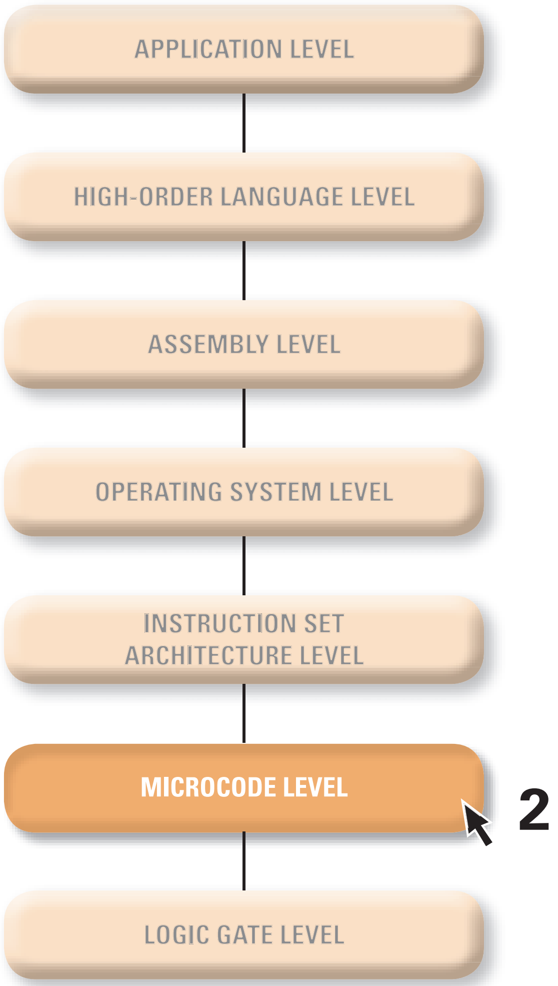
LEVEL
2

Microcode



CHAPTER 12

Computer Organization

TABLE OF CONTENTS

This final chapter shows the connection between the combinational and sequential circuits at Level LG1 and the machine at Level ISA3. It describes how hardware devices can be connected at the Mc2 level of abstraction to form black boxes at successively higher levels of abstraction to eventually construct the Pep/9 computer.

12.1 Constructing a Level-ISA3 Machine

FIGURE 12.1 is a block diagram of the Pep/9 computer. It shows the CPU divided into a data section and a control section. The data section receives data from and sends data to the main memory subsystem and the disk. The control section issues the control signals to the data section and to the other components of the computer.

The CPU Data Section

FIGURE 12.2 is the data section of the Pep/9 CPU. The sequential devices in the figure are shaded to distinguish them from the combinational devices. The CPU registers at the top of the figure are identical to the two-port register bank of Figure 11.52.

The control section, not shown in the figure, is to the right of the data section. The control lines coming in from the right come from the control section. The control lines are rendered as dashed lines in Figure 12.1, but they are solid lines in Figure 12.2. There are two kinds of control signals—combinational circuit controls and clock pulses. Names of the clock pulses all end in Ck and all act to clock data into a register or flip-flop. For example,

FIGURE 12.1
Block diagram of the Pep/9 computer.

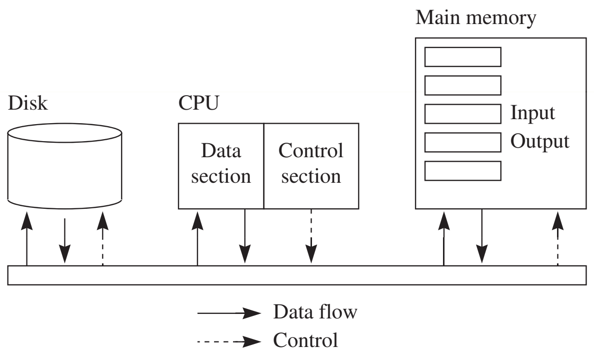
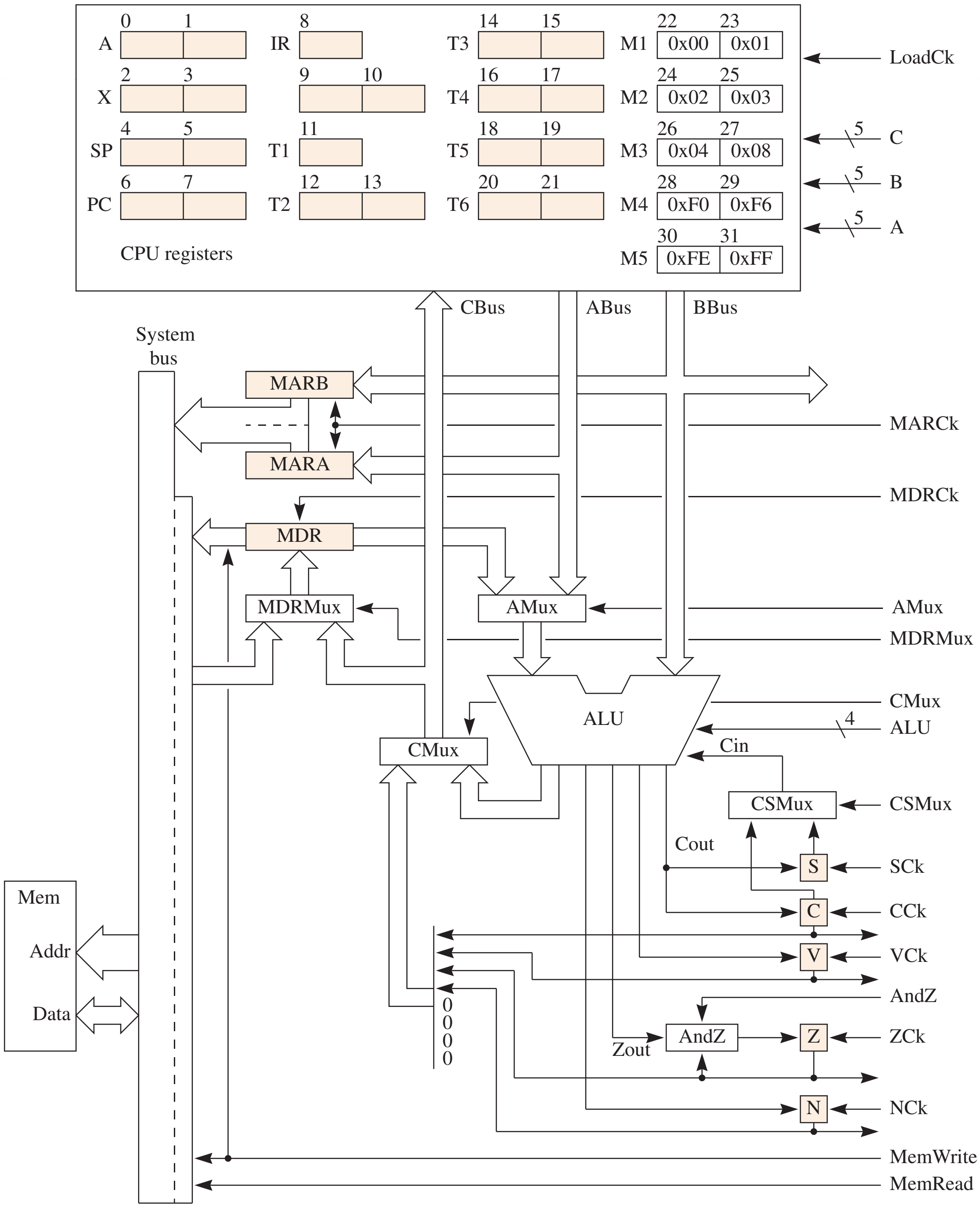


FIGURE 12.2
The data section of the Pep/9 CPU.



MDRCk is the clock input for the MDR. When it is pulsed, the input from the MDRMux is clocked into the MDR.

The main system bus

The system bus on the left of the figure is the main system bus of Figure 12.1, to which main memory and the disk devices are attached. It consists of 8 bidirectional data lines, 16 address lines, and 2 control lines labeled *MemWrite* and *MemRead* at the bottom of the figure. MAR is the memory address register, divided into MARA, the high-order byte, and MARB, the low-order byte. The box labeled *Mem* is a 64-KiB memory subsystem. The 16 address lines on the bus are unidirectional, so that the output of the MAR connects to the input of the address port of the memory subsystem over the bus. MDR is the eight-bit memory data register. Because the data bus is bidirectional, there is a set of eight tri-state buffers (not shown in the figure) between MDR and the bus that are enabled by the MemWrite control line. The MemWrite line connects to the Write Enable (WE) line of the memory subsystem over the main system bus. The MemRead line connects to the Output Enable (OE) line. All the other buses in Figure 12.2 represented by the wide arrows are eight-bit unidirectional data buses, including, for example, ABus, BBus, CBus, and the bus connecting the data lines of the main system bus to the box labeled *MDRMux*.

The memory address register, MAR

The memory data register, MDR

The multiplexers—AMux, CMux, and MDRMux

Each multiplexer—AMux, CMux, and MDRMux—is a bank of eight two-input multiplexers with their control lines connected together to form the single control line in Figure 12.2. For example, the control line labeled *AMux* in the figure is connected to each of the eight control lines in the bank of eight multiplexers in the block labeled *AMux*. A multiplexer control line routes the signal through a multiplexer as follows:

Multiplexer control signals

- 0 on a multiplexer control line routes the left input to the output.
- 1 on a multiplexer control line routes the right input to the output.

For example, if the MDRMux control line is 0, MDRMux routes the content of the system bus to the MDR. If the control line is 1, it routes the data from the CBus to the MDR. Similarly, if the AMux control line is 0, AMux routes the content of MDR to the left input of the ALU. Otherwise, it routes the data from the ABus to the left input of the ALU.

The CSMux is different from the other multiplexers because it switches only one line instead of eight lines. If the CSMux control line is 0, the multiplexer sends the S bit to the Cin of the ALU. If the CSMux control line is 1, the multiplexer sends the C bit to the Cin of the ALU.

The block labeled *ALU* is the arithmetic logic unit of Figure 10.54. It provides the 16 functions listed in Figure 10.55 via the four control lines labeled ALU in Figure 12.2. The status bits—N, Z, V, C, and S—are each one D flip-flop. For example, the box labeled *C* is a D flip-flop that stores the

The status bits—N, Z, V, C, and S

value of the carry bit. The D input to the flip-flop is the Cout signal from the ALU. The Q output of the flip-flop is at the top, into the left input of CSMux. The Q output is also at the bottom of the box labeled *C*. The clock input of the flip-flop is the control signal labeled *CCk*. The outputs of each of the status bits feed into the low-order nybble of the left bus into the CMux. The high-order nybble is hardwired to four zeros. The outputs of each of the status bits are also sent to the control section.

The shadow carry bit

The box labeled *S* is the shadow carry bit. The C bit is visible to the programmer as the carry bit at Level ISA3. The arithmetic instructions set the C bit, and the conditional branch instructions test it. The shadow carry bit S, however, is hidden from the programmer at the ISA3 abstraction level. When the ALU performs an arithmetic operation, it can take its Cin input from either the C bit or the S bit through CSMux. The system can also clock the Cout output from the ALU into either the C bit or the S bit. The question of which bit to use depends on whether the computation at Level Mc2 should alter the carry bit for the programmer at Level ISA3. If the computation is performing an arithmetic operation like `ADDA` for which the carry bit should be set, Cout from the ALU would be clocked into the C carry bit. If the computation is performing an internal operation like incrementing the program counter, Cout from the ALU would be clocked into the S shadow carry bit.

At the ISA level, each register is 16 bits, but the internal data paths of the CPU are only 8 bits wide. To perform an operation on one 16-bit quantity requires two operations on 8-bit quantities at the Mc2 level. For example, the Z bit must be set to 1 if the 16 bits of the result are all zeros, which happens when the Zout signal from the ALU is 1 for both 8-bit operations. The combinational box labeled *AndZ* facilitates the computation of the Z bit. Its output is connected to the input of the D flip-flop for the Z bit. It has three inputs—the AndZ input from the control section, the Zout output from the ALU, and the Q output from the D flip-flop for the Z bit. FIGURE 12.3 shows the truth table for the box. It operates in one of the following two modes:

Operation of the AndZ circuit in Figure 12.2

- If the AndZ control signal is 0, Zout passes directly through to the output.
- If the AndZ control signal is 1, Zout AND Z passes to the output.

The Z bit is, therefore, loaded with either the Zout signal from the ALU or the Zout signal ANDed with the current value of the Z bit. Which one depends on the AndZ signal from the control section. Implementation of the AndZ circuit is an exercise at the end of the chapter.

The data flow is one big loop, starting with the 32 eight-bit registers at the top and proceeding via ABus and BBus through AMux to the ALU,

FIGURE 12.3
The truth table for the AndZ combinational circuit in Figure 12.2.

Input			
AndZ	Z	Zout	Output
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

through CMux, and finally back to the bank of 32 registers via CBus. Data from main memory can be injected into the loop from the system bus, through the MDRMux to the MDR. From there, it can go through the AMux, the ALU, and the CMux to any of the CPU registers. To send the content of a CPU register to memory, you can pass it through the ALU via the ABus and AMux, through the CMux and MDRMux into the MDR. From there it can go to the memory subsystem over the system bus.

The control section has 34 control output lines and 12 input lines. The 34 output lines control the flow of data around the data section loop and specify the processing that is to occur along the way. The 12 input lines come from the 8 lines of BBus plus 4 lines from the status bits, which the control section can test for certain conditions. Later in this chapter is a description of how the control section generates the proper control signals. In the following description of the data section, assume for now that you can set the control lines to any desired values for any cycle.

The von Neumann Cycle

The heart of the Pep/9 computer is the von Neumann cycle. The data section in Figure 12.2 implements the von Neumann cycle. It really is nothing more than plumbing. In the same way that water in your house runs through the pipes controlled by various faucets and valves, signals (electrons, literally) flow through the wires of the buses controlled by various multiplexers. Along the way, the signals can flow through the ALU, where they can be processed as required. This section shows the control signals necessary to implement

FIGURE 12.4
A pseudocode description at Level Mc2 of the von Neumann execution cycle.

```
do {
    Fetch the instruction specifier at address in PC
    PC ← PC + 1
    Decode the instruction specifier
    if (the instruction is not unary) {
        Fetch the high-order byte of the operand specifier as specified by PC
        PC ← PC + 1
        Fetch the low-order byte of the operand specifier as specified by PC
        PC ← PC + 1
    }
    Execute the instruction fetched
}
while ((the stop instruction does not execute) && (the instruction is legal))
```

the von Neumann cycle. It includes the implementation of some typical instructions in the Pep/9 instruction set and leaves the implementation of others as problems at the end of the chapter.

Figure 4.32 shows the pseudocode description of the steps necessary to execute a program at Level ISA3. The do loop is the von Neumann cycle. At Level Mc2, the data section of the CPU operates on 8-bit quantities, even though the operand specifier part of the instruction register is a 16-bit quantity. The CPU fetches the operand specifier in two steps: the high-order byte followed by the low-order byte. The control section increments PC by 1 after fetching each byte. FIGURE 12.4 is a pseudocode description at Level Mc2 of the von Neumann execution cycle.

The control section sends control signals to the data section to implement the von Neumann cycle. FIGURE 12.5 is the control sequence to fetch the instruction specifier and to increment PC by 1. The figure does not show the method by which the control section determines whether the instruction is unary.

Each numbered line in Figure 12.5 is a CPU clock cycle and consists of a set of control signals that are input into the combinational devices, usually followed by a clock pulse into one or more registers. The combinational signals, denoted by the equals sign, must be set up for a long enough period of time to let the data reach the register before being clocked into the register. The combinational signals are applied concurrently and are

FIGURE 12.5
The control signals to fetch the instruction specifier and increment PC by 1.

```
// Fetch the instruction specifier and increment PC by 1

UnitPre: IR=0x000000, PC=0x00FF, Mem[0x00FF]=0xAB, S=0
UnitPost: IR=0xAB0000, PC=0x0100

// MAR <- PC.
1. A=6, B=7; MARCk
// Fetch instruction specifier.
2. MemRead
3. MemRead
4. MemRead, MDRMux=0; MDRCk
// IR <- instruction specifier.
5. AMux=0, ALU=0, CMux=1, C=8; LoadCk

// PC <- PC plus 1, low-order byte first.
6. A=7, B=23, AMux=1, ALU=1, CMux=1, C=7; SCk, LoadCk
7. A=6, B=22, AMux=1, CSMux=1, ALU=2, CMux=1, C=6; LoadCk
```

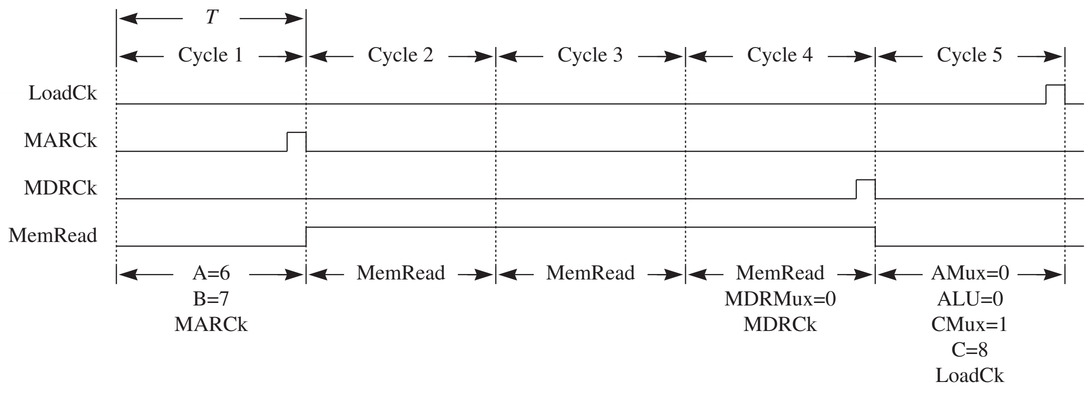
therefore separated from each other by a comma, which is the concurrent separator. The combinational signals are separated from the clock signals by a semicolon, which is the sequential separator, because the clock pulses are applied after the combinational signals have been set. Comments are denoted by double forward slashes (//).

FIGURE 12.6 shows the clock cycles corresponding to the lines numbered 1 through 5 in Figure 12.5. The period of a cycle T in seconds is specified by the frequency f of the system clock in Hz according to $T = 1/f$. The greater the frequency of your computer, as measured by its GHz rating, the shorter the period of one cycle and the faster your computer will execute, all other things being equal. So, what limits the speed of the CPU? The period must be long enough to allow the signals to flow through the combinational circuits and be presented to the inputs of the registers (which are the sequential circuits) before the next clock pulse arrives.

Transferring the PC to the MAR

For example, at the beginning of cycle 1 of Figure 12.6, `A=6` sets the five A lines to 6 (dec), which is 00110 (bin), and `B=7` sets the five B lines to 7 (dec), which is 00111 (bin). Figure 12.2 shows that `A=6` and `B=7` access the high-order byte and the low-order byte of the PC. It takes time for the A and B signals to propagate through the combinational addressing circuits in the

FIGURE 12.6
Timing diagram of the first five cycles of Figure 12.5.



register bank and for the content of the PC to be placed on ABus and BBus. The period *T* is long enough to allow those signals to be set up and present at the inputs of MARA and MARB before the MARCk clock pulse occurs at the end of cycle 1 shown in Figure 12.6. After the clock pulse at the end of cycle 1, the content of PC is in MARA and MARB.

Transferring a byte from memory to the MDR

At the beginning of cycle 2, the address signals from MARA and MARB begin to propagate over the system bus to the memory subsystem. Now the `MemRead` signal activates the OE line on the chip in the memory subsystem that is selected by the address decoding circuitry. There are so many propagation delays in the memory subsystem that it usually takes many CPU cycles before the data is available on the main system bus. The Pep/9 computer models this fact by requiring `MemRead` on three consecutive cycles after the address is clocked into the MAR. So the `MemRead` signal is asserted high in cycles 2, 3, and 4.

At the end of cycle 4, because the same address was present in the memory address register (MAR) for three consecutive cycles and MemRead was asserted during those three cycles, the data from that address in memory is present on the system bus. In cycle 4, `MDRMux=0` sets the MDR multiplexer line to 0, which routes the data from the system bus through the multiplexer to the input of the memory data register (MDR). The `MDRCk` pulse on cycle 4 clocks the instruction specifier into the MDR.

Sending data from the MDR to the register bank

Cycle 5 sends the instruction specifier from the MDR into the instruction register, IR, as follows. First, `AMux=0` sets the AMux control line to 0, which routes the MDR to the output of the AMux multiplexer. Next, `ALU=0` sets the ALU control line to 0, which passes the data through the

ALU unchanged, as specified in Figure 10.55. `CMux=1` routes the data from the ALU to the CBus. Then, `C=8` sets C to 8, which specifies the instruction register, as Figure 12.2 shows. Finally, `LoadCk` clocks the content of the MDR into the instruction register.

In cycle 5, Figure 12.6 shows the LoadCk clock pulse at the end of the cycle. The clock period *T* must be long enough to allow the content of the MDR to propagate through the multiplexer and the ALU before the LoadCk. The control section designer must count the number of gate delays through those combinational circuits to determine the minimum time that must elapse before the data is clocked into the instruction register. The period *T* is set long enough to accommodate those gate delays.

Incrementing PC by 1

Cycles 6–7 increment PC by 1. On cycle 6, `A=7` puts the low-order byte of PC on ABus, and `B=23` puts constant 1 on BBus. `AMux=1` selects the ABus to pass through the multiplexer. `ALU=1` selects the A plus B function of the arithmetic logic unit, so the ALU adds 1 to the low-order byte of PC. `CMux=1` routes the sum to the CBus, and `C=7` puts the sum back into the low-order byte of PC. In the same cycle, `SCk` saves the carry out of the addition in the shadow carry S bit. The period *T* is long enough for the data to flow through the combinational devices specified by the other control signals in cycle 6 before `SCk` stores Cout from the ALU into S and `LoadCk` stores the result of the ALU into the low-order byte of PC.

If the original low-order byte of PC is 1111 1111 (bin), then adding 1 will cause a carry to the high-order byte. On cycle 7, `A=6` puts the high-order byte of PC onto the ABus, and `B=22` puts constant 0 onto the BBus. `AMux=1` routes ABus through the multiplexer to the ALU, and `CSMux=1` routes the saved shadow carry S bit to Cin of the ALU. `ALU=2` selects the A plus B plus Cin function for the ALU, adding the saved carry from the low-order byte to the high-order byte of PC. `CMux=1` routes the result to the CBus, and `C=6` directs the data on the CBus to be loaded into the high-order byte of PC, which is stored with the `LoadCk` pulse.

The Pep/9 CPU software available with this text allows you to write control sequences like Figure 12.5 and simulate their execution on the Pep/9 data section. The software provides unit tests for the correctness of the sequences specified by UnitPre and UnitPost statements. A UnitPre statement can set the value of a memory location, a register bank location, or a status bit to any arbitrary value before the control sequences execute. A UnitPost statement tests the value of a memory location, a register bank location, or a status bit after the control sequences execute.

Microcode unit tests

For example, in Figure 12.5 the UnitPre statement sets the IR to all zeros, the PC to 00FF (hex), Mem[00FF] to AB (hex) (the instruction specifier for `CPWX this,s`), and the shadow bit S to 0. With these initial conditions, fetching the instruction specifier and incrementing the program counter should put AB (hex) in the instruction register and should increment the program counter to 0100 (hex). Because the UnitPost statement specifies these values, the software automatically tests them at the conclusion of the control sequence simulation and displays a message about whether these postconditions are satisfied.

Combining cycles

It is possible to reduce the number of cycles in Figure 12.5 by combining cycles. The control sequence in FIGURE 12.7 does the same processing as the control sequence in Figure 12.5 but with only five cycles instead of seven. Cycle 1 puts a copy of PC in the MAR, and that original copy stays in the MAR during cycles 2, 3, and 4. Consequently, the value of PC can be incremented during cycles 2 and 3 without disturbing its original value in the MAR. The control sequence in Figure 12.7 combines cycle 6 with cycle 2 from Figure 12.5. During cycle 2, the ABus, BBus, and CBus are not used in Figure 12.5. Thus, the low-order byte can be incremented by 1 in this cycle concurrently with the system waiting for the memory read. Similarly, cycle 7 is combined with cycle 3. Reducing the number of cycles from 7 to 5 is a savings in time of 2/7 = 29%.

You cannot arbitrarily combine cycles in a control sequence. You must remember that a numbered line in a control sequence like Figure 12.5 represents one CPU cycle. Some cycles depend on the results from previous cycles. For example, you cannot combine cycles 4 and 5 in Figure 12.5, because cycle 5 depends on the results from cycle 4. Cycle 4 sets the content

FIGURE 12.7
Combining cycles of Figure 12.5.

```
// Fetch the instruction specifier and increment PC by 1

UnitPre: IR=0x000000, PC=0x00FF, Mem[0x00FF]=0xAB, S=0
UnitPost: IR=0xAB0000, PC=0x0100

// MAR <- PC.
1. A=6, B=7; MARCk
// Fetch instruction specifier, PC <- PC + 1.
2. MemRead, A=7, B=23, AMux=1, ALU=1, CMux=1, C=7; SCk, LoadCk
3. MemRead, A=6, B=22, AMux=1, CSMux=1, ALU=2, CMux=1, C=6; LoadCk
4. MemRead, MDRMux=0; MDRCk
// IR <- instruction specifier.
5. AMux=0, ALU=0, CMux=1, C=8; LoadCk
```

of the MDR, and cycle 5 uses the content of the MDR. Therefore, cycle 5 must happen after cycle 4.

Hardware concurrency is an important issue in computer organization. Designers are always on the alert to use hardware concurrency to improve performance. The seven-cycle sequence of Figure 12.5 would certainly not be used in a real machine, because combining cycles in Figure 12.7 gives a performance boost with no increase in circuitry.

Although the details of the control section are not shown, you can imagine how it would test the instruction just fetched to determine whether it is unary. The control section would set B to 8 to put the instruction specifier on BBus, which it could then test. If the fetched instruction is not unary, the control section must fetch the operand specifier, incrementing PC accordingly. The control sequence to fetch the operand specifier and increment PC is a problem at the end of this chapter.

After fetching an instruction, the control section tests the instruction specifier to determine which of the Pep/9 ISA3 instructions to execute. The control signals to execute the instruction depend not only on the opcode, but on the register-r field and the addressing-aaa field also. FIGURE 12.8 shows the relationship between the operand and the operand specifier (OprndSpec) for each addressing mode.

