

CS6303 – COMPUTER ARCHITECTURE

LESSION NOTES

UNIT I OVERVIEW & INSTRUCTIONS

8 GREAT IDEAS

1. Design for Moore's Law



MOORE'S LAW The one constant for computer designers is rapid change, which is driven largely by Moore's Law. It states that integrated circuit resources double every 18–24 months. Moore's Law resulted from a 1965 prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel. As computer designs can take years, the resources available per chip can easily double or quadruple between the start and finish of the project. Like a skeet shooter, computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts. We use an "up and to the right" Moore's Law graph to represent designing for rapid change.

2. Use Abstraction to Simplify Design



ABSTRACTION Both computer architects and programmers had to invent techniques to make themselves more productive, for otherwise design time would lengthen as dramatically as resources grew by Moore's Law. A major productivity technique for hardware and software is to use abstractions to represent the design at different levels of representation; lower-level details are hidden to offer a simpler model at higher levels. We'll use the abstract painting icon to represent this second great idea.

3. Make the common case fast



COMMON CASE FAST Making the common case fast will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is often easier to enhance. This common sense advice implies that you know what the common case is, which is only possible with careful experimentation and measurement. We use a sports car as the icon for making the common case fast, as the most common trip has one or two passengers, and it's surely easier to make a fast sports car than a fast minivan.

4. Performance via parallelism



PARALLELISM Since the dawn of computing, computer architects have offered designs that get more performance by performing operations in parallel. We'll see many examples of parallelism in this book. We use multiple jet engines of a plane as our icon for parallel performance.

5. Performance via pipelining



PIPELINING A particular pattern of parallelism is so prevalent in computer architecture that it merits its own name: pipelining. For example, before fire engines, a "bucket brigade" would respond to a fire, which many cowboy movies show in response to a dastardly act by the villain. The townsfolk form a human chain to carry a water source to fire, as they could much more quickly move buckets up the chain instead of individuals running back and forth. Our pipeline icon is a sequence of pipes, with each section representing one stage of the pipeline.

6. Performance via prediction

Following the saying that it can be better to ask for forgiveness than to ask for permission, the next great idea is prediction. In some cases it can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too

expensive and your prediction is relatively accurate. We use the fortune-teller's crystal ball as our prediction icon.

7. Hierarchy of memories



HIERARCHY Programmers want memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost. Architects have found that they can address these conflicting demands with a hierarchy of memories, with the fastest, smallest, and most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom. Caches give the programmer the illusion that main memory is nearly as fast as the top of the hierarchy and nearly as big and cheap as the bottom of the hierarchy. We use a layered triangle icon to represent the memory hierarchy. The shape indicates speed, cost, and size: the closer to the top, the faster and more expensive per bit the memory; the wider the base of the layer, the bigger the memory.

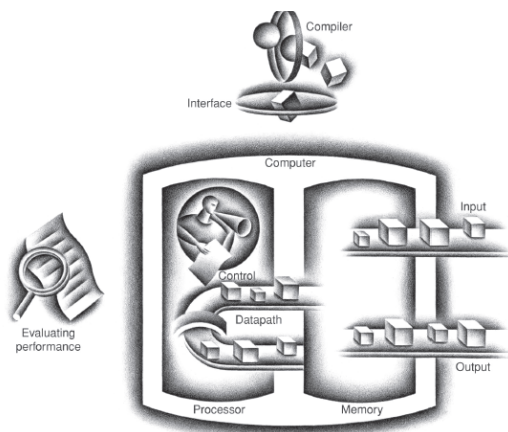
8. Dependability via redundancy



DEPENDABILITY Computers not only need to be fast; they need to be dependable. Since any physical device can fail, we make systems dependable by including redundant components that can take over when a failure occurs and to help detect failures. We use the tractor-trailer as our icon, since the dual tires on each side of its rear axels allow the truck to continue driving even when one tire fails. (Presumably, the truck driver heads immediately to a repair facility so the flat tire can be fixed, thereby restoring redundancy!)

COMPONENTS OF COMPUTER SYSTEM

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor. Figure 1.5 shows the standard organization of a computer. This organization is independent of hardware technology: you can place every piece of every computer, past and present, into one of these five categories.



Through the Looking Glass

The most fascinating I/O device is probably the graphics display. Most personal mobile devices use liquid crystal displays (LCDs) to get a thin, low-power display. The LCD is not the source of light; instead, it controls the transmission of light. A typical LCD includes rod-shaped molecules in a liquid that form a twisting helix that bends light entering the display, from either a light source behind the display or less often from reflected light. The rods straighten out when a current is applied and no longer bend the light. Since the liquid crystal material is between two screens polarized at 90 degrees, the light cannot pass through unless it is bent.

Today, most LCD displays use an active matrix that has a tiny transistor switch at each pixel to precisely control current and make sharper images. A red-green-blue mask associated with each dot on the display determines the intensity of the three-color components in the final image; in a color active matrix LCD, there are three transistor switches at each point.

The image is composed of a matrix of picture elements, or pixels, which can be represented as a matrix of bits, called a *bit map*. Depending on the size of the screen and the resolution, the display matrix in a typical tablet ranges in size from 1024 _ 768 to 2048 _ 1536. A color display might use 8 bits for each of the three colors (red, blue, and green), for 24 bits per pixel, permitting millions of different colors to be displayed.

Touchscreen

While PCs also use LCD displays, the tablets and smartphones of the PostPC era have replaced the keyboard and mouse with touch sensitive displays, which has the wonderful user interface advantage of users pointing directly what they are interested in rather than indirectly with a mouse. While there are a variety of ways to implement a touch screen, many tablets today use capacitive sensing. Since people are electrical conductors, if an insulator like glass is covered with a transparent conductor, touching distorts the electrostatic field of the screen, which results in a change in capacitance. This technology can allow multiple touches simultaneously, which allows gestures that can lead to attractive user interfaces.

