The MIPS ISA is described in detail in the book *MIPS RISC Architecture* by Gerry Kane. The MIPS architecture is described in more detail in the book *Computer Organization and Architecture: A Hardware Software Interface*, by David A. Patterson and John L. Hennessy. A very concise description of the MIPS ISA is presented in the following section.

The Single Instruction Computer: It has also been shown in the past that a microprocessor can be designed with a single instruction. This single instruction should be able to access memory operands, do arithmetic operations and do control transfers. A subtract instruction that operates on memory operands, writes results to memory, and branches to an address if the result of the subtraction is negative can be used to write any program. Will such a single instruction microprocessor qualify to be called a RISC? Probably not. Although it is a single instruction computer, it is not a register-register architecture, and it is not an ISA that supports simple operations. We would classify it under a CISC category since every instruction is a complex branch and memory access instruction. More discussion of such a computer and illustration of a program written using the single instruction can be found in [Patterson/Hennessey].

The MIPS ISA

The MIPS instruction set architecture (ISA) contains a set of simple arithmetic, logical, memory-access, branch, and jump instructions. The architecture emphasizes simplicity and excludes instructions that could possibly take longer than the most common instructions.

There are 32 general-purpose registers in the MIPS architecture. Each register is 32 bits wide. The MIPS registers are often referred to as \$0, \$1, \$2, ..., and \$31. The MIPS instructions follow a **three-address format** for ALU instructions, meaning they specify two source addresses and one destination address. For example, an add instruction that adds registers \$3 and \$4 and writes the result to \$5 is written as

```
add $5, $3, $4
```

Each group of instructions is described in the following subsections.

Arithmetic Instructions

The MIPS ISA contains instructions for performing addition, subtraction, multiplication, and division of integers. The various arithmetic instructions are summarized in Table 9-1. Addition and subtraction of signed or unsigned quantities can be accomplished using the *add*, *addu*, *sub*, and *subu* instructions. Signed arithmetic instructions detect overflows, whereas unsigned arithmetic instructions do not detect overflows.

For example the instruction

```
sub $5, $3, $4
```

will subtract the value in register $4 from the value in register $3 and write the result to register $5. It is a signed instruction, and overflow will be detected. When an overflow is detected, it is handled as an exception. The address of the instruction that caused the exception is saved, and control is transferred to the operating system, which handles the exception.

TABLE 9-1: Arithmetic Instructions in the MIPS ISA

Instruction	Example	Meaning	Comments
Add	add $s1, $s2, $s3	$s1 = $s2 + $s3	Overflow detected
Subtract	sub $s1, $s2, $s3	$s1 = $s2 − $s3	Overflow detected
Add immediate	addi $s1, $s2, k	$s1 = $s2 + k	k is a 16 bit constant; sign extended and added; 2's complement overflow detected
Add unsigned	addu $s1, $s2, $s3	$s1 = $s2 + $s3	Overflow not detected
Subtract unsigned	subu $s1, $s2, $s3	$s1 = $s2 − $s3	Overflow not detected
Add immediate unsigned	addiu $s1, $s2, k	$s1 = $s2 + k	k is an unsigned 16-bit constant;
Move from co-processor register	mfc0 $s1, $epc	$s1 = $epc	epc is Exception Program Counter
Multiply	mult $s2, $s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
Multiply unsigned	multu $s2, $s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
Divide	div $s2, $s3	Lo = $s2 / $s3 Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
Divide unsigned	divu $s2, $s3	Lo = $s2 / $s3 Hi = $s2 mod $s3	Unsigned quotient and remainder
Move from Hi	mfhi $s1	$s1 = Hi	Copy Hi to $s1
Move from Lo	mflo $s1	$s1 = Lo	Copy Lo to $s1

Addition of the contents of a register with an immediate value specified in the instruction can be done using the *addi* and *addiu* instructions. The instruction

```
addi $5, $3, 400
```

will add the value in register $3 to the immediate constant 400 and write the result to register $5. The immediate constant is sign extended before the addition. The action of the *addiu* instruction is similar, except that the *addiu* instruction never causes an overflow exception.

Multiplication of two 32-bit quantities results in a 64-bit result that cannot be contained in one MIPS register. Hence, two special registers called Hi and Lo are used by the MIPS processors to hold the products. Use of implied Hi and Lo registers, is certainly a deviation from the RISC philosophy. Table 9-1 illustrates the multiply and divide instructions in the MIPS ISA along with the use of the Hi and Lo registers. The use of these special registers also necessitate special instructions

to transfer data from these registers to the required destination registers. The *mfhi* and *mflo* accomplish this task.

Logical Instructions

The logical instructions in the MIPS ISA are presented in Table 9-2. The MIPS ISA contains logical instructions for performing bit-wise AND and OR of register contents. The *and* and *or* instructions perform these operations for register operands. The *andi* and *ori* instructions can be used when one operand is in a register and the other operand is an immediate constant. The *sll* and *srl* instructions are provided to perform logical left and right shifts of register contents (with zero fill). The number of shifts is encoded as an immediate value in the instruction.

TABLE 9-2: Logical Instructions in the MIPS ISA

Instruction	Example	Meaning	Comments
and	and $s1, $s2, $s3	$s1 = $s2 AND $s3	logical AND
or	or $s1, $s2, $s3	$s1 = $s2 OR $s3	logical OR
and immediate	andi $s1, $s2, k	$s1 = $s2 AND k	k is a 16-bit constant; k is 0-extended first.
or immediate	ori $s1, $s2, k	$s1 = $s2 OR k	k is a 16-bit constant; k is 0-extended first
shift left logical	sll $s1, $s2, k	$s1 = $s2 << k	Shift Left by 5-bit constant k
shift right logical	srl $s1, $s2, k	$s1 = $s2 >> k	Shift right by 5-bit constant k

Memory Access Instructions

The only instructions in the MIPS ISA to access the memory are load and store instructions. A load instruction transfers data from memory to the specified register. A store instruction transfers data from a register to the specified memory address.

The RISC researchers investigated the number of addressing modes that are needed to efficiently code high-level language programs such as those in C. They concluded that one addressing mode with a base register and an offset was sufficient. The only addressing mode that is supported for memory instructions in the MIPS processor is this addressing mode with one base register and an offset. The memory address is computed as the sum of the register contents and the offset specified in the instruction.

Consider the MIPS load instruction

```
lw $5, 100($4)
```

This instruction computes the memory address as the sum of the value in register $4 and the offset 100. So if register $4 contains 4000, the effective address is 4100. The content of memory location 4100 is moved to register $5 in the processor. In the case of `sw $6, 100($8)`, the content of register $6 is written to the memory location pointed to by the sum of the contents of register $8 and 100.

A group of 32 bits is called a word in the MIPS world. MIPS has instructions to load and store words, half-words (16 bits) or bytes (8 bits). These instructions are summarized in Table 9-3.

TABLE 9-3: Memory Access Instructions in the MIPS ISA

Instruction	Example	Meaning	Comments
load word	lw $s1, k($s2)	$s1 = Memory[$s2 + k]	Read a word (32 bits) from memory; memory address = Register content + k ; k is 16 bit offset
store word	sw $s1, k($s2)	Memory[$s2 + k] = $s1	Write a word (32 bits) to memory; memory address = Register content + k; k is 16 bit offset
load half word	lh $s1, k($s2)	$s1 = Memory[$s2 + k]	Read a word (16 bits) from memory; sign-extend and load into register
store half word	sh $s1, k($s2)	Memory[$s2 + k] = $s1	Write a half-word (16 bits) to memory
load byte	lb $s1, k($s2)	$s1 = Memory[$s2 + k]	Read byte from memory; sign extend and load to register
store byte	sb $s1, k($s2)	Memory[$s2 + k] = $s1	Write byte to memory
load byte unsigned	lbu $s1, k($s2)	$s1 = Memory[$s2 + k]	Read byte from memory; byte is 0-extended
load upper immediate	lui $s1, k	$s1 = k * 2^{16}	Loads constant k to upper 16 bits of register

Control Transfer Instructions

Typically program execution proceeds in a sequential fashion, but loops, procedures, functions, and sub-routines change the program control flow. A microprocessor needs branch and jump instructions in order to accomplish transfer of control whenever non-sequential control flow is required. The MIPS ISA includes two conditional branch instructions, *branch on equal (beq)* and *branch on not equal (bne)* as illustrated in Table 9-4.

The MIPS instruction

```
beq $5, $4, 25
```

will compare the contents of $5 and $4 and branch to PC + 4 + 100 if $4 and $5 are equal. The constant offset provided in the branch instruction is specified in terms of the number of instructions from the current PC (program counter). MIPS

Conditional Control–Related Instructions in the MIPS ISA

Instruction	Example	Meaning	Comments
branch on equal	beq $s1, $s2, k	If ($s1 $s2) go to PC + 4 + k*4	Branch if registers are equal; PC-relative branch; Target = PC + 4 + Offset*4; k is sign-extended
branch on not equal	bne $s1, $s2, k	If ($s1 != $s2) go to PC + 4 + k*4	Branch if registers are not equal; PC relative branch; Target = PC+ 4 + Offset*4; k is sign-extended
set on less than	slt $s1, $s2, $s3	If ($s2 $s3) $s1 1; else $s1 = 0;	Compare and Set (2's complement)
set on less than immediate	slti $s1, $s2, k	If ($s2 k) $s1 1; else $s1 0;	Compare and Set; k is 16-bit constant; sign-extended and compared
set on less than unsigned	sltu $s1, $s2, $s3	If ($s2 $s3) $s1 1; else $s1 0;	Compare and Set; natural numbers
set on less than immediate unsigned	sltiu $s1, $s2, k	If ($s2 k) $s1 1; else $s1 0;	Compare and set ; natural numbers; K the16-bit constant is sign extended; no overflow

uses byte addressing; hence, the offset in words is multiplied by 4 to get the offset in bytes. The program counter is assumed to point to the next instruction at PC + 4 already; hence, the target address is computed as PC + 4 + 4 * offset. The offset is 16 bits long; however, one bit is used for sign. Branching is thus possible to only + / 32K.

