Code Example 6.28 CALCULATING THE BRANCH TARGET ADDRESS

MIPS Assembly Code

```
0xA4          beq  $t0, $0, else
0xA8          addi $v0, $0, 1
0xAC          addi $sp, $sp, 8
0xB0          jr   $ra
0xB4    else: addi $a0, $a0, -1
0xB8          jal  factorial
```

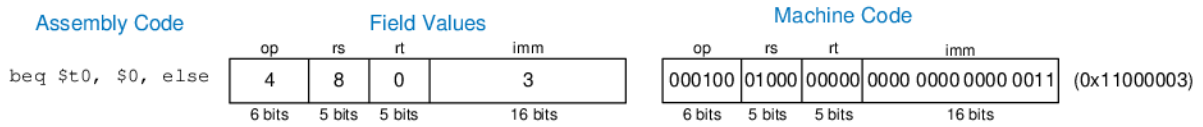


Figure 6.28 Machine code for `beq`

Example 6.8 CALCULATING THE IMMEDIATE FIELD FOR PC-RELATIVE ADDRESSING

Calculate the immediate field and show the machine code for the branch not equal (`bne`) instruction in the following program.

```
# MIPS assembly code
0x40 loop: add  $t1, $a0, $s0
0x44       lb   $t1, 0($t1)
0x48       add  $t2, $a1, $s0
0x4C       sb   $t1, 0($t2)
0x50       addi $s0, $s0, 1
0x54       bne  $t1, $0, loop
0x58       lw   $s0, 0($sp)
```

Solution: Figure 6.29 shows the machine code for the `bne` instruction. Its branch target address, 0x40, is 6 instructions behind `PC`+4 (0x58), so the immediate field is −6.

Figure 6.29 `bne` machine code

Pseudo-Direct Addressing

In *direct addressing*, an address is specified in the instruction. The jump instructions, `j` and `jal`, ideally would use direct addressing to specify a

Code Example 6.29 CALCULATING THE JUMP TARGET ADDRESS

MIPS Assembly Code

```
0x0040005C          jal     sum
...
0x004000A0  sum:    add     $v0, $a0, $a1
```

32-bit *jump target address* (*JTA*) to indicate the instruction address to execute next.

Unfortunately, the J-type instruction encoding does not have enough bits to specify a full 32-bit JTA. Six bits of the instruction are used for the `opcode`, so only 26 bits are left to encode the JTA. Fortunately, the two least significant bits, $JTA_{1:0}$, should always be 0, because instructions are word aligned. The next 26 bits, $JTA_{27:2}$, are taken from the `addr` field of the instruction. The four most significant bits, $JTA_{31:28}$, are obtained from the four most significant bits of `PC`+4. This addressing mode is called *pseudo-direct*.

Code Example 6.29 illustrates a `jal` instruction using pseudo-direct addressing. The JTA of the `jal` instruction is 0x004000A0. Figure 6.30 shows the machine code for this `jal` instruction. The top four bits and bottom two bits of the JTA are discarded. The remaining bits are stored in the 26-bit address field (`addr`).

The processor calculates the JTA from the J-type instruction by appending two 0's and prepending the four most significant bits of `PC`+4 to the 26-bit address field (`addr`).

Because the four most significant bits of the JTA are taken from `PC`+4, the jump range is limited. The range limits of branch and jump instructions are explored in Exercises 6.23 to 6.26. All J-type instructions, `j` and `jal`, use pseudo-direct addressing.

Note that the jump register instruction, `jr`, is *not* a J-type instruction. It is an R-type instruction that jumps to the 32-bit value held in register `rs`.

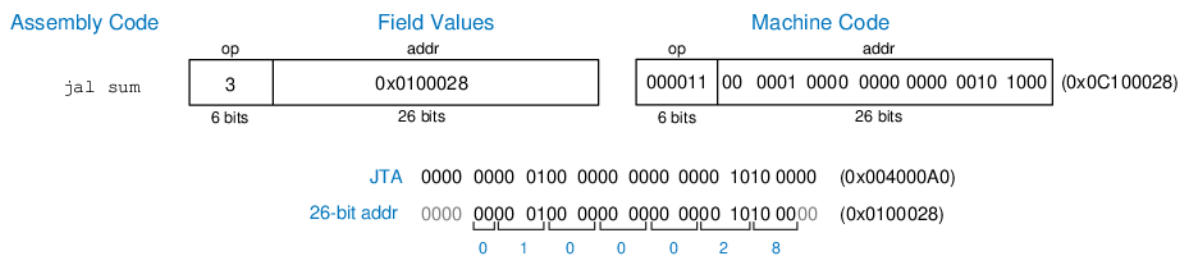


Figure 6.30 `jal` **machine code**

6.6 LIGHTS, CAMERA, ACTION: COMPILING, ASSEMBLING, AND LOADING

Up until now, we have shown how to translate short high-level code snippets into assembly and machine code. This section describes how to compile and assemble a complete high-level program and how to load the program into memory for execution.

We begin by introducing the MIPS *memory map*, which defines where code, data, and stack memory are located. We then show the steps of code execution for a sample program.

6.6.1 The Memory Map

With 32-bit addresses, the MIPS *address space* spans 2^{32} bytes = 4 gigabytes (GB). Word addresses are divisible by 4 and range from 0 to 0xFFFFFFFC. Figure 6.31 shows the MIPS memory map. The MIPS architecture divides the address space into four parts or *segments*: the text segment, global data segment, dynamic data segment, and reserved segments. The following sections describes each segment.

The Text Segment

The *text segment* stores the machine language program. It is large enough to accommodate almost 256 MB of code. Note that the four most significant bits of the address in the text space are all 0, so the `j` instruction can directly jump to any address in the program.

The Global Data Segment

The *global data segment* stores global variables that, in contrast to local variables, can be seen by all procedures in a program. Global variables

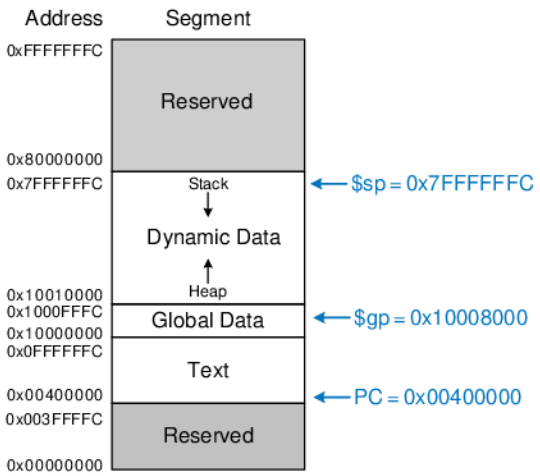


Figure 6.31 MIPS memory map

are defined at *start-up*, before the program begins executing. These variables are declared outside the main procedure in a C program and can be accessed by any procedure. The global data segment is large enough to store 64 KB of global variables.

Global variables are accessed using the global pointer (`$gp`), which is initialized to 0x100080000. Unlike the stack pointer (`$sp`), `$gp` does not change during program execution. Any global variable can be accessed with a 16-bit positive or negative offset from `$gp`. The offset is known at assembly time, so the variables can be efficiently accessed using base addressing mode with constant offsets.

The Dynamic Data Segment

The *dynamic data segment* holds the stack and the *heap*. The data in this segment are not known at start-up but are dynamically allocated and deallocated throughout the execution of the program. This is the largest segment of memory used by a program, spanning almost 2 GB of the address space.

As discussed in Section 6.4.6, the stack is used to save and restore registers used by procedures and to hold local variables such as arrays. The stack grows downward from the top of the dynamic data segment (0x7FFFFFFC) and is accessed in last-in-first-out (LIFO) order.

The heap stores data that is allocated by the program during runtime. In C, memory allocations are made by the `malloc` function; in C++ and Java, `new` is used to allocate memory. Like a heap of clothes on a dorm room floor, heap data can be used and discarded in any order. The heap grows upward from the bottom of the dynamic data segment.

If the stack and heap ever grow into each other, the program's data can become corrupted. The memory allocator tries to ensure that this never happens by returning an out-of-memory error if there is insufficient space to allocate more dynamic data.

Grace Hopper, 1906–1992. Graduated from Yale University with a Ph.D. in mathematics. Developed the first compiler while working for the Remington Rand Corporation and was instrumental in developing the COBOL programming language. As a naval officer, she received many awards, including a World War II Victory Medal and the National Defence Service Medal.

The Reserved Segments

The *reserved segments* are used by the operating system and cannot directly be used by the program. Part of the reserved memory is used for interrupts (see Section 7.7) and for memory-mapped I/O (see Section 8.5).

6.6.2 Translating and Starting a Program

Figure 6.32 shows the steps required to translate a program from a high-level language into machine language and to start executing that program. First, the high-level code is compiled into assembly code. The assembly code is assembled into machine code in an *object file*. The linker combines the machine code with object code from libraries and other files to produce an entire executable program. In practice, most compilers perform all three steps of compiling, assembling, and linking. Finally, the loader loads

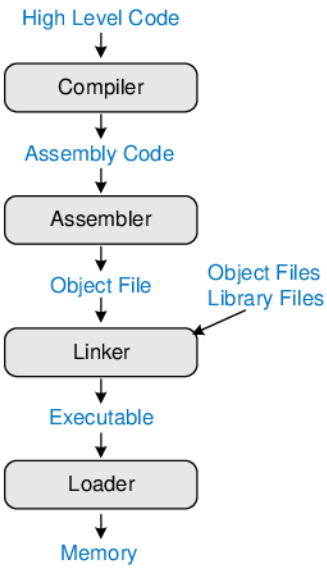


Figure 6.32 Steps for translating and starting a program

the program into memory and starts execution. The remainder of this section walks through these steps for a simple program.

Step 1: Compilation

A compiler translates high-level code into assembly language. Code Example 6.30 shows a simple high-level program with three global

Code Example 6.30 COMPILING A HIGH-LEVEL PROGRAM

High-Level Code

```
int f, g, y; // global variables



int main (void)
{
  f = 2;
  g = 3;
  y = sum (f, g);
  return y;
}



int sum (int a, int b) {
  return (a + b);
}
```

MIPS Assembly Code

```
.data
f:
g:
y:

.text
main:
  addi  $sp, $sp, -4  # make stack frame
  sw    $ra, 0($sp)   # store $ra on stack
  addi  $a0, $0, 2    # $a0 = 2
  sw    $a0, f        # f = 2
  addi  $a1, $0, 3    # $a1 = 3
  sw    $a1, g        # g = 3
  jal   sum           # call sum procedure
  sw    $v0, y        # y = sum (f, g)
  lw    $ra, 0($sp)   # restore $ra from stack
  addi  $sp, $sp, 4   # restore stack pointer
  jr    $ra           # return to operating system

sum:
  add   $v0, $a0, $a1 # $v0 = a + b
  jr    $ra           # return to caller
```

variables and two procedures, along with the assembly code produced by a typical compiler. The `.data` and `.text` keywords are *assembler directives* that indicate where the text and data segments begin. Labels are used for global variables `f`, `g`, and `y`. Their storage location will be determined by the assembler; for now, they are left as symbols in the code.

Step 2: Assembling

The assembler turns the assembly language code into an *object file* containing machine language code. The assembler makes two passes through the assembly code. On the first pass, the assembler assigns instruction addresses and finds all the *symbols*, such as labels and global variable names. The code after the first assembler pass is shown here.

```
0x00400000  main:  addi $sp, $sp, -4
0x00400004         sw   $ra, 0($sp)
0x00400008         addi $a0, $0, 2
0x0040000C         sw   $a0, f
0x00400010         addi $a1, $0, 3
0x00400014         sw   $a1, g
0x00400018         jal  sum
0x0040001C         sw   $v0, y
0x00400020         lw   $ra, 0($sp)
0x00400024         addi $sp, $sp, 4
0x00400028         jr   $ra
0x0040002C  sum:   add  $v0, $a0, $a1
0x00400030         jr   $ra
```

The names and addresses of the symbols are kept in a *symbol table*, as shown in Table 6.4 for this code. The symbol addresses are filled in after the first pass, when the addresses of labels are known. Global variables are assigned storage locations in the global data segment of memory, starting at memory address 0x10000000.

On the second pass through the code, the assembler produces the machine language code. Addresses for the global variables and labels are taken from the symbol table. The machine language code and symbol table are stored in the object file.

Table 6.4 Symbol table

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Step 3: Linking

Most large programs contain more than one file. If the programmer changes only one of the files, it would be wasteful to recompile and reassemble the other files. In particular, programs often call procedures in library files; these library files almost never change. If a file of high-level code is not changed, the associated object file need not be updated.

The job of the linker is to combine all of the object files into one machine language file called the *executable*. The linker relocates the data and instructions in the object files so that they are not all on top of each other. It uses the information in the symbol tables to adjust the addresses of global variables and of labels that are relocated.

In our example, there is only one object file, so no relocation is necessary. Figure 6.33 shows the executable file. It has three sections: the executable file header, the text segment, and the data segment. The executable file header reports the text size (code size) and data size (amount of globally declared data). Both are given in units of bytes. The text segment gives the instructions and the addresses where they are to be stored.

