taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns. We want to resolve these concerns. Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer."Analysts estimate that this recall cost Intel $300 million.

This story brings up a few points for everyone to ponder. How much cheaper would it have been to fix the bug in July 1994? What was the cost to repair the damage to Intel's reputation? And what is the corporate responsibility in disclosing bugs in a product so widely used and relied upon as a microprocessor?

In April 1997 another floating-point bug was revealed in the Pentium Pro and Pentium II microprocessors. When the floating-point-to-integer store instructions (`fist`, `fistp`) encounter a negative floating-point number that is too large to fit in a 16- or 32-bit word after being converted to integer, they set the wrong bit in the FPO status word (precision exception instead of invalid operation exception). To Intel's credit, this time they publicly acknowledged the bug and offered a software patch to get around it—quite a different reaction from what they did in 1994.

4.11 Concluding Remarks

Computer arithmetic is distinguished from paper-and-pencil arithmetic by the constraints of limited precision. This limit may result in invalid operations through calculating numbers larger or smaller than the predefined limits. Such anomalies, called "overflow" or "underflow," may result in exceptions or interrupts, emergency events similar to unplanned subroutine calls. Chapter 5 discusses exceptions in more detail.

Floating-point arithmetic has the added challenge of being an approximation of real numbers, and care needs to be taken to ensure that the computer number selected is the representation closest to the actual number. The challenges of imprecision and limited representation are part of the inspiration for the field of numerical analysis.

Over the years, computer arithmetic has become largely standardized, greatly enhancing the portability of programs. Two's complement binary integer arithmetic and IEEE 754 binary floating-point arithmetic are found in the vast majority of computers sold today. For example, every desktop computer sold since this book was first printed follows these conventions.

A side effect of the stored-program computer is that bit patterns have no inherent meaning. The same bit pattern may represent a signed integer, unsigned integer, floating-point number, instruction, and so on. It is the instruction that operates on the word that determines its meaning.

With the explanation of computer arithmetic in this chapter comes a description of much more of the MIPS instruction set. One point of confusion is the instructions covered in these chapters versus instructions executed by MIPS chips versus the instructions accepted by MIPS assemblers. The next two figures try to make this clear.

Figure 4.52 lists the MIPS instructions covered in Chapters 3 and 4. We call the set of instructions on the left-hand side of the figure the *MIPS core*. The instructions on the right we call the *MIPS arithmetic core*. On the left of Figure 4.53 are the instructions the MIPS processor executes that are not found in Figure 4.52. We call the full set of hardware instructions *MIPS I*. On the right of Figure 4.53 are the instructions accepted by the assembler that are not part of MIPS I. We call this set of instructions *Pseudo MIPS*.

MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
and	and	R	move from system control (EPC)	mfc0	R
and immediate	andi	I	floating-point add single	add.s	R
or	or	R	floating-point add double	add.d	R
or immediate	ori	I	floating-point subtract single	sub.s	R
shift left logical	sll	R	floating-point subtract double	sub.d	R
shift right logical	srl	R	floating-point multiply single	mul.s	R
load upper immediate	lui	I	floating-point multiply double	mul.d	R
load word	lw	I	floating-point divide single	div.s	R
store word	sw	I	floating-point divide double	div.d	R
load byte unsigned	lbu	I	load word to floating-point single	lwc1	I
store byte	sb	I	store word to floating-point single	swc1	I
branch on equal	beq	I	branch on floating-point true	bc1t	I
branch on not equal	bne	I	branch on floating-point false	bc1f	I
jump	j	J	floating-point compare single	c.x.s	R
jump and link	jal	J	(x = eq, neq, lt, le, gt, ge)		
jump register	jr	R	floating-point compare double	c.x.d	R
set less than	slt	R	(x = eq, neq, lt, le, gt, ge)		
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

FIGURE 4.52 The MIPS instruction set covered so far. This book concentrates on the instructions in the left column.

Figure 4.54 gives the popularity of the MIPS instructions for two programs: gcc and spice. All instructions are listed that were responsible for at least 0.5% of the instructions executed. The table following summarizes that information:

INTEL - 1012

Instruction subset	gcc	spice
MIPS core	95%	45%
MIPS arithmetic core	0%	49%
Remaining MIPS I	5%	6%

Note that although programmers and compiler writers may use MIPS I to have a richer menu of options, MIPS core instructions dominate gcc execution, and the integer core plus arithmetic core dominate spice.

Remaining MIPS I	Name	Format	Pseudo MIPS	Name	Format
exclusive or ($rs \oplus rt$)	xor	R	move	move	rd,rs
exclusive or immediate	xori	I	absolute value	abs	rd,rs
nor ($\neg(rs \vee rt)$)	nor	R	not ($\neg rs$)	not	rd,rs
shift right arithmetic	sra	R	negate *(signed or unsigned)*	neg*s*	rd,rs
shift left logical variable	sllv	R	rotate left	rol	rd,rs,rt
shift right logical variable	srlv	R	rotate right	ror	rd,rs,rt
shift right arith. variable	srav	R	mult. & don't check oflw *(signed or uns.)*	mul*s*	rd,rs,rt
			multiply & check oflw *(signed or uns.)*	mulo*s*	rd,rs,rt
move to Hi	mthi	R	divide and check overflow	div	rd,rs,rt
move to Lo	mtlo	R	divide and don't check overflow	divu	rd,rs,rt
load halfword	lh	I	remainder *(signed or unsigned)*	rem*s*	rd,rs,rt
load halfword unsigned	lhu	I	load immediate	li	rd,imm
store halfword	sh	I	load address	la	rd,addr
load word left *(unaligned)*	lwl	I	load double	ld	rd,addr
load word right *(unaligned)*	lwr	I	store double	sd	rd,addr
store word left *(unaligned)*	swl	I	unaligned load word	ulw	rd,addr
store word right *(unaligned)*	swr	I	unaligned store word	usw	rd,addr
branch on less than zero	bltz	I	unaligned load halfword *(signed or uns.)*	ulh*s*	rd,addr
branch on less or equal zero	blez	I	unaligned store halfword	ush	rd,addr
branch on greater than zero	bgtz	I	branch	b	Label
branch on ≥ zero	bgez	I	branch on equal zero	beqz	rs,L
branch on ≥ zero and link	bgezal	I	branch on ≥ *(signed or unsigned)*	bge*s*	rs,rt,L
branch on < zero and link	bltzal	I	branch on > *(signed or unsigned)*	bgt*s*	rs,rt,L
jump and link register	jalr	R	branch on ≤ *(signed or unsigned)*	ble*s*	rs,rt,L
return from exception	rfe	R	branch on < *(signed or unsigned)*	blt*s*	rs,rt,L
system call	syscall	R	set equal	seq	rd,rs,rt
break *(cause exception)*	break	R	set not equal	sne	rd,rs,rt
move from FP to integer	mfc1	R	set greater or equal *(signed or unsigned)*	sge*s*	rd,rs,rt
move to FP from integer	mtc1	R	set greater than *(signed or unsigned)*	sgt*s*	rd,rs,rt
FP move *(s or d)*	mov.*f*	R	set less or equal *(signed or unsigned)*	sle*s*	rd,rs,rt
FP absolute value *(s or d)*	abs.*f*	R	set less than *(signed or unsigned)*	sle*s*	rd,rs,rt
FP negate *(s or d)*	neg.*f*	R	load to floating point *(s or d)*	l.*f*	rd,addr
FP convert *(w, s, or d)*	cvt.*ff*	R	store from floating point *(s or d)*	s.*f*	rd,addr
FP compare un *(s or d)*	c.xn.*f*	R			

FIGURE 4.53 Remaining MIPS I and "Pseudo MIPS" instruction sets. Appendix A describes all these instructions. *f* means single (s) and double precision (d) versions of the floating-point instruction, and *s* means signed and unsigned (u) versions.

Core MIPS	Name	gcc	spice	Arithmetic core + MIPS I	Name	gcc	spice
add	add	0%	0%	FP add double	add.d	0%	4%
add immediate	addi	0%	0%	FP subtract double	sub.d	0%	3%
add unsigned	addu	9%	10%	FP multiply double	mul.d	0%	5%
add immediate unsigned	addiu	17%	1%	FP divide double	div.d	0%	2%
subtract unsigned	subu	0%	1%	load word to FP single	l.s	0%	24%
and	and	1%	0%	store word to FP single	s.s	0%	9%
and immediate	andi	2%	1%	branch on FP true	bc1t	0%	1%
shift left logical	sll	5%	5%	branch on FP false	bc1f	0%	1%
shift right logical	srl	0%	1%	FP compare double	c.x.d	0%	1%
load upper immediate	lui	2%	6%	move to FP	mtc1	0%	2%
load word	lw	21%	7%	move from FP	mfc2	0%	2%
store word	sw	12%	2%	convert float integer	cut	0%	1%
load byte	lb	1%	0%	shift right arithmetic	sra	2%	0%
store byte	sb	1%	0%	load half	lh	1%	0%
branch on equal (zero)	beq	9%	3%	branch less than zero	bltz	1%	0%
branch on not equal (zero)	bne	8%	2%	branch greater or equal zero	bgez	1%	0%
jump and link	jal	1%	1%	branch less or equal zero	blez	0%	1%
jump register	jr	1%	1%				
set less than	slt	2%	0%				
set less than immediate	slti	1%	0%				
set less than unsigned	sltu	1%	0%				
set less than imm. uns.	sltiu	1%	0%				

FIGURE 4.54 The frequency of the MIPS instructions for two programs, gcc and spice. Calculated from "pixie" output of the full MIPS I. (Pixie is an instruction measurement tool from MIPS.) All instructions that accounted for at least 0.5% of the instructions executed in either gcc or spice are included in the table. Thus the integer multiply and divide instructions are not listed because they were responsible for less than 0.5% of the instructions executed. Pseudoinstructions are converted into MIPS I before execution, and hence do not appear here.

For the rest of the book, we concentrate on the MIPS core instructions—the integer instruction set excluding multiply and divide—to make the explanation of computer design easier. As we can see, the MIPS core includes the most popular MIPS instructions, and be assured that understanding a computer that runs the MIPS core will give you sufficient background to understand even more ambitious machines.

INTEL - 1012

4.12 Historical Perspective and Further Reading

Gresham's Law ("Bad money drives out Good") for computers would say, "The Fast drives out the Slow even if the Fast is wrong."

W. Kahan, 1992

At first it may be hard to imagine a subject of less interest than the correctness of computer arithmetic or its accuracy, and harder still to understand why a subject so old and mathematical should be so controversial. Computer arithmetic is as old as computing itself, and some of the subject's earliest notions, like the economical reuse of registers during serial multiplication and division, still command respect today. Maurice Wilkes [1985] recalled a conversation about that notion during his visit to the United States in 1946, before the earliest stored-program machine had been built:

> *. . . a project under von Neumann was to be set up at the Institute of Advanced Studies in Princeton. . . . Goldstine explained to me the principal features of the design, including the device whereby the digits of the multiplier were put into the tail of the accumulator and shifted out as the least significant part of the product was shifted in. I expressed some admiration at the way registers and shifting circuits were arranged . . . and Goldstine remarked that things of that nature came very easily to von Neumann.*

There is no controversy here; it can hardly arise in the context of exact integer arithmetic so long as there is general agreement on what integer the correct result should be. However, as soon as approximate arithmetic enters the picture, so does controversy, as if one person's "negligible" must be another's "everything."

The First Dispute

Floating-point arithmetic kindled disagreement before it was ever built. John von Neumann was aware of Konrad Zuse's proposal for a computer in Germany in 1939 that was never built, probably because the floating point made it appear too complicated to finish before the Germans expected World War II to end. Hence von Neumann refused to include it in the machine he built at Princeton. In an influential report coauthored in 1946 with H. H. Goldstine and A. W. Burks, he gave the arguments for and against floating point. In favor:

> *. . . to retain in a sum or product as many significant digits as possible and . . . to free the human operator from the burden of estimating and inserting into a problem "scale factors"—multiplication constants which serve to keep numbers within the limits of the machine.*

Floating point was excluded for several reasons:

> *There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point, one must waste memory capacity which could otherwise be used for carrying more digits per word. It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.*

The argument seems to be that most bits devoted to exponent fields would be bits wasted. Experience has proved otherwise.

One software approach to accommodate reals without floating-point hardware was called *floating vectors;* the idea was to compute at runtime one scale factor for a whole array of numbers, choosing the scale factor so that the array's biggest number would barely fill its field. By 1951, James H. Wilkinson had used this scheme extensively for matrix computations. The problem proved to be that a program might encounter a very large value, and hence the scale factor must accommodate these rare large numbers. The common numbers would thus have many leading 0s, since all numbers had to use a single scale factor. Accuracy was sacrificed because the least significant bits had to be lost on the right to accommodate leading 0s. This wastage became obvious to practitioners on early machines that displayed all their memory bits as dots on cathode ray tubes (like TV screens) because the loss of precision was visible. Where floating point deserved to be used, no practical alternative existed.

Thus true floating-point hardware became popular because it was useful. By 1957, floating-point hardware was almost ubiquitous. A decimal floating-point unit was available for the IBM 650; and soon the IBM 704, 709, 7090, 7094 . . . series would offer binary floating-point hardware for double as well as single precision.

As a result, everybody had floating point, but every implementation was different.

Diversity versus Portability

Since roundoff introduces some error into almost all floating-point operations, to complain about another bit of error seems picayune. So for 20 years nobody complained much that those operations behaved a little differently on different machines. If software required clever tricks to circumvent those idiosyncrasies and finally deliver results correct in all but the last several bits, such tricks were deemed part of the programmer's art. For a long time, matrix computations mystified most people who had no notion of error analysis; perhaps this continues to be true. That may be why people are still surprised that

INTEL - 1012

numerically stable matrix computations depend upon the quality of arithmetic in so few places, far fewer than are generally supposed. Books by Wilkinson and widely used software packages like Linpack and Eispack sustained a false impression, widespread in the early 1970s, that a modicum of skill sufficed to produce *portable* numerical software.

Portable here means that the software is distributed as source code in some standard language to be compiled and executed on practically any commercially significant machine, and that it will then perform its task as well as any other program performs that task on that machine. Insofar as numerical software has often been thought to consist entirely of machine-independent mathematical formulas, its portability has often been taken for granted; the mistake in that presumption will become clear shortly.

Packages like Linpack and Eispack cost so much to develop—over a hundred dollars per line of Fortran delivered—that they could not have been developed without U.S. government subsidy; their portability was a precondition for that subsidy. But nobody thought to distinguish how various components contributed to their cost. One component was algorithmic—devise an algorithm that deserves to work on at least one computer despite its roundoff and over/underflow limitations. Another component was the software engineering effort required to achieve and confirm portability to the diverse computers commercially significant at the time; this component grew more onerous as ever more diverse floating-point arithmetics blossomed in the 1970s.

And yet scarcely anybody realized how much that diversity inflated the cost of such software packages.

A Backward Step

Early evidence that somewhat different arithmetics could engender grossly different software development costs was presented in 1964. It happened at a meeting of SHARE, the IBM mainframe users' group, at which IBM announced System/360, the successor to the 7094 series. One of the speakers described the tricks he had been forced to devise to achieve a level of quality for the S/360 library that was not quite so high as he had previously achieved for the 7094.

Part of the trouble could have been foretold by von Neumann had he still been alive. In 1948 he and Goldstine had published a lengthy error analysis so difficult and so pessimistic that hardly anybody paid attention to it. It did predict correctly, however, that computations with larger arrays of data would probably fall prey to roundoff more often. IBM S/360s had bigger memories than 7094s, so data arrays could grow bigger, and they did. To make matters worse, the S/360s had narrower single precision words (32 bits versus 36) and used a cruder arithmetic (hexadecimal or base 16 versus binary or base 2) with consequently poorer worst-case precision (21 significant bits versus 27) than

old 7094s. Consequently, software that had almost always provided (barely) satisfactory accuracy on 7094s too often produced inaccurate results when run on S/360s. The quickest way to recover adequate accuracy was to replace old codes' single precision declarations with double precision before recompilation for the S/360. This practice exercised S/360 double precision far more than had been expected.

The early S/360s' worst troubles were caused by lack of a guard digit in double precision. This lack showed up in multiplication as a failure of identities like $1.0 * x = x$ because multiplying x by 1.0 dropped x's last hexadecimal digit (4 bits). Similarly, if x and y were very close but had different exponents, subtraction dropped off the last digit of the smaller operand before computing $x - y$. This last aberration in double precision undermined a precious theorem that single precision then (and now) honored: If $1/2 \le x/y \le 2$, then no rounding error can occur when $x - y$ is computed; it must be computed exactly.

Innumerable computations had benefited from this minor theorem, most often unwittingly, for several decades before its first formal announcement and proof. We had been taking all this stuff for granted.

The identities and theorems about exact relationships that persisted, despite roundoff, with reasonable implementations of approximate arithmetic were not appreciated until they were lost. Previously, all that had been thought to matter were precision (how many significant digits were carried) and range (the spread between over/underflow thresholds). Since the S/360s' double precision had more precision and wider range than the 7094s', software was expected to continue to work at least as well as before. But it didn't.

Programmers who had matured into program managers were appalled at the cost of converting 7094 software to run on S/360s. A small subcommittee of SHARE proposed improvements to the S/360 floating point. This committee was surprised and grateful to get a fair part of what they asked for from IBM, including all-important guard digits. By 1968, these had been retrofitted to S/360s in the field at considerable expense; worse than that was customers' loss of faith in IBM's infallibility (a lesson learned by Intel 30 years later). IBM employees who can remember the incident still shudder.

The People Who Built the Bombs

Seymour Cray was associated for decades with the CDC and Cray computers that were, when he built them, the world's biggest and fastest. He always understood what his customers wanted most: *speed*. And he gave it to them even if, in so doing, he also gave them arithmetics more "interesting" than anyone else's. Among his customers have been the great government laboratories like those at Livermore and Los Alamos, where nuclear weapons were designed. The challenges of "interesting" arithmetics were pretty tame to people who had to overcome Mother Nature's challenges.

INTEL - 1012

Perhaps all of us could learn to live with arithmetic idiosyncrasy if only one computer's idiosyncrasies had to be endured. Instead, when accumulating different computers' different anomalies, software dies the Death of a Thousand Cuts. Here is an example from Cray's machines:

```
if (x == 0.0)    y = 17.0 else y = z/x
```

Could this statement be stopped by a divide-by-zero error? On a CDC 6600 it could. The reason was a conflict between the 6600's adder, where x was compared with 0.0, and the multiplier and divider. The adder's comparison examined x's leading 13 bits, which sufficed to distinguish zero from normal nonzero floating-point numbers x. The multiplier and divider examined only 12 leading bits. Consequently, tiny numbers existed that were nonzero to the adder but zero to the multiplier and divider! To avoid disasters with these tiny numbers, programmers learned to replace statements like the one above by

```
if (1.0*x == 0.0)    y = 17.0 else y = z/x
```

But this statement is unsafe to use in would-be portable software because it malfunctions obscurely on other computers designed by Cray, the ones marketed by Cray Research, Inc. If x is so huge that 2.0 * x would overflow, then 1.0 * x may overflow too! Overflow happens because Cray computers check the product's exponent *before* the product's exponent has been normalized, just to save the delay of a single AND gate.

In case you think the statement above is safe to use now for portable software, since computers of the CDC 6600 era are no longer commercially significant, you should be warned that it can lead to overflow on a Cray computer even if z is almost as tiny as x; the trouble here is that the Cray computes not z/x but z * (1/x), and the reciprocal can overflow even though the desired quotient is unexceptionable. A similar difficulty troubles the Intel i860s used in its massively parallel computers. The would-be programmer of portable code faces countless dilemmas like these whenever trying to program for the full range of existing computers.

Rounding error anomalies that are far worse than the over/underflow anomaly just discussed also affect Cray computers. The worst error comes from the lack of a guard digit in add/subtract, an affliction of IBM S/360s. Further bad luck for software is occasioned by the way Cray economized his multiplier; about one-third of the bits that normal multiplier arrays generate have been left out of his multipliers because they would contribute less than a unit to the last place of the final Cray-rounded product. Consequently, a Cray's multiplier errs by almost a bit more than might have been expected. This error is compounded when division takes three multiplications to improve an approximate reciprocal of the divisor and then multiply the numerator by it. Square root compounds a few more multiplication errors.

The fast way drove out the slow, even though the fast was occasionally slightly wrong.

Making the World Safe for Floating Point, or Vice Versa

William Kahan was an undergraduate at the University of Toronto in 1953 when he learned to program its Ferranti-Manchester Mark-I computer. Because he entered the field early, Kahan became acquainted with a wide range of devices and a large proportion of the personalities active in computing; the numbers of both were small at that time. He has performed computations on slide rules, desktop mechanical calculators, tabletop analog differential analyzers, and so on; he used all but the earliest electronic computers and calculators mentioned in this book.