A quantity in square brackets is a memory address. To execute the instruction, the control section must provide control signals to the data

FIGURE 12.8
The addressing modes for the Pep/9 computer.

Addressing mode	Operand
Immediate	OprndSpec
Direct	Mem[OprndSpec]
Indirect	Mem[Mem[OprndSpec]]
Stack-relative	Mem[SP + OprndSpec]
Stack-relative deferred	Mem[Mem[SP + OprndSpec]]
Indexed	Mem[OprndSpec + X]
Stack-indexed	Mem[SP + OprndSpec + X]
Stack-deferred indexed	Mem[Mem[SP + OprndSpec] + X]

section to compute the memory address. For example, to execute an instruction that uses the indexed addressing mode, the control section must perform a 16-bit addition of the content of the operand specifier (registers 9 and 10) and X (registers 2 and 3). The result of this addition is then loaded into MAR in preparation for a memory read in case of an `LDWr` instruction, or memory write in case of an `STWr` instruction.

The control sequence to implement the first part of the von Neumann execution cycle in Figure 12.5 looks suspiciously like a program in some low-level programming language. The sequence is the microcode language at Level Mc2. The job of control section designers is to devise circuits that, in effect, program the data section to implement the instructions at Level ISA3, the instruction set architecture level.

The next few examples show the Mc2 control sequences necessary to execute some representative ISA3 instructions. Each example assumes that the instruction has been fetched and PC incremented accordingly. Each statement in the program is written on a separate, numbered line and consists of a set of combinational signals to route data through a multiplexer or to select a function for the ALU, possibly followed by one or several clock pulses to load some registers. Keep in mind that a program at this level of abstraction (Level Mc2) consists of the control signals necessary to implement just one instruction at the higher level of abstraction (Level ISA3).

The Store Byte Direct Instruction

FIGURE 12.9 shows the control sequence to execute the instruction

```
STBA there,d
```

where `there` is a symbol. The RTL specification for the `STBr` instruction is

byte Oprnd $\leftarrow$ r$\langle 8..15 \rangle$

Because the instruction specifies direct addressing, the operand is Mem[OprndSpec]. That is, the operand specifier is the address in memory of the operand. The instruction stores the least significant byte of the accumulator into the memory cell at that address. The status bits are not affected.

This example, as well as those that follow, assumes that the operand specifier is already in the instruction register. That is, it assumes that the fetch, decode, and increment parts of the von Neumann execution cycle have already transpired. The programs show only the execute part of the von Neumann cycle. The unit test shows that if the least significant byte of the

FIGURE 12.9
The control signals to implement the store byte instruction with direct addressing.

```
// STBA there,d
// RTL: byteOprnd <- A<8..15>
// Direct addressing: Oprnd = Mem[OprndSpec]

UnitPre: IR=0xF1000F, A=0x00AB
UnitPost: Mem[0x000F]=0xAB

// MAR <- OprndSpec.
1. A=9, B=10; MARCk
// Initiate write, MBR <- A<low>.
2. MemWrite, A=1, AMux=1, ALU=0, CMux=1, MDRMux=1; MDRCk
3. MemWrite
4. MemWrite
```

accumulator has AB (hex) and the operand specifier has 000F (hex), then after the statement executes, Mem[000F] must have AB (hex).

Transferring OprndSpec to MAR

Cycle 1 transfers the operand specifier into the memory address register. `A=9` puts the high-order byte of the operand specifier on the ABus, `B=10` puts the low-order byte of the operand specifier on the BBus, and `MARCk` clocks the ABus and BBus into the MAR registers.

Transferring A<low> to MAR

Cycle 2 transfers the low-order byte of the accumulator into the MDR. `A=1` puts the low-order byte of the accumulator onto the ABus, `AMux=1` routes it through the AMux into the ALU, `ALU=0` passes it through the ALU unchanged, `CMux=1` routes it onto the CBus, `MDRMux=1` routes it through MDRMux to the MDR, and `MDRCk` latches the data into the MDR. This cycle also initiates the memory write.

Completing memory write

Cycles 3 and 4 complete the memory write, storing the data that is in the MDR to main memory at the address that is in the MAR. As with memory reads, memory writes require three consecutive cycles of the MemWrite line to give the memory subsystem time to get the address from the bus for routing the data. The data to be transferred needs to be in the MDR only during the last memory write cycle. The store instructions do not affect the status bits at the ISA level. Consequently, none of the cycles in the control sequence for `STBA` pulse NCk, ZCk, VCk, or CCk.

Bus Protocols

To fetch the instruction specifier requires a read from memory over the system bus, and to store a byte requires a write to memory over the system bus. In practice, every bus in a computer system has timing specifications that other components of the system must follow to transfer information over the bus.

The bus protocol for a memory read over the Pep/9 system bus requires three consecutive cycles, with MemRead asserted on each cycle. The read operation must adhere to the following specification:

The memory read bus protocol

- You must clock the address into the MAR before the first MemRead cycle.
- You must clock the data into the MDR from the system bus on or before the third MemRead cycle.
- On the third MemRead cycle, you cannot clock a new value into MAR in anticipation of a following memory operation.

You might be tempted to clock in a new address on the third MemRead cycle, reasoning that the address gets clocked in at the *end* of the third cycle and so would be present on the system bus during all of the first two cycles and most of the third cycle. However, the system bus protocol does not allow you to do so.

The bus protocol for a memory write requires three consecutive cycles, with MemWrite asserted on each cycle. The write operation must adhere to the following specification:

The memory write bus protocol

- You must clock the address into the MAR before the first MemWrite cycle.
- On the first or second MemWrite cycle, you can clock the data to be written into the MDR.
- On the third MemWrite cycle, you can clock a new data value into the MDR in anticipation of a following memory write. However, you cannot clock a new address value into the MAR in anticipation of a following memory operation.

Figure 12.9 shows the data value to be written to memory clocked into the MDR during cycle 2. It could just as easily be clocked into the MDR during cycle 3, which is the second MemWrite cycle. It is possible to first clock the data into the MDR, then to clock the address into the MAR, then to initiate the memory write. But that would require an extra cycle. It is more efficient to combine the clocking of the data into the MBR concurrently with one of the MemWrite cycles.

The Store Word Direct Instruction

FIGURE 12.10 shows the control sequence to execute the instruction

```
STWA there,d
```

where `there` is a symbol. The RTL specification for the `STWr` instruction is

Oprnd ← r

The difference between `STWA` and `STBA` is that `STWA` stores two bytes instead of one. Because the bytes are stored in consecutive addresses and the operand specifier is the address of where the first byte will be stored, the microcode adds 1 to that address to get the address of where to store the second byte.

FIGURE 12.10
The control signals to implement the store word instruction with direct addressing.

```
// STWA there,d
// RTL: Oprnd <- A
// Direct addressing: Oprnd = Mem[OprndSpec]

UnitPre: IR=0xE100FF, A=0xABCD, S=0
UnitPost: Mem[0x00FF]=0xABCD

// UnitPre: IR=0xE101FE, A=0xABCD, S=1
// UnitPost: Mem[0x01FE]=0xABCD

// MAR <- OprndSpec.
1. A=9, B=10; MARCk
// Initiate write, MDR <- A<high>.
2. MemWrite, A=0, AMux=1, ALU=0, CMux=1, MDRMux=1; MDRCk
// Continue write, T2 <- OprndSpec + 1.
3. MemWrite, A=10, B=23, AMux=1, ALU=1, CMux=1, C=13; SCk, LoadCk
4. MemWrite, A=9, B=22, AMux=1, CSMux=1, ALU=2, CMux=1, C=12; LoadCk

// MAR <- T2.
5. A=12, B=13; MARCk
// Initiate write, MDR <- A<low>.
6. MemWrite, A=1, AMux=1, ALU=0, CMux=1, MDRMux=1; MDRCk
7. MemWrite
8. MemWrite
```

Transferring OprndSpec to MAR and adding 1 to OprndSpec

Cycles 1 through 4 in Figure 12.10 are identical to cycles 1 through 4 in Figure 12.9, with two exceptions. First, the microcode stores the high-order (leftmost) byte of the accumulator instead of the low-order byte. Second, in parallel with the memory writes in cycles 3 and 4, it adds 1 to the operand specifier and stores it in temporary register T2.

Transferring OprndSpec to MAR and writing to memory

Cycles 5 through 8 store the low-order byte of the accumulator to memory at the address computed in register T2. Cycle 5 puts the content of T2 into the MAR. Cycle 6 puts the low-order byte of the accumulator in the MDR and initiates the memory write. Cycles 7 and 8 complete the memory write. The two unit tests check for the possibility of an internal carry on the address computation. In the first test, the internal carry is 1, and in the second it is 0.

The Add Immediate Instruction

FIGURE 12.11 shows the control sequence to implement

```
ADDA this,i
```

The RTL specification for `ADDr` is

$$r \leftarrow r + \text{Oprnd}\ ;\ N \leftarrow r < 0\ ,\ Z \leftarrow r = 0\ ,\ V \leftarrow \{\mathit{overflow}\}\ ,\ C \leftarrow \{\mathit{carry}\}$$

The instruction adds the operand to register r and puts the sum in register r, in this case the accumulator. Because the instruction uses immediate addressing, the operand is the operand specifier. As usual, this example assumes that the instruction specifier has already been fetched and is in the instruction register.

The instruction affects all four of the status bits. However, the data section of the Pep/9 CPU can operate only on 8-bit quantities, even though the accumulator holds a 16-bit value. To do the addition, the control sequence must add the low-order bytes first and save the shadow carry from the low-order addition to compute the sum of the high-order bytes. It sets N to 1 if the two-byte quantity is negative when interpreted as a signed integer; otherwise, it clears N to 0. The sign bit of the most significant byte determines the value of N. It sets Z to 1 if the two-byte quantity is all zeros; otherwise, it clears Z to 0. So, unlike the N bit, the values of both the high-order and the low-order bytes determine the value of Z.

Adding low-order byte

Cycle 1 adds the low-order byte of the accumulator to the low-order byte of the operand specifier. `A=1` puts the low-order byte of the accumulator on the ABus, and `B=10` puts the low-order byte of the operand specifier on the BBus. `AMux=1` routes the ABus through the multiplexer, `ALU=1` selects the A plus B function of the ALU, `CMux=1` routes the sum to the CBus, `C=1` directs the output of the ALU to be stored in the low-order byte of the accumulator,

FIGURE 12.11
The control signals to implement the add instruction with immediate addressing.

```
// ADDA this,i
// RTL: A <- A + Oprnd; N <- A<0, Z <- A=0, V <- {overflow}, C <- {carry}
// Immediate addressing: Oprnd = OprndSpec

UnitPre: IR=0x700FF0, A=0x0F11, N=1, Z=1, V=1, C=1, S=0
UnitPost: A=0x1F01, N=0, Z=0, V=0, C=0

// UnitPre: IR=0x707FF0, A=0x0F11, N=0, Z=1, V=0, C=1, S=0
// UnitPost: A=0x8F01, N=1, Z=0, V=1, C=0

// UnitPre: IR=0x70FF00, A=0xFFAB, N=0, Z=1, V=1, C=0, S=1
// UnitPost: A=0xFEAB, N=1, Z=0, V=0, C=1

// UnitPre: IR=0x70FF00, A=0x0100, N=1, Z=0, V=1, C=0, S=1
// UnitPost: A=0x0000, N=0, Z=1, V=0, C=1

// A<low> <- A<low> + Oprnd<low>, Save shadow carry.
1. A=1, B=10, AMux=1, ALU=1, AndZ=0, CMux=1, C=1; ZCk, SCk, LoadCk
// A<high> <- A<high> plus Oprnd<high> plus saved carry.
2. A=0, B=9, AMux=1, CSMux=1, ALU=2, AndZ=1, CMux=1, C=0; NCk, ZCk, VCk, CCk, LoadCk
```

and `LoadCk` clocks it in. In the same cycle, `AndZ=0` sends Zout through to the output of the AndZ combinational circuit, which is presented as input to the Z one-bit register (a D flip-flop). `ZCk` latches the bit into the Z bit, while `SCk` latches the carry out into the shadow carry S bit.

Adding high-order byte

Cycle 2 adds the high-order byte of the accumulator to the high-order byte of the operand specifier. `A=0` puts the high-order byte of the accumulator on the ABus, and `B=9` puts the high-order byte of the operand specifier on the BBus. `CMux=1` routes the shadow carry bit S to Cin of the ALU, `AMux=1` routes the ABus through the multiplexer, `ALU=2` selects the A plus B plus Cin function of the ALU, `CMux=1` routes the sum to the CBus, `C=0` directs it to be stored in the high-order byte of the accumulator, and `LoadCk` clocks it in. `AndZ=1` sends Zout AND Z through to the output of the AndZ combinational circuit, which is presented as input to the Z bit. `ZCk` latches the value into the status bit. A 1 will be latched into the Z bit if and only if both Zout and Z are 1. The value of Z was saved with ZCk from cycle 1, and so it contains 1 if and only if the low-order sum was all zeros. Consequently, the final value of Z is 1 if and only if all 16 bits of the sum are zeros. The other three status bits—N, V, and C—reflect the status of the high-order addition. They are saved with NCk, VCk, and CCk on cycle 2.

Figure 12.11 shows four unit tests for this ISA3 instruction implementation. Each unit test results in a different final set of values for status bits NZVC and is activated by uncommenting the test. Because the `ADDA` instruction affects all four status bits, their initial values are set to the opposite values they should have in the final state. For example, in the second unit test, the final values of the status bits should be NZVC = 1010. Therefore, the initial values for that unit test are NZVC = 0101.

The Load Word Indirect Instruction

FIGURE 12.12 shows the control sequence for

```
LDWX this,n
```

The RTL specification for `LDWr` is

$$r \leftarrow \text{Oprnd}\ ;\ N \leftarrow r < 0\ ,\ Z \leftarrow r = 0$$

This instruction loads two bytes from main memory into the index register. Because the instruction uses indirect addressing, the operand is Mem[Mem[OprndSpec]], as Figure 12.8 shows. The operand specifier is the address of the address of the operand. The control sequence must fetch a word from memory, which it uses as the address of the operand, requiring yet another fetch to get the operand.

FIGURE 12.13 shows the effect of executing the control sequence of Figure 12.12, assuming that symbol `this` has the value 0012 (hex) and the initial values in memory are the ones shown at addresses 0012 and 26D1. Mem[0012] contains 26D1 (hex), which is the address of the operand. Mem[26D1] contains the operand 53AC (hex), which the instruction must load into the index register. The figure shows the register addresses for each register affected by the control sequence. The first byte of the instruction register, CA, is the instruction specifier for the `LDWX` instruction using indirect addressing.

Transferring Mem[OprndSpec] to T3<high> and adding 1 to OprndSpec

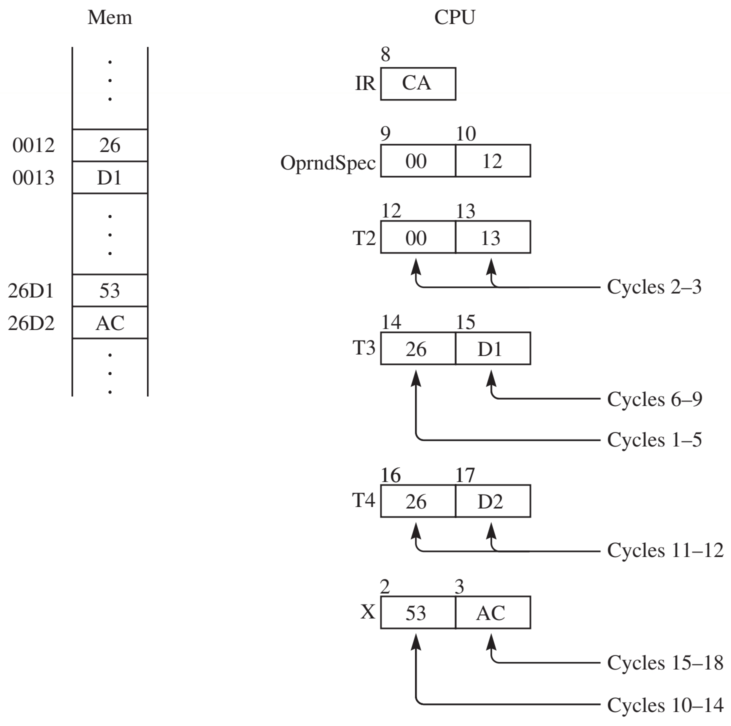
Cycles 1–5 transfer Mem[OprndSpec] to the high-order byte of temporary register T3. On cycle 1, `A=9` and `B=10` put the operand specifier on the ABus and BBus, and `MARCk` clocks them in to the memory address register. Cycle 2 initiates a memory read, cycle 3 continues it, and cycle 4 completes it. Cycle 5 routes the data from the MDR into the high-order byte of T3 in the usual way.

Cycles 2–3 add 1 to the operand specifier and store the result in temporary register T2 concurrently with the above operation during the memory read cycles. Cycle 2 adds 1 to the low-order byte of the operand

FIGURE 12.12
The control signals to implement the load word instruction with indirect addressing.

```
// LDWX this,n
// RTL: X <- Oprnd; N <- X<0, Z <- X=0
// Indirect addressing: Oprnd = Mem[Mem[OprndSpec]]

UnitPre: IR=0xCA0012, Mem[0x0012]=0x26D1, Mem[0x26D1]=0x53AC
UnitPre: N=1, Z=1, V=0, C=1, S=1
UnitPost: X=0x53AC, N=0, Z=0, V=0, C=1

// UnitPre: IR=0xCA0012, X=0xEEEE, Mem[0x0012]=0x00FF, Mem[0x00FF]=0x0000
// UnitPre: N=1, Z=0, V=1, C=0, S=1
// UnitPost: X=0x0000, N=0, Z=1, V=1, C=0

// T3<high> <- Mem[OprndSpec], T2 <- OprndSpec + 1.
1. A=9, B=10; MARCk
2. MemRead, A=10, B=23, AMux=1, ALU=1, CMux=1, C=13; SCk, LoadCk
3. MemRead, A=9, B=22, AMux=1, CSMux=1, ALU=2, CMux=1, C=12; LoadCk
4. MemRead, MDRMux=0; MDRCk
5. A=12, B=13, AMux=0, ALU=0, CMux=1, C=14; MARCk, LoadCk

// T3<low> <- Mem[T2].
6. MemRead
7. MemRead
8. MemRead, MDRMux=0; MDRCk
9. AMux=0, ALU=0, CMux=1, C=15; LoadCk

// Assert: T3 contains the address of the operand.
// X<high> <- Mem[T3], T4 <- T3 + 1.
10. A=14, B=15; MARCk
11. MemRead, A=15, B=23, AMux=1, ALU=1, CMux=1, C=17; SCk, LoadCk
12. MemRead, A=14, B=22, AMux=1, CSMux=1, ALU=2, CMux=1, C=16; LoadCk
13. MemRead, MDRMux=0; MDRCk
14. A=16, B=17, AMux=0, ALU=0, AndZ=0, CMux=1, C=2; NCk, ZCk, MARCk, LoadCk

// X<low> <- Mem[T4].
15. MemRead
16. MemRead
17. MemRead, MDRMux=0; MDRCk
18. AMux=0, ALU=0, AndZ=1, CMux=1, C=3; ZCk, LoadCk
```

FIGURE 12.13
The result of executing the control sequence of Figure 12.12.



specifier, and cycle 3 takes care of a possible carry out from the low-order addition to the high-order addition. Cycle 5 puts the computed value of T2 into the MAR over the ABus and BBus at the same time that it puts the MDR into T3 over the CBus.

Transferring Mem[T2] to T3<low>

Cycles 6–9 use the address computed in T2 to fetch the low-order byte of the address of the operand. Cycle 6 initiates the memory read, cycle 7 continues the memory read, and cycle 8 completes the read and latches the byte from memory into the MDR. Cycle 9 routes the byte from the MDR through AMux and into the low-order byte of T3.

Transferring Mem[T3] to X<high> and adding 1 to T3 storing in T4

At this point, we can assert that temporary register T3 contains the address of the operand, 26D1 (hex) in this example. Finally, we can get the first byte of the operand and load it into the first byte of the index register. Cycles 10–14 perform that load. The RTL specification for `LDWr` shows that the instruction affects the N and Z bits. The N bit is determined by the sign bit of the most significant byte. Consequently, cycle 14 includes the clock pulse NCk to save the N bit as the byte goes through the ALU. The Z bit depends on the value of both bytes, so cycle 14 also contains `AndZ=0` and `ZCk` to save the Zout signal in the Z register. Cycles 11 and 12 increment T3, putting the result in T4 concurrently with the memory read.

Transferring Mem[T4] to X<low>

Cycles 15–18 get the second byte into the index register. Cycle 18 contains `AndZ=1` so that the Z value from the low-order byte (stored in cycle 14) will be ANDed with the Zout from the high-order byte. `ZCk` stores the correct Z value for the 16-bit quantity that the instruction loads.

The first unit test initializes the values to correspond to the values of Figure 12.13. With these initial values, the final value of Z is 0, because the value loaded is not all zeros. The second unit test loads two bytes of all zeros into the index register, and so the final value of Z is 1. The second unit test also tests the internal carry with the shadow bit when it increments 00FF (hex).

The Arithmetic Shift Right Instruction

FIGURE 12.14 shows the control sequence to execute the unary instruction

```
ASRA
```

The RTL specification for the `ASRr` instruction is

$$C \leftarrow r\langle 15\rangle \text{ , } r\langle 1..15\rangle \leftarrow r\langle 0..14\rangle \text{ ; } N \leftarrow r < 0 \text{ , } Z \leftarrow r = 0$$

The V bit is unaffected by `ASRr` because it is impossible to overflow with a shift right instruction. The `ASRr` instruction is unary, so there are no memory accesses. That makes the control sequence nice and short.

Because the ALU computes only with 8-bit quantities, it must break the 16-bit shift into two 8-bit computations. Figure 10.62 shows the four shift and rotate computations that the ALU can perform. To do the arithmetic shift right, the control sequence does an arithmetic shift right of the high-order byte followed by a rotate right of the low-order byte.

ASR of high-order byte

In cycle 1, `A=0` puts the high-order byte of the accumulator on the ABus, `AMux=1` sends it to the ALU, `ALU=13` selects the arithmetic shift right operation, `CMux=1` and `C=0` direct the result to be stored back into the accumulator, and `LoadCk` stores it. `AndZ=0` routes the Zout from the shift operation to the Z register, and `ZCk` saves it. `NCk` saves the N bit from the high-order operation, which will be its final value. `SCk` saves the shadow carry S bit from the high-order operation, which will not be the final value of C.

ROR of low-order byte

In cycle 2, `A=1` puts the low-order byte of the accumulator on the ABus, `AMux=1` sends it to the ALU, `CSMux=1` selects the shadow carry bit for Cin of the ALU, `ALU=14` selects the rotate right operation, `CMux=1` and `C=1` direct the result to be stored back into the accumulator, and `LoadCk` stores it. `AndZ` causes the AndZ combinational circuit to perform the AND operation on

FIGURE 12.14
The control signals to implement the unary ASRA instruction.

```
// ASRA
// RTL: C <- A<15>, A<1..15> <- A<0..14>; N <- A<0, Z <- A=0

UnitPre: IR=0x0C0000, A=0xFF01, N=1, Z=1, V=1, C=0, S=0
UnitPost: A=0xFF80, N=1, Z=0, V=1, C=1

// UnitPre: IR=0x0C0000, A=0x7E00, N=1, Z=1, V=0, C=0, S=1
// UnitPost: A=0x3F00, N=0, Z=0, V=0, C=0

// UnitPre: IR=0x0C0000, A=0x0001, N=1, Z=1, V=0, C=0, S=1
// UnitPost: A=0x0000, N=0, Z=1, V=0, C=1

// Arithmetic shift right of high-order byte.
1. A=0, AMux=1, ALU=13, AndZ=0, CMux=1, C=0; NCk, ZCk, SCk, LoadCk
// Rotate right of low-order byte.
2. A=1, AMux=1, CSMux=1, ALU=14, AndZ=1, CMux=1, C=1; ZCk, CCk, LoadCk
```

Zout and Z, which ZCk stores in Z as its final value. CCk stores Cout as the final value of C.

Figure 12.14 shows three unit tests for the ASRA implementation. The first tests duplication of the sign bit when it is 1 and the shadow carry bit when it is 1. The second tests duplication of the sign bit when it is 0 and the shadow carry bit when it is 0. The third tests the implementation when the final result is all zeros and the final Z bit is 1. In all three unit tests, the value of the V bit is unchanged.

The CPU Control Section

Figure 12.1 shows the CPU divided into a data section and a control section. Given a sequence of control signals necessary to implement an ISA3 instruction, such as the sequence in Figure 12.9 to implement the STBA instruction, the problem is determining how to design the control section to generate that sequence of signals.