The figure shows the instructions in human-readable format next to the machine code for ease of interpretation, but the executable file includes only machine instructions. The data segment gives the address of each global variable. The global variables are addressed with respect to the base address given by the global pointer, `$gp`. For example, the

Executable file header	Text Size	Data Size	
	0x34 (52 bytes)	0xC (12 bytes)	
Text segment	**Address**	**Instruction**	
	0x00400000	0x23BDFFFC	addi $sp, $sp, –4
	0x00400004	0xAFBF0000	sw $ra, 0 ($sp)
	0x00400008	0x20040002	addi $a0, $0, 2
	0x0040000C	0xAF848000	sw $a0, 0x8000 ($gp)
	0x00400010	0x20050003	addi $a1, $0, 3
	0x00400014	0xAF858004	sw $a1, 0x8004 ($gp)
	0x00400018	0x0C10000B	jal 0x0040002C
	0x0040001C	0xAF828008	sw $v0, 0x8008 ($gp)
	0x00400020	0x8FBF0000	lw $ra, 0 ($sp)
	0x00400024	0x23BD0004	addi $sp, $sp, –4
	0x00400028	0x03E00008	jr $ra
	0x0040002C	0x00851020	add $v0, $a0, $a1
	0x00400030	0x03E00008	jr $ra
Data segment	**Address**	**Data**	
	0x10000000	f	
	0x10000004	g	
	0x10000008	y	

Figure 6.33 Executable

first store instruction, `sw $a0, 0x8000($gp)`, stores the value 2 to the global variable `f`, which is located at memory address 0x10000000. Remember that the offset, 0x8000, is a 16-bit signed number that is sign-extended and added to the base address, `$gp`. So, `$gp` + 0x8000 = 0x10008000 + 0xFFFF8000 = 0x10000000, the memory address of variable `f`.

Step 4: Loading

The operating system loads a program by reading the text segment of the executable file from a storage device (usually the hard disk) into the text segment of memory. The operating system sets `$gp` to 0x10008000 (the middle of the global data segment) and `$sp` to 0x7FFFFFFC (the top of the dynamic data segment), then performs a `jal 0x00400000` to jump to the beginning of the program. Figure 6.34 shows the memory map at the beginning of program execution.

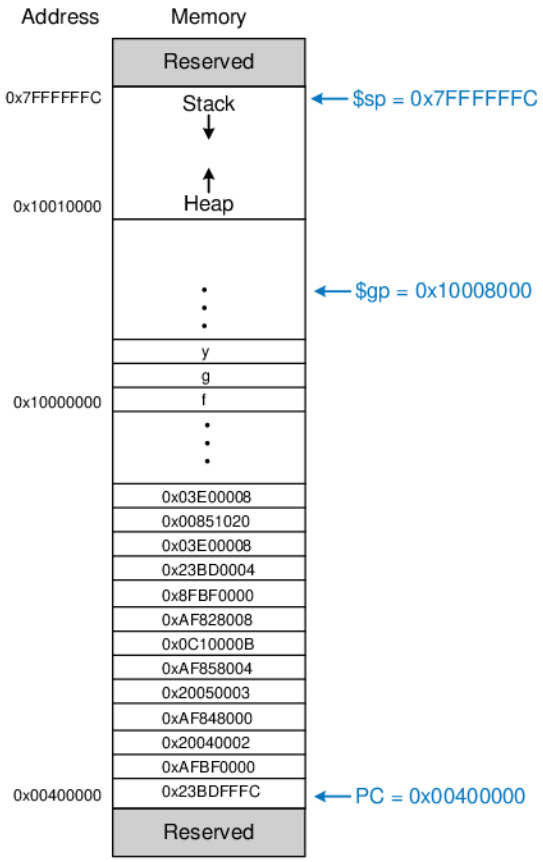


Figure 6.34 Executable loaded in memory

6.7 ODDS AND ENDS*

This section covers a few optional topics that do not fit naturally elsewhere in the chapter. These topics include pseudoinstructions, exceptions, signed and unsigned arithmetic instructions, and floating-point instructions.

6.7.1 Pseudoinstructions

If an instruction is not available in the MIPS instruction set, it is probably because the same operation can be performed using one or more existing MIPS instructions. Remember that MIPS is a reduced instruction set computer (RISC), so the instruction size and hardware complexity are minimized by keeping the number of instructions small.

However, MIPS defines *pseudoinstructions* that are not actually part of the instruction set but are commonly used by programmers and compilers. When converted to machine code, pseudoinstructions are translated into one or more MIPS instructions.

Table 6.5 gives examples of pseudoinstructions and the MIPS instructions used to implement them. For example, the load immediate pseudoinstruction (`li`) loads a 32-bit constant using a combination of `lui` and `ori` instructions. The `multiply` pseudoinstruction (`mul`) provides a three-operand multiply, multiplying two registers and putting the 32 least significant bits of the result into a third register. The no operation pseudoinstruction (`nop`, pronounced "no op") performs no operation. The `PC` is incremented by 4 upon its execution. No other registers or memory values are altered. The machine code for the `nop` instruction is 0x00000000.

Some pseudoinstructions require a temporary register for intermediate calculations. For example, the pseudoinstruction `beq $t2,` $\text{imm}_{15:0}$`, Loop` compares `$t2` to a 16-bit immediate, $\text{imm}_{15:0}$. This pseudoinstruction

Table 6.5 Pseudoinstructions

Pseudoinstruction	Corresponding MIPS Instructions
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`mul $s0, $s1, $s2`	`mult $s1, $s2` `mflo $s0`
`clear $t0`	`add $t0, $0, $0`
`move $s1, $s2`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

Table 6.6 Pseudoinstruction using `$at`

Pseudoinstruction	Corresponding MIPS Instructions
`beq $t2, imm15:0, Loop`	`addi $at, $0, imm15:0` `beq $t2, $at, Loop`

requires a temporary register in which to store the 16-bit immediate. Assemblers use the assembler register, `$at`, for such purposes. Table 6.6 shows how the assembler uses `$at` in converting a pseudoinstruction to real MIPS instructions. We leave it as Exercise 6.31 to implement other pseudoinstructions such as rotate left (`rol`) and rotate right (`ror`).

6.7.2 Exceptions

An *exception* is like an unscheduled procedure call that jumps to a new address. Exceptions may be caused by hardware or software. For example, the processor may receive notification that the user pressed a key on a keyboard. The processor may stop what it is doing, determine which key was pressed, save it for future reference, then resume the program that was running. Such a hardware exception triggered by an input/output (I/O) device such as a keyboard is often called an *interrupt*. Alternatively, the program may encounter an error condition such as an undefined instruction. The program then jumps to code in the *operating system* (*OS*), which may choose to terminate the offending program. Software exceptions are sometimes called *traps*. Other causes of exceptions include division by zero, attempts to read nonexistent memory, hardware malfunctions, debugger breakpoints, and arithmetic overflow (see Section 6.7.3).

The processor records the cause of an exception and the value of the `PC` at the time the exception occurs. It then jumps to the *exception handler* procedure. The exception handler is code (usually in the OS) that examines the cause of the exception and responds appropriately (by reading the keyboard on a hardware interrupt, for example). It then returns to the program that was executing before the exception took place. In MIPS, the exception handler is always located at 0x80000180. When an exception occurs, the processor always jumps to this instruction address, regardless of the cause.

The MIPS architecture uses a special-purpose register, called the `Cause` register, to record the cause of the exception. Different codes are used to record different exception causes, as given in Table 6.7. The exception handler code reads the `Cause` register to determine how to handle the exception. Some other architectures jump to a different exception handler for each different cause instead of using a `Cause` register.

MIPS uses another special-purpose register called the Exception Program Counter (`EPC`) to store the value of the `PC` at the time an exception

Table 6.7 Exception cause codes

Exception	Cause
hardware interrupt	0x00000000
system call	0x00000020
breakpoint/divide by 0	0x00000024
undefined instruction	0x00000028
arithmetic overflow	0x00000030

takes place. The processor returns to the address in `EPC` after handling the exception. This is analogous to using `$ra` to store the old value of the `PC` during a `jal` instruction.

The `EPC` and `Cause` registers are not part of the MIPS register file. The `mfc0` (move from coprocessor 0) instruction copies these and other special-purpose registers into one of the general purpose registers. Coprocessor 0 is called the *MIPS processor control*; it handles interrupts and processor diagnostics. For example, `mfc0 $t0, Cause` copies the `Cause` register into `$t0`.

The `syscall` and `break` instructions cause traps to perform system calls or debugger breakpoints. The exception handler uses the `EPC` to look up the instruction and determine the nature of the system call or breakpoint by looking at the fields of the instruction.

`$k0` and `$k1` are included in the MIPS register set. They are reserved by the OS for exception handling. They do not need to be saved and restored during exceptions.

In summary, an exception causes the processor to jump to the exception handler. The exception handler saves registers on the stack, then uses `mfc0` to look at the cause and respond accordingly. When the handler is finished, it restores the registers from the stack, copies the return address from `EPC` to `$k0` using `mfc0`, and returns using `jr $k0`.

6.7.3 Signed and Unsigned Instructions

Recall that a binary number may be signed or unsigned. The MIPS architecture uses two's complement representation of signed numbers. MIPS has certain instructions that come in signed and unsigned flavors, including addition and subtraction, multiplication and division, set less than, and partial word loads.

Addition and Subtraction

Addition and subtraction are performed identically whether the number is signed or unsigned. However, the interpretation of the results is different.

As mentioned in Section 1.4.6, if two large signed numbers are added together, the result may incorrectly produce the opposite sign. For example, adding the following two huge positive numbers gives a negative

result: 0x7FFFFFFF + 0x7FFFFFFF = 0xFFFFFFFE = −2. Similarly, adding two huge negative numbers gives a positive result, 0x80000001 + 0x80000001 = 0x00000002. This is called arithmetic *overflow*.

The C language ignores arithmetic overflows, but other languages, such as Fortran, require that the program be notified. As mentioned in Section 6.7.2, the MIPS processor takes an exception on arithmetic overflow. The program can decide what to do about the overflow (for example, it might repeat the calculation with greater precision to avoid the overflow), then return to where it left off.

MIPS provides signed and unsigned versions of addition and subtraction. The signed versions are `add`, `addi`, and `sub`. The unsigned versions are `addu`, `addiu`, and `subu`. The two versions are identical except that signed versions trigger an exception on overflow, whereas unsigned versions do not. Because C ignores exceptions, C programs technically use the unsigned versions of these instructions.

Multiplication and Division

Multiplication and division behave differently for signed and unsigned numbers. For example, as an unsigned number, 0xFFFFFFFF represents a large number, but as a signed number it represents −1. Hence, 0xFFFFFFFF × 0xFFFFFFFF would equal 0xFFFFFFFE00000001 if the numbers were unsigned but 0x0000000000000001 if the numbers were signed.

Therefore, multiplication and division come in both signed and unsigned flavors. `mult` and `div` treat the operands as signed numbers. `multu` and `divu` treat the operands as unsigned numbers.

Set Less Than

Set less than instructions can compare either two registers (`slt`) or a register and an immediate (`slti`). Set less than also comes in signed (`slt` and `slti`) and unsigned (`sltu` and `sltiu`) versions. In a signed comparison, 0x80000000 is less than any other number, because it is the most negative two's complement number. In an unsigned comparison, 0x80000000 is greater than 0x7FFFFFFF but less than 0x80000001, because all numbers are positive.

Beware that `sltiu` sign-extends the immediate before treating it as an unsigned number. For example, `sltiu $s0, $s1, 0x8042` compares `$s1` to 0xFFFF8042, treating the immediate as a large positive number.

Loads

As described in Section 6.4.5, byte loads come in signed (`lb`) and unsigned (`lbu`) versions. `lb` sign-extends the byte, and `lbu` zero-extends the byte to fill the entire 32-bit register. Similarly, MIPS provides signed and unsigned half-word loads (`lh` and `lhu`), which load two bytes into the lower half and sign- or zero-extend the upper half of the word.

6.7.4 Floating-Point Instructions

The MIPS architecture defines an optional floating-point coprocessor, known as coprocessor 1. In early MIPS implementations, the floating-point coprocessor was a separate chip that users could purchase if they needed fast floating-point math. In most recent MIPS implementations, the floating-point coprocessor is built in alongside the main processor.

MIPS defines 32 32-bit floating-point registers, `$f0-$f31`. These are separate from the ordinary registers used so far. MIPS supports both single- and double-precision IEEE floating point arithmetic. Double-precision (64-bit) numbers are stored in pairs of 32-bit registers, so only the 16 even-numbered registers (`$f0, $f2, $f4, . . . , $f30`) are used to specify double-precision operations. By convention, certain registers are reserved for certain purposes, as given in Table 6.8.

Floating-point instructions all have an `opcode` of 17 (10001_2). They require both a `funct` field and a `cop` (coprocessor) field to indicate the type of instruction. Hence, MIPS defines the *F-type* instruction format for floating-point instructions, shown in Figure 6.35. Floating-point instructions come in both single- and double-precision flavors. `cop` = 16 (10000_2) for single-precision instructions or 17 (10001_2) for double-precision instructions. Like R-type instructions, F-type instructions have two source operands, `fs` and `ft`, and one destination, `fd`.

Instruction precision is indicated by `.s` and `.d` in the mnemonic. Floating-point arithmetic instructions include addition (`add.s, add.d`), subtraction (`sub.sz, sub.d`), multiplication (`mul.s, mul.d`), and division (`div.s, div.d`) as well as negation (`neg.s, neg.d`) and absolute value (`abs.s, abs.d`).

