (5,0)] and range [(0,1)…(4,1) + one byte of (5,1)]. This first fill requires only one cache line write.

The second read [(5,0)@byte2…(7,0)] and [(0,2)…(1,2) + two bytes of (2,2)] and [(5,1)@byte2…(7,1)] and [(0,3)…(1,3) + two bytes of (2,3)]. The second fill uses both available depth cache write cycles.

The third and last read fills [(2,2)@byte3 … (7,2)] and [(2,3)@byte3 … (7,3)]. The last fill only requires one cache line write.

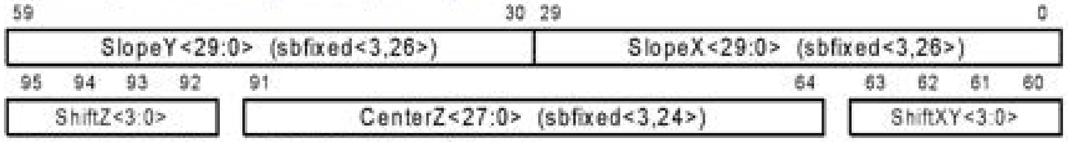
As the cache line is filled a per-cache-line fill count in the non-CAM tag is incremented. This fill count tells the data cache fill logic which third of the cache line to fill. The valid bit(s) (BOZO??) are turned upon completing the final fill of the cache line. The first cache line with the depths for pixels [(0,0)…(7,3)] is filled first.



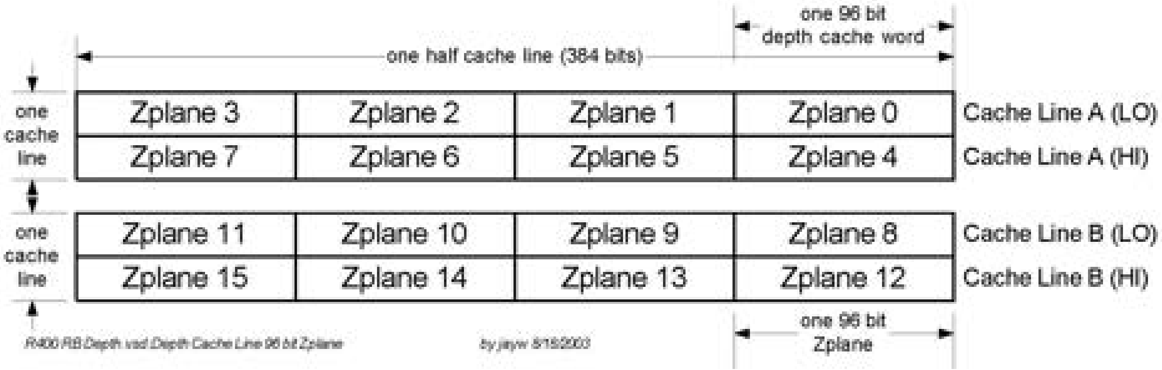
| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7,1 | 6,1 | 5,1 | 4,1 | 3,1 | 2,1 | 1,1 | 0,1 | 7,0 | 6,0 | 5,0 | 4,0 | 3,0 | 2,0 | 1,0 | 0,0 | Cache Line A (LO) |
| 7,3 | 6,3 | 5,3 | 4,3 | 3,3 | 2,3 | 1,3 | 0,3 | 7,2 | 6,2 | 5,2 | 4,2 | 3,2 | 2,2 | 1,2 | 0,2 | Cache Line A (HI) |
| 7,5 | 6,5 | 5,5 | 4,5 | 3,5 | 2,5 | 1,5 | 0,5 | 7,4 | 6,4 | 5,4 | 4,4 | 3,4 | 2,4 | 1,4 | 0,4 | Cache Line B (LO) |
| 7,7 | 6,7 | 5,7 | 4,7 | 3,7 | 2,7 | 1,7 | 0,7 | 7,6 | 6,6 | 5,6 | 4,6 | 3,6 | 2,6 | 1,6 | 0,6 | Cache Line B (HI) |

## 6.5 Pmasks & 96 bit Zplane Depth Cache Word Format

The definitive definition and description of Zplanes is in the document "//depot/r400/doc_lib/design/chip/memory/-R400_MemoryFormat.pdf" section "Zplane Depth Representation".



Two cache lines are used for storing the sixteen Zplanes:



| | | | | |
|---|---|---|---|---|
| Zplane 3 | Zplane 2 | Zplane 1 | Zplane 0 | Cache Line A (LO) |
| Zplane 7 | Zplane 6 | Zplane 5 | Zplane 4 | Cache Line A (HI) |
| Zplane 11 | Zplane 10 | Zplane 9 | Zplane 8 | Cache Line B (LO) |
| Zplane 15 | Zplane 14 | Zplane 13 | Zplane 12 | Cache Line B (HI) |

In the Frame Buffer Zplanes are painful to deal with. It's not just that Zplanes cross microtile and even the dual microtile read size. The Zplane format introduces the third data element of the depth cache, Zplane Pmasks (Plane-Masks). Each pixel sample's depth is represented by one of up to sixteen Zplanes. This requires that each pixel in the frame buffer have a four ([0…15]) bit Zplane Pmask for encoding it's representative depth. The Pmask is stored preceding the Zplane data and after the stencil data (if not 16 bit depth which has no stencil data). Since the Pmask is S*N*64 bits (where N is one, two or four bits per Pmask and S is one of {1,2,3,4,6,8}) there are combinations of N and S that will result in the Zplane depth data being not even microtile aligned. Note: Three bits per Pmask are actually stored and processed as four bits per Pmask.

The Pmask data is the first depth data read as the RBD uses this data early to determine the exact usage of the Zplanes by the tile. Along with the Pmask data some Zplane data is usually read, when (N*S)%4 != 0, i.e. when (N,S) is one of { (1,1), (2,1), (1,2), (1,3), (1,6) }.

> NOTE: The (N,S) combination of (4,1) is not allowed in the frame buffer. The frame buffer size for Zplane depth data is constrained to be less than or equal to the size that required by the depth being its native URF (16 or 24) bit format. For single sample this constrains the number of Zplanes to be:
>
> Samples*Size of Pmask/Tile + $N_{Zplane}$*(Size of Zplane) <= Samples*Size of 16 bit URF/Tile

$1*64*(4bits/depth) + N_{Zplane}*(96bits/Zplane) <= 1*64*16bits/depth$

$N_{Zplane} <= (1*64*16bits/depth – 1*64*(4bits/depth))/(96bits/Zplane)$

$N_{Zplane} <= 64(16-4)/96 = 2*12/3$

$N_{Zplane} <= 8$

Therefore while $N_{Zplane}$ may be 8 for N=4, S=1, it can not be 16, in this case. The spec (Perforce: //depot/R400/doc_lib/design/chip/memory/R400_MemoryFormat.doc) section "Zplane Storage Formats", table "Depth Storage Sizes in Micro-Tiles" fully analyzes this issue. The (N,S) combination of (4,1), aka >8 Zplanes for one sample surfaces, is the only combination that restricts using the R400 RB's full 16 Zplane capability. Single sample tiles inside the RB may have [8…16] Zplanes, however as they are flushed from the depth cache they will be written as explicit depth values and the Zmask is changed to "Separate".

## 6.6 CAM Tags

Cache line tags in the depth cache are not a pure physical address since the depth cache does not maintain the exact same organization as the frame buffer's depth surface. The CAM portion of a depth cache line tag is a composite of five fields.

R400 RB Depth.vsd Depth Cache CAM Entry by jayw 6/16/2003

The first field of the depth cache CAM tag is the top 25 bits of the physical address of the **start** of the 8x8 pixel tile's depth data. (Explanation: The smallest size of a data cache tile is 64 16 bit depth values or 128 bytes, therefore leaving out the bottom log2(128) bits of the 32 bit device address gives us bits 31:7, or 25 bits. All other sizes are multiples of 128 bytes.)

The second field is three bits and contains the data type:

| Code | Type | Description |
|---|---|---|
| 0 | DEPTH_LO | Depth for pixels (0,0)…(7,3) |
| 1 | DEPTH_HI | Depth for pixels (0,4)…(7,7) |
| 2 | STENCIL_PMASK | Stencil & Pmask |
| 3 | Reserved | reserved |

The third field is three bits for multisample surfaces and contains the sample number ([0…7]) of the depth information. Since the stencil and Pmask are stored as data elements of a size proportional to the number of samples for multisample surfaces, the 'sample number' is an offset into the surface's stencil data.

The fourth field is two dirty bits, one bit per cache line half. The dirty is one if any depth or stencil value (depending upon the 'Type' of the depth cache line) is required to be written to the frame buffer before the cache line contents are discarded. Note: The Pmask information is never 'dirty' and does not contribute to the status of this bit. The Pmask is written depending upon whether the depth data is dirty and whether it is of Zplane format. *BOZO: a third dirty bit would be optimal for ZMASK = SEPARATE/EXPANDED.*

The fifth field is the three valid bits. The meaning of the valid depends upon the data type of the cache line. (Explanation: Memory fills to the depth cache arrive as 256 bits; therefore 24 bit depth takes three reads to fill a cache line. Also Pmasks are read separately from the stencil data and only require only a single valid per cache line. Stencil data takes two 256 bit reads per cache line and therefore require two valids.)

| Format | Valids[2] (MSB) | Valids[1] | Valids[0] (LSB) |
|---|---|---|---|
| DEPTH_LO | Valid Depth for pixels (5,2)…(7,2) & (0,3)…(7,3) | Valid Depth for pixels (2,1)…(7,1) & (0,2)…(4,2) | Valid Depth for pixels (0,0)…(7,0) & (0,1)…(1,1) |
| DEPTH_HI | Valid Depth for pixels (5,6)…(7,6) & (0,7)…(7,7) | Valid Depth for pixels (2,5)…(7,5) & (0,6)…(4,6) | Valid Depth for pixels (0,4)…(7,4) & (0,5)…(1,5) |
| STENCIL_PMASK | Valid Pmask for pixels (0,0)…(7,7) | Valid Stencil for pixels (0,4)…(7,7) | Valid Stencil for pixels (0,0)…(7,3) |
| Reserved | *reserved* | *reserved* | *reserved* |

The state of a cache line:

| Valid | Dirty | Inflight Count | Description |
|---|---|---|---|
| 0 | 0 | 0 | Empty, available for reuse.<br>Occurs only after a hard/soft reset or flush & invalidate. |
| 0 | 0 | >0 | Empty and in use, awaiting to be filled/marked-valid. |
| 0 | 1 | n/a | *not a valid cache line state.* |
| 1 | 0 | 0 | Valid, available for reallocation |
| 1 | 0 | >0 | Valid and in use. |
| 1 | 1 | 0 | Valid and Dirty, must be flushed before reuse. |
| 1 | 1 | >0 | Valid, Dirty and in use. |

## 6.7 Depth Surface Device Addresses

The address calculations for the depth cache are quite complex. Starting from the general equations (Perforce //depot/r400/doc_lib/design/chip/memory/R400_MemoryFormat.pdf) we can simplify and reduce to the required forms.

Depth surface dimensionality effects address generation. The depth surface dimensionality of a depth surface is the context register field **RB_COLOR0_INFO.COLOR0_ARRA{** XE "**RB_COLOR0_INFO**:COLOR0_ARRAY" **}Y** which may be either **ARRAY_2D** or **ARRAY_3D_SLICE**. If depth or stencil are enabled then the field **RB_COLOR0_INFO.COLOR0_TILING{** XE "**RB_COLOR0_INFO**:COLOR0_TILING" **}** must be **ARRAY_TILED**. A depth surface may not be linear or 1D.

### 6.7.1 *Tile starting device address*

The complexity of calculating the start of a tile's depth data in its depth surface is greatly simplified by several facts.

1. The X and Y quad and pixel offset bits are zero, i.e. X[2:0] and Y[2:0] are zero.
2. 16 bit depth surfaces have no stencil and are fixed with a data element 'Size' of 2 bytes per pixel.
3. 24 bit depth surfaces have stencil and are fixed with a data element 'Size' of 4 bytes per pixel.
4. There is no offset within the tile since we're just calculating the starting address, i.e. 'TileBase' is 0.
5. TileSize is (16 or 32 bits/pixel) * 64 pixels * 'S' samples = 128*S (16 bit depth) or 256*S bytes (24 bit depth) per tile.
6. 'DataSize' is irrelevant for the tile starting address.
7. Depth surfaces are always tiled.
8. Z is a three bit slice value, from **RB_COLOR0_INFO.COLOR0_SLIC{** XE "**RB_COLOR0_INFO**:COLOR0_SLICE" **}E**. Note: "*This value applies to all color buffers and the depth buffer.*"
9. 'SURFACE_PITCH' is **RB_SURFACE_EXTENT.SURFACE_PITCH{** XE "**RB_SURFACE_EXTENT**:SURFACE_PITCH" **}**.
10. 'SURFACEBASE' is **RB_DEPTH_BASE.DEPTH_BAS{** XE "**RB_DEPTH_BASE**:DEPTH_BASE" **}E**.

Starting from the general equations and reducing…

The original equations for each field are shown in italics on the first equation line. The reduced results using the above follow the original equations and are expanded, if necessary, according to the number of pipes. The use of log2(Pipes) and log2(Size) are used often; the log2(16 bit depth) is 1 (Byte) and log2(32 bit depth) is 2 (Bytes). Curly braces '{' and '}' are used to denote bit concatenation into a resulting multibit value. The subscripts are used to denote the particular bit of the X, Y or Z tile coordinates.

| Field Name | Bits | Pip-es | Equation |
|---|---|---|---|
| TileSize | 6 + log2(Size) | - | *64*Size* |
| TileBase | 6 + log2(Size) | | 0 |
| BankSelect2D | 1 | - | *(Y/16) mod 2* |
| | 1 | - | **$Y_4$** |
| BankSelect3D | 1 | - | *(Y/8 + Z/4) mod 2* |
| | 1 | - | **$Y_3$^$Z_2$** |
| MemSelect2D | log2(Pipes) | - | *(X/8 + (Y/8 mod 2)*(Pipes/2)) mod Pipes* |
| | 0 | 1 | ***NULL-ZERO-BITS*** |
| | 1 | 2 | **$X_3$^$Y_3$** |
| | 2 | 4 | **{$X_4$^$Y_3$, $X_3$}** |
| MemSelect3D | log2(Pipes) | - | *(X/8 + BankSelect*(Pipes/2)) mod Pipes* |
| | 0 | 1 | ***NULL-ZERO-BITS*** |
| | 1 | 2 | **$X_3$^$Y_3$^$Z_2$** |
| | 2 | 4 | **{$X_4$^$Y_3$^$Z_2$, $X_3$}** |
| Subset2D | 1+log2(Pipes) | - | *BankSelect2D + 2*MemSelect2D* |
| | 1 | 1 | **$Y_4$** |
| | 2 | 2 | **{$X_3$^$Y_3$,$Y_4$}** |
| | 3 | 4 | **{$X_4$^$Y_3$, $X_3$, $Y_4$}** |
| Subset3D | 1+log2(Pipes) | - | *BankSelect2D + 2*MemSelect2D* |
| | 1 | 1 | **$Y_3$^$Z_2$** |
| | 2 | 2 | **{ $X_3$^$Y_3$^$Z_2$, $Y_3$^$Z_2$}** |
| | 3 | 4 | **{$X_4$^$Y_3$^$Z_2$, $X_3$, $Y_3$^$Z_2$}** |
| TileNumber2D | 3-log2(Pipes) | - | *((X mod 32)/8/Pipes)*2 + (Y/8 mod 2)* |
| | 3 | 1 | **$X_4$, $X_3$, $Y_3$** |
| | 2 | 2 | **$X_4$, $Y_3$** |
| | 1 | 4 | **$Y_3$** |
| TileNumber3D | 4-log2(Pipes) | - | *(Z mod 4) + ((X mod 32)/8/Pipes)*4* |

| | | | |
|---|---|---|---|
| | 4 | 1 | **{$X_4$, $X_3$, $Z_1$, $Z_0$}** |
| | 3 | 2 | **{$X_4$, $Z_1$, $Z_0$}** |
| | 2 | 4 | **{$Z_1$, $Z_0$}** |
| MicroByte2D&3D | 5 + log2(Size) | - | _(X mod 8 + ((Y mod 8)/2)*8)*Size_ |
| | 5 + log2(Size) | - | **0** |
| TileOffset2D&3D | 6 + log2(Size) | - | _(MicroByte2D&3D mod 16) + (Y mod 2)*16 + (MicroByte2D&3D/16)*32_ |
| | 6 + log2(Size) | - | **0** |
| TileAddr2D&3D | 6 + log2(Size) | - | TileBase + TileOffset2D&3D |
| | | | **0** |
| MacroOffset2D | 16 | - | _(X/32) + (Y/32) * (Pitch/32);_ |
| | 16 | - | **X[12:5] + Y[12:5]*SURFACE_PITCH[13:5]** |
| MacroOffset3D | 16 | - | _(X/32) + (Pitch/32) * ((Y/16) + (Height/16)*(Z/4))_ |
| | 16 | - | **X[12:5] + (Y[12:4]+ SURFACE_HEIGHT[13:4]*Z[2])*SURFACE_PITCH[13:5]** |
| SubsetOffset2D | [20…3]+ log2(TileSize) - log2(Pipes) | - | _MacroOffset2D*TileSize*16/Subsets + TileNumber2D*TileSize + TileAddr2D_ |
| | | - | TileSize*(MacroOffset2D* 16/Subsets + TileNumber2D) |
| | | - | TileSize*((Pipes ? 2,4,8)* MacroOffset2D + (Pipes ? $Y_3$, {$X_4$, $Y_3$}, {$X_4$, $X_3$, $Y_3$}) |
| | | - | TileSize*((Pipes ? {MacroOffset2D,Y[3]}, {MacroOffset2D, $X_4$,$Y_3$}, {MacroOffset2D, $X_4$, $X_3$, $Y_3$}) |
| | | 1 | **({MacroOffset2D, $X_4$, $X_3$, $Y_3$}) << log2(TileSize)** |
| | | 2 | **({MacroOffset2D, $X_4$, $Y_3$}) << log2(TileSize)** |
| | | 4 | **({MacroOffset2D, $Y_3$} << log2(TileSize)** |
| SubsetOffset3D | [20…3]+ log2(TileSize) -log2(Pipes) | - | _MacroOffset3D*TileSize*32/Subsets + TileNumber3D*TileSize + TileAddr3D_ |
| | | - | TileSize*(MacroOffset3D*32/Subsets + TileNumber3D) |
| | | - | TileSize*((Pipes ? 4,8,16)*MacroOffset3D + (Pipes ? {$Z_1$, $Z_0$}, {$X_4$, $Z_1$, $Z_0$}, {$X_4$, $X_3$, $Z_1$, $Z_0$})) |
| | | - | TileSize*(Pipes ? {MacroOffset3D, $Z_1$, $Z_0$}, {MacroOffset3D, $X_4$, $Z_1$, $Z_0$}, {MacroOffset3D, $X_4$, $X_3$, $Z_1$, $Z_0$}) |
| | | 1 | **{MacroOffset3D, $X_4$, $X_3$, $Z_1$, $Z_0$} << log2(TileSize)** |
| | | 2 | **{MacroOffset3D, $X_4$, $Z_1$, $Z_0$} << log2(TileSize)** |
| | | 4 | **{MacroOffset3D, $Z_1$, $Z_0$} << log2(TileSize)** |

Whew! Now we can calculate the 'LocalAddr' using a single MAC (Multiply and Accumulate) and some muxing.

| Field Name | Bits | Pip-es | Equation |
|---|---|---|---|
| LocalAddr2D | 31 | - | _SurfaceBase/Subsets + SubsetOffset2D_ |
| | | | SURFACEBASE[31:12]<<(11 - log2(Pipes)) + ((Pipes ? {MacroOffset2D,$Y_3$}, {MacroOffset2D, $X_4$,$Y_3$}, {MacroOffset2D, $X_4$, $X_3$,$Y_3$}) << log2(TileSize) |
| | | | SURFACEBASE[31:12]<<(11 - log2(Pipes)) +<br>MacroOffset2D << (3+log2(TileSize)-log2(Pipes)) +<br>((Pipes ? $Y_3$, {$X_4$,$Y_3$}, {$X_4$, $X_3$,$Y_3$}) << log2(TileSize) |
| | | | **(SURFACEBASE[31:12]<<(8 – log2(TileSize)) + MacroOffset2D)**<br>**<< (3+log2(TileSize)-log2(Pipes)) +**<br>**((Pipes ? $Y_3$, {$X_4$,$Y_3$}, {$X_4$, $X_3$,$Y_3$}) << log2(TileSize)** |
| | | | (SURFACEBASE[31:12]<<(8 – log2(TileSize)) + MacroOffset2D)<br><< (2 - log2(Pipes)) + (Pipes ? _null-0_, $X_4$, {$X_4$, $X_3$ }) << (1+log2(TileSize)) +<br>$Y_3$ << log2(TileSize) _// Note the LSB is invariant_ |
| | | | **{MAC2D, (Pipes ? $Y_3$, {$X_4$,$Y_3$}, {$X_4$, $X_3$,$Y_3$})} << log2(TileSize)** |
| | | | |
| LocalAddr3D | 31 | | _SurfaceBase/Subsets + SubsetOffset3D_ |
| | | | SURFACEBASE[31:12]<<(11 - log2(Pipes)) + (Pipes ? {MacroOffset3D, $Z_1$, $Z_0$}, {MacroOffset3D, $X_4$, $Z_1$, $Z_0$}, {MacroOffset3D, $X_4$, $X_3$, $Z_1$, $Z_0$})<< log2(TileSize) |
| | | | SURFACEBASE[31:12]<<(11 - log2(Pipes)) +<br>MacroOffset3D << (4+log2(TileSize)-log2(Pipes)) +<br>(Pipes ? {$Z_1$, $Z_0$}, {$X_4$, $Z_1$, $Z_0$}, {$X_4$, $X_3$, $Z_1$, $Z_0$})<< log2(TileSize) |
| | | | (SURFACEBASE[31:12]<<(7 - log2(TileSize)) +MacroOffset3D)<br><< (4+log2(TileSize)-log2(Pipes)) +<br>(Pipes ? {$Z_1$, $Z_0$}, {$X_4$, $Z_1$, $Z_0$}, {$X_4$, $X_3$, $Z_1$, $Z_0$})<< log2(TileSize) |
| | | | **(SURFACEBASE[31:12]<<(8 - log2(TileSize)) +(MacroOffset3D<<1) +**<br>**(Pipes ? $Z_1$, $X_4$, $X_4$)) << (3+log2(TileSize)-log2(Pipes)) +**<br>**(Pipes ? $Z_0$, {$Z_1$, $Z_0$}, {$X_3$, $Z_1$, $Z_0$})<< log2(TileSize)** |
| | | | ((SURFACEBASE[31:12]<<(8 - log2(TileSize)) +(MacroOffset3D<<1)) << (2-log2(Pipes))<br>+ (Pipes ? $Z_1$, {$X_4$, $Z_1$}, {$X_4$, $X_3$, $Z_1$}))<< (1+log2(TileSize))<br>$Z_0$<< log2(TileSize) _// Note the LSB is invariant_ |
| | | | **{(MAC3D), (Pipes ? $Z_0$, {$Z_1$, $Z_0$}, {$X_3$, $Z_1$, $Z_0$})} << log2(TileSize)** |

| | | | |
|---|---|---|---|
| **MAC2D** | **21** | | **(SURFACEBASE[31:12]<<(8 – log2(TileSize)) +**<br>**X[12:5] +**<br>**Y[12:5]*SURFACE_PITCH[13:5])** |
| | | | |
| **MAC3D** | **21** | | **(SURFACEBASE[31:12]<<(8 - log2(TileSize)) +**<br>**((X[12:5]<<1 \| (Pipes ? $Z_1$, $X_4$, $X_4$)) +**<br>**(Y[12:4]*SURFACE_PITCH[13:5])<<1 +**<br>**(Z[2]*SURFACE_HEIGHT[13:4]*SURFACE_PITCH[13:5])<<1** |
| | | | |

The MAC plus a small 3-to-1 mux (based on the dimension and depth size) gives LocalAddr. The final Device Address is just one more level of muxing (based on the number of pipes):

| Field Name | Bits | Pipes | Equation |
|---|---|---|---|
| DeviceAddress [5:0] | 6 | - | *LocalAddr[5:0]* |
| | 6 | - | **0** |
| DeviceAddress [log2(Pipes) +5:6] | log2(Pipes) | - | *MemSelect2D or MemSelect3D* |
| | | | **2D: (Pipes ? {$X_4$^$Y_3$, $X_3$}, $X_3$^$Y_3$, NULL)**<br>**3D: (Pipes ? {$X_4$^$Y_3$^$Z_2$, $X_3$}, $X_3$^$Y_3$^$Z_2$, NULL)** |
| DeviceAddress [6+log2(Pipes)] | 1 | - | *LocalAddr[6]* |
| | 1 | - | **0** |
| DeviceAddress [10:7+log2(Pipes)] | 4-log2(Pipes) | - | *LocalAddr[10-log2(Pipes):7]* |
| | | | |
| DeviceAddress [11] | 1 | - | *BankSelect2D or BankSelect3D* |
| | | - | **2D: $Y_4$; 3D: $Y_3$^$Z_2$** |
| DeviceAddress [31:12] | 20 | - | *LocalAddr[30-log2(Pipes) : 11-log2(Pipes)]* |
| | | - | |

### 6.7.2 *TileBase*

The 'TileBase' is used when accessing 24 bit depth surfaces to skip over the stencil data or to skip over the previous sample's depth data. or multisample 16 bit depth surfaces.

| Depth Size | Zmask | TileBase (bytes) | From Start of Tile to beginning of: |
|---|---|---|---|
| 16 | - | 0 | Tile |
| 16 | SEPARATE<br>EXPANDED | 128*(S-1) | The depth data for sample 'S'. |
| 16 | ZPLANE(*N*) | 0 | The depth data. |
| 24 | - | 0 | Tile and also start of Stencil |
| 24 | EXPANDED | 256*(S-1) | The depth data for sample 'S'. |
| 24 | SEPARATE | 320*S-256 | The depth data for sample 'S'. (64*S+256*(S-1)) |
| 24 | ZPLANE(*N*) | 64*S | The Pmask data. |
| 24 | ZPLANE(*N*) | (64+8*log2(*N*))*S | The depth data. (64*S+8*log2(*N*)*S) |

### 6.7.3 *DataSize*

The 'DataSize' along with 'TileBase' is used in the general addressing equations to access individual data elements within a depth tile.

| Depth Size (bits) | Zmask | Depth 'DataSize' (bytes) | Stencil 'DataSize' (bytes) |
|---|---|---|---|
| 16 | - | 2 | N/A |
| 24 | EXPANDED | 4 | N/A |
| 24 | SEPARATE | 3 | 64*'S' |
| 24 | ZPLANE(*N*) | 12***bozo *not really per pixel data* | 64*'S' |

## 6.8 Non-CAM Tags

### 6.8.1 *Inflight*

This is incremented once for every tile seen by the early depth cache probe logic, thereby reserving the cache line and preventing its accidental deallocation before use. The inflight count for a cache line is decremented when a tile has completed depth and stencil testing. The inflight count is an unsigned 6 bits per cache line.

### 6.8.2 *Descriptor*

Each cache line has a format assigned. This is used when the line is flushed to determine how to unpack, reformat and swizzle the data into the frame buffer.

For depth cache lines storing depth data the descriptor describes the frame buffer's current format. When flushing depth information this provides the information that is needed to generate a new Zmask, be 'smart' about how much to write back and what depth format.

| **Descriptor Bits[1:0]** | **Type** | **Description** |
|---|---|---|
| 0 | DEPTH_LO *or* DEPTH_HI | 16 bit Depth |
| 1 | DEPTH_LO *or* DEPTH_HI | 24 bit Depth (URF or FP) |
| 2 | DEPTH_LO *or* DEPTH_HI | 96 bit Depth (Zplane) |
| 3 | DEPTH_LO *or* DEPTH_HI | Reserved |

| **Descriptor Bits[9:2]** | **Type** | **Description** |
|---|---|---|
| - | DEPTH_LO | Tile cache index for Zmask and Smask |

The stencil is uncompressed/compressed in FB is used to optimize the flushing of stencil data. If the stencil has changed format then all of the tile's stencil data must be written. If the stencil has not changed its compression state then only the dirty stencils need be written.

| **Descriptor Bit** | **Type** | **Description** |
|---|---|---|
| 0 | STENCIL_PMASK | Stencil is uncompressed in FB. |
| 1 | STENCIL_PMASK | Compressed Pmask, unusable until uncompressed.<br>*BOZO how is the compression ration remembered?* |

# 7. Depth&Stencil Surface

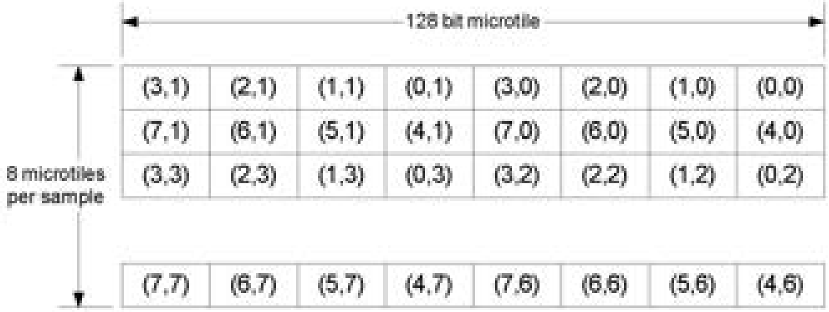
## 7.1 ZMASK Background (0)

## 7.2 ZMASK ZplaneN

# 8. ZMASK Separate (6)

## 8.1 16 Bit Depth

This format is indentical to ZMASK Expanded (7) when the depth value is 16 bits. No stencil information exists in a depth surface with 16 bit depth values. This is the most trivial surface to load and flush, the data is aligned and a full 256 bit memory operation is a single depth cache in size. This is the smallest depth surface with just 256 bytes per sample.

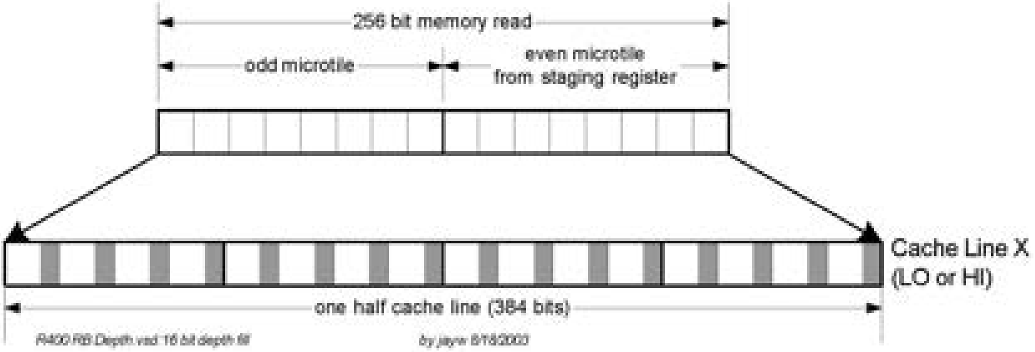


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| (3,1) | (2,1) | (1,1) | (0,1) | (3,0) | (2,0) | (1,0) | (0,0) |
| (7,1) | (6,1) | (5,1) | (4,1) | (7,0) | (6,0) | (5,0) | (4,0) |
| (3,3) | (2,3) | (1,3) | (0,3) | (3,2) | (2,2) | (1,2) | (0,2) |
| (7,7) | (6,7) | (5,7) | (4,7) | (7,6) | (6,6) | (5,6) | (4,6) |

R400 RB Depth.vsd ZMASK Separate 6 by jayw 8/18/2003

### 8.1.1 *Data Cache Fills*

A single 256 bit read request generates a single aligned data cache write (384 bits). The first microtile is retained pending the arrival of the second microtile of the read. The 16 bit depth values are padded to 24 bits and written. The half cache line is marked valid.



R400 RB Depth.vsd 16 bit depth fill by jayw 8/18/2003

Writes are just as obviously simple. There are no alignment concerns in either direction, the data in the frame buffer is always aligned in this format.

### 8.1.2 *24 Bit Depth & Stencil*

# 9. ZMASK Expanded (7)

## 9.1 16 Bit Depth

This format is indentical to ZMASK Separate (6) when the depth value is 16 bits. No stencil information exists in a depth surface with 16 bit depth values.

## 9.2 Packed 24 Bit Depth with Stencil

This format has low performance and is only intended as a software or texture unit readable and writable surface. It is unfortunately the first depth surface to get working. The depth surface is stored into three depth cache lines per sample; one for the stencil data and two for the depth values.

A whole tile is read whenever any depth or stencil information of the tile is needed. This is not optimim particularily for the multisample cases, but this restriction is expediant.

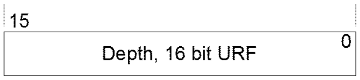
The stencil write assembles both read microtiles into a single 8 pixel stencil write (one cache word or 64 bits plus the 32 bits of Pmask data is zeroed to aid debug). The depth write assembles both read microtiles into an aligned single 8 pixel depth write (192 bits = two cache words).

Writing the tile from the depth cache will generate just 8*S 256 bit memory writes. Assembly of the write data with the stencil data is performed outside the cache with the aid of an L0 stencil cache line.

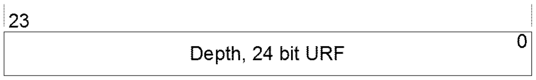
# R400 RB Depth Drawings

*R400 RB Depth.vsd:TITLE*

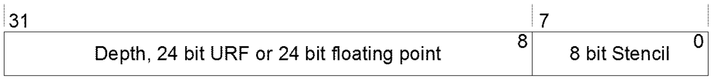
*by jayw 9/8/2003*



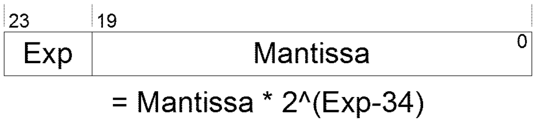
*R400 RB Depth.vsd:Depth 16 URF*
*by jayw 9/8/2003*



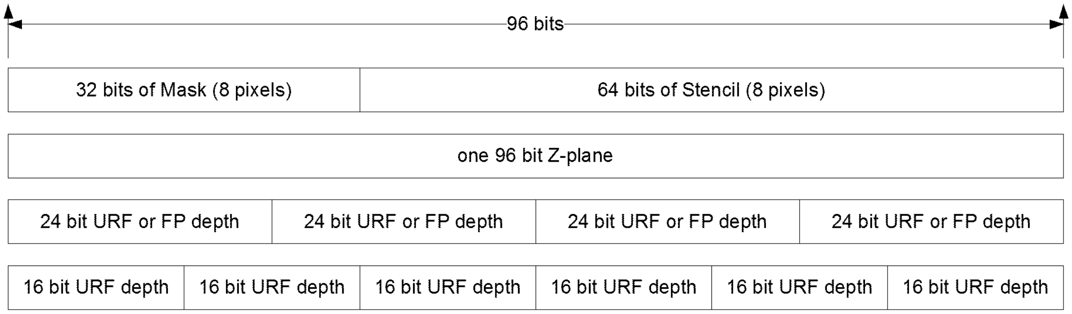
*R400 RB Depth.vsd:Depth, 24 bit URF*
*by jayw 9/8/2003*



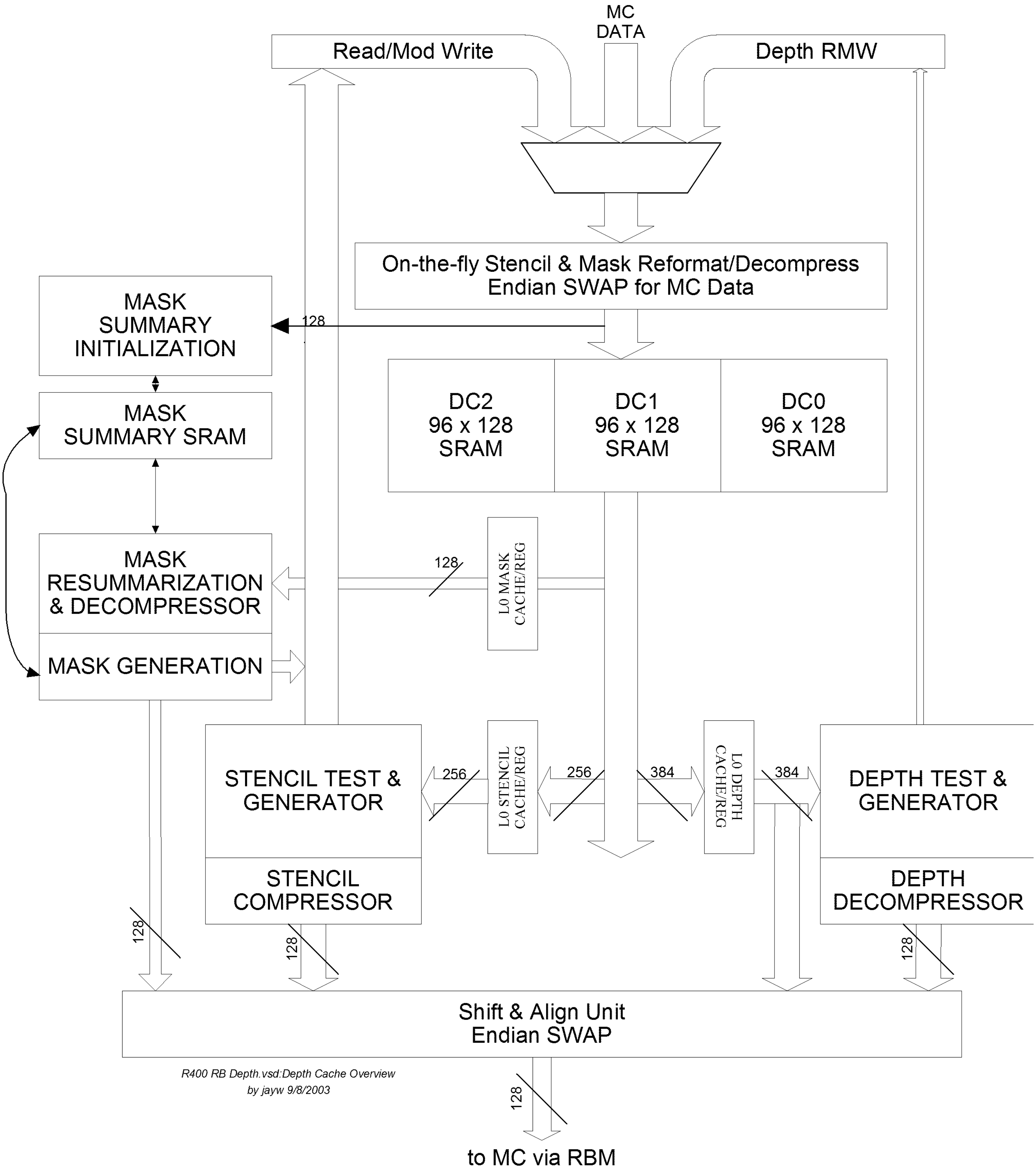
*R400 RB Depth.vsd:Depth, 32 bit with stencil*
*by jayw 9/8/2003*



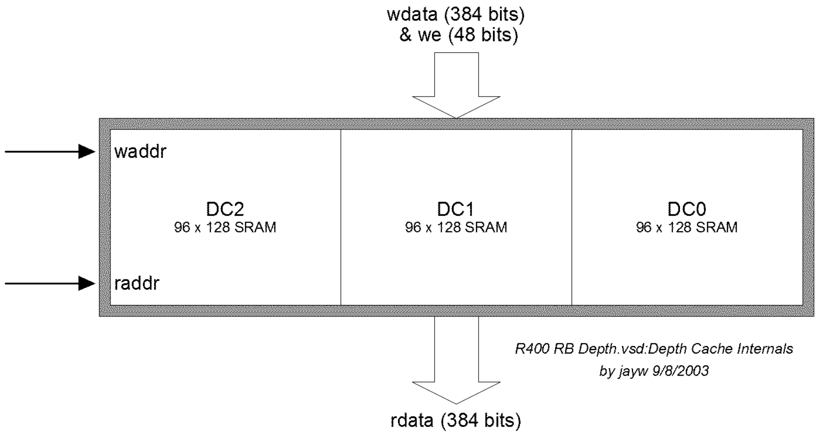
*R400 RB Depth.vsd:Depth, 24 bit floating point*
*by jayw 9/8/2003*



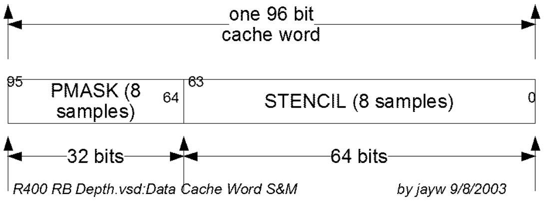
*R400 RB Depth.vsd:Cache Word* *by jayw 9/8/2003*

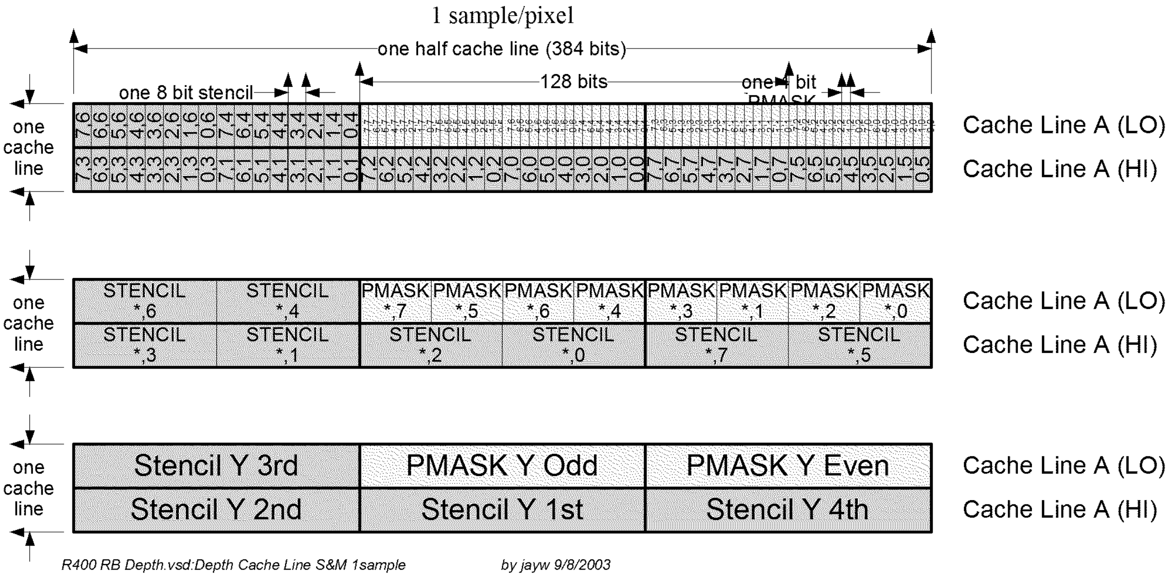


*R400 RB Depth.vsd:Depth Cache Overview*
*by jayw 9/8/2003*

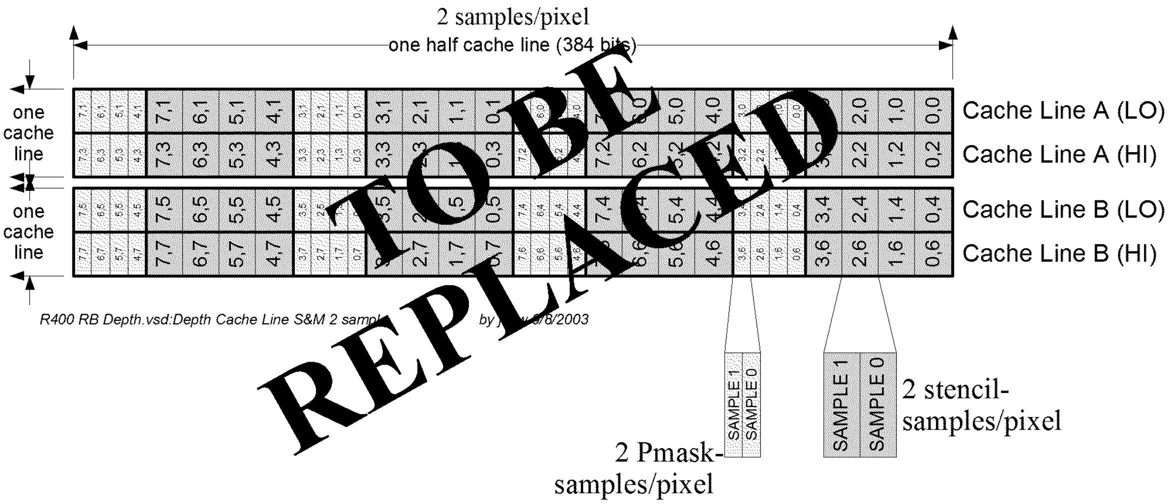


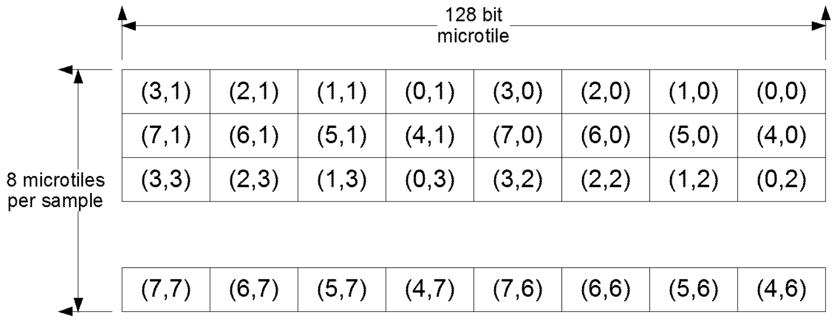
*R400 RB Depth.vsd:Depth Cache Internals*
*by jayw 9/8/2003*


one 96 bit cache word
95
PMASK (8 samples)
64
63
STENCIL (8 samples)
0
32 bits
64 bits
R400 RB Depth.vsd:Data Cache Word S&M
by jayw 9/8/2003



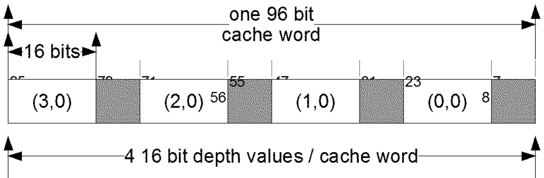
*R400 RB Depth.vsd:Depth Cache Line S&M 1sample* *by jayw 9/8/2003*


2 samples/pixel
one half cache line (384 bits)
one cache line
one cache line
Cache Line A (LO)
Cache Line A (HI)
Cache Line B (LO)
Cache Line B (HI)
R400 RB Depth.vsd:Depth Cache Line S&M 2 sampl
by j
8/8/2003
SAMPLE 1
SAMPLE 0
2 Pmask-samples/pixel
SAMPLE 1
SAMPLE 0
2 stencil-samples/pixel

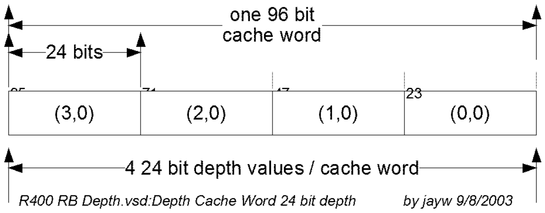


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| (3,1) | (2,1) | (1,1) | (0,1) | (3,0) | (2,0) | (1,0) | (0,0) |
| (7,1) | (6,1) | (5,1) | (4,1) | (7,0) | (6,0) | (5,0) | (4,0) |
| (3,3) | (2,3) | (1,3) | (0,3) | (3,2) | (2,2) | (1,2) | (0,2) |
| (7,7) | (6,7) | (5,7) | (4,7) | (7,6) | (6,6) | (5,6) | (4,6) |

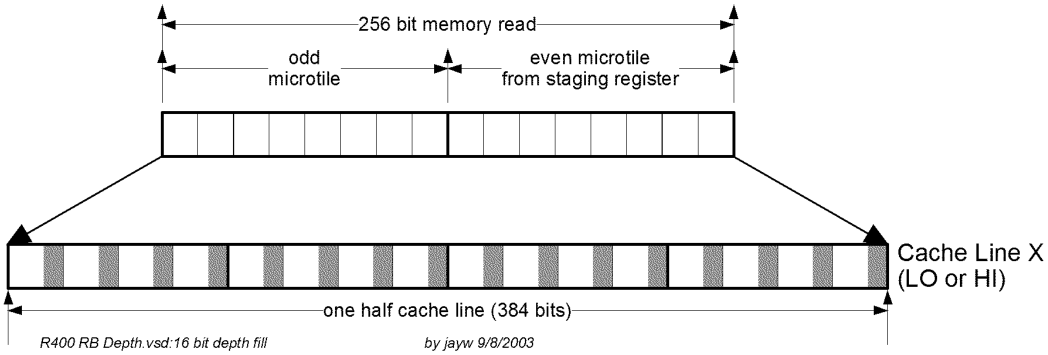
*R400 RB Depth.vsd:ZMASK Separate 6* *by jayw 9/8/2003*

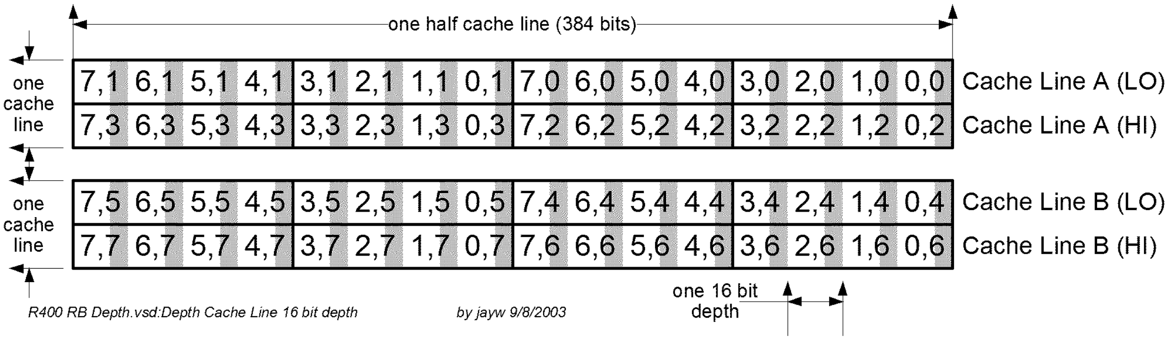


*R400 RB Depth.vsd:Depth Cache Word 16 bit depth* *by jayw 9/8/2003*



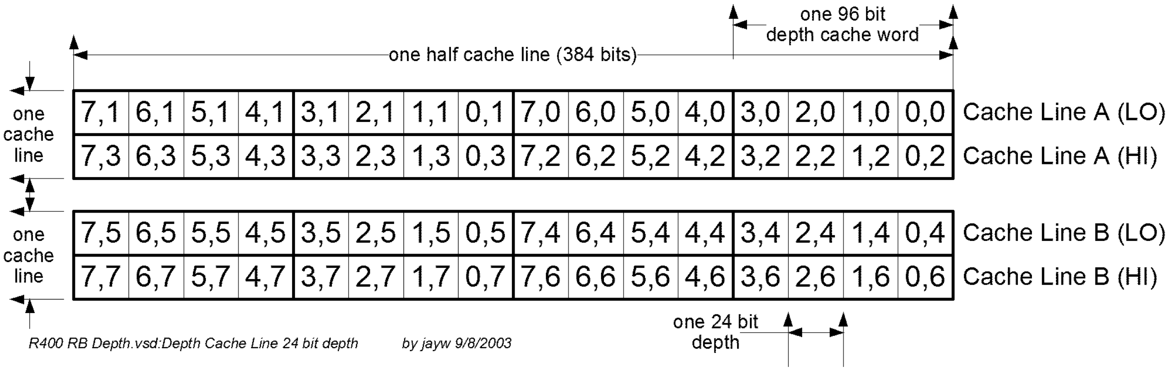
*R400 RB Depth.vsd:Depth Cache Word 24 bit depth* *by jayw 9/8/2003*


256 bit memory read
odd microtile
even microtile from staging register
Cache Line X (LO or HI)
one half cache line (384 bits)
R400 RB Depth.vsd:16 bit depth fill
by jayw 9/8/2003



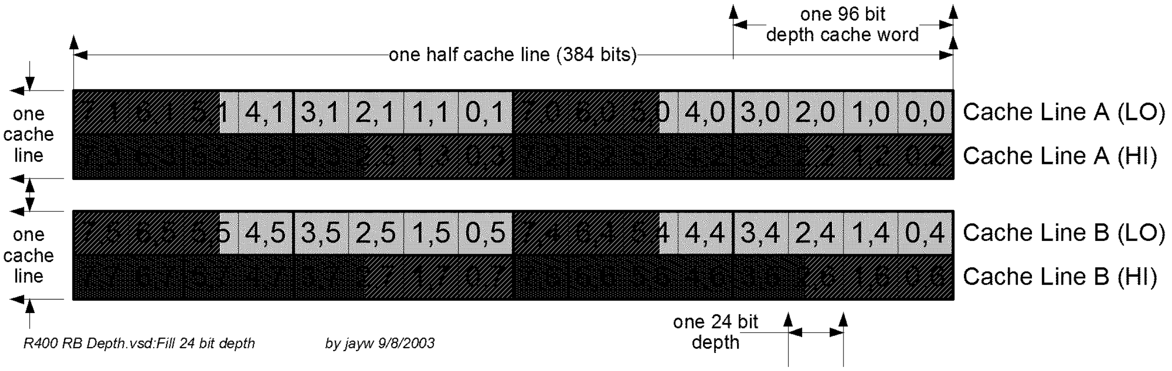
| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7,1 | 6,1 | 5,1 | 4,1 | 3,1 | 2,1 | 1,1 | 0,1 | 7,0 | 6,0 | 5,0 | 4,0 | 3,0 | 2,0 | 1,0 | 0,0 | Cache Line A (LO) |
| 7,3 | 6,3 | 5,3 | 4,3 | 3,3 | 2,3 | 1,3 | 0,3 | 7,2 | 6,2 | 5,2 | 4,2 | 3,2 | 2,2 | 1,2 | 0,2 | Cache Line A (HI) |
| 7,5 | 6,5 | 5,5 | 4,5 | 3,5 | 2,5 | 1,5 | 0,5 | 7,4 | 6,4 | 5,4 | 4,4 | 3,4 | 2,4 | 1,4 | 0,4 | Cache Line B (LO) |
| 7,7 | 6,7 | 5,7 | 4,7 | 3,7 | 2,7 | 1,7 | 0,7 | 7,6 | 6,6 | 5,6 | 4,6 | 3,6 | 2,6 | 1,6 | 0,6 | Cache Line B (HI) |

*R400 RB Depth.vsd:Depth Cache Line 16 bit depth* *by jayw 9/8/2003*

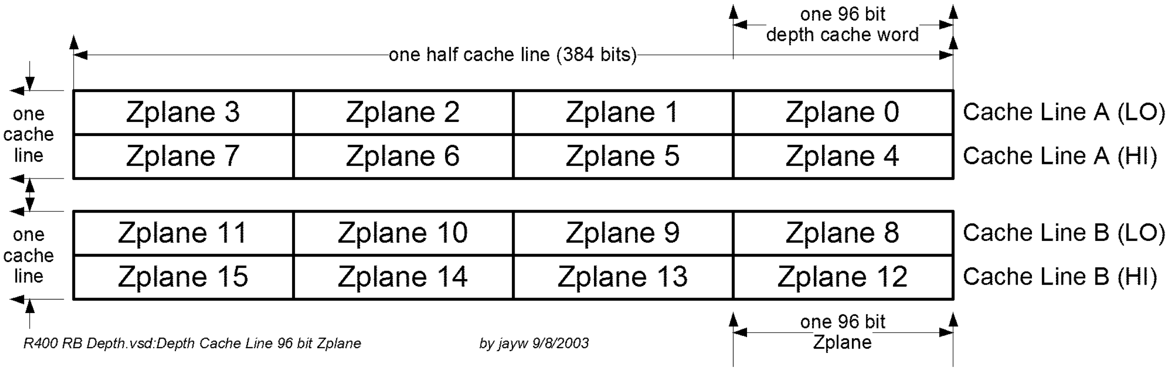


| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7,1 | 6,1 | 5,1 | 4,1 | 3,1 | 2,1 | 1,1 | 0,1 | 7,0 | 6,0 | 5,0 | 4,0 | 3,0 | 2,0 | 1,0 | 0,0 | Cache Line A (LO) |
| 7,3 | 6,3 | 5,3 | 4,3 | 3,3 | 2,3 | 1,3 | 0,3 | 7,2 | 6,2 | 5,2 | 4,2 | 3,2 | 2,2 | 1,2 | 0,2 | Cache Line A (HI) |
| 7,5 | 6,5 | 5,5 | 4,5 | 3,5 | 2,5 | 1,5 | 0,5 | 7,4 | 6,4 | 5,4 | 4,4 | 3,4 | 2,4 | 1,4 | 0,4 | Cache Line B (LO) |
| 7,7 | 6,7 | 5,7 | 4,7 | 3,7 | 2,7 | 1,7 | 0,7 | 7,6 | 6,6 | 5,6 | 4,6 | 3,6 | 2,6 | 1,6 | 0,6 | Cache Line B (HI) |

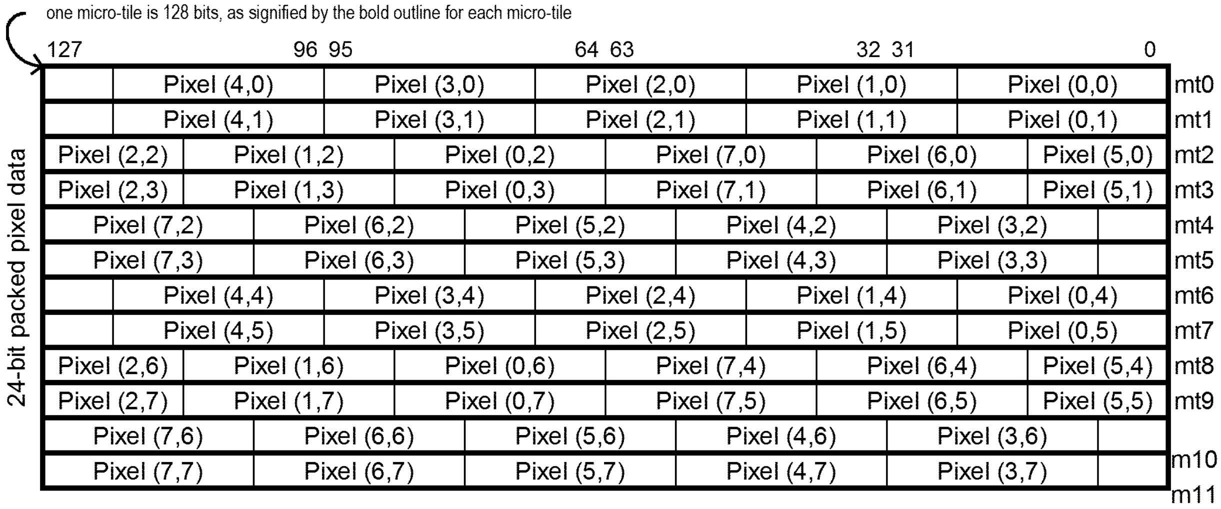
*R400 RB Depth.vsd:Depth Cache Line 24 bit depth* *by jayw 9/8/2003*



*R400 RB Depth.vsd:Fill 24 bit depth* *by jayw 9/8/2003*



*R400 RB Depth.vsd:Depth Cache Line 96 bit Zplane* *by jayw 9/8/2003*



one micro-tile is 128 bits, as signified by the bold outline for each micro-tile

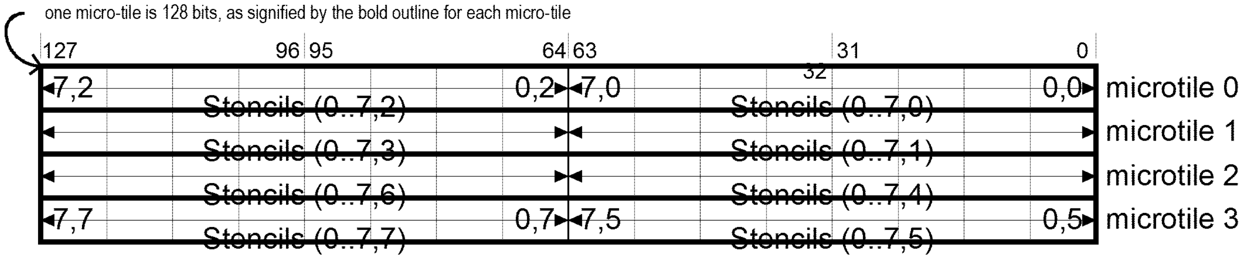
| | | | | | | |
|---|---|---|---|---|---|---|
| | Pixel (4,0) | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) | mt0 |
| | Pixel (4,1) | Pixel (3,1) | Pixel (2,1) | Pixel (1,1) | Pixel (0,1) | mt1 |
| Pixel (2,2) | Pixel (1,2) | Pixel (0,2) | Pixel (7,0) | Pixel (6,0) | Pixel (5,0) | mt2 |
| Pixel (2,3) | Pixel (1,3) | Pixel (0,3) | Pixel (7,1) | Pixel (6,1) | Pixel (5,1) | mt3 |
| Pixel (7,2) | Pixel (6,2) | Pixel (5,2) | Pixel (4,2) | Pixel (3,2) | | mt4 |
| Pixel (7,3) | Pixel (6,3) | Pixel (5,3) | Pixel (4,3) | Pixel (3,3) | | mt5 |
| | Pixel (4,4) | Pixel (3,4) | Pixel (2,4) | Pixel (1,4) | Pixel (0,4) | mt6 |
| | Pixel (4,5) | Pixel (3,5) | Pixel (2,5) | Pixel (1,5) | Pixel (0,5) | mt7 |
| Pixel (2,6) | Pixel (1,6) | Pixel (0,6) | Pixel (7,4) | Pixel (6,4) | Pixel (5,4) | mt8 |
| Pixel (2,7) | Pixel (1,7) | Pixel (0,7) | Pixel (7,5) | Pixel (6,5) | Pixel (5,5) | mt9 |
| Pixel (7,6) | Pixel (6,6) | Pixel (5,6) | Pixel (4,6) | Pixel (3,6) | | m10 |
| Pixel (7,7) | Pixel (6,7) | Pixel (5,7) | Pixel (4,7) | Pixel (3,7) | | m11 |

*R400 RB Depth.vsd:24 bit microtile FB format* *by lseiler from R400_Memory_Tiles.vsd::FB2dAltMicro*

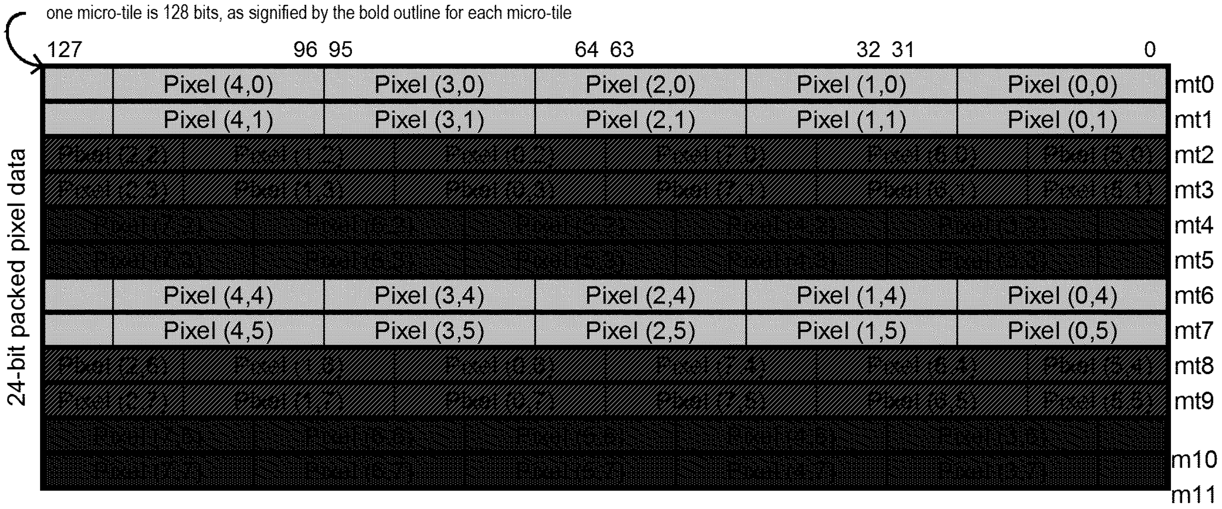
one micro-tile is 128 bits, as signified by the bold outline for each micro-tile

| 127 96 | 95 64 | 63 32 | 31 0 | |
|---|---|---|---|---|
| 7,3 Stencils (0..7,3) | Stencils (0..7,1) | Stencils (0..7,2) | Stencils (0..7,0) 0,0 | microtile 0 |
| 7,7 Stencils (0..7,7) | Stencils (0..7,5) | Stencils (0..7,6) | Stencils (0..7,4) 0,4 | microtile 1 |

*R400 RB Depth.vsd:4 bit microtile FB format* *by jayw 9/8/2003*

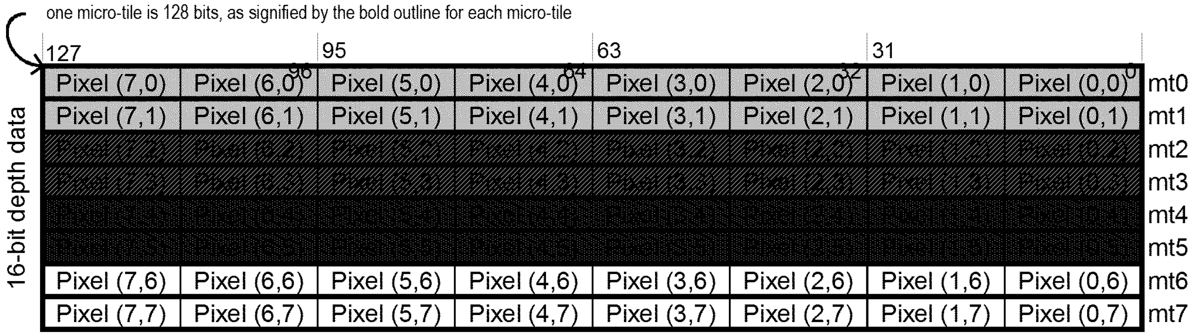

one micro-tile is 128 bits, as signified by the bold outline for each micro-tile

*R400 RB Depth.vsd:8 bit stencil FB format* *by jayw 9/8/2003*



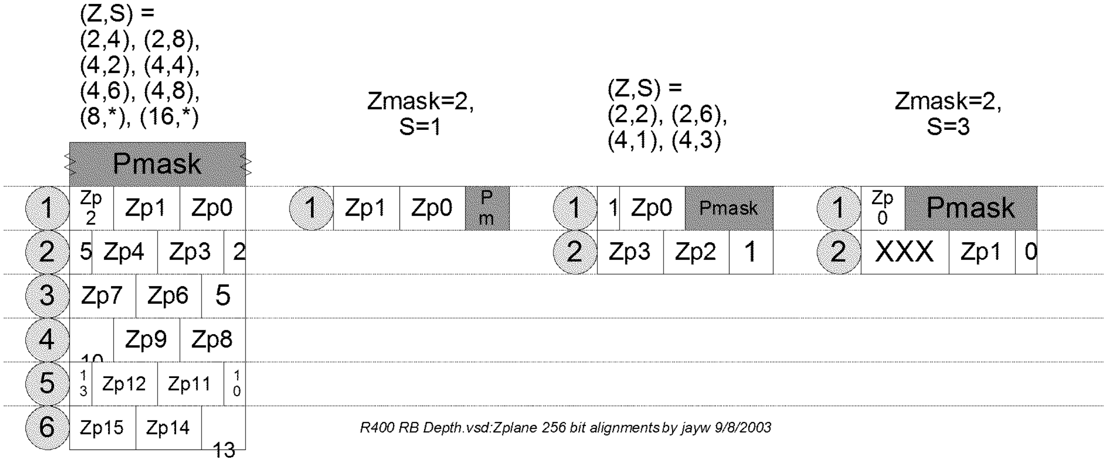
*R400 RB Depth.vsd:24 bit microtile FB format colo* *by lseiler from R400_Memory_Tiles.vsd::FB2dAltMicro*

Total size is 128 Bytes per sample. This is the smallest depth surface.

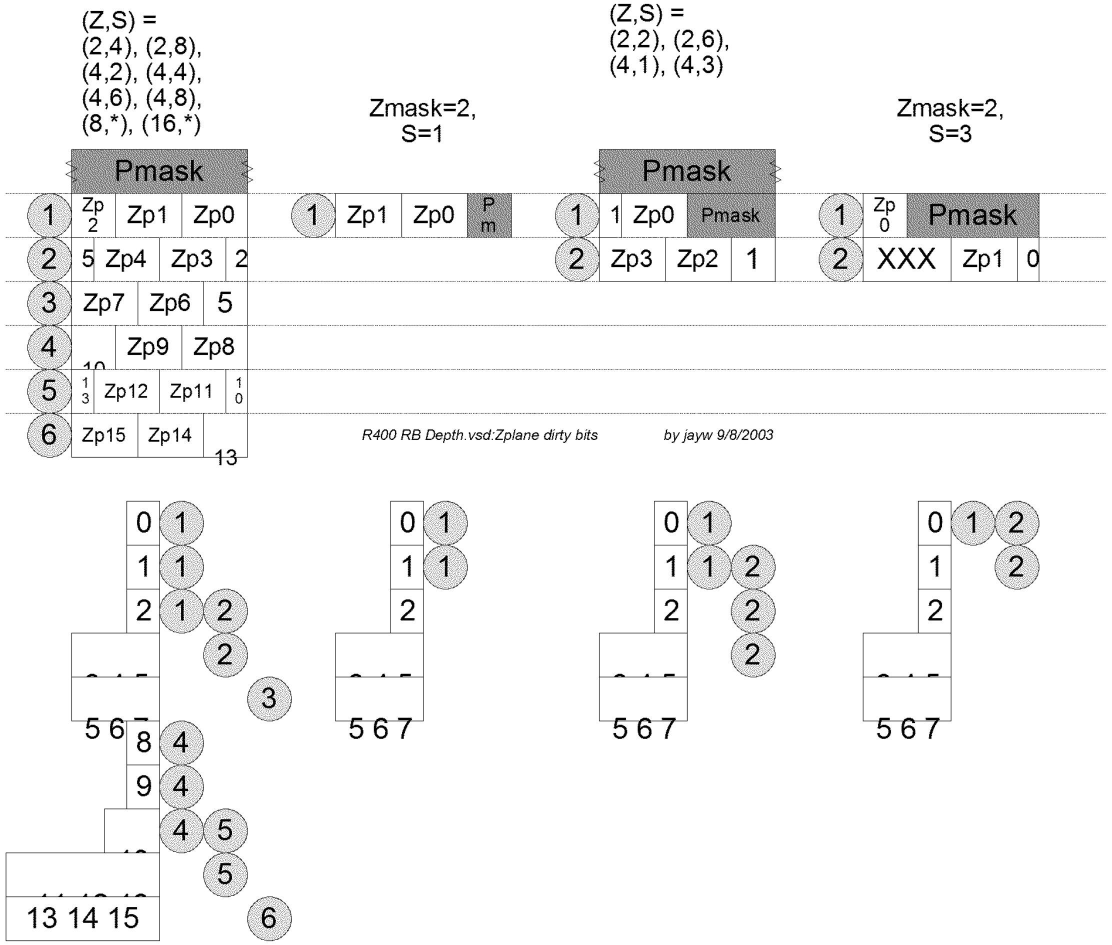


| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Pixel (7,0) | Pixel (6,0) | Pixel (5,0) | Pixel (4,0) | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) | mt0 |
| Pixel (7,1) | Pixel (6,1) | Pixel (5,1) | Pixel (4,1) | Pixel (3,1) | Pixel (2,1) | Pixel (1,1) | Pixel (0,1) | mt1 |
| Pixel (7,2) | Pixel (6,2) | Pixel (5,2) | Pixel (4,2) | Pixel (3,2) | Pixel (2,2) | Pixel (1,2) | Pixel (0,2) | mt2 |
| Pixel (7,3) | Pixel (6,3) | Pixel (5,3) | Pixel (4,3) | Pixel (3,3) | Pixel (2,3) | Pixel (1,3) | Pixel (0,3) | mt3 |
| Pixel (7,4) | Pixel (6,4) | Pixel (5,4) | Pixel (4,4) | Pixel (3,4) | Pixel (2,4) | Pixel (1,4) | Pixel (0,4) | mt4 |
| Pixel (7,5) | Pixel (6,5) | Pixel (5,5) | Pixel (4,5) | Pixel (3,5) | Pixel (2,5) | Pixel (1,5) | Pixel (0,5) | mt5 |
| Pixel (7,6) | Pixel (6,6) | Pixel (5,6) | Pixel (4,6) | Pixel (3,6) | Pixel (2,6) | Pixel (1,6) | Pixel (0,6) | mt6 |
| Pixel (7,7) | Pixel (6,7) | Pixel (5,7) | Pixel (4,7) | Pixel (3,7) | Pixel (2,7) | Pixel (1,7) | Pixel (0,7) | mt7 |

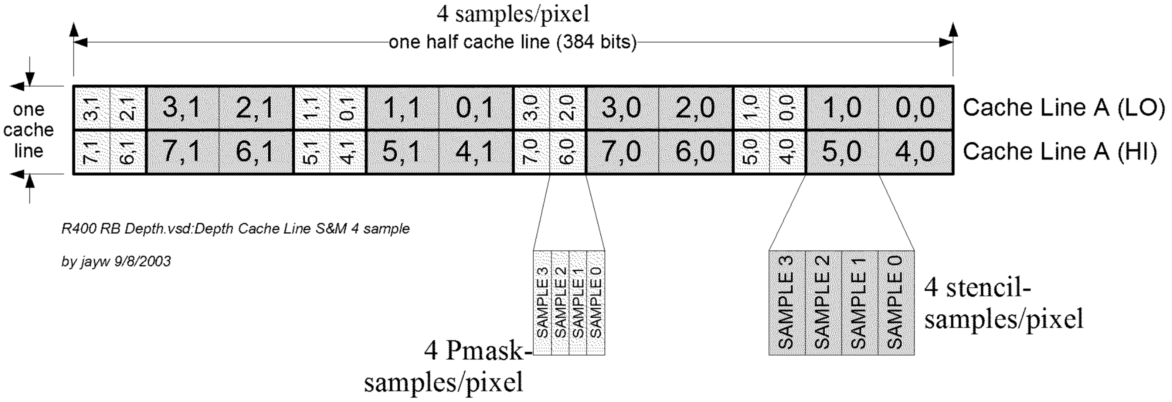
*R400 RB Depth.vsd:16 bit FB format* *by jayw 9/8/2003*



*R400 RB Depth.vsd:Zplane 256 bit alignments by jayw 9/8/2003*

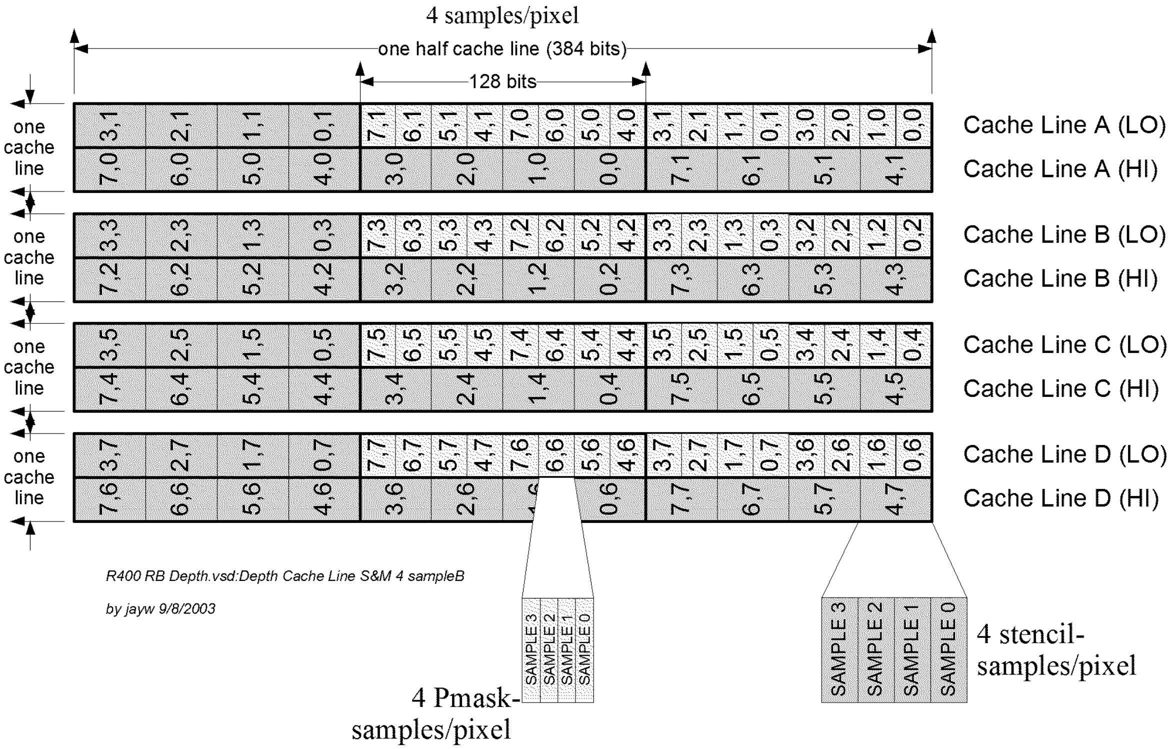


R400 RB Depth.vsd:Zplane dirty bits by jayw 9/8/2003



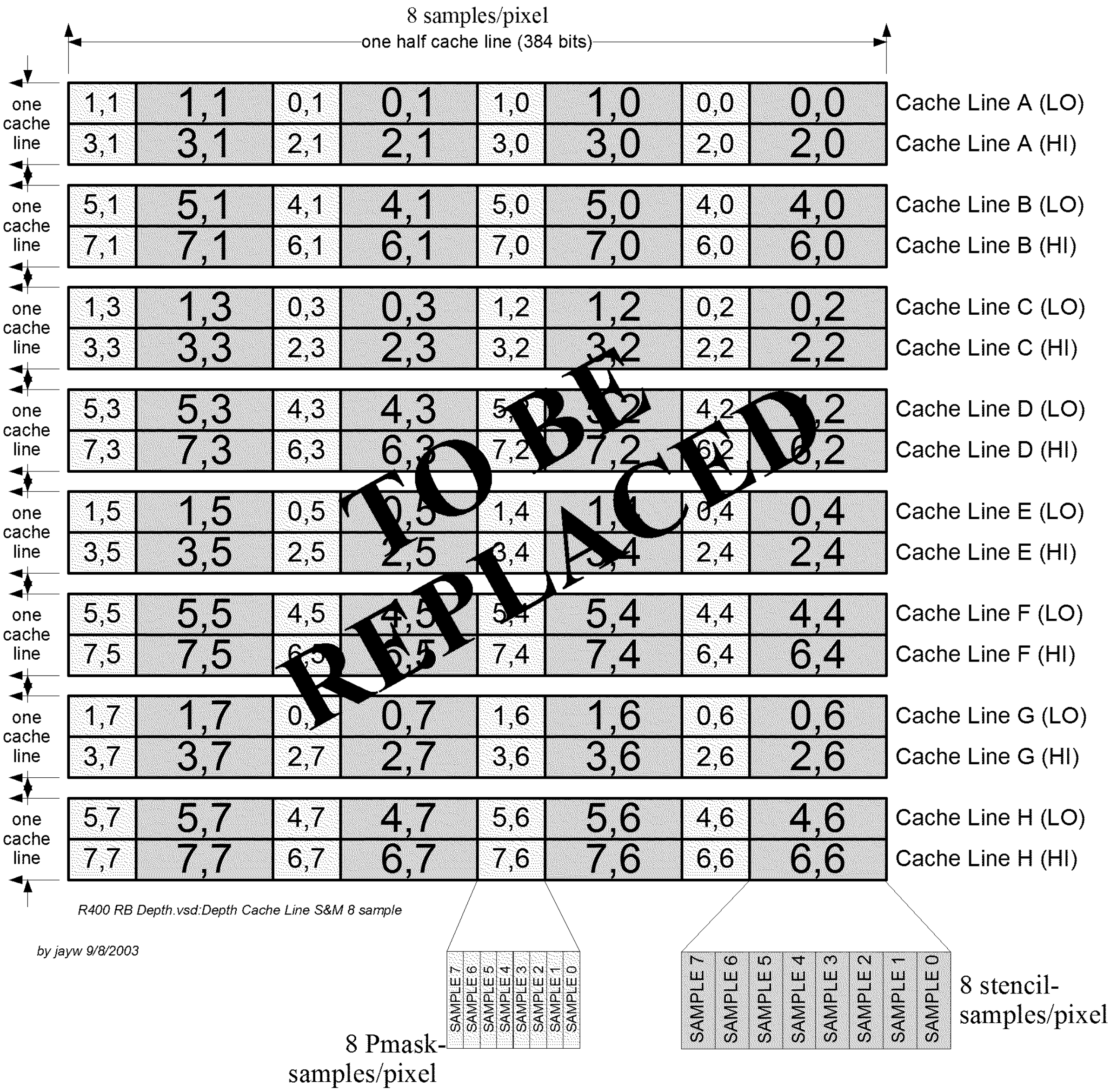
*R400 RB Depth.vsd:Depth Cache Line S&M 4 sample*

*by jayw 9/8/2003*



*R400 RB Depth.vsd:Depth Cache Line S&M 4 sampleB*

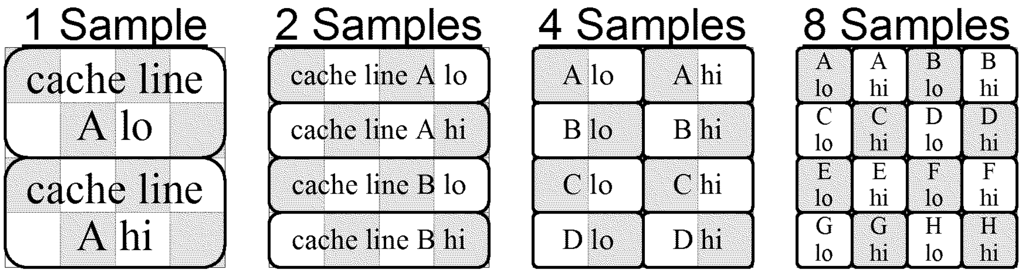
*by jayw 9/8/2003*



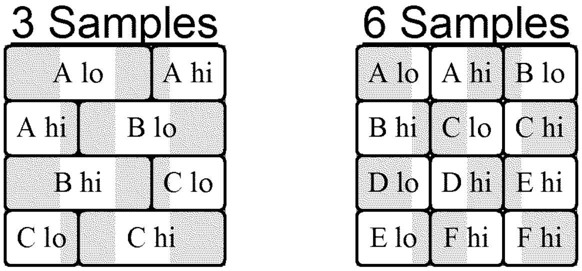
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1,1 | 1,1 | 0,1 | 0,1 | 1,0 | 1,0 | 0,0 | 0,0 | Cache Line A (LO) |
| 3,1 | 3,1 | 2,1 | 2,1 | 3,0 | 3,0 | 2,0 | 2,0 | Cache Line A (HI) |
| 5,1 | 5,1 | 4,1 | 4,1 | 5,0 | 5,0 | 4,0 | 4,0 | Cache Line B (LO) |
| 7,1 | 7,1 | 6,1 | 6,1 | 7,0 | 7,0 | 6,0 | 6,0 | Cache Line B (HI) |
| 1,3 | 1,3 | 0,3 | 0,3 | 1,2 | 1,2 | 0,2 | 0,2 | Cache Line C (LO) |
| 3,3 | 3,3 | 2,3 | 2,3 | 3,2 | 3,2 | 2,2 | 2,2 | Cache Line C (HI) |
| 5,3 | 5,3 | 4,3 | 4,3 | 5,2 | 5,2 | 4,2 | 4,2 | Cache Line D (LO) |
| 7,3 | 7,3 | 6,3 | 6,3 | 7,2 | 7,2 | 6,2 | 6,2 | Cache Line D (HI) |
| 1,5 | 1,5 | 0,5 | 0,5 | 1,4 | 1,4 | 0,4 | 0,4 | Cache Line E (LO) |
| 3,5 | 3,5 | 2,5 | 2,5 | 3,4 | 3,4 | 2,4 | 2,4 | Cache Line E (HI) |
| 5,5 | 5,5 | 4,5 | 4,5 | 5,4 | 5,4 | 4,4 | 4,4 | Cache Line F (LO) |
| 7,5 | 7,5 | 6,5 | 6,5 | 7,4 | 7,4 | 6,4 | 6,4 | Cache Line F (HI) |
| 1,7 | 1,7 | 0,7 | 0,7 | 1,6 | 1,6 | 0,6 | 0,6 | Cache Line G (LO) |
| 3,7 | 3,7 | 2,7 | 2,7 | 3,6 | 3,6 | 2,6 | 2,6 | Cache Line G (HI) |
| 5,7 | 5,7 | 4,7 | 4,7 | 5,6 | 5,6 | 4,6 | 4,6 | Cache Line H (LO) |
| 7,7 | 7,7 | 6,7 | 6,7 | 7,6 | 7,6 | 6,6 | 6,6 | Cache Line H (HI) |

*R400 RB Depth.vsd:Depth Cache Line S&M 8 sample*

*by jayw 9/8/2003*

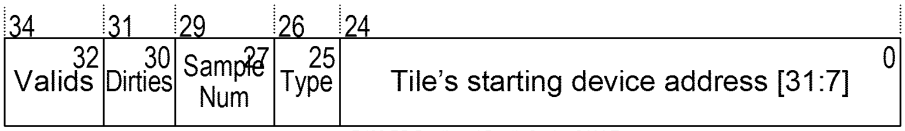


*R400 RB Depth.vsd:Depth Cache Line S&M as Tiles* *by jayw 9/8/2003*



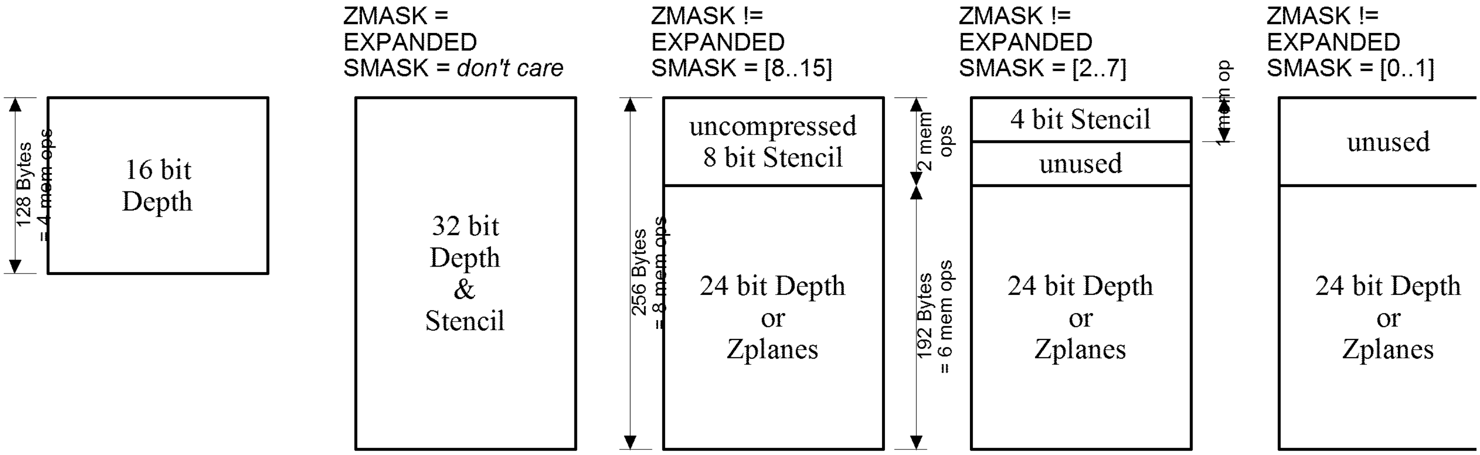
*R400 RB Depth.vsd:Depth Cache Line S&M as Tiles2*

*by jayw 9/8/2003*



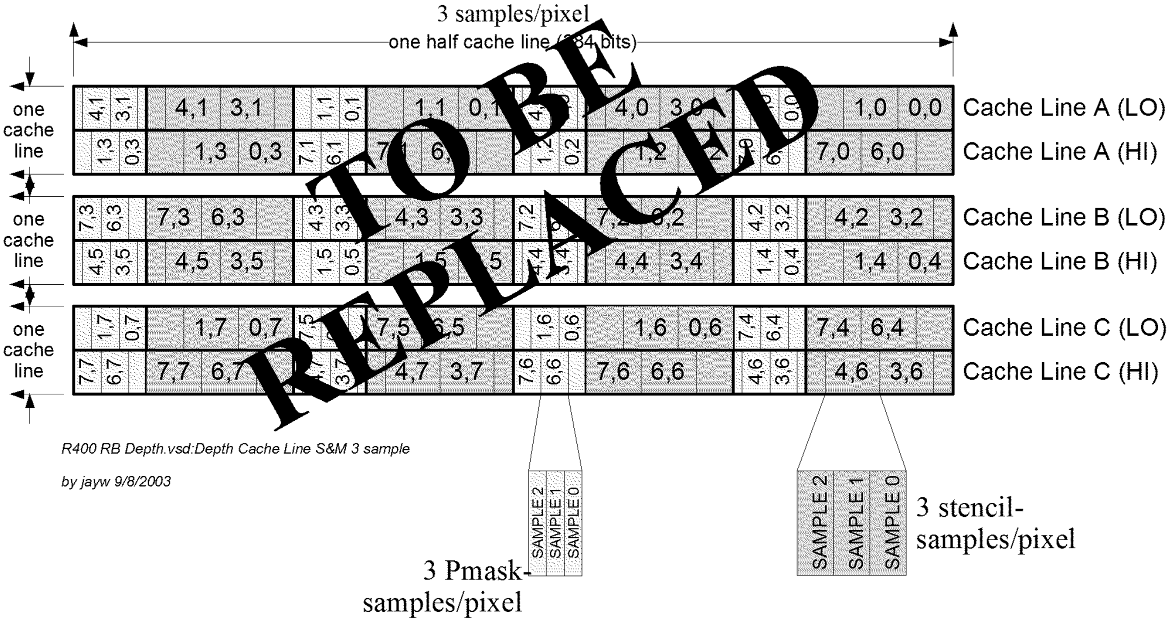
*R400 RB Depth.vsd:Depth Cache CAM Entry*

*by jayw 9/8/2003*



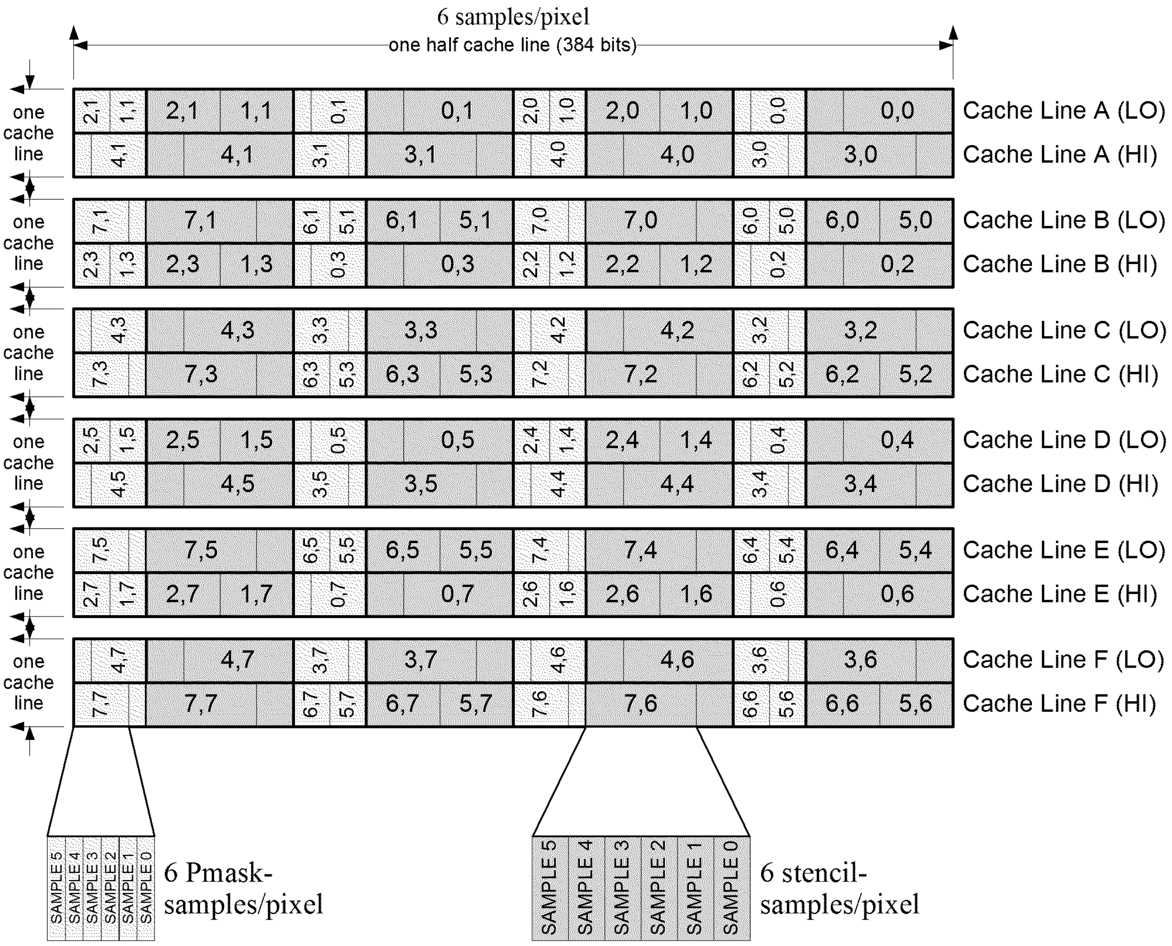
*R400 RB Depth.vsd:Tile Format Stencil Overview*

