Programming the Hawaii Parallel Computer

Richard Halverson, Jr.
Art Lew

University of Hawaii at Manoa
Honolulu, HI

Abstract

A new application of field programmable gate-arrays is featured in the prototype of the University of Hawaii parallel computer (HPC). User programs are compiled and then executed partly or completely in one or more field programmable gate-arrays (FPGAs). Some compile-for-FPGA systems have yet to effectively implement full high-level language loop constructs. In this paper we show how the conditional logic used for generating jump addresses can be consolidated into one jump address calculation and computed in a FPGA. In addition, we show how this coprocessing can be easily added to reduce the number of memory transactions and cycles it takes to execute a program. An overview of the HPC shared memory architecture is presented. A shortest path programming example begins by describing the steps for automatically generating the FPGA source code and concludes with the results of a load-store analysis comparing execution with and without the FPGA.

1. Introduction

The HPC prototype is a 4" by 6" five slot platform containing its own microprocessor bus into which processor/computers can be plugged. The system was built for demonstration purposes only, at an absolute minimal cost. One slot is reserved for the main processor board, which is an Intel MCS-51 8031 microprocessor with bus master logic and an RS-232 port for connecting to a host IBM PC. Subordinate processors plug in and interface through an expandable two-port RAM on each processor board, which allows the system to function like a cache-only shared memory multiprocessor. Present hardware allows configurations of up to three parallel 8031 microprocessors and one XILINX board. The XILINX board contains five XILINX 3090 FPGAs and five 8K-byte static RAMs. Parallel programs can be written for execution (a) exclusively on the 8031 array in SIMD fashion, (b) exclusively on the XILINX board, and (c) partly on the 8031s and partly on the XILINX board.

One difficulty in executing programs completely in reprogrammable gate arrays is handling looping. Many compile-for-FPGA projects have made great progress towards synthesizing logic from C-like blocks within loops of user programs but much less progress towards implementing *all* varieties of looping mechanisms found in today's high level programming languages. Since FPGAs still offer quite minimal reprogrammable gate counts, the added parallel processing with the custom coprocessor approach is still necessarily fine-grained. One problem with these fine-grained systems is that the resulting program often requires an increase in the number of transactions across the system bus because operands still must be loaded and retrieved between the coprocessor and the main processor.

For example, the PRISM-II platform contains an Am29050 main processor with slots for several triple XILINX 4010 boards for custom coprocessing. So far from [1], they have reported quite good results on "single-pass" functions without loops. Single pass functions pose a problem because the main processor must transfer the operands and results back and forth from memory which may increase memory transactions overall. A goal of PRISM-II is to implement all the loop constructs of C which will increase the grain size and surely reduce overall the number of memory transactions.

A Reconfigurable Processor Unit (RPU) described in [2] is an array of reprogrammable FPGAs attached to a memory. Their overall goal of compiling a subset of C into FPGA code appears similar to PRISM's but they instead seem to have focused on implementing specific parallel models with limited and specialized constructs for looping. Unlike PRISM, however, it appears that a RPU is capable of fetching its own data from memory and writing results back which eliminates any extra load-store transactions by the main processor. The RPU as described appears most similar to our XILINX board.

Splash 2 contains one or more boards each with an array of 16 well connected XILINX 4010 chips [3]. The architecture does an excellent job supporting pipelined and SIMD processor configurations. Splash 2, for example, can be programmed in dbC, which is a superset

of C used on other SIMD computers. The dbC preprocessor produces C that runs on the Sun and VHDL which define SIMD processors with an instruction set tailored to the application, one or more of which fit into each XILINX chip. When the actual program executes, looping is still handled in the Sun, which transmits SIMD instructions to the Splash 2 board(s).

An Anyboard [4] is a six XILINX chip highly configurable board which plugs into an IBM PC. Since Anyboard is for prototyping hardware, their SOLDER language (similar to C) does provide if-then constructs but other program control constructs of C would have limited value. This is because when programming in the Anyboard environment, the user thinks in terms of designing hardware whereas in the other compile-to-FPGA projects (including ours), the goal is to translate high level language programs (where the user is thinking about writing software). Anyboard's design mapping tool for partitioning a design across many chips, however, would be useful in *any* compile-to-FPGA project where a compiled function may consume more than one FPGA.