Table 6.8 MIPS floating-point register set

Name	Number	Use
`$fv0-$fv1`	0, 2	procedure return values
`$ft0-$ft3`	4, 6, 8, 10	temporary variables
`$fa0-$fa1`	12, 14	procedure arguments
`$ft4-$ft5`	16, 18	temporary variables
`$fs0-$fs5`	20, 22, 24, 26, 28, 30	saved variables

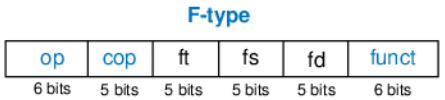


Figure 6.35 F-type machine instruction format

Floating-point branches have two parts. First, a compare instruction is used to set or clear the *floating-point condition flag* (`fpcond`). Then, a conditional branch checks the value of the flag. The compare instructions include equality (`c.seq.s`/`c.seq.d`), less than (`c.lt.s`/`c.lt.d`), and less than or equal to (`c.le.s`/`c.le.d`). The conditional branch instructions are `bc1f` and `bc1t` that branch if `fpcond` is FALSE or TRUE, respectively. Inequality, greater than or equal to, and greater than comparisons are performed with `seq`, `lt`, and `le`, followed by `bc1f`.

Floating-point registers are loaded and stored from memory using `lwc1` and `swc1`. These instructions move 32 bits, so two are necessary to handle a double-precision number.

6.8 REAL-WORLD PERSPECTIVE: IA-32 ARCHITECTURE*

Almost all personal computers today use IA-32 architecture microprocessors. IA-32 is a 32-bit architecture originally developed by Intel. AMD also sells IA-32 compatible microprocessors.

The IA-32 architecture has a long and convoluted history dating back to 1978, when Intel announced the 16-bit 8086 microprocessor. IBM selected the 8086 and its cousin, the 8088, for IBM's first personal computers. In 1985, Intel introduced the 32-bit 80386 microprocessor, which was backward compatible with the 8086, so it could run software developed for earlier PCs. Processor architectures compatible with the 80386 are called IA-32 or x86 processors. The Pentium, Core, and Athlon processors are well known IA-32 processors. Section 7.9 describes the evolution of IA-32 microprocessors in more detail.

Various groups at Intel and AMD over many years have shoehorned more instructions and capabilities into the antiquated architecture. The result is far less elegant than MIPS. As Patterson and Hennessy explain, "this checkered ancestry has led to an architecture that is difficult to explain and impossible to love." However, software compatibility is far more important than technical elegance, so IA-32 has been the *de facto* PC standard for more than two decades. More than 100 million IA-32 processors are sold every year. This huge market justifies more than $5 billion of research and development annually to continue improving the processors.

IA-32 is an example of a *Complex Instruction Set Computer* (*CISC*) architecture. In contrast to RISC architectures such as MIPS, each CISC instruction can do more work. Programs for CISC architectures usually require fewer instructions. The instruction encodings were selected to be more compact, so as to save memory, when RAM was far more expensive than it is today; instructions are of variable length and are often less than 32 bits. The trade-off is that complicated instructions are more difficult to decode and tend to execute more slowly.

Table 6.9 Major differences between MIPS and IA-32

Feature	MIPS	IA-32
# of registers	32 general purpose	8, some restrictions on purpose
# of operands	3 (2 source, 1 destination)	2 (1 source, 1 source/destination)
operand location	registers or immediates	registers, immediates, or memory
operand size	32 bits	8, 16, or 32 bits
condition codes	no	yes
instruction types	simple	simple and complicated
instruction encoding	fixed, 4 bytes	variable, 1–15 bytes

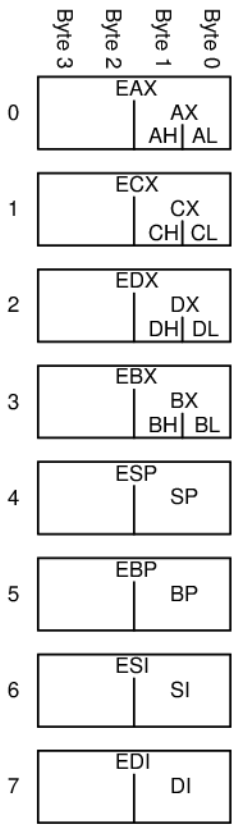


Figure 6.36 IA-32 registers

This section introduces the IA-32 architecture. The goal is not to make you into an IA-32 assembly language programmer, but rather to illustrate some of the similarities and differences between IA-32 and MIPS. We think it is interesting to see how IA-32 works. However, none of the material in this section is needed to understand the rest of the book. Major differences between IA-32 and MIPS are summarized in Table 6.9.

6.8.1 IA-32 Registers

The 8086 microprocessor provided eight 16-bit registers. It could separately access the upper and lower eight bits of some of these registers. When the 32-bit 80386 was introduced, the registers were extended to 32 bits. These registers are called `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI`, and `EDI`. For backward compatibility, the bottom 16 bits and some of the bottom 8-bit portions are also usable, as shown in Figure 6.36.

The eight registers are almost, but not quite, general purpose. Certain instructions cannot use certain registers. Other instructions always put their results in certain registers. Like `$sp` in MIPS, `ESP` is normally reserved for the stack pointer.

The IA-32 program counter is called `EIP` (the *extended instruction pointer*). Like the MIPS `PC`, it advances from one instruction to the next or can be changed with branch, jump, and subroutine call instructions.

6.8.2 IA-32 Operands

MIPS instructions always act on registers or immediates. Explicit load and store instructions are needed to move data between memory and the registers. In contrast, IA-32 instructions may operate on registers, immediates, or memory. This partially compensates for the small set of registers.

MIPS instructions generally specify three operands: two sources and one destination. IA-32 instructions specify only two operands. The first is a source. The second is both a source and the destination. Hence, IA-32 instructions always overwrite one of their sources with the result. Table 6.10 lists the combinations of operand locations in IA-32. All of the combinations are possible except memory to memory.

Like MIPS, IA-32 has a 32-bit memory space that is byte-addressable. However, IA-32 also supports a much wider variety of memory *addressing modes*. The memory location is specified with any combination of a *base register*, *displacement*, and a *scaled index register*. Table 6.11 illustrates these combinations. The displacement can be an 8-, 16-, or 32-bit value. The scale multiplying the index register can be 1, 2, 4, or 8. The base + displacement mode is equivalent to the MIPS base addressing mode for loads and stores. The scaled index provides an easy way to access arrays or structures of 2-, 4-, or 8-byte elements without having to issue a sequence of instructions to generate the address.

Table 6.10 Operand locations

Source/ Destination	Source	Example	Meaning
register	register	`add EAX, EBX`	`EAX <- EAX + EBX`
register	immediate	`add EAX, 42`	`EAX <- EAX + 42`
register	memory	`add EAX, [20]`	`EAX <- EAX + Mem[20]`
memory	register	`add [20], EAX`	`Mem[20] <- Mem[20] + EAX`
memory	immediate	`add [20], 42`	`Mem[20] <- Mem[20] + 42`

Table 6.11 Memory addressing modes

Example	Meaning	Comment
`add EAX, [20]`	`EAX <- EAX + Mem[20]`	displacement
`add EAX, [ESP]`	`EAX <- EAX + Mem[ESP]`	base addressing
`add EAX, [EDX+40]`	`EAX <- EAX + Mem[EDX+40]`	base + displacement
`add EAX, [60+EDI*4]`	`EAX <- EAX + Mem[60+EDI*4]`	displacement + scaled index
`add EAX, [EDX+80+EDI*2]`	`EAX <- EAX + Mem[EDX+80+EDI*2]`	base + displacement + scaled index

Table 6.12 Instructions acting on 8-, 16-, or 32-bit data

Example	Meaning	Data Size
`add AH, BL`	`AH <- AH + BL`	8-bit
`add AX, -1`	`AX <- AX + 0xFFFF`	16-bit
`add EAX, EDX`	`EAX <- EAX + EDX`	32-bit

While MIPS always acts on 32-bit words, IA-32 instructions can operate on 8-, 16-, or 32-bit data. Table 6.12 illustrates these variations.

6.8.3 Status Flags

IA-32, like many CISC architectures, uses *status flags* (also called *condition codes*) to make decisions about branches and to keep track of carries and arithmetic overflow. IA-32 uses a 32-bit register, called `EFLAGS`, that stores the status flags. Some of the bits of the `EFLAGS` register are given in Table 6.13. Other bits are used by the operating system.

The architectural state of an IA-32 processor includes `EFLAGS` as well as the eight registers and the `EIP`.

6.8.4 IA-32 Instructions

IA-32 has a larger set of instructions than MIPS. Table 6.14 describes some of the general purpose instructions. IA-32 also has instructions for floating-point arithmetic and for arithmetic on multiple short data elements packed into a longer word. `D` indicates the destination (a register or memory location), and `S` indicates the source (a register, memory location, or immediate).

Note that some instructions always act on specific registers. For example, 32 × 32-bit multiplication always takes one of the sources from `EAX` and always puts the 64-bit result in `EDX` and `EAX`. `LOOP` always

Table 6.13 Selected EFLAGS

Name	Meaning
`CF` (Carry Flag)	Carry out generated by last arithmetic operation. Indicates overflow in unsigned arithmetic. Also used for propagating the carry between words in multiple-precision arithmetic.
`ZF` (Zero Flag)	Result of last operation was zero.
`SF` (Sign Flag)	Result of last operation was negative (msb = 1).
`OF` (Overflow Flag)	Overflow of two's complement arithmetic.

Table 6.14 Selected IA-32 instructions

Instruction	Meaning	Function
ADD/SUB	add/subtract	D = D + S / D = D − S
ADDC	add with carry	D = D + S + CF
INC/DEC	increment/decrement	D = D + 1 / D = D − 1
CMP	compare	Set flags based on D − S
NEG	negate	D = −D
AND/OR/XOR	logical AND/OR/XOR	D = D op S
NOT	logical NOT	D = $\overline{D}$
IMUL/MUL	signed/unsigned multiply	EDX:EAX = EAX × D
IDIV/DIV	signed/unsigned divide	EDX:EAX/D EAX = Quotient; EDX = Remainder
SAR/SHR	arithmetic/logical shift right	D = D >>> S / D = D >> S
SAL/SHL	left shift	D = D << S
ROR/ROL	rotate right/left	Rotate D by S
RCR/RCL	rotate right/left with carry	Rotate CF and D by S
BT	bit test	CF = D[S] (the S*th* bit of D)
BTR/BTS	bit test and reset/set	CF = D[S]; D[S] = 0 / 1
TEST	set flags based on masked bits	Set flags based on D AND S
MOV	move	D = S
PUSH	push onto stack	ESP = ESP − 4; Mem[ESP] = S
POP	pop off stack	D = MEM[ESP]; ESP = ESP + 4
CLC, STC	clear/set carry flag	CF = 0 / 1
JMP	unconditional jump	relative jump: EIP = EIP + S absolute jump: EIP = S
Jcc	conditional jump	if (flag) EIP = EIP + S
LOOP	loop	ECX = ECX − 1 if ECX ≠ 0 EIP = EIP + imm
CALL	procedure call	ESP = ESP − 4; MEM[ESP] = EIP; EIP = S
RET	procedure return	EIP = MEM[ESP]; ESP = ESP + 4

Table 6.15 Selected branch conditions

Instruction	Meaning	Function After `cmp d, s`
JZ/JE	jump if ZF = 1	jump if D = S
JNZ/JNE	jump if ZF = 0	jump if D ≠ S
JGE	jump if SF = OF	jump if D ≥ S
JG	jump if SF = OF and ZF = 0	jump if D > S
JLE	jump if SF ≠ OF or ZF = 1	jump if D ≤ S
JL	jump if SF ≠ OF	jump if D < S
JC/JB	jump if CF = 1	
JNC	jump if CF = 0	
JO	jump if OF = 1	
JNO	jump if OF = 0	
JS	jump if SF = 1	
JNS	jump if SF = 0	

stores the loop counter in ECX. PUSH, POP, CALL, and RET use the stack pointer, ESP.

Conditional jumps check the flags and branch if the appropriate condition is met. They come in many flavors. For example, JZ jumps if the zero flag (ZF) is 1. JNZ jumps if the zero flag is 0. The jumps usually follow an instruction, such as the compare instruction (CMP), that sets the flags. Table 6.15 lists some of the conditional jumps and how they depend on the flags set by a prior compare operation.

6.8.5 IA-32 Instruction Encoding

The IA-32 instruction encodings are truly messy, a legacy of decades of piecemeal changes. Unlike MIPS, whose instructions are uniformly 32 bits, IA-32 instructions vary from 1 to 15 bytes, as shown in Figure 6.37.[2] The opcode may be 1, 2, or 3 bytes. It is followed by four optional fields: ModR/M, SIB, Displacement, and Immediate. ModR/M specifies an addressing mode. SIB specifies the scale, index, and base

[2] It is possible to construct 17-byte instructions if all the optional fields are used. However, IA-32 places a 15-byte limit on the length of legal instructions.

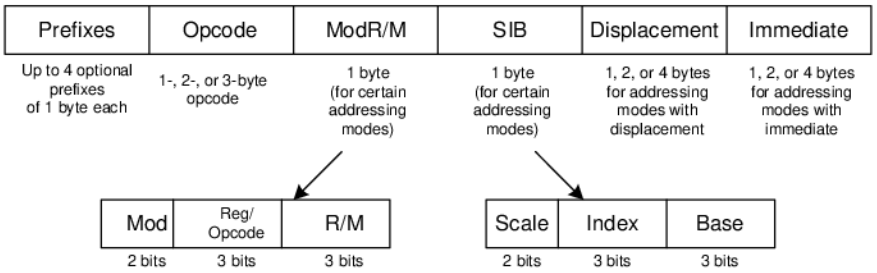


Figure 6.37 IA-32 instruction encodings

registers in certain addressing modes. `Displacement` indicates a 1-, 2-, or 4-byte displacement in certain addressing modes. And `Immediate` is a 1-, 2-, or 4-byte constant for instructions using an immediate as the source operand. Moreover, an instruction can be preceded by up to four optional byte-long prefixes that modify its behavior.