*by jayw 9/8/2003*

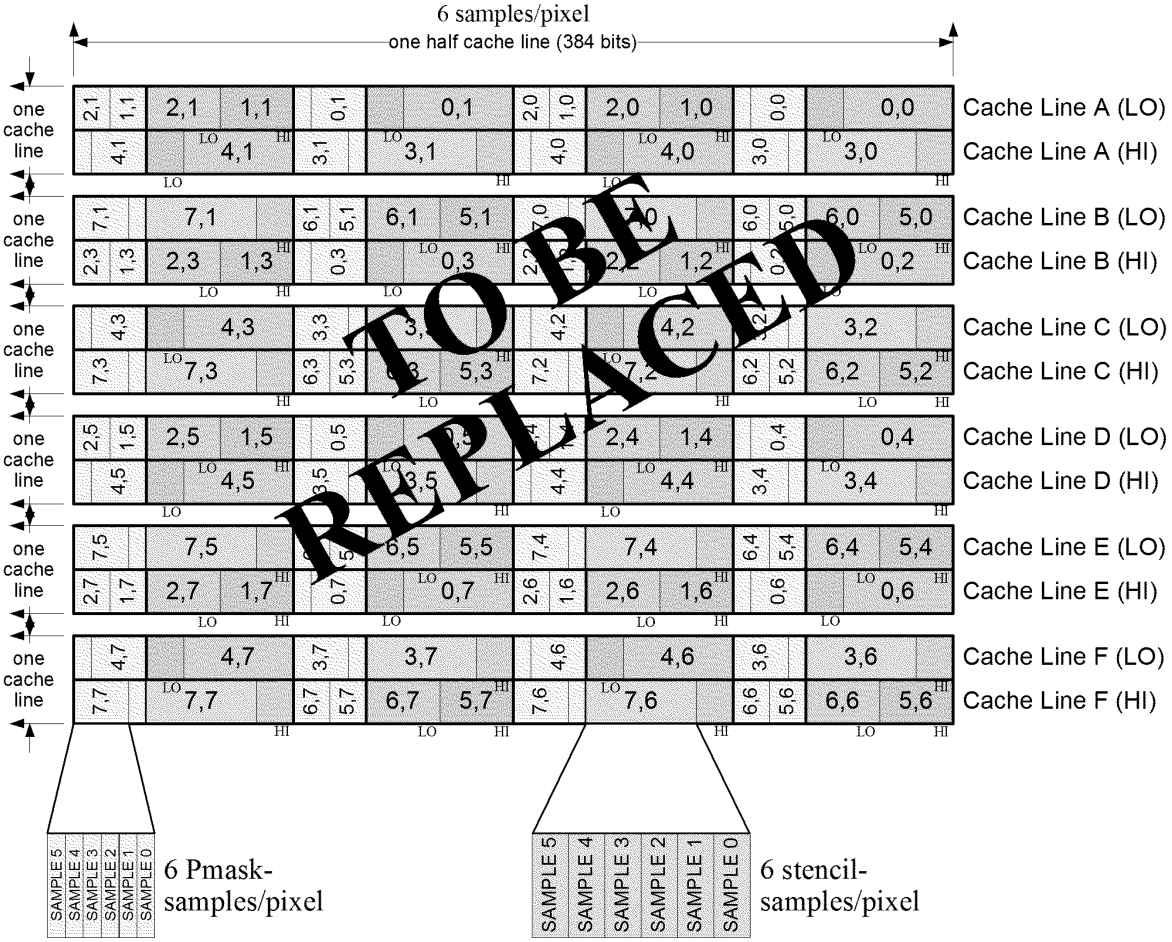


*R400 RB Depth.vsd:Depth Cache Line S&M 3 sample*

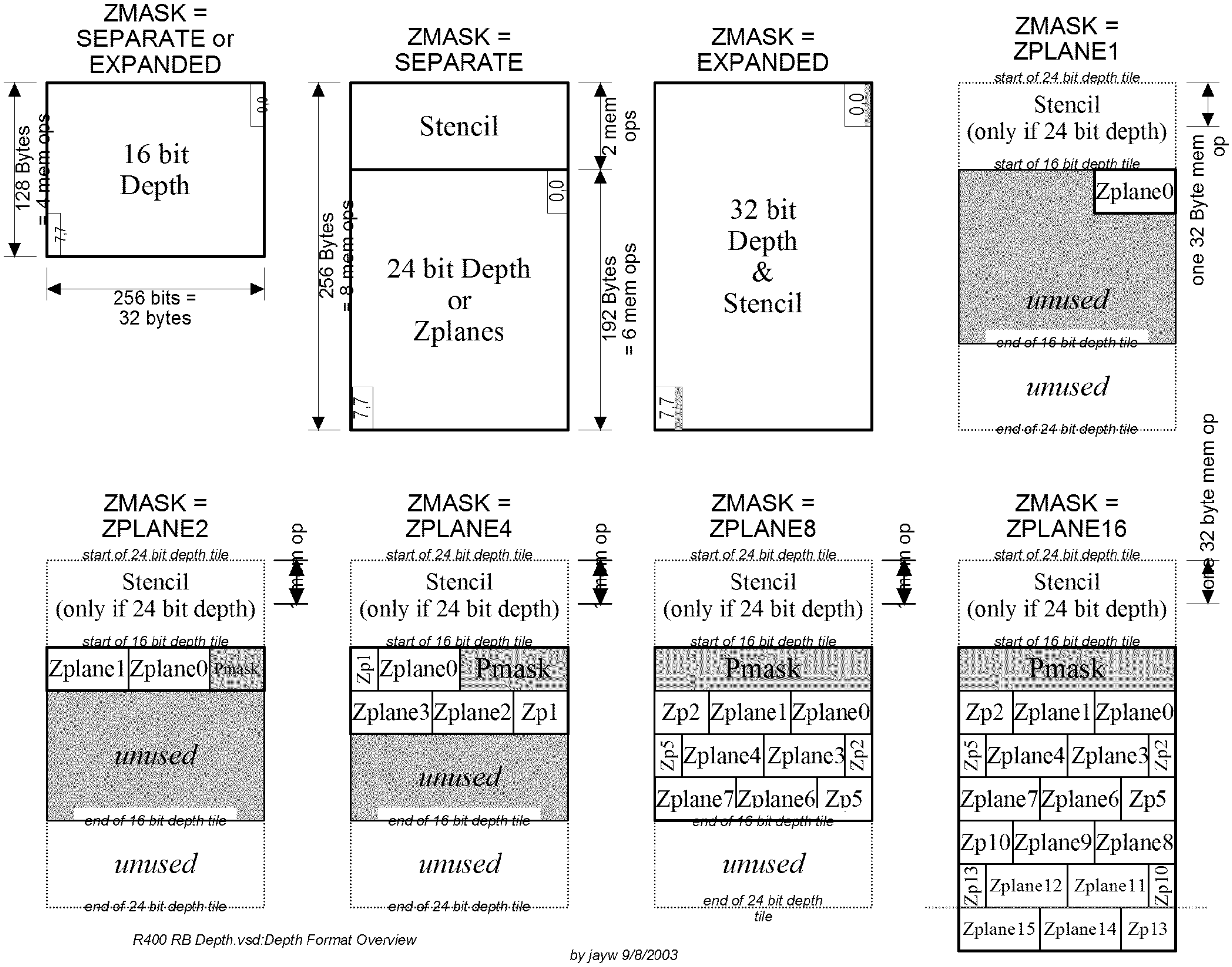
*by jayw 9/8/2003*



*R400 RB Depth.vsd:Depth Cache Line S&M 6 sample* *by jayw 9/8/2003*

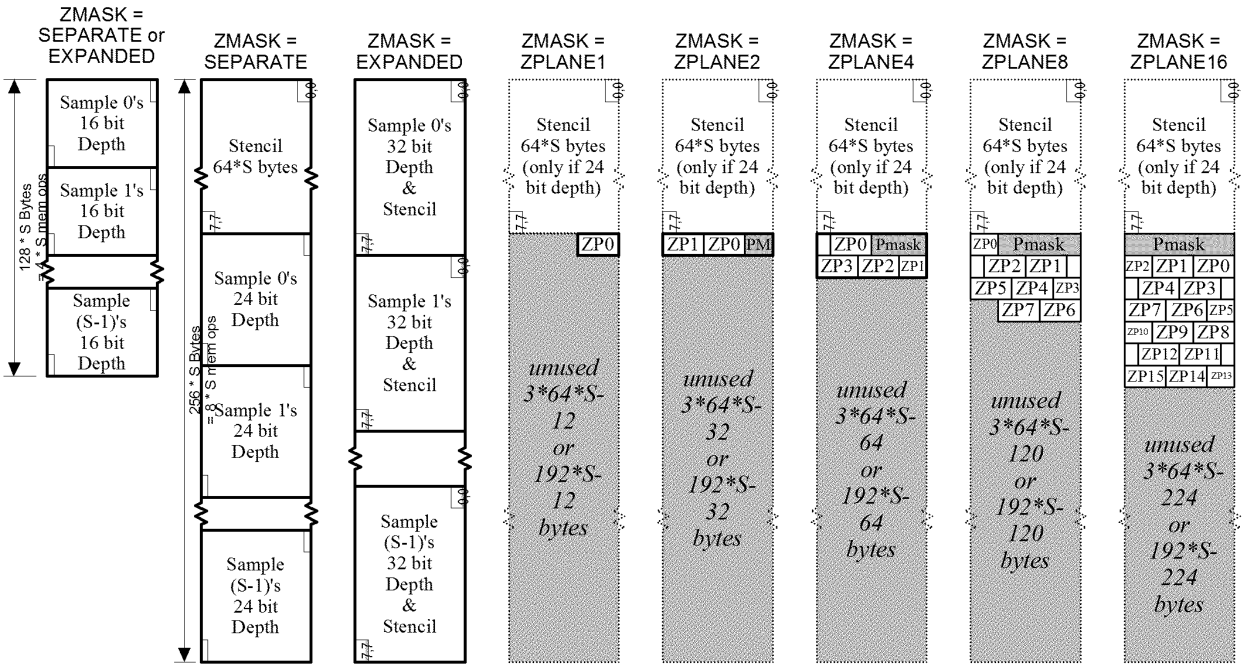


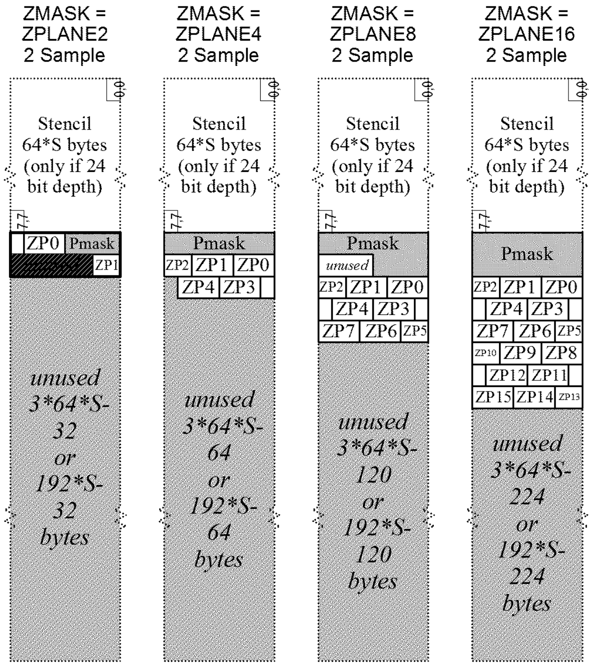
*R400 RB Depth.vsd:Depth Cache Line S&M 6 SampleB* *by jayw 9/8/2003*

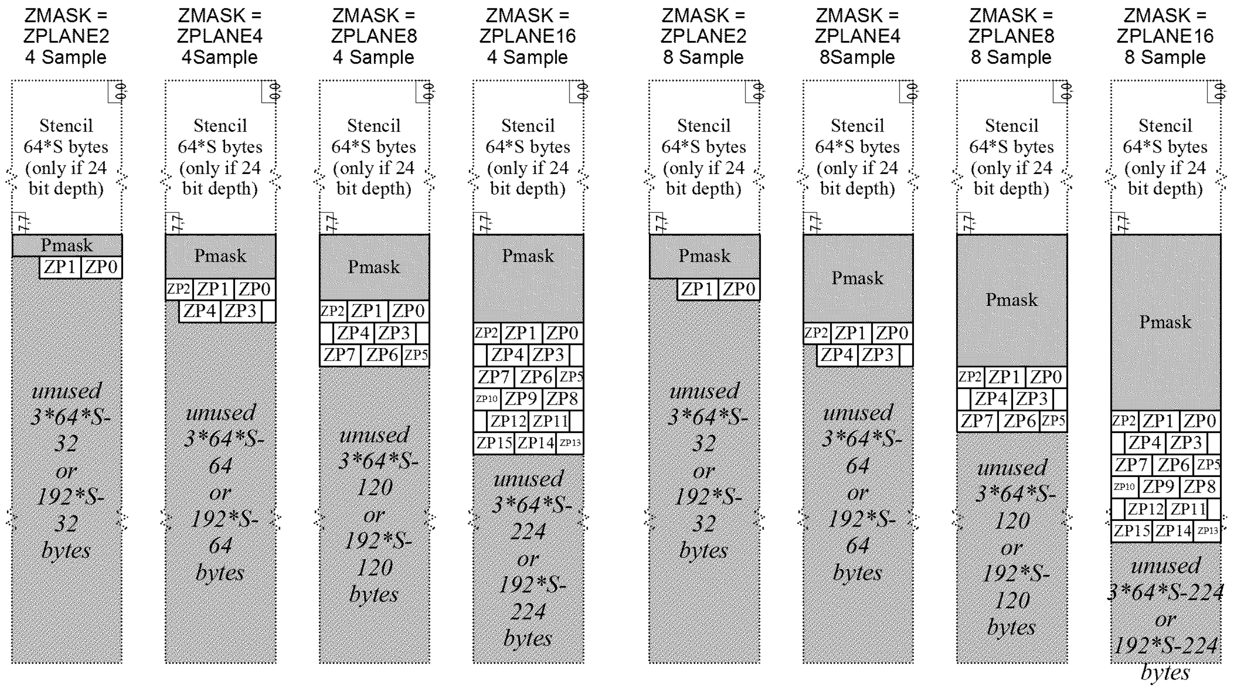


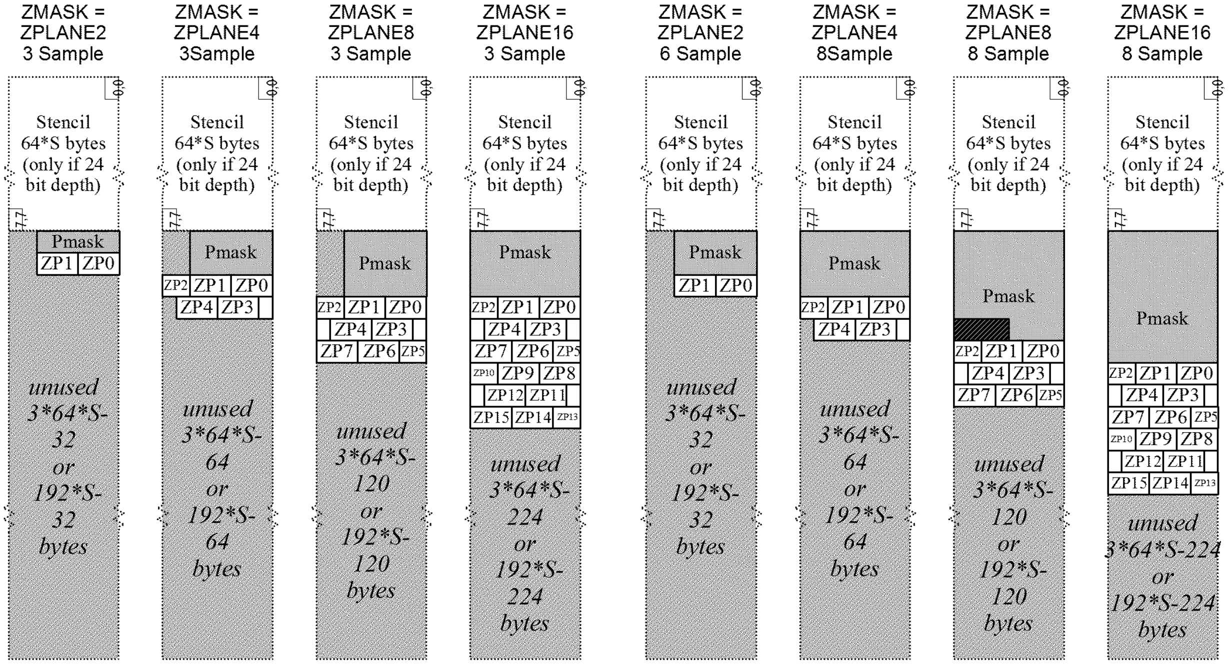
R400 RB Depth.vsd:Depth Format Overview

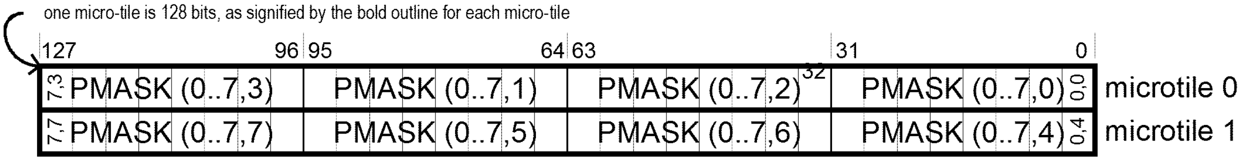
by jayw 9/8/2003


ZMASK = SEPARATE or EXPANDED
ZMASK = SEPARATE
ZMASK = EXPANDED
ZMASK = ZPLANE1
ZMASK = ZPLANE2
ZMASK = ZPLANE4
ZMASK = ZPLANE8
ZMASK = ZPLANE16
128 * S Bytes
= 4 * S mem ops
Sample 0's 16 bit Depth
Sample 1's 16 bit Depth
Sample (S-1)'s 16 bit Depth
256 * S Bytes
= 8 * S mem ops
Stencil 64*S bytes
Sample 0's 24 bit Depth
Sample 1's 24 bit Depth
Sample (S-1)'s 24 bit Depth
Sample 0's 32 bit Depth & Stencil
Sample 1's 32 bit Depth & Stencil
Sample (S-1)'s 32 bit Depth & Stencil
Stencil 64*S bytes (only if 24 bit depth)
0,0
7,7
ZP0
unused 3*64*S-12 or 192*S-12 bytes
ZP1
ZP0
PM
unused 3*64*S-32 or 192*S-32 bytes
ZP0
Pmask
ZP3
ZP2
ZP1
unused 3*64*S-64 or 192*S-64 bytes
ZP0
Pmask
ZP2
ZP1
ZP5
ZP4
ZP3
ZP7
ZP6
unused 3*64*S-120 or 192*S-120 bytes
Pmask
ZP2
ZP1
ZP0
ZP4
ZP3
ZP7
ZP6
ZP5
ZP10
ZP9
ZP8
ZP12
ZP11
ZP15
ZP14
ZP13
unused 3*64*S-224 or 192*S-224 bytes


ZMASK = ZPLANE2 2 Sample
ZMASK = ZPLANE4 2 Sample
ZMASK = ZPLANE8 2 Sample
ZMASK = ZPLANE16 2 Sample
Stencil 64*S bytes (only if 24 bit depth)
ZP0
Pmask
ZP1
unused 3*64*S-32 or 192*S-32 bytes
ZP2 ZP1 ZP0
ZP4 ZP3
unused 3*64*S-64 or 192*S-64 bytes
unused
ZP7 ZP6 ZP5
unused 3*64*S-120 or 192*S-120 bytes
ZP10 ZP9 ZP8
ZP12 ZP11
ZP15 ZP14 ZP13
unused 3*64*S-224 or 192*S-224 bytes


ZMASK = ZPLANE2 4 Sample
Stencil 64*S bytes (only if 24 bit depth)
Pmask
ZP1 ZP0
unused 3*64*S-32 or 192*S-32 bytes
ZMASK = ZPLANE4 4Sample
Stencil 64*S bytes (only if 24 bit depth)
Pmask
ZP2 ZP1 ZP0
ZP4 ZP3
unused 3*64*S-64 or 192*S-64 bytes
ZMASK = ZPLANE8 4 Sample
Stencil 64*S bytes (only if 24 bit depth)
Pmask
ZP2 ZP1 ZP0
ZP4 ZP3
ZP7 ZP6 ZP5
unused 3*64*S-120 or 192*S-120 bytes
ZMASK = ZPLANE16 4 Sample
Stencil 64*S bytes (only if 24 bit depth)
Pmask
ZP2 ZP1 ZP0
ZP4 ZP3
ZP7 ZP6 ZP5
ZP10 ZP9 ZP8
ZP12 ZP11
ZP15 ZP14 ZP13
unused 3*64*S-224 or 192*S-224 bytes
ZMASK = ZPLANE2 8 Sample
Stencil 64*S bytes (only if 24 bit depth)
Pmask
ZP1 ZP0
unused 3*64*S-32 or 192*S-32 bytes
ZMASK = ZPLANE4 8Sample
Stencil 64*S bytes (only if 24 bit depth)
Pmask
ZP2 ZP1 ZP0
ZP4 ZP3
unused 3*64*S-64 or 192*S-64 bytes
ZMASK = ZPLANE8 8 Sample
Stencil 64*S bytes (only if 24 bit depth)
Pmask
ZP2 ZP1 ZP0
ZP4 ZP3
ZP7 ZP6 ZP5
unused 3*64*S-120 or 192*S-120 bytes
ZMASK = ZPLANE16 8 Sample
Stencil 64*S bytes (only if 24 bit depth)
Pmask
ZP2 ZP1 ZP0
ZP4 ZP3
ZP7 ZP6 ZP5
ZP10 ZP9 ZP8
ZP12 ZP11
ZP15 ZP14 ZP13
unused 3*64*S-224 or 192*S-224 bytes

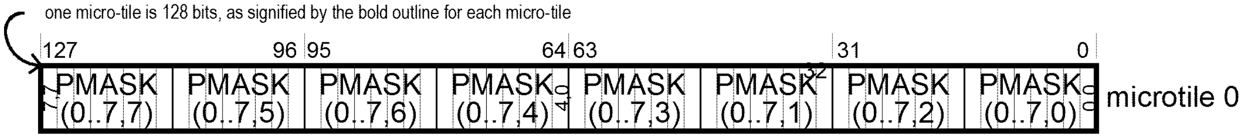

ZMASK = ZPLANE2 3 Sample
ZMASK = ZPLANE4 3Sample
ZMASK = ZPLANE8 3 Sample
ZMASK = ZPLANE16 3 Sample
ZMASK = ZPLANE2 6 Sample
ZMASK = ZPLANE4 8Sample
ZMASK = ZPLANE8 8 Sample
ZMASK = ZPLANE16 8 Sample
Stencil 64*S bytes (only if 24 bit depth)
Pmask
ZP0
ZP1
ZP2
ZP3
ZP4
ZP5
ZP6
ZP7
ZP8
ZP9
ZP10
ZP11
ZP12
ZP13
ZP14
ZP15
unused 3*64*S-32 or 192*S-32 bytes
unused 3*64*S-64 or 192*S-64 bytes
unused 3*64*S-120 or 192*S-120 bytes
unused 3*64*S-224 or 192*S-224 bytes



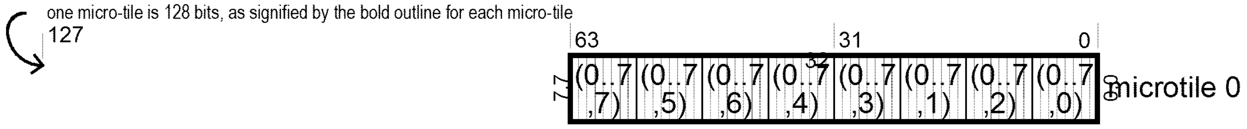
one micro-tile is 128 bits, as signified by the bold outline for each micro-tile

| 127 | 96 95 | 64 63 | 32 31 | 0 | |
|---|---|---|---|---|---|
| 7,3 | PMASK (0..7,3) | PMASK (0..7,1) | PMASK (0..7,2) | PMASK (0..7,0) 0,0 | microtile 0 |
| 7,7 | PMASK (0..7,7) | PMASK (0..7,5) | PMASK (0..7,6) | PMASK (0..7,4) 0,4 | microtile 1 |

*R400 RB Depth.vsd:4 bit microtile PMASK FB Format* *by jayw 9/8/2003*



*R400 RB Depth.vsd:2 bit microtile PMASK FB Format* *by jayw 9/8/2003*

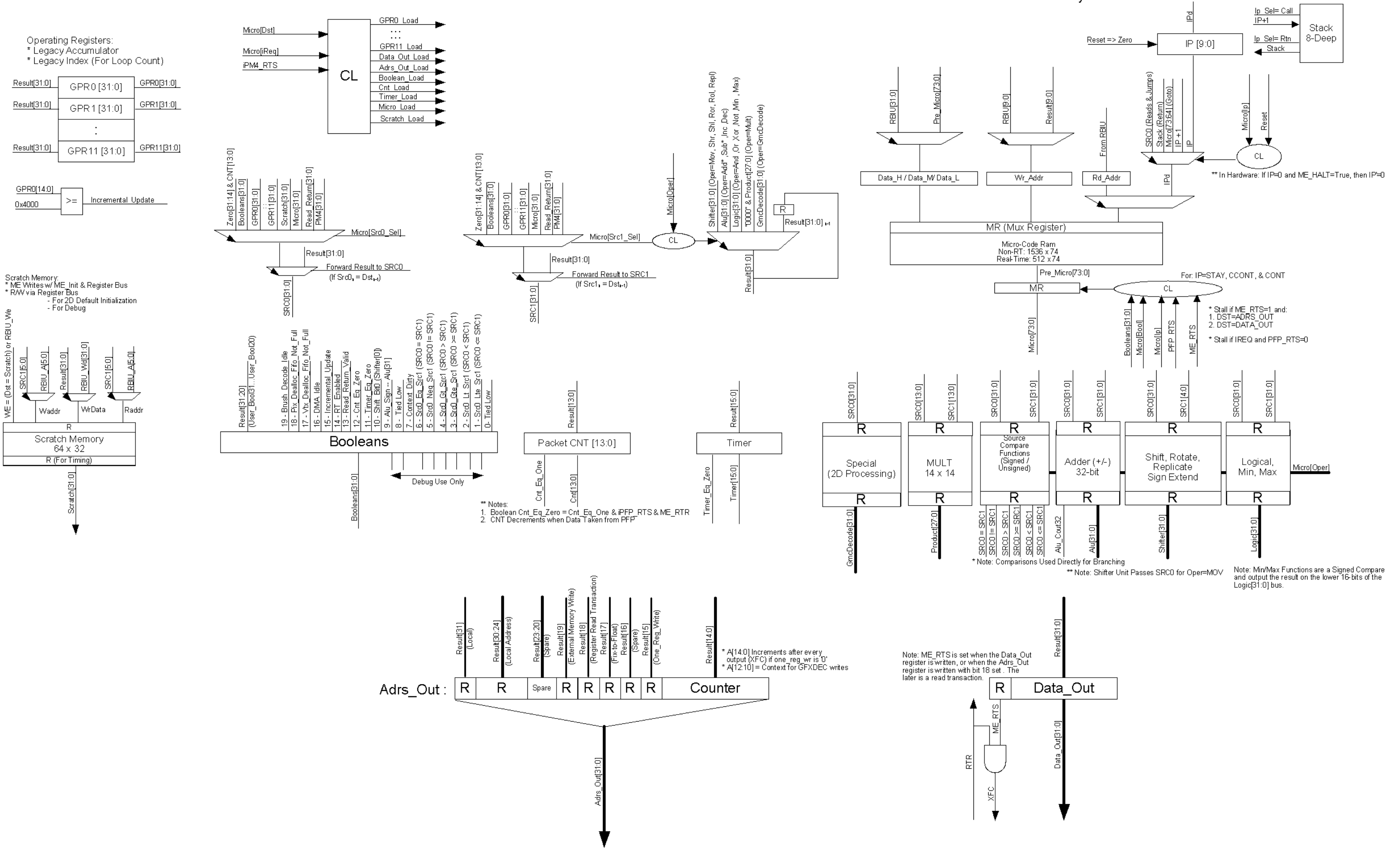


one micro-tile is 128 bits, as signified by the bold outline for each micro-tile

R400 RB Depth.vsd:1 bit microtile PMASK FB Format by jayw 9/8/2003

# R400 CP Micro Engine Details

Updated: 9/23/2003
John A. Carey



** In Hardware: If IP=0 and ME_HALT=True, then IP'=0

* Stall if ME_RTS=1 and:
1. DST=ADRS_OUT
2. DST=DATA_OUT

* Stall if IREQ and PFP_RTS=0

** Notes:
1. Boolean Cnt_Eq_Zero = Cnt_Eq_One & iPFP_RTS & ME_RTR
2. CNT Decrements when Data Taken from PFP

* Note: Comparisons Used Directly for Branching

** Note: Shifter Unit Passes SRC0 for Oper=MOV

Note: Min/Max Functions are a Signed Compare and output the result on the lower 16-bits of the Logic[31:0] bus.

* A[14:0] Increments after every output (XFC) if one_reg_wr is '0'
* A[12:10] = Context for GFXDEC writes

Note: ME_RTS is set when the Data_Out register is written, or when the Adrs_Out register is written with bit 18 set. The later is a read transaction.

| Opcode Tasks | GetBorderColorFraction | GetCompTexLOD | GetGradients | GetWeights | SetTexLOD | SetGradients{H,V} | FetchVertex | FetchTextureMap | FetchMultiSample |
|---|---|---|---|---|---|---|---|---|---|
| tp_input | BORDER_COLOR = ARGB White<br>BORDER_SIZE = 0<br>DATA_FORMAT = FMT_8<br>FORMAT_COMP_X = unsigned<br>NUM_FORMAT_ALL = RF | DATA_FORMAT = FMT_16<br>FORMAT_COMP_X = signed<br>NUM_FORMAT_ALL = INT | DATA_FORMAT = FMT_16_16_FLOAT | DATA_FORMAT = FMT_8_8<br>FORMAT_COMP_X = unsigned<br>NUM_FORMAT_ALL = INT | Normal | Normal | DIM = 1D<br>Different state mux'g | Normal | DIM = 2D |
| tp_lod_deriv | Normal | Normal (don't care) | Normal (don't care) | Normal | Convert to 16-bit float for tp_lod_getset | Use pix_mask to find 1st fetch_addr<br>Convert to 16-bit float for tp_lod_getset | Normal | Normal | Normal |
| tp_lod_aniso | Normal | aniso dx = {1'b0, comp_lod}<br>(output on aniso achieves broadcast) | lod p3..p0 = dydv, dydh<br>cycle 0: aniso dy,dx = dxdv, dxdh | Normal | Outputs zeroed to single cycle | Outputs zeroed to single cycle | Zero outputs to single cycle | Normal | Normal |
| tp_lod_fifo | Normal | Normal | cycle 1: aniso dy,dx = dydv, dydh<br>cycle 0: aniso dy,dx = dxdv, dxdh | Normal | Normal | Normal | Different state mux'g | Normal | Normal |
| tp_lod_getset | N/A | N/A | N/A | N/A | Separate storage for get/set, grad/lod | Separate storage for get/set, grad/lod | N/A | N/A | N/A |
| tp_addresser | Normal | Pass LOD thru<br>TA_FA_ws[3:0] = adx[15:12]<br>TA_FA_yf[5:0] = adx[11:6]<br>TA_FA_xf[5:0] = adx[5:0]<br>(can make exactly like GetGradients)<br>Remove logic to zero valids for this opcode | Pass LOD thru<br>TA_FA_y_inc = ady[15]<br>TA_FA_x_inc = ady[14]<br>TA_FA_coord[1:0] = ady[13:12]<br>TA_FA_mip_id[3:0] = ady[11:8]<br>TA_FA_ws[11:0] = { ady[7:0], adx[15:12] }<br>TA_FA_yf[5:0] = adx[11:6]<br>TA_FA_xf[5:0] = adx[5:0]<br>Remove logic to zero valids for this opcode | Normal | Normal | Normal | Mux in vertex control<br>Zero blend fractions | Normal | Normal |
| tp_fetch | Clear TC valids<br>Mux 0 for texel, 1 for border data to tp_ch_blend | Clear TC valids<br>Put LOD in tp_rr_fifo in 4 top-left texels as 16-bit fixed | Clear TC valids<br>Put d*from tp_rr_fifo in 4 top-left texels as 16-bit floats | Clear TC valids<br>wh->x,z, wv->y,w to tp_ch_blend | Clear TC valids | Clear TC valids | Normal | Normal | Normal |
| tp_pipe_valids | Normal | Normal | Normal | Normal | Normal (don't care) | Normal (don't care) | Normal | Normal | Normal |
| tp_ch_blend | Normal | Normal | Normal | Normal | Normal (don't care) | Normal (don't care) | Different state mux'g | Normal | Normal |
| tp_tt | Normal | Normal | Normal | Normal | Normal (don't care) | Normal (don't care) | Different state mux'g | Normal | Normal |
| tp_hicolor | Normal | Normal | Normal | Normal | Normal (don't care) | Normal (don't care) | Different state mux'g | Normal | Normal |
| sp_tp_formatter | Normal | Normal | Normal | Normal | Normal (don't care) | Normal (don't care) | Different state mux'g | Normal | Normal |
| tpc_fifos | DATA_FORMAT = FMT_16<br>FORMAT_COMP_X = signed<br>NUM_FORMAT_ALL = RF<br>EXP_ADJUST_ALL = 0 | DATA_FORMAT = FMT_16<br>FORMAT_COMP_X = signed<br>NUM_FORMAT_ALL = INT<br>EXP_ADJUST_ALL = -7 | DATA_FORMAT = FMT_16_16_FLOAT<br>EXP_ADJUST_ALL = 0 | DATA_FORMAT = FMT_8_8<br>FORMAT_COMP_X = unsigned<br>NUM_FORMAT_ALL = INT<br>EXP_ADJUST_ALL = -8 | Normal (don't care) | Normal (don't care) | DIM = 1D<br>Different state mux'g | Normal | DIM = 2D<br>Different state to TC |
| opcode_enc[2:0] | 4 (tp_parameters.v) | 5 (tp_parameters.v) | 6 (tp_parameters.v) | 7 (tp_parameters.v) | 3 (tp_parameters.v) | 3 (tp_parameters.v) | 0 | 1 | TBD |

| BCP B RTL | Affected blocks | Owner | |
|---|---|---|---|
| shrink u*_FL_TA_lod_grad from 16 to 10 bits | tp.tree, tp_addresser | Manoo | |
| aniso walk optimizations/step size changes | tp_lod_aniso | Tien | RTL in, pending EMU/release |
| | tpc_walker | Tien | RTL in, pending EMU/release |
| | tp_lod_fifo | Tien | RTL in, pending EMU/release |
| | tp_addresser | Tien | RTL in, pending EMU/release |
| add Ws bit | tp_* | Manoo, Steve | release in progress |
| | | | |
| Post BCP/Optional? | | | |
| Encoded DATA_FORMAT optimzations | | | |
| tp_pipe_valids - channel-encoded data formats for tp_hicolor | | | |
| tp_border | | | |
| sp_tp_formatter - anyhwere full texture data format is used | | | |
| remove of 3,6,8 multisample | tp_lod_deriv | | Simple, rerun emu-generated .mc include, remove cases from testbench |
| LOD gradient precision issue | | | Artifacts with larg tex maps (2K and higher) |
| support denorms in LOD blocks | tp_lod* | | |
| possible optimizations with explicit 1 logic | tp_lod* | | |
| FMT_1_1_1_1 | various | | |
| stackable textures | many | | |
| expand opcode enc | many | | |
| implement NO_ZERO RF expand and verify | tp_hicolor, emu | | 2 wks design/verify, already supported for vFetch formats |
| | | | |
| EMU Tasks | | | |
| opcodes (FetchMultiSample only) | | Jocelyn | |
| border color, various formats | | Steve | pending release |
| tp_sqsp.dmp | | Jocelyn | |
| shrink u*_FL_TA_lod_grad from 16 to 10 bits | | | |
| aniso walk optimizations/step size changes | | | |
| add Ws bit | | Joceyln | release in progress |
| LOD gradient precision issue | | Jocelyn | Post BCP/Optional? |
| implement NO_ZERO RF expand and verify | | Jocelyn | Post BCP/Optional? |
| face_id in top_left_coord MSBs instead | | Jocelyn | Post BCP/Optional? |
| hardware accurate TPC_TC_state_rts | | Jocelyn | Post BCP/Optional? |
| | | | |
| Verification Tasks | | | |
| Opcodes | See above | | |
| pre-RF expand cleanup/signed support | tp_fetch, tp_pre_rf_* | | |
| Expand simd to 2 bits | Various | | |
| proper vtx rf expand enable to sp_tp_formatter | tpc_fifos? | | |
| pipe thru pix_mask to formatter | sp_tp_formatter | | |
| verify 32bpp RF expansion | tp_hicolor, emu | Steve, Jocelyn | |
| tp4_tc/gc level debug | | Chris, Ray, Kevin, and the rest of us… | |
| Cubic verification | | | |
| standalone tp_lod_deriv, tp_addresser | | Chris, Vishal | |
| tp4_tc | | George | |
| tp_addresser | | | |
| Texture Border, 3D case | | Manoo, Jocelyn | |
| Interlaced Field Bit | | Manoo | |
| Vertex Fetch channel add, channel is non-zero | | Manoo | |
| tp_addresser standalone | | | |
| pix_mask | tp_addresser | Manoo, George, Jocelyn | |
| aniso | tp_addresser | Manoo, George, Jocelyn | |
| test plan | | George, Carlos, Tien | Complete item-by-item for us, summary req'd for Microsoft |
| updated aniso mode control | tp_lod_aniso | Tien | Stupid constant changed a while back .. After RTL was in of course |
| pre-RF expand assertions for some formats | EMU | Jocelyn | |
| unsigned biased border color generation | EMU | Steve | |
| tp_lod_aniso mismatches | | Jocelyn, Tien, Vishal | |
| Vertex Fetch channel add, channel is non-zero | tp_addresser | Manoo | |
| xenos reduncancy | tp_input/tp_output | | |
| connect proper cube map control | tp_input | | 50% there, need DIM field to move |
| | tp_addresser | | |
| texture border (BORDER_SIZE = 1) | | Jocelyn | |
| | | | |
| Misc Tasks | | | |
| debug regs review | | Vishal?? | RTL, .blk file, verification tasks |
| perf regs review | | George?? | RTL, .blk file, verification tasks |
| Coverage | tp_* | block owners, Vishal, George | |
| Update TP spec | tp_* | Jocelyn | |

|  | ORIGINATE DATE | EDIT DATE | DOCUMENT-REV. NUM. | PAGE |
|---|---|---|---|---|
|  | 13 October, 2000 | [date \@ "d MMMM, | Version 0.16 | 1 of 42 |

AUTHOR: John Carey

| ISSUE TO | COPY NO. |
|---|---|
|  |  |

# Specification of the Register Backbone Manager (RBBM) R400

## THIS DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

APPROVALS

| Name/Dept | Signature/Date |
|---|---|
|  |  |
|  |  |
|  |  |
|  |  |

Remarks:

# 1. Table of Contents

## 2. Table of Figures

# 3. Revision History

| | | |
|---|---|---|
| 05 July 01 | Version 0.000 | Baseline from R300 (Khan) Version 1.16 Specification. ***Spec does not represent R400 Design Decisions***. -- J. Carey |
| 23 July 01 | Version 0.001 | Sweeping changes to bring spec closer to R400 desires. Old text not relevant has been either deleted or striked-out. – J. Carey |
| 10 August 01 | Version 0.01 | Read Return Bus Diagram and Text, Register to Capture Read Error Addresses, … |
| 18 October 01 | Version 0.02 | Sweeping changes to establish a POR. – John Carey |
| 16 Nov. 01 | Version 0.04 | Version 0.03 officially not released. Added Diagrams for Transactions, Removed CGM, HDP, Updated Interfaces for MC, SQ, SX, and RB. Added "Slow Client" transactions. Go Signals to CG are programmable. |
| 23 January 02 | Version 0.05 | Add Implicit 2D←→ Synchronization. Combined VGA, TVOUT, VIP, and Display RTR signals into DC_RBBM_RTR and DC_RBBM_nrtRTR. Clarified Interrupt Generation from the RBBM. Global Register Bus (GRB) Output is 16:2 {128 KBytes}. Added VGA Clock Go/Active to RBBM. Some Clarifications on Address Mapping. |
| 12 Feb. 02 | Version 0.06 | Removed bits 19:17 from HI_RBBM_A. Clarified Host and Command Data Swapping. Renamed core clock input to SCLK_P_RBBM. Updated slow client protocol. Added DMA Init Skew Control. Added signals for 2*N determination. |
| 13 Feb. 02 | Version 0.07 | Release for Review. No Changes from Version 0.06. |
| 28 Feb. 02 | Version 0.08 | Debug Bus Connections, Command FIFO Size Adjustment, Skew Control Update. Combined all interrupts from DISP to DISP_RBBM_int and combined all interrupts from VIP to VIP_RBBM_int, and renamed IDCT interrupt. Renamed clock to SCLK_P (no unit designation). Added interrupts inputs for all clients. Unused interrupts will be tied low at the RBBM instance. |
| 26 March 02 | Version 0.09 | Added ROM Go/Active Pair. Updated BIF-to-RBBM Interface for slip buffer. Clarification of stall for wait_until condition. Added four go/active pairs for the MC clock assertions per the Power Management Meeting on 03-06-2002. Removed weights for arbitration and clarified arbitration. Updates from review with Tushar Shah. Updates on wait_until signals from David Glen. Fix signal names and comments on CP-to-RBBM interface. Add note on transactions that do not decode to a block. Updates from BIF and VIP review: HI renamed to BIF, added soft reset for SC. Removed WAIT_IDCT_SEMAPHORE from RT stream per IDCT design review. |
| June 17 02 | Version 0.10 | Renamed signals from Display/Overlay engine. Added Idle signals to CP. Removed Stat_Gui_Idle signal. Updated wait_until per definitions of signals from the display/overlay engine. Added VGT's soft reset signal. RBBM asserts RBBM_regclk_active during reset. Added MASTER_INT_SIGNAL interrupt register and removed references to the GEN_INT_* registers. Fix typo in RBBM_PERF_CNTL register. Added CGM interface rtr's. Added RBBM_STATUS2 register. Removed Reference of "Go" signals from spec. Removed HIRQ_PENDING from GUI_ACTIVE equation. Add VGT_RBBM_no_dma_busy and remove slow client protocol. Added ROM soft Reset. |
| October 29, 2002 | Version 0.11 | Updates for Performance Counters. |
| Nov. 20 2002 | Version 0.12 | Rename Field in RBBM_Status Register. |
| Jan. 7 2003 | Version 0.13 | Add NQ Wait Equation to Spec. |
| March 24, 2003 | Version 0.14 | Fix Typo in Section 7.2. |
| April 24, 2003 | Version 0.15 | Add interface signals to DB block. |
| April 25, 2003 | Version 0.16 | Only adding RBBM_DB_soft_reset for DB block addition. |

# 4. Open Issues / Items to Do (Updated: 04-26-2002)

1) Possible removal of all shared registers. If so, the RBBM would implement the MASTER_INT_SIGNAL Read only register. Each bit is active (high) only if that block is currently asserting a interrupt signal to RBBM. *A : 04-26-2002. This is plan of record per a meeting with Phil Rogers, Rodney A. and Jeffrey C. on 04-23-2002.*

2) Need to resolve the performance counter connections. *A: 04-10-2002: All busy signals will be wired to the performance counters.*

3) Need review of the status signals that are listed in the RBBM_STATUS register. *A: Done with Mark Fowler and Steve Morein on 04-10-02.*

4) For "idleclean" (and idle), the RBBM either gets "idle" and "clean" signals from all the clients OR the "empty" status of the "drain-o" FIFO in the CP is wired to the RBBM. In the later case, the Driver would need to insert an "event drain-o" before the wait_until for idleclean. The RBBM would then wait until the "drain-o" FIFO is empty. *A: 04-09-2002: The RBBM will receive "busy" and "clean" signals from each of the clients and use these for the wait idle/clean function.*

5) Need to resolve the WAIT_UNTIL signals for Real-Time and Non-Real-Time transactions. *A: Done.*

6) Do we need strap bits to tell the RBBM the number of "repeater flops". *A: 04-01-2002: No, the strap signals would need connection to the top-level route. The RBBM will assume a fixed number of repeater flops maximum.*

7) Do we need a WAIT_IDCT_SEMAPHORE on the RT stream path? *A: No per IDCT design review on 03-26-2002. The WAIT_REG_MEM function will be used instead for RT streams.*

8) Need feedback on the list of Go/Active signal pairs output by the RBBM (S. Morein Action). It is expected that the list will get shorter. *A: RBBM outputs single "reg_active_sclk" signal to clients instead of go/active signals.*

9) Possible Go/Active pair addition for controlling the registers for the read bus and repeater flops. *A: RBBM outputs single "reg_active_sclk" signal to clients instead of go/active signals.*

10) Need address that RBBM should use for queued/non-queued determination for Host Transactions. *A: From the RBBM design review, it was determined to remove the Host's queued path through the RBBM.*

11) The SLICEDONE wait_until is replaced with some wait reg or waits on a frame buffer address equaling a certain value. The RTS rendering the overlay could write these values and the primary PM4 stream waiting on the real-time can poll on the address containing the expected value to proceed (D. Glen). *A: Removed the STAT_DISP_OV0_SLICEDONE and WAIT_OV0_SLICEDONE wait logic.*

12) Interrupts from Video Blocks. *A: D. Glen: TV, HI and AIC do not need interrupt lines into RBBM. VGA, DISP and VIP do.*

13) Possible Go/Active Additions: 4 for RB. *A: 03-06-2002 (Power Management Meeting) – No. These may need 4 busy signals back to the CG, but they only need one clock control per unit.*

14) Alternative for "Slow Client" mode documented by David Glen. *A: 02-11-2002: Slow Clients will send pulse train to the RBBM.*

15) Need to resolve how to decode the upper bits [19:17] of address from the Host? *A: 01-29-2002: Address bits 19:17 will not be wired to CP. VGA registers are memory-mapped registers, IO decode addresses are the same as memory-mapped registers, and the BIOS is written BIF →MH.*

16) Adding the ability of the Host "Queued" transactions to be send down the "Non-Queued" path via a debug bit may not be necessary if the register map is duplicated where each location has both a "Queued" and "Non-Queued" address. *A: The address map is not duplicated, so it is included in the RBBM design.*

17) Need to finalize address width after register specification is released. A: Global Register Bus addresses 128 KBytes.

18) Need to resolve the Interrupt signals into the RBBM. Perhaps these should just be input into the BIF? A: No. The RBBM generates a GUI_IDLE interrupt and an interrupt for read errors.

19) Perhaps the 2D ←→ 3D synchronization function belongs in the CP? A: Yes. But the RBBM still has the WAIT_IDLE and CPSCRATCH implicit synchronization.

20) Should Host read transactions to "queued" registers be sent down the "queued" path so that their result reflects the affect of any initiators that proceeded? The read may stall however if a WAIT_UNTIL is in-progress which may hinder debug in a "hang" situation. A. Debug bit allows all host read transactions to go through non-queued path.

21) How does the RBBM handle the assembly of read data from shared registers (i.e. Registers with bits spread across multiple units). A: CP read logic accumulates the shared data until a "read latency timer" expires. There is a decoder in the RBBM to identify shared registers and the read latency for these registers is based on the "read latency timer".

22) Where is the Host Blit Immediate Data Swapping Function? *A: Per conversation on 08-23-01 with S. Morein, the CP will skip fetching of the Immediate Host data. The 3D engine will fetch the immediate data and do the data swapping. The CP will need some mechanism (Read Address Jump or Speculative Prefetching) to be able to skip over data in the command stream. 10-17-2001: The data swapping logic will be removed from the RBBM. The swapping logic will be either in the Legacy 2D logic or will be in the shader where the data will be fetched.*

# 5. R400 Overview (Updated: 06-17-2002)

The RBBM merges register writes and reads from the Bus Interface Unit (BIF a.k.a. Host) and the Command Processor (CP) then broadcasts them to the rest of the blocks in the chip. The RBBM can access up to 128K Bytes of register space (32K DWORDs).

Transactions from the CP can be sent down either a queued (CF) or non-queued (CP) path. The RBBM looks at the NQ flag from the CP. If the NQ flag is set, the transaction from the CP is sent down the non-queued path. See the CP unit specification for details on controlling the NQ flag.

In addition, there are two other paths for transactions from the CP – a Real-Time (RT) path and a path that the Command Processor uses to issue Index DMA requests from its Pre-Fetch Parser (PF Path). Separate SEND signals from the CP are used to distinguish these transactions.

In general, non-queued transactions can pass queued transactions. If it is important for non-queued register writes to be held off by a queued register write the CP must not send the non-queued register write until it has determined that the queued register write has completed. The CP can use chip status (i.e. reading chip status from registers) to perform this synchronization.

There is no ordering between the CP and Host transactions within the RBBM. Writes from both senders may become interleaved on the global register bus.

Real-Time transactions from the CP will pass both the non-queued and queued transactions. These have the highest priority in the arbitration for access to the global register bus.

The CP's Pre-Fetch Parser (PFP) issues initiators to the VGT's Index DMA engine via the PF path. These requests pass the CP's Micro Engine and other "queued" transactions in the RBBM. This is done so that the Index DMA requests will over-lap the writing of state data. This is a low bandwidth, high priority path. These transactions have the next highest priority after Real-Time transactions.

Arbitration Summary for the Global Register Bus:

> Real-Time: Always Wins. Removed from Arbitration if RTRs are not asserted.
> Index DMA (PF): Always Wins if No Real-Time. Removed from arbitration if skew count limit exceeded or if corresponding ready-to-receives are not asserted.
> HI, CF, CP: Round-Robin Priority. Each removed from arbitration if corresponding ready-to-receives are not asserted.

Unlike prior implementations, the RBBM does not perform address re-mapping. All the clients to the RBBM "see" the entire flat address map.

In R400, clock gating is performed in the clients. Therefore there are no handshaking signals between the RBBM and CG for enabling clocks as in past chips. The RBBM simply asserts the RBBM_regclk_active signal to all clients whenever it has a transaction in any of its data paths.

Neither the BIF nor the CP will be allowed to issue more that one read request at any give time. This is enforced by logic in the RBBM. There will be only one outstanding read on the GRB at any one time. The RBBM will allow write requests to be processed if a read transaction is outstanding however.

## 5.1 Implicit Synchronization (Updated: 03-11-2002)

In prior chips, the RBBM implemented 2D/3D synchronization. This involved stalling a 2D initiator/trigger register write if 3D was not idle and stalling a 3D initiator/trigger register write if 2D was not idle. In R400, there is no separate 2D engine. The RBBM therefore cannot distinguish between a 2D or 3D initiator. The CP therefore performs the 2D ←→ 3D synchronization in R400. Note that if the Driver is not using the CP for 2D/3D, then it needs to do this synchronization itself.

The RBBM implements the following legacy implicit synchronization: ISYNC_WAIT_IDLEGUI and ISYNC_CPSCRATCH_IDLEGUI in the queued (CF) pipeline. See the RBBM_ISYNC_CNTL register for details.

## 5.2 Client Clock Synchronization (Updated: 06-17-2002)

At the overall chip level, the RBBM is responsible for making sure that a client of a transaction has its clock running. Whenever the RBBM receives a transaction from either the BIF or CP, it asserts the "Register Clock Active" signal. The latency of the RBBM will provide for several clocks of the "active" signal being asserted before the transaction is presented on to the Global Register Bus (GRB).

The RBBM will keep the "Register Clock Active" signal asserted as long as it has a transaction in any of its pipelines.

The RBBM will de-assert the "Register Clock Active" signal after a programmable number of clocks after the last transaction has been issued. The "issued" condition includes the return of any read data or the terminal count of the READ_INTERVAL counter (Multi-Target Registers and Read Error Condition).

The programmability of the de-assertion time for the "Register Clock Active" signal is provided as the REGCLK_DEASSERT_TIME in the RBBM_CNTL register.

## 5.3 Interrupt Generation (Updated: 04-26-2002)

The RBBM has interrupt control, status, and acknowledge registers, which are used to control the operation of the interrupts generated by the RBBM.

The RBBM generates the following interrupts:

1. Non-Real-Time GUI Idle
2. Read Error

The RBBM is also the collection point for the discrete interrupt signals in the chip. Interrupt signals from the clients are input into the RBBM, logically "OR'd" with each other -- and the RBBM's internal interrupt -- registered, and sent out as a single interrupt to the Bus Interface unit (BIF). All clients have an interrupt input. If a client does not implement an interrupt, then its corresponding input to the RBBM is tied low at the RBBM's instantiation.

The RBBM has a register that can be read by the Driver to determine which unit is interrupting: MASTER_INT_SIGNAL.

## 5.4 Host and Command Data Swapping (Updated: 01-31-2002)

The RBBM performs endian swapping on the data from the Bus Interface Unit (BIF). When the RBBM_BIF_swp signal from the BIF is set the RBBM will perform the following swap of on the data:

0xAABBCCDD becomes 0xDDCCBBAA

The RBBM_BIF_swp signal only applies to transactions from the BIF that are not Command Stream Push data. For Command Stream Push data (i.e. writes to the CP_PUSH_* registers), only the CPQ_DATA_SWAP control in the RBBM_CNTL register is used to determine if a swap should occur.

Note that the byte enables are not ordinarily swapped with the data. There is however a SWAP_BE control bit in the RBBM_DEBUG register. If this bit is set and RBBM_BIF_swp is set, then the byte enables are also swapped for both BIF and Command Stream Push data.

Note that byte enables are swapped for CP_PUSH_* write requests when RBBM_BIF_swp and SWAP_BE are set. Where as for data to be swapped, RBBM_BIF_swp may not be set but CPQ_DATA_SWAP should be set.

## 5.5 What Happened to Upper Address Bits from BIF? (Updated: 02-25-2002)

In prior chips, the RBBM decoded address bits [19:16] from the BIF to determine the following decode spaces – MMR, IO, VGA, and BIOS.

Bit 16 was used to determine BIOS0 or BIOS1.

Bit 17 was not used in prior RBBM designs.

Bits 19:18 were used to determine the decode spaces as follows:

MMR_DEC = BIF_Address[19:18] = "00"

IO_DEC = BIF_Address[19:18] = "01"

VGADEC = BIF_Address[19:18] = "10"

BIOSDEC = BIF_Address[19:18] = "11"

In R400, the memory-mapped address space is 128KBytes. Bit 16 is used as the most-significant bit of the address.

Bits 19:17 are not needed by the RBBM because:

1. The address re-mapping logic is not in the R400 RBBM, so the Memory Mapped Registers (MMR) and IO decode space must have the same address 16:2.
2. The VGADEC register space maps directly to the Memory-Mapped Registers, so detection of VGADEC is no longer needed.
3. The ROM is accessed via the register bus for configuration, writing, and indirect reading. These locations are within the 128KBytes of memory-mapped register space.

Therefore the address width from the BIF to the RBBM is only 16:2, which represents the 32K DWORDs that make-up the 128K Byte Memory-Mapped Register aperture.

Note: The BIOS (ROM) image is accessible via read requests to the Memory Hub.

## 5.6 Reset Behavior (Updated: 04-06-2002)

The RBBM will drive all '0's onto both Write Data buses during reset, and hold that value on the data bus after reset, until the first request comes in from one of the requesters.

The RBBM will also assert the RBBM_regclk_active signal during both hard reset and soft reset so that the zero value will propagate through any client's interface.

## 5.7 VGT DMA Draw Initiator Deadlock (Updated: 06-05-2002)

The CP's Pre-Fetch Parser (PFP) issues index DMA requests to the VGT by writing the VGT_DMA_BASE and VGT_DMA_SIZE registers for the DRAW_INDX packets. Because we desire to hide the fetch latencies of the index DMA operations, these DMA requests are issued long before the corresponding DRAW_INITIATOR is sent to the VGT. Both the DMA request and the DRAW_INITIATOR enter the VGT through the same interface. So, care must be taken so as not to write so many DMA requests that the interface into the VGT is backed-up. This would block the ability for the CP to write the DRAW_INITIATOR. If this happens then the chip is deadlocked.

To prevent this situation, the RBBM will increment a counter every time it writes the VGT_DMA_BASE or VGT_DMA_SIZE register through the PF path. The counter is decremented by 2 when the RBBM writes to the VGT_DRAW_INITIATOR register with the SOURCE_SELECT field set to "VGT DMA Data" is sent to the Global Register Bus.

The count is reset to zero on reset. The RBBM will stop issuing index DMA requests to the VGT (i.e. stall the PFP path) when the count value is greater than or equal to the SKEW_TOP_THRESHOLD. The SKEW_TOP_THRESHOLD value programmed in the RBBM_SKEW_CNTL register. The SKEW_TOP_THRESHOLD value must be a non-zero even number.

## 5.8 Support for Explicit Synchronization (Updated: 04-02-2002)

There are two WAIT_UNTIL event engines – one for Real-Time Streams and another one for non-Real-Time Streams. Each of these has its own independent "WAIT_UNTIL" control register. The operation of the WAIT_UNTIL event engines however is identical.

A write to the "WAIT_UNTIL" register is actually written into the Command FIFO. The value written is a command to the Event Engine, which resides at the output of the Command FIFO. The value specifies that a certain status condition should be met before allowing the next command to be read from the Command FIFO. Essentially, the value written into the "WAIT_UNTIL" register is a read mask for reading consecutive data from that FIFO. If multiple bits in the write value are ON, then ALL conditions must be met before the write can be un-blocked.

Typically, the write to the wait_until register is preceded with a write to an initiator register. This initiator changes the status that the subsequent wait_until condition is programmed to wait for. When the RBBM sees a WAIT_UNTIL register write on the output of the Command FIFO, it will not evaluate the status signals until all prior write transactions have made it to the client. The RBBM ensures this by waiting for the following condition to be met:

Wait_For = "All CP Non-Queued and Prior Queued (CF) Transactions Out of RBBM" + 48 Clocks

The reason for making sure non-queued transactions are complete is because Initiators may also be sent through non-queued path.

The 48 extra clocks allow time for the client to process the initiator transaction and for the status signals to stabilize.

Reads from the WAIT_UNTIL registers return the value on the status signals that qualify the wait condition.

For example, if the value being written has the WAIT_CRTC_VLINE bit ON, then consecutive data from the FIFO will not be read until the CRTC_VLINE condition becomes true.

Another example is the WAIT_CMDFIFO bit. This allows the programmer to stall the command stream at the bottom of the Command FIFO until there are at least CMDFIFO_ENTRIES number of occupied entries in the Command FIFO behind it. This gives the programmer a mechanism to guarantee that a certain sequence of writes will occur in rapid succession once the stall condition has been met. The WAIT_CMDFIFO is only provided in the non-Real-Time queued path.

## 5.9 NQ Wait Until (Updated: 02-07-2003)

The Non-Queued Wait Until function has the same operation as in prior chips. If either the BIF or the CP writes the NQWAIT_UNTIL register, subsequent transactions will not be processed until the GUI is idle. The BIF writes directly to this register and the CP uses the WAIT_FOR_IDLE packet. In either case, the wait operation is before the decision of "queued" versus "non-queued". Unlike the explicit sync, this wait function also affects the non-queued paths – thus its name.

If initiated directly from the Host, the RBBM waits for all prior transactions in the Host to be flushed, all transactions in the CF pipe to be flushed, and then it waits for 48 clocks before sampling the Non-RT GUI_ACTIVE to be idle.

If initiated directly from the CP, the RBBM waits for all prior transactions in the CF pipe to be flushed and all transactions in the CP paths' scheduler to be flushed, and then it waits 48 clocks before sampling the Non-RT GUI_ACTIVE to be idle.

The 48 clock wait is to make sure all the pending initiators have been received by the clients (i.e. Not stuck in the repeater flops).

```
Non-RT GUI_ACTIVE = ( not CP_RBBM_rt_enable and ( RC_RBBM_cntx0_busy or SQ_RBBM_cntx0_busy or SC_RBBM_cntx0_busy) ) or
                    ( PA_RBBM_busy or VGT_RBBM_no_dma_busy or SQ_RBBM_cntx17_busy or RC_RBBM_cntx17_busy or
                      SC_RBBM_cntx17_busy );
```

## 5.10 Field Affecting Operation (Updated: 10-24-2002)

Whenever BIF or CP requests for a write operation for either RBBM_CNTL or RBBM_SKEW_CNTL register, all the subsequent requests in any of the RBBM pipes are held off till the RBBM write operation is complete.

# 6. Register Backbone Protocols

This section is intended to summarize the constraints that the Register Backbone structure places on targets that reside on that backbone. It is a centralized place where designers of blocks that have target interfaces on the backbone can quickly see what is required of them.

## 6.1 Write Protocol (Updated: 06-17-2002)

The RBBM issues queued (CF), non-queued (HI and CP), real-time (RT), and pre-fetch parser (PFP) transactions that are broadcast on the same bus.

### *6.1.1 Non-Real Time Write Transactions (HI, CF, CP, PFP Paths)*

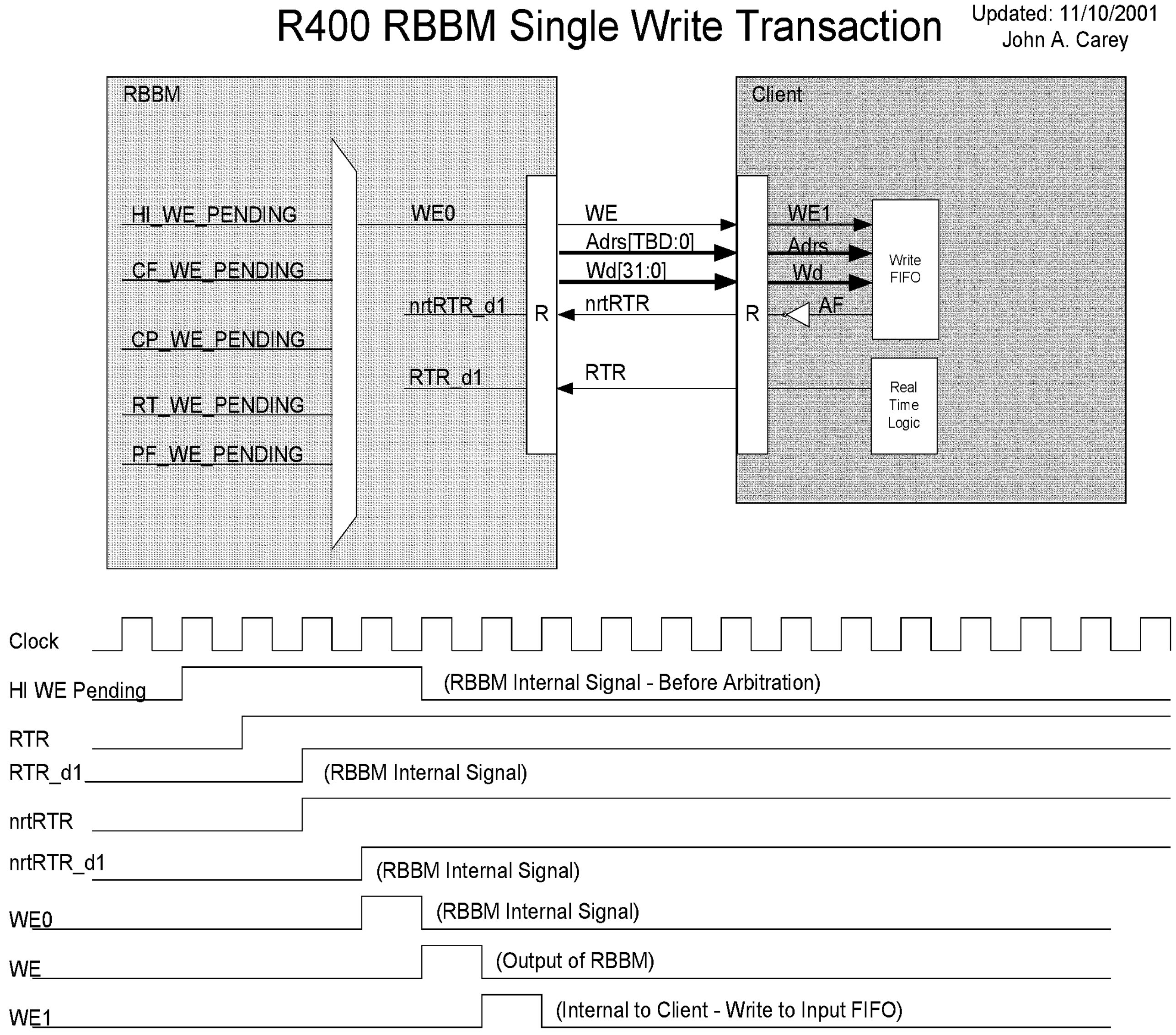
All Non Real-Time (HI, CF, CP, PFP) write transactions follow the following rules:

1. The RBBM waits until both the RTR and nrtRTR signals have been asserted from all of the clients.
2. Only at this point does the transaction win access past the internal path arbiter. The RBBM then pulses the RBBM_WE and asserts the Address, Byte Enables, and Data onto the Global Register Bus.

Note that the register bus and all control signals into and out-of the RBBM and Clients are registered. All Global Register Bus (GRB) signals may be pipelined "N" times as determined by the chip layout. The minimum for N = 2 (Sender + Receiver). The clients therefore must be able to accept 2*N additional transactions after its de-assertion of their ready-to-receive signal. The ready-to-receive can be implemented as an "almost full" condition for a FIFO where the client is placing the transactions from the RBBM.

The only reason that a client should de-assert its ready-to-receive is if its input buffer (a.k.a. "Skid Buffer") becomes almost full as a result of a write operation.

Note that it is up to the client as to whether they implement either of the nrtRTR or RTR signals. For any client that does not implement these signals, the corresponding inputs to the RBBM will be tied high at the RBBM instance.


R400 RBBM Single Write Transaction

Updated: 11/10/2001
John A. Carey

**Figure 6-1: Single Write Waveform**

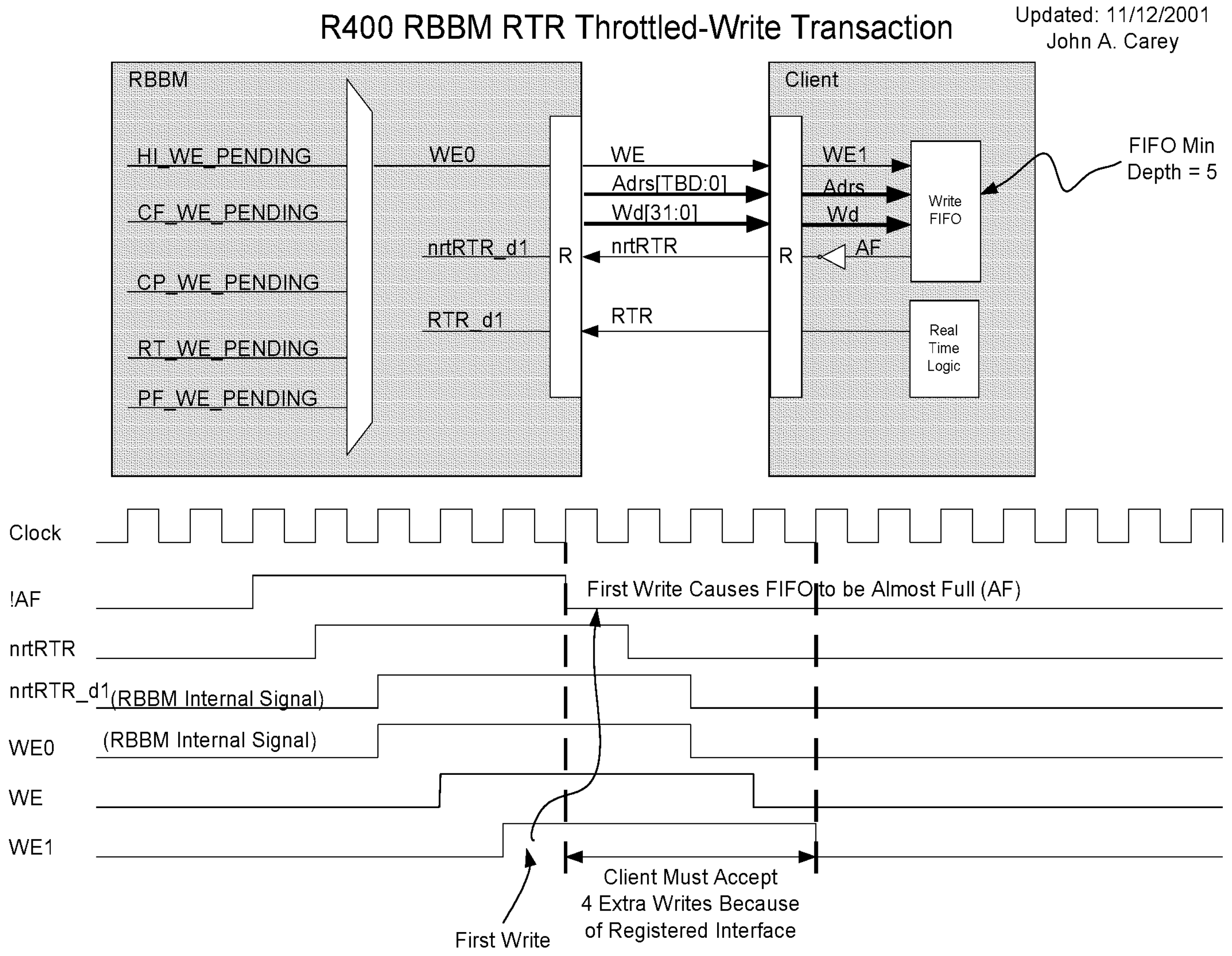


**Figure 6-2: RTR-Throttled Write Waveform**

### *6.1.2 Real-Time Write Transactions (RT Path)*

For Real-Time writes, the transaction rules are the same as the Non-Real Time transactions except that the RBBM looks at only the status of the RTR signals from the clients. The nrtRTR signals will be ignored.

## 6.2 Read Protocol (Updated: 06-17-2002)

A read request is sent out when RBBM_RE is high and the address is valid. RBBM_RE and Address will only be valid for one clock cycle. Read requests are throttled by the targeted client's ready-to-receive signals the same as write transactions. It is assumed that the clients will process these transactions through the same slip FIFOs as write transactions in order for the read operation to reflect all prior write transactions.

Only one read is outstanding at any time. If the BIF and CP both make read requests they will be serialized by the RBBM.

There is only one path from the BIF to the register bus. This is used for both reads and writes.

CP read transactions will be routed to either the queued (CF) or non-queued (CP) path depending on the NQ flag from the CP. Reads from the Real-Time stream will pass through the RT path. The Pre-Fetch Parser in the CP will not issue read operations, so no reads will traverse the PF path.

Some number of clock cycles after RBBM_RE is asserted, the RBBM will receive the return data back. The delay will be a function of the layout of the chip, which dictates the wiring of the read chains.

The read return "bus" is the bitwise OR of all the clients that can respond to a read request. All clients that are not targeted for the read operation must drive a '0' on the bus. The wiring of the read return bus is a tree of point-to-point connections. At each connection, the inputs are registered, OR'd with the client's signals, registered again, and then driven to the next connection.

The RBBM also implements a read error function. If there is not a strobe detected by the time the timer expires, the RBBM completes the read transaction and asserts the read error interrupt. It is up to the interrupt service routine to determine what to do about the read error.

As stated earlier, that the RBBM will issue write transactions when a read operation is in-progress. The RBBM will however stall on the next read transaction.

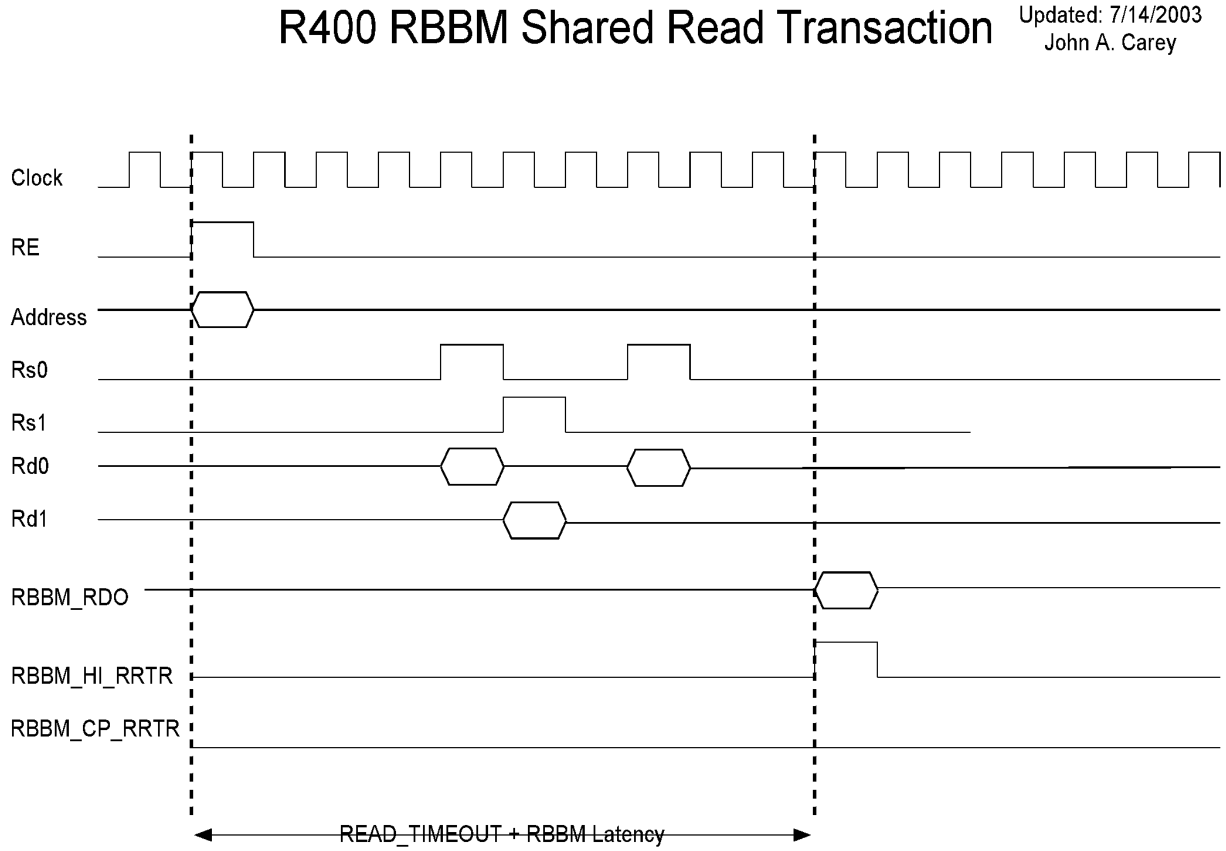


Figure 6-3: Read Transaction

## 6.3 Non-Targeted Transactions (Updated: 06-17-2002)

It is legal for none of the clients to be a target of a transaction. The RBBM does not "know" this to be the case however; it just issues the transaction when all the clients have asserted their RTR and nrtRTR signals.

If no client claims a write transaction, it will just disappear on the Global Register Bus. If no client claims a read transaction, it will most-likely end up as a "read error" as none of the clients will assert a read strobe.

## 6.4 Notes on Byte Enables (Updated: 03-26-2002)

Here are the Byte-Enable preservation rules:

1. BIF transactions preserve the byte enables.
2. CP "queued" transactions do not preserve the byte enables. They are hard-coded to "1111".
3. CP "non-queued" transactions preserve the byte enables from the CP. This is to allow the CP's DMA engine to perform byte writes (See below).
4. Real-Time Path Does not implement byte enables. They are hard-coded to "1111".
5. Pre-Fetch Parser Path does not implement byte enables. They are hard-coded to "1111".

Notes:

- a. The CP's Micro Engine (ME) does not support byte enables. It can only do 32-bit (DWORD) writes to registers and memory.
- b. The CP's DMA Engine has a control bit (NQ Flag) to indicate whether its transactions go through the CF or CP path. If the transactions go through the CP path, then the byte enables will be preserved.

# 7. External Interfaces

The Register Backbone Manager communicates with other modules of the Graphics Controller device, as summarized in Figure 7-1 below.

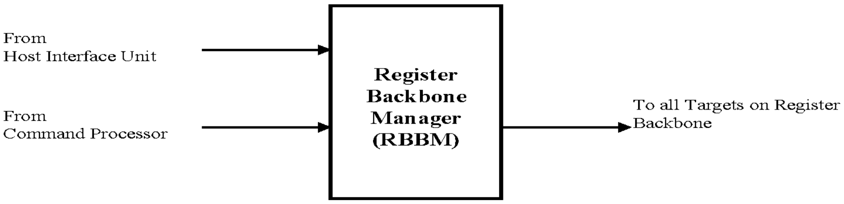


**Figure 7-1: External Interfaces of Register Backbone Manager**

## 7.1 System Interface CG-to-RBBM (Updated: 04-02-2002)

| Pin Name | Vector | Type | Description | Note |
|---|---|---|---|---|
| sclk | | I | Permanent Core Clock. | |
| srst | | I | Reset, synchronized to SCLK. | |

## 7.2 Bus Interface (BIF) -to- RBBM Interface (Updated: 03-24-2003)

This interface is used when the BIF wishes to access registers that reside in target modules connected to the register backbone.

All the inputs are registered and a "slip FIFO" is implemented on the interface. The RBBM will de-assert the RBBM_BIF_rdy when it is "almost full" in order to be able to safely consume transactions already in-flight. The BIF will only send a transaction (Write/Read) to the RBBM if its registered version of the RBBM_BIF_RDY signal is asserted.

The "slip FIFO" and its "almost full" signal is set so that up to 2 repeater flops can be inserted between the BIF and the RBBM.

| Pin Name | Vector | Type | Description | Note |
|---|---|---|---|---|
| BIF_RBBM_a | 16:2 | I | Register Address for Read/Write transaction. | |
| BIF_RBBM_op | | I | Opcode. Specifies a read or write transaction.<br>(0=Read, 1=Write). | |
| BIF_RBBM_wd | 31:0 | I | Write Data Bus. | |
| BIF_RBBM_be | 3:0 | I | Byte Enables.<br>One bit for each byte of the Wd bus.<br>Bit 0 corresponds to byte on Wd[7:0]<br>Bit 1 corresponds to byte on Wd[15:7]<br>Bit 2 corresponds to byte on Wd[23:16]<br>Bit 3 corresponds to byte on Wd[31:24]<br>(0=Disable Read/Write, 1=Enable Read/Write | |
| BIF_RBBM_swp | | I | Indicates whether endian swap should be performed on HI reads/writes.<br>Excludes pushed command data to CP. | |
| BIF_RBBM_send | | I | Send: Single Pulse for Either Write or Read Transactions. | |
| RBBM_BIF_rdy | | O | RBBM Ready to Receive a Write or Read Transaction from BIF. | |
| RBBM_BIF_int | | O | Interrupt to the BIF. | |

## 7.3 CP-to- RBBM Interface (Updated: 03-19-2002)

This interface is used when the CP wishes to access registers that reside in target modules connected to the register backbone. The CP writes state data, constant data, shader code DMA initiators, and other register writes through this interface.

The "slip FIFO" and its "almost full" signal on the CP input interface is set so that up to 2 repeater flops can be inserted between the CP and the RBBM.

Note 1: The RBBM does not do endian swapping on the data from the CP.

| Pin Name | Vector | Type | Description | Note |
|---|---|---|---|---|
| CP_RBBM_send | | I | CP Send Transaction to RBBM.<br>1. Transaction from Micro Engine<br>2. Transaction from DMA Engine | |
| CP_RBBM_rt_send | | I | CP Send for Real-Time Transaction | |
| CP_RBBM_pf_send | | I | CP Send Pre-Parser Transaction | |
| CP_RBBM_a | 16:2 | I | Register Address for Read/Write transaction. | |
| CP_RBBM_op | | I | Opcode. Specifies a read or write transaction.<br>(0=Read, 1=Write). | |
| CP_RBBM_nq | | I | Use Non-Queued Path<br>0 = Transaction processed through "queued" path<br>1 = Transaction processed through "non-queued" path. | |
| CP_RBBM_wd | 31:0 | I | Write Data Bus. | |
| CP_RBBM_be | 3:0 | I | Byte Mask (For 32-bit Transfers) | |
| RBBM_CP_rdy | | O | Non-Real Time Ready to Receive. The CP uses a delayed version of this signal so the RBBM asserts this as an "almost full" with respect to its receiving FIFO. | |
| RBBM_CP_rt_rdy | | O | Real-Time Stream Ready to Receive. The CP uses a delayed version of this signal so the RBBM asserts this as an "almost full" with respect to its receiving FIFO. | |
| RBBM_CP_pf_rdy | | O | Pre-Parser Transaction Ready to Receive. The CP uses a delayed version of this signal so the RBBM asserts this as an "almost full" with respect to its receiving FIFO. | |

## 7.4 RBBM –to- CP/BIF Read Return Data (Updated: 12-21-2001)

This is the read return data that is wired from the RBBM to the CP and BIF. The read data is qualified by a pulse on the "valid" signal. The RBBM will assert an interrupt if a read error occurs. The "valid" signal will still be asserted to appropriate read-requester however for read errors.

| Pin Name | Vector | Type | Description | Note |
|---|---|---|---|---|
| RBBM_rd | 31:0 | O | Read Return Data to CP and HI. Note: Read return data for the BIF can be swapped in the RBBM. | |
| RBBM_BIF_valid | | O | Read Data Valid to the BIF. | |
| RBBM_CP_valid | | O | Read Data Valid to the CP. | |

## 7.5 RBBM-to-Target(s) Interface (Updated: 04-24-2003)

This interface is used to perform read and write transactions to targets containing registers that reside in the register address space of the Graphics Controller device. The RBBM can access up to 128K Bytes of register space.

The RBBM is itself a client on the register bus, but transactions to its registers are snooped internal to the RBBM. The interface signals for the RBBM client therefore do not appear in this list.

The ready-to-receive signals (*_RTR and *_nrtRTR) are registered before being used by the RBBM. The clients must either have storage to absorb extra transfers because of this delay or throttle the RTR signal accordingly.

Note 1: The DC_RBBM_RTR and DC_RBBM_nrtRTR signals are combined RTR signals for the VGA, TVOUT, VIP, and Display.

| Pin Name | Vector | Type | Description | Note |
|---|---|---|---|---|
| RBBM_we | | O | Write Enable (Send) (Address and Data are Valid). | |
| RBBM_a | 16:2 | O | Register Address for Read/Write Transaction. | |
| RBBM_wd | 31:0 | O | 32-bit Write Data Bus (To 32-bit Clients) | |
| RBBM_be | 3:0 | O | Register Byte Mask<br>Bit 0 corresponds to byte on Wd[7:0]<br>Bit 1 corresponds to byte on Wd[15:7]<br>Bit 2 corresponds to byte on Wd[23:16]<br>Bit 3 corresponds to byte on Wd[31:24]<br>(0=Disable Read/Write, 1=Enable Read/Write). | Optional for Clients |
| | | | | |
| RBBM_re | | O | Read Enable (Address is Valid, Data is "Don't Care") | |
| RBB_rs0 | | I | Register BackBone Read Return Strobe 0 | |
| RBB_rs1 | | I | Register BackBone Read Return Strobe 1 | |
| RBB_rd0 | 31:0 | I | Register BackBone Read Return Data 0 | |
| RBB_rd1 | 31:0 | I | Register BackBone Read Return Data 1 | |
| | | | | |
| CP_RBBM_rtr | | I | Command Processor Real-Time Ready to Receive | CP Does Not Implement |
| BIF_RBBM_rtr | | I | Bus Interface Unit (BIF) Real-Time Ready to Receive | |
| AIC_RBBM_rtr | | I | AIC Ready to Real-Time Ready to Receive | |
| PA_RBBM_rtr | | I | Primitive Assembly Real-Time Ready to Receive | |
| VGT_RBBM_rtr | | I | VGT Real-Time Ready to Receive | VGT Does Not Implement |
| MC0_RBBM_rtr | | I | Memory Controller #0 Real-Time Ready to Receive | |
| MC1_RBBM_rtr | | I | Memory Controller #1 Real-Time Ready to Receive | |
| MC2_RBBM_rtr | | I | Memory Controller #2 Real-Time Ready to Receive | |
| MC3_RBBM_rtr | | I | Memory Controller #3 Real-Time Ready to Receive | |
| MH_RBBM_rtr | | I | Memory Hub Real-Time Ready to Receive | |
| DC_RBBM_rtr | | I | VGA, TV, VIP, Display Real-Time Ready to Receive | |
| CG_RBBM_rtr | | I | Clock Generator Real-Time Ready to Receive | |
| CGM_RBBM_rtr | | I | Memory Clock Generator Real-Time Ready to Receive | |
| IDCT_RBBM_rtr | | I | IDCT Engine Real-Time Ready to Receive | |
| SQ_RBBM_rtr | | I | Sequencer Real-Time Ready to Receive | |
| SX0_RBBM_rtr | | I | Shader Export #0 Real-Time Ready to Receive | |
| SX1_RBBM_rtr | | I | Shader Export #1 Real Time Ready to Receive | |
| RC_RBBM_rtr | | I | Renderer Central Real-Time Ready to Receive | |
| RB0_RBBM_rtr | | I | Renderer Backend #0 Real-Time Ready to Receive | |
| RB1_RBBM_rtr | | I | Renderer Backend #1 Real-Time Ready to Receive | |
| RB2_RBBM_rtr | | I | Renderer Backend #2 Real-Time Ready to Receive | |
| RB3_RBBM_rtr | | I | Renderer Backend #3 Real-Time Ready to Receive | |
| ROM_RBBM_rtr | | I | ROM Real-Time Ready to Receive | |
| DEBUG_RBBM_rtr | | I | Debug Bus Controller Real-Time Ready to Receive | |
| | | | | |
| CP_RBBM_nrtrtr | | I | Command Processor Ready to Receive | |
| BIF_RBBM_nrtrtr | | I | Bus Interface Unit Ready to Receive | |
| AIC_RBBM_nrtrtr | | I | AIC Ready to Receive | |

| Pin Name | Vector | Type | Description | Note |
|---|---|---|---|---|
| PA_RBBM_nrtrtr | | I | Primitive Assembly Ready to Receive | |
| VGT_RBBM_nrtrtr | | I | VGT Ready to Receive | |
| MC0_RBBM_nrtrtr | | I | Memory Controller #0 Ready to Receive | |
| MC1_RBBM_nrtrtr | | I | Memory Controller #1 Ready to Receive | |
| MC2_RBBM_nrtrtr | | I | Memory Controller #2 Ready to Receive | |
| MC3_RBBM_nrtrtr | | I | Memory Controller #3 Ready to Receive | |
| MH_RBBM_nrtrtr | | I | Memory Hub Ready to Receive | |
| DC_RBBM_nrtrtr | | I | VGA, TVOUT, VIP, Display Ready to Receive | |
| CG_RBBM_nrtrtr | | I | Clock Generator Ready to Receive | |
| CGM_RBBM_nrtrtr | | I | Memory Clock Generator Ready to Receive | |
| IDCT_RBBM_nrtrtr | | I | IDCT Engine Ready to Receive | |
| SQ_RBBM_nrtrtr | | I | Sequencer Ready to Receive | |
| SX0_RBBM_nrtrtr | | I | Shader Export #0 Ready to Receive | |
| SX1_RBBM_nrtrtr | | I | Shader Export #1 Ready-to-Receive | |
| RC_RBBM_nrtrtr | | I | Renderer Central Ready to Receive | |
| RB0_RBBM_nrtrtr | | I | Renderer Backend #0 Ready to Receive | |
| RB1_RBBM_nrtrtr | | I | Renderer Backend #1 Ready to Receive | |
| RB2_RBBM_nrtrtr | | I | Renderer Backend #2 Ready to Receive | |
| RB3_RBBM_nrtrtr | | I | Renderer Backend #3 Ready to Receive | |
| ROM_RBBM_nrtrtr | | I | ROM Ready to Receive | |
| DEBUG_RBBM_nrtrtr | | I | Debug Bus Controller Ready to Receive | |

## 7.6 RBBM-to-Target Soft Resets (Updated: 04-24-2003)

The soft reset signals are from a register that in the RBBM that is written by a normal register write. The reset signals will remain set until the corresponding bit in the SOFT_RESET register is cleared.

| Pin Name | Vector | Type | Description | Note |
|---|---|---|---|---|
| RBBM_CP_soft_reset | | O | Soft Reset to Command Processor (CP). | |
| RBBM_BIF_soft_reset | | O | Soft Reset to Bus Interface Unit (BIF). | |
| RBBM_PA_soft_reset | | O | Soft Reset to Primitive Assembly (PA, VGT, and SC) | |
| RBBM_MH_soft_reset | | O | Soft Reset to Memory Hub (MH) | |
| RBBM_SQ_soft_reset | | O | Soft Reset to Sequencer (SQ) (None for SP) | |
| RBBM_SX_soft_reset | | O | Soft Reset to Shader Export (SX) | |
| RBBM_RB_soft_reset | | O | Soft Reset to Render Back End (RB). | |
| RBBM_RC_soft_reset | | O | Soft Reset to Render Central (RC) | |
| RBBM_MC_soft_reset | | O | Soft Reset to Memory Controller (MC). | |
| RBBM_VIP_soft_reset | | O | Soft Reset to Video Input Port (VIP). | |
| RBBM_DISP_soft_reset | | O | Soft Reset to Display Engine (DISP). | |
| RBBM_CG_soft_reset | | O | Soft Reset to Clock Generator (CG, CGM). | |
| RBBM_IDCT_soft_reset | | O | Soft Reset to the IDCT (IDCT ☺) | |
| RBBM_VGA_soft_reset | | O | Soft Reset to the VGA | |
| RBBM_SC_soft_reset | | O | Soft Reset to the Scan Converter (SC) | |
| RBBM_VGT_soft_reset | | O | Soft Reset to the Vertex Grouper Tessellator (VGT) | |
| RBBM_ROM_soft_reset | | O | Soft Reset to the ROM Controller | |
| RBBM_DB_soft_reset | | O | Soft Reset to the DB Block | |

## 7.7 Status/Interrupt Interface (Updated: 04-24-2003)

Status and Interrupt signals that are used to for wait_until, general status, and interrupt logic in the RBBM.