Having only two conditional branch instructions is in contrast to CISC processors, which provide branch on less than, branch on greater than, branch on higher than, branch on lower than, branch on carry, branch on overflow, branch on negative, and several other such conditional branch instructions. The MIPS philosophy was that only two conditional branch instructions are necessary and that checking of other conditions can be accomplished using separate instructions. In order to facilitate checking of less than and greater than, MIPS ISA provides the set on less than *(slt)* instructions. These are explicit compare instructions that will set an explicit destination register to 1 or 0 depending on the results of the compare. The *slt* instruction is used along with a *bne* or *beq* instruction to create the effect of branch on less than, branch on greater than, and so forth. These instructions are used for implementing **loop** and **if-then-else** statements from high-level languages.

The MIPS ISA also includes three unconditional jump instructions as illustrated in Table 9-5. These instructions are used for implementing function and procedure calls as well as returns.

TABLE 9-5: Unconditional Control Transfer Instructions in the MIPS ISA

Instruction	Example	Meaning	Comments
jump	j addr	Go to addr*4; i.e., PC <= addr*4	Target address = Imm offset * 4; addr is 26-bits
jump register	jr $reg	Go to $reg; i.e., PC <= $reg	$reg contains 32-bit target address
jump and link	jal addr	return address = PC + 4; go to addr*4	Used for procedure call. Return address saved in the link register $31

The **jump (j)** instruction transfers control to the address specified in the instruction. Since the MIPS instruction is 32 bits wide, the number of bits available for encoding the address will be (32 – number of opcode bits). In the MIPS, the opcode consumes 6 bits; therefore, only 26 bits are available for the address in the jump instruction. In order to increase the range of addresses to which control can be transferred, MIPS designers consider the specified address as a word address (instead of a byte address) and multiply the specified address by 4 to obtain the resulting byte address.

The **jump register (jr)** instruction is an **indirect jump**. In contrast, the jump instruction *j* described in the previous paragraph is called a **direct** jump because the target address is directly specified in the instruction itself. In the case of the *jr* instruction, the target address is in the register. This type of branch instruction is very useful for implementing case statements from high-level languages.

The **jump and link (jal)** instruction is specifically designed for procedure calls. It computes the target address from the offset specified in the instruction, but in addition to transferring control to that address, it also saves the return address in link register $31. The return address means the address control should return to after the subroutine or procedure call is completed. The return address is equal to the current PC + 4, since every instruction is four bytes wide and PC + 4 is the address of the instruction following the current instruction (the *jal* instruction).

We have described the major classes of instructions in the MIPS ISA. In order to become familiar with the instructions, let us practice some assembly language programming.

Example

Write a MIPS assembly language program for the following program which adds two arrays x(i) and y(i), each of which has 100 elements.

```
for i=0; i<100; i++      ; repeat 100 times
y(i) = x(i) + y(i)       ; add ith element of the arrays
```

Assume that the x and y arrays start at locations 4000 and 8000 (decimal).

Answer:

```
        andi   $3, $3, 0       ; initialize loop counter $3 to 0
        andi   $2, $2, 0       ; clear register for loop bound
        addi   $2, $2, 400     ; loop bound
$label: lw     $15, 4000($3)   ; load x(i) to R15
```

```
lw      $14, 8000($3)   ; load y(i) to R14
add     $24, $15, $14   ; x(i) + y(i)
sw      $24, 8000($3)   ; save new y(i)
addi    $3, $3, 4       ; update address register,
                          address= address + 4
bne     $3, $2, $label  ; check if loop counter=loop
                          bound
```

Several microprocessors with the MIPS ISA have been designed since the MIPS R2000 was designed during the 1980s. In those days, the main processor could not integrate the floating-point unit. Hence, the floating-point units were implemented as a math coprocessor, the MIPS R2010. Currently, the floating-point unit is integrated with the main CPU. The MIPS R2000 was followed by the MIPS R3000, R4000, R8000, R10000, R12000, and R14000. They all have the MIPS ISA but have different implementations with different levels of pipelining and different techniques to obtain high performance.

MIPS Instruction Encoding

Adhering to the RISC philosophy, all instructions in the MIPS processor have the same width, 32 bits. In a move towards simplicity, there are only three different instruction formats for the MIPS instructions. The three formats are called **R-format, I-format,** and **J-format**, as illustrated in Table 9-6.

TABLE 9-6: Instruction Formats in the MIPS ISA

Format	Fields						Comments
	6 bits 31..26	5 bits 25..21	5 bits 20..16	5 bits 15..11	5 bits 10..6	6 bits 5..0	All MIPS instruction are 32 bits.
R-format	opcode	rs	rt	rd	shamt	F_Code (funct)	ALU instructions except immediate, Jump Register (JR)
I-format	opcode	rs	rt	address/immediate			Load, store, Immediate ALU, beq, bne
J-format	opcode	target address					Jump (J)

The **R-format** is primarily for ALU instructions, which require three operands. These ALU instructions have two source operands (input registers) and one destination address (result register) to be specified. The jump register instruction (jr) also uses this format. The instruction consists of six fields, the first of which is the 6-bit **opcode** field. The opcode field is followed by the three register fields ***rs***, ***rt***, and ***rd***, each of which takes 5 bits. The first two are the source register fields and the third is the destination register field. The next field is called shift amount (***shamt***) field,

TABLE 9-7: Instruction Encoding for the MIPS Instructions

Name	Format	Fields						Instruction (operation dest, src1, src2)
		Bits 31..26	Bits 25..21	Bits 20-16	Bits 15-11	Bits 10-6	Bits 5..0	
Add	R	0	2	3	1	0	32	add $1, $2, $s3
Sub	R	0	2	3	1	0	34	sub $1, $2, $3
addi	I	8	2	1	100			addi $1, $2, 100
addu	R	0	2	3	1	0	33	addu $1, $2, $3
subu	R	0	2	3	1	0	35	subu $1, $2, $3
addiu	I	9	2	1	100			addiu $1, $2, 100
mfc0	R	16	0	1	14	0	0	mfc0 $1, $epc
mult	R	0	2	3	0	0	24	mult $2, $3
multu	R	0	2	3	0	0	25	multu $2, $3
div	R	0	2	3	0	0	26	div $2, $3
divu	R	0	2	3	0	0	27	divu $2, $3
mfhi	R	0	0	0	1	0	16	mfhi $1
mflo	R	0	0	0	1	0	18	mflo $1
and	R	0	2	3	1	0	36	and $1, $2, $3
or	R	0	22	3	1	0	37	or $1, $2, $3
andi	I	12	2	1	100			andi $1, $2, 100
ori	I	13	2	1	100			ori $1, $2, 100
sll	R	0	0	2	1	10	0	sll $1, $2, 10
srl	R	0	0	2	1	10	2	srl $1, $2, 10
lw	I	35	2	1	100			lw $1, 100($2)
sw	I	43	2	1	100			sw $1, 100($2)
lui	I	15	0	1	100			lui $1, 100
beq	I	4	1	2	25			beq $1, $2, 100
bne	I	5	1	2	25			bne $1, $2, 100
slt	R	0	2	3	1	0	42	slt $1, $2, $3
slti	I	10	2	1	100			slti $1, $2, 100
sltu	R	0	22	3	1	0	43	sltu $1, $2, $3
sltiu	I	11	2	1	100			sltiu $1, $2, 100
j	J	2	2500					j 10000
jr	R	0	31	0	0	0	8	jr $31
jal	J	3	2500					jal 10000

which is used to specify the amount of shifting to be done in shift instructions. Any number between 0 and 31 can be specified as the shift amount. This field is used only in shift instructions. The last field is an additional opcode field, called the function field **funct** or **F_code**. The first opcode field can encode only 2^6 or 64 instructions. The MIPS processor does have more than 64 instructions considering the different variations of loads (byte load, half-word load, word load, floating point loads, etc.). Hence, more than 6 bits are required to fully specify an instruction. The MIPS designers chose a scheme in which the first 6 bits are 0 for the R-format instructions, and then use an additional field to specifically identify the instruction. The last 6 bits of the instruction are used to further identify the instruction in the case of the R-format ALU instructions.

The **I-format** is for load/store instructions, branch instructions, and ALU instructions that need an immediate constant to be specified in the instruction. These instructions need only two registers to be specified in addition to the immediate constant. The opcode field takes 6 bits, and the two register fields take 5 bits each. The remaining 16 bits are used as an immediate constant to specify an operand for instructions such as *addi* or to specify the offset in a load/store instruction or to specify the branch offset in a conditional branch instruction.

The **J-format** is for jump instructions. The first 6 bits of the instruction word are used for the opcode, and the remaining 26 bits of the instruction are used to specify the branch target. The branch target address is calculated by multiplying the 26-bit target by 4, since the jump offset is specified as a word address rather than byte address. MIPS uses byte addressing for accessing instructions and data.

Table 9-7 illustrates the instruction encoding for the MIPS instructions we have discussed. The opcode, source, and destination are assigned the same field in the instruction format as much as possible. The first 6 bits (bits 31 – 26) are for the opcode in all three formats. The source and destination register fields are in similar positions (bits 25–21, bits 20–16 and bits 15–11) as much as possible. This greatly simplifies decoding.

The encoding is very regular; however, compromises had to be made to accommodate various instructions into the same width. For instance, the destination register appears in different fields in 3-register and 2-register formats. Similarly, in a load instruction, the second register field is a destination register, whereas in a store instruction, it is the source of the data to be stored. In spite of these irregularities, one can say that the encoding is largely regular.