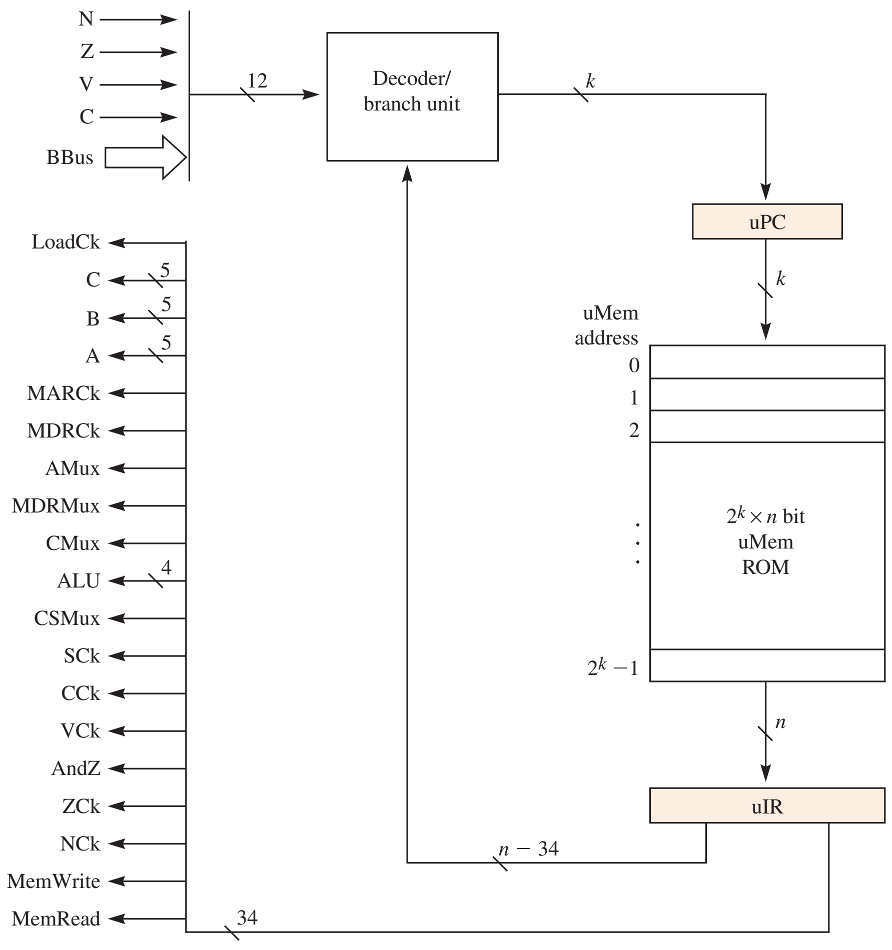
Level Mc2

The idea behind microcode is that a sequence of control signals is, in effect, a program. Figure 12.4, which is a description of the von Neumann cycle, even looks like a C program. One way to design the control section is to create a lower-level von Neumann machine as a microcode level of abstraction that lies between ISA3 and LG1. Like all levels of abstraction, this level has its own language consisting of a set of microprogramming statements. The control section is its own micromachine with its own micromemory, uMem; its own microprogram counter, uPC; and its own microinstruction register, uIR. Unlike the machine at Level ISA3, the machine at Level Mc2 has only one program that is burned into uMem ROM. Once the chip is manufactured, the microprogram can never be changed. The program contains a single loop whose sole purpose is to implement the ISA3 von Neumann cycle.

uMem, uPC, and uIR

FIGURE 12.15 shows the control section of Pep/9 implemented in microcode at Level Mc2. The data section of Figure 12.2 has 34 control lines

FIGURE 12.15
A microcode implementation of the control section of the Pep/9 CPU.



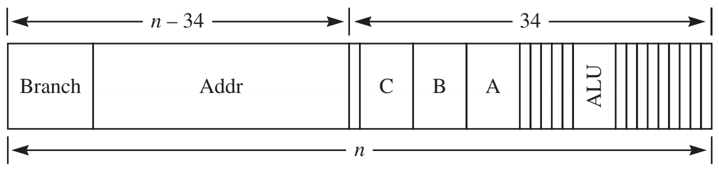
coming from the right to control the data flow. These correspond to the 34 control lines going to the left from Figure 12.15. In the data section of Figure 12.2, there are a total of 12 data lines going to the right to the control section—8 from the BBus and 4 from the status bits. These correspond to the 12 data lines coming from the right in Figure 12.15.

The microprogram counter contains the address of the next microinstruction to execute. uPC is k bits wide, so that it can point to any of the 2^k instructions in uMem. A microinstruction is n bits wide, so that is the width of each cell in uMem and also the width of uIR. At Level Mc2, there is no reason to require that the width of a microinstruction be an even power of 2. n can be any oddball value that you want to make it. This flexibility is due to the fact that uMem contains only instructions with no data. Because instructions and data are not commingled, there is no need to require the memory cell size to accommodate both.

FIGURE 12.16 shows the instruction format of a microinstruction. The rightmost 34 bits are the control signals to send to the data section. The remaining field consists of two parts—a Branch field and an Addr field. The program counter at Level ISA3 is incremented because the normal flow of control is to have the instructions stored and executed sequentially in main memory. The only deviation from this state of affairs is when PC changes due to a branch instruction. uPC at Level Mc2, however, is not incremented. Instead, every microinstruction contains within it information to compute the address of the next microinstruction. The Branch field specifies *how* to compute the address of the next microinstruction, and the Addr field contains data to be used in the computation.

For example, if the next microinstruction does not depend on any of the 12 signals from the data section, Branch will specify an unconditional branch, and Addr will be the address of the next microinstruction. In effect, every instruction is a branch instruction. To execute a set of microinstructions in sequence, you make each microinstruction an unconditional branch to the next one. The Decoder/branch unit in Figure 12.15 is designed to pass Addr straight through to uPC when the Branch field specifies an unconditional branch, regardless of the values of the 12 lines from the data section.

FIGURE 12.16
The instruction format of a microinstruction.



An example where a conditional microbranch is necessary is the implementation of `BRLT`. The RTL specification of `BRLT` is

$$N = 1 \Rightarrow PC \leftarrow Oprnd$$

If the N bit is 1, PC gets the operand. To implement `BRLT`, a microinstruction must check the value of the N bit and either do nothing or branch to a sequence of microinstructions that replaces the PC with the operand. This microinstruction would contain a Branch field that specifies that the next address is computed by combining N with Addr. If N is 0, the computation will produce one address, and if N is 1, it will produce another.

In general, conditional microbranches work by computing the address of the next microinstruction from Addr and whatever signals from the data section that the condition depends on. The biggest conditional branch of all is the branch that decides which ISA3 instruction to execute—in other words, the decode part of the von Neumann cycle. The microinstruction to decode an ISA instruction would have 8 in its B field, which would put the first byte of IR on the BBus. (See Figure 12.2 for the register address of the IR.) The Branch field would specify an instruction decode, and the Decoder/branch unit would be designed to output the address of the first microinstruction in the sequence to implement that instruction.

The details of the Branch and Addr fields in a microinstruction as well as the implementation of the Decoder/branch unit are beyond the scope of this text. Although Figures 12.15 and 12.16 ignore many practical issues in the design of a microcode level, they do illustrate the essential design elements of Level Mc2.

12.2 Performance

From a theoretical perspective, all real von Neumann computing machines are equivalent in their computation abilities. Given a mechanism to connect an infinite amount of disk memory to a machine, it is equivalent in computing power to a Turing machine. The only difference between what Pep/9 can compute and what the world's largest supercomputer can compute is the time it takes to perform the computation. Granted, it might take a million years for Pep/9 to compute the solution to a problem that a supercomputer could compute in a microsecond, but theoretically they can do the same things.

From a practical perspective, time matters. All other things being equal, faster is better. Although the data section in Figure 12.2 at Level LG1 can implement Pep/9 at Level ISA3, the question is, how fast? The fundamental source of increased performance is the space/time tradeoff. Hardware engineers can decrease the time of the computation by adding circuitry—that is, increasing the space—on the chip. There are two aspects of time in all computations—the time to perform the computation and the time to move information between components of the computer system.

Three common techniques for increasing performance in a computer system are:

The three common techniques for increasing performance

- Increasing the width of the data bus
- Inserting a cache between the CPU and the memory subsystem
- Increasing hardware parallelism with pipelining

The first two techniques decrease the time to move information between main memory and the CPU. The third technique decreases the time to perform the computation. All three techniques increase space in the system to decrease the execution time. This section describes how increasing the data bus width and using a cache improve performance. The next section describes the pipeline design of the MIPS machine.

The Data Bus Width and Memory Alignment

The most straightforward way to decrease the time to move information between main memory and the CPU is to increase the width of the data bus. If you increase the data bus width from 8 lines to 16 lines, then each memory read will get two bytes from memory instead of one. Figure 12.12 shows the implementation of a load instruction with indirect addressing. Cycles 2–4 get a low-order byte and cycles 6–8 get a high-order byte. If the data bus had 16 lines, then both bytes could be read in one memory access with only three cycles instead of six.

Figure 12.2 shows the system bus with 16 lines for addresses and 8 lines for data. To accommodate the 16-bit width of the address bus, the memory address register has two parts, MARA and MARB. If the data bus has 16 lines instead of 8, then the memory data register must have two parts as well. FIGURE 12.17 shows the design of a Pep/9 CPU that has a data bus with 16 lines. The register bank is identical to that of Figure 12.2 and is not shown in this figure. The memory subsystem on the left is also not shown. The two parts of the memory data register are MDREven and MDROdd. The increase in space comes from the extra wires on the data bus and the circuitry for the extra memory data register. That increase in space makes possible the decrease in the computation time.

The space/time tradeoff

If the system is designed to access memory in two-byte chunks, you could design main memory to be word-addressable instead of byte-addressable. In the early days of computing, manufacturers designed memory subsystems with different cell sizes, some of which had an address

FIGURE 12.17
The data section of the Pep/9 CPU with a two-byte data bus.

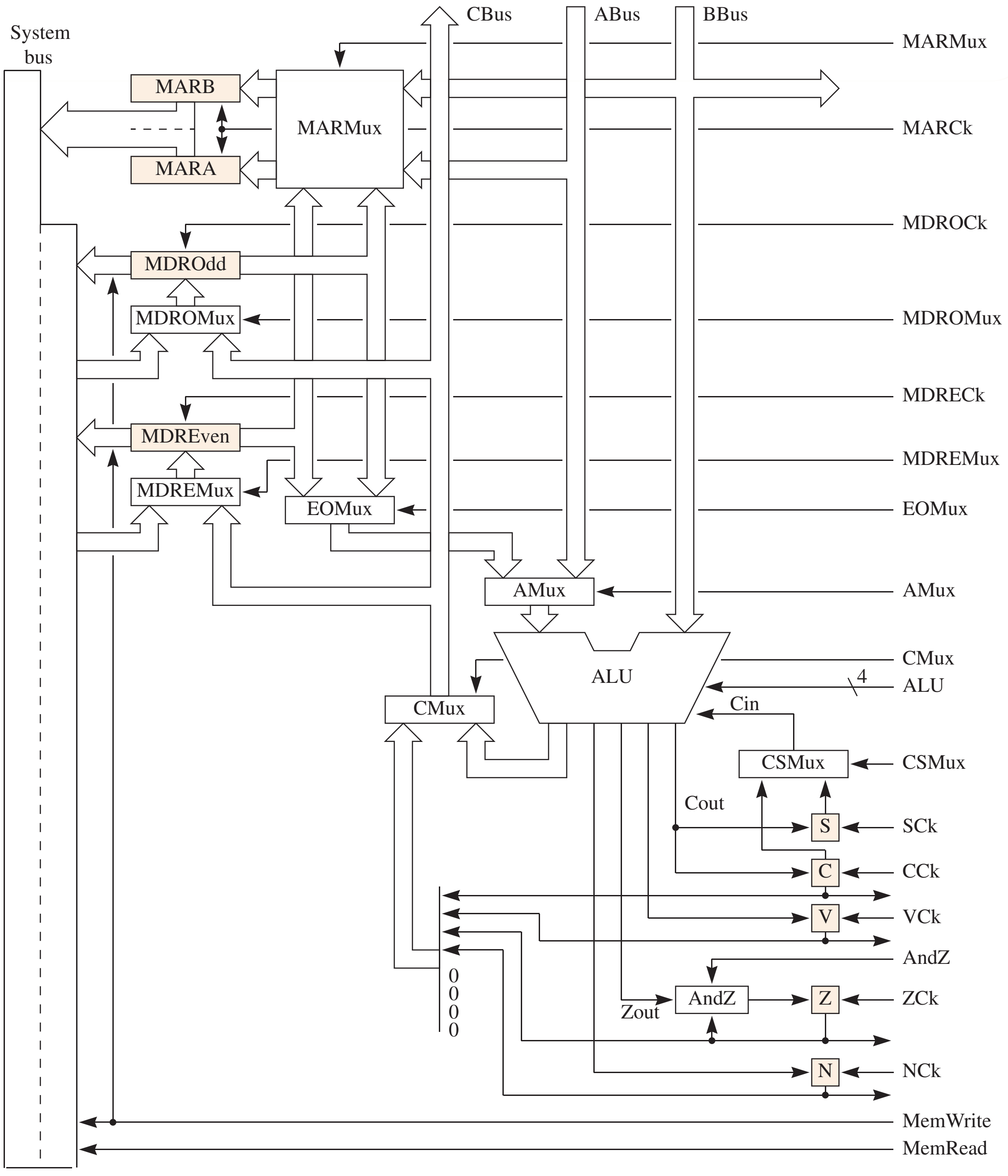


FIGURE 12.18
Addresses of data delivered by the memory subsystem to the CPU with a two-byte data bus.

CPU Address Request	Delivered	
	MDREven	MDROdd
0AB6	0AB6	0AB7
0AB7	0AB6	0AB7
0AB8	0AB8	0AB9
0AB9	0AB8	0AB9

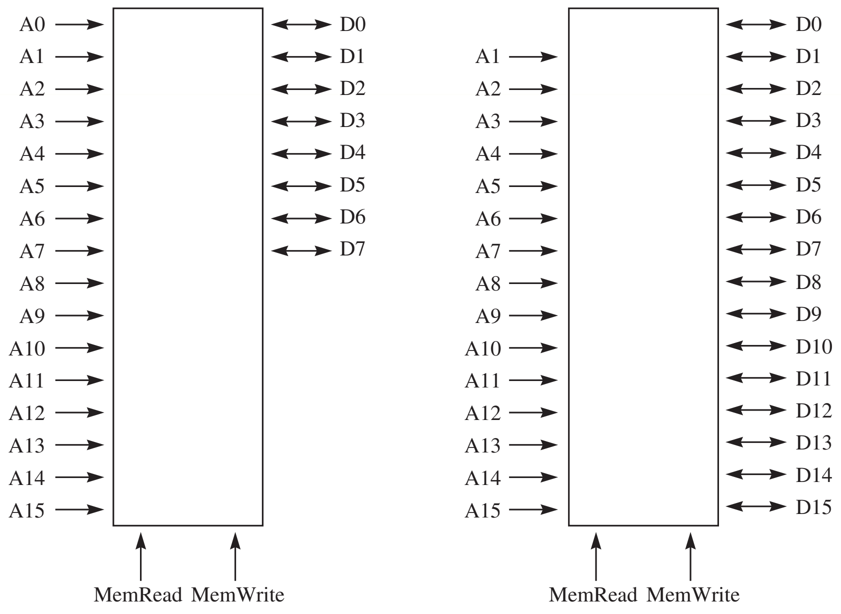
for each two-byte word instead of for each byte. Today, however, virtually all computer memories are byte-addressable.

FIGURE 12.18 shows the addresses of the data delivered by the memory subsystem to the CPU in response to some example CPU requests. If the CPU puts 0AB6 in the MAR, the memory will deliver the content of Mem[0AB6] in MDREven and Mem[0AB7] in MDROdd. That is, it delivers the byte at the requested address and the byte at the *following* address. However, if the CPU puts 0AB7 in the MAR, the memory system delivers the same two bytes in the same data registers. That is, it delivers the byte at the requested address and the byte at the *preceding* address. It always delivers data from an even address to MDREven and from an odd address to MDROdd, regardless of whether the CPU memory request is even or odd.

FIGURE 12.19 shows the memory pinout of the chips for the 8-line data bus of Figure 12.2 and for the 16-line data bus of Figure 12.17. One obvious difference between the two pinouts is the set of 16 data lines for the two-byte bus as opposed to the 8 data lines for the one-byte bus. Another difference is the absence of the lowest-order address line A0 in the two-byte bus in part (b). Why is it missing? Because it is never used. Figure 12.18 shows that a memory request from 0AB6 and one from 0AB7 produce the same access—namely, Mem[0AB6] delivered to MDREven and Mem[0AB7] delivered to MDROdd. Because 0AB6 (hex) is 0000 1010 1011 0110 (bin) and 0AB7 (hex) is 0000 1010 1011 0111 (bin), the last bit A0 is irrelevant to the memory access. The address line A0 literally does not need to be on the bus, which makes it physically a 15-line address bus even though it is logically a 16-line address bus.

Figure 12.2 for the one-byte data bus shows the MDR output connected to the AMux input so you can inject it into the ABus–CBus data loop. With the two-byte data bus of Figure 12.17, however, there are two memory data registers. The EOMux, which stands for *even-odd multiplexer*, selects one of

FIGURE 12.19
The pinout diagrams for two Pep/9 main memory chips.



(a) The chip of Figure 12.2 with an 8-line data bus.

(b) The chip of Figure 12.17 with a 16-line data bus.

the two memory data registers to be sent to the AMux input and from there to the ABus–CBus data loop. It works like the other multiplexers, with an EOMux control value of 0 routing the MDREven output to the input of AMux and a control value of 1 routing the MDROdd output to the input of AMux.

Another performance feature of the Pep/9 CPU design of Figure 12.17 is an additional two-byte data path from the MDR registers to the MAR registers. This path alleviates a bottleneck in those cases when a memory address is read from memory and subsequently needs to be sent to the MAR. Instead of routing the address one byte at a time from the MDR registers, through the register bank, and then to the MAR, this two-byte data path allows both bytes of the MDR registers to be sent to the MAR registers in only one cycle. The box labeled *MARMux* is a 16-bit multiplexer controlled by the MARMux signal. Here is the rule for the control signal:

MARMux control signals

- 0 on the MARMux control line routes MDREven to MARA and MDROdd to MARB.
- 1 on the MARMux control line routes ABus to MARA and BBus to MARB.

Even though increasing the data bus width from one to two bytes increases the performance, it does complicate the von Neumann cycle. With the one-byte data bus, you first fetch the instruction specifier, which is one byte. If the instruction is nonunary, you then fetch the operand specifier, which is two bytes. Suppose you start the von Neumann cycle with the program loaded at address 0000. After the read from memory, you will have the instruction specifier in MDREven and the following byte in MDROdd. If the first instruction is unary, the byte in MDROdd is the instruction specifier of the following instruction. If the first instruction is nonunary, the byte in MDROdd is the first byte of the operand specifier. In either case, it is efficient to store the second byte in temporary register T1 in the register bank to be used later without having to reread it from memory.

FIGURE 12.20 shows the control sequence to fetch the instruction specifier and increment the PC. It is similar to the microcode sequence of Figure 12.5, which requires only five cycles. The extra cycle in Figure 12.20 is cycle 6, which stores the subsequent byte in register T1 in anticipation of the next fetch.

The time savings becomes apparent only when you consider the instruction specifier fetch when you know that a previous memory access has prefetched it into register T1. FIGURE 12.21 shows the control sequence

FIGURE 12.20
The fetch and increment part of the von Neumann cycle with the two-byte data bus.

```
// Fetch the instruction specifier and increment PC by 1
// Assume: PC is even and pre-fetch the next byte

UnitPre: IR=0x000000, PC=0x00FE, Mem[0x00FE]=0xABCD, S=1
UnitPost: IR=0xAB0000, PC=0x00FF, T1=0xCD

// MAR <- PC.
1. A=6, B=7, MARMux=1; MARCk
// Initiate fetch, PC <- PC + 1.
2. MemRead, A=7, B=23, AMux=1, ALU=1, CMux=1, C=7; SCk, LoadCk
3. MemRead, A=6, B=22, AMux=1, CSMux=1, ALU=2, CMux=1, C=6; LoadCk
4. MemRead, MDREMux=0, MDROMux=0; MDRECk, MDROCk
// IR <- MDREven, T1 <- MDROdd.
5. EOMux=0, AMux=0, ALU=0, CMux=1, C=8; LoadCk
6. EOMux=1, AMux=0, ALU=0, CMux=1, C=11; LoadCk
```

FIGURE 12.21
The fetch and increment part of the von Neumann cycle with pre-fetched instruction specifier.

```
// Fetch the instruction specifier and increment PC by 1
// Assume instruction specifier has been pre-fetched

UnitPre: IR=0x000000, PC=0x01FF, T1=0x12, S=0
UnitPost: IR=0x120000, PC=0x0200

// Fetch instruction specifier.
1. A=11, AMux=1, ALU=0, CMux=1, C=8; CCk

// PC <- PC plus 1.
2. A=7, B=23, AMux=1, ALU=1, CMux=1, C=7; SCk, LoadCk
3. A=6, B=22, AMux=1, CSMux=1, ALU=2, CMux=1, C=6; LoadCk
```

under this scenario. No memory read is required at all. The entire sequence is only three cycles. The control section can easily determine at the beginning of the cycle whether the instruction specifier was prefetched. If the program counter is even, it was not and the sequence in Figure 12.20 is necessary. If PC is odd, it uses the sequence in Figure 12.21.

A similar consideration is used for fetching the operand specifier. If the program counter is even, then no byte has been prefetched. The control section can get both bytes of the operand specifier with only one memory access. If the program counter is odd, then the first byte of the operand specifier has been prefetched. Again, only one memory access is required, which gets the second byte of the operand specifier and prefetches the subsequent instruction specifier. Another performance enhancement is the ability to increment PC by 2 only once instead of incrementing it by 1 twice, which is necessary with the one-byte data bus, as in Figure 12.4. Implementation of the control sequences to fetch the operand specifier under these scenarios is a problem for the student at the end of the chapter.

Memory Alignment

Using register T1 to store prefetched bytes increases the performance of the fetch part of the von Neumann cycle. Maximizing the performance of the execute part of the von Neumann cycle requires memory alignment of both data and program statements. For example, FIGURE 12.22(a) shows a code fragment from the program of Figure 5.27. Global variable `exam1` is stored at

FIGURE 12.22
Data alignment in the program of Figure 5.26.

```
0000   120009           BR       main
              bonus:    .EQUATE 10
0003   0000   exam1:    .BLOCK   2
0005   0000   exam2:    .BLOCK   2
0007   0000   score:    .BLOCK   2
              ;
0009   310003 main:     DECI     exam1,d
000C   310005           DECI     exam2,d
000F   C10003           LDWA     exam1,d
0012   610005           ADDA     exam2,d
```

(a) Without data alignment.

```
0000   12000A           BR       main
              bonus:    .EQUATE 10
0003   00               .ALIGN   2
0004   0000   exam1:    .BLOCK   2
0006   0000   exam2:    .BLOCK   2
0008   0000   score:    .BLOCK   2
              ;
000A   310004 main:     DECI     exam1,d
000D   310006           DECI     exam2,d
0010   C10004           LDWA     exam1,d
0013   610006           ADDA     exam2,d
```

(b) With data alignment.

fixed location 0003. Consider execution of the `LDWA` instruction. It loads the accumulator with the value stored at 0003. Because 0003 is odd, a memory request by the CPU from this location puts Mem[0002] in MDREven and Mem[0003] in MDROdd. Even though the memory access loads two bytes from memory, it loads only the first byte of `exam1` into the memory data register. The CPU can use only the value in MDROdd and needs to make a second memory access for the value from Mem[0004]. The benefit of the two-byte data bus is lost, as two memory accesses are still required.

Alignment of data

If the first byte of `exam1` is stored at an even address, both bytes can be accessed with a single memory access. Figure 12.22(b) shows the same program but with an additional `.ALIGN` dot command inserted before the declaration of `exam1`. The effect of a `.ALIGN` command is to insert an extra zero byte if necessary so that the code generated by the following line will begin at an even address. In Figure 12.22(b), the extra byte generated by the alignment dot command forces `exam1` to be stored at 0004 instead of at 0003. With this program, the `LDWA` instruction causes the CPU to request a memory access from 0004, which puts Mem[0004] in MDREven and Mem[0005] in MDROdd. Only one memory access is required to load the value of the integer.

The `.ALIGN` assembler directive

Alignment commands in assembly languages take an integer argument that is a power of 2. In Figure 12.22(b), the alignment command is

```
.ALIGN 2
```

The argument `2` inserts enough zero bytes in the code to force the code on the next line to be at an even address—that is, at an address divisible by 2. Suppose you increase the performace further by designing a four-byte data bus. You would then have four memory data registers—MDR0, MDR1, MDR2, and MDR3. The EOMux of Figure 12.17 would be a four-input multiplexer with two control lines instead of one. The memory chip of Figure 12.17(b) would have 32 data lines instead of 16 and would be missing address lines A0 and A1 instead of just A0. The alignment command

```
    .ALIGN 4
```

would insert enough zero bytes in the code to force the code on the next line to be at the next address divisible by 4. Similarly, the alignment command

```
    .ALIGN 8
```

would insert enough zero bytes in the code to force the code on the next line to be at the next address divisible by 8, appropriate for an eight-byte data bus.

Alignment on the run-time stack

Figure 12.22 shows how the alignment command maximizes performance with global variables stored at a fixed location in memory. The same optimization technique is necessary for local variables and parameters stored on the run-time stack. With a two-byte data bus, the initial value of the stack pointer is aligned to an even address. The compiler must generate assembly language code, possibly padded with zero bytes so that each variable is aligned with an even-address boundary in memory. For example, the trap mechanism described in Section 8.2 shows the process control block on the system stack. With a two-byte data bus, the instruction specifier is stored in a two-byte cell padded with a zero byte even though it is only one byte long. Each register is stored at an even address, and the NZVC bits are stored in a two-byte cell padded with a zero byte.

FIGURE 12.23 shows the implementation of the load word instruction with indirect addressing using the two-byte bus design. The unit test assumes that the operand specifier is 0012, which is even, and that Mem[0012] is 26D2, which is also even. Thus, it assumes data alignment on two-byte boundaries. Cycle 5 shows how the content of both bytes from the MDR can be sent to the MAR in one cycle. Compared to Figure 12.12, which is the implementation of the same instruction with the one-byte bus, this implementation has only two memory accesses instead of four for a total of 10 cycles instead of 18. The savings in time is thus 8 cycles out of the original 18, or 44%.

Alignment of program statements

Executing a program correctly with the two-byte data bus of Figure 12.17 requires the alignment of some program statements in memory. Branch statements cause a problem with the wide bus because they change the value of the program counter. Figure 12.21 shows the sequence for the

FIGURE 12.23
The two-byte bus implementation of the load word instruction with indirect addressing.

```
// LDWX this,n
// RTL: X <- Oprnd; N <- X<0, Z <- X=0
// Indirect addressing: Oprnd = Mem[Mem[OprndSpec]]

UnitPre: IR=0xCA0012, Mem[0x0012]=0x26D2, Mem[0x26D2]=0x53AC
UnitPre: N=1, Z=1, V=0, C=1
UnitPost: X=0x53AC, N=0, Z=0, V=0, C=1

// MDR <- Mem[OprndSpec].
1. A=9, B=10, MARMux=1; MARCk
2. MemRead
3. MemRead
4. MemRead, MDROMux=0, MDREMux=0; MDROCk, MDRECk

// MAR <- MDR.
5. MARMux=0; MARCk

// MDR <- two-byte operand.
6. MemRead
7. MemRead
8. MemRead, MDROMux=0, MDREMux=0; MDROCk, MDRECk

// X <- MDR, high-order first.
9. EOMux=0, AMux=0, ALU=0, AndZ=0, CMux=1, C=2; NCk, ZCk, LoadCk
10. EOMux=1, AMux=0, ALU=0, AndZ=1, CMux=1, C=3; ZCk, LoadCk
```

fetch part of the von Neumann cycle if PC is odd. The hardware assumes the instruction specifier is prefetched and stored in register T1. Suppose the program branches to a target statement stored at an odd location. To execute the von Neumann cycle for the target instruction, the hardware will access T1 for the instruction specifier even though it was not prefetched. For a program to work correctly with such hardware, every symbol defined on a statement that is the target of a branch instruction must be located on an even address. After the branch occurs, the program counter will be even. The fetch sequence of Figure 12.20 will execute and fetch the instruction specifier of the target instruction correctly.