Notes:

1. 04-09-02: No "busy" signals will be provided from MC for either debug or synchronization.
2. 04-09-02: No "busy" signal will be provided from the Clock Generator.
3. 04-10-02: The MH_Clean signal is not used for the "wait_idleclean" determination because the Renderer Backend writes directly to the Memory Controller.

| Pin Name | Vector | Type | Description | Note |
|---|---|---|---|---|
| IDCT_semaphore | | I | Semaphore Status from IDCT.<br>Used for "wait IDCT semaphore" | Confirmed for R400 by Daniel Wong. |
| RC_RBBM_cntx0_clean | | I | Renderer Common is clean for context #0.<br>Used for "wait idle/clean" | Updated 04-09-02 |
| RC_RBBM_cntx17_clean | | I | Renderer Common is clean for contexts #1 to #7<br>Used for "wait idle/clean" | Updated 04-09-02 |
| RC_RBBM_cntx0_busy | | I | Renderer Common is busy for context #0.<br>Used for "wait idle and wait idle/clean" | Updated 04-09-02 |
| RC_RBBM_cntx17_busy | | I | Renderer Common is busy for contexts #1 to #7<br>Used for "wait idle and wait idle/clean" | Updated 04-09-02 |
| MH_clean | | I | Memory Hub is Clean.<br>Excludes Display Processing.<br>Used for debug status. | Updated 04-09-02 |
| MH_hdp_clean | | I | Status from the HDP indicating its write path is clean and that the HI is clean.<br>Does not reflect the status of the read path of the HDP.<br>Used for "wait host idle/clean" | Renamed: 03-26-02<br>The MH combines the HI signal into this signal. |
| SQ_RBBM_cntx0_busy | | I | Sequencer is busy for context #0.<br>Used for "wait idle and wait idle/clean". | Updated 04-09-02 |
| SQ_RBBM_cntx17_busy | | I | Sequencer is busy for context #1 to #7.<br>Used for "wait idle and wait idle/clean". | Updated 04-09-02 |
| VGT_RBBM_busy | | I | VGT is busy.<br>Used for debug status. | Non-Real-Time Only |
| VGT_RBBM_no_dma_busy | | I | Busy signal from the VGT that does not contain the index DMA status. This is used for the wait_until and nqwait_until conditions. | |
| PA_RBBM_busy | | I | Primitive Assembly is busy.<br>Used for "wait idle and wait idle/clean". | Non-Real-Time Only |
| SC_RBBM_cntx0_busy | | I | Scan Converter is busy for context #0. | Updated 04-10-02 |
| SC_RBBM_cntx17_busy | | I | Scan Converter is busy for context #1 to #7. | Updated 04-10-02 |
| TPC_RBBM_busy | | I | Texture Pipeline Common is busy.<br>Used for debug status. | Updated 06-21-02 |
| TC_RBBM_busy | | I | Texture Cache is Busy<br>Used for debug status. | Updated: 06-21-02 |
| u0_SX_RBBM_busy | | I | Shader Export #0 busy.<br>Used for debug status. | Updated 04-10-02 |
| u1_SX_RBBM_busy | | I | Shader Export #1 busy.<br>Used for debug status. | Updated 04-10-02 |
| MH_RBBM_coherency_busy | | I | Surface Coherency Logic is busy.<br>Used for debug status. | Need Correct Name. |
| PAD_extern_signal | | I | Status Signal from Pads of Chip. | See Extern_Trig_Cntl Register |
| VIP_RBBM_h0dma_idle | | I | VIP's Host DMA Channel 0 is Idle. | Renamed: 03-26-2002 |
| VIP_RBBM_h1dma_idle | | I | VIP's Host DMA Channel 1 is Idle. | Renamed: 03-26-2002 |
| VIP_RBBM_h2dma_idle | | I | VIP's Host DMA Channel 2 is Idle. | Renamed: 03-26-2002 |
| VIP_RBBM_h3dma_idle | | I | VIP's Host DMA Channel 3 is Idle. | Renamed: 03-26-2002 |
| CP_RBBM_nrt_busy | | I | Non-RT portion of CP is busy | |
| CP_RBBM_rt_busy | | I | Real-Time portion of CP is busy | |
| CP_RBBM_dma_busy | | I | Status of CP's DMA Engine.<br>Used for "wait CP_DMA_IDLE" | |
| CP_RBBM_rt_enable | | I | Real-Time is Enabled.<br>RBBM uses to qualify the "context #0" busy/clean signals for real-time. | |
| D1OVL_update_pending | | I | Status from 1st Overlay Controller (Flip Pending). | Used for wait_until. |

| Pin Name | Vector | Type | Description | Note |
|---|---|---|---|---|
| D2OVL_update_pending | | I | Status from 2nd Overlay Controller (Flip Pending). | Used for wait_until. |
| D1MODE_vline | | I | Status from 1st Display Controller. | Used for wait_until. |
| D2MODE_vline | | I | Status from 2nd Display Controller. | Used for wait_until. |
| D1GRPH_update_pending | | I | "Pending Flip" status from 1st Display Controller. | Used for wait_until. |
| D2GRPH_update_pending | | I | "Pending Flip" status from 2nd Display Controller. | Used for wait_until. |
| BIF_RBBM_agp_flush | | I | AGP Flush Complete | Renamed: 03-26-02 |
| Interrupts | | | | |
| CP_RBBM_int | | I | Interrupt from CP | New for R400 |
| PA_RBBM_int | | I | Interrupt from the PA | Not in R400 |
| VGT_RBBM_int | | I | Interrupt from the VGT | Not in R400 |
| MH_RBBM_int | | I | Interrupt from HDP (MH) | New for R400 |
| MC0_RBBM_int | | I | Interrupt 0 from the MC | |
| MC1_RBBM_int | | I | Interrupt 1 from the MC | |
| DISP_RBBM_int | | I | Combined Interrupt from Display | Display combines all its interrupts into one (CRTC & CRTC2) D. Glen Confirmed Need. |
| VIP_RBBM_int | | I | Combined Interrupt from VIP | VIP combines all its interrupts into one (CAP0, HDMA, I2C, HOST, etc.). D. Glen Confirmed Need. |
| VGA_RBBM_int | | I | Interrupt from the VGA | Needed per D. Glen |
| CG_RBBM_int | | I | Interrupt from the CG | |
| IDCT_RBBM_int | | I | Interrupt from IDCT | Renamed from INT_IDCT_TIMESTAMP |
| SQ_RBBM_int | | I | Interrupt from the SQ | |
| SX_RBBM_int | | I | Interrupt from the SX | |
| RC_RBBM_int | | I | Interrupt from the Renderer Common | |
| RB_RBBM_int | | I | Interrupt from the Renderer Backend | |
| DEBUG_RBBM_int | | I | Interrupt from the Debug Unit | |
| Status Outputs | | | | |
| RBBM_BIF_hi_busy | | O | The RBBM is processing a transaction from the BIF. This signal will be de-asserted when all transactions from the BIF have been processed by the RBBM and the RBBM has received a new RTR from the clients. It will also be asserted for 32 additional clocks by the RBBM to account for the transaction to get through the client's slip buffers. | HDP uses this signal to stall writes to surfaces until prior surface parameters have been written by the RBBM. |
| RBBM_CP_nrt_idle | | O | Non-Real-Time graphics pipe is idle. | Used by the CP to clear the context valid flags for surface coherency. |
| RBBM_CP_rt_idle | | O | Real-Time graphics pipe is idle. | Used by the CP to clear the context #0 flag if real-time is enabled. |
| RBBM_extern_signal | | O | Conditioned External Signal from the Pads. | Wired to CP for Initiating Real-Time Streams. |

## 7.8 RBBM-to-Clock Generator (CG) Interface (Updated: 06-05-2002)

This interface is used to turn on clocks to clients that are being accessed. Whenever the RBBM has a transaction from either the BIF or CP, it asserts the register clock active signal until the end of the transaction (EOT). The clients do the clock gating to enable the clock to their register interface logic. The signal is also asserted during reset.

The end of transaction (EOT) is defined as follows:

Write Transaction – EOT is when the RBBM pulses the RBBM_WE write strobe.

Read Transaction – EOT is when the RBBM receives a read strobe and has sent the data to the BIF or CP.

After EOT or reset the RBBM keeps the signal asserted for the duration of REGCLK_DEASSERT_TIME, which is programmed in the RBBM_CNTL register. The REGCLK_DEASSERT_TIME has a hardware default value of 0xF.

Signal Table for Block RBBM:

| Pin Name | Vector | Type | Description | Note |
|---|---|---|---|---|
| RBBM_regclk_active | | O | Turn on clock all client's register interface. | |

## 7.9 Debug Bus Interface (Updated: 02-13-2002)

This following interface is to connect to the debug bus.

| Signal Name | Vector | Type | Description | Note |
|---|---|---|---|---|
| DEBUG_bus_in | 11:0 | I | Debug Bus Input | |
| DEBUG_bus_out | 11:0 | O | Debug Bus Output | |
| DEBUG_block_sel | 5:0 | I | Unit Select | RBBM = 0x02 |
| DEBUG_group_sel | 5:0 | I | Selects Local RBBM Signals to Output | See RBBM' Test Bus Specification. |

# 8. Architectural Overview

## 8.1 Backbone Bus Topology

Figure 8-1 is a picture of how the RBBM fits into the top-level chip from a hardware-wiring point of view.

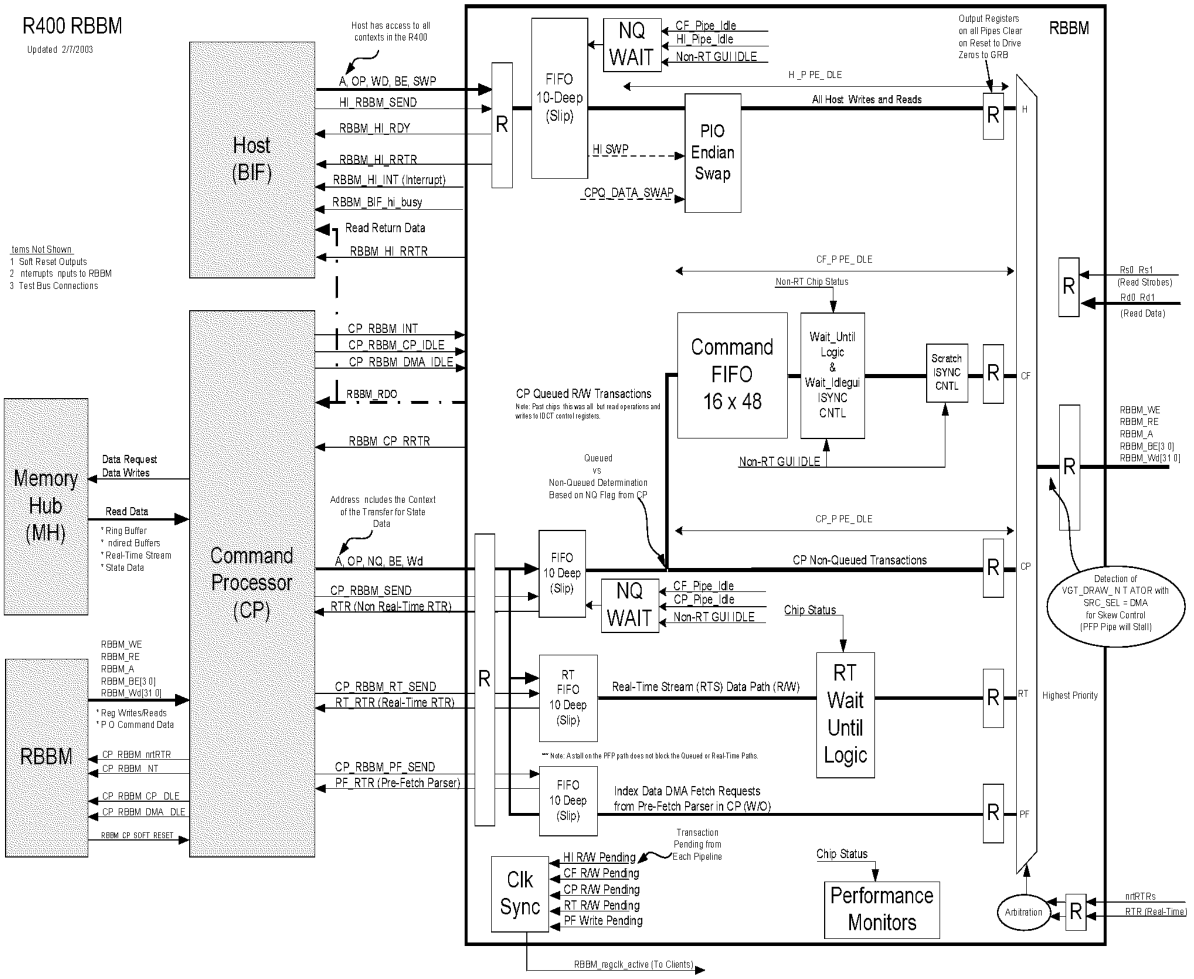


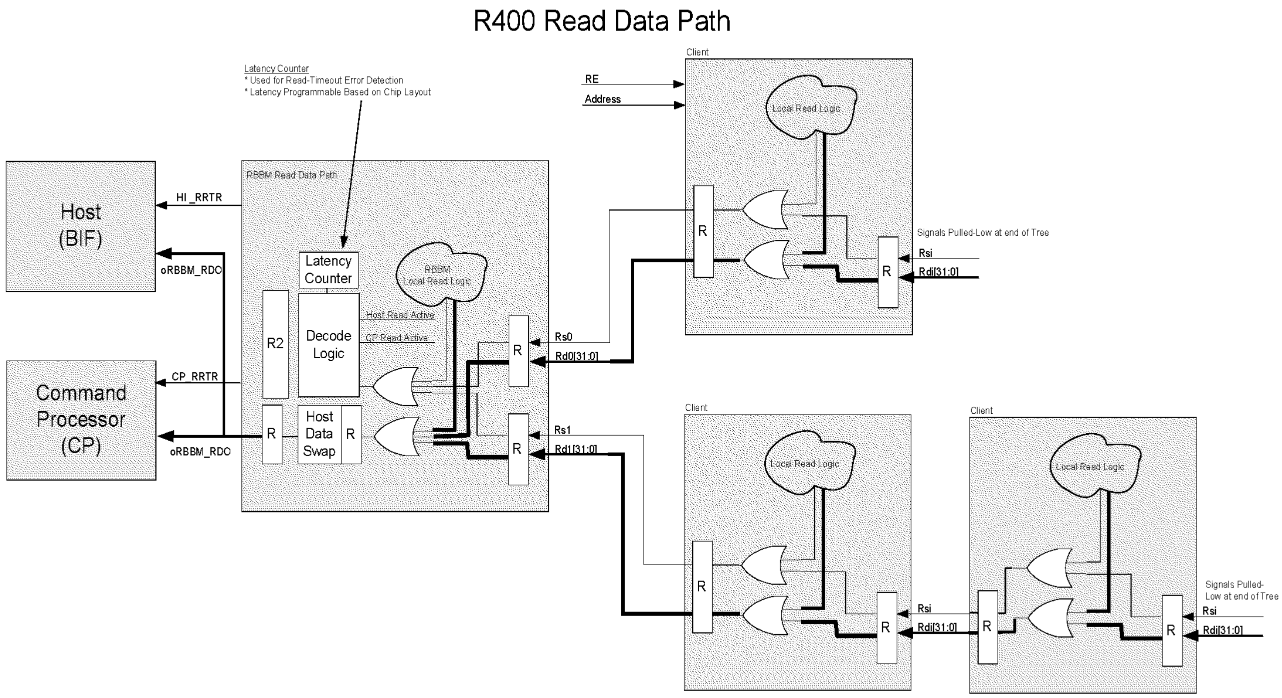
**Figure 8-1: RBBM in the Chip-Level Environment**

The backbone bus has a command that is broadcast to all targets. The command consists of an Address, Byte Enables (BE), and RBBM_WE or RBBM_RE. The RBBM can access up to 128K Bytes of register space.

On writes, when the command is presented on the backbone, so is the Write Data (Wd). The protocol on the backbone allows for single-cycle writes to any target.

Like writes, the read command is also sent in one clock, but the return data may take several clocks to return. Read data is returned from targets via 32-bit buses that are registered within the clients and finally wired to the RBBM. The RBBM does data swapping on the BIF data if needed, registers the data, and outputs the read data to the CP and BIF.

It is the target's responsibility to pass the daisy-chain data from its input to its output when it is not responding to a read transaction on the backbone. The read return path is shown in Figure 8-2.



Updated: 4/26/2002
John Carey

Figure 8-2: Read Data Chain

## 8.2 Address Re-Mapping, Auto-Reg Usage, Address Decoding (Updated: 06-17-2002)

Unlike prior chips, the RBBM does not re-map addresses to a client's "local-linear" space. Because of this, if a register appears in both the IO and Memory-Mapped spaces, then its address in both of these spaces must be identical.

In past designs, multiple addresses could be re-mapped to the same register. For R400, the clients must take care of this by decoding multiple addresses for a particular register.

In past designs, multiple clients could have the same register and the RBBM would broadcast the transaction by asserting multiple unary ready-to-send signals to the clients. In R400, there is a common transaction and the clients themselves decide as to whether to accept the transaction (Read or Write).

# 9. Register Descriptions

The Register Backbone Manager itself has a target interface from the Register Backbone of the chip. Through this interface, any master on the register backbone can program control registers inside the RBBM.

## 9.1 RBBM Control Register (03-07-2002)

The RBBM control register controls the read timeout, pushed command data swapping, and arbitration hysteresis.

The RBBM control register is mapped to the same address in the IO and MMR decode spaces..

**RBBM_CNTL**
**Miscellaneous RBBM Control Register**

| Field Name | Bit(s) | Description |
|---|---|---|
| Reserved | 31:21 | Reserved |
| CPQ_DATA_SWAP | 20 | Endian Swap Control for writes to the Command Stream Queue.<br>0 = No swap.<br>1 = 32-bit swap: 0xAABBCCDD becomes 0xDDCCBBAA.<br>Default = 0. |
| Reserved | 19:17 | Reserved |
| REGCLK_DEASSERT_TIME | 16:8 | Number of clocks the RBBM will wait before de-asserting the "Register Clock Active" signal.<br>Default = 0x0F |
| READ_TIMEOUT | 7:0 | Programmable delay after the start of a read operation that a timeout will occur.<br>This is used for the detection of an error during a read operation.<br>The timeout counter is set to this value when a read operation starts and counts down every 16 clocks. The maximum delay therefore is 4080 clocks (10.20 usec).<br>Minimum time programmed into this register should equal to the depth of the read-return data path. (Default = 0x0F) |

**RBBM_SKEW_CNTL**
**Skew Control Register**

| Field Name | Bit(s) | Description |
|---|---|---|
| RESERVED | 31:10 | Reserved |
| SKEW_COUNT | 9:5 | Current Value of the Skew Counter. Read-Only. |
| SKEW_TOP_THRESHOLD | 4:0 | Upper Skew Counter Threshold. Default = 0x04.<br>Must be a non-zero even number. |

## 9.2 Non-Real-Time WAIT_UNTIL Control Register (Updated: 10-24-2002)

Non-Real Time Busy = ( not CP_RBBM_rt_enable and ( RC_RBBM_cntx0_busy or SQ_RBBM_cntx0_busy or SC_RBBM_cntx0_busy) ) or
( PA_RBBM_busy or VGT_RBBM_no_dma_busy or SQ_RBBM_cntx17_busy or RC_RBBM_cntx17_busy or
SC_RBBM_cntx17_busy );

Non-Real Time Clean = (( not CP_RBBM_rt_enable and RC_RBBM_cntx0_clean ) or CP_RBBM_rt_enable ) and RC_RBBM_cntx17_clean;

The Non-Queued flag must be set to ‘0’ in the CP when writing to this register.

**WAIT_UNTIL (WO) – Only CP Accesses Register**
**Explicit Synchronization Register**

| Field Name | Bit(s) | Description |
|---|---|---|
| ENG_DISPLAY_SELECT | 31 | Determines if WAIT_UNTIL condition is for Display/Overlay Engine 1 or 2.<br>(Applicable only to the WAIT_OV_FLIP and wait conditions for bits 3:0.) |
| WAIT_BOTH_CRTC_FLIP | 30 | Wait for ‘Pending Flip’ signal to be OFF from both display engines. |
| Unused | 29:28 | Unused |
| WAIT_IDCT_SEMAPHORE | 27 | Wait for IDCT Semaphore |
| Reserved | 26:24 | Reserved |
| CMDFIFO_ENTRIES | 23:20 | Number of entries (1 to 15) to wait for if the WAIT_CMDFIFO bit is ON.<br>The FIFO is 16-deep, but one of the entries is taken by the Wait_Until command. |
| WAIT_EXTERN_SIGNAL | 19 | Wait for Conditioned External Signal to be Set (See Extern_Trig_Cntl Register). |
| WAIT_HOST_IDLECLEAN | 18 | Wait for Host Interface/Host Data Path to be idle and clean. This is the MH_hdp_clean signal being asserted. |
| WAIT_3D_IDLECLEAN | 17 | Wait for processing to be done and the caches in the RB to be flushed. |
| WAIT_2D_IDLECLEAN | 16 | Same as WAIT_3D_IDLECLEAN in R400. |
| WAIT_3D_IDLE | 15 | Wait for processing to be done. |
| WAIT_2D_IDLE | 14 | Same as WAIT_3D_IDLE in R400. |
| WAIT_AGP_FLUSH | 13 | Wait for AGP Flush to complete (When BIF_RBBM_agp_flush signal is ‘0’.) |
| Reserved | 12 | Reserved |
| WAIT_OV_FLIP | 11 | Wait for selected overlay flip signal to be de-asserted.<br>Eng_Display_Select controls which overlay flip signal to observe.<br>D1OVL_update_pending<br>D2OVL_update_pending |
| WAIT_CMDFIFO | 10 | Wait until there are at least “CMDFIFO_ENTRIES” values in the Command FIFO. |
| Reserved | 9 | Reserved |
| WAIT_CP_DMA_IDLE | 8 | Wait for CP DMA Channel to be idle. |
| WAIT_DMA_VIPH3_IDLE | 7 | Wait for VIP Host DMA Channel 3 to be idle |
| WAIT_DMA_VIPH2_IDLE | 6 | Wait for VIP Host DMA Channel 2 to be idle |
| WAIT_DMA_VIPH1_IDLE | 5 | Wait for VIP Host DMA Channel 1 to be idle |
| WAIT_DMA_VIPH0_IDLE | 4 | Wait for VIP Host DMA Channel 0 to be idle |
| WAIT_CRTC_VLINE | 3 | Wait for selected VLINE signal to be asserted.<br>Eng_Display_Select controls which overlay flip signal to observe.<br>D1MODE_vline<br>D2MODE_vline |
| WAIT_FE_CRTC_VLINE | 2 | Wait for Falling Edge of selected VLINE signal.<br>Eng_Display_Select controls which overlay flip signal to observe.<br>D1MODE_vline<br>D2MODE_vline |
| WAIT_RE_CRTC_VLINE | 1 | Wait for Rising Edge of selected VLINE signal.<br>Eng_Display_Select controls which overlay flip signal to observe.<br>D1MODE_vline<br>D2MODE_vline |
| WAIT_CRTC_PFLIP | 0 | Wait for the ‘Pending Flip’ signal to be de-asserted.<br>Eng_Display_Select controls which overlay flip signal to observe.<br>D1GRPH_update_pending<br>D2GRPH_update_pending |

## 9.3 Real-Time WAIT_UNTIL Control Register (Updated: 10-24-2002)

The RTS_WAIT_UNTIL register controls the wait logic for the real-time stream.

Real Time Busy = CP_RBBM_rt_enable and (RC_RBBM_cntx0_busy or SQ_RBBM_cntx0_busy or SC_RBBM_cntx0_busy);

Real Time Clean = ( CP_RBBM_rt_enable and RC_RBBM_cntx0_clean ) or ( not CP_RBBM_rt_enable);

Notes:

1. IDCT Semaphore is not included. The Wait_Reg_Mem function will be used as an alternative for Real-Time streams.

**RTS_WAIT_UNTIL (WO) – Only CP Accesses Register**
**Real-Time Stream Explicit Synchronization Register**

| Field Name | Bit(s) | Description |
|---|---|---|
| ENG_DISPLAY_SELECT | 31 | Determines if WAIT_UNTIL condition is for Display/Overlay Engine 1 or 2. (Applicable only to the WAIT_OV_FLIP and wait conditions for bits 3:0.) |
| WAIT_BOTH_CRTC_FLIP | 30 | Wait for 'Pending Flip' signal to be OFF from both display engines |
| Unused | 29:20 | Unused |
| WAIT_EXTERN_SIGNAL | 19 | Wait for Conditioned External Signal to be Set (See Extern_Trig_Cntl Register). |
| WAIT_HOST_IDLECLEAN | 18 | Wait for Host Interface/Host Data Path to be idle and clean |
| Reserved | 17 | Reserved |
| WAIT_RT_IDLECLEAN | 16 | Wait for Real-Time to be Idle and Clean. |
| Reserved | 15 | Reserved |
| WAIT_RT_IDLE | 14 | Wait for Real-Time to be Idle. |
| WAIT_AGP_FLUSH | 13 | Wait for AGP Flush to complete (When BIF_RBBM_agp_flush signal is '0'.) |
| Reserved | 12 | Reserved |
| WAIT_OV_FLIP | 11 | Wait for selected overlay flip signal to be de-asserted.<br>Eng_Display_Select controls which overlay flip signal to observe.<br>D1OVL_update_pending<br>D2OVL_update_pending |
| Unused | 10:9 | Unused |
| WAIT_CP_DMA_IDLE | 8 | Wait for CP DMA Channel to be idle. |
| WAIT_DMA_VIPH3_IDLE | 7 | Wait for VIP Host DMA Channel 3 to be idle. |
| WAIT_DMA_VIPH2_IDLE | 6 | Wait for VIP Host DMA Channel 2 to be idle. |
| WAIT_DMA_VIPH1_IDLE | 5 | Wait for VIP Host DMA Channel 1 to be idle. |
| WAIT_DMA_VIPH0_IDLE | 4 | Wait for VIP Host DMA Channel 0 to be idle. |
| WAIT_CRTC_VLINE | 3 | Wait for selected VLINE signal to be asserted.<br>Eng_Display_Select controls which overlay flip signal to observe.<br>D1MODE_vline<br>D2MODE_vline |
| WAIT_FE_CRTC_VLINE | 2 | Wait for Falling Edge of selected VLINE signal.<br>Eng_Display_Select controls which overlay flip signal to observe.<br>D1MODE_vline<br>D2MODE_vline |
| WAIT_RE_CRTC_VLINE | 1 | Wait for Rising Edge of selected VLINE signal.<br>Eng_Display_Select controls which overlay flip signal to observe.<br>D1MODE_vline<br>D2MODE_vline |
| WAIT_CRTC_PFLIP | 0 | Wait for the 'Pending Flip' signal to be de-asserted.<br>Eng_Display_Select controls which overlay flip signal to observe.<br>D1GRPH_update_pending<br>D2GRPH_update_pending |

## 9.4 Non-Queued Wait Register (Updated: 03-15-2002)

This register appears only in the CP's non-real-time path through the RBBM. After being written, consecutive transactions will be held-off until the idle status (defined below) is met. Note that the data is stalled prior to the Command FIFO and Event Engine in the RBBM and that both "queued" and "non-queued" transactions are held-off.

The CP PM4 packet "WAIT_FOR_IDLE" writes to this register on the CP data path.

**NQWAIT_UNTIL (WO)**
**Non-Queued Explicit Synchronization Register**

| Field Name | Bit(s) | Description |
|---|---|---|
| RESERVED | 31:1 | Reserved |
| WAIT_GUI_IDLE | 0 | Wait for graphics engine to be idle with non-Real Time processing, and for Command FIFO to be empty. |

## 9.5 Performance Monitoring Registers (Updated: 10-29-2002)

The RBBM also shadows the CP_PERFMON_CNTL[3:0] for resetting, enabling, and disabling the counters.

**RBBM_PERFCOUNTER1_HI (RO)**
**Performance Counter #1 High**

| Field Name | Bit(s) | Description |
|---|---|---|
| Reserved | 31:16 | Reserved |
| PERF_COUNT1_HI | 15:0 | Performance Counter #1 High Bits. |

**RBBM_PERFCOUNTER1_LO (RO)**
**Performance Counter #1 Low**

| Field Name | Bit(s) | Description |
|---|---|---|
| PERF_COUNT1_LO | 31:0 | Performance Counter #1 Lower Bits. |

**RBBM_PERFCOUNTER2_HI (RO)**
**Performance Counter #2 High**

| Field Name | Bit(s) | Description |
|---|---|---|
| Reserved | 31:16 | Reserved |
| PERF_COUNT2_HI | 15:0 | Performance Counter #2 High Bits. |

**RBBM_PERFCOUNTER2_LO (RO)**
**Performance Counter #2 Low**

| Field Name | Bit(s) | Description |
|---|---|---|
| PERF_COUNT2_LO | 31:0 | Performance Counter #2 Lower Bits. |

**RBBM_PERFCOUNTER1_SELECT (R/W)**
**Performance Counter #1 Select**

| Field Name | Bit(s) | Description |
|---|---|---|
| Reserved | 31:6 | Reserved |
| PERF_COUNT1_SEL | 5:0 | Counter #1 Input Select. |

**RBBM_PERFCOUNTER2_SELECT (R/W)**
**Performance Counter #2 Select**

| Field Name | Bit(s) | Description |
|---|---|---|
| Reserved | 31:6 | Reserved |
| PERF_COUNT2_SEL | 5:0 | Counter #2 Input Select. |

## 9.6 Debug Registers (Updated: 10-25-2002)

The debug register is in the design in case there is a need to have any static register settings.

BIF can write to this register by snooping it without going through the RBBM's scheduling (arbitration). When BIF requests a write to this register , it will ignore the RBBM_BIF_rdy. This write request will be decoded as soon as it enters RBBM after input register stage and the very next clock, RBBM_DEBUG register will be written. If IGNORE_RTR is set, then it will allow RBBM to release the pending requests in the scheduler. The advantage of BIF able to write to the RBBM_DEBUG register without RBBM_BIF_rdy is, it allows system to come out of stuck condition, which might have occurred due to one of the clients not being ready. CP can also write to this register but as if it is writing to any other register. In case CP and BIF are trying to write to this register at the same time, BIF will over-write CP and write from CP will be ignored.

Note: As the write request to RBBM_DEBUG register from BIF is snooped within, it will not be allowed to get scheduled later in the pipe. Also, the read request for this register is similar to other register reads (will not be snooped). If read and write request to this register occurs at the same time then the previously registered data will be put on the read bus.

**RBBM_DEBUG**
**RBBM_DEBUG Register**

| Field Name | Bit(s) | Description |
|---|---|---|
| RBBM_DEBUG | 31:16 | Reserved |
| HYSTERESIS_RT_GUI_ACTIVE | 15:12 | Keep asserting RT_GUI_ACTIVE for these many more clocks after it is deasserted. Default=0. |
| HYSTERESIS_NRT_GUI_ACTIVE | 11:8 | Keep asserting GUI_ACTIVE for these many more clocks after it is deasserted. Default=0. |
| UNUSED | 7:3 | Reserved |
| IGNORE_CP_SCHED_NQ_BIF | 4 | If set the CP Paths' Scheduler Status is Ignored in the Host's NQ wait conditions. Default = 0. |
| IGNORE_CP_SCHED_ISYNC | 3 | If set the CP Paths' Scheduler Status is Ignored in the Isync wait conditions. Default = 0. |
| IGNORE_CP_SCHED_WU | 2 | If set the CP Paths' Scheduler Status is Ignored in the Wait_Until wait conditions. Default = 0. |
| IGNORE_RTR | 1 | If set, the RBBM will ignore all real-time and non-real-time ready-to-receive signals when processing a transaction.<br>Default = 0. |
| SWAP_BE | 0 | Swaps byte enables of BIF Data for endian swaps if set. Default = 0. |

## RBBM_READ_ERROR
## Captures Error Information for Read Operations that Timed-Out

| Field Name | Bit(s) | Description |
|---|---|---|
| READ_ERROR | 31 | Read Error in RBBM<br>Read:<br>0 – No Error<br>1 – Read error occurred in RBBM during read operation<br>Write:<br>0 – Force Clear the Error condition status<br>1 – Set Read Error (For state restoration from power-down)<br>Cleared when RDERR_INT_ACK bit is set in RBBM_INT_ACK register. |
| READ_REQUESTER | 30 | Indicates whether a BIF or CP read resulted in an error condition.<br>Read:<br>0 – CP<br>1 – Host<br>Write: (For state restoration from power-down)<br>0 – CP<br>1 – Host |
| UNUSED | 29:17 | Unused |
| READ_ADDRESS | 16:2 | Read: Target address for the read operation where an error was detected.<br>Write: Target address for the read operation where an error was detected.<br>(Write is for state restoration from power-down) |
| Reserved | 1:0 | Reserved |

## 9.7 Soft Reset Register (Updated: 08-01-2002)

When a bit is set in this register, the soft reset signal to the corresponding unit will be set. The signal will remain asserted until the bit is cleared.

The RBBM_SOFT_RESET register is mapped to the same address in the IO and MMR decode spaces.

| RBBM_SOFT_RESET<br>Soft Reset Generation | | |
|---|---|---|
| **Field Name** | **Bit(s)** | **Description** |
| Reserved | 31:19 | Reserved for Future Assignment |
| SOFT_RESET_CGM | 18 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_ROM | 17 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_VGT | 16 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_SC | 15 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_IDCT | 14 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_VGA | 13 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_CG | 12 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_DISP | 11 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_VIP | 10 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_DB | 9 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_MC | 8 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_RB | 7 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_SX | 6 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_SQ | 5 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_RC | 4 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_MH | 3 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_PA | 2 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_BIF | 1 | Soft Reset to this block. Default = 0 |
| SOFT_RESET_CP | 0 | Soft Reset to this block. Default = 0 |

## 9.8 RBBM Status (Updated 10-24-2002)

The RBBM status register provides status for transactions in the RBBM and "Busy" status for other clients in the chip.

**RBBM_STATUS (RO)**
**Chip & RBBM Status Register**

| Field Name | Bit(s) | Description |
|---|---|---|
| GUI_ACTIVE | 31 | Non-Real-Time Graphics Pipe is Busy<br>GUI_ACTIVE = Non-RT_Busy (From Wait_Until Register) or CP_RBBM_nrt_busy or PFRQ_PENDING or CFRQ_PENDING or CPRQ_PENDING or VGT_DMA_BUSY. |
| RC_RBBM_cntx0_busy | 30 | Renderer Common is Busy for Context #0. |
| RC_RBBM_cntx17_busy | 29 | Renderer Common is Busy for Contexts #1 through #7. |
| SQ_RBBM_cntx0_busy | 28 | Sequencer is Busy for Context #0. |
| SQ_RBBM_cntx17_busy | 27 | Sequencer is Busy for Context #1 through #7. |
| VGT_RBBM_busy | 26 | Vertex Grouper Tessellator is Busy. |
| PA_RBBM_busy | 25 | Primitive Assembly is Busy. |
| SC_RBBM_cntx0_busy | 24 | Scan Converter is Busy for Context #0. |
| SC_RBBM_cntx17_busy | 23 | Scan Converter is Busy for Contexts #1 through #7 |
| TPC_RBBM_busy | 22 | Texture Pipeline Common is Busy. |
| u0_SX_RBBM_busy | 21 | Shader Export #0 is Busy. |
| u1_SX_RBBM_busy | 20 | Shader Export #1 is Busy. |
| MH_RBBM_coherency_busy | 19 | Surface Coherency Logic is Busy. |
| MH_Clean | 18 | Memory Hub is Clean. |
| CP_RBBM_rt_enable | 17 | Real-Time Enable from the CP. |
| CP Non-Real-Time Busy | 16 | Command Processor is Busy for non-Real-Time (CP_RBBM_nrt_busy). |
| CP Real-Time Busy | 15 | Command Processor is Busy for Real-Time (CP_RBBM_rt_busy). |
| RBBM_WU_BUSY | 14 | The RBBM's Non-Real Time Wait Until Unit is Busy.<br>i.e. Waiting for a Non-Real-Time synchronization event. |
| RBBM_RT_WU_BUSY | 13 | The RBBM's Real-Time Wait Until Unit is Busy<br>i.e. Waiting for a Real-Time synchronizing event. |
| VGT_BUSY_NO_DMA | 12 | Busy from the VGT that does not include the Index DMA engine status. |
| PFRQ_PENDING | 11 | There is a request from the CP's Pre-Fetch Parser Pending in the RBBM. |
| CFRQ_PENDING | 10 | There is a queued request from the CP Pending in the RBBM.<br>Includes the RBBM_EE_BUSY status. |
| CPRQ_PENDING | 9 | There is a non-queued request from the CP Pending in the RBBM. |
| HIRQ_PENDING | 8 | There is a BIF request Pending in the RBBM. |
| RTRQ_PENDING | 7 | There is a Real-Time request Pending in the RBBM.<br>Includes the RBBM_RT_WU_BUSY status. |
| RT_GUI_ACTIVE | 6 | Real-Time Graphics Pipe is Busy<br>RT_GUI_ACTIVE = RT_Busy (From RTS_Wait_Until Register) or CP_RBBM_rt_busy or RTRQ_PENDING. |
| TC_RBBM_busy | 5 | Texture Cache is Busy. |
| CMDFIFO_AVAIL | 4:0 | Number of available entries in the Command FIFO. |

## 9.9 RBBM Status #2 (Updated 05-22-2002)

The RBBM Status #2 register provides visibility for input signals that are not included in the RBBM_STATUS register.

**RBBM_STATUS2 (RO)**
**RBBM Status #2 Register**

| Field Name | Bit(s) | Description |
|---|---|---|
| RC_RBBM_cntx17_clean | 31 | RC is clean for contexts #1 through #7. |
| RC_RBBM_cntx0_clean | 30 | RC is clean for context #0. |
| Reserved | 29:14 | Reserved |
| IDCT_semaphore | 13 | Semaphore Status from IDCT. |
| PAD_extern_signal | 12 | Status Signal from Pads of Chip. |
| VIP_RBBM_h0dma_idle | 11 | VIP's Host DMA Channel 0 is Idle. |
| VIP_RBBM_h1dma_idle | 10 | VIP's Host DMA Channel 1 is Idle. |
| VIP_RBBM_h2dma_idle | 9 | VIP's Host DMA Channel 2 is Idle. |
| VIP_RBBM_h3dma_idle | 8 | VIP's Host DMA Channel 3 is Idle. |
| D1OVL_update_pending | 7 | Flip Pending from Overlay 1. |
| D2OVL_update_pending | 6 | Flip Pending from Overlay 2. |
| D1MODE_vline | 5 | Status from 1st Display Controller. |
| D2MODE_vline | 4 | Status from 2nd Display Controller. |
| D1GRPH_update_pending | 3 | Pending Flip status from 1st Display Controller. |
| D2GRPH_update_pending | 2 | Pending Flip status from 2nd Display Controller. |
| BIF_RBBM_agp_flush | 1 | AGP Flush. |
| MH_hdp_clean | 0 | HDP Clean. |

## 9.10 Implicit Sync Control (Updated 10-25-2002)

The implicit sync controls in the RBBM are only for legacy controls not related to 2D/3D switches. The 2D/3D synchronization is managed in the CP. The implicit sync controls only affect the non-real-time path (CF path).

Isync control waits for CF's scheduler request to be empty and CP's scheduler request to be empty, then it waits for 48 clocks to sample Non-RT graphics pipe idle. The 48 clock wait is to make sure all the pending initiators have been received by the clients (i.e. Not stuck in the repeater flops).

**RBBM_ISYNC_CNTL**
**Implicit Synchronization Control**

| Field Name | Bit(s) | Description |
|---|---|---|
| Reserved | 31:6 | Reserved |
| ISYNC_CPSCRATCH_IDLEGUI | 5 | A write to any of the CP's Scratch Registers stalls if the Graphics Engine is busy processing non-Real Time transactions.<br>Default = 0. |
| ISYNC_WAIT_IDLEGUI | 4 | A write to the WAIT_UNTIL register stalls if the Graphics Engine is busy processing non-Real Time transactions.<br>Default = 0. |
| Reserved | 3:0 | Reserved |

## 9.11 RBBM Interrupt Registers (Updated: 04-26-2002)

The following registers control the interrupt functionality in the RBBM.

**RBBM_INT_CNTL (R/W)**
**RBBM Interrupt Control**

| Field Name | Bit(s) | Description |
|---|---|---|
| Reserved | 31:20 | Reserved |
| GUI_IDLE_INT_MASK | 19 | GUI Idle Mask<br>The mask is used to enable the generation of the interrupt.<br>0 – Disabled (Default – Also cleared on Reset)<br>1 – Enabled: Interrupt will generate interrupt to the BIF. |
| Reserved | 18:1 | Reserved |
| RDERR_INT_MASK | 0 | Interrupt Mask for a detected Read Error.<br>The mask is used to enable the generation of the interrupt.<br>0 – Disabled (Default – Also cleared on Reset)<br>1 – Enabled: Interrupt will generate interrupt to the BIF. |

**RBBM_INT_STATUS (R/W)**
**RBBM Interrupt STATUS**

| Field Name | Bit(s) | Description |
|---|---|---|
| Reserved | 31:20 | Reserved |
| GUI_IDLE_INT_STAT | 19 | Interrupt status for GUI Idle.<br><u>Read:</u><br>0 – No Event (Interrupt did not occur)<br>1 – Event Occurred<br><u>Write:</u><br>0 – No affect. (Default – Also cleared on Reset)<br>1 – Set Interrupt Status (For state restoration from power-down). |
| Reserved | 18:1 | Reserved |
| RDERR_INT_STAT | 0 | Interrupt status for a detected Read Error.<br><u>Read:</u><br>0 – No Event (Interrupt did not occur)<br>1 – Event Occurred<br><u>Write:</u><br>0 – No affect. (Default – Also cleared on Reset)<br>1 – Set Interrupt Status (For state restoration from power-down). |

**RBBM_INT_ACK (WO)**
**RBBM Interrupt Acknowledge**

| Field Name | Bit(s) | Description |
|---|---|---|
| Reserved | 31:20 | Reserved |
| GUI_IDLE_INT_ACK | 19 | Interrupt acknowledge for a GUI Idle Interrupt.<br><u>Write:</u><br>0 – No affect.<br>1 – Clear Interrupt Status (Non-Persistent Write). |
| Reserved | 18:1 | Reserved |
| RDERR_INT_ACK | 0 | Interrupt acknowledge for a detected Read Error.<br><u>Write:</u><br>0 – No affect.<br>1 – Clear Interrupt Status (Non-Persistent Write). |

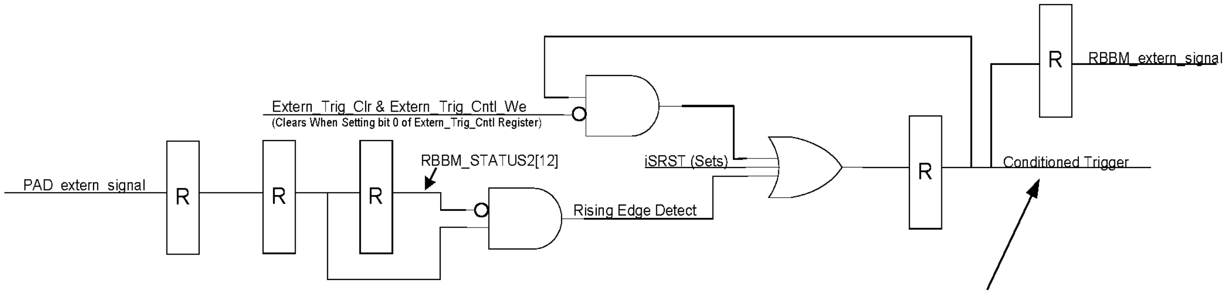
| MASTER_INT_SIGNAL (RO)<br>Master Interrupt Signal – Read Only | | |
|---|---|---|
| **Field Name** | **Bit(s)** | **Description** |
| RBBM_INT_STAT | 31 | Interrupt from the RBBM |
| CP_INT_STAT | 30 | Interrupt from the CP |
| Reserved | 29:16 | Reserved |
| MC1_INT_STAT | 15 | Interrupt 1 from the Memory Controller |
| MC0_INT_STAT | 14 | Interrupt 0 from the Memory Controller |
| VGT_INT_STAT | 13 | Interrupt from the Vertex Grouper Tessellator |
| PA_INT_STAT | 12 | Interrupt from the Primitive Assembly |
| DEBUG_INT_STAT | 11 | Interrupt from the DEBUG Unit. |
| RB_INT_STAT | 10 | Interrupt from the Renderer Backend |
| RC_INT_STAT | 9 | Interrupt from the Renderer Common |
| SX_INT_STAT | 8 | Interrupt from the Shader Export |
| SQ_INT_STAT | 7 | Interrupt from the Sequencer |
| CG_INT_STAT | 6 | Interrupt from the Clock Generator |
| MH_INT_STAT | 5 | Interrupt from the Memory Hub |
| Reserved | 4 | Reserved |
| IDCT_INT_STAT | 3 | Interrupt from the IDCT |
| VIP_INT_STAT | 2 | Interrupt from the VIP |
| VGA_INT_STAT | 1 | Interrupt from the VGA |
| DISPLAY_INT_STAT | 0 | Interrupt from Display Engine |

## 9.12 External Trigger Control Register (Updated: 10-24-2002)

The RBBM inputs the signal "PAD_extern_signal" from the pads of the chip. This input is double-registered and then a rising edge detection is performed to set a set/reset flop. The output of the S/R flop is used for both the Real-Time and Non-Real-Time WAIT_EXTERN_SIGNAL wait_until conditions. It is also output from the RBBM as RBBM_extern_signal and is used for initiating Real-Time stream. The following is a diagram of how the PAD_extern_signal is processed:

Updated: 10/24/2002
John Carey

R400 External Trigger



1. Non-RT Wait Until: WAIT_EXTERN_SIGNAL
2. Real-Time Wait_Until: WAIT_EXTERN_SIGNAL
3. Output as RBBM_extern_signal for RT stream initiation
4. Wired to Extern_Trig_Cntl[1] as Extern_Trig_Read

**EXTERN_TRIG_CNTL**
**PAD External Signal Control**

| Field Name | Bit(s) | Description |
|---|---|---|
| Reserved | 31:2 | Reserved |
| EXTERN_TRIG_READ | 1 | Read-Only:<br>0 – Indicates WAIT condition is active for the External Trigger.<br>1 – Indicates WAIT condition is not active for the External Trigger. |
| EXTERN_TRIG_CLR | 0 | Write-Only:<br>0 – No affect.<br>1 – Clears External Trigger Set/Reset Flop (Non-Persistent Write). |

# 10. Performance Counters (Updated: 11-07-2002)

The RBBM contains two identical 48-bit Performance Counters to aid in measuring the performance of various blocks. This is accomplished by counting the number of clock cycles for which various status signals are asserted (true) from their respective blocks over some period of time. The following combinations are possible by setting PERFCOUNTER_SELECTs to the appropriate listed value:

| PERF_SEL[5:0] | Signal Combination for Counting |
|---|---|
| 0x0 | Tie High – Count Number of Clocks |
| 0x1 | Non-Real-Time Busy |
| 0x2 | RC_RBBM_cntx0_busy |
| 0x3 | RC_RBBM_cntx17_busy |
| 0x4 | SQ_RBBM_cntx0_busy |
| 0x5 | SQ_RBBM_cntx17_busy |
| 0x6 | VGT_RBBM_busy |
| 0x7 | VGT_RBBM_no_dma_busy |
| 0x8 | PA_RBBM_busy |
| 0x9 | SC_RBBM_cntx0_busy |
| 0xA | SC_RBBM_cntx17_busy |
| 0xB | TPC_RBBM_busy |
| 0xC | TC_RBBM_busy |
| 0xD | u0_SX_RBBM_busy |
| 0xE | u1_SX_RBBM_busy |
| 0xF | MH_RBBM_coherency_busy |
| 0x10 | CP_RBBM_nrt_busy |
| 0x11 | CP_RBBM_rt_busy |
| 0x12 | CP_RBBM_dma_busy |
| 0x13 | Non-RT Waiting on IDCT Semaphore ** |
| 0x14 | Non-RT Waiting for "Both CRTC PFLIP" ** |
| 0x15 | Non-RT Waiting for External Trigger ** |
| 0x16 | Non-RT Waiting for CP's DMA to go Idle ** |
| 0x17 | Non-RT Waiting for AGP Flush ** |
| 0x18 | Non-RT Waiting for Host Idle Clean Status ** |
| 0x19 | Non-RT Waiting for Graphics Pipe to be Idle ** |
| 0x1A | Non-RT Waiting for Graphics Pipe to be Idle and Clean ** |
| 0x1B | RBBM_BIF_int – Combined Interrupt Signal to the BIF. |

***** Note: These performance conditions do not include the 48 additional clocks that the RBBM waits for the previous initiators to arrive at the clients before evaluating the wait condition.***

## 11. Design for Test (Updated: 10-29-2001)

Signals in the RBBM are connected to the Test Bus for observation at the pins of the chip. The RBBM decodes select 0x02 as in prior designs. Refer to the R400 RBBM test bus document for details on the wiring.

## 12. Power Management (Updated: 04-02-2002)

The clock within the RBBM is a permanent clock. No clock gating is used to disable the RBBM clock. The input “sclk” is run through the ati_master_permanent clock gating module so that the RBBM’s clock is aligned to other clients in the design.

## 13. Physical Aspects (Updated: 04-02-2002)

16x48 RAM for the Command FIFO will be included in the R400 design.

| | ORIGINATE DATE | EDIT DATE | DOCUMENT-REV. NUM. | PAGE |
|---|---|---|---|---|
| | 10 July, 2003 | [date \@ "d MMMM, | GEN-CXXXXX-REVA | 1 of 34 |

**Author:** Larry Seiler

| | |
|---|---|
| **Issue To:** | **Copy No:** |

# R400 Memory Format Specification

## Version 0.10

**Overview:** This specification describes how pixels and other data are stored in the frame buffer.

AUTOMATICALLY UPDATED FIELDS:

| | |
|---|---|
| **Document Location :** | C:\r400\doc_lib\design\chip\memory\R400_MemoryFormat.doc |
| **Current Intranet Search Title**: | R400 Frame Buffer Layout Specification |

| APPROVALS | |
|---|---|
| Name/Dept | Signature/Date |
| | |
| | |
| | |

Remarks:

THIS DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

## Table Of Contents

## Table Of Figures

## Table Of Tables

# Revision Changes:

| | |
|---|---|
| **Rev 0.1 (Larry Seiler)**<br>Date: March 8, 2001 | First draft |
| **Rev 0.2 (Larry Seiler)**<br>Date: June 25, 2001 | Complete rewrite. Added 1D and 3D formats and modified the 2D formats. Added addressing modes, and some information on depth compression. |
| **Rev 0.3 (Larry Seiler)**<br>Date: July 3, 2001 | Added more 3D formats and compressed formats. Various minor changes. |
| **Rev 0.4 (Larry Seiler)**<br>Date: October ???, 2001 | Partially updated version |
| **Rev 0.5 (Larry Seiler)**<br>**Date: December 20, 2001** | Revised micro-tile formats, significant additions. |
| **Rev 0.6 (Larry Seiler)**<br>**Date: January 29, 2002** | Updated color and depth compression formats, added pixel format descriptions, added tiling for per-tile data and non-standard pixels. |
| **Rev 0.7 (Larry Seiler)**<br>**Date: March 5, 2002** | Change to 32x32 2D macro-tile size and 32x16x4 3D macro-tile size. Added addressing equations and 2D mipmap packing. |
| **Rev 0.8 (Larry Seiler)**<br>**Date: June 4, 2002** | File name and location changed. All surfaces now have a 4KB alignment constraint for allocation. Pitch must be 256-byte multiple for linear arrays. LocalBase eliminated from tiling equations. Zplane format changed. Multi-sample color format changed. |
| **Rev 0.8B (Larry Seiler)**<br>**Date:** | Changed Zplane format to include the MultiSample bit, changed a parenthesis in 3D Ytile computation |
| **Rev 0.9 (Larry Seiler)**<br>**Date: May, 2003, minor edits July 10, 2003** | Added alternate 2D tiling formats for efficient depth buffering and to support depth with a 3D slice. |
| **Rev 0.10 (Larry Seiler)**<br>**Date: October 1, 2003** | Change depth format: separate/expanded modes interleave depth values for multi-sampling and Pmask data is now stored after the Zplanes. |

# 1. Background

This document describes how the R400 family maps pixels and other data into local (video) memory and system (AGP) memory. The R400 uses 32-bit device addresses that map both local memory and system memory within a $2^{32}$-byte device address space. Local memory is accessed through one, two or four memory controllers that each access up to $2^{28}$-bytes (256M-bytes) of on-board memory.

Data can be organized into 1D, 2D or 3D arrays. Types of data include coordinates/normals, uncompressed pixels/texels, uncompressed depth/stencil values, and a variety of compressed data formats. 2D and 3D arrays can be organized in interleaved tiles for higher performance. All array sizes can be stored in a linear format, which stores pixels along a scanline at sequential device addresses, but R400 does not support depth buffer operations in linear format.

The subsections below describe some important common characteristics of all of the memory formats in all of the array sizes. These topics include dividing local memory into disjoint subsets, address alignment constraints, and address modes. Following sections describe the bit formats of individual data elements (pixels, texels, etc.), tiling formats for 1D, 2D, and 3D arrays of data elements, and special tile formats for multi-sample data and depth/stencil data.

## 1.1 System vs. Local Memory

Data can be stored either in local (on-board or frame buffer) memory or in system (AGP/PCI) memory. A specified range of the device address space maps to local memory and the rest maps to system memory.

Logic blocks that compute device addresses need not be aware of whether an address is in local or system memory, though some logic blocks (e.g. the Display Controller) require data in local memory due to latency issues. The system memory bus has significantly lower bandwidth than the local memory bus and has tremendously longer read latency. AGP memory accesses use either 256-bit or 512-bit bursts.

{Give specific ranges for AGP latency. Reference Tom's memory space spec.}

## 1.2 Local Memory Subsets

The R400 family accesses local on-board memory through one to four independent memory controllers. The R400 has four memory controllers, each of which provides access to one quarter of the local memory. The RV400 has two memory controllers, each of which provides access to half of the local memory. A future integrated version of the chip will have a single memory controller.

Each memory controller is associated with a corresponding render backend block. Each render backend can only access the local memory associated with its own memory controller. Hence the local memory is divided into subsets, so that each memory controller stores the pixels or other data elements that are processed by its associated render backend. All of the render backends can access system memory.

DDRAM memory is divided into multiple banks and pages. Each bank is in effect a separate memory array inside the DDRAM. Each page contains bits from a single row in the internal DDRAM memory array for a given bank. The size of a page in bytes depends on the specific DDRAM part and the number that are used for a single memory controller. Accessing data within the same page of a bank (which is called "column access") is dramatically faster than accessing data in a different page of the bank (which is called "row access"). Accordingly, much effort in the tiling design goes into grouping accesses that are on the same page and avoiding situations where two different pages in a bank must be accessed without intervening accesses in other banks.

Each memory controller contains two separate memory subsets, in order to improve memory access efficiency. The R400 family supports DDRAM memory that is organized into four banks: A, B, C, and D, so each of the two memory subsets per memory controller contains data from two of the banks. For each memory controller, one subset includes banks A and B and the other subset includes banks C and D. Since R400 has four memory controllers, it has eight memory subsets, which are called ab0, ab1, ab2, ab3, cd0, cd1, cd2, and cd3. RV400 has two memory controllers, so its four memory subsets are called ab0, ab1, cd0, and cd1.

Within a memory subset, memory accesses alternate between two banks at the page boundaries. Consider memory subset ab0. The first word of this memory subset is in page 0 of bank A. This is followed by the rest of the words of bank A in page 0, then by page 0 of bank B. Next comes page 1 of bank A, then page 1 of bank B, etc. As a result, the two banks are interleaved on a page-by-page basis. The figure below illustrates this bank alternation for a DDRAM with pages consisting of 256 words. The word size depends on the specific DDRAM type. Each MC reads 64-bit words from multiple DDRAMs in parallel, depending on the DDRAM configuration.

| page 0 bank A | | | | page 0 bank B | | | | page 1 bank A | | | | page 1 bank B | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| word 0 | word 1 | ... | word 255 | word 0 | word 1 | ... | word 255 | word 0 | word 1 | ... | word 255 | word 0 | word 1 | ... | word 255 |

**Figure 1: Local Memory Subset for Banks AB**

Dividing the memory into subsets in this fashion ensures that sequential accesses within a memory subset are always either within the same page or are within different banks. For example, suppose that a sequential access starts at page 0, bank A, word 255. A large number of accesses to bank B occur before accessing bank A, page 1, word 0. This gives the DDRAM array time to perform the slow “row access” to bank A, in parallel with the fast “column accesses” within bank B.

## 1.3 Units of Memory

This document describes memory formats in terms of four levels of addressable units in local and system memory. Each addressable unit has different address alignment constraints.

Individual data elements have a wide variety of sizes. Each occupies $2^N$-bytes for some value of N and each is naturally aligned on a $2^N$-byte boundary. The most common size is 4-bytes (32-bits), which could represent, for example, a 32-bit ARGB pixel or a 32-bit IEEE floating point value.

**Data elements** are organized into 16-byte (128-bit) micro-tiles. This corresponds to the read/write bus size between each memory controller and its render backend, though each memory access reads or writes an aligned pair of micro-tiles.A micro-tile may contain four 32-bit floats or a larger number of smaller data elements. For 1D arrays, a micro-tile always contains sequential data in the 1D array. For 2D and 3D arrays, a micro-tile contains data from a single scanline for 16-bit, 32-bit, 64-bit, and 128-bit data elements.

**Tiles** are variable-sized. For 1D arrays, each tile stores exactly 64-bytes (512-bits), which stores a variable number of data elements, e.g. 64 8-bit pixels or four 128-bit pixels that each contain four 32-bit floats. For 2D and 3D arrays of pixels or texels, each tile stores an 8x8 array of data elements, which occupies a multiple of 64-bytes (512-bits). The most typical data element size for 2D arrays is 32-bits, which results in a 256-byte (2K-bit) tile size. 2D arrays can also store a single per-tile data element, instead of 64 pixel or texel data elements. The Render Backend uses this mode.

A **macro-tile** is the basic unit of memory allocation in both linear and tiled formats. For linear arrays, each macro-tile is exactly 4K-bytes and contains exactly 16 tiles. For 2D tiled arrays, each macro-tile contains 32x32 pixels, arranged as a 4x4 sub-array of 8x8 pixel tiles. For 3D tiled arrays each macro-tile contains 32x16x4 pixels, arranged as a 4x2x4 sub-array of 8x8x1 pixel tiles.

Finally, a **surface** is a contiguous range of device address space that is all interpreted the same way, e.g. as either a linear array or a 2D or 3D tiled array with a specific pitch and pixel size. Each surface must start on a 4K-byte aligned address in device address space.

# 2. Data Element Formats

This section describes pixel formats, texel formats, per-tile data formats, and other types of data elements that the R400 reads and writes. These formats are used in the Memory Hub (MH) block to support client memory accesses, in the Texel Central (TC) block to read and interpolate data for the shader programs, and in the Render Backend (RB) block to read and write data render data.

The following subsections divide the data element formats into four groups. Displayable pixel formats are fully supported by R400, including displaying them to the monitor. Renderable pixel formats are usable as render targets with full support for alpha blending (with one exception), as well as for texture inputs. The Texel-Only Pixel formats may be used as texel inputs or render targets, but cannot be alpha blended. The Texel-Only formats cannot be used as render targets. Finally, the Special Data Formats have specific, limited uses.

Each pixel contains between one and four components, named C0 through C3. The TC and RB blocks both contain pixel format descriptors that specify how to interpret the components as numbers and how to map them to the four components that are computed in the shader pipe. The MH block passes them through without interpretation.

{**Note:** define the number formats here: floating-point, repeating fraction, integer, etc.}

## 2.1 Displayable Pixel Formats

The pixel formats described below are fully supported by the MH, TC, and RB blocks. Additionally, each of these frame buffer formats may be displayed to the monitor.

**{List the displayable pixel formats. Questions: (1) is GRPH_SWAP_RB supported for all formats, or only for the ones listed in the display spec? (2) How are YUV modes supported, and which ones? (3) How does the display logic support unsigned vs. signed vs. float number formats? All three expand differently for use with the linear interpolation table.}**

## 2.2 Renderable Pixel Formats

Renderable pixel formats are those that the RB (render backend) block can produce. They are also fully supported by the MH and TC blocks. Each pixel contains between one and four components, named C0 through C3. The MH block transfers whole pixels without interpreting their content. The TC and RB blocks both contain pixel format descriptors that specify how to interpret the components as numbers and how to map them to the four components that are computed in the shader pipe. The TC allows an arbitrary mapping of the input components to the shader pipe components. The RB supports more limited component mappings, as described below.

The figures in this subsection list multiple names for each renderable pixel format. Names of the form FMT_* are enumeration constants from enum type SurfaceFormat. Names of the form COLOR_* are enumeration constants from enum type ColorFormat, which contains the subset of surface formats that are renderable. Each COLOR_* enumeration name has the same value as a corresponding FMT_* enumeration name.

The figure below illustrates the single-component renderable pixel formats, with their enumeration names. The 8-bit format has three separate pairs of names to support format numbers used by legacy code. The 16-bit formats can either be explicitly floating point or else may use one of several fixed-point number formats. For the 32-bit component size, only the floating-point format is renderable, and the Render Backend cannot alpha blend that component size. The Render Backend can map either the shader pipe Red or Alpha channel to C0.

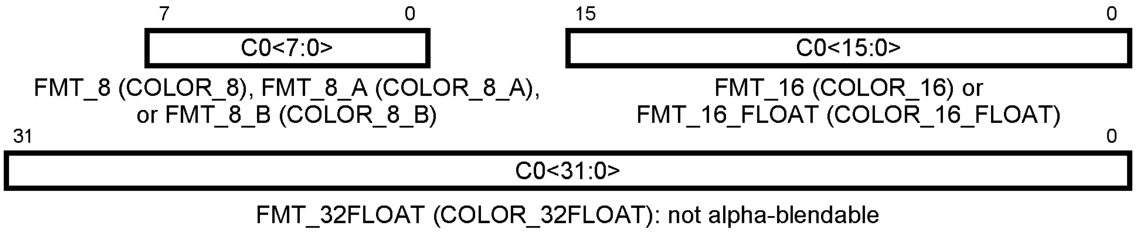
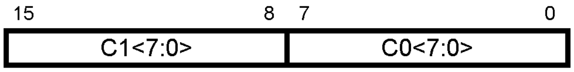
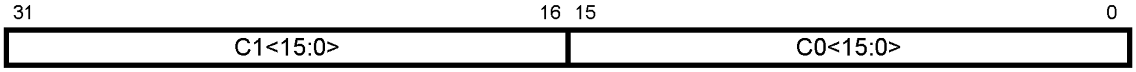


**Figure 2: One-Component Renderable Pixel Formats**

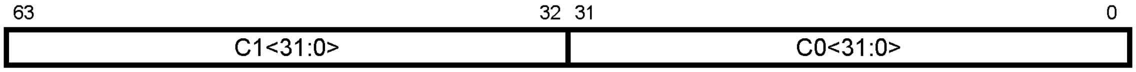
The next figure illustrates the two-component renderable pixel formats. Each format contains two equal size components, labeled C0 and C1. As for the single-component formats, the Render Backend cannot render to 32-bit components unless they are floating point and cannot alpha blend even floating point 32-bit components. The Render Backend can map either the shader pipe GR or AR components to C1 and C0, in that order.



FMT_8_8 (COLOR_8_8)



FMT_16_16 (COLOR_16_16) or FMT_16_16_FLOAT (COLOR_16_16_FLOAT)



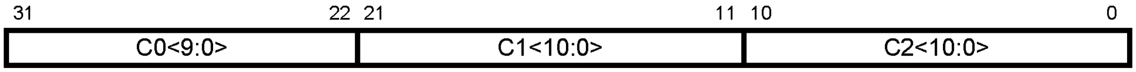
FMT_32FLOAT (COLOR_32FLOAT): not alpha-blendable

**Figure 3: Two-Component Renderable Pixel Formats**

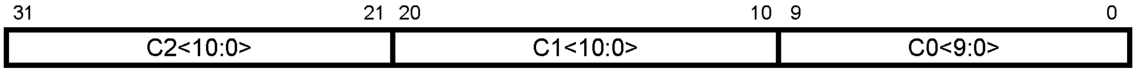
The next figure illustrates the three-component renderable pixel formats. Each format contains two components of one size and one component that is one bit different in size, labeled C0, C1 and C2. The Render Backend can map either the shader pipe BGR or RGB components to C2, C1 and C0, in that order.

FMT_5_6_5 (COLOR_5_6_5)

FMT_6_5_5 (COLOR_6_5_5)



FMT_10_11_11 (COLOR_10_11_11)



FMT_11_11_10 (COLOR_11_11_10)

**Figure 4: Three-Component Renderable Pixel Formats**

The final figure illustrates the four-component renderable pixel formats. Two of the formats reduce the size of the C3 component and the rest provide an equal number of bits to each component. As for the two-component formats, the RB can render either floating-point or fixed-point 16-bit components, but only floating-point 32-bit components, and it cannot alpha-blend 32-bit components. The Render Backend can map either the shader pipe ABGR or ARGB components to C3, C2, C1 and C0, in that order.

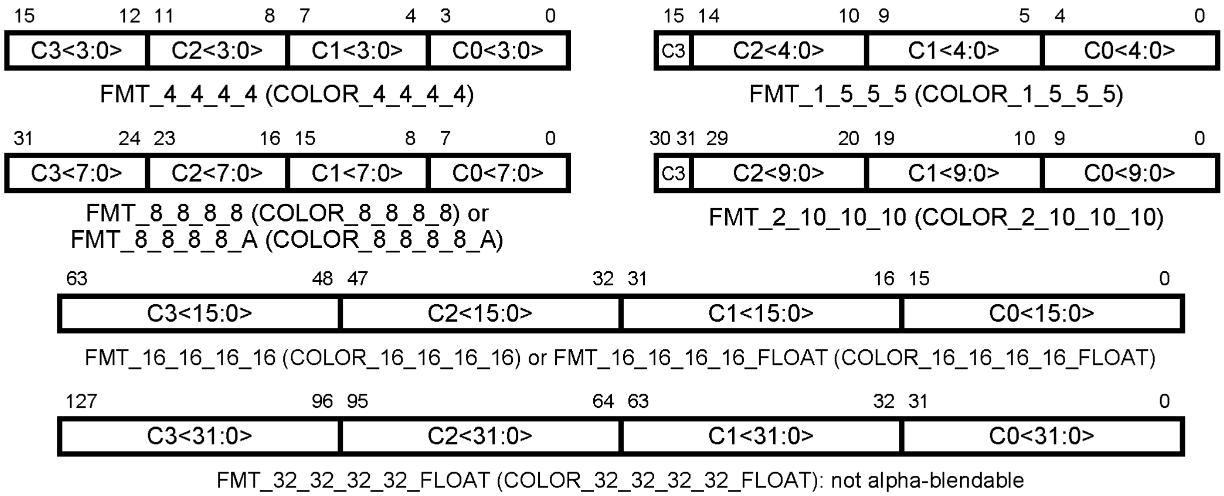
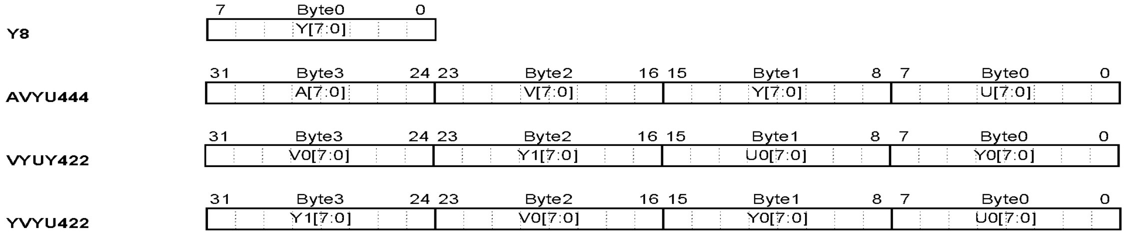


Figure 5: Four-Component Renderable Pixel Formats

## 2.3 Texel-Only Formats

The pixel formats described below are supported by the MH and TC but cannot be read or written by the RB. The MH passes them through without interpreting the bits in each pixel. The MH and TC also support the renderable pixel formats, which are described in the preceeding subsection.

**{Describe the YUV formats.}**



The DX formats provide texture compression. The bitmap formats store glyphs in one of several bit orders. **{provide more detail.}**

**DXT1**

63 Byte7 56 55 Byte6 48 47 Byte5 40 39 Byte4 32

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T33 | T32 | T31 | T30 | T23 | T22 | T21 | T20 | T13 | T12 | T11 | T10 | T03 | T02 | T01 | T00 |

31 Byte3 24 23 Byte2 16 15 Byte1 8 7 Byte0 0

| | | | | | |
|---|---|---|---|---|---|
| R1[4:0] | G1[5:0] | B1[4:0] | R0[4:0] | G0[5:0] | B0[4:0] |

**DXT2/DXT3**

127 Byte15 120119 Byte14 112111 Byte13 104103 Byte12 96

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T33 | T32 | T31 | T30 | T23 | T22 | T21 | T20 | T13 | T12 | T11 | T10 | T03 | T02 | T01 | T00 |

95 Byte11 88 87 Byte10 80 79 Byte9 72 71 Byte8 64

| | | | | | |
|---|---|---|---|---|---|
| R1[4:0] | G1[5:0] | B1[4:0] | R0[4:0] | G0[5:0] | B0[4:0] |

63 Byte7 56 55 Byte6 48 47 Byte5 40 39 Byte4 32

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A33[3:0] | A32[3:0] | A31[3:0] | A30[3:0] | A23[3:0] | A22[3:0] | A21[3:0] | A20[3:0] |

31 Byte3 24 23 Byte2 16 15 Byte1 8 7 Byte0 0

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A13[3:0] | A12[3:0] | A11[3:0] | A10[3:0] | A03[3:0] | A02[3:0] | A01[3:0] | A00[3:0] |

**DXT4/DXT5**

127 Byte15 120119 Byte14 112111 Byte13 104103 Byte12 96

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T33 | T32 | T31 | T30 | T23 | T22 | T21 | T20 | T13 | T12 | T11 | T10 | T03 | T02 | T01 | T00 |

95 Byte11 88 87 Byte10 80 79 Byte9 72 71 Byte8 64

| | | | | | |
|---|---|---|---|---|---|
| R1[4:0] | G1[5:0] | B1[4:0] | R0[4:0] | G0[5:0] | B0[4:0] |

63 Byte7 56 55 Byte6 48 47 Byte5 40 39 Byte4 32

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| T33 | T32 | T31 | T30 | T23 | T22 | T21 | T20 | T13 | T12 | T11 |

31 Byte3 24 23 Byte2 16 15 Byte1 8 7 Byte0 0

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | T10 | T03 | T02 | T01 | T00 | A1[7:0] | A0[7:0] |

---

Each value in DXT1 format is treated as a 64-bit pixel that decodes to 4x4 texels. Each value in DXT2 to DXT5 format is treated as a 128-bit pixel that decodes to 4x4 texels. In linear format, N sequential DXTC pixels decode to a 4Nx4 region of texels. In tiled format, 64 sequential DCTC pixels form an 8x8 tile, which decodes to a 32x32 region of texels.

**{Include the depth and depth/stencil formats.}**

**{Describe the following formats:}**

| |
|---|
| 1 (1D only) |
| 1_REVERSE (1D only) |
| |

| |
|---|
| 16_MPEG |
| 16_16_MPEG |
| 8_INTERLACED |
| 16_INTERLACED (fixed)<br>16_INTERLACED (float)<br>16_INTERLACED (expand) |
| 32_AS_8_INTERLACED |
| 32_AS_8 |

## 2.4 Special Data Formats

This subsection describes arrayable data elements that have specific, limited purposes.

The Render Backend uses an array of Tile Data words, which store 32-bits for each 8x8 pixel tile. The Tile Data word stores compression and hierarchical information for each tile. The figure below illustrate the Tile Data word. The Cmask field stores the compression format for the Color0 buffer. The Zmask field stores the compression format for depth data in the Depth/Stencil buffer. The Smask field stores the compression format and hierarchical data for the stencil data in the Depth/Stencil buffer. Finally, the Zrange field encodes bounds on the minimum and maximum depth values in the tile for hierarchical depth kills.

| 31 12 | 11 8 | 7 4 | 3 0 |
|---|---|---|---|
| Zrange<19:0> | Smask<3:0> | Zmask<3:0> | Cmask<3:0> |

Figure 6: 32-Bit Tile Data Word for Render Backend

**{Document the depth formats}** Each16-bit pixel consists of a 16-bit repeating fraction depth value, which represents the range [0..1]. Each 32-bit pixel represents an 8-bit stencil value in the low order byte and a 24-bit depth value in the high bytes. The depth is either a 24-bit repeating fraction or a floating point representation with a 4-bit exponent, which represents the range [0..2), though values greater than 1 are not allowed.

{This includes all of the uncompressed formats that are not destination color formats, including bitmap formats, YUV formats, uncompressed depth/stencil values, etc.}

# 3. 1D Tiled Memory Formats

This section describes tiled memory formats for 1D arrays. In system memory, there is no difference between 1D tiled format and linear format. In local memory, the tiling format describes how the 1D data is interleaved across the memory subsets. For Local Tiled and System Tiled mode, the memory allocated for a 1D array must start and end on a 4Kbyte boundary. For System Linear mode, the array must start and end on a 64-byte boundary.

## 3.1 1D Micro-Tile Formats

The figure below shows 1D micro-tile formats within a 64-byte tile for 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit data elements. In each case, the tile size is 64-bytes (512-bits) or four 16-byte (128-bit) micro-tiles. Each row in the figure below is a single micro-tile.

| 127 | | | 96 | 95 | | | 64 | 63 | | | 32 | 31 | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D 15 | D 14 | D 13 | D 12 | D 11 | D 10 | D 9 | D 8 | D 7 | D 6 | D 5 | D 4 | D 3 | D 2 | D 1 | D 0 |
| D 31 | D 30 | D 29 | D 28 | D 27 | D 26 | D 25 | D 24 | D 23 | D 22 | D 21 | D 20 | D 19 | D 18 | D 17 | D 16 |
| D 47 | D 46 | D 45 | D 44 | D 43 | D 42 | D 41 | D 40 | D 39 | D 38 | D 37 | D 36 | D 35 | D 34 | D 33 | D 32 |
| D 63 | D 62 | D 61 | D 60 | D 59 | D 58 | D 57 | D 56 | D 55 | D 54 | D 53 | D 52 | D 51 | D 50 | D 49 | D 48 |

8-bit data

| 127 | 96 | 95 | 64 | 63 | 32 | 31 | 0 |
|---|---|---|---|---|---|---|---|
| D 7 | D 6 | D 5 | D 4 | D 3 | D 2 | D 1 | D 0 |
| D 15 | D 14 | D 13 | D 12 | D 11 | D 10 | D 9 | D 8 |
| D 23 | D 22 | D 21 | D 20 | D 19 | D 18 | D 17 | D 16 |
| D 31 | D 30 | D 29 | D 28 | D 27 | D 26 | D 25 | D 24 |

16-bit data

| 127 96 | 95 64 | 63 32 | 31 0 |
|---|---|---|---|
| Data 3 | Data 2 | Data 1 | Data 0 |
| Data 7 | Data 6 | Data 5 | Data 4 |
| Data 11 | Data 10 | Data 9 | Data 8 |
| Data 15 | Data 14 | Data 13 | Data 12 |

32-bit data

| 127 96 95 64 | 63 32 31 0 |
|---|---|
| Data 1 | Data 0 |
| Data 3 | Data 2 |
| Data 5 | Data 4 |
| Data 7 | Data 6 |

64-bit data

| 127 96 95 64 63 32 31 0 |
|---|
| Data 0 |
| Data 1 |
| Data 2 |
| Data 3 |

128-bit data

**Figure 7: 1D Micro-Tile Data Formats**

The texture unit also supports packed 1-bit pixel formats for use as text fonts. {Draw a figure describing these formats, including the bit order.}

to be specified

**Figure 8: 1D Micro-Tile 1-bit Data Formats**

Finally, note that bytes are stored in local memory packed from lsb to msb of successive data elements, that is, little-endian order. Three byte swap reorderings are supported for data stored in system memory. R400 automatically converts between the different byte swap modes, based on a field that is stored with the offset for each surface.

## 3.2 1D Macro-Tile Formats

The following figure shows the organization of tiles within a macro-tile for 1D arrays. Each 1D macro-tile occupies 4K-bytes. The left-hand figure shows the organization in system memory, which is simply a linear sequence of 64 64-byte tiles. The column in the center of the figure gives the byte address relative to to the start of the macro-tile, for each tile.

| system memory | one MC with two memory subsets | | two MCs with four memory subsets | four MCs with eight memory subsets |
|---|---|---|---|---|
| tile 0 | ab, tile 0 | 0x000 | ab0, tile 0 | ab0, tile 0 |
| tile 1 | ab, tile 1 | 0x040 | ab1, tile 0 | ab1, tile 0 |
| tile 2 | ab, tile 2 | 0x080 | ab0, tile 1 | ab2, tile 0 |
| tile 3 | ab, tile 3 | 0x0C0 | ab1, tile 1 | ab3, tile 0 |
| . . . | . . . | . . . | . . . | . . . |
| tile 30 | ab, tile 30 | 0x780 | ab0, tile 15 | ab2, tile 7 |
| tile 31 | ab, tile 31 | 0x7C0 | ab1, tile 15 | ab3, tile 7 |
| tile 32 | cd, tile 0 | 0x800 | cd0, tile 0 | cd0, tile 0 |
| tile 33 | cd, tile 1 | 0x840 | cd1, tile 0 | cd1, tile 0 |
| tile 34 | cd, tile 2 | 0x880 | cd0, tile 1 | cd2, tile 0 |
| tile 35 | cd, tile 3 | 0x8C0 | cd1, tile 1 | cd3, tile 0 |
| . . . | . . . | . . . | . . . | . . . |
| tile 62 | cd, tile 30 | 0xF80 | cd0, tile 15 | cd2, tile 7 |
| tile 63 | cd, tile 31 | 0xFC0 | cd1, tile 15 | cd3, tile 7 |

**Figure 9: 1D Macro-Tile Formats**

The remainder of the above figure shows the arrangement of micro-tiles in local memory for 2-8 memory subsets (1-4 memory controllers). Tiles are numbered separately within each memory subset. These formats spread sequential array accesses evenly across the memory controllers at a relatively fine granularity. Within a memory controller, these formats produce a burst within one memory subset before switching to the other memory subset. The goal is to produce a large enough burst within a bank to cover the time required to open a page in a different bank, while keeping the bursts small enough to reduce the buffering required to spread sequential accesses across all of the memory controllers.

The linear array form can also be used for 2D or 3D arrays. A 2D or 3D linear arrays is first converted to a 1D array by computing *Index = X + Y*Pitch + Z*Height*Pitch*. R400 can read texture maps from 2D and 3D linear arrays and can write to them for bitblts. R400 does not support rendering (alpha blending and/or depth buffering) to 2D or 3D arrays that are stored in linear format.

## 3.3 1D Address Equations

This subsection presents equations for computing addresses in a 1D array. These are also used in computing 2D and 3D array addresses (subsections 4.4 and 5.3). These equations are also implemented in address conversion library code (address.h and address.c). Boldface represents names of parameters that are used in the C library.

The first list below defines parameters that are constant for a given surface. *Size* and *Subsets* may be derived from ***DataSize*** and ***Pipes***, but are defined separately to simplify the equations. ***SurfaceBase*** is the byte address of the start of the surface, which must be 4K-byte aligned. ***SurfaceBase*** may be expressed relative to the start of the entire $2^{32}$-byte device address space or within a subrange of the complete device address space, provided that the subrange is also 4K-byte aligned.

| | |
|---|---|
| Size | Bytes per pixel: can be 1, 2, 4, 8, or 16 (or fractions of a byte for non-pixel data) |
| **DataSize** | 64 times Size, equals the total bytes of data in a 2D or 3D tile |
| **Pipes** | Total number of Render Backend/Memory Controller pipelines: 1, 2, or 4 |
| Subsets | Total number of memory subsets: equals twice the number of pipelines |
| **SurfaceBase** | Byte address of pixel zero in device address space or subrange, must be 4K-byte aligned |

The second list below names parameters that depend on which pixel is accessed in the 1D array. 1D address equations use the parameters in the first list and one or more of the parameters in the second list to define the remaining parameters in the second list. *MemSelect* and *BankSelect* may be derived from ***Subset***, or vice versa. *MacroNumber*, *TileNumber*, and *TileAddr* are temporary values used in computing the other parameters.

| | |
|---|---|
| **Index** | Pixel index into the array |
| Byte**Addr** | Byte address of the pixel in device address space (or a 4K-byte aligned subrange) |
| **Local**Addr | Byte address of the pixel within its memory subset, starting at byte 0 in the address range |
| MemSelect | Number of the memory controller that stores this pixel |
| BankSelect | 0 for banks AB or 1 for banks CD, together with MemSelect determines the memory subset |
| **Subset** | Subset number, which equals BankSelect + 2*MemSelect |
| MacroNumber | Sequential number of the macro-tile containing the pixel, starting from device address 0 |
| TileNumber | Sequential number of the tile containing the pixel, within its memory subset and its macro-tile |
| TileAddr | Byte address of the pixel within its tile, which is entirely contained in a single memory subset |

The following equations use ***Index*** to compute the other address terms, particularly *Byte**Addr***. This is used to convert an array access into a device address. Typically ***LocalAddr*** is not required as part of this step, but it is included for completeness.

```
TileAddr     = (Index*Size) mod 64;                 // 64 bytes per tile
TileNumber   = ((Index*Size/64) mod 32)  / Pipes;   // cycle through MCs within each half-macro-tile
MacroNumber  = (Index*Size + SurfaceBase) / 4096;   // 4096 bytes per macro-tile
MemSelect    = (Index*Size/64) mod Pipes;           // cycle through MCs each 64 bytes
BankSelect   = (Index*Size/2048) mod 2;             // CD banks are in high half of each macro-tile
Subset       = BankSelect + 2*MemSelect;            // subset number
ByteAddr     = SurfaceBase + Index*Size;
LocalAddr    = MacroNumber*4096/Subsets + TileAddr + TileNumber*64;
```

The following equations use *Byte**Addr*** to compute the other address terms, particularly ***LocalAddr*** and ***Subset***. This is used to convert a device address into a local memory address within a particular memory subset. Typically ***Index*** is not required as part of this step, but it is included for completeness.

```
TileAddr     = ByteAddr mod 64;                     // 64 bytes per tile
TileNumber   = ((ByteAddr/64) mod 32) / Pipes;      // cycle through MCs within each half-macro-tile
MacroNumber  = (ByteAddr) / 4096;                   // 4096 bytes per macro-tile
```

MemSelect = (ByteAddr/64) mod Pipes; // cycle through MCs each 64 bytes
BankSelect = (ByteAddr/2048) mod 2; // CD banks are in high half of each macro-tile
**Subset** = BankSelect + 2*MemSelect; // subset number
**Index** = (ByteAddr – SurfaceBase) / Size;
**Local**Addr = MacroNumber*4096/Subsets + TileAddr + TileNumber*64;

The final set of equations use ***LocalAddr*** and ***Subset*** to compute the other address terms, particularly *Byte**Addr***. The equations for 2D and 3D arrays use (*X*,*Y*) addresses to produce ***LocalAddr*** and ***Subset***, which these equations convert into device addresses. Typically only *Byte**Addr*** is required as the result of this step but the others are provided for completeness.

MemSelect = Subset / 2;
BankSelect = Subset mod 2;
TileAddr = LocalAddr mod 64; // 64 bytes per tile
TileNumber = (LocalAddr /64) mod (64/Subsets); // 64 tiles in a macro-tile, over all the subsets
MacroNumber = LocalAddr * Subsets / 4096; // 4096 bytes per macro-tile
Byte**Addr** = 4096*MacroNumber+2048*BankSelect+32*Subsets*TileNumber +64*MemSelect+TileAddr;
**Index** = (ByteAddr – SurfaceBase) / Size;

# 4. 2D Tiled Memory Formats

This section describes how the R400 family stores tiled 2D data arrays. Each tile contains an 8x8 array of data elements. Each 8x8 tile has a micro-tile format that depends on the data element size. Each 2D macro-tile contains a 4x4 array of tiles, which covers 32x32 pixels. This is different from 1D formats, where each tile and macro-tile stores a fixed number of bytes. Like the 1D formats, each 2D tiled surface must start on a 4K-byte boundary.

## 4.1 2D Micro-Tile Formats

R400 arranges pixels within 8x8 tiles in order to meet two conflicting goals. First, sequential accesses from memory should contain pixels from a roughly square region within the tile. This improves efficiency by a modest amount, due to rendering locality. Second, display updates must be efficient with only one line buffer. This implies that each 256-bit memory access should contain pixels from only two scanlines.

To meet these goals, R400 stores even scanlines of the 8x8 tile in even-numbered micro-tiles and stores odd scanlines in odd-numbered micro-tiles. The figure below shows the order of micro-tiles within an 8x8 tile for the five pixel sizes. For 128-bit pixels, each micro-tile covers a single pixel. For 8-bit pixels, each 256-bit access includes pixels from four different scanlines. This requires display accesses to throw away half of the 8-bit data that it reads, but this loss of efficiency is acceptable for 8-bit pixels. For all other pixel sizes, a 256-bit access reads from just two scanlines.

| |
|---|
| Micro-Tile 0 <63:0> |
| Micro-Tile 1 <63:0> |
| Micro-Tile 0 <127:64> |
| Micro-Tile 1 <127:64> |
| Micro-Tile 2 <63:0> |
| Micro-Tile 3 <63:0> |
| Micro-Tile 2 <127:64> |
| Micro-Tile 3 <127:64> |

8-bit Pixels

| |
|---|
| Micro-Tile 0 |
| Micro-Tile 1 |
| Micro-Tile 2 |
| Micro-Tile 3 |
| Micro-Tile 4 |
| Micro-Tile 5 |
| Micro-Tile 6 |
| Micro-Tile 7 |

16-bit Pixels

| | |
|---|---|
| MT 0 | MT 2 |
| MT 1 | MT 3 |
| MT 4 | MT 6 |
| MT 5 | MT 7 |
| MT 8 | MT 10 |
| MT 9 | MT 11 |
| MT 12 | MT 14 |
| MT 13 | MT 15 |

32-bit Pixels

| | | | |
|---|---|---|---|
| 0 | 2 | 4 | 6 |
| 1 | 3 | 5 | 7 |
| 8 | 10 | 12 | 14 |
| 9 | 11 | 13 | 15 |
| 16 | 18 | 20 | 22 |
| 17 | 19 | 21 | 23 |
| 24 | 26 | 28 | 30 |
| 25 | 27 | 29 | 31 |

64-bit Pixels

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 |
| 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
| 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
| 17 | 19 | 21 | 23 | 25 | 27 | 29 | 31 |
| 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 |
| 33 | 35 | 37 | 39 | 41 | 43 | 45 | 47 |
| 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 |
| 49 | 51 | 53 | 55 | 57 | 59 | 61 | 63 |

128-bit Pixels

**Figure 10: 2D Micro-tile Layout Within Tiles**

For 32-bit and larger pixels, the patterns above map each 256-bit access to a 1:1 or 2:1 region within the tile, therefore meeting the square region criterion. 16-bit pixels have a less efficient 4:1 region, but this is acceptable since smaller pixels require less memory bandwidth. Smaller pixels are also less important in the R400 timeframe. This micro-tile format is efficient enough that it is used for all uncompressed 2D pixel and texel arrays, even though texels do not need to be displayed to the screen.

The figure below shows the (x,y) address of the pixels or texels inside the 16-byte (128-bit) micro-tiles. Only the first two micro-tiles are shown for each pixel size. Except for 8-bit pixels, these micro-tiles only include data from the first two rows of the tile. For 8-bit pixels, they cover the first four rows of the 8x8 tile.

one micro-tile is 128 bits, as signified by the bold outline for each micro-tile

| | 127:120 | 119:112 | 111:104 | 103:96 | 95:88 | 87:80 | 79:72 | 71:64 | 63:56 | 55:48 | 47:40 | 39:32 | 31:24 | 23:16 | 15:8 | 7:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8-bit data | (7,2) | (6,2) | (5,2) | (4,2) | (3,2) | (2,2) | (1,2) | (0,2) | (7,0) | (6,0) | (5,0) | (4,0) | (3,0) | (2,0) | (1,0) | (0,0) |
| | (7,3) | (6,3) | (5,3) | (4,3) | (3,3) | (2,3) | (1,3) | (0,3) | (7,1) | (6,1) | (5,1) | (4,1) | (3,1) | (2,1) | (1,1) | (0,1) |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 16-bit data | Pixel (7,0) | Pixel (6,0) | Pixel (5,0) | Pixel (4,0) | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) |
| | Pixel (7,1) | Pixel (6,1) | Pixel (5,1) | Pixel (4,1) | Pixel (3,1) | Pixel (2,1) | Pixel (1,1) | Pixel (0,1) |

| | | | | |
|---|---|---|---|---|
| 32-bit data | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) |
| | Pixel (3,1) | Pixel 2,1) | Pixel (1,1) | Pixel (0,1) |

| | | |
|---|---|---|
| 64-bit data | Pixel (1,0) | Pixel (0,0) |
| | Pixel (1,1) | Pixel (0,1) |

| | |
|---|---|
| 128-bit data | Pixel (0,0) |
| | Pixel (0,1) |

**Figure 11: Pixel Format within 2D Micro-Tiles**

## 4.2 2D Macro-Tile Formats

Each 2D macro-tile stores a 32x32 array of pixels, organized into 16 8x8 tiles. The macro-tile format depends on the number of memory subsets, which is twice the number of memory controllers. The following figure shows the layout of 8x8 tiles within 32x32 macro-tiles for 1, 2, and 4 memory controllers. Bold lines mark 32x32 macro-tiles. Light lines mark 8x8 tiles. The upper line of text in each 8x8 tile specifies the memory subset and the lower line of text specifies the order of the tiles within their memory subset.

The three macro-tile formats have several properties in common. First, each macro-tile allocates an equal number of 8x8 tiles to each memory subset, which makes it simpler to allocate memory. Second, tile addresses in memory increase from left to right within each macro-tile and between macro-tiles on the same scanline. Finally, moving vertically by one macro-tile increments the tile address by a value *L*, which is equal to the pitch (line length) in pixels, divided by four times the number of memory controllers. The pitch must be a multiple of 32 pixels.

one MC with two memory subsets

| | | | |
|---|---|---|---|
| ab 0 | ab 2 | ab 4 | ab 6 |
| ab 1 | ab 3 | ab 5 | ab 7 |
| cd 0 | cd 2 | cd 4 | cd 6 |
| cd 1 | cd 3 | cd 5 | cd 7 |
| ab L | ab 2+L | ab 4+L | ab 6+L |
| ab 1+L | ab 3+L | ab 5+L | ab 7+L |
| cd L | cd 2+L | cd 4+L | cd 6+L |
| cd 1+L | cd 3+L | cd 5+L | cd 7+L |

two MCs with four memory subsets

| | | | |
|---|---|---|---|
| ab0 0 | ab1 0 | ab0 2 | ab1 2 |
| ab1 1 | ab0 1 | ab1 3 | ab0 3 |
| cd0 0 | cd1 0 | cd0 2 | cd1 2 |
| cd1 1 | cd0 1 | cd1 3 | cd0 3 |
| ab0 L | ab1 L | ab0 2+L | ab1 2+L |
| ab1 1+L | ab0 1+L | ab1 3+L | ab0 3+L |
| cd0 L | cd1 L | cd0 2+L | cd1 2+L |
| cd1 1+L | cd0 1+L | cd1 3+L | cd0 3+L |

four MCs with eight memory subsets

| | | | |
|---|---|---|---|
| ab0 0 | ab1 0 | ab2 0 | ab3 0 |
| ab2 1 | ab3 1 | ab0 1 | ab1 1 |
| cd0 0 | cd1 0 | cd2 0 | cd3 0 |
| cd2 1 | cd3 1 | cd0 1 | cd1 1 |
| ab0 L | ab1 L | ab2 L | ab3 L |
| ab2 1+L | ab3 1+L | ab0 1+L | ab1 1+L |
| cd0 L | cd1 L | cd2 L | cd3 L |
| cd2 1+L | cd3 1+L | cd0 1+L | cd1 1+L |

**Figure 12: 2D Macro-Tile Mappings**

Another common property is that if there are N memory controllers, then each Nx1 row of tiles places a tile in each memory controller. This is necessary for efficient display accesses. The display reads across rows of 8x8 tiles and sometimes requires a significant fraction of the total memory bandwidth. Alternating between the memory controllers allows the display to spread its bandwidth equally between them. This also makes it more efficient to render large primitives, since the Scan Converter steps horizontally before it steps vertically. The bank alternation within a memory subset ensures that page crossings do not occur while rendering horizontal swaths of pixels.

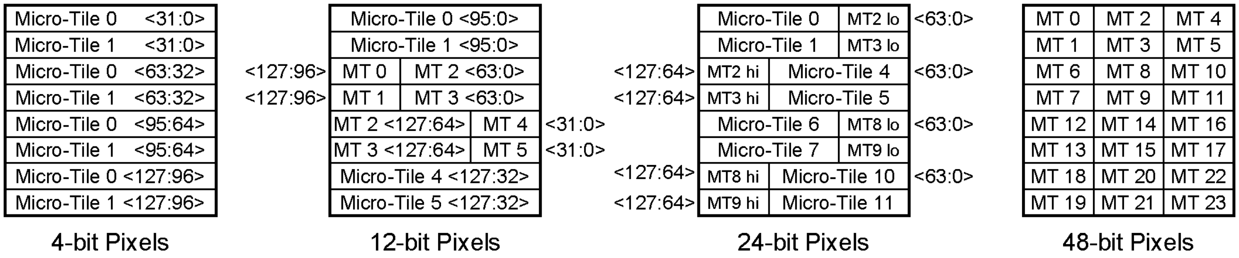
The remaining property that the macro-tile formats have in common is that the upper half of each macro-tile uses bank AB memory subsets and the lower half uses bank CD memory subsets. This reduces page crossings when rendering vertical swaths of pixels. A vertical line first touches two tiles in AB subsets, followed by two tiles in CD subsets. The next tile is once again in an AB subset and could be on a different page of the same bank as the initial accesses. Interspersing the CD subset accesses makes it more likely that the Memory Controller will have accesses to perform while waiting for the bank to become ready. However, vertical motion is not as efficient as horizontal motion in these macro-tile formats.