Kahan's desire to deliver reliable software led to an interest in error analysis that intensified during two years of postdoctoral study in England, where he became acquainted with Wilkinson. In 1960, he resumed teaching at Toronto, where an IBM 7090 had been acquired, and was granted free rein to tinker with its operating system, Fortran compiler, and runtime library. (He denies that he ever came near the 7090 hardware with a soldering iron but admits asking to do so.) One story from that time illuminates how misconceptions and numerical anomalies in computer systems can incur awesome hidden costs.

A graduate student in aeronautical engineering used the 7090 to simulate the wings he was designing for short takeoffs and landings. He knew such a wing would be difficult to control if its characteristics included an abrupt onset of stall, but he thought he could avoid that. His simulations were telling him otherwise. Just to be sure that roundoff was not interfering, he had repeated many of his calculations in double precision and gotten results much like those in single; his wings had stalled abruptly in both precisions. Disheartened, the student gave up.

Meanwhile Kahan replaced IBM's logarithm program (ALOG) with one of his own, which he hoped would provide better accuracy. While testing it, Kahan reran programs using the new version of ALOG. The student's results changed significantly; Kahan approached him to find out what had happened.

The student was puzzled. Much as the student preferred the results produced with the new ALOG—they predicted a gradual stall—he knew they must be wrong because they disagreed with his double precision results. The discrepancy between single and double precision results disappeared a few days later when a new release of IBM's double precision arithmetic software for the 7090 arrived. (The 7090 had no double precision hardware.) He went on to write a thesis about it and to build the wings; they performed as predicted. But that is not the end of the story.

In 1963, the 7090 was replaced by a faster 7094 with double precision floating-point hardware but with otherwise practically the same instruction set as the 7090. Only in double precision and only when using the new hardware did the wing stall abruptly again. A lot of time was spent to find out why. The 7094 hardware turned out, like the superseded 7090 software and the subsequent early S/360s, to lack a guard bit in double precision. Like so many programmers on those machines and on Cray's, the student discovered a trick to

INTEL - 1012

compensate for the lack of a guard digit; he wrote the expression `(0.5 - x) + 0.5` in place of `1.0 - x`. Nowadays we would blush if we had to explain why such a trick might be necessary, but it solved the student's problem.

Meanwhile the lure of California was working on Kahan and his family; they came to Berkeley and he to the University of California. An opportunity presented itself in 1974 when accuracy questions induced Hewlett-Packard's calculator designers to call in a consultant. The consultant was Kahan, and his work dramatically improved the accuracy of HP calculators, but that is another story. Fruitful collaboration with congenial co-workers, however, fortified him for the next and crucial opportunity.

It came in 1976, when John F. Palmer at Intel was empowered to specify the "best possible" floating-point arithmetic for all of Intel's product line. The 8086 was imminent, and an 8087 floating-point coprocessor for the 8086 was contemplated. (A *coprocessor* is simply an additional chip that accelerates a portion of the work of a processor; in this case, it accelerated floating-point computation.)

Palmer had obtained his Ph.D. at Stanford a few years before and knew whom to call for counsel of perfection—Kahan. They put together a design that obviously would have been impossible only a few years earlier and looked not quite possible at the time. But a new Israeli team of Intel employees led by Rafi Navé felt challenged to prove their prowess to Americans and leaped at an opportunity to put something impossible on a chip—the 8087.

By now, floating-point arithmetics that had been merely diverse among mainframes had become chaotic among microprocessors, one of which might be host to a dozen varieties of arithmetic in ROM firmware or software. Robert G. Stewart, an engineer prominent in IEEE activities, got fed up with this anarchy and proposed that the IEEE draft a decent floating-point standard. Simultaneously, word leaked out in Silicon Valley that Intel was going to put on one chip some awesome floating point well beyond anything its competitors had in mind. The competition had to find a way to slow Intel down, so they formed a committee to do what Stewart requested.

Meetings of this committee began in late 1977 with a plethora of competing drafts from innumerable sources and dragged on into 1985 when IEEE Standard 754 for Binary Floating Point was made official. The winning draft was very close to one submitted by Kahan, his student Jerome T. Coonen, and Harold S. Stone, a professor visiting Berkeley at the time. Their draft was based on the Intel design, with Intel's permission of course, as simplified by Coonen. Their harmonious combination of features, almost none of them new, had at the outset attracted more support within the committee and from outside experts like Wilkinson than any other draft, but they had to win nearly unanimous support within the committee to win official IEEE endorsement, and that took time.

The First IEEE 754 Chips

In 1980, Intel became tired of waiting and released the 8087 for use in the IBM PC. The floating-point architecture of the companion 8087 had to be retrofitted into the 8086 opcode space, making it inconvenient to offer two operands per instruction as found in the rest of the 8086. Hence the decision for one operand per instruction using a stack: "The designer's task was to make a Virtue of this Necessity." (Kahan's [1990] history of the stack architecture selection for the 8087 is entertaining reading.)

Rather than the classical stack architecture, which has no provision for avoiding common subexpressions from being pushed and popped from memory into the top of the stack found in registers, Intel tried to combine a flat register file with a stack. The reasoning was that the restriction of the top of stack as one operand was not so bad since it only required the execution of an `FXCH` instruction (which swapped registers) to get the same result as a two-operand instruction, and `FXCH` was much faster than the floating-point operations of the 8087.

Since floating-point expressions are not that complex, Kahan reasoned that eight registers meant that the stack would rarely overflow. Hence he urged that the 8087 use this hybrid scheme with the provision that stack overflow or stack underflow would interrupt the 8086 so that interrupt software could give the illusion to the compiler writer of an unlimited stack for floating-point data.

The Intel 8087 was implemented in Israel, and 7500 miles and 10 time zones made communication difficult from California. According to Palmer and Morse (*The 8087 Primer*, J. Wiley, New York, 1984, p. 93):

> *Unfortunately, nobody tried to write a software stack manager until after the 8087 was built, and by then it was too late; what was too complicated to perform in hardware turned out to be even worse in software. One thing found lacking is the ability to conveniently determine if an invalid operation is indeed due to a stack overflow. . . . Also lacking is the ability to restart the instruction that caused the stack overflow . . .*

The result is that the stack exceptions are too slow to handle in software. As Kahan [1990] says:

> *Consequently, almost all higher-level languages' compilers emit inefficient code for the 80x87 family, degrading the chip's performance by typically 50% with spurious stores and loads necessary simply to preclude stack over/underflow. . . .*
>
> *I still regret that the 8087's stack implementation was not quite so neat as my original intention. . . . If the original design had been realized, compilers today would use the 80x87 and its descendents more efficiently, and Intel's competitors could more easily market faster but compatible 80x87 imitations.*

In 1982, Motorola announced its 68881, which found a place in Sun 3s and Macintosh IIs; Apple had been a supporter of the proposal from the beginning. Another Berkeley graduate student, George S. Taylor, had soon designed a high-speed implementation of the proposed standard for an early superminicomputer (ELXSI 6400). The standard was becoming de facto before its final draft's ink was dry.

An early rush of adoptions gave the computing industry the false impression that IEEE 754, like so many other standards, could be implemented easily by following a standard recipe. Not true. Only the enthusiasm and ingenuity of its early implementors made it look easy.

In fact, to implement IEEE 754 correctly demands extraordinarily diligent attention to detail; to make it run fast demands extraordinarily competent ingenuity of design. Had the industry's engineering managers realized this, they might not have been so quick to affirm that, as a matter of policy, "We conform to all applicable standards."

IEEE 754 Today

Today the computing industry is enmeshed in a host of standards that evolve continuously as technology changes. The floating-point standards IEEE 754/854 (they are practically the same) stand in somewhat splendid isolation only because nobody wishes to repeat the protracted wrangling that surrounded their birth, when, with unprecedented generosity, the representatives of hardware interests acceded to the demands of those few who represented the interests of mathematical and numerical software.

Unfortunately, the compiler-writing community was not represented adequately in the wrangling, and some of the features didn't balance language and compiler issues against other points. That community has been slow to make IEEE 754's unusual features available to the applications programmer. Humane exception handling is one such unusual feature; directed rounding another. Without compiler support, these features have atrophied.

The successful parts of IEEE 754 are that it is a widely implemented standard with a common floating-point format, it requires minimum accuracy to one-half ulp in the least significant bit, and that operations must be commutative.

At present, IEEE 754/854 have been implemented to a considerable degree of fidelity in at least part of the product line of every North American computer manufacturer. The only significant exceptions are the DEC VAX, IBM S/370 descendants, and Cray Research vector supercomputers, and all three are being replaced by compliant machines. Even Cray Research, now a division of Silicon Graphics, announced that successors to the T90 vector computer will conform "to some degree" to ease the transfer of data files and portable software between Crays and the desktop computers through which Cray users have come to access their machines nowadays.

In 1989, the Association for Computing Machinery, acknowledging the benefits conferred upon the computing industry by IEEE 754, honored Kahan with the Turing Award. On accepting it, he thanked his many associates for their diligent support, and his adversaries for their blunders.

So . . . not all errors are bad.

To Probe Further

If you are interested in learning more about floating point, two publications by David Goldberg [1991, 1995] are good starting points; they abound with pointers to further reading. Several of the stories told above come from Kahan [1972, 1983]. The latest word on the state of the art in computer arithmetic is often found in the *Proceedings* of the latest IEEE-sponsored Symposium on Computer Arithmetic, held every two years; the 13th was held in 1997.

Burks, A. W., H. H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," *Report to the U.S. Army Ordnance Dept.*, p. 1; also in *Papers of John von Neumann*, W. Aspray and A. Burks, eds., MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, 97–146, 1987.

This classic paper includes arguments against floating-point hardware.

Goldberg, D. [1991]. "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys* 23(1), 5–48.

Another good introduction to floating-point arithmetic by the same author, this time with emphasis on software.

Goldberg, D. [1995]. "Computer arithmetic," *Appendix A of Computer Architecture: A Quantitative Approach*, second edition, J. L. Hennessy and D. A. Patterson, Morgan Kaufmann Publishers, San Francisco.

A more advanced introduction to integer and floating-point arithmetic, with emphasis on hardware. It covers sections 4.6–4.8 of this book in just 10 pages, leaving another 45 pages for advanced topics.

Kahan, W. [1972]. "A survey of error-analysis," in *Info. Processing 71* (Proc. IFIP Congress 71 in Ljubljana), vol. 2, pp. 1214–39, North-Holland Publishing, Amsterdam.

This survey is a source of stories on the importance of accurate arithmetic.

Kahan, W. [1983]. "Mathematics written in sand," *Proc. Amer. Stat. Assoc. Joint Summer Meetings of 1983, Statistical Computing Section*, pp. 12–26.

The title refers to silicon and is another source of stories illustrating the importance of accurate arithmetic.

Kahan, W. [1990]. "On the advantage of the 8087's stack," unpublished course notes, Computer Science Division, University of California at Berkeley.

What the 8087 floating-point architecture could have been.

INTEL - 1012

Kahan, W. [1997]. Available via a link to Kahan's homepage at *www.mkp.com/books_catalog/cod/links.htm*.

A collection of memos related to floating point, including "Beastly Numbers" (another less famous Pentium bug),"Notes on the IEEE Floating Point Arithmetic" (including comments on how some features are atrophying), and "The Baleful Effects of Computing Benchmarks" (on the unhealthy preoccupation on speed versus correctness, accuracy, ease of use, flexibility, . . .).

Koren, I. [1993]. *Computer Arithmetic Algorithms*, Prentice Hall, Englewood Cliffs, NJ.

A textbook aimed at seniors and first-year graduate students that explains fundamental principles of basic arithmetic, as well as complex operations such as logarithmic and trigonometric functions.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

This computer pioneer's recollections include the derivation of the standard hardware for multiply and divide developed by von Neumann.

4.13 Key Terms

These terms reflect the key ideas in the chapter. Check the Glossary for definitions of the terms you are unsure of.

AND gate	floating point	round
AND operation	guard	scientific notation
arithmetic logic unit (ALU)	hexadecimal	significand
biased notation	least significant bit	single precision
Booth's algorithm	most significant bit	sticky bit
divisor	normalized	underflow
double precision	overflow	units in the last place (ulp)
exclusive OR gate	quotient	
exponent	remainder	

4.14 Exercises

Never give in, never give in, never, never, never—in nothing, great or small, large or petty—never give in.

Winston Churchill, address at Harrow School, 1941

4.1 [3] <§4.2> Convert 512_{ten} into a 32-bit two's complement binary number.

4.2 [3] <§4.2> Convert $-1{,}023_{ten}$ into a 32-bit two's complement binary number.

4.3 [5] <§4.2> Convert $-4{,}000{,}000_{ten}$ into a 32-bit two's complement binary number.

4.4 [5] <§4.2> What decimal number does this two's complement binary number represent: $1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0000\ 1100_{two}$?

4.5 [5] <§4.2> What decimal number does this two's complement binary number represent: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$?

4.6 [5] <§4.2> What decimal number does this two's complement binary number represent: $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$?

4.7 [5] <§4.2> What binary number does this hexadecimal number represent: $7fff\ fffa_{hex}$? What decimal number does it represent?

4.8 [5] <§4.2> What hexadecimal number does this binary number represent: $1100\ 1010\ 1111\ 1110\ 1111\ 1010\ 1100\ 1110_{two}$?

4.9 [5] <§4.2> Why doesn't MIPS have a subtract immediate instruction?

4.10 [10] <§4.2> Find the shortest sequence of MIPS instructions to determine the absolute value of a two's complement integer. Convert this instruction (accepted by the MIPS assembler):

```
abs    $t2,$t3
```

This instruction means that register `$t2` has a copy of register `$t3` if register `$t3` is positive, and the two's complement of register `$t3` if `$t3` is negative. (Hint: It can be done with three instructions.)

4.11 [10] <§4.2> Two friends, Harry and David, are arguing. Harry says, "All integers greater than zero and exactly divisible by six have exactly two 1s in their binary representation." David disagrees. He says, "No, but all such numbers have an even number of 1s in their representation." Do you agree with Harry or with David, or with neither? (Hint: Look for counterexamples.)

4.12 [15] <§4.4> Consider the following code used to implement the instruction

```
sllv $s0, $s1, $s2
```

which uses the least significant 5 bits of the value in register `$s2` to specify the amount register `$s1` should be shifted left:

```
         .data
mask:    .word  0xfffff83f
         .text
start:   lw     $t0, mask
         lw     $s0, shifter
         and    $s0,$s0,$t0
         andi   $s2,$s2,0x1f
         sll    $s2,$s2,6
         or     $s0,$s0,$s2
         sw     $s0, shifter
shifter: sll    $s0,$s1,0
```

INTEL - 1012

Add comments to the code and write a paragraph describing how it works. Note that the two `lw` instructions are pseudoinstructions that use a label to specify a memory address that contains the word of data to be loaded. Why do you suppose that writing "self-modifying code" such as this is a bad idea (and oftentimes not actually allowed)?

4.13 [10] <§4.2> If `A` is a 32-bit address, typically an instruction sequence such as

```
lui $t0, A_upper
ori $t0, $t0, A_lower
lw $s0, 0($t0)
```

can be used to load the word at `A` into a register (in this case, `$s0`). Consider the following alternative, which is more efficient:

```
lui $t0, A_upper_adjusted
lw $s0, A_lower($t0)
```

Describe how `A_upper` is adjusted to allow this simpler code to work. (Hint: `A_upper` needs to be adjusted because `A_lower` will be sign-extended.)

4.14 [15] <§§3.4, 4.2, 4.8> The Big Picture on page 299 mentions that bits have no inherent meaning. Given the bit pattern:

1000 1111 1110 1111 1100 0000 0000 0000

what does it represent, assuming that it is

a. a two's complement integer?

b. an unsigned integer?

c. a single precision floating-point number?

d. a MIPS instruction?

You may find Figures 3.18 (page 153), 4.48 (page 292), and A.19 (page A-54) useful.

4.15 [10] <§§4.2, 4.4, 4.8> This exercise is similar to Exercise 4.14, but this time use the bit pattern

0000 0000 0000 0000 0000 0000 0000 0000

4.16 [10] <§4.3> One of the differences between Sun's SPARC architecture and the MIPS architecture we've been studying is that the load word instruction on the SPARC can specify the address either as the sum of two registers' contents or as one register's contents plus a constant offset (i.e., the way MIPS does). The paper "An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks" (R. F. Cmelik, S. I. Kong, D. R. Ditzel, and E. J. Kelly, *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991) reports that on the SPARC, the gcc benchmark has 15% of its loads use the register + register version (with neither register being `$zero`). Assume that the same would be true on the MIPS, if it were modified to have this extra addressing option for `lw` instructions. Using the data from Figure 4.54, what percentage of gcc's instructions could be eliminated with this architectural modification? Why?

4.17 [10] <§4.3> Find the shortest sequence of MIPS instructions to determine if there is a carry out from the addition of two registers, say, registers `$t3` and `$t4`. Place a 0 or 1 in register `$t2` if the carry out is 0 or 1, respectively. (Hint: It can be done in two instructions.)

4.18 [15] <§4.3> {Ex. 4.17} Find the shortest sequence of MIPS instructions to perform double precision integer addition. Assume that one 64-bit, two's complement integer is in registers `$t4` and `$t5` and another is in registers `$t6` and `$t7`. The sum is to be placed in registers `$t2` and `$t3`. In this example, the most significant word of the 64-bit integer is found in the even-numbered registers, and the least significant word is found in the odd-numbered registers. (Hint: It can be done in four instructions.)

4.19 [15] <§4.3> Suppose that all of the conditional branch instructions except `beq` and `bne` were removed from the MIPS instruction set along with `slt` and all of its variants (`slti`, `sltu`, `sltui`). Show how to perform

```
slt $t0, $s0, $s1
```

using the modified instruction set in which `slt` is not available. (Hint: It requires more than two instructions.)

4.20 [10] <§4.4> The following MIPS instruction sequence could be used to implement a new instruction that has two register operands. Give the instruction a name and describe what it does. Note that register `$t0` is being used as a temporary.

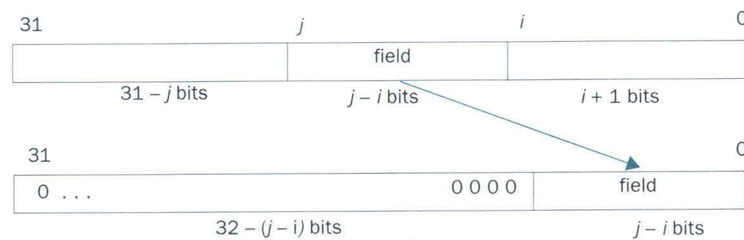
```
srl $s1, $s1, 1    #
sll $t0, $s0, 31   # These 4 instructions accomplish
srl $s0, $s0, 1    # "new $s0 $s1"
or  $s1, $s1, $t0  #
```

4.21 [5] <§4.4> Instead of using a special hardware multiplier, it is possible to multiply using shift and add instructions. This is particularly attractive when multiplying by small constants. Suppose we want to put five times the value

INTEL - 1012

of `$s0` into `$s1`, ignoring any overflow that may occur. Show a minimal sequence of MIPS instructions for doing this without using a multiply instruction.

4.22 [15] <§4.4> Some computers have explicit instructions to extract an arbitrary field from a 32-bit register and to place it in the least significant bits of a register. The figure below shows the desired operation:



Find the shortest sequence of MIPS instructions that extracts a field for the constant values $i = 7$ and $j = 19$ from register `$s0` and places it in register `$s1`. (Hint: It can be done in two instructions.)

4.23 [15] <§4.5> The ALU supported set on less than (`slt`) using just the sign bit of the adder. Let's try a set on less than operation using the values -7_{ten} and 6_{ten}. To make it simpler to follow the example, let's limit the binary representations to 4 bits: 1001_{two} and 0110_{two}.

$$1001_{two} - 0110_{two} = 1001_{two} + 1010_{two} = 0011_{two}$$

This result would suggest that $-7 > 6$, which is clearly wrong. Hence we must factor in overflow in the decision. Modify the 1-bit ALU in Figure 4.17 on page 238 to handle `slt` correctly. Make your changes on a photocopy of this figure to save time.

4.24 [20] <§4.6> Find the shortest sequence of MIPS instructions to perform double precision integer multiplication. Try to do it in 35 instructions or less. Assume that one 64-bit, *unsigned* integer is in registers `$t4` and `$t5` and another is in registers `$t6` and `$t7`. The 128-bit product is to be placed in registers `$t0`, `$t1`, `$t2`, and `$t3`. In this example, the most significant word is found in the lower-numbered registers, and the least significant word is found in the higher-numbered registers. (Hint: Write out the formula for $(a \times 2^{32} + b) \times (c \times 2^{32} + d)$.)

4.25 [5] <§4.8> Show the IEEE 754 binary representation for the floating-point number 10_{ten} in single and double precision.