To increase the reader's familiarity with the MIPS instruction encoding, let us practice some machine coding.

Example

Create the machine code equivalent of the following assembly language program.

```
        andi   $3, $3, 0        ; initialize loop counter $3 to 0
        andi   $2, $2, 0        ; clear register for loop bound
        addi   $2, $2, 4000     ; loop bound register
$label: lw     $15, 4000($3)    ; load x(i) to R15
        lw     $14, 8000($3)    ; load y(i) to R14
        add    $24, $15, $14    ; x(i) + y(i)
```

```
sw      $24, 8000($3)     ; save new y(i)
addi    $3, $3, 4         ; update address register,
                            address= address + 4
bne     $3, $2, $label
```

Answer: The first instruction

```
andi    $3, $3, 0
```

can be translated as follows. Table 9-7 shows that the opcode for **andi** is 12. Hence, the first 6 bits for the first instruction will be 001100 as indicated in row 1 (after the header row) of Table 9-8. The source register field is next. It should be 00011, because the source register is $3. The destination register field is next. It should be 00011, because the destination register is $3. The immediate constant is 0, and it leads to sixteen 0s in bits 0 to 15. This explains the contents of row 1. In hex representation, it becomes 3063 0000.

We will also explain the encoding of the last instruction, bne $3, $2, label. The opcode is 5 (i.e., 000101). The next field corresponds to register $3, so it is 00011. The next field is 00010 to indicate the register $2. The byte offset should be −24, but the instruction is supposed to contain the word offset, which is −24 divided by 4 (i.e., −6). In 2's complement representation, it is 1010. Sign extending to fill the sixteen bits, one gets 1111111111111010, which will occupy bits 0 to 15. Machine code corresponding to all the instructions is shown in Table 9-8.

TABLE 9-8: MIPS Machine Code for Example 2; Binary as Well as Hex Representations Are Shown

Instruction	Bits 31–26	Bits 25–21	Bits 20–16	Bits 15–11	Bits 10–6	Bits 5–0	Equivalent Hex
andi $3, $3, 0	001100	00011	00011	00000	00000	000000	3063 0000
andi $2, $2, 0	001100	00010	00010	00000	00000	000000	3042 0000
addi $2, $2, 4000	001000	00010	00010	00001	11110	100000	2042 0FA0
lw $15, 4000($3)	100011	00011	01111	00001	11110	100000	8C6F 0FA0
lw $14, 8000($3)	100011	00011	01110	00011	11101	000000	8C6E 1F40
add $24, $15, $14	000000	01111	01110	11000	00000	100000	01EE C020
sw $24, 8000($3)	101011	00011	11000	00011	11101	000000	B478 1F40
addi $3, $3, 4	001000	00011	00011	00000	00000	000100	2063 0004
bne $3, $2, –6	000101	00011	00010	11111	11111	111010	1462 FFFA

Implementation of a MIPS Subset

In this section, we describe a simple implementation of a subset of the MIPS ISA. This subset, illustrated in Table 9-9, includes most of the important instructions, including ALU, memory access, and branch instructions. What we present in this section is a naïve implementation of this instruction set. Modern microprocessors implement features such as multiple instruction issue, out-of-order execution, branch prediction, pipelining, and so forth. For the sake of simplicity, what is presented here is a simple in-order, non-pipelined implementation. Some of the exercise problems describe other implementations that will provide better performance.

TABLE 9-9: Subset of MIPS Instructions Implemented in This Chapter

Arithmetic	add subtract add immediate
Logical	and or and immediate or immediate shift left logical shift right logical
Data Transfer	load word store word
Conditional Branch	branch on equal branch on not equal set on less than
Unconditional Branch	jump jump register

Design of the Data Path

In order to design a microprocessor, first we will examine the sequence of operations during execution of instructions. Then we will describe the nature of the hardware required to accomplish the instruction execution. In general, any microprocessor works in the following manner:

1. The processor **fetches** an instruction.
2. It **decodes** the instruction that was fetched. Decoding means identifying what the instruction is.
3. It reads the operands and **executes** the instruction. For a RISC ISA, for arithmetic instructions, the operands are in registers. The registers that contain the input operands are called source registers. For memory access instructions, addresses are computed using registers and memory is accessed. After execution, the processor **writes** the result of the instruction execution into the destination. The destination is a register for all instructions other than the store instruction, which has to write the result into the memory.

Hence, the design must contain a **unit to fetch** the instructions, a **unit to decode** the instructions, an arithmetic and logic unit (**ALU**) to execute the instructions, a **register file** to hold the operands, and the **memory** that stores instructions and data. These components are described in the following subsections.

Instruction Fetch Unit

In general, a microprocessor has a special register called the program counter (PC), which points to the next instruction in the instruction memory (or in the instruction caches). The PC sends this address to the instruction memory, which sends the instruction back. The processor increments the PC to point to the next instruction to be fetched. A block diagram for this unit is shown in Figure 9-1.

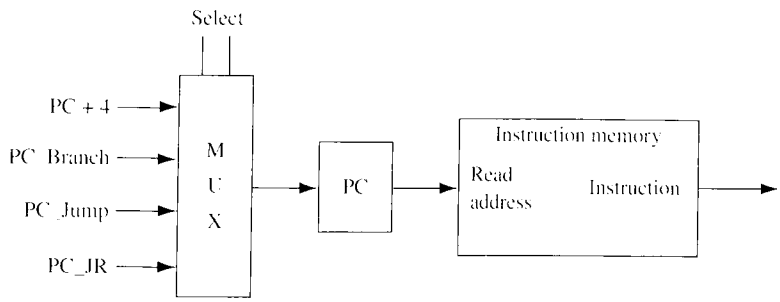


FIGURE 9-1: Block Diagram for Instruction Fetch

The next PC is one of the following, depending on the current instruction:

(a) PC + 4: For instructions other than branch and jump instructions, the next instruction is at address PC + 4, since four bytes are needed for the current instruction.

(b) PC_Branch: In the branch (bne and beq) instructions, the next PC is obtained by adding the offset in the instruction to the current PC. In the MIPS ISA, the branch offset is provided as a signed word offset (number of words to jump forward or backward). First the word offset is sign extended, then converted to a byte offset by multiplying by 4, and finally added to the current PC. Thus, the next PC for branch instructions is

```
PC_Branch = PC + 4 + Offset * 4.
```

(c) PC_Jump: In the jump (J) instruction, the jump target address is provided in the instruction itself. In the MIPS ISA, the opcode field takes 6 out of the 32 bits. Hence, the biggest jump address that can be encoded is only 26 bits. In order to compute the 32-bit jump address, first, the 26-bit word address in the instruction is shifted twice to the left, resulting in a 28-bit address, which is a byte address. Then it is concatenated with the four highest bits of the PC, yielding a 32-bit address. Thus, the next PC for jump instructions is

```
PC_Jump = PC[31..28] || Address * 4,
```

where || stands for concatenation. Note that this symbol is different from the concatenate symbol in Verilog. This is the symbol typically used in the MIPS ISA references [Kane, Gerry, *MIPS RISC Architecture* Prentice Hall, 1989].

(d) PC_JR: In the jump register (JR) instruction, the jump target is obtained from the register specified in the instruction. Thus, the next PC for a JR instruction is

```
PC_JR = [REG],
```

where [REG] indicates contents of the register.

The appropriate target addresses are computed and fed to the PC. A multiplexer is used to select between the branch target, jump target, jump register target, or PC + 4, depending on the instruction.

There are several choices as to when the target addresses are computed. The default target, PC + 4, can be computed at instruction fetch itself, since it needs no

information other than the PC itself. In conditional branch instructions, the branch target (PC_Branch) computation can be done as soon as the instruction is read; however, whether the branch is taken or not will not be known until the registers are read and compared. In the case of the jump instruction, the target (PC_Jump) can be computed as soon as the instruction is fetched, since the information for the target is available in the instruction itself. In a jump register (*jr*) instruction, the branch target (PC_JR) can be computed after the register is read.

Figure 9-1 also shows the instruction memory unit. In the initial design, we will use a separate instruction memory and separate data memory, in alignment with the popular scheme of separate instruction and data caches found in modern processors. We will not be designing a cache memory; however, we will assume the presence of on-chip instruction memory that can be accessed by the processor in one cycle after the address is provided to it.

Instruction Decode Unit

Decoding is fairly simple due to the simplicity of the RISC ISA. One can observe from Table 9-7 that the instruction formats in the MIPS ISA are very regular and uniform. The first 6 bits of the instruction specify the opcode in most cases. But, as described in Section 9.3, for the R-format ALU instructions, the first 6 bits are 0, and the last 6 bits of the instruction, called **F_code**, need to be used to further identify the instruction.

The opcode is used to identify the instruction and the instruction format used by the instruction. The uniformity of the instruction format allows many of the instruction fields to be directly used for register addressing and control-signal generation. The instruction opcode bits are fed to a control unit that generates the various control signals.