Opening the Box

Figure 1.7 shows the contents of the Apple iPad 2 tablet computer. Unsurprisingly, of the five classic components of the computer, I/O dominates this reading device. The list of I/O devices includes a capacitive multitouch LCD display, front facing camera, rear facing camera, microphone, headphone jack, speakers, accelerometer, gyroscope, Wi-Fi network, and Bluetooth network. The datapath, control, and memory are a tiny portion of the components. The small rectangles in Figure 1.8 contain the devices that drive our advancing technology, called integrated circuits and nicknamed chips. The A5 package seen in the middle of in Figure 1.8 contains two ARM processors that operate with a clock rate

of 1 GHz. The *processor* is the active part of the computer, following the instructions of a program to the letter. It adds numbers, tests numbers, signals I/O devices to activate, and so on. Occasionally, people call the processor the CPU, for the more bureaucratic-sounding central processor unit.

Cache memory

It consists of a small, fast memory that acts as a buffer for the DRAM memory. (The nontechnical definition of *cache* is a safe place for hiding things.) Cache is built using a different memory technology, static random access memory (SRAM). SRAM is faster but less dense, and hence more expensive, than DRAM (see Chapter 5). SRAM and DRAM are two layers of the memory hierarchy.

A Safe Place for Data

Thus far, we have seen how to input data, compute using the data, and display data. If we were to lose power to the computer, however, everything would be lost because the memory inside the computer is volatile—that is, when it loses power, it forgets. In contrast, a DVD disk doesn't forget the movie when you turn off the power to the DVD player, and is thus a nonvolatile memory technology.

Communicating with Other Computers

We've explained how we can input, compute, display, and save data, but there is still one missing item found in today's computers: computer networks. Just as the processor shown in [Figure 1.5](#) is connected to memory and I/O devices, networks interconnect whole computers, allowing computer users to extend the power of computing by including communication. Networks have become so popular that they are the backbone of current computer systems; a new personal mobile device or server without a network interface would be ridiculed. Networked computers have several major advantages:

Communication: Information is exchanged between computers at high speeds.

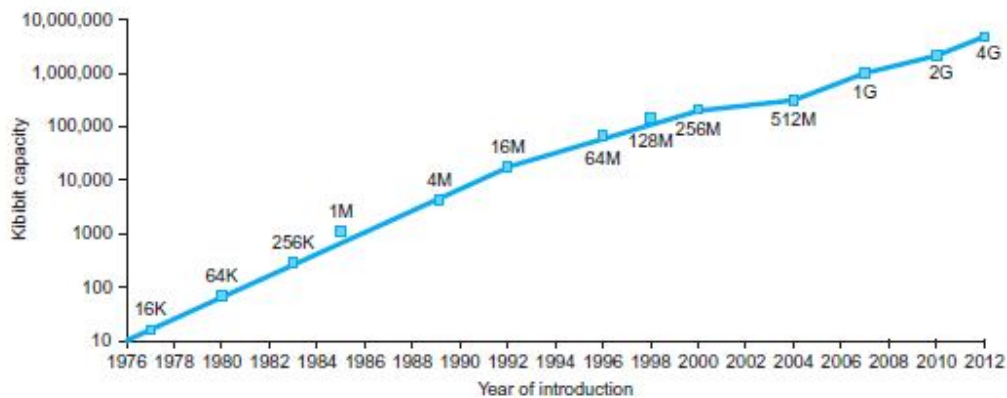
Resource sharing: Rather than each computer having its own I/O devices, computers on the network can share I/O devices.

Nonlocal access: By connecting computers over long distances, users need not be near the computer they are using.

Networks vary in length and performance, with the cost of communication increasing according to both the speed of communication and the distance that information travels. Perhaps the most popular type of network is *Ethernet*. It can be up to a kilometer long and transfer at up to 40 gigabits per second.

Technologies for Building Processors and Memory

Processors and memory have improved at an incredible rate, because computer designers have long embraced the latest in electronic technology to try to win the race to design a better computer. It has been used over time, with an estimate of the relative performance per unit cost for each technology. Since this technology shapes what computers will be able to do and how quickly they will evolve, we believe all computer professionals should be familiar with the basics of integrated circuits.



A **transistor** is simply an on/off switch controlled by electricity. The *integrated circuit* (IC) combined dozens to hundreds of transistors into a single chip. When Gordon Moore predicted the continuous doubling of resources, he was predicting the growth rate of the number of transistors per chip. To describe the tremendous increase in the number of transistors from hundreds to millions, the adjective *very large scale* is added to the term, creating the abbreviation *VLSI*, for **very large-scale integrated circuit**.

This rate of increasing integration has been remarkably stable. Figure 1.11 shows the growth in DRAM capacity since 1977. For decades, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times! To understand how manufacture integrated circuits, we start at the beginning. The manufacture of a chip begins with **silicon**, a substance found in sand. Because silicon does not conduct electricity well, it is called a **semiconductor**. With a special chemical process, it is possible to add materials to silicon that allow tiny areas to transform into one of three devices: □ Excellent conductors of electricity (using either microscopic copper or aluminum wire) been used over time, with an estimate of the relative performance per unit cost for each technology. Since this technology shapes what computers will be able to do and how quickly they will evolve, we believe all computer professionals should be familiar with the basics of integrated circuits.