Finally, note that the first memory controller on odd rows of 8x8 tiles is offset by 1 for two memory controllers and is offset by two for four memory controllers. This has the effect that each 2x2 block of tiles hits each memory controller the same number of times. Putting together all of these properties, the 8x8 tiles nearest to any tile that are in the same memory controller are either on the same page of the same bank or are in a different bank. Further, with two or four memory controllers, moving horizontally or vertically to an adjacent tile also moves to a different memory controller.

## 4.3 Special 2D Micro-Tile Formats

The previous section describes micro-tile formats for standard pixel sizes. The Render Backend requires several additional pixel sizes for depth, stencil, and multifragment mask data. These pixels require special micro-tile formats that are only read and written by the Render Backend. Additionally, the Render Backend requires a micro-tile format that stores a single data element per tile, instead of a data element per pixel.

The following figure illustrates the micro-tile packing formats for non-standard pixel sizes. Like the standard micro-tile formats, these formats put even scanlines into even micro-tiles and odd scanlines into odd micro-tiles. The smallest allowed pixel size is 4-bits, since at that size an entire tile occupies one even and one odd micro-tile. The 4-bit and 12-bit formats do not permit byte masking of individual pixels. All but the 4-bit format cause pixels to cross micro-tile boundaries. All but the 48-bit format cause micro-tiles to touch multiple scanlines (as does the 8-bit micro-tile format). For these reasons, surfaces that use these formats are not readable or writable by software through the Memory Hub. They are only read and written by the Render Backend.



4-bit Pixels

| Micro-Tile |
|---|
| Micro-Tile 0 <31:0> |
| Micro-Tile 1 <31:0> |
| Micro-Tile 0 <63:32> |
| Micro-Tile 1 <63:32> |
| Micro-Tile 0 <95:64> |
| Micro-Tile 1 <95:64> |
| Micro-Tile 0 <127:96> |
| Micro-Tile 1 <127:96> |

12-bit Pixels

| | | | |
|---|---|---|---|
| | Micro-Tile 0 <95:0> | | |
| | Micro-Tile 1 <95:0> | | |
| <127:96> | MT 0 | MT 2 <63:0> | |
| <127:96> | MT 1 | MT 3 <63:0> | |
| | MT 2 <127:64> | MT 4 | <31:0> |
| | MT 3 <127:64> | MT 5 | <31:0> |
| | Micro-Tile 4 <127:32> | | |
| | Micro-Tile 5 <127:32> | | |

24-bit Pixels

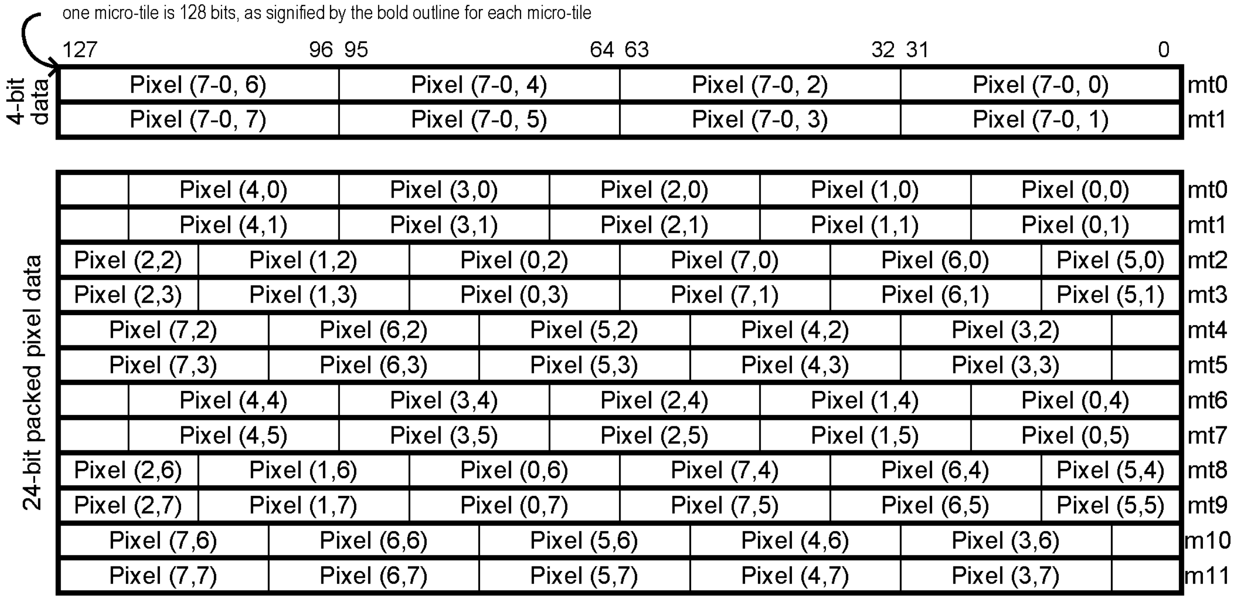
| | | | |
|---|---|---|---|
| | Micro-Tile 0 | MT2 lo | <63:0> |
| | Micro-Tile 1 | MT3 lo | |
| <127:64> | MT2 hi | Micro-Tile 4 | <63:0> |
| <127:64> | MT3 hi | Micro-Tile 5 | |
| | Micro-Tile 6 | MT8 lo | <63:0> |
| | Micro-Tile 7 | MT9 lo | |
| <127:64> | MT8 hi | Micro-Tile 10 | <63:0> |
| <127:64> | MT9 hi | Micro-Tile 11 | |

48-bit Pixels

| | | |
|---|---|---|
| MT 0 | MT 2 | MT 4 |
| MT 1 | MT 3 | MT 5 |
| MT 6 | MT 8 | MT 10 |
| MT 7 | MT 9 | MT 11 |
| MT 12 | MT 14 | MT 16 |
| MT 13 | MT 15 | MT 17 |
| MT 18 | MT 20 | MT 22 |
| MT 19 | MT 21 | MT 23 |

**Figure 13: 2D Micro-tiling for Nonstandard Pixels**

The figure below shows the (x,y) address of the pixels or texels inside the 16-byte (128-bit) micro-tiles for the 4-bit and 24-bit pixel sizes. Each row specifies a different micro-tile. 4-bit data occupies just two micro-tiles. 24-bit data requires 12 micro-tiles, with some pixels splitting across micro-tile boundaries. 12-bit pixels and 48-bit pixels require 6 and 24 micro-tiles, respectively.



one micro-tile is 128 bits, as signified by the bold outline for each micro-tile

4-bit data

| 127 – 96 | 95 – 64 | 63 – 32 | 31 – 0 | |
|---|---|---|---|---|
| Pixel (7-0, 6) | Pixel (7-0, 4) | Pixel (7-0, 2) | Pixel (7-0, 0) | mt0 |
| Pixel (7-0, 7) | Pixel (7-0, 5) | Pixel (7-0, 3) | Pixel (7-0, 1) | mt1 |

24-bit packed pixel data

| | | | | | | |
|---|---|---|---|---|---|---|
| | Pixel (4,0) | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) | mt0 |
| | Pixel (4,1) | Pixel (3,1) | Pixel (2,1) | Pixel (1,1) | Pixel (0,1) | mt1 |
| Pixel (2,2) | Pixel (1,2) | Pixel (0,2) | Pixel (7,0) | Pixel (6,0) | Pixel (5,0) | mt2 |
| Pixel (2,3) | Pixel (1,3) | Pixel (0,3) | Pixel (7,1) | Pixel (6,1) | Pixel (5,1) | mt3 |
| Pixel (7,2) | Pixel (6,2) | Pixel (5,2) | Pixel (4,2) | Pixel (3,2) | | mt4 |
| Pixel (7,3) | Pixel (6,3) | Pixel (5,3) | Pixel (4,3) | Pixel (3,3) | | mt5 |
| | Pixel (4,4) | Pixel (3,4) | Pixel (2,4) | Pixel (1,4) | Pixel (0,4) | mt6 |
| | Pixel (4,5) | Pixel (3,5) | Pixel (2,5) | Pixel (1,5) | Pixel (0,5) | mt7 |
| Pixel (2,6) | Pixel (1,6) | Pixel (0,6) | Pixel (7,4) | Pixel (6,4) | Pixel (5,4) | mt8 |
| Pixel (2,7) | Pixel (1,7) | Pixel (0,7) | Pixel (7,5) | Pixel (6,5) | Pixel (5,5) | mt9 |
| Pixel (7,6) | Pixel (6,6) | Pixel (5,6) | Pixel (4,6) | Pixel (3,6) | | m10 |
| Pixel (7,7) | Pixel (6,7) | Pixel (5,7) | Pixel (4,7) | Pixel (3,7) | | m11 |

**Figure 14: 4-bit and 24-bit Micro-Tile Formats**

Finally, data that is stored on a per-tile basis is not micro-tiled at all. Instead, each tile simply stores a specified number of bits. The Render Backend stores 32-bits per tile to record the tile's compression. In this case, the macro-tile format is exactly the same as described in the previous section, except that there may be fewer than 512-bits per tile. {**Note**: say a lot more about this.}

one MC with two memory subsets

| | | | |
|---|---|---|---|
| ab 0.0 | ab 0.2 | ab 1.0 | ab 1.2 |
| ab 0.1 | ab 0.3 | ab 1.1 | ab 1.3 |
| cd 0.0 | cd 0.2 | cd 1.0 | cd 1.2 |
| cd 0.1 | cd 0.3 | cd 1.1 | cd 1.3 |

two MCs with four memory subsets

| | | | |
|---|---|---|---|
| ab0 0.0 | ab1 0.0 | ab0 0.2 | ab1 0.2 |
| ab1 0.1 | ab0 0.1 | ab1 0.3 | ab0 0.3 |
| cd0 0.0 | cd1 0.0 | cd0 0.2 | cd1 0.2 |
| cd1 0.1 | cd0 0.1 | cd1 0.3 | cd0 0.3 |

four MCs with eight memory subsets

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ab0 0.0 | ab1 0.0 | ab2 0.0 | ab3 0.0 | ab0 0.2 | ab1 0.2 | ab2 0.2 | ab3 0.2 |
| ab2 0.1 | ab3 0.1 | ab0 0.1 | ab1 0.1 | ab2 0.3 | ab3 0.3 | ab0 0.3 | ab1 0.3 |
| cd0 0.0 | cd1 0.0 | cd2 0.0 | cd3 0.0 | cd0 0.2 | cd1 0.2 | cd2 0.2 | cd3 0.2 |
| cd2 0.1 | cd3 0.1 | cd0 0.1 | cd1 0.1 | cd2 0.3 | cd3 0.3 | cd0 0.3 | cd1 0.3 |

**Figure 15: 2D Macro-Tiling for 32-Bit Per-Tile Data**

## 4.4 Alternate 2D Macro-Tile Formats

There are three variations of the standard 2D macro-tile formats. These variations exist to improve performance for depth buffering and to allow a 2D depth buffer to be used in conjunction with a slice from a 3D color buffer. R400 does not support these alternate for display buffers, but they may be used for rendering, memory apertures, and texture mapping . {Check whether the 3D slice format actually gets supported for texture maps.} As with the figures in section , these figures show a box per 8x8 tile, with the upper line of text in each box listing the banks and RB number and the lower line of text giving the order in which the tiles are stored within their memory subset.

The figure below shows an alternate 2D tiling pattern that swaps the AB and CD bank assignment relative to the standard pattern that is described in section 4.2. This macro-tile pattern is particularly appropriate for depth buffers. If the same 2D tiling is used for both the depth buffer and the color buffer, then large area operations will tend to cause the Render Backend to read and write both of them in the AB banks or both of them in the CD banks. Using the bank-swapped alternate tiling illustrated below for the depth buffer increases the number of different banks that are likely to be open at the same time, thus increasing memory efficiency.

one MC with two memory subsets

| | | | |
|---|---|---|---|
| cd<br>0 | cd<br>2 | cd<br>4 | cd<br>6 |
| cd<br>1 | cd<br>3 | cd<br>5 | cd<br>7 |
| ab<br>0 | ab<br>2 | ab<br>4 | ab<br>6 |
| ab<br>1 | ab<br>3 | ab<br>5 | ab<br>7 |
| cd<br>L | cd<br>2+L | cd<br>4+L | cd<br>6+L |
| cd<br>1+L | cd<br>3+L | cd<br>5+L | cd<br>7+L |
| ab<br>L | ab<br>2+L | ab<br>4+L | ab<br>6+L |
| ab<br>1+L | ab<br>3+L | ab<br>5+L | ab<br>7+L |

two MCs with four memory subsets

| | | | |
|---|---|---|---|
| cd0<br>0 | cd1<br>0 | cd0<br>2 | cd1<br>2 |
| cd1<br>1 | cd0<br>1 | cd1<br>3 | cd0<br>3 |
| ab0<br>0 | ab1<br>0 | ab0<br>2 | ab1<br>2 |
| ab1<br>1 | ab0<br>1 | ab1<br>3 | ab0<br>3 |
| cd0<br>L | cd1<br>L | cd0<br>2+L | cd1<br>2+L |
| cd1<br>1+L | cd0<br>1+L | cd1<br>3+L | cd0<br>3+L |
| ab0<br>L | ab1<br>L | ab0<br>2+L | ab1<br>2+L |
| ab1<br>1+L | ab0<br>1+L | ab1<br>3+L | ab0<br>3+L |

four MCs with eight memory subsets

| | | | |
|---|---|---|---|
| cd0<br>0 | cd1<br>0 | cd2<br>0 | cd3<br>0 |
| cd2<br>1 | cd3<br>1 | cd0<br>1 | cd1<br>1 |
| ab0<br>0 | ab1<br>0 | ab2<br>0 | ab3<br>0 |
| ab2<br>1 | ab3<br>1 | ab0<br>1 | ab1<br>1 |
| cd0<br>L | cd1<br>L | cd2<br>L | cd3<br>L |
| cd2<br>1+L | cd3<br>1+L | cd0<br>1+L | cd1<br>1+L |
| ab0<br>L | ab1<br>L | ab2<br>L | ab3<br>L |
| ab2<br>1+L | ab3<br>1+L | ab0<br>1+L | ab1<br>1+L |

**Figure 16: Bank-Swapped 2D Macro-Tile Mappings**

A 2D depth buffer may be used with a single slice of a 3D color buffer. This requires a 2D tiling pattern that maps each pixel to the same RB that it is mapped to in the 3D tiling pattern. Section 5.2 describes 3D macro-tiling patterns, which map pixel in an (x,y) column to one of two RBs, depending on the value of Z. One of the two RB assignments matches the RB assignments in the standard 2D macro-tiling pattern. The other RB assignment swaps the RB numbers. The swapped 2D macro-tiling pattern below matches this alternate RB mapping for 3D slices. A 2D depth surface may be used with slices of a 3D color array by selecting either the standard or this swapped macro-tiling pattern, depending on the slice selected from the 3D array.

one MC with two memory subsets

| | | | |
|---|---|---|---|
| ab 0 | ab 2 | ab 4 | ab 6 |
| ab 1 | ab 3 | ab 5 | ab 7 |
| cd 0 | cd 2 | cd 4 | cd 6 |
| cd 1 | cd 3 | cd 5 | cd 7 |
| ab L | ab 2+L | ab 4+L | ab 6+L |
| ab 1+L | ab 3+L | ab 5+L | ab 7+L |
| cd L | cd 2+L | cd 4+L | cd 6+L |
| cd 1+L | cd 3+L | cd 5+L | cd 7+L |

two MCs with four memory subsets

| | | | |
|---|---|---|---|
| ab1 0 | ab0 0 | ab1 2 | ab0 2 |
| ab0 1 | ab1 1 | ab0 3 | ab1 3 |
| cd1 0 | cd0 0 | cd1 2 | cd0 2 |
| cd0 1 | cd1 1 | cd0 3 | cd1 3 |
| ab1 L | ab0 L | ab1 2+L | ab0 2+L |
| ab0 1+L | ab1 1+L | ab0 3+L | ab1 3+L |
| cd1 L | cd0 L | cd1 2+L | cd0 2+L |
| cd0 1+L | cd1 1+L | cd0 3+L | cd1 3+L |

four MCs with eight memory subsets

| | | | |
|---|---|---|---|
| ab2 0 | ab3 0 | ab0 0 | ab1 0 |
| ab0 1 | ab1 1 | ab2 1 | ab3 1 |
| cd2 0 | cd3 0 | cd0 0 | cd1 0 |
| cd0 1 | cd1 1 | cd2 1 | cd3 1 |
| ab2 L | ab3 L | ab0 L | ab1 L |
| ab0 1+L | ab1 1+L | ab2 1+L | ab3 1+L |
| cd2 L | cd3 L | cd0 L | cd1 L |
| cd0 1+L | cd1 1+L | cd2 1+L | cd3 1+L |

**Figure 17: RB-Swapped 2D Macro-Tile Mappings**

The final new macro-tiling pattern is a combination of the preceeding two. The macro-tilings illustrated above and below allow selecting a 2D pattern that either does or does not swap the AB and CD banks relative to the 3D slices that do not match the standard 2D macro-tiling. The 3D macro-tiling described in section 5.2 matches the pattern below because it swaps both RBs and banks. So the pattern above, that swaps just the RBs, should be used to cause a 2D depth surface to use a different bank for each tile than the 2D tile pattern uses.

one MC with two memory subsets

| | | | |
|---|---|---|---|
| cd 0 | cd 2 | cd 4 | cd 6 |
| cd 1 | cd 3 | cd 5 | cd 7 |
| ab 0 | ab 2 | ab 4 | ab 6 |
| ab 1 | ab 3 | ab 5 | ab 7 |
| cd L | cd 2+L | cd 4+L | cd 6+L |
| cd 1+L | cd 3+L | cd 5+L | cd 7+L |
| ab L | ab 2+L | ab 4+L | ab 6+L |
| ab 1+L | ab 3+L | ab 5+L | ab 7+L |

two MCs with four memory subsets

| | | | |
|---|---|---|---|
| cd1 0 | cd0 0 | cd1 2 | cd0 2 |
| cd0 1 | cd1 1 | cd0 3 | cd1 3 |
| ab1 0 | ab0 0 | ab1 2 | ab0 2 |
| ab0 1 | ab1 1 | ab0 3 | ab1 3 |
| cd1 L | cd0 L | cd1 2+L | cd0 2+L |
| cd0 1+L | cd1 1+L | cd0 3+L | cd1 3+L |
| ab1 L | ab0 L | ab1 2+L | ab0 2+L |
| ab0 1+L | ab1 1+L | ab0 3+L | ab1 3+L |

four MCs with eight memory subsets

| | | | |
|---|---|---|---|
| cd2 0 | cd3 0 | cd0 0 | cd1 0 |
| cd0 1 | cd1 1 | cd2 1 | cd3 1 |
| ab2 0 | ab3 0 | ab0 0 | ab1 0 |
| ab0 1 | ab1 1 | ab2 1 | ab3 1 |
| cd2 L | cd3 L | cd0 L | cd1 L |
| cd0 1+L | cd1 1+L | cd2 1+L | cd3 1+L |
| ab2 L | ab3 L | ab0 L | ab1 L |
| ab0 1+L | ab1 1+L | ab2 1+L | ab3 1+L |

**Figure 18: Dual-Swapped 2D Macro-Tile Mappings**

## 4.5 2D Address Equations

This subsection presents equations for computing addresses in a 2D array. This is a two-step process that makes use of the 1D array equations of subsection 3.3. The first list below defines parameters that are constant for a given surface. The second list names parameters that depend on which pixel is accessed in the array. 2D address equations use the parameters in the first list and one or more of the parameters in the second list to define the remaining parameters in the second list.

| | |
|---|---|
| Size | Bytes per pixel: can be 1, 2, 4, 8, or 16 (or 1/8 for 1-bit pixels) |
| **DataSize** | 64 times Size, equals the total bytes of data in a 2D tile |
| **Pipes** | Total number of Render Backend/Memory Controller pipelines: 1, 2, or 4 |
| Subsets | Total number of memory subsets: equals twice the number of pipelines |

| | |
|---|---|
| **SurfaceBase** | Byte address of pixel zero in device address space, must be 4K-byte aligned |
| **TileSize** | Bytes per tile: equals 64*Size, except for special tile formats that contain multiple pixel arrays |
| **TileBase** | Byte address of first pixel in a tile: normally zero, a multiple of 64 for special tile formats |
| **Pitch** | The width of each scanline in pixels, must be a multiple of 32 |
| **AltBank** | Boolean that selects alternate 2D macro-tile pattern that exchanges banks AB and CD |
| **SwapRB** | Boolean that selects swapping the RB numbers in the 2D macro-tile pattern |
| | |
| **X**, **Y** | Pixel location in the 2D array |
| **Local**Addr | Byte address of the pixel within its memory subset, starting from device address 0 |
| SubsetOffset | Byte address of the pixel within its memory subset, starting from pixel (0,0) of the surface |
| MemSelect | Number of the memory controller that stores this pixel |
| BankSelect | 0 for banks AB or 1 for banks CD, together with MemSelect determines the memory subset |
| **Subset** | Subset number, equals BankSelect + 2*MemSelect |
| MacroOffset | Sequential number of the macro-tile containing the pixel, starting from SurfaceBase |
| TileNumber | Sequential number of the tile containing the pixel, within its memory subset and its macro-tile |
| TileAddr | Byte address of the pixel within its tile, starting from the first byte of the tile |
| TileOffset | Byte address of the pixel within its tile, relative to TileBase (which is normally zero) |

The following equations use ***X*** and ***Y*** to compute the other address terms, particularly ***Local**Addr* and ***Subset***. The final set of equations in subsection 3.3 uses these results to produce a device address.

```
BankSelect    = ((Y/16) mod 2) ^ AltBank;                // Banks change for high/low half of each macro-tile
MemSelect     = (X/8 + ((Y/8 mod 2)^SwapRB)*(Pipes/2)) mod Pipes;   // Offset memory in alternate rows
Subset        = BankSelect + 2*MemSelect;                // subset number
TileNumber    = ((X mod 32)/8/Pipes)*2 + (Y/8 mod 2);    // Odd tile numbers are in odd rows
MicroByte     = (X mod 8 + ((Y mod 8)/2)*8)*Size         // Byte address within tile for even scanlines
                                                         // Odd scanlines get odd micro-tiles within an 8x8 tile
TileOffset    = (MicroByte mod 16) + (Y mod 2)*16 + (MicroByte/16)*32;
TileAddr      = TileBase + TileOffset;                   // The tile may contain other data as well
MacroOffset   = (X/32) + (Y/32) * (Pitch/32);            // There are Pitch/32 macro-tiles per row
SubsetOffset  = MacroOffset*TileSize*16/Subsets + TileNumber*TileSize + TileAddr;
LocalAddr     = SurfaceBase/Subsets + SubsetOffset;
```

The following equations use ***AltBank***, ***SwapRB***. ***Local**Addr* and ***Subset*** to compute the other address terms, particularlty the (***X***, ***Y***) array address. The final set of equations in subsection 3.3 convert a device address into ***Local**Addr* and ***Subset*** and these equations complete the conversion to an (***X***, ***Y***) array address.

```
MemSelect     = Subset / 2;
BankSelect    = Subset mod 2;
SubsetOffset  = LocalAddr – SurfaceBase/Subsets;         // subset address in surface
TileAddr      = SubsetOffset mod TileSize;               // byte address within the tile
TileOffset    = TileAddr - TileBase;                     // byte address within subset of the tile
MacroOffset   = SubsetOffset * Subsets / 16 / TileSize;  // 16*TileSize bytes per macro-tile
TileNumber    = SubsetOffset/TileSize mod (16/Subsets);  // 16 8x8 tiles per macro-tile over all subsets
MicroByte     = TileOffset mod 16 + (TileOffset/32)*16;  // byte address within even micro-tiles
Ymacro        = MacroOffset*32/Pitch;                    // Macro-tile offset vertically
Xmacro        = MacroOffset mod Pitch/32;                // macro-tile offset horizontally
Ytile         = (BankSelect^AltBank)*2 + (TileNumber mod 2);// tile 0, 1, 2, or 3 vertically in macro-tile
Xtile         = (MemSelect + ((Y/8 mod 2) ^SwapRB)*Pipes/2) mod Pipes + (TileNumber/2)*Pipes;
Ymicro        = ((MicroByte mod 16)/8 + (TileOffset/32)*2) / Size;      // row pair in tile due to micro-tiling
Xbyte         = MicroByte mod (8*Size);                  // byte address within first 8x1 scanline
Y             = Ymacro*32 + Ytile*8 + Ymicro*2 + (TileOffset/16 mod 2);
X             = Xmacro*32 + Xtile*8 + Xbyte/Size;
```

Note that these equations also work for non-standard pixel sizes and per-tile data. For 4-bit or 24-bit pixels, set Size to ½ or 3, set *TileSize* to the number of bytes in the tile, e.g. 64*32 and set *TileBase* to the starting byte in the tile for the pixel data, e.g. 0 or 8*64. For per-tile data, set *TileSize* to the number of bytes of data per tile and set *X* and *Y* to multiples of 8, that is, the lowest pixel address for the specified tile. This forces *MacroOffset* to zero, which causes *Size* to be ignored.

# 5. 3D Tiled Memory Formats

This section describes how the R400 family stores tiled 3D data arrays. Each tile contains an 8x8x1 array of data elements. Each 8x8x1 tile has a micro-tile format that depends on the data element size. Each 3D macro-tile contains a 4x1x4 array of tiles, which covers 32x8x4 pixels. This is different from 1D formats, where each tile and macro-tile stores a fixed number of bytes. Like the 1D formats, each 3D tiled surface must start on a 4K-byte boundary. Additionally, each NxMx4 slice of the 3D array must start on a 4K-byte boundary, so that individual 3D slices may be accessed as if they are a 2D array. {Is the 4K-byte restriction necessary?}

## 5.1 3D Micro-Tile Formats

Tiles in 3D arrays cover 8x8x1 data elements, even though the best aspect ratio for 3D tiles is probably a 4x4x4 array of data elements. That aspect ratio would provide the greatest degree of locality for random reads, for example, and therefore should be more efficient. However, the implementation is simpler if 3D tile formats are similar to 2D tile formats, for two reasons. First, this reduces the amount of multiplexing and address decoding required to read 3D texels. Second, it makes it simpler to render to (X, Y) slices within the 3D array. Therefore, the 3D tile formats encode an 8x8x1 tile of data elements, in exactly the same way as for 2D tiles.

The figure below shows how an 8x8x1 tile divides into micro-tiles for different pixel sizes. The numbers are the relative micro-tile addresses within the tile. 64-bits is the maximum size allowed for texels in 3D arrays. Unlike 2D arrays, 3D arrays do not support 4-bit and 24-bit pixel sizes.

| |
|---|
| Micro-Tile 0 <63:0> |
| Micro-Tile 1 <63:0> |
| Micro-Tile 0 <127:64> |
| Micro-Tile 1 <127:64> |
| Micro-Tile 2 <63:0> |
| Micro-Tile 3 <63:0> |
| Micro-Tile 2 <127:64> |
| Micro-Tile 3 <127:64> |

8-bit Pixels

| |
|---|
| Micro-Tile 0 |
| Micro-Tile 1 |
| Micro-Tile 2 |
| Micro-Tile 3 |
| Micro-Tile 4 |
| Micro-Tile 5 |
| Micro-Tile 6 |
| Micro-Tile 7 |

16-bit Pixels

| | |
|---|---|
| MT 0 | MT 2 |
| MT 1 | MT 3 |
| MT 4 | MT 6 |
| MT 5 | MT 7 |
| MT 8 | MT 10 |
| MT 9 | MT 11 |
| MT 12 | MT 14 |
| MT 13 | MT 15 |

32-bit Pixels

| | | | |
|---|---|---|---|
| 0 | 2 | 4 | 6 |
| 1 | 3 | 5 | 7 |
| 8 | 10 | 12 | 14 |
| 9 | 11 | 13 | 15 |
| 16 | 18 | 20 | 22 |
| 17 | 19 | 21 | 23 |
| 24 | 26 | 28 | 30 |
| 25 | 27 | 29 | 31 |

64-bit Pixels

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 |
| 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
| 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
| 17 | 19 | 21 | 23 | 25 | 27 | 29 | 31 |
| 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 |
| 33 | 35 | 37 | 39 | 41 | 43 | 45 | 47 |
| 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 |
| 49 | 51 | 53 | 55 | 57 | 59 | 61 | 63 |

128-bit Pixels

**Figure 19: 3D Micro-Tile Layout Within Tiles**

The figure below shows the format of texels inside each 16-byte (128-bit) micro-tiles. Pixels from even scanlines are in the lower 64-bits of each micro-tile and pixels from odd scanlines are in the upper 64-bits.

one micro-tile is 128 bits, as signified by the bold outline for each micro-tile

| | 127:120 | 119:112 | 111:104 | 103:96 | 95:88 | 87:80 | 79:72 | 71:64 | 63:56 | 55:48 | 47:40 | 39:32 | 31:24 | 23:16 | 15:8 | 7:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8-bit data | (7,2) | (6,2) | (5,2) | (4,2) | (3,2) | (2,2) | (1,2) | (0,2) | (7,0) | (6,0) | (5,0) | (4,0) | (3,0) | (2,0) | (1,0) | (0,0) |
| | (7,3) | (6,3) | (5,3) | (4,3) | (3,3) | (2,3) | (1,3) | (0,3) | (7,1) | (6,1) | (5,1) | (4,1) | (3,1) | (2,1) | (1,1) | (0,1) |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 16-bit data | Pixel (7,0) | Pixel (6,0) | Pixel (5,0) | Pixel (4,0) | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) |
| | Pixel (7,1) | Pixel (6,1) | Pixel (5,1) | Pixel (4,1) | Pixel (3,1) | Pixel (2,1) | Pixel (1,1) | Pixel (0,1) |

| | | | | |
|---|---|---|---|---|
| 32-bit data | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) |
| | Pixel (3,1) | Pixel 2,1) | Pixel (1,1) | Pixel (0,1) |

| | | |
|---|---|---|
| 64-bit data | Pixel (1,0) | Pixel (0,0) |
| | Pixel (1,1) | Pixel (0,1) |

| | |
|---|---|
| 128-bit data | Pixel (0,0) |
| | Pixel (0,1) |

**Figure 20: Pixel Format within 3D Micro-Tiles**

{**Note**: We should find a way to determine if we lose significant performance by not implementing 4x4x4 3D tiles.}

## 5.2 3D Macro-Tile Formats

The 3D macro-tile formats store a 32x16x4 array of data elements. This size allows reasonably efficient movement through the 3D array in either the X, Y, or Z directions, as described below. Although the tile size is 16 in Y, software should constrain the size in Y to a multiple of 32. That guarantees that each NxMx4 slab of 3D data occupies a multiple of 4K-bytes, even for 8-bit data elements. It is also simpler than enforcing a different Y height constraint for 3D arrays than for 2D arrays. The macro-tile size is expressed as 32x16x4, however, rather than as 32x32x4, because the 32x8x4 macro-tile size stores a contiguous array of bytes within each of the memory subsets. A 32x32x4 region includes bytes from two discotguous regions within each memory subset, unless the pitch happens to equal 32.

The following figures show the layout of 8x8x1 tiles within 32x16x4 macro-tiles for 3D arrays. Each figure shows two macro-tiles (slab 0 and slab 1) comprising a 32x16x8 region, since even and odd slices in Z use a different subset pattern. Each row of a figure shows the four slices within a macro-tile. Light lines mark tiles within the macro-tiles. The upper line of text in each tile specifies the memory subset. The lower line specifies the tile number within that memory subset. *S* equals the number of tiles per subset in a slice of the 3D array.

slab 0, slice 0 (Z=0)

| | | | |
|---|---|---|---|
| ab<br>0 | ab<br>4 | ab<br>8 | ab<br>12 |
| cd<br>0 | cd<br>4 | cd<br>8 | cd<br>12 |

slab 0, slice 1 (Z=1)

| | | | |
|---|---|---|---|
| ab<br>1 | ab<br>5 | ab<br>9 | ab<br>13 |
| cd<br>1 | cd<br>5 | cd<br>9 | cd<br>13 |

slab 0, slice 2 (Z=2)

| | | | |
|---|---|---|---|
| ab<br>2 | ab<br>6 | ab<br>10 | ab<br>14 |
| cd<br>2 | cd<br>6 | cd<br>10 | cd<br>14 |

slab 0, slice 3 (Z=3)

| | | | |
|---|---|---|---|
| ab<br>3 | ab<br>7 | ab<br>11 | ab<br>15 |
| cd<br>3 | cd<br>7 | cd<br>11 | cd<br>15 |

slab 1, slice 0 (Z=4)

| | | | |
|---|---|---|---|
| cd<br>S | cd<br>4+S | cd<br>8+S | cd<br>12+S |
| ab<br>S | ab<br>4+S | ab<br>8+S | ab<br>12+S |

slab 1, slice 1 (Z=5)

| | | | |
|---|---|---|---|
| cd<br>1+S | cd<br>5+S | cd<br>9+S | cd<br>13+S |
| ab<br>1+S | ab<br>5+S | ab<br>9+S | ab<br>13+S |

slab 1, slice 2 (Z=6)

| | | | |
|---|---|---|---|
| cd<br>2+S | cd<br>6+S | cd<br>10+S | cd<br>14+S |
| ab<br>2+S | ab<br>6+S | ab<br>10+S | ab<br>14+S |

slab 1, slice 3 (Z=7)

| | | | |
|---|---|---|---|
| cd<br>3+S | cd<br>7+S | cd<br>11+S | cd<br>15+S |
| ab<br>3+S | ab<br>7+S | ab<br>11+S | ab<br>15+S |

**Figure 21: 3D Macro-Tile Two Subset Format**

The figure above shows the tiled format for two memory subsets. This occurs when there is just one rendering pipeline. Movement in Y or Z through the 3D array hits both memory subsets. Movement in X hits only one memory subset, but alternates between the two banks within that subset. If the page size is 256 64-bit words and pixels are 64-bits in size or smaller, horizontally adjacent tiles are either in the same page of the same bank or are in different banks. This is not true for 128-bit pixels, but in that case sweeping across a single tile hits eight 256-bit accesses in the same page.

slab 0, slice 0 (Z=0)

| | | | |
|---|---|---|---|
| ab0 0 | ab1 0 | ab0 4 | ab1 4 |
| cd1 0 | cd0 0 | cd1 4 | cd0 4 |

slab 0, slice 1 (Z=1)

| | | | |
|---|---|---|---|
| ab0 1 | ab1 1 | ab0 5 | ab1 5 |
| cd1 1 | cd0 1 | cd1 5 | cd0 5 |

slab 0, slice 2 (Z=2)

| | | | |
|---|---|---|---|
| ab0 2 | ab1 2 | ab0 6 | ab1 6 |
| cd1 2 | cd0 2 | cd1 6 | cd0 6 |

slab 0, slice 3 (Z=3)

| | | | |
|---|---|---|---|
| ab0 3 | ab1 3 | ab0 7 | ab1 7 |
| cd1 3 | cd0 3 | cd1 7 | cd0 7 |

slab 1, slice 0 (Z=4)

| | | | |
|---|---|---|---|
| cd1 S | cd0 S | cd1 4+S | cd0 4+S |
| ab0 S | ab1 S | ab0 4+S | ab1 4+S |

slab 1, slice 1 (Z=5)

| | | | |
|---|---|---|---|
| cd1 1+S | cd0 1+S | cd1 5+S | cd0 5+S |
| ab0 1+S | ab1 1+S | ab0 5+S | ab1 5+S |

slab 1, slice 2 (Z=6)

| | | | |
|---|---|---|---|
| cd1 2+S | cd0 2+S | cd1 6+S | cd0 6+S |
| ab0 2+S | ab1 2+S | ab0 6+S | ab1 6+S |

slab 1, slice 3 (Z=7)

| | | | |
|---|---|---|---|
| cd1 3+S | cd0 3+S | cd1 7+S | cd0 7+S |
| ab0 3+S | ab1 3+S | ab0 7+S | ab1 7+S |

**Figure 22: 3D Macro-Tile Four Subset Format**

The figures above and below show the tiled format for four and eight memory subsets. Movement in Y or Z through the 3D array hits two memory subsets in different pipelines. Movement in X hits half of the memory subsets, one per pipeline. If the page size is 256 64-bit words and pixels are 64-bits in size or smaller, horizontally adjacent tiles in the same pipeline are either in the same page of the same bank or are in different banks. This is not true for 128-bit pixels, but in that case sweeping across a single tile hits eight 256-bit accesses in the same page.

slab 0, slice 0 (Z=0)

| | | | |
|---|---|---|---|
| ab0 0 | ab1 0 | ab2 0 | ab3 0 |
| cd2 0 | cd3 0 | cd0 0 | cd1 0 |

slab 0, slice 1 (Z=1)

| | | | |
|---|---|---|---|
| ab0 1 | ab1 1 | ab2 1 | ab3 1 |
| cd2 1 | cd3 1 | cd0 1 | cd1 1 |

slab 0, slice 2 (Z=2)

| | | | |
|---|---|---|---|
| ab0 2 | ab1 2 | ab2 2 | ab3 2 |
| cd2 2 | cd3 2 | cd0 2 | cd1 2 |

slab 0, slice 3 (Z=3)

| | | | |
|---|---|---|---|
| ab0 3 | ab1 3 | ab2 3 | ab3 3 |
| cd2 3 | cd3 3 | cd0 3 | cd1 3 |

slab 1, slice 0 (Z=4)

| | | | |
|---|---|---|---|
| cd2 S | cd3 S | cd0 S | cd1 S |
| ab0 S | ab1 S | ab2 S | ab3 S |

slab 1, slice 1 (Z=5)

| | | | |
|---|---|---|---|
| cd2 1+S | cd3 1+S | cd0 1+S | cd1 1+S |
| ab0 1+S | ab1 1+S | ab2 1+S | ab3 1+S |

slab 1, slice 2 (Z=6)

| | | | |
|---|---|---|---|
| cd2 2+S | cd3 2+S | cd0 2+S | cd1 2+S |
| ab0 2+S | ab1 2+S | ab2 2+S | ab3 2+S |

slab 1, slice 3 (Z=7)

| | | | |
|---|---|---|---|
| cd2 3+S | cd3 3+S | cd0 3+S | cd1 3+S |
| ab0 3+S | ab1 3+S | ab2 3+S | ab3 3+S |

**Figure 23: 3D Macro-Tile Eight Subset Format**

## 5.3 3D Address Equations

This subsection presents equations for computing addresses in a 3D array. This is a two-step process that makes use of the 1D array equations of subsection 3.3. The first list below defines parameters that are constant for a given surface. The second list names parameters that depend on which pixel is accessed in the array. 3D address equations use the parameters in the first list and one or more of the parameters in the second list to define the remaining parameters in the second list.

| | |
|---|---|
| Size | Bytes per pixel: can be 1, 2, 4, 8, or 16 |
| **DataSize** | 64 times Size, equals the total bytes of data in the 3D tile |
| **Pipes** | Total number of Render Backend/Memory Controller pipelines: 1, 2, or 4 |
| Subsets | Total number of memory subsets: equals twice the number of pipelines |
| **SurfaceBase** | Byte address of pixel zero in device address space, must be 4K-byte aligned |
| **TileSize** | Bytes per tile (should always equal 64*Size for 3D arrays, defined for consistency with 2D) |
| **TileBase** | Byte address of first pixel in a tile (should always be zero for 3D arrays) |
| **Pitch** | The width of each scanline in pixels, must be a multiple of 32 |
| **Height** | The height of each slice in scanlines, must be a multiple of 16 (should be multiple of 32) |

| | |
|---|---|
| **X**, **Y**, **Z** | Pixel location in the 3D array |
| **Local**Addr | Byte address of the pixel within its memory subset, starting from device address 0 |
| SubsetOffset | Byte address of the pixel within its memory subset, starting from pixel (0,0) of the surface |

| | |
|---|---|
| MemSelect | Number of the memory controller that stores this pixel |
| BankSelect | 0 for banks AB or 1 for banks CD, together with MemSelect determines the memory subset |
| **Subset** | Subset number, equals BankSelect + 2*MemSelect |
| MacroOffset | Sequential number of the macro-tile containing the pixel, starting from SurfaceBase |
| TileNumber | Sequential number of the tile containing the pixel, within its memory subset and its macro-tile |
| TileAddr | Byte address of the pixel within its tile, starting from the first byte of the tile |
| TileOffset | Byte address of the pixel within its tile, relative to TileBase (which is normally zero) |

The following equations use *X*, *Y* and *Z* to compute the other address terms. This is used to convert an array access into a *Subset* and *LocalAddr*. The final set of equations in subsection 3.3 uses these results to produce a device address.

```
BankSelect    = (Y/8 + Z/4) mod 2;                       // CD banks alternate every 8 Y and 4 Z
MemSelect     = (X/8 + BankSelect*(Pipes/2)) mod Pipes;  // Offset memory in alternate rows and slabs
Subset        = BankSelect + 2*MemSelect;                // subset number
TileNumber    = (Z mod 4) + ((X mod 32)/8/Pipes)*4;      // Groups of four tiles vertically
MicroByte     = (X mod 8 + ((Y mod 8)/2)*8)*Size         // Byte address within tile for even scanlines
                                              // Odd scanlines get odd micro-tiles within an 8x8 tile
TileOffset    = (MicroByte mod 16) + (Y mod 2)*16 + (MicroByte/16)*32;
TileAddr      = TileBase + TileOffset;                   // The tile may contain other data as well
MacroOffset   = (X/32) + (Pitch/32) * ((Y/16) + (Height/16)*(Z/4));
SubsetOffset  = MacroOffset*TileSize*32/Subsets + TileNumber*TileSize + TileAddr;
LocalAddr     = SurfaceBase/Subsets + SubsetOffset;
```

The following equations use *LocalAddr*, *BankSelect* and *MemSelect* to compute the other address terms. This is used to convert a device address into an (*X*, *Y*) array address. The final set of equations in subsection 3.3 produces *LocalAddr*, *BankSelect* and *MemSelect* from a device address.

```
SubsetOffset  = LocalAddr – SurfaceBase/Subsets;         // relative subset address in surface
TileAddr      = SubsetOffset mod TileSize;               // byte address within the tile
TileOffset    = TileAddr - TileBase;                     // byte address within subset of the tile
MacroOffset   = SubsetOffset * Subsets / 32 / TileSize;  // 32*TileSize bytes per macro-tile
TileNumber    = SubsetOffset/TileSize mod (32/Subsets);  // 32 8x8 tiles per macro-tile over all subsets
MicroByte     = TileOffset mod 16 + (TileOffset/32)*16;
Z             = (TileNumber mod 4) + (MacroOffset*32*16/Pitch/Height)*4;
Ymacro        = (MacroOffset*32/Pitch mod Height/16);    // Macro-tiles vertically within a slab
Ytile         = (BankSelect + (Z/4 mod 2)) mod 2;        // tile 0 or1 vertically in macro-tile
Y             = Ymacro*16 + Ytile*8 + (TileOffset/16/Size)*2 + (TileOffset/16 mod 2);
Xmacro        = MacroOffset mod Pitch/32;
Xtile         = (MemSelect + BankSelect*Pipes/2) mod Pipes + (TileNumber/4)*Pipes;
Xbyte         = MicroByte mod (8*Size);                  // Byte address within first 8x1 scanline
X             = Xmacro*32 + Xtile*8 + Xbyte/Size;
```

# 6. Mipmap Storage

{This section is for any special issues involving texture storage that belong in a whole-chip document instead of in the TC block spec. At present the only such issue is mipmap storage.}

## 6.1 Packing 2D Mipmaps

Small 2D surfaces waste a lot of space if each dimension must be increased to a multiple of 32. This is a particular problem for mipmap chains, which produce many small mipmaps. For example, if each mipmap produced by a 32x32 texture map requires a full macro-tile, then the mipmap chain requires 6*32*32 = 6144 pixels instad of 1365 pixels. The problem is worse for small texels, since each surface must start on a 4K-byte boundary. With 8-bit texels, the mipmap chain would require 6*4K-bytes = 24K-bytes, instead of 1365-bytes.

R400 solves this problem in two ways. First, each texture is specified with two surface descriptors. The first points to the base texture map, which has dimensions that are increased to multiples of 32. The second points to the start of the mipmap chain and R400 automatically computes the starting address of each subsequent mipmap in the chain. Each texture map in the mipmap chain has its dimensions increased to a power of 2 and each starts at an address that is a multiple of 4K-bytes.

Additionally, R400 packs the small mipmaps at the tail of the mipmap chain into a single 32x32 tile. Each mipmap has a position in the final tile that is based solely on the maximum of the width and height of that mipmap. The figure below shows the layout. Each mipmap in the chain is increased in size, if necessary, to a square mipmap with width and height equal to a power of two. The location where each mipmap is stored depends solely on its (increased) size. Any mipmap in the chain that has width > 16 or height > 16 is stored as a separate surface that uses a multiple of 4K-bytes. The final mipmaps also require a minimum of 4K-bytes, which is larger than a single 32x32 macro-tile for 8-bit and 16-bit texels.

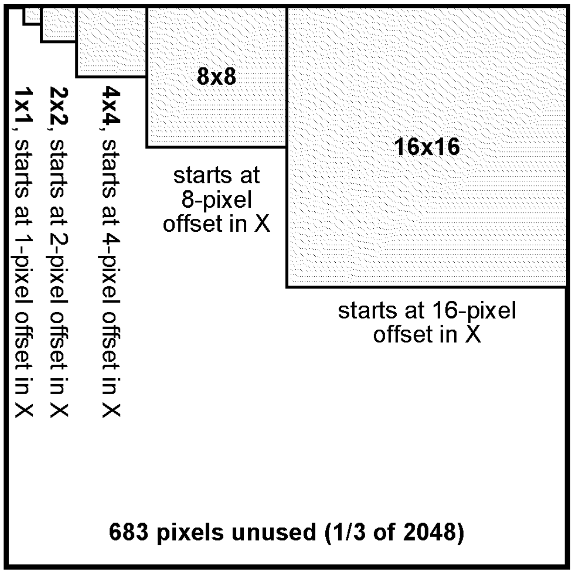
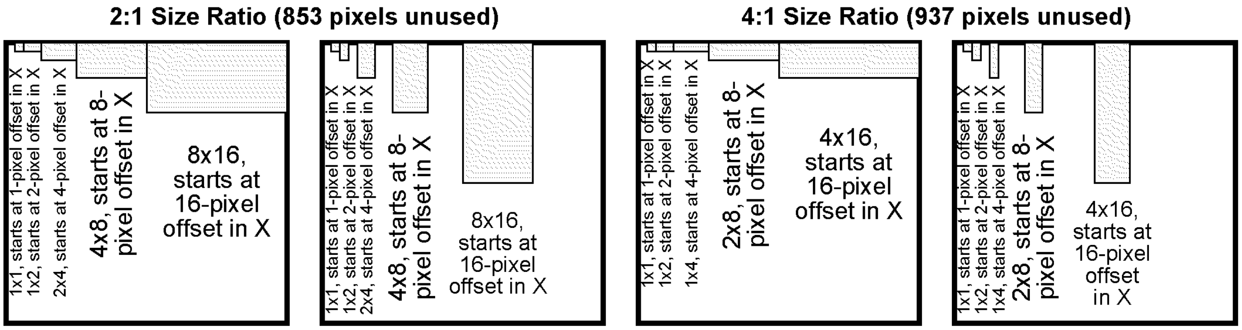


Figure 24: Mipmap Chain Storage Offsets

If the mipmaps are actually squares with width and height equal to a power of two, the above format uses 341 of 1024 pixels in the two tiles, wasting 683 or 2/3 of the pixels in the tile. The figure below shows examples of mipmaps with non-square aspect ratios and the number of pixels wasted in each case. Note that for each mipmap, texel (0,0) is stored in the same location as for the corresponding square mipmaps in the figure above.


2:1 Size Ratio (853 pixels unused)

4:1 Size Ratio (937 pixels unused)

Figure 25: Mipmap Chain Unused Pixels

Rendering to mipmaps that are packed this way requires altering the window offset. For example, to render to a mipmap that is expanded to 16x16, set the base address to the start of the macro-tile and increase the X offset by 16.

Future chips may use different offsets or packing formats, so the driver should obtain mipmap positions and offsets from code that is delivered with the hardware.

## 6.2 2D Mipmap Equations

{Describe how R400 computes the position of each mipmap in the chain and the total memory required for any mipmap chain.}

## 6.3 Packing 3D Mipmaps

{Define a 3D mipmap packing format.}

{Proposal: pack the final 3D mipmaps into a 32x32x32 cube, which contains 16 3D macro-tiles. Each 3D mipmap is expanded to a cube with all three sizes equal to a power of two that is greater than or equal to its largest dimension. }

{Variant: The same as the above, except only force the X and Y dimensions to match. Allow the Z dimension to be a power of two that is less than X or Y. This causes the packed mipmap to use a variable number of tiles.}

## 6.4 3D Mipmap Equations

{Describe how R400 computes the position of each mipmap in the chain and the total memory required for any mipmap chain.}

# 7. Destination Color Compression

R400 supports rendering to pixels with 1, 2, 3, 4, 6, or 8 samples per pixel. To a large extent, the aliased mode (1 sample per pixel) is just a special case of the multi-sample modes (2, 3, 4, 6, or 8 samples per pixel), though there are some operations, such as multi-buffer rendering, that are available only for single-sample pixels.

R400 stores multi-sample color data as fragments. A fragment is a pixel color together with a mask that specifies the samples within the pixel where that color is visible. As a result, if an operation writes a single color to an entire pixel, e.g. in the interior of a triangle, only one color (plus the mask) is necessary to describe the entire pixel. If there are S samples per pixel, the pixel could have as many as S fragments, but multiple fragments are only needed if multiple triangles are visible within a single pixel. Unless triangles are extremely small, it is quite common for a pixel to have just one fragment. The maximum number of fragments per pixel within an 8x8 tile is also typically small, so fragments result in significant compression. {For example, if there are eight samples per pixel and an average of less than 2 colors per pixel within a tile, storing fragments results in approximately 4x compression relative to supersampling.}

The following subsections describe the format of color data and fragment mask data.

## 7.1 Destination Color Format

R400 stores multi-sample pixels in two separate surfaces: one surface for pixel colors and a separate surface for the fragment masks (Fmasks). R400 uses a 4-bit Cmask field to specify storage format for the fragment mask and color. The figure below illustrates the color storage format for each value of Cmask.

If Cmask=Background, no color or fragment data is stored for the tile. Instead, each pixel is treated as having a single fragment that covers the entire pixel and is equal to the color_clear value. No data needs to be stored in the color surface in this mode.

If Cmask=Expanded, R400 stores a separate color for each sample. Starting at the base address, R400 stores a complete 2D tiled array for the color at sample 0, followed by a complete 2D tiled array for the color at sample 1, and so forth for the total number of samples per pixel. This format allows the texture logic and software to read multi-sample data by reading S individual 2D arrays for S-sample pixels.

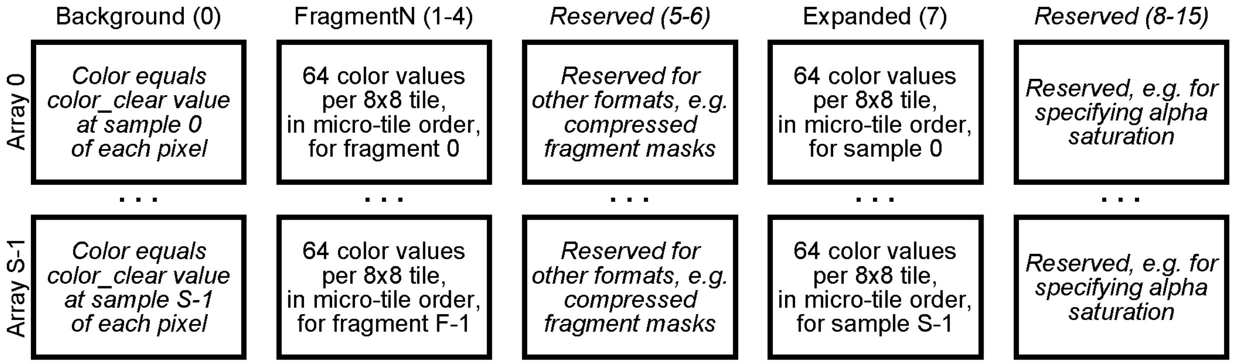


**Figure 26: Fragment Mask Bit Format**

The FragmentN formats allow compression when there are 2 or more samples per pixel. The choices are Fragment1, Fragment2, Fragment4, and Fragment 8 modes, which encode tiles with a maximum of 1, 2, 4, or 8 fragments in any single pixel. In these Cmask modes, each successive 2D array stores a single color per pixel from fragment 0 up to the maximum number of fragments. If the triangles in a scene are relatively large, then most tiles are likely to have at most one or two fragments per pixel. Storing different fragment colors in separate 2D arrays allows more tiles to share the same DDRAM page. This allows larger DDRAM page bursts when there are a small number of fragments per pixel.

## 7.2 Fragment Mask Format

A fragment mask consists of a set of n-bit fragment mask values, or Fmasks, with one such number per sample in each pixel. Each pixel within an 8x8 tile uses the same number of bits per Fmask. The number of bits in each Fmask depends on the maximum number of fragments per pixel within an 8x8 tile. If each pixel contains exactly one fragment, then no Fmask is required, since a single color completely covers each pixel. If a pixel in the tile contains 2 fragments but none contain more, then each Fmask requires 1-bit to select between one of two fragments per pixel. Similarly, 2-bit fragment masks are required if there is a maximum of 3 or 4fragments per pixel and 3-bit fragment masks are required if there is a maximum of 5 to 8 fragments per pixel. There cannot be more than 8 fragments per pixel, since there cannot be more than 8 samples that could have separate colors.

The Fmask buffer stores one or more 64-bit words per tile. Each 64-bit word stores one bit of Fmask data for one sample of each pixel in the 8x8 tile. The figure below shows the correspondence between the pixels of the tile and the bits in one 64-bit Fmask word.

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| word 0 | (7,1) | (6,1) | (5,1) | (4,1) | (3,1) | (2,1) | (1,1) | (0,1) | (7,0) | (6,0) | (5,0) | (4,0) | (3,0) | (2,0) | (1,0) | (0,0) |
| word 1 | (7,3) | (6,3) | (5,3) | (4,3) | (3,3) | (2,3) | (1,3) | (0,3) | (7,2) | (6,2) | (5,2) | (4,2) | (3,2) | (2,2) | (1,2) | (0,2) |
| word 2 | (7,5) | (6,5) | (5,5) | (4,5) | (3,5) | (2,5) | (1,5) | (0,5) | (7,4) | (6,4) | (5,4) | (4,4) | (3,4) | (2,4) | (1,4) | (0,4) |
| word 3 | (7,7) | (6,7) | (5,7) | (4,7) | (3,7) | (2,7) | (1,7) | (0,7) | (7,6) | (6,6) | (5,6) | (4,6) | (3,6) | (2,6) | (1,6) | (0,6) |

**Figure 27: Fragment Mask 1-Bit/Pixel Format**

The following figure shows multiple 64-bit words that store one Fmask bit each for each of S samples per pixel. The complete Fmask data stores 1, 2 or 3 copies of this set of S 64-bit words. This storge structure allows the RB to read and write either 1-bit, 2-bits, or 3-bits for each sample in the tile. The following table shows the number of micro-tiles required to store a tile of Fmask data for each number of samples and each

| Fmask bit for S samples | 63 48 | 47 32 | 31 16 | 15 0 |
|---|---|---|---|---|
| | word 3, sample 0 | word 2, sample 0 | word 1, sample 0 | word 0, sample 0 |
| | | . . . | | |
| | word 3, sample S-1 | word 2, sample S-1 | word 1, sample S-1 | word 0, sample S-1 |

**Figure 28: Fragment Mask 1-Bit/Sample Format**

| Samples per Pixel | Cmask = Fragment1 | Cmask = Fragment2 | Cmask = Fragment4 | Cmask = Fragment8 |
|---|---|---|---|---|
| **1 sample per pixel** | 0 of 0 micro-tiles | *(not used)* | *(not used)* | *(not used)* |
| **2 samples per pixel** | 0 of 1 micro-tiles | 1 of 1 micro-tiles | *(not used)* | *(not used)* |
| **3 samples per pixel** | 0 of 3 micro-tiles | 1.5 of 3 micro-tiles | 3 of 3 micro-tiles | *(not used)* |
| **4 samples per pixel** | 0 of 4 micro-tiles | 2 of 4 micro-tiles | 4 of 4 micro-tiles | *(not used)* |
| **6 samples per pixel** | 0 of 9 micro-tiles | 3 of 9 micro-tiles | 6 of 9 micro-tiles | 9 of 9 micro-tiles |
| **8 samples per pixel** | 0 of 12 micro-tiles | 4 of 12 micro-tiles | 8 of 12 micro-tiles | 12 of 12 micro-tiles |

**Table 1: Fmask Storage Required Per Tile**

Finally, the following figure shows the layout of a single tile of Fmask data for each number of samples per pixel and each FragmentN mode. Each row represents S*64-bits of Fmask data. Note that the FragmentN modes are not used when there is only one sample per pixel. In that case, the only allowed Cmask modes are Background and Expanded.

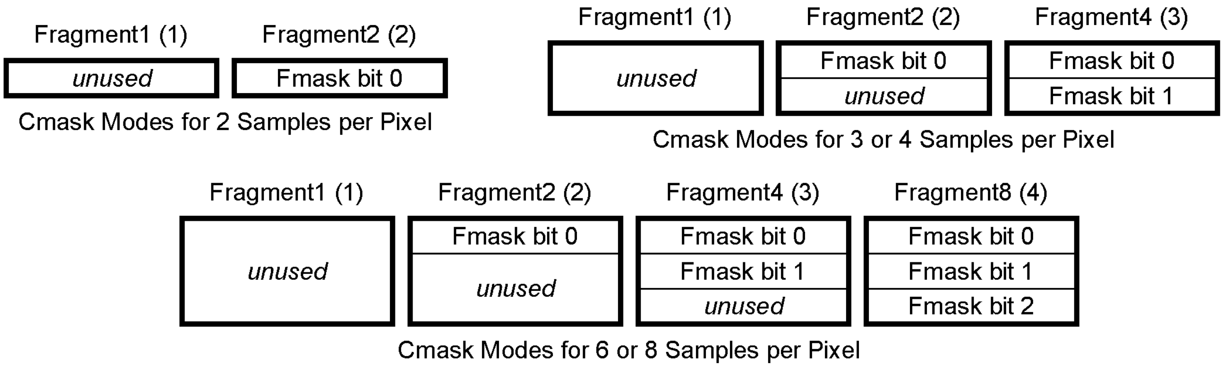


**Figure 29: Fragment Mask Bit Storages Format**

# 8. Depth and Stencil Formats

This section describes how the R400 family stores depth and stencil data at varying levels of compression. R400 supports 1, 2, 3, 4, 6, or 8 samples per pixel. Multi-sampled 8x8 tile formats must allocate enough frame buffer memory to be able to fall back to storing a separate value per sample in the cases where the data cannot be compressed. Therefore compression reduces the amount of data that must be read or written, but not the amount of memory that must be allocated.

The figure below illustrates the formats for storing depth data in a tile, depending on the 4-bit Zmask field in the 32-bit tile data word for each tile. If Zmask=Expanded, single-sample depth data is stored in standard micro-tile format as 16-bit or 32-bit pixels. This Expanded format allows single-sample depth and stencil values to be read and written by software and by the texture logic. All other depth formats are only readable and writable by the Render Backend depth logic and by address utility code that translates the compressed formats.

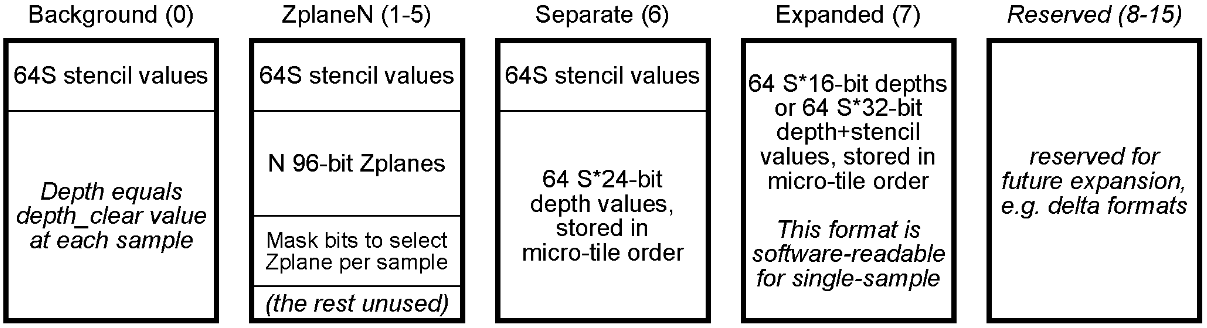


**Figure 30: Depth Storage Formats**

Remaining values of Zmask represent depth compression formats that store the stencil bits separately from depth bits. This allows depth and stencil to be accessed and compressed independently. If Zmask=Background, no depth data is stored in the tile. Instead, each depth value equals the depth_clear value. If Zmask=1-5, depth data is represented as Zplanes, which are described in the following subsection. If Zmask=Separate, depths are stored as a packed array of 16-bit or 24-bit values. The toal number of depth values is 64S, where S is the number of samples per pixel. These are stored as S adjacent arrays of 64 packed depth values, one per sample, similar to the format for storing multiple color fragments.

The following subsections describe stencil compression and Zplane compression.

## 8.1 Compressed Stencil Formats

The stencil buffer stores 8-bits per sample and is stored together with the depth buffer. Rendering operations can modify a pixel's stencil value based on the result of the depth test and on comparing the current stencil value to a reference value. The reference comparison can also be used to disable modifying the depth and color of the pixel. Allowed stencil modification operations are keeping the old value, setting it to zero, replacing it with the reference value, incrementing it, decrementing it, and inverting it.

The following is a brief summary of common uses of the stencil buffer. See the OpenGL Programming Guide for more information on these algorithms, except for shadow volumes, which is a more recent technique. Most of these algorithms use just two stencil values and set the same stencil value at each pixel that is written in a given triangle. The shadow volume method uses a range of values [base–N..base+N] for some base stencil value, where N depends on the number of overlapping shadows.

1) Stippling and irregular masking: set the stencil to define the pixels that can be updated.
2) Capping: invert the stencil on each pixel update to find places where clipping exposes an object's interior.
3) Non-convex polygons: invert the stencil on each pixel update to find the interior of a non-convex polygon.
4) Write once: change the stencil when writing the pixel; don't write if the stencil has already been changed.
5) Decals: change the stencil when writing the pixel; only write the decal where the stencil was changed.
6) Shadow Volumes: inc/dec the stencil based on projections of occluding objects, to find shadow regions.

Given these usages, R400 supports four types of stencil compression. The following table shows the four stencil compression modes, as selected by the 4-bit Smask field of the 32-bit tile data word. A fast stencil clear of the tile sets Smask to zero, indicating that the entire tile equals the stencil clear color. If Smask!=0, the lower three bits specify whether any stencil values in the tile are greater than (bit 2), less than (bit 1), or equal to (bit 0) a specified stencil compare value. If there aren't any stencil values greater than or less than the stencil compare value, then they all equal the stencil compare value, so again no bits need to be stored for the stencil values.

| Smask | Stencil Compression Mode |
|---|---|
| 0000 | 0-bits per stencil: every stencil in the entire tile is equal to the stencil clear value |
| 0001 | 0-bits per stencil: every stencil in the entire tile is equal to the stencil compare value |
| 0010-0111 | 4-bits per stencil: each stencil is the sum of the stencil base value plus an unsigned 4-bit offset |
| 1000 | 8-bits per stencil: every stencil in the entire tile is equal to the stencil clear value |
| 1001 | 8-bits per stencil: every stencil in the entire tile is equal to the stencil compare value |
| 1001-1111 | 8-bits per stencil: each stencil stores the full 8-bit value |

**Table 2: Stencil Compression Modes**

If the stencils are not all equal to the clear color or the compare color, then there are still two choices. A stencil base value may be used to compress the stencils. The stencil base value is typically set to max(0, stencil_compare – 8). If all the stencil values are in the range [base .. base+15], then a 4-bit offset is sufficient to specify each stencil value, relative to the stencil base. This is primarily useful for multi-sampled pixels. Finally, full 8-bit values may be stored for each stencil if any stencil values are outside the base range or if the stencil surface needs to be decompressed in order to allow software or the texture controller to read them.

If Zmask!=Expanded, stencil values are stored packed together at the start of the tile. If Zmask=Expanded, 8-bit stencils are interleaved with 24-bit depth values to produce 32-bit depth/stencil values, regardless of the value of Smask. This is the only interaction between stencil compression and depth compression. Zmask is only set to Expanded when writing a tile with depth compression disabled or after expanding the depth buffer to uncompressed format.

The table below shows the number of micro-tiles required to store stencil values, depending on the number of samples per pixel. These sizes apply for all depth compression modes except for Lockable. In Lockable mode, stencil values are stored as the lower byte of 32-bit words, for which the upper 24-bits are a depth value. Stencil compression is not available together with the Lockable depth mode, which is only produced as a result of a specific operation that converts single-sample depth/stencil values into a form that is directly readable and writable by software.

| Samples per Pixel | Smask = 0000-0001 | Smask = 0010-0111 | Smask = 1000-1111 |
|---|---|---|---|
| **1 sample per pixel** | 0 micro-tiles | 2 micro-tiles | 4 micro-tiles |
| **2 samples per pixel** | 0 micro-tiles | 4 micro-tiles | 8 micro-tiles |
| **3 samples per pixel** | 0 micro-tiles | 6 micro-tiles | 12 micro-tiles |
| **4 samples per pixel** | 0 micro-tiles | 8 micro-tiles | 16 micro-tiles |
| **6 samples per pixel** | 0 micro-tiles | 12 micro-tiles | 24 micro-tiles |
| **8 samples per pixel** | 0 micro-tiles | 16 micro-tiles | 32 micro-tiles |

**Table 3: Stencil Storage Sizes in Micro-Tiles**

A future chip could provide delta-encoded compression for stencil values in place of R4000's base/offset compression. This should allow significantly higher compression ratios for multi-sample stencil buffers.

If the stencils are stored as 8-bit values, they are micro-tiled in the stanard way for 8-bit pixels. Compressed 4-bit stencils are stored in a special micro-tile format that uses a 64-bit micro-tile instead of the standard 128-bit micro-tile. As for 8-bit stencils, the format depends on the number of samples. The compressed stencil formats are identical to the formats used for 4-bit Pmask values, which are described in subsection 8.4 below.

## 8.2 Zplane Depth Representation

R300 introduced compressing depth values by storing a plane equation for each triangle that intersects a tile. The plane equation allows the depth logic to compute a depth value at each sample, so that it is not necessary to store the individual depth values. The tile also stores a mask that specifies which of multiple plane equations to use at each sample.

The R400 Zplane compression format adapts the R300 technique to R400's 8x8 tiles and provides higher precision. As in R300, each Zplane is associated with a single triangle. Therefore, if four triangles are visible in an 8x8 tile, then the compressed tile must store four Zplanes and a 2-bit per sample mask that specifies which Zplane is visible at each sample. If only one triangle is visible in an 8x8 tile, then the compressed tile only needs to store that triangle's Zplane.

The following figure shows the 96-bit Zplane format used in R400. A Zplane contains six values. The slope in X and Y per subpixel (SlopeX and SlopeY) are each specified as a 30-bit fixed point S3.26 number. The depth value at the center of the 8x8 tile (CenterZ) is a 27-bit fixed point S3.23 number. Larger values for SlopeX, SlopeY, and CenterZ must be wrapped to these ranges by dropping higher order bits in fixed-point notation. So long as the depth values computed at sample points inside the primitive are in the range [-8..8), dropping the higher order bits does not affect the final depth value computed by R400 at sample points inside the primitive. The MultiSample bit is described below.

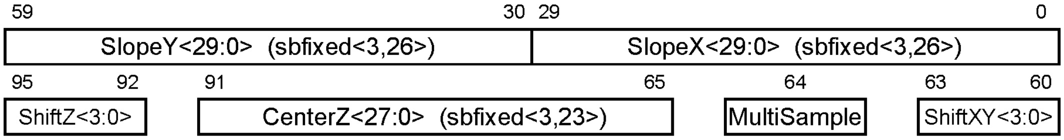


**Figure 31: Per-Triangle Zplane Format**

The ShiftXY and ShiftZ fields specify bit shift values for the SlopeX, SlopeY, and CenterZ fields, so that they can specify more accurate values with smaller ranges. The figure below shows how ShiftZ affects CenterZ and how ShiftXY affects SlopeX and SlopeY. When the shift is zero, the fixed-point value is converted to an S3.42 fixed-point value (for example) by appending low order zeros, which leaves the numeric value unchanged. Larger ShiftXY or ShiftZ values shift the fixed-point value right by the specified number of bits, sign extending the high order bits. These shifted values represent numbers in the range $[-2^{3-shift}..2^{3-shift})$. When ShiftZ==15, CenterZ represents values as small as $2^{-38}$. When ShiftXY==15, SlopeX and SlopeY represent values as small as $2^{-41}$. The slopes represent the change in depth per subpixel, so the smallest nonzero change in depth is $2^{-37}$ per pixel or $2^{-34}$ per tile. The smallest nonzero magnitude representable in the 24-bit floating-point depth format is also $2^{-34}$, so Zplanes allow specifying a slope of one lsb per tile in all depth formats.

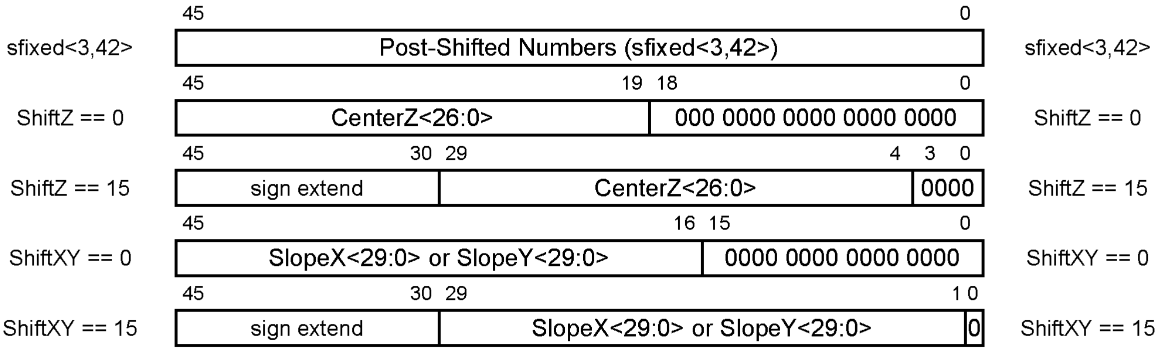


**Figure 32: Zplane Format Shifted Numbers**

The MultiSample bit specifies whether this Zplane was rendered with multisampling disabled, so that all samples are at the same location, or whether this Zplane was rendered with multisampling enabled, so that multiple samples occur at different locations within the pixel. Clients may enable and disable multisampling while rendering a scene, e.g. disabling it to render high quality anti-aliased lines with alpha blending. Disabling multisampling does not reduce the number of sample points per pixel – it simply moves them all to the same location. The MultiSample bit ensures that when a Zplane is converted to individual samples, this occurs using the multisample state that was valid at the time that the Zplane was generated, so that expanding the Zplane produces the same depth values that would occur if storing a separate depth value per sample.

Computing CenterZ and ShiftZ from a floating-point value FloatZ is straightforward. First convert FloatZ into an S3.42 value FixedZ, truncating low order bits of precision instead of rounding and dropping higher order bits to wrap around to this range. If FloatZ < –4 or FloatZ >= 4, then ShiftZ = 0 and CenterZ = (FixedZ + $2^{-24}$) <45:18>, that is, round off the lower 19 bits of FixedZ and use the remaining higher order bits as CenterZ. For smaller magnitudes of FloatZ, count the number of high order bits B in FixedZ that match the sign bit, up to 15. For example, in 0xF2, three high order bits match the sign bit. This can also be determined from the exponent of FloatZ. Then ShiftZ = B and CenterZ = (FixedZ + $2^{-24-B}$) <45–B:19–B>, that is, truncate the upper B bits of FixedZ and round off the lower 18–B bits of FixedZ. Finally, check whether rounding FixedZ caused CenterZ to overflow. If so, then ShiftZ = B+1 and CenterZ = 0x4000000 (2.0).

Computing SlopeX, SlopeY and ShiftXY from floating-point values FloatX and FloatY is similar. First convert FloatX and FloatY into S3.42 values FixedX and FixedY, truncating low order bits of precision instead of rounding and dropping higher order bits to wrap around to this range. If FloatX < –4, FloatX >= 4, FloatY < –4, or FloatY >= 4, then ShiftXY = 0, SlopeX = (FixedX + $2^{-27}$) <45:16>, and SlopeY = (FixedY + $2^{-27}$) <45:16>. For smaller magnitudes, count the number of high order bits BX and BY in FixedX and FixedY that match the sign bit, up to 15, and set B = min(BX,BY). This can also be determined from the exponents of FloatX and FloatY. Then ShiftXY = B, SlopeX = (FixedX + $2^{-27-B}$) <45–B:16–B>, and SlopeY = (FixedY + $2^{-27-B}$) <45–B:16–B>. Finally, check whether rounding FixedX or FixedY caused SlopeX or SlopeY to overflow. If so, then ShiftXY = B+1 and the overflowing slope or slope equals 0x10000000 (2.0).

Depth values and slopes can be significantly larger than the S3.N fixed-point formats supported above. R400 uses two's complement wraparound for depths and depth gradients. For example, if the true value of SlopeX is 8, the value stored in the Zplane format must be wrapped around to –8. In general, if a slope or CenterZ is outside the range [-8..8), represent it as a fixed point value, truncate higher order bits of integer precision and treat integer bit<3> as the sign bit to get the value to store in the Zplane format, along with a shift value of zero. This works provided that depth values computed at sample points inside the triangle are always in the range [-8..8).

## 8.3 Zplane Storage Formats

The figures below illustrates the five Zplane formats, which differ based on the number of Zplanes required in the tile and on whether there is one or multiple samples per pixel. Each Zplane occupies 96-bits, so each pair of Zplanes requires three 64-bit words. The Pmask data stores plane mask bits that specify which Zplane to use at each sample in the tile. For Zplane2 mode, there is a 1-bit Pmask per sample or an S-bit Pmask per pixel. For Zplane4 mode, there is a 2-bit Pmask per sample or a 2S-bit Pmask per pixel. For Zplane8 and Zplane16 modes, there is a 4-bit Pmask per sample or a 4S-bit Pmask per pixel.

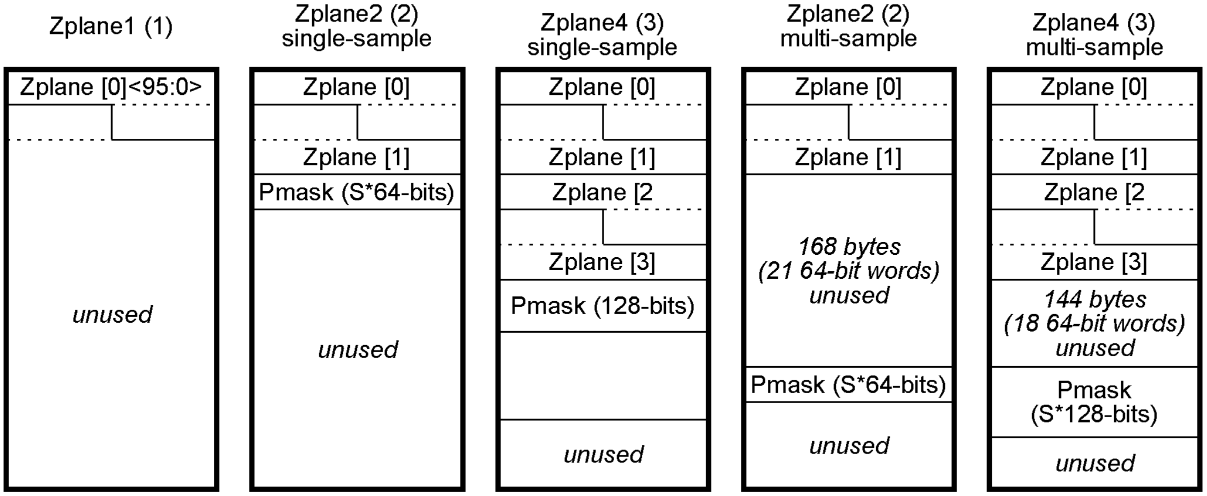


**Figure 33: One to Four Zplane Depth Tile Formats**

The single-sample and multi-sample storage formats differ in the location of the Pmask bits. Single-sample modes pack the Pmask bits immediately after the Zplanes. This reduces the number of 256-bit memory accesses required to read or write the compressed depth. Multi-sample formats all store the Pmask bits starting 16*96-bits after the start of the Zplane data. This simplifies the logic without increasing the number of 256-bit accesses.

Note that the Zplane16 mode only stores all 16 Zplanes for multi-sample depth values. For 32-bit single-sample depth values, Zplane16 mode only stores 12 Zplanes, due to the limited size of the tile. Zpane16 format is not supported for 16-bit single-sample depth values, since there is only room in the tile for 8 Zplanes.

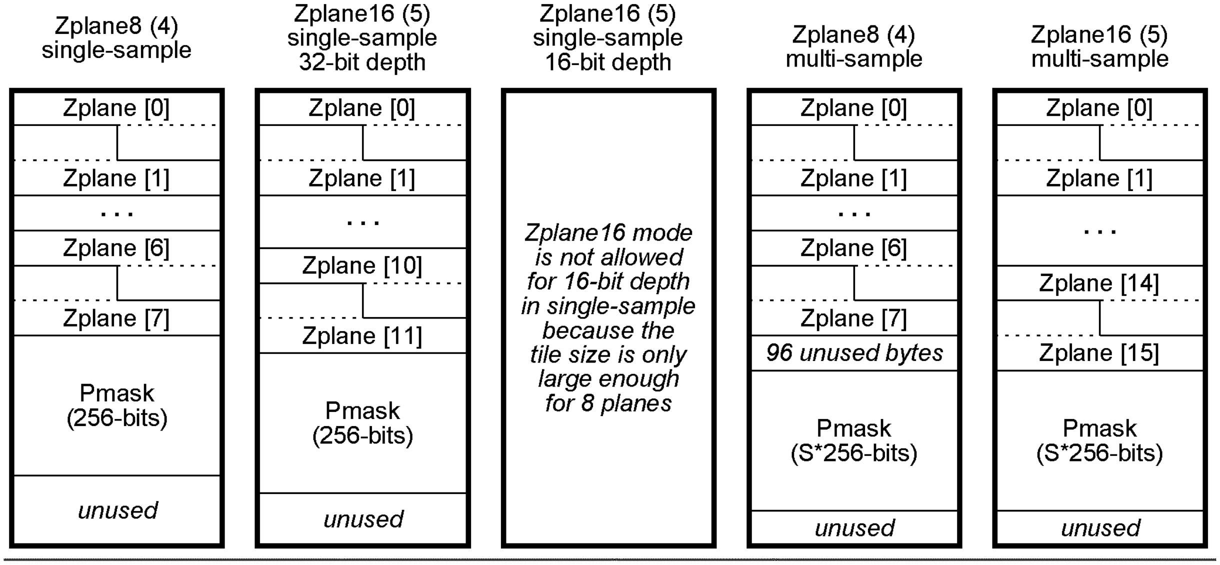


Figure 34: Eight or More Zplane Depth Tile Formats

The following table shows how many 128-bit (16-byte) micro-tiles are required to store an 8x8 tile of 16-bit or 24-bit depth data for each of the Zplane modes. To determine the number of 256-bit memory accesses required for each format, divide by 2 and round up. For comparison, the table also lists the number of micro-tiles required in the Separate and Expanded formats. The micro-tiles required for Expanded format are listed in parenthese and includes storage for stencil data, if any. Typically, the total tile size for depth/stencil data equals the size of Expanded mode and the amount of memory allocated for depth data within each tile equals the size of the Separate format. Zplane16 mode Requires too many micro-tiles when there is one sample per pixel, so Zplane16 mode can only be used with 2 or more samples per pixel.

| **Samples per Pixel** | **Zplane1 (1)** | **Zplane2 (2)** | **Zplane4 (3)** | **Zplane8 (4)** | **Zplane16 (5)** | Separate (or Expanded) (6-7) | |
|---|---|---|---|---|---|---|---|
| | | | | | | **16-bit depth** | **24-bit depth** |
| **1** | 0.75 | 2 | 4 | 8 | (N/A) | 8 (8) | 12 (16) |
| **2** | 0.75 | 2.5 | 5 | 10 | 16 | 16 (16) | 24 (32) |
| **3** | 0.75 | 3.0 | 6 | 12 | 18 | 24 (24) | 36 (48) |
| **4** | 0.75 | 3.5 | 7 | 14 | 20 | 32 (32) | 48 (64) |
| **6** | 0.75 | 4.5 | 9 | 18 | 24 | 48 (48) | 72 (96) |
| **8** | 0.75 | 5.5 | 11 | 22 | 28 | 64 (64) | 96 (128) |

**Table 4: Depth Storage Sizes in Micro-Tiles**

The storage required by the Zplane formats goes up significantly as the number of samples increases, since the number of bits required to store the mask is proportional to the number of samples. A future chip could achieve higher levels of multi-sample compression by encoding the mask data, taking advantage of the fact that adjacent samples typically use the same Zplane.

## 8.4 Pmask Storage Formats

The compressed depth formats store a Pmask value per sample, which specifies the Zplane that must be used to compute the depth at that sample. Zplane8 and Zplane16 modes store 4-bits per Pmask, Zplane 4 mode stores 2-bits per Pmask, and Zplane2 mode stores 1-bit per Plask. Zplane1 mode does not require a Pmask, since the entire tile is covered by a single Zplane.

The Pmask data is stored as an 8x8 tile in a modified micro-tile format. The pmask values for all of the samples for a pixel are combined into a single pmask-pixel, which is then micro-tiled using 64-bit micro-tiles for 4-bit Pmasks, 32-bit micro-tiles for 2-bit Pmasks, and 16-bit micro-tiles for 1-bit Pmasks. This causes the Pmask micro-tiling to match up with the micro-tiling for 8-bit stencil values, which makes it easier for R400 to interleave corresponding stencil and Pmask data in the internal depth/stencil cache.

The figure below shows the Pmask storage pattern for 8-sample pixels. Even and odd scanlines interleave each 16N-bits, where N is the number of bits per Pmask. Each pmask-pixel contains eight N-bit Pmask values. Therefore, a single 16N-bit interleave is filled by two pmask-pixels. As a result, the interleave pattern stores Pmask values for two horizontally adjacent pixels on an even scanline, then two horizontally adjacent pixels on an odd scanline, and so forth. This is like the standard micro-tile format, except that the micro-tile size is 16N-bits instead of 128-bits.

| | 64N-1 48N | 48N-1 32N | 32N-1 16N | 16N-1 0 |
|---|---|---|---|---|
| **0** | pmask (2-3, 1) | pmask (2-3, 0) | pmask (0-1, 1) | pmask (0-1, 0) |
| **1** | pmask (6-7, 1) | pmask (6-7, 0) | pmask (4-5, 1) | pmask (4-5, 0) |
| **2** | pmask (2-3, 3) | pmask (2-3, 2) | pmask (0-1, 3) | pmask (0-1, 2) |
| **3** | pmask (6-7, 3) | pmask (6-7, 2) | pmask (4-5, 3) | pmask (4-5, 2) |
| **4** | pmask (2-3, 5) | pmask (2-3, 4) | pmask (0-1, 5) | pmask (0-1, 4) |
| **5** | pmask (6-7, 5) | pmask (6-7, 4) | pmask (4-5, 5) | pmask (4-5, 4) |
| **6** | pmask (2-3, 7) | pmask (2-3, 6) | pmask (0-1, 7) | pmask (0-1, 6) |
| **7** | pmask (6-7, 7) | pmask (6-7, 6) | pmask (4-5, 7) | pmask (4-5, 6) |

*N-bit Pmask, 8-sample: 16N-bit interleave of even/odd scanlines*

**Figure 35: Pmask Interleave For 8 Samples**

The next figure shows the Pmask storage patterns for 1-sample, 2-sample, and 4-sample pixels. As before, even and odd scanlines interleave each 16N-bits, where N is the number of bits per Pmask. Each pmask-pixel contains eight N-bit Pmask values. Therefore, a single 16N-bit interleave is filled by two pmask-pixels. As a result, the interleave pattern stores Pmask values for two horizontally adjacent pixels on an even scanline, then two horizontally adjacent pixels on an odd scanline, and so forth. This is like the standard micro-tile format, except that the micro-tile size is 16N-bits instead of 128-bits.

| | 64N-1 56N | 56N-1 48N | 48N-1 40N | 40N-1 32N | 32N-1 24N | 24N-1 16N | 16N-1 8N | 8N-1 0 |
|---|---|---|---|---|---|---|---|---|
| **0** | pmsk(0-7,7) | pmsk(0-7,5) | pmsk(0-7,6) | pmsk(0-7,4) | pmsk(0-7,3) | pmsk(0-7,1) | pmsk(0-7,2) | pmsk(0-7,0) |

*N-bit Pmask, 1-sample: 16N-bit interleave of even/odd scanlines*

| | 64N-1 48N | 48N-1 32N | 32N-1 16N | 16N-1 0 |
|---|---|---|---|---|
| **0** | pmask (0-7, 3) | pmask (0-7, 2) | pmask (0-7, 1) | pmask (0-7, 0) |
| **1** | pmask (0-7, 7) | pmask (0-7, 6) | pmask (0-7, 5) | pmask (0-7, 4) |

*N-bit Pmask, 2-sample: 16N-bit interleave of even/odd scanlines*

| | 64N-1 48N | 48N-1 32N | 32N-1 16N | 16N-1 0 |
|---|---|---|---|---|
| **0** | pmask (4-7, 1) | pmask (4-7, 0) | pmask (0-3, 1) | pmask (0-3, 0) |
| **1** | pmask (4-7, 3) | pmask (4-7, 2) | pmask (0-3, 3) | pmask (0-3, 2) |
| **2** | pmask (4-7, 5) | pmask (4-7, 4) | pmask (0-3, 5) | pmask (0-3, 4) |
| **3** | pmask (4-7, 7) | pmask (4-7, 6) | pmask (0-3, 7) | pmask (0-3, 6) |

*N-bit Pmask, 4-sample: 16N-bit interleave of even/odd scanlines*

**Figure 36: Pmask Interleave For 1, 2, or 4 Samples**

The final figure shows the Pmask storage patterns for 3-sample and 6-sample pixels. For these non-power-of-two sample sizes, the Pmask values for some samples of a pixel may be in a different 16N-bit interleave from the Pmask values for the rest of the samples of a pixel. This is notated by marking pixels with letters *a* to *c* for 3-sample pixels or *a* to *f* for 6-sample pixels. So for example, the interleave marked "pmask (0a-5a, 0)" stores all of the Pmask values for Y=0 and X=0 through 4, and also stores the Pmask for the first sample of pixel (5,0).

<table>
<tr><th></th><th>64N-1 56N</th><th>56N-1 48N</th><th>48N-1 40N</th><th>40N-1 32N</th><th>32N-1 24N</th><th>24N-1 16N</th><th>16N-1 8N</th><th>8N-1 0</th></tr>
<tr><td>0</td><td>pmsk(0a-2b,3)</td><td>pmsk(5b-7c,1)</td><td>pmsk(0a-2b,2)</td><td>pmsk(5b-7c,0)</td><td colspan="2">pmask (0a-5a, 1)</td><td colspan="2">pmask (0a-5a,0)</td></tr>
<tr><td>1</td><td colspan="2">pmask (0a-5a, 5)</td><td colspan="2">pmask (0a-5a,4)</td><td colspan="2">pmask (2c-7c, 3)</td><td colspan="2">pmask (2c-7,2)</td></tr>
<tr><td>2</td><td colspan="2">pmask (2c-7c, 7)</td><td colspan="2">pmask (2c-7c,6)</td><td>pmsk(0a-2b,7)</td><td>pmsk(5b-7c,5)</td><td>pmsk(0a-2b,6)</td><td>pmsk(5b-7c,4)</td></tr>
</table>

*N-bit Pmask, 3-sample: 16N-bit interleave of even/odd scanlines*

| | 64N-1 48N | 48N-1 32N | 32N-1 16N | 16N-1 0 |
|---|---|---|---|---|
| 0 | pmask (2e-5b, 1) | pmask (2e-5b,0) | pmask (0a-2d, 1) | pmask (0a-2d,0) |
| 1 | pmask (0a-2d, 3) | pmask (0a-2d,2) | pmask (5c-7f,1) | pmask (5c-7f,0) |
| 2 | pmask (5c-7f,3) | pmask (5c-7f,2) | pmask (2e-5b, 3) | pmask (2e-5b,2) |
| 3 | pmask (2e-5b, 5) | pmask (2e-5b,4) | pmask (0a-2d, 5) | pmask (0a-2d,4) |
| 4 | pmask (0a-2d, 7) | pmask (0a-2d,6) | pmask (5c-7f,5) | pmask (5c-7f,4) |
| 5 | pmask (5c-7f,7) | pmask (5c-7f,6) | pmask (2e-5b, 7) | pmask (2e-5b,6) |

*N-bit Pmask, 6-sample: 16N-bit interleave of even/odd scanlines*

Figure 37: Pmask Interleave For 4 and 6 Samples

Another way to think of these storage formats is to treat them as compressed versions of 8-bit storage formats. Imagine padding out the Pmask values to 8-bits, and then storing them as 8S-bit pixels, where there are S samples. This storage format would use the standard 128-bit microtiling. The 4-bit Pmask format above is the same as this 8-bit format, except with bits<7:4> of each byte omitted. This results in a 64-bit micro-tile, as described above. Similarly, 2-bit and 1-bit Pmask values are stored the same as this 8-bit format, except with bits<7:2> and bits<7:1> omitted from each byte, respectively. The RB logic stores each Pmask value with the stencil value for the same sampleThis compression format makes it easier for the RB logic to match them up, since it means that the sequence of stencil values matches the sequence of Pmask values in memory.

|  | ORIGINATE DATE | EDIT DATE | DOCUMENT-REV. NUM. | PAGE |
|---|---|---|---|---|
|  | 10 July, 2003 | 1 October, 2003 | GEN-CXXXXX-REVA | 1 of 34 |

**Author:** Larry Seiler

**Issue To:**

**Copy No:**

# R400 Memory Format Specification

## Version 0.10

**Overview:** This specification describes how pixels and other data are stored in the frame buffer.

AUTOMATICALLY UPDATED FIELDS:
**Document Location :** C:\r400\doc_lib\design\chip\memory\R400_MemoryFormat.doc
**Current Intranet Search Title:** R400 Frame Buffer Layout Specification

| APPROVALS |  |
|---|---|
| Name/Dept | Signature/Date |
|  |  |
|  |  |
|  |  |

Remarks:

THIS DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

## Table Of Contents

## Table Of Figures

## Table Of Tables

## Revision Changes:

| | |
|---|---|
| **Rev 0.1 (Larry Seiler)**<br>Date: March 8, 2001 | First draft |
| **Rev 0.2 (Larry Seiler)**<br>Date: June 25, 2001 | Complete rewrite. Added 1D and 3D formats and modified the 2D formats. Added addressing modes, and some information on depth compression. |
| **Rev 0.3 (Larry Seiler)**<br>Date: July 3, 2001 | Added more 3D formats and compressed formats. Various minor changes. |
| **Rev 0.4 (Larry Seiler)**<br>Date: October ???, 2001 | Partially updated version |
| **Rev 0.5 (Larry Seiler)**<br>**Date: December 20, 2001** | Revised micro-tile formats, significant additions. |
| **Rev 0.6 (Larry Seiler)**<br>**Date: January 29, 2002** | Updated color and depth compression formats, added pixel format descriptions, added tiling for per-tile data and non-standard pixels. |
| **Rev 0.7 (Larry Seiler)**<br>**Date: March 5, 2002** | Change to 32x32 2D macro-tile size and 32x16x4 3D macro-tile size. Added addressing equations and 2D mipmap packing. |
| **Rev 0.8 (Larry Seiler)**<br>**Date: June 4, 2002** | File name and location changed. All surfaces now have a 4KB alignment constraint for allocation. Pitch must be 256-byte multiple for linear arrays. LocalBase eliminated from tiling equations. Zplane format changed. Multi-sample color format changed. |
| **Rev 0.8B (Larry Seiler)**<br>**Date:** | Changed Zplane format to include the MultiSample bit, changed a parenthesis in 3D Ytile computation |
| **Rev 0.9 (Larry Seiler)**<br>**Date: May, 2003, minor edits July 10, 2003** | Added alternate 2D tiling formats for efficient depth buffering and to support depth with a 3D slice. |
| **Rev 0.10 (Larry Seiler)**<br>**Date: October 1, 2003** | Change depth format: separate/expanded modes interleave depth values for multi-sampling and Pmask data is now stored after the Zplanes. |

# 1. Background

This document describes how the R400 family maps pixels and other data into local (video) memory and system (AGP) memory. The R400 uses 32-bit device addresses that map both local memory and system memory within a $2^{32}$-byte device address space. Local memory is accessed through one, two or four memory controllers that each access up to $2^{28}$-bytes (256M-bytes) of on-board memory.