Chameleon is a workstation [5] based on LSI Logic's LR33000 32 bit RISC processor that has a Configurable Array Logic (CAL) array of more than 6,000 gates attached to the system bus. The CAL array can be configured as a coprocessor with its own memory and I/O. The Debora language used to program the logic array is C like, but intended for describing the state transitions of sequential logic. All statements execute in parallel except those "guarded" using the IF construct. Except for the IF statement, there are no other traditional language constructs for defining control flow.

One way to use the XILINX board on a HPC system is for it to compute jump addresses from the variable operands of conditional expressions in a program. This can be performed mechanically by first representing the program as a decision table. The computations are boolean in nature and implementing these operations in an FPGA is straightforward. By connecting the FPGA directly to the system bus to allow registers to capture operands *as they are written to memory* allows operands to be loaded into the FPGA coprocessor with no extra loads or stores by the main processor.

This paper describes how the looping control for any high level language program can be implemented totally in a reprogrammable FPGA to increase performance and reduce processor bus transactions. Section 2 begins by reviewing decision tables. Section 3 describes the HPC architecture and its execution model. Section 4 explains the XILINX chip interface to the system bus and how they are programmed. Section 5 introduces the shortest path program which we hand compile into MCS-51 processor code and a PALASM definition source file, the latter of which will be compiled using the XILINX tools. Section 6 shows the results the XILINX compilation. Section 7 gives the results of a load-store analysis comparing optimum processor load-store counts with and without the FPGA. Section 8 concludes with our current and future research plans.

2. Decision Tables

The decision table is used as the high-level programming language to compile because (a) it simplifies loop address computation for FPGA implementation, (b) it is in fact more expressive than conventional imperative languages, and (c) it is general purpose, in that it has been shown previously that any computer program can be expressed in a decision table form [6].

As Figure 1 illustrates, a decision table consists of four quadrants. The upper left contains condition stubs, which are expressions that can be evaluated all at once. The upper right quadrant lists the condition entries, which define columns of possible expression result combinations. Multiple 'T' entries in a column indicate logically ANDed condition stubs which must be true for the "rule" (column) to fire. The lower right quadrant contains the action entries that indicate row by row with X's, which action stub statements (in the lower left quadrant) are to be executed when the rule fires. Note that the right half of the table (the entry table) is simply an AND-OR array, containing boolean inputs and outputs. The process of translating any program (i.e., flowchart) into a decision table is mechanical and explained in [6].

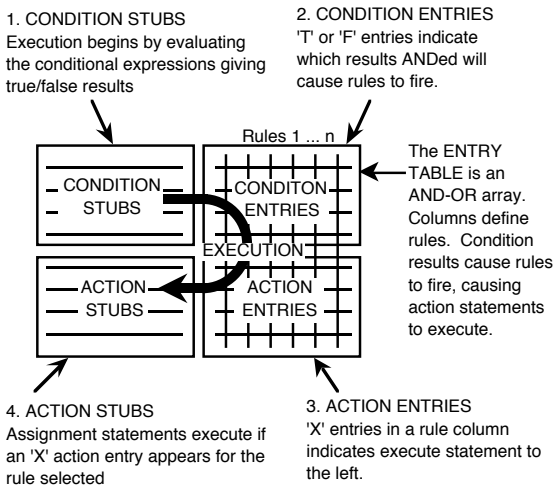

Figure 1. Decision Table

A decision table program executes by first evaluating all the condition stubs simultaneously. The results feed the entry table logic which selects a unique rule. This rule selection will be performed completely in the FPGA, with no processor intervention. The selection of the rule defines a program entry point address for the

processor. There, code for all the action stubs in a selected entry column is executed. The whole process repeats (starting with the reevaluation of the condition stubs) until a selected rule causes the program to terminate. A special loop variable *lambda* helps manage the iteration process. An example of a Pascal program translated into a decision table is shown in Section 5.

3. HPC Architecture

The Hawaii Parallel Computer is a type of cache-only shared memory multiprocessor. Processors may be heterogeneous. As shown in Figure 2, the architecture can consist of p processors, designated μP_0 through μP_{p-1}. Each is attached to a two-port memory, which can be read and written asynchronously by the main CPU (μP_0) on one side, and the single subordinate processor on the other.