The `ModR/M` byte uses the 2-bit `Mod` and 3-bit `R/M` field to specify the addressing mode for one of the operands. The operand can come from one of the eight registers, or from one of 24 memory addressing modes. Due to artifacts in the encodings, the `ESP` and `EBP` registers are not available for use as the base or index register in certain addressing modes. The `Reg` field specifies the register used as the other operand. For certain instructions that do not require a second operand, the `Reg` field is used to specify three more bits of the `opcode`.

In addressing modes using a scaled index register, the `SIB` byte specifies the index register and the scale (1, 2, 4, or 8). If both a base and index are used, the `SIB` byte also specifies the base register.

MIPS fully specifies the instruction in the `opcode` and `funct` fields of the instruction. IA-32 uses a variable number of bits to specify different instructions. It uses fewer bits to specify more common instructions, decreasing the average length of the instructions. Some instructions even have multiple opcodes. For example, `add AL, imm8` performs an 8-bit add of an immediate to `AL`. It is represented with the 1-byte `opcode`, 0x04, followed by a 1-byte immediate. The `A` register (`AL`, `AX`, or `EAX`) is called the *accumulator*. On the other hand, `add D, imm8` performs an 8-bit add of an immediate to an arbitrary destination, `D` (memory or a register). It is represented with the 1-byte `opcode`, 0x80, followed by one or more bytes specifying `D`, followed by a 1-byte immediate. Many instructions have shortened encodings when the destination is the accumulator.

In the original 8086, the `opcode` specified whether the instruction acted on 8- or 16-bit operands. When the 80386 introduced 32-bit operands, no new opcodes were available to specify the 32-bit form.

Instead, the same opcode was used for both 16- and 32-bit forms. An additional bit in the *code segment descriptor* used by the OS specifies which form the processor should choose. The bit is set to 0 for backward compatibility with 8086 programs, defaulting the opcode to 16-bit operands. It is set to 1 for programs to default to 32-bit operands. Moreover, the programmer can specify prefixes to change the form for a particular instruction. If the prefix 0x66 appears before the opcode, the alternative size operand is used (16 bits in 32-bit mode, or 32 bits in 16-bit mode).

6.8.6 Other IA-32 Peculiarities

The 80286 introduced *segmentation* to divide memory into segments of up to 64 KB in length. When the OS enables segmentation, addresses are computed relative to the beginning of the segment. The processor checks for addresses that go beyond the end of the segment and indicates an error, thus preventing programs from accessing memory outside their own segment. Segmentation proved to be a hassle for programmers and is not used in modern versions of the Windows operating system.

IA-32 contains string instructions that act on entire strings of bytes or words. The operations include moving, comparing, or scanning for a specific value. In modern processors, these instructions are usually slower than performing the equivalent operation with a series of simpler instructions, so they are best avoided.

As mentioned earlier, the 0x66 prefix is used to choose between 16- and 32-bit operand sizes. Other prefixes include ones used to lock the bus (to control access to shared variables in a multiprocessor system), to predict whether a branch will be taken or not, and to repeat the instruction during a string move.

Intel and Hewlett-Packard jointly developed a new 64-bit architecture called IA-64 in the mid 1990's. It was designed from a clean slate, bypassing the convoluted history of IA-32, taking advantage of 20 years of new research in computer architecture, and providing a 64-bit address space. However, IA-64 has yet to become a market success. Most computers needing the large address space now use the 64-bit extensions of IA-32.

The bane of any architecture is to run out of memory capacity. With 32-bit addresses, IA-32 can access 4 GB of memory. This was far more than the largest computers had in 1985, but by the early 2000s it had become limiting. In 2003, AMD extended the address space and register sizes to 64 bits, calling the enhanced architecture AMD64. AMD64 has a compatibility mode that allows it to run 32-bit programs unmodified while the OS takes advantage of the bigger address space. In 2004, Intel gave in and adopted the 64-bit extensions, renaming them Extended Memory 64 Technology (EM64T). With 64-bit addresses, computers can access 16 exabytes (16 billion GB) of memory.

For those curious about more details of the IA-32 architecture, the IA-32 Intel Architecture Software Developer's Manual, is freely available on Intel's Web site.

6.8.7 The Big Picture

This section has given a taste of some of the differences between the MIPS RISC architecture and the IA-32 CISC architecture. IA-32 tends to have shorter programs, because a complex instruction is equivalent to a series of simple MIPS instructions and because the instructions are encoded to minimize memory use. However, the IA-32 architecture is a hodgepodge of features accumulated over the years, some of which are no longer useful but must be kept for compatibility with old programs. It has too few registers, and the instructions are difficult to decode. Merely explaining the instruction set is difficult. Despite all these failings, IA-32 is firmly entrenched as the dominant computer architecture for PCs, because the value of software compatibility is so great and because the huge market justifies the effort required to build fast IA-32 microprocessors.

6.9 SUMMARY

To command a computer, you must speak its language. A computer architecture defines how to command a processor. Many different computer architectures are in widespread commercial use today, but once you understand one, learning others is much easier. The key questions to ask when approaching a new architecture are

- What is the data word length?
- What are the registers?
- How is memory organized?
- What are the instructions?

MIPS is a 32-bit architecture because it operates on 32-bit data. The MIPS architecture has 32 general-purpose registers. In principle, almost any register can be used for any purpose. However, by convention, certain registers are reserved for certain purposes, for ease of programming and so that procedures written by different programmers can communicate easily. For example, register 0, `$0`, always holds the constant 0, `$ra` holds the return address after a `jal` instruction, and `$a0-$a3` and `$v0 - $v1` hold the arguments and return value of a procedure. MIPS has a byte-addressable memory system with 32-bit addresses. The memory map was described in Section 6.6.1. Instructions are 32 bits long and must be word aligned. This chapter discussed the most commonly used MIPS instructions.

The power of defining a computer architecture is that a program written for any given architecture can run on many different implementations of that architecture. For example, programs written for the Intel

Pentium processor in 1993 will generally still run (and run much faster) on the Intel Core 2 Duo or AMD Athlon processors in 2006.

In the first part of this book, we learned about the circuit and logic levels of abstraction. In this chapter, we jumped up to the architecture level. In the next chapter, we study microarchitecture, the arrangement of digital building blocks that implement a processor architecture. Microarchitecture is the link between hardware and software engineering. And, we believe it is one of the most exciting topics in all of engineering: you will learn to build your own microprocessor!

Exercises

Exercise 6.1 Give three examples from the MIPS architecture of each of the architecture design principles: (1) simplicity favors regularity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises. Explain how each of your examples exhibits the design principle.

Exercise 6.2 The MIPS architecture has a register set that consists of 32 32-bit registers. Is it possible to design a computer architecture without a register set? If so, briefly describe the architecture, including the instruction set. What are advantages and disadvantages of this architecture over the MIPS architecture?

Exercise 6.3 Consider memory storage of a 32-bit word stored at memory word 42 in a byte addressable memory.

(a) What is the byte address of memory word 42?

(b) What are the byte addresses that memory word 42 spans?

(c) Draw the number 0xFF223344 stored at word 42 in both big-endian and little-endian machines. Your drawing should be similar to Figure 6.4. Clearly label the byte address corresponding to each data byte value.

Exercise 6.4 Explain how the following program can be used to determine whether a computer is big-endian or little-endian:

```
li $t0, 0xABCD9876
sw $t0, 100($0)
lb $s5, 101($0)
```

Exercise 6.5 Write the following strings using ASCII encoding. Write your final answers in hexadecimal.

(a) SOS

(b) Cool!

(c) (your own name)

Exercise 6.6 Show how the strings in Exercise 6.5 are stored in a byte-addressable memory on (a) a big-endian machine and (b) a little-endian machine starting at memory address 0x1000100C. Use a memory diagram similar to Figure 6.4. Clearly indicate the memory address of each byte on each machine.

Exercise 6.7 Convert the following MIPS assembly code into machine language. Write the instructions in hexadecimal.

```
add  $t0, $s0, $s1
lw   $t0, 0x20($t7)
addi $s0, $0, -10
```

Exercise 6.8 Repeat Exercise 6.7 for the following MIPS assembly code:

```
addi $s0, $0, 73
sw   $t1, -7($t2)
sub  $t1, $s7, $s2
```

Exercise 6.9 Consider I-type instructions.

(a) Which instructions from Exercise 6.8 are I-type instructions?

(b) Sign-extend the 16-bit immediate of each instruction from part (a) so that it becomes a 32-bit number.

Exercise 6.10 Convert the following program from machine language into MIPS assembly language. The numbers on the left are the instruction address in memory, and the numbers on the right give the instruction at that address. Then reverse engineer a high-level program that would compile into this assembly language routine and write it. Explain in words what the program does. `$a0` is the input, and it initially contains a positive number, `n`. `$v0` is the output.

```
0x00400000   0x20080000
0x00400004   0x20090001
0x00400008   0x0089502a
0x0040000c   0x15400003
0x00400010   0x01094020
0x00400014   0x21290002
0x00400018   0x08100002
0x0040001c   0x01001020
```

Exercise 6.11 The `nori` instruction is not part of the MIPS instruction set, because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: `$t0 = $t1 NOR 0xF234`. Use as few instructions as possible.

Exercise 6.12 Implement the following high-level code segments using the `slt` instruction. Assume the integer variables `g` and `h` are in registers `$s0` and `$s1`, respectively.

(a)
```
if (g > h)
    g = g + h;
else
    g = g - h;
```

(b)
```
if (g >= h)
    g = g + 1;
else
    h = h - 1;
```

(c)
```
if (g <= h)
    g = 0;
else
    h = 0;
```

Exercise 6.13 Write a procedure in a high-level language for `int find42(int array[], int size)`. `size` specifies the number of elements in the array. `array` specifies the base address of the array. The procedure should return the index number of the first array entry that holds the value 42. If no array entry is 42, it should return the value −1.

Exercise 6.14 The high-level procedure `strcpy` copies the character string `x` to the character string `y`.

```
// high-level code
void strcpy(char x[], char y[]) {
  int i = 0;

  while (x[i] != 0) {
    y[i] = x[i];
    i = i + 1;
  }
}
```

(a) Implement the `strcpy` procedure in MIPS assembly code. Use `$s0` for `i`.

(b) Draw a picture of the stack before, during, and after the `strcpy` procedure call. Assume `$sp` = 0x7FFFFF00 just before `strcpy` is called.

Exercise 6.15 Convert the high-level procedure from Exercise 6.13 into MIPS assembly code.

This simple string copy program has a serious flaw: it has no way of knowing that `y` has enough space to receive `x`. If a malicious programmer were able to execute `strcpy` with a long string `x`, the programmer might be able to write bytes all over memory, possibly even modifying code stored in subsequent memory locations. With some cleverness, the modified code might take over the machine. This is called a buffer overflow attack; it is employed by several nasty programs, including the infamous Blaster worm, which caused an estimated $525 million in damages in 2003.

Exercise 6.16 Each number in the *Fibonacci series* is the sum of the previous two numbers. Table 6.16 lists the first few numbers in the series, *fib(n)*.

Table 6.16 Fibonacci series

n	1	2	3	4	5	6	7	8	9	10	11	. . .
fib(n)	1	1	2	3	5	8	13	21	34	55	89	. . .

(a) What is *fib(n)* for $n = 0$ and $n = -1$?

(b) Write a procedure called `fib` in a high-level language that returns the Fibonacci number for any nonnegative value of n. Hint: You probably will want to use a loop. Clearly comment your code.

(c) Convert the high-level procedure of part (b) into MIPS assembly code. Add comments after every line of code that explain clearly what it does. Use the SPIM simulator to test your code on fib(9).

Exercise 6.17 Consider the MIPS assembly code below. `proc1`, `proc2`, and `proc3` are non-leaf procedures. `proc4` is a leaf procedure. The code is not shown for each procedure, but the comments indicate which registers are used within each procedure.

```
0x00401000    proc1:   . . .          # proc1 uses $s0 and $s1
0x00401020             jal proc2
 . . .
0x00401100    proc2:   . . .          # proc2 uses $s2 - $s7
0x0040117C             jal proc3
 . . .
0x00401400    proc3:   . . .          # proc3 uses $s1 - $s3
0x00401704             jal proc4
 . . .
0x00403008    proc4:   . . .          # proc4 uses no preserved
                                      # registers
0x00403118             jr $ra
```

(a) How many words are the stack frames of each procedure?

(b) Sketch the stack after `proc4` is called. Clearly indicate which registers are stored where on the stack. Give values where possible.