4.26 [5] <§4.8> This exercise is similar to Exercise 4.25, but this time replace the number 10_{ten} with 10.5_{ten}.

4.27 [10] <§4.8> This exercise is similar to Exercise 4.25, but this time replace the number 10_{ten} with 0.1_{ten}.

4.28 [10] <§4.8> This exercise is similar to Exercise 4.25, but this time replace the number 10_{ten} with the decimal fraction $-2/3$.

4.29 [10] <§4.8> Write a simple C program that inputs a floating-point number and shows its bit representation in hexadecimal.

4.30 [10] <§4.8> Write a simple C++ program that inputs a floating-point number and shows its bit representation in hexadecimal.

4.31 [10] <§4.8> A single precision IEEE number is stored in memory at address X. Write a sequence of MIPS instructions to multiply the number at X by 2 and store the result back at X. Accomplish this without using any floating-point instructions (don't worry about overflow).

4.32 [10] <§4.11> For the program gcc (Figure 4.54 on page 311), find the 10 most frequently executed MIPS instructions. List them in order of popularity, from most used to least used. Show the rank, name, and percentage of instructions executed for each instruction. If there is a tie for a given rank, list all instructions that tie with the same rank, even if this results in more than 10 instructions.

4.33 [10] <§4.11> This exercise is similar to Exercise 4.32, but this time replace the program gcc with the program spice.

4.34 <§4.11> {Ex. 4.32, 4.33} These questions examine the relative frequency of instructions in different programs.

a. [5] Which instructions are found both in the answer to Exercise 4.32 and in the answer to Exercise 4.33?

b. [5] What percentage of gcc instructions executed is due to the instructions identified in Exercise 4.34a?

c. [5] What percentage of gcc instructions executed is due to the instructions identified in Exercise 4.32?

d. [5] What percentage of spice instructions executed is due to the instructions identified in Exercise 4.34a?

e. [5] What percentage of spice instructions executed is due to the instructions identified in Exercise 4.33?

4.35 [10] <§4.11> {Ex. 4.32–4.34} If you were designing a machine to execute the MIPS instruction set, what are the five instructions that you would try to make as fast as possible, based on the answers to Exercises 4.32 through 4.34? Give your rationale.

INTEL - 1012

4.36 [15] <§§2.3, 4.11> Using Figure 4.54 on page 311, calculate the average clock cycles per instruction (CPI) for the program gcc. Figure 4.55 gives the average CPI per instruction category, taking into account cache misses and other effects. Assume that instructions omitted from the table have a CPI of 1.0.

Instruction category	Average CPI
Loads and stores	1.4
Conditional branch	1.8
Jumps	1.2
Integer multiply	10.0
Integer divide	30.0
Floating-point add and subtract	2.0
Floating-point multiply, single precision	4.0
Floating-point multiply, double precision	5.0
Floating-point divide, single precision	12.0
Floating-point divide, double precision	19.0

FIGURE 4.55 CPI for MIPS instruction categories.

4.37 [15] <§§2.3, 4.11> This exercise is similar to Exercise 4.36, but this time replace the program gcc with the program spice.

4.38 [2 weeks] Write a simulator for a subset of the MIPS instruction set using MIPS instructions and the SPIM simulator described in Appendix A. Your simulator should execute hand-assembled programs that are located in the data segment of the SPIM simulator and should use `$v0` and `$v1` for input and output. Other portions of the data segment can be used for storing the memory contents and register values of your virtual machine. Your implementation can use any of the MIPS instructions, but your simulator need only support a smaller subset of the instruction set (e.g., the instructions appearing in Chapters 5 and 6). (Additional details regarding this assignment are available at *www.mkp.com/cod2e.htm.*)

4.39 [1 week] {Ex. 4.38} Add an exception handler to the simulator you developed for Exercise 4.38. Your simulator should generate a simulated exception if a misaligned word is accessed via an `lw`, `sw`, or `jr` instruction. The exception handler should print out an error message identifying the offending address (within the simulation) and then realign the access, perform the instruction, and resume executing the simulated program. (Additional details regarding this assignment are available at *www.mkp.com/cod2e.htm.*)

In More Depth

Logical Instructions

The full MIPS instruction set has two more logical operations not mentioned thus far: `xor` and `nor`. The operation `xor` stands for exclusive OR, and `nor` stands for not OR. The table that follows defines these operations on a bit-by-bit basis. These instructions will be useful in the following two exercises.

A	B	A xor B	A nor B
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	0

4.40 [15] <§4.4> Show the minimal MIPS instruction sequence for a new instruction called `swap` that exchanges two registers. After the sequence completes, the Destination register has the original value of the Source register, and the Source register has the original value of the Destination register. Convert this instruction:

```
swap $s0,$s1
```

The hard part is that this sequence *must use only these two registers!* (Hint: It can be done in three instructions if you use the new logical instructions. What is the value of `(A xor B xor A)`?)

4.41 [5] <§4.4> Show the minimal MIPS instruction sequence for a new instruction called `not` that takes the one's complement of a Source register and places it in a Destination register. Convert this instruction (accepted by the MIPS assembler):

```
not $s0,$s1
```

(Hint: It can be done in one instruction if you use the new logical instructions.)

4.42 [20] <§4.5> A simple check for overflow during addition is to see if the CarryIn to the most significant bit is *not* the same as the CarryOut of the most significant bit. Prove that this check is the same as in Figure 4.4 on page 222.

4.43 [10] <§4.5> Draw the gates for the Sum bit of an adder, given the equation on page 234.

INTEL - 1012

4.44 [5] <§4.5> Rewrite the equations on page 247 for a carry-lookahead logic for a 16-bit adder using a new notation. First use the names for the CarryIn signals of the individual bits of the adder. That is, use c4, c8, c12, . . . instead of C1, C2, C3, ... Also, let $P_{i,j}$ mean a propagate signal for bits i to j, and $G_{i,j}$ mean a generate signal for bits i to j. For example, the equation

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

can be rewritten as

$$c8 = G_{7,4} + (P_{7,4} \cdot G_{3,0}) + (P_{7,4} \cdot P_{3,0} \cdot c0)$$

This more general notation is useful in creating wider adders.

4.45 [15] <§4.5> {Ex. 4.44} Write the equations for the carry-lookahead logic for a *64-bit* adder using the new notation from Exercise 4.44 and using 16-bit adders as building blocks. Include a drawing similiar to Figure 4.24 in your solution.

4.46 [10] <§4.5> Now calculate the relative performance of adders. Assume that hardware corresponding to any equation containing only OR or AND terms, such as the equations for *pi* and *gi* on page 242, takes one time unit T. Equations that consist of the OR of several AND terms, such as the equations for c1, c2, c3, and c4 on page 243, would thus take two time units, 2T, because it would take T to produce the AND terms and then an additional T to produce the result of the OR. Calculate the numbers and performance ratio for 4-bit adders for both ripple carry and carry lookahead. If the terms in equations are further defined by other equations, then add the appropriate delays for those intermediate equations, and continue recursively until the actual input bits of the adder are used in an equation. Include a drawing of each adder labeled with the calculated delays and the path of the worst-case delay highlighted.

4.47 [15] <§4.5> This exercise is similar to Exercise 4.46, but this time calculate the relative speeds of a 16-bit adder using ripple carry only, ripple carry of 4-bit groups that use carry lookahead, and the carry-lookahead scheme on page 242.

4.48 [15] <§4.5> {Ex. 4.45} This exercise is similar to Exercises 4.46 and 4.47, but this time calculate the relative speeds of a 64-bit adder using ripple carry only, ripple carry of 4-bit groups that use carry lookahead, ripple carry of 16-bit groups that use carry lookahead, and the carry-lookahead scheme from Exercise 4.45.

4.49 [10] <§4.5> There are times when we want to add a collection of numbers together. Suppose you wanted to add four 4-bit numbers (A,B, E, F) using 1-bit full adders. Let's ignore carry lookahead for now. You would likely connect the 1-bit adders in the organization in the top of Figure 4.56. Below the traditional organization is a novel organization of full adders. Try adding four numbers using both organizations to convince yourself that you get the same answer.

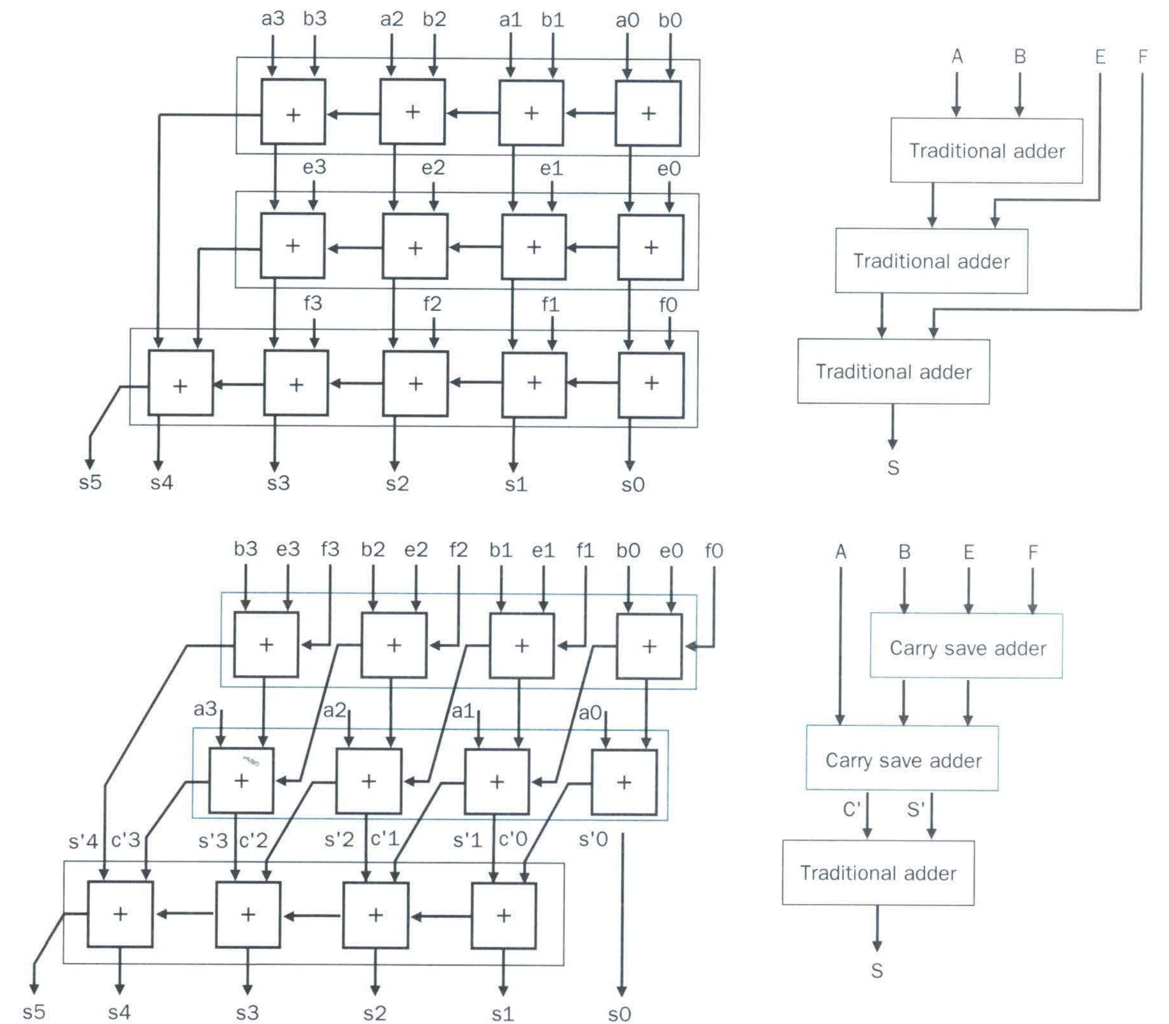


FIGURE 4.56 Traditional ripple carry and carry save addition of four 4-bit numbers. The details are shown on the left, with the individual signals in lowercase, and the corresponding higher-level blocks are on the right, with collective signals in uppercase. Note that the sum of four n-bit numbers can take n+2 bits.

4.50 [5] <§4.5> {Ex. 4.49} Assume that the time delay through each 1-bit adder is 2T. Calculate the time of adding four 4-bit numbers to the organization at the top versus the organization in the bottom in Figure 4.56.

INTEL - 1012

In More Depth

Carry Save Adders

Exercises 4.49 and 4.50 motivate an organization that uses the 1-bit adder in Figure 4.10 on page 232 in a way it was not intended. Although this piece of hardware is simple and fast, the problem comes from trying to get the CarryIn signal calculated in a timely fashion across several adders.

We can think of the adder instead as a hardware device that can add three inputs together (*ai*, *bi*, *ci*) and produce two outputs (s, *ci+1*). When we are just adding two numbers together, there is little we can do with this observation, but when we are adding more than two operands, it is possible to reduce the cost of the carry. The idea is to form two independent sums, called S′ (sum bits) and C′ (carry bits). At the end of the process, we need to add C′ and S′ together using a normal adder. This technique of delaying carry propagation until the end of a sum of numbers is called *carry save addition*. The block drawing on the lower right of Figure 4.56 shows the organization, with two levels of carry save adders connected by a single normal adder.

4.51 [10] <§4.5> {Ex. 4.47, 4.50} Calculate the delays to add four 16-bit numbers using full carry-lookahead adders versus carry save with a carry-lookahead adder forming the final sum. (The time unit T in Exercises 4.46 and 4.50 is the same.)

4.52 [20] <§4.5, 4.6> {Ex. 4.47} Perhaps the most likely case of adding many numbers at once in a computer would be when trying to multiply more quickly by using many adders to add many numbers in a single clock cycle. Compared to the multiply algorithm in Figure 4.32 on page 258, a carry save scheme with many adders could multiply more than 10 times faster.

This exercise estimates the cost and speed of a combinational multiplier to multiply two positive 16-bit numbers. Assume that you have 16 intermediate terms M15, M14, . . . , M0, called *partial products*, that contain the multiplicand ANDed with multiplier bits m15, m14, . . . , m0.

The idea is to use carry save adders to reduce the n operands into $2n/3$ in parallel groups of three, and do this repeatedly until you get two large numbers to add together with a traditional adder.

First show the block organization of the 16-bit carry save adders to add these 16 terms, as shown on the right in Figure 4.56. Then calculate the delays to add these 16 numbers. Compare this time to the iterative multiplication scheme in Figure 4.32 on page 258 but only assume 16 iterations using a 16-bit adder that has full carry lookahead whose speed was calculated in Exercise 4.47.

4.53 [30] <§4.6> The original reason for Booth's algorithm was to reduce the number of operations by avoiding operations when there were strings of 0s and 1s. Revise the algorithm on page 260 to look at 3 bits at a time and compute the product 2 bits at a time. Fill in the following table to determine the 2-bit Booth encoding:

Current bits		Previous bit	Operation	Reason
ai+1	*ai*	*ai–1*		
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Assume that you have both the multiplicand and 2 × multiplicand already in registers. Explain the reason for the operation on each line, and show a 6-bit example that runs faster using this algorithm. (Hint: Try dividing to conquer; see what the operations would be in each of the eight cases in the table using a 2-bit Booth algorithm, and then optimize the pair of operations.)

4.54 [30] <§4.6, 4.7> The division algorithm in Figure 4.40 on page 270 is called *restoring division*, since each time the result of subtracting the divisor from the dividend is negative you must add the divisor back into the dividend to restore the original value. Recall that shift left is the same as multiplying by two. Let's look at the value of the left half of the Remainder again, starting with step 3b of the divide algorithm and then going to step 2:

$$(\text{Remainder} + \text{Divisor}) \times 2 - \text{Divisor}$$

This value is created from restoring the Remainder by adding the Divisor, shifting the sum left, and then subtracting the Divisor. Simplifying the result we get

$$\text{Remainder} \times 2 + \text{Divisor} \times 2 - \text{Divisor} = \text{Remainder} \times 2 + \text{Divisor}$$

Based on this observation, write a *nonrestoring division* algorithm using the notation of Figure 4.40 that does not add the Divisor to the Remainder in step 3b. Show that your algorithm works by dividing $0000\ 0111_{\text{two}}$ by 0010_{two}.

INTEL - 1012

4.55 [5] <§4.8> Add $6.42_{ten} \times 10^1$ to $9.51_{ten} \times 10^2$, assuming that you have only three significant digits, first with guard and round digits and then without them.

4.56 [5] <§4.8> This exercise is similar to Exercise 4.55, but this time use the numbers $8.76_{ten} \times 10^1$ and $1.47_{ten} \times 10^2$.

4.57 [25] <§4.8> Derive the floating-point algorithm for division as we did for addition and multiplication on pages 280 through 288. First divide $1.110_{ten} \times 10^{10}$ by $1.100_{ten} \times 10^{-5}$, showing the same steps that we did in the example starting on page 282. Then derive the floating-point division algorithm using a format similar to the multiplication algorithm in Figure 4.46 on page 289.

4.58 [30] <§4.8> The elaboration on page 300 explains the four rounding modes of IEEE 754 and the extra bit, called the *sticky bit*, needed in addition to the 2 bits called *guard* and *round*. Guard is the first bit, round is the second bit, and sticky represents whether the remaining bits are 0 or not. Fill in the following table with logical equations that are functions of guard (g), round (r), and sticky (s) for the result of a floating-point addition that creates Sum. Let p be the proper number of bits in the significand for a given precision and Sum_p be the pth most significant bit of Sum. A blank box means that the p most significant bits of the sum are correctly rounded. If you place an equation in a box, a false equation means that the p bits are correctly rounded; a true equation means add 1 to the pth most significant bit of Sum.

Rounding mode	Sum >> 0	Sum < 0
Toward $-\infty$		
Toward $+\infty$		
Truncate		
Nearest even		

4.59 [30] <§4.8> The elaboration on page 300 mentions that IEEE 754 has two special symbols that are floating-point operands: infinity and Not a Number (NaN). There are also small numbers called *denorms*, which are not normalized. Because these special symbols and numbers are not used very frequently, implementations that employ a mix of both hardware and software techniques are sometimes used. For example, instead of using complicated hardware to handle these special cases, an exception is generated and they are handled in software. Many implementation options exist, each of which has unique performance characteristics. Your task is to benchmark several different machines for floating-point operations as the operands vary from normal numbers to these special cases. Be sure to state your conclusions by comparing the performance of different machines with one another and describing their similarities

and differences. What impact are your results likely to have on software designers who must choose whether or not to make use of the special features in the IEEE 754 standard?

4.60 [30] <§4.5> If you have access to a computer containing a MIPS processor, write a loop in assembly language that sets registers `$k0` (`$26`) and `$k1` (`$27`) to an initial value, and then loop for several seconds, checking the contents of these registers. Print the values if they change. See the elaboration on page 225 for an explanation of why they change. Can you find a reason for the particular values you observe?

INTEL - 1012

5

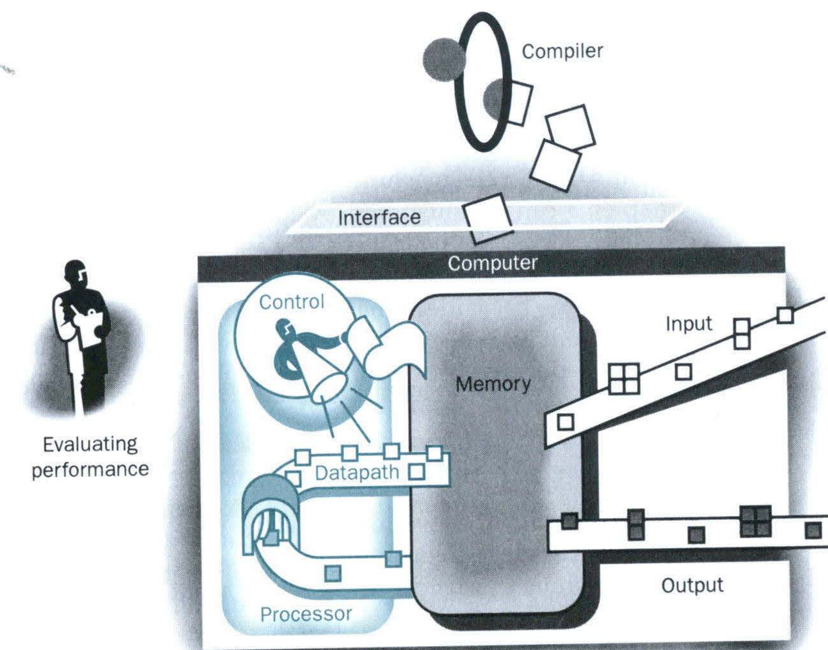
The Processor: Datapath and Control

In a major matter,
no details are small.