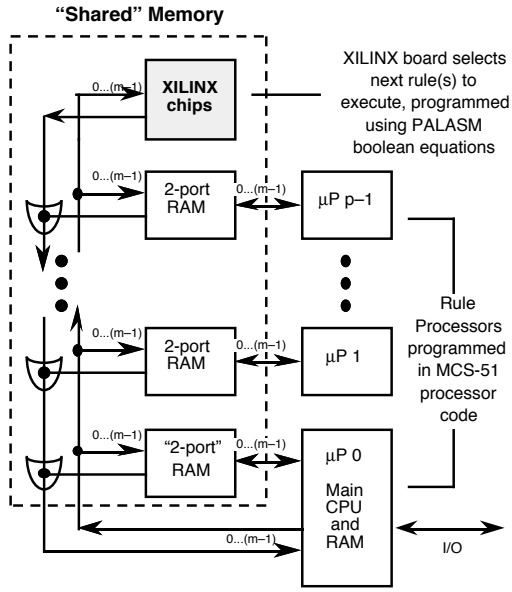

Figure 2. HPC Architecture

All the RAMs map into the same address space on the main CPU side (0..m–1) so the data can be written to all RAMs simultaneously. (Each RAM can also be addressed separately.) The RAM outputs on the main CPU side are wire-ORed, so as long as subordinate processors write to exclusive locations, the main CPU will be able to read subordinate processor output, without having to keep track of which processor wrote it. Although this means that each processor has free read-write capability to and from the RAM, interprocessor communication requires intervention by the main CPU. Before one processor can read what another one wrote, the main CPU must read and re-write (i.e., "refresh") the location, so *all* RAMs contain the same information (at that location).

Figure 3 illustrates the HPC decision table execution model. A rule is provided by the FPGA chip which is "refreshed" to be visible to all processors. If a processor has code to execute for that particular rule, it does so, independently of other processors. When the processor is complete, it waits for the next rule to be issued.

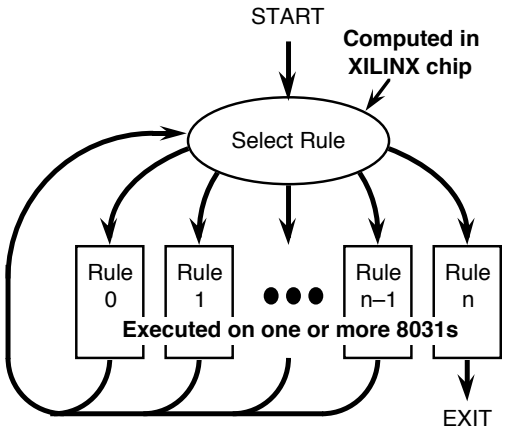

Figure 3. HPC Execution Model

As the model illustrates, more than one processor can execute in parallel. This allows the HPC to support research in adapting this model to shared memory multi-microprocessor programming. In the shortest path example below, however, only one processor will be necessary.

4. XILINX Board Programming

Eventually, the programs which configure the FPGAs will be produced automatically by compiler from a high-level programming language, therefore it was best to select a text based boolean equation method for defining the FPGA logic. Several hardware description languages exist for VLSI designs. We chose PALASM because it is a flexible yet simple language which was originally developed back in the 1970's, for designing field programmable medium scale integrated circuit chips. In order to provide a standard PALASM program format for different chip sizes and pin configurations, a FPGA interface circuit "shell" can be used as illustrated in Figure 4. This interface circuit contains all the specific pin assignment details for a particular FPGA part and its interconnect with the other circuit components. Within it is defined a PALASM module which contains the program for latching data when written, computing the expressions and multiplexing the results out. The PALASM macro with pins defined as shown uses the PALASM ".PDS" file format. Address, control, data in and data out pin names are predefined, in order, using the

same names as in the outer schematic. Following the EQUATIONS statement appears the compiler generated boolean equations which define the input registers, combinational logic for computing the expressions and output multiplexers for the particular user program. Tools for programming XILINX chips are described in more detail in [7].