FIGURE 12.24 shows the program alignment that is necessary for the program of Figure 6.8. Part (a) shows the initial branch to main as a branch

FIGURE 12.24
Program alignment in the program of Figure 6.8.

```
0000  120003            BR       main
             limit:     .EQUATE 100
             num:       .EQUATE 0
             ;
0003  580002 main:      SUBSP    2,i
0006  330000            DECI     num,s
0009  C30000 if:        LDWA     num,s
000C  A00064            CPWA     limit,i
000F  160018            BRLT     else
0012  49001F            STRO     msg1,d
0015  12001B            BR       endIf
0018  490025 else:      STRO     msg2,d
001B  500002 endIf:     ADDSP    2,i
001E  00                STOP
```

(a) Without program alignment.

```
0000  120004            BR       main
             limit:     .EQUATE 100
             num:       .EQUATE 0
             ;
0003  00                .ALIGN   2
0004  580002 main:      SUBSP    2,i
0007  330000            DECI     num,s
000A  C30000 if:        LDWA     num,s
000D  A00064            CPWA     limit,i
0010  16001A            BRLT     else
0013  490022            STRO     msg1,d
0016  12001E            BR       endIf
0019  26                NOP0
001A  490028 else:      STRO     msg2,d
001D  26                NOP0
001E  500002 endIf:     ADDSP    2,i
0021  00                STOP
```

(b) With program alignment.

to an instruction at 0003, which is odd. The `.ALIGN` statement in part (b) fixes that problem, forcing the main program code to begin at 0004, which is even. Inserting a `.ALIGN` statement at the beginning of every function, including `main`, is required. The program contains two other branch targets, `if` and `endIf`. To force these statements to lie on even addresses, the assembler must insert the unary no operation instruction `NOP0` before each target instruction. In Pep/9, the `NOP0` instruction is a trap instruction and so would cause a big performance degradation that would nullify the benefit of the wide data bus. All real processors have native no-operation instructions that execute in one cycle and are used to align program statements.

Two other statements that modify the program counter are `call` and `ret`. The `call` statement is a three-byte instruction that puts the incremented program counter on the run-time stack as the return address. The `ret` statement branches to that address. Thus, the address following the `call` statement must be at an even address. It follows that every `call` statement must be at an odd address and can therefore never be the target of a branch instruction. To fix a program that contains a branch to a `call` statement, you must branch instead to a no-operation instruction at an even address just before the `call`.

The Definition of an *n*-Bit Computer

The generally accepted meaning of "an n-bit computer" is that n is the number of bits in the MAR and in the CPU registers that are visible at Level ISA3. Because the registers visible at Level ISA3 can hold addresses, the registers are usually the same width as the MAR, so this definition is unambiguous. For example, Pep/9 is clearly a 16-bit computer because the registers visible at the ISA3 level that hold data, like the accumulator and the index register, are 16-bit cells. Furthermore, registers that hold addresses, like the program counter and the stack pointer, are also 16-bit cells.

There is frequently confusion about this definition, especially in marketing. An n-bit computer does not necessarily have n-bit data buses within the data section of its CPU. Nor does it necessarily have n-bit registers in its register bank at Level LG1. All these widths can be less than n. Pep/9 is an example of such a machine. Even though it is a 16-bit machine, the ABus, BBus, and CBus are only 8 bits wide. At the Mc2 level, the registers in its register bank are only 8 bits wide.

The classic example is the IBM 360 family, which was introduced in 1964. It was the first family of computers whose models all had the same ISA3 instruction sets and registers. It was a 32-bit machine, but depending on the model, the data buses in the CPU were 8-bit, 16-bit, or 32-bit. Because the LG1 details are hidden from the programmer at Level ISA3, all the software written and debugged on one model in the family ran unchanged on a different model. The only perceived difference between models was performance as measured by execution time. The concept was revolutionary at the time and promoted the design of computers based on levels of abstraction.

Nor must an n-bit computer have n address lines in the main system bus. The width of the MAR determines the maximum number of addressable bytes the system can access. An n-byte computer can access 2^n bytes, as FIGURE 12.25 shows. For example, Pep/9 has a 16-bit MAR and so can access a maximum of 2^{16} bytes or, equivalently, 64 KiB.

Many laptop computers today have 64-bit processors with a theoretical maximum main memory size of 2^{64} bytes, or 17,179,869,184 GiB. Because it will never be physically possible to have that much main memory installed, such systems simply advertise a maximum main memory size and design the system with the corresponding number of lines in the address bus. For example, suppose a computer manufacturer advertises a 64-bit computer for which you can install a maximum of 32 GiB of memory. Because the CPU is a 64-bit processor, its internal MAR is 64 bits wide. To save space on

FIGURE 12.25
Maximum memory limits as a function of the MAR width.

MAR Width	Number of Addressable Bytes
8	256
16	64 Ki
32	4 Gi
64	17,179,869,184 Gi

the circuit board, however, the manufacturer designs the system bus with only 35 address lines connected to MAR lines A0 through A34. Lines A35 through A63 of the MAR remain unconnected. Because 2^{35} bytes is 32 GiB, the product will allow the user to install up to 32 GiB of memory.

It is even possible for the number of data lines in the main system bus to be greater than *n*. For example, Pep/9, which is a 16-bit computer, could have a 32-bit wide data bus that would fetch 8 bytes with each memory read. In the same way that the microcode sequence in Figure 12.20 prefetches an additional byte that eliminates the need for a later memory access, all the unused bytes from the 8-byte memory access can be saved in the CPU, possibly eliminating many future memory accesses. The idea of prefetching more data than necessary to eliminate future memory accesses is the basis of cache memories described in the next section.

Assuming that the internal registers, the internal buses, and the data/address buses on the system bus all have the same width, increasing that common width will increase the performance. Increasing the common width of these components has a large impact on the size of the chip. All the circuits, including the ALU, the multiplexers, and the registers, must be increased to accommodate the larger width. The history of computers shows a progression toward ever-wider buses and registers. FIGURE 12.26 shows how the Intel CPUs increased in bus width from 4 bits to 64 bits. The 4004 was the first microprocessor on a chip. The first 64-bit processor from Intel was a version of the Pentium 4 chip. The market is in the process of completing the transition from 32-bit machines to 64-bit machines. Most desktop and laptop machines today have 64-bit processors, while most mobile devices have 32-bit processors.

The progression is possible only because technology advances at such a regular pace. Gordon Moore, the founder of Intel, observed in 1965 that the density of transistors in an integrated circuit had doubled every year and would continue to do so for the foreseeable future. The rate has slowed

FIGURE 12.26
Historic register/bus widths.

Chip	Date	Register Width
4004	1971	4-bit
8008	1972	8-bit
8086	1978	16-bit
80386	1985	32-bit
x86–64	2004	64-bit

somewhat, so that today the so-called *Moore's law* now states that the density of transistors in an integrated circuit doubles every 18 months. This pace cannot be maintained forever, because the miniaturization will eventually reach the dimensions of an atom, which cannot be subdivided. Exactly when Moore's law will cease has been hotly debated in the past, with many people predicting its demise, only to see it continue on.

Moore's law

The big push for 64-bit computers was because multimedia applications and large databases needed an address space greater than 4 GiB. The transition to 64-bit computers will probably be the last of its kind, however, because 64 address lines can access 16 billion GiB. The increase from one generation to the next is not polynomial, but exponential. For many years, 32-bit computers had 32-bit MARs but only 24 address lines on the main memory bus. The 8 high-order address bits were ignored. Users could install up to 16 MiB (2^{24}) on such machines, which was plenty of memory at the time. It will be the same with 64-bit computers long into the future, where the number of external address lines will be less than the internal width of the MAR, depending on the needs of the market.

Cache Memories

The Pep/9 control sequence has a requirement that memory reads and writes require three cycles because of the excessive time it takes to access the memory subsystem over the main system bus. Although this requirement puts some realism into the model, in practice the speed mismatch between main memory access time and the CPU cycle time is more severe. Suppose it took 10 cycles instead of just 3 for a main memory access. You can imagine what the control sequences would look like—many cycles of MemReads. Most of the time the CPU would be waiting for memory reads, wasting cycles that could be used for making progress on the job.

But what if you could predict the future? If you knew which words from memory would be required by the program ahead of time, you could set up a small amount of expensive, high-speed memory right next to the CPU, called a *cache*, and fetch the instructions and data from main memory ahead of time. The information would then be available for the data section immediately. Of course, no one can predict the future, so this scheme is impossible. Still, even if you could not predict the future with 100% accuracy, what if you could predict it with 95% accuracy? Those times your prediction was correct, the memory access time from the cache would be nearly instantaneous. If the percentage of time you were correct was high enough, the time savings would be substantial.

Cache memory

The problem is determining how to make the prediction. Suppose you could tap the address lines and monitor all the memory requests that a CPU makes when it executes a typical job. Would you expect to see the sequence of addresses come out at random? That is, given one address request, would you expect to see the next one close to it, or would you expect the next one to be at some random location far away?

There are two reasons you should expect to see successive memory requests close together, based on the two things stored in memory. First, the CPU must access instructions during the fetch part of the von Neumann cycle. As long as no branches execute to faraway instructions, it will be requesting from addresses that are all clumped together. Second, the CPU must access data from memory. Recall that the assembly language programs in Chapter 6 all have their data clumped together in memory. It is true that applications and the heap are stored in low memory, and the operating system and the run-time stack are stored in high memory, but you should be able to visualize that for periods of time, the accesses will all be of bytes from the same neighborhood.

The phenomenon that memory accesses are not random is called *locality of reference*. If memory accesses were totally random, then a cache would be totally useless because it would be impossible to predict which bytes from memory to preload. Fortunately, typical access requests exhibit two kinds of locality:

The two types of locality of reference

- *Spatial locality*—An address close to the previously requested address is likely to be requested in the near future.
- *Temporal locality*—The previously requested address *itself* is likely to be requested in the near future.

Temporal locality comes from the common use of loops in programs.

Cache hits and misses

When the CPU requests a load from memory, the cache subsystem first checks to see if the data requested has already been loaded into the cache, an event called a *cache hit*. If so, it delivers the data straightaway. If not, an event called a *cache miss* occurs. The data is fetched from main memory, and the CPU must wait substantially longer for it. When the data finally arrives, it is loaded into the cache and given to the CPU. Because the data was just requested, there is a high probability that it will be requested again in the near future. Keeping it in the cache takes advantage of temporal locality. You can take advantage of spatial locality by bringing into the cache not only the data that is requested, but also a clump of data in the neighborhood of the requested byte. Even though you have brought in some bytes that have not been requested yet, you are preloading them based on a prediction of the future. The probability that they will be accessed in the near future is high because their addresses are close to the address of the previously accessed byte.

Why not build main memory with high-speed circuits like the cache, put it where the cache is, and dispense with the cache altogether? Because high-speed memory circuits require so much more area on a chip. There is a huge size and speed difference between the fastest memory technology and the slowest. It is the classic space/time tradeoff. The more space you are willing to devote per memory cell, the faster you can make the memory operate.

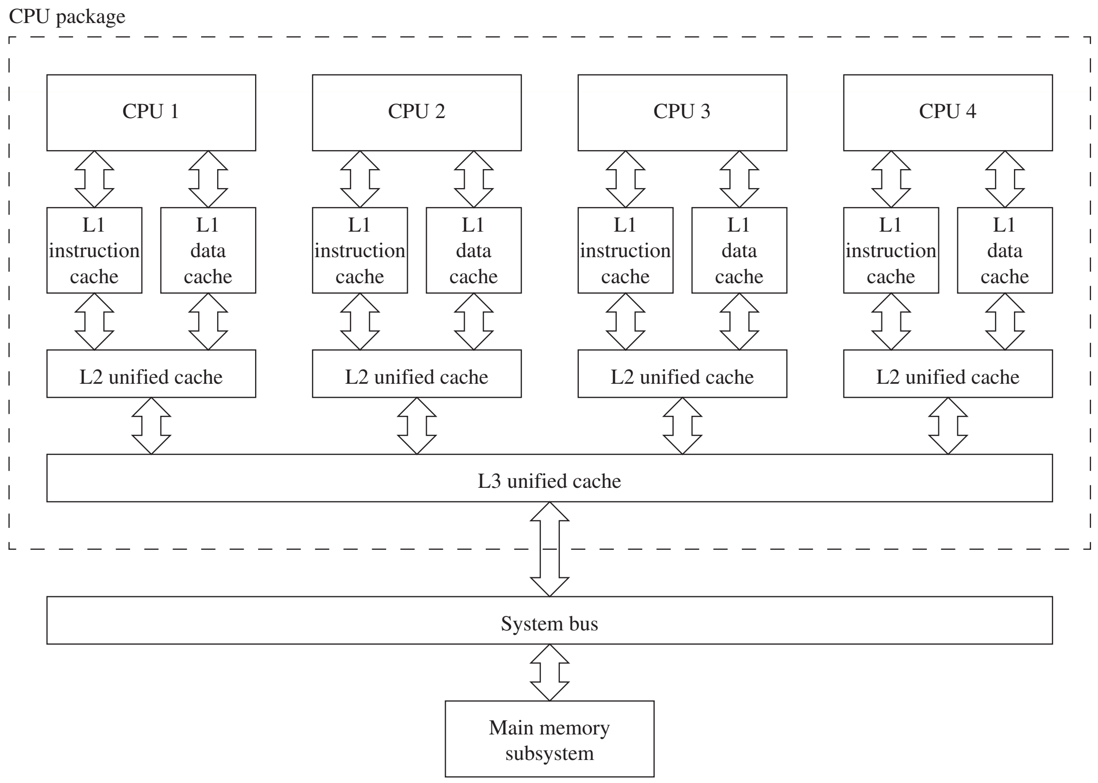
Memory technology provides a range of designs between these two extremes. Corresponding to this range, three levels of cache are typical between the CPU and the main memory subsystem:

Three levels of cache in a computer system

- Split L1 instruction and data cache—smallest, fastest, closest to the CPU
- Unified L2 cache—between the L1 and L3 caches
- Unified L3 cache—largest, slowest, farthest from the CPU

FIGURE 12.27 shows the three levels for a quad-core CPU. In the figure, the dashed line marks the boundary of a single package—that is, a single part

FIGURE 12.27
Three levels of cache in a typical computer system.



that is mounted on the circuit board of the computer system. The L1 cache is smaller and faster than the L2 cache, which is smaller and faster than the L3 cache, which is smaller and faster than the main memory subsystem. Typical sizes are 32 to 64 KiB per core for the L1 cache. (To show how tiny Pep/9 is, its entire main memory would fit inside a typical L1 cache.) The L1 cache runs at CPU speeds. The L2 cache usually runs at one-half or one-quarter the speed of the L1 cache and is four to eight times larger. The L3 cache is usually another factor of four to eight times slower and larger than the L2 cache. The cache next to main memory is also called *last-level cache*. Some designs omit the L3 cache altogether, and others have more than three levels.

The CPU makes a distinction between an instruction fetch as part of the von Neumann cycle and a data fetch of an operand. Accordingly, the L1 cache is split between an instruction cache and a data cache. The L1 cache receives memory requests from the CPU and passes the requests on to the L2 cache in case of a cache miss. The L2 cache is known as a *unified cache* because it stores instructions and data intermixed with no distinction between them. Each core also has its own L2 cache. In the event of a cache miss in the L2 cache, the L2 cache passes the requests on to the L3 cache, a unified cache that is shared between all the cores. In the event of a cache miss in the L3 cache, the L3 cache passes the requests on to the main memory subsystem.

There are two kinds of cache design:

Two types of cache design

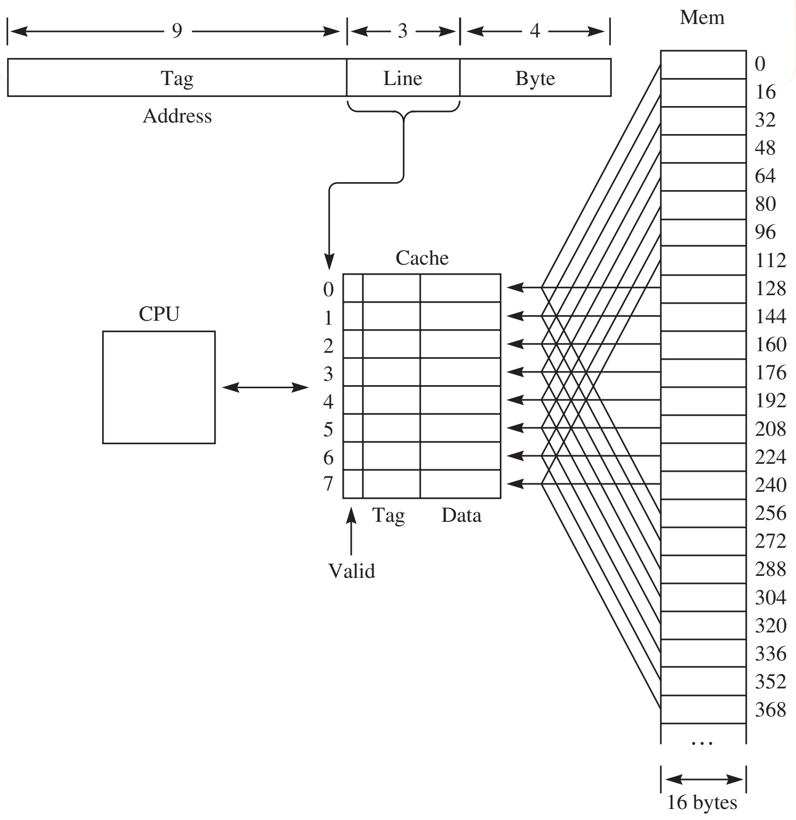
› Direct-mapped cache
› Set-associative cache

The simpler of the two is direct-mapped cache, an example of which is shown in FIGURE 12.28. As usual, the example is unrealistically small to help facilitate the description.

The example is for a system with 16 address lines and 2^{16} = 64 KiB main memory. Memory is divided into 16-byte chunks called *cache lines*. On a cache miss, the system loads not only the requested byte, but also all 16 bytes of the cache line that contains the requested byte. The cache itself is a miniature memory with eight cells addressed from 0 to 7. Each cell is divided into three fields—Valid, Tag, and Data. The Data field is that part of the cache cell that holds a copy of the cache line from memory. The Valid field is a single bit that is 1 if the cache cell contains valid data from memory and 0 if it does not.

The address field is divided into three parts—Tag, Line, and Byte. The Byte field is 4 bits, corresponding to the fact that 2^4 = 16, and there are 16 bytes per cache line. The Line field is 3 bits, corresponding to the fact that 2^3 = 8, and there are eight cells in the cache. The Tag field holds the remainder of the bits in the 16-bit address, so it has 16 – 3 – 4 = 9 bits. A cache cell holds a copy of the Tag field from the address along with the data

FIGURE 12.28
A direct-mapped cache.



from memory. In this example, each cache cell holds a total of 1 + 9 + 128 = 138 bits. As there are eight cells in the cache, the entire cache holds a total of 138 × 8 = 1104 bits.

FIGURE 12.29 is a pseudocode descritption of the operation of a direct-mapped cache. When the system starts up, it sets all the Valid bits in the cache to 0. The very first memory request will be a miss. The address of the cache line from memory to be loaded is simply the Tag field of the address, with the last four bits set to 0. That line is fetched from memory and stored in the cache cell with the Valid bit set to 1 and the Tag field extracted from the address and stored as well.

If another request asks for a byte in the same line, it will be a hit. The system extracts the Line field from the address, goes to that line in the cache, and determines that the Valid bit is 1 and that the Tag field of the request matches the Tag field stored in the cache cell. It extracts the byte or word from the Data part of the cache cell and gives it to the CPU without a memory read. If the Valid bit

FIGURE 12.29
A pseudocode description of the operation of a direct-mapped cache.

```
Extract the Line field from the CPU memory address request
Retrieve the Valid/Tag/Data cache entry from the Line row
if (Valid == 0) {
    Cache miss, memory fetch
} else if (Tag from cache != Tag from memory request) {
    Cache miss, memory fetch
} else {
    Cache hit, use Data field from cache
}
```

is 1 but the Tag fields do not match, it is a miss. A memory request is necessary, and the Tag and Data fields replace the old ones in the same cache cell.

Example 12.1 The CPU requests the byte at address 3519 (dec). What are the nine bits of the Tag field, what are the four bits of the Byte field, and which cell of the cache stores the data? Converting to binary and extracting the fields gives

3519 (dec) = 000011011 011 1111 (bin)

The nine bits in the Tag field are 000011011, the four bits in the Byte field are 1111, and the data is stored at address 011 (bin) = 3 (dec) in the cache. ■

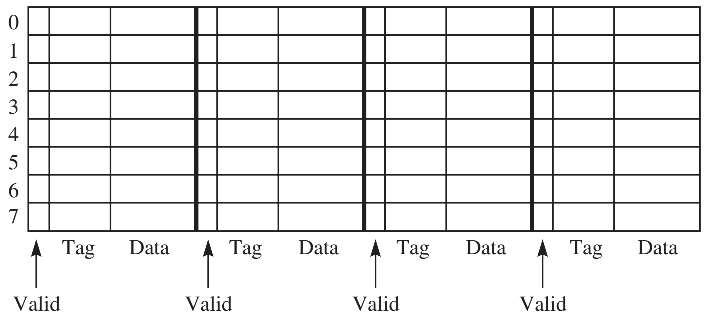
A deficiency of direct-mapped caches

Figure 12.28 shows that the blocks of memory at addresses 16, 144, 272, . . . , all contend for the same cache entry at address 1. Because there are nine bits in the Tag field, there are $2^9 = 512$ blocks in memory that contend for each cache cell, which can hold only one at a time. There is a pattern of requests resulting in a high cache miss rate that arises from switching back and forth between two fixed areas of memory. An example is a program with pointers on the run-time stack at high memory in Pep/9 and the heap at low memory. A program that accesses pointers and the cells to which they point will have such an access pattern. If it happens that the pointer and the cell to which it points have the same Line field in their addresses, the miss rate will increase substantially.

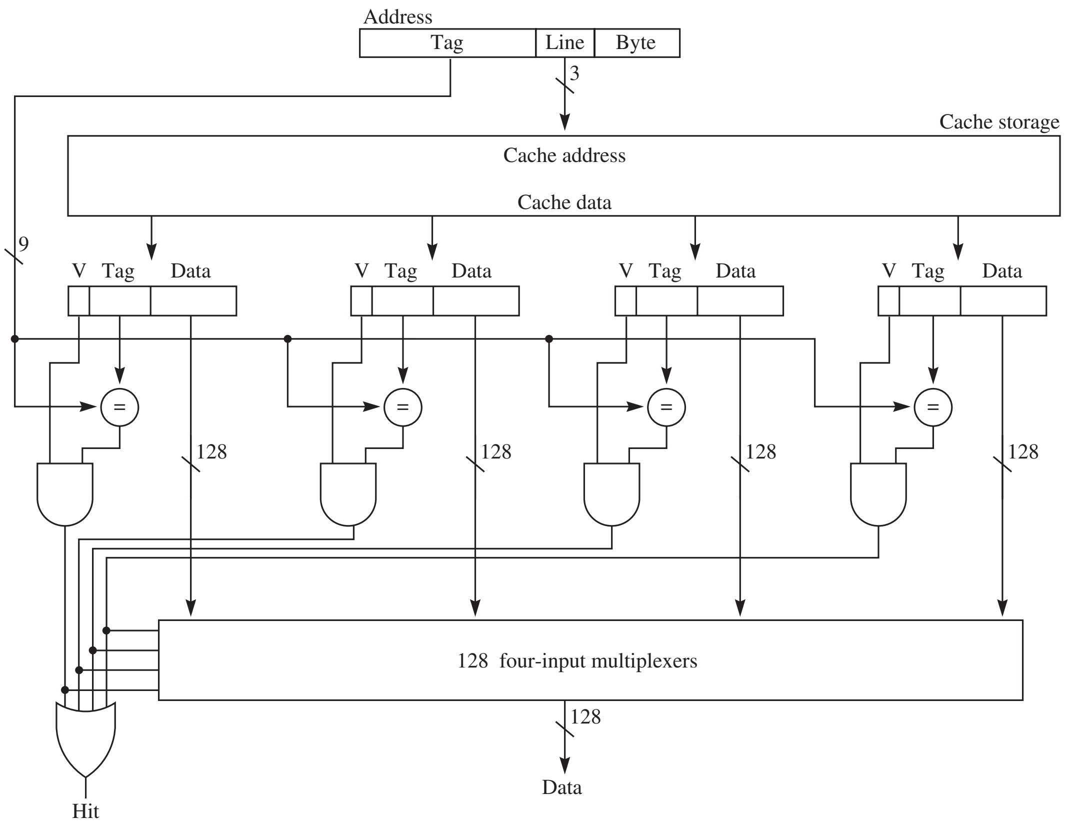
Set-associative caches

Set-associative caches are designed to alleviate this problem. Instead of having each cache entry hold just one cache line from memory, it can hold several. FIGURE 12.30(a) shows a four-way set-associative cache. It duplicates the cache of Figure 12.29 four times and allows a set of up to four blocks of memory with the same Line fields to be in the cache at any

FIGURE 12.30
A four-way set-associative cache.



(a) Block diagram of cache storage.



(b) Implementation of read circuit.

time. The access circuitry is more complex than the circuitry for a direct-mapped cache. For each read request, the hardware must check all four parts of the cache cell in parallel and route the one with the matching Line field if there is one.