A **transistor** is simply an on/off switch controlled by electricity. The *integrated circuit* (IC) combined dozens to hundreds of transistors into a single chip. When Gordon Moore predicted the continuous doubling of resources, he was predicting the growth rate of the number of transistors per chip. To describe the tremendous increase in the number of transistors from hundreds to millions, the adjective *very large scale* is added to the term, creating the abbreviation *VLSI*, for **very large-scale integrated circuit**. This rate of increasing integration has been remarkably stable. Figure 1.11 shows the growth in DRAM capacity since 1977. For decades, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times! To understand how manufacture integrated circuits, we start at the beginning. The manufacture of a chip begins with **silicon**, a substance found in sand. Because silicon does not conduct electricity well, it is called a **semiconductor**. With a special chemical process, it is possible to add materials to silicon that allow tiny areas to transform into one of three devices:

- Excellent conductors of electricity (using either microscopic copper or
- Excellent insulators from electricity (like plastic sheathing or glass)
- Areas that can conduct or insulate under special conditions (as a switch)

Transistors fall in the last category. A VLSI circuit, then, is just billions of combinations of conductors, insulators, and switches manufactured in a single small package. aluminum wire)

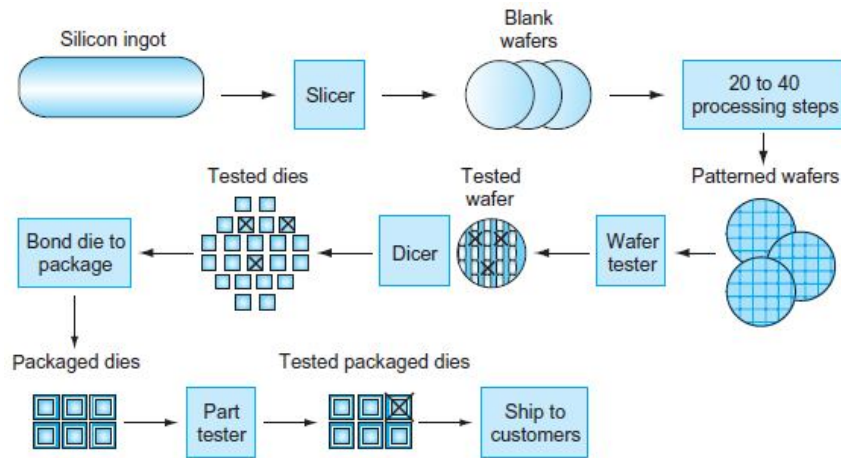


FIGURE 1.12 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see Figure 1.13). These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Then, the wafers are diced into dies (see Figure 1.9). In this figure, one wafer produced 20 dies, of which 17 passed testing. (X means the die is bad.) The yield of good dies in this case was 17/20, or 85%. These good dies are then bonded into packages and tested one more time before shipping the packaged parts to customers. One bad packaged part was found in this final test.

Elaboration: The cost of an integrated circuit can be expressed in three simple equations:

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$

$$\text{Dies per wafer} \approx \frac{\text{Wafer area}}{\text{Die area}}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2))^2}$$

The first equation is straightforward to derive. The second is an approximation, since it does not subtract the area near the border of the round wafer that cannot accommodate the rectangular dies (see Figure 1.13). The final equation is based on empirical observations of yields at integrated circuit factories, with the exponent related to the number of critical processing steps.

Hence, depending on the defect rate and the size of the die and wafer, costs are generally not linear in the die area.

Performance

Defining Performance

When we say one computer has better performance than another, what do we mean? Although this question might seem simple, an analogy with passenger airplanes shows how subtle the question of performance can be. Figure 1.14 lists some typical passenger airplanes, together with their cruising speed, range, and capacity. If we wanted to know which of the planes in this table had the best performance, we would first need to define performance. For example, considering different measures of performance, we see that the plane with the highest cruising speed was the Concorde (retired from service in 2003), the plane with the longest range is the DC-8, and the plane with the largest capacity is the 747.

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers × m.p.h.)
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

FIGURE 1.14 The capacity, range, and speed for a number of commercial airplanes. The last column shows the rate at which the airplane transports passengers, which is the capacity times the cruising speed (ignoring range and takeoff and landing times).

Throughput and Response Time

Do the following changes to a computer system increase throughput, decrease response time, or both?

1. Replacing the processor in a computer with a faster version
 2. Adding additional processors to a system that uses multiple processors for separate tasks—for example, searching the web
- Decreasing response time almost always improves throughput. Hence, in case

1, both response time and throughput are improved. In case 2, no one task gets work done faster, so only throughput increases. If, however, the demand for processing in the second case was almost as large as the throughput, the system might force requests to queue up. In this case, increasing the throughput could also improve response time, since it would reduce the waiting time in the queue. Thus, in many real computer systems, changing either execution time or throughput often affects the other. In discussing the performance of computers, we will be primarily concerned with response time for the first few chapters. To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\begin{aligned} \text{Performance}_X &> \text{Performance}_Y \\ \frac{1}{\text{Execution time}_X} &> \frac{1}{\text{Execution time}_Y} \\ \text{Execution time}_Y &> \text{Execution time}_X \end{aligned}$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

Relative Performance

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

We know that A is n times as fast as B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times as fast as B.

In the above example, we could also say that computer B is 1.5 times *slower than* computer A, since

$$\frac{\text{Performance}_A}{\text{Performance}_B} = 1.5$$

means that

$$\frac{\text{Performance}_A}{1.5} = \text{Performance}_B$$

The Power Wall

Figure 1.16 shows the increase in clock rate and power of eight generations of Intel microprocessors over 30 years. Both clock rate and power increased rapidly for decades, and then flattened off recently. The reason they grew together is that they are correlated, and the reason for their recent slowing is that we have run into the practical power limit for cooling commodity microprocessors.