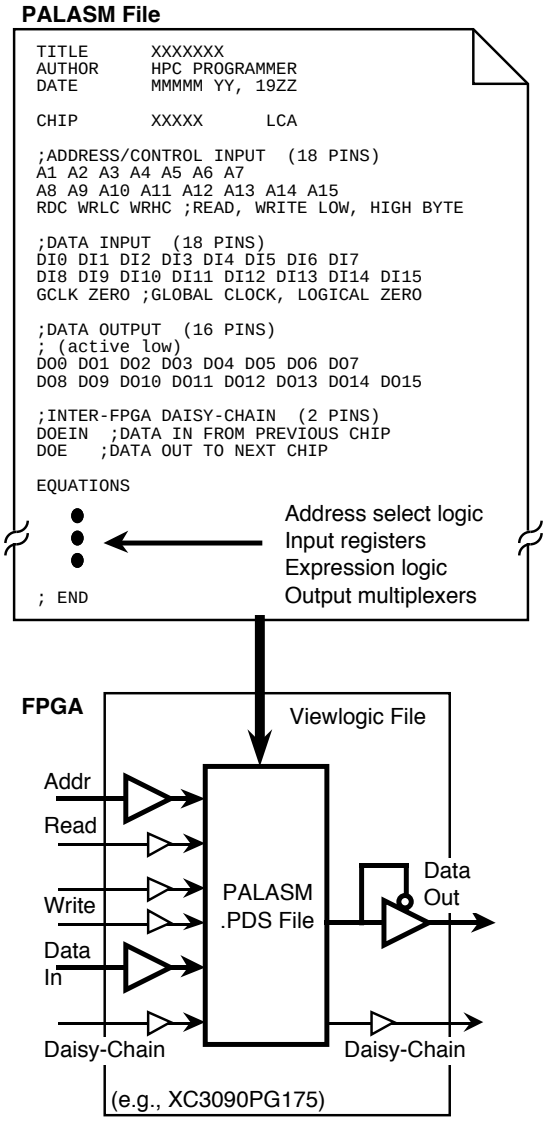

Figure 4. Programming the FPGA

5. Example: Shortest Path Program

To illustrate program compilation and execution of a decision table, a shortest path program which first appeared in [8] will be used. Figure 5 shows a Pascal version of a shortest path algorithm for a topologically sorted graph with distances stored in a two dimensional array *d*. As we see, the first two statements after the begin statement will be executed once. The body of the outer while loop (including the comparison of *i* with 1) will be executed $n-1$ times. The inner while loop executes n-1 times by being inside the outer loop, times $n/2$ more times. Within the most inner loop, the if test and the *j* increment always occurs with the body of the if-then executing depending on characteristics of the data.

Figure 6 shows an equivalent decision table [6]. Rule 0 executes once at the beginning, which sets *lambda*=1. In the second pass, with *lambda*=1, either Rule 1 or 6 will execute, depending on if $i \geq 1$. Rule 1 will execute $n-1$ times, each time setting *lambda*=2. Rule 6 will execute once at the end. When *lambda*=2, Rules 2 and 5 compete, depending on if $j \leq n$. Rule 2 executes $n(n-1)/2$ times while Rule 5 executes $n-1$ times. If $j \leq n$ and *lambda*=2, then lambda is set to 3 causing Rule 3 or 4 to execute next. In the worst case, $d[i,j]+f[j]<min$ causes Rule 3 to execute $n(n-1)/2$ times with Rule 4 executing not at all. In the best case, Rule 3 executes $n(n\text{-}1)/2\text{-}(n\text{-}1)$ times while Rule 4 executes n-1 times. These iteration equations for the loops in Pascal and rules in the decision table will be used for the load store analysis described in section 7.

6. Compiling

Figure 7 contains the memory map and a flow chart showing assignment statements which must be compiled into MCS-51 assembly language. The *d* array is allocated starting at location 0000. The *f* and *t* arrays are above that at 3200 and 32A0. The scalar variables are allocated starting at 3340 with *n*, followed by *i*, *j*, *min*, *ptr*, *lambda*, *Rule•3*, "*d*[*i*,*j*]" and "*f*[*j*]". "*d*[*i*,*j*]" and "*f*[*j*]" are specially allocated locations which hold the value of *d*[*i*,*j*] and *f*[*j*] depending on the current values of *i* and *j*. They are required because they are needed in the FPGA calculation of the rule, as are all variables marked with an asterisk (*). These variables will be latched in registers in the FPGA whenever written. The carrot (^) indicates an output of the FPGA, which in this case is the selected next rule to execute multiplied by 3, which will serve as an offset into a jump table in the 8031 program (as LJMP instructions are 3 bytes long).

The 8031 program selects a rule by reading memory location 334E into the accumulator, which contains the next rule to execute multiplied by 3. It then performs a jump into a table of jump instructions to the code for that rule. When the rule execution is complete, it jumps to the top to select the next rule. As shown in the diagram, executing Rule 6 causes the program to terminate.