Exercise 6.18 Ben Bitdiddle is trying to compute the function $f(a, b) = 2a + 3b$ for nonnegative b. He goes overboard in the use of procedure calls and recursion and produces the following high-level code for procedures `f` and `f2`.

```
// high-level code for procedures f and f2
int f(int a, int b) {
  int j;
  j = a;
  return j + a + f2(b);
}

int f2(int x)
{
  int k;
  k = 3;
  if (x == 0) return 0;
  else return k + f2(x-1);
}
```

Ben then translates the two procedures into assembly language as follows. He also writes a procedure, `test`, that calls the procedure `f(5, 3)`.

```
# MIPS assembly code
# f: $a0 = a, $a1 = b, $s0 = j f2: $a0 = x, $s0 = k

0x00400000  test:addi  $a0, $0, 5     # $a0 = 5 (a = 5)
0x00400004       addi  $a1, $0, 3     # $a1 = 3 (b = 3)
0x00400008       jal   f              # call f(5,3)
0x0040000c  loop:j     loop           # and loop forever

0x00400010  f:   addi  $sp, $sp, -16  # make room on the stack
                                      # for $s0, $a0, $a1, and $ra
0x00400014       sw    $a1, 12($sp)   # save $a1 (b)
0x00400018       sw    $a0, 8($sp)    # save $a0 (a)
0x0040001c       sw    $ra, 4($sp)    # save $ra
0x00400020       sw    $s0, 0($sp)    # save $s0
0x00400024       add   $s0, $a0, $0   # $s0 = $a0 (j = a)
0x00400028       add   $a0, $a1, $0   # place b as argument for f2
0x0040002c       jal   f2             # call f2(b)
0x00400030       lw    $a0, 8($sp)    # restore $a0 (a) after call
0x00400034       lw    $a1, 12($sp)   # restore $a1 (b) after call
0x00400038       add   $v0, $v0, $s0  # $v0 = f2(b) + j
0x0040003c       add   $v0, $v0, $a0  # $v0 = (f2(b) + j) + a
0x00400040       lw    $s0, 0($sp)    # restore $s0
0x00400044       lw    $ra, 4($sp)    # restore $ra
0x00400048       addi  $sp, $sp, 16   # restore $sp (stack pointer)
0x0040004c       jr    $ra            # return to point of call

0x00400050  f2:  addi  $sp, $sp, -12  # make room on the stack for
                                      # $s0, $a0, and $ra
0x00400054       sw    $a0, 8($sp)    # save $a0 (x)
0x00400058       sw    $ra, 4($sp)    # save return address
0x0040005c       sw    $s0, 0($sp)    # save $s0
0x00400060       addi  $s0, $0, 3     # k = 3
```

```
0x00400064        bne   $a0, $0, else  # x = 0?
0x00400068        addi  $v0, $0, 0     # yes: return value should be 0
0x0040006c        j     done           # and clean up
0x00400070  else: addi  $a0, $a0, -1   # no: $a0 = $a0 - 1 (x = x - 1)
0x00400074        jal   f2             # call f2(x - 1)
0x00400078        lw    $a0, 8($sp)    # restore $a0 (x)
0x0040007c        add   $v0, $v0, $s0  # $v0 = f2(x - 1) + k
0x00400080  done: lw    $s0, 0($sp)    # restore $s0
0x00400084        lw    $ra, 4($sp)    # restore $ra
0x00400088        addi  $sp, $sp, 12   # restore $sp
0x0040008c        jr    $ra            # return to point of call
```

You will probably find it useful to make drawings of the stack similar to the one in Figure 6.26 to help you answer the following questions.

(a) If the code runs starting at `test`, what value is in `$v0` when the program gets to `loop`? Does his program correctly compute $2a + 3b$?

(b) Suppose Ben deletes the instructions at addresses 0x0040001C and 0x00400040 that save and restore `$ra`. Will the program (1) enter an infinite loop but not crash; (2) crash (cause the stack to grow beyond the dynamic data segment or the PC to jump to a location outside the program); (3) produce an incorrect value in `$v0` when the program returns to loop (if so, what value?), or (4) run correctly despite the deleted lines?

(c) Repeat part (b) when the instructions at the following instruction addresses are deleted:

(i) 0x00400018 and 0x00400030 (instructions that save and restore `$a0`)

(ii) 0x00400014 and 0x00400034 (instructions that save and restore `$a1`)

(iii) 0x00400020 and 0x00400040 (instructions that save and restore `$s0`)

(iv) 0x00400050 and 0x00400088 (instructions that save and restore `$sp`)

(v) 0x0040005C and 0x00400080 (instructions that save and restore `$s0`)

(vi) 0x00400058 and 0x00400084 (instructions that save and restore `$ra`)

(vii) 0x00400054 and 0x00400078 (instructions that save and restore `$a0`)

Exercise 6.19 Convert the following `beq`, `j`, and `jal` assembly instructions into machine code. Instruction addresses are given to the left of each instruction.

(a)

```
0x00401000            beq $t0, $s1, Loop
0x00401004             . . .
0x00401008             . . .
0x0040100C     Loop:   . . .
```

(b)

```
0x00401000            beq $t7, $s4, done
 . . .                . . .
0x00402040     done:  . . .
```

(c)

```
0x0040310C     back:  . . .
 . . .                . . .
0x00405000            beq $t9, $s7, back
```

(d)

```
0x00403000            jal proc
 . . .                . . .
0x0041147C     proc:  . . .
```

(e)

```
0x00403004     back:  . . .
 . . .                . . .
0x0040400C     j      back
```

Exercise 6.20 Consider the following MIPS assembly language snippet. The numbers to the left of each instruction indicate the instruction address.

```
0x00400028        add  $a0, $a1, $0
0x0040002c        jal  f2
0x00400030 f1:    jr   $ra
0x00400034 f2:    sw   $s0, 0($s2)
0x00400038        bne  $a0, $0, else
0x0040003c        j    f1
0x00400040 else:  addi $a0, $a0, -1
0x00400044        j    f2
```

(a) Translate the instruction sequence into machine code. Write the machine code instructions in hexadecimal.

(b) List the addressing mode used at each line of code.

Exercise 6.21 Consider the following C code snippet.

```
// C code
   void set_array(int num) {
   int i;
   int array[10];

   for (i = 0; i < 10; i = i + 1) {
     array[i] = compare(num, i);
   }
  }
  int compare(int a, int b) {
   if (sub(a, b) >= 0)
     return 1;
   else
```

```
  return 0;
}
int sub (int a, int b) {
  return a - b;
}
```

(a) Implement the C code snippet in MIPS assembly language. Use `$s0` to hold the variable `i`. Be sure to handle the stack pointer appropriately. The array is stored on the stack of the `set_array` procedure (see Section 6.4.6).

(b) Assume `set_array` is the first procedure called. Draw the status of the stack before calling `set_array` and during each procedure call. Indicate the names of registers and variables stored on the stack and mark the location of `$sp`.

(c) How would your code function if you failed to store `$ra` on the stack?

Exercise 6.22 Consider the following high-level procedure.

```
// high-level code
int f(int n, int k) {
  int b;

  b = k + 2;
  if (n == 0) b = 10;
  else b = b + (n * n) + f(n - 1, k + 1);
  return b * k;
}
```

(a) Translate the high-level procedure `f` into MIPS assembly language. Pay particular attention to properly saving and restoring registers across procedure calls and using the MIPS preserved register conventions. Clearly comment your code. You can use the MIPS `mult`, `mfhi`, and `mflo` instructions. The procedure starts at instruction address 0x00400100. Keep local variable `b` in `$s0`.

(b) Step through your program from part (a) by hand for the case of `f(2, 4)`. Draw a picture of the stack similar to the one in Figure 6.26(c). Write the register name and data value stored at each location in the stack and keep track of the stack pointer value (`$sp`). You might also find it useful to keep track of the values in `$a0`, `$a1`, `$v0`, and `$s0` throughout execution. Assume that when `f` is called, `$s0` = 0xABCD and `$ra` = 0x400004. What is the final value of `$v0`?

Exercise 6.23 What is the range of instruction addresses to which conditional branches, such as `beq` and `bne`, can branch in MIPS? Give your answer in number of instructions relative to the conditional branch instruction.

Exercise 6.24 The following questions examine the limitations of the jump instruction, `j`. Give your answer in number of instructions relative to the jump instruction.

(a) In the worst case, how far can the jump instruction (`j`) jump forward (i.e., to higher addresses)? (The worst case is when the jump instruction cannot jump far.) Explain using words and examples, as needed.

(b) In the best case, how far can the jump instruction (`j`) jump forward? (The best case is when the jump instruction can jump the farthest.) Explain.

(c) In the worst case, how far can the jump instruction (`j`) jump backward (to lower addresses)? Explain.

(d) In the best case, how far can the jump instruction (`j`) jump backward? Explain.

Exercise 6.25 Explain why it is advantageous to have a large address field, `addr`, in the machine format for the jump instructions, `j` and `jal`.

Exercise 6.26 Write assembly code that jumps to the instruction 64 Minstructions from the first instruction. Recall that 1 Minstruction = 2^{20} instructions = 1,048,576 instructions. Assume that your code begins at address 0x00400000. Use a minimum number of instructions.

Exercise 6.27 Write a procedure in high-level code that takes a ten-entry array of 32-bit integers stored in little-endian format and converts it to big-endian format. After writing the high-level code, convert it to MIPS assembly code. Comment all your code and use a minimum number of instructions.

Exercise 6.28 Consider two strings: `string1` and `string2`.

(a) Write high-level code for a procedure called `concat` that concatenates (joins together) the two strings: `void concat(char[] string1, char[] string2, char[] stringconcat)`. The procedure does not return a value. It concatenates `string1` and `string2` and places the resulting string in `stringconcat`. You may assume that the character array `stringconcat` is large enough to accommodate the concatenated string.

(b) Convert the procedure from part (a) into MIPS assembly language.

Exercise 6.29 Write a MIPS assembly program that adds two positive single-precision floating point numbers held in `$s0` and `$s1`. Do not use any of the MIPS floating-point instructions. You need not worry about any of the encodings that are reserved for special purposes (e.g., 0, NANs, INF) or numbers that overflow or underflow. Use the SPIM simulator to test your code. You will need to manually set the values of `$s0` and `$s1` to test your code. Demonstrate that your code functions reliably.

Exercise 6.30 Show how the following MIPS program would be loaded into memory and executed.

```
# MIPS assembly code
main:
  lw $a0, x
  lw $a1, y
  jal diff
  jr $ra
diff:
  sub $v0, $a0, $a1
  jr $ra
```

(a) First show the instruction address next to each assembly instruction.

(b) Draw the symbol table showing the labels and their addresses.

(c) Convert all instructions into machine code.

(d) How big (how many bytes) are the data and text segments?

(e) Sketch a memory map showing where data and instructions are stored.

Exercise 6.31 Show the MIPS instructions that implement the following pseudoinstructions. You may use the assembler register, `$at`, but you may not corrupt (overwrite) any other registers.

(a) `beq $t1,` $\text{imm}_{31:0}$`, L`

(b) `ble $t3, $t5, L`

(c) `bgt $t3, $t5, L`

(d) `bge $t3, $t5, L`

(e) `addi $t0, $2,` $\text{imm}_{31:0}$

(f) `lw $t5,` $\text{imm}_{31:0}$`($s0)`

(g) `rol $t0, $t1, 5` (rotate `$t1` left by 5 and put the result in `$t0`)

(h) `ror $s4, $t6, 31` (rotate `$t6` right by 31 and put the result in `$s4`)

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs (but are usually open to any assembly language).

Question 6.1 Write MIPS assembly code for swapping the contents of two registers, `$t0` and `$t1`. You may not use any other registers.

Question 6.2 Suppose you are given an array of both positive and negative integers. Write MIPS assembly code that finds the subset of the array with the largest sum. Assume that the array's base address and the number of array elements are in `$a0` and `$a1`, respectively. Your code should place the resulting subset of the array starting at base address `$a2`. Write code that runs as fast as possible.

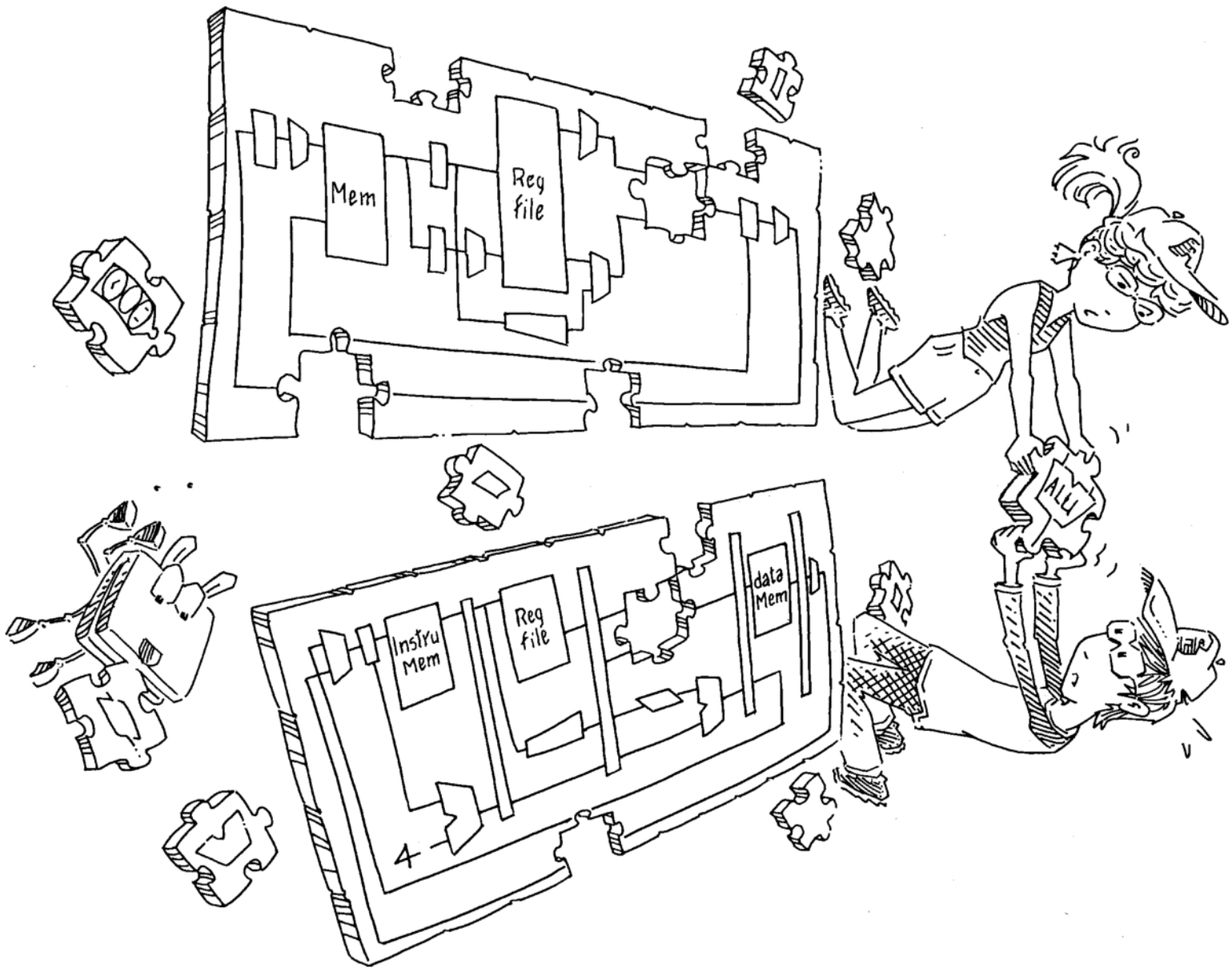
Question 6.3 You are given an array that holds a C string. The string forms a sentence. Design an algorithm for reversing the words in the sentence and storing the new sentence back in the array. Implement your algorithm using MIPS assembly code.