Instruction Execution Unit

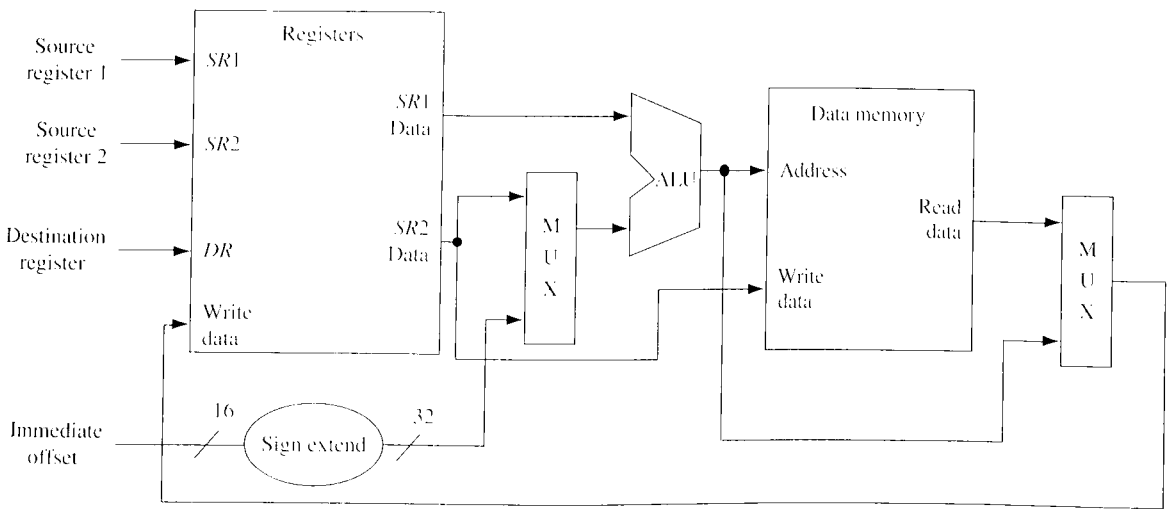
Once the instruction is identified at the decode stage, the next task is to read the operands and perform the operation. In RISC instruction sets, the operands are in registers. The MIPS architecture contains 32 registers, and these registers are collectively referred to as the **register file**. The register file should have at least two read ports to support reading two operands at the same time, and it should have one write port.

The operation of the register file is as follows. The registers that hold the input operands are called **source registers**, and the register that should receive the result is called the **destination register**. The source register addresses are applied to the register file. The register file will produce the data from the corresponding registers on the output data lines. This data is fed to the arithmetic and logic unit (**ALU**), which executes the instruction. The ALU contains functional units such as adders, shifters, and the like. It may also include more complex units such as multipliers, although our restricted design here does not include multiplication.

In most instructions, the result from the ALU should be written into the destination register. To accomplish this, the ALU result is applied to the input data lines of the register file. The destination register name and the **register write (RegW) command** is applied to the register file. That causes the input data to get written into the destination register.

Figure 9-2 shows a block diagram of the data path that is required to execute the ALU and memory instructions. The data path includes an ALU, which will perform the following operations: add, sub, and, and or. In the case of R-format instructions, both operands for the ALU are read from the register file. In the case of the I-format instructions, the immediate constant in the instruction is sign extended to create the second operand. Since one of the ALU operands comes from either the register file or the sign extender, a multiplexer is required to select the appropriate operand.

FIGURE 9-2: Required Data Path for Computation and Memory Instructions



The ALU is also required for non-arithmetic instructions. For memory access instructions, we have to first calculate the address to be accessed. The ALU can be used for calculating the address. For address calculation for load and store instructions, the first operand is obtained from the register specified in the instruction, and the second operand is obtained by sign extending the immediate offset specified in the instruction.

The ALU is required for conditional branch instructions as well. As you know, MIPS has only two branch instructions—branch on equal (*beq*) and branch not equal (*bne*). The comparison for determining whether the registers are equal can be done by the ALU. Both operands for this comparison can be obtained from the register file. The data path must also include a **data memory unit**, because load and store instructions have to access the data memory unit. Modern microprocessors contain on-chip data caches. We will not be designing a cache memory, but we will assume the presence of on-chip data memory that can be accessed by the instructions in one cycle after the address is provided to the memory.

Figure 9-2 also shows use of several multiplexers and how the different bits of the instruction are connected to the register file. As Table 9-6 illustrates, bits 21 to 25 of the instruction contains one of the source register addresses in all ALU instructions. Hence, these bits can be directly connected to the first source register address of the register file. Any instruction with a second register source contains the register address in bits 16 to 20. Hence, these bits can also be directly connected to the source register address of the register file. However, the destination register address appears in

different fields in different instructions. In R-format instructions, the destination register address appears in bits 11 to 15. In I-format instructions, however, the destination address is in bits 16 to 20. Hence, a multiplexer is required to choose the appropriate destination register address. Another multiplexer chooses between the immediate operand or the register operand for the ALU. A third multiplexer is used to select whether ALU output or memory data will be written to the destination register.

Overall Data Path

The overall data path is shown in Figure 9-3. It integrates the fetch and execute hardware from Figures 9-1 and 9-2 and adds other required elements for correct operation. In addition, control signals are also shown.

FIGURE 9-3: Overall Data Path

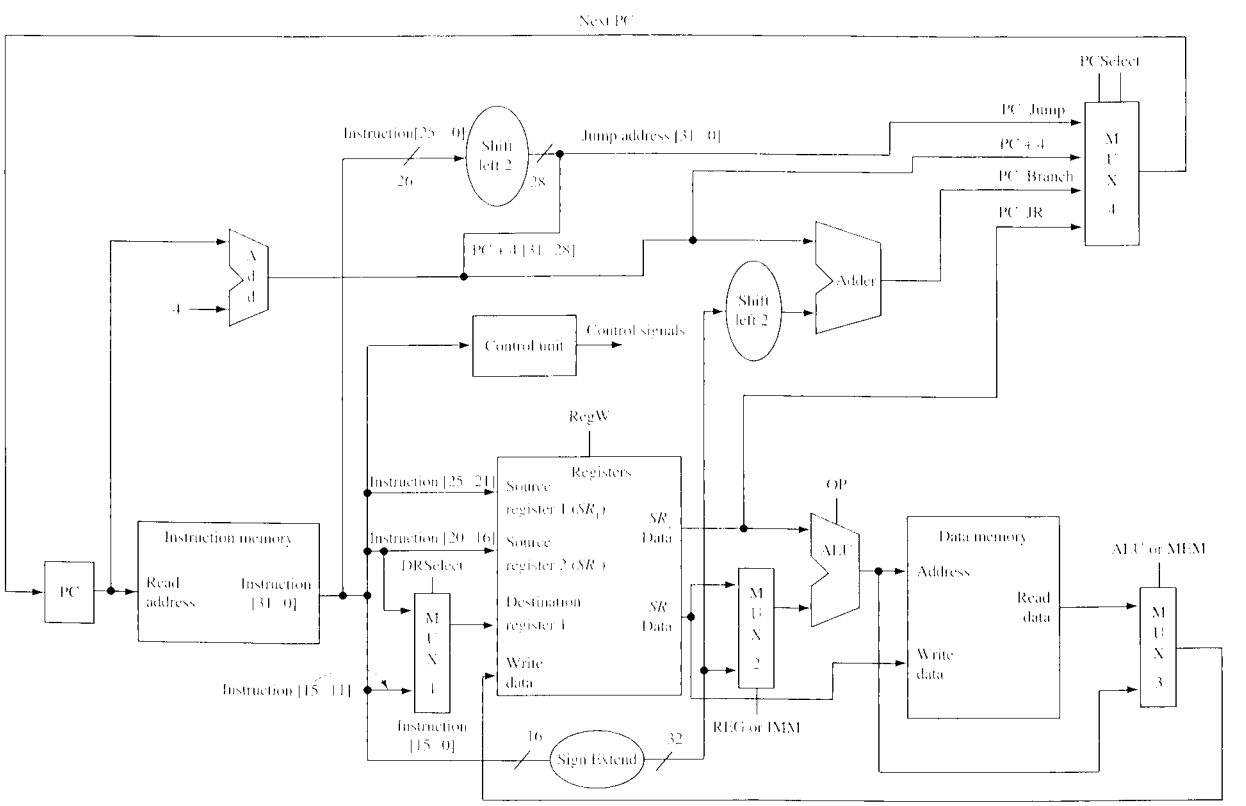


Figure 9-3 also illustrates the details of the computation of the target addresses in the various kinds of instructions. Default next address of PC + 4 is calculated with an adder. Addition of the branch offset to the PC is also done using a separate adder.

Several multiplexers are shown in this data path:

- MUX 1 selects a destination register address from an appropriate register field depending on the instruction format. For R-format instructions, bits 20-16 yield the destination address, and for I-format instructions, bits 15-11 of instruction provide the destination address.

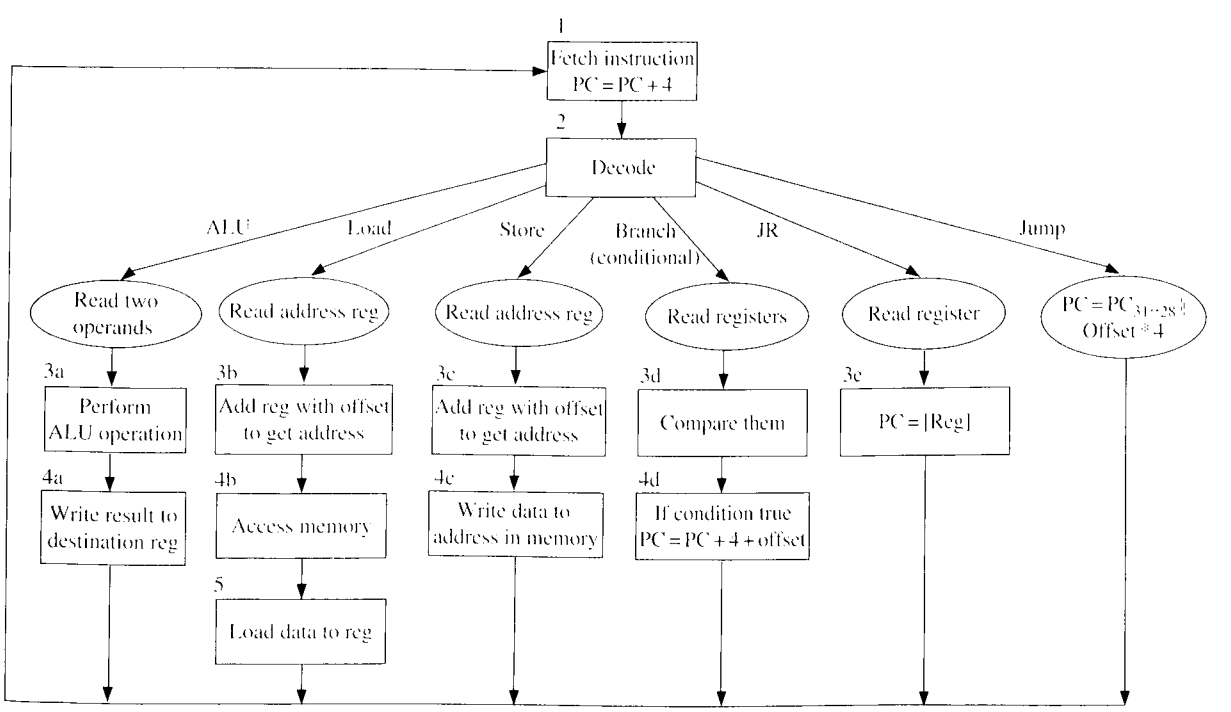
- MUX 2 selects whether the second operand for ALU comes from a register or an immediate constant. For R-format ALU instructions and conditional branch instructions, a register is chosen. For I-format ALU instructions, the immediate constant provides the operand.
- MUX 3 selects between the memory or the ALU output for data to go into the destination register. For load instructions, the memory data is chosen.
- MUX 4 selects between the four possible next PC values depending on the type of instruction.