Figure 12.30(b) shows the details of the read circuit. The circle with the equals sign is a comparator that outputs 1 if the inputs are equal and 0 otherwise. The bank of 128 multiplexers is simpler than usual. Four-input multiplexers usually have two select lines that must be decoded, but the four select lines of this multiplexer are already decoded. The output labeled *Hit* is 1 on a cache hit, in which case the output labeled *Data* is the data from the cache line with the same Tag field as the one in the requested address. Otherwise Hit is 0.

Set-associative cache line replacement

Another complication with set-associative caches is the decision that must be made when a cache miss occurs and all four parts of the cache cell are occupied. The question is, which of the four parts should be overwritten by the new data from memory? One technique is to use the least recently used (LRU) algorithm. In a two-way, set-associative cache, only one extra bit is required per cache cell to keep track of which cell was least recently used. But in a four-way set-associative cache, it is considerably more complicated to keep track of the least recently used. You must maintain a list of four items in order of use and update the list on every cache request. One approximation to LRU for a four-way cache uses three bits. One bit specifies which group was least recently used, and the bits within each group specify which item in that group was least recently used.

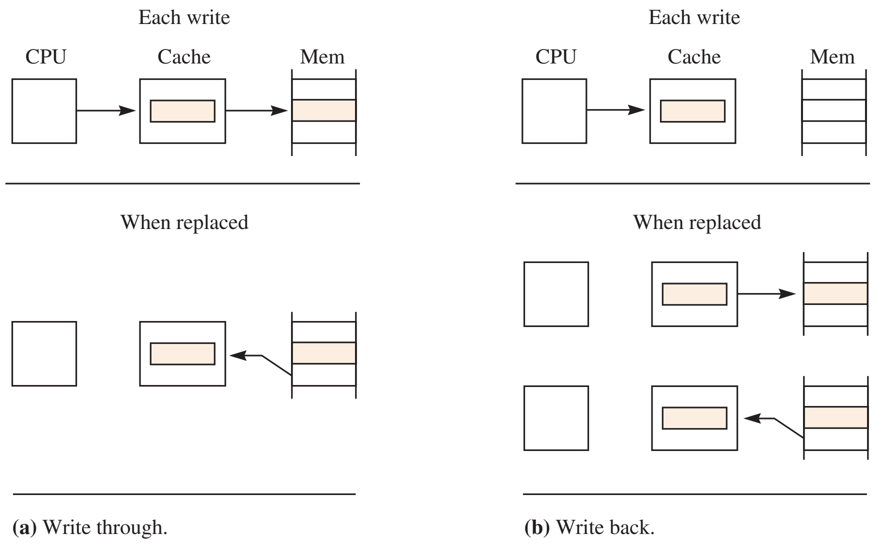
Regardless of whether the cache is direct-mapped or set-associative, system designers must decide how to handle memory writes with caches. There are two possibilities with cache hits:

Two cache write policies with cache hits

- *Write through*—Every write request updates the cache and the corresponding block in memory.
- *Write back*—A write request updates only the cache copy. A write to memory happens only when the cache line is replaced.

FIGURE 12.31 shows the two possibilities. Write through is the simpler design. While the system is writing to memory, the CPU can continue processing. When the cache line needs to be replaced, the value in memory is guaranteed to already have the latest update. The problem is an excessive amount of bus traffic when you get a burst of write requests. Write back minimizes the bus traffic, which could otherwise affect performance of other components wanting to use the main system bus. At any given time, however, memory does not have the most recent copy of the current value of the variable. Also, there is a delay when the cache line must be replaced,

FIGURE 12.31
Cache write policies with cache hits.



because memory must be updated before the new data can be loaded into the cache. By design, this event should happen rarely with a high percentage of cache hits.

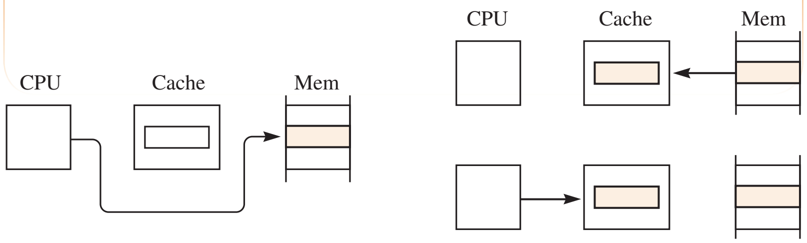
Cache write policies with cache misses

Another issue to resolve with caches is what to do with a write request in conjunction with a cache miss. The policy known as *write allocation* brings the block from memory into the cache, possibly replacing another cache line, and then updates the cache line according to its normal cache write strategy. Without write allocation, a memory write is initiated, bypassing the cache altogether. The idea here is that the CPU can continue its processing concurrently with the completion of the memory write.

FIGURE 12.32 shows the cache write policies without and with write allocation. Although either cache write policy with cache misses can be combined with either cache write policy with cache hits, write allocation is normally used in caches with write back for cache hits. Write-through caches also tend to not use write allocation in order to keep the design simple. The design choice of most caches is either parts (a) of Figures 12.31 and 12.32 on the one hand or parts (b) of Figures 12.31 and 12.32 on the other.

The discussion of cache memories should sound familiar if you have read the discussion of virtual memory in Section 9.2. In the same way that the motivation behind virtual memory is the mismatch between the small size of main memory and the size of an executable program stored on a hard drive, the motivation behind caches is the mismatch between the small size

FIGURE 12.32
Cache write policies with a cache miss.



(a) Without write allocation. **(b)** With write allocation.

of the cache and the size of main memory. The LRU page replacement policy in a virtual memory system corresponds to the LRU replacement policy of a cache line. Cache is to main memory as main memory is to disk. In both cases, there is a *memory hierarchy* that spans two extremes of a small high-speed memory subsystem that must interface with a large low-speed memory subsystem. Design solutions in both hierarchies rely on locality of reference. It is no accident that designs in both fields have common issues and solutions. Another indication of the universality of these principles is the hash table data structure in software. In Figure 12.28, you should have recognized the mapping from main memory to cache as essentially a hash function. Set-associative caches even look like hash tables where you resolve collisions by chaining, albeit with an upper limit on the length of the chain.

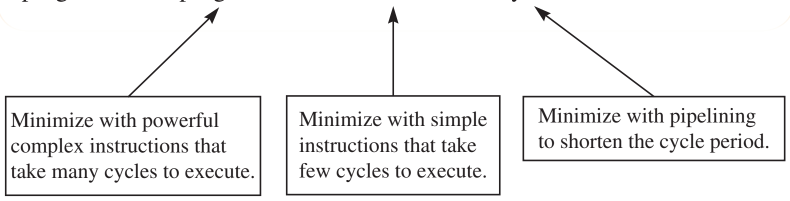
Memory hierarchies

The System Performance Equation

The ultimate performance metric of a machine is how fast it executes. FIGURE 12.33, the system performance equation, shows that the time it takes to execute a program is the product of three factors. The word *instruction* in

FIGURE 12.33
The system performance equation for a von Neumann machine.

$$\frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}$$


Minimize with powerful complex instructions that take many cycles to execute.
Minimize with simple instructions that take few cycles to execute.
Minimize with pipelining to shorten the cycle period.

the equation means Level-ISA3 instruction, and the word *cycle* means the von Neumann cycle.

There is a relationship between the first two factors that is not shared by the third. A decrease in the first factor usually leads to an increase in the second factor, and vice versa. That is, if you design your ISA3 machine in such a way that you decrease the number of instructions it takes to execute a given program, you usually must pay the price of increasing the number of cycles it takes to execute each instruction. Conversely, if you design your ISA3 machine so that each instruction takes as few cycles as possible, you usually must make the instructions so simple that it takes more of them to execute a given program.

The third factor in the equation is based on parallelism and comes into play by reorganizing the control section so that more subsystems on the integrated circuit can operate concurrently. There is no tradeoff between it and either of the first two factors. You can introduce pipelining in the design to decrease the time per cycle, and the time per program will decrease, whether you have chosen to minimize the first factor at the expense of the second, or the second at the expense of the first.

RISC Versus CISC

Early in the history of computing, designers concentrated on the first factor. That approach was motivated in part by the high cost of main memory. Programs needed to be as small as possible just so they could fit in the available memory. Instruction sets and addressing modes were designed to make it easy for compilers to translate from HOL6 to Asmb5. Pep/9 is an example of such a design. In this case, the motivation is for pedagogical reasons and not for reasons of limited main memory. The assembly language is designed to teach the principles of translation between levels of abstraction typical in computer systems. Those principles are easiest to learn when the translation process is straightforward.

By the early 1980s, the computing landscape had changed. Hardware was getting cheaper, and memory subsystems were getting bigger. Some designers, notably John Cocke at IBM, David Patterson at UC Berkeley, and John Hennessy at Stanford, began to promote designs based on decreasing the second factor at the expense of increasing the first. Their designs were characterized by a much smaller number of ISA3 instructions and fewer addressing modes. The moniker for their designs was *reduced instruction set computer* (RISC, pronounced *risk*). In contrast, the older designs began to be called *complex instruction set computers* (CISCs, pronounced *sisks*).

RISC and CISC

The following is a list of RISC design principles. The first principle is the primary one from which the others necessarily follow.

1. Each ISA3 instruction except for load and store executes in one cycle.
2. The control section has no microcode.

3. Every ISA3 instruction is the same length.
4. There are only a few simple addressing modes.
5. All arithmetic and logic operations take place entirely in the CPU.
6. Data sections are designed for deep pipelining.

Pep/9, although it is a tiny processor, is a complex instruction set computer.

The first RISC design principle

The first RISC design principle is that each ISA3 instruction except for load and store executes in one cycle. The second factor in the system performance equations shows that you can minimize the execution time of a program by minimizing the number of cycles per instruction. The RISC design principle takes this optimization technique to the extreme by making the second factor exactly one. RISC machines execute with one cycle per ISA3 instruction, even including the fetch part and the increment part of the von Neumann cycle.

The second RISC design principle

The second RISC design principle is that the control section has no microcode. Figure 12.15 shows the control section of Pep/9, a CISC machine. Because a Pep/9 ISA3 instruction takes many cycles to execute, its control section must deliver a sequence of control signals to the data section. In a RISC machine, the Mc2 level is missing altogether because a sequence of control signals is no longer necessary to drive the data section.

The third RISC design principle

The third RISC design principle is that every ISA3 instruction is the same length. The fetch part of the Pep/9 von Neumann cycle fetches the instruction specifier, determines if the instruction is unary, and if not, fetches the operand specifier. The fetch process must, therefore, take more than one cycle. If each instruction must execute in one cycle, then every instruction must be the same length.

The fourth RISC design principle

The fourth RISC design principle is that there are only a few simple addressing modes. Pep/9 has the eight addressing modes of Figure 12.8. Of these eight modes, a RISC machine could not have the complex modes indirect, stack-relative deferred, or stack-deferred indexed because each one would take multiple memory fetches. Operand address computations that would be provided at the Mc2 level in hardware must be provided by the compiler in software. For example, if an HOL6 program has a data structure that needs stack-deferred indexed addressing, the compiler must generate the code to compute Mem[SP + OprndSped] + X] at the ISA3 level because it is not provided by the hardware at the Mc2 level. Consequently, the application code for a RISC machine is larger than the equivalent application code for a CISC machine.

The fifth RISC design principle

The fifth RISC design principle is that all arithmetic and logic operations take place entirely in the CPU. Consider the Pep/9 `ADDA` instruction with direct addressing. It adds Mem[OprndSpec] to the accumulator and puts the result in the accumulator. The addition operation requires a memory access, which consumes many memory access cycles. To minimize memory accesses, RISC machines have a large bank of general-purpose registers instead of a small bank of special-purpose registers. The processor performs arithmetic and logic operations with a register addressing mode.

For example, the Pep/9 translation of

```
x = y + z;
```

into assembly language is

Code for an accumulator machine

```
LDWA y,d
ADDA z,d
STWA x,d
```

where `x`, `y`, and `z` are global variables stored in a fixed location of memory. All three assembly language statements require a memory access. The preceding translation pattern makes Pep/9 a so-called *accumulator machine*. In a RISC machine, the equivalent code would be

Code for a load/store machine

```
LDW r1,y,d   ;Load y into register r1
LDW r2,z,d   ;Load z into register r2
ADD r3,r1,r2 ;Add r1 + r2 and put the sum in r3
STW r3,x,d   ;Store register r3 into x
```

Registers r1, r2, and r3 are three registers from a large set of general-purpose registers. The `ADD` instruction uses the register addressing mode and does not access memory. The preceding translation pattern makes RISC processors so-called *load/store machines*.

It may seem like the execution time would increase on the RISC machine with the above example because there are still three memory accesses plus the addition. In the long run, memory accesses are minimized over the CISC machine because the large register bank acts like a cache for recently used variables. The compiler maintains an internal symbol table for the set of recently used variables. The first time a variable is accessed, the compiler assigns it to one of the registers. When arithmetic and logic operations are performed in a function with the variables, only the register copies are used, with no memory accesses. The generated code stores variable values to memory only when necessary—for example, when a `return` statement executes.

The sixth RISC design principle

The sixth RISC design principle is that data sections are designed for deep pipelining. This principle is not unique to RISC processors, as CISC processors also use pipelining to decrease the cycle period. However, the simplicity of the instruction set and the small number of simple addressing modes allow RISC processors to more efficiently implement deep pipelining.

Microcode in x86 Systems

The x86 processors built by Intel and AMD are the most widely used CISC processors on the market. The details of their internal designs, including the design of their Mc2 level microcode control units, are proprietary and considered to be company secrets. The companies even encrypt the microcode on the chip in an effort to keep it hidden. Through limited publications and reverse engineering, however, we do have a partial picture of the designs.

Modern x86 chips use microcode only for those instructions in their ISA3 instruction set that are complex. Simple instructions that require just a few cycles are implemented with finite-state machines directly in the hardware that output RISC-like operations (ROPs) of a fixed length. More complex instructions use a full Mc2 layer of abstraction.

FIGURE 12.34 shows the microcode sequence for an AMD chip that implements the complex `movsb` instruction, whose mnemonic stands for *move string byte*. This one ISA3 instruction copies a string of ASCII bytes from one memory location to another. It uses two index registers in the x86 register bank—ESI, the index register for the source, and EDI, the index register for the destination—and one register ECX for the byte count. The CPU also has a direction flag that the assembly language programmer can set and clear. To use the `movsb` instruction, the programmer first puts the address of the source string in ESI, puts the address of the destination in EDI, puts the byte count in ECX, and sets or clears the direction flag, depending on whether the copy should be left to right or right to left. Then, the single execution of `movsb` at the ISA3 level performs the entire copy using a loop at the Mc2 microcode level.

The microcode in Figure 12.34 is more symbolic than Pep/9 microcode. For example, the microcode instruction at cycle 2 sets the Z bit if the content of the ECX register is all zeros. It takes the OR operation of the ECX register with itself and sets the Z bit accordingly. The equivalent Pep/9 microcode sequence for testing the high-order byte of the index register and setting the Z bit accordingly is

```
2. A=2, B=2, AMux=1, ALU=7, AndZ=0; ZCk
```

In the Pep/9 version, the numeric address of the register 2 is required, while the AMD microcode version uses the symbolic name `ecx`. Pep/9 microcode requires the numeric value 7 on the ALU control line to select the OR operation, while the AMD microcode version uses the symbolic name `OR`. The Pep/9 version requires explicit control values for the multiplexer and clock pulse, while these signals are implicit in the AMD microcode. The AMD microassembler must generate these control signals and values when it translates the microcode.

FIGURE 12.34
AMD microcode for the `movsb` instruction.

```
1. LDDF                ;load direction flag to latch in functional unit
2. OR ecx, ecx         ;test if ECX is zero
3. JZ end              ;terminate string move if ECX is zero
loop:
4. MOVFM+ tmp0, [esi]  ;move to tmp data from source and inc/dec ESI
5. MOVTM+ [edi], tmp0  ;move the data to destination and inc/dec EDI
6. DECXJNZ loop        ;dec ECX and repeat until zero
end:
7. EXIT
```

12.3 The MIPS Machine

Since the 1980s, almost all newly designed CPUs have been RISC machines. Two prominent RISC machines are the ARM chip, which dominates the mobile phone and tablet markets, and the MIPS chip, which is based on the Stanford design and is used in servers and Nintendo game consoles. *MIPS* is an acronym for *microprocessor without interlocked pipeline stages*. There is one CISC design that continues to dominate the desktop and laptop market—namely, Intel's x86-64 family of processors, the latest chip family listed in Figure 12.26. Its continued dominance is due in large part to the compatibility that each family has maintained with its predecessor. It is expensive to migrate applications and operating systems to chips with different ISA3 instructions and addressing modes. Furthermore, CISC designers were able to adopt the RISC philosophy by creating a level of abstraction with a RISC core, the details of which are hidden at a low level, that implemented the CISC machine at Level ISA3.

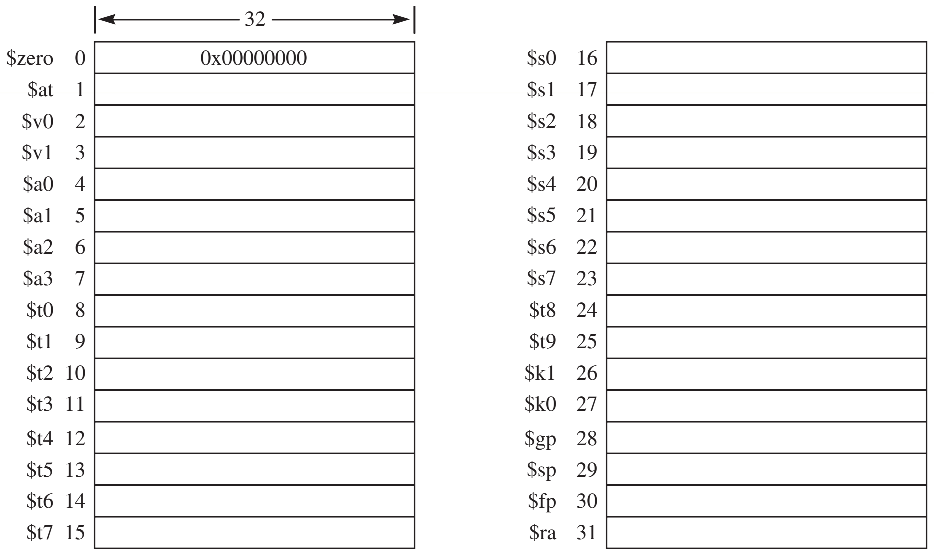
The Register Set

The MIPS machine is a classic example of a commercially produced load/store machine. The 32-bit version has 32 32-bit registers in its CPU. The 64-bit version of MIPS also has 32 registers, but each register is 64 bits wide. FIGURE 12.35 is a drawing to scale of the registers in the 32-bit MIPS CPU compared to those in Pep/9.

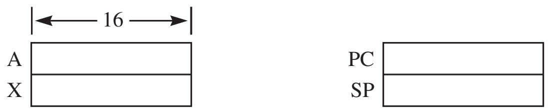
Each register has a special assembler designation that begins with a dollar sign. $zero is a constant zero register similar to register 22 in Figure 12.2 but visible at Level ISA3. $v0 and $v1 are for values returned by a subroutine, and $a0 through $a3 are for arguments to a subroutine, similar to the calling protocol for operator `malloc` in Section 6.5. Registers that begin with $t are temporary, not preserved across a function call; and those that begin with $s are saved registers, preserved across a function call. The $k registers are reserved for the operating system kernel. $gp is the global pointer, $sp is the stack pointer, $fp is the frame pointer, and $ra is the return address.

Pep/9 is a tiny machine compared to most microprocessors. Figure 12.25 shows that there are 16 times more bits in the MIPS CPU registers than the Pep/9 CPU registers. And that does not count another set of floating-point registers that MIPS has and Pep/9 does not. Even with this big mismatch in size, in two respects MIPS is simpler than Pep/9—it has fewer addressing modes, and its instructions are all the same length, namely, four bytes. For enhanced performance, the memory alignment issue forces

FIGURE 12.35
Comparison of the 32-bit MIPS and Pep/9 CPU registers.



(a) MIPS registers.



(b) Pep/9 registers.

the first byte of each instruction to be stored at an address evenly divisible by four. FIGURE 12.36 shows the von Neumann cycle for the MIPS machine. There is no `if` statement to determine the size of the instruction.

Figure 12.36 shows the von Neumann cycle for MIPS as an endless loop, which is more realistic than the cycle for Pep/9. Real machines have no `STOP` instruction because the operating system continues to execute when an application terminates.

The Addressing Modes

In contrast to Pep/9, which has eight addressing modes, MIPS has the five addressing modes of FIGURE 12.37. Each of the five addressing modes uses one of the three instruction types—either I-type, R-type, or J-type.

FIGURE 12.36
A pseudocode description of the MIPS von Neumann execution cycle.

```
do {
    Fetch the instruction at the address in PC
    PC ← PC + 4
    Decode the instruction specifier
    Execute the instruction fetched
}
while (true)
```

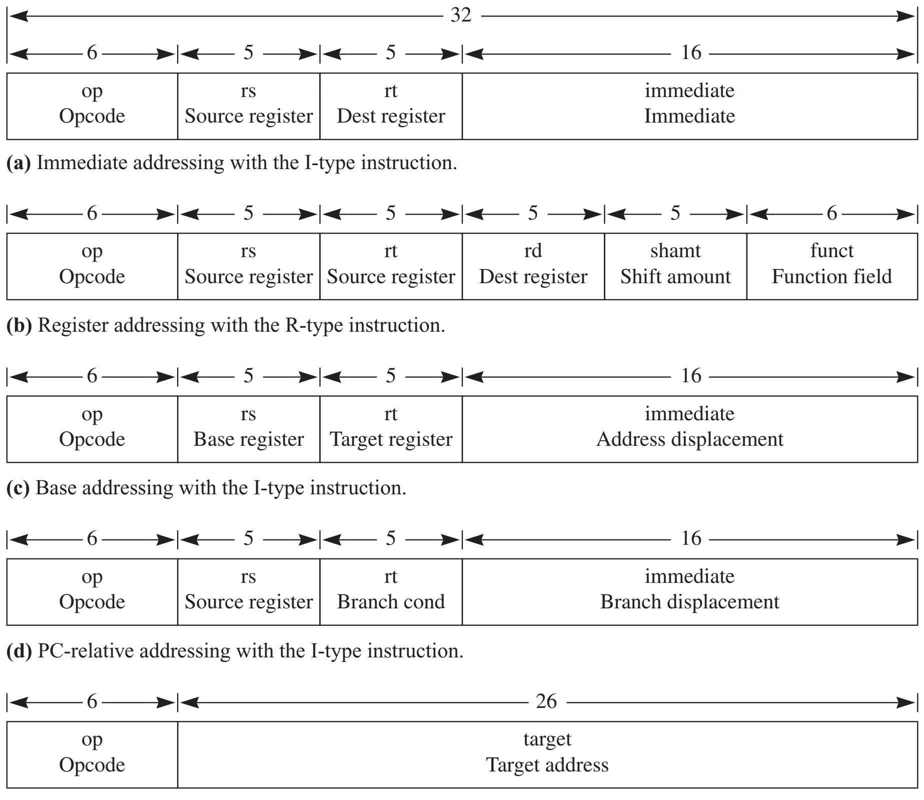
FIGURE 12.37
The MIPS addressing modes.

Addressing Mode	Instruction Type	Operands		
		Destination	Source	Source
Immediate	I-type	Reg[rt]	Reg[rs]	SE(im)
Register	R-type	Reg[rd]	Reg[rs]	Reg[rt]
Base with load	I-type	Reg[rt]	Mem[Reg[rb] + SE(im)]	
Base with store	I-type	Mem[Reg[rb] + SE(im)]	Reg[rt]	
PC-relative	I-type	PC	PC + 4	SE(im × 4)
Pseudodirect	J-type	PC	(PC + 4)⟨0..3⟩ : (ta × 4)	

FIGURE 12.38 shows the instruction format for each addressing mode. Instructions with the addressing modes immediate, base, and PC-relative are I-type instructions. Instructions with register addressing are R-type, and instructions with pseudodirect addressing are J-type instructions. A MIPS instruction always consists of a six-bit opcode to specify the instruction and one or more operand specifiers. The rs field, when present, is always at bit location 6..10, the rt field is always at 11..15, and rd is always at 16..20. This chapter specifies bit locations starting with the leftmost bit as 0 and numbering from left to right to be consistent with Pep/9 notation. Standard MIPS notation is to start with the rightmost bit as 0 and to number from right to left.

MIPS bit numbering notation

FIGURE 12.38
The MIPS instruction formats corresponding to the addressing modes.



(a) Immediate addressing with the I-type instruction.

(b) Register addressing with the R-type instruction.

(c) Base addressing with the I-type instruction.

(d) PC-relative addressing with the I-type instruction.

(e) Pseudodirect addressing with the J-type instruction.

Because there are 32 registers in the register bank in Figure 12.35(a), and 2^5 is 32, it takes five bits to access one of them. The designations rs, rt, and rd are standard MIPS notations for five-bit register fields in an instruction. Figure 12.37 shows that rs is always a source register, and rd is always a destination register; but rt, which stands for *target register*, can be either a source or a destination register. The notation *Reg* in Figure 12.37 stands for *register* and is analogous to *Mem*, which stands for *memory*. Reg[r] indicates the content of the register r. For example, if an instruction with immediate addressing executes, Figure 12.37 shows Reg[rt] as the destination operand. If the five-bit rt field in Figure 12.38(a) has the value 10011 (bin), which is 19 (dec), then the destination register \$s3 will contain the result of the operation because \$s3 is register 19 in Figure 12.35.

Sign extension

The function SE(im) is sign extension of the immediate operand. If the sign bit of the 16-bit operand is 0, indicating a positive quantity, the 16-bit operand is expanded to 32 bits by prepending 16 additional zeros. If the sign bit of the 16-bit quantity is 1, the operand is expanded by prepending 16 additional ones. For example, the 16-bit quantity 7C9B (hex) expands to the 32-bit quantity 00007C9B, and 8C9B expands to FFFF8C9B. Sign extension does not change the decimal value of the operand.

Immediate addressing

Figure 12.38(a) shows the instruction format for immediate addressing. The immediate operand cannot contain 32 bits because it must be part of a 32-bit instruction. It is sign extended to 32 bits and combined with the other source operand Reg[rs], and the result is put in Reg[rt]. With immediate addressing, rt is a destination register.