Question 6.4 Design an algorithm for counting the number of 1's in a 32-bit number. Implement your algorithm using MIPS assembly code.

Question 6.5 Write MIPS assembly code to reverse the bits in a register. Use as few instructions as possible. Assume the register of interest is `$t3`.

Question 6.6 Write MIPS assembly code to test whether overflow occurs when `$t2` and `$t3` are added. Use a minimum number of instructions.

Question 6.7 Design an algorithm for testing whether a given string is a palindrome. (Recall, that a palindrome is a word that is the same forward and backward. For example, the words "wow" and "racecar" are palindromes.) Implement your algorithm using MIPS assembly code.


Mem
Reg file
ALU
Instru Mem
Reg file
data Mem
4

Microarchitecture

7

7.1 INTRODUCTION

In this chapter, you will learn how to piece together a MIPS microprocessor. Indeed, you will puzzle out three different versions, each with different trade-offs between performance, cost, and complexity.

To the uninitiated, building a microprocessor may seem like black magic. But it is actually relatively straightforward, and by this point you have learned everything you need to know. Specifically, you have learned to design combinational and sequential logic given functional and timing specifications. You are familiar with circuits for arithmetic and memory. And you have learned about the MIPS architecture, which specifies the programmer's view of the MIPS processor in terms of registers, instructions, and memory.

This chapter covers *microarchitecture*, which is the connection between logic and architecture. Microarchitecture is the specific arrangement of registers, ALUs, finite state machines (FSMs), memories, and other logic building blocks needed to implement an architecture. A particular architecture, such as MIPS, may have many different microarchitectures, each with different trade-offs of performance, cost, and complexity. They all run the same programs, but their internal designs vary widely. We will design three different microarchitectures in this chapter to illustrate the trade-offs.

This chapter draws heavily on David Patterson and John Hennessy's classic MIPS designs in their text *Computer Organization and Design*. They have generously shared their elegant designs, which have the virtue of illustrating a real commercial architecture while being relatively simple and easy to understand.

7.1.1 Architectural State and Instruction Set

Recall that a computer architecture is defined by its instruction set and *architectural state*. The architectural state for the MIPS processor consists

David Patterson was the first in his family to graduate from college (UCLA, 1969). He has been a professor of computer science at UC Berkeley since 1977, where he coinvented RISC, the Reduced Instruction Set Computer. In 1984, he developed the SPARC architecture used by Sun Microsystems. He is also the father of *RAID* (*Redundant Array of Inexpensive Disks*) and *NOW* (*Network of Workstations*).

John Hennessy is president of Stanford University and has been a professor of electrical engineering and computer science there since 1977. He coinvented RISC. He developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems. As of 2004, more than 300 million MIPS microprocessors have been sold.

In their copious free time, these two modern paragons write textbooks for recreation and relaxation.

of the program counter and the 32 registers. Any MIPS microarchitecture must contain all of this state. Based on the current architectural state, the processor executes a particular instruction with a particular set of data to produce a new architectural state. Some microarchitectures contain additional *nonarchitectural state* to either simplify the logic or improve performance; we will point this out as it arises.

To keep the microarchitectures easy to understand, we consider only a subset of the MIPS instruction set. Specifically, we handle the following instructions:

- R-type arithmetic/logic instructions: `add`, `sub`, `and`, `or`, `slt`
- Memory instructions: `lw`, `sw`
- Branches: `beq`

After building the microarchitectures with these instructions, we extend them to handle `addi` and `j`. These particular instructions were chosen because they are sufficient to write many interesting programs. Once you understand how to implement these instructions, you can expand the hardware to handle others.

7.1.2 Design Process

We will divide our microarchitectures into two interacting parts: the *datapath* and the *control*. The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers. MIPS is a 32-bit architecture, so we will use a 32-bit datapath. The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

A good way to design a complex system is to start with hardware containing the state elements. These elements include the memories and the architectural state (the program counter and registers). Then, add blocks of combinational logic between the state elements to compute the new state based on the current state. The instruction is read from part of memory; load and store instructions then read or write data from another part of memory. Hence, it is often convenient to partition the overall memory into two smaller memories, one containing instructions and the other containing data. Figure 7.1 shows a block diagram with the four state elements: the program counter, register file, and instruction and data memories.

In Figure 7.1, heavy lines are used to indicate 32-bit data busses. Medium lines are used to indicate narrower busses, such as the 5-bit address busses on the register file. Narrow blue lines are used to indicate

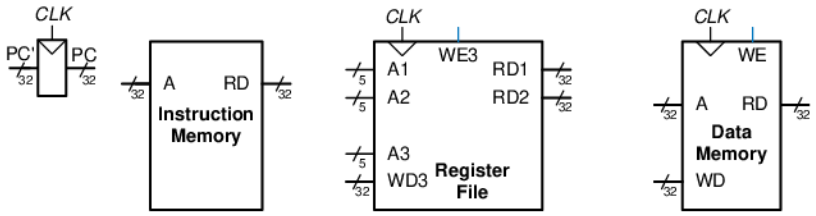


Figure 7.1 State elements of MIPS processor

control signals, such as the register file write enable. We will use this convention throughout the chapter to avoid cluttering diagrams with bus widths. Also, state elements usually have a reset input to put them into a known state at start-up. Again, to save clutter, this reset is not shown.

The *program counter* is an ordinary 32-bit register. Its output, *PC*, points to the current instruction. Its input, *PC'*, indicates the address of the next instruction.

The *instruction memory* has a single read port.[1] It takes a 32-bit instruction address input, *A*, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, *RD*.

The 32-element × 32-bit *register file* has two read ports and one write port. The read ports take 5-bit address inputs, *A1* and *A2*, each specifying one of $2^5 = 32$ registers as source operands. They read the 32-bit register values onto read data outputs *RD1* and *RD2*, respectively. The write port takes a 5-bit address input, *A3*; a 32-bit write data input, *WD*; a write enable input, *WE3*; and a clock. If the write enable is 1, the register file writes the data into the specified register on the rising edge of the clock.

The *data memory* has a single read/write port. If the write enable, *WE*, is 1, it writes data *WD* into address *A* on the rising edge of the clock. If the write enable is 0, it reads address *A* onto *RD*.

The instruction memory, register file, and data memory are all read *combinationally*. In other words, if the address changes, the new data appears at *RD* after some propagation delay; no clock is involved. They are written only on the rising edge of the clock. In this fashion, the state of the system is changed only at the clock edge. The address, data, and write enable must setup sometime before the clock edge and must remain stable until a hold time after the clock edge.

Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits. The microprocessor is

Resetting the PC

At the very least, the program counter must have a reset signal to initialize its value when the processor turns on. MIPS processors initialize the PC to `0xBFC00000` on reset and begin executing code to start up the operating system (OS). The OS then loads an application program at `0x00400000` and begins executing it. For simplicity in this chapter, we will reset the PC to `0x00000000` and place our programs there instead.

[1] This is an oversimplification used to treat the instruction memory as a ROM; in most real processors, the instruction memory must be writable so that the OS can load a new program into memory. The multicycle microarchitecture described in Section 7.4 is more realistic in that it uses a combined memory for instructions and data that can be both read and written.

built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit. Indeed, the processor can be viewed as a giant finite state machine, or as a collection of simpler interacting state machines.

7.1.3 MIPS Microarchitectures

In this chapter, we develop three microarchitectures for the MIPS processor architecture: single-cycle, multicycle, and pipelined. They differ in the way that the state elements are connected together and in the amount of nonarchitectural state.

The *single-cycle microarchitecture* executes an entire instruction in one cycle. It is easy to explain and has a simple control unit. Because it completes the operation in one cycle, it does not require any nonarchitectural state. However, the cycle time is limited by the slowest instruction.

The *multicycle microarchitecture* executes instructions in a series of shorter cycles. Simpler instructions execute in fewer cycles than complicated ones. Moreover, the multicycle microarchitecture reduces the hardware cost by reusing expensive hardware blocks such as adders and memories. For example, the adder may be used on several different cycles for several purposes while carrying out a single instruction. The multicycle microprocessor accomplishes this by adding several nonarchitectural registers to hold intermediate results. The multicycle processor executes only one instruction at a time, but each instruction takes multiple clock cycles.

The *pipelined microarchitecture* applies pipelining to the single-cycle microarchitecture. It therefore can execute several instructions simultaneously, improving the throughput significantly. Pipelining must add logic to handle dependencies between simultaneously executing instructions. It also requires nonarchitectural pipeline registers. The added logic and registers are worthwhile; all commercial high-performance processors use pipelining today.

We explore the details and trade-offs of these three microarchitectures in the subsequent sections. At the end of the chapter, we briefly mention additional techniques that are used to get even more speed in modern high-performance microprocessors.

7.2 PERFORMANCE ANALYSIS

As we mentioned, a particular processor architecture can have many microarchitectures with different cost and performance trade-offs. The cost depends on the amount of hardware required and the implementation technology. Each year, CMOS processes can pack more transistors on a chip for the same amount of money, and processors take advantage

of these additional transistors to deliver more performance. Precise cost calculations require detailed knowledge of the implementation technology, but in general, more gates and more memory mean more dollars. This section lays the foundation for analyzing performance.

There are many ways to measure the performance of a computer system, and marketing departments are infamous for choosing the method that makes their computer look fastest, regardless of whether the measurement has any correlation to real world performance. For example, Intel and Advanced Micro Devices (AMD) both sell compatible microprocessors conforming to the IA-32 architecture. Intel Pentium III and Pentium 4 microprocessors were largely advertised according to clock frequency in the late 1990s and early 2000s, because Intel offered higher clock frequencies than its competitors. However, Intel's main competitor, AMD, sold Athlon microprocessors that executed programs faster than Intel's chips at the same clock frequency. What is a consumer to do?

The only gimmick-free way to measure performance is by measuring the execution time of a program of interest to you. The computer that executes your program fastest has the highest performance. The next best choice is to measure the total execution time of a collection of programs that are similar to those you plan to run; this may be necessary if you haven't written your program yet or if somebody else who doesn't have your program is making the measurements. Such collections of programs are called *benchmarks*, and the execution times of these programs are commonly published to give some indication of how a processor performs.

The execution time of a program, measured in seconds, is given by Equation 7.1.

$$Execution\ Time = \left(\#\ instructions\right)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right) \quad (7.1)$$

The number of instructions in a program depends on the processor architecture. Some architectures have complicated instructions that do more work per instruction, thus reducing the number of instructions in a program. However, these complicated instructions are often slower to execute in hardware. The number of instructions also depends enormously on the cleverness of the programmer. For the purposes of this chapter, we will assume that we are executing known programs on a MIPS processor, so the number of instructions for each program is constant, independent of the microarchitecture.

The number of cycles per instruction, often called *CPI*, is the number of clock cycles required to execute an average instruction. It is the reciprocal of the throughput (instructions per cycle, or *IPC*). Different microarchitectures have different CPIs. In this chapter, we will assume

we have an ideal memory system that does not affect the CPI. In Chapter 8, we examine how the processor sometimes has to wait for the memory, which increases the CPI.

The number of seconds per cycle is the clock period, T_c. The clock period is determined by the critical path through the logic on the processor. Different microarchitectures have different clock periods. Logic and circuit designs also significantly affect the clock period. For example, a carry-lookahead adder is faster than a ripple-carry adder. Manufacturing advances have historically doubled transistor speeds every 4–6 years, so a microprocessor built today will be much faster than one from last decade, even if the microarchitecture and logic are unchanged.

The challenge of the microarchitect is to choose the design that minimizes the execution time while satisfying constraints on cost and/or power consumption. Because microarchitectural decisions affect both CPI and T_c and are influenced by logic and circuit designs, determining the best choice requires careful analysis.