Instruction Execution Flow

Figure 9-4 illustrates the flow of execution for a possible implementation. The first step is fetch for all instructions. The address in the program counter (PC) is sent to the instruction memory unit. All instructions also need to update the PC to point to the next instruction. While the PC should be updated differently for branch or jump instructions, the vast majority of instructions are in sequence; hence, the PC can be updated to point to the next instruction in sequence. Branch and jump instructions can later modify the PC appropriately.

The second step is decode. Depending on the opcode that is encountered, different actions follow. For R-type instructions, and for some I-type instructions (e.g., *bne* and *beq*), both ALU operands are read from registers. For other I-type instructions, one operand is read from the register file, and the immediate constant in the instruction is sign extended as the other operand. Reading of a register source satisfies the requirements for a jump register *(jr)* instruction,

FIGURE 9-4: Flow Chart for Instruction Processing



which is an R-type instruction. The ALU operation required for each instruction is identified during the decode step. For instance, the *bne* and *beq* instructions need a subtract operation. The load and store instructions require an add operation. If the jump opcode is encountered, a jump target is calculated. Since the jump instruction does not need any further action, flow of control can go to step 1.

Step 3 is the actual execution of the instructions. Depending on the instruction, different ALU operations are performed during this step. The different actions are shown in boxes labeled 3a, 3b, and so on for the different types of instructions. Each instruction goes through only one of these operations depending on what type of instruction it is. All instructions other than the jump instruction must come to this step. The jump register *(jr)* instruction does not need any arithmetic operation. The content of the register fetched during step 2 must be loaded into the PC. For load and store instructions, the ALU performs an addition to calculate the memory address.

Step 4 varies widely between the instructions. Arithmetic and logic instructions (of R-type and I-type) can write their computation result to the destination register. Branch instructions must examine their condition and decide to take the branch or not. If the branch is to be taken, the branch target address is calculated. For load instructions, a memory read operation is initiated. For memory store instructions, the data from the second source register is steered to the memory, and a memory write operation is initiated. This is the final step for all instructions other than load instructions.

Step 5 is required only for load instructions. The data output from memory is written into the destination register.

One can implement this instruction flow in various ways. In the most naïve implementation, one can have a very slow clock, and the processor will perform all operations required for each instruction during one clock cycle. The disadvantage with this scheme is that all instructions will be as slow as the slowest instruction, because the clock cycle has to be long enough for the slowest instruction. Another option is to do an implementation whereby each instruction takes multiple cycles but just enough cycles to finish all operations for each class of instruction. For instance, Figure 9-4 can be considered as an ASM chart with each box taking one cycle. In this case, a jump instruction can finish in two cycles, while an ALU instruction needs four cycles, and a load instruction takes five cycles. In the following section, we present the Verilog model of such an implementation.

Verilog Model

The Verilog model for the processor is organized as shown in Figure 9-5. The instruction memory, data memory, and register file are created as components with their architecture and entity descriptions. The main code, the MIPS entity, embeds the control sequencing the instructions through the various stages of its operation.

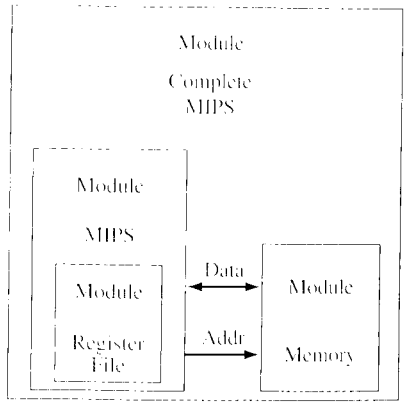


FIGURE 9-5 Organization of the Verilog Model for the Processor

We combine the instruction and data memory units to be a single memory and illustrate the use of the address and data buses. Later we use a test bench to test the functionality of the designed modules. The model we create is synthesizable into hardware. If the models are synthesized into an FPGA, appropriate display modules such as an LCD or 7-segment displays should be included so that we can observe outputs.

Let us model the register and memory components first.

Verilog Model for the Register File

Figure 9-6 shows the Verilog model for the register file. The REG entity is used to represent the 32 MIPS registers. Each register is 32 bits long. The destination register address is *DR*, and the source register addresses are *SR1* and *SR2*. Since there are 32 registers, *DR*, *SR1*, and *SR2* are 5 bits each. The outputs *Reg1* and *Reg2* are the contents of the registers specified by *SR1* and *SR2*. *Reg1* is fed straight to the ALU. *Reg2* can be used as a second ALU input or as the input to data memory in the case of store instructions. The control signal *RegW* is used to control the write operation to the register file. If *RegW* is true, the data on lines *Reg_In* is written into the register pointed to by *DR*.

It is desirable to perform register reads synchronously. It is also desirable to have the register files close to the arithmetic and logical units. Hence in FPGA-based designs, register files should be in the distributed RAM. If the reads are performed synchronously as in the provided code, some synthesis tools may not map the register file into the distributed RAM (it might be mapped into block RAM). Modern synthesis tools allow designers to explicitly choose distributed RAM or block RAM as desired from a menu option. In FPGA-based designs, it is important to check that the major modules are mapped into the regions expected by the designer. One can check synthesis reports from FPGA tools to ascertain this.

Verilog Model for Memory

Figure 9-7 illustrates the Verilog code for the memory unit. The Verilog model is similar to the SRAM model that we created in Chapter 8. Although Figures 9-3 and 9-5 illustrated separate instruction and data memories, for convenience and for

FIGURE 9-6 Verilog Code for Register File

```
module Register(CLK, RegW, DR, SR1, SR2, Reg_In, ReadReg1, ReadReg2);
    input CLK, RegW;
    input [4:0] DR, SR1, SR2;
    input [31:0] Reg_In;
    output reg [31:0] ReadReg1, ReadReg2;

    reg [31:0] REG [0:31];
    integer i;

    initial begin
        ReadReg1 = 0;
        ReadReg2 = 0;
          for (i = 0; i < 32; i = i+1) begin
              REG[i] = 0;
          end
    end

    always @(posedge CLK) begin
        if(RegW == 1'b1)
            REG[DR] <= Reg In[31:0];
        ReadReg1 <= REG[SR1];
        ReadReg2 <= REG[SR2];
    end
endmodule
```

illustrating use of address and data buses, we have used a unified memory module that stores both instructions and data. The memory consists of 128 locations, each 32 bits wide. We assume that the instructions are the first 64 words in the array, and the other 64 words are allocated for data memory. The signal *ADDR* specifies the location in memory to be read from or stored to. The address bus is actually 32 bits wide, but we use only the 7 lower bits since we implement only a small memory

The address bus will be driven by the processor appropriately for instruction and data access. The address input may come from the program counter for reading the instruction or from the ALU that computes the address to access the data portion of the memory. The chip select (*CS*) and write enable (*WE*) signals allow the processor to control the reads and writes. When *CS* and *WE* are true, the data on *Mem_Bus* gets written to the memory location pointed to by address *ADDR*. The writing is shown to occur synchronously at the negative edge of the clock.

As you know from Chapter 6, the Xilinx Spartan/Virtex FPGAs contain dedicated block RAM. It is desirable to have the block RAM. Modern synthesis tools allow designers to explicitly choose distributed RAM or block RAM as desired.