French Proverb

5.1 **Introduction** 338
5.2 **Building a Datapath** 343
5.3 **A Simple Implementation Scheme** 351
5.4 **A Multicycle Implementation** 377
5.5 **Microprogramming: Simplifying Control Design** 399
5.6 **Exceptions** 410
5.7 **Real Stuff: The Pentium Pro Implementation** 416
5.8 **Fallacies and Pitfalls** 419
5.9 **Concluding Remarks** 421
5.10 **Historical Perspective and Further Reading** 423
5.11 **Key Terms** 426
5.12 **Exercises** 427

The Five Classic Components of a Computer



INTEL - 1012

5.1 Introduction

In Chapter 2, we saw that the performance of a machine was determined by three key factors: instruction count, clock cycle time, and clock cycles per instruction (CPI). The compiler and the instruction set architecture, which we examined in Chapters 3 and 4, determine the instruction count required for a given program. However, both the clock cycle time and the number of clock cycles per instruction are determined by the implementation of the processor. In this chapter, we construct the datapath and control unit for two different implementations of the MIPS instruction set.

We will be designing an implementation that includes a subset of the core MIPS instruction set:

- The memory-reference instructions load word (`lw`) and store word (`sw`)
- The arithmetic-logical instructions `add`, `sub`, `and`, `or`, and `slt`
- The instructions branch equal (`beq`) and jump (`j`), which we add last

This subset does not include all the integer instructions (for example, multiply and divide are missing), nor does it include any floating-point instructions. However, the key principles used in creating a datapath and designing the control will be illustrated. The implementation of the remaining instructions is similar.

In examining the implementation, we will have the opportunity to see how the instruction set architecture determines many aspects of the implementation, and how the choice of various implementation strategies affects the clock rate and CPI for the machine. Many of the key design principles introduced in Chapter 3 can be illustrated by looking at the implementation, such as the guidelines *Make the common case fast* and *Simplicity favors regularity.* In addition, most concepts used to implement the MIPS subset in this chapter and the next are the same basic ideas that are used to construct a broad spectrum of computers, from high-performance machines to general-purpose microprocessors to special-purpose processors, which are used increasingly in products ranging from VCRs to automobiles.

An Overview of the Implementation

In Chapters 3 and 4, we looked at the core MIPS instructions, including the integer arithmetic-logical instructions, the memory-reference instructions, and the branch instructions. Much of what needs to be done to implement these instructions is the same, independent of the exact class of instruction. For every instruction, the first two steps are identical:

1. Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction we need to read only one register, but most other instructions require that we read two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact opcode.

Even across different instruction classes there are some similarities. For example, all instruction classes use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and branches for comparison. As we can see, the simplicity and regularity of the instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

After using the ALU, the actions required to complete the different instruction classes differ. A memory-reference instruction will need to access the memory either to write data for a store or read data for a load. An arithmetic-logical instruction must write the data from the ALU back into a register. Lastly, for a branch instruction, we may need to change the next instruction address based on the comparison.

Figure 5.1 shows the high-level view of a MIPS implementation. In the remainder of the chapter, we refine this view to fill in the details, which requires that we add further functional units, increase the number of connections between units, and, of course, add a control unit to control what actions are taken for different instruction classes. Before we begin to create a more complete implementation, we need to discuss a few principles of logic design.

A Word about Logic Conventions and Clocking

To discuss the design of a machine, we must decide how the logic implementing the machine will operate and how the machine is clocked. This section reviews a few key ideas in digital logic that we will use extensively in this chapter. If you have little or no background in digital logic, you will find it helpful to read through Appendix B before continuing. Section B.9 presents the key terms introduced in Appendix B and is useful as a quick check-up if you want to review your logic design background.

When designing logic, it is often convenient for the designer to change the mapping between a logically true or false signal and the high or low voltage level. Thus, in some parts of a design, a signal that is logically asserted may actually be an electrically low signal, while in others an electrically high signal is asserted. To maintain consistency, we will use the word *asserted* to indicate a signal that is logically high and *assert* to specify that a signal should be driven logically high.

The functional units in the MIPS implementation consist of two different types of logic elements: elements that operate on data values and elements that contain state. The elements that operate on data values are all *combinational*, which means that their outputs depend only on the current inputs. Given the same input, a combinational element always produces the same output. The ALU shown in Figure 5.1 and discussed in detail in Chapter 4 is a combinational element. Given a set of inputs, it always produces the same output because it has no internal storage.

Other elements in the design are not combinational, but instead contain *state*. An element contains state if it has some internal storage. We call these elements *state elements* because, if we pulled the plug on the machine, we could restart it by loading the state elements with the values they contained before

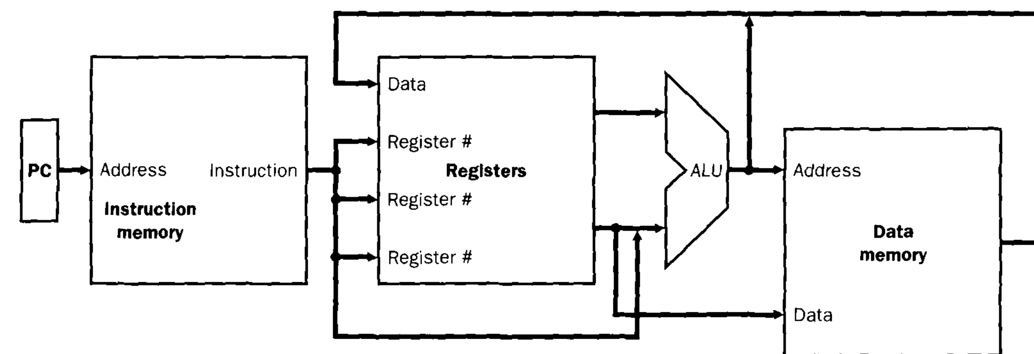


FIGURE 5.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which requires some control logic, as we will see.

we pulled the plug. Furthermore, if we saved and restored the state elements, it would be as if the machine had never lost power. Thus, these state elements completely characterize the machine. In Figure 5.1, the instruction and data memories as well as the registers are all examples of state elements.

A state element has at least two inputs and one output. The required inputs are the data value to be written into the element, and the clock, which determines when the data value is written. The output from a state element provides the value that was written in an earlier clock cycle. For example, one of the logically simplest state elements is a D-type flip-flop (see Appendix B), which has exactly these two inputs (a value and a clock) and one output. In addition to flip-flops, our MIPS implementation also uses two other types of state elements: memories and registers, both of which appear in Figure 5.1. The clock is used to determine when the state element should be written; a state element can be read at any time.

Logic components that contain state are also called *sequential* because their outputs depend on both their inputs and the contents of the internal state. For example, the output from the functional unit representing the registers depends both on the register numbers supplied and on what was written into the registers previously. The operation of both the combinational and sequential elements and their construction are discussed in more detail in Appendix B.

Clocking Methodology

A *clocking methodology* defines when signals can be read and when they can be written. It is important to specify the timing of reads and writes because, if a signal is written at the same time it is read, the value of the read could correspond to the old value, the newly written value, or even some mix of the two! Needless to say, computer designs cannot tolerate such unpredictability. A clocking methodology is designed to prevent this circumstance.

For simplicity, we will assume an *edge-triggered* clocking methodology. An edge-triggered clocking methodology means that any values stored in the machine are updated only on a clock edge. Thus, the state elements all update their internal storage on the clock edge. Because only state elements can store a data value, any collection of combinational logic must have its inputs coming from a set of state elements and its outputs written into a set of state elements. The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle.

Figure 5.2 shows the two state elements surrounding a block of combinational logic, which operates in a single clock cycle: All signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle. The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

INTEL - 1012

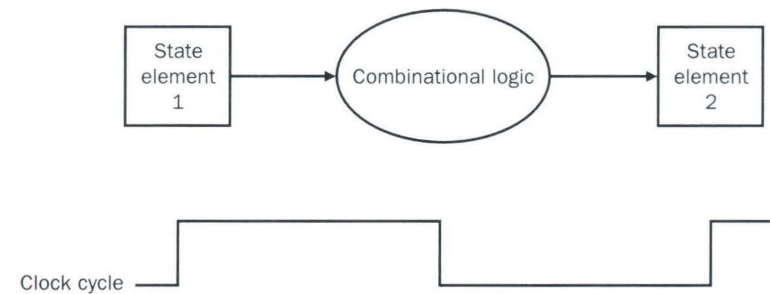


FIGURE 5.2 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements, including memory, are assumed to be edge-triggered.

For simplicity, we do not show a write control signal when a state element is written on every active clock edge. In contrast, if a state element is not updated on every clock, then an explicit write control signal is required. Both the clock signal and the write control signal are inputs, and the state element is changed only when the write control signal is asserted and a clock edge occurs.

An edge-triggered methodology allows us to read the contents of a register, send the value through some combinational logic, and write that register in the same clock cycle, as shown in Figure 5.3. It doesn't matter whether we assume that all writes take place on the rising clock edge or on the falling clock edge, since the inputs to the combinational logic block cannot change except on the chosen clock edge. With an edge-triggered timing methodology, there is *no* feedback within a single clock cycle, and the logic in Figure 5.3 works correctly. In Appendix B we briefly discuss additional timing constraints (such as set-up and hold times) as well as other timing methodologies.

Nearly all of these state and logic elements will have inputs and outputs that are 32 bits wide, since that is the width of most of the data handled by the processor. We will make it clear whenever a unit has an input or output that is other than 32 bits in width. The figures will indicate *buses*, which are signals wider than 1 bit, with thicker lines. At times we will want to combine several buses to form a wider bus; for example, we may want to obtain a 32-bit bus by combining two 16-bit buses. In such cases, labels on the bus lines will make it clear that we are concatenating buses to form a wider bus. Arrows are also added to help clarify the direction of the flow of data between elements. Finally, color indicates a control signal as opposed to a signal that carries data; this distinction will become clearer as we proceed through this chapter.

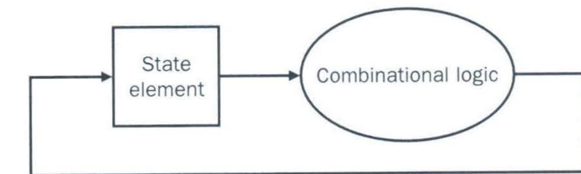


FIGURE 5.3 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values. Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. Feedback cannot occur within 1 clock cycle because of the edge-triggered update of the state element. If feedback were possible, this design could not work properly. Our designs in this chapter and the next rely on the edge-triggered timing methodology and structures like the one shown in this figure.

The MIPS Subset Implementation

We will start with a simple implementation that uses a single long clock cycle for every instruction and follows the general form of Figure 5.1. In this first design, every instruction begins execution on one clock edge and completes execution on the next clock edge.

While easier to understand, this approach is not practical, since it would be slower than an implementation that allows different instruction classes to take different numbers of clock cycles, each of which could be much shorter. After designing the control for this simple machine, we will look at an implementation that uses multiple clock cycles for each instruction. This implementation is more realistic but also requires more complex control.

In this chapter, we will take the specification of the control to the level of logic equations or finite state machine specifications. From either representation, a modern computer-aided design (CAD) system can synthesize a hardware implementation; Appendix C shows how this is done. Before closing the chapter, we will discuss how exceptions, mentioned in Chapter 4, are implemented.

5.2 Building a Datapath

A reasonable way to start a datapath design is to examine the major components required to execute each class of MIPS instruction. Let's start by looking at which datapath elements each instruction needs and build up the sections of the datapath for each instruction class from these elements. When we show the datapath elements, we will also show their control signals.

INTEL - 1012

The first element we will need is a place to store the instructions of a program. A memory unit, which is a state element, is used to hold and supply instructions given an address, as shown in Figure 5.4. The address of the instruction must also be kept in a state element, which we call the *program counter* (PC), also shown in Figure 5.4. Lastly, we will need an adder to increment the PC to the address of the next instruction. This adder, which is combinational, can be built from the ALU we designed in the last chapter simply by wiring the control lines so that the control always specifies an add operation. We will draw such an ALU with the label *Add*, as in Figure 5.4, to indicate that it has been permanently made an adder and cannot perform the other ALU functions.

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later. The datapath for this step, shown in Figure 5.5, uses the three elements from Figure 5.4.

Now let's consider the R-format instructions (see Figure 3.19 on page 154). They all read two registers, perform an ALU operation on the contents of the registers, and write the result. We call these instructions either *R-type instructions* or *arithmetic-logical instructions* (since they perform arithmetic or logical

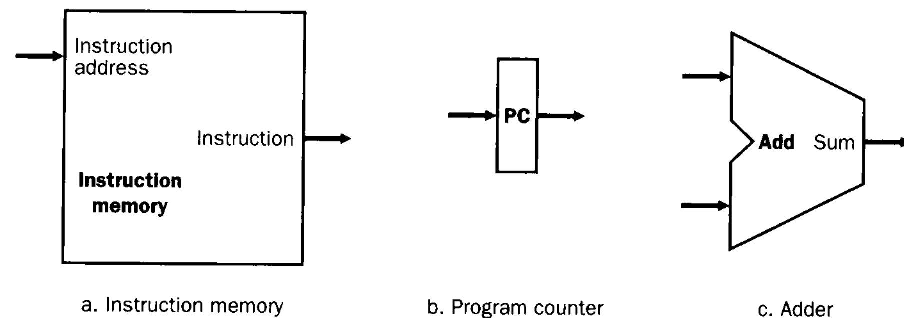


FIGURE 5.4 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory is only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) Since the instruction memory unit can only be read, we do not include a read control signal; this simplifies the design. The program counter is a 32-bit register that will be written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always perform an add of its two 32-bit inputs and place the result on its output.

operations). This instruction class includes `add`, `sub`, and `slt`, which were introduced in Chapter 3, as well as `and` and `or`, which were introduced in Chapter 4. Recall that a typical instance of such an instruction is `add $t1,$t2,$t3`, which reads `$t2` and `$t3` and writes `$t1`.

The processor's 32 registers are stored in a structure called a *register file*. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the machine. In addition, we will need an ALU to operate on the values read from the registers.

Because the R-format instructions have three register operands, we will need to read two data words from the register file and write one data word into the register file for each instruction. For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers. To write a data word, we will need two inputs: one to specify the *register number* to be written and one to supply the *data* to be written into the register. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge. Thus, we need a total of four inputs (three for register

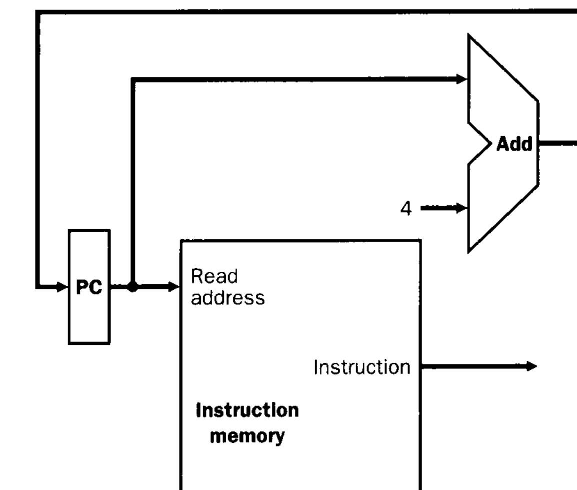


FIGURE 5.5 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

INTEL - 1012

numbers and one for data) and two outputs (both for data), as shown in Figure 5.6. The register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$), whereas the data input and two data output buses are each 32 bits wide.

The ALU, shown in Figure 5.6, is controlled by the 3-bit signal described in Chapter 4. The ALU takes two 32-bit inputs and produces a 32-bit result.

The datapath for these R-type instructions, which uses the register file and the ALU of Figure 5.6, is shown in Figure 5.7. Since the register numbers come from fields of the instruction, we show the instruction, which comes from Figure 5.5, as connected to the register number inputs of the register file.

Next, consider the MIPS load word and store word instructions, which have the general form: `lw $t1,offset_value($t2)` or `sw $t1,offset_value ($t2)`. These instructions compute a memory address by adding the base reg-

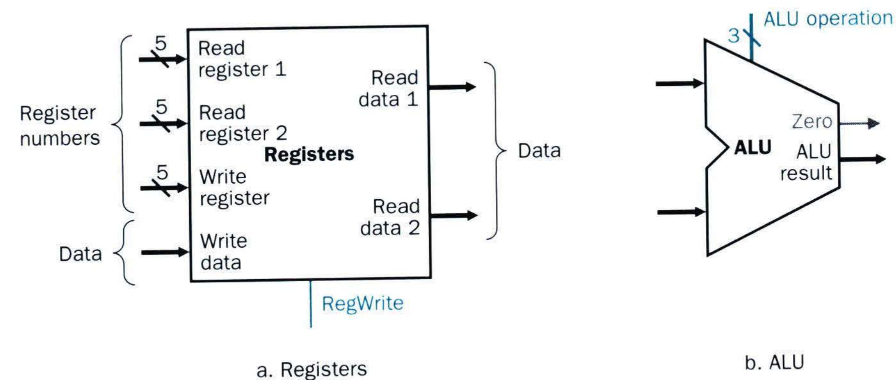


FIGURE 5.6 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in section B.5 of Appendix B. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 3 bits wide, using the ALU designed in the previous chapter (see Figure 4.19 on page 237). We will use the Zero detection output of the ALU shortly to implement branches. The overflow output will not be needed until section 5.6, when we discuss exceptions; we omit it until then.

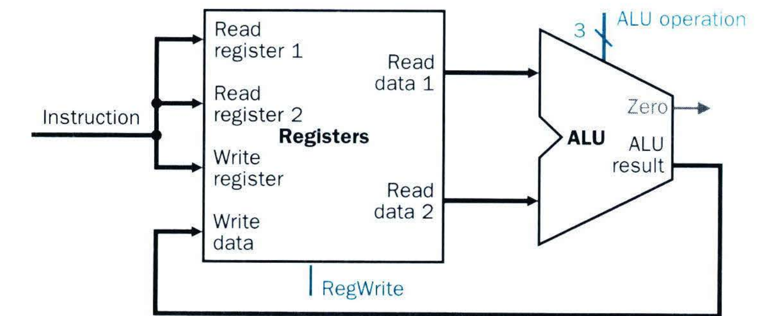


FIGURE 5.7 The datapath for R-type instructions. The ALU discussed in Chapter 4 can be controlled to provide all the basic ALU functions required for R-type instructions.

ister, which is `$t2`, to the 16-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in `$t1`. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is `$t1`. Thus, we will need both the register file and the ALU shown in Figure 5.6.

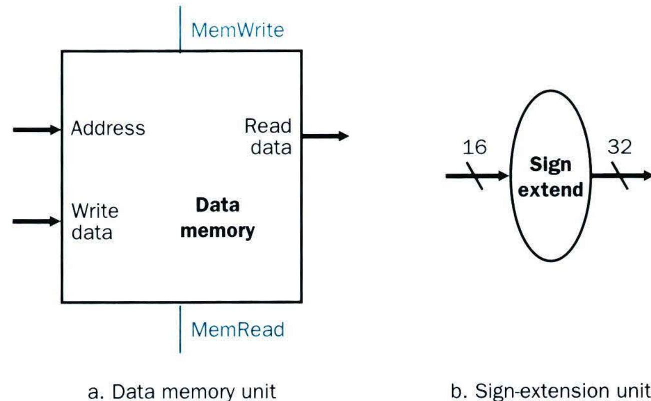
In addition, we will need a unit to sign-extend the 16-bit offset field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to. The data memory must be written on store instructions; hence, it has both read and write control signals, an address input, as well as an input for the data to be written into memory. Figure 5.8 shows these two elements.

Figure 5.9 shows how to combine these elements to build the datapath for a load word or a store word instruction, assuming that the instruction has already been fetched. The register number inputs for the register file come from fields of the instruction, as does the offset value, which after sign extension becomes the second ALU input.

The `beq` instruction has three operands, two registers that are compared for equality, and a 16-bit offset used to compute the branch target address relative to the branch instruction address. Its form is `beq $t1,$t2,offset`. To implement this instruction, we must compute the branch target address by adding the sign-extended offset field of the instruction to the PC. There are two details in the definition of branch instructions (see Chapter 3) to which we must pay attention:

- The instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch.

INTEL - 1012



a. Data memory unit b. Sign-extension unit

FIGURE 5.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 5.6, are the data memory unit and the sign extension unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output (see Chapter 4, page 216). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See section B.5 of Appendix B for a further discussion of how real memory chips work.