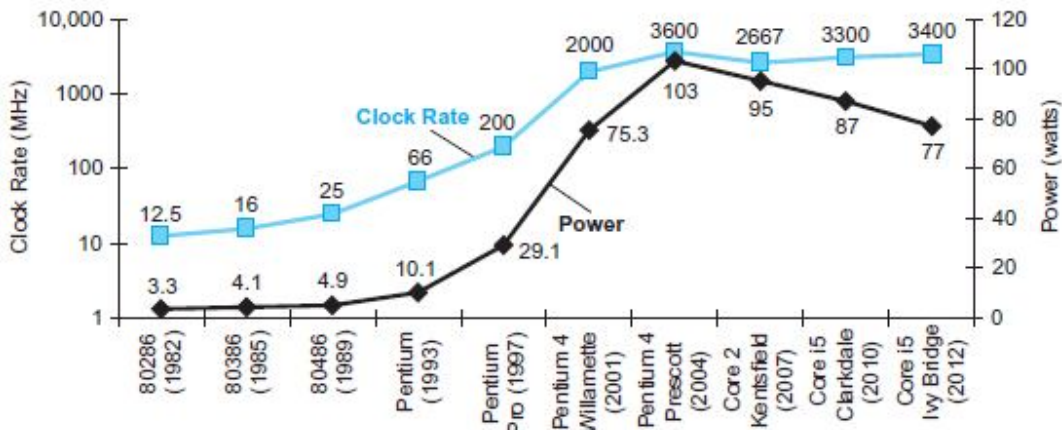


FIGURE 1.16 Clock rate and Power for Intel x86 microprocessors over eight generations and 25 years. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip. The Core i5 pipelines follow in its footsteps.

The dominant technology for integrated circuits is called CMOS (complementary metal oxide semiconductor). For CMOS, the primary source of energy consumption is so-called dynamic energy—that is, energy that is consumed when transistors switch states from 0 to 1 and vice versa. The dynamic energy depends on the capacitive loading of each transistor and the voltage applied:

$$\text{Energy} \propto \text{Capacitive load} \times \text{Voltage}^2$$

This equation is the energy of a pulse during the logic transition of $0 \rightarrow 1 \rightarrow 0$ or $1 \rightarrow 0 \rightarrow 1$. The energy of a single transition is then

$$\text{Energy} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2$$

The power required per transistor is just the product of energy of a transition and the frequency of transitions:

$$\text{Power} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

Frequency switched is a function of the clock rate. The capacitive load per transistor is a function of both the number of transistors connected to an output (called the *fanout*) and the technology, which determines the capacitance of both wires and transistors.

The Sea Change: The Switch from Uniprocessors to Multiprocessors

The power limit has forced a dramatic change in the design of microprocessors. Figure 1.17 shows the improvement in response time of programs for desktop microprocessors over time. Since 2002, the rate has slowed from a factor of 1.5 per year to a factor of 1.2 per year.

Rather than continuing to decrease the response time of a single program running on the single processor, as of 2006 all desktop and server companies are shipping microprocessors with multiple processors per chip, where the benefit is often more on throughput than on response time. To reduce confusion between the words processor and microprocessor, companies refer to processors as “cores,” and such microprocessors are generically called multicore microprocessors.

Hence, a “quadcore” microprocessor is a chip that contains four processors or four cores. In the past, programmers could rely on innovations in hardware, architecture, and compilers to double performance of their programs every 18 months without having to change a line of code. Today, for programmers to get significant improvement in response time, they need to rewrite their programs to take advantage of multiple processors. Moreover, to get the historic benefit of running faster on new microprocessors, programmers will have to continue to improve performance of their code as the number of cores increases.

To reinforce how the software and hardware systems work hand in hand, we use a special section, *Hardware/Software Interface*, throughout the book, with the first one appearing below. These elements summarize important insights at this critical interface.

Parallelism has always been critical to performance in computing, but it was often hidden. Chapter 4 will explain **pipelining**, an elegant technique that runs programs faster by overlapping the execution of instructions. This is one example of *instruction-level parallelism*, where the parallel nature of the hardware is abstracted away so the programmer and compiler can think of the hardware as executing instructions sequentially.

Forcing programmers to be aware of the parallel hardware and to explicitly rewrite their programs to be parallel had been the “third rail” of computer architecture, for companies in the past that depended on such a change in behavior failed (see Section 6.15). From this historical perspective, it’s startling that the whole IT industry has bet its future that programmers will finally successfully switch to explicitly parallel programming.

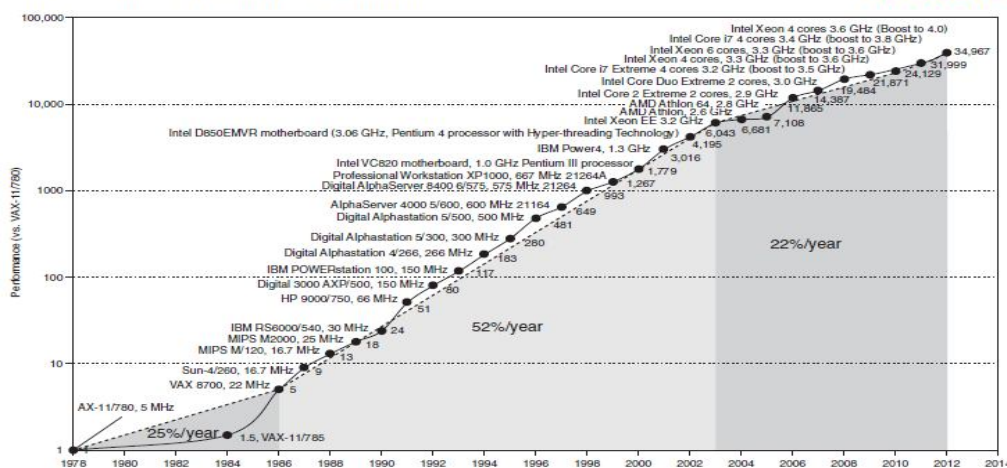


FIGURE 1.17 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.10). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. The higher annual performance improvement of 52% since the mid-1980s meant performance was about a factor of seven higher in 2002 than it would have been had it stayed at 25%. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 22% per year.

Operations of the Computer Hardware

Every computer must be able to perform arithmetic. The MIPS assembly language notation `add a, b, c` instructs a computer to add the two variables `b` and `c` and to put their sum in `a`.

This notation is rigid in that each MIPS arithmetic instruction performs only one operation and must always have exactly three variables. For example, suppose we want to place the sum of four variables `b`, `c`, `d`, and `e` into variable `a`. (In this section we are being deliberately vague about what a “variable” is; in the next section we’ll explain in detail.)