Loading specific contents into the memory can be accomplished by reading from a file. For instance, in order to load a program from file into our memory module, it can be done using the statement

```
$readmemh("MIPS_Instructions.txt", RAM);
```

FIGURE 9-7: Verilog Code for the Unified Instruction/Data Memory

```
module Memory(CS, WE, CLK, ADDR, Mem_Bus);
     input CS, WE, CLK;
     input [31:0] ADDR;
     inout [31:0] Mem_Bus;

     reg [31:0] data_out;
     reg [31:0] RAM [0:127];

     integer i;
     reg[6:0] counter;

     initial begin
          for (i=0; i<128; i=i+1)
             begin
                 RAM[i] = 32'd0; //initialize all locations to 0
             end
               $readmemh("MIPS_Instructions.txt", RAM);
               //this optional statement can be inserted to read initial values
               //from a file
      end

    assign Mem_Bus = ((CS == 1'b0) || (WE == 1'b1)) ? 32'bZ : data_out;

    always @(negedge CLK) begin
          if((CS == 1'b1) && (WE == 1'b1))
              RAM[ADDR] <= Mem_Bus[31:0];
          data_out <= RAM[ADDR];
    end
endmodule
```

This statement (currently a comment in the text) can be included into the **initial** block of the code if memory has to be loaded with specific contents. The `$readmemh` which reads hex values was explained in Chapter 8. In this case, the file `MIPS_Instructions.txt` is assumed to be a text file with a 32-bit instruction in hex format in each row. In our case, we will load MIPS instructions into the memory. The contents of the file look like

```
30000000
20010006
```

if the MIPS instructions we want to load into memory are

```
andi $0, $0, 0
addi $1, $0, 6
```

For simplicity, the address is shown as a word address in the Verilog code for the memory. In the actual MIPS processor, the memory is byte addressable. Therefore, each instruction memory access should obtain the data found in the specified location concatenated with the next three memory locations. For example, if address = 0, the instruction register must be loaded with the contents of MEM[0],

MEM[1], MEM[2], and MEM[3]. The instructions are stored depending on the endian-ness of the machine. Many modern microprocessors support both big-endian and little-endian approaches.

Little-Endian and Big-Endian

When we store 16-bit or 32-bit data into byte-addressable memory, there are two possible ways to store the data: little-endian and big-endian. In a little-endian system, the least significant byte in the sequence is stored first. In a big-endian system, the most significant byte in the sequence is stored at the lowest storage address (i.e., first). Let us consider how a MIPS instruction will be stored into byte-addressable memory in the two systems. The MIPS instruction **andi $3, $3, 0** will be encoded as **30630000 (hex).** When this instruction is stored at address 2000, depending on whether big-endian or little-endian system is used, the memory will look as follows:

Address	Big-Endian Representation of 30630000 hex	Little-Endian Representation of 30630000 hex
2000	30	00
2001	63	00
2002	00	63
2003	00	30

Verilog Code for the Processor CPU

In this section, we present the Verilog code for the central processing unit (CPU) of the microprocessor. The register module that was created in the earlier section is used here. Figure 9-8 shows a Verilog model for the MIPS instructions in Table 9-9. It generally follows the flow in Figure 9-4, implementing the fetch, decode, and execute phases of an instruction. The various statements and variables used in the code are explained in the succeeding pages.

FIGURE 9-8: Verilog Code for the MIPS Subset Implementation

```
`define opcode  instr[31:26]
`define sr1  instr[25:21]
`define sr2  instr[20:16]
`define f_code  instr[5:0]
`define numshift  instr[10:6]

module MIPS (CLK, RST, CS, WE, ADDR, Mem_Bus);
    input CLK, RST;
    output reg CS, WE;
```

```
    output [31:0] ADDR;
    inout [31:0] Mem_Bus;

    //special instructions (opcode == 000000), values of F code (bits 5-0):
    parameter add  = 6'b100000;
    parameter sub  = 6'b100010;
    parameter xor1 = 6'b100110;
    parameter and1 = 6'b100100;
    parameter or1  = 6'b100101;
    parameter slt  = 6'b101010;
    parameter srl  = 6'b000010;
    parameter sll  = 6'b000000;
    parameter jr   = 6'b001000;
    //non-special instructions, values of opcodes:
    parameter addi = 6'b001000;
    parameter andi = 6'b001100;
    parameter ori  = 6'b001101;
    parameter lw   = 6'b100011;
    parameter sw   = 6'b101011;
    parameter beq  = 6'b000100;
    parameter bne  = 6'b000101;
    parameter j    = 6'b000010;
    //instruction format
    parameter R = 2'd0;
    parameter I = 2'd1;
    parameter J = 2'd2;
    //internal signals
    reg [5:0] op, opsave;
    wire [1:0] format;
    reg [31:0] instr, pc, npc, alu_result;
    wire [31:0] imm_ext, alu_in_A, alu_in_B, reg_in, readreg1, readreg2;
    reg [31:0] alu_result_save;
    reg alu_or_mem, alu_or_mem_save, regw, writing, reg_or_imm,
reg_or_imm_save;
    reg fetchDorI;
    wire [4:0] dr;
    reg [2:0] state, nstate;

   //combinational
    assign imm_ext = (instr[15] == 1)? {16'hFFFF, instr[15:0]} : {16'h0000,
           instr[15:0]};//Sign extend immediate field
    assign dr = (format == R)? instr[15:11] : instr[20:16];
   //Destination Register MUX (MUX1)
    assign alu_in_A = readreg1;
    assign alu_in_B = (reg_or_imm_save)? imm_ext : readreg2;
   //ALU MUX (MUX2)
    assign reg_in = (alu_or_mem_save)? Mem_Bus : alu_result_save;
   //Data MUX
    assign format = (`opcode == 6'd0)? R : ((`opcode == 6'd2)? J : I);
    assign Mem_Bus = (writing)? readreg2 : 32'bZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
   //drive memory bus only during writes
```

```
    assign ADDR = (fetchDorI)? pc : alu_result_save;
   //ADDR Mux
    REG Register(CLK, regw, dr, `sr1, `sr2, reg_in, readreg1, readreg2);

    initial begin
        op = and1;
        opsave = and1;
        state = 3'b0;
        nstate = 3'b0;
        alu_or_mem = 0;
        regw = 0;
        fetchDorI = 0;
        writing = 0;
        reg_or_imm = 0;
        reg_or_imm_save = 0;
        alu_or_mem_save = 0;
    end

always @(*)
begin
      fetchDorI = 0; CS = 0; WE = 0; regw = 0; writing = 0; alu_result = 32'd0;
      npc = pc; op = jr; reg_or_imm = 0; alu_or_mem = 0;
      case (state)

         0: begin //fetch
               npc = pc + 32'd1; CS = 1; nstate = 3'd1;
               fetchDorI = 1;
             end

         1: begin //decode
               nstate = 3'd2; reg_or_imm = 0; alu_or_mem = 0;
               if (format == J) begin //jump, and finish
                    npc = {pc[31:26], instr[25:0]};
                    nstate = 3'd0;
               end
               else if (format == R) //register instructions
                    op = `f_code;
               else if (format == I) begin //immediate instructions
                    reg_or_imm = 1;
                    if(`opcode == lw) begin
                      op = add;
                      alu_or_mem = 1;
                    end
                    else if ((`opcode == lw)||(`opcode == sw)||(`opcode == addi))
                              op = add;
                    else if ((`opcode == beq)||(`opcode == bne)) begin
                       op = sub;
                       reg_or_imm = 0;
                    end
                    else if (`opcode == andi) op = and1;
                    else if (`opcode == ori) op = or1;
               end
          end
```

```
            2: begin //execute
                nstate = 3'd3;
                if (opsave == and1) alu_result = alu_in_A & alu_in_B;
                else if (opsave == or1) alu_result = alu_in_A | alu_in_B;
                else if (opsave == add) alu_result = alu_in_A + alu_in_B;
                else if (opsave == sub) alu_result = alu_in_A - alu_in_B;
                else if (opsave == srl) alu_result = alu_in_B >> `numshift;
                else if (opsave == sll) alu_result = alu_in_B << `numshift;
                else if (opsave == slt) alu_result = (alu_in_A < alu_in_B)? 32'd1 :
                        32'd0;
                else if (opsave == xor1) alu_result = alu_in_A ^ alu_in_B;
                if (((alu_in_A == alu_in_B)&&(`opcode == beq)) || ((alu_in_A != alu_
                   in_B)&&(`opcode == bne))) begin
                    npc = pc + imm_ext;
                    nstate = 3'd0;
                end
                else if ((`opcode == bne)||(`opcode == beq)) nstate = 3'd0;
                else if (opsave == jr) begin
                    npc = alu_in_A;
                    nstate = 3'd0;
                end
            end

            3:  begin //prepare to write to mem
                nstate = 3'd0;
                if ((format == R)||(`opcode == addi)||(`opcode == andi)||(`opcode == ori))
                   regw = 1;
                else if (`opcode == sw) begin
                    CS = 1;
                    WE = 1;
                    writing = 1;
                end
                else if (`opcode == lw) begin
                    CS = 1;
                    nstate = 3'd4;
                end
            end

           4:   begin
                nstate = 3'd0;
                CS = 1;
                if (`opcode == lw) regw = 1;
            end
        endcase
    end //always

    always @(posedge CLK) begin

           if (RST) begin
                state <= 3'd0;
                pc <= 32'd0;
            end
            else begin
```

```
                state <= nstate;
                pc <= npc;
            end

            if (state == 3'd0) instr <= Mem_Bus;
            else if (state == 3'd1) begin
                opsave <= op;
                reg_or_imm_save <= reg_or_imm;
                alu_or_mem_save <= alu_or_mem;
            end
            else if (state == 3'd2) alu_result_save <= alu_result;

    end //always
endmodule
```

For readability of the code, we have also used constant declarations to associate the various opcodes with the corresponding codes from Table 9-7. For instance, the load instruction *lw* has 35 as its opcode, and the store instruction *sw* has 43 as its opcode. Several statements, such as the following, are used in order to denote the various opcodes.

```
parameter lw    = 6'b100011;
parameter sw    = 6'b101011;
```

This could also be done using compiler directive `define as follows:

```
`define lw    6'b100011
`define sw    6'b101011
```

but if `define is used, one should remember to use (`) with each macro substitution. For example,

```
op <= add;
```

in the code should be substituted with

```
op <= `add; // The (`) is required
```

if `define add is used.