There are many other factors that affect overall computer performance. For example, the hard disk, the memory, the graphics system, and the network connection may be limiting factors that make processor performance irrelevant. The fastest microprocessor in the world doesn't help surfing the Internet on a dial-up connection. But these other factors are beyond the scope of this book.

7.3 SINGLE-CYCLE PROCESSOR

We first design a MIPS microarchitecture that executes instructions in a single cycle. We begin constructing the datapath by connecting the state elements from Figure 7.1 with combinational logic that can execute the various instructions. Control signals determine which specific instruction is carried out by the datapath at any given time. The controller contains combinational logic that generates the appropriate control signals based on the current instruction. We conclude by analyzing the performance of the single-cycle processor.

7.3.1 Single-Cycle Datapath

This section gradually develops the single-cycle datapath, adding one piece at a time to the state elements from Figure 7.1. The new connections are emphasized in black (or blue, for new control signals), while the hardware that has already been studied is shown in gray.

The program counter (PC) register contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. Figure 7.2 shows that the PC is simply connected to the address input of the instruction memory. The instruction memory reads out, or *fetches*, the 32-bit instruction, labeled *Instr*.

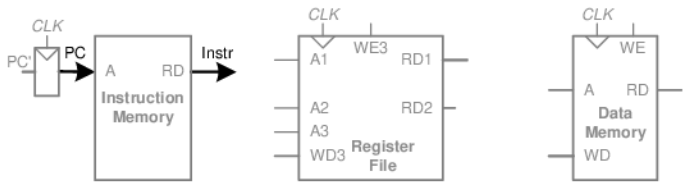


Figure 7.2 Fetch instruction from memory

The processor's actions depend on the specific instruction that was fetched. First we will work out the datapath connections for the `lw` instruction. Then we will consider how to generalize the datapath to handle the other instructions.

For a `lw` instruction, the next step is to read the source register containing the base address. This register is specified in the `rs` field of the instruction, $Instr_{25:21}$. These bits of the instruction are connected to the address input of one of the register file read ports, *A1*, as shown in Figure 7.3. The register file reads the register value onto *RD1*.

The `lw` instruction also requires an offset. The offset is stored in the immediate field of the instruction, $Instr_{15:0}$. Because the 16-bit immediate might be either positive or negative, it must be sign-extended to 32 bits, as shown in Figure 7.4. The 32-bit sign-extended value is called *SignImm*. Recall from Section 1.4.6 that sign extension simply copies the sign bit (most significant bit) of a short input into all of the upper bits of the longer output. Specifically, $SignImm_{15:0} = Instr_{15:0}$ and $SignImm_{31:16} = Instr_{15}$.

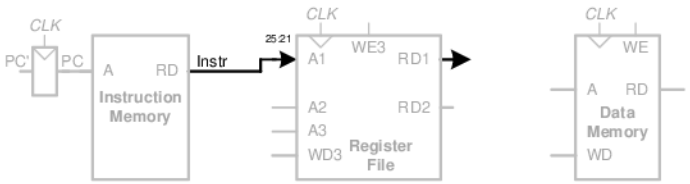


Figure 7.3 Read source operand from register file

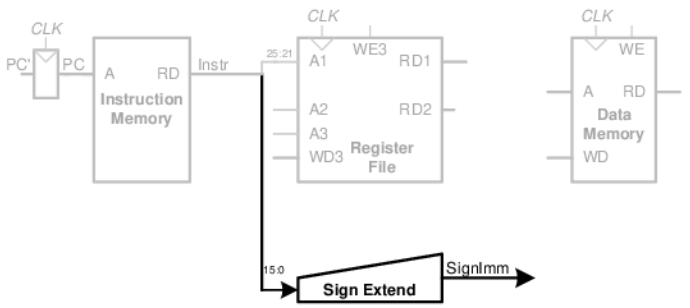


Figure 7.4 Sign-extend the immediate

The processor must add the base address to the offset to find the address to read from memory. Figure 7.5 introduces an ALU to perform this addition. The ALU receives two operands, *SrcA* and *SrcB*. *SrcA* comes from the register file, and *SrcB* comes from the sign-extended immediate. The ALU can perform many operations, as was described in Section 5.2.4. The 3-bit *ALUControl* signal specifies the operation. The ALU generates a 32-bit *ALUResult* and a *Zero* flag, that indicates whether *ALUResult* == 0. For a `lw` instruction, the *ALUControl* signal should be set to 010 to add the base address and offset. *ALUResult* is sent to the data memory as the address for the load instruction, as shown in Figure 7.5.

The data is read from the data memory onto the *ReadData* bus, then written back to the destination register in the register file at the end of the cycle, as shown in Figure 7.6. Port 3 of the register file is the

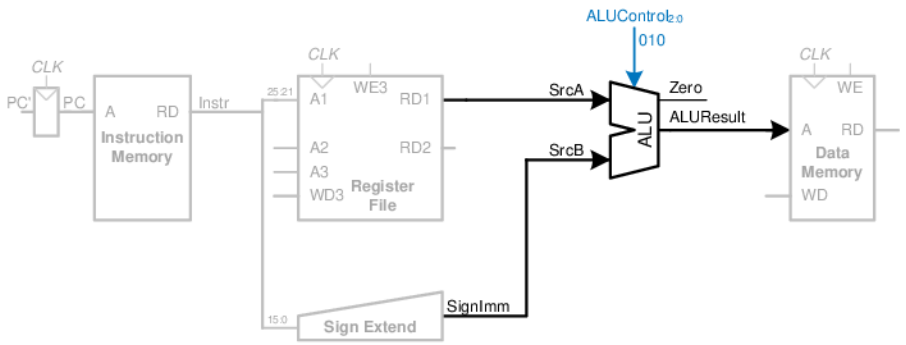


Figure 7.5 Compute memory address

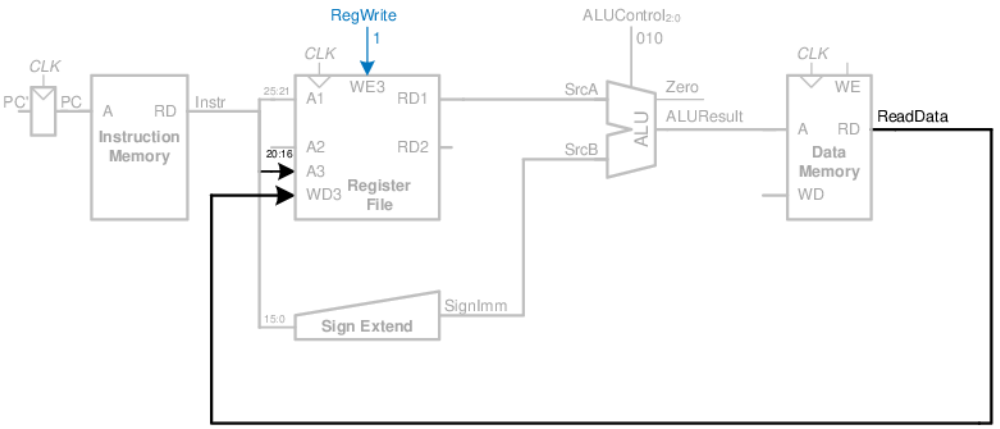


Figure 7.6 Write data back to register file

write port. The destination register for the `lw` instruction is specified in the `rt` field, $Instr_{20:16}$, which is connected to the port 3 address input, *A3*, of the register file. The *ReadData* bus is connected to the port 3 write data input, *WD3*, of the register file. A control signal called *RegWrite* is connected to the port 3 write enable input, *WE3*, and is asserted during a `lw` instruction so that the data value is written into the register file. The write takes place on the rising edge of the clock at the end of the cycle.

While the instruction is being executed, the processor must compute the address of the next instruction, *PC'*. Because instructions are 32 bits = 4 bytes, the next instruction is at *PC* + 4. Figure 7.7 uses another adder to increment the *PC* by 4. The new address is written into the program counter on the next rising edge of the clock. This completes the datapath for the `lw` instruction.

Next, let us extend the datapath to also handle the `sw` instruction. Like the `lw` instruction, the `sw` instruction reads a base address from port 1 of the register and sign-extends an immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported by the datapath.

The `sw` instruction also reads a second register from the register file and writes it to the data memory. Figure 7.8 shows the new connections for this function. The register is specified in the `rt` field, $Instr_{20:16}$. These bits of the instruction are connected to the second register file read port, *A2*. The register value is read onto the *RD2* port. It is connected to the write data port of the data memory. The write enable port of the data memory, *WE*, is controlled by *MemWrite*. For a `sw` instruction, *MemWrite* = 1, to write the data to memory; *ALUControl* = 010, to add the base address

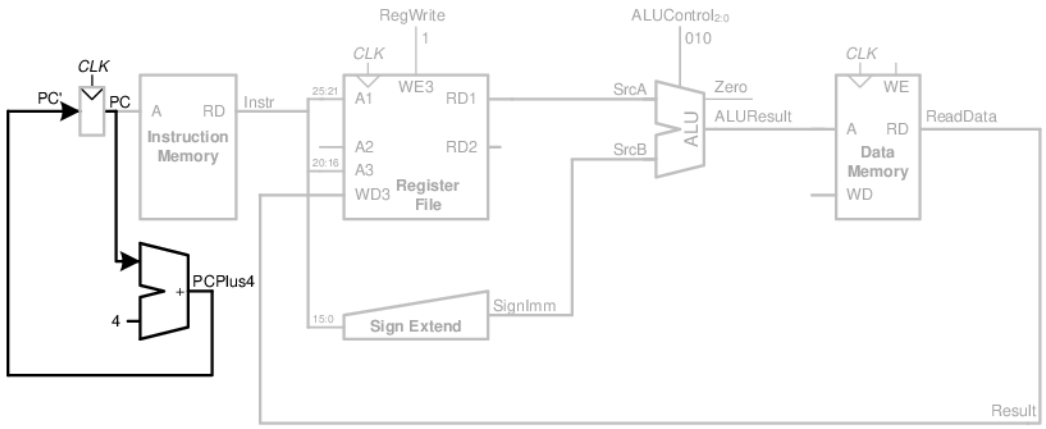


Figure 7.7 Determine address of next instruction for PC

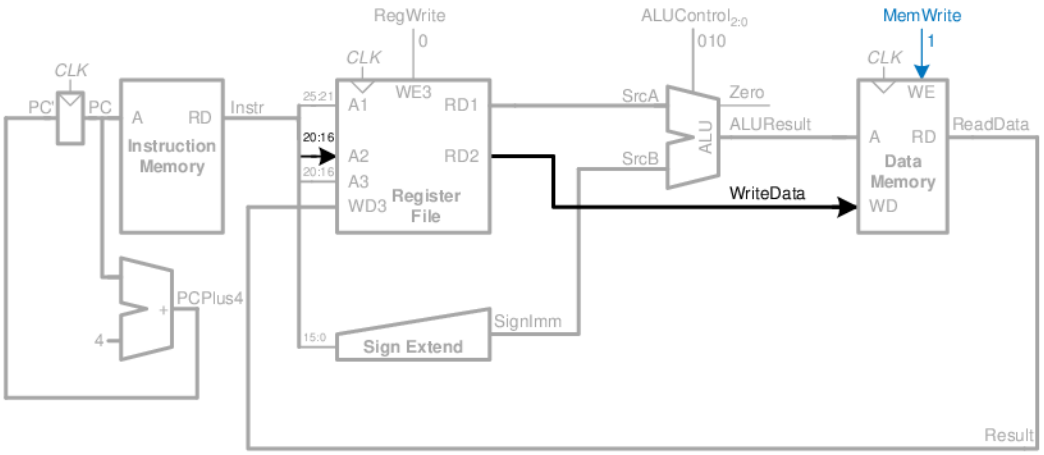


Figure 7.8 **Write data to memory for `sw` instruction**

and offset; and *RegWrite* = 0, because nothing should be written to the register file. Note that data is still read from the address given to the data memory, but that this *ReadData* is ignored because *RegWrite* = 0.

Next, consider extending the datapath to handle the R-type instructions `add`, `sub`, `and`, `or`, and `slt`. All of these instructions read two registers from the register file, perform some ALU operation on them, and write the result back to a third register file. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware, using different *ALUControl* signals.

Figure 7.9 shows the enhanced datapath handling R-type instructions. The register file reads two registers. The ALU performs an operation on these two registers. In Figure 7.8, the ALU always received its *SrcB* operand from the sign-extended immediate (*SignImm*). Now, we add a multiplexer to choose *SrcB* from either the register file *RD2* port or *SignImm*.

The multiplexer is controlled by a new signal, *ALUSrc*. *ALUSrc* is 0 for R-type instructions to choose *SrcB* from the register file; it is 1 for `lw` and `sw` to choose *SignImm*. This principle of enhancing the datapath's capabilities by adding a multiplexer to choose inputs from several possibilities is extremely useful. Indeed, we will apply it twice more to complete the handling of R-type instructions.