The following sequence of instructions adds the four variables:

```
add a, b, c # The sum of b and c is placed in a
add a, a, d # The sum of b, c, and d is now in a
add a, a, e # The sum of b, c, d, and e is now in a
```

Thus, it takes three instructions to sum the four variables. The words to the right of the sharp symbol (`#`) on each line above are *comments* for the human reader, so the computer ignores them.

MIPS ASSEMBLY LANGUAGE CODE

Category	Instruction	Example	Meaning	Comments
Arithmetic	<code>add</code>	<code>add \$s1,\$s2,\$s3</code>	$\$s1 = \$s2 + \$s3$	Three register operands
	<code>subtract</code>	<code>sub \$s1,\$s2,\$s3</code>	$\$s1 = \$s2 - \$s3$	Three register operands
	<code>add immediate</code>	<code>addi \$s1,\$s2,20</code>	$\$s1 = \$s2 + 20$	Used to add constants
Data transfer	<code>load word</code>	<code>lw \$s1,20(\$s2)</code>	$\$s1 = \text{Memory}[\$s2 + 20]$	Word from memory to register
	<code>store word</code>	<code>sw \$s1,20(\$s2)</code>	$\text{Memory}[\$s2 + 20] = \$s1$	Word from register to memory
	<code>load half</code>	<code>lh \$s1,20(\$s2)</code>	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	<code>load half unsigned</code>	<code>lhu \$s1,20(\$s2)</code>	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	<code>store half</code>	<code>sh \$s1,20(\$s2)</code>	$\text{Memory}[\$s2 + 20] = \$s1$	Halfword register to memory
	<code>load byte</code>	<code>lb \$s1,20(\$s2)</code>	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	<code>load byte unsigned</code>	<code>lbu \$s1,20(\$s2)</code>	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	<code>store byte</code>	<code>sb \$s1,20(\$s2)</code>	$\text{Memory}[\$s2 + 20] = \$s1$	Byte from register to memory
	<code>load linked word</code>	<code>ll \$s1,20(\$s2)</code>	$\$s1 = \text{Memory}[\$s2 + 20]$	Load word as 1st half of atomic swap
	<code>store condition. word</code>	<code>sc \$s1,20(\$s2)</code>	$\text{Memory}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ or } 1$	Store word as 2nd half of atomic swap
Logical	<code>load upper immed.</code>	<code>lui \$s1,20</code>	$\$s1 = 20 * 2^{16}$	Loads constant in upper 16 bits
	<code>and</code>	<code>and \$s1,\$s2,\$s3</code>	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	<code>or</code>	<code>or \$s1,\$s2,\$s3</code>	$\$s1 = \$s2 \$s3$	Three reg. operands; bit-by-bit OR
	<code>nor</code>	<code>nor \$s1,\$s2,\$s3</code>	$\$s1 = \sim (\$s2 \$s3)$	Three reg. operands; bit-by-bit NOR
	<code>and immediate</code>	<code>andi \$s1,\$s2,20</code>	$\$s1 = \$s2 \& 20$	Bit-by-bit AND reg with constant
	<code>or immediate</code>	<code>ori \$s1,\$s2,20</code>	$\$s1 = \$s2 20$	Bit-by-bit OR reg with constant
Conditional branch	<code>shift left logical</code>	<code>sll \$s1,\$s2,10</code>	$\$s1 = \$s2 \ll 10$	Shift left by constant
	<code>shift right logical</code>	<code>srl \$s1,\$s2,10</code>	$\$s1 = \$s2 \gg 10$	Shift right by constant
	<code>branch on equal</code>	<code>beq \$s1,\$s2,25</code>	if $(\$s1 == \$s2)$ go to PC + 4 + 100	Equal test; PC-relative branch
	<code>branch on not equal</code>	<code>bne \$s1,\$s2,25</code>	if $(\$s1 \neq \$s2)$ go to PC + 4 + 100	Not equal test; PC-relative
	<code>set on less than</code>	<code>slt \$s1,\$s2,\$s3</code>	if $(\$s2 < \$s3)$ $\$s1 = 1$; else $\$s1 = 0$	Compare less than; for <code>beq</code> , <code>bne</code>
	<code>set on less than unsigned</code>	<code>sltu \$s1,\$s2,\$s3</code>	if $(\$s2 < \$s3)$ $\$s1 = 1$; else $\$s1 = 0$	Compare less than unsigned
<code>set less than immediate</code>	<code>slti \$s1,\$s2,20</code>	if $(\$s2 < 20)$ $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant	
<code>set less than immediate unsigned</code>	<code>sltiu \$s1,\$s2,20</code>	if $(\$s2 < 20)$ $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant unsigned	

Compiling Two C Assignment Statements Into MIPS

This segment of a C program contains the five variables *a*, *b*, *c*, *d*, and *e*. Since Java evolved from C, this example and the next few work for either high-level programming language:

```
a = b + c;  
d = a - e;
```

The translation from C to MIPS assembly language instructions is performed by the *compiler*. Show the MIPS code produced by a compiler.

A MIPS instruction operates on two source operands and places the result in one destination operand. Hence, the two simple statements above compile directly into these two MIPS assembly language instructions:

```
add a, b, c  
sub d, a, e
```

Operands of the Computer Hardware

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers, like MIPS. (See [Section 2.21](#) for the history of the number of registers.) Thus, continuing in our top-down, stepwise evolution of the symbolic representation of the MIPS language, in this section we have added the restriction that the three operands of MIPS arithmetic instructions must each be chosen from one of the 32 32-bit registers. The reason for the limit of 32 registers may be found in the second of our three underlying design principles of hardware technology:

Design Principle 2: Smaller is faster.