Register addressing

Figure 12.38(b) shows the instruction format for register addressing. All the arithmetic and logic operations use register addressing with two source registers and one destination register. For example, one instruction can add the values of two different variables and give their sum to a third. The function field is, in effect, an expanded opcode. If the opcode field is 000000 (bin), the instruction uses register addressing and the function field determines the operation. For example, if the function field is 100000, the operation is addition, and if it is 100010, the operation is subtraction. The Pep/9 shift instructions shift only one bit location either to the left or to the right. The MIPS processor can shift a register multiple times in one cycle. The shift amount field specifies how many bits to shift. With register addressing, rt is a source register.

Base addressing

Figure 12.38(c) shows the instruction format for base addressing. Of all the addressing modes, this is the only one that accesses main memory. The rs field designates the base register field and is written as *rb* in Figure 12.37. The instruction computes the memory address by adding the sign-extended immediate field to the content of the base register. For load instructions, memory is the source and register rt is the destination. For store instructions, register rt is the source and memory is the destination.

PC-relative addressing

Figure 12.38(d) shows the instruction format for PC-relative addressing. The conditional branch instructions use this addressing mode to change the program counter. If the condition of the branch is satisfied, the program counter is changed with the following specification:

$$\text{PC} \leftarrow (\text{PC} + 4) + \text{SE}(\text{im} \times 4)$$

MIPS instructions are 32 bits in length, so they are aligned on 4-byte boundaries when stored in main memory. Because you would never access an instruction that is not aligned, the immediate operand is multiplied by 4, which is equivalent to a left shift by 2. The shifted operand is sign extended and added to the incremented value of the program counter. The quantity added to the PC is a signed 18-bit quantity—16 bits from the immediate operand, then shifted left 2 more bits. The range of values for an 18-bit signed integer is -2^{17} to $2^{17} - 1$. Therefore, with PC-relative addressing, you can branch backward and forward from the current PC by 128 KiB. To branch beyond this limit requires the use of another addressing mode. Most conditional branches are within this limit. For example, an `endFor` symbol is not far from its corresponding `for` symbol because the body of the loop is usually smaller than 128 KiB.

Figure 12.38(d) shows that the rt field specifies the branch condition. The rt field with PC-relative addressing is another instance of an expanded opcode. If the six-bit opcode field is 000001, then the instruction is a conditional branch, and the rt field specifies the condition. For example, if the rt field is 00000, the branch is on rs less than zero, but if the rt field is 00001, the branch is on rs greater than or equal to zero. Some conditional branch instructions have a six-bit opcode field different from 000001.

All processors have some form of PC-relative addressing that is used by branch instructions. Chapter 6 gives examples of PC-relative addressing in the x86 architecture. Pep/9 does not need PC-relative addressing because every application program is loaded into memory starting at address 0000. The assembler can know the absolute memory address of every global variable and instruction of the program and calculate the value of each symbol accordingly. With Pep/9, a branch to `endFor` uses the symbol as the absolute address of the instruction.

In practice, operating systems manage multiple processes simultaneously, with the code for the processes scattered at various locations throughout the memory map. It is impossible for the assembler to know at translation time where in memory the program will be loaded and executed, so direct addressing is rarely useful. When a MIPS assembler encounters a branch to `endFor,` it counts the number of bytes between the use of the symbol and the definition of the symbol, which it generates as the immediate value in the PC-relative branch. Regardless of where the program is loaded in memory, the number of bytes between the use of the symbol and its definition is unchanged, and the code will execute correctly. The same concept is used in Pep/9 stack-relative addressing, where the memory location of a local variable is specified by its offset from the stack pointer. With PC-relative addressing, the offset is from the program counter instead of the stack pointer.

Pseudodirect addressing

Figure 12.38(e) shows the instruction format for pseudodirect addressing used with the unconditional branch instruction `j`, which stands for *jump*. The program counter is changed with the following specification.

$$PC \leftarrow (PC + 4)\langle 0 .. 3\rangle : (ta \times 4)$$

Quantity ta is the 26-bit target address, which is shifted left 2 bits, increasing its length to 28 bits. The colon in the specification is the concatenation operator. The first 4 significant bits of the incremented program counter are concatenated with the 28 bits of the shifted target address to produce a 32-bit address for the PC.

With pseudodirect addressing, you are still limited to one-sixteenth of the memory map specified by the first four bits of the program counter. Neither PC-relative nor pseudodirect addressing modes allow unrestricted access to the entire 4 GiB address space. MIPS provides the jump register instruction `jr` with base addressing to access any address in the entire address space. The program counter is changed with the following specification:

The jump register instruction

$$\text{PC} \leftarrow \text{Reg}[\text{rb}]$$

where rb is the base register in the rs field of Figure 12.38(c). It requires the programmer to compute the address and put it in the base register before executing the unconditional branch.

The Instruction Set

FIGURE 12.39 is a summary of some MIPS instructions. The first column is the mnemonic of the instruction for MIPS assembly language. Operand specifiers labeled `sssss` and `ttttt` are five-bit source register fields, those labeled `ddddd` are five-bit destination register fields, and those labeled `bbbbb` are five-bit base register fields. A field of `i` characters is an immediate operand specifier, which is sign-extended for addition and zero-extended for AND and OR. A field of `a` characters is an address operand specifier, which is a sign-extended offset in an address calculation. Operand specifiers labeled `hhhhh` are five-bit shift amounts for the shift instructions.

Following are some examples of C code fragments with their translation into MIPS assembly language. The MIPS processor uses its bank of 32 registers as a cache for variables. The first time a variable is accessed, the compiler associates a register with that value. Later accesses use the variable copy in the register instead of accessing memory again. With Pep/9, global variables are stored at a fixed absolute address in memory. With MIPS, global variables are accessed relative to \$gp, the global pointer.

Example 12.2 Figure 5.27 declares three global variables, `exam1`, `exam2` and `score`, as well as a constant `bonus`, which equates to 10. The C compiler might install the three globals with offsets 0, 4, and 8 from \$gp and associate them with registers \$s1, \$s2, and \$s3 in the register bank. It translates the Cstatement

```
score = (exam1 + exam2) / 2 + bonus;
```

FIGURE 12.39
A few instructions from the MIPS instruction set.

Mnemonic	Meaning	Binary Instruction Encoding							
add	Add	0000	00ss	ssst	tttt	dddd	d000	0010	0000
addi	Add immediate	0010	00ss	sssd	dddd	iiii	iiii	iiii	iiii
sub	Subtract	0000	00ss	ssst	tttt	dddd	d000	0010	0010
and	Bitwise AND	0000	00ss	ssst	tttt	dddd	d000	0010	0100
andi	Bitwise AND immediate	0011	00ss	sssd	dddd	iiii	iiii	iiii	iiii
or	Bitwise OR	0000	00ss	ssst	tttt	dddd	d000	0010	0101
ori	Bitwise OR immediate	0011	01ss	sssd	dddd	iiii	iiii	iiii	iiii
sll	Shift left logical	0000	0000	000t	tttt	dddd	dhhh	hh00	0000
sra	Shift right arithmetic	0000	0000	000t	tttt	dddd	dhhh	hh00	0011
srl	Shift right logical	0000	0000	000t	tttt	dddd	dhhh	hh00	0010
lb	Load byte	1000	00bb	bbbd	dddd	aaaa	aaaa	aaaa	aaaa
lw	Load word	1000	11bb	bbbd	dddd	aaaa	aaaa	aaaa	aaaa
lui	Load upper immediate	0011	1100	000d	dddd	iiii	iiii	iiii	iiii
sb	Store byte	1010	00bb	bbbt	tttt	aaaa	aaaa	aaaa	aaaa
sw	Store word	1010	11bb	bbbt	tttt	aaaa	aaaa	aaaa	aaaa
beq	Branch if equal to	0001	00ss	ssst	tttt	aaaa	aaaa	aaaa	aaaa
bgez	Branch if greater than or equal to zero	0000	01ss	sss0	0001	aaaa	aaaa	aaaa	aaaa
bgtz	Branch if greater than zero	0001	11ss	sss0	0000	aaaa	aaaa	aaaa	aaaa
blez	Branch if less than or equal to zero	0001	10ss	sss0	0000	aaaa	aaaa	aaaa	aaaa
bltz	Branch if less than zero	0000	01ss	sss0	0000	aaaa	aaaa	aaaa	aaaa
bne	Branch if not equal to	0001	01ss	ssst	tttt	aaaa	aaaa	aaaa	aaaa
j	Jump	0000	10aa	aaaa	aaaa	aaaa	aaaa	aaaa	aaaa
jr	Jump register	0000	00bb	bbb0	0000	0000	0000	0000	1000

to MIPS assembly language as

```
lw $s1,0($gp)    # Load exam1 into register $s1
lw $s2,4($gp)    # Load exam2 into register $s2
add $s3,$s1,$s2 # Register $s3 gets exam1 + exam2
sra $s3,$s3,1    # Shift right register $s3 one bit
addi $s3,$s3,10 # Register $s3 gets $s3 + 10
sw $s3,8($gp)    # score gets $s3
```

Comments begin with the # character. In MIPS assembly language, the first argument after the mnemonic is usually the destination. For example, in the first `lw` instruction, \$s1 is the destination register, and in the `add` instruction, \$s3 is the destination register. The `sw` instruction is an exception to this general rule. The RTL specification of the `lw` instruction is

The `lw` instruction

$$\text{Reg}[\text{rt}] \leftarrow \text{Mem}[\text{Reg}[\text{rb}] + \text{SE}(\text{im})]$$

and the specification of the `sw` instruction is

The `sw` instruction

$$\text{Mem}[\text{Reg}[\text{rb}] + \text{SE}(\text{im})] \leftarrow \text{Reg}[\text{rt}]$$

In both of these instructions, the base register \$gp is in parentheses preceded by the immediate operand in decimal notation. The RTL specification of the `add` instruction is

The `add` instruction

$$\text{Reg}[\text{rd}] \leftarrow \text{Reg}[\text{rs}] + \text{Reg}[\text{rt}]$$

The destination register rd is \$s3, the source register rs is \$s1, and the target register rt is \$s2. The mnemonic `addi` stands for *add immediate* and obviously uses immediate addressing. The machine language translation of these instructions is

```
100011 11100 10001 0000000000000000
100011 11100 10010 0000000000000100
000000 10001 10010 10011 00000100000
000000 00000 10011 10011 00001 000011
001000 10011 10011 0000000000001010
101011 11100 10011 0000000000001000
```

For base addressing, Figure 12.38(c) shows the base register field next to the opcode field. The above `lw` and `sw` instructions have \$gp as the base register. Figure 12.35 shows \$gp as register 28 (dec) = 11100 (bin), which is the register field next to the opcode field in the preceding machine code. For the `add` instruction with register addressing, the fields in the machine code are in the order rs, rt, rd, consistent with Figure 12.38(b), while the fields in the assembly code are in the order rd, rs, rt, with the destination field first. In Figure 12.35, you can identify the binary fields in the preceding machine code for the global variables as follows:

- $s1 is register 17 (dec) = 10001 (bin).
- $s2 is register 18 (dec) = 10010 (bin).
- $s3 is register 19 (dec) = 10011 (bin). ■

It is not feasible to store an entire array in the register bank because most arrays have too many elements to fit the available registers. To process arrays, the C compiler associates a register with the address of the first element of the array and uses base addressing to access an element of the array. Pep/9 accesses the element of an array with indexed addressing, for which the operand is specified as

Oprnd = Mem[OprndSpec + X]

A significant difference between Pep/9 and MIPS is the use of the register to access an array element. In Pep/9, the register X contains the value of the index. In the MIPS machine, the register rb contains the address of the first element of the array—in other words, the base of the array. That is why register rb is called the *base register* and this addressing mode is called *base addressing*.

Example 12.3 Suppose the C compiler associates $s1 with array `a`, $s2 with variable `g`, and $s3 with array `b`. It translates the statement

```
a[2] = g + b[3];
```

to MIPS assembly language as

```
lw $t0,12($s3)   # Register $t0 gets b[3]
add $t0,$s2,$t0  # Register $t0 gets g + b[3]
sw $t0,8($s1)    # a[2] gets g + b[3]
```

The load instruction has 12 for the address field, because it is accessing `b[3]`, each word is four bytes, and 3 × 4 = 12. Similarly, the store instruction has 8 in the address field because of the index value in `a[2]`. The machine language translation of these instructions is

```
100011 10011 01000 0000000000001100
000000 10010 01000 01000 00000 100000
101011 10001 01000 0000000000001000
```

Figure 12.35 shows that $t0 is register 8 (dec) = 01000 (bin), $s3 is register 19 (dec) = 10011 (bin), $s2 is register 18 (dec) = 10010 (bin), and $s1 is register 17 (dec) = 10001 (bin). ■

The situation is a bit more complicated if you want to access the element of an array whose index is a variable. MIPS has no index register that does what the index register in Pep/9 does. Consequently, the compiler must generate code to add the index value to the address of the first element of the array to get the address of the element referenced. In Pep/9, this addition is done automatically at Level Mc2 with indexed addressing. But, the design philosophy of load/store machines is to have few addressing modes even at the expense of needing more statements in the program.

In Pep/9, words are two bytes, and so the index must be shifted left once to multiply it by 2. In MIPS, words are four bytes, and so the index must be shifted left twice to multiply it by 4. The MIPS instruction `sll`, for *shift left logical*, uses the shamt field in Figure 12.38(b) to specify the amount of the shift.

A word is four bytes in MIPS.

Example 12.4 The MIPS assembly language statement to shift the content of \$s0 seven bits to the left and put the result in \$t2 is

```
sll $t2,$s0,7
```

The machine language translation is

```
000000 00000 10000 01010 00111 000000
```

The first field is the opcode. The second field is not used by this instruction and is set to all zeros. The third is the rt field, which indicates \$s0, register 16 (dec) = 10000. The fourth is the rd field, which indicates \$t2, register 10 (dec) = 01010 (bin). The fifth is the shamt field, which indicates the shift amount. The last is the funct field used with the opcode to indicate the `sll` instruction. ■

Example 12.5 Assuming that the C compiler associates \$s0 with variable `i`, \$s1 with array `a`, and \$s2 with variable `g`, it translates the statement

```
g = a[i];
```

into MIPS assembly language as follows.

```
sll $t0,$s0,2    # $t0 gets $s0 times 4
add $t0,$s1,$t0  # $t0 gets the address of a[i]
lw $s2,0($t0)    # $s2 gets a[i]
```

Note the 0 in the address field of the load instruction. ■

Like Pep/9, the MIPS machine has a stack register \$sp in its register bank. Unlike Pep/9, there is no special `ADDSP` instruction because the `addi` instruction can access any of the 32 registers in the register bank, including \$sp. To allocate storage on the run-time stack, execute `addi` with a negative immediate value and with \$sp as both the source and destination register.

Example 12.6 To allocate four bytes of storage on the run-time stack, you would execute

```
addi $sp,$sp,-4 # $sp <- $sp - 4
```

where –4 is not an address but the immediate operand. The machine language translation is

```
001000 11101 11101 1111111111111100
```

where \$sp is register 29. ■

You may have noticed a limitation with the `addi` instruction. The constant field is only 16 bits wide, but MIPS is a 32-bit machine. You should be able to add constants with 32-bit precision using immediate addressing. Here is another example of the RISC architecture philosophy. The goal is simple instructions with few addressing modes. The Pep/9 design permits instructions with different widths—that is, both unary and nonunary instructions. Figure 12.4 shows how this decision complicates the fetch part of the von Neumann cycle. The hardware must fetch the instruction specifier, then decode it to determine whether to fetch an operand specifier. This complexity is completely counter to the load/store philosophy, which demands simple instructions that can be decoded quickly. The simplicity goal demands that all instructions be the same length.

But if all instructions are 32 bits wide, how could one instruction possibly contain a 32-bit immediate constant? There would be no room in the instruction format for the opcode. Here is where decreasing the second factor in Figure 12.33 at the expense of the first factor comes into play. The solution to the problem of a 32-bit immediate constant is to require the execution of two instructions. For this job, MIPS provides `lui`, which stands for *load upper immediate*. It sets the high-order 16 bits of a register to the immediate operand and the low-order bits to all zeros. A second instruction is required to set the low-order 16 bits, usually the OR immediate instruction `ori`.

Example 12.7 Assuming that the compiler associates register \$s2 with variable `g`, it translates the C statement

```
g = 491521;
```

to MIPS assembly language as

```
lui $s2,0x0007
ori $s2,$s2,0x8001
```

The decimal number 491521 requires more than 16 bits in binary, and 491521 (dec) = 0007 8001 (hex).

MIPS Computer Organization

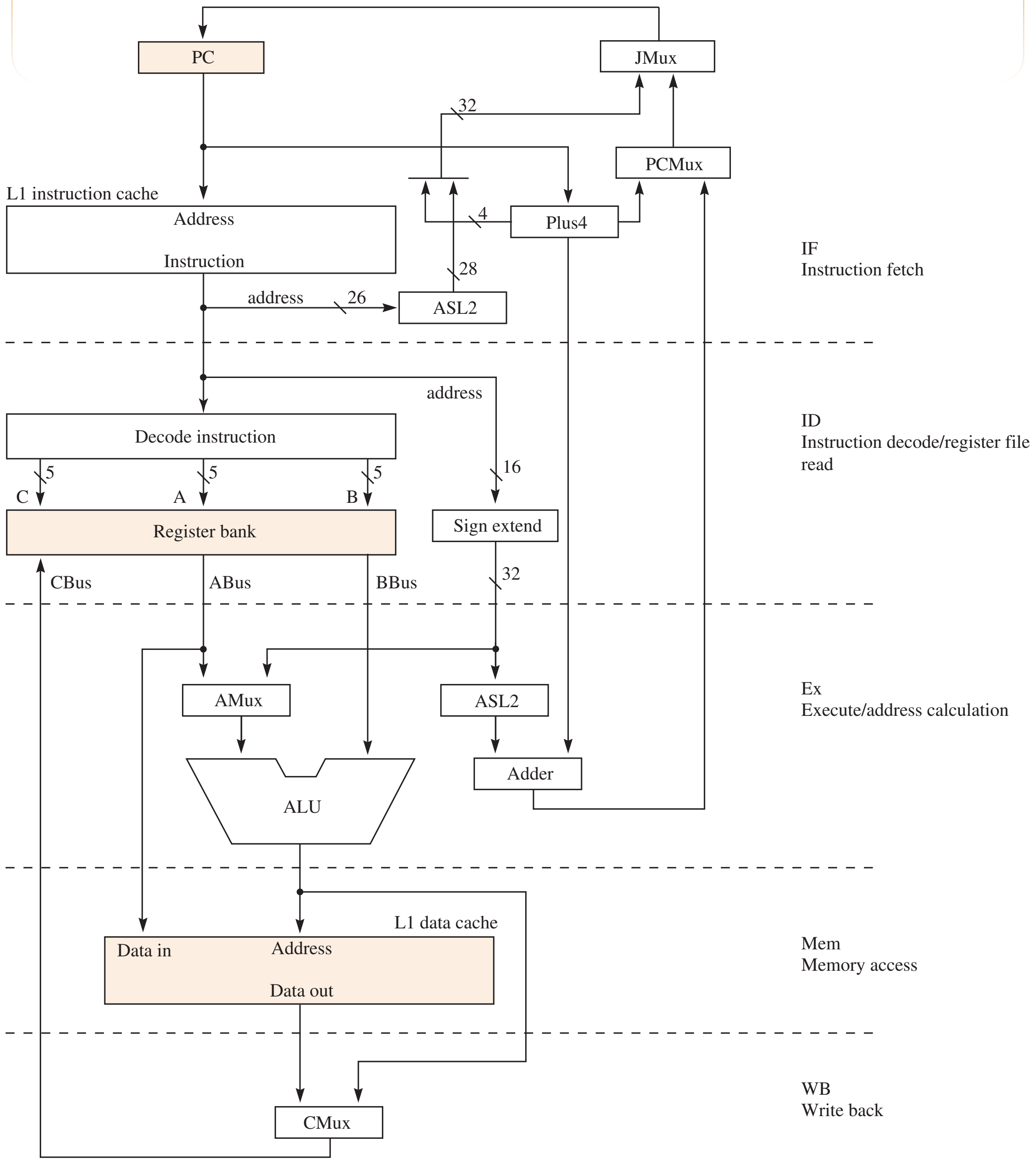
FIGURE 12.40 shows the data section of the MIPS CPU. The box labeled *Register bank* is a two-port bank of the 32-bit registers in Figure 12.35(a). The data section has the same basic organization as the data section of Pep/9 in Figure 12.2 with an ABus and BBus that feed through the primary ALU, whose output eventually goes over the CBus to the bank of CPU registers. A significant difference in the organization is the L1 instruction and data caches in the path. As most cache memories exhibit a hit rate above 90%, we can assume that memory reads and writes operate at full CPU speed. There are no MemRead or MemWrite delays, except on those rare occasions when you get a cache miss. The ABus, BBus, and CBus and the buses into and out of the ALU; the L1 data cache; and the JMux, PCMux, and CMux multiplexers are all 32 bits.

Unlike Pep/9, the program counter is not one of the general registers in the register bank. Instead, it is in effect the memory address register to the L1 instruction cache. There is no separate MAR other than the program counter itself. Similarly, the output of the ALU is, in effect, the memory address register to the L1 data cache. Likewise, there is no separate MDR. Instead, the ABus is, in effect, the memory data register to the L1 data cache.

The CPU does not write to the instruction cache. It simply requests a read from the address specified by PC. The cache subsystem delivers the instruction from the cache immediately on a hit. On a miss, it delays the CPU and eventually reads the instruction from the L2 cache, writes it to the L1 cache, and notifies the CPU that it can continue. Because the CPU never writes to the instruction cache, it treats the cache as if it were a combinational circuit. That is why the instruction cache is not shaded in Figure 12.40.

The primary design goal of a RISC machine is for every ISA3 instruction to execute in one von Neumann cycle, including the fetch and increment part of the cycle. The next value of the program counter must be computed concurrently with the computation of the execute part of the cycle. The data section contains a set of specialized combinational circuits for the program counter that operate concurrently with the ALU. The ASL2 units output an arithmetic shift left by 2 bits. The Sign extend unit outputs a 32-bit sign extension from a 16-bit input. The Plus4 unit is hardwired to add 4 to a 32-bit quantity. Its bottom and right outputs are the 32-bit sum, but its left output is the 4 most significant bits of the sum. Because all instructions are exactly four bytes long, the increment part of the von Neumann cycle is simpler than that of Pep/9 and can be implemented with these specialized hardware units without tying up the main ALU or consuming cycles.

FIGURE 12.40
The MIPS data section. Sequential circuits are shaded.



The box labeled *Decode instruction* in Figure 12.40 is a circuit whose 32 inputs are all the bits of the instruction that is executing. MIPS has no Mc2 level that emits a sequence of control signals. The Decode unit corresponds to the control section of a CISC machine and outputs the control signals for the single cycle to implement the instruction. Its bottom outputs are the three five-bit fields A, B, and C, which feed into the register bank and correspond to the A, B, and C control signals into the Pep/9 register bank of Figure 12.2.

FIGURE 12.41 hides the specialized hardware units of the previous figure and shows the other control outputs of the Decode unit. Besides the A, B, and C outputs, there are eight other control signals, as follows:

The MIPS control signals

- *JMux control*—If 0, select left input. If 1, select PCMux.
- *PCMux control*—If 0, select Plus4. If 1, select Adder.
- *AMux control*—If 0, select ABus. If 1, select Sign extend.
- *CMux control*—If 0, select Data out. If 1, select ALU.
- *ALU control*—Multi-line function select.
- *PCCk*—Clock pulse for program counter.
- *LoadCk*—Clock pulse for write to Register bank.
- *DCCk*—Clock pulse for write to L1 data cache.

The multiplexer control signals follow the Pep/9 convention for consistency. Specifically, a zero-control signal selects the left input, and a one-control signal selects the right input. Actual MIPS hardware differs from this convention. MIPS hardware documentation also differs from the naming convention for the hardware units, which is used in this text for consistency with the Pep/9 system.

PC is clocked on every cycle and drives the primary loop. The register bank and the data cache are not always clocked, depending on the instruction executed. The following examples show the control signals from the Decode unit to implement some ISA3 MIPS instructions.