In Figure 7.8, the register file always got its write data from the data memory. However, R-type instructions write the *ALUResult* to the register file. Therefore, we add another multiplexer to choose between *ReadData* and *ALUResult*. We call its output *Result*. This multiplexer is controlled by another new signal, *MemtoReg*. *MemtoReg* is 0

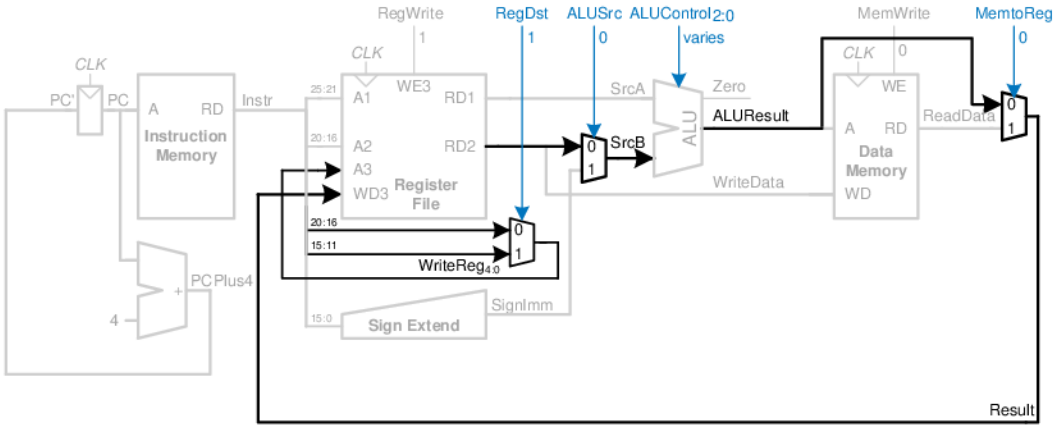


Figure 7.9 Datapath enhancements for R-type instruction

for R-type instructions to choose *Result* from the *ALUResult*; it is 1 for `lw` to choose *ReadData*. We don't care about the value of *MemtoReg* for `sw`, because `sw` does not write to the register file.

Similarly, in Figure 7.8, the register to write was specified by the `rt` field of the instruction, $Instr_{20:16}$. However, for R-type instructions, the register is specified by the `rd` field, $Instr_{15:11}$. Thus, we add a third multiplexer to choose *WriteReg* from the appropriate field of the instruction. The multiplexer is controlled by *RegDst*. *RegDst* is 1 for R-type instructions to choose *WriteReg* from the `rd` field, $Instr_{15:11}$; it is 0 for `lw` to choose the `rt` field, $Instr_{20:16}$. We don't care about the value of *RegDst* for `sw`, because `sw` does not write to the register file.

Finally, let us extend the datapath to handle `beq`. `beq` compares two registers. If they are equal, it takes the branch by adding the branch offset to the program counter. Recall that the offset is a positive or negative number, stored in the `imm` field of the instruction, $Instr_{31:26}$. The offset indicates the number of instructions to branch past. Hence, the immediate must be sign-extended and multiplied by 4 to get the new program counter value: $PC' = PC + 4 + SignImm \times 4$.

Figure 7.10 shows the datapath modifications. The next *PC* value for a taken branch, *PCBranch*, is computed by shifting *SignImm* left by 2 bits, then adding it to *PCPlus4*. The left shift by 2 is an easy way to multiply by 4, because a shift by a constant amount involves just wires. The two registers are compared by computing $SrcA - SrcB$ using the ALU. If *ALUResult* is 0, as indicated by the *Zero* flag from the ALU, the registers are equal. We add a multiplexer to choose PC' from either *PCPlus4* or *PCBranch*. *PCBranch* is selected if the instruction is

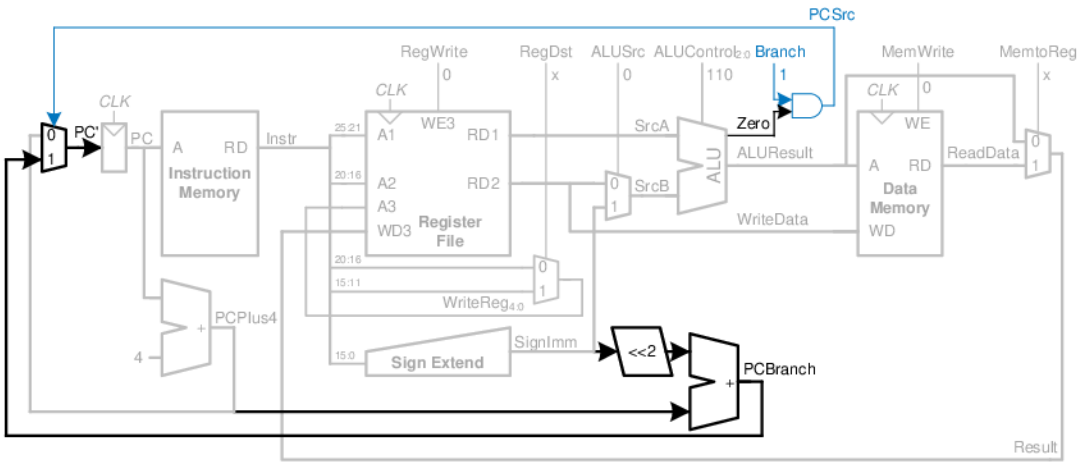


Figure 7.10 Datapath enhancements for `beq` instruction

a branch and the *Zero* flag is asserted. Hence, *Branch* is 1 for `beq` and 0 for other instructions. For `beq`, *ALUControl* = 110, so the ALU performs a subtraction. *ALUSrc* = 0 to choose *SrcB* from the register file. *RegWrite* and *MemWrite* are 0, because a branch does not write to the register file or memory. We don't care about the values of *RegDst* and *MemtoReg*, because the register file is not written.

This completes the design of the single-cycle MIPS processor datapath. We have illustrated not only the design itself, but also the design process in which the state elements are identified and the combinational logic connecting the state elements is systematically added. In the next section, we consider how to compute the control signals that direct the operation of our datapath.

7.3.2 Single-Cycle Control

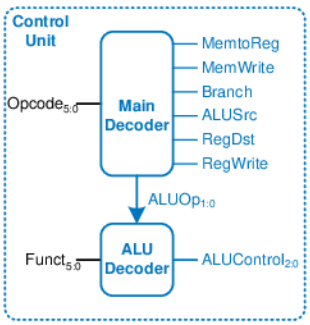


Figure 7.12 Control unit internal structure

The control unit computes the control signals based on the `opcode` and `funct` fields of the instruction, $Instr_{31:26}$ and $Instr_{5:0}$. Figure 7.11 shows the entire single-cycle MIPS processor with the control unit attached to the datapath.

Most of the control information comes from the `opcode`, but R-type instructions also use the `funct` field to determine the ALU operation. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic, as shown in Figure 7.12. The *main decoder* computes most of the outputs from the `opcode`. It also determines a 2-bit *ALUOp* signal. The ALU decoder uses this *ALUOp* signal in conjunction with the `funct` field to compute *ALUControl*. The meaning of the *ALUOp* signal is given in Table 7.1.

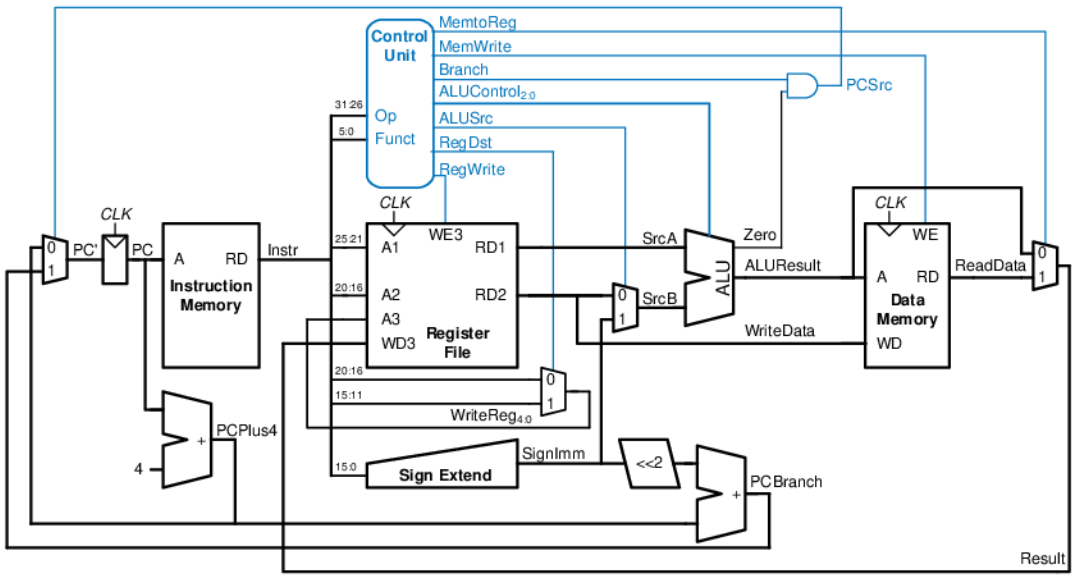


Figure 7.11 Complete single-cycle MIPS processor

Table 7.1 *ALUOp* encoding

ALUOp	Meaning
00	add
01	subtract
10	look at `funct` field
11	n/a

Table 7.2 is a truth table for the ALU decoder. Recall that the meanings of the three *ALUControl* signals were given in Table 5.1. Because *ALUOp* is never 11, the truth table can use don't care's X1 and 1X instead of 01 and 10 to simplify the logic. When *ALUOp* is 00 or 01, the ALU should add or subtract, respectively. When *ALUOp* is 10, the decoder examines the `funct` field to determine the *ALUControl*. Note that, for the R-type instructions we implement, the first two bits of the `funct` field are always 10, so we may ignore them to simplify the decoder.

The control signals for each instruction were described as we built the datapath. Table 7.3 is a truth table for the main decoder that summarizes the control signals as a function of the opcode. All R-type instructions use the same main decoder values; they differ only in the

Table 7.2 ALU decoder truth table

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

Table 7.3 Main decoder truth table

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

ALU decoder output. Recall that, for instructions that do not write to the register file (e.g., sw and beq), the *RegDst* and *MemtoReg* control signals are don't cares (X); the address and data to the register write port do not matter because *RegWrite* is not asserted. The logic for the decoder can be designed using your favorite techniques for combinational logic design.

Example 7.1 SINGLE-CYCLE PROCESSOR OPERATION

Determine the values of the control signals and the portions of the datapath that are used when executing an or instruction.

Solution: Figure 7.13 illustrates the control signals and flow of data during execution of the or instruction. The PC points to the memory location holding the instruction, and the instruction memory fetches this instruction.

The main flow of data through the register file and ALU is represented with a dashed blue line. The register file reads the two source operands specified by $Instr_{25:21}$ and $Instr_{20:16}$. *SrcB* should come from the second port of the register

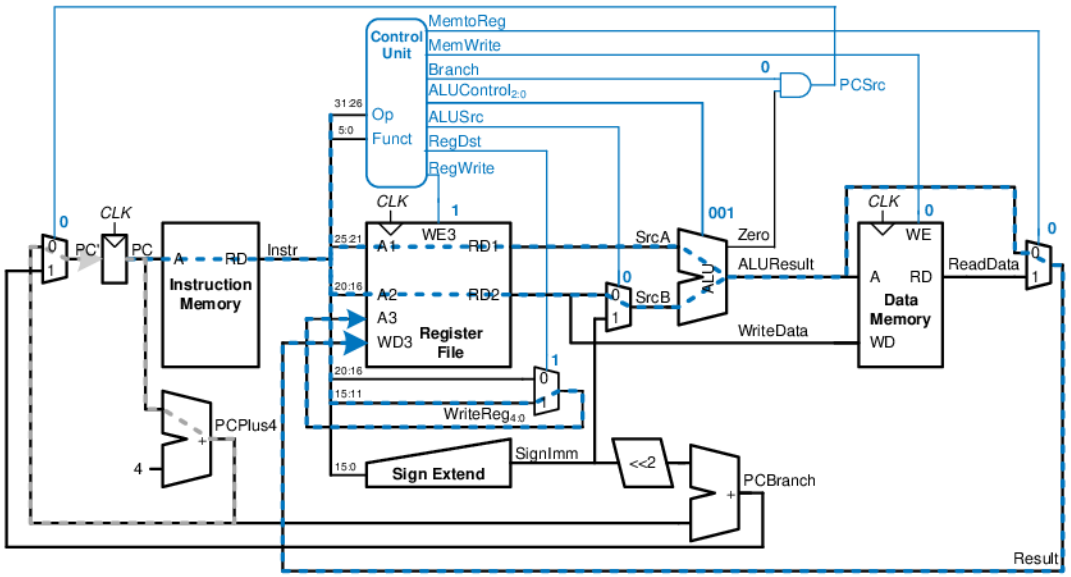


Figure 7.13 Control signals and data flow while executing `or` instruction

file (not *SignImm*), so *ALUSrc* must be 0. `or` is an R-type instruction, so *ALUOp* is 10, indicating that *ALUControl* should be determined from the `funct` field to be 001. *Result* is taken from the ALU, so *MemtoReg* is 0. The result is written to the register file, so *RegWrite* is 1. The instruction does not write memory, so *MemWrite* = 0.

The selection of the destination register is also shown with a dashed blue line. The destination register is specified in the `rd` field, $Instr_{15:11}$, so *RegDst* = 1.

The updating of the PC is shown with the dashed gray line. The instruction is not a branch, so *Branch* = 0 and, hence, *PCSrc* is also 0. The PC gets its next value from *PCPlus4*.

Note that data certainly does flow through the nonhighlighted paths, but that the value of that data is unimportant for this instruction. For example, the immediate is sign-extended and data is read from memory, but these values do not influence the next state of the system.

7.3.3 More Instructions

We have considered a limited subset of the full MIPS instruction set. Adding support for the `addi` and `j` instructions illustrates the principle of how to handle new instructions and also gives us a sufficiently rich instruction set to write many interesting programs. We will see that