A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther. Guidelines such as “smaller is faster” are not absolutes; 31 registers may not be faster than 32. Yet, the truth behind such observations causes computer designers to take them seriously. In this case, the designer must balance the craving of programs for more registers with the designer’s desire to keep the clock cycle fast. Another reason for not using more than 32 is the number of bits it would take in the instruction format, as Section 2.5 demonstrates.

Compiling a C Assignment Using Registers

It is the compiler’s job to associate program variables with registers. Take, for instance, the assignment statement from our earlier example:

```
f = (g + h) - (i + j);
```

The variables *f*, *g*, *h*, *i*, and *j* are assigned to the registers *\$s0*, *\$s1*, *\$s2*, *\$s3*, and *\$s4*, respectively. What is the compiled MIPS code?

The compiled program is very similar to the prior example, except we replace the variables with the register names mentioned above plus two temporary registers, \$t0 and \$t1, which correspond to the temporary variables above:

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```

Memory Operands

Recall the five components of a computer introduced in Chapter 1 and repeated on page 61. The processor can keep only a small amount of data in registers, but computer memory contains billions of data elements. Hence, data structures (arrays and structures) are kept in memory.

As explained above, arithmetic operations occur only on registers in MIPS instructions; thus, MIPS must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**. To access a word in memory, the instruction must supply the memory **address**. Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0. For example, in Figure 2.2, the address of the third data element is 2, and the value of Memory [2] is 10.

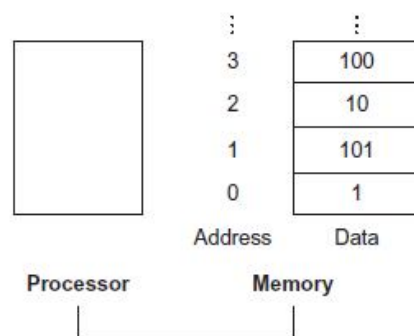


FIGURE 2.2 Memory addresses and contents of memory at those locations. If these elements were words, these addresses would be incorrect, since MIPS actually uses byte addressing, with each word representing four bytes. Figure 2.3 shows the memory addressing for sequential word addresses.

Logical Operations

Although the first computers operated on full words, it soon became clear that it was useful to operate on fields of bits within a word or even on individual bits. Examining characters within a word, each of which is stored as 8 bits, is one example of such an operation (see Section 2.9). It follows that operations were added to programming languages and instruction set architectures to simplify, among other things, the packing and unpacking of bits into words. These instructions are called logical operations. Figure 2.8 shows logical operations in C, Java, and MIPS.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

FIGURE 2.8 C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

The first class of such operations is called *shift s*. They move all the bits in a word to the left or right, filling the emptied bits with 0s. For example, if register \$s0 contained

0000 0000 0000 0000 0000 0000 0000 1001two = 9ten

and the instruction to shift left by 4 was executed, the new value would be:

0000 0000 0000 0000 0000 0000 1001 0000two = 144ten

The dual of a shift left is a shift right. The actual name of the two MIPS shift instructions are called *shift left logical* (sll) and *shift right logical* (srl). The following instruction performs the operation above, assuming that the original value was in register \$s0 and the result should go in register \$t2:

sll \$t2,\$s0,4 # reg \$t2 = reg \$s0 << 4 bits

We delayed explaining the *shamt* field in the R-format. Used in shift instructions, it stands for *shift amount*. Hence, the machine language version of the instruction above is

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

Instructions for Making Decisions

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels. MIPS assembly language includes two decision-making instructions, similar to an *if* statement with a *go to*. The first instruction is

beq register1, register2, L1

This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic beq stands for *branch if equal*. The second instruction is

bne register1, register2, L1

It means go to the statement labeled L1 if the value in register1 does *not* equal the value in register2. The mnemonic bne stands for *branch if not equal*. These two instructions are traditionally called **conditional branches**.

Compiling *If-then-else* Into Conditional Branches

In the following code segment, f, g, h, i, and j are variables. If the five variables f through j correspond to the five registers \$s0 through \$s4, what is the compiled MIPS code for this C *if* statement?

```
if (i == j) f = g + h; else f = g - h;
```

Figure 2.9 shows a flowchart of what the MIPS code should do. The first expression compares for equality, so it would seem that we would want the branch if registers are equal instruction (`beq`). In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the *if* (the label `Else` is defined below) and so we use the branch if registers are *not* equal instruction (`bne`):

```
bne $s3,$s4,Else # go to Else if i ≠ j
```

The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

```
add $s0,$s1,$s2 # f = g + h (skipped if i ≠ j)
```

We now need to go to the end of the *if* statement. This example introduces another kind of branch, often called an *unconditional branch*. This instruction says that the processor always follows the branch. To distinguish between conditional and unconditional branches, the MIPS name for this type of instruction is *jump*, abbreviated as `j` (the label `Exit` is defined below).

```
j Exit # go to Exit
```

The assignment statement in the *else* portion of the *if* statement can again be compiled into a single instruction. We just need to append the label `Else` to this instruction. We also show the label `Exit` that is after this instruction, showing the end of the *if-then-else* compiled code:

```
Else:sub $s0,$s1,$s2 # f = g - h (skipped if i = j)
Exit:
```

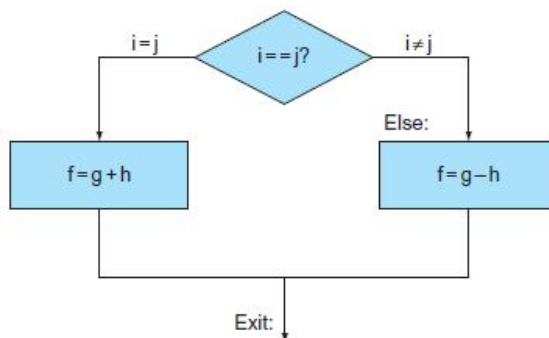


FIGURE 2.9 Illustration of the options in the *if* statement above. The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

Loops

Decisions are important both for choosing between two alternatives—found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.