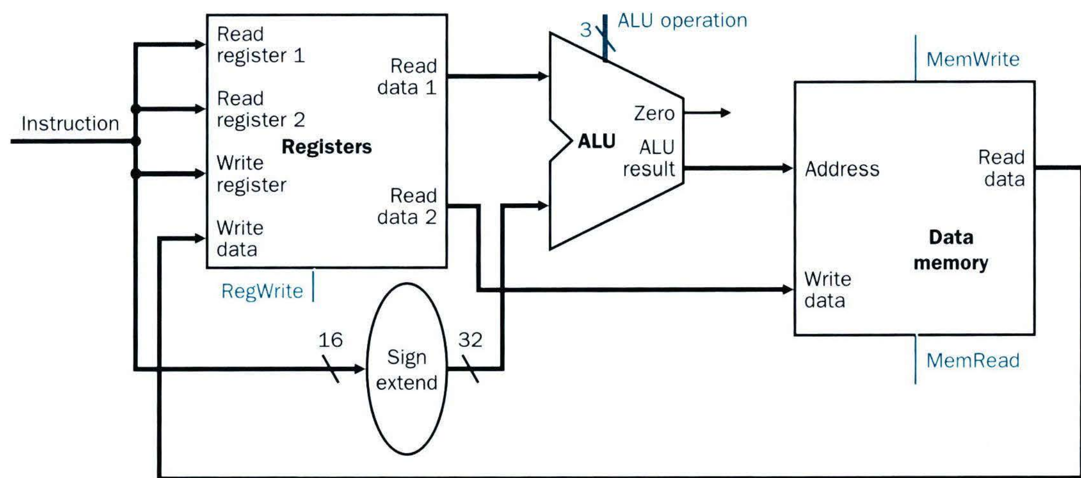


FIGURE 5.9 The datapath for a load or store does a register access, followed by a memory address calculation, then a read or write from memory, and a write into the register file if the instruction is a load.

Since we compute PC + 4 (the address of the next instruction) in the instruction fetch datapath, it is easy to use this value as the base for computing the branch target address.

- The architecture also states that the offset field is shifted left 2 bits so that it is a word offset; this shift increases the effective range of the offset field by a factor of four.

To deal with the latter complication, we will need to shift the offset field by two.

In addition to computing the branch target address, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address. When the condition is true (i.e., the operands are equal), the branch target address becomes the new PC, and we say that the branch is *taken*. If the operands are not equal, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the branch is *not taken*.

Thus, the branch datapath must do two operations: compute the branch target address and compare the register contents. (Branches also require that we modify the instruction fetch portion of the datapath, which we will deal with shortly.) Figure 5.10 shows the branch datapath. To compute the branch target address, the branch datapath includes a sign extension unit, just like that in Figure 5.8, and an adder. To perform the compare, we need to use the register file shown in Figure 5.6 to supply the two register operands (although we will not need to write into the register file). In addition, the comparison can be done using the ALU we designed in Chapter 4. Since that ALU provides an output signal that indicates whether the result was 0, we can send the two register operands to the ALU with the control set to do a subtract. If the Zero signal out of the ALU unit is asserted, we know that the two values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equal test of branches. Later, we will show exactly how to connect the control signals of the ALU for use in the datapath.

The jump instruction operates by replacing the lower 28 bits of the PC with the lower 26 bits of the instruction shifted left by 2 bits. This shift is accomplished simply by concatenating 00 to the jump offset (as described in the elaboration in Chapter 3, page 150).

Now that we have examined the datapaths needed for the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation. The datapaths shown in Figures 5.5, 5.7, 5.9, and 5.10 will be the building blocks for two different implementations. In the next section, we will create an implementation that uses a single long clock cycle for every instruction. In section 5.4, we will look at an implementation that uses multiple shorter clock cycles for every instruction.

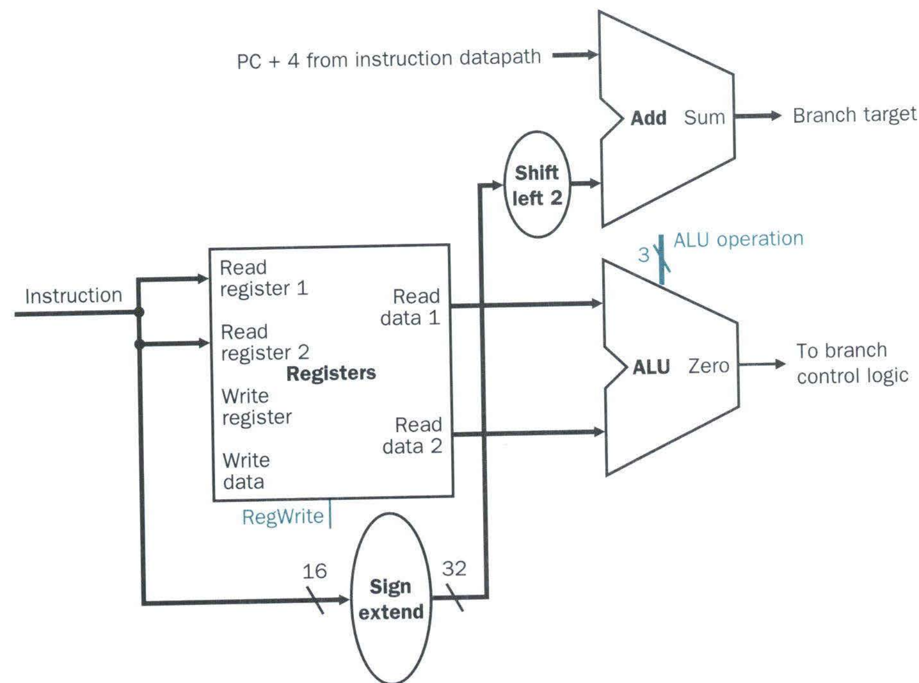


FIGURE 5.10 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. The unit labeled *Shift left 2* is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended offset field; no actual shift hardware is needed, since the amount of the "shift" is constant. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only "sign bits." Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

Elaboration: In the MIPS instruction set, branches are *delayed*, meaning that the instruction immediately following the branch is always executed, *independent* of whether the branch condition is true or false. When the condition is false, the execution looks like a normal branch. When the condition is true, a delayed branch first executes the instruction immediately following the branch in sequential instruction order before jumping to the specified branch target address. The motivation for delayed branches arises from how pipelining affects branches (see section 6.6). For simplicity, we ignore delayed branches in this chapter and implement a nondelayed `beq` instruction.

5.3 A Simple Implementation Scheme

In this section, we look at what might be thought of as the simplest possible implementation of our MIPS subset. We build this simple datapath and its control by assembling the datapath segments of the last section and adding control lines as needed. This simple implementation covers load word (`lw`), store word (`sw`), branch equal (`beq`), and the arithmetic-logical instructions `add`, `sub`, `and`, `or`, and `set on less than`. We will later enhance the design to include a jump instruction (`j`).

Creating a Single Datapath

Suppose we were going to build a datapath from the pieces we looked at in Figures 5.5, 5.7, 5.9, and 5.10. The simplest datapath might attempt to execute all instructions in 1 clock cycle. This means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated. We therefore need a memory for instructions separate from one for data. Although some of the functional units will need to be duplicated when the individual datapaths of the previous section are combined, many of the elements can be shared by different instruction flows.

To share a datapath element between two different instruction classes, we may need to allow multiple connections to the input of an element and have a control signal select among the inputs. This selection is commonly done with a device called a *multiplexor*, although this device might better be called a *data selector*. The multiplexor, which was introduced in the last chapter (Figure 4.8 on page 231), selects from among several inputs based on the setting of its control lines.

Example

Composing Datapaths

The arithmetic-logical (or R-type) instruction datapath of Figure 5.7 on page 347 and the memory instruction datapath of Figure 5.9 on page 348 are quite similar. The key differences are the following:

- The second input to the ALU unit is either a register (if it's an R-type instruction) or the sign-extended lower half of the instruction (if it's a memory instruction).
- The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

INTEL - 1012

Show how to combine the two datapaths using multiplexors, without duplicating the functional units that are in common in Figures 5.7 and 5.9. Ignore the control of the multiplexors.

Answer

To combine the two datapaths and use only a single register file and an ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus one multiplexor is placed at the ALU input and another at the data input to the register file. Figure 5.11 shows the combined datapath.

The instruction fetch portion of the datapath, shown in Figure 5.5 on page 345, can easily be added to the datapath in Figure 5.11. Figure 5.12 shows the result. The combined datapath includes a memory for instructions and a separate memory for data. This combined datapath requires both an adder and an ALU, since the adder is used to increment the PC while the other ALU is used for executing the instruction in the same clock cycle.

Now we can combine all the pieces to make a simple datapath for the MIPS architecture by adding the datapath for branches from Figure 5.10 on page 350. Figure 5.13 on page 354 shows the datapath we obtain by composing the separate pieces. The branch instruction uses the main ALU for comparison of the register operands, so we must keep the adder in Figure 5.10 for computing the branch target address. An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC.

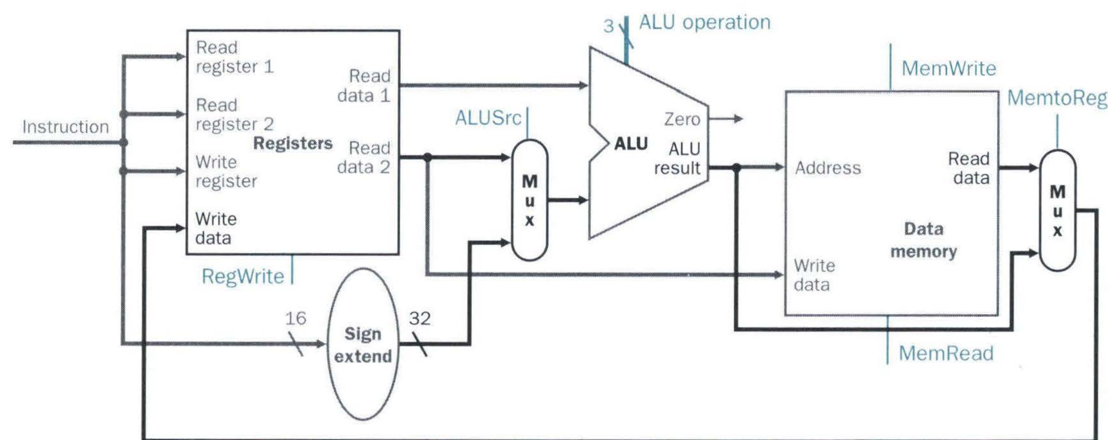


FIGURE 5.11 Combining the datapaths for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 5.7 and 5.9 by adding multiplexors. The added multiplexors and connections have been highlighted. The control lines for the multiplexors are also shown.

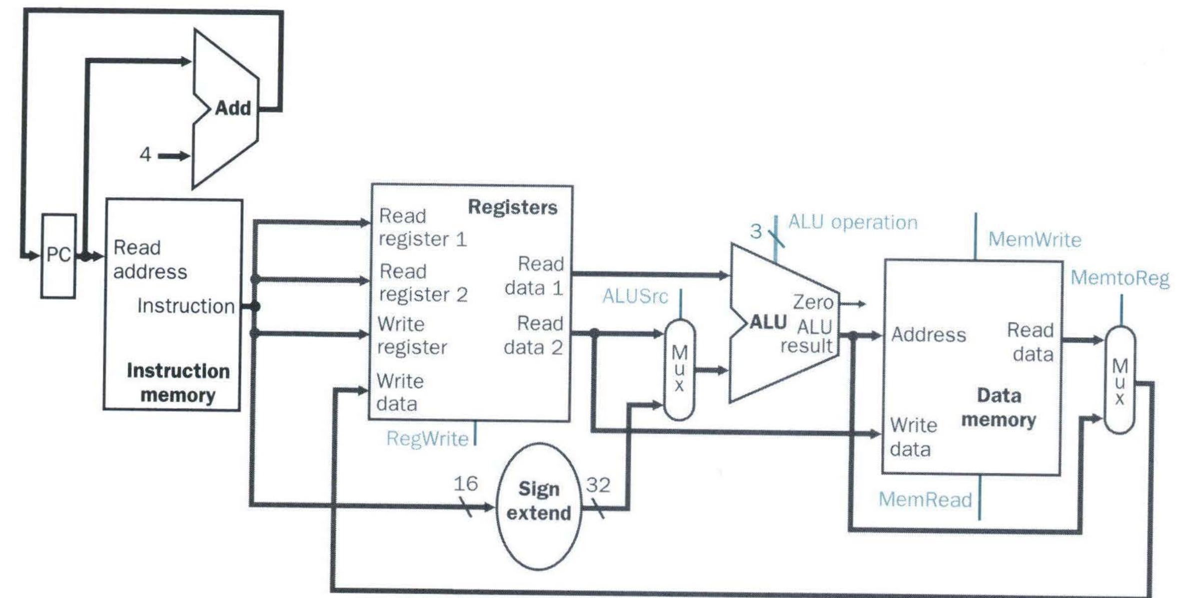


FIGURE 5.12 The instruction fetch portion of the datapath from Figure 5.5 is appended to the datapath of Figure 5.11 that handles memory and ALU instructions. The addition is highlighted. The result is a datapath that supports many operations of the MIPS instruction set—branches and jumps are the major missing pieces.

Now that we have completed this simple datapath, we can add the control unit. The control unit must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control. The ALU control is different in a number of ways, and it will be useful to design it first before we design the rest of the control unit.

The ALU Control

Recall from Chapter 4 that the ALU has three control inputs. Only five of the possible eight input combinations are used. Figure 4.20 on page 240 showed the five following combinations:

ALU control input	Function
000	AND
001	OR
010	add
110	subtract
111	set on less than

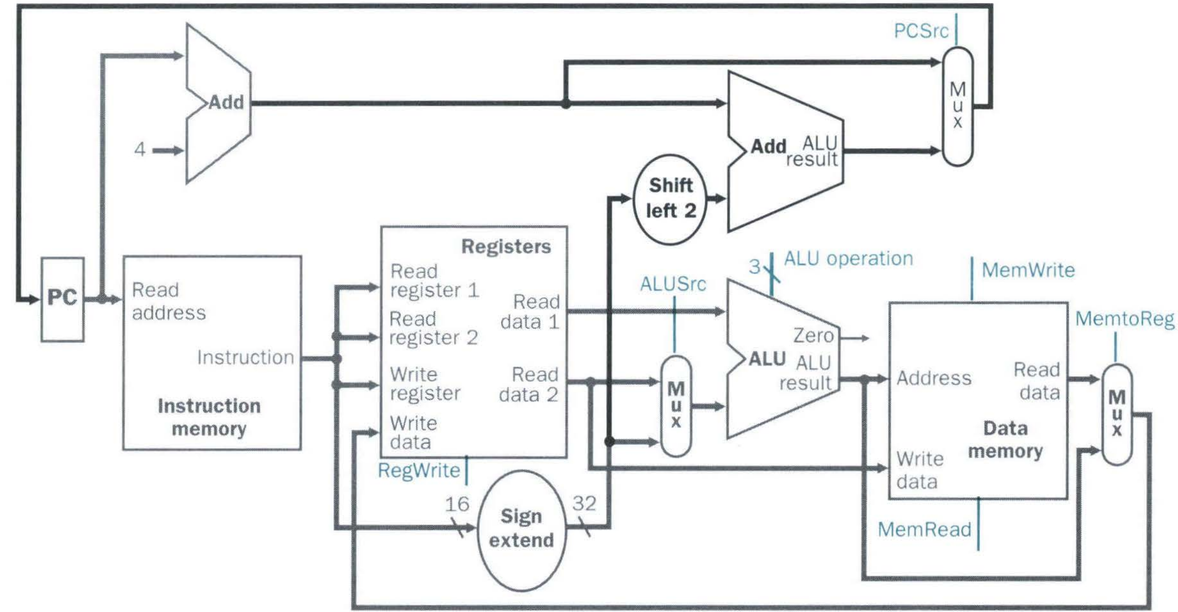


FIGURE 5.13 The simple datapath for the MIPS architecture combines the elements required by different instruction classes. This datapath can execute the basic instructions (load/store word, ALU operations, and branches) in a single clock cycle. The additions to Figure 5.12, which are all highlighted, are used to implement branches. The datapath components for branches come from Figure 5.10. A multiplexor is also needed, since the value written into the PC can be either the sequentially incremented PC or the branch target PC. The support for jumps will be added later.

Depending on the instruction class, the ALU will need to perform one of these five functions. For load word and store word instructions, we use the ALU to compute the memory address by addition. For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field in the low-order bits of the instruction (see Chapter 3, page 118). For branch equal, the ALU must perform a subtraction.

We can generate the 3-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract (01) for `beq`, or determined by the operation encoded in the funct field (10). The output of the ALU control unit is a 3-bit signal that directly controls the ALU by generating one of the five 3-bit combinations shown previously.

In Figure 5.14, we show how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code. For completeness, the relationship between the ALUOp bits and the instruction opcode is also shown. Later in this chapter we will see how the ALUOp bits are generated from the main control unit.

This style of using multiple levels of decoding (i.e., the main control unit generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit) is a common implementation technique. Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially increase the speed of the control unit. Such optimizations are important, since the control unit is often performance-critical.

There are several different ways to implement the mapping from the 2-bit ALUOp field and the 6-bit funct field to the three ALU operation control bits. Because only a small number of the 64 possible values of the function field are of interest and the function field is used only when the ALUOp bits equal 10, we can use a small piece of logic that recognizes the subset of possible values and causes the correct setting of the ALU control bits.

As a step in designing this logic, it is useful to create a truth table for the interesting combinations of the function code field and the ALUOp bits, as we've done in Figure 5.15; this truth table shows how the 3-bit ALU control is set depending on these two input fields. Since the full truth table is very large (2^8 = 256 entries) and we don't care about the value of the ALU control for many of these input combinations, we show only the truth table entries for which the ALU control must have a specific value. Throughout this chapter, we will use

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

FIGURE 5.14 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we "don't care" about the value of the function code, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input.

INTEL - 1012

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

FIGURE 5.15 The truth table for the three ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

this practice of showing only the truth table entries that must be asserted and not showing those that are all zero or don't care. (This practice has a disadvantage, which we discuss in section C.2 of Appendix C.)

Because in many instances we do not care about the values of some of the inputs and to keep the tables compact, we also include "don't-care" terms. A don't-care term in this truth table (represented by an X in an input column) indicates that the output does not depend on the value of the input corresponding to that column. For example, when the ALUOp bits are 00, as in the first line of the table in Figure 5.15, we always set the ALU control to 010, independent of the function code. In this case, then, the function code inputs will be don't cares in this line of the truth table. Later, we will see examples of another type of don't-care term. If you are unfamiliar with the concept of don't-care terms, see Appendix B for more information.

Once the truth table has been constructed, it can be optimized and then turned into gates. This process is completely mechanical. Thus, rather than show the final steps here, we describe the process and the result in section C.2 of Appendix C.

Designing the Main Control Unit

Now that we have described how to design an ALU that uses the function code and a 2-bit signal as its control inputs, we can return to looking at the rest of the control. To start this process, let's identify the fields of an instruction and the control lines that are needed for the datapath we constructed in Figure 5.13 on page 354. To understand how to connect the fields of an instruction to the datapath, it is useful to review the formats of the three instruction classes: the R-type, branch, and load/store instructions. These formats are shown in Figure 5.16.

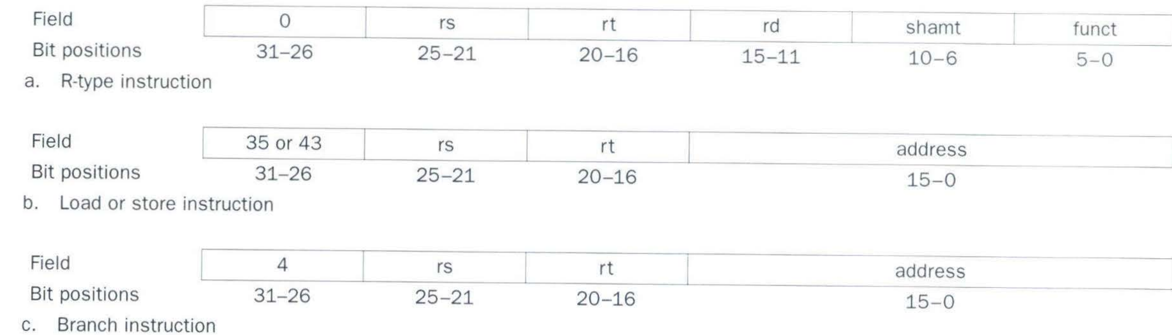


FIGURE 5.16 The three instruction classes (R-type, load and store, and branch) use two different instruction formats. The jump instructions use another format, which we will discuss shortly. (a) Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. The ALU function is in the funct field and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are `add`, `sub`, `and`, `or`, and `slt`. The shamt field is used only for shifts; we will ignore it in this chapter. (b) Instruction format for load (opcode = 35_{ten}) and store (opcode = 43_{ten}) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory. (c) Instruction format for branch equal (opcode = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC to compute the branch target address.