Data can be organized into 1D, 2D or 3D arrays. Types of data include coordinates/normals, uncompressed pixels/texels, uncompressed depth/stencil values, and a variety of compressed data formats. 2D and 3D arrays can be organized in interleaved tiles for higher performance. All array sizes can be stored in a linear format, which stores pixels along a scanline at sequential device addresses, but R400 does not support depth buffer operations in linear format.

The subsections below describe some important common characteristics of all of the memory formats in all of the array sizes. These topics include dividing local memory into disjoint subsets, address alignment constraints, and address modes. Following sections describe the bit formats of individual data elements (pixels, texels, etc.), tiling formats for 1D, 2D, and 3D arrays of data elements, and special tile formats for multi-sample data and depth/stencil data.

## 1.1 System vs. Local Memory

Data can be stored either in local (on-board or frame buffer) memory or in system (AGP/PCI) memory. A specified range of the device address space maps to local memory and the rest maps to system memory.

Logic blocks that compute device addresses need not be aware of whether an address is in local or system memory, though some logic blocks (e.g. the Display Controller) require data in local memory due to latency issues. The system memory bus has significantly lower bandwidth than the local memory bus and has tremendously longer read latency. AGP memory accesses use either 256-bit or 512-bit bursts.

{Give specific ranges for AGP latency. Reference Tom's memory space spec.}

## 1.2 Local Memory Subsets

The R400 family accesses local on-board memory through one to four independent memory controllers. The R400 has four memory controllers, each of which provides access to one quarter of the local memory. The RV400 has two memory controllers, each of which provides access to half of the local memory. A future integrated version of the chip will have a single memory controller.

Each memory controller is associated with a corresponding render backend block. Each render backend can only access the local memory associated with its own memory controller. Hence the local memory is divided into subsets, so that each memory controller stores the pixels or other data elements that are processed by its associated render backend. All of the render backends can access system memory.

DDRAM memory is divided into multiple banks and pages. Each bank is in effect a separate memory array inside the DDRAM. Each page contains bits from a single row in the internal DDRAM memory array for a given bank. The size of a page in bytes depends on the specific DDRAM part and the number that are used for a single memory controller. Accessing data within the same page of a bank (which is called "column access") is dramatically faster than accessing data in a different page of the bank (which is called "row access"). Accordingly, much effort in the tiling design goes into grouping accesses that are on the same page and avoiding situations where two different pages in a bank must be accessed without intervening accesses in other banks.

Each memory controller contains two separate memory subsets, in order to improve memory access efficiency. The R400 family supports DDRAM memory that is organized into four banks: A, B, C, and D, so each of the two memory subsets per memory controller contains data from two of the banks. For each memory controller, one subset includes banks A and B and the other subset includes banks C and D. Since R400 has four memory controllers, it has eight memory subsets, which are called ab0, ab1, ab2, ab3, cd0, cd1, cd2, and cd3. RV400 has two memory controllers, so its four memory subsets are called ab0, ab1, cd0, and cd1.

Within a memory subset, memory accesses alternate between two banks at the page boundaries. Consider memory subset ab0. The first word of this memory subset is in page 0 of bank A. This is followed by the rest of the words of bank A in page 0, then by page 0 of bank B. Next comes page 1 of bank A, then page 1 of bank B, etc. As a result, the two banks are interleaved on a page-by-page basis. The figure below illustrates this bank alternation for a DDRAM with pages consisting of 256 words. The word size depends on the specific DDRAM type. Each MC reads 64-bit words from multiple DDRAMs in parallel, depending on the DDRAM configuration.

| page 0 bank A | | | | page 0 bank B | | | | page 1 bank A | | | | page 1 bank B | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| word 0 | word 1 | ... | word 255 | word 0 | word 1 | ... | word 255 | word 0 | word 1 | ... | word 255 | word 0 | word 1 | ... | word 255 |

**Figure 1: Local Memory Subset for Banks AB**

Dividing the memory into subsets in this fashion ensures that sequential accesses within a memory subset are always either within the same page or are within different banks. For example, suppose that a sequential access starts at page 0, bank A, word 255. A large number of accesses to bank B occur before accessing bank A, page 1, word 0. This gives the DDRAM array time to perform the slow "row access" to bank A, in parallel with the fast "column accesses" within bank B.

## 1.3 Units of Memory

This document describes memory formats in terms of four levels of addressable units in local and system memory. Each addressable unit has different address alignment constraints.

Individual data elements have a wide variety of sizes. Each occupies $2^N$-bytes for some value of N and each is naturally aligned on a $2^N$-byte boundary. The most common size is 4-bytes (32-bits), which could represent, for example, a 32-bit ARGB pixel or a 32-bit IEEE floating point value.

**Data elements** are organized into 16-byte (128-bit) micro-tiles. This corresponds to the read/write bus size between each memory controller and its render backend, though each memory access reads or writes an aligned pair of micro-tiles.A micro-tile may contain four 32-bit floats or a larger number of smaller data elements. For 1D arrays, a micro-tile always contains sequential data in the 1D array. For 2D and 3D arrays, a micro-tile contains data from a single scanline for 16-bit, 32-bit, 64-bit, and 128-bit data elements.

**Tiles** are variable-sized. For 1D arrays, each tile stores exactly 64-bytes (512-bits), which stores a variable number of data elements, e.g. 64 8-bit pixels or four 128-bit pixels that each contain four 32-bit floats. For 2D and 3D arrays of pixels or texels, each tile stores an 8x8 array of data elements, which occupies a multiple of 64-bytes (512-bits). The most typical data element size for 2D arrays is 32-bits, which results in a 256-byte (2K-bit) tile size. 2D arrays can also store a single per-tile data element, instead of 64 pixel or texel data elements. The Render Backend uses this mode.

A **macro-tile** is the basic unit of memory allocation in both linear and tiled formats. For linear arrays, each macro-tile is exactly 4K-bytes and contains exactly 16 tiles. For 2D tiled arrays, each macro-tile contains 32x32 pixels, arranged as a 4x4 sub-array of 8x8 pixel tiles. For 3D tiled arrays each macro-tile contains 32x16x4 pixels, arranged as a 4x2x4 sub-array of 8x8x1 pixel tiles.

Finally, a **surface** is a contiguous range of device address space that is all interpreted the same way, e.g. as either a linear array or a 2D or 3D tiled array with a specific pitch and pixel size. Each surface must start on a 4K-byte aligned address in device address space.

# 2. Data Element Formats

This section describes pixel formats, texel formats, per-tile data formats, and other types of data elements that the R400 reads and writes. These formats are used in the Memory Hub (MH) block to support client memory accesses, in the Texel Central (TC) block to read and interpolate data for the shader programs, and in the Render Backend (RB) block to read and write data render data.

The following subsections divide the data element formats into four groups. Displayable pixel formats are fully supported by R400, including displaying them to the monitor. Renderable pixel formats are usable as render targets with full support for alpha blending (with one exception), as well as for texture inputs. The Texel-Only Pixel formats may be used as texel inputs or render targets, but cannot be alpha blended. The Texel-Only formats cannot be used as render targets. Finally, the Special Data Formats have specific, limited uses.

Each pixel contains between one and four components, named C0 through C3. The TC and RB blocks both contain pixel format descriptors that specify how to interpret the components as numbers and how to map them to the four components that are computed in the shader pipe. The MH block passes them through without interpretation.

{**Note:** define the number formats here: floating-point, repeating fraction, integer, etc.}

## 2.1 Displayable Pixel Formats

The pixel formats described below are fully supported by the MH, TC, and RB blocks. Additionally, each of these frame buffer formats may be displayed to the monitor.

**{List the displayable pixel formats. Questions: (1) is GRPH_SWAP_RB supported for all formats, or only for the ones listed in the display spec? (2) How are YUV modes supported, and which ones? (3) How does the display logic support unsigned vs. signed vs. float number formats? All three expand differently for use with the linear interpolation table.}**

## 2.2 Renderable Pixel Formats

Renderable pixel formats are those that the RB (render backend) block can produce. They are also fully supported by the MH and TC blocks. Each pixel contains between one and four components, named C0 through C3. The MH block transfers whole pixels without interpreting their content. The TC and RB blocks both contain pixel format descriptors that specify how to interpret the components as numbers and how to map them to the four components that are computed in the shader pipe. The TC allows an arbitrary mapping of the input components to the shader pipe components. The RB supports more limited component mappings, as described below.

The figures in this subsection list multiple names for each renderable pixel format. Names of the form FMT_* are enumeration constants from enum type SurfaceFormat. Names of the form COLOR_* are enumeration constants from enum type ColorFormat, which contains the subset of surface formats that are renderable. Each COLOR_* enumeration name has the same value as a corresponding FMT_* enumeration name.

The figure below illustrates the single-component renderable pixel formats, with their enumeration names. The 8-bit format has three separate pairs of names to support format numbers used by legacy code. The 16-bit formats can either be explicitly floating point or else may use one of several fixed-point number formats. For the 32-bit component size, only the floating-point format is renderable, and the Render Backend cannot alpha blend that component size. The Render Backend can map either the shader pipe Red or Alpha channel to C0.

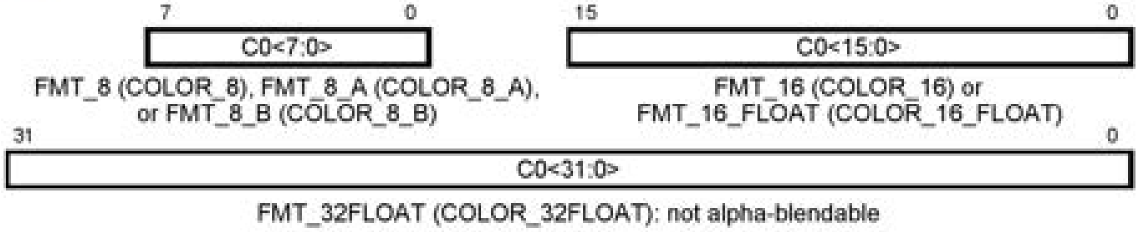


**Figure 2: One-Component Renderable Pixel Formats**

The next figure illustrates the two-component renderable pixel formats. Each format contains two equal size components, labeled C0 and C1. As for the single-component formats, the Render Backend cannot render to 32-bit components unless they are floating point and cannot alpha blend even floating point 32-bit components. The Render Backend can map either the shader pipe GR or AR components to C1 and C0, in that order.

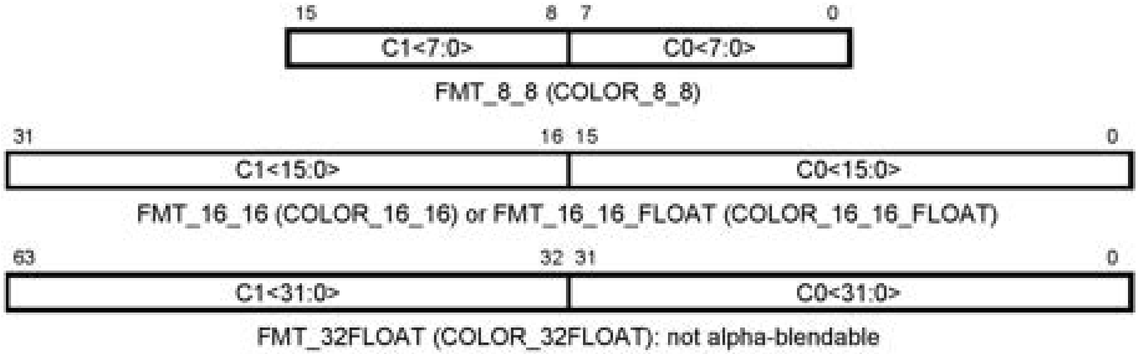


**Figure 3: Two-Component Renderable Pixel Formats**

The next figure illustrates the three-component renderable pixel formats. Each format contains two components of one size and one component that is one bit different in size, labeled C0, C1 and C2. The Render Backend can map either the shader pipe BGR or RGB components to C2, C1 and C0, in that order.

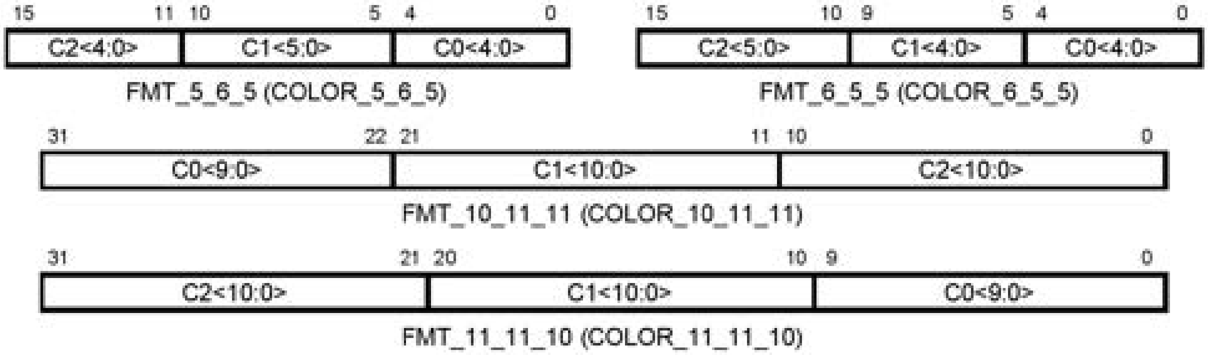


**Figure 4: Three-Component Renderable Pixel Formats**

The final figure illustrates the four-component renderable pixel formats. Two of the formats reduce the size of the C3 component and the rest provide an equal number of bits to each component. As for the two-component formats, the RB can render either floating-point or fixed-point 16-bit components, but only floating-point 32-bit components, and it cannot alpha-blend 32-bit components. The Render Backend can map either the shader pipe ABGR or ARGB components to C3, C2, C1 and C0, in that order.

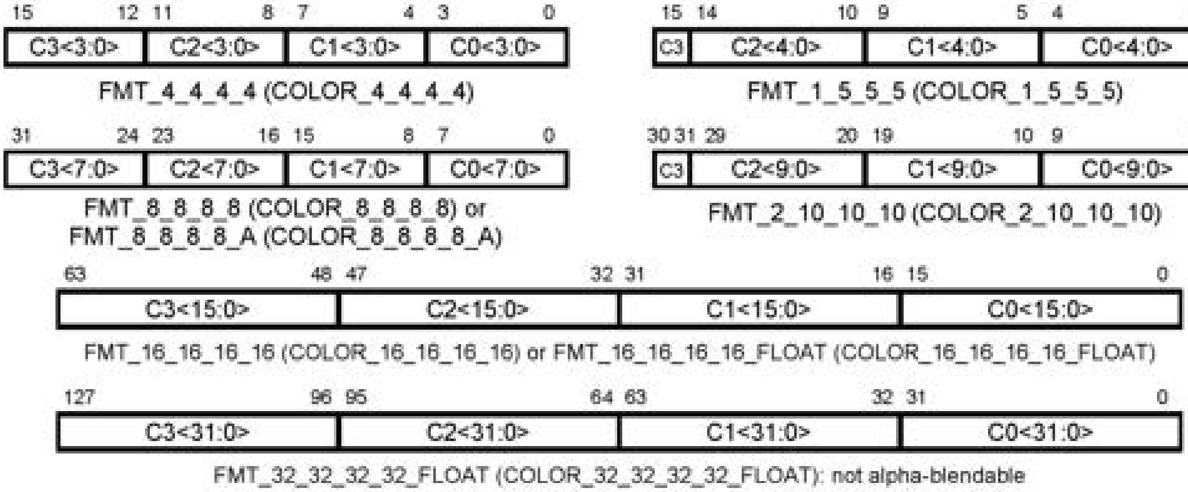
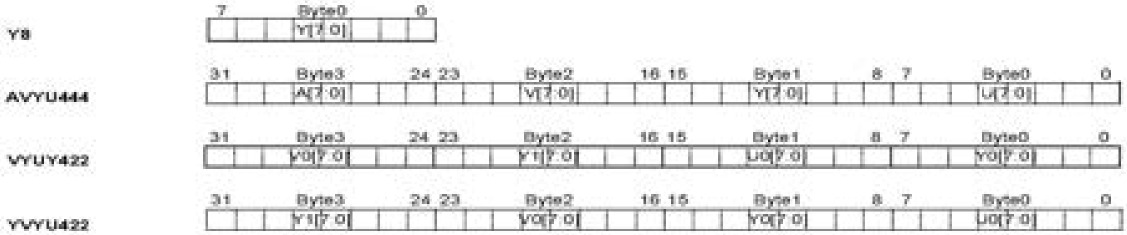


Figure 5: Four-Component Renderable Pixel Formats

## 2.3 Texel-Only Formats

The pixel formats described below are supported by the MH and TC but cannot be read or written by the RB. The MH passes them through without interpreting the bits in each pixel. The MH and TC also support the renderable pixel formats, which are described in the preceeding subsection.

**{Describe the YUV formats.}**



---

The DX formats provide texture compression. The bitmap formats store glyphs in one of several bit orders. **{provide more detail.}**

---

Each value in DXT1 format is treated as a 64-bit pixel that decodes to 4x4 texels. Each value in DXT2 to DXT5 format is treated as a 128-bit pixel that decodes to 4x4 texels. In linear format, N sequential DXTC pixels decode to a 4Nx4 region of texels. In tiled format, 64 sequential DCTC pixels form an 8x8 tile, which decodes to a 32x32 region of texels.

**<u>{Include the depth and depth/stencil formats.}</u>**

**{Describe the following formats:}**

| |
|---|
| 1 (1D only) |
| 1_REVERSE (1D only) |
| |

| |
|---|
| 16_MPEG |
| 16_16_MPEG |
| 8_INTERLACED |
| 16_INTERLACED (fixed)<br>16_INTERLACED (float)<br>16_INTERLACED (expand) |
| 32_AS_8_INTERLACED |
| 32_AS_8 |

## 2.4 Special Data Formats

This subsection describes arrayable data elements that have specific, limited purposes.

The Render Backend uses an array of Tile Data words, which store 32-bits for each 8x8 pixel tile. The Tile Data word stores compression and hierarchical information for each tile. The figure below illustrate the Tile Data word. The Cmask field stores the compression format for the Color0 buffer. The Zmask field stores the compression format for depth data in the Depth/Stencil buffer. The Smask field stores the compression format and hierarchical data for the stencil data in the Depth/Stencil buffer. Finally, the Zrange field encodes bounds on the minimum and maximum depth values in the tile for hierarchical depth kills.

| 31 12 | 11 8 | 7 4 | 3 0 |
|---|---|---|---|
| Zrange<19:0> | Smask<3:0> | Zmask<3:0> | Cmask<3:0> |

Figure 6: 32-Bit Tile Data Word for Render Backend

**{Document the depth formats}** Each16-bit pixel consists of a 16-bit repeating fraction depth value, which represents the range [0..1]. Each 32-bit pixel represents an 8-bit stencil value in the low order byte and a 24-bit depth value in the high bytes. The depth is either a 24-bit repeating fraction or a floating point representation with a 4-bit exponent, which represents the range [0..2), though values greater than 1 are not allowed.

{This includes all of the uncompressed formats that are not destination color formats, including bitmap formats, YUV formats, uncompressed depth/stencil values, etc.}

# 3. 1D Tiled Memory Formats

This section describes tiled memory formats for 1D arrays. In system memory, there is no difference between 1D tiled format and linear format. In local memory, the tiling format describes how the 1D data is interleaved across the memory subsets. For Local Tiled and System Tiled mode, the memory allocated for a 1D array must start and end on a 4Kbyte boundary. For System Linear mode, the array must start and end on a 64-byte boundary.

## 3.1 1D Micro-Tile Formats

The figure below shows 1D micro-tile formats within a 64-byte tile for 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit data elements. In each case, the tile size is 64-bytes (512-bits) or four 16-byte (128-bit) micro-tiles. Each row in the figure below is a single micro-tile.

| 127 | | | 96 | 95 | | | 64 | 63 | | | 32 | 31 | | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D 15 | D 14 | D 13 | D 12 | D 11 | D 10 | D 9 | D 8 | D 7 | D 6 | D 5 | D 4 | D 3 | D 2 | D 1 | D 0 |
| D 31 | D 30 | D 29 | D 28 | D 27 | D 26 | D 25 | D 24 | D 23 | D 22 | D 21 | D 20 | D 19 | D 18 | D 17 | D 16 |
| D 47 | D 46 | D 45 | D 44 | D 43 | D 42 | D 41 | D 40 | D 39 | D 38 | D 37 | D 36 | D 35 | D 34 | D 33 | D 32 |
| D 63 | D 62 | D 61 | D 60 | D 59 | D 58 | D 57 | D 56 | D 55 | D 54 | D 53 | D 52 | D 51 | D 50 | D 49 | D 48 |

8-bit data

| 127 | 96 | 95 | 64 | 63 | 32 | 31 | 0 |
|---|---|---|---|---|---|---|---|
| D 7 | D 6 | D 5 | D 4 | D 3 | D 2 | D 1 | D 0 |
| D 15 | D 14 | D 13 | D 12 | D 11 | D 10 | D 9 | D 8 |
| D 23 | D 22 | D 21 | D 20 | D 19 | D 18 | D 17 | D 16 |
| D 31 | D 30 | D 29 | D 28 | D 27 | D 26 | D 25 | D 24 |

16-bit data

| 127 96 | 95 64 | 63 32 | 31 0 |
|---|---|---|---|
| Data 3 | Data 2 | Data 1 | Data 0 |
| Data 7 | Data 6 | Data 5 | Data 4 |
| Data 11 | Data 10 | Data 9 | Data 8 |
| Data 15 | Data 14 | Data 13 | Data 12 |

32-bit data

| 127 96 95 64 | 63 32 31 0 |
|---|---|
| Data 1 | Data 0 |
| Data 3 | Data 2 |
| Data 5 | Data 4 |
| Data 7 | Data 6 |

64-bit data

| 127 96 95 64 63 32 31 0 |
|---|
| Data 0 |
| Data 1 |
| Data 2 |
| Data 3 |

128-bit data

**Figure 7: 1D Micro-Tile Data Formats**

The texture unit also supports packed 1-bit pixel formats for use as text fonts. {Draw a figure describing these formats, including the bit order.}

to be specified

Figure 8: 1D Micro-Tile 1-bit Data Formats

Finally, note that bytes are stored in local memory packed from lsb to msb of successive data elements, that is, little-endian order. Three byte swap reorderings are supported for data stored in system memory. R400 automatically converts between the different byte swap modes, based on a field that is stored with the offset for each surface.

## 3.2 1D Macro-Tile Formats

The following figure shows the organization of tiles within a macro-tile for 1D arrays. Each 1D macro-tile occupies 4K-bytes. The left-hand figure shows the organization in system memory, which is simply a linear sequence of 64 64-byte tiles. The column in the center of the figure gives the byte address relative to to the start of the macro-tile, for each tile.

| system memory | one MC with two memory subsets | | two MCs with four memory subsets | four MCs with eight memory subsets |
|---|---|---|---|---|
| tile 0 | ab, tile 0 | 0x000 | ab0, tile 0 | ab0, tile 0 |
| tile 1 | ab, tile 1 | 0x040 | ab1, tile 0 | ab1, tile 0 |
| tile 2 | ab, tile 2 | 0x080 | ab0, tile 1 | ab2, tile 0 |
| tile 3 | ab, tile 3 | 0x0C0 | ab1, tile 1 | ab3, tile 0 |
| . . . | . . . | . . . | . . . | . . . |
| tile 30 | ab, tile 30 | 0x780 | ab0, tile 15 | ab2, tile 7 |
| tile 31 | ab, tile 31 | 0x7C0 | ab1, tile 15 | ab3, tile 7 |
| tile 32 | cd, tile 0 | 0x800 | cd0, tile 0 | cd0, tile 0 |
| tile 33 | cd, tile 1 | 0x840 | cd1, tile 0 | cd1, tile 0 |
| tile 34 | cd, tile 2 | 0x880 | cd0, tile 1 | cd2, tile 0 |
| tile 35 | cd, tile 3 | 0x8C0 | cd1, tile 1 | cd3, tile 0 |
| . . . | . . . | . . . | . . . | . . . |
| tile 62 | cd, tile 30 | 0xF80 | cd0, tile 15 | cd2, tile 7 |
| tile 63 | cd, tile 31 | 0xFC0 | cd1, tile 15 | cd3, tile 7 |

Figure 9: 1D Macro-Tile Formats

The remainder of the above figure shows the arrangement of micro-tiles in local memory for 2-8 memory subsets (1-4 memory controllers). Tiles are numbered separately within each memory subset. These formats spread sequential array accesses evenly across the memory controllers at a relatively fine granularity. Within a memory controller, these formats produce a burst within one memory subset before switching to the other memory subset. The goal is to produce a large enough burst within a bank to cover the time required to open a page in a different bank, while keeping the bursts small enough to reduce the buffering required to spread sequential accesses across all of the memory controllers.

The linear array form can also be used for 2D or 3D arrays. A 2D or 3D linear arrays is first converted to a 1D array by computing *Index = X + Y*Pitch + Z*Height*Pitch*. R400 can read texture maps from 2D and 3D linear arrays and can write to them for bitblts. R400 does not support rendering (alpha blending and/or depth buffering) to 2D or 3D arrays that are stored in linear format.

## 3.3 1D Address Equations

This subsection presents equations for computing addresses in a 1D array. These are also used in computing 2D and 3D array addresses (subsections 4.4 and 5.3). These equations are also implemented in address conversion library code (address.h and address.c). Boldface represents names of parameters that are used in the C library.

The first list below defines parameters that are constant for a given surface. *Size* and *Subsets* may be derived from ***DataSize*** and ***Pipes***, but are defined separately to simplify the equations. ***SurfaceBase*** is the byte address of the start of the surface, which must be 4K-byte aligned. ***SurfaceBase*** may be expressed relative to the start of the entire $2^{32}$-byte device address space or within a subrange of the complete device address space, provided that the subrange is also 4K-byte aligned.

| | |
|---|---|
| Size | Bytes per pixel: can be 1, 2, 4, 8, or 16 (or fractions of a byte for non-pixel data) |
| **DataSize** | 64 times Size, equals the total bytes of data in a 2D or 3D tile |
| **Pipes** | Total number of Render Backend/Memory Controller pipelines: 1, 2, or 4 |
| Subsets | Total number of memory subsets: equals twice the number of pipelines |
| **SurfaceBase** | Byte address of pixel zero in device address space or subrange, must be 4K-byte aligned |

The second list below names parameters that depend on which pixel is accessed in the 1D array. 1D address equations use the parameters in the first list and one or more of the parameters in the second list to define the remaining parameters in the second list. *MemSelect* and *BankSelect* may be derived from ***Subset***, or vice versa. *MacroNumber*, *TileNumber*, and *TileAddr* are temporary values used in computing the other parameters.

| | |
|---|---|
| **Index** | Pixel index into the array |
| Byte**Addr** | Byte address of the pixel in device address space (or a 4K-byte aligned subrange) |
| **Local**Addr | Byte address of the pixel within its memory subset, starting at byte 0 in the address range |
| MemSelect | Number of the memory controller that stores this pixel |
| BankSelect | 0 for banks AB or 1 for banks CD, together with MemSelect determines the memory subset |
| **Subset** | Subset number, which equals BankSelect + 2*MemSelect |
| MacroNumber | Sequential number of the macro-tile containing the pixel, starting from device address 0 |
| TileNumber | Sequential number of the tile containing the pixel, within its memory subset and its macro-tile |
| TileAddr | Byte address of the pixel within its tile, which is entirely contained in a single memory subset |

The following equations use ***Index*** to compute the other address terms, particularly *Byte**Addr***. This is used to convert an array access into a device address. Typically ***LocalAddr*** is not required as part of this step, but it is included for completeness.

```
TileAddr      = (Index*Size) mod 64;             // 64 bytes per tile
TileNumber    = ((Index*Size/64) mod 32)  / Pipes;   // cycle through MCs within each half-macro-tile
MacroNumber   = (Index*Size + SurfaceBase) / 4096;   // 4096 bytes per macro-tile
MemSelect     = (Index*Size/64) mod Pipes;       // cycle through MCs each 64 bytes
BankSelect    = (Index*Size/2048) mod 2;         // CD banks are in high half of each macro-tile
Subset        = BankSelect + 2*MemSelect;        // subset number
ByteAddr      = SurfaceBase + Index*Size;
LocalAddr     = MacroNumber*4096/Subsets + TileAddr + TileNumber*64;
```

The following equations use *Byte**Addr*** to compute the other address terms, particularly ***Local**Addr* and ***Subset***. This is used to convert a device address into a local memory address within a particular memory subset. Typically ***Index*** is not required as part of this step, but it is included for completeness.

TileAddr = ByteAddr mod 64; // 64 bytes per tile
TileNumber = ((ByteAddr/64) mod 32) / Pipes; // cycle through MCs within each half-macro-tile
MacroNumber = (ByteAddr) / 4096; // 4096 bytes per macro-tile
MemSelect = (ByteAddr/64) mod Pipes; // cycle through MCs each 64 bytes
BankSelect = (ByteAddr/2048) mod 2; // CD banks are in high half of each macro-tile
**Subset** = BankSelect + 2*MemSelect; // subset number
**Index** = (ByteAddr – SurfaceBase) / Size;
**Local**Addr = MacroNumber*4096/Subsets + TileAddr + TileNumber*64;

The final set of equations use ***Local**Addr* and ***Subset*** to compute the other address terms, particularly *Byte**Addr***. The equations for 2D and 3D arrays use (*X*,*Y*) addresses to produce ***Local**Addr* and ***Subset***, which these equations convert into device addresses. Typically only *Byte**Addr*** is required as the result of this step but the others are provided for completeness.

MemSelect = Subset / 2;
BankSelect = Subset mod 2;
TileAddr = LocalAddr mod 64; // 64 bytes per tile
TileNumber = (LocalAddr /64) mod (64/Subsets); // 64 tiles in a macro-tile, over all the subsets
MacroNumber = LocalAddr * Subsets / 4096; // 4096 bytes per macro-tile
Byte**Addr** = 4096*MacroNumber+2048*BankSelect+32*Subsets*TileNumber +64*MemSelect+TileAddr;
**Index** = (ByteAddr – SurfaceBase) / Size;

# 4. 2D Tiled Memory Formats

This section describes how the R400 family stores tiled 2D data arrays. Each tile contains an 8x8 array of data elements. Each 8x8 tile has a micro-tile format that depends on the data element size. Each 2D macro-tile contains a 4x4 array of tiles, which covers 32x32 pixels. This is different from 1D formats, where each tile and macro-tile stores a fixed number of bytes. Like the 1D formats, each 2D tiled surface must start on a 4K-byte boundary.

## 4.1 2D Micro-Tile Formats

R400 arranges pixels within 8x8 tiles in order to meet two conflicting goals. First, sequential accesses from memory should contain pixels from a roughly square region within the tile. This improves efficiency by a modest amount, due to rendering locality. Second, display updates must be efficient with only one line buffer. This implies that each 256-bit memory access should contain pixels from only two scanlines.

To meet these goals, R400 stores even scanlines of the 8x8 tile in even-numbered micro-tiles and stores odd scanlines in odd-numbered micro-tiles. The figure below shows the order of micro-tiles within an 8x8 tile for the five pixel sizes. For 128-bit pixels, each micro-tile covers a single pixel. For 8-bit pixels, each 256-bit access includes pixels from four different scanlines. This requires display accesses to throw away half of the 8-bit data that it reads, but this loss of efficiency is acceptable for 8-bit pixels. For all other pixel sizes, a 256-bit access reads from just two scanlines.

| | |
|---|---|
| Micro-Tile 0 | <63:0> |
| Micro-Tile 1 | <63:0> |
| Micro-Tile 0 | <127:64> |
| Micro-Tile 1 | <127:64> |
| Micro-Tile 2 | <63:0> |
| Micro-Tile 3 | <63:0> |
| Micro-Tile 2 | <127:64> |
| Micro-Tile 3 | <127:64> |

8-bit Pixels

| |
|---|
| Micro-Tile 0 |
| Micro-Tile 1 |
| Micro-Tile 2 |
| Micro-Tile 3 |
| Micro-Tile 4 |
| Micro-Tile 5 |
| Micro-Tile 6 |
| Micro-Tile 7 |

16-bit Pixels

| | |
|---|---|
| MT 0 | MT 2 |
| MT 1 | MT 3 |
| MT 4 | MT 6 |
| MT 5 | MT 7 |
| MT 8 | MT 10 |
| MT 9 | MT 11 |
| MT 12 | MT 14 |
| MT 13 | MT 15 |

32-bit Pixels

| | | | |
|---|---|---|---|
| 0 | 2 | 4 | 6 |
| 1 | 3 | 5 | 7 |
| 8 | 10 | 12 | 14 |
| 9 | 11 | 13 | 15 |
| 16 | 18 | 20 | 22 |
| 17 | 19 | 21 | 23 |
| 24 | 26 | 28 | 30 |
| 25 | 27 | 29 | 31 |

64-bit Pixels

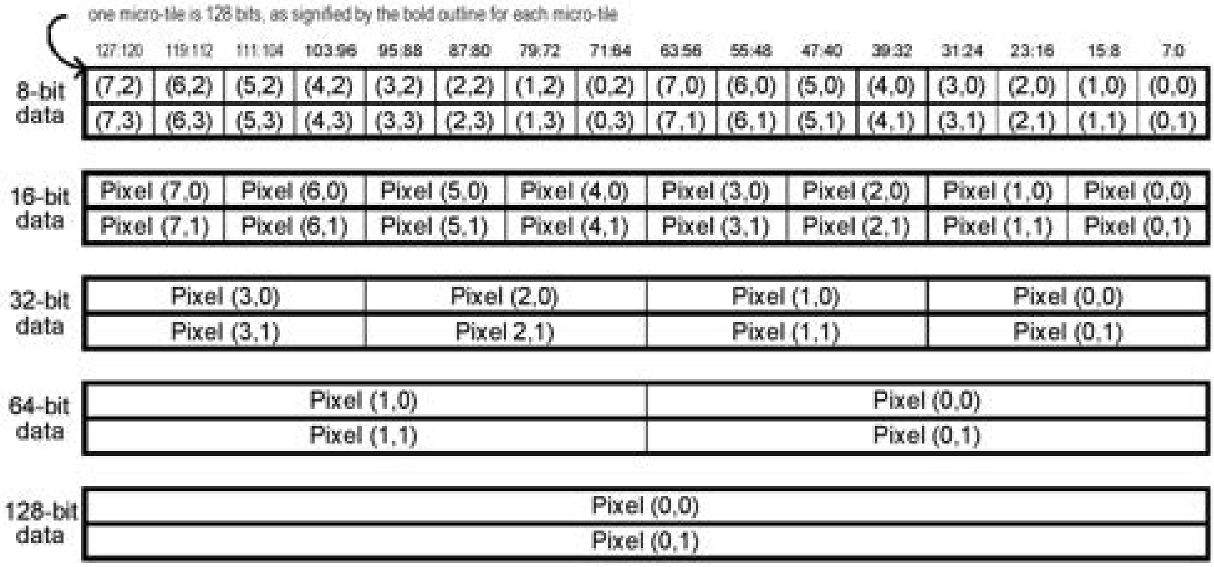
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 |
| 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
| 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
| 17 | 19 | 21 | 23 | 25 | 27 | 29 | 31 |
| 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 |
| 33 | 35 | 37 | 39 | 41 | 43 | 45 | 47 |
| 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 |
| 49 | 51 | 53 | 55 | 57 | 59 | 61 | 63 |

128-bit Pixels

**Figure 10: 2D Micro-tile Layout Within Tiles**

For 32-bit and larger pixels, the patterns above map each 256-bit access to a 1:1 or 2:1 region within the tile, therefore meeting the square region criterion. 16-bit pixels have a less efficient 4:1 region, but this is acceptable since smaller pixels require less memory bandwidth. Smaller pixels are also less important in the R400 timeframe. This micro-tile format is efficient enough that it is used for all uncompressed 2D pixel and texel arrays, even though texels do not need to be displayed to the screen.

The figure below shows the (x,y) address of the pixels or texels inside the 16-byte (128-bit) micro-tiles. Only the first two micro-tiles are shown for each pixel size. Except for 8-bit pixels, these micro-tiles only include data from the first two rows of the tile. For 8-bit pixels, they cover the first four rows of the 8x8 tile.



one micro-tile is 128 bits, as signified by the bold outline for each micro-tile

| | 127:120 | 119:112 | 111:104 | 103:96 | 95:88 | 87:80 | 79:72 | 71:64 | 63:56 | 55:48 | 47:40 | 39:32 | 31:24 | 23:16 | 15:8 | 7:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8-bit data | (7,2) | (6,2) | (5,2) | (4,2) | (3,2) | (2,2) | (1,2) | (0,2) | (7,0) | (6,0) | (5,0) | (4,0) | (3,0) | (2,0) | (1,0) | (0,0) |
| | (7,3) | (6,3) | (5,3) | (4,3) | (3,3) | (2,3) | (1,3) | (0,3) | (7,1) | (6,1) | (5,1) | (4,1) | (3,1) | (2,1) | (1,1) | (0,1) |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 16-bit data | Pixel (7,0) | Pixel (6,0) | Pixel (5,0) | Pixel (4,0) | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) |
| | Pixel (7,1) | Pixel (6,1) | Pixel (5,1) | Pixel (4,1) | Pixel (3,1) | Pixel (2,1) | Pixel (1,1) | Pixel (0,1) |

| | | | | |
|---|---|---|---|---|
| 32-bit data | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) |
| | Pixel (3,1) | Pixel 2,1) | Pixel (1,1) | Pixel (0,1) |

| | | |
|---|---|---|
| 64-bit data | Pixel (1,0) | Pixel (0,0) |
| | Pixel (1,1) | Pixel (0,1) |

| | |
|---|---|
| 128-bit data | Pixel (0,0) |
| | Pixel (0,1) |

**Figure 11: Pixel Format within 2D Micro-Tiles**

## 4.2 2D Macro-Tile Formats

Each 2D macro-tile stores a 32x32 array of pixels, organized into 16 8x8 tiles. The macro-tile format depends on the number of memory subsets, which is twice the number of memory controllers. The following figure shows the layout of 8x8 tiles within 32x32 macro-tiles for 1, 2, and 4 memory controllers. Bold lines mark 32x32 macro-tiles. Light lines mark 8x8 tiles. The upper line of text in each 8x8 tile specifies the memory subset and the lower line of text specifies the order of the tiles within their memory subset.

The three macro-tile formats have several properties in common. First, each macro-tile allocates an equal number of 8x8 tiles to each memory subset, which makes it simpler to allocate memory. Second, tile addresses in memory increase from left to right within each macro-tile and between macro-tiles on the same scanline. Finally, moving vertically by one macro-tile increments the tile address by a value *L*, which is equal to the pitch (line length) in pixels, divided by four times the number of memory controllers. The pitch must be a multiple of 32 pixels.

one MC with two memory subsets

| | | | |
|---|---|---|---|
| ab 0 | ab 2 | ab 4 | ab 6 |
| ab 1 | ab 3 | ab 5 | ab 7 |
| cd 0 | cd 2 | cd 4 | cd 6 |
| cd 1 | cd 3 | cd 5 | cd 7 |
| ab L | ab 2+L | ab 4+L | ab 6+L |
| ab 1+L | ab 3+L | ab 5+L | ab 7+L |
| cd L | cd 2+L | cd 4+L | cd 6+L |
| cd 1+L | cd 3+L | cd 5+L | cd 7+L |

two MCs with four memory subsets

| | | | |
|---|---|---|---|
| ab0 0 | ab1 0 | ab0 2 | ab1 2 |
| ab1 1 | ab0 1 | ab1 3 | ab0 3 |
| cd0 0 | cd1 0 | cd0 2 | cd1 2 |
| cd1 1 | cd0 1 | cd1 3 | cd0 3 |
| ab0 L | ab1 L | ab0 2+L | ab1 2+L |
| ab1 1+L | ab0 1+L | ab1 3+L | ab0 3+L |
| cd0 L | cd1 L | cd0 2+L | cd1 2+L |
| cd1 1+L | cd0 1+L | cd1 3+L | cd0 3+L |

four MCs with eight memory subsets

| | | | |
|---|---|---|---|
| ab0 0 | ab1 0 | ab2 0 | ab3 0 |
| ab2 1 | ab3 1 | ab0 1 | ab1 1 |
| cd0 0 | cd1 0 | cd2 0 | cd3 0 |
| cd2 1 | cd3 1 | cd0 1 | cd1 1 |
| ab0 L | ab1 L | ab2 L | ab3 L |
| ab2 1+L | ab3 1+L | ab0 1+L | ab1 1+L |
| cd0 L | cd1 L | cd2 L | cd3 L |
| cd2 1+L | cd3 1+L | cd0 1+L | cd1 1+L |

**Figure 12: 2D Macro-Tile Mappings**

Another common property is that if there are N memory controllers, then each Nx1 row of tiles places a tile in each memory controller. This is necessary for efficient display accesses. The display reads across rows of 8x8 tiles and sometimes requires a significant fraction of the total memory bandwidth. Alternating between the memory controllers allows the display to spread its bandwidth equally between them. This also makes it more efficient to render large primitives, since the Scan Converter steps horizontally before it steps vertically. The bank alternation within a memory subset ensures that page crossings do not occur while rendering horizontal swaths of pixels.

The remaining property that the macro-tile formats have in common is that the upper half of each macro-tile uses bank AB memory subsets and the lower half uses bank CD memory subsets. This reduces page crossings when rendering vertical swaths of pixels. A vertical line first touches two tiles in AB subsets, followed by two tiles in CD subsets. The next tile is once again in an AB subset and could be on a different page of the same bank as the initial accesses. Interspersing the CD subset accesses makes it more likely that the Memory Controller will have accesses to perform while waiting for the bank to become ready. However, vertical motion is not as efficient as horizontal motion in these macro-tile formats.

Finally, note that the first memory controller on odd rows of 8x8 tiles is offset by 1 for two memory controllers and is offset by two for four memory controllers. This has the effect that each 2x2 block of tiles hits each memory controller the same number of times. Putting together all of these properties, the 8x8 tiles nearest to any tile that are in the same memory controller are either on the same page of the same bank or are in a different bank. Further, with two or four memory controllers, moving horizontally or vertically to an adjacent tile also moves to a different memory controller.

## 4.3 Special 2D Micro-Tile Formats

The previous section describes micro-tile formats for standard pixel sizes. The Render Backend requires several additional pixel sizes for depth, stencil, and multifragment mask data. These pixels require special micro-tile formats that are only read and written by the Render Backend. Additionally, the Render Backend requires a micro-tile format that stores a single data element per tile, instead of a data element per pixel.

The following figure illustrates the micro-tile packing formats for non-standard pixel sizes. Like the standard micro-tile formats, these formats put even scanlines into even micro-tiles and odd scanlines into odd micro-tiles. The smallest allowed pixel size is 4-bits, since at that size an entire tile occupies one even and one odd micro-tile. The 4-bit and 12-bit formats do not permit byte masking of individual pixels. All but the 4-bit format cause pixels to cross micro-tile boundaries. All but the 48-bit format cause micro-tiles to touch multiple scanlines (as does the 8-bit micro-tile format). For these reasons, surfaces that use these formats are not readable or writable by software through the Memory Hub. They are only read and written by the Render Backend.

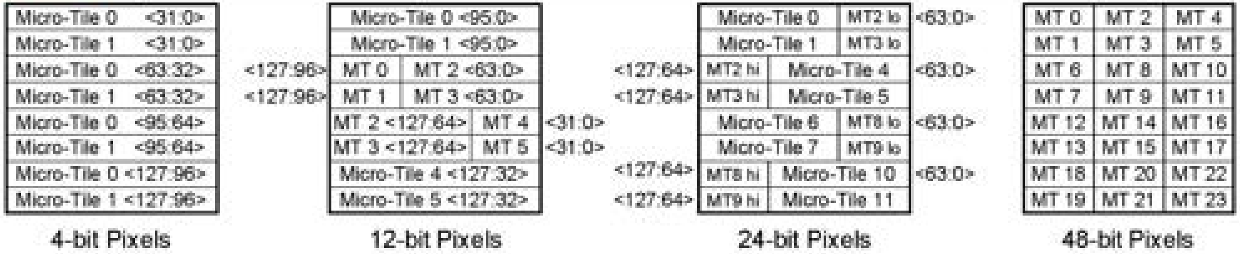
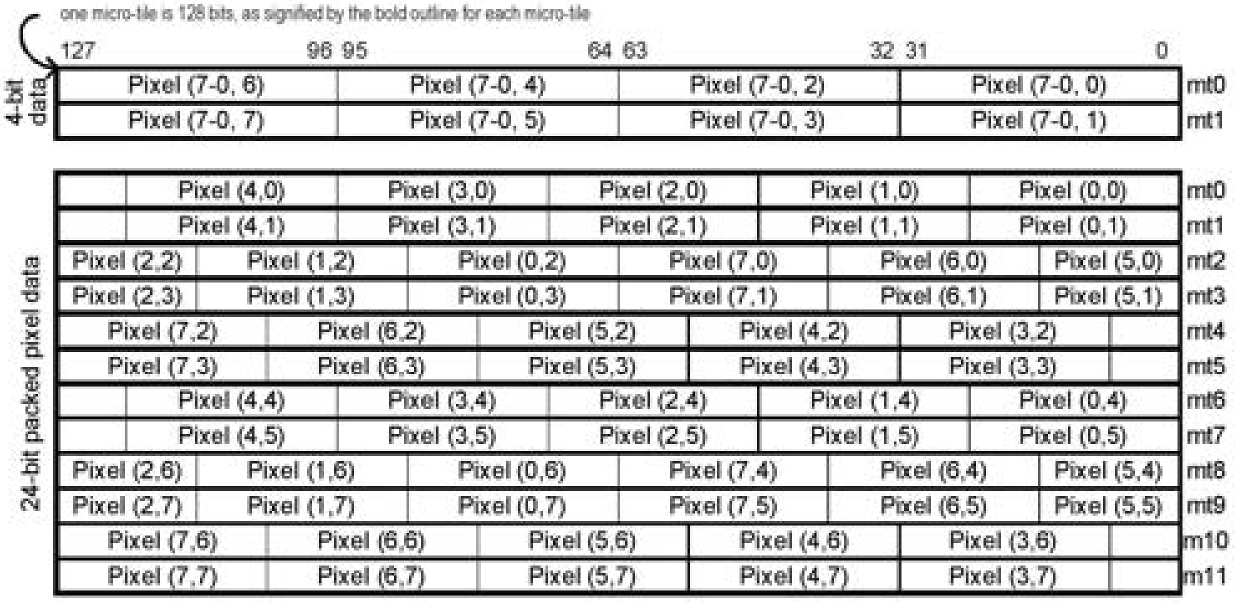


**Figure 13: 2D Micro-tiling for Nonstandard Pixels**

The figure below shows the (x,y) address of the pixels or texels inside the 16-byte (128-bit) micro-tiles for the 4-bit and 24-bit pixel sizes. Each row specifies a different micro-tile. 4-bit data occupies just two micro-tiles. 24-bit data requires 12 micro-tiles, with some pixels splitting across micro-tile boundaries. 12-bit pixels and 48-bit pixels require 6 and 24 micro-tiles, respectively.



| 127–96 | 95–64 | 63–32 | 31–0 | |
|---|---|---|---|---|
| Pixel (7-0, 6) | Pixel (7-0, 4) | Pixel (7-0, 2) | Pixel (7-0, 0) | mt0 |
| Pixel (7-0, 7) | Pixel (7-0, 5) | Pixel (7-0, 3) | Pixel (7-0, 1) | mt1 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | Pixel (4,0) | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) | mt0 |
| | Pixel (4,1) | Pixel (3,1) | Pixel (2,1) | Pixel (1,1) | Pixel (0,1) | mt1 |
| Pixel (2,2) | Pixel (1,2) | Pixel (0,2) | Pixel (7,0) | Pixel (6,0) | Pixel (5,0) | mt2 |
| Pixel (2,3) | Pixel (1,3) | Pixel (0,3) | Pixel (7,1) | Pixel (6,1) | Pixel (5,1) | mt3 |
| Pixel (7,2) | Pixel (6,2) | Pixel (5,2) | Pixel (4,2) | Pixel (3,2) | | mt4 |
| Pixel (7,3) | Pixel (6,3) | Pixel (5,3) | Pixel (4,3) | Pixel (3,3) | | mt5 |
| | Pixel (4,4) | Pixel (3,4) | Pixel (2,4) | Pixel (1,4) | Pixel (0,4) | mt6 |
| | Pixel (4,5) | Pixel (3,5) | Pixel (2,5) | Pixel (1,5) | Pixel (0,5) | mt7 |
| Pixel (2,6) | Pixel (1,6) | Pixel (0,6) | Pixel (7,4) | Pixel (6,4) | Pixel (5,4) | mt8 |
| Pixel (2,7) | Pixel (1,7) | Pixel (0,7) | Pixel (7,5) | Pixel (6,5) | Pixel (5,5) | mt9 |
| Pixel (7,6) | Pixel (6,6) | Pixel (5,6) | Pixel (4,6) | Pixel (3,6) | | m10 |
| Pixel (7,7) | Pixel (6,7) | Pixel (5,7) | Pixel (4,7) | Pixel (3,7) | | m11 |

**Figure 14: 4-bit and 24-bit Micro-Tile Formats**

Finally, data that is stored on a per-tile basis is not micro-tiled at all. Instead, each tile simply stores a specified number of bits. The Render Backend stores 32-bits per tile to record the tile's compression. In this case, the macro-tile format is exactly the same as described in the previous section, except that there may be fewer than 512-bits per tile. {**Note**: say a lot more about this.}

one MC with two memory subsets

| | | | |
|---|---|---|---|
| ab 0.0 | ab 0.2 | ab 1.0 | ab 1.2 |
| ab 0.1 | ab 0.3 | ab 1.1 | ab 1.3 |
| cd 0.0 | cd 0.2 | cd 1.0 | cd 1.2 |
| cd 0.1 | cd 0.3 | cd 1.1 | cd 1.3 |

two MCs with four memory subsets

| | | | |
|---|---|---|---|
| ab0 0.0 | ab1 0.0 | ab0 0.2 | ab1 0.2 |
| ab1 0.1 | ab0 0.1 | ab1 0.3 | ab0 0.3 |
| cd0 0.0 | cd1 0.0 | cd0 0.2 | cd1 0.2 |
| cd1 0.1 | cd0 0.1 | cd1 0.3 | cd0 0.3 |

four MCs with eight memory subsets

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ab0 0.0 | ab1 0.0 | ab2 0.0 | ab3 0.0 | ab0 0.2 | ab1 0.2 | ab2 0.2 | ab3 0.2 |
| ab2 0.1 | ab3 0.1 | ab0 0.1 | ab1 0.1 | ab2 0.3 | ab3 0.3 | ab0 0.3 | ab1 0.3 |
| cd0 0.0 | cd1 0.0 | cd2 0.0 | cd3 0.0 | cd0 0.2 | cd1 0.2 | cd2 0.2 | cd3 0.2 |
| cd2 0.1 | cd3 0.1 | cd0 0.1 | cd1 0.1 | cd2 0.3 | cd3 0.3 | cd0 0.3 | cd1 0.3 |

**Figure 15: 2D Macro-Tiling for 32-Bit Per-Tile Data**

## 4.4 Alternate 2D Macro-Tile Formats

There are three variations of the standard 2D macro-tile formats. These variations exist to improve performance for depth buffering and to allow a 2D depth buffer to be used in conjunction with a slice from a 3D color buffer. R400 does not support these alternate for display buffers, but they may be used for rendering, memory apertures, and texture mapping . {Check whether the 3D slice format actually gets supported for texture maps.} As with the figures in section , these figures show a box per 8x8 tile, with the upper line of text in each box listing the banks and RB number and the lower line of text giving the order in which the tiles are stored within their memory subset.

The figure below shows an alternate 2D tiling pattern that swaps the AB and CD bank assignment relative to the standard pattern that is described in section 4.2. This macro-tile pattern is particularly appropriate for depth buffers. If the same 2D tiling is used for both the depth buffer and the color buffer, then large area operations will tend to cause the Render Backend to read and write both of them in the AB banks or both of them in the CD banks. Using the bank-swapped alternate tiling illustrated below for the depth buffer increases the number of different banks that are likely to be open at the same time, thus increasing memory efficiency.

one MC with two memory subsets

| | | | |
|---|---|---|---|
| cd<br>0 | cd<br>2 | cd<br>4 | cd<br>6 |
| cd<br>1 | cd<br>3 | cd<br>5 | cd<br>7 |
| ab<br>0 | ab<br>2 | ab<br>4 | ab<br>6 |
| ab<br>1 | ab<br>3 | ab<br>5 | ab<br>7 |
| cd<br>L | cd<br>2+L | cd<br>4+L | cd<br>6+L |
| cd<br>1+L | cd<br>3+L | cd<br>5+L | cd<br>7+L |
| ab<br>L | ab<br>2+L | ab<br>4+L | ab<br>6+L |
| ab<br>1+L | ab<br>3+L | ab<br>5+L | ab<br>7+L |

two MCs with four memory subsets

| | | | |
|---|---|---|---|
| cd0<br>0 | cd1<br>0 | cd0<br>2 | cd1<br>2 |
| cd1<br>1 | cd0<br>1 | cd1<br>3 | cd0<br>3 |
| ab0<br>0 | ab1<br>0 | ab0<br>2 | ab1<br>2 |
| ab1<br>1 | ab0<br>1 | ab1<br>3 | ab0<br>3 |
| cd0<br>L | cd1<br>L | cd0<br>2+L | cd1<br>2+L |
| cd1<br>1+L | cd0<br>1+L | cd1<br>3+L | cd0<br>3+L |
| ab0<br>L | ab1<br>L | ab0<br>2+L | ab1<br>2+L |
| ab1<br>1+L | ab0<br>1+L | ab1<br>3+L | ab0<br>3+L |

four MCs with eight memory subsets

| | | | |
|---|---|---|---|
| cd0<br>0 | cd1<br>0 | cd2<br>0 | cd3<br>0 |
| cd2<br>1 | cd3<br>1 | cd0<br>1 | cd1<br>1 |
| ab0<br>0 | ab1<br>0 | ab2<br>0 | ab3<br>0 |
| ab2<br>1 | ab3<br>1 | ab0<br>1 | ab1<br>1 |
| cd0<br>L | cd1<br>L | cd2<br>L | cd3<br>L |
| cd2<br>1+L | cd3<br>1+L | cd0<br>1+L | cd1<br>1+L |
| ab0<br>L | ab1<br>L | ab2<br>L | ab3<br>L |
| ab2<br>1+L | ab3<br>1+L | ab0<br>1+L | ab1<br>1+L |

**Figure 16: Bank-Swapped 2D Macro-Tile Mappings**

A 2D depth buffer may be used with a single slice of a 3D color buffer. This requires a 2D tiling pattern that maps each pixel to the same RB that it is mapped to in the 3D tiling pattern. Section 5.2 describes 3D macro-tiling patterns, which map pixel in an (x,y) column to one of two RBs, depending on the value of Z. One of the two RB assignments matches the RB assignments in the standard 2D macro-tiling pattern. The other RB assignment swaps the RB numbers. The swapped 2D macro-tiling pattern below matches this alternate RB mapping for 3D slices. A 2D depth surface may be used with slices of a 3D color array by selecting either the standard or this swapped macro-tiling pattern, depending on the slice selected from the 3D array.

one MC with two memory subsets

| | | | |
|---|---|---|---|
| ab 0 | ab 2 | ab 4 | ab 6 |
| ab 1 | ab 3 | ab 5 | ab 7 |
| cd 0 | cd 2 | cd 4 | cd 6 |
| cd 1 | cd 3 | cd 5 | cd 7 |
| ab L | ab 2+L | ab 4+L | ab 6+L |
| ab 1+L | ab 3+L | ab 5+L | ab 7+L |
| cd L | cd 2+L | cd 4+L | cd 6+L |
| cd 1+L | cd 3+L | cd 5+L | cd 7+L |

two MCs with four memory subsets

| | | | |
|---|---|---|---|
| ab1 0 | ab0 0 | ab1 2 | ab0 2 |
| ab0 1 | ab1 1 | ab0 3 | ab1 3 |
| cd1 0 | cd0 0 | cd1 2 | cd0 2 |
| cd0 1 | cd1 1 | cd0 3 | cd1 3 |
| ab1 L | ab0 L | ab1 2+L | ab0 2+L |
| ab0 1+L | ab1 1+L | ab0 3+L | ab1 3+L |
| cd1 L | cd0 L | cd1 2+L | cd0 2+L |
| cd0 1+L | cd1 1+L | cd0 3+L | cd1 3+L |

four MCs with eight memory subsets

| | | | |
|---|---|---|---|
| ab2 0 | ab3 0 | ab0 0 | ab1 0 |
| ab0 1 | ab1 1 | ab2 1 | ab3 1 |
| cd2 0 | cd3 0 | cd0 0 | cd1 0 |
| cd0 1 | cd1 1 | cd2 1 | cd3 1 |
| ab2 L | ab3 L | ab0 L | ab1 L |
| ab0 1+L | ab1 1+L | ab2 1+L | ab3 1+L |
| cd2 L | cd3 L | cd0 L | cd1 L |
| cd0 1+L | cd1 1+L | cd2 1+L | cd3 1+L |

**Figure 17: RB-Swapped 2D Macro-Tile Mappings**

The final new macro-tiling pattern is a combination of the preceeding two. The macro-tilings illustrated above and below allow selecting a 2D pattern that either does or does not swap the AB and CD banks relative to the 3D slices that do not match the standard 2D macro-tiling. The 3D macro-tiling described in section 5.2 matches the pattern below because it swaps both RBs and banks. So the pattern above, that swaps just the RBs, should be used to cause a 2D depth surface to use a different bank for each tile than the 2D tile pattern uses.

one MC with two memory subsets

| | | | |
|---|---|---|---|
| cd 0 | cd 2 | cd 4 | cd 6 |
| cd 1 | cd 3 | cd 5 | cd 7 |
| ab 0 | ab 2 | ab 4 | ab 6 |
| ab 1 | ab 3 | ab 5 | ab 7 |
| cd L | cd 2+L | cd 4+L | cd 6+L |
| cd 1+L | cd 3+L | cd 5+L | cd 7+L |
| ab L | ab 2+L | ab 4+L | ab 6+L |
| ab 1+L | ab 3+L | ab 5+L | ab 7+L |

two MCs with four memory subsets

| | | | |
|---|---|---|---|
| cd1 0 | cd0 0 | cd1 2 | cd0 2 |
| cd0 1 | cd1 1 | cd0 3 | cd1 3 |
| ab1 0 | ab0 0 | ab1 2 | ab0 2 |
| ab0 1 | ab1 1 | ab0 3 | ab1 3 |
| cd1 L | cd0 L | cd1 2+L | cd0 2+L |
| cd0 1+L | cd1 1+L | cd0 3+L | cd1 3+L |
| ab1 L | ab0 L | ab1 2+L | ab0 2+L |
| ab0 1+L | ab1 1+L | ab0 3+L | ab1 3+L |

four MCs with eight memory subsets

| | | | |
|---|---|---|---|
| cd2 0 | cd3 0 | cd0 0 | cd1 0 |
| cd0 1 | cd1 1 | cd2 1 | cd3 1 |
| ab2 0 | ab3 0 | ab0 0 | ab1 0 |
| ab0 1 | ab1 1 | ab2 1 | ab3 1 |
| cd2 L | cd3 L | cd0 L | cd1 L |
| cd0 1+L | cd1 1+L | cd2 1+L | cd3 1+L |
| ab2 L | ab3 L | ab0 L | ab1 L |
| ab0 1+L | ab1 1+L | ab2 1+L | ab3 1+L |

**Figure 18: Dual-Swapped 2D Macro-Tile Mappings**

## 4.5 2D Address Equations

This subsection presents equations for computing addresses in a 2D array. This is a two-step process that makes use of the 1D array equations of subsection 3.3. The first list below defines parameters that are constant for a given surface. The second list names parameters that depend on which pixel is accessed in the array. 2D address equations use the parameters in the first list and one or more of the parameters in the second list to define the remaining parameters in the second list.

| | |
|---|---|
| Size | Bytes per pixel: can be 1, 2, 4, 8, or 16 (or 1/8 for 1-bit pixels) |
| **DataSize** | 64 times Size, equals the total bytes of data in a 2D tile |
| **Pipes** | Total number of Render Backend/Memory Controller pipelines: 1, 2, or 4 |

| | |
|---|---|
| Subsets | Total number of memory subsets: equals twice the number of pipelines |
| **SurfaceBase** | Byte address of pixel zero in device address space, must be 4K-byte aligned |
| **TileSize** | Bytes per tile: equals 64*Size, except for special tile formats that contain multiple pixel arrays |
| **TileBase** | Byte address of first pixel in a tile: normally zero, a multiple of 64 for special tile formats |
| **Pitch** | The width of each scanline in pixels, must be a multiple of 32 |
| **AltBank** | Boolean that selects alternate 2D macro-tile pattern that exchanges banks AB and CD |
| **SwapRB** | Boolean that selects swapping the RB numbers in the 2D macro-tile pattern |
| | |
| **X, Y** | Pixel location in the 2D array |
| **Local**Addr | Byte address of the pixel within its memory subset, starting from device address 0 |
| SubsetOffset | Byte address of the pixel within its memory subset, starting from pixel (0,0) of the surface |
| MemSelect | Number of the memory controller that stores this pixel |
| BankSelect | 0 for banks AB or 1 for banks CD, together with MemSelect determines the memory subset |
| **Subset** | Subset number, equals BankSelect + 2*MemSelect |
| MacroOffset | Sequential number of the macro-tile containing the pixel, starting from SurfaceBase |
| TileNumber | Sequential number of the tile containing the pixel, within its memory subset and its macro-tile |
| TileAddr | Byte address of the pixel within its tile, starting from the first byte of the tile |
| TileOffset | Byte address of the pixel within its tile, relative to TileBase (which is normally zero) |

The following equations use ***X*** and ***Y*** to compute the other address terms, particularly ***Local****Addr* and ***Subset***. The final set of equations in subsection 3.3 uses these results to produce a device address.

```
BankSelect    = ((Y/16) mod 2) ^ AltBank;              // Banks change for high/low half of each macro-tile
MemSelect     = (X/8 + ((Y/8 mod 2)^SwapRB)*(Pipes/2)) mod Pipes;    // Offset memory in alternate rows
Subset        = BankSelect + 2*MemSelect;                   // subset number
TileNumber    = ((X mod 32)/8/Pipes)*2 + (Y/8 mod 2);       // Odd tile numbers are in odd rows
MicroByte     = (X mod 8 + ((Y mod 8)/2)*8)*Size            // Byte address within tile for even scanlines
                                                       // Odd scanlines get odd micro-tiles within an 8x8 tile
TileOffset    = (MicroByte mod 16) + (Y mod 2)*16 + (MicroByte/16)*32;
TileAddr      = TileBase + TileOffset;                      // The tile may contain other data as well
MacroOffset   = (X/32) + (Y/32) * (Pitch/32);               // There are Pitch/32 macro-tiles per row
SubsetOffset  = MacroOffset*TileSize*16/Subsets + TileNumber*TileSize + TileAddr;
LocalAddr     = SurfaceBase/Subsets + SubsetOffset;
```

The following equations use ***AltBank***, ***SwapRB***, ***Local****Addr* and ***Subset*** to compute the other address terms, particularly the (***X***, ***Y***) array address. The final set of equations in subsection 3.3 convert a device address into ***Local****Addr* and ***Subset*** and these equations complete the conversion to an (***X***, ***Y***) array address.

```
MemSelect     = Subset / 2;
BankSelect    = Subset mod 2;
SubsetOffset  = LocalAddr – SurfaceBase/Subsets;            // subset address in surface
TileAddr      = SubsetOffset mod TileSize;                  // byte address within the tile
TileOffset    = TileAddr - TileBase;                        // byte address within subset of the tile
MacroOffset   = SubsetOffset * Subsets / 16 / TileSize;     // 16*TileSize bytes per macro-tile
TileNumber    = SubsetOffset/TileSize mod (16/Subsets);     // 16 8x8 tiles per macro-tile over all subsets
MicroByte     = TileOffset mod 16 + (TileOffset/32)*16; // byte address within even micro-tiles
Ymacro        = MacroOffset*32/Pitch;                       // Macro-tile offset vertically
Xmacro        = MacroOffset mod Pitch/32;                   // macro-tile offset horizontally
Ytile         = (BankSelect^AltBank)*2 + (TileNumber mod 2);// tile 0, 1, 2, or 3 vertically in macro-tile
Xtile         = (MemSelect + ((Y/8 mod 2) ^SwapRB)*Pipes/2) mod Pipes + (TileNumber/2)*Pipes;
Ymicro        = ((MicroByte mod 16)/8 + (TileOffset/32)*2) / Size;        // row pair in tile due to micro-tiling
Xbyte         = MicroByte mod (8*Size);                     // byte address within first 8x1 scanline
Y             = Ymacro*32 + Ytile*8 + Ymicro*2 + (TileOffset/16 mod 2);
X             = Xmacro*32 + Xtile*8 + Xbyte/Size;
```

Note that these equations also work for non-standard pixel sizes and per-tile data. For 4-bit or 24-bit pixels, set Size to ½ or 3, set *TileSize* to the number of bytes in the tile, e.g. 64*32 and set *TileBase* to the starting byte in the tile for the pixel data, e.g. 0 or 8*64. For per-tile data, set *TileSize* to the number of bytes of data per tile and set *X* and *Y* to multiples of 8, that is, the lowest pixel address for the specified tile. This forces *MacroOffset* to zero, which causes *Size* to be ignored.

# 5. 3D Tiled Memory Formats

This section describes how the R400 family stores tiled 3D data arrays. Each tile contains an 8x8x1 array of data elements. Each 8x8x1 tile has a micro-tile format that depends on the data element size. Each 3D macro-tile contains a 4x1x4 array of tiles, which covers 32x8x4 pixels. This is different from 1D formats, where each tile and macro-tile stores a fixed number of bytes. Like the 1D formats, each 3D tiled surface must start on a 4K-byte boundary. Additionally, each NxMx4 slice of the 3D array must start on a 4K-byte boundary, so that individual 3D slices may be accessed as if they are a 2D array. {Is the 4K-byte restriction necessary?}

## 5.1 3D Micro-Tile Formats

Tiles in 3D arrays cover 8x8x1 data elements, even though the best aspect ratio for 3D tiles is probably a 4x4x4 array of data elements. That aspect ratio would provide the greatest degree of locality for random reads, for example, and therefore should be more efficient. However, the implementation is simpler if 3D tile formats are similar to 2D tile formats, for two reasons. First, this reduces the amount of multiplexing and address decoding required to read 3D texels. Second, it makes it simpler to render to (X, Y) slices within the 3D array. Therefore, the 3D tile formats encode an 8x8x1 tile of data elements, in exactly the same way as for 2D tiles.

The figure below shows how an 8x8x1 tile divides into micro-tiles for different pixel sizes. The numbers are the relative micro-tile addresses within the tile. 64-bits is the maximum size allowed for texels in 3D arrays. Unlike 2D arrays, 3D arrays do not support 4-bit and 24-bit pixel sizes.

| | |
|---|---|
| Micro-Tile 0 | <63:0> |
| Micro-Tile 1 | <63:0> |
| Micro-Tile 0 | <127:64> |
| Micro-Tile 1 | <127:64> |
| Micro-Tile 2 | <63:0> |
| Micro-Tile 3 | <63:0> |
| Micro-Tile 2 | <127:64> |
| Micro-Tile 3 | <127:64> |

8-bit Pixels

| |
|---|
| Micro-Tile 0 |
| Micro-Tile 1 |
| Micro-Tile 2 |
| Micro-Tile 3 |
| Micro-Tile 4 |
| Micro-Tile 5 |
| Micro-Tile 6 |
| Micro-Tile 7 |

16-bit Pixels

| | |
|---|---|
| MT 0 | MT 2 |
| MT 1 | MT 3 |
| MT 4 | MT 6 |
| MT 5 | MT 7 |
| MT 8 | MT 10 |
| MT 9 | MT 11 |
| MT 12 | MT 14 |
| MT 13 | MT 15 |

32-bit Pixels

| | | | |
|---|---|---|---|
| 0 | 2 | 4 | 6 |
| 1 | 3 | 5 | 7 |
| 8 | 10 | 12 | 14 |
| 9 | 11 | 13 | 15 |
| 16 | 18 | 20 | 22 |
| 17 | 19 | 21 | 23 |
| 24 | 26 | 28 | 30 |
| 25 | 27 | 29 | 31 |

64-bit Pixels

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 |
| 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
| 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
| 17 | 19 | 21 | 23 | 25 | 27 | 29 | 31 |
| 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 |
| 33 | 35 | 37 | 39 | 41 | 43 | 45 | 47 |
| 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 |
| 49 | 51 | 53 | 55 | 57 | 59 | 61 | 63 |

128-bit Pixels

**Figure 19: 3D Micro-Tile Layout Within Tiles**

The figure below shows the format of texels inside each 16-byte (128-bit) micro-tiles. Pixels from even scanlines are in the lower 64-bits of each micro-tile and pixels from odd scanlines are in the upper 64-bits.

one micro-tile is 128 bits, as signified by the bold outline for each micro-tile

| | 127:120 | 119:112 | 111:104 | 103:96 | 95:88 | 87:80 | 79:72 | 71:64 | 63:56 | 55:48 | 47:40 | 39:32 | 31:24 | 23:16 | 15:8 | 7:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8-bit data | (7,2) | (6,2) | (5,2) | (4,2) | (3,2) | (2,2) | (1,2) | (0,2) | (7,0) | (6,0) | (5,0) | (4,0) | (3,0) | (2,0) | (1,0) | (0,0) |
| | (7,3) | (6,3) | (5,3) | (4,3) | (3,3) | (2,3) | (1,3) | (0,3) | (7,1) | (6,1) | (5,1) | (4,1) | (3,1) | (2,1) | (1,1) | (0,1) |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 16-bit data | Pixel (7,0) | Pixel (6,0) | Pixel (5,0) | Pixel (4,0) | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) |
| | Pixel (7,1) | Pixel (6,1) | Pixel (5,1) | Pixel (4,1) | Pixel (3,1) | Pixel (2,1) | Pixel (1,1) | Pixel (0,1) |

| | | | | |
|---|---|---|---|---|
| 32-bit data | Pixel (3,0) | Pixel (2,0) | Pixel (1,0) | Pixel (0,0) |
| | Pixel (3,1) | Pixel 2,1) | Pixel (1,1) | Pixel (0,1) |

| | | |
|---|---|---|
| 64-bit data | Pixel (1,0) | Pixel (0,0) |
| | Pixel (1,1) | Pixel (0,1) |

| | |
|---|---|
| 128-bit data | Pixel (0,0) |
| | Pixel (0,1) |

**Figure 20: Pixel Format within 3D Micro-Tiles**

{**Note**: We should find a way to determine if we lose significant performance by not implementing 4x4x4 3D tiles.}

## 5.2 3D Macro-Tile Formats

The 3D macro-tile formats store a 32x16x4 array of data elements. This size allows reasonably efficient movement through the 3D array in either the X, Y, or Z directions, as described below. Although the tile size is 16 in Y, software should constrain the size in Y to a multiple of 32. That guarantees that each NxMx4 slab of 3D data occupies a multiple of 4K-bytes, even for 8-bit data elements. It is also simpler than enforcing a different Y height constraint for 3D arrays than for 2D arrays. The macro-tile size is expressed as 32x16x4, however, rather than as 32x32x4, because the 32x8x4 macro-tile size stores a contiguous array of bytes within each of the memory subsets. A 32x32x4 region includes bytes from two discotguous regions within each memory subset, unless the pitch happens to equal 32.

The following figures show the layout of 8x8x1 tiles within 32x16x4 macro-tiles for 3D arrays. Each figure shows two macro-tiles (slab 0 and slab 1) comprising a 32x16x8 region, since even and odd slices in Z use a different subset pattern. Each row of a figure shows the four slices within a macro-tile. Light lines mark tiles within the macro-tiles. The upper line of text in each tile specifies the memory subset. The lower line specifies the tile number within that memory subset. *S* equals the number of tiles per subset in a slice of the 3D array.

slab 0, slice 0 (Z=0)

| | | | |
|---|---|---|---|
| ab<br>0 | ab<br>4 | ab<br>8 | ab<br>12 |
| cd<br>0 | cd<br>4 | cd<br>8 | cd<br>12 |

slab 0, slice 1 (Z=1)

| | | | |
|---|---|---|---|
| ab<br>1 | ab<br>5 | ab<br>9 | ab<br>13 |
| cd<br>1 | cd<br>5 | cd<br>9 | cd<br>13 |

slab 0, slice 2 (Z=2)

| | | | |
|---|---|---|---|
| ab<br>2 | ab<br>6 | ab<br>10 | ab<br>14 |
| cd<br>2 | cd<br>6 | cd<br>10 | cd<br>14 |

slab 0, slice 3 (Z=3)

| | | | |
|---|---|---|---|
| ab<br>3 | ab<br>7 | ab<br>11 | ab<br>15 |
| cd<br>3 | cd<br>7 | cd<br>11 | cd<br>15 |

slab 1, slice 0 (Z=4)

| | | | |
|---|---|---|---|
| cd<br>S | cd<br>4+S | cd<br>8+S | cd<br>12+S |
| ab<br>S | ab<br>4+S | ab<br>8+S | ab<br>12+S |

slab 1, slice 1 (Z=5)

| | | | |
|---|---|---|---|
| cd<br>1+S | cd<br>5+S | cd<br>9+S | cd<br>13+S |
| ab<br>1+S | ab<br>5+S | ab<br>9+S | ab<br>13+S |

slab 1, slice 2 (Z=6)

| | | | |
|---|---|---|---|
| cd<br>2+S | cd<br>6+S | cd<br>10+S | cd<br>14+S |
| ab<br>2+S | ab<br>6+S | ab<br>10+S | ab<br>14+S |

slab 1, slice 3 (Z=7)

| | | | |
|---|---|---|---|
| cd<br>3+S | cd<br>7+S | cd<br>11+S | cd<br>15+S |
| ab<br>3+S | ab<br>7+S | ab<br>11+S | ab<br>15+S |

**Figure 21: 3D Macro-Tile Two Subset Format**

The figure above shows the tiled format for two memory subsets. This occurs when there is just one rendering pipeline. Movement in Y or Z through the 3D array hits both memory subsets. Movement in X hits only one memory subset, but alternates between the two banks within that subset. If the page size is 256 64-bit words and pixels are 64-bits in size or smaller, horizontally adjacent tiles are either in the same page of the same bank or are in different banks. This is not true for 128-bit pixels, but in that case sweeping across a single tile hits eight 256-bit accesses in the same page.

slab 0, slice 0 (Z=0)

| | | | |
|---|---|---|---|
| ab0 0 | ab1 0 | ab0 4 | ab1 4 |
| cd1 0 | cd0 0 | cd1 4 | cd0 4 |

slab 0, slice 1 (Z=1)

| | | | |
|---|---|---|---|
| ab0 1 | ab1 1 | ab0 5 | ab1 5 |
| cd1 1 | cd0 1 | cd1 5 | cd0 5 |

slab 0, slice 2 (Z=2)

| | | | |
|---|---|---|---|
| ab0 2 | ab1 2 | ab0 6 | ab1 6 |
| cd1 2 | cd0 2 | cd1 6 | cd0 6 |

slab 0, slice 3 (Z=3)

| | | | |
|---|---|---|---|
| ab0 3 | ab1 3 | ab0 7 | ab1 7 |
| cd1 3 | cd0 3 | cd1 7 | cd0 7 |

slab 1, slice 0 (Z=4)

| | | | |
|---|---|---|---|
| cd1 S | cd0 S | cd1 4+S | cd0 4+S |
| ab0 S | ab1 S | ab0 4+S | ab1 4+S |

slab 1, slice 1 (Z=5)

| | | | |
|---|---|---|---|
| cd1 1+S | cd0 1+S | cd1 5+S | cd0 5+S |
| ab0 1+S | ab1 1+S | ab0 5+S | ab1 5+S |

slab 1, slice 2 (Z=6)

| | | | |
|---|---|---|---|
| cd1 2+S | cd0 2+S | cd1 6+S | cd0 6+S |
| ab0 2+S | ab1 2+S | ab0 6+S | ab1 6+S |

slab 1, slice 3 (Z=7)

| | | | |
|---|---|---|---|
| cd1 3+S | cd0 3+S | cd1 7+S | cd0 7+S |
| ab0 3+S | ab1 3+S | ab0 7+S | ab1 7+S |

**Figure 22: 3D Macro-Tile Four Subset Format**

The figures above and below show the tiled format for four and eight memory subsets. Movement in Y or Z through the 3D array hits two memory subsets in different pipelines. Movement in X hits half of the memory subsets, one per pipeline. If the page size is 256 64-bit words and pixels are 64-bits in size or smaller, horizontally adjacent tiles in the same pipeline are either in the same page of the same bank or are in different banks. This is not true for 128-bit pixels, but in that case sweeping across a single tile hits eight 256-bit accesses in the same page.

slab 0, slice 0 (Z=0)

| | | | |
|---|---|---|---|
| ab0 0 | ab1 0 | ab2 0 | ab3 0 |
| cd2 0 | cd3 0 | cd0 0 | cd1 0 |

slab 0, slice 1 (Z=1)

| | | | |
|---|---|---|---|
| ab0 1 | ab1 1 | ab2 1 | ab3 1 |
| cd2 1 | cd3 1 | cd0 1 | cd1 1 |

slab 0, slice 2 (Z=2)

| | | | |
|---|---|---|---|
| ab0 2 | ab1 2 | ab2 2 | ab3 2 |
| cd2 2 | cd3 2 | cd0 2 | cd1 2 |

slab 0, slice 3 (Z=3)

| | | | |
|---|---|---|---|
| ab0 3 | ab1 3 | ab2 3 | ab3 3 |
| cd2 3 | cd3 3 | cd0 3 | cd1 3 |

slab 1, slice 0 (Z=4)

| | | | |
|---|---|---|---|
| cd2 S | cd3 S | cd0 S | cd1 S |
| ab0 S | ab1 S | ab2 S | ab3 S |

slab 1, slice 1 (Z=5)

| | | | |
|---|---|---|---|
| cd2 1+S | cd3 1+S | cd0 1+S | cd1 1+S |
| ab0 1+S | ab1 1+S | ab2 1+S | ab3 1+S |

slab 1, slice 2 (Z=6)

| | | | |
|---|---|---|---|
| cd2 2+S | cd3 2+S | cd0 2+S | cd1 2+S |
| ab0 2+S | ab1 2+S | ab2 2+S | ab3 2+S |

slab 1, slice 3 (Z=7)

| | | | |
|---|---|---|---|
| cd2 3+S | cd3 3+S | cd0 3+S | cd1 3+S |
| ab0 3+S | ab1 3+S | ab2 3+S | ab3 3+S |

**Figure 23: 3D Macro-Tile Eight Subset Format**

## 5.3 3D Address Equations

This subsection presents equations for computing addresses in a 3D array. This is a two-step process that makes use of the 1D array equations of subsection 3.3. The first list below defines parameters that are constant for a given surface. The second list names parameters that depend on which pixel is accessed in the array. 3D address equations use the parameters in the first list and one or more of the parameters in the second list to define the remaining parameters in the second list.

| | |
|---|---|
| Size | Bytes per pixel: can be 1, 2, 4, 8, or 16 |
| **DataSize** | 64 times Size, equals the total bytes of data in the 3D tile |
| **Pipes** | Total number of Render Backend/Memory Controller pipelines: 1, 2, or 4 |
| Subsets | Total number of memory subsets: equals twice the number of pipelines |
| **SurfaceBase** | Byte address of pixel zero in device address space, must be 4K-byte aligned |
| **TileSize** | Bytes per tile (should always equal 64*Size for 3D arrays, defined for consistency with 2D) |
| **TileBase** | Byte address of first pixel in a tile (should always be zero for 3D arrays) |
| **Pitch** | The width of each scanline in pixels, must be a multiple of 32 |
| **Height** | The height of each slice in scanlines, must be a multiple of 16 (should be multiple of 32) |

| | |
|---|---|
| **X, Y, Z** | Pixel location in the 3D array |
| **Local**Addr | Byte address of the pixel within its memory subset, starting from device address 0 |

| | |
|---|---|
| SubsetOffset | Byte address of the pixel within its memory subset, starting from pixel (0,0) of the surface |
| MemSelect | Number of the memory controller that stores this pixel |
| BankSelect | 0 for banks AB or 1 for banks CD, together with MemSelect determines the memory subset |
| **Subset** | Subset number, equals BankSelect + 2*MemSelect |
| MacroOffset | Sequential number of the macro-tile containing the pixel, starting from SurfaceBase |
| TileNumber | Sequential number of the tile containing the pixel, within its memory subset and its macro-tile |
| TileAddr | Byte address of the pixel within its tile, starting from the first byte of the tile |
| TileOffset | Byte address of the pixel within its tile, relative to TileBase (which is normally zero) |

The following equations use *X*, *Y* and *Z* to compute the other address terms. This is used to convert an array access into a *Subset* and *LocalAddr*. The final set of equations in subsection 3.3 uses these results to produce a device address.

```
BankSelect    = (Y/8 + Z/4) mod 2;                       // CD banks alternate every 8 Y and 4 Z
MemSelect     = (X/8 + BankSelect*(Pipes/2)) mod Pipes;  // Offset memory in alternate rows and slabs
Subset        = BankSelect + 2*MemSelect;                // subset number
TileNumber    = (Z mod 4) + ((X mod 32)/8/Pipes)*4;      // Groups of four tiles vertically
MicroByte     = (X mod 8 + ((Y mod 8)/2)*8)*Size         // Byte address within tile for even scanlines
                                                    // Odd scanlines get odd micro-tiles within an 8x8 tile
TileOffset    = (MicroByte mod 16) + (Y mod 2)*16 + (MicroByte/16)*32;
TileAddr      = TileBase + TileOffset;                   // The tile may contain other data as well
MacroOffset   = (X/32) + (Pitch/32) * ((Y/16) + (Height/16)*(Z/4));
SubsetOffset  = MacroOffset*TileSize*32/Subsets + TileNumber*TileSize + TileAddr;
LocalAddr     = SurfaceBase/Subsets + SubsetOffset;
```

The following equations use *LocalAddr*, *BankSelect* and *MemSelect* to compute the other address terms. This is used to convert a device address into an (*X*, *Y*) array address. The final set of equations in subsection 3.3 produces *LocalAddr*, *BankSelect* and *MemSelect* from a device address.

```
SubsetOffset  = LocalAddr – SurfaceBase/Subsets;         // relative subset address in surface
TileAddr      = SubsetOffset mod TileSize;               // byte address within the tile
TileOffset    = TileAddr - TileBase;                     // byte address within subset of the tile
MacroOffset   = SubsetOffset * Subsets / 32 / TileSize;  // 32*TileSize bytes per macro-tile
TileNumber    = SubsetOffset/TileSize mod (32/Subsets);  // 32 8x8 tiles per macro-tile over all subsets
MicroByte     = TileOffset mod 16 + (TileOffset/32)*16;
Z             = (TileNumber mod 4) + (MacroOffset*32*16/Pitch/Height)*4;
Ymacro        = (MacroOffset*32/Pitch mod Height/16); // Macro-tiles vertically within a slab
Ytile         = (BankSelect + (Z/4 mod 2)) mod 2;        // tile 0 or1 vertically in macro-tile
Y             = Ymacro*16 + Ytile*8 + (TileOffset/16/Size)*2 + (TileOffset/16 mod 2);
Xmacro        = MacroOffset mod Pitch/32;
Xtile         = (MemSelect + BankSelect*Pipes/2) mod Pipes + (TileNumber/4)*Pipes;
Xbyte         = MicroByte mod (8*Size);                  // Byte address within first 8x1 scanline
X             = Xmacro*32 + Xtile*8 + Xbyte/Size;
```

# 6. Mipmap Storage

{This section is for any special issues involving texture storage that belong in a whole-chip document instead of in the TC block spec. At present the only such issue is mipmap storage.}

## 6.1 Packing 2D Mipmaps

Small 2D surfaces waste a lot of space if each dimension must be increased to a multiple of 32. This is a particular problem for mipmap chains, which produce many small mipmaps. For example, if each mipmap produced by a 32x32 texture map requires a full macro-tile, then the mipmap chain requires 6*32*32 = 6144 pixels instad of 1365 pixels. The problem is worse for small texels, since each surface must start on a 4K-byte boundary. With 8-bit texels, the mipmap chain would require 6*4K-bytes = 24K-bytes, instead of 1365-bytes.

R400 solves this problem in two ways. First, each texture is specified with two surface descriptors. The first points to the base texture map, which has dimensions that are increased to multiples of 32. The second points to the start of the mipmap chain and R400 automatically computes the starting address of each subsequent mipmap in the chain. Each texture map in the mipmap chain has its dimensions increased to a power of 2 and each starts at an address that is a multiple of 4K-bytes.

Additionally, R400 packs the small mipmaps at the tail of the mipmap chain into a single 32x32 tile. Each mipmap has a position in the final tile that is based solely on the maximum of the width and height of that mipmap. The figure below shows the layout. Each mipmap in the chain is increased in size, if necessary, to a square mipmap with width and height equal to a power of two. The location where each mipmap is stored depends solely on its (increased) size. Any mipmap in the chain that has width > 16 or height > 16 is stored as a separate surface that uses a multiple of 4K-bytes. The final mipmaps also require a minimum of 4K-bytes, which is larger than a single 32x32 macro-tile for 8-bit and 16-bit texels.

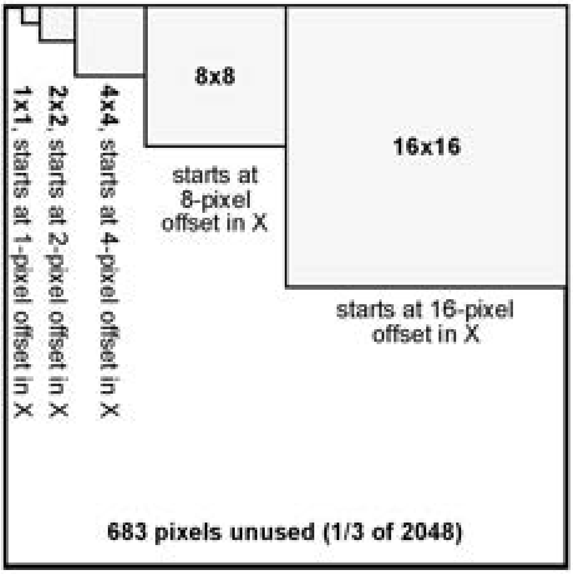
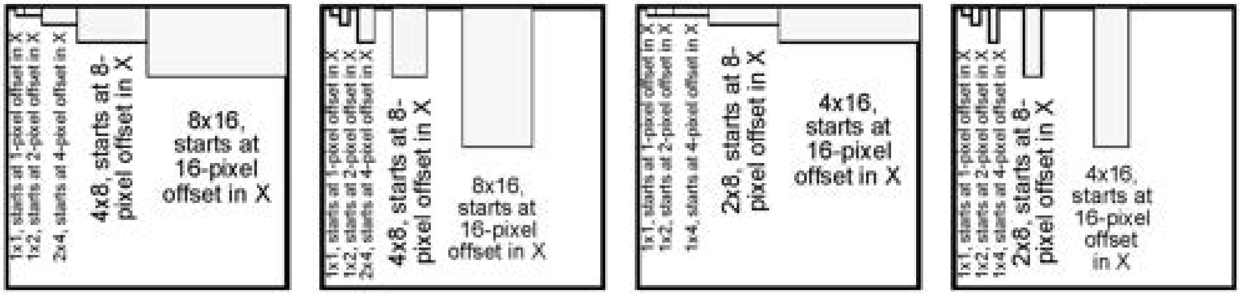


Figure 24: Mipmap Chain Storage Offsets

If the mipmaps are actually squares with width and height equal to a power of two, the above format uses 341 of 1024 pixels in the two tiles, wasting 683 or 2/3 of the pixels in the tile. The figure below shows examples of mipmaps with non-square aspect ratios and the number of pixels wasted in each case. Note that for each mipmap, texel (0,0) is stored in the same location as for the corresponding square mipmaps in the figure above.


2:1 Size Ratio (853 pixels unused)

4:1 Size Ratio (937 pixels unused)

**Figure 25: Mipmap Chain Unused Pixels**

Rendering to mipmaps that are packed this way requires altering the window offset. For example, to render to a mipmap that is expanded to 16x16, set the base address to the start of the macro-tile and increase the X offset by 16. Future chips may use different offsets or packing formats, so the driver should obtain mipmap positions and offsets from code that is delivered with the hardware.

## 6.2 2D Mipmap Equations

{Describe how R400 computes the position of each mipmap in the chain and the total memory required for any mipmap chain.}

## 6.3 Packing 3D Mipmaps

{Define a 3D mipmap packing format.}

{Proposal: pack the final 3D mipmaps into a 32x32x32 cube, which contains 16 3D macro-tiles. Each 3D mipmap is expanded to a cube with all three sizes equal to a power of two that is greater than or equal to its largest dimension. }

{Variant: The same as the above, except only force the X and Y dimensions to match. Allow the Z dimension to be a power of two that is less than X or Y. This causes the packed mipmap to use a variable number of tiles.}

## 6.4 3D Mipmap Equations

{Describe how R400 computes the position of each mipmap in the chain and the total memory required for any mipmap chain.}

# 7. Destination Color Compression

R400 supports rendering to pixels with 1, 2, 3, 4, 6, or 8 samples per pixel. To a large extent, the aliased mode (1 sample per pixel) is just a special case of the multi-sample modes (2, 3, 4, 6, or 8 samples per pixel), though there are some operations, such as multi-buffer rendering, that are available only for single-sample pixels.

R400 stores multi-sample color data as fragments. A fragment is a pixel color together with a mask that specifies the samples within the pixel where that color is visible. As a result, if an operation writes a single color to an entire pixel, e.g. in the interior of a triangle, only one color (plus the mask) is necessary to describe the entire pixel. If there are S samples per pixel, the pixel could have as many as S fragments, but multiple fragments are only needed if multiple triangles are visible within a single pixel. Unless triangles are extremely small, it is quite common for a pixel to have just one fragment. The maximum number of fragments per pixel within an 8x8 tile is also typically small, so fragments result in significant compression. {For example, if there are eight samples per pixel and an average of less than 2 colors per pixel within a tile, storing fragments results in approximately 4x compression relative to supersampling.}

The following subsections describe the format of color data and fragment mask data.

## 7.1 Destination Color Format

R400 stores multi-sample pixels in two separate surfaces: one surface for pixel colors and a separate surface for the fragment masks (Fmasks). R400 uses a 4-bit Cmask field to specify storage format for the fragment mask and color. The figure below illustrates the color storage format for each value of Cmask.

If Cmask=Background, no color or fragment data is stored for the tile. Instead, each pixel is treated as having a single fragment that covers the entire pixel and is equal to the color_clear value. No data needs to be stored in the color surface in this mode.

If Cmask=Expanded, R400 stores a separate color for each sample. Starting at the base address, R400 stores a complete 2D tiled array for the color at sample 0, followed by a complete 2D tiled array for the color at sample 1, and so forth for the total number of samples per pixel. This format allows the texture logic and software to read multi-sample data by reading S individual 2D arrays for S-sample pixels.