The bits of the instruction word are decoded into the relevant fields using continuous assign statements. The most significant 6 bits of the instruction form the *opcode*. The lowest 6 bits of the instruction are denoted with the alias *f_code*. The shift amount in shift instructions is denoted using *numshift*. The two register source fields are aliased to *sr1* and *sr2*. The following statements accomplish this aliasing:

```
`define opcode  instr[31:26]

`define sr1  instr[25:21]

`define sr2  instr[20:16]

`define f_code  instr[5:0]

`define numshift  instr[10:6]
```

Sign extension of the immediate quantity is accomplished by the following statement:

```
assign imm_ext = (instr[15] == 1)? {16'hFFFF, instr[15:0]} :
{16'h0000, instr[15:0]};//Sign extend immediate field
```

The following are the signals used in the Verilog model:

MIPS Processor Model Signals

CLK (input)	Clock
RST (input)	Synchronous reset
CS (output)	Memory chip select. When `cs` is active and `WE` is inactive, the memory module outputs the memory contents at the address specified by Addr to `mem_bus`.
WE (output)	Memory write enable. When `WE` and `cs` are active, the memory module stores the contents of `mem_bus` to the location specified by Addr during the falling edge of the clock.
ADDR (ouput)	Memory address. During state 0 (fetch instruction from memory), ADDR is connected to the PC. Otherwise, it is connected to the ALU result (32 bits).
Mem_Bus (in/out)	Memory bus; carries data to and from the memory module. The MIPS module outputs to the bus during memory writes. The memory module outputs to the bus during memory reads. When not in use, the bus is at 'hi-Z' (32 bits).
Op	ALU operation select; determines the specific operation (e.g, add, and, or) to be performed by ALU. Determined during decode.
Format	Indicates whether the current instruction is of R, I, or J format.
Instr	The current instruction (32 bits)
imm_ext	Sign extended immediate constant from the instruction (32 bits)
Pc	Current program counter (32 bits).
Npc	Next program counter (32 bits).
readreg1	Contents of the first source register (sr1) (32 bits).
readreg2	Contents of the second source register (sr2) (32 bits).
reg_in	Data input to registers. When executing a load instruction, `reg_in` is connected to the memory bus. Otherwise, it is connected to the ALU result (32 bits).
alu_in_A	First operand for the ALU (32 bits).
alu_in_B	Second operand for the ALU. `alu_in_B` is connected to `imm_ext` during immediate mode instructions. Otherwise, it is connected to `readreg2` (32 bits).
alu_result	Output of ALU (32 bits).
alu_or_mem	Select signal for the `reg_in` multiplexer; indicates whether the register input should come from the memory or the ALU.
reg_or_imm	Select signal for the `alu_in_B` multiplexer; determines whether the second ALU operand is a register output or sign-extended immediate constant.
Regw	Indicates if the destination register should be written to. Some instructions do not write any results to a register (e.g, branch, store).
fetchDorI	Select signal for the `Address` multiplexer; determines whether ADDR is the location of an instruction to be fetched or the location of data to be read or written.
Writing	Control signal for the MIPS processor output to the memory bus. Except during memory writes, the output is 'hi-Z' so the bus can be used by other modules. *Note*: `writing` cannot be replaced with `WE`, because `WE` is of mode `out`. *writing* is used in mode *in* as well.
Dr	Address of destination register (5 bits).
State	Current state.
nState	Next state.

Since we have used separate clock cycles for the fetch operation, the decode operation, the execute operation, and so on, it is necessary to save signals created during each stage for later use. The statements such as

```
opsave <= op;

reg_or_imm_Save <= reg_or_imm;

alu_or_mem_save <= alu_or_mem;

alu_result_save <= alu_result;
```

are used in the clocked process for saving (explicit latching) of the relevant signals.

Two always blocks are used in the code: (1) a clocked process for the sequential part of the code and (2) another for the combinational block.

The multiplexer at the input of the program counter is not explicitly coded. The various data transfers are coded behaviorally in the various states. A good synthesizer will be able to generate the multiplexer to accomplish the various data transfers. Similarly, the multiplexer to select the destination register address is also not explicitly coded. If the synthesis tool generates inefficient hardware for this multiplexed data transfer, one can code the multiplexer into the data path and generate control signals for the select signals.

Complete MIPS

The processor module and the memory are integrated to yield the complete MIPS model. Component descriptions are created for the processor and the memory units. These components are instantiated into the Complete_MIPS module. We have also brought out the address and data buses as outputs from the high-level entity. If no outputs are shown in an entity, when the code is synthesized, it results in empty blocks. Depending on the synthesis tool, unused signals (and corresponding nets) may be deleted from the synthesized circuit. Bringing the address and data buses allows the testing of the modules observing these buses.

FIGURE 9-9: Verilog Code Integrating the Processor and Memory Modules

```
module Complete_MIPS(CLK, RST, A_Out, D_Out);

    input CLK;
    input RST;
    output [31:0] A_Out;
    output [31:0] D_Out;

    wire CS, WE;
    wire [31:0] ADDR, Mem_Bus;
    assign A_Out = ADDR;
    assign D_Out = Mem_Bus;

    MIPS CPU(CLK, RST, CS, WE, ADDR, Mem_Bus);
    Memory MEM(CS, WE, CLK, ADDR, Mem_Bus);

endmodule
```

We synthesized the foregoing model. The Xilinx ISE tools targeted for a Spartan 3 FPGA yield 1108 4-input LUTs, 660 slices, 111 flip-flops, and one block RAM. The register file takes 194 4-input LUTs. Since one LUT can give 16 bits of storage, thirty-two 32-bit registers would need the storage from 64 LUTs. Since the register file has two read ports, it would need 128 LUTs. Additional LUTs are required for the address decoder and the control signals. In order to implement the design on a prototyping board, interface to the input and display modules should be added.

Testing the Processor Model

The overall MIPS Verilog model is tested using a test bench. The test bench must verify the proper operation of each implemented instruction. The test bench consists of a MIPS program with test instructions and Verilog code to load the program into memory and verify the program's output. We use a constant array of instructions that we want to write into the memory and a constant array of expected outputs to which we will compare the result of the processor execution.

The model was tested for the following instructions. The expected action of each instruction is also shown as follows:

```
x"30000000", -- andi $0, $0, 0 => $0 = 0
x"20010006", -- addi $1, $0, 6 => $1 = 6
x"34020012", -- ori $2, $0, 18 => $2 = 18
x"00221820", -- add $3, $1, $2 => $3 = $1 + $2 = 24
x"00412022", -- sub $4, $2, $1 => $4 = $2 - $1 = 12
x"00222824", -- and $5, $1, $2 => $5 = $1 and $2 = 2
x"00223025", -- or $6, $1, $2  => $6 = $1 or $2 = 22
x"0022382A", -- slt $7, $1, $2 => $7 = 1 because $1<$2
x"00024100", -- sll $8, $2, 4  => $8 = 18 * 16 = 288
x"00014842", -- srl $9, $1, 1  => $9 = 6/2 = 3
x"10220001", -- beq $1, $2, 1  => Will not branch. $10 wrong if fails.
x"8C0A0004", -- lw $10, 4($0)  => $10 = 5th instr = x"00412022" = 4268066
x"14620001", -- bne $1, $2, 1  => Branch to PC+1+1 (not PC+4+4 because
                                  word-addressed memory instead of byte-
                                  addressed)
x"30210000", -- andi $1, $1, 0  => $1 = 0 (skipped if bne worked correctly)
x"08000010", -- j 16            => PC = $16.  $2 wrong if j malfunctioned
x"30420000", -- andi $2, $2, 0  => $2 = 0 (skipped if j worked correctly)
x"00400008", -- jr $2           => PC = $2 = 18. $3 wrong if jr malfunction
x"30630000", -- andi $3, $3, 0  => $3 = 0 (skipped if jr worked correctly)
x"AC010040", -- sw $1, 64($0)   => Mem(64) = $1 = 6
x"AC020041", -- sw $2, 65($0)   => Mem(65) = $2 = 18
x"AC030042", -- sw $3, 66($0)   => Mem(66) = $3 = 24
x"AC040043", -- sw $4, 67($0)   => Mem(67) = $4 = 12
x"AC050044", -- sw $5, 68($0)   => Mem(68) = $5 = 2
x"AC060045", -- sw $6, 69($0)   => Mem(69) = $6 = 22
x"AC070046", -- sw $7, 70($0)   => Mem(70) = $7 = 1
x"AC080047"  -- sw $8, 71($0)   => Mem(71) = $8 = 288
x"AC090048", -- sw $9, 72($0)   => Mem(72) = $9 = 3
x"AC0A0049", -- sw $10, 73($0)  => Mem(73) = $10 = x00412022
```

File I/O is used to load these instructions into the memory module. The file "MIPS_Instructions.txt" contains the following hex data:

```
30000000
20010006
34020012
00221820
00412022
00222824
00223025
0022382A
00024100
00014842
10220001
8C0A0004
14620001
30210000
08000010
30420000
00400008
30630000
AC010040
AC020041
AC030042
AC040043
AC050044
AC060045
AC070046
AC080047
AC090048
AC0A0049
```

The memory module in Figure 9-7 must include the following line in order to appropriately initialize the code memory:

```
$readmemh("MIPS_Instructions.txt", RAM);
```

Note that now the memory is connected to the processor and test bench, and that means both our test bench and the processor will try to control the two signals at the same time. One way to resolve this is to put MUXes at the input ports of the memory. There are a few MUXes for that purpose: *Address_Mux* (for choosing the address), *CS_Mux* for choosing the *CS* signal, and *WE_Mux* (for choosing the WE signal). The select signal for the MUXes is *init*. When the *init* signal is 1, the three MUXes select the address and the *CS* and *WE* signals from the test bench. Otherwise, these signals from the processor are chosen. We also assert the reset of our CPU throughout the initialization process to make sure the CPU does not run until the test bench finishes writing the instructions into the memory. When *init* is 0, the CPU and memory are connected for normal operation.