There are several major observations about this instruction format that we will rely on:

- The op field, also called the *opcode*, is always contained in bits 31–26. We will refer to this field as Op[5-0].
- The two registers to be read are always specified by the rs and rt fields, at positions 25–21 and 20–16. This is true for the R-type instructions, branch equal, and for store.
- The base register for load and store instructions is always in bit positions 25–21 (rs).
- The 16-bit offset for branch equal, load, and store is always in positions 15–0.
- The destination register is in one of two places. For a load it is in bit positions 20–16 (rt), while for an R-type instruction it is in bit positions 15–11 (rd). Thus we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

Using this information, we can add the instruction labels and extra multiplexor (for the Write register number input of the register file) to the simple datapath. Figure 5.17 shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the

INTEL - 1012

FIGURE 5.17 The datapath of Figure 5.13 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20–16).	The register destination number for the Write register comes from the rd field (bits 15–11).
RegWrite	None	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 5.18 The effect of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. The clock is never gated externally to a state element, since this can create timing problems. (See Appendix B for further discussion of this problem.)

control signals for the multiplexors. Since all the multiplexors have two inputs, they each require a single control line.

Figure 5.17 shows seven single-bit control lines plus the 2-bit ALUOp control signal. We have already defined how the ALUOp control signal works, and it is useful to define what the seven other control signals do informally before we determine how to set these control signals during instruction execution. Figure 5.18 describes the function of these seven control lines.

Now that we have looked at the function of each of the control signals, we can look at how to set them. The control unit can set all but one of the control signals based solely on the opcode field of the instruction. The PCSrc control line is the exception. That control line should be set if the instruction is branch on equal (a decision that the control unit can make) *and* the Zero output of the ALU, which is used for equality comparison, is true. To generate the PCSrc signal, we will need to AND together a signal from the control unit, which we call *Branch*, with the Zero signal out of the ALU.

These nine control signals (seven from Figure 5.18 and two for ALUOp) can now be set on the basis of six input signals to the control unit, which are the opcode bits. The datapath with the control unit and the control signals are shown in Figure 5.19.

Before we try to write a set of equations or a truth table for the control unit, it will be useful to try to define the control function informally. Because the setting of the control lines depends only on the opcode, we define whether each control signal should be 0, 1, or don't care (X), for each of the opcode values.

INTEL - 1012

FIGURE 5.19 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus we drop the signal name in subsequent figures.

Figure 5.20 defines how the control signals should be set for each opcode; this information follows directly from Figures 5.14, 5.18, and 5.19.

Operation of the Datapath

With the information contained in Figures 5.18 and 5.20, we can design the control unit logic, but before we do that, let's look at how each instruction uses the datapath. In the next few figures, we show the flow of three different instruction classes through the datapath. The asserted control signals and active datapath elements are highlighted in each of these. Note that a multiplexor whose control is 0 has a definite action, even if its control line is not highlighted. Multiple-bit control signals are highlighted if any constituent signal is asserted.

Let's begin with an R-type instruction, such as `add $t1,$t2,$t3`. Rather than looking at the entire datapath as one piece of combinational logic, it is easier to think of an instruction executing in a series of steps, focusing our attention on the portion of the datapath associated with each step. There are four steps to execute an R-type instruction:

1. An instruction is fetched from the instruction memory and the PC is incremented. Figure 5.21 shows this first step. The active units and asserted control lines are highlighted. The same format is followed for the next three steps.

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

FIGURE 5.20 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of the table corresponds to the R-format instructions (`add`, `sub`, `and`, `or`, and `slt`). For all these instructions, the source register fields are rs and rt and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set. Furthermore, an R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field. The second and third rows of this table give the control signal settings for `lw` and `sw`. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register. The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0—since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

INTEL - 1012

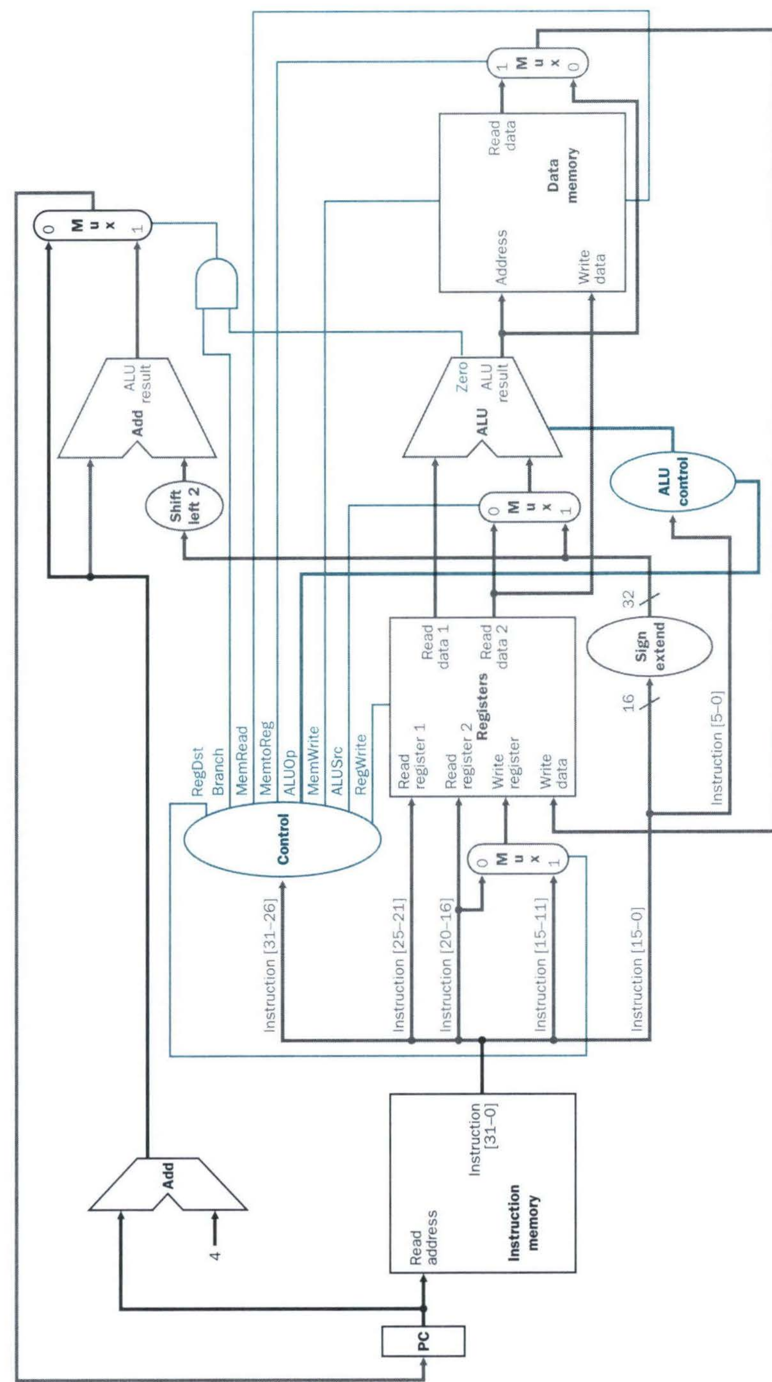


FIGURE 5.21 The first step of an R-type instruction performs a fetch from instruction memory and increments the PC. The portions active in this step are highlighted; the light portions are not active at this step, though some will be active later in the cycle.

2. Two registers, `$t2` and `$t3`, are read from the register file as shown in Figure 5.22 on page 364. The main control unit computes the setting of the control lines during this step also.

3. The ALU operates on the data read from the register file, using the function code (bits 5–0, which is the funct field, of the instruction) to generate the ALU function. Figure 5.23 on page 365 shows the operation of this step.

4. The result from the ALU is written into the register file using bits 15–11 of the instruction to select the destination register (`$t1`). Figure 5.24 on page 366 shows the final step added to the previous three.

Remember that this implementation is combinational. That is, it is not really a series of four distinct steps. The datapath really operates in a single clock cycle, and the signals within the datapath can vary unpredictably during the clock cycle. The signals stabilize roughly in the order of the steps given above because the flow of information follows this order. Thus, Figure 5.24 shows not only the action of the last step, but essentially the operation of the entire datapath when the clock cycle actually ends.

We can illustrate the execution of a load word, such as

```
lw $t1, offset($t2)
```

in a style similar to Figure 5.24. Figure 5.25 on page 368 shows the active functional units and asserted control lines for a load. We can think of a load instruction as operating in five steps (similar to the R-type executed in four):

1. An instruction is fetched from the instruction memory and the PC is incremented.
2. A register (`$t2`) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (`offset`).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file; the register destination is given by bits 20–16 of the instruction (`$t1`).

INTEL - 1012

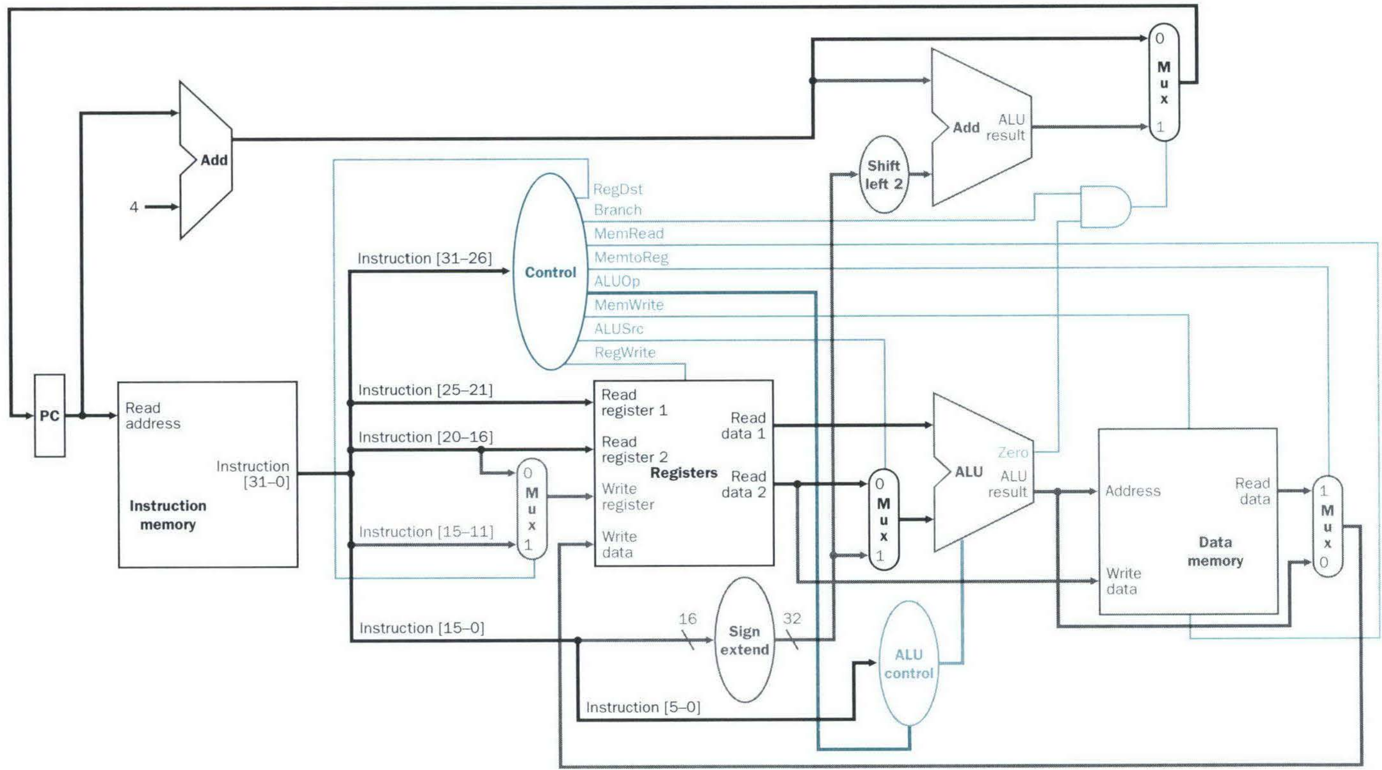


FIGURE 5.22 The second phase in the execution of R-type instructions reads the two source registers from the register file. The main control unit also uses the opcode field to determine the control line setting. These units become active in addition to the units active during the instruction fetch portion, shown in Figure 5.21.

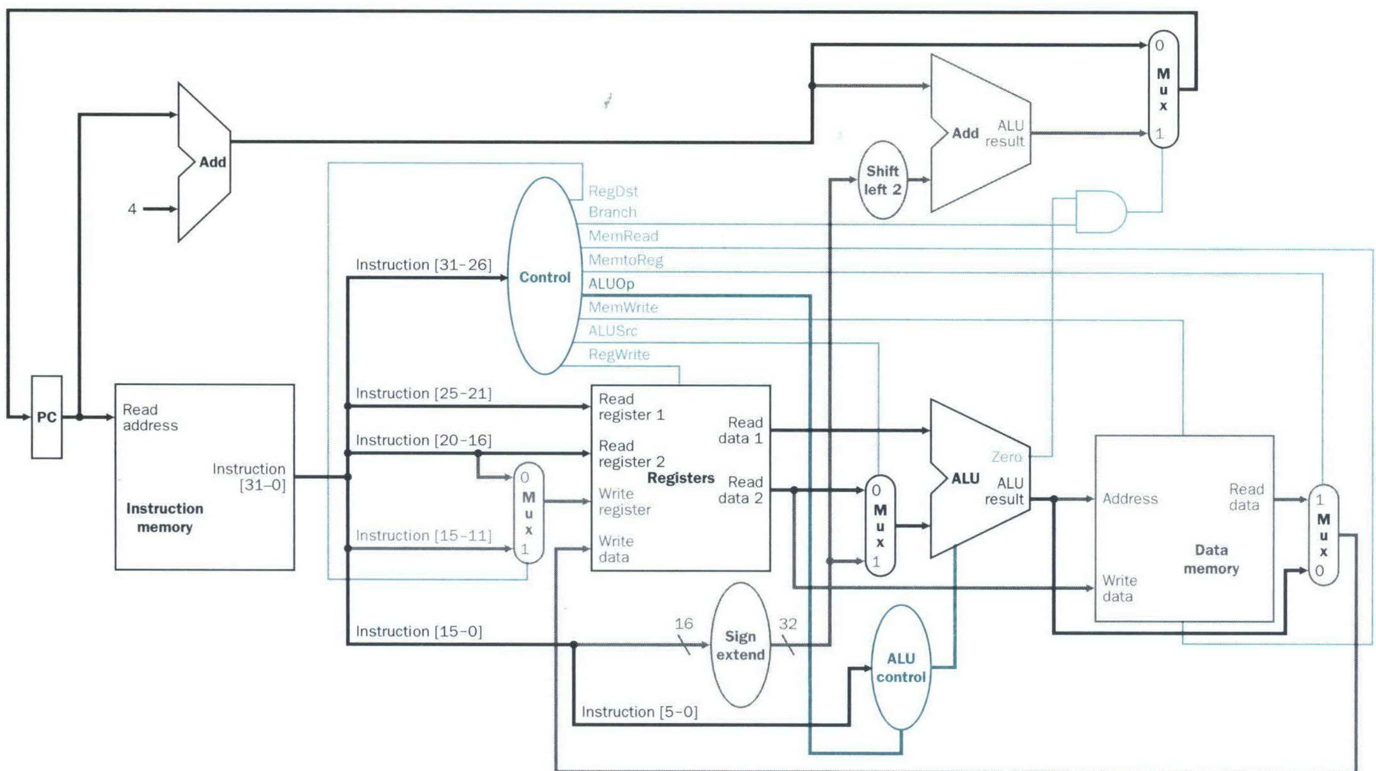


FIGURE 5.23 The third phase of execution for R-type instructions involves the ALU operating on the register data operands. The control line values are all set, and the ALU control has been computed. The ALU operates on the data.

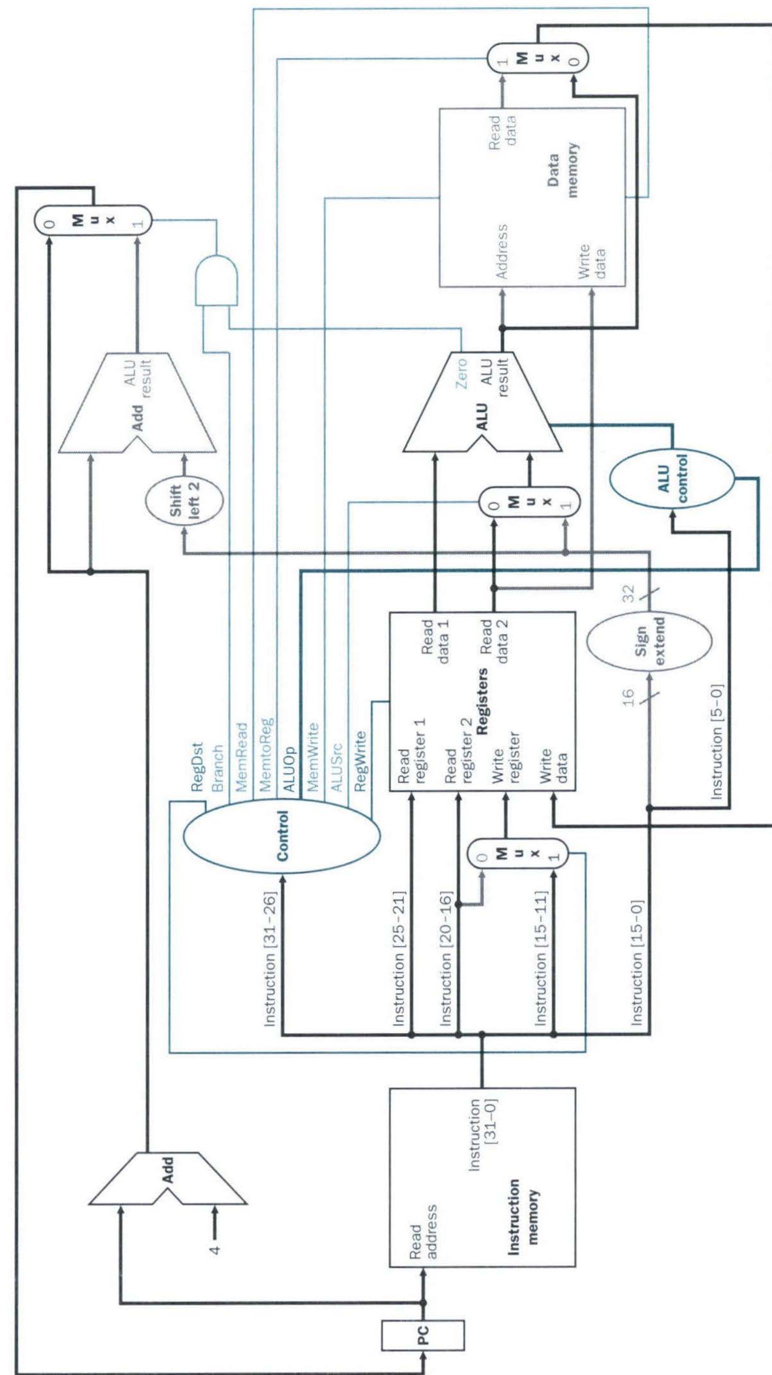


FIGURE 5.24 The final step in an R-type instruction, writing the result, is added to the active units shown for the previous three steps in Figure 5.23. The PC is also updated at the end of this phase. Because the datapath is combinational, this step shows all the active units and asserted control lines when they are stable. Observe that if the instruction is one that uses the same register as both an input and output (such as `add R1,R1,R1`) it works correctly: the value read from the registers is the value of `R1` written at the end of some earlier clock cycle, while the value to be written into the registers by the instruction is not actually written into the register until the clock edge at the end of the current clock cycle.

Finally, we can show the operation of the branch-on-equal instruction, such as `beq $t1,$t2,offset`, in the same fashion. It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address. Figure 5.26 shows the four steps in execution:

1. An instruction is fetched from the instruction memory and the PC is incremented.
2. Two registers, `$t1` and `$t2`, are read from the register file.
3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (`offset`) shifted left by two; the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

In the next section, we will examine machines that are truly sequential, namely, those in which each of these steps is a distinct clock cycle.