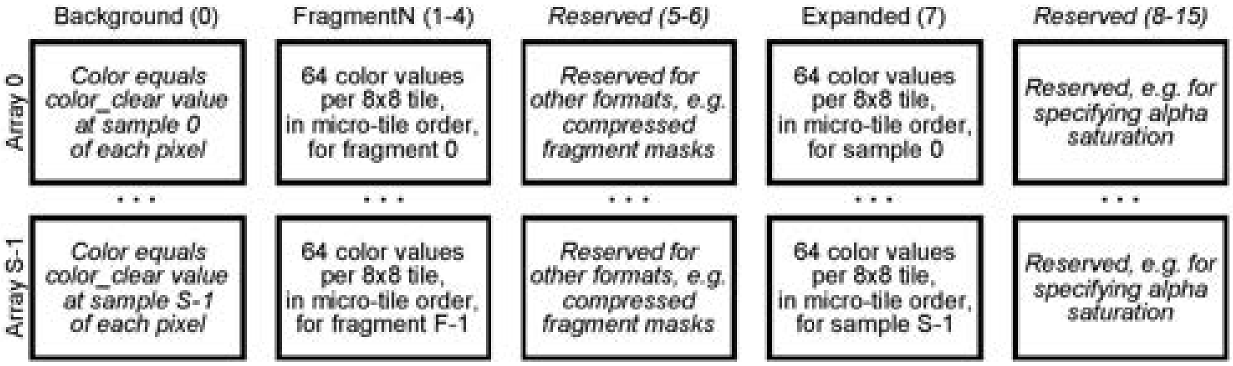


**Figure 26: Fragment Mask Bit Format**

The FragmentN formats allow compression when there are 2 or more samples per pixel. The choices are Fragment1, Fragment2, Fragment4, and Fragment 8 modes, which encode tiles with a maximum of 1, 2, 4, or 8 fragments in any single pixel. In these Cmask modes, each successive 2D array stores a single color per pixel from fragment 0 up to the maximum number of fragments. If the triangles in a scene are relatively large, then most tiles are likely to have at most one or two fragments per pixel. Storing different fragment colors in separate 2D arrays allows more tiles to share the same DDRAM page. This allows larger DDRAM page bursts when there are a small number of fragments per pixel.

## 7.2 Fragment Mask Format

A fragment mask consists of a set of n-bit fragment mask values, or Fmasks, with one such number per sample in each pixel. Each pixel within an 8x8 tile uses the same number of bits per Fmask. The number of bits in each Fmask depends on the maximum number of fragments per pixel within an 8x8 tile. If each pixel contains exactly one fragment, then no Fmask is required, since a single color completely covers each pixel. If a pixel in the tile contains 2 fragments but none contain more, then each Fmask requires 1-bit to select between one of two fragments per pixel. Similarly, 2-bit fragment masks are required if there is a maximum of 3 or 4fragments per pixel and 3-bit fragment masks are required if there is a maximum of 5 to 8 fragments per pixel. There cannot be more than 8 fragments per pixel, since there cannot be more than 8 samples that could have separate colors.

The Fmask buffer stores one or more 64-bit words per tile. Each 64-bit word stores one bit of Fmask data for one sample of each pixel in the 8x8 tile. The figure below shows the correspondence between the pixels of the tile and the bits in one 64-bit Fmask word.

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| word 0 | (7,1) | (6,1) | (5,1) | (4,1) | (3,1) | (2,1) | (1,1) | (0,1) | (7,0) | (6,0) | (5,0) | (4,0) | (3,0) | (2,0) | (1,0) | (0,0) |
| word 1 | (7,3) | (6,3) | (5,3) | (4,3) | (3,3) | (2,3) | (1,3) | (0,3) | (7,2) | (6,2) | (5,2) | (4,2) | (3,2) | (2,2) | (1,2) | (0,2) |
| word 2 | (7,5) | (6,5) | (5,5) | (4,5) | (3,5) | (2,5) | (1,5) | (0,5) | (7,4) | (6,4) | (5,4) | (4,4) | (3,4) | (2,4) | (1,4) | (0,4) |
| word 3 | (7,7) | (6,7) | (5,7) | (4,7) | (3,7) | (2,7) | (1,7) | (0,7) | (7,6) | (6,6) | (5,6) | (4,6) | (3,6) | (2,6) | (1,6) | (0,6) |

**Figure 27: Fragment Mask 1-Bit/Pixel Format**

The following figure shows multiple 64-bit words that store one Fmask bit each for each of S samples per pixel. The complete Fmask data stores 1, 2 or 3 copies of this set of S 64-bit words. This storge structure allows the RB to read and write either 1-bit, 2-bits, or 3-bits for each sample in the tile. The following table shows the number of micro-tiles required to store a tile of Fmask data for each number of samples and each

| | 63 48 | 47 32 | 31 16 | 15 0 |
|---|---|---|---|---|
| Fmask bit for S samples | word 3, sample 0 | word 2, sample 0 | word 1, sample 0 | word 0, sample 0 |
| | . . . | | | |
| | word 3, sample S-1 | word 2, sample S-1 | word 1, sample S-1 | word 0, sample S-1 |

**Figure 28: Fragment Mask 1-Bit/Sample Format**

| Samples per Pixel | Cmask = Fragment1 | Cmask = Fragment2 | Cmask = Fragment4 | Cmask = Fragment8 |
|---|---|---|---|---|
| **1 sample per pixel** | 0 of 0 micro-tiles | *(not used)* | *(not used)* | *(not used)* |
| **2 samples per pixel** | 0 of 1 micro-tiles | 1 of 1 micro-tiles | *(not used)* | *(not used)* |
| **3 samples per pixel** | 0 of 3 micro-tiles | 1.5 of 3 micro-tiles | 3 of 3 micro-tiles | *(not used)* |
| **4 samples per pixel** | 0 of 4 micro-tiles | 2 of 4 micro-tiles | 4 of 4 micro-tiles | *(not used)* |
| **6 samples per pixel** | 0 of 9 micro-tiles | 3 of 9 micro-tiles | 6 of 9 micro-tiles | 9 of 9 micro-tiles |
| **8 samples per pixel** | 0 of 12 micro-tiles | 4 of 12 micro-tiles | 8 of 12 micro-tiles | 12 of 12 micro-tiles |

**Table 1: Fmask Storage Required Per Tile**

Finally, the following figure shows the layout of a single tile of Fmask data for each number of samples per pixel and each FragmentN mode. Each row represents S*64-bits of Fmask data. Note that the FragmentN modes are not used when there is only one sample per pixel. In that case, the only allowed Cmask modes are Background and Expanded.

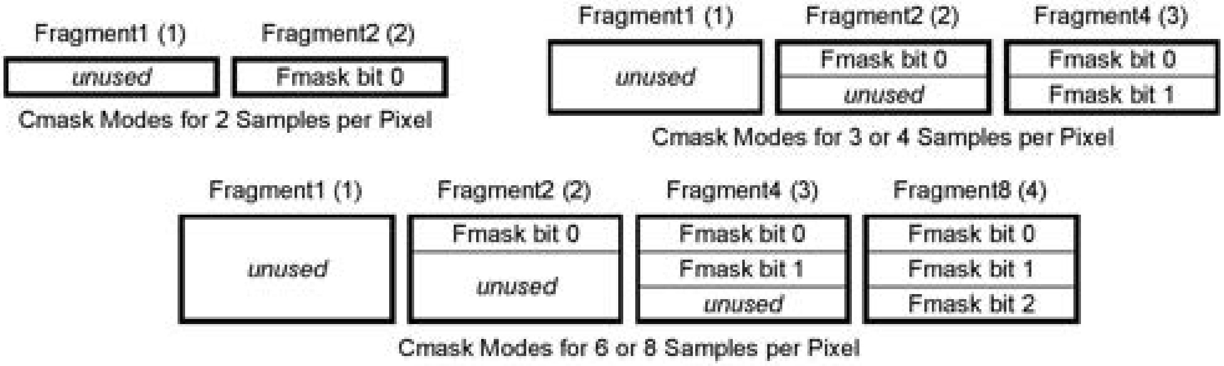


**Figure 29: Fragment Mask Bit Storages Format**

# 8. Depth and Stencil Formats

This section describes how the R400 family stores depth and stencil data at varying levels of compression. R400 supports 1, 2, 3, 4, 6, or 8 samples per pixel. Multi-sampled 8x8 tile formats must allocate enough frame buffer memory to be able to fall back to storing a separate value per sample in the cases where the data cannot be compressed. Therefore compression reduces the amount of data that must be read or written, but not the amount of memory that must be allocated.

The figure below illustrates the formats for storing depth data in a tile, depending on the 4-bit Zmask field in the 32-bit tile data word for each tile. If Zmask=Expanded, single-sample depth data is stored in standard micro-tile format as 16-bit or 32-bit pixels. This Expanded format allows single-sample depth and stencil values to be read and written by software and by the texture logic. All other depth formats are only readable and writable by the Render Backend depth logic and by address utility code that translates the compressed formats.

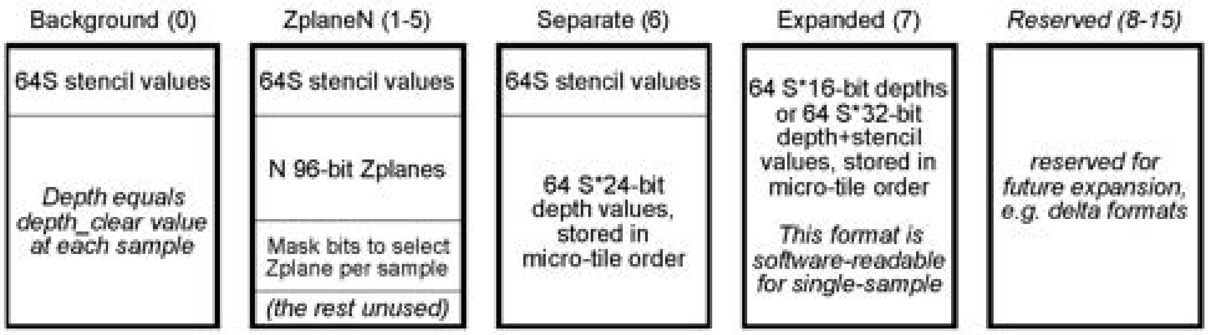


**Figure 30: Depth Storage Formats**

Remaining values of Zmask represent depth compression formats that store the stencil bits separately from depth bits. This allows depth and stencil to be accessed and compressed independently. If Zmask=Background, no depth data is stored in the tile. Instead, each depth value equals the depth_clear value. If Zmask=1-5, depth data is represented as Zplanes, which are described in the following subsection. If Zmask=Separate, depths are stored as a packed array of 16-bit or 24-bit values. The toal number of depth values is 64S, where S is the number of samples per pixel. These are stored as S adjacent arrays of 64 packed depth values, one per sample, similar to the format for storing multiple color fragments.

The following subsections describe stencil compression and Zplane compression.

## 8.1 Compressed Stencil Formats

The stencil buffer stores 8-bits per sample and is stored together with the depth buffer. Rendering operations can modify a pixel's stencil value based on the result of the depth test and on comparing the current stencil value to a reference value. The reference comparison can also be used to disable modifying the depth and color of the pixel. Allowed stencil modification operations are keeping the old value, setting it to zero, replacing it with the reference value, incrementing it, decrementing it, and inverting it.

The following is a brief summary of common uses of the stencil buffer. See the OpenGL Programming Guide for more information on these algorithms, except for shadow volumes, which is a more recent technique. Most of these algorithms use just two stencil values and set the same stencil value at each pixel that is written in a given triangle. The shadow volume method uses a range of values [base–N..base+N] for some base stencil value, where N depends on the number of overlapping shadows.

1) Stippling and irregular masking: set the stencil to define the pixels that can be updated.
2) Capping: invert the stencil on each pixel update to find places where clipping exposes an object's interior.
3) Non-convex polygons: invert the stencil on each pixel update to find the interior of a non-convex polygon.

4) Write once: change the stencil when writing the pixel; don't write if the stencil has already been changed.
5) Decals: change the stencil when writing the pixel; only write the decal where the stencil was changed.
6) Shadow Volumes: inc/dec the stencil based on projections of occluding objects, to find shadow regions.

Given these usages, R400 supports four types of stencil compression. The following table shows the four stencil compression modes, as selected by the 4-bit Smask field of the 32-bit tile data word. A fast stencil clear of the tile sets Smask to zero, indicating that the entire tile equals the stencil clear color. If Smask!=0, the lower three bits specify whether any stencil values in the tile are greater than (bit 2), less than (bit 1), or equal to (bit 0) a specified stencil compare value. If there aren't any stencil values greater than or less than the stencil compare value, then they all equal the stencil compare value, so again no bits need to be stored for the stencil values.

| Smask | Stencil Compression Mode |
|---|---|
| 0000 | 0-bits per stencil: every stencil in the entire tile is equal to the stencil clear value |
| 0001 | 0-bits per stencil: every stencil in the entire tile is equal to the stencil compare value |
| 0010-0111 | 4-bits per stencil: each stencil is the sum of the stencil base value plus an unsigned 4-bit offset |
| 1000 | 8-bits per stencil: every stencil in the entire tile is equal to the stencil clear value |
| 1001 | 8-bits per stencil: every stencil in the entire tile is equal to the stencil compare value |
| 1001-1111 | 8-bits per stencil: each stencil stores the full 8-bit value |

**Table 2: Stencil Compression Modes**

If the stencils are not all equal to the clear color or the compare color, then there are still two choices. A stencil base value may be used to compress the stencils. The stencil base value is typically set to max(0, stencil_compare – 8). If all the stencil values are in the range [base .. base+15], then a 4-bit offset is sufficient to specify each stencil value, relative to the stencil base. This is primarily useful for multi-sampled pixels. Finally, full 8-bit values may be stored for each stencil if any stencil values are outside the base range or if the stencil surface needs to be decompressed in order to allow software or the texture controller to read them.

If Zmask!=Expanded, stencil values are stored packed together at the start of the tile. If Zmask=Expanded, 8-bit stencils are interleaved with 24-bit depth values to produce 32-bit depth/stencil values, regardless of the value of Smask. This is the only interaction between stencil compression and depth compression. Zmask is only set to Expanded when writing a tile with depth compression disabled or after expanding the depth buffer to uncompressed format.

The table below shows the number of micro-tiles required to store stencil values, depending on the number of samples per pixel. These sizes apply for all depth compression modes except for Lockable. In Lockable mode, stencil values are stored as the lower byte of 32-bit words, for which the upper 24-bits are a depth value. Stencil compression is not available together with the Lockable depth mode, which is only produced as a result of a specific operation that converts single-sample depth/stencil values into a form that is directly readable and writable by software.

| Samples per Pixel | Smask = 0000-0001 | Smask = 0010-0111 | Smask = 1000-1111 |
|---|---|---|---|
| **1 sample per pixel** | 0 micro-tiles | 2 micro-tiles | 4 micro-tiles |
| **2 samples per pixel** | 0 micro-tiles | 4 micro-tiles | 8 micro-tiles |
| **3 samples per pixel** | 0 micro-tiles | 6 micro-tiles | 12 micro-tiles |
| **4 samples per pixel** | 0 micro-tiles | 8 micro-tiles | 16 micro-tiles |
| **6 samples per pixel** | 0 micro-tiles | 12 micro-tiles | 24 micro-tiles |
| **8 samples per pixel** | 0 micro-tiles | 16 micro-tiles | 32 micro-tiles |

**Table 3: Stencil Storage Sizes in Micro-Tiles**

A future chip could provide delta-encoded compression for stencil values in place of R4000's base/offset compression. This should allow significantly higher compression ratios for multi-sample stencil buffers.

If the stencils are stored as 8-bit values, they are micro-tiled in the stanard way for 8-bit pixels. Compressed 4-bit stencils are stored in a special micro-tile format that uses a 64-bit micro-tile instead of the standard 128-bit micro-tile. As for 8-bit stencils, the format depends on the number of samples. The compressed stencil formats are identical to the formats used for 4-bit Pmask values, which are described in subsection 8.4 below.

## 8.2 Zplane Depth Representation

R300 introduced compressing depth values by storing a plane equation for each triangle that intersects a tile. The plane equation allows the depth logic to compute a depth value at each sample, so that it is not necessary to store the individual depth values. The tile also stores a mask that specifies which of multiple plane equations to use at each sample.

The R400 Zplane compression format adapts the R300 technique to R400's 8x8 tiles and provides higher precision. As in R300, each Zplane is associated with a single triangle. Therefore, if four triangles are visible in an 8x8 tile, then the compressed tile must store four Zplanes and a 2-bit per sample mask that specifies which Zplane is visible at each sample. If only one triangle is visible in an 8x8 tile, then the compressed tile only needs to store that triangle's Zplane.

The following figure shows the 96-bit Zplane format used in R400. A Zplane contains six values. The slope in X and Y per subpixel (SlopeX and SlopeY) are each specified as a 30-bit fixed point S3.26 number. The depth value at the center of the 8x8 tile (CenterZ) is a 27-bit fixed point S3.23 number. Larger values for SlopeX, SlopeY, and CenterZ must be wrapped to these ranges by dropping higher order bits in fixed-point notation. So long as the depth values computed at sample points inside the primitive are in the range [-8..8), dropping the higher order bits does not affect the final depth value computed by R400 at sample points inside the primitive. The MultiSample bit is described below.

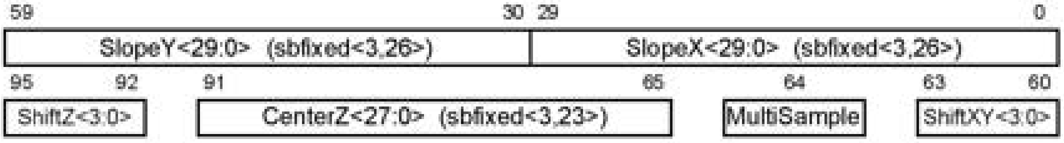


**Figure 31: Per-Triangle Zplane Format**

The ShiftXY and ShiftZ fields specify bit shift values for the SlopeX, SlopeY, and CenterZ fields, so that they can specify more accurate values with smaller ranges. The figure below shows how ShiftZ affects CenterZ and how ShiftXY affects SlopeX and SlopeY. When the shift is zero, the fixed-point value is converted to an S3.42 fixed-point value (for example) by appending low order zeros, which leaves the numeric value unchanged. Larger ShiftXY or ShiftZ values shift the fixed-point value right by the specified number of bits, sign extending the high order bits. These shifted values represent numbers in the range [$-2^{3-\text{shift}}$..$2^{3-\text{shift}}$). When ShiftZ==15, CenterZ represents values as small as $2^{-38}$. When ShiftXY==15, SlopeX and SlopeY represent values as small as $2^{-41}$. The slopes represent the change in depth per subpixel, so the smallest nonzero change in depth is $2^{-37}$ per pixel or $2^{-34}$ per tile. The smallest nonzero magnitude representable in the 24-bit floating-point depth format is also $2^{-34}$, so Zplanes allow specifying a slope of one lsb per tile in all depth formats.

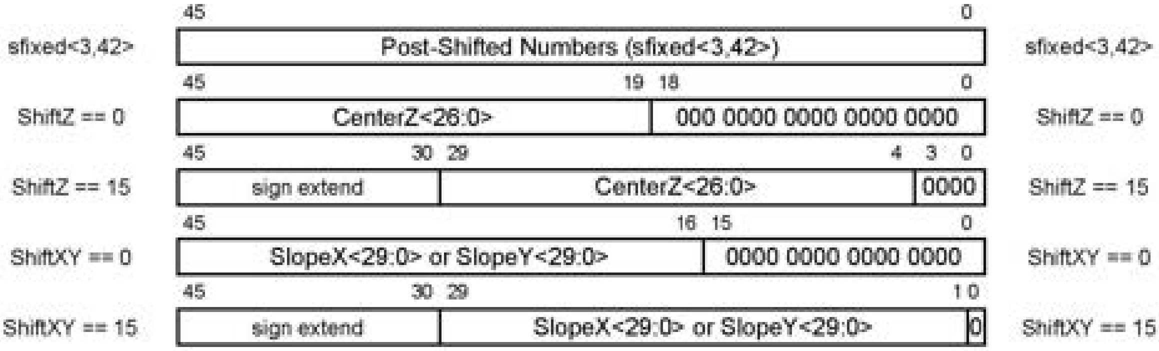


**Figure 32: Zplane Format Shifted Numbers**

The MultiSample bit specifies whether this Zplane was rendered with multisampling disabled, so that all samples are at the same location, or whether this Zplane was rendered with multisampling enabled, so that multiple samples occur at different locations within the pixel. Clients may enable and disable multisampling while rendering a scene, e.g. disabling it to render high quality anti-aliased lines with alpha blending. Disabling multisampling does not reduce the number of sample points per pixel – it simply moves them all to the same location. The MultiSample bit ensures that when a Zplane is converted to individual samples, this occurs using the multisample state that was valid at the time that the Zplane was generated, so that expanding the Zplane produces the same depth values that would occur if storing a separate depth value per sample.

Computing CenterZ and ShiftZ from a floating-point value FloatZ is straightforward. First convert FloatZ into an S3.42 value FixedZ, truncating low order bits of precision instead of rounding and dropping higher order bits to wrap around to this range. If FloatZ < –4 or FloatZ >= 4, then ShiftZ = 0 and CenterZ = (FixedZ + $2^{-24}$) <45:18>, that is, round off the lower 19 bits of FixedZ and use the remaining higher order bits as CenterZ. For smaller magnitudes of FloatZ, count the number of high order bits B in FixedZ that match the sign bit, up to 15. For example, in 0xF2, three high order bits match the sign bit. This can also be determined from the exponent of FloatZ. Then ShiftZ = B and CenterZ = (FixedZ + $2^{-24-B}$) <45–B:19–B>, that is, truncate the upper B bits of FixedZ and round off the lower 18–B bits of FixedZ. Finally, check whether rounding FixedZ caused CenterZ to overflow. If so, then ShiftZ = B+1 and CenterZ = 0x4000000 (2.0).

Computing SlopeX, SlopeY and ShiftXY from floating-point values FloatX and FloatY is similar. First convert FloatX and FloatY into S3.42 values FixedX and FixedY, truncating low order bits of precision instead of rounding and dropping higher order bits to wrap around to this range. If FloatX < –4, FloatX >= 4, FloatY < –4, or FloatY >= 4, then ShiftXY = 0, SlopeX = (FixedX + $2^{-27}$) <45:16>, and SlopeY = (FixedY + $2^{-27}$) <45:16>. For smaller magnitudes, count the number of high order bits BX and BY in FixedX and FixedY that match the sign bit, up to 15, and set B = min(BX,BY). This can also be determined from the exponents of FloatX and FloatY. Then ShiftXY = B, SlopeX = (FixedX + $2^{-27-B}$) <45–B:16–B>, and SlopeY = (FixedY + $2^{-27-B}$) <45–B:16–B>. Finally, check whether rounding FixedX or FixedY caused SlopeX or SlopeY to overflow. If so, then ShiftXY = B+1 and the overflowing slope or slope equals 0x10000000 (2.0).

Depth values and slopes can be significantly larger than the S3.N fixed-point formats supported above. R400 uses two's complement wraparound for depths and depth gradients. For example, if the true value of SlopeX is 8, the value stored in the Zplane format must be wrapped around to –8. In general, if a slope or CenterZ is outside the range [-8..8), represent it as a fixed point value, truncate higher order bits of integer precision and treat integer bit<3> as the sign bit to get the value to store in the Zplane format, along with a shift value of zero. This works provided that depth values computed at sample points inside the triangle are always in the range [-8..8).

## 8.3 Zplane Storage Formats

The figures below illustrates the five Zplane formats, which differ based on the number of Zplanes required in the tile and on whether there is one or multiple samples per pixel. Each Zplane occupies 96-bits, so each pair of Zplanes requires three 64-bit words. The Pmask data stores plane mask bits that specify which Zplane to use at each sample in the tile. For Zplane2 mode, there is a 1-bit Pmask per sample or an S-bit Pmask per pixel. For Zplane4 mode, there is a 2-bit Pmask per sample or a 2S-bit Pmask per pixel. For Zplane8 and Zplane16 modes, there is a 4-bit Pmask per sample or a 4S-bit Pmask per pixel.

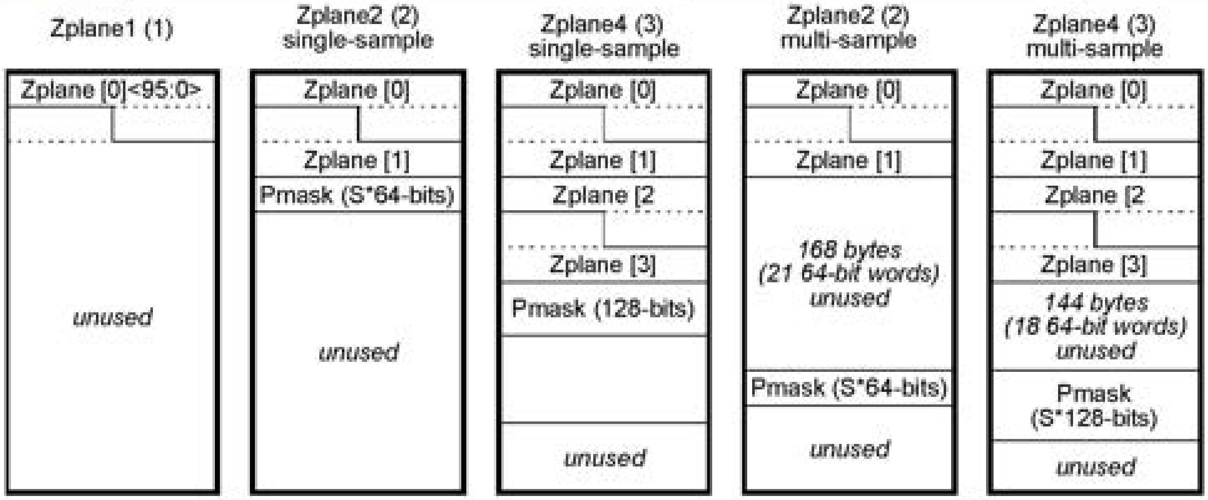


**Figure 33: One to Four Zplane Depth Tile Formats**

The single-sample and multi-sample storage formats differ in the location of the Pmask bits. Single-sample modes pack the Pmask bits immediately after the Zplanes. This reduces the number of 256-bit memory accesses required to read or write the compressed depth. Multi-sample formats all store the Pmask bits starting 16*96-bits after the start of the Zplane data. This simplifies the logic without increasing the number of 256-bit accesses.

Note that the Zplane16 mode only stores all 16 Zplanes for multi-sample depth values. For 32-bit single-sample depth values, Zplane16 mode only stores 12 Zplanes, due to the limited size of the tile. Zpane16 format is not supported for 16-bit single-sample depth values, since there is only room in the tile for 8 Zplanes.

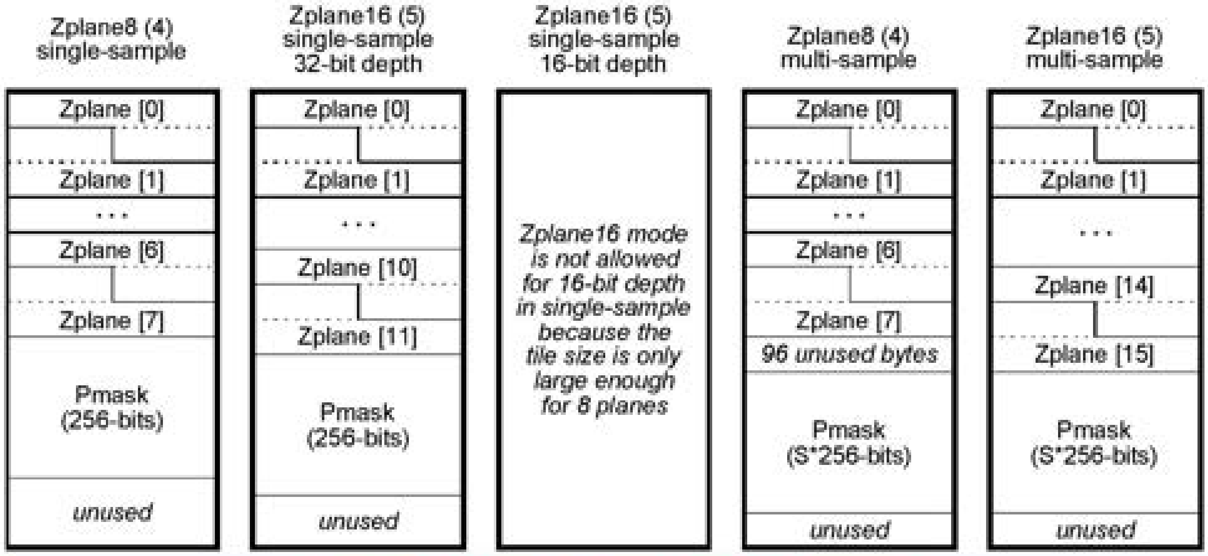


**Figure 34: Eight or More Zplane Depth Tile Formats**

The following table shows how many 128-bit (16-byte) micro-tiles are required to store an 8x8 tile of 16-bit or 24-bit depth data for each of the Zplane modes. To determine the number of 256-bit memory accesses required for each format, divide by 2 and round up. For comparison, the table also lists the number of micro-tiles required in the Separate and Expanded formats. The micro-tiles required for Expanded format are listed in parenthese and includes storage for stencil data, if any. Typically, the total tile size for depth/stencil data equals the size of Expanded mode and the amount of memory allocated for depth data within each tile equals the size of the Separate format. Zplane16 mode

Requires too many micro-tiles when there is one sample per pixel, so Zplane16 mode can only be used with 2 or more samples per pixel.

| Samples per Pixel | Zplane1 (1) | Zplane2 (2) | Zplane4 (3) | Zplane8 (4) | Zplane16 (5) | Separate (or Expanded) (6-7) | |
|---|---|---|---|---|---|---|---|
| | | | | | | **16-bit depth** | **24-bit depth** |
| **1** | 0.75 | 2 | 4 | 8 | (N/A) | 8 (8) | 12 (16) |
| **2** | 0.75 | 2.5 | 5 | 10 | 16 | 16 (16) | 24 (32) |
| **3** | 0.75 | 3.0 | 6 | 12 | 18 | 24 (24) | 36 (48) |
| **4** | 0.75 | 3.5 | 7 | 14 | 20 | 32 (32) | 48 (64) |
| **6** | 0.75 | 4.5 | 9 | 18 | 24 | 48 (48) | 72 (96) |
| **8** | 0.75 | 5.5 | 11 | 22 | 28 | 64 (64) | 96 (128) |

**Table 4: Depth Storage Sizes in Micro-Tiles**

The storage required by the Zplane formats goes up significantly as the number of samples increases, since the number of bits required to store the mask is proportional to the number of samples. A future chip could achieve higher levels of multi-sample compression by encoding the mask data, taking advantage of the fact that adjacent samples typically use the same Zplane.

## 8.4 Pmask Storage Formats

The compressed depth formats store a Pmask value per sample, which specifies the Zplane that must be used to compute the depth at that sample. Zplane8 and Zplane16 modes store 4-bits per Pmask, Zplane 4 mode stores 2-bits per Pmask, and Zplane2 mode stores 1-bit per Plask. Zplane1 mode does not require a Pmask, since the entire tile is covered by a single Zplane.

The Pmask data is stored as an 8x8 tile in a modified micro-tile format. The pmask values for all of the samples for a pixel are combined into a single pmask-pixel, which is then micro-tiled using 64-bit micro-tiles for 4-bit Pmasks, 32-bit micro-tiles for 2-bit Pmasks, and 16-bit micro-tiles for 1-bit Pmasks. This causes the Pmask micro-tiling to match up with the micro-tiling for 8-bit stencil values, which makes it easier for R400 to interleave corresponding stencil and Pmask data in the internal depth/stencil cache.

The figure below shows the Pmask storage pattern for 8-sample pixels. Even and odd scanlines interleave each 16N-bits, where N is the number of bits per Pmask. Each pmask-pixel contains eight N-bit Pmask values. Therefore, a single 16N-bit interleave is filled by two pmask-pixels. As a result, the interleave pattern stores Pmask values for two horizontally adjacent pixels on an even scanline, then two horizontally adjacent pixels on an odd scanline, and so forth. This is like the standard micro-tile format, except that the micro-tile size is 16N-bits instead of 128-bits.

| | 64N-1 48N | 48N-1 32N | 32N-1 16N | 16N-1 0 |
|---|---|---|---|---|
| **0** | pmask (2-3, 1) | pmask (2-3, 0) | pmask (0-1, 1) | pmask (0-1, 0) |
| **1** | pmask (6-7, 1) | pmask (6-7, 0) | pmask (4-5, 1) | pmask (4-5, 0) |
| **2** | pmask (2-3, 3) | pmask (2-3, 2) | pmask (0-1, 3) | pmask (0-1, 2) |
| **3** | pmask (6-7, 3) | pmask (6-7, 2) | pmask (4-5, 3) | pmask (4-5, 2) |
| **4** | pmask (2-3, 5) | pmask (2-3, 4) | pmask (0-1, 5) | pmask (0-1, 4) |
| **5** | pmask (6-7, 5) | pmask (6-7, 4) | pmask (4-5, 5) | pmask (4-5, 4) |
| **6** | pmask (2-3, 7) | pmask (2-3, 6) | pmask (0-1, 7) | pmask (0-1, 6) |
| **7** | pmask (6-7, 7) | pmask (6-7, 6) | pmask (4-5, 7) | pmask (4-5, 6) |

*N-bit Pmask, 8-sample: 16N-bit interleave of even/odd scanlines*

**Figure 35: Pmask Interleave For 8 Samples**

The next figure shows the Pmask storage patterns for 1-sample, 2-sample, and 4-sample pixels. As before, even and odd scanlines interleave each 16N-bits, where N is the number of bits per Pmask. Each pmask-pixel contains eight N-bit Pmask values. Therefore, a single 16N-bit interleave is filled by two pmask-pixels. As a result, the interleave pattern stores Pmask values for two horizontally adjacent pixels on an even scanline, then two horizontally adjacent pixels on an odd scanline, and so forth. This is like the standard micro-tile format, except that the micro-tile size is 16N-bits instead of 128-bits.

| | 64N-1 56N | 56N-1 48N | 48N-1 40N | 40N-1 32N | 32N-1 24N | 24N-1 16N | 16N-1 8N | 8N-1 0 |
|---|---|---|---|---|---|---|---|---|
| 0 | pmsk(0-7,7) | pmsk(0-7,5) | pmsk(0-7,6) | pmsk(0-7,4) | pmsk(0-7,3) | pmsk(0-7,1) | pmsk(0-7,2) | pmsk(0-7,0) |

*N-bit Pmask, 1-sample: 16N-bit interleave of even/odd scanlines*

| | 64N-1 48N | 48N-1 32N | 32N-1 16N | 16N-1 0 |
|---|---|---|---|---|
| 0 | pmask (0-7, 3) | pmask (0-7, 2) | pmask (0-7, 1) | pmask (0-7, 0) |
| 1 | pmask (0-7, 7) | pmask (0-7, 6) | pmask (0-7, 5) | pmask (0-7, 4) |

*N-bit Pmask, 2-sample: 16N-bit interleave of even/odd scanlines*

| | 64N-1 48N | 48N-1 32N | 32N-1 16N | 16N-1 0 |
|---|---|---|---|---|
| 0 | pmask (4-7, 1) | pmask (4-7, 0) | pmask (0-3, 1) | pmask (0-3, 0) |
| 1 | pmask (4-7, 3) | pmask (4-7, 2) | pmask (0-3, 3) | pmask (0-3, 2) |
| 2 | pmask (4-7, 5) | pmask (4-7, 4) | pmask (0-3, 5) | pmask (0-3, 4) |
| 3 | pmask (4-7, 7) | pmask (4-7, 6) | pmask (0-3, 7) | pmask (0-3, 6) |

*N-bit Pmask, 4-sample: 16N-bit interleave of even/odd scanlines*

Figure 36: Pmask Interleave For 1, 2, or 4 Samples

The final figure shows the Pmask storage patterns for 3-sample and 6-sample pixels. For these non-power-of-two sample sizes, the Pmask values for some samples of a pixel may be in a different 16N-bit interleave from the Pmask values for the rest of the samples of a pixel. This is notated by marking pixels with letters *a* to *c* for 3-sample pixels or *a* to *f* for 6-sample pixels. So for example, the interleave marked "pmask (0a-5a, 0)" stores all of the Pmask values for Y=0 and X=0 through 4, and also stores the Pmask for the first sample of pixel (5,0).

| | 64N-1 56N | 56N-1 48N | 48N-1 40N | 40N-1 32N | 32N-1 24N | 24N-1 16N | 16N-1 8N | 8N-1 0 |
|---|---|---|---|---|---|---|---|---|
| 0 | pmsk(0a-2b,3) | pmsk(5b-7c,1) | pmsk(0a-2b,2) | pmsk(5b-7c,0) | pmask (0a-5a, 1) | | pmask (0a-5a,0) | |
| 1 | pmask (0a-5a, 5) | | pmask (0a-5a,4) | | pmask (2c-7c, 3) | | pmask (2c-7,2) | |
| 2 | pmask (2c-7c, 7) | | pmask (2c-7c,6) | | pmsk(0a-2b,7) | pmsk(5b-7c,5) | pmsk(0a-2b,6) | pmsk(5b-7c,4) |

*N-bit Pmask, 3-sample: 16N-bit interleave of even/odd scanlines*

| | 64N-1 48N | 48N-1 32N | 32N-1 16N | 16N-1 0 |
|---|---|---|---|---|
| 0 | pmask (2e-5b, 1) | pmask (2e-5b,0) | pmask (0a-2d, 1) | pmask (0a-2d,0) |
| 1 | pmask (0a-2d, 3) | pmask (0a-2d,2) | pmask (5c-7f,1) | pmask (5c-7f,0) |
| 2 | pmask (5c-7f,3) | pmask (5c-7f,2) | pmask (2e-5b, 3) | pmask (2e-5b,2) |
| 3 | pmask (2e-5b, 5) | pmask (2e-5b,4) | pmask (0a-2d, 5) | pmask (0a-2d,4) |
| 4 | pmask (0a-2d, 7) | pmask (0a-2d,6) | pmask (5c-7f,5) | pmask (5c-7f,4) |
| 5 | pmask (5c-7f,7) | pmask (5c-7f,6) | pmask (2e-5b, 7) | pmask (2e-5b,6) |

*N-bit Pmask, 6-sample: 16N-bit interleave of even/odd scanlines*

Figure 37: Pmask Interleave For 4 and 6 Samples

Another way to think of these storage formats is to treat them as compressed versions of 8-bit storage formats. Imagine padding out the Pmask values to 8-bits, and then storing them as 8S-bit pixels, where there are S samples. This storage format would use the standard 128-bit microtiling. The 4-bit Pmask format above is the same as this 8-bit format, except with bits<7:4> of each byte omitted. This results in a 64-bit micro-tile, as described above. Similarly, 2-bit and 1-bit Pmask values are stored the same as this 8-bit format, except with bits<7:2> and bits<7:1> omitted from each byte, respectively. The RB logic stores each Pmask value with the stencil value for the same sampleThis compression format makes it easier for the RB logic to match them up, since it means that the sequence of stencil values matches the sequence of Pmask values in memory.

# Kaleidoscope Clock Block
# Architecture and Implementation Spec

***AUTHOR***

**JIMMY LAU**

**Revision 1.1~~1~~3**

File Path: //depot/r400/doc_lib/design/blocks/dc/dccg/DCCG.doc~~*March 31,*Jan 26 *2000 4:03:00 PM*~~

***WARNING***
***This document contains confidential information that could be substantially detrimental to the interest of ATI Technologies Inc. through unauthorized use or disclose.***

## Table Of Contents

# 1 Open Issues

List all open issues. Include short description of resolution when closed. This should not be detailed.

## 1.1 Open Feature Issues

1
Issue:
Resolution:

2
Issue:
Resolution:

## 1.2 Open Implementation Issues

1
Issue:
Resolution:

2
Issue:
Resolution:

# 2 Top Level Diagram

DCCG Top Level Diagram : //depot/r400/doc_lib/design/blocks/dc/dccg/dccg.vsd

# 3 Introduction to DCCG

## 3.1 Purpose/Application

DCCG provides core clock and pixel clock branches to various blocks inside DC1 and DC2. It receives clock inputs from external pins or display PLLs, performs reference, feedback and post divisions on the pixel clocks, and finally gate the various core and pixel clock branches based on activities and power states of the chip.

## 3.2 Feature Requirements

### 3.2.1 SCLK Domain Feature List

- Provide gated clock branches for different blocks in DC and DO.
- Busy signals can be extended.
- Provide clock-on signals to monitor the behavior of gated clocks.
- Provide reset signals according to the different reset conditions for different gated clock branches.

### 3.2.2 PCLK Domain Feature List

- Provide functional gated clock branches for different blocks in DO only.
- Fractional feedback divider by using slip request circuitry for PLLs.
- Fractional post divider by using slip request circuitry for PLLs.
- Reference divider for PLLs.
- The clock root can be one of PLL1, PLL2 and external clock source.
- Provide TVCLK branch by using DTO
- Provide clock branches for dual-link(or two single links) for TMDS
- Anti-glitch circuitry to minimize glitch when the frequency is changed.
- One-shot mode for debug purpose.
- Test counter to be read back to monitor the frequency of the clock roots.
- Clock branches can be selected to the Generic A or Generic B output pin through Dispout.
- Different TST clocks for different branches in the TST mode.

## 3.3 Relationship to Other Blocks

The DC is split into DC1 and DC2 in order to bring down the gate count. The SCLK and PCLK branches are distributed as shown in the diagram.

Clock Distribution Diagram for DC and DO

## 3.4 Structure

## 3.5 References

| Purpose | File Path & Name |
|---|---|
| Test Plan | |
| Architectural Outline | |
| Block spec for block that includes this block (if applicable) | |
| Block specs for all sub-blocks of this block (if applicable) | |
| Programming Guide | |
| Others | |

# 4 Input Data and Signal Interface

## 4.1 Input Interface Spec(s)

| Signal Name | Source | Description |
|---|---|---|
| IO_DCCG_pciclk | IO | Raw PCI clock. |
| IO_DCCG_xtalin | IO | Crystal clock. |
| IO_DCCG_generic_sig1 | IO | Generic pin 1 as clock input. |
| IO_DCCG_generic_sig2 | IO | Generic pin 2 as clock input. |
| IO_DCCG_scan_clk | IO | Scan clock input. |
| IO_DCCG_test_clk | IO | Test clock input. |
| CG_DCCG_sclk | CG | Global SCLK. |
| CG_DCCG_sclk_rst | CG | Reset for global SCLK. |
| CG_DCCG_hi_reset | CG | De-glitched PCI reset. |
| CG_DCCG_soft_reset | CG | Global reset. |
| VGA_DCCG_clock_speed | VGA | Pixel clock speed in VGA mode.<br>0 = 25 MHz<br>1 = 28 MHz |
| VGA_DCCG_crtc1_mode_en | VGA | Indicates whether VGA mode is enabled for CRTC1. |
| VGA_DCCG_crtc2_mode_en | VGA | Indicates whether VGA mode is enabled for CRTC2. |
| VGA_DCCG_crtc1_timing_sel | VGA | Indicates whether VGA timing mode is selected for CRTC1. |
| VGA_DCCG_crtc2_timing_sel | VGA | Indicates whether VGA timing mode is selected for CRTC2. |
| P1PLL_DCCG_oclk | PPLL | Output clock from display PLL 1. |
| P1PLL_DCCG_oslclk0 | PPLL | Slipable output clock for post divider from display PLL 1. |
| P1PLL_DCCG_oslclk1 | PPLL | Slipable output clock for feedback divider from display PLL 1. |
| P1PLL_DCCG_slip0_ack | PPLL | Slip clock acknowledge for post divider from display PLL 1. |
| P1PLL_DCCG_slip1_ack | PPLL | Slip clock acknowledge for fractional feedback divider from display PLL 1. |
| P2PLL_DCCG_oclk | P2PLL | Output clock from display PLL 2. |
| P2PLL_DCCG_oslclk0 | P2PLL | Slipable output clock for post divider from display PLL 2. |
| P2PLL_DCCG_oslclk1 | P2PLL | Slipable output clock for feedback divider from display PLL 2. |
| P2PLL_DCCG_slip0_ack | P2PLL | Slip clock acknowledge for post divider from display PLL 2. |
| P2PLL_DCCG_slip1_ack | P2PLL | Slip clock acknowledge for fractional feedback divider from display PLL 2. |
| DISP_DCCG_crtc1_en | DISP | CRTC1 enable, as pixel clock gating condition. |
| DISP_DCCG_crtc2_en | DISP | CRTC2 enable, as pixel clock gating condition. |
| DISP_DCCG_daca_en | DISP | DACA enable, as pixel clock gating condition. |
| DISP_DCCG_dacb_en | DISP | DACB enable, as pixel clock gating condition. |
| DISP_DCCG_tmdsa_en | DISP | TMDSA enable, as pixel clock gating condition. |
| DISP_DCCG_hdcp_en | DISP | HDCP enable, as pixel clock gating condition. |
| DISP_DCCG_dvoa_en | DISP | DVOA enable, as pixel clock gating condition. |
| DISP_DCCG_lvds_en | DISP | LVDS enable, as pixel clock gating condition. |
| DISP_DCCG_tv_en | DISP | TVOUT enable, as pixel clock gating condition. |
| DISP_DCCG_crtc1_vsync_event | DISP | VSYNC occurs for CRTC1. |
| DISP_DCCG_crtc2_vsync_event | DISP | VSYNC occurs for CRTC2. |
| DISP_DCCG_crtc1_hsync_event | DISP | HSYNC occurs for CRTC1. |
| DISP_DCCG_crtc2_hsync_event | DISP | HSYNC occurs for CRTC2. |

| | | |
|---|---|---|
| RBBM_DCCG_regclk_active | RBBM | SCLK gating condition from RBBM. |
| CG_BLK_pm_disable | CG | SCLK gating condition from CG. |
| MH_memclk_active | MH | SCLK gating condition from MH. |
| DCCIF_memclk_active | DCCIF | SCLK gating condition from DCCIF. |
| VGA_busy | VGA | VGA is busy, do not gate the SCLK_G_VGA. |
| DISP_DCP_busy | DISP | DCP is busy, do not gate the SCLK_G_DCP. |
| DISP_SCL1_busy | DISP | SCL1 is busy, do not gate SCLK_G_SCLK1. |
| DISP_SCL1C_busy | DISP | SCL1C is busy, do not gate SCLK_G_SCLK1C. |
| DISP_SCL2_busy | DISP | SCL2 is busy, do not gate SCLK_G_SCLK2. |
| DISP_SCL2C_busy | DISP | SCL2C is busy, do not gate SCLK_G_SCLK2C. |
| RBBMIF_DCCG_addr [16:2] | RBBMIF | Address for DCCG register access. |
| RBBMIF_DCCG_rstr | RBBMIF | Read strobe for DCCG register access. |
| RBBMIF_DCCG_wstr | RBBMIF | Write strobe for DCCG register access. |
| RBBMIF_DCCG_wben [3:0] | RBBMIF | Byte enable for DCCG register access. |
| RBBMIF_DCCG_dec | RBBMIF | Decode signal specific for DCCG register access. |
| RBBMIF_DCCG_wrdata [31:0] | RBBMIF | Write data for DCCG register access. |
| OTHERS_DCCG_rddata [31:0] | ? | Input from register read data daisy chain. |

### 4.2

### 4.3 Used Subset of Shared Bus

### 4.4 Control Signals Affecting Input Data

### 4.5 Input Interface State Machine(s)

# 5 Output Data and Signal Interface

## 5.1 Output Interface Spec(s)

| Signal Name | Destination | Description |
|---|---|---|
| DCCG_P1PLL_reset | PPLL | Reset signal to display PLL 1. |
| DCCG_P1PLL_sleep | PPLL | Power down signal to display PLL 1. |
| DCCG_P1PLL_pcp [2:0] | PPLL | Charge pump gain for display PLL 1. |
| DCCG_P1PLL_pdc [1:0] | PPLL | Duty cycle for display PLL 1. |
| DCCG_P1PLL_pvg [2:0] | PPLL | VCO gain for display PLL 1. |
| DCCG_P1PLL_tcpoff | PPLL | Force output of charge pump for display PLL 1 to become high impedance. |
| DCCG_P1PLL_tvcomax | PPLL | When TCPOFF is high, force VCO to run at maximum possible frequency for display PLL 1. |
| DCCG_P1PLL_refsel | PPLL | Selects reference clock input for display PLL 1. |
| DCCG_P1PLL_fbsel | PPLL | Selects feedback clock input for display PLL 1. |
| DCCG_P1PLL_refclk0 | PPLL | Reference clock 0 for display PLL 1. |
| DCCG_P1PLL_refclk1 | PPLL | Reference clock 1 for display PLL 1. |
| DCCG_P1PLL_fbclk0 | PPLL | Feedback clock 0 for display PLL 1. |
| DCCG_P1PLL_fbclk1 | PPLL | Feedback clock 1 for display PLL 1. |
| DCCG_P1PLL_slip0_req | PPLL | Slip clock request for post divider to display PLL 1. |
| DCCG_P1PLL_slip1_req | PPLL | Slip clock request for fractional feedback divider to display PLL 1. |
| DCCG_P2PLL_reset | P2PLL | Reset signal to display PLL 2. |
| DCCG_P2PLL_sleep | P2PLL | Power down signal to display PLL 2. |
| DCCG_P2PLL_pcp [2:0] | P2PLL | Charge pump gain for display PLL 2. |
| DCCG_P2PLL_pdc [1:0] | P2PLL | Duty cycle for display PLL 2. |
| DCCG_P2PLL_pvg [2:0] | P2PLL | VCO gain for display PLL 2. |
| DCCG_P2PLL_tcpoff | P2PLL | Force output of charge pump for display PLL 2 to become high impedance. |
| DCCG_P2PLL_tvcomax | P2PLL | When TCPOFF is high, force VCO to run at maximum possible frequency for display PLL 2. |
| DCCG_P2PLL_refsel | P2PLL | Selects reference clock input for display PLL 2. |
| DCCG_P2PLL_fbsel | P2PLL | Selects feedback clock input for display PLL 2. |
| DCCG_P2PLL_refclk0 | P2PLL | Reference clock 0 for display PLL 2. |
| DCCG_P2PLL_refclk1 | P2PLL | Reference clock 1 for display PLL 2. |
| DCCG_P2PLL_fbclk0 | P2PLL | Feedback clock 0 for display PLL 2. |
| DCCG_P2PLL_fbclk1 | P2PLL | Feedback clock 1 for display PLL 2. |
| DCCG_P2PLL_slip0_req | P2PLL | Slip clock request for post divider to display PLL 2. |
| DCCG_P2PLL_slip1_req | P2PLL | Slip clock request for fractional feedback divider to display PLL 2. |
| DCCG_DISP_pclk_crtc1 | DISP | PCLK for CRTC 1. |
| DCCG_DISP_pclk_crtc2 | DISP | PCLK for CRTC 2. |
| DCCG_DISP_pclk_daca | DISP | PCLK for DAC A |
| DCCG_DISP_pclk_dacb | DISP | PCLK for DAC B |
| DCCG_DISP_pclk_tmdsa | DISP | PCLK for TMDSA. |
| DCCG_DISP_pclk_hdcp | DISP | PCLK for HDCP. |

| | | |
|---|---|---|
| DCCG_DISP_pclk_dvoa | DISP | PCLK for DVOA. |
| DCCG_DISP_pclk_lvds | DISP | PCLK for LVDS. |
| DCCG_DISP_pclk_tv | DISP | PCLK for TVOUT. |
| DCCG_DISP_pclk_crtc1_rst | DISP | PCLK reset for CRTC 1. |
| DCCG_DISP_pclk_crtc2_rst | DISP | PCLK reset for CRTC 2. |
| DCCG_DISP_pclk_daca_rst | DISP | PCLK reset for DAC A |
| DCCG_DISP_pclk_dacb_rst | DISP | PCLK reset for DAC B |
| DCCG_DISP_pclk_tmdsa_rst | DISP | PCLK reset for TMDSA. |
| DCCG_DISP_pclk_hdcp_rst | DISP | PCLK reset for HDCP. |
| DCCG_DISP_pclk_dvoa_rst | DISP | PCLK reset for DVOA. |
| DCCG_DISP_pclk_lvds_rst | DISP | PCLK reset for LVDS. |
| DCCG_DISP_pclk_tv_rst | DISP | PCLK reset for TVOUT. |
| DCCG_sclk_p_dc | DC | Permanent SCLK for DC. |
| DCCG_sclk_r_rbbmif | RBBMIF | Globally gated SCLK for RBBMIF. |
| DCCG_sclk_m | DMIF, DCCIF, DCCARB | Globally gated SCLK for DMIF, DCCIF, DCCARB. |
| DCCG_sclk_r_disp | DISP | Register clock for DISP and VGA. |
| DCCG_sclk_r_tvout | TVOUT | Register clock for TVOUT. |
| DCCG_sclk_r_vip | VIP | Register clock for VIP. |
| DCCG_sclk_g_vga | VGA | Functionally gated SCLK for VGA. |
| DCCG_sclk_g_dcp | DCP, LB | Functionally gated SCLK for DCP and write buffer in LB. |
| DCCG_sclk_g_scl1 | SCL1 | Functionally gated SCLK for data pipe of scaler 1. |
| DCCG_sclk_g_scl1c | SCL1C | Functionally gated SCLK for control logic of scaler 1. |
| DCCG_sclk_g_scl2 | SCL2 | Functionally gated SCLK for data pipe of scaler 2. |
| DCCG_sclk_g_scl2c | SCL2C | Functionally gated SCLK for control logic of scaler 2. |
| DCCG_sclk_r_rbbmif_rst | RBBMIF | Reset for DCCG_sclk_r_rbbmif. |
| DCCG_sclk_m_rst | DMIF, DCCIF, DCCARB | Reset for DCCG_sclk_m. |
| DCCG_sclk_r_disp_rst | DISP | Reset for DCCG_sclk_r_disp. |
| DCCG_sclk_r_tvout_rst | TVOUT | Reset for DCCG_sclk_r_tvout. |
| DCCG_sclk_r_vip_rst | VIP | Reset for DCCG_sclk_r_vip. |
| DCCG_sclk_g_vga_rst | VGA | Reset for DCCG_sclk_g_vga. |
| DCCG_sclk_g_dcp_rst | DCP, LB | Reset for DCCG_sclk_g_dcp. |
| DCCG_sclk_g_scl1_rst | SCL1 | Reset for DCCG_sclk_g_scl1. |
| DCCG_sclk_g_scl1c_rst | SCL1C | Reset for DCCG_sclk_g_scl1c. |
| DCCG_sclk_g_scl2_rst | SCL2 | Reset for DCCG_sclk_g_scl2. |
| DCCG_sclk_g_scl2c_rst | SCL2C | Reset for DCCG_sclk_g_scl2c. |
| DCCG_OTHERS_rddata [31:0] | ? | Output to register read data daisy chain. |

## 5.2 Control Signals Affecting Output Data

## 5.3 Output Interface State Machine(s)

# 6 Data Processing Algorithms

# 7 Hardware Implementation

## 7.1 Pipelining and Processing Logic Blocks

## 7.2 Memory Requirements

## 7.3 State Machine(s) and Control Logic

## 7.4 Timing and Data Flow Control

## 7.5 Clocking and Power Management

# 8 Gate Area and Macros

## 8.1 Pre-Implementation Estimates of Pipelining and Gate Area

## 8.2 Actual Gate Area

# 9 Block Programming Guide

## 9.1 Register Field List

| Field Name | Bits | R/W | Default | Description |
|---|---|---|---|---|
| VGA25_PPLL_REF_DIV_SRC [2:0] | 3 | R/W | 0x0 | Source clock selection for the display PLLs reference divider in VGA timing mode with 25MHz pixel clock.<br>0 = XTALIN<br>1 = GENERIC_SIG1<br>2 = GENERIC_SIG2<br>3 = spread spectrum clock ? |
| VGA28_PPLL_REF_DIV_SRC [2:0] | 3 | R/W | 0x0 | Source clock selection for the display PLLs reference divider in VGA timing mode with 28MHz pixel clock. |
| EXT1_PPLL_REF_DIV_SRC [2:0] | 3 | R/W | 0x0 | Source clock selection for display PLL1 reference divider in extended timing mode. |
| EXT2_PPLL_REF_DIV_SRC [2:0] | 3 | R/W | 0x0 | Source clock selection for display PLL2 reference divider in extended timing mode. |
| VGA25_PPLL_REF_DIV [9:0] | 10 | R/W | 0x? | Reference divider value for display PLLs in VGA timing mode with 25MHz pixel clock. |
| VGA28_PPLL_REF_DIV [9:0] | 10 | R/W | 0x? | Reference divider value for display PLLs in VGA timing mode with 28MHz pixel clock. |
| EXT1_PPLL_REF_DIV [9:0] | 10 | R/W | 0x0 | Reference divider value for display PLL1 in extended timing mode. |
| EXT2_PPLL_REF_DIV [9:0] | 10 | R/W | 0x0 | Reference divider value for display PLL2 in extended timing mode. |
| EXT1_PPLL_UPDATE_LOCK | 1 | R/W | 0x0 | Lock bit for double buffering of display PLL1 registers in extended timing mode. If lock bit is enabled, any update in the registers will not be copied to the active buffers. Otherwise, any update in the registers will be copied to the active buffers in the next rising edge of the reference clock input to the reference divider. |
| EXT1_PPLL_UPDATE_PENDING | 1 | R | | Readback for double buffering status of display PLL1 registers in extended timing mode.<br>0 = update done<br>1 = update pending |
| EXT1_PPLL_UPDATE_SYNC | 1 | R/W | 0x0 | Selects trigger position of double buffering of display PLL1 registers in extended timing mode.<br>0 = update ASAP<br>1 = update in VSYNC region |
| EXT2_PPLL_UPDATE_LOCK | 1 | R/W | 0x0 | Lock bit for double buffering of display PLL2 registers in extended timing mode. |
| EXT2_PPLL_UPDATE_PENDING | 1 | R | | Readback for double buffering status of display PLL2 registers in extended timing mode. |
| EXT2_PPLL_UPDATE_SYNC | 1 | R/W | 0x0 | Selects trigger position of double buffering of display PLL2 registers in extended timing mode. |

| | | | | |
|---|---|---|---|---|
| EXT1_HTOT_SLIP [2:0] | 3 | R/W | 0x0 | Selects number of 1/5 PLLCLK phase slips to do in display PLL1 in extended timing mode, at every HSYNC, for post divider. |
| EXT1_HTOT_CNTL_EDGE | 1 | R/W | 0x0 | Selects which HSYNC edge the slip correction for post divider is done for display PLL1 in extended timing mode. |
| EXT1_HTOT_CNTL_W | 1 | W<br>(W trigger) | 0x0 | Writing to this register triggers the update of the slip count from pending buffer to active buffer. It is associated with the HTOTAL slip control in display PLL1 in extended timing mode. |
| EXT2_HTOT_SLIP [2:0] | 3 | R/W | 0x0 | Selects number of 1/5 PLLCLK phase slips to do in display PLL2 in extended timing mode, at every HSYNC, for post divider. |
| EXT2_HTOT_CNTL_EDGE | 1 | R/W | 0x0 | Selects which HSYNC edge the slip correction for post divider is done for display PLL2 in extended timing mode. |
| EXT2_HTOT_CNTL_W | 1 | W<br>(W trigger) | 0x0 | Writing to this register triggers the update of the slip count from pending buffer to active buffer. It is associated with the HTOTAL slip control in display PLL2 in extended timing mode. |
| VGA25_PPLL_FB_DIV [10:0] | 11 | R/W | 0x0 | Feedback divider value for display PLLs in VGA timing mode with 25MHz pixel clock. |
| VGA28_PPLL_FB_DIV [10:0] | 11 | R/W | 0x0 | Feedback divider value for display PLLs in VGA timing mode with 28MHz pixel clock. |
| EXT1_PPLL_FB_DIV [10:0] | 11 | R/W | 0x0 | Feedback divider value for display PLL1 in extended timing mode. |
| EXT2_PPLL_FB_DIV [10:0] | 11 | R/W | 0x0 | Feedback divider value for display PLL2 in extended timing mode. |
| VGA25_PPLL_FB_DIV_FRACTION [2:0] | 3 | R/W | 0x0 | Selects number of 1/5 PLLCLK phase slips for display PLLs fractional feedback divider in VGA timing mode with 25MHz pixel clock. |
| VGA28_PPLL_FB_DIV_FRACTION [2:0] | 3 | R/W | 0x0 | Selects number of 1/5 PLLCLK phase slips for display PLLs fractional feedback divider in VGA timing mode with 28MHz pixel clock. |
| EXT1_PPLL_FB_DIV_FRACTION [2:0] | 3 | R/W | 0x0 | Selects number of 1/5 PLLCLK phase slips for display PLL1 fractional feedback divider in extended timing mode. |
| EXT2_PPLL_FB_DIV_FRACTION [2:0] | 3 | R/W | 0x0 | Selects number of 1/5 PLLCLK phase slips for display PLL2 fractional feedback divider in extended timing mode. |
| VGA25_PCLK_SRC [2:0] | 3 | R/W | 0x0 | Selects source of main pixel clocks in VGA timing mode with 25MHz pixel clock.<br>0 = CPUCLK<br>1 = SCAN clock<br>2 = output clock of display PLL.<br>3 = GENERIC_SIG1<br>4 = GENERIC_SIG2 |
| VGA28_PCLK_SRC [2:0] | 3 | R/W | 0x0 | Selects source of main pixel clocks in VGA timing mode with 28MHz pixel clock. |

| | | | | |
|---|---|---|---|---|
| EXT1_PCLK_SRC [2:0] | 3 | R/W | 0x0 | Selects source of main pixel clock 1 in extended timing mode. |
| EXT2_PCLK_SRC [2:0] | 3 | R/W | 0x0 | Selects source of main pixel clock 2 in extended timing mode. |
| VGA25_PPLL_POST_DIV [3:0] | 4 | R/W | 0x0 | Post divider value for display PLLs in VGA timing mode with 25MHz pixel clock.<br>0 = divided by 1<br>1 = divided by 2<br>2 = divided by 4<br>3 = divided by 8<br>4 = divided by 3<br>5 = divided by 16<br>6 = divided by 6<br>7 = divided by 12<br>8 = divided by 24<br>9 = divided by 32 |
| VGA28_PPLL_POST_DIV [3:0] | 4 | R/W | 0x0 | Post divider value for display PLLs in VGA timing mode with 28MHz pixel clock. |
| EXT1_PPLL_POST_DIV [3:0] | 4 | R/W | 0x0 | Post divider value for display PLL1 in extended timing mode. |
| EXT2_PPLL_POST_DIV [3:0] | 4 | R/W | 0x0 | Post divider value for display PLL2 in extended timing mode. |
| VGA25_PPLL_PCP [2:0] | 3 | R/W | 0x4 | Charge pump gain for display PLLs in VGA timing mode with 25MHz pixel clock. |
| VGA25_PPLL_PDC [1:0] | 2 | R/W | 0x2 | Duty cycle for display PLLs in VGA timing mode with 25MHz pixel clock. |
| VGA25_PPLL_PVG [2:0] | 3 | R/W | 0x4 | VCO gain for display PLLs in VGA timing mode with 25MHz pixel clock. |
| VGA25_PPLL_TCPOFF | 1 | R/W | 0x0 | Force output of charge pump for display PLLs to become high-impedance in VGA timing mode with 25MHz pixel clock. |
| VGA25_PPLL_TVCOMAX | 1 | R/W | 0x0 | when TCPOFF is high, force VCO to run at maximum possible frequency for display PLLs in VGA timing mode with 25MHz pixel clock. |
| VGA28_PPLL_PCP [2:0] | 3 | R/W | 0x4 | Charge pump gain for display PLLs in VGA timing mode with 28MHz pixel clock. |
| VGA28_PPLL_PDC [1:0] | 2 | R/W | 0x2 | Duty cycle for display PLLs in VGA timing mode with 28MHz pixel clock. |
| VGA28_PPLL_PVG [2:0] | 3 | R/W | 0x4 | VCO gain for display PLLs in VGA timing mode with 28MHz pixel clock. |
| VGA28_PPLL_TCPOFF | 1 | R/W | 0x0 | Force output of charge pump for display PLLs to become high-impedance in VGA timing mode with 28MHz pixel clock. |
| VGA28_PPLL_TVCOMAX | 1 | R/W | 0x0 | when TCPOFF is high, force VCO to run at maximum possible frequency for display PLLs in VGA timing mode with 28MHz pixel clock. |
| EXT1_PPLL_PCP [2:0] | 3 | R/W | 0x4 | Charge pump gain for display PLL1 in extended timing mode. |
| EXT1_PPLL_PDC [1:0] | 2 | R/W | 0x2 | Duty cycle for display PLL1 in extended timing mode. |
| EXT1_PPLL_PVG [2:0] | 3 | R/W | 0x4 | VCO gain for display PLL1 in extended timing mode. |

| | | | | |
|---|---|---|---|---|
| EXT1_PPLL_TCPOFF | 1 | R/W | 0x0 | Force output of charge pump for display PLL1 to become high-impedance in extended timing mode. |
| EXT1_PPLL_TVCOMAX | 1 | R/W | 0x0 | when TCPOFF is high, force VCO to run at maximum possible frequency for display PLL1 in extended timing mode. |
| EXT2_PPLL_PCP [2:0] | 3 | R/W | 0x4 | Charge pump gain for display PLL2 in extended timing mode. |
| EXT2_PPLL_PDC [1:0] | 2 | R/W | 0x2 | Duty cycle for display PLL2 in extended timing mode. |
| EXT2_PPLL_PVG [2:0] | 3 | R/W | 0x4 | VCO gain for display PLL2 in extended timing mode. |
| EXT2_PPLL_TCPOFF | 1 | R/W | 0x0 | Force output of charge pump for display PLL2 to become high-impedance in extended timing mode. |
| EXT2_PPLL_TVCOMAX | 1 | R/W | 0x0 | when TCPOFF is high, force VCO to run at maximum possible frequency for display PLL2 in extended timing mode. |
| P1PLL_REFCLK_SEL | 1 | R/W | 0x0 | Enables jitter rejecter on display PLL1 reference clock. |
| P1PLL_FBCLK_SEL | 1 | R/W | 0x0 | Enables jitter rejecter on display PLL1 feedback clock. |
| P1PLL_RESET | 1 | R/W | 0x1 | Resets display PLL1. |
| P1PLL_SLEEP | 1 | R/W | 0x1 | Power down display PLL1. |
| P2PLL_REFCLK_SEL | 1 | R/W | 0x0 | Enables jitter rejecter on display PLL2 reference clock. |
| P2PLL_FBCLK_SEL | 1 | R/W | 0x0 | Enables jitter rejecter on display PLL2 feedback clock. |
| P2PLL_RESET | 1 | R/W | 0x1 | Resets display PLL2. |
| P2PLL_SLEEP | 1 | R/W | 0x1 | Power down display PLL2. |
| P1PLL_TIMING_MODE_STATUS [1:0] | 2 | R | | Readback of timing mode in which display PLL 1 is running in.<br>0 = extended timing mode.<br>1 = reserved.<br>2 = VGA timing mode with 25MHz pixel clock.<br>3 = VGA timing mode with 28MHz pixel clock. |
| P2PLL_TIMING_MODE_STATUS [1:0] | 2 | R | | Readback of timing mode in which display PLL 2 is running in. |
| PCLK_CRTC1_RESET | 1 | R/W | 0x0 | Soft reset of PCLK_CRTC1. |
| PCLK_CRTC1_GATE_DISABLE | 1 | R/W | 0x0 | Disables functional gating of PCLK_CRTC1. |
| PCLK_CRTC2_RESET | 1 | R/W | 0x0 | Soft reset of PCLK_CRTC2. |
| PCLK_CRTC2_GATE_DISABLE | 1 | R/W | 0x0 | Disables functional gating of PCLK_CRTC2. |
| PCLK_DACA_RESET | 1 | R/W | 0x0 | Soft reset of PCLK_DACA. |
| PCLK_DACA_GATE_DISABLE | 1 | R/W | 0x0 | Disables functional gating of PCLK_DACA. |
| PCLK_DACB_RESET | 1 | R/W | 0x0 | Soft reset of PCLK_DACB. |
| PCLK_DACB_GATE_DISABLE | 1 | R/W | 0x0 | Disables functional gating of PCLK_DACB. |
| PCLK_TMDSA_RESET | 1 | R/W | 0x0 | Soft reset of PCLK_TMDSA. |
| PCLK_TMDSA_GATE_DISABLE | 1 | R/W | 0x0 | Disables functional gating of PCLK_TMDSA. |

| | | | | |
|---|---|---|---|---|
| PCLK_HDCP_RESET | 1 | R/W | 0x0 | Soft reset of PCLK_HDCP. |
| PCLK_HDCP_GATE_DISABLE | 1 | R/W | 0x0 | Disables functional gating of PCLK_HDCP. |
| PCLK_DVOA_RESET | 1 | R/W | 0x0 | Soft reset of PCLK_DVOA. |
| PCLK_DVOA_GATE_DISABLE | 1 | R/W | 0x0 | Disables functional gating of PCLK_DVOA. |
| PCLK_LVDS_RESET | 1 | R/W | 0x0 | Soft reset of PCLK_LVDS. |
| PCLK_LVDS_GATE_DISABLE | 1 | R/W | 0x0 | Disables functional gating of PCLK_LVDS. |
| PCLK_TV_RESET | 1 | R/W | 0x0 | Soft reset of PCLK_TV. |
| PCLK_TV_GATE_DISABLE | 1 | R/W | 0x0 | Disables functional gating of PCLK_TV. |
| CRTC1_CLK_SRC | 1 | R/W | 0x0 | Selects clock source for PCLK_CRTC1.<br>0 = main pixel clock 1.<br>1 = main pixel clock 2. |
| CRTC1_CLK_INV | 1 | R/W | 0x0 | Chooses inverted clock source for PCLK_CRTC1.<br>0 = non-inverted clock source.<br>1 = inverted clock source. |
| CRTC2_CLK_SRC | 1 | R/W | 0x0 | Selects clock source for PCLK_CRTC2. |
| CRTC2_CLK_INV | 1 | R/W | 0x0 | Chooses inverted clock source for PCLK_CRTC2. |
| DACA_CLK_SRC | 1 | R/W | 0x0 | Selects clock source for PCLK_DACA. |
| DACA_CLK_INV | 1 | R/W | 0x0 | Chooses inverted clock source for PCLK_DACA. |
| DACB_CLK_SRC | 1 | R/W | 0x0 | Selects clock source for PCLK_DACB. |
| DACB_CLK_INV | 1 | R/W | 0x0 | Chooses inverted clock source for PCLK_DACB. |
| TMDSA_CLK_SRC | 1 | R/W | 0x0 | Selects clock source for PCLK_TMDSA. |
| TMDSA_CLK_INV | 1 | R/W | 0x0 | Chooses inverted clock source for PCLK_TMDSA. |
| HDCP_CLK_SRC | 1 | R/W | 0x0 | Selects clock source for PCLK_HDCP. |
| HDCP_CLK_INV | 1 | R/W | 0x0 | Chooses inverted clock source for PCLK_HDCP. |
| DVO_CLK_SRC | 1 | R/W | 0x0 | Selects clock source for PCLK_DVO. |
| DVOA_CLK_INV | 1 | R/W | 0x0 | Chooses inverted clock source for PCLK_DVOA. |
| LVDS_CLK_SRC | 1 | R/W | 0x0 | Selects clock source for PCLK_LVDS. |
| LVDS_CLK_INV | 1 | R/W | 0x0 | Chooses inverted clock source for PCLK_LVDS. |
| REGCLK_DISP_RESET | 1 | R/W | 0x0 | Soft reset of SCLK_R_DISP. |
| REGCLK_DISP_GATE_DISABLE | 1 | R/W | 0x0 | Disable gating of SCLK_R_DISP. |
| REGCLK_TV_RESET | 1 | R/W | 0x0 | Soft reset of SCLK_R_TVOUT. |
| REGCLK_TV_GATE_DISABLE | 1 | R/W | 0x0 | Disable gating of SCLK_R_TVOUT. |
| REGCLK_VIP_RESET | 1 | R/W | 0x0 | Soft reset of SCLK_R_VIP. |
| REGCLK_VIP_GATE_DISABLE | 1 | R/W | 0x0 | Disable gating of SCLK_R_VIP. |
| SCLK_VGA_RESET | 1 | R/W | 0x0 | Soft reset of SCLK_G_VGA. |

| | | | | |
|---|---|---|---|---|
| SCLK_VGA_GATE_DISABLE | 1 | R/W | 0x0 | Disable gating of SCLK_G_VGA. |
| SCLK_DCP_RESET | 1 | R/W | 0x0 | Soft reset of SCLK_G_DCP. |
| SCLK_DCP_GATE_DISABLE | 1 | R/W | 0x0 | Disable gating of SCLK_G_DCP. |
| SCLK_SCL1_RESET | 1 | R/W | 0x0 | Soft reset of SCLK_G_SCL1. |
| SCLK_SCL1_GATE_DISABLE | 1 | R/W | 0x0 | Disable gating of SCLK_G_SCL1. |
| SCLK_SCL1C_RESET | 1 | R/W | 0x0 | Soft reset of SCLK_G_SCL1C. |
| SCLK_SCL1C_GATE_DISABLE | 1 | R/W | 0x0 | Disable gating of SCLK_G_SCL1C. |
| SCLK_SCL2_RESET | 1 | R/W | 0x0 | Soft reset of SCLK_G_SCL2. |
| SCLK_SCL2_GATE_DISABLE | 1 | R/W | 0x0 | Disable gating of SCLK_G_SCL2. |
| SCLK_SCL2C_RESET | 1 | R/W | 0x0 | Soft reset of SCLK_G_SCL2C. |
| SCLK_SCL2C_GATE_DISABLE | 1 | R/W | 0x0 | Disable gating of SCLK_G_SCL2C. |

# 10 sub-module Specification

## 10.1 SCLK Domain

Although the CG block is responsible for the generation of the core clock from the PLLs for the entire chip, the SG provides the gated version of this core clock for the DC. These gated clock branches are generated based on the different clock gating conditions from the various blocks. The single gater structure is shown in the Figure of "SCLK gater.

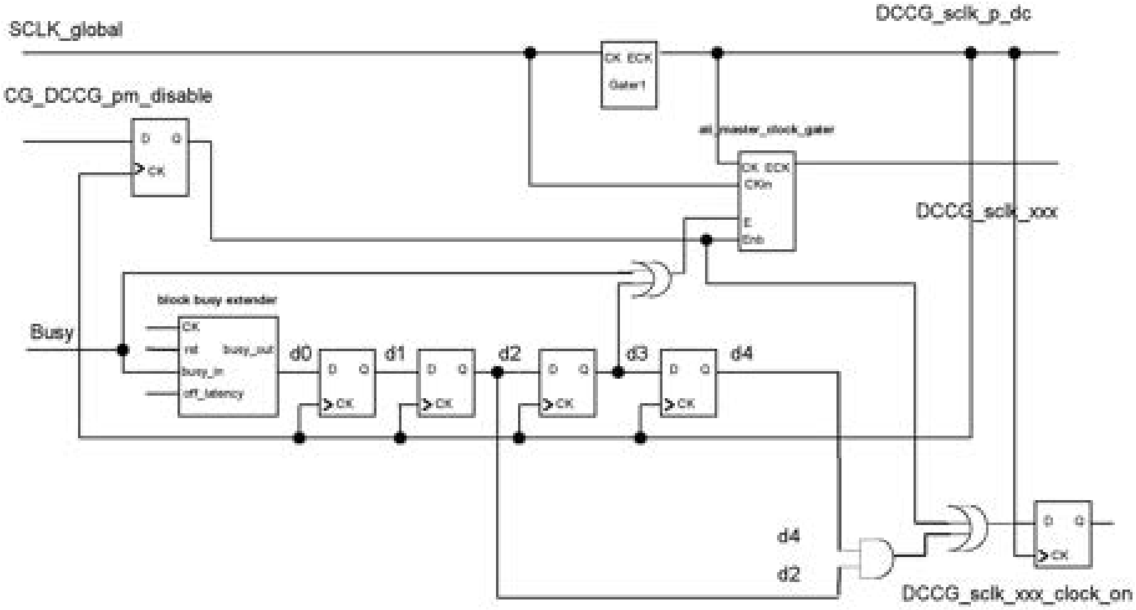


Figure 1 SCLK Gater

The "block busy extender" extends the busy signal to a number of "turn off latency" programmed by a register of 6-bits. There is an "or" gate and a flop inside "ati master clock gater". The result of "or" gate, which is "OR" of the enable signal and the permanent disable signal, drives the gater to turn on the gated clock. The pipelined flops are used to turn on the gated clock as soon as possible when the rising edge arrives, but turn off the gated clock a few clocks later when the falling edge occurs. The clock-on signal is asserted a few clocks later when the gated clock is turned on and de-asserted a few clocks early when the gated clock is turned off. In this way, it guarantees that within the "high" period of clock-on signal, there is always a few clocks margin for the gated clock. The timing diagram is shown in the Figure of "SCLK Timing Diagram".

It should be pointed that the gated clock branches for Scaler1 and Scaler2 are slight different from all the other branches. The clock-on signal is asserted 2 clocks early in order to save two entries of synchronization fifo.

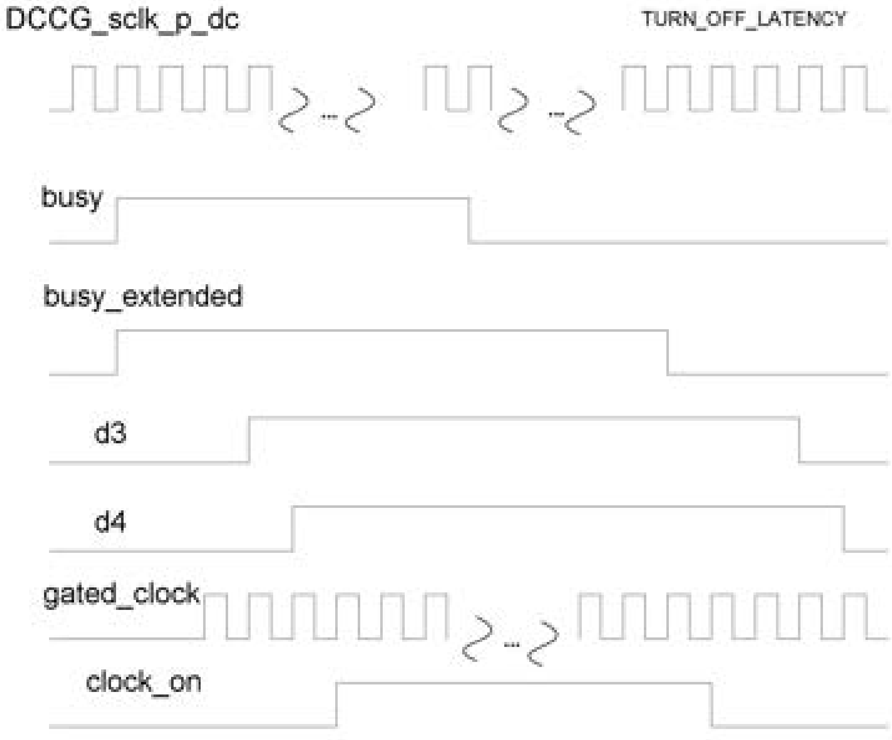


**Figure 2 SCLK Timing Diagram**

The SG receives busy signals or active signals from the various blocks in DC. It combines these signals to generate the internal busy signals to drive the "SCLK gater". If the SCLK_GATE_DISABLE remains to high, which is programmed through register write, all gated clock function is disabled in SCLK domain. Different register can also control the gated clock function for individual sclk branch.

The gated clock conditions of different branches are listed bellow.

sclk_r_rbbmif_busy = Q_RBBM_DCCG_regclk_active | Q_SCLK_GATE_DISABLE

sclk_m_busy = Q_SCLK_Q_DCT_DC_memreq | DISP_DCCG_cursor_busy | DISP_DCCG_icon_busy |
DCCIF_dc_busy | VIP_DCCG_busy |
Q_SCLK_M_GATE_DISABLE Q_SCLK_GATE_DISABLE

sclk_r_disp_busy = RBBMIF_DCCG_dispreg_busy | Q_REGCLK_DISP_GATE_DISABLE |
Q_SCLK_GATE_DISABLE

sclk_r_tvout_busy = RBBMIF_DCCG_tvoutreg_busy | Q_REGCLK_TV_GATE_DISABLE |
Q_SCLK_GATE_DISABLE

sclk_r_vip_busy = RBBMIF_DCCG_vgareg_busy | RBBMIF_DCCG_vipreg_busy |
Q_REGCLK_VIP_GATE_DISABLE | Q_SCLK_GATE_DISABLE

sclk_g_vga_busy = VGA_DCCG_busy | Q_SCLK_VGA_GATE_DISABLE | Q_SCLK_GATE_DISABLE

sclk_g_vip_busy = VIP_DCCG_busy | Q_SCLK_VIP_GATE_DISABLE | Q_SCLK_GATE_DISABLE

sclk_g_dcp_busy = DCP_DCCG_busy | RUN_CLK_DEBUG_DCP_LB_SCL_SCLK |
Q_SCLK_DCP_GATE_DISABLE | Q_SCLK_GATE_DISABLE

sclk_g_scl1_busy = SCL1_DCCG_busy | RUN_CLK_DEBUG_DCP_LB_SCL_SCLK |
Q_SCLK_DCP_GATE_DISABLE | Q_SCLK_GATE_DISABLE

sclk_g_scl2_busy = SCL2_DCCG_busy | RUN_CLK_DEBUG_DCP_LB_SCL_SCLK |
Q_SCLK_DCP_GATE_DISABLE | Q_SCLK_GATE_DISABLE

## 10.2 PCLK Domain

The DCCG in DO is responsible for generating pixel clocks for different display mode. The frequency source can be PLL1, PLL2 or external source depending on the frequency that the mode requires. The top-level diagram of PCLK shows in the Figure PCLK Top Diagram. The main pixel clock1 and main pixel clock2 can be considered as the two clock roots.

PCLK Top Diagram

### 10.2.1 DISPLAY PLL DIVIDERS

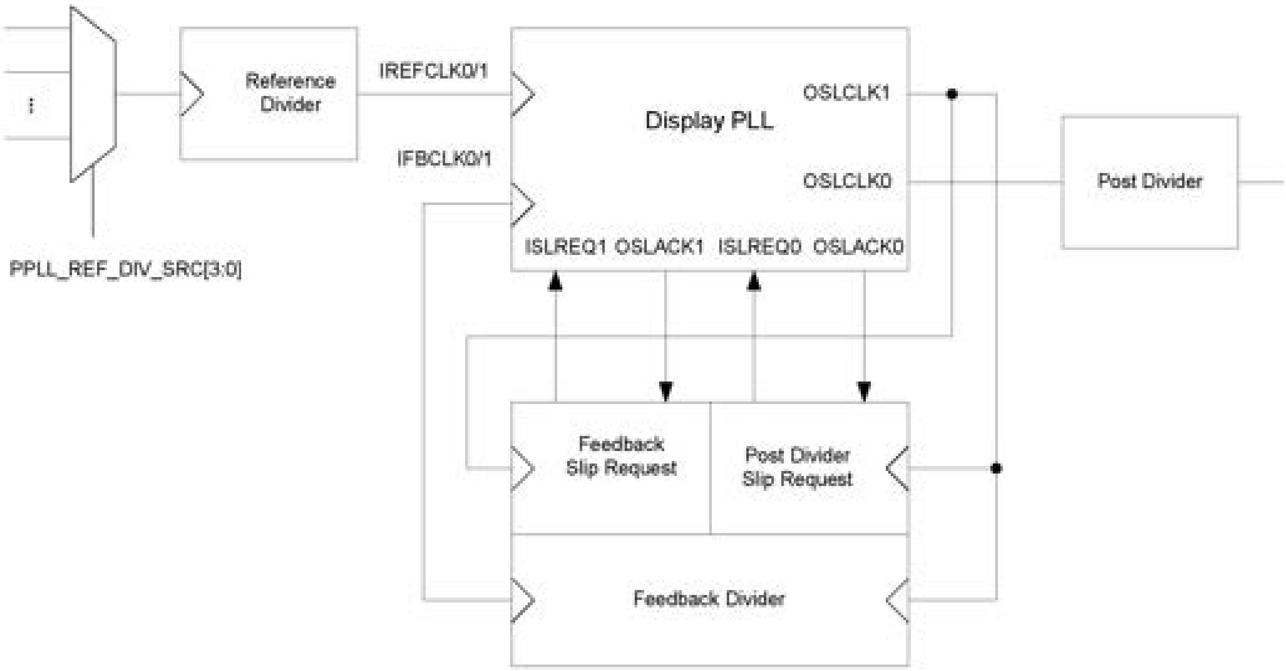


Figure 3 Display PLL with Dividers

There are three dividers associated with each display PLL.

(1) Reference Divider (Incoming clock is divided by a factor of M)
(2) Feedback Divider (Incoming clock is divided by a factor of N)
(3) Post Divider (Incoming clock is divider by a factor of P)

These three dividers, together with the frequency of the input clock, will determine the frequency of the output clock.

$$f_{out} = f_{in} \times \frac{N}{M \times P}$$

In the following subsections, we will look at each divider in more detail.

### 10.2.2 Reference Divider

The diagram below depicts the reference divider.

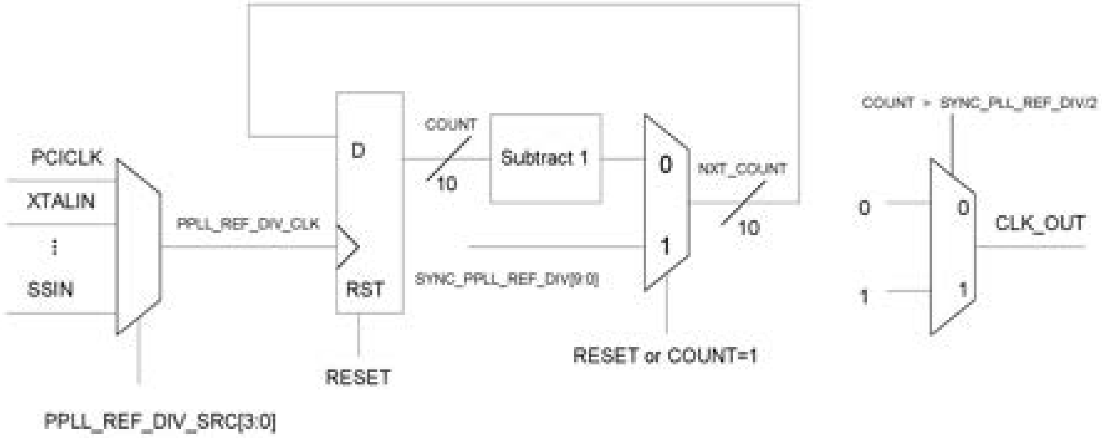


**Figure 4 Reference Divider of PLL**

The input to the reference divider is one of the following clock sources :

(1) PCICLK
(2) Crystal clock
(3) GENERICA
(4) GENERICB
(5) VPCLK
(6) VIPCLK
(7) DVOCLK1
(8) HSYNCA
(9) HSYNCB
(10) SSIN

The signal PPLL_REF_DIV_SRC controls which one of the above clock sources goes into the reference divider. PPLL_REF_DIV_SRC gets its value from one of the four sets of registers, depending on the timing mode (VGA or extended) and which display PLL (P1PLL or P2PLL).

The algorithm used inside the reference divider is quite simple. At reset, the counter inside the reference divider is initially loaded with the divider value (M), from one of the 4 sets of reference divider registers, also depending on the timing mode and which display PLL. The counter, which runs at the input clock of the reference divider, will be decrement by 1 after each clock. When the counter reaches a value of one, it will be loaded with an updated value of M. The output clock have a value of 1 when the counter value is larger than M/2, and a value of 0 when counter is smaller than or equal to M/2. Therefore, its frequency will be 1/M of the input frequency and its duty cycle is almost 50%. The output clock of the reference divider will go directly to the reference clock input of the display PLL.

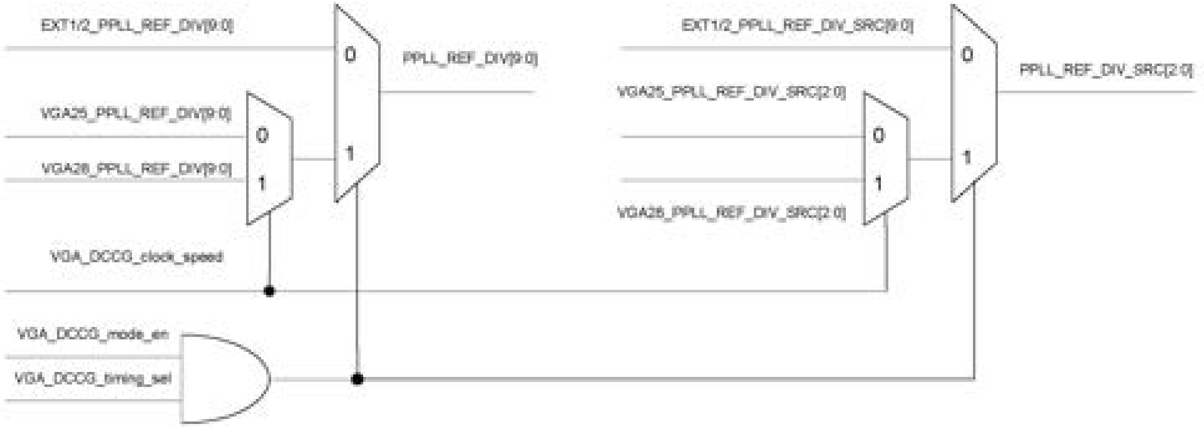


**Figure 5 PLL Reference Divider Source Selection**

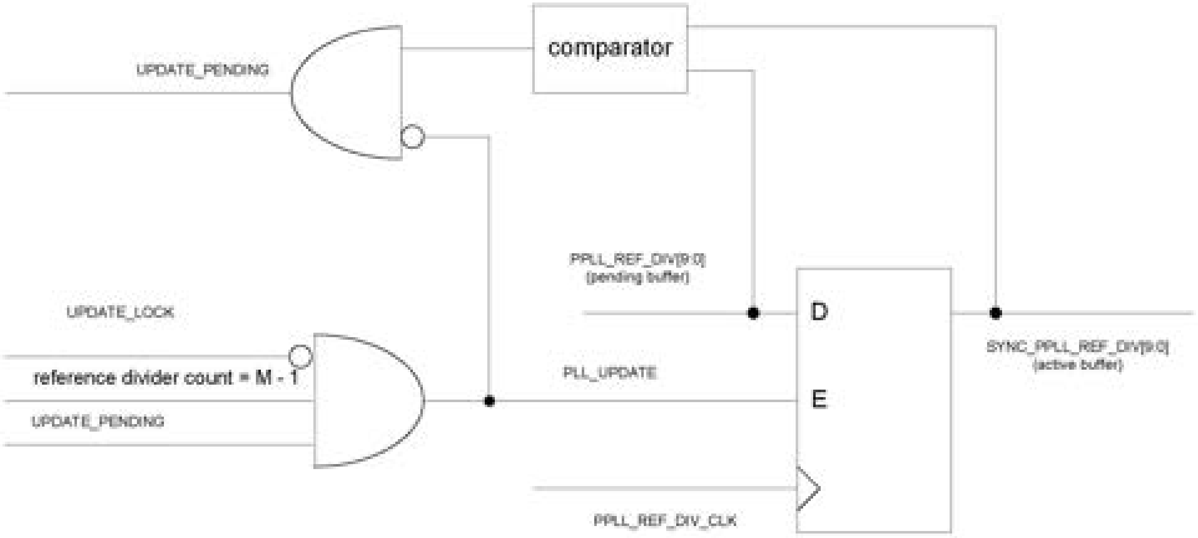
The above Figure describes the logic to choose which one of the four registers is used as the reference divider value and reference divider clock source. If both VGA_DCCG_mode_en and VGA_DCCG_timing_sel are 1, that means VGA timing mode. Reference divider will use either VGA25_xxx or VGA28_xxx, depending on speed of pixel clock in VGA timing mode, determined by VGA_DCCG_clock_speed. If extended timing mode, reference divider for PLL 1 will use EXT1_xxx and reference divider for PLL 2 will use EXT2_xxx.

When the reference divider register is updated, the input to the reference divider not updated right away. The register value is first copied to a pending buffer and stays there. When certain conditions are met, the value in the pending buffer will be copied to the active buffer, which is the input to the reference divider. Double buffering works as follows :

(1) Before any register update is performed, the lock bit associated with the register is set to 1.
(2) Then the register is updated. This will set the update pending bit associated with the register to 1. The new register value will be copied to a pending buffer and stays there.
(3) Set the lock bit to 0. The state machine will then wait for the counter in the reference divider to wrap around from 0 to M, then to M-1. Then the new value in the pending buffer will be copied to the active buffer. Also the updated pending bit will be cleared to 0. Notice that if lock bit is kept to be 1, update of the active buffer will never occur even when the counter reaches M-1.
(4) Set the lock bit back to 1.