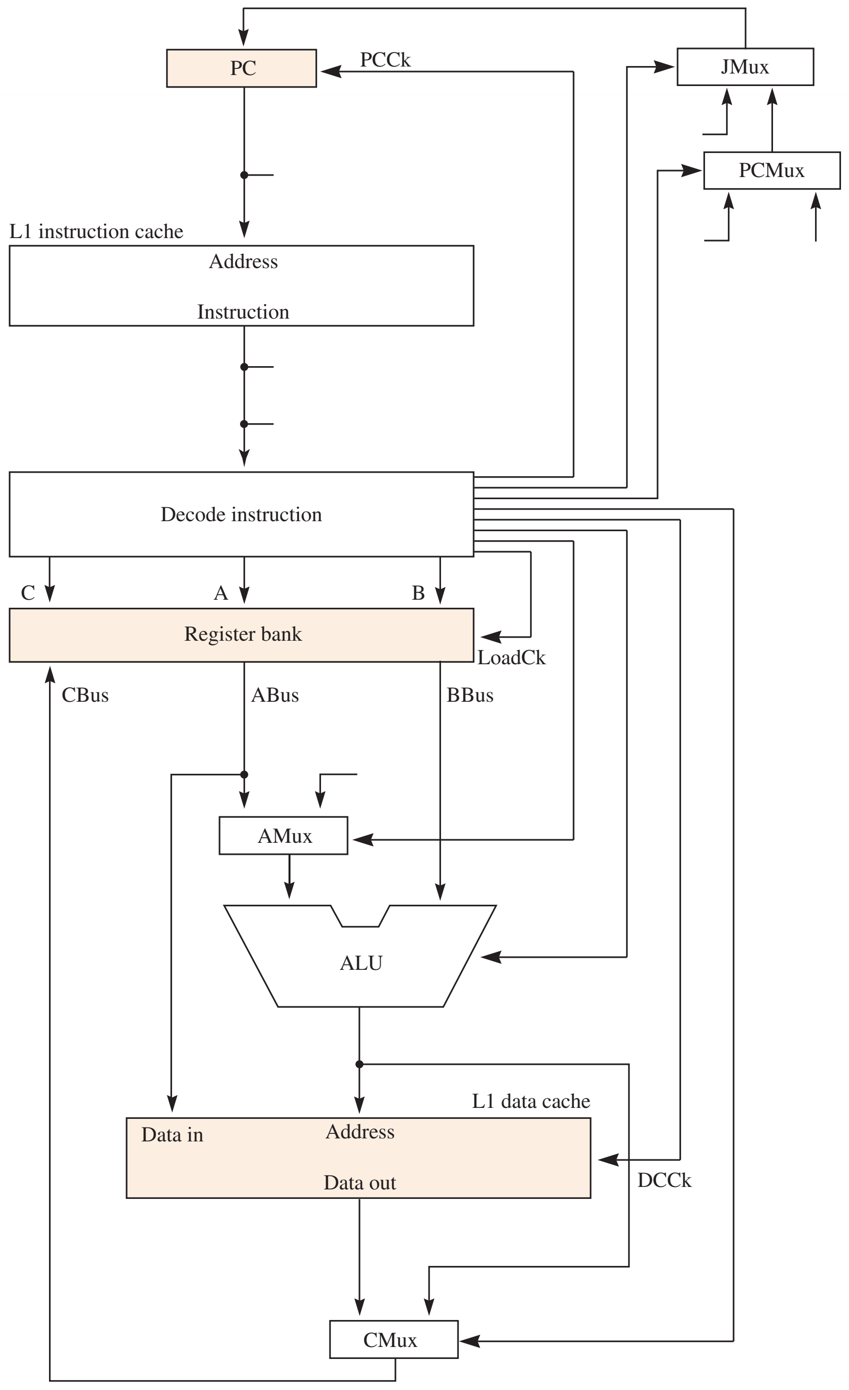
Example 12.8 The jump instruction uses pseudodirect addressing. The ASL2 box in the IF section of Figure 12.40 shifts the address two places to the left with the result concatenated with the first four (high-order) bits of the incremented PC. This is the hardware implementation with specialized hardware units of the pseudodirect addressing mode of Figure 12.37. The jump instruction requires the following control signals:

Jump instruction

```
1. JMux=0; PCCk
```

Before the cycle executes, PC has the address of the jump instruction, the 26-bit address field is presented to the ASL2 input, the first 4 bits of

FIGURE 12.41
The control signals from the Decode unit in the MIPS data section.



the incremented PC concatenated with the ASL2 output are presented to JMux, and the output of JMux is presented to PC. The clock pulse updates PC. ■

MIPS has no status bits.

Unlike Pep/9, the MIPS processor has no NZVC status bits for its conditional branch instructions. Instead, the comparison is always between two registers in the register bank, one of which might be the $zero register. To test for overflow with an arithmetic instruction, programs use a version of the instruction that generates a trap on overflow. Otherwise, they use an unsigned version, which does not trap on overflow. For example, the `add` instruction triggers a system trap on overflow. The `addu` instruction, which stands for *add unsigned*, does the same operation but does not trap on overflow.

Example 12.9 The conditional branch instructions alter the program counter in one of two ways, depending on the condition. The Decode unit compares two registers and outputs different control signals, depending on the outcome of the test. If the branch is not taken, it outputs

Branch instructions

```
1. PCMux=0, JMux=1; PCCk
```

Before the cycle executes, PC has the address of the conditional branch instruction, and the incremented value from Plus4 is presented to PCMux, passed through JMux, and clocked into PC. If the branch is taken, it outputs

```
1. PCMux=1, JMux=1; PCCk
```

Before the cycle executes, PC has the address of the conditional branch instruction *and* the 16-bit address field is presented to the ASL2 input in the Ex section, the ASL2 output and incremented PC are presented to the adder in the Ex section, the adder output is presented to PCMux, the PCMux output is presented to JMux, and the output of JMux is presented to PC. The clock pulse updates PC. ■

The store instructions are facilitated by a clever arrangement of components in the data section. ABus provides a path from the register bank directly to the data input of the L1 data cache. Furthermore, the output of the primary ALU output goes to the address lines of the data cache. Hence, the addition for the address computation of a store instruction is done by the primary ALU and not a special-purpose hardware unit. PC is updated and the data is written to the data cache simultaneously.

Example 12.10 The store word instruction `sw` has RTL specification

$$\text{Mem}[\text{Reg}[\text{rb}] + \text{SE}(\text{im})] \leftarrow \text{Reg}[\text{rt}]$$

Because it updates PC and writes to memory, the cycle requires simultaneous clock pulses PCCk and DCCk. The control signals are

Store instruction

```
1. PCMux=0, JMux=1, A=rt, AMux=1, B=rb, ALU=A plus B;
PCCk, DCCk
```

The `PCMux=0` and `JMux=1` signals simply present the incremented PC to PC. The `A=rt` signal puts the content of the rt source register on ABus, which is presented as data to the cache. The `AMux=1` signal selects the address field of the instruction as the left input to the ALU, and the `B=rb` signal puts the base register on BBus as the right input to the ALU. Selecting the addition function presents the address computation on the address lines of the data cache. ▮

The register instructions use the primary ALU for their processing but do not write to memory. Therefore, the output of the ALU has a path through CMux to the register bank. As with store instructions, PC and the register bank are updated simultaneously.

Example 12.11 The add instruction `add` has RTL specification

Reg[rd] ← Reg[rs] + Reg[rt]

Because it updates PC and writes to the register bank, the cycle requires simultaneous clock pulses PCCk and LoadCk. The control signals are

Add instruction

```
1. PCMux=0, JMux=1, A=rs, AMux=0, B=rt, ALU=A plus B,
   CMux=1, C=rd; PCCk, LoadCk
```

The `PCMux=0` and `JMux=1` signals present the incremented PC to PC. The `A=rs` signal puts the content of the rs source register on ABus, which is presented as data to the cache through AMux with the `AMux=0` signal. The `B=rt` signal puts the base register on BBus as the right input to the ALU. Selecting the addition function presents the result on CBus through CMux with the `CMux=1` signal. Signal `C=rd` addresses the register bank for the destination register rd. ▮

The control signals for the load instructions are left as an exercise at the end of the chapter.

Pipelining

The instant PC changes at the beginning of a cycle, the data must propagate from it through the combinational circuits in the following order:

Stages in the MIPS data section

1. IF: The instruction cache, the Plus4 adder, the shifter and multiplexers

2. ID: The Decode instruction box, the Register bank, the sign extend box
3. Ex: The AMux, the ASL2 shifter, the ALU, the adder
4. Mem: The data cache
5. WB: The CMux, the address decoder of the Register bank

The CPU designers must set the period of the clock long enough to allow for the data to be presented to the sequential circuits—PC, Register bank, and data cache—before clocking the data into them. **FIGURE 12.42** shows a time line of several instructions executing, one after the other. The boxes represent the propagation time of each stage.

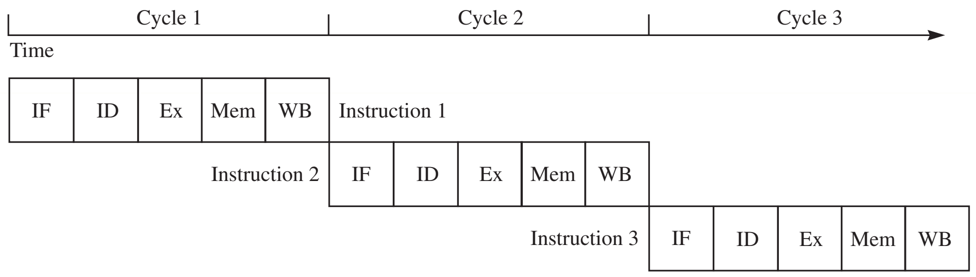
The situation in Figure 12.42 is analogous to a single craftsman who must build a piece of furniture. He has a shop with all the tools to do three things—cut the wood, assemble with clamps and glue, and paint. As there is only one craftsman, he builds his furniture in the same sequence, one piece of furniture after the other. With two other craftsmen, there are several ways to increase the number of pieces produced per day. The other craftsmen could acquire their own tools, and all three could work concurrently, cutting the wood for three pieces at the same time, assembling the three pieces at the same time, and painting them at the same time. It is true that output per time is tripled, but at the cost of all those extra tools.

The assembly line analogy

A more economical alternative is to recognize that when one person is using the clamps and glue, the tools for cutting the wood could be used for starting the next piece of furniture. Similarly, when the first piece is being painted, the second can be assembled and the wood for the third can be cut. You should recognize this organization as the basis of the factory assembly line.

The resources corresponding to the tools are the combinational circuits in the five areas listed above—instruction fetch, instruction decode/register

FIGURE 12.42
Instruction execution without pipelining.



file read, execute/address calculation, memory access, and write back. The idea of our CPU pipeline is to increase the number of cycles it takes to execute an instruction by a factor of five, but to decrease the period of each cycle by a factor of five. It might seem at first glance that there would be no net benefit. But by overlapping the execution of one instruction with the next, you get parallelism that increases the number of instructions executed per second.

Boundary registers

To implement this idea requires a modification to the data path of Figure 12.40. The results of each stage must be saved to be used as input for the next stage. At the boundary of each of the five stages, you must put a set of registers, as FIGURE 12.43 shows. At the end of each new shortened cycle, the data from every data path that crosses to the next stage below it gets stored in a boundary register.

You can get a perfect five-times decrease in the cycle time only if the propagation delays at each stage are exactly equal. In fact, they are only approximately equal. So the new cycle time must be the propagation time of the lengthiest delay of all the shortened stages. When choosing where to put the boundary registers to implement a pipeline, a designer must strive to evenly divide the stages.

FIGURE 12.44 shows how pipelining works. At startup the pipeline is empty. In cycle 1, the first instruction is fetched. In cycle 2, the second instruction is fetched concurrently with the first instruction being decoded and the register bank being read. In cycle 3, the third instruction is fetched concurrently with the second instruction being decoded and the register bank being read and the first instruction executing, and so on. The speedup comes from putting more parts of the circuit to use at the same time. Pipelining is a form of parallelism. In theory, a perfect pipeline with five stages increases by five times the number of instructions executing per second, once the pipeline is filled.

That's the good news. The bad news is that a whole host of problems can throw a monkey wrench into this rosy scenario. There are two kinds of problems, called *hazards*:

Piplining hazards

- Control hazards from unconditional and conditional branches
- Data hazards from data dependencies between instructions

Each of these hazards is due to an instruction that cannot complete the task at one stage in the pipeline because it needs the result of a previous instruction that has not finished executing. A hazard causes the instruction that cannot continue to stall, which creates a bubble in the pipeline that must be flushed out before peak performance is restored.

FIGURE 12.45(a) shows the execution of a pipeline from startup with no hazards. The second group of five boxes on the first line represents the

FIGURE 12.43
The MIPS data section with pipelining.

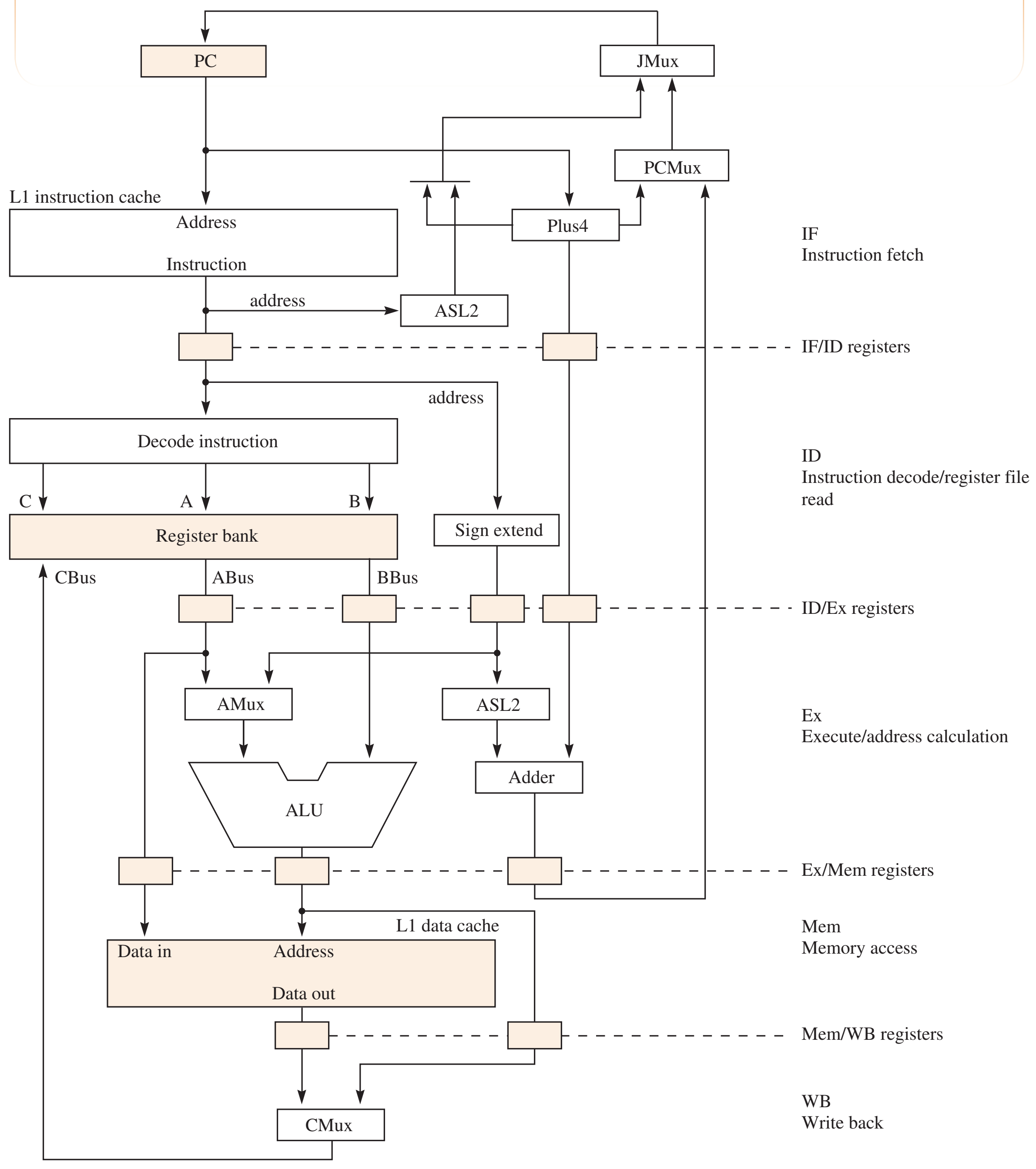


FIGURE 12.44
Instruction execution with pipelining.

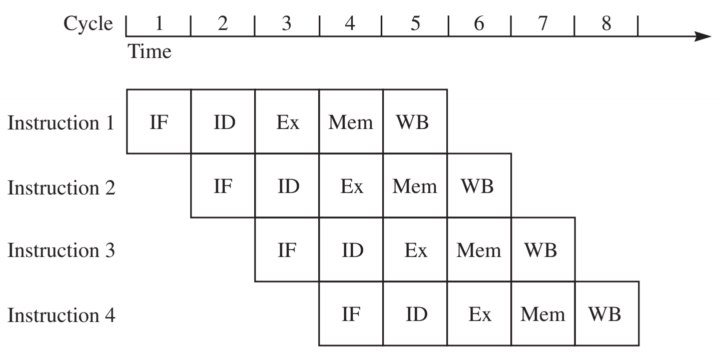
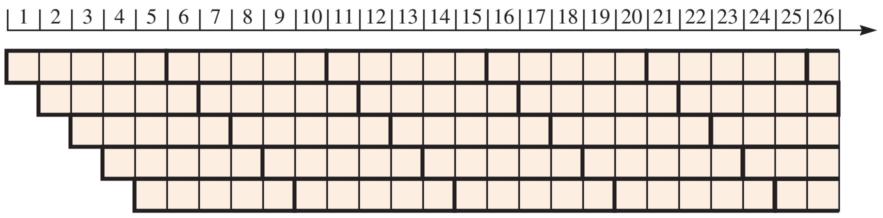
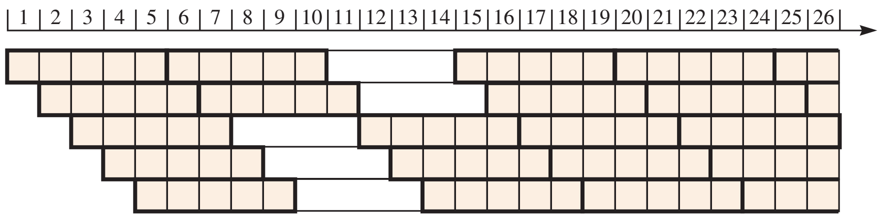


FIGURE 12.45
The effect of a hazard on a pipeline.



(a) No hazards.



(b) A branch hazard.

sixth instruction to execute, the second on the second line represents the seventh, and so on. Starting with cycle 5, the pipeline operates at peak efficiency.

Consider what happens when a branch instruction executes. Suppose instruction 7, which starts at cycle 7 on the second line, is a branch instruction. Assuming that the updated program counter is not available for the next instruction until the completion of this instruction, instruction

8 and every instruction after it must stall. Figure 12.45(b) shows the bubble as not shaded. The effect is as if the pipeline must start over at cycle 12. FIGURE 12.46 shows that branch instructions account for 15% of executing statements in a typical program on a MIPS machine. So roughly every seventh instruction must delay four cycles.

There are several ways to reduce the penalty of a control hazard. Figure 12.45(b) assumes that the result of the branch instruction is not available until after the write-back stage. But branch instructions do not modify the register bank. So, to decrease the length of the bubble the system could eliminate the write-back stage of branch instructions under the assumption that the next instruction has been delayed. The extra control hardware to do that would decrease the length of the bubble from four cycles to three.

Eliminate the write-back stage of the branch instructions.

FIGURE 12.46
Frequency of execution of MIPS instructions.

Instruction	Frequency
Arithmetic	50%
Load/Store	35%
Branch	15%

Conditional branches present another opportunity to minimize the effects of the hazard. Suppose the branch penalty is three cycles with the addition of the extra control hardware, and the computer is executing the following MIPS program:

```
beq $s1,$s2,4
add $s3,$s3,$s4
sub $s5,$s5,$s6
andi $s7,$s7,15
sll $s0,$s0,2
ori $s3,$s3,1
```

The first instruction is a branch if equal. The address field is 4, which means a branch to the fourth instruction after the next one. Consequently, if the branch is taken, it will be to the `ori` instruction. If the branch is not taken, `add` will execute next.

Figure 12.45(b) shows a lot of wasted parallelism. While the bubble is being flushed out of the pipeline, many stages are idle. You do not know whether you should be executing `add`, `sub`, and `andi` while waiting for the results of `beq`. But you can execute them anyway assuming that the branch is not taken. If the branch is in fact not taken, you have eliminated the bubble altogether. If it is taken, you are no worse off in terms of bubble delay than you would have been if you had not started the instruction after `beq`. In that case, flush the bubble from the pipeline.

Assume that conditional branches are not taken.

The problem is the circuitry required to clean up your mess if your assumption is wrong and the branch is taken. You must keep track of any instructions in the interim, before you discover whether the branch is taken, and not allow them to irreversibly modify the data cache or register bank. When you discover that the branch is not taken, you can commit to those changes.

Assuming the branch is not taken is really a crude form of predicting the future. It is like going to the racetrack and betting on the same horse regardless of previous outcomes. You can let history be your guide with a technique known as *dynamic branch prediction*. When a branch statement executes, keep track of whether the branch is taken or not. If it is taken, store its destination address. The next time the instruction executes, predict the same outcome. If it was taken the previous time, continue filling the pipeline with the instructions at the branch destination. If it was not, fill the pipeline with the instructions following the branch instruction.

Dynamic branch prediction

The scheme described above is called *one-bit branch prediction* because one bit is necessary to keep track of whether the branch was taken or not. A one-bit storage cell defines a finite-state machine with two states, corresponding to whether you predict the branch will be taken or not. **FIGURE 12.47** shows the finite-state machine.

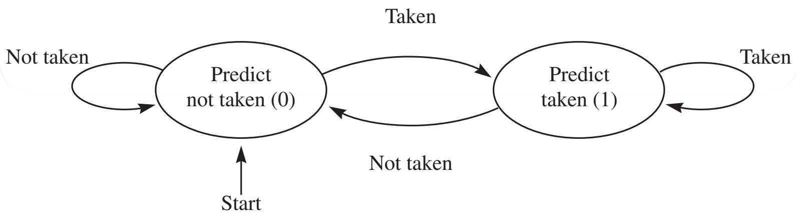
Using the racetrack analogy, perhaps you should not be so quick to change the horse to bet on. Suppose the same horse you have been betting on has won three times in a row. The next time out, another horse wins. Would you really want to change your bet to the other horse based on only one result, discounting the history of previous wins? It is similar to what happens when you have a program with nested loops. Suppose the inner loop executes four times each time the outer loop executes once. The compiler translates the code for the inner loop with a conditional branch that is taken four times in a row, followed by one branch not taken to terminate the loop. Here is the sequence of branches taken and the one-bit dynamic prediction based on Figure 12.47:

Taken:	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N
Prediction:	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y
Incorrect:	x				x	x				x	x				x	x				x

With one-bit dynamic branch prediction, the branch of every inner loop will always be mispredicted twice for each execution of the outer loop.

To overcome this deficiency, it is common to use two bits to predict the next branch. The idea is that if you have a run of branches taken and you

FIGURE 12.47
The state transition diagram for one-bit dynamic branch prediction.



encounter one branch not taken, you do not change your prediction right away. The criterion to change is to get two consecutive branches not taken. FIGURE 12.48 shows that you can be in one of four states. The two shaded states are the ones where you have a run of two consecutive identical branch types—the two previous branches either both taken or both not taken. The states not shaded are for the situation where the two previous branches were different. If you have a run of branches taken, you are in state 00. If the next branch is not taken, you go to state 01, but still predict the branch after that will be taken. A trace of the finite-state machine of Figure 12.48 with the branch sequence above shows that the prediction is correct every time after the outer loop gets started.

Build two pipelines.

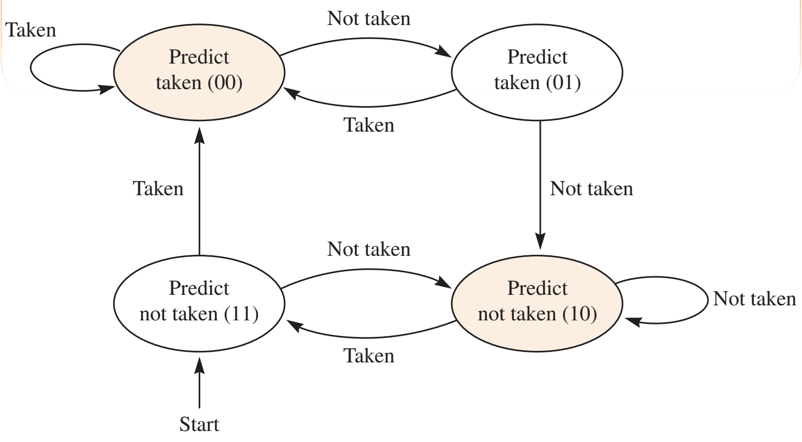
Another technique used on deep pipelines, where the penalty would be more severe if your branch assumption is not correct, is to have duplicate pipelines with duplicate program counters, fetch circuitry, and all the rest. When you decode a branch instruction, you initiate both pipelines, filling one with instructions assuming the branch is not taken and the other assuming it is. When you find out which pipeline is valid, you discard the other and continue on. This solution is quite expensive, but you have no bubbles regardless of whether the branch is taken or not.

A data hazard happens when one instruction needs the result of a previous instruction and must stall until it gets it. It is called a read-after-write (RAW) hazard. An example is the code sequence

A RAW data hazard

```
add $s2,$s2,$s3 # write $s2
sub $s4,$s4,$s2 # read $s2
```

FIGURE 12.48
The state transition diagram for two-bit dynamic branch prediction.



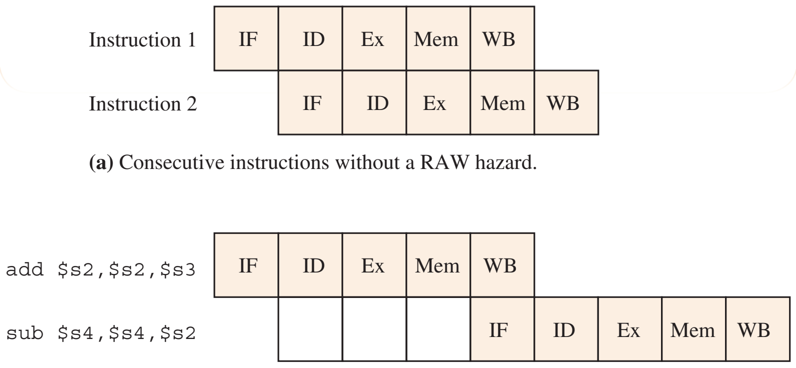
The `add` instruction changes the value of \$s2, which is used in the `sub` instruction. Figure 12.43 shows that the add instruction will update the value of \$s2 at the end of the write-back stage, WB. Also, the `sub` instruction reads from the register bank at the end of the instruction decode/register file read stage, ID. FIGURE 12.49(a) shows two instructions without a RAW hazard, and 12.49(b) shows how the two instructions must overlap with the data hazard. The `sub` instruction's ID stage must come after the `add` instruction's WB stage. The result is that the `sub` instruction must stall, creating a three-cycle bubble.

If there were another instruction with no hazard between `add` and `sub`, the bubble would be only two cycles long. With two hazardless instructions between them, the bubble gets reduced to one cycle in length, and with three, it disappears altogether. This observation brings up a possibility: If you could find some hazardless instructions nearby that needed to be executed sometime anyway, why not just execute them out of order, sticking them in between the `add` and `sub` instructions to fill up the bubble? You might object that mixing up the order in which instructions are executed will change the results of the algorithm. That is true in some cases, but not in all. If there are many arithmetic operations in a block of code, an optimizing compiler can analyze the data dependencies and rearrange the statements to reduce the bubbles in the pipeline without changing the result of the algorithm. Alternatively, a human assembly language programmer can do the same thing.

Instruction reordering

This is an example of the price to be paid for abstraction. A level of abstraction is supposed to simplify computation at one level by hiding the details at a lower level. It would certainly be simpler if the assembly language

FIGURE 12.49
The effect of a RAW data hazard on a pipeline.