Finalizing the Control

Now that we have seen how the instructions operate in steps, let's continue with the control implementation. The control function can be precisely defined using the contents of Figure 5.20 on page 361. The outputs are the control lines, the input is the 6-bit opcode field, Op [5–0]. Thus we can create a truth table for each of the outputs. Before doing so, let's write down the encoding for each of the opcodes of interest in Figure 5.20, both as a decimal number and as a series of bits that are input to the control unit:

Name	Opcode in decimal	Opcode in binary					
		Op5	Op4	Op3	Op2	Op1	Op0
R-format	0_{ten}	0	0	0	0	0	0
`lw`	35_{ten}	1	0	0	0	1	1
`sw`	43_{ten}	1	0	1	0	1	1
`beq`	4_{ten}	0	0	0	1	0	0

INTEL - 1012

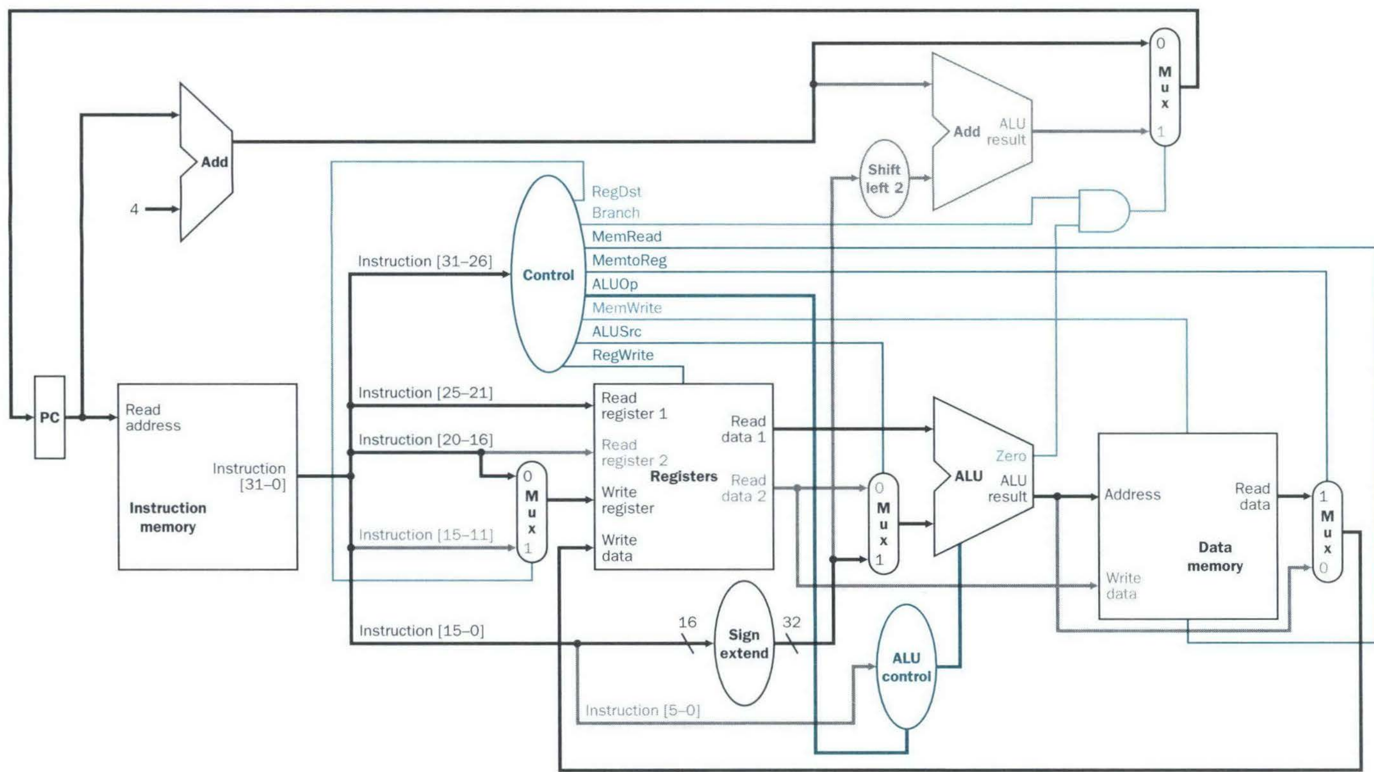


FIGURE 5.25 The operation of a load instruction with the simple datapath control scheme. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occur.

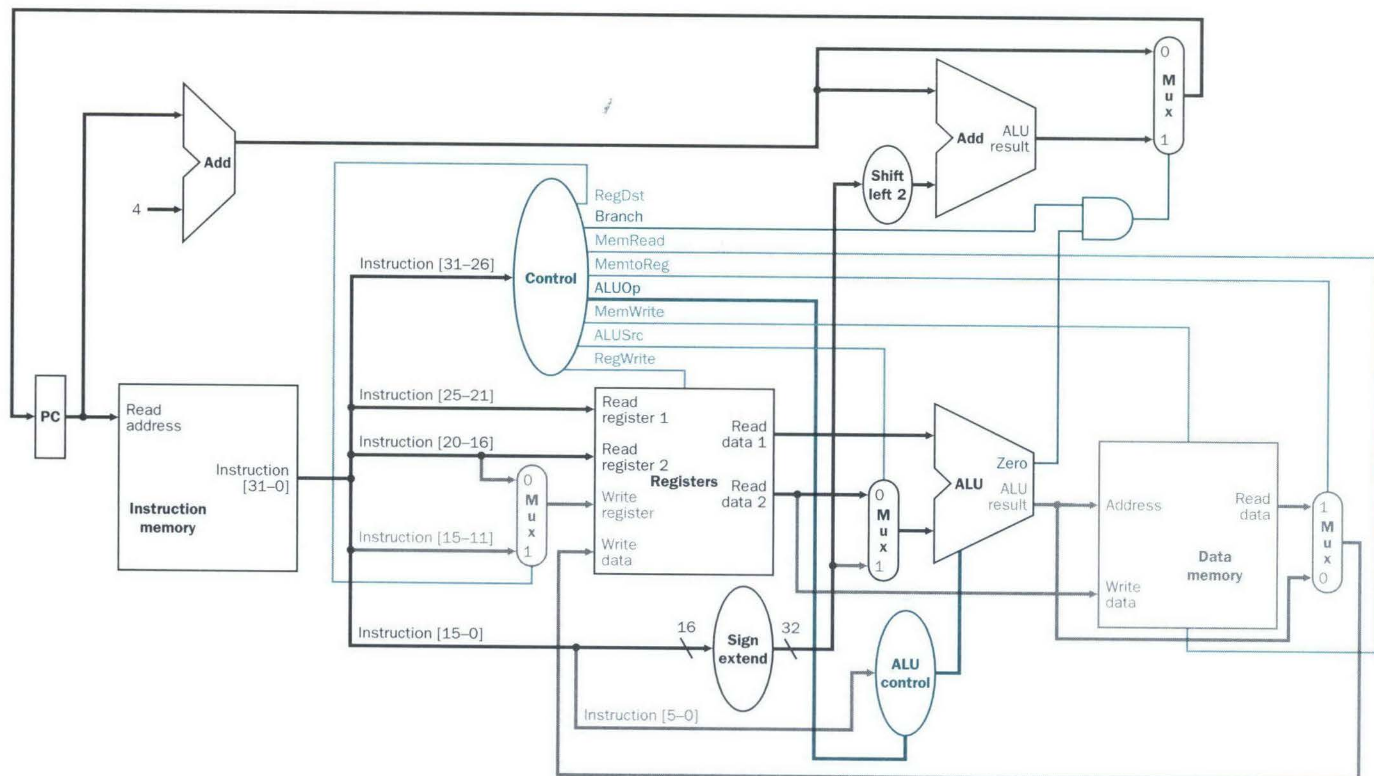


FIGURE 5.26 The datapath in operation for a branch equal instruction. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.

INTEL - 1012

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURE 5.27 The control function for the simple single-cycle implementation is completely specified by this truth table. The top half of the table gives the combinations of input signals that correspond to the four opcodes that determine the control output settings. (Remember that Op [5–0] corresponds to bits 31–26 of the instruction, which is the op field.) The bottom portion of the table gives the outputs. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression $\overline{Op5} \cdot \overline{Op2}$, since this is sufficient to distinguish the R-format instructions from `lw`, `sw`, and `beq`. We do not take advantage of this simplification, since the rest of the MIPS opcodes are used in a full implementation.

Using this information, we can now describe the logic in the control unit in one large truth table that combines all the outputs, as in Figure 5.27. It completely specifies the control function, and we can implement it directly in gates in an automated fashion. We show this final step in section C.2 in Appendix C.

Now, let's add the jump instruction to show how the basic datapath and control can be extended to handle other instructions in the instruction set.

Example

Implementing Jumps

Figure 5.19 on page 360 shows the implementation of many of the instructions we looked at in Chapter 3. One class of instructions missing is that of the jump instruction. Extend the datapath and control of Figure 5.19 to include the jump instruction. Describe how to set any new control lines.

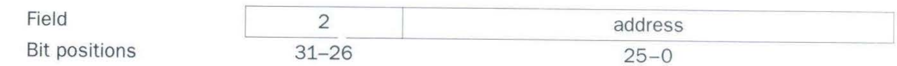


FIGURE 5.28 Instruction format for the jump instruction (opcode = 2). The destination address for a jump instruction is formed by concatenating the upper 4 bits of the current PC + 4 to the 26-bit address field in the jump instruction and adding 00 as the 2 low-order bits.

Answer

The jump instruction looks somewhat like a branch instruction but computes the target PC differently and is not conditional. Like a branch, the low-order 2 bits of a jump address are always 00_{two}. The next lower 26 bits of this 32-bit address come from the 26-bit immediate field in the instruction, as shown in Figure 5.28. The upper 4 bits of the address that should replace the PC come from the PC of the jump instruction plus four. Thus, we can implement a jump by storing into the PC the concatenation of

- the upper 4 bits of the current PC + 4 (these are bits 31–28 of the sequentially following instruction address)
- the 26-bit immediate field of the jump instruction
- the bits 00_{two}

Figure 5.29 shows the addition of the control for jump added to Figure 5.19. An additional multiplexor is used to select the source for the new PC value, which is either the incremented PC (PC + 4), the branch target PC, or the jump target PC. One additional control signal is needed for the additional multiplexor. This control signal, called *Jump,* is asserted only when the instruction is a jump—that is, when the opcode is 2.

Why a Single-Cycle Implementation Is Not Used

Although the single-cycle design will work correctly, it would not be used in modern designs because it is inefficient. To see why this is so notice that the clock cycle must have the same length for every instruction in this single-cycle design, and the CPI (see Chapter 2) will therefore be 1. Of course, the clock cycle is determined by the longest possible path in the machine. This path is almost certainly a load instruction, which uses five functional units in series: the instruction memory, the register file, the ALU, the data memory, and the register file. Although the CPI is 1, the overall performance of a single-cycle implementation is not likely to be very good, since several of the instruction classes could fit in a shorter clock cycle.

INTEL - 1012

FIGURE 5.29 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

Performance of Single-Cycle Machines

Example

Assume that the operation times for the major functional units in this implementation are the following:

- Memory units: 2 nanoseconds (ns)
- ALU and adders: 2 ns
- Register file (read or write): 1 ns

Assuming that the multiplexors, control unit, PC accesses, sign extension unit, and wires have no delay, which of the following implementations would be faster and by how much?

1. An implementation in which every instruction operates in 1 clock cycle of a fixed length.
2. An implementation where every instruction executes in 1 clock cycle using a variable-length clock, which for each instruction is only as long as it needs to be. (Such an approach is not terribly practical, but it will allow us to see what is being sacrificed when all the instructions must execute in a single clock of the same length.)

To compare the performance, assume the following instruction mix: 24% loads, 12% stores, 44% ALU instructions, 18% branches, and 2% jumps.

Answer

Let's start by comparing the CPU execution times. Recall from Chapter 2 that

$$\text{CPU execution time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

Since CPI must be 1, we can simplify this to

$$\text{CPU execution time} = \text{Instruction count} \times \text{Clock cycle time}$$

INTEL - 1012

We need only find the clock cycle time for the two implementations, since the instruction count and CPI are the same for both implementations. The critical path for the different instruction classes is as follows:

Instruction class	Functional units used by the instruction class				
ALU type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

Using these critical paths, we can compute the required length for each instruction class:

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
ALU type	2	1	2	0	1	6 ns
Load word	2	1	2	2	1	8 ns
Store word	2	1	2	2		7 ns
Branch	2	1	2			5 ns
Jump	2					2 ns

The clock cycle for a machine with a single clock for all instructions will be determined by the longest instruction, which is 8 ns. (This timing is approximate, since our timing model is quite simplistic. In reality, the timing of modern digital systems is complex, often allowing time to be borrowed from one clock cycle for use in the next.)

A machine with a variable clock will have a clock cycle that varies between 2 ns and 8 ns. We can find the average clock cycle length for a machine with a variable-length clock using the information above and the instruction frequency distribution.

Thus, the average time per instruction with a variable clock is

$$\text{CPU clock cycle} = 8 \times 24\% + 7 \times 12\% + 6 \times 44\% + 5 \times 18\% + 2 \times 2\%$$
$$= 6.3 \text{ ns}$$

Since the variable clock implementation has a shorter average clock cycle, it is clearly faster. Let's find the performance ratio:

$$\frac{\text{CPU performance}_{\text{variable clock}}}{\text{CPU performance}_{\text{single clock}}} = \frac{\text{CPU execution time}_{\text{single clock}}}{\text{CPU execution time}_{\text{variable clock}}}$$

$$= \frac{\text{IC} \times \text{CPU clock cycle}_{\text{single clock}}}{\text{IC} \times \text{CPU clock cycle}_{\text{variable clock}}}$$

$$= \frac{\text{CPU clock cycle}_{\text{single clock}}}{\text{CPU clock cycle}_{\text{variable clock}}}$$

$$= \frac{8}{6.3} = 1.27$$

The variable clock implementation would be 1.27 times faster. Unfortunately, implementing a variable-speed clock for each instruction class is extremely difficult, and the overhead for such an approach could be larger than any advantage gained. As we will see in the next section, an alternative is to use a shorter clock cycle that does less work and then vary the number of clock cycles for the different instruction classes.

The penalty for using the single-cycle design with a fixed clock cycle is significant, but might be considered acceptable for this small instruction set. However, if we tried to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design wouldn't work well at all. Let's look at an example with floating point.

Example

Performance of a Single-Cycle CPU with Floating-Point Instructions

Suppose we have a floating-point unit that requires 8 ns for a floating-point add and 16 ns for a floating-point multiply. All the other functional unit times are as in the previous example, and a floating-point instruction is like an arithmetic-logical instruction, except that it uses the floating-point ALU rather than the main ALU. Find the performance ratio between an implementation in which the clock cycle is different for each instruction class and an implementation in which all instructions have the same clock cycle time. Assume the following:

INTEL - 1012

- All loads take the same time and comprise 31% of the instructions.
- All stores take the same time and comprise 21% of the instructions.
- R-format instructions comprise 27% of the mix.
- Branches comprise 5% of the instructions, while jumps comprise 2%.
- FP add and subtract take the same time and together total 7% of the instructions.
- FP multiply and divide take the same time and together total 7% of the instructions.

Answer

From the previous example, we know that

$$\frac{\text{CPU performance}_{\text{variable clock}}}{\text{CPU performance}_{\text{single clock}}} = \frac{\text{CPU clock cycle}_{\text{single clock}}}{\text{CPU clock cycle}_{\text{variable clock}}}$$

The cycle time for the single-cycle machine will be equal to the longest instruction time, which is floating-point multiply. The time for a floating-point multiply, and thus the clock cycle, is 2 + 1 + 16 + 1 = 20 ns.

Consider a machine whose instructions have different cycle times. The time for a floating-point add instruction is 2 + 1 + 8 + 1 = 12 ns. Multiplying the cycle times by the instruction frequencies tells us that the average clock length will be

$$\text{CPU clock cycle} = 8 \times 31\% + 7 \times 21\% + 6 \times 27\% + 5 \times 5\%$$
$$+ 2 \times 2\% + 12 \times 7\% + 20 \times 7\% = 8.0 \text{ ns}$$

The improvement in performance is

$$\frac{\text{CPU performance}_{\text{variable clock}}}{\text{CPU performance}_{\text{single clock}}} = \frac{\text{CPU clock cycle}_{\text{single clock}}}{\text{CPU clock cycle}_{\text{variable clock}}}$$

$$= \frac{20}{7} = 2.9$$

A variable clock would allow us to improve performance by 2.9 times.

Similarly, if we had a machine with more powerful operations and addressing modes, instructions could vary from three or four functional unit delays to tens or even hundreds of functional unit delays. In addition, because we must assume that the clock cycle is equal to the worst-case delay for all instructions, we can't use implementation techniques that reduce the delay of the common case but do not improve the worst-case cycle time. A single-cycle implementation thus violates our key design principle of making the common case fast.

In addition, with this single-cycle implementation, each functional unit can be used only once per clock; therefore, some functional units must be duplicated, raising the cost of the implementation. A single-cycle design is inefficient both in its performance and in its hardware cost!

We can avoid these difficulties by using implementation techniques that have a shorter clock cycle—derived from the basic functional unit delays—and that require multiple clock cycles for each instruction. The next section explores this alternative implementation scheme. In Chapter 6, we'll look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath, but is much more efficient. Pipelining gains efficiency by overlapping the execution of multiple instructions, increasing hardware utilization and improving performance.

5.4 A Multicycle Implementation

In an earlier example, we broke each instruction into a series of steps corresponding to the functional unit operations that were needed. We can use these steps to create a *multicycle implementation*. In a multicycle implementation, each *step* in the execution will take 1 clock cycle. The multicycle implementation allows a functional unit to be used more than once per instruction, as long as it is used on different clock cycles. This sharing can help reduce the amount of hardware required. The ability to allow instructions to take different numbers of clock cycles and the ability to share functional units within the execution of a single instruction are the major advantages of a multicycle design. Figure 5.30 shows the abstract version of the multicycle datapath. Comparing this to the datapath for the single-cycle version shown in Figure 5.13 on page 354, we can see the following differences:

- A single memory unit is used for both instructions and data.
- There is a single ALU, rather than an ALU and two adders.
- One or more registers are added after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle.

INTEL - 1012

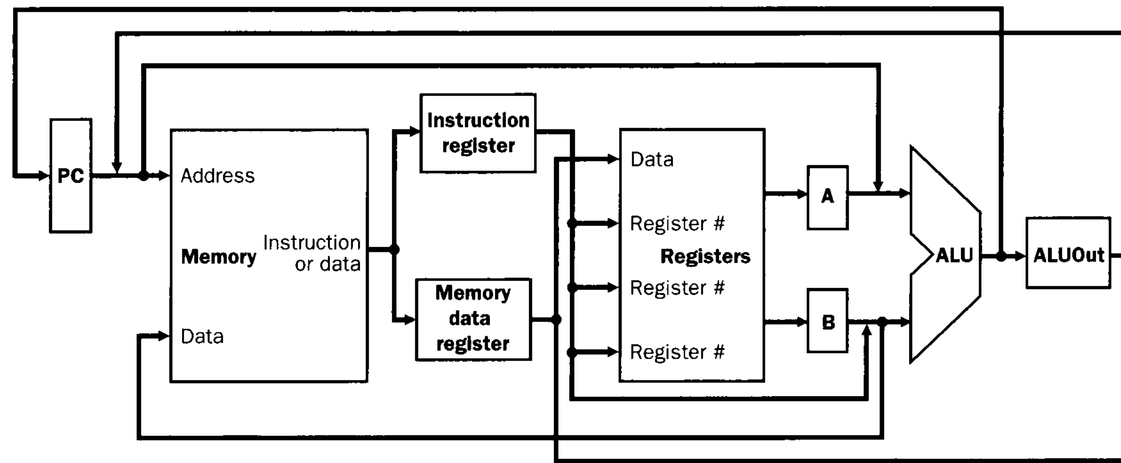


FIGURE 5.30 The high-level view of the multicycle datapath. This picture shows the key elements of the datapath: a shared memory unit, a single ALU shared among instructions, and the connections among these shared units. The use of shared functional units requires the addition or widening of multiplexors as well as new temporary registers that hold data between clock cycles of the same instruction. The additional registers are the Instruction register (IR), the Memory data register (MDR), A, B, and ALUOut.

At the end of a clock cycle, all data that are used in subsequent clock cycles must be stored in a state element. Data used by *subsequent instructions* in a later clock cycle is stored into one of the programmer-visible state elements (i.e., the register file, the PC, or the memory). In contrast, data used by the *same instruction* in a later cycle must be stored into one of these additional registers.

Thus the position of the additional registers is determined by the two factors: what combinational units will fit in a clock cycle and what data are needed in later cycles implementing the instruction. In this multicycle design, we assume that the clock cycle can accommodate at most one of the following operations: a memory access, a register file access (two reads or one write), or an ALU operation. Thus any data produced by one of these three functional units (the memory, the register file, or the ALU) must be saved into a temporary register for use on a later cycle.

The following temporary registers are added to meet these requirements:

- The Instruction register (IR) and the Memory data register (MDR) are added to save the output of the memory for an instruction read and a data read, respectively. Two separate registers are used, since, as will be clear shortly, both values are needed during the same clock cycle.

- The A and B registers are used to hold the register operand values read from the register file.
- The ALUOut register holds the output of the ALU.

All the registers except the IR hold data only between a pair of adjacent clock cycles and will thus not need a write control signal. The IR needs to hold the instruction until the end of execution of that instruction, and thus will require a write control signal. This distinction will become more clear when we show the individual clock cycles for each instruction.

Because several functional units are shared for different purposes, we need both to add multiplexors and to expand existing multiplexors. For example, since one memory is used for both instructions and data, we need a multiplexor to select between the two sources for a memory address, namely the PC (for instruction access) and ALUOut (for data access).