As the MIPS program executes, each test instruction stores its result in a different register. The last 10 instructions perform a series of *sw* operations that store registers 1-10 to memory. Each of these instructions places the contents of a different register onto the bus as it executes. During the memory write stage, the testbench will compare the value of these registers (by looking at the bus value) with the expected output. No explicit checks for branch instructions are in the test sequence; however, if a branch does not execute as expected, an error will be detected because the result register values for the instructions after the branch instructions will be incorrect.

In the actual MIPS processor, register $0 is always 0. We did not implement that in the register file. Hence we clear register $0 using an instruction. The first instruction in the test sequence does that. In normal MIPS processor code, you will not find instructions with register $0 as the destination. Essentially, writes to register $0 are ignored.

FIGURE 9-10: Test Bench for the Processor Model

```
module mips_testbench;
    reg CLK;
    wire CS, WE;

    parameter N = 10;
    reg[31:0] expected[N:1];

    wire[31:0] Address, Address_Mux, Mem_Bus_Wire;
    reg[31:0] AddressTB;
    wire WE_Mux, CS_Mux;
    reg init, rst, WE_TB, CS_TB;

    integer i;

    MIPS CPU(CLK, rst, CS, WE, Address, Mem_Bus_Wire);
    Memory MEM(CS_Mux, WE_Mux, CLK, Address_Mux, Mem_Bus_Wire);

    assign Address_Mux = (init)? AddressTB : Address;
    assign WE_Mux = (init)? WE_TB : WE;
    assign CS_Mux = (init)? CS_TB : CS;

    always
        #10 CLK = ~CLK;

    initial begin
        expected[1] = 32'h00000006; // $1 content=6 decimal
        expected[2] = 32'h00000012; // $2 content=18 decimal
        expected[3] = 32'h00000018; // $3 content=24 decimal
        expected[4] = 32'h0000000C; // $4 content=12 decimal
        expected[5] = 32'h00000002; // $5 content=2
        expected[6] = 32'h00000016; // $6 content=22 decimal
```

```
        expected[7] = 32'h00000001; // $7 content=1
        expected[8] = 32'h00000120; // $8 content=288 decimal
        expected[9] = 32'h00000003; // $9 content=3
        expected[10] = 32'h00412022; // $10 content=5th instr
        CLK = 0;
    end

    always begin
        rst = 1;
        @(posedge CLK); //wait until posedge CLK
        //Initialize the instructions from the testbench
        init <= 1; CS_TB <= 1; WE_TB <= 1;
        @(posedge CLK);
        CS_TB <= 0; WE_TB <= 0; init <= 0;
        @(posedge CLK);
        rst <= 0;
        for(i = 1; i <= N; i = i+1) begin
            @(posedge WE); // When a store word is executed
            @(negedge CLK);
            if (Mem_Bus_Wire != expected[i])
                $display("Output mismatch: got %d, expect %d", Mem_Bus_Wire,
                        expected[i]);
        end
        $display("Testing Finished:");
        $stop;
    end
endmodule
```

The following command file was used to test the Verilog model. The full path length of the signals is mentioned in the add list command so that the simulation happens correctly. All the signals that we are interested in are not available in the topmost entity, which here is the test bench. In such cases, the full path describing the signal (specifically pointing to the component in which the signal is appearing) must be provided for correct simulation.

```
add list -hex sim:/mips_testbench/CPU/instr
add list -unsigned sim:/mips_testbench/CPU/npc
add list -unsigned sim:/mips_testbench/CPU/pc
add list -unsigned sim:/mips_testbench/CPU/state
add list -unsigned sim:/mips_testbench/CPU/alu_ina
add list -unsigned sim:/mips_testbench/CPU/alu_inb
add list -signed sim:/mips_testbench/CPU/alu_result
add list -signed sim:/mips_testbench/CPU/addr
configure list -delta collapse
run 2000
```

The simulation results are illustrated in the following table:

MIPS Instruction	ns	Instr	PC	State	ALU_InA	ALU_InB	ALU_Result	Addr
andi $0, $0, 0	050	30000000	0	0	x	x	0	0
	070	30000000	1	1	0	0	0	X
	090	30000000	1	2	0	0	0	X
	110	30000000	1	3	0	0	0	0
addi $1, $0, 6	130	30000000	1	0	0	0	0	1
	150	20010006	2	1	0	6	0	0
	170	20010006	2	2	0	6	6	0
	190	20010006	2	3	0	6	0	6
ori $2, $0, 18	210	20010006	2	0	0	6	0	2
	230	34020012	3	1	0	18	0	6
	250	34020012	3	2	0	18	18	6
	270	34020012	3	3	0	18	0	18
add $3, $1, $2	290	34020012	3	0	0	18	0	3
	310	00221820	4	1	6	6176	0	18
	330	00221820	4	2	6	18	24	18
	350	00221820	4	3	6	18	0	24
sub $4, $2, $1	370	00221820	4	0	6	18	0	4
	390	00412022	5	1	18	6	0	24
	410	00412022	5	2	18	6	12	24
	430	00412022	5	3	18	6	0	12
and $5, $1, $2	450	00412022	5	0	18	6	0	5
	470	00222824	6	1	6	18	0	12
	490	00222824	6	2	6	18	2	12
	510	00222824	6	3	6	18	0	2
or $6, $1, $2	530	00222824	6	0	6	18	0	6
	550	00223025	7	1	6	18	0	2
	570	00223025	7	2	6	18	22	2
	590	00223025	7	3	6	18	0	22
slt $7, $1, $2	610	00223025	7	0	6	18	0	7
	630	0022382A	8	1	6	18	0	22
	650	0022382A	8	2	6	18	1	22
	670	0022382A	8	3	6	18	0	1
sll $8, $2, 4	690	0022382A	8	0	6	18	0	8
	710	00024100	9	1	0	18	0	1
	730	00024100	9	2	0	18	288	1
	750	00024100	9	3	0	18	0	288
srl $9, $1, 1	770	00024100	9	0	0	18	0	9
	790	00014842	10	1	0	6	0	288
	810	00014842	10	2	0	6	3	288

	830	00014842	10	3	0	6	0	3
beq $1, $2, 1	850	00014842	10	0	0	6	0	10
	870	10220001	11	1	6	18	0	3
	890	10220001	11	2	6	18	-12	3
lw $10, 4($0)	910	10220001	11	0	6	18	0	11
	930	8C0A0004	12	1	0	-	0	-12
	950	8C0A0004	12	2	0	4	4	-12
	970	8C0A0004	12	3	0	4	0	4
	990	8C0A0004	12	4	0	4	0	4
bne $1, $2, 1	1010	8C0A0004	12	0	0	4	0	12
	1030	14620001	13	1	24	1	0	4
	1050	14620001	13	2	24	18	6	4
j 16	1070	14620001	14	0	24	18	0	14
	1090	08000010	15	1	0	0	0	6
jr $2	1110	08000010	16	0	0	0	0	16
	1130	00400008	17	1	18	0	0	6
	1150	00400008	17	2	18	0	0	6
sw $1, 64($0)	1170	00400008	18	0	18	0	0	18
	1190	AC010040	19	1	0	6	0	0
	1210	AC010040	19	2	0	64	64	0
	1230	AC010040	19	3	0	64	0	64

The presented data corresponds to the cycles once the instruction fetch by the processor begins (i.e., the initialization cycles are not shown). Only the first store instruction is shown here. But all store instructions are tested in the test bench. The tenth store instruction finishes by 1950 ns. More comprehensive tests can be devised by loading the data from the stored locations in the memory.

In this chapter, we have presented a popular RISC instruction set, the MIPS. We presented a design for a subset of the MIPS instruction set starting from the instruction set specification. A synthesizable Verilog model for the MIPS subset was presented. Use of a test bench to test the processor model was illustrated. The processor model was at the behavioral level and can be easily extended to include other instructions.

Problems

What does the term ISA mean? Do the Pentium 4 –> IBM POWER7 and Pentium 3 –> IBM POWER8 have the same ISA?

Microprocessor *X* has 30 instructions in its instruction set, and microprocessor *Y* has 45 instructions in its instruction set. You are told that *Y* is a RISC processor. Can you conclusively say that *X* is a RISC processor? Why or why not?

List four important characteristics that make a processor RISC type.

Compare the RISC ISA and the CISC ISA and list the benefits of each ISA. Which ISA will typically yield -> consume higher cycles per instruction (CPI)?

What is the difference between the MIPS `addi` instruction and the `addiu` instruction?

What is the difference between branch instructions and jump instructions?

What is the machine language encoding for the following MIPS instructions? Give the answers in hexadecimal (hex). All offsets are in decimal.

(i) `add $6, $7 , $8`
(ii) `lw $5, 4($6)`
(iii) `addiu $3, $2, -2000`
(iv) `sll $3, $7, 12`
(v) `beq $6, $5, -16`
(vi) `j 4000`

What is the machine language encoding for the following MIPS instructions? Give the answers in hexadecimal (hex). All offsets are in decimal.

(i) `addi $5, $4 ,4000`
(ii) `sw $5, 20($3)`
(iii) `addu $4, $5, $3`
(iv) `bne $2, $3, 32`
(v) `jr $5`
(vi) `jal 8000`

What MIPS instruction do the following hexadecimal (hex) numbers correspond to? If it is not any instruction shown in Table 9-7, denote it as an illegal opcode.

(i) 33333300
(ii) 8D8D8D8D
(iii) 1777FF00
(iv) BDBD00BD
(v) 01010101

What MIPS instruction do the following hexadecimal (hex) numbers correspond to? If it is not any instruction shown in Table 9-7, denote it as an illegal opcode.

(i) 20202020
(ii) 00E70018
(iii) 13D300C8
(iv) 0192282A
(v) 0F6812A4

Write a MIPS assembly language program for the following pseudocode segment:

```
for(i = 0; i < 100; i++)
  x(i) = x(i) * y(i)
```