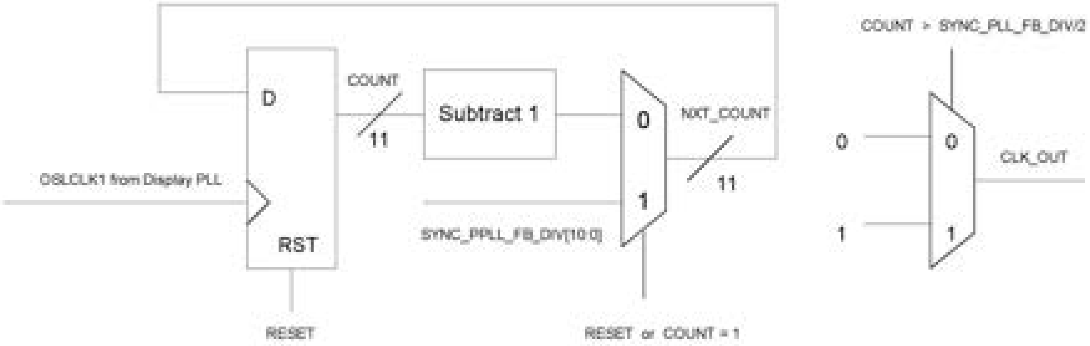
The reason to use double buffering is that we want the values for the reference and feedback dividers to be updated at the same moment. Although the registers for different dividers are written at different times, double buffering can make sure that they arrive to the active buffers of the dividers at the same moment. Double buffering can also be delayed until the next VSYNC. This is enabled by setting register EXT1/2_PPLL_UPDATE_SYNC to 1.

Note that double buffering is not enabled in VGA timing mode.



### 10.2.3 Feedback Divider

The diagram below depicts the feedback divider.



The feedback divider takes the output clock from the display PLL and divides the frequency by N, the output clock of the feedback divider goes to the feedback clock input of the display PLL.

The algorithm used inside the feedback divider is similar to the reference divider. At reset, the counter inside the feedback divider is initially loaded with the divider value (N), from one of the four feedback divider registers, depending on the timing mode and which display PLL. The counter, which runs at the slipable output clock from display PLL, is decremented by 1 after each clock. When the counter reaches 1, it will be loaded with an updated value of N. The output clock will have a value 1 when the counter is larger than N/2, and a value of 0 when counter is smaller than or equal to N/2. The output clock of the feedback divider will have a frequency 1/N of the input clock, and it also have an almost 50% duty cycle.

The feedback divider value can come from one of the four registers. If both VGA_DCCG_mode_en and VGA_DCCG_timing_sel are 1, feedback divider will use either VGA25_xxx or VGA28_xxx, depending on value of VGA_DCCG_clock_speed. Otherwise, feedback divider for PLL 1 will use EXT1_xxx and feedback divider for PLL 2 will use EXT2_xxx. The below Figure describes the muxing logic.

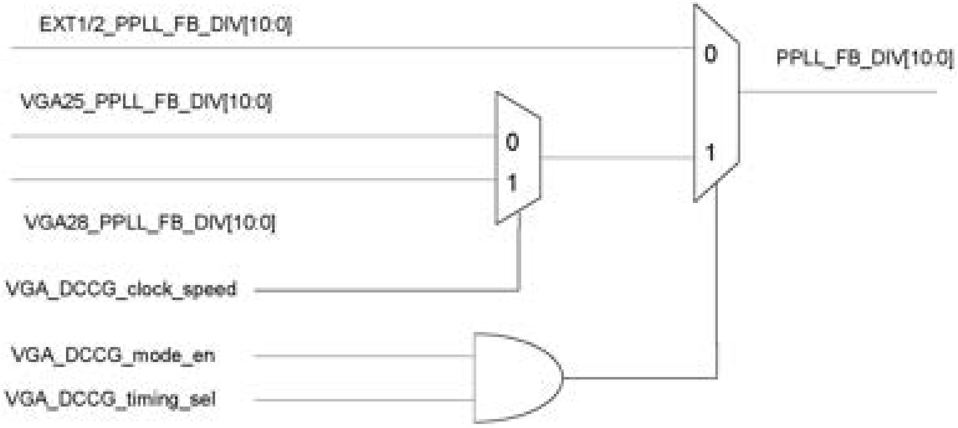


Figure 6 PLL Feedback Divider Source Register Selection

Double buffering is also used to update the active buffers for feedback divider values. The algorithm used for reference and feedback dividers are identical. Values of both dividers are updated at the same time.

### 10.2.4 Slip Requestor for Fractional Feedback Divider

The diagram below depicts the slip requestor for fractional feedback divider.

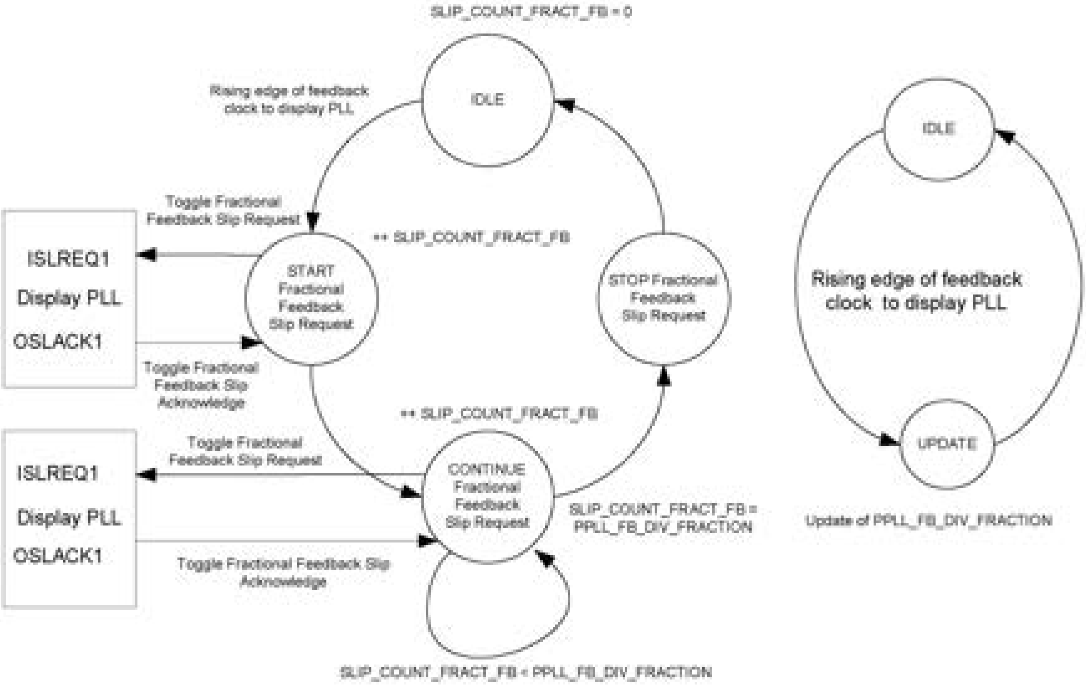


**Figure 7 PLL Feedback Divider Slip Requestor State Diagram**

The counter algorithm used in the feedback divider can only divide the input clock frequency by an integer value. In order to divide the input clock frequency by a fractional value, a technique called slip request is employed. At every output clock from the feedback divider, a certain number of slip request is made to the output clock of the display PLL. Each slip request will make one output clock of the PLL elongated by 1/5 of the period. That is equivalent to adding 1/5 to the feedback divider value. Therefore, the counter inside the feedback divider, together with the slip request will be able to make a fractional feedback divider, with fractional part of the divider value rounded to the nearest 0.2.

Look at the state diagram on the left hand side of the above Figure.

(1) In the IDLE state, the counter inside the feedback divider is initially loaded with the feedback divider value (N). At each input clock of the feedback divider, the counter is decremented by 1. When the counter reaches a value of 1, counter will be wrapped around to N. The transition of a counter value from 1 to N is also the rising edge of the output clock from the feedback divider. At that moment, the state machine will transition to the START state. Also note that slip counter is initially set to 0.

(2) In the START state, the slip request circuit will make a slip request by toggling the level of the request signal. At the same time, the slip counter will increment from initial value of 0 to 1. The display PLL acknowledges the slip request by toggling the level of the acknowledge signal, and at the same time, slip the output clock of the PLL by 1/5 of the period. Then the state machine will transition to the CONTINUE state.

(3) In the CONTINUE state, the slip counter will be checked against the number of slip requests the slip circuit is supposed to make. If it has not finished all the slip requests, it will continue to make them and increment the slip counter by 1 for each request made. Once it has finished make all slip request, and that each request is acknowledged and performed by the display PLL, the state machine will transition to the stop state.

(4) In the STOP state, the slip request circuit will not make any slip request, and the slip counter will be reset to 0. Then the state machine will return to the IDLE state and wait for another rising edge of the output clock from the feedback divider.

The number of slip request to make in each period of the output clock from the feedback divider is stored in the signal PPLL_FB_DIV_FRACTION. The signal gets its value from one of the four sets of registers, depending on the timing mode (VGA or extended) and which display PLL (P1PLL or P2PLL). If both VGA_DCCG_mode_en and VGA_DCCG_timing_sel are 1, fractional feedback divider will use either VGA25_xxx or VGA28_xxx, depending on value of VGA_DCCG_clock_speed. Otherwise, fractional feedback divider for PLL 1 will use EXT1_xxx and fractional feedback divider for PLL 2 will use EXT2_xxx. Figure 4A describes the muxing logic.

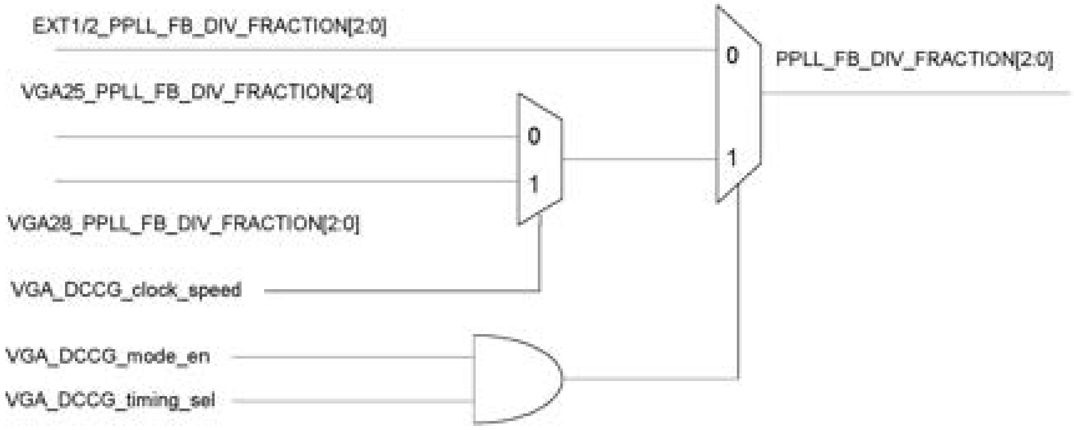


**Figure 8 PLL Fractional Feedback Divider Register Source Selection**

Double buffering is also used to update the active buffers for fractional feedback divider values. The algorithm used for reference and fractional feedback dividers are similar. The main difference is that the number of slips to make per feedback clock is updated at the time when the counter in the feedback divider changes from 1 to N. This will make sure that the slip circuit will perform an updated number of slips at the same time when the counter in the feedback divider is updated with a new value. That will effectively update the integral and fractional part of the divider value at the same time.

### 10.2.5 Post Divider

The diagram below depicts the post divider.

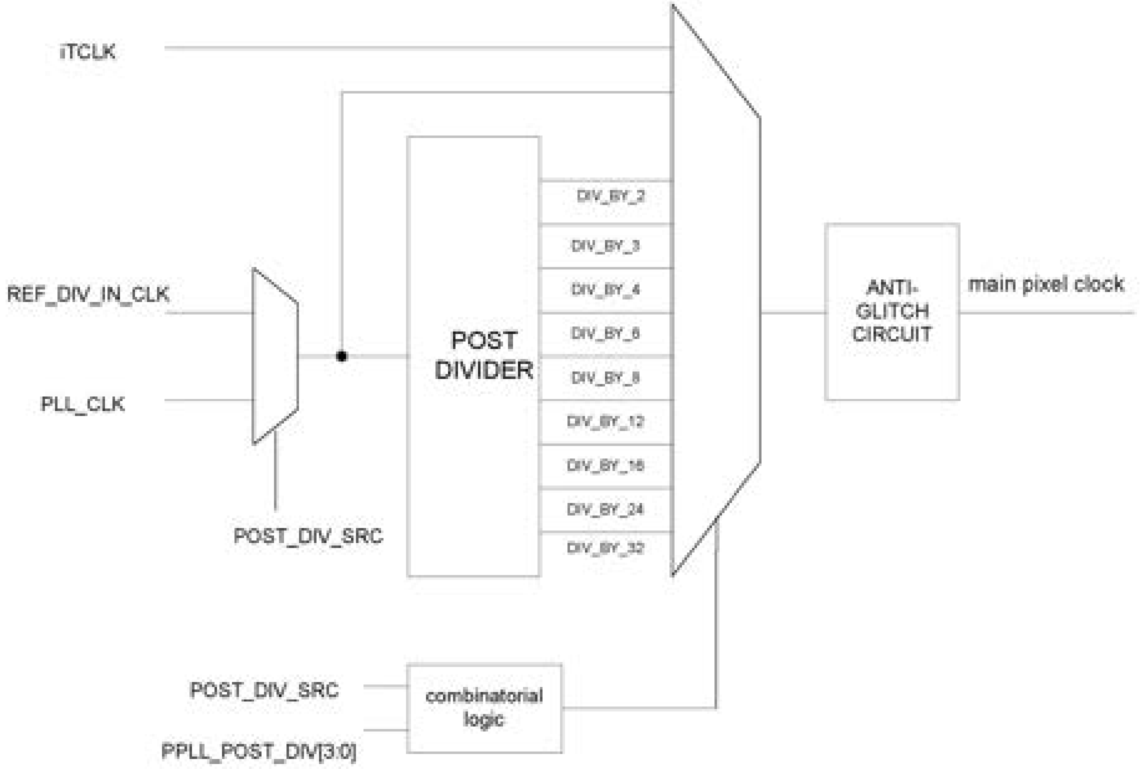


**Figure 9 PLL Post Divider and the Mux**

The following are the clock sources of the display PLL post divider :

(1) External clock, input of reference divider
(2) PLL clock

The signal POST_DIV_SRC determines the clock source to the input of the post divider. POST_DIV_SRC gets its value from one of the four sets of registers, depending on the timing mode (VGA or extended) and which display PLL (P1PLL or P2PLL). The signal PPLL_POST_DIV will determine the value for post divider. PPLL_POST_DIV also gets its value from one of the four sets of registers, depending on timing mode and which PLL. If both VGA_DCCG_mode and VGA_DCCG_timing_sel are 1, post divider will use either VGA25_xxx or VGA28_xxx for its clock source and divider value, depending on value of VGA_DCCG_clock_speed. Otherwise, post divider for PLL 1 will use EXT1_xxx and post divider for PLL2 will use EXT2_xxx.

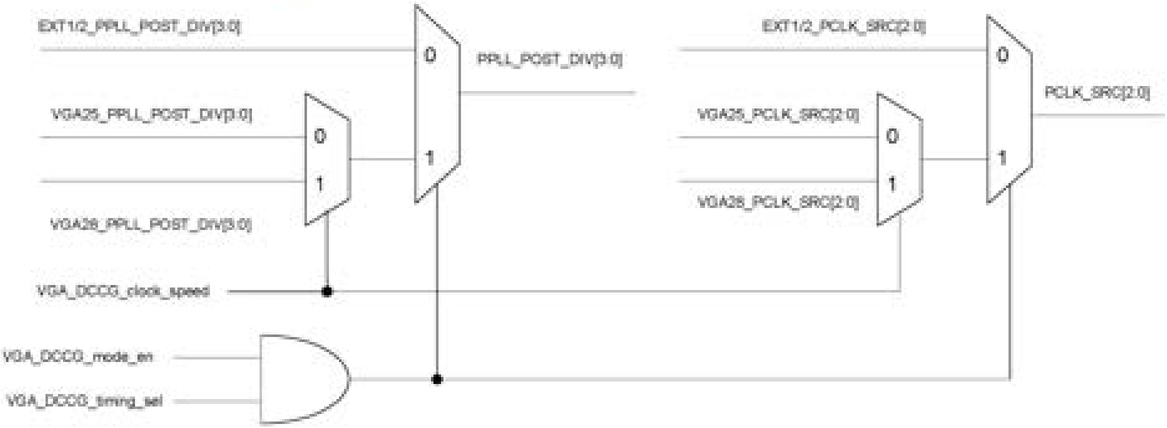
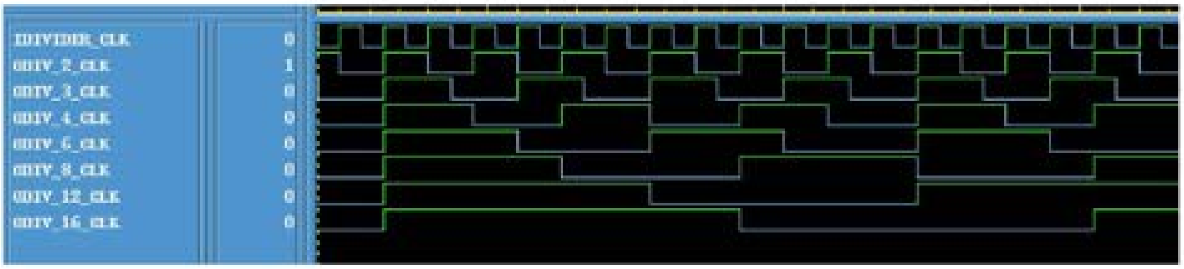


Figure 10 PLL Post Divider Register Source Selection

The post divider uses the input clock to generate a limited number of clock outputs whose periods are integral multiples of the input clock. The following multiples are generated : 2, 3, 4, 6, 8, 12, 16, 24, 32. Instead of using the counter algorithm employed by reference and feedback dividers, the following algorithm is used to generate these divided clocks :

```
If (reset) then
   DIV_2_CLK = 0
   DIV_3A_CLK = 0
   DIV_3B_CLK = 0
   DIV_4_CLK = 0
   DIV_6_CLK = 0
   DIV_8_CLK = 0
   DIV_12_CLK = 0
   DIV_16_CLK = 0
   DIV_24_CLK = 0
   DIV_32_CLK = 0
Else if (At rising edge of input clock) then
   DIV_2_CLK   = not (DIV_2_CLK and (DIV_3A_CLK or DIV_3B_CLK or DIV_6_CLK) )
   DIV_3A_CLK  = not DIV_3B_CLK and not DIV_3A_CLK
   DIV_3B_CLK  = DIV_3A_CLK
   DIV_4_CLK   = not (DIV_4_CLK xor DIV_2_CLK)
   DIV_6_CLK   = not (DIV_6_CLK xor (DIV_3A_CLK or DIV_3B_CLK) )
   DIV_8_CLK   = not (DIV_8_CLK xor (DIV_2_CLK or DIV_4_CLK) )
   DIV_12_CLK  = not (DIV_12_CLK xor (DIV_3A_CLK or DIV_3B_CLK or DIV_6_CLK) )
   DIV_16_CLK  = not (DIV_16_CLK xor (DIV_2_CLK or DIV_4_CLK or DIV_8_CLK) )
   DIV_24_CLK  = not (DIV_24_CLK xor (DIV_3A_CLK or DIV_3B_CLK or DIV_6_CLK or DIV_12_CLK) )
   DIV_32_CLK  = not (DIV_32_CLK xor (DIV_2_CLK or DIV_4_CLK or DIV_8_CLK or DIV_16_CLK) )
End if

(At falling edge of input clock)
   DIV_3D_CLK  = DIV_3A_CLK
   DIV_3E_CLK  = DIV_3D_CLK

DIV_3_CLK  = DIV_3A_CLK or DIV_3D_CLK (just combinatorial)
```

The waveform of the divided clocks looks like the diagram below (DIV_24_CLK and DIV_32_CLK does not exist in R300) :



Look at figure 5 again. The divided clocks, together with all the input clock sources of the post divider will go into a larger mux. The larger mux uses signals PCLK_SRC and PPLL_POST_DIV to choose one of the input clocks to be the main pixel clock. PPLL_POST_DIV determines the divider value of the post divider. The following post divider values are available : 1, 2, 3, 4, 6, 8, 12, 16, 24 and 32. Notice that both PCLK_SRC and PPLL_POST_DIV are not double buffered.

When either the clock source to the post divider or the post divider value is changed, it can cause instability and glitches to the main pixel clock. Therefore, an anti-glitch circuit is needed.

Figure 6 describes how the anti-glitch functions.

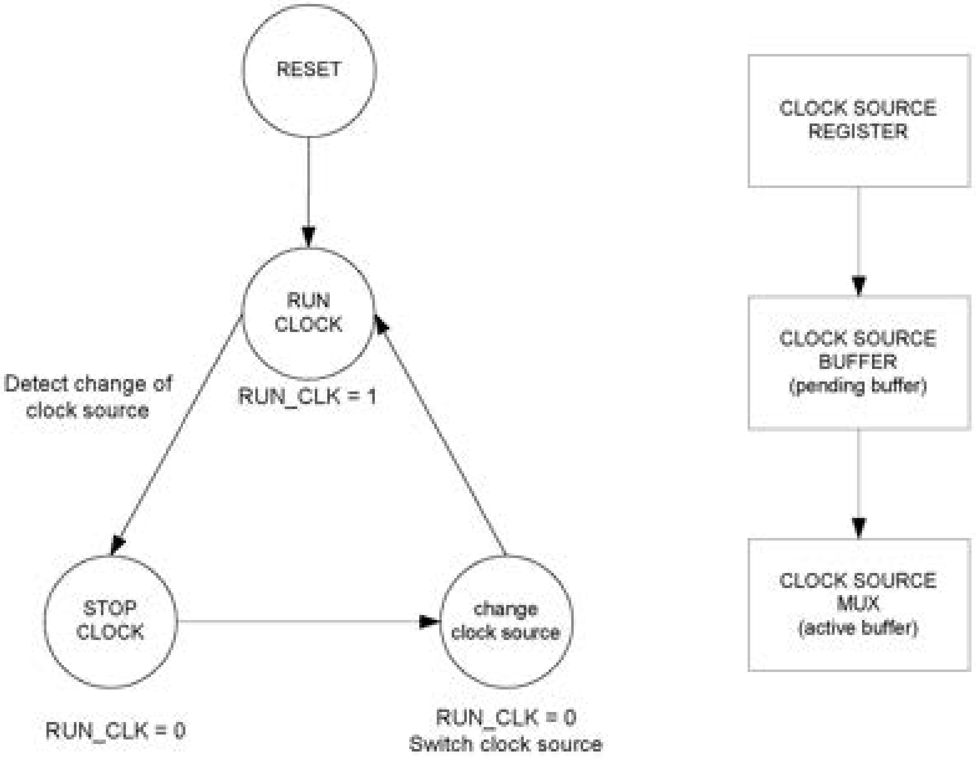


**Figure 11 Anti-glitch Circuit State Diagram**

In figure 5, all the divided clocks and the input clock sources for the post divider will be the source of the bigger mux. Whenever the clock sources to the bigger mux changes, caused by either change in PCLK_SRC or PPLL_POST_DIV, there will be a change of clock source to the bigger mux.

In figure 6, the anti-glitch circuit state machine is initially transitioned from RESET state to "RUN CLOCK" state, in which the main pixel clock is running smoothly. Whenever the clock source register or the post divider value register changes, the value in the pending buffer will be different from the register values. The state machine will detect this change of clock source and transition to the "STOP CLOCK" state. In the "STOP CLOCK" state, the main pixel clock is temporarily gated off. The state machine then transitions to the "change clock source" state. In this state, the register changes will then be transferred to the pending buffer and finally to the active buffer. The change in the active buffer will switch the clock source of the main pixel clock. Notice that the main pixel clock is still gated off in this state. After the switch of the main pixel clock source, the state machine will transition back to the "RUN CLOCK" state. The main pixel clock resumes running again. So the main function of the anti-glitch circuit is to stop the main pixel clock, switch the clock source and resumes the clock.

### 10.2.6 Slip Requestor for Post Divider

Another feature related to the display PLL is that the output clock can be slipped a number of times for each line of the display. This is accomplished by adding a slip request circuit to the display PLL output clock. The slip requestor for post divider is similar to the slip requestor for the fractional feedback divider. They are different in the conditions when the slip request will be made. Notice that this slip requestor is not enabled in VGA timing mode.

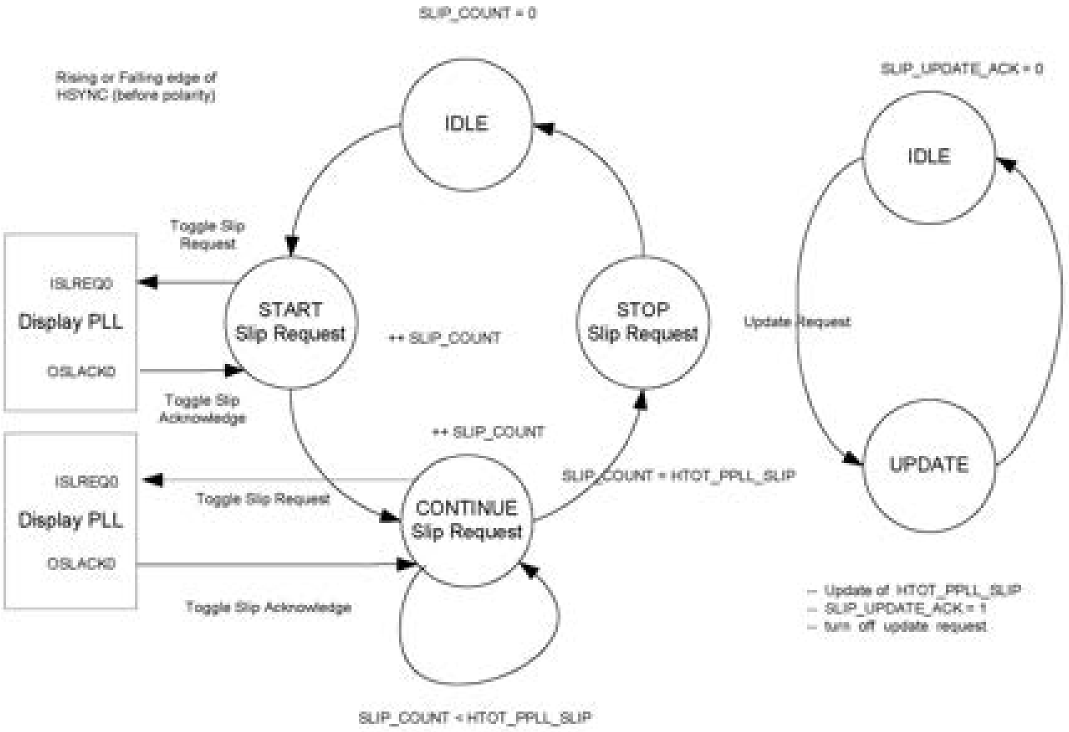


Figure 12 PLL Post Divider Slip Requestor

The state machine of the slip requestor for post divider works as follows :

(1) In the IDLE state, the slipable output clock from the display PLL is running at a regular frequency, i.e. period of each clock cycle is the same. At the HSYNC of each display line, the state machine will transition to the START state. The register EXT1/2_HTOT_CNTL_EDGE determines whether the transition of state happen in the positive or negative edge of HSYNC.

(2) In the START state, the slip request circuit will make a slip request by toggling the level of the request signal. At the same time, the slip counter will increment from initial value of zero to one. The display PLL acknowledges the slip request by toggling the level of the acknowledge signal, and at the same time, slip the output clock of the PLL by 1/5 of the period. Then the state machine will transition to the CONTINUE state.

(3) In the CONTINUE state, the slip counter will be checked against the number of slip requests the slip circuit is supposed to make. If it has not finished all the slip requests, it will continue to make them and increment the slip counter by 1 for each request made. Once it has finished make all slip request, and that each request is acknowledged and performed by the display PLL, the state machine will transition to the stop state.

(4) In the STOP state, the slip request circuit will not make any slip request, and the slip counter will be reset to 0. Then the state machine will return to the IDLE state and wait for the HSYNC in the next line.

The number of slip request to make in each horizontal line of the display is stored in the active buffer HTOT_PPLL_SLIP. When the slip count register is updated, it will be copied to the pending buffer and stays there. The new value in the pending buffer will not be copied to the active buffer until there is a trigger event. The trigger event is the write trigger signal generated by writing to the register EXT1/2_HTOT_CNTL_W. This trigger event will generate an update request to update the active buffer that holds the number of slips to be performed per horizontal line. After the new value is copied from the pending buffer to the active buffer, an acknowledge signal is generated to turn off the request. This is illustrated in the state diagram on the right hand side of figure 7.

The post divider slip circuit can be disabled by setting the register EXT1/2_HTOT_SLIP to 0.

### 10.2.7 Functional Clock Gating for Pixel Clock

The two main pixel clocks, one for each CRTC is functionally gated to provide pixel clocks for each display device. The clock branches include PCLK_CRTC1, PCLK_CRTC2, PCLK_DACA, PCLK_DACB, PCLK_TMDSA, PCLK_HDCP, PCLK_DVOA. Each clock branch have different gating conditions, but they are usually the enables for the device associated with the clock branch. The functional gating of each branch can be disabled by individual register bit.

The below diagram depicts the muxing and functional clock gating of each branch of the pixel clock.

PCLK_DIAGRAM

Main pixel clock 1 and main pixel clock 2, and there inversions, serve as inputs to the PCLK_CRTC1 and PCLK_CRTC2, as well as pixel clock branches for various devices. They will first go through a mux and an anti-glitch circuit . Then output of the anti-glitch circuit will be functionally gated by individual conditions of each display device to become the pixel clock for that device. The table below describes the clock source muxing and functional gating of each pixel clock branch.

| CLOCK | Clock muxing register | Clock source | Clock gating enable | Feature enable | Clock Status |
|---|---|---|---|---|---|
| PCLK_CRTC1 | CRTC1_CLK_SRC = 0 | Pixel clock 1 | PCLK_CRTC1_GATE_DIS = 1 | | Always run |
| | CRTC1_CLK_SRC = 1 | Pixel clock 2 | PCLK_CRTC1_GATE_DIS = 0 | CRTC1_EN = 0 | Stop |
| | | | | CRTC1_EN = 1 | Always run |
| PCLK_CRTC2 | CRTC2_CLK_SRC = 0 | Pixel clock 1 | PCLK_CRTC2_GATE_DIS = 1 | | Always run |
| | CRTC2_CLK_SRC = 1 | Pixel clock 2 | PCLK_CRTC2_GATE_DIS = 0 | CRTC2_EN = 0 | Stop |
| | | | | CRTC2_EN = 1 | Always run |
| PCLK_DACA | DACA_CLK_SRC = 0 | Pixel clock 1 | PCLK_DACA_GATE_DIS = 1 | | Always run |
| | DACA_CLK_SRC = 1 | Pixel clock 2 | PCLK_DACA_GATE_DIS = 0 | DACA_EN = 0 | Stop |
| | | | | DACA_EN = 1 | Always run |
| PCLK_DACB | DACB_CLK_SRC = 0 | Pixel clock 1 | PCLK_DACB_GATE_DIS = 1 | | Always run |
| | DACB_CLK_SRC = 1 | Pixel clock 2 | PCLK_DACB_GATE_DIS = 0 | DACB_EN = 0 | Stop |
| | | | | DACB_EN = 1 | Always run |
| PCLK_TMDSA | TMDSA_CLK_SRC = 0 | Pixel clock 1 | PCLK_TMDSA_GATE_DIS = 1 | | Always run |
| | TMDSA_CLK_SRC = 1 | Pixel clock 2 | PCLK_TMDSA_GATE_DIS = 0 | TMDSA_EN = 0 | Stop |
| | | | | TMDSA_EN = 1 | Always run |
| PCLK_HDCP | HDCP_CLK_SRC = 0 | Pixel clock 1 | PCLK_HDCP_GATE_DIS = 1 | | Always run |

| | | | | | |
|---|---|---|---|---|---|
| | HDCP_CLK_SRC = 1 | Pixel clock 2 | PCLK_HDCP_GATE_DIS = 0 | HDCP_EN = 0 | Stop |
| | | | | HDCP_EN = 1 | Always run |
| PCLK_DVOA | DVOA_CLK_SRC = 0 | Pixel clock 1 | PCLK_DVOA_GATE_DIS = 1 | | Always run |
| | DVOA_CLK_SRC = 1 | Pixel clock 2 | PCLK_DVOA_GATE_DIS = 0 | DVOA_EN = 0 | Stop |
| | | | | DVOA_EN = 1 | Always run |

## 10.3 TMDS Links

The feature of R500 includes a second internal TMDS link in order to support display resolution larger than 1600x1200 at 60Hz refresh rate. These dual links operates at two different modes, which are dual-link mode and two-single-link mode.

In the dual link mode, the display clock is running at 330 MHz. The two separate fifo store the pixel data for even pixels and odd pixels. Although one TMDS link can only support 165 MHz pixel rate, with the second link, a total bandwidth of 330Mpixel per second can be supported.

In the two single link mode, the two TMDS links operates separately at the frequency of 165 MHz. The clock source in these two single links can be from the same crtc1 or crtc2 clock source. They also can be from the different crtc1 and crtc2 clock source depending on the mode setting.

The DCCG block in DO is responsible for generating clocks for TMDSA and TMDSB through the clock tree synthesis. The frequency of these two clocks is 330 MHz in the dual-link mode and 165 MHz in the two-single-link mode.

The TMDSCLKBLK generates the clean version of the clocks. In the dual link mode, the data synchronizer needs “divide-by-two” version of clock in order to read the data from the separate data fifo of even pixels and odd pixels. If “divide-by-two” is done in TMDSCLKBLK, the frequency of TMDSA_DIRECT and TMDSB_DIRECT is 165 MHz. Otherwise the frequency is 330 MHz. This means that this “divide-by-two” will be done the TMDS macro. In the case of two-single-links, the frequency is 165 MHz.

The clock source selection depends on the mode control and the source control.

```
// if (dual link tmds)
     if (tmds clock from crtc1)
             pclk_tmdsa = pclk_crtc1
             pclk_tmdsb = pclk_crtc1
     else
             pclk_tmdsa = pclk_crtc2
             pclk_tmdsb = pclk_crtc2
  else if (single link mode)
      if (tmdsa clock from crtc1)
             pclk_tmdsa = pclk_crtc1
      else
             pclk_tmdsa= pclk_crtc2
      if(tmdsb clock from crtc1)
             pclk_tmdsb = pclk_crtc1
      else
             pclk_tmdsb = pclk_crtc2
```

The detailed clock generation diagram shows in the following link.

DUAL_LINK_TMDS_CLOCK

## 10.4 TV_VCLK Generation

A method of Discrete Time Oscillator (DTO) is used to generate TV_VCLK. The algorithm used by DTO is basically a scaling algorithm.

Let's take NTSC as an example:

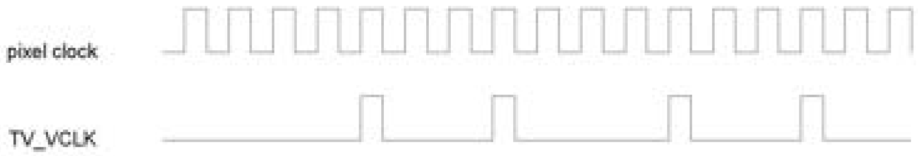
If the scaling ratio for NTSC = 35/121 = 0.289256

The accumulator is reset at 0. After that at each cycle of pixel clock, it is incremented by the scaling ratio. At each pixel clock cycle where the integer part of the accumulator is incremented, the TV_VCLK will generate a pixel clock pulse. Otherwise, TV_VCLK will be zero.

DTO also outputs the phase coefficient for the upsampler at each pixel clock. The fractional part of the accumulator is used as the phase coefficient.

| Pixel clock cycle | Accumulator value | TV_VCLK |
|---|---|---|
| 0 | 0 | zero |
| 1 | 0.289256 | zero |
| 2 | 0.578512 | zero |
| 3 | 0.867768 | zero |
| 4 | 1.157024 | pixel clock pulse |
| 5 | 1.446280 | zero |
| 6 | 1.735536 | zero |
| 7 | 2.024792 | pixel clock pulse |
| 8 | 2.314048 | zero |
| 9 | 2.603304 | zero |
| 10 | 2.892560 | zero |
| 11 | 3.181816 | pixel clock pulse |
| 12 | 3.471072 | zero |
| 13 | 3.760328 | zero |
| 14 | 4.049584 | pixel clock pulse |

The pixel clock and TV_VCLK will look like the below diagram.



In the hardware implementation, we want to calculation in fix point instead of floating point. Therefore, we need to approximate the scaling ratio with an unsigned integer. In R500, a 32-bit register field is defined to represent the scaling ratio.

DTO_VCLK_INC[31:0]

The following formula is used to calculate the register value based on scaling ratio:

DTO_VCLK_INC = integer part(scaling_ratio x $2^{31}$)

The size of the accumulator is defined as 32 bits for R500.

tvclk_accumulator[31:0]

The accumulator is initially reset to 0. At each pixel clock cycle, tvclk_accumulator is incremented by DTO_VCLK_INC. At the pixel clock cycle where tvclk_accumulator[31] is 1, the accumulator has overflowed. DTO will output a pixel clock pulse at this cycle. Also at each pixel clock cycle, the 8-bit msb (bit 30 to 23) of the accumulator will be used as the phase coefficient for the upsampler.

After the accumulator has been incremented with the scaling ratio for N times, where N is the denominator of the scaling ratio, its fractional part should be equal to zero, with the integer part equal to the numerator of the scaling ratio. However, since we round off the fractional part when we calculate DTO_VCLK_INC, this will not happen. To correct this, we need to add an offset to the accumulator every N pixel clock cycles, where N is the denominator of the scaling ratio.

In order to keep track of the number of cycles the accumulator has been incremented, we define a 12-bit register to store the denominator of the scaling ratio.

DTO_VCLK_DENOMIN[11:0] = Denominator of the scaling ratio – 1

The counter tvclk_denominator_count is initialized to 0. At each pixel clock cycle, it is incremented by one. When counter reaches DTO_VCLK_DENOMIN, it will be reset to 0. At this same cycle, the accumulator is incremented by (DTO_VCLK_INC + offset) to compensate for the round off of DTO_VCLK_DENOMIN.

A 32-bit register is defined to represent the value added to the accumulator every N cycles, where N is the denominator of the scaling ratio.

DTO_VCLK_INC_CORR[31:0]

DTO_VCLK_INC_CORR = Numerator x $2^{31}$ – DTO_VCLK_DENOMIN x DTO_VCLK_INC

The algorithm to generate TV_VCLK_EN is described as below.

```
//algorithm
At every pixel clock cycle
If(reset)
        tvclk_accumulator <= 0
        tvclk_denominator_count <= 0
        upsampler_phase <= 0

else
        if(tvclk_denominator_count == DTO_VCLK_DENOMINATOR)
                tvclk_accumulator <= tvclk_accumulator[30:0] + DTO_VCLK_INC_CORR
                tvclk_denominator_count <= 0
        else
                tvclk_accumulator <= tvclk_accumulator[30:0] + DTO_VCLK_INC
                tvclk_denominator_count <= tvclk_denominator_count + 1

if(reset)
        upsampler_phase = 0
        TV_VCLK_EN <= 0
Else
        Upsampler_phase <= tvclk_accumulator[30:23]
        TV_VCLK_EN = tvclk_accumulator[31]
```

## 10.5 One-shot Dynamic Stopping/Running Clock for Debug Purpose

One-shot dynamic stopping/running clock has been implemented as a new feature in R400 and R500 compared to previous projects. This feature makes clocks of a certain braches stopped or run dynamically in order to analyze the status of the chip through the debug bus. After a particular clock branch is stopped, it can be advanced one or more clock cycles through register-write and will stop again when it is finished.

There are two modes of one-shot dynamic stopping/running clock, manual mode and trigger mode. The manual mode is under register control, which register-write makes clock stopping or running at a particular moment. In the trigger mode, the clock is stopped when a trigger event on the test debug data bus is detected as the matching pattern that is expected.

The clocks branches are controlled by one-shot debug mode are listed bellow.

SCG in DC branches

- System clock branch used by DCP (sclk_g_dcp)
- System clock branch used by VGA renderer (sclk_g_vga)

DCCG in DO branches

- System clock branches used by SCL1 and SCL2 (sclk_g_scl1, sclk_g_scl2)
- Pixel clock branch used by the primary main pixel clock source (main_pix1clk)
- Pixel clock branch used by the secondary main pixel clock source (main_pix2clk)

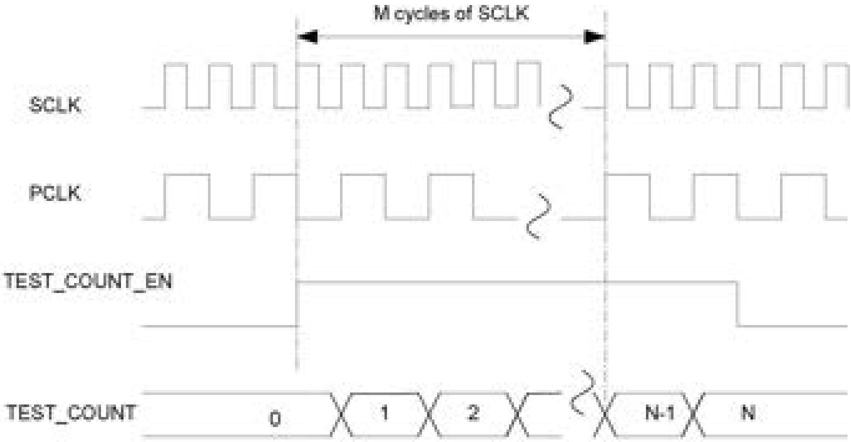
The registers are listed bellow.

```
SCG:
SCG_ONE_SHOT_CLOCKING_CNTL
{
  SCG_ONE_SHOT_CLOCKING_MODE [1:0]
  SCG_TEST_DEBUG_DATA_TRIGGER_ONE_SHOT_EN
};

SCG_ONE_SHOT_STOP_CLOCKS_CNTL
{
  SCG_ONE_SHOT_STOP_DCP_SCLK
  SCG_ONE_SHOT_STOP_VGA_SCLK
};

SCG_ONE_SHOT_RUN_CLOCKS_CNTL
{
  SCG_ONE_SHOT_RUN_DCP_SCLK
  SCG_DCP_SCLK_OVERALL_STATUS
  SCG_ONE_SHOT_RUN_VGA_SCLK
  SCG_VGA_SCLK_OVERALL_STATUS
};

SCG_ONE_SHOT_RUN_CLOCKS_COUNT
{
 SCG_ONE_SHOT_RUN_DCP_SCLK_COUNT [7:0]
 SCG_ONE_SHOT_RUN_VGA_SCLK_COUNT [7:0]
};

DCCG_SCL_ONE_SHOT_STOP_CLOCKS_CNTL
{
```

```
 DCCG_ONE_SHOT_STOP_SCL_SCLK
};

DCCG_SCL_ONE_SHOT_RUN_CLOCKS_CNTL
{
 DCCG_ONE_SHOT_RUN_SCL_SCLK
 DCCG_SCL_SCLK_OVERALL_STATUS
};

DCCG_SCL_ONE_SHOT_RUN_SCL_SCLK_COUNT
{
 DCCG_ONE_SHOT_RUN_SCL_SCLK_COUNT
};

ONE_SHOT_STOP_CLOCKS_CNTL_MIRROR
{
 SCG_ONE_SHOT_STOP_DCP_SCLK
 DCCG_ONE_SHOT_STOP_SCL_SCLK
};

ONE_SHOT_RUN_CLOCKS_CNTL_MIRROR
{
 SCG_ONE_SHOT_RUN_DCP_SCLK
 DCCG_ONE_SHOT_RUN_SCL_SCLK
};


DCCG:

DCCG_ONE_SHOT_CLOCKING_CNTL
{
  DCCG_ONE_SHOT_CLOCKING_MODE [1:0]
  DCCG_TEST_DEBUG_DATA_TRIGGER_ONE_SHOT_EN
};

DCCG_ONE_SHOT_STOP_CLOCKS_CNTL
{
  DCCG_ONE_SHOT_STOP_PIX1CLK
  DCCG_ONE_SHOT_STOP_PIX2CLK
};

DCCG_ONE_SHOT_RUN_CLOCKS_CNTL
{
  DCCG_ONE_SHOT_RUN_PIX1CLK_
  DCCG_PIX1CLK_OVERALL_STATUS
  DCCG_ONE_SHOT_RUN_PIX2CLK
  DCCG_PIX2CLK_OVERALL_STATUS
};

DCCG_ONE_SHOT_RUN_CLOCKS_COUNT
{
   DCCG_ONE_SHOT_RUN_PIX1CLK_COUNT [7:0]
 DCCG_ONE_SHOT_RUN_PIX2CLK_COUNT [7:0]
};
```

## 10.6 Read Back Test Counter

The test counter is implemented to calculate pixel clock frequency. This pixel clock can be selected from one of the pclk_main1 and pclk_main2. The method is using SCLK as a reference. If the test counter enable is set, the test counter will increment by 1. This test counter enable bit is programmed through RBBMIF. It can be enabled a number of clock cycles of SCLK and then disabled. After that the RBBM will read back the counter number to calculate the frequency for pixel clock.



The register field is defined as below.

TEST_COUNT_MUX_CLK (select pclk_main 1 or pclk_main 2)
TEST_COUNT_EN
RESET_TEST_COUNT
TEST_COUNT[23:0] (read only)

If the TEST_COUNT_EN is set for M cycles of SCLK, and the TEST_COUNT value is N cycles of pixel clock. The pixel clock frequency is calculated as below formula:

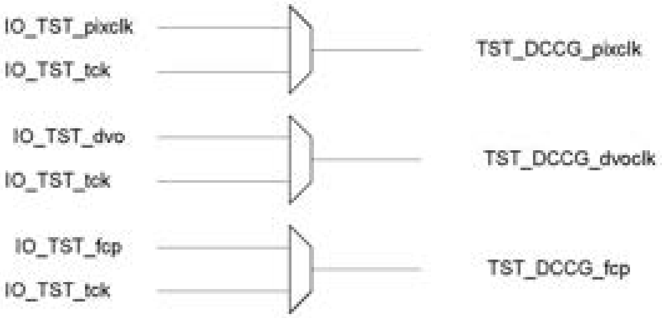
PCLK_frequency = N * SCLK_frequency/M

## 10.7 Clock TST Control

The TST control for SCLK domain is included in CG block. The PCLK domain TST control is done inside DCCG. There are four test clock sources:

TST_DCCG_test_pixclk
TST_DCCG_test_dvoclk
TST_DCCG_test_fcp
TST_DCCG_test_tck

In the scan debug mode, all these TST sources come from IO_TST_TCK. However in the functional test mode, they are from the different IO ports.



These TST clock sources are used for the different PCLK branches as below description.

```
If(TST_DCCG_test_sel[1])
      PIX1CLK = TST_DCCG_test_pixclk
      PIX2CLK = TST_DCCG_test_pixclk
      PCLK_DVOCLK_C/D = TST_DCCG_test_dvoclk
      PCLK_FCP = TST_DCCG_test_fcp
      PCLK_tv = TST_DCCG_test_pixclk
      Tvclk = TST_DCCG_test_tck
end
```

# Revision Changes

This section is optional for changes to the document before the first official release to other groups (rev 1.0). After that point, all changes must be briefly detailed in this section.

**Rev 0.1 Jimmy Lau**
Date: June 4, 2002
Initial revision.
**Rev 0.2 Jimmy Lau**
Date: June 4, 2002
Fix the problem that top level diagram cannot be accessed.
Add sections 3.1, 3.2 and 3.3
**Rev 0.3 Jimmy Lau**
Date: June 9, 2002
Major update after specs review.
**Rev 1.0 Jie Zhou**
Date: April 4, 2003
Major update for R500. Add TMDS Link, TVCLK generation, one-shot mode for debug purpose, Read back test counter, TST mode.
**Rev 1.1 Jie Zhou**
Date: Oct 8, 2003
Add DTO, test counter, fix Hyperlink

|  | ORIGINATE DATE | EDIT DATE | DOCUMENT-REV. NUM. | PAGE |
|---|---|---|---|---|
|  | 4 November, 20023~~1~~ | [date \@ "d MMMM, | GEN-CXXXXX-REVA | 1 of 25 |

**Author:** Larry Seiler

**Issue To:** **Copy No:**

# R400 Memory Controller Architectural Specification

## Version 0.8~~7~~

**Overview:** This is an architectural specification for the R400 Memory Controller block (MC). It provides an overview of the required capabilities and expected uses of the block. It also describes the block interfaces, internal sub-blocks, and provides internal state diagrams.

AUTOMATICALLY UPDATED FIELDS:
**Document Location :**
C:\depot\r400\doc_lib\design\blocks\mc\R400_MemCtl.doc~~D:\Perforce\r400\arch\d~~

| APPROVALS | |
|---|---|
| Name/Dept | Signature/Date |
|  |  |
|  |  |
|  |  |

Remarks:

THIS DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

## Table Of Contents

## Table Of Figures

## Table Of Tables

## Revision History:

| | |
|---|---|
| **Rev 0.1 (Larry Seiler)**<br>Date: March 6, 2001 | First draft. |
| **Rev 0.2 (Larry Seiler)**<br>Date: April 13, 2001 | Complete rewrite. Divides the MC into three sections: routing engine, ordering engine, and protocol engine, with updated interfaces. Does not describe how the ordering engine selects requests. |
| **Rev 0.3 (Larry Seiler)**<br>Date: May 9, 2001 | Complete update of the Ordering Engine, details of Protocol Engine and External Interfaces, updates to Routing Engine and lesser changes. |
| **Rev 0.4 (Larry Seiler)**<br>Date: June 6, 2001 | Added internal interfaces, modified the MCO texture queue, added ordering examples and other updates. |
| **Rev 0.5 (Larry Seiler)**<br>**Date: August 9, 2001** | Eliminated AutoTag & changed how tag is returned, changed many bus names, added XY address bits to RB address request, many lesser changes. (N.B.: Rev 0.5a just fixes typos in the interface tables) |
| **Rev 0.6 (Ken Correll)**<br>Date: Dec 13, 2001 | Updated interface tables. |
| **Rev 0.7 (Bob Bloemer and Bei Wang)**<br>**Date: Jan. 14, 2002** | Extensive changes. |
| **Rev 0.8 (Bob Bloemer and Bei Wang)** | Updated client interface block diagram and interface specs. |

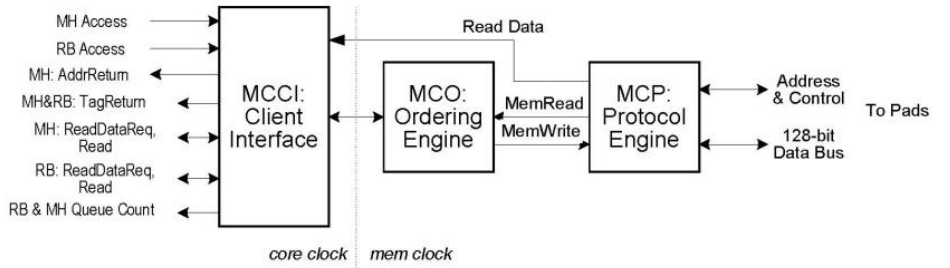
# 1. Overview

The Memory Controller (MC) reads and writes the local memory and converts between the memory clock domain and the core clock domain. There is one MC per render pipeline. The R400 and R450 have four render pipelines and hence four memory controllers. The RV400 has two renderpipelines and hence two memory controllers.

Each MC accepts read and write requests from the global Memory Hub (MH) and from the Render Backend (RB) in its render pipeline. The MC also transfers data between the MH and its RB. The memory clock can be slower than the core clock or can be up to twice as fast as the core clock, so the MC converts between the two clock rates.

## 1.1 Top Level Block Diagram

The MC can be thought of as containing three sub-blocks. The Client Interface (MCCI) puts requests into the proper queues and routes read and write data. The Ordering Engine (MCO) handles the memory clock end of the queues and schedules the memory accesses. The Protocol Engine (MCP) buffers read/write commands, read data, and write data and performs the actual memory accesses. The figure below shows the relationship of these three sub-blocks and the names of the busses that connect them to each other and to other blocks.



**Formatted:** Font: Times New Roman, Bold, Complex Script Font: Bold

**Figure 1: MC Top Level Block Diagram**

The clock crossing from core clock to mem clock occurs in the queues and dual ported memories that connect the client interface to the rest of the MC. With the heads of the queues in the core clock domain the clients can access the queues without waiting to cross a clock boundary. By having the tails of the queues in the memory clock domain the ordering engine sees all the queues and can select the next access knowing the state of each DRAM bank.

The pins transfer 64 bits to or from the DRAMs twice per clock. In the pads this is converted to 128 bits once per clock. Read and write data is stored in 128-bit wide memories; transfers to or from the pads can occur at a rate of 128 bits per mem clock. The design is optimized for a memory clock to core clock ratio of 1 or somewhat higher. However the design will function correctly at any ratio.

## 1.2 Memory Configurations

The MC supports regular DDRAM2-style parts and Elpida DDRAM, both of which have four banks and 8 or 9 column address bits. The MC does not support non-DDR SDRAM or SGRAM. ~~The MC supports only a 64-bit wide memory data bus. We will support up to one rank of 64M x 16b (1Gb) DRAMS, or 512 MB per MC. This support applies to the logic in the MH and MC. This will increase the width of address field on MH_MC_access bus to 23 bits, plus one bit for the subset field. Note that 1Gb DRAMs require one additional address pin per MC over the original design of 512MB for 4 MCs; whether that is included is a separate issue. This support would allow a total of 2 GB on four MCs. This is not necessarily a supported configuration, due to issues outside of the MC.~~

The maximum memory size is 512 MB total, 256 MB per MC.  11-14 rows and 8-10 column bits are supported.  Two ranks are allowed, but in that configuration the number of row bits is limited to 13.  The latest DDR II spec calls for 8 banks in the 1Gb and larger parts; this is not supported in the MC.

| Class | I/O V. | On-Chip Term. | Data Inv. | Size | Speed | Width | Banks | Rows | Cols | Supported In R400? |
|---|---|---|---|---|---|---|---|---|---|---|
| GDDR III | 1.8v. | High | ✓ | 256Mb | 600 MHz. | X 32 | 4 | 12 | 9 | ✓ |
| GDDR II | 1.8v. | Mid | | 128Mb | 500 MHz. | X 32 | 4 | 12 | 8 | ✓ |
| DDR II | 1.8v. | ? | | 256Mb | 333 MHz. | X 16 | 4 | 13 | 9 | ✓ |
| | 1.8v. | ? | | 512Mb | 333 MHz. | X 16 | 4 | 13 | 10 | ✓ |
| Elpida | 1.8v. | Mid | ✓ | 64Mb | 300 MHz. | X 32 | 4 | 11 | 8 | ✓ |
| Elpida | 1.8v. | Mid | ✓ | 128Mb | ? | X 32 | 4 | ? | ? | ✓ |
| GDDR I | 2.5v. -3.0v. | None | | 128Mb & 256Mb | To 500 MHz. | X 32 | 4 | 12 | 8 & 9 | |
| | 1.8v. | None | | 128Mb & 256Mb | To 500 MHz. | X 32 | 4 | 12 | 8 & 9 | ✓ |
| DDR I | 2.5v. | None | | 64Mb to 256Mb | To 300 MHz. | X 16 | 4 | 12 & 13 | 8 & 9 | |

The MC uses Fetch-4 mode on all memory parts. Fetch-4 mode uses a burst of two memory clocks to read or write four 64-bit data words. Fetch-2 mode (1-clock bursts of two data words) will never be used, even on slower DDRAMs that support this mode, since fetch-2 does not provide extra slots on the command bus for activate and prefetch commands. It now appears that all DDRAMs under consideration for R400 will support Fetch-4 mode, so R400 will not use Fetch-8 mode (a four clock burst of eight data words), even on parts that support it.

A key issue for the memory controller design is finding ways to hide the latency between finishing with a row in a particular bank until a different row can be accessed in that bank. Typical row cycle times for the least expensive parts are in the range of 60ns, which is 18 to 30 memory clocks for 300MHz to 500MHz DDRAMs. This is the design center, though the MC should also be able to take advantage of more expensive parts with shorter row cycle times, e.g. down to 36ns, or 13-22 clocks. The time from the last access in one row to the first access of a different row in the same bank is typically 2/3 of the row cycle time, or 12 to 30 memory clocks for 300MHz to 500MHz DDRAMs.

The MC hides prefetch and row access latency by scheduling accesses to other banks while a given bank is changing rows. The MC selects accesses from multiple address queues, based on the latency requirements of each client and the time needed to hide precharge and row activate cycles. With the sole exception of texture read accesses, the MC does not change the order of accesses within an address queue. Therefore, clients should be designed so that they typically group together accesses that are in the same row.

Another key issue is minimizing the number of transitions between reading and writing on the memory bus. On older, DDRAM1 parts, a sequence of reads->writes->reads wastes 4-5 clocks on the data bus, due to turnaround time on the bus and inside the memory parts. With DDRAM2 parts, current estimates are for 7-12 clocks wasted on the data bus, with larger numbers for the faster parts. A enhanced write proposal would reduce this to 2-4 clocks, but even if parts are designed that support enhanced writes, we cannot depend on using those parts. Therefore, read and write buffers need to be large enough to reduce the number of read->write->read transitions.

Finally, we must allow the memory clock to run faster than the core clock, to give us more configuration flexibility.
This requires that the MC support more than 128-bits per clock for both address requests and data transfers. Each MC address request specifies a 256-bit addressable unit. Most requests will read or write the entire 256-bits, though some may only access half of that data. Both the RB and the MH can make one request per clock, so address requests exceed 128-bits per clock. Data transfers occur over separate 128-bit read and write busses. Given a mix of reads and writes, the MC can therefore support a memory clock rate up to twice as fast as the core clock rate. In practice, the memory clock is unlikely to ever be more than 50% faster than the core clock.

## 1.3 Memory Formats

The R400 Memory Format Specification describes the format of data in local memory. Some of the details are important for this specification and are reproduced here. All data stored in the local memory uses tiled formats, where the precise tiling format depends on the dimensionality of the data: 1D, 2D, or 3D. The MH can translate the tiled formats so that its clients can access memory linearly, but direct accesses from the RB always used tiled formats.

The R400 frame buffer formats divide local memory into eight disjoint subsets. Each memory controller contains two subsets that are organized by banks. In memory controller 0, for example, subset ab0 contains all of the memory words addressed in banks A and B. Subset cd0 contains all of the memory words addressed in banks C and D. RV400 frame buffer formats are similar, except that the RV400 divides local memory into four disjoint subsets, since the RV400 has only two memory controllers instead of four.

The 2D and 3D tiling formats interleave even/odd scanline pairs within each 128-bit access. Adjacent 128-bit words step through an 8x8 pixel tile before stepping to the next tile. Since the smallest pixel size is 8-bits, this means that the smallest allocatable unit of memory is 512-bits. After filling a page in bank A (for example) with 8x8 blocks, the next 128-bit word comes from the same page on bank B. When bank B is full, the next 128-bit word comes from the next page in bank A, etc. As a result, stepping horizontally to the next 8x8 tile can never change to a different page within the same bank. Vertically, 2D and 3D tiling alternates between the AB and CD memory subsets, so that stepping vertically also can never change to a different page within the same bank. 3D tiling also alternates memory subsets when stepping in the Z direction.
{Open Issue: *How many different DDRAM types will R400 support?* DDRI, SGRAM, DDRII, and 500MHz type, maybe Elpida and Infineon memory. Elpida has 64Mbit memory at 350-400Mhz, but market is coalescing to 128Mbit at this frequency range. But Elpida has the advantage of internal termination (helpful for mobile products)}

# 2. Client Interface (MCCI)

## 2.1 MCCI Block Diagram

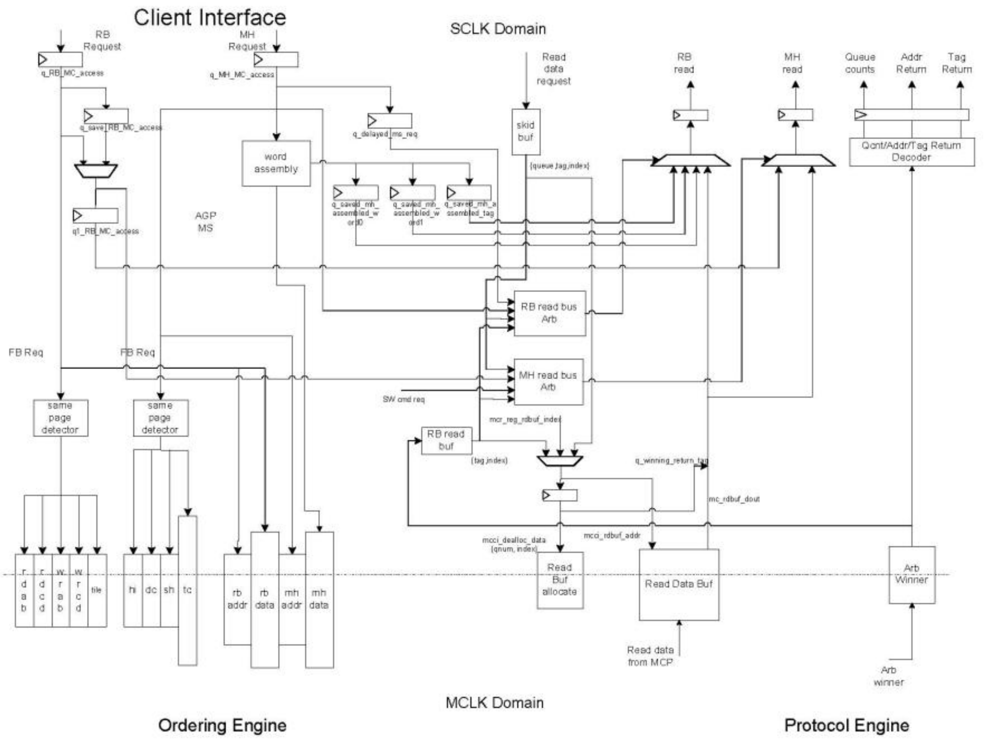
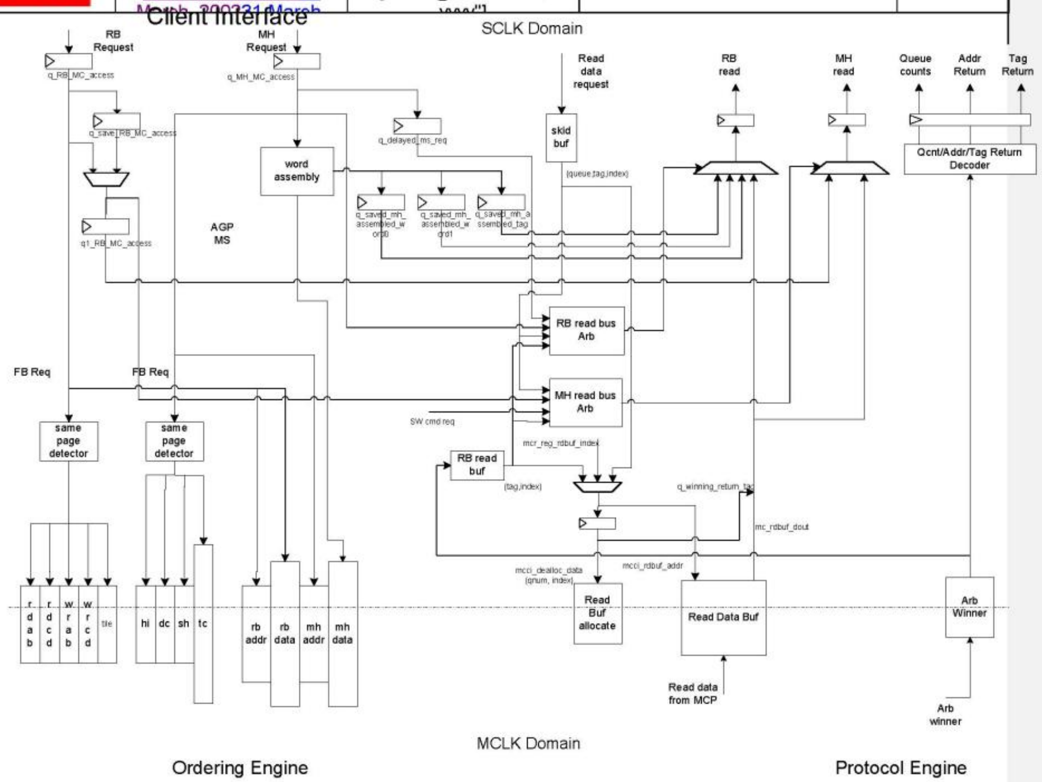


Figure 2: MCCI Block Diagram

The MCCI (MC Client Interface) is the interface between the requestors MH, RB and the MC Ordering engine and Protocol engine. It organizes requests from MH and RB into different queues before passing them on to the Ordering Engine. It reorganizes 32-bit words from MH write requests into 128-bit words. It handles address and tag return depending on the result of arbitration from Ordering Engine. It passes read data coming back from the Protocol Engine on to the requestors MH and RB. It routes requests and data between MH and RB. The boundary between MCCI and the rest of MC is also the interface between SCLK and MCLK.



~~Figure 2: MCCI Block Diagram~~

## 2.2 Receiving Requests from RB and MH

The MH and RB each has their own access bus to the MC (MH Access Bus and RB Access Bus). Each Access bus has the Request portion and the Write Data portion each with their own valid bit (validrequest and validdata). Each portion also has their own destination fields (requestqueue and dataqueue) indicating different requests from different sources to be deposited into different destination queues (to either Ordering engine or the other requestor, MH or RB). Since the two access busses feed into two different same-page detection logic and a non-overlapping set of queues in the MCCI, there is no conflict when the two sends information simultaneously. Instead of having handshake signals, only a send signal is needed per bus to indicate that the bus has something to send. All requests can be for 128-bits of data or 256-bits. On the data side of the bus, the data field of the MH bus is 32 bits wide and that of the RB bus is 128 bits wide. There is also a one-bit word field to indicate which half of the 256-bits the current cycle of data is for. Also for every byte of data there is one bit of bytevalid. For the MH bus, since the data cycle is only 32-bits, there is a 2-bit select field to indicate the position of the word within a 128-bit word. There is also an endofword field to signal the end of a 128-bit transfer. Write data can precede the write request, but the last cycle of data has to match the write request. Other cycles of the data can be matched with other read requests. This increases the utilization of the access busses. There is no longer any wordvalid bits since we eliminated the half populated case. RB will always access 256-bit data at a time. Even if it has only 128 bits of valid data it will still send the other cycle with the proper masks. MH will always send at least one access bus transfer for each of the two 128b words of write data. MH can also send "dummy_write" (also must be 2 cycles) that has all data mask set to zero. The MCCI will mark that request with a same page bit and return a tag on this request. But dummy_write would create bubbles on MCP pipelines.

For those accesses that read or write to the frame buffer, their information is organized into two or three sets of queues (depending on whether the access is read or write) going from MCCI to the Ordering Engine. There are the arbitration queues that contain just enough information for the Ordering Engine to pick the next winner. There are the address store queues and the write data store queues that can be indexed by the Ordering Engine once a winner has been picked to access the complete request information. Each of those groups of queues is divided according to destination and all requests within the same queues are in order. For example, an RB Write AB request would have its arbitration information in its own arbitration queue, address queue and data queue, but they are all in the same order. This way when the queues cross over to the Ordering Engine, the MCO can arbitrate for the winner and then depending on the destination queue index into the corresponding address and data queues and simply select the top entry to access the complete request information. Each of the arbitration queues is 8 deep, as is each of the address queues. The data queues are 128-bits wide and 16 deep each with every 2 entries corresponding to an address queue entry. Tile and host queues will be 8 deep as well.

But before the information can be stored in the queues, they have to go through some organization. For the arbitration queues, there is the same page logic. The same page logic operates on the request portion of the bus. It compares the address (rank, bank, and row) and operation (read or write) of the latest request to that of the last request in the same queue to generate a same page bit. This bit will be later used by the Ordering Engine to determine whether there is any same page advantage to be exploited in arbitration. The MCCI then puts just enough address information (rank, bank, and op) together with the same page bit into the corresponding destination queue awaiting arbitration. A special case is one of the MH's clients, Texture Cache, a read only client. Requests from this client go into a special CAM to be reordered to extract more of the same page sequentiality (explained in 2.5). The output of the CAM is then fed into two arbitration queues for AB and CD bank accesses, as well as the Address queues. In addition, the data portion of the MH access bus has to be assembled into 128-bit words suitable for storage in the MH data queue.

## 2.3 Returning Information to MH and RB

The MCCI needs to communicate information back to the MH and RB as well. For those requests that access the frame buffer, once the arbitration has selected a winner and passed that information to the MCCI and its Read Buffer Allocation logic, the MCCI needs to send a queue count signal back to MH and RB to indicate which queue has been selected winner so that the requester will know that a FIFO space has been freed up in that particular queue. This is passed through the MH QueueCount Bus and RB QueueCount Bus. If the access is a write, then tag information (for MH) or queue information (for RB) should also be returned via the RBMH TagReturn bus upon winning. (Is this necessary?) Meanwhile when the access is a read, the Read Buffer Allocation logic allocates a Read Buffer space to the winner and then sends back on the TagReturn bus the index of the buffer space together with the tag/queue information. The source is always noted to indicate whether the request is from MH or RB. The MH or RB can then request for that read data through RB readdatareq bus and MH readdatareq bus with the proper index indicated. The MCCI will return the appropriate data if the valid bit for that entry is set. When there are read data requests from both RB and MH they will be serviced one from each at a time to maintain fairness. There will need to be skid buffer in addition to storage buffer for read data requests. Note that tag return is a don't care for RB writes to frame buffer, but since the logic already exists for RB reads anyway, always return tag.

The Read Data Buffer has one write port and one read port. The write port interfaces to the Protocol Engine and the read port to the MH and RB read bus. The Read Buffer has a Valid bit associated with each pair of 128-bit word. This valid bit is set when the MCP writes that location and is cleared when RB or MH reads that location. There is also a Allocated bit associated with each entry. Once the entry is allocated to a particular read request, the Allocated bit will be set to 1. Upon receiving data into that location, the Valid bit will be set to 1 as well. As soon as the read data has been passed back on the read bus to MH or RB, both the Allocated bit and the Valid bit are cleared. Both the Valid bit and the Allocated bit are determined on the core clock side. The indices of freed up entries are sent from core clock side to MCLK side to be allocated to winning reads. The indices of data returned from MCP are sent across the other direction to mark the entry as valid. Indices are synchronized through FIFOs when crossing boundaries in either direction. The read data buffer will have 128 pairs of 128-bit entries, so will hold the data from 64 read requests. To prevent deadlock, minimum allocations are set for (groups of) clients. The following minimum and maximum number of buffers will be enforced:

- RB: minimum 8 for all RB queues; a programmable maximum, tentatively set to 32.
- Host: 1, minimum and maximum. There can be only one read outstanding.

- Shared: minimum 1; a programmable maximum.
- Texture: minimum 1, but this should be revisited when the architecture of the cache is set. A programmable maximum.
- Display: 16 minimum and maximum. Logic will be included to hold off the display requests until 8 read buffers are free.

To further prevent deadlock, TC should not allow multiple reads on the same cacheline. Also no client should have a read dependent on a previous read.

~~Any write operation to frame buffer that the RB performs needs to have the corresponding location in the Texture Cache invalidated. They are Write AB, WriteCD, and Tile writes. The information used to invalidate the cache is the coordinate and base fields related to those requests. So for each of the three types of requests, there needs to be an 8-deep FIFO from the RB access bus to the Address Return decoding logic storing the coord and base information. These queues behave similarly to the address and data queues that pass the clock boundary to the Ordering Engine, except they stay within SCLK. When a write operation wins arbitration, the Arb winner logic will select the top entry in one of the three queues and send that back to MH on Address Return bus.~~
~~{Open Issue: There is talk from Steve that texture cache invalidation might be handled entirely outside of MC (in SW?) which would eliminate the need for these coordinate and base queues}~~

There is also a routing functionality in the MCCI. Certain access requests are not destined for the frame buffer and the MCCI needs to route these requests to MH and RB accordingly. Whenever RB performs a system memory access or texture cache fetch, the MCCI has to route the corresponding address and data to the MH. It does so by stealing cycles from the MH Read bus, with the proper source and destination noted on the busses. A system memory write requires three cycles of the MH Read bus, one to send the address and two to send the data. The address is always sent first. Likewise when the MH has data to send back to the RB in response to a TC fetch or system memory read, cycles will be stolen from the RB read bus to send these data with the proper source indicated.

A couple of restrictions on RB->MH requests:

- An AGP write request is 2 cycles on the RB_MC_access bus, but it has to be followed by 1 cycle free of other RB->MH requests.
- No two RB->MH requests can appear on the RB_MC_access bus at the same time.

**Formatted:** Bullets and Numbering

These restrictions are necessary because of the limited amount of request buffering available in MC. If two requests were sent on the same cycle, and there is also a contention with the MH read data return, only one of the incoming requests would be saved and buffered.

MH->RB requests such as AGP read data return and multi-sample requests are always sent in 8 data cycles on the MH_MC_access bus.

## 2.4 Render Backend Queues

The Render Backend reads and writes depth data, color data, and tile data. The RB can read or write local memory or can write this data to the MH for transfer to system memory. The RB also responds to requests from the Texture Unit by computing shadow coverage and filtered pixel colors and sending the results to the Texture Unit via the MH. The RB has six request queues that are divided among these different types of accesses.

The RB Tile queue stores read and write requests for tile data. The tile data stores hierarchical Z and color/depth compression information for each 8x8 tile, in groups of 16 tiles. This queue requires few entries, since there will usually be very few requests outstanding at the same time. The queue stores both read and write requests.

The two RB Read queues store read requests for color and depth data. One queue holds the requests to the local bank A/B subset, the other to the local C/D subset. Now requests for color are not likely to be on the same page as requests for depth. So if color and depth requests are interleaved one color request followed by one depth request, multiple requests to the same page will be separated in the queue and not detected. So when the RB merges the color request stream with the depth stream it should keep the requests to the same page together.

The two RB Write queues store write requests for color and depth data. As for reads, one queue holds the requests to the local bank A/B subset, the other to the local C/D subset. And as for reads the RB should keep writes to the same page together.

Finally, the RB External AGP and TC fetch queues are used for system memory requests. This request is not passed to the Ordering Engine; instead, it is sent over the MH Read buss on the next few clocks without using queues. Other traffic on these busses is delayed. Addresses that are marked as coming from the RB External queue represent external accesses, so the MH converts them into AGP transactions.

The RB processes data in units of 512-bits. All 8x8 tiles of color and depth data occupy a multiple of 512-bits in the frame buffer. Even the tile data comes in units of 512-bits. A tile of 32-bit pixels occupies 2048 bits. The RB may not need to access all of an 8x8 tile, but typically the RB accesses multiple 256-bit words within each tile. As a result, requests in the RB Read and Write queues have a lot of coherency, so that multiple sequential requests are in the same page.

## 2.5 Memory Hub Queues

The Memory Hub processes memory accesses from the Display Unit, the Texture Unit, the AGP/PCI, and a variety of low bandwidth sources, such as the Video Interface Port (VIP). The MH merges all of these accesses onto a single address request bus to the MC. Each request specifies one of four request queues, depending on the source of the request. As with the RB request queues, most of the queued address requests are stored in a common dual ported memory.

The Host Access queue stores writes and possibly one read from AGP/PCI accesses where R400/RV400 is the slave. Additionally, servicing this queue is Urgent if there is a read in the queue, since the host may block until the read is serviced. Otherwise the queue is low priority. Host reads and writes go into the same queue to ensure that order is preserved.

The Shared queue stores read and write requests from several MH clients. This queue has high priority since these clients may stall if they don't get data in time. To handle the case where the real time clients are not receiving the accesses they require, the MH can make the shared queue urgent.

The Display Read queue stores long sequences of reads from up to four different surfaces: a main display surface and an overlay surface for two video outputs. This queue has the lowest priority, except when the MH signals that Display Read is Urgent. Then it has a very high priority, since a display artifact may occur if the Display Unit does not receive data in time. The output of the display queue is enabled for non-eff1 arbitration only if there is space for at least eight requests in the read buffer. Eff1 requests are same page requests that would follow a beginning non-eff1 request. Thus at least eight same page requests would be processed without interruption if they are present in the queue.

Texture read requests are first entered into a CAM to attempt to group accesses on the same page. The MH receives texture requests from the L2 texture cache in an arbitrary order, since the pipelines are not synchronized and since texture accesses would be somewhat chaotic even if they were. The L2 cache removes requests for texels that are already in the L2 cache (or that it already requested) and passes the rest of the requests to the MC.

The Texture queue accomplishes this reordering by providing a 16 entry CAM. The figure below illustrates the CAM entries. Each contains one address request register and a 16-bit comparator, which compares the contents against the most recent texture read address. Priority logic selects the oldest address request register that matches the most recent texture read address, or the oldest address request register if none of them match. When the priority logic selects a texture read as the next access,

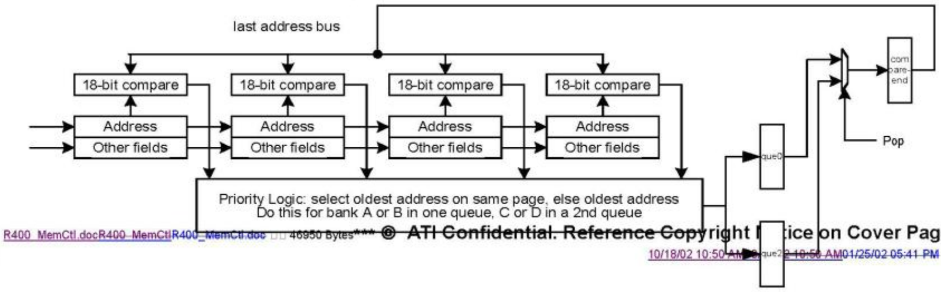


**Figure 4~~3~~: Content Addressable Texture Queue**

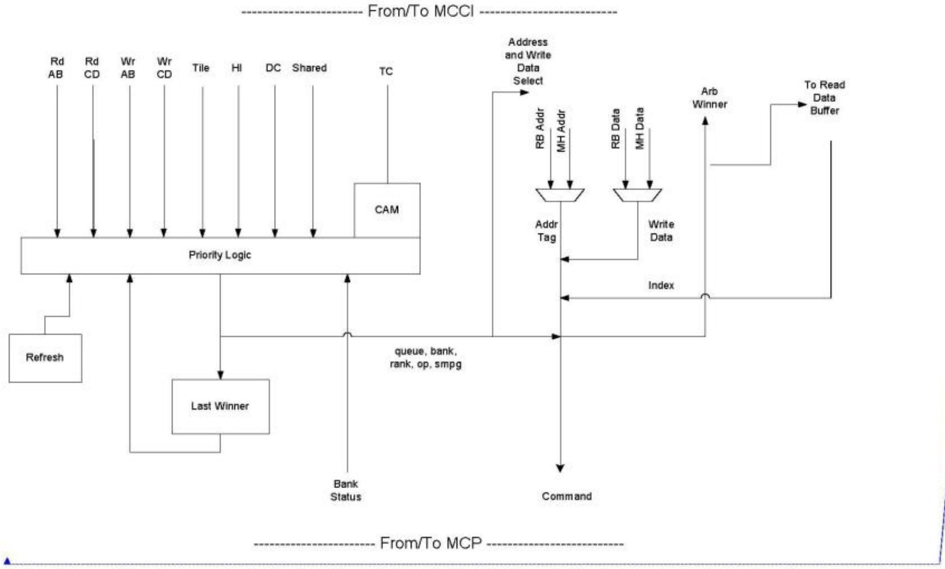
that register is read out and the registers with newer requests transfer their results to close the gap. As a result, the texture queue is able to group texture requests onto the same page within the most recent 16 requests.

# 3. Ordering Engine (MCO)

The MC Ordering Engine (MCO) fetches read/write address requests from queues and schedules the memory accesses. There are multiple request queues. Memory accesses occur in order within a given queue. The Ordering Engine selects which queue to service based on latency requirements for the different clients and based on minimizing dead cycles on the memory bus. The ordering engine works entirely on memory clock.

## 3.1 MCO Block Diagram

The MC Ordering Engine inputs address requests from the Render Backend and the Memory Hub. It produces a sequence of memory access commands for the MC Protocol Engine and a sequence of accesses that are returned to the Memory Hub. The figure below shows the basic structure of the MCO. The Priority Logic looks at the next address from each queue, combines this with information about the efficiency of various accesses and whether a queue has been declared "urgent", and decides on the next address to send to the MC Protocol Engine.



**Formatted:** Font: Times New Roman, Bold

**Figure 54~~46~~: Ordering Engine Block Diagram**

Ten queues feed the priority logic. The storage for these queues is separate for each queue so that it can respond more quickly. The priority logic only receives from each FIFO the five bits needed for the arbitration decision:

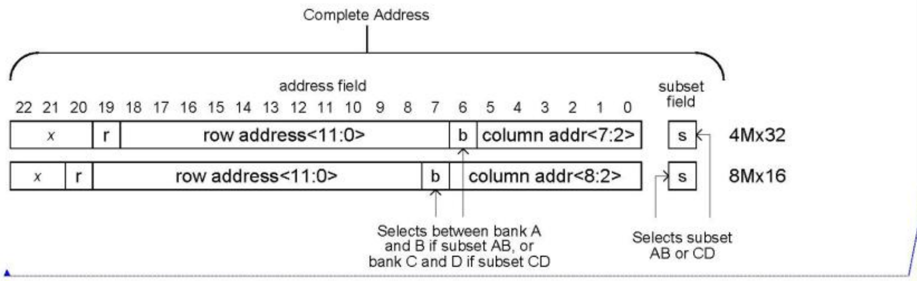
- 2 bits DRAM bank
- 1 bit DRAM rank
- 1 bit operation (0 -> read, 1 -> write)
- 1 bit same page (1 -> this request is on the same page as the last one in this queue)

The address and other data for the request are stored in the address and write data memories. These are addressed by a pointer for each queue. The winner of arbitration is immediately placed in a short queue to cross the clock boundary to the client interface. There each winner is decoded and the appropriate transfer is generated back to the client.

The read buffers are managed by the memory controller. They are allocated when the read operation enters the Protocol Engine, and released when the client reads the data. A valid bit is set when the read data is stored into the buffer. Client reads of the buffer are held up until the data is marked valid.

The operation of the Ordering Engine proceeds as follows. The arbiter selects the top of one of the queues as the next operation to be done based on information saved about the previous winner. That queue entry is popped off and sent several places. It goes to the address and write data memories; the correct queue pointer is used to pop off the address and the write data if necessary. The winner is also sent to the client interface for client notification and to the read buffer allocation logic to receive a read buffer index if needed. When all of this completes, the access is sent to the protocol engine for processing.

Figure 7 below shows some examples of how the MCO interprets the address as passed down from the MCCI. The complete address is indicated with the two fields: address and subset. The actual bank number is indicated with a combination of subset and bank bits. The subset bit "s" indicates whether the request is for subset (bank group) AB or CD. In MCCI, this bit selects the destination queue the request would go into and therefore is placed in its own field apart from the address field in the Access bus. Between the row address and column address is the bank select bit that chooses between Bank A and B if subset AB, or between Bank C and D if subset CD. This bit is so placed to avoid having nearby pixels on different rows of the same bank, but rather same row different banks. "r" selects between two ranks of memory chips. The number of bits indicating row address depends on the DRAM type. So is the number of bits for column address. However, since the address selects a burst of 4, the least significant two column bits are truncated. "x" specifies bits that are unused. Note bank A,B,C,D corresponds to 0,1,2,3 on DRAMs.



**Formatted:** Font: Times New Roman, Bold, Complex Script Font: Bold

**Figure 655~~7~~: Address Subfields**

## 3.2 Priority Logic

The priority logic can be viewed as a simple fixed priority arbiter that selects the highest priority request as the next winner. However each queue can enter the arbitration at several different levels; which levels depends on the state of three fairness counters and the properties of the queue, the request at the head of the queue, and the last winner. These counters implement a limited round robin algorithm among some of the queues.

It is useful to think of queues making requests at several different efficiency levels. These efficiency level definitions are similar to those used in prior memory controllers; they relate the current request to the previous winner:

- efficiency 1. (EFF1) The current request has the same operation and is on the same page as the previous winner.
- efficiency 2. (EFF2) The current request has the same operation and rank but a different bank in relation to the previous winner, or it's EFF1.
- efficiency 3. (EFF3) The current request is to a different bank or rank relative to the previous winner, or it's EFF1.
- efficiency 4. (EFF4) All requests.

Note that if a request qualifies at a certain efficiency level, it will also qualify at all lower levels.

The priority logic takes in the bank status from from the protocol engine. This status indicates when a particular bank can take a request to another page. There is no point issuing a request to a busy bank, so the priority logic removes from consideration all requests to busy banks, other than EFF1 requests. This keeps the ordering engine from issuing requests to a bank while it is precharging. Note that this approach eliminates the requirement to have the EFF3 level of requests in arbitration. The reason to look for another bank is to avoid choosing a bank that is busy. But that can't happen when the bank status is used. A modified EFF2 is still used to give priority to the same operation and rank. So the efficiency definitions are modified as follows:

- efficiency 1. (EFF1) Same as above.
- modified efficiency 2. (EFF2') The current request has the same operation and rank as the previous winner, and it is to a non-busy bank.
- modified efficiency 4. (EFF4') The current request is to a non-busy bank.

In general the priority logic puts requests that are more efficient (with a lower efficiency number) at higher priority. Fairness counters are used to limit the number of contiguous requests of a particular type. This hopefully prevents many accesses of the same type from locking out indefinitely otherwise higher priority requests that are not as efficient. One fairness counter, SAME_PAGE_COUNT, counts the number of contiguous same page accesses. When the counter expires, EFF1 requests are shut off. Since an EFF1 request is probably to a busy bank, it will not make a request at EFF2' or EFF4' until the bank closes. This makes it likely that some other EFF2' or EFF4' request will win. Another counter, SAME_PAGE_COUNT_URGENT, also counts contiguous accesses at EFF1. If the counter expires and an urgent request is pending, EFF1 requests are shut off. Since urgent requests then become the highest priority, one of them will win. This counter has no effect unless it is set to a value less than the SAME_PAGE_COUNT counter. Its purpose is to allow urgent requests to break a string of EFF1 requests sooner than the SAME_PAGE_COUNT counter would. The SAME_OP_COUNT counter counts contiguous non-EFF1 accesses made with the same operation. When the count expires all EFF2' accesses are shut off. So EFF2' requests are forced to EFF4'.

The MH Host, Display, and Shared queues may be marked by the MH to be urgent. This status causes their requests to enter arbitration at a higher level regardless of their efficiency. This is meant to be used only by time critical clients when their accesses are being held off too long. Since ignoring efficiency will reduce total memory performance, the client should not use urgency often. The definition and handling of urgency differs for each of the queues that can be urgent:

- Display: Urgent status will be set when the interface wire is pulsed. It will be cleared when the display read data buffers are full.
- Host: Urgent status will be set when the interface wire is pulsed. It will be cleared when the queue is empty. The host can only have one read outstanding, and fast writes will be distributed over the four MCs, so the queue will empty quickly.
- Shared: Urgent status is declared as long as the interface wire is asserted. Currently only the xDCT block uses urgent. The MH will assert shared urgent to all four MCs when it detects xDCT urgent. This approach may cause the urgent state to be held longer than necessary, but using urgent with the shared queue does not have a huge effect on priority. The shared queue is already near the top of the list, and DRAM bank status is never ignored. So urgent here only causes the shared queue to go above the RB tile queue and bypass the same op efficiency level.

Within each efficiency level the requests are normally ordered as follows, highest priority first:

- RB tile
- MH shared
- MH Texture A/B and MH Texture C/D
- RB read A/B and RB read C/D
- RB write A/B and RB write C/D
- MH Display

For each of MH Texture, RB read, and RB write, the A/B and C/D queues are in a round robin. For each non-EFF1 arbitration, the priority of A/B and C/D are switched for all three queue pairs. This attempts to give all banks equal priority.

Both RB and MH Texture are likely to be high bandwidth clients. Since the texture unit is earlier in the pipeline it should have higher priority. However it may be that allowing the RB some minimum bandwidth will improve performance, so the programmable TEXTURE_WIN_COUNT fairness counter was added. Texture non-EFF1 wins are counted; when the count expires both MH Texture queues are made lower priority than the RB queues. Within each efficiency level the order becomes:

- RB tile
- MH shared
- RB read A/B and RB read C/D
- RB write A/B and RB write C/D
- MH Texture A/B and MH Texture C/D
- MH Display

This priority is maintained until any RB request wins. Then the counter is reloaded and the priority returns to normal.

The overall priority is then, highest first:

- EFF1
- Urgent MH Display
- Urgent DRAM refresh
- Urgent MH host
- Urgent MH shared
- EFF2
- EFF4
- MH host
- DRAM refresh

Note that only one queue may have an EFF1 request because same page is detected only within a queue. These requests will probably be handled special in the priority logic, and not go through the arbiter.

### 3.2.1 *Write-Read Bus Turnaround Enhancement*

Current DRAMs cannot transfer data for many clocks during a transition from write to read. After the last write there is a several clock delay before a read command may be issued. Then the data bus is idle until the CAS latency period is past. To increase the data bus utilization and thus DRAM throughput an enhancement is being considered that would allow additional write data to be transferred during this transition period. The commands and addresses to write this data would be sent after the read commands are finished, at the point of read to write transition.

This raises several challenges for the ordering and protocol engines. At the time the Ordering Engine makes the transition from write to read, it must issue the read accesses at full rate. But it must also select several writes that it will issue at the later read to write transitions. These writes will have their data sent to the Protocol Engine for transmission to the DRAMs, but the addresses and commands must be saved somewhere. How is the Ordering Engine to know what are the correct writes to select to be sent after the reads? What bank will the last reads be to? {To be further discussed}

## 3.3 Refresh Generator

Data is maintained in the DRAMs by issuing Auto Refresh commands. The Auto Refresh approach requires that the MC issue periodic refresh commands to the DRAMs with sufficient frequency to ensure that each row is refreshed within the maximum refresh time specified by the DRAM manufacturer. The row address counters are maintained within the DRAM, and the same row in each bank is refreshed simultaneously.

The refresh generator is implemented in three stages. First, there is a simple divide-by-64 counter that runs on the memory clock. Second, there is a software-programmable timer that defines the nominal row refresh interval. Typically, DRAMs require a 7.8 us row refresh rate. Third, there is a refresh request log counter. The request log counter is initialized to 7, then increments for every required refresh and decrements for every refresh actually performed. Refresh requests are made as long as the counter is non-zero, and an urgent condition exists if the counter is greater than 11 (decimal). The requests return to normal once the counter is below 9. This allows at least 4 refresh requests to pass through whenever the urgency wins arbitration. This method allows refreshing to "work ahead" when there is no meaningful work in the queues, and to tolerate some period of denial. The maximum refresh interval to any single row will be within 1% of the requirement with this method.

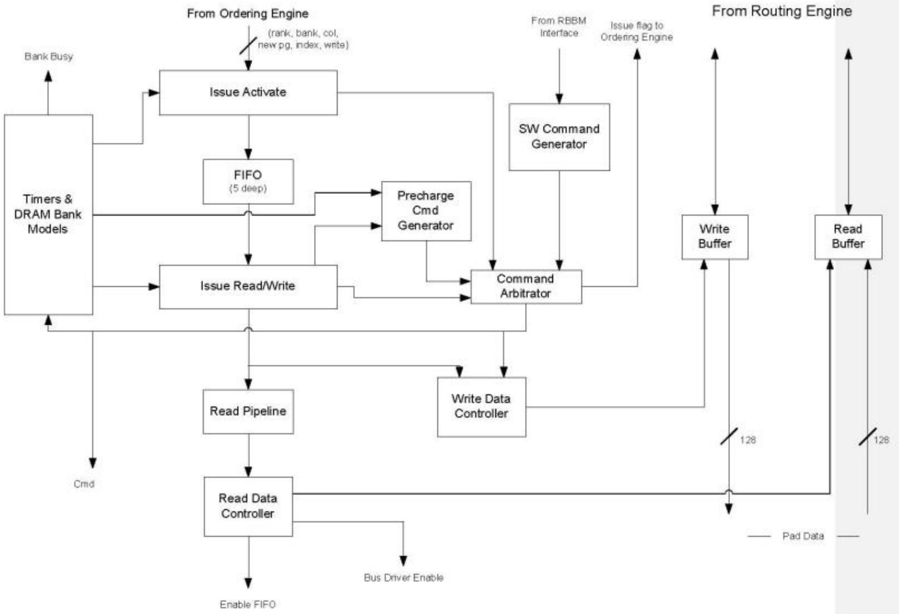
*Issue: Do we support ½ clock speed and Auto Refresh? If so we need to make the first divider programmable. Or we can select between two counters, divide by 64, and divide by 32.*

# 4. Protocol Engine (MCP)

The MC Protocol Engine (MCP) initializes the DDRAM and performs burst reads, burst writes and refresh cycles under the control of the MC Ordering Engine (MCO). The first subsection below describes the basic functionality of the MCP. The second subsection describes the commands passed to the MCP from the MCO.