Since "*d*[*i*,*j*]" and "*f*[*j*]" are only used when *lambda*=3, then they need to be updated only when *lambda* is set equal to 3. The only time this occurs is in Rule 2, so as

shown in Figure 7, only Rule 2 must contain code to update "*d*[*i*,*j*]" from *d*[*i*,*j*] and "*f*[*j*]" from *f*[*j*]. Rule 3 executes when *lambda*=3 and is the only rule that uses *d*[*i*,*j*] and *f*[*j*], therefore it can use "*d*[*i*,*j*]" and "*f*[*j*]" to save having to compute the address of *d*[*i*,*j*] and *f*[*j*] over again.

```
begin
  f[n] := 0;
  i := n - 1
  while i >= 1 do
    begin
      min := maxint;
      ptr := n + 1;
      j := i + 1;
      while j <= n do
        begin
          if d[i,j] + f[j] < min then
            begin
              min := d[i,j] + f[j]
              ptr := j
            end; {if}
          j := j + 1
        end; {while j <= n}
      f[i] := min;
      t[i] := ptr;
      i := i - 1
    end;  {while i >= 1}
end
```

Figure 5. Pascal Shortest Path

Rule:	0 1 2 3 4 5
lambda =	0 1 2 3 3 2 1
i >= 1	- T - - - - F
j <= n	- - T - - F -
d[i,j]+f[j] < min	- - - T F - -
f[n] := 0	X - - - - - -
i := n - 1	X - - - - - -
min := maxint	- X - - - - -
ptr := n + 1	- X - - - - -
j := i + 1	- X - - - - -
min := d[i,j] + f[j]	- - - X - - -
ptr := j	- - - X - - -
j := j + 1	- - - X X - -
f[i] := min	- - - - - - X -
t[i] := ptr	- - - - - - X -
i := i - 1	- - - - - - X -
exit	- - - - - - - X
lambda :=	1 2 3 2 2 1 %

Figure 6. Decision Table Shortest Path

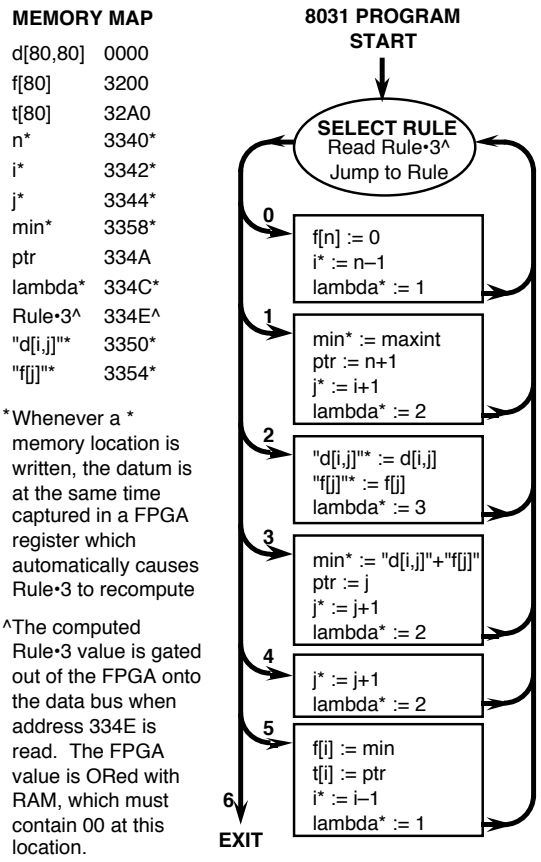

Figure 7. Memory Map and 8031 Program

The calculation of the next rule (and its multiplication by 3) is performed in the XILINX chip. As Figure 7 indicates, within the chip are allocated registers for *lambda*, *i*, *j*, *n*, "*d*[*i*,*j*]", "*f*[*j*]" and *min*. They are clocked from the write signal and enabled from address bus decoders indicating when their respective memory location is being written by the processor. This means that the register locations always contain the most up to date value, without any extra memory transactions being required of the processor. Whenever a register changes, *Rule•3* is automatically recalculated in combinational logic.

As Figure 8 suggests, prespecified "operator macros" are useful for automating the generation of the PALASM code. U1 is "instantiated" from a generic 8-bit in, 1-bit out ">0" macro. U2's output says whether its two inputs are equal. U3's two inputs sum to its output, which feeds U4. The single bit outputs of U1, U2 and U4 are the results of the three condition stubs in the decision table. These three bits, along with the L1 and L0 bits of *lambda* feed U5, which determines which rule to execute next.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.