Replacing the three ALUs of the single-cycle datapath by a single ALU means that the single ALU must accommodate all the inputs that used to go to the three different ALUs. Handling the additional inputs requires two changes to the datapath:

1. An additional multiplexor is added for the first ALU input. The multiplexor chooses between the A register and the PC.
2. The multiplexor on the second ALU input is changed from a two-way to a four-way multiplexor. The two additional inputs to the multiplexor are the constant 4 (used to increment the PC) and the sign-extended and shifted offset field (used in the branch address computation).

Figure 5.31 shows the details of the datapath with these additional multiplexors. By introducing a few registers and multiplexors, we are able to reduce the number of memory units from two to one and eliminate two adders. Since registers and multiplexors are fairly small compared to a memory unit or ALU, this could yield a substantial reduction in the hardware cost.

Because the datapath shown in Figure 5.31 takes multiple clock cycles per instruction, it will require a different set of control signals. The programmer-visible state units (the PC, the memory, and the registers) as well as the IR will need write control signals. The memory will also need a read signal. We can use the ALU control unit from the single-cycle datapath (see Figures 5.15 and Appendix C) to control the ALU here as well. Finally, each of the two-input multiplexors requires a single control line, while the four-input multiplexor requires two control lines. Figure 5.32 shows the datapath of Figure 5.31 with these control lines added.

INTEL - 1012

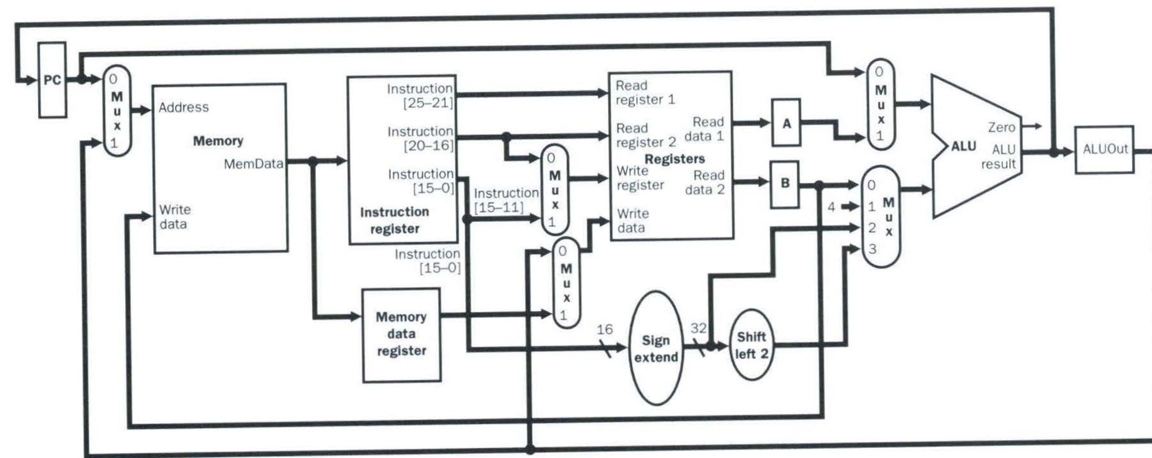


FIGURE 5.31 Multicycle datapath for MIPS handles the basic instructions. Although this datapath supports normal incrementing of the PC, a few more connections and a multiplexor will be needed for branches and jumps; we will add these shortly. The additions versus the single-clock datapath include several registers (IR, MDR, A, B, ALUOut), a multiplexor for the memory address, a multiplexor for the top ALU input, and expanding the multiplexor on the bottom ALU input into a four-way selector. These small additions allow us to remove two adders and a memory unit.

The multicycle datapath still requires additions to support branches and jumps; after these additions, we will see how the instructions are sequenced and then generate the datapath control.

With the jump instruction and branch instruction, there are three possible sources for the value to be written into the PC:

1. The output of the ALU, which is the value PC + 4 during instruction fetch. This value should be stored directly into the PC.
2. The register ALUOut, which is where we will store the address of the branch target after it is computed.
3. The lower 26 bits of the Instruction register (IR) shifted left by two and concatenated with the upper 4 bits of the incremented PC, which is the source when the instruction is a jump.

As we observed when we implemented the single-cycle control, the PC is written both unconditionally and conditionally. During a normal increment and jumps, the PC is written unconditionally. If the instruction is a conditional branch, the incremented PC is replaced with the value in ALUOut only if the two designated registers are equal. Thus the control needs two PC write signals, which we will call PCWrite and PCWriteCond.

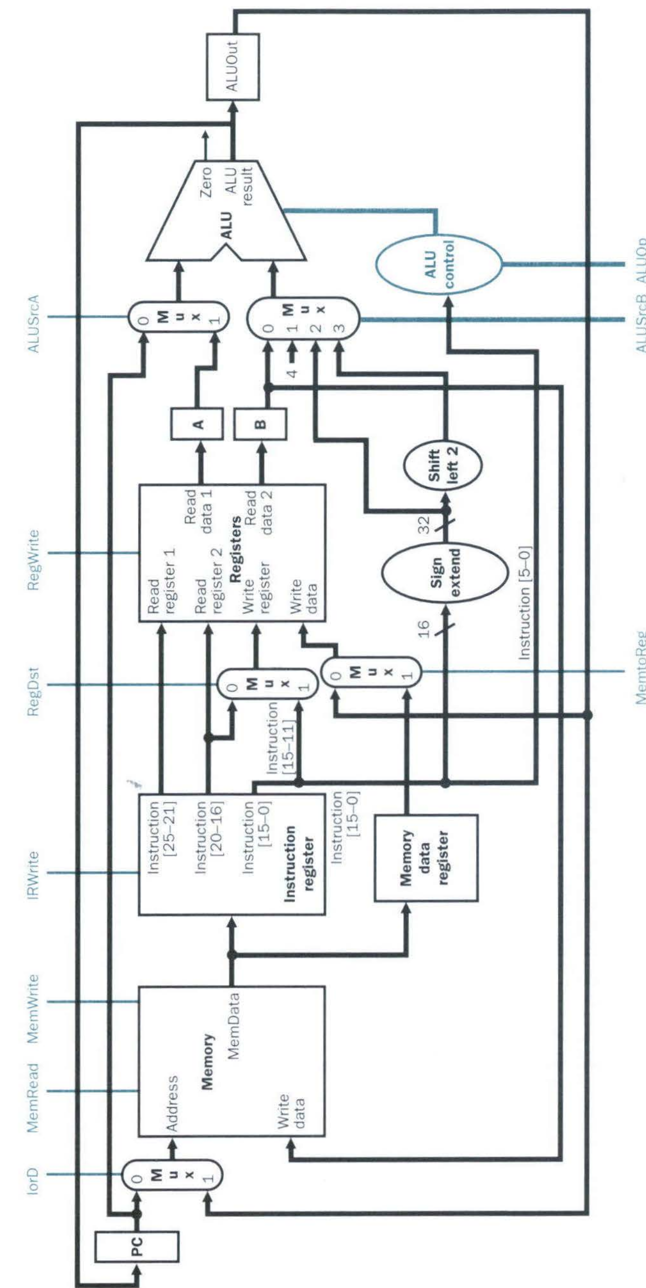


FIGURE 5.32 The multicycle datapath from Figure 5.31 with the control lines shown. The signals ALUOp and ALUSrcB are 2-bit control signals, while all the other control lines are 1-bit signals. Neither register A nor B requires a write signal, since their contents are only read on the cycle immediately after it is written. The memory data register has been added to hold the data from a load when the data returns from memory. Data from a load returning from memory cannot be written directly into the register file since the clock cycle cannot accommodate the time required for both the memory access and the register file write. The MemRead signal has been moved to the top of the memory unit to simplify the figures. The full set of datapaths and control lines for branches will be added shortly.

INTEL - 1012

We need to connect these two control signals to the PC write control. Just as we did in the single-cycle datapath, we will use a few gates to derive the PC write control signal from PCWrite, PCWriteCond, and the Zero signal of the ALU, which is used to detect if the two register operands of a `beq` are equal. To determine whether the PC should be written during a conditional branch, we AND together the Zero signal of the ALU with the PCWriteCond. The output of this AND gate is then ORed with PCWrite, which is the unconditional PC write signal. The output of this OR gate is connected to the write control signal for the PC.

Figure 5.33 shows the complete multicycle datapath and control unit, including the additional control signals and multiplexor for implementing the PC updating.

Before examining the steps to execute each instruction, let us informally examine the effect of all the control signals (just as we did for the single-cycle design in Figure 5.18 on page 359). Figure 5.34 shows what each control signal does when asserted and deasserted.

Elaboration: To reduce the number of signal lines interconnecting the functional units, designers can use *shared buses*. A shared bus is a set of lines that connect multiple units; in most cases, they include multiple sources that can place data on the bus and multiple readers of the value. Just as we reduced the number of functional units for the datapath, we can reduce the number of buses interconnecting these units by sharing the buses. For example, there are six sources coming to the ALU; however, only two of them are needed at any one time. Thus, a pair of buses can be used to hold values that are being sent to the ALU. Rather than placing a large multiplexor in front of the ALU, a designer can use a shared bus and then ensure that only one of the sources is driving the bus at any point. Although this saves signal lines, the same number of control lines will be needed to control what goes on the bus. The major drawback to using such bus structures is a potential performance penalty, since a bus is unlikely to be as fast as a point-to-point connection.

Breaking the Instruction Execution into Clock Cycles

Given the datapath in Figure 5.33, we now need to look at what should happen in each clock cycle of the multicycle execution, since this will determine what additional control signals may be needed, as well as the setting of the control signals. Our goal in breaking the execution into clock cycles should be to balance the amount of work done in each cycle, so that we minimize the clock cycle time. We can begin by breaking the execution of any instruction into a series of steps, each taking 1 clock cycle, which will be roughly balanced in length. For example, we will restrict each step to contain at most one

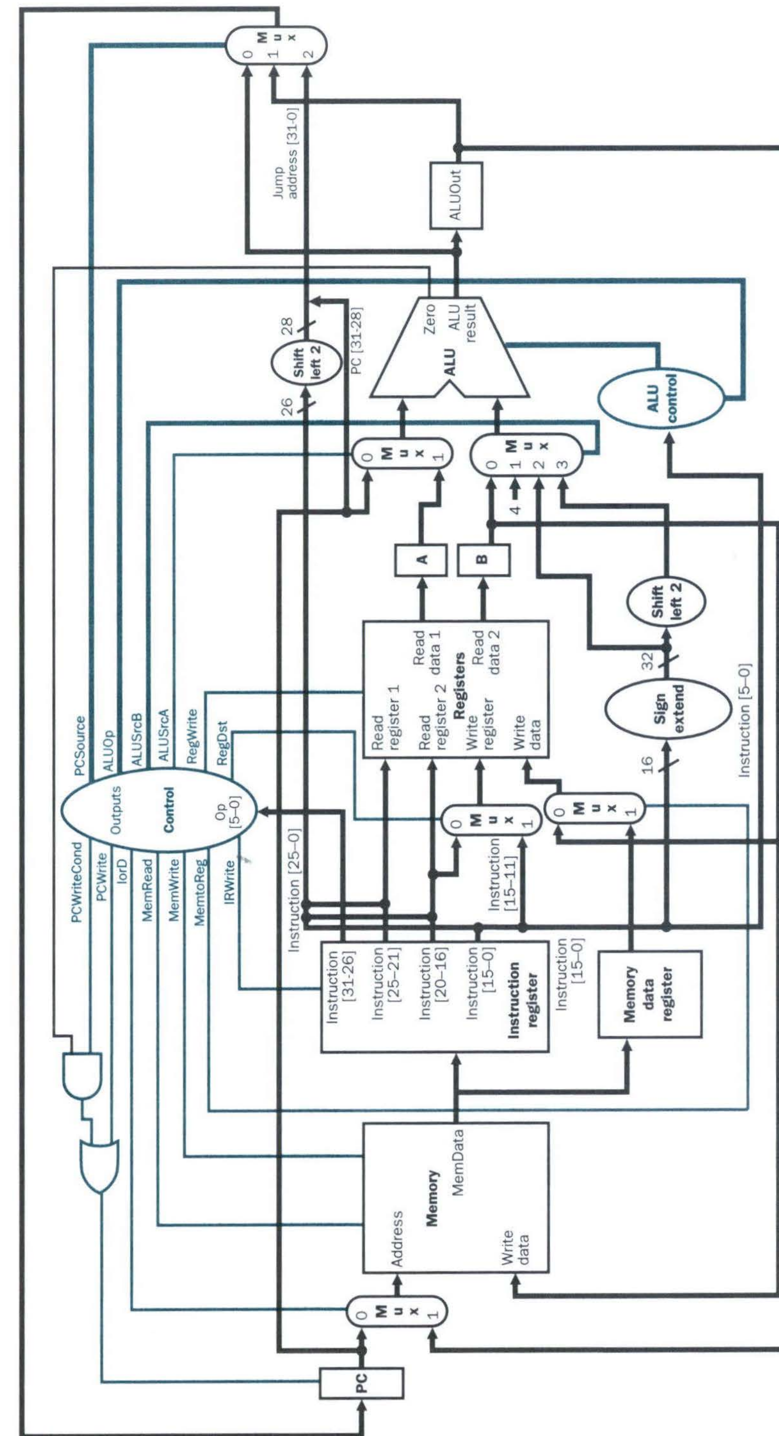


FIGURE 5.33 The complete datapath for the multicycle implementation together with the necessary control lines. The control lines of Figure 5.32 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure 5.32 include the multiplexor used to select the source of a new PC value (at the top right); two gates used to combine the PC write signals (top left); and the control signals PCSource, PCWrite, and PCWriteCond. The PCWriteCond signal is ANDed with the Zero output of the ALU to decide whether a branch should be taken; the resulting signal is ORed with the control signal PCWrite to generate the actual write control signal for the PC. In addition, the output of the IR is rearranged to send the lower 26 bits (the jump address) to the logic used to select the next PC. These 26 bits are shifted to the left by two, adding 2 low-order 0 bits; these 28 bits are then concatenated with the high-order 4 bits of the PC, which has already been incremented.

INTEL - 1012

Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None	The output of the memory is written into the IR.
PCWrite	None	The PC is written; the source is controlled by PCSource.
PCWriteCond	None	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals

Signal name	Value	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25–0] shifted left 2 bits and concatenated with PC + 4[31–28]) is sent to the PC for writing.

FIGURE 5.34 The action caused by the setting of each control signal in Figure 5.33 on page 383. The top table describes the 1-bit control signals, while the bottom table describes the 2-bit signals. Only those control lines that affect multiplexors have an action when they are deasserted. This information is similar to that in Figure 5.18 on page 359 for the single-cycle datapath, but adds several new control lines (IRWrite, PCWrite, PCWriteCond, ALUSrcB, and PCSource) and removes control lines that are no longer used or have been replaced (PCSrc, Branch, and Jump).

ALU operation, or one register file access, or one memory access. With this restriction, the clock cycle could be as short as the longest of these operations.

Recall that at the end of every clock cycle any data values that will be needed on a subsequent cycle must be stored into a register, which can be either one of the major state elements (e.g., the PC, the register file, or the memory), a temporary register written on every clock cycle (e.g., A, B, MDR, or ALUOut), or a temporary register with write control (e.g., IR). Also remember that because our design is edge-triggered, we can continue to read the current value of a register; the new value does not appear until the next clock cycle.

In the single-cycle datapath, each instruction uses a set of datapath elements to carry out its execution. Many of the datapath elements operate in series, using the output of another element as an input. Some datapath elements operate in parallel; for example, the PC is incremented and the instruction is read at the same time. A similar situation exists in the multicycle datapath. All the operations listed in one step occur in parallel within 1 clock cycle, while successive steps operate in series in different clock cycles. The limitation of one ALU operation, one memory access, and one register file access determines what can fit in one step.

Notice that we distinguish between reading from or writing into the PC or one of the stand-alone registers and reading from or writing into the register file. In the former case, the read or write is part of a clock cycle, while reading or writing a result into the register file takes an additional clock cycle. The reason for this distinction is that the register file has additional control and access overhead compared to the single stand-alone registers. Thus keeping the clock cycle short motivates dedicating separate clock cycles for register file accesses.

The potential execution steps and their actions are given below. Each instruction needs from three to five of these steps:

1. Instruction fetch step

Fetch the instruction from memory and compute the address of the next sequential instruction:

```
IR = Memory[PC];
PC = PC + 4;
```

Operation: Send the PC to the memory as the address, perform a read, and write the instruction into the Instruction register (IR), where it will be stored. Also, increment the PC by four. To implement this step, we will need to assert the control signals MemRead and IRWrite, and set IorD to 0 to select the PC as the source of the address. We also increment the PC by four in this stage, which requires setting the ALUSrcA signal to 0 (sending the PC to the ALU), the ALUSrcB signal to 01 (sending 4 to the ALU), and ALUOp to 00 (to make the ALU add). Finally, we will also want to store the incremented instruction address back into the PC, which requires setting PC source to 00 and setting PCWrite. The increment of the PC and the instruction memory access can occur in parallel. The new value of the PC is not visible until the next clock cycle. (The incremented PC will also be stored into ALUOut, but this action is benign.)

INTEL - 1012

2. Instruction decode and register fetch step

In the previous step and in this one, we do not yet know what the instruction is, so we can perform only actions that are either applicable to all instructions (such as fetching the instruction in step 1) or are not harmful, in case the instruction isn't what we think it might be. Thus, in this step we can read the two registers indicated by the rs and rt instruction fields, since it isn't harmful to read them even if it isn't necessary. The values read from the register file may be needed in later stages, so we read them from the register file and store the values into the temporary registers A and B.

We will also compute the branch target address with the ALU, which also is not harmful because we can ignore the value if the instruction turns out not to be a branch. The potential branch target is saved in ALUOut.

Performing these "optimistic" actions early has the benefit of decreasing the number of clock cycles needed to execute an instruction. We can do these optimistic actions early because of the regularity of the instruction formats. For instance, if the instruction has two register inputs, they are always in the rs and rt fields; and if the instruction is a branch, the offset is always the low-order 16 bits:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend (IR[15-0]) << 2);
```

Operation: Access the register file to read registers rs and rt and store the results into the registers A and B. Since A and B are overwritten on every cycle, the register file can be read on every cycle with the values stored into A and B. This step also computes the branch target address and stores the address in ALUOut, where it will be used on the next clock cycle if the instruction is a branch. This requires setting ALUSrcA to 0 (so that the PC is sent to the ALU), ALUSrcB to the value 11 (so that the sign-extended and shifted offset field is sent to the ALU), and ALUOp to 00 (so the ALU adds). The register file accesses and computation of branch target occur in parallel.

After this clock cycle, determining the action to take can depend on the instruction contents.

3. Execution, memory address computation, or branch completion

This is the first cycle during which the datapath operation is determined by the instruction class. In all cases, the ALU is operating on the operands prepared in the previous step, performing one of four functions, depending on the instruction class. We specify the action to be taken depending on the instruction class:

Memory reference:

```
ALUOut = A + sign-extend (IR[15-0]);
```

Operation: The ALU is adding the operands to form the memory address. This requires setting ALUSrcA to 1 (so that the first ALU input is register A) and setting ALUSrcB to 10 (so that the output of the sign extension unit is used for the second ALU input). The ALUOp signals will need to be set to 00 (causing the ALU to add).

Arithmetic-logical instruction (R-type):

```
ALUOut = A op B;
```

Operation: The ALU is performing the operation specified by the function code on the two values read from the register file in the previous cycle. This requires setting ALUSrcA = 1 and setting ALUSrcB = 00 (together causing the registers A and B to be used as the ALU inputs). The ALUOp signals will need to be set to 10 (so that the funct field is used to determine the ALU control signal settings).

Branch:

```
if (A == B) PC = ALUOut;
```

Operation: The ALU is used to do the equal comparison between the two registers read in the previous step. The Zero signal out of the ALU is used to determine whether or not to branch. This requires setting ALUSrcA = 1 and setting ALUSrcB = 00 (so that the register file outputs are the ALU inputs). The ALUOp signals will need to be set to 01 (causing the ALU to subtract) for equality testing. The PCWriteCond signal will need to be asserted to update the PC if the Zero output of the ALU is asserted. By setting PCSource to 01, the value written into the PC will come from ALUOut, which holds the branch target address computed in the previous cycle. For conditional branches that are taken, we actually write the PC twice: once from the output of the ALU (during the Instruction decode/register fetch) and once from ALUOut (during the Branch completion step). The value written into the PC last is the one used for the next instruction fetch.

Jump:

```
PC = PC [31-28] || (IR[25-0]<<2);
```

Operation: The PC is replaced by the jump address. PCSource is set to direct the jump address to the PC, and PCWrite is asserted to write the jump address into the PC.

INTEL - 1012