## 4.1 MCP Block Diagram

The MC Protocol Engine (MCP) accepts the winning requests' address command and write data from the MC Ordering Engine (MCO). It pipelines the request as controlled by the DRAM Timing Generator and outputs command and write data on the memory bus. The Timing Generator also sends bank status information (whether a particular bank is busy or not) back to the MCO to help it select the next winner. The MCP accepts a read index from the MCCI for read operations and sends back the read data to MCCI to be written into the Read data buffer at location indicated by the Index. There is a software controlled command unit which can hold off the MCP pipeline and inject custom commands for purposes of debugging, controlling self refresh, or managing initialization and ACPI power state transitions.



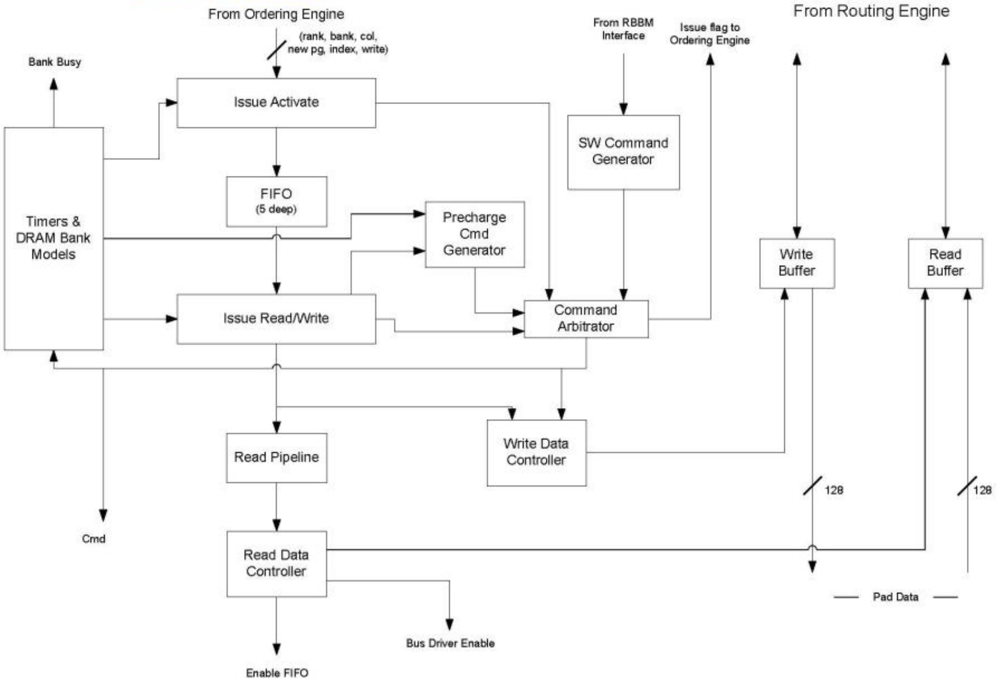


**Figure 7669: Protocol Engine Block Diagram**

Internally, the DDRAM control logic contains multiple registers that store the commands being processed at multiple pipeline delays. This allows the control logic to issue commands, issue read data, and capture write data at different timing delays. It also allows the control logic to look ahead in the queue to decide when to close or open a page in each bank. The control logic also enforces protocol constraints, including prefetch delays, row access delays, read/write transition delays and the special constraints required by Elpida DDRAMs.

## 4.2 Software DRAM Command Unit

Sometimes it is necessary to send commands to the DRAMs that are not associated with read or write operations. Initialization and power down mode entry and exit are two cases. Hardware diagnostics is another case. Rather than anticipating all possible command sequences in the protocol engine controllers, this general purpose unit has been added. Through RBBM interface writes:

- Any command can be sent to the DRAM.
- Commands can be separated by a specified time.
- Any command can be specified as having read or write data associated with it. The timing of the reception of read data or transmission of write data is controlled through the same timers used by the rest of the protocol engine.
- CKE can be manipulated.

~~The software command unit is to be controlled through software writes to the register MC.DRAM_CMD.~~ The register has the following format:

**Formatted:** Heading 2

**Formatted:** Bullets and Numbering

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Tab after: 0.5" + Indent at: 0.5"

**Formatted:** Bullets and Numbering

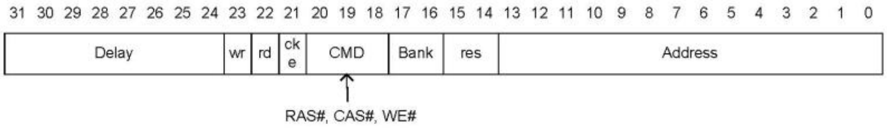


**Figure 8~~7~~7~~10~~: MC DRAM_CMD Register Definition**

The command register has fields that spell out the address and command (RAS#, CAS#, WE#, CKE, etc) as would appear on the external memory bus. It also has a delay field in units of MCLKs to indicate when the next DRAM command can happen. Almost any command can be issued through software. The R400 will be using a wide mixture of DRAMs including both DDR and DDR II. The initialization methods of many of those DDR II type DRAMs have not completely been finalized. Having a software controlled command register allows the flexibility of implementing these init and reset sequences in software. In addition, the power management state machine performs many initializations, precharge, and self refresh. The same benefit of flexibility applies if that state machine is implemented in software.

Whether a command send write data or brings in read data is determined by the wr and rd bits, not the CMD bits. So setting CMD to write alone will not send write data; the wr bit must be set as well. Reads behave similarly. Do not set both wr and rd bits.

### 4.2.1 *Procedure for Sending DRAM Commands*

**Formatted:** Heading 3

**Formatted:** Bullets and Numbering

Any commands send through this facility are not coordinated with read, write, and refresh commands generated by the hardware controllers. So these must first be shut off. Note that shutting off refresh can cause the contents of the DRAM to be lost unless the refreshes are resumed in time.

**Formatted:** List, Bulleted + Level: 1 + Aligned at: 0.25" + Tab after: 0.5" + Indent at: 0.5"

**Formatted:** Bullets and Numbering

**Formatted:** List, Bulleted + Level: 1 + Aligned at: 0.25" + Tab after: 0.5" + Indent at: 0.5"

- If necessary, wait for required operations to complete. An idle MC can be detected by MC_STATUS_0 = 0.
- Clear MC_ENABLE and MEM_REFRESH_RQ_EN in MC_CNTL. This shuts off queue and refresh processing.
- Wait to make sure all operations are finished. Again, MC_STATUS can be checked.
- Set MC_POINTER = 0.
- Write up to 16 commands into MC_DRAM_CMD. This doesn't send out the commands, it just loads a buffer. Each write autoincrements MC_POINTER.
- Send the buffer of commands to the DRAMs by setting ISSUE_DRAM_CMDS in MC_CNTL. Leave MC_POINTER alone; it's used to mark the last command in the buffer.

Any subsequent RBBM operations will be held up until all commands are issued to the DRAM.

### 4.2.2 *Procedure for Initializing Write Data*

**Formatted:** Heading 3

**Formatted:** Bullets and Numbering

If any if the commands issued through the software command unit send write data, that data must be set up before the commands are issued. The write data is pushed into a fifo before the commands are issued. The fifo can hold the data and mask for up to 8 write commands. Each write command requires exactly 9 RBBM writes to the MC_DRAM_DATA register. The first write is 32 bits of byte enables, bit 0 for the first byte and bit 31 for the last. The next eight writes contain the 256 bits of data, with the first write being the least significant word and the last the most significant. Note that the RBBM writes byte enables; one implies write. When they are sent to the DRAM they are byte masks; one implies don't write. So the byte enables are inverted by the MC.

### 4.2.3 *Procedure for Accessing Read Data*

Formatted: Bullets and Numbering

Any data read by commands issued through the software command unit is stored in the read data buffer, like all read commands. Each time ISSUE_DRAM_CMDS is set, the read data is stored beginning at address 0 of the read data buffer and continuing to the next higher address for each read. Note that any old data in the read data buffer will be overwritten.

Once the DRAM reads are complete the read data can be accessed by setting the MC_POINTER register to the correct address and reading the MC_RDBUF_DATA register. The data from each read occupies 8 addresses as seen by MC_POINTER. Each read of MC_RDBUF_DATA increments MC_POINTER by one. So to access the data from the first read set MC_POINTER to 0 and read MC_RDBUF_DATA 8 times. MC_POINTER will now be 8, the address of the first word of the second DRAM read command.

### 4.2.4 *Procedure for Looping a Set of Commands*

Formatted: Bullets and Numbering

It is possible to loop continuously through up to 16 commands loaded into the command buffer. To do this the MC must have just been reset, without any intervening DRAM write commands. Normally the DRAM initialization sequence does not write the DRAM, so this is OK. Next load the commands as described above and initialize any needed write data, also as described above. There must be exactly enough write data for the write commands in one pass throught the command loop. Each pass will resend the same sequence of write data as the first. To specifiy looping, set the LOOP_DRAM_CMDS bit in the MC_DEBUG register. Then set ISSUE_DRAM_CMDS as before.

The only way to terminate the loop is to reset the MC, either by chip reset or a MC soft reset. If DRAM reads are included in the loop, the data is stored in the read data buffer starting at location 0. Each pass of the loop does not reset the buffer address. The address register has a cycle of 32 read commands, and so continuously stores read data in buffers 0 to 31, and repeating. Each buffer uses 8 address of MC_POINTER, as described above.

Formatted: Font color: Auto

~~{Open Issue: Are software reads and writes to the RBBM in order? If so software can read a register after the write to make sure the command has been executed.}~~

~~{Open Issue: Is the SW approach to power management adequate? Is it OK for R400 desktop product to not do self refresh? Not that much difference in power. Current plan is to let the CP handle the ACPI power state transitions. Steve and Greg are working on it.}~~

## ~~4.2~~4.3 Protocol Engine Commands

Formatted: Bullets and Numbering

The MC Ordering Engine (MCO) uses the MemCmd bus to send burst read, burst write, and refresh commands to the MCP. The Protocol Engine always executes the commands in order. The following is a description of the fields of each command. These fields support up to 32Mbits of memory behind each data pin. This is enough to support up to a 512MB frame buffer using two 64-bit wide memory controllers.

Bank (2-bits): 0: bank A; 1: bank B; 2: bank C: 3: bank D
ColAddr (7-bits): Stores column address bits<8:2> for a burst 4 access
RowAddr (13-bits): {what is the maximum number of row address bits?}
Rank (1-bit): selects bwtween two ranks of memory chips that use the same data pins
Operation (3-bits): Specifies the memory operation with its address context (see table below)
Index (6-bits): Specifies the location(s) to use in the Memory Read or Memory Write buffer

The table below describes the operations supported by the MCP. The MCO schedules refresh cycles in the same way that it schedules other address requests. The MCO issues a separate burst read or burst write cycle for each 4-word burst. With a 64-bit wide memory controller, these are 256-bit bursts. The MCO also marks each read and write burst to indicate that it is on a different page or the same page as the previous access to the same bank.

| Code | Name | Description | Notes |
|---|---|---|---|
| 000 | Null | No-op | MCP ignores this invalid command |
| 001 | Refresh | Initiate an auto-refresh cycle | MCO decides when MCP must refresh |
| 010 | | (reserved) | {Do we need commands for special operations?} |
| 011 | | (reserved) | |
| 100 | R_Diff | Read burst for the RB, on a different page | This burst is on a different page from the most recent burst to the same bank. |
| 101 | W_Diff | Write burst for the RB, on a different page | |
| 110 | R_Same | Read burst for the RB, on the same page | This burst is on the same page as the most recent burst to the same bank. |
| 111 | W_Same | Write burst for the RB, on the same page | |

**Table 1~~1~~13: MCP Command Bus Operations**

The MCP uses the Index field to select where to access the Memory Read buffer for a given burst. The Index specifies an aligned pair of 128-bit entries in the queue. 128-bit bursts use either the lower or upper half, depending whether the burst column address is even or odd.

# 5. Bus and Pin Interfaces

The three Memory Controller sub-blocks interface to the Memory Hub (MH), the Render Backend (RB), and the local memory pins. This section describes these interfaces and the busses between the MC sub-blocks. The first subsection below describes address interfaces between the MC Ordering Engine (MCO) and the RB and MH. The second subsection below describes data interfaces between the MC Routing Engine (MCR) and the RB and MH. The third subsection describes interfaces between the MC Protocol Engine (MCP) and the other two MC sub-blocks. The final subsection describes the local memory pin interface.

Note that in the tables that follow the convention used is that MC*n* refers to any of the distinct instantiations of the MC, MC0, MC1, MC2, or MC3.

## 5.1 MCCI Interfaces with MH and RB

There are four types of busses between the MCCI and the MH and RB. All ~~except for Address return bus~~ are defined for both RB and MH:

- Access bus: Sends request address and write data from MH and RB to MCCI
- Queue count bus: Returns indicator about each arbitration queue freeing up to MH and RB
- Read data request bus: For MH and RB to request read data in the MCCI read data buffer once they received the tag/index.
- Read data return bus: Returns read data from MCCI to RB or MH.

### 5.1.1 *Access Bus*

MH Access specifies a 256-bit burst read or write address and associated information. The MH may send whenever it has either valid address or valid data and will indicate so by asserting the send bit. Address together with subset select the 256-bit word to access (refer to section 3.1 for more on address format). Write selects a read or write transfer and Requestqueue selects one of 4 different arbitration queues that store requests from Memory Hub clients (TC will be split into different queues after the CAM). MH also sends down a tag field with each request and keeps track of them through tagreturn bus. Urgent bits specifies whether the MCO should raise the priority of this Queue. Dataqueue selects destination for data which could be the RB. Since TC and DC are read only clients, they are not valid as write data queues. MH transfers 32 bits per cycle due to routing constraints as well as the fact that it is relatively low bandwidth (as compared to RB). Select specifies the relative position of the 32 bit word in the 128 bit word. Endofword is important to signal end of an 128 bit transfer, especially for the first half of a 256-bit. For the second half the indication can be derived by the fact that the write request will accompany the data in the same cycle, but MH guarantees the Endofword to be set correctly for either half. AGP read data return and multi-sample requests are always sent in 8 data phases.

| Name | Bits | Description |
|---|---|---|
| MH_MC*n*_access_send | 1 | R400 standard flow control |
| | | |
| MH_MC*n*_access_validrequest | 1 | request portion of transfer is valid |
| MH_MC*n*_access_requestqueue | 3 | Selects one of the MH request queues<br>0: HI<br>1: TC<br>2: DC<br>3: Shared<br>4:7 not valid target for requests |
| MH_MC*n*_access_address | 24 | Address of this 256-bit access within this memory controller, allows reach of 512MB per MC which equals the system maximum. Format is Device Addr – FB_START |
| MH_MC*n*_access_write | 1 | 0: read request; 1: write request |
| MH_MC*n*_access_tag | 6 | MH generated tag associated with this access |
| | | |
| MH_MC*n*_access_validdata | 1 | data portion of transfer is valid |
| MH_MC*n*_access_dataqueue | 3 | Selects one of the MH request queues<br>0: HI<br>1: TC<br>2: DC<br>3: Shared<br>4: not used<br>5: dest is RB in response to system memory read<br>6: dest is RB in conjunction with TC fetch_shadow request<br>7: dest is RB in conjunction with TC fetch_multisample request |
| MH_MC*n*_access_word | 1 | Selects a 128-bit word out of the 256-bit buffer location |
| MH_MC*n*_access_bytevalid | 4 | Overloaded field:<br>If data_queue is 0, 1, 2, or 3 then use as bytevalids. If 1, the corresponding byte should be written by the destination<br>If data_queue is 5, then these lines transfer the tag associated with the RB read back to the RB in multiple phases.<br>phase0: bits[3:0] = tag[3:0]<br>phase1: bits[3:0] = tag[7:4]<br>phase2: bits[0] = tag[8] |
| MH_MC*n*_access_select | 2 | Selects which 32 bit section of the 128 bit word is being transferred |
| MH_MC*n*_access_data | 32 | data for this cycle |
| MH_MC*n*_access_endofword | 1 | Indicates that this transfer completes this 128 bit word |
| | | |
| MH_MC*n*_access_dcurgent | 1 | Display requests, pulsed, serviced while read buffers are available |
| MH_MC*n*_access_sharedurgent | 1 | Shared requests, level, urgent priority until signal is lowered |
| MH_MC*n*_access_hiurgent | 1 | Host requests, pulsed, serviced until request queue is empty |

**Table 2: MC Access Interface**

~~**Table 224: MH Access Interface**~~

RB Access bus is similar to the MH access bus. It transfers 128 bits of write data rather than 32 and therefore requires no Endofword bit. The Address is 26 bits instead of 23 bits. This allows the RB to specify a 256-bit access out of the $2^{32}$-byte system address space for AGP accesses. Also the RB does not have a tag field because it does not keep track of the requests. The TC fetch requests (shadow and multisample) are a different type of access. These requests are not made to the MC but to MH, so they have special formats and the request is actually sent in the data field of the access bus rather than the "normal" request fields. This is why the requestqueue field does not use codes 6 and 7.

| Name | Bits | Description |
|---|---|---|
| RBn_MCn_access_validrequest | 1 | The ~~R~~request (RB*n*_MC*n*_access_write) is valid |
| ~~RB*n*_MC*n*_access_subset~~ | ~~1~~ | ~~0: use the AB memory subset; 1: use the CD memory subset Not valid for AGP accesses~~ |
| RB*n*_MC*n*_access_address | 27~~6~~ | Address of this 256-bit aligned access. ($2^{32}/2^{5}=2^{27}$)~~access within the selected memory subset~~ |
| RB*n*_MC*n*_access_write | 1 | 0: read request; 1: write request |
| RB*n*_MC*n*_access_tag | 9 | Tag, to be returned to the RB with the read data |
| RB*n*_MC*n*_access_requestqueue | 3 | Selects one of six~~five~~ RB request queues<br>0: ReadAB<br>1: ReadCD<br>2: WriteAB<br>3: WriteCD<br>4: Tile<br>5: AGP access<br>6: TC fetch_shadow(not used)~~reserved~~<br>7: TC fetch_multisample (not valid for any RB request, data only)~~reserved~~ |
| | | |
| RBn_MCn_access_validdata | 1 | Request is valid |
| RB*n*_MC*n*_access_dataqueue | 3 | Selects one of four RB write request queues~~ for this transaction~~<br>0: ReadAB (not valid for an RB write request)<br>1: ReadCD (not valid for an RB write request)<br>2: WriteAB<br>3: WriteCD<br>4: Tile<br>5: AGP access<br>6: TC fetch_shadow ~~(not valid for any RB request)~~(not used)<br>7: TC fetch_multisample ~~(not valid for any RB request)~~ |
| RB*n*_MC*n*_access_word | 1 | ~~Selects a 128-bit word out of the 256-bit buffer location~~Valid only for writes. Coincident with the data.<br>0: The least significant 128 bits; 1: The most significant 128 bits. |
| RB*n*_MC*n*_access_data | 128 | The 128-bit data for this transfer |
| RB*n*_MC*n*_access_bytevalid | 16 | ~~If 1, the corresponding byte should be written by the destination~~1: The corresponding byte must be written; 0: The byte must not be written to memory |

**Table 43~~35~~: RB Access Interface**

### 5.1.2 *Queue Count Bus*

The MH and RB QueueCount Interfaces allow the MC's clients to track how many request queue entries are available for use. MH and RB each keeps internal counters for the available entries in each arbitration queue. They are decremented for every request sent and incremented when the corresponding queuecount signal is asserted.

| Name | Bits | Description |
|---|---|---|
| MC*n*_ MH_queuecount_hi | 1 | HI request queue has had one entry read out |
| MC*n*_ MH_queuecount_tc | 1 | TC request queue has had one entry read out |
| MC*n*_ MH_queuecount_dc | 1 | DC request queue has had one entry read out |
| MC*n*_ MH_queuecount_shared | 1 | Shared request queue has had one entry read out |

**Table 54~~46~~: MH QueueCount Interface**

| Name | Bits | Description |
|---|---|---|
| MC*n*_RB*n*_queuecount_readab | 1 | ReadAB request queue has had one entry read out |
| MC*n*_RB*n*_queuecount_readcd | 1 | ReadCD request queue has had one entry read out |
| MC*n*_RB*n*_queuecount_writeab | 1 | WriteAB request queue has had one entry read out |
| MC*n*_RB*n*_queuecount_writecd | 1 | WriteCD request queue has had one entry read out |
| MC*n*_RB*n*_queuecount_tile | 1 | Tile request queue has had one entry read out |

**Table 65~~57~~: RB QueueCount Interface**

### 5.1.3 *Tag Return Bus*

The TagReturn interface for the MH and RB provides tags and or indices that indicate when the MC has completed 256-bit reads and writes. The MH and the RB both read this interface and select the tags for their own MC requests. The MH and RB must be able to accept a tag on every clock cycle, so no handshake signal is required. The index field is interpreted the same way by both MH and RB, but the tag field is read by MH as the tag that it sent on the access bus, read by RB as the queue identifier as on the access bus.

| Name | Bits | Description |
|---|---|---|
| MC*n*_tagreturn_send | 1 | data is valid on this cycle |
| MC*n*_tagreturn_index | 6 | Selects an aligned pair of 128-bit words in the read buffer |
| MC*n*_tagreturn_tag | 6 | Either the MH tag or the RB queue identifier |
| MC*n*_tagreturn_write | 1 | Write request if 1, else read request |
| MC*n*_tagreturn_source | 1 | 0: MH request; 1: RB request |

**Table 76~~69~~: RBMH TagReturn Interface**

### ~~5.1.4~~ *Read Data Request Bus*

Formatted: Bullets and Numbering

### ~~5.1.5~~5.1.4

~~The RB Read Data Request interface specifies a 256-bit read from the Memory Data Read Buffer.~~

| ~~Name~~ | ~~Bits~~ | ~~Description~~ |
|---|---|---|
| ~~RB*n*_MC*n*_readdatareq_send~~ | ~~1~~ | ~~Ready to send~~ |
| ~~MC*n*_RB*n*_readdatareq_rtr~~ | ~~1~~ | ~~Ready to receive a new RB read request~~ |
| | | |
| ~~RB*n*_MC*n*_readdatareq_index~~ | ~~6~~ | ~~Selects a 256-bit location to read from the data buffer~~ |

~~**Table 714: RB Read Data Request (readdatareq) Interface**~~

The MH Read Data Request interface specifies a 256-bit read from the Memory Data Read Buffer.

| Name | Bits | Description |
|---|---|---|
| MH_MC*n*_readdatareq_send | 1 | Data is valid on this clock |
| MC*n*_MH_readdatareq_rtr | 1 | Ready to receive a new MH read request |
| MH_MC*n*_readdatareq_index | 6 | Selects a 256-bit location to read from the data buffer |
| MH_MC*n*_readdatareq_tag | 6 | Tag for this transaction |
| MH_MCn_readdatareq_queue | 2 | which request queue generated this read data<br>0: HI<br>1: TC<br>2: DC<br>3: Shared~~Queue for this transaction~~ |

**Table 98~~815~~: MH Read Data Request (readdatareq) Interface**

### ~~5.1.6~~5.1.5 *Read Data Return Bus*

The RB Read interface specifies a 128-bit read of data into the RB. The source field indicates whether the data comes from MC (frame buffer) or the MH. As a result the index field is overloaded to indicate index to read data buffer if the source is MC, or indicate the nature of the data if the source is MH. Word field specifies whether this transfer is the upper or lower half of the 256-bit word.

| Name | Bits | Description |
|---|---|---|
| MC*n*_RB*n*_read_send | 1 | Data is valid on this clock |
| MC*n*_RB*n*_read_tag | 9 | Overloaded field:<br>source == MC: The RB's original read request tag is returned<br>source == MH: Encoded value specifying nature of the data<br>tag = 0x40: Multi-sample request from TC<br>Others: As indicated by MH |
| MC*n*_RB*n*_read_word | 1 | 128-bit word out of the 256-bit buffer location |
| MC*n*_RB*n*_read_source | 1 | 0: source is MC; 1:source is MH; |
| MC*n*_RB*n*_read_data | 128 | The 128-bit data for this transfer |

**Table 109~~20~~: RB Read Interface**

The MH Read interface is defined similarly to the RB Read Interface, except the index is not returned.

| Name | Bits | Description |
|---|---|---|
| MC*n*_MH_read_send | 1 | Data is valid on this clock |
| MC*n*_MH_read_tag | 6 | Overloaded field:<br>source == mc~~0~~: tag associated with this transaction<br>source == rb~~1~~: encoded value specifying nature of data<br>Tag = 0: RB AGP access descriptor~~external AGP access address~~<br>Tag = 5: RB AGP write data portion~~write data going to system memory~~<br>Tag = 6: data to TC in response to Fetch_shadow (no longer valid)<br>Tag = 7: data to TC in response to Fetch_multisample |
| MC*n*_MH_read_word | 1 | 128-bit word out of the 256-bit buffer location |
| MC*n*_MH_read_source | 1 | 0: source is MC;1: source is RB |
| MC*n*_MH_read_data | 128 | If source = = rb~~1~~ and tag == 0:<br>bits [0]: 0:read, 1:write ~~subset~~<br>bits [31:5~~22:1~~]: Device address of access [31:5]~~address~~<br>bits [40:32~~23~~]: AGP request tag~~write~~<br><br>~~bits [127:24] unused~~<br>if source == mc~~Else~~:<br>Read data returned to MH~~The 128-bit data for this transfer~~ |
| MC*n*_MH_read_bytevalid | 16 | If 1, the corresponding byte should be written by the destination. Not used when source = 1 and tag = 0. |

**Table 11~~1010~~2~~1~~: MH Read Interface**

## 5.2 MC Pin Interface

***{To be specified}***

TP Block Performance

The first 3 sections of this document identify areas that affect performance. The fourth section defines the performace cases that need to be verified along with details explanations of how to construct these tests.

3 things control flow within the TP:

tpc_walker state machine
tpc_aligner state machine
tpc_rr_fifo latency FIFO

The state machines are counters that count down form a variable maximum count. For performance, each state machine must:

generate the proper number of cycles
determine the worst case cycles over the 4 TPs properly

Both state machines also read from FIFOs separating them from the previous pipeline. Thus, it is necessary to verify the following:

Proper FIFO functionality, size, and watermarks
Optimal read logic.

## 1. tpc_walker

### 1.1 State machine

For mipmapping where TP0..3 require a maximum of n cycles, tpc_walker must generate n cycles for each cycle from the SQ/SP, n=[1..2].
For volume filtering where TP0..3 require a maximum of n cycles, tpc_walker must generate n cycles for each cycle from the SQ/SP, n=[1..2].
For hicolor cases where TP0..3 require a maximum of n cycles, tpc_walker must generate n cycles for each cycle from the SQ/SP, n=[1..4].
For anisotropic cases where TP0..3 require a maximum of n cycles, tpc_walker must generate n cycles for each cycle from the SQ/SP, n=[1,2,4,6,8,10,12,14,16].

For a combination of hicolor, mipmapped, volume filtering, and anisotropic filtering modes where TP0..3 require a maximum of h, m, v, and a cycles for each mode respectively, tp_walker must generate h*m*v*a cycles for each cycle from the SQ/SP, h=[1..4], m=[1..2], v=[1..2], a=[1,2,4,6,8,10,12,14,16].

### 1.2 Control

Is the tpc_walker state machine sending TP_SQ_dec back to SQ as early as possible?

Are the following FIFOs read by the walker state machine properly sized to work with the SQ_TP interface?

tpc_walker_fifo
tp_lod_fifo
tp_coord_fifo

## 2. tpc_aligner

### 2.1 State machine

For alignment cases where TP0..3 require a maximum of n cycles, does tpc_aligner generate n cycles for each cycle from the SQ/SP, n=[1,2,4].

### 2.2 Control

Is the FIFO read enable optimal?

Is the RTR that stalls the previous pipeline section optimal? This involves the FIFO control logic as well as FIFO sizes and watermarks.

Are RTRs from following pipelines being efficiently handled (no extra stalls)?

## 3. tpc_blend

TPC_TC_rtr and whether the tpc_rr_fifo latency FIFO is full are the 2 condition that stall the tpc_aligner. The latency FIFO will cause stall, but it has been sized to handle typical memory latencies.

### 3.1 Control

Is tpc_rr_fifo sized properly? That is, is there an unresonable amount of stalling for actual memory latencies

## 4. Cases to Test

Assuming no external bottlenecks (0 memory latency, 100% cache hits), each cycle requires the following number of cycles to complete:

h*m*v*a*al

| | |
|---|---|
| h | # of cycles to handle the DATA_FORMAT [1,2,3,4] |
| m | mip filter mode [1,2] |
| v | volume filte rmode [1,2] |
| a | aniso filter mode [12,4,6,8,10,12,14,16] |
| al | alignment multicyling [1,2,4] |

Ideal performance can be computed using the observations above as a basis.

### 4.1 General Test Characteristics

All tests must render a destination region large enough to reach steady state performance. The tests must also eliminate TC and memory system bottlenecks. Unless otherewise notes, the tests in sections 4.3, 4.4, and 4.5 have the following characteristics:

- Destination regions is 128x128 defined by V0, V1, V2, and V3. This area can be filled using two triangles defined V0, V1, V3, and V0, V3, and V2.

  V0.[x,y] = [0.0, 0.0]
  V1.[x,y] = [128.0, 0.0]
  V2.[x,y] = [0.0, 128.0]
  V3.[x,y] = [128.0,128.0]

- BORDER_SIZE = 0, BORDER_COLOR = ARGB White

- The base texture map is defined by:

  Width = 128
  Height = 128

Depth = 2

These are to be programmed into the const.SIZE according to the dimension of the texture map required: normally 2D, 3D for cases with volume filter.

- DIM=2D, unless volume filtering is enabled, in which case it is set to 3D
- MAX_ANISO set to 16:1 to avoid concealing incorrect aniso ratio setup with a clamp. Where the case name does not specify an aniso ratio, it is disabled.
- Clamp modes X, Y, and Z should all be set to “clamp to border color”. This will remove TC and the memory subsystem as bottlenecks
- The texture coordinates (s, t, w) should be programmed as follows

V0.[s,t] = [2.0, 2.0]
V1.[s,t] = [2.0+(n*sqrt(2)), 2.0]
V2.[s,t] = [2.0, 2.0+sqrt(2)]
V3.[s,t] = [2.0+(n*sqrt(2)), 2.0+sqrt(2)]
V{0..3}.p(aka r) = 0.5f

The s and t coordinates are meant to create a 50-50 blend between the mip levels 0 and 1 (base map and the next one down) n is defined by the desired aniso ratio, n:1. The w coord of 0.5 will create a texture z coord of 1, which will require a 50-50 blend between 2 z layers when volume filter is turned on. When vol filter is off, it will just clamp to one of the levels.

IF POSSIBLE, BECAUSE ALL TEXELS MAP TO BORDER, IT MAY NOT BE NECESSARY TO INITIALIZE THE TEXTURE.

### 4.2 Gathering of Performance data

The number of cycles elapsed can be approximated by counting the time between the first rising and last falling edge on the TP_SP_data_valid signal. Ideal cycles have been calculate with:

128*128 / 16 pixels/clock * h * m * v * a

h, m, v, a defined at the start of section 4.

## 4.3 FMT_8_8_8_8 Even-sized Texture Maps

| Mode | Pixels | hicolor | mip | volume | aniso | Ideal | Actual | % |
|---|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |
| mip linear, vol point, aniso disabled | 16384 | 1 | 2 | 1 | 1 | 2048 | 2080 | 98% |
| mip point, vol linear, aniso disabled | 16384 | 1 | 1 | 2 | 1 | 2048 | 2080 | 98% |
| mip point, vol point, aniso 2:1 | 16384 | 1 | 1 | 1 | 2 | 2048 | 2080 | 98% |
| mip point, vol point, aniso 4:1 | 16384 | 1 | 1 | 1 | 4 | 4096 | 4156 | 99% |
| mip point, vol point, aniso 6:1 | 16384 | 1 | 1 | 1 | 6 | 6144 | 6236 | 99% |
| mip point, vol point, aniso 8:1 | 16384 | 1 | 1 | 1 | 8 | 8192 | 8312 | 99% |
| mip point, vol point, aniso 10:1 (a) | 16384 | 1 | 1 | 1 | 10 | 10240 | 10392 | 99% |
| mip point, vol point, aniso 12:1 | 16384 | 1 | 1 | 1 | 12 | 12288 | 12468 | 99% |
| mip point, vol point, aniso 14:1 (a) | 16384 | 1 | 1 | 1 | 14 | 14336 | 14544 | 99% |
| mip point, vol point, aniso 16:1 | 16384 | 1 | 1 | 1 | 16 | 16384 | 16624 | 99% |
| mip linear, vol point, aniso 2:1 | 16384 | 1 | 2 | 1 | 2 | 4096 | 4156 | 99% |
| mip linear, vol point, aniso 4:1 | 16384 | 1 | 2 | 1 | 4 | 8192 | 8312 | 99% |
| mip linear, vol point, aniso 8:1 | 16384 | 1 | 2 | 1 | 8 | 16384 | 16624 | 99% |
| mip linear, vol point, aniso 16:1 | 16384 | 1 | 2 | 1 | 16 | 32768 | 33248 | 99% |
| mip point, vol linear, aniso 2:1 | 16384 | 1 | 1 | 2 | 2 | 4096 | 4156 | 99% |
| mip point, vol linear, aniso 4:1 | 16384 | 1 | 1 | 2 | 4 | 8192 | 8312 | 99% |
| mip point, vol linear, aniso 8:1 | 16384 | 1 | 1 | 2 | 8 | 16384 | 16624 | 99% |
| mip point, vol linear, aniso 16:1 | 16384 | 1 | 1 | 2 | 16 | 32768 | 33248 | 99% |
| mip linear, vol linear, aniso 2:1 | 16384 | 1 | 2 | 2 | 2 | 8192 | 8312 | 99% |
| mip linear, vol linear, aniso 4:1 | 16384 | 1 | 2 | 2 | 4 | 16384 | 16624 | 99% |
| mip linear, vol linear, aniso 8:1 | 16384 | 1 | 2 | 2 | 8 | 32768 | 33248 | 99% |
| mip linear, vol linear, aniso 16:1 | 16384 | 1 | 2 | 2 | 16 | 65536 | 66496 | 99% |
| mip point, vol point, aniso 16:1 clamped to 1:1 (1) | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |
| mip point, vol point, aniso 16:1 clamped to 2:1 (1) | 16384 | 1 | 1 | 1 | 2 | 2048 | 2076 | 99% |
| mip point, vol point, aniso 16:1 clamped to 4:1 (1) | 16384 | 1 | 1 | 1 | 4 | 4096 | 4156 | 99% |
| mip point, vol point, aniso 16:1 clamped to 8:1 (1) | 16384 | 1 | 1 | 1 | 8 | 8192 | 8312 | 99% |
| mip point, vol point, aniso 16:1 clamped to 16:1 (1) | 16384 | 1 | 1 | 1 | 16 | 16384 | 16624 | 99% |
| mip linear frac=0, vol point, aniso disabled | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |
| mip point, vol linear frac=0, aniso disabled (b) | 16384 | 1 | 1 | 2 | 1 | 2048 | 2076 | 99% |
| mip linear tri_juice 1/6 frac=1/8, vol point, aniso disabled | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |
| mip linear tri_juice 1/4 frac=7/32, vol point, aniso disabled | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |
| mip linear tri_juice 3/8 frac=21/64, vol point, aniso disabled | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |

(1) The following texture coordinates (s, t, w) should be programmed with different values from above:
V1.[s,t] = [2.0+(16*sqrt(2)), 2.0]
V3.[s,t] = [2.0+(16*sqrt(2)), 2.0+sqrt(2)]
MAX_ANISO set to ratio as specified by "clamped to n:1".

(a) Setting up an exact n:1 ratio caused the actual stepping to be n+2:1. The ratio was modified to slightly less than n:1 to acheive the numbers listed. Using the exact ratio, the actual number of cycles would have been the same as that for the n+2:1 ratio.

(b) To simplify control of the TP, zero fractions in the depth coordinate will still generated 2 cycles. This is because the decision to multicycle is made before the fraction can be found.

## 4.4 FMT_16_16_16_16 Even-sized Texture Maps

| Mode | Pixels | hicolor | mip | volume | aniso | Ideal | Actual | % |
|---|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled | 16384 | 2 | 1 | 1 | 1 | 2048 | 2080 | 98% |
| mip linear, vol point, aniso disabled | 16384 | 2 | 2 | 1 | 1 | 4096 | 4160 | 98% |
| mip point, vol linear, aniso disabled | 16384 | 2 | 1 | 2 | 1 | 4096 | 4160 | 98% |
| mip point, vol point, aniso 2:1 | 16384 | 2 | 1 | 1 | 2 | 4096 | 4160 | 98% |
| mip point, vol point, aniso 4:1 | 16384 | 2 | 1 | 1 | 4 | 8192 | 8316 | 99% |
| mip point, vol point, aniso 6:1 | 16384 | 2 | 1 | 1 | 6 | 12288 | 12476 | 98% |
| mip point, vol point, aniso 8:1 | 16384 | 2 | 1 | 1 | 8 | 16384 | 16632 | 99% |
| mip point, vol point, aniso 10:1 (a) | 16384 | 2 | 1 | 1 | 10 | 20480 | 20792 | 98% |
| mip point, vol point, aniso 12:1 | 16384 | 2 | 1 | 1 | 12 | 24576 | 24948 | 99% |
| mip point, vol point, aniso 14:1 (a) | 16384 | 2 | 1 | 1 | 14 | 28672 | 29108 | 99% |
| mip point, vol point, aniso 16:1 | 16384 | 2 | 1 | 1 | 16 | 32768 | 33264 | 99% |
| mip linear, vol point, aniso 2:1 | 16384 | 2 | 2 | 1 | 2 | 8192 | 8316 | 99% |
| mip linear, vol point, aniso 4:1 | 16384 | 2 | 2 | 1 | 4 | 16384 | 16632 | 99% |
| mip linear, vol point, aniso 8:1 | 16384 | 2 | 2 | 1 | 8 | 32768 | 33264 | 99% |
| mip linear, vol point, aniso 16:1 | 16384 | 2 | 2 | 1 | 16 | 65536 | 66528 | 99% |
| mip point, vol linear, aniso 2:1 | 16384 | 2 | 1 | 2 | 2 | 8192 | 8316 | 99% |
| mip point, vol linear, aniso 4:1 | 16384 | 2 | 1 | 2 | 4 | 16384 | 16632 | 99% |
| mip point, vol linear, aniso 8:1 | 16384 | 2 | 1 | 2 | 8 | 32768 | 33264 | 99% |
| mip point, vol linear, aniso 16:1 | 16384 | 2 | 1 | 2 | 16 | 65536 | 66528 | 99% |
| mip linear, vol linear, aniso 2:1 | 16384 | 2 | 2 | 2 | 2 | 16384 | 16632 | 99% |
| mip linear, vol linear, aniso 4:1 | 16384 | 2 | 2 | 2 | 4 | 32768 | 33264 | 99% |
| mip linear, vol linear, aniso 8:1 | 16384 | 2 | 2 | 2 | 8 | 65536 | 66528 | 99% |
| mip linear, vol linear, aniso 16:1 | 16384 | 2 | 2 | 2 | 16 | 131072 | 133056 | 99% |
| mip point, vol point, aniso 16:1 clamped to 1:1 (1) | 16384 | 2 | 1 | 1 | 1 | 2048 | 2080 | 98% |
| mip point, vol point, aniso 16:1 clamped to 2:1 (1) | 16384 | 2 | 1 | 1 | 2 | 4096 | 4156 | 99% |
| mip point, vol point, aniso 16:1 clamped to 4:1 (1) | 16384 | 2 | 1 | 1 | 4 | 8192 | 8316 | 99% |
| mip point, vol point, aniso 16:1 clamped to 8:1 (1) | 16384 | 2 | 1 | 1 | 8 | 16384 | 16632 | 99% |
| mip point, vol point, aniso 16:1 clamped to 16:1 (1) | 16384 | 2 | 1 | 1 | 16 | 32768 | 33264 | 99% |

## 4.5 FMT_32_32_32_32 Even-sized Texture Maps

| Mode | Pixels | hicolor | mip | volume | aniso | Ideal | Actual | % |
|---|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled | 16384 | 4 | 1 | 1 | 1 | 4096 | 4160 | 98% |
| mip linear, vol point, aniso disabled | 16384 | 4 | 2 | 1 | 1 | 8192 | 8320 | 98% |
| mip point, vol linear, aniso disabled | 16384 | 4 | 1 | 2 | 1 | 8192 | 8320 | 98% |
| mip point, vol point, aniso 2:1 | 16384 | 4 | 1 | 1 | 2 | 8192 | 8320 | 98% |
| mip point, vol point, aniso 4:1 | 16384 | 4 | 1 | 1 | 4 | 16384 | 16636 | 98% |
| mip point, vol point, aniso 6:1 | 16384 | 4 | 1 | 1 | 6 | 24576 | 24956 | 98% |
| mip point, vol point, aniso 8:1 | 16384 | 4 | 1 | 1 | 8 | 32768 | 33272 | 98% |
| mip point, vol point, aniso 10:1 (a) | 16384 | 4 | 1 | 1 | 10 | 40960 | 41592 | 98% |
| mip point, vol point, aniso 12:1 | 16384 | 4 | 1 | 1 | 12 | 49152 | 49908 | 98% |
| mip point, vol point, aniso 14:1 (a) | 16384 | 4 | 1 | 1 | 14 | 57344 | 58228 | 98% |
| mip point, vol point, aniso 16:1 | 16384 | 4 | 1 | 1 | 16 | 65536 | 66544 | 98% |
| mip linear, vol point, aniso 2:1 | 16384 | 4 | 2 | 1 | 2 | 16384 | 16636 | 98% |
| mip linear, vol point, aniso 4:1 | 16384 | 4 | 2 | 1 | 4 | 32768 | 33272 | 98% |
| mip linear, vol point, aniso 8:1 | 16384 | 4 | 2 | 1 | 8 | 65536 | 66544 | 98% |
| mip linear, vol point, aniso 16:1 | 16384 | 4 | 2 | 1 | 16 | 131072 | 133088 | 98% |
| mip point, vol linear, aniso 2:1 | 16384 | 4 | 1 | 2 | 2 | 16384 | 16636 | 98% |
| mip point, vol linear, aniso 4:1 | 16384 | 4 | 1 | 2 | 4 | 32768 | 33272 | 98% |
| mip point, vol linear, aniso 8:1 | 16384 | 4 | 1 | 2 | 8 | 65536 | 66544 | 98% |
| mip point, vol linear, aniso 16:1 | 16384 | 4 | 1 | 2 | 16 | 131072 | 133088 | 98% |
| mip linear, vol linear, aniso 2:1 | 16384 | 4 | 2 | 2 | 2 | 32768 | 33272 | 98% |
| mip linear, vol linear, aniso 4:1 | 16384 | 4 | 2 | 2 | 4 | 65536 | 66544 | 98% |
| mip linear, vol linear, aniso 8:1 | 16384 | 4 | 2 | 2 | 8 | 131072 | 133088 | 98% |
| mip linear, vol linear, aniso 16:1 | 16384 | 4 | 2 | 2 | 16 | 262144 | 266176 | 98% |
| mip point, vol point, aniso 16:1 clamped to 1:1 (1) | 16384 | 4 | 1 | 1 | 1 | 4096 | 4160 | 98% |
| mip point, vol point, aniso 16:1 clamped to 2:1 (1) | 16384 | 4 | 1 | 1 | 2 | 8192 | 8316 | 99% |
| mip point, vol point, aniso 16:1 clamped to 4:1 (1) | 16384 | 4 | 1 | 1 | 4 | 16384 | 16636 | 98% |
| mip point, vol point, aniso 16:1 clamped to 8:1 (1) | 16384 | 4 | 1 | 1 | 8 | 32768 | 33272 | 98% |
| mip point, vol point, aniso 16:1 clamped to 16:1 (1) | 16384 | 4 | 1 | 1 | 16 | 65536 | 66544 | 98% |

## 4.6 FMT_8_8_8_8 Odd-sized Texture Maps

Odd-sized texture maps are used to stress the aligner state machine, both with and without the presence of walker multicycling. Both cases vary slightly from the standard form described in 4.1.

Non-anisotropic filtering cases:

- The base texture map is defined by:

  Width = 7
  Height = 7
  Depth = 4

  These are programmed into the const.SIZE according to the dimension of the texture map required.

- MAX_ANISO set as specified in the test case name

- Clamp modes X, Y, and Z are set to wrap. The clamp-to-border trick can't be used since that would optimize the number of texel requests to one through the aligner, making it unnecessary to multicycle.

- The texture coordinates (s, t, w) should be programmed as follows

  V0.[s,t] = [0.0, 0.0], [0.0, 0.5], [0.5, 0.0]
  V1.[s,t] = [0.0, 0.0], [0.0, 0.5], [0.5, 0.0]
  V2.[s,t] = [0.0, 0.0], [0.0, 0.5], [0.5, 0.0]
  V3.[s,t] = [0.0, 0.0], [0.0, 0.5], [0.5, 0.0]
  V{0..3}.p(aka r) = 0.5f

  Constant s and t coordinates minimize the amount of cache accesses and create predictable alignment behavior. The 3 values above used are for 4-cycle, 2-cycle horizontal, and 2-cycle vertical alignment stalls, respectively.

- SetGradients* commands are used to set up a desired LOD and aniso ratio.

Anisotropic filtering cases are the same as the non-anisotropic odd-mapped cases except:

- The texture coordinates (s, t, w) should be programmed as follows

  V0.[s,t] = [0.0, 0.0]
  V1.[s,t] = [0.0 + s', 128.0 + t']
  V2.[s,t] = [(i*128.0) + s', 128.0 + t']
  V3.[s,t] = [(i*128.0) + s', 0.0 + t']
  V{0..3}.p(aka r) = 0.5f

  S' is 0.0 if an alignment stall is desired in the horizontal dimension or 0.5 if not. T' is 0.0 if an alignment stall is desired in the vertical dimension or 0.5 if not. The parameter i is used to set up an i:1 anisotropic ratio. The endpoints being 128.0 mean that each sample will land on the same point in the texture map, just on different copies.

| Mode | Pixels | ch | mip | vol | aniso | align | Ideal | Actual | % |
|---|---|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled, all ee | 16384 | 1 | 1 | 1 | 1 | 4 | 4096 | 4156 | 99% |
| mip point, vol point, aniso disabled, 2 ee x | 16384 | 1 | 1 | 1 | 1 | 2 | 2048 | 2076 | 99% |
| mip point, vol point, aniso disabled, 2 ee y | 16384 | 1 | 1 | 1 | 1 | 2 | 2048 | 2076 | 99% |
| mip linear, vol point, aniso disabled, all ee | 16384 | 1 | 2 | 1 | 1 | 4 | 8192 | 8312 | 99% |
| mip linear, vol point, aniso disabled, 2 ee x | 16384 | 1 | 2 | 1 | 1 | 2 | 4096 | 4156 | 99% |
| mip linear, vol point, aniso disabled, 2 ee y | 16384 | 1 | 2 | 1 | 1 | 2 | 4096 | 4156 | 99% |
| mip point, vol linear, aniso disabled, all ee | 16384 | 1 | 1 | 2 | 1 | 4 | 8192 | 8312 | 99% |
| mip point, vol linear, aniso disabled, 2 ee x | 16384 | 1 | 1 | 2 | 1 | 2 | 4096 | 4156 | 99% |
| mip point, vol linear, aniso disabled, 2 ee y | 16384 | 1 | 1 | 2 | 1 | 2 | 4096 | 4156 | 99% |
| mip linear, vol linear, aniso disabled, all ee | 16384 | 1 | 2 | 2 | 1 | 4 | 16384 | 16624 | 99% |
| mip linear, vol linear, aniso disabled, 2 ee x | 16384 | 1 | 2 | 2 | 1 | 2 | 8192 | 8312 | 99% |
| mip linear, vol linear, aniso disabled, 2 ee y | 16384 | 1 | 2 | 2 | 1 | 2 | 8192 | 8312 | 99% |
| mip point, vol point, aniso 2:1, all ee | 16384 | 1 | 1 | 1 | 2 | 4 | 8192 | 8312 | 99% |
| mip point, vol point, aniso 16:1, all ee | 16384 | 1 | 1 | 1 | 16 | 4 | 65536 | 66496 | 99% |
| mip linear, vol point, aniso 2:1, all ee | 16384 | 1 | 2 | 1 | 2 | 4 | 16384 | 16624 | 99% |
| mip linear, vol point, aniso 16:1, all ee | 16384 | 1 | 2 | 1 | 16 | 4 | 131072 | 132992 | 99% |
| mip point, vol linear, aniso 2:1, all ee | 16384 | 1 | 1 | 2 | 2 | 4 | 16384 | 16624 | 99% |
| mip point, vol linear, aniso 16:1, all ee | 16384 | 1 | 1 | 2 | 16 | 4 | 131072 | 132992 | 99% |
| mip linear, vol linear, aniso 2:1, all ee | 16384 | 1 | 2 | 2 | 2 | 4 | 32768 | 33248 | 99% |
| mip linear, vol linear, aniso 16:1, all ee | 16384 | 1 | 2 | 2 | 16 | 4 | 262144 | 265984 | 99% |

## 4.7 FMT_16_16_16_16 Odd-sized Texture Maps

| Mode | Pixels | ch | mip | vol | aniso | align | Ideal | Actual | % |
|---|---|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled, all ee | 16384 | 2 | 1 | 1 | 1 | 4 | 8192 | 8316 | 99% |
| mip point, vol point, aniso disabled, 2 ee x | 16384 | 2 | 1 | 1 | 1 | 2 | 4096 | 4156 | 99% |
| mip point, vol point, aniso disabled, 2 ee y | 16384 | 2 | 1 | 1 | 1 | 2 | 4096 | 4156 | 99% |
| mip linear, vol point, aniso disabled, all ee | 16384 | 2 | 2 | 1 | 1 | 4 | 16384 | 16632 | 99% |
| mip linear, vol point, aniso disabled, 2 ee x | 16384 | 2 | 2 | 1 | 1 | 2 | 8192 | 8320 | 98% |
| mip linear, vol point, aniso disabled, 2 ee y | 16384 | 2 | 2 | 1 | 1 | 2 | 8192 | 8316 | 99% |
| mip point, vol linear, aniso disabled, all ee | 16384 | 2 | 1 | 2 | 1 | 4 | 16384 | 16632 | 99% |
| mip point, vol linear, aniso disabled, 2 ee x | 16384 | 2 | 1 | 2 | 1 | 2 | 8192 | 8316 | 99% |
| mip point, vol linear, aniso disabled, 2 ee y | 16384 | 2 | 1 | 2 | 1 | 2 | 8192 | 8316 | 99% |
| mip linear, vol linear, aniso disabled, all ee | 16384 | 2 | 2 | 2 | 1 | 4 | 32768 | 33264 | 99% |
| mip linear, vol linear, aniso disabled, 2 ee x | 16384 | 2 | 2 | 2 | 1 | 2 | 16384 | 16632 | 99% |
| mip linear, vol linear, aniso disabled, 2 ee y | 16384 | 2 | 2 | 2 | 1 | 2 | 16384 | 16632 | 99% |
| mip point, vol point, aniso 2:1, all ee | 16384 | 2 | 1 | 1 | 2 | 4 | 16384 | 16632 | 99% |
| mip point, vol point, aniso 16:1, all ee | 16384 | 2 | 1 | 1 | 16 | 4 | 131072 | 133056 | 99% |
| mip linear, vol point, aniso 2:1, all ee | 16384 | 2 | 2 | 1 | 2 | 4 | 32768 | 33264 | 99% |
| mip linear, vol point, aniso 16:1, all ee | 16384 | 2 | 2 | 1 | 16 | 4 | 262144 | 266112 | 99% |
| mip point, vol linear, aniso 2:1, all ee | 16384 | 2 | 1 | 2 | 2 | 4 | 32768 | 33264 | 99% |
| mip point, vol linear, aniso 16:1, all ee | 16384 | 2 | 1 | 2 | 16 | 4 | 262144 | 266112 | 99% |
| mip linear, vol linear, aniso 2:1, all ee | 16384 | 2 | 2 | 2 | 2 | 4 | 65536 | 66528 | 99% |
| mip linear, vol linear, aniso 16:1, all ee | 16384 | 2 | 2 | 2 | 16 | 4 | 524288 | 532224 | 99% |

## 4.8 FMT_32_32_32_32 Odd-sized Texture Maps

| Mode | Pixels | ch | mip | vol | aniso | align | Ideal | Actual | % |
|---|---|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled, all ee | 16384 | 4 | 1 | 1 | 1 | 4 | 16384 | 16636 | 98% |
| mip point, vol point, aniso disabled, 2 ee x | 16384 | 4 | 1 | 1 | 1 | 2 | 8192 | 8316 | 99% |
| mip point, vol point, aniso disabled, 2 ee y | 16384 | 4 | 1 | 1 | 1 | 2 | 8192 | 8316 | 99% |
| mip linear, vol point, aniso disabled, all ee | 16384 | 4 | 2 | 1 | 1 | 4 | 32768 | 33272 | 98% |
| mip linear, vol point, aniso disabled, 2 ee x | 16384 | 4 | 2 | 1 | 1 | 2 | 16384 | 16636 | 98% |
| mip linear, vol point, aniso disabled, 2 ee y | 16384 | 4 | 2 | 1 | 1 | 2 | 16384 | 16636 | 98% |
| mip point, vol linear, aniso disabled, all ee | 16384 | 4 | 1 | 2 | 1 | 4 | 32768 | 33272 | 98% |
| mip point, vol linear, aniso disabled, 2 ee x | 16384 | 4 | 1 | 2 | 1 | 2 | 16384 | 16636 | 98% |
| mip point, vol linear, aniso disabled, 2 ee y | 16384 | 4 | 1 | 2 | 1 | 2 | 16384 | 16636 | 98% |
| mip linear, vol linear, aniso disabled, all ee | 16384 | 4 | 2 | 2 | 1 | 4 | 65536 | 66544 | 98% |
| mip linear, vol linear, aniso disabled, 2 ee x | 16384 | 4 | 2 | 2 | 1 | 2 | 32768 | 33272 | 98% |
| mip linear, vol linear, aniso disabled, 2 ee y | 16384 | 4 | 2 | 2 | 1 | 2 | 32768 | 33272 | 98% |
| mip point, vol point, aniso 2:1, all ee | 16384 | 4 | 1 | 1 | 2 | 4 | 32768 | 33272 | 98% |
| mip point, vol point, aniso 16:1, all ee | 16384 | 4 | 1 | 1 | 16 | 4 | 262144 | 266176 | 98% |
| mip linear, vol point, aniso 2:1, all ee | 16384 | 4 | 2 | 1 | 2 | 4 | 65536 | 66544 | 98% |
| mip linear, vol point, aniso 16:1, all ee | 16384 | 4 | 2 | 1 | 16 | 4 | 524288 | 532352 | 98% |
| mip point, vol linear, aniso 2:1, all ee | 16384 | 4 | 1 | 2 | 2 | 4 | 65536 | 66544 | 98% |
| mip point, vol linear, aniso 16:1, all ee | 16384 | 4 | 1 | 2 | 16 | 4 | 524288 | 532352 | 98% |
| mip linear, vol linear, aniso 2:1, all ee | 16384 | 4 | 2 | 2 | 2 | 4 | 131072 | 133088 | 98% |
| mip linear, vol linear, aniso 16:1, all ee | 16384 | 4 | 2 | 2 | 16 | 4 | 1048576 | 1064704 | 98% |

## A1. Performance Counters

Though not directly related to block performance, these counters have been included here for completeness.

### A1.1 TPC status

tpc_busy
tpc_stalled
tpc_starved
tpc_working

TPC is busy when anything from the top (SQ_TP ati_dff_in) to the bottom (TP_SP_data_valid ati_dff_out) is busy.

It's hard to visualize what stalled, starved, and working look like when viewing TPC as a whole. There may be starved and stalled situations within TPC, but not from around its periphery. These signals will thus be defined according the following:

TPC is starved when the lower pipe (blend) is idle waiting for data from the TC but the TPC as a whole is busy. This can be true if any part of the upper pipe (lod, aniso) is busy. This is a little misleading, since the upper pipe may be doing useful work.

TPC is stalled when:

- TC is stalling the upper pipe and lower pipe is empty. The second part of that is important, since if the lower pipe is busy and the TC is stalling the upper pipe, TPC as a whole is still doing useful work.
- TPC must wait for the proper phase before writing data to the SP. This one's a tricky one as it really isn't a stall. It's more like a deferring buffer

TPC is doing work when any of the pipelines are busy. This is tpc_busy excluding the fifos. This busy includes:

- input instr/const gatherer - TI_TCG_busy
- walker pipe - TW_TCG_busy
- aligner pipe - AL_TCG_busy
- blend pipe - TB_TCG_busy
- formatter pipe – SP_TCG_busy (??)

The 3 definitions above do not add up cleanly, but hopefully provide some extra useful information. I believe the tpc_busy count to be the only accurate count at this level.

### A1.2 TPC walker status

This set of registers monitors the status of the TPC/TP pipe from the walker state machine up to the SQ_TP interface. Note that the input instr/const gathering logic is separte in the TPC status but is included with this walker state.

tpc_walker_busy
tpc_walker_stalled
tpc_walker_starved
tpc_walker_working

The walker is busy is if any of the following blocks are busy:

- Input instr/const gatherer – TI_TCG_busy
- Walker pipe – TW_TCG_busy
- Walker fifo, state machine and read control – FW_TCG_busy

The walker is stalled if the aligner is not ready to receive, indidated by TW_TA_rtr.

The walker is starved if the walker fifo is not empty , but the input and walker pipes are. This indicates that the walker is still busy trying to send data on to the aligner, but the SQ/SP is not sending more data along and thus starving the input and walker pipes.

The walker is doing useful work if the input and walker pipes are busy.

### A1.3 TPC aligner status

This set of registers monitors the status of the TPC/TP pipe from top of the aligner pipe to the TC and top of the latency fifo.

tpc_aligner_busy
tpc_aligner_stalled
tpc_aligner_starved
tpc_aligner_working

The aligner is busy is if any of the following blocks are busy:

- Aligner pipe – AL_TCG_busy
- Aligner fifo, state machine and read control – FA_TCG_busy

The aligner is stalled if:

- TC not ready to receive, indicated by a flopped version TC_TPC_rtr.
- Blender not ready to receive, indicated by TA_TB_rtr.

The aligner is starved if the aligner fifo is not empty , but the input and walker pipes are. This idicates that the aligner is still busy trying to send data on to the TC and latency FIFOs, but the walker is not sending more data along and thus starving the aligner pipe.

The aligner is doing useful work if the aligner pipe is busy.

### A1.4 TPC blender status

These registers monitor the TPC/TP pipe the top of the latency FIFO down to the bottom of the output FIFO (or bottom of the sp_tp_formatter, TBD).

tpc_blend_busy
tpc_blend_stalled
tpc_blend_starved
tpc_blend_working

The blender is busy is if any of the following blocks are busy:

- Latency fifo (read return FIFO) – FR_TCG_busy
- Blender pipe – TB_TCG_busy
- Output fifo, state machine and read control – FO_TCG_busy
- Formatter pipe – SP_TCG_busy

The blender is unstallable, so we will modify this counter to count the number of cycles a read from the output is delayed because of the incorrect phase.

The blender is starved if the latency fifo is not empty and no data is coming back from the cache. This is when q_rfifo_empty and rfifo_ren are low.

The blender is doing useful work if the blender and formatter pipes are busy.

### A1.5 TPC Counts

0 - # of valid cycles on the output of the walker state machine
1 - # of phases with any 1:1 aniso, bilin, or point sampling
2 - # of phases with any aniso (>1:1 ratio) filtering
3 - # of phases with any mip filtering
4 - # of phases with any volume filtering
5 - # of phases with mip and volume filtering
6 - # of phases with mip and aniso (>1:1 ratio) filtering
7 - # of phases with volume and aniso (>1:1 ratio) filtering
8 - # of phases with 2:1 aniso sampling
9 - # of phases with 4:1 aniso sampling
10 - # of phases with 6:1 aniso sampling
11 - # of phases with 8:1 aniso sampling
12 - # of phases with 10:1 aniso sampling
13 - # of phases with 12:1 aniso sampling
14 - # of phases with 14:1 aniso sampling
15 - # of phases with 16:1 aniso sampling
16 - # of phases with mip, volume and aniso (>1:1 ratio) filtering
17 - # of 2-cycle misaligned phases
18 - # of 4-cycle misaligned phases

### A1.6 TP Counts

0 - # of quads that are point samples (including 1:1 aniso samples that are point sampled)
1 - # of quads that are bilinearly filtered (including 1:1 aniso samples that are bilinearly filtered)
2 - # of quads with any aniso (>1:1 ratio) filtering
3 - # of quads with any mip filtering
4 - # of quads with any volume filtering
5 - # of quads with mip and volume filtering
6 - # of quads with mip and aniso (>1:1 ratio) filtering
7 - # of quads with volume and aniso (>1:1 ratio) filtering
8 - # of quads with 2:1 aniso sampling
9 - # of quads with 4:1 aniso sampling
10 - # of quads with 6:1 aniso sampling
11 - # of quads with 8:1 aniso sampling
12 - # of quads with 10:1 aniso sampling
13 - # of quads with 12:1 aniso sampling
14 - # of quads with 14:1 aniso sampling
15 - # of quads with 16:1 aniso sampling
16 - # of quads with mip, volume and aniso (>1:1 ratio) filtering
17 - # of 2-cycle misaligned quads
18 - # of 4-cycle misaligned quads
19 - no valid pixels in quad
20 - 1 valid pixel in quad
21 - 2 valid pixels in quad
22 - 3 valid pixels in quad
23 - 4 valid pixels in quad

### A1.7 Summary

TPC

0 - # of valid cycles on the output of the walker state machine
1 - # of phases with any 1:1 aniso, bilin, or point sampling
2 - # of phases with any aniso (>1:1 ratio) filtering
3 - # of phases with any mip filtering
4 - # of phases with any volume filtering
5 - # of phases with mip and volume filtering
6 - # of phases with mip and aniso (>1:1 ratio) filtering
7 - # of phases with volume and aniso (>1:1 ratio) filtering
8 - # of phases with 2:1 aniso sampling
9 - # of phases with 4:1 aniso sampling
10 - # of phases with 6:1 aniso sampling
11 - # of phases with 8:1 aniso sampling
12 - # of phases with 10:1 aniso sampling
13 - # of phases with 12:1 aniso sampling
14 - # of phases with 14:1 aniso sampling
15 - # of phases with 16:1 aniso sampling
16 - # of phases with mip, volume and aniso (>1:1 ratio) filtering
17 - # of 2-cycle misaligned phases
18 - # of 4-cycle misaligned phases
24 - tpc_busy
25 - tpc_stalled
26 - tpc_starved
27 - tpc_working
28 - tpc_walker_busy
29 - tpc_walker_stalled
30 - tpc_walker_starved
31 - tpc_walker_working
32 - tpc_aligner_busy
33 - tpc_aligner_stalled
34 - tpc_aligner_starved
35 - tpc_aligner_working
36 - tpc_blend_busy
37 - tpc_blend_stalled
38 - tpc_blend_starved
39 - tpc_blend_working

TP0..3

0 - # of quads that are point samples (including 1:1 aniso samples that are point sampled)
1 - # of quads that are bilinearly filtered (including 1:1 aniso samples that are bilinearly filtered)
2 - # of quads with any aniso (>1:1 ratio) filtering
3 - # of quads with any mip filtering
4 - # of quads with any volume filtering
5 - # of quads with mip and volume filtering
6 - # of quads with mip and aniso (>1:1 ratio) filtering
7 - # of quads with volume and aniso (>1:1 ratio) filtering
8 - # of quads with 2:1 aniso sampling
9 - # of quads with 4:1 aniso sampling
10 - # of quads with 6:1 aniso sampling
11 - # of quads with 8:1 aniso sampling
12 - # of quads with 10:1 aniso sampling
13 - # of quads with 12:1 aniso sampling

14 - # of quads with 14:1 aniso sampling
15 - # of quads with 16:1 aniso sampling
16 - # of quads with mip, volume and aniso (>1:1 ratio) filtering
17 - # of 2-cycle misaligned quads
18 - # of 4-cycle misaligned quads
19 - no valid pixels in quad
20 - 1 valid pixel in quad
21 - 2 valid pixels in quad
22 - 3 valid pixels in quad
23 - 4 valid pixels in quad

### A1.8 How TPC pipe sections map to TP pipe sections

TPC input instr/const gatherer -> TP input instr/const gather
TPC walker pipeline -> TP LOD deriv and aniso pipelines
TPC walker FIFO -> TP LOD and COORD FIFOs
TPC aligner pipeline -> TP addresser pipeline
TPC aligner FIFO -> TP aligner FIFO
TPC latency FIFO -> TP aligner logic (no state), TP_TC interface, TP latency FIFO
TPC blend pipe -> TP blend pipeline (ch_blend, tt, hicolor)
TPC output FIFO -> TP output FIFO
SP_TP_formatter -> SP_TP_formatter

| **Mode** | **Pixels** | **hicolor** | **mip** | **volume** | **aniso** | **Ideal** | **Actual** | **%** |
|---|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |
| mip linear, vol point, aniso disabled | 16384 | 1 | 2 | 1 | 1 | 2048 | 2080 | 98% |
| mip point, vol linear, aniso disabled | 16384 | 1 | 1 | 2 | 1 | 2048 | 2080 | 98% |
| mip point, vol point, aniso 2:1 | 16384 | 1 | 1 | 1 | 2 | 2048 | 2080 | 98% |
| mip point, vol point, aniso 4:1 | 16384 | 1 | 1 | 1 | 4 | 4096 | 4156 | 99% |
| mip point, vol point, aniso 6:1 | 16384 | 1 | 1 | 1 | 6 | 6144 | 6236 | 99% |
| mip point, vol point, aniso 8:1 | 16384 | 1 | 1 | 1 | 8 | 8192 | 8312 | 99% |
| mip point, vol point, aniso 10:1 (a) | 16384 | 1 | 1 | 1 | 10 | 10240 | 10392 | 99% |
| mip point, vol point, aniso 12:1 | 16384 | 1 | 1 | 1 | 12 | 12288 | 12468 | 99% |
| mip point, vol point, aniso 14:1 (a) | 16384 | 1 | 1 | 1 | 14 | 14336 | 14544 | 99% |
| mip point, vol point, aniso 16:1 | 16384 | 1 | 1 | 1 | 16 | 16384 | 16624 | 99% |
| mip linear, vol point, aniso 2:1 | 16384 | 1 | 2 | 1 | 2 | 4096 | 4156 | 99% |
| mip linear, vol point, aniso 4:1 | 16384 | 1 | 2 | 1 | 4 | 8192 | 8312 | 99% |
| mip linear, vol point, aniso 8:1 | 16384 | 1 | 2 | 1 | 8 | 16384 | 16624 | 99% |
| mip linear, vol point, aniso 16:1 | 16384 | 1 | 2 | 1 | 16 | 32768 | 33248 | 99% |
| mip point, vol linear, aniso 2:1 | 16384 | 1 | 1 | 2 | 2 | 4096 | 4156 | 99% |
| mip point, vol linear, aniso 4:1 | 16384 | 1 | 1 | 2 | 4 | 8192 | 8312 | 99% |
| mip point, vol linear, aniso 8:1 | 16384 | 1 | 1 | 2 | 8 | 16384 | 16624 | 99% |
| mip point, vol linear, aniso 16:1 | 16384 | 1 | 1 | 2 | 16 | 32768 | 33248 | 99% |
| mip linear, vol linear, aniso 2:1 | 16384 | 1 | 2 | 2 | 2 | 8192 | 8312 | 99% |
| mip linear, vol linear, aniso 4:1 | 16384 | 1 | 2 | 2 | 4 | 16384 | 16624 | 99% |
| mip linear, vol linear, aniso 8:1 | 16384 | 1 | 2 | 2 | 8 | 32768 | 33248 | 99% |
| mip linear, vol linear, aniso 16:1 | 16384 | 1 | 2 | 2 | 16 | 65536 | 66496 | 99% |
| mip point, vol point, aniso 16:1 clamped to 1:1 (1) | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |
| mip point, vol point, aniso 16:1 clamped to 2:1 (1) | 16384 | 1 | 1 | 1 | 2 | 2048 | 2076 | 99% |
| mip point, vol point, aniso 16:1 clamped to 4:1 (1) | 16384 | 1 | 1 | 1 | 4 | 4096 | 4156 | 99% |
| mip point, vol point, aniso 16:1 clamped to 8:1 (1) | 16384 | 1 | 1 | 1 | 8 | 8192 | 8312 | 99% |
| mip point, vol point, aniso 16:1 clamped to 16:1 (1) | 16384 | 1 | 1 | 1 | 16 | 16384 | 16624 | 99% |
| mip linear frac=0, vol point, aniso disabled | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |
| mip point, vol linear frac=0, aniso disabled (b) | 16384 | 1 | 1 | 2 | 1 | 2048 | 2076 | 99% |
| mip linear tri_juice 1/6 frac=1/8, vol point, aniso disabled | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |
| mip linear tri_juice 1/4 frac=7/32, vol point, aniso disabled | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |
| mip linear tri_juice 3/8 frac=21/64, vol point, aniso disabled | 16384 | 1 | 1 | 1 | 1 | 1024 | 1040 | 98% |

| **Mode** | **Pixels** | **hicolor** | **mip** | **volume** | **aniso** | **Ideal** | **Actual** |
|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled | 16384 | 2 | 1 | 1 | 1 | 2048 | 2080 |
| mip linear, vol point, aniso disabled | 16384 | 2 | 2 | 1 | 1 | 4096 | 4160 |
| mip point, vol linear, aniso disabled | 16384 | 2 | 1 | 2 | 1 | 4096 | 4160 |
| mip point, vol point, aniso 2:1 | 16384 | 2 | 1 | 1 | 2 | 4096 | 4160 |
| mip point, vol point, aniso 4:1 | 16384 | 2 | 1 | 1 | 4 | 8192 | 8316 |
| mip point, vol point, aniso 6:1 | 16384 | 2 | 1 | 1 | 6 | 12288 | 12476 |
| mip point, vol point, aniso 8:1 | 16384 | 2 | 1 | 1 | 8 | 16384 | 16632 |
| mip point, vol point, aniso 10:1 (a) | 16384 | 2 | 1 | 1 | 10 | 20480 | 20792 |
| mip point, vol point, aniso 12:1 | 16384 | 2 | 1 | 1 | 12 | 24576 | 24948 |
| mip point, vol point, aniso 14:1 (a) | 16384 | 2 | 1 | 1 | 14 | 28672 | 29108 |
| mip point, vol point, aniso 16:1 | 16384 | 2 | 1 | 1 | 16 | 32768 | 33264 |
| mip linear, vol point, aniso 2:1 | 16384 | 2 | 2 | 1 | 2 | 8192 | 8316 |
| mip linear, vol point, aniso 4:1 | 16384 | 2 | 2 | 1 | 4 | 16384 | 16632 |
| mip linear, vol point, aniso 8:1 | 16384 | 2 | 2 | 1 | 8 | 32768 | 33264 |
| mip linear, vol point, aniso 16:1 | 16384 | 2 | 2 | 1 | 16 | 65536 | 66528 |
| mip point, vol linear, aniso 2:1 | 16384 | 2 | 1 | 2 | 2 | 8192 | 8316 |
| mip point, vol linear, aniso 4:1 | 16384 | 2 | 1 | 2 | 4 | 16384 | 16632 |
| mip point, vol linear, aniso 8:1 | 16384 | 2 | 1 | 2 | 8 | 32768 | 33264 |
| mip point, vol linear, aniso 16:1 | 16384 | 2 | 1 | 2 | 16 | 65536 | 66528 |
| mip linear, vol linear, aniso 2:1 | 16384 | 2 | 2 | 2 | 2 | 16384 | 16632 |
| mip linear, vol linear, aniso 4:1 | 16384 | 2 | 2 | 2 | 4 | 32768 | 33264 |
| mip linear, vol linear, aniso 8:1 | 16384 | 2 | 2 | 2 | 8 | 65536 | 66528 |
| mip linear, vol linear, aniso 16:1 | 16384 | 2 | 2 | 2 | 16 | 131072 | 133056 |
| mip point, vol point, aniso 16:1 clamped to 1:1 (1) | 16384 | 2 | 1 | 1 | 1 | 2048 | 2080 |
| mip point, vol point, aniso 16:1 clamped to 2:1 (1) | 16384 | 2 | 1 | 1 | 2 | 4096 | 4156 |
| mip point, vol point, aniso 16:1 clamped to 4:1 (1) | 16384 | 2 | 1 | 1 | 4 | 8192 | 8316 |
| mip point, vol point, aniso 16:1 clamped to 8:1 (1) | 16384 | 2 | 1 | 1 | 8 | 16384 | 16632 |
| mip point, vol point, aniso 16:1 clamped to 16:1 (1) | 16384 | 2 | 1 | 1 | 16 | 32768 | 33264 |

| % |
|---|
| 98% |
| 98% |
| 98% |
| 98% |
| 99% |
| 98% |
| 99% |
| 98% |
| 99% |
| 99% |
| 99% |
| 99% |
| 99% |
| 99% |
| 99% |
| 99% |
| 99% |
| 99% |
| 99% |
| 99% |
| 99% |
| 99% |
| 99% |
| 98% |
| 99% |
| 99% |
| 99% |
| 99% |

| **Mode** | **Pixels** | **hicolor** | **mip** | **volume** | **aniso** | **Ideal** | **Actual** |
|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled | 16384 | 4 | 1 | 1 | 1 | 4096 | 4160 |
| mip linear, vol point, aniso disabled | 16384 | 4 | 2 | 1 | 1 | 8192 | 8320 |
| mip point, vol linear, aniso disabled | 16384 | 4 | 1 | 2 | 1 | 8192 | 8320 |
| mip point, vol point, aniso 2:1 | 16384 | 4 | 1 | 1 | 2 | 8192 | 8320 |
| mip point, vol point, aniso 4:1 | 16384 | 4 | 1 | 1 | 4 | 16384 | 16636 |
| mip point, vol point, aniso 6:1 | 16384 | 4 | 1 | 1 | 6 | 24576 | 24956 |
| mip point, vol point, aniso 8:1 | 16384 | 4 | 1 | 1 | 8 | 32768 | 33272 |
| mip point, vol point, aniso 10:1 (a) | 16384 | 4 | 1 | 1 | 10 | 40960 | 41592 |
| mip point, vol point, aniso 12:1 | 16384 | 4 | 1 | 1 | 12 | 49152 | 49908 |
| mip point, vol point, aniso 14:1 (a) | 16384 | 4 | 1 | 1 | 14 | 57344 | 58228 |
| mip point, vol point, aniso 16:1 | 16384 | 4 | 1 | 1 | 16 | 65536 | 66544 |
| mip linear, vol point, aniso 2:1 | 16384 | 4 | 2 | 1 | 2 | 16384 | 16636 |
| mip linear, vol point, aniso 4:1 | 16384 | 4 | 2 | 1 | 4 | 32768 | 33272 |
| mip linear, vol point, aniso 8:1 | 16384 | 4 | 2 | 1 | 8 | 65536 | 66544 |
| mip linear, vol point, aniso 16:1 | 16384 | 4 | 2 | 1 | 16 | 131072 | 133088 |
| mip point, vol linear, aniso 2:1 | 16384 | 4 | 1 | 2 | 2 | 16384 | 16636 |
| mip point, vol linear, aniso 4:1 | 16384 | 4 | 1 | 2 | 4 | 32768 | 33272 |
| mip point, vol linear, aniso 8:1 | 16384 | 4 | 1 | 2 | 8 | 65536 | 66544 |
| mip point, vol linear, aniso 16:1 | 16384 | 4 | 1 | 2 | 16 | 131072 | 133088 |
| mip linear, vol linear, aniso 2:1 | 16384 | 4 | 2 | 2 | 2 | 32768 | 33272 |
| mip linear, vol linear, aniso 4:1 | 16384 | 4 | 2 | 2 | 4 | 65536 | 66544 |
| mip linear, vol linear, aniso 8:1 | 16384 | 4 | 2 | 2 | 8 | 131072 | 133088 |
| mip linear, vol linear, aniso 16:1 | 16384 | 4 | 2 | 2 | 16 | 262144 | 266176 |
| mip point, vol point, aniso 16:1 clamped to 1:1 (1) | 16384 | 4 | 1 | 1 | 1 | 4096 | 4160 |
| mip point, vol point, aniso 16:1 clamped to 2:1 (1) | 16384 | 4 | 1 | 1 | 2 | 8192 | 8316 |
| mip point, vol point, aniso 16:1 clamped to 4:1 (1) | 16384 | 4 | 1 | 1 | 4 | 16384 | 16636 |
| mip point, vol point, aniso 16:1 clamped to 8:1 (1) | 16384 | 4 | 1 | 1 | 8 | 32768 | 33272 |
| mip point, vol point, aniso 16:1 clamped to 16:1 (1) | 16384 | 4 | 1 | 1 | 16 | 65536 | 66544 |

| % |
|---|
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 98% |
| 99% |
| 98% |
| 98% |
| 98% |

| Mode | Pixels | ch | mip | vol | aniso | align | Ideal | Actual | % |
|---|---|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled, all ee | 16384 | 1 | 1 | 1 | 1 | 4 | 4096 | 4156 | 99% |
| mip point, vol point, aniso disabled, 2 ee x | 16384 | 1 | 1 | 1 | 1 | 2 | 2048 | 2076 | 99% |
| mip point, vol point, aniso disabled, 2 ee y | 16384 | 1 | 1 | 1 | 1 | 2 | 2048 | 2076 | 99% |
| mip linear, vol point, aniso disabled, all ee | 16384 | 1 | 2 | 1 | 1 | 4 | 8192 | 8312 | 99% |
| mip linear, vol point, aniso disabled, 2 ee x | 16384 | 1 | 2 | 1 | 1 | 2 | 4096 | 4156 | 99% |
| mip linear, vol point, aniso disabled, 2 ee y | 16384 | 1 | 2 | 1 | 1 | 2 | 4096 | 4156 | 99% |
| mip point, vol linear, aniso disabled, all ee | 16384 | 1 | 1 | 2 | 1 | 4 | 8192 | 8312 | 99% |
| mip point, vol linear, aniso disabled, 2 ee x | 16384 | 1 | 1 | 2 | 1 | 2 | 4096 | 4156 | 99% |
| mip point, vol linear, aniso disabled, 2 ee y | 16384 | 1 | 1 | 2 | 1 | 2 | 4096 | 4156 | 99% |
| mip linear, vol linear, aniso disabled, all ee | 16384 | 1 | 2 | 2 | 1 | 4 | 16384 | 16624 | 99% |
| mip linear, vol linear, aniso disabled, 2 ee x | 16384 | 1 | 2 | 2 | 1 | 2 | 8192 | 8312 | 99% |
| mip linear, vol linear, aniso disabled, 2 ee y | 16384 | 1 | 2 | 2 | 1 | 2 | 8192 | 8312 | 99% |
| mip point, vol point, aniso 2:1, all ee | 16384 | 1 | 1 | 1 | 2 | 4 | 8192 | 8312 | 99% |
| mip point, vol point, aniso 16:1, all ee | 16384 | 1 | 1 | 1 | 16 | 4 | 65536 | 66496 | 99% |
| mip linear, vol point, aniso 2:1, all ee | 16384 | 1 | 2 | 1 | 2 | 4 | 16384 | 16624 | 99% |
| mip linear, vol point, aniso 16:1, all ee | 16384 | 1 | 2 | 1 | 16 | 4 | 131072 | 132992 | 99% |
| mip point, vol linear, aniso 2:1, all ee | 16384 | 1 | 1 | 2 | 2 | 4 | 16384 | 16624 | 99% |
| mip point, vol linear, aniso 16:1, all ee | 16384 | 1 | 1 | 2 | 16 | 4 | 131072 | 132992 | 99% |
| mip linear, vol linear, aniso 2:1, all ee | 16384 | 1 | 2 | 2 | 2 | 4 | 32768 | 33248 | 99% |
| mip linear, vol linear, aniso 16:1, all ee | 16384 | 1 | 2 | 2 | 16 | 4 | 262144 | 265984 | 99% |

| Mode | Pixels | ch | mip | vol | aniso | align | Ideal | Actual | % |
|---|---|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled, all ee | 16384 | 2 | 1 | 1 | 1 | 4 | 8192 | 8316 | 99% |
| mip point, vol point, aniso disabled, 2 ee x | 16384 | 2 | 1 | 1 | 1 | 2 | 4096 | 4156 | 99% |
| mip point, vol point, aniso disabled, 2 ee y | 16384 | 2 | 1 | 1 | 1 | 2 | 4096 | 4156 | 99% |
| mip linear, vol point, aniso disabled, all ee | 16384 | 2 | 2 | 1 | 1 | 4 | 16384 | 16632 | 99% |
| mip linear, vol point, aniso disabled, 2 ee x | 16384 | 2 | 2 | 1 | 1 | 2 | 8192 | 8320 | 98% |
| mip linear, vol point, aniso disabled, 2 ee y | 16384 | 2 | 2 | 1 | 1 | 2 | 8192 | 8316 | 99% |
| mip point, vol linear, aniso disabled, all ee | 16384 | 2 | 1 | 2 | 1 | 4 | 16384 | 16632 | 99% |
| mip point, vol linear, aniso disabled, 2 ee x | 16384 | 2 | 1 | 2 | 1 | 2 | 8192 | 8316 | 99% |
| mip point, vol linear, aniso disabled, 2 ee y | 16384 | 2 | 1 | 2 | 1 | 2 | 8192 | 8316 | 99% |
| mip linear, vol linear, aniso disabled, all ee | 16384 | 2 | 2 | 2 | 1 | 4 | 32768 | 33264 | 99% |
| mip linear, vol linear, aniso disabled, 2 ee x | 16384 | 2 | 2 | 2 | 1 | 2 | 16384 | 16632 | 99% |
| mip linear, vol linear, aniso disabled, 2 ee y | 16384 | 2 | 2 | 2 | 1 | 2 | 16384 | 16632 | 99% |
| mip point, vol point, aniso 2:1, all ee | 16384 | 2 | 1 | 1 | 2 | 4 | 16384 | 16632 | 99% |
| mip point, vol point, aniso 16:1, all ee | 16384 | 2 | 1 | 1 | 16 | 4 | 131072 | 133056 | 99% |
| mip linear, vol point, aniso 2:1, all ee | 16384 | 2 | 2 | 1 | 2 | 4 | 32768 | 33264 | 99% |
| mip linear, vol point, aniso 16:1, all ee | 16384 | 2 | 2 | 1 | 16 | 4 | 262144 | 266112 | 99% |
| mip point, vol linear, aniso 2:1, all ee | 16384 | 2 | 1 | 2 | 2 | 4 | 32768 | 33264 | 99% |
| mip point, vol linear, aniso 16:1, all ee | 16384 | 2 | 1 | 2 | 16 | 4 | 262144 | 266112 | 99% |
| mip linear, vol linear, aniso 2:1, all ee | 16384 | 2 | 2 | 2 | 2 | 4 | 65536 | 66528 | 99% |
| mip linear, vol linear, aniso 16:1, all ee | 16384 | 2 | 2 | 2 | 16 | 4 | 524288 | 532224 | 99% |

| Mode | Pixels | ch | mip | vol | aniso | align | Ideal | Actual | % |
|---|---|---|---|---|---|---|---|---|---|
| mip point, vol point, aniso disabled, all ee | 16384 | 4 | 1 | 1 | 1 | 4 | 16384 | 16636 | 98% |
| mip point, vol point, aniso disabled, 2 ee x | 16384 | 4 | 1 | 1 | 1 | 2 | 8192 | 8316 | 99% |
| mip point, vol point, aniso disabled, 2 ee y | 16384 | 4 | 1 | 1 | 1 | 2 | 8192 | 8316 | 99% |
| mip linear, vol point, aniso disabled, all ee | 16384 | 4 | 2 | 1 | 1 | 4 | 32768 | 33272 | 98% |
| mip linear, vol point, aniso disabled, 2 ee x | 16384 | 4 | 2 | 1 | 1 | 2 | 16384 | 16636 | 98% |
| mip linear, vol point, aniso disabled, 2 ee y | 16384 | 4 | 2 | 1 | 1 | 2 | 16384 | 16636 | 98% |
| mip point, vol linear, aniso disabled, all ee | 16384 | 4 | 1 | 2 | 1 | 4 | 32768 | 33272 | 98% |
| mip point, vol linear, aniso disabled, 2 ee x | 16384 | 4 | 1 | 2 | 1 | 2 | 16384 | 16636 | 98% |
| mip point, vol linear, aniso disabled, 2 ee y | 16384 | 4 | 1 | 2 | 1 | 2 | 16384 | 16636 | 98% |
| mip linear, vol linear, aniso disabled, all ee | 16384 | 4 | 2 | 2 | 1 | 4 | 65536 | 66544 | 98% |
| mip linear, vol linear, aniso disabled, 2 ee x | 16384 | 4 | 2 | 2 | 1 | 2 | 32768 | 33272 | 98% |
| mip linear, vol linear, aniso disabled, 2 ee y | 16384 | 4 | 2 | 2 | 1 | 2 | 32768 | 33272 | 98% |
| mip point, vol point, aniso 2:1, all ee | 16384 | 4 | 1 | 1 | 2 | 4 | 32768 | 33272 | 98% |
| mip point, vol point, aniso 16:1, all ee | 16384 | 4 | 1 | 1 | 16 | 4 | 262144 | 266176 | 98% |
| mip linear, vol point, aniso 2:1, all ee | 16384 | 4 | 2 | 1 | 2 | 4 | 65536 | 66544 | 98% |
| mip linear, vol point, aniso 16:1, all ee | 16384 | 4 | 2 | 1 | 16 | 4 | 524288 | 532352 | 98% |
| mip point, vol linear, aniso 2:1, all ee | 16384 | 4 | 1 | 2 | 2 | 4 | 65536 | 66544 | 98% |
| mip point, vol linear, aniso 16:1, all ee | 16384 | 4 | 1 | 2 | 16 | 4 | 524288 | 532352 | 98% |
| mip linear, vol linear, aniso 2:1, all ee | 16384 | 4 | 2 | 2 | 2 | 4 | 131072 | 133088 | 98% |
| mip linear, vol linear, aniso 16:1, all ee | 16384 | 4 | 2 | 2 | 16 | 4 | 1048576 | 1064704 | 98% |

TP Debug Registers

## 1. Overview

The following is an overview of the TP registers space.

0x0000..0x0007 – TP/TC functional
0x0008..0x000b – TCR debug/extra
0x000c..0x000f – TCF debug/extra

0x0010..0x0013 – Unused

0x0014..0x002b – TCR performance
0x002c..0x005b – TCF performance
0x005c..0x005f – TP/TC clock gating

0x0060..0x0067 – Unused
0x0060 TPC_DEBUG0
0x0064 TPC_DEBUG1

0x0068..0x006b – TP functional

0x006c..0x006f – TP debug/extra

0x0070..0x00ff – TP functional/performance

0x0074..0x00d3 – TP functional/performance

0x0100..0x011b – VC functional
0x011c..0x014f – VC debug

0x0150..0x0167 – TCM performance

//0x0200..0x029f – TP debug regs
// 0x0200..0x21f – TPC debug regs
// 0x0220..0x23f – TP0 debug regs
// 0x0240..0x25f – TP1 debug regs
// 0x0260..0x27f – TP2 debug regs
// 0x0280..0x29f – TP3 debug regs

0x0300..0x035f – TC debug regs
0x0380..0x03bf – TC debug regs

There is a large region of free space at 0x2** where a number of debug registers can be added. However, trying to fill this area with TPC/TP signals is actually hard, perhaps overkill. The following sections define TP/TPC debug registers withg minimal additional register locations, using

## 2. TPC

```
TPC_CNTL_STATUS              <TPDEC:0x0004> 32 {
  TPC_INPUT_BUSY         0    NUM R;
  TPC_TC_FIFO_BUSY       1    NUM R; // includes fifo empty
  TPC_STATE_FIFO_BUSY    2    NUM R; // includes fifo empty
  TPC_FETCH_FIFO_BUSY    3    NUM R; // includes fifo empty
  TPC_WALKER_PIPE_BUSY   4    NUM R;
```

```
  TPC_WALK_FIFO_BUSY     5    NUM R;
  TPC_WALKER_BUSY        6    NUM R;
  TPC_ALIGNER_PIPE_BUSY  8    NUM R;
  TPC_ALIGN_FIFO_BUSY    9    NUM R; // includes fifo empty
  TPC_ALIGNER_BUSY      10    NUM R;
  TPC_RR_FIFO_BUSY      12    NUM R; // includes fifo empty
  TPC_BLEND_PIPE_BUSY   13    NUM R;
  TPC_OUT_FIFO_BUSY     14    NUM R; // includes fifo empty
  TPC_BLEND_BUSY        15    NUM R;
  TF_TW_RTS             16    NUM R; // DEBUG field
  TF_TW_STATE_RTS       17    NUM R; // DEBUG field
  TF_TW_RTR             19    NUM R; // DEBUG field
  TW_TA_RTS             20    NUM R; // DEBUG field
  TW_TA_TT_RTS          21    NUM R; // DEBUG field
  TW_TA_LAST_RTS        22    NUM R; // DEBUG field
  TW_TA_RTR             23    NUM R; // DEBUG field
  TA_TB_RTS             24    NUM R; // DEBUG field
  TA_TB_TT_RTS          25    NUM R; // DEBUG field
  TA_TB_RTR             27    NUM R; // DEBUG field
  TA_TF_RTS             28    NUM R; // DEBUG field
  TA_TF_TC_FIFO_REN     29    NUM R; // DEBUG field
  TP_SQ_dec             30    NUM R; // DEBUG field
  TPC_BUSY              31    NUM R;
};

TPC_DEBUG_0                  <TPDEC:0x006c> 32 {
  LOD_CNTL              1:0   NUM R; // DEBUG field
  IC_CTR                3:2   NUM R; // DEBUG field
  WALKER_CNTL           7:4   NUM R; // DEBUG field
  ALIGNER_CNTL          10:8  NUM R; // DEBUG field
  PREV_TC_STATE_VALID   12    NUM R; // q_tfifo_q_valid from tpc_fifos.v
  WALKER_STATE          25:16 NUM R; // DEBUG field
  ALIGNER_STATE         27:26 NUM R; // DEBUG field
  CLK_GATE_OVERRIDE     28     NUM R;
  REG_CLK_EN            29     NUM R;
  TPC_CLK_EN            30     NUM R;
  SQ_TP_WAKEUP          31     NUM R;
};

TPC_DEBUG_1                  <TPDEC:0x0070> 32 {
};
```

## 3. TP

```
TP@_CNTL_STATUS              <TPDEC:0x0000> 32 {
  TP_INPUT_BUSY          0    NUM R;
  TP_LOD_BUSY            1    NUM R;
  TP_LOD_FIFO_BUSY       2    NUM R;
  TP_ADDR_BUSY           3    NUM R;
  TP_ALIGN_FIFO_BUSY     4    NUM R;
  TP_ALIGNER_BUSY        5    NUM R;
  TP_TC_FIFO_BUSY        6    NUM R;
  TP_RR_FIFO_BUSY        7    NUM R;
  TP_FETCH_BUSY          8    NUM R;
  TP_CH_BLEND_BUSY       9    NUM R;
  TP_TT_BUSY            10    NUM R;
  TP_HICOLOR_BUSY       11    NUM R;
  TP_BLEND_BUSY         12    NUM R;
  TP_OUT_FIFO_BUSY      13    NUM R;
  TP_OUTPUT_BUSY        14    NUM R;
  IN_LC_RTS             16    NUM R;
  LC_LA_RTS             17    NUM R;
  LA_FL_RTS             18    NUM R;
  FL_TA_RTS             19    NUM R;
  TA_FA_RTS             20    NUM R;
  TA_FA_TT_RTS          21    NUM R;
  FA_AL_RTS             22    NUM R;
  FA_AL_TT_RTS          23    NUM R;
  AL_TF_RTS             24    NUM R;
```

```
  AL_TF_TT_RTS          25    NUM R;
  TF_TB_RTS             26    NUM R;
  TF_TB_TT_RTS          27    NUM R;
  TB_TT_RTS             28    NUM R;
  TB_TT_TT_RESET        29    NUM R;
  TB_TO_RTS             30    NUM R;
  TP_BUSY               31    NUM R;
};

TP@_DEBUG                    <TPDEC:0x0000> 32 {
  Q_LOD_CNTL            1:0   NUM R; // DEBUG field
  CLK_GATE_OVERRIDE     2     NUM R;
  Q_SQ_TP_WAKEUP        3     NUM R;
  FL_TA_ADDRESSER_CNTL  20:4  NUM R; // DEBUG field
  REG_CLK_EN            21    NUM R;
  PERF_CLK_EN           22    NUM R;
  TP_CLK_EN             23    NUM R;
  Q_WALKER_CNTL         27:24 NUM R; // DEBUG field
  Q_ALIGNER_CNTL        30:28 NUM R; // DEBUG field
};
```