(a) Consecutive instructions without a RAW hazard.

(b) Consecutive instructions with a RAW hazard.

programmer or the compiler designer could generate assembly language statements in the most convenient order at Level ISA3, without knowing the details of the pipeline at Level LG1. Adding a level of abstraction always comes with a performance penalty. The question is whether the performance penalty is worth the benefits that come with the simplicity. Using the details of Level LG1 is a tradeoff of the simplicity of abstraction for performance. Another example of the same tradeoff is to design ISA3 programs while taking into account the properties of the cache subsystem.

Trading off abstraction for performance

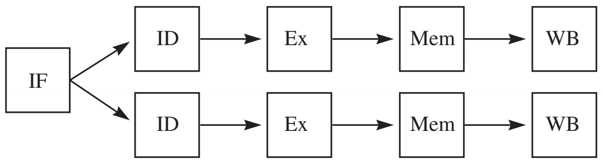
Another technique called *data forwarding* can alleviate data hazards. Figure 12.43 shows that the results of the add instruction from the ALU are clocked into an Ex/Mem boundary register at the conclusion of the Ex stage. For the add instruction, it is simply clocked into a Mem/WB boundary register at the end of the Mem stage and finally into the register bank at the end of the WB stage. If you set up a data path between the Ex/Mem register containing the result of the `add` and one of the ID/Ex registers from which `sub` would normally get the result from the register bank, then the only alignment requirement in Figure 12.49(b) is that the ID stage of `sub` follows the Ex stage of `add`. Doing so still leaves a bubble, but it is only one cycle long.

Data forwarding

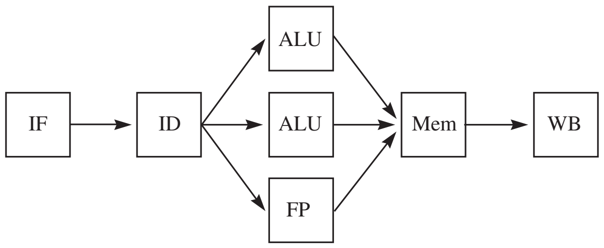
A *superscalar* design exploits the idea that two instructions with no data dependencies can execute in parallel. FIGURE 12.50 shows two approaches. In part (a), you simply build two separate pipelines. There is one fetch unit that is fast enough to fetch more than one instruction in one cycle. It can

Superscalar machines

FIGURE 12.50
Superscalar machines.



(a) Dual pipelines.



(b) Multiple execution units.

issue up to two instructions per cycle concurrently. Scheduling is complex, because data dependencies across the two pipelines must be managed.

Figure 12.50(b) is based on the fact that the execution unit Ex is usually the weakest link in the chain, because its propagation delays are longer than those in the other stages in the pipeline. Floating point units are particularly time consuming compared to integer-processing circuits. The box labeled *FP* in the figure is a floating point unit that implements IEEE 754. Each one of the execution units can be three times slower than the other stages in the pipeline. However, if their inputs and outputs are staggered in time and they work in parallel, the execution stage will not slow down the pipeline.

With superscalar machines, the instruction scheduler must consider other types of data hazards. A write-after-read (WAR) hazard occurs when one instruction writes to a register after the previous one reads it. An example is the following MIPS code:

A WAR data hazard

```
add $s3,$s3,$s2 # read $s2
sub $s2,$s4,$s5 # write $s2
```

In the pipelined machine of Figure 12.43, this sequence is not a hazard, because `add` clocks $s2 into the ID/Ex register at the end of its ID stage, and `sub` writes it at the end of its WB stage. A superscalar machine reorders instructions to minimize bubbles, so it might start the execution of `sub` before it starts `add`. If it does so, it must ensure that the WB stage of `sub` comes after the ID stage of `add`.

In a perfect pipeline, increasing the clock frequency by a factor of *k* using a pipeline with *k* stages increases the performance by a factor of *k*. What is to prevent you from carrying this design to its logical conclusion with a cycle time equal to one gate delay? What prevents you are all the complexities from control hazards and data hazards that reduce the performance from the ideal. There comes a point at which increasing the length of the pipeline and increasing the frequency decrease performance.

But there is one more benefit to increasing the clock frequency—advertising. The megahertz rating on personal computers can be a factor in a consumer's mind when making a decision of which computer to purchase. The *megahertz myth* says that of two machines with two different megahertz ratings, the one with more megahertz has higher performance. You can now understand why the myth is not true. Increasing the frequency by increasing the number of stages in the pipeline increases the hazard performance penalties. The question is how effective the design is in combating those penalties. Using only megahertz as a measure of performance also neglects the interplay between the factors of the performance equation in Figure 12.33. The ultimate question is not just how many cycles per second your CPU can crank out; it is also how much useful work is done per cycle to get the program executed.

The megahertz myth

Future trends

At this time in computing history, pipelining technology has hit a plateau. Although advances in clock speed still occur in commercial CPU chips, they do not do so at the same pace as before. Moore's law continues to hold, with digital circuit engineers providing more total gates per square millimeter on the chips. But the current trend in CPU design is to use the extra circuitry to simply duplicate the entire CPU, fitting multiple CPU cores into a single package. This trend is expected to accelerate in the future. The big challenge will be for software designers to use parallel programming techniques, like those introduced in Section 8.3, to harness the power provided by these multicore CPU chips.

12.4 Conclusion

Pep/9 illustrates the fundamental nature of a von Neumann computer. The data section consists of combinational and sequential circuits. It is one big finite-state machine. Input to the data section consists of data from main memory and control signals from the control section. Output from the data section is also sent to main memory and the control section. Each time the control section sends a clock pulse to a state register, the machine makes a transition to a different state.

Finite-state machines are the basis of all computing.

In a real computer, the number of states is huge but finite. The Pep/9 data section of Figure 12.2 has 25 writable 8-bit registers and 5 status bits, for a total of 205 bits of storage, or 2^{205} states. With 8 inputs from the data lines on the main system bus and 34 control inputs from the control section, the number of transitions from each state is 2^{42}. Considered as a finite-state machine, the machine has a total number of transitions of 2^{205} times 2^{42}, or $2^{247} = 10^{74}$. The number of atoms in the earth is estimated to be only about 10^{50}. And Pep/9 is a tiny computer! At the most fundamental level, no matter how complex the system, computing is nothing more than executing a finite-state machine with peripheral storage.

Simplifications in the Model

The Pep/9 computer illustrates the basic organizational idea behind all real von Neumann machines. Of course, many simplifications are made to keep the machine easy to understand.

One low-level detail that is different in real hardware implementations is the use of edge-triggered flip-flops throughout the integrated circuit instead of master–slave. Both kinds of flip-flop solve the feedback problem. Because the master–slave principle is easier to understand than the edge-triggered principle, that is the one that is carried throughout the presentation.

Another simplification is the interface of the CPU and main memory with the main system bus. With real computers, the timing constraints are more complex than simply putting the addresses on the bus, waiting for three cycles with MemRead asserted, and assuming that data can be clocked into MDR. A single memory access requires more than just two cycles, and there is a protocol that specifies in more detail how long the address lines must be asserted and precisely when the data must be clocked into a register on the CPU side.

Another issue with the main system bus is how it is shared among the CPU, main memory, and the peripheral devices. In practice, the CPU is not always in control of the bus. Instead, the bus has its own processor that arbitrates between competing devices when too many of them want to use the bus at the same time. An example is with *direct memory access* (DMA), in which data flows directly from a disk over the bus to main memory not under control of the CPU. The advantage of DMA is that the CPU can spend its cycles doing useful work executing programs without diverting them to control the peripherals.

Direct memory access

Other topics beyond the scope of this text include assembler macros, linkers, popular peripheral buses like USB and Thunderbolt, supercomputers, and the whole field of computer networks. When you study computer networks, you will find that abstraction is central. Computer systems are designed as layers of abstraction with the details of each level hidden from the level above, and Internet communication protocols are designed the same way. Each level of abstraction exists to do one thing and provides a service to the next higher level, hiding the details of how the service is provided.

The Big Picture

Now consider the big picture from Level App7 all the way down to Level LG1. Suppose a user is entering data in a database system with an application at Level App7. She wants to enter a numerical value, so she types it and executes an Enter command. What lies behind such a seemingly innocuous action?

A C programmer wrote the database system, including the procedure to input the numeric value. The C program was compiled to assembly language, which in turn was assembled into machine language. A compiler designer wrote the compiler, and an assembler designer wrote the assembler. The compiler and assembler, being automatic translators, both contain a lexical analysis phase, a parsing phase, and a code-generating phase. The lexical analysis phase is based on finite-state machines.

The C programmer also used a finite-state machine in the numeric input procedure. The compiler translated each C statement in that procedure to many assembly language statements. The assembler, however, translated each assembly language statement into one machine language statement. So the code to process the user's Enter command was expanded into many C commands, each of which was expanded into many ISA3-level commands.

Each ISA3-level command in turn was translated into the control section signals to fetch the instruction and to execute it. Each control signal is input to a multiplexer or some other combinational device, or it is a pulse to clock a value into a state register. The sequential circuits are governed by the laws of finite-state machines.

Each register is an array of flip-flops, and each flip-flop is a pair of latches designed with the master–slave principle. Each latch is a pair of NOR gates with simple cross-coupled feedback connections. Each combinational part of the data section is an interconnection of only a few different types of gates. The behavior of each gate is governed by the laws of Boolean algebra. Ultimately, the user's Enter command translates into electrical signals that flow through the individual gates.

The user's Enter command may be interrupted by the operating system if it is executing in a multiprogramming system. The Enter command may generate a page fault, in which case the operating system may need to execute a page replacement algorithm to determine the page to copy back to disk.

Of course, all these events happen without the user getting any hint of what is going on at the lower levels of the system. Design inefficiencies at any level can tangibly slow the processing to the extent that the user may curse the computer. Remember that the design of the entire system from Level App7 to Level LG1 is constrained by the fundamental space/time tradeoff.

The connection between one signal flowing through a single gate in some multiplexer at Level LG1 and the user executing an Enter command at Level App7 may seem remote, but it does exist. Literally millions of gates must cooperate to perform a task for the user. So many devices can be organized into a useful machine only by structuring the system into successive levels of abstraction.

Levels of abstraction

It is remarkable that each level of abstraction consists of a few simple concepts. At Level LG1, either the NAND or NOR gate is sufficient to construct any combinational circuit. There are only four basic types of flip-flops, all of which can be produced from the SR flip-flop. The simple von Neumann cycle is the controlling force at Level ISA3 behind the operation of the machine. At Level OS4, a process is a running program that can be interrupted by storing its process control block. Assembly language at Level Asmb5 is a simple one-to-one translation to machine language. A high-order language at Level HOL6 is a one-to-many translation to a lower-level language.

Finite-state machines

The concept of a finite-state machine permeates the entire level structure. Finite-state machines are the basis of lexical analysis for automatic translators, and they also describe sequential circuits. The process control block stores the state of a process.

Simplicity is the key to harnessing complexity.

All sciences have simplicity and structure as their goals. In the natural sciences, the endeavor is to discover the laws of nature that explain the most phenomena with the fewest number of mathematical laws or concepts. Computer scientists have also discovered that simplicity is the key to harnessing complexity. It is possible to construct a machine as complicated as a computer only because of the simple concepts that govern its behavior at every level of abstraction.

Chapter Summary

The central processing unit (CPU) is divided into a data section and a control section. The data section has a bank of registers, some or all of which are visible to the Level-ISA3 programmer. Processing occurs in a loop, with data coming from the register bank on the ABus and BBus, through the ALU, then back to the register bank on the CBus. Data is injected into the loop from main memory via the main system bus and the memory data register at the address specified by the memory address register.

The function of the control section is to send a sequence of control signals to the data section to implement the ISA3 instruction set. The machine is controlled by the von Neumann cycle—fetch, decode, increment, execute, repeat. In a CISC machine like Pep/9, the control signals must direct the data section to fetch the operand, which may take many cycles because of complex addressing modes. A RISC machine like MIPS has few addressing modes and simple instructions so that each instruction executes with only one cycle.

Three common sources of increased performance are increasing the data bus width, inserting cache memory between the CPU and main memory, and pipelining. All three are based on the fundamental space/time tradeoff.

Increasing the data bus width requires more space for the wires on the bus and additional data registers and buses in the data section of the CPU. This increase in space makes possible a decrease in the time to move information between main memory and the CPU. All computer memories are byte-addressable. Exploiting the increased parallelism with a bus that is wider than one byte requires memory alignment of data and programs in assembly language.

Cache memory solves the problem of the extreme mismatch between the fast speed of the CPU and the slow speed of main memory. A cache is a small high-speed memory unit that contains a copy of data from main

memory likely to be accessed by the CPU. It relies on the spatial and temporal locality of reference present in all real programs.

Performance enhancements are based on three components of execution time specified by the performance equation

$$\frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}$$

CISC machines minimize the first factor at the expense of the second. RISC machines, also called *load/store machines*, minimize the second factor at the expense of the first. Both organizations can use the third factor to increase performance, primarily through pipelining.

Computers with complex instructions and many addressing modes were popular early in the history of computing. They are characterized by the Mc2 level of abstraction, in which the control section has its own micromemory, microprogram counter, and microinstruction register. The microprogram of the control section produces the control sequences to implement the ISA3 instruction set. A characteristic of load/store computers is the absence of Level Mc2 because each of its simple instructions can be implemented in one cycle.

Pipelining is analogous to an assembly line in a factory. To implement a pipeline, you subdivide the cycle by putting boundary registers in the data path of the data section. The effect is to increase the number of cycles per instruction but decrease the cycle time proportionally. When the pipeline is full, you execute one instruction per cycle through the parallelism inherent in the pipeline. However, control hazards and data hazards decrease the performance from the theoretical ideal. Techniques to deal with hazards include branch prediction, instruction reordering, and data forwarding. Superscalar machines duplicate pipelines or execution units for greater parallelism.

Exercises

Section 12.1

1. Draw the individual lines of the eight-bit bus between MDR and the main memory bus. Show the tri-state buffers and the connection to the MemWrite line.

2. Design the three-input, one-output combinational circuit AndZ of Figure 12.2. **(a)** Minimize the AND-OR circuit with a Karnaugh map.

(b) Minimize the OR-AND circuit with a Karnaugh map. **(c)** Which one is better?

3. Figure 12.7 combines cycle 6 with cycle 2 and cycle 7 with cycle 3 in Figure 12.5 to speed up the von Neumann cycle. Can you combine cycle 6 with cycle 3 and cycle 7 with cycle 4 instead? Explain.

4. Figure 12.7 combines cycle 6 with cycle 2 and cycle 7 with cycle 3 in Figure 12.5 to speed up the von Neumann cycle. Can you combine cycle 6 with cycle 4 and cycle 7 with cycle 5 instead? Explain.

Section 12.2

* **5.** The text predicts that we will never need to transition from 64-bit computers to 128-bit computers because we will never need main memories bigger than 16 billion GiB. Silicon crystals have a plane consisting of 0.5-nm square tiles, each tile containing two atoms. **(a)** Assuming that you could manufacture a memory so dense as to store one bit per atom on one plane of silicon atoms (and neglecting the interconnection issues with the wires), what would be the length of the side of a square chip necessary to store the maximum number of bytes addressable by a 64-bit computer? Show your calculation. **(b)** Does this calculation support the prediction? Explain.

6. The CPU requests the byte at address 4675 (dec) with the cache of Figure 12.28. **(a)** What are the nine bits of the Tag field? **(b)** What are the four bits of the Byte field? **(c)** Which cell of the cache stored the data?

7. A CPU can address 16 MiB of main memory. It has a direct-mapped cache in which it stores 256 eight-byte cache lines. **(a)** How many bits are required for a memory address? **(b)** How many bits are required for the Byte field of the address? **(c)** How many bits are required for the Line field of the address? **(d)** How many bits are required for the Tag field of the address? **(e)** How many bits are required for the Data field of each cache entry? **(f)** How many bits total are required for all the fields of one cache entry? **(g)** How many bits total are required for the entire cache?

8. If the CPU of Exercise 7 had a two-way set-associative cache, again with 256 eight-byte cache lines, how many bits would be required for each cache entry?

9. For Figure 12.30, **(a)** draw the implementation of a comparator, the circle with the equals sign. (*Hint:* Consider the truth table of an XOR followed by an inverter, sometimes called an *XNOR gate*.) **(b)** Draw the input and output connections to and from the 128 four-input multiplexers. **(c)** Draw the implementation of one of the 128 four-input multiplexers. You may use ellipses (. . .) in parts (a) and (b) of the exercise.

10. A direct-mapped cache is at one extreme of cache designs, with set-associative caches in the middle. At the opposite extreme is the *fully associative cache*, where there is in essence only one entry in the cache of Figure 12.29(a), and the Line field of the address has zero bits—that is, it is missing altogether. An address consists of only the Tag field and the Byte field. **(a)** In Figure 12.29, instead of having 8 cache cells, each with 4 lines, you could use the same number of bits with 1 cache cell having 32 lines. Would this design increase the cache hit percentage over that in Figure 12.29? Explain. **(b)** How many comparators in the read circuit would be required for the cache of part (a)?

Fully associative caches

11. Suppose a CPU can address 1 MByte of main memory. It has a fully associative cache (see Exercise 10) with 16 32-byte cache lines. **(a)** How many bits are required for a memory address? **(b)** How many bits are required for the Byte field of the address? **(c)** How many bits are required for the Tag field of the address? **(d)** How many bits are required for the Data field of each cache entry? **(e)** How many bits total are required for the entire cache?

Section 12.3

***12.** **(a)** Suppose a C compiler for the MIPS machine associates $s4 with array `a`, $s5 with variable `g`, and $s6 with array `b`. How does it translate

```
a[4] = g + b[5];
```

into MIPS assembly language? **(b)** Write the machine language translation of the instructions in part (a).

13. **(a)** Write the MIPS assembly language statement to shift the content of register $s2 nine bits to the left and put the result in $t5. **(b)** Write the machine language translation of the instruction in (a).

14. **(a)** Suppose a C compiler for the MIPS machine associates $s4 with variable `g`, $s5 with array `a`, and $s6 with variable `i`. How does it translate

```
g = a[i];
```

into MIPS assembly language? **(b)** Write the machine language translation of the instructions in part (a).

15. **(a)** Suppose a C compiler for the MIPS machine associates \$s4 with variable `g`, \$s5 with array `a`, and \$s6 with variable `i`. How does it translate

```
a[i] = g;
```

into MIPS assembly language? **(b)** Write the machine language translation of the instructions in part (a).

16. **(a)** Suppose a C compiler for the MIPS machine associates \$s4 with variable `g`, \$s5 with array `a`, and \$s6 with variable `i`. How does it translate

```
g = a[i+3];
```

into MIPS assembly language? **(b)** Write the machine language translation of the instructions in part (a).

17. **(a)** Suppose a C compiler for the MIPS machine associates \$s5 with array `a` and \$s6 with variable `i`. How does it translate

```
a[i] = a[i+1];
```

into MIPS assembly language? **(b)** Write the machine language translation of the instructions in part (a).

18. **(a)** Write the MIPS assembly language statement to allocate 12 bytes of storage on the run-time stack. **(b)** Write the machine language translation of the instruction in part (a).

19. **(a)** Suppose a C compiler for the MIPS machine associates \$s5 with variable `g`. How does it translate

```
g = 529371;
```

into MIPS assembly language? **(b)** Write the machine language translation of the instructions in part (a).

20. **(a)** What is the RTL specification for the `lw` instruction? **(b)** For Figure 12.40, write the control signals to execute the `lw` instruction.

21. In Figure 12.43, **(a)** how many bits are in each of the two IF/ID boundary registers? **(b)** How many bits are in each of the four ID/Ex boundary registers? **(c)** How many bits are in each of the three Ex/Mem boundary registers? **(d)** How many bits are in each of the two Mem/WB boundary registers?

22. For Figure 12.45(b), place a checkmark for each circuit that is idle in each of the cycles in the table below, and list the total number of circuits that are idle for each cycle.

Cycle	7	8	9	10	11	12	13	14	15	16
IF										
ID										
Ex										
Mem										
WB										
Number idle										

23. Suppose the five-stage pipeline of Figure 12.45(a) executes a branch 15% of the time, each branch causing the next instruction that executes to stall until the completion of the branch, as in Figure 12.45(b). **(a)** What is the percentage increase in the number of cycles over the ideal pipeline with no bubbles? **(b)** Suppose an n-stage pipeline executes a branch x% of the time, each branch causing the next instruction that executes to stall until the completion of the branch. What is the percentage increase in the number of cycles over the ideal pipeline with no bubbles?

24. The text states that you can eliminate the write-back stage of an unconditional branch under the assumption that the next instruction has been delayed. **(a)** Draw cycles 7 through 16 of Figure 12.45(b) with that design. **(b)** Complete the table of Exercise 22 with that design.

25. **(a)** In Figure 12.47 for one-bit dynamic branch prediction, what pattern of Taken outcomes will produce the maximum percentage of incorrect predictions? What is the maximum percentage? **(b)** In Figure 12.48 for two-bit dynamic branch prediction, what pattern of Taken outcomes will produce the maximum percentage of incorrect predictions? What is the maximum percentage?

26. Construct the one-input finite-state machine of Figure 12.48 to implement two-bit dynamic branch prediction. Minimize your circuit with a Karnaugh map. **(a)** Use two SR flip-flops. **(b)** Use two JK flip-flops. **(c)** Use two D flip-flops. **(d)** Use two T flip-flops.

27. The finite-state machine of Figure 12.48 for branch prediction moves from predicting no branch to predicting branch only if two consecutive branches were made (and similarly for moving from predicting branch to predicting no branch). **(a)** Draw a finite-state machine that moves from predicting no branch to predicting branch only if *three* consecutive branches were made (and similarly for moving from predicting branch to predicting no branch). **(b)** How many prediction bits would be necessary for the implementation of the machine?

Problems

The problems in this chapter are to write the control sequences to implement ISA3 instructions at Level Mc2. For each problem, write your implementation in the Pep/9 CPU simulator. The Help feature of the application has unit tests for each problem, which you must use to test your implementation. For all problems, use the fewest cycles possible.

Section 12.1

28. Write the control sequence with the one-byte data bus for the von Neumann cycle to fetch the operand specifier and increment PC accordingly. Assume the instruction specifier has been fetched and the control section has determined that the instruction is nonunary.

29. Write the control sequence with the one-byte data bus to implement the following unary ISA3 instructions. Assume the instruction has been fetched and the program counter has been incremented.

*(a) `MOVSPA` (b) `MOVFLGA`
(c) `MOVAFLG` (d) `NOTA`
(e) `NEGA` (f) `ROLA`
(g) `RORA`

30. Write the control sequence with the one-byte data bus to implement the `ASLA` instruction, which does an arithmetic shift left of the accumulator and puts the result back in the accumulator. The RTL specification of `ASLA` at the ISA3 level shows that, unlike `ASRA`, the V bit is set to indicate an overflow when the quantity is interpreted as a signed integer. At the Mc2 level, an `ASLA` instruction is implemented as an ASL on the low-order byte followed by an ROL on the high-order byte. Even though the ROL operation does not set the V bit at the ISA3 level, Figure 10.55 shows that the ALU function for ROL does compute the V bit at the Mc2 level. So, you can access the V output from the ALU on the ROL operation. Assume the instruction has been fetched and the program counter has been incremented.

31. Write the control sequence with the one-byte data bus to implement the following nonunary ISA3 instructions. Assume that the instruction has been fetched and the program counter has been incremented. Note that the operand is already in the instruction register (IR).

*(**a**) `SUBA this,i` (**b**) `ANDA this,i`
(**c**) `ORA this,i` (**d**) `CPWA this,i`
(**e**) `CPBA this,i` (**f**) `LDWA this,i`
(**g**) `LDBA this,i`

32. Write the control sequence with the one-byte data bus to implement the following nonunary ISA3 instructions. Assume that the instruction has been fetched and the program counter has been incremented.

*(**a**) `LDWA here,d` (**b**) `LDWA here,s`
(**c**) `LDWA here,sf` (**d**) `LDWA here,x`
(**e**) `LDWA here,sx` (**f**) `LDWA here,sfx`
(**g**) `STWA there,n` (**h**) `STWA there,s`
(**i**) `STWA there,sf` (**j**) `STWA there,x`
(**k**) `STWA there,sx` (**l**) `STWA there,sfx`

33. Write the control sequence with the one-byte data bus to implement the following flow of control ISA3 instructions. Assume that the instruction has been fetched and the program counter has been incremented. Because `DECO` is a trap instruction, its addressing mode is irrelevant to its implementation. All trap instructions have the same implementation at the ISA3 level.

(**a**) `BR main` (**b**) `BR guessJT,x`
(**c**) `CALL alpha` (**d**) `RET`
(**e**) `DECO 0x0003,d` (**f**) `RETTR`

Section 12.2

34. Write the control sequence with the two-byte data bus for the von Neumann cycle to fetch the operand specifier and increment PC accordingly. (**a**) Assume the program counter is even so that the first byte of the operand specifier has not been prefetched. (**b**) Assume the program counter is odd so that the first byte of the operand specifier has been prefetched. Prefetch the following instruction specifier.

35. Write the control sequence with the two-byte data bus to implement the ISA3 instructions of Problem 32. Assume that all addresses and word operands are aligned at even addresses. Compare the number of cycles with the corresponding number of cycles for the one-byte data bus and compute the percentage savings in the number of cycles for the two-byte bus design.

36. Write the control sequence with the two-byte data bus to implement the ISA3 instructions of Problem 33. Assume that all addresses and word operands are aligned at even addresses. Compare the number of cycles with the corresponding number of cycles for the one-byte data bus and compute the percentage savings in the number of cycles for the two-byte bus design. Part (e) is a trap instruction, and part (f) is the return from trap instruction. Both parts assume an aligned system stack. The modified RTL specification for the aligned system stack is provided in the Pep/9 CPU simulator along with the usual unit tests.

37. Insert the `.ALIGN` dot command and the `NOP0` instruction to align the following Pep/9 assembly language programs for the Pep/9 processor with the two-byte data bus. Remember that all `CALL` statements must be at odd addresses so the return address will be at an even address. The source code for the original nonaligned program is available in the Help facility of the Pep/9 application. Test your aligned programs.

(a) Figure 5.22
*__(b)__ Figure 6.10
(c) Figure 6.12
(d) Figure 6.18
(e) Figure 6.21