Compiling a *while* Loop In C

Here is a traditional loop in C:

```
while (save[i] == k)
    i += 1;
```

Assume that *i* and *k* correspond to registers *\$s3* and *\$s5* and the base of the array *save* is in *\$s6*. What is the MIPS assembly code corresponding to this C segment?

The first step is to load *save[i]* into a temporary register. Before we can load *save[i]* into a temporary register, we need to have its address. Before we can add *i* to the base of array *save* to form the address, we must multiply the index *i* by 4 due to the byte addressing problem. Fortunately, we can use shift left logical, since shifting left by 2 bits multiplies by 2^2 or 4 (see page 88 in the prior section). We need to add the label *Loop* to it so that we can branch back to that instruction at the end of the loop:

```
Loop: sll $t1,$s3,2    # Temp reg $t1 = i * 4
```

To get the address of *save[i]*, we need to add *\$t1* and the base of *save* in *\$s6*:

```
add $t1,$t1,$s6    # $t1 = address of save[i]
```

Now we can use that address to load *save[i]* into a temporary register:

```
lw $t0,0($t1)    # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if *save[i] ≠ k*:

```
bne $t0,$s5, Exit # go to Exit if save[i] ≠ k
```

The next instruction adds 1 to *i*:

```
addi $s3,$s3,1    # i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
    j    Loop      # go to Loop
Exit:
```

(See the exercises for an optimization of this sequence.)

Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table** or **jump table**, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the jump table into a register. It then needs to jump using the address in the register. To support such situations, computers like MIPS include a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction. We'll see an even more popular use of `jr` in the next section.

MIPS Addressing for 32-bit Immediates and Addresses

Although keeping all MIPS instructions 32 bits long simplifies the hardware, there are times where it would be convenient to have a 32-bit constant or 32-bit address. This section starts with the general solution for large constants, and then shows the optimizations for instruction addresses used in branches and jumps.

32-Bit Immediate Operands

Although constants are frequently short and fit into the 16-bit field, sometimes they are bigger. The MIPS instruction set includes the instruction *load upper immediate* (`lui`) specifically to set the upper 16 bits of a constant in a register, allowing a subsequent instruction to specify the lower 16 bits of the constant. [Figure 2.17](#) shows the operation of `lui`.

Loading a 32-Bit Constant

What is the MIPS assembly code to load this 32-bit constant into register `$s0`?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

First, we would load the upper 16 bits, which is 61 in decimal, using `lui`:

```
lui $s0, 61    # 61 decimal = 0000 0000 0011 1101 binary
```

The value of register `$s0` afterward is

```
0000 0000 0011 1101 0000 0000 0000 0000
```

The next step is to insert the lower 16 bits, whose decimal value is 2304:

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

The final value in register `$s0` is the desired value:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

The machine language version of `lui $t0, 255 # $t0 is register 8:`

001111	00000	01000	0000 0000 1111 1111
--------	-------	-------	---------------------

Contents of register `$t0` after executing `lui $t0, 255:`

0000 0000 1111 1111	0000 0000 0000 0000
---------------------	---------------------

FIGURE 2.17 The effect of the `lui` instruction. The instruction `lui` transfers the 16-bit immediate constant field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s.

Addressing in Branches and Jumps

The MIPS jump instructions have the simplest addressing. They use the final MIPS instruction format, called the *J-type*, which consists of 6 bits for the operation field and the rest of the bits for the address field. Thus,

```
j 10000 # go to location 10000
```

could be assembled into this format (it's actually a bit more complicated, as we will see):

2	10000
6 bits	26 bits

where the value of the jump opcode is 2 and the jump address is 10000.

Unlike the jump instruction, the conditional branch instruction must specify two operands in addition to the branch address. Thus,

```
bne $s0,$s1,Exit # go to Exit if $s0 ≠ $s1
```

is assembled into this instruction, leaving only 16 bits for the branch address:

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

If addresses of the program had to fit in this 16-bit field, it would mean that no program could be bigger than 2^{16} , which is far too small to be a realistic option today. An alternative would be to specify a register that would always be added to the branch address, so that a branch instruction would calculate the following:

$$\text{Program counter} = \text{Register} + \text{Branch address}$$

This sum allows the program to be as large as 2^{32} and still be able to use conditional branches, solving the branch address size problem. Then the question is, which register?

The answer comes from seeing how conditional branches are used. Conditional branches are found in loops and in *if* statements, so they tend to branch to a nearby instruction. For example, about half of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away. Since the *program counter* (PC) contains the address of the current instruction, we can branch within $\pm 2^{15}$ words of the current instruction if we use the PC as the register to be added to the address. Almost all loops and *if* statements are much smaller than 2^{16} words, so the PC is the ideal choice.

This form of branch addressing is called **PC-relative addressing**. As we shall see in Chapter 4, it is convenient for the hardware to increment the PC early to point to the next instruction. Hence, the MIPS address is actually relative to the address of the following instruction (PC + 4) as opposed to the current instruction (PC). It is yet another example of making the common case fast, which in this case is addressing nearby instructions.

Showing Branch Offset In Machine Language

The *while* loop on pages 92–93 was compiled into this MIPS assembler code:

```

Loop: sll $t1,$s3,2      # Temp reg $t1 = 4 * i
      add $t1,$t1,$s6    # $t1 = address of save[i]
      lw  $t0,0($t1)    # Temp reg $t0 = save[i]
      bne $t0,$s5, Exit # go to Exit if save[i] ≠ k
      addi $s3,$s3,1    # i = i + 1
      j   Loop         # go to Loop
Exit:

```

If we assume we place the loop starting at location 80000 in memory, what is the MIPS machine code for this loop?

The assembled instructions and their addresses are:

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

MIPS Addressing Mode Summary

Multiple forms of addressing are generically called **addressing modes**. Figure 2.18 shows how operands are identified for each addressing mode. The MIPS addressing modes are the following:

1. *Immediate addressing*, where the operand is a constant within the instruction itself
2. *Register addressing*, where the operand is a register
3. *Base or displacement addressing*, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
4. *PC-relative addressing*, where the branch address is the sum of the PC and a constant in the instruction
5. *Pseudodirect addressing*, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

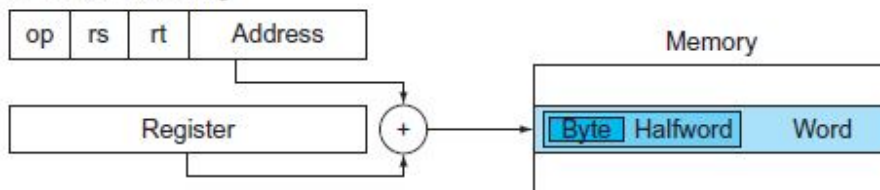
1. Immediate addressing



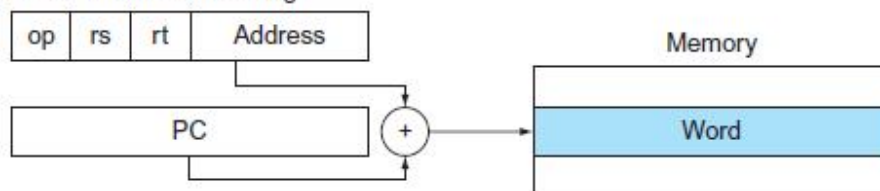
2. Register addressing



3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing

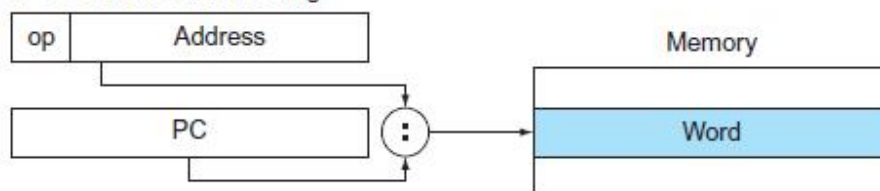


FIGURE 2.18 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC. Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (*addi*) and register (*add*) addressing.

UNIT II ARITHMETIC OPERATIONS

Introduction

Computer words are composed of bits; thus, words can be represented as binary numbers. Chapter 2 shows that integers can be represented either in decimal or binary form, but what about the other numbers that commonly occur? For example:

- What about fractions and other real numbers?
- What happens if an operation creates a number bigger than can be represented?
- And underlying these questions is a mystery: How does hardware really multiply or divide numbers?

The goal of this chapter is to unravel these mysteries including representation of real numbers, arithmetic algorithms, hardware that follows these algorithms, and the implications of all this for instruction sets. These insights may explain quirks that you have already encountered with computers. Moreover, we show how to use this knowledge to make arithmetic-intensive programs go much faster.

Addition and Subtraction

Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: the appropriate operand is simply negated before being added.

Binary Addition and Subtraction

Let's try adding 6_{ten} to 7_{ten} in binary and then subtracting 6_{ten} from 7_{ten} in binary.

$$\begin{array}{r} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\ + \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\ \hline = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{\text{two}} = 13_{\text{ten}} \end{array}$$

The 4 bits to the right have all the action; [Figure 3.1](#) shows the sums and carries. The carries are shown in parentheses, with the arrows showing how they are passed.

Subtracting 6_{ten} from 7_{ten} can be done directly:

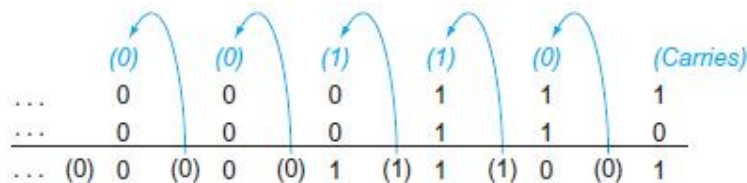


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is $0 + 1 + 1$. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of $1 + 1 + 1$, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is $1 + 0 + 0$, yielding a 1 sum and no carry.

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
 -\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\
 \hline
 =\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}
 \end{array}$$

or via addition using the two's complement representation of -6 :

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
 +\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{\text{two}} = -6_{\text{ten}} \\
 \hline
 =\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}
 \end{array}$$

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 3.2 Overflow conditions for addition and subtraction.

We have just seen how to detect overflow for two's complement numbers in a computer. What about overflow with unsigned integers? Unsigned integers are commonly used for memory addresses where overflows are ignored.

The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

- Add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow.
- Add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do *not* cause exceptions on overflow.

Because C ignores overflows, the MIPS C compilers will always generate the unsigned versions of the arithmetic instructions `addu`, `addiu`, and `subu`, no matter what the type of the variables. The MIPS Fortran compilers, however, pick the appropriate arithmetic instructions, depending on the type of the operands.

Appendix B describes the hardware that performs addition and subtraction, which is called an **Arithmetic Logic Unit** or **ALU**.

Multiplication

Now that we have completed the explanation of addition and subtraction, we are ready to build the more vexing operation of multiplication.

First, let's review the multiplication of decimal numbers in longhand to remind ourselves of the steps of multiplication and the names of the operands. For reasons that will become clear shortly, we limit this decimal example to using only the digits 0 and 1. Multiplying 1000_{ten} by 1001_{ten} :

$$\begin{array}{r}
 \text{Multiplicand} \quad 1000_{\text{ten}} \\
 \text{Multiplier} \quad \times \quad 1001_{\text{ten}} \\
 \hline
 1000 \\
 0000 \\
 0000 \\
 1000 \\
 \hline
 \text{Product} \quad 1001000_{\text{ten}}
 \end{array}$$

The first operand is called the *multiplicand* and the second the *multiplier*. The final result is called the *product*. As you may recall, the algorithm learned in grammar school is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier, and shifting the intermediate product one digit to the left of the earlier intermediate products.

In this example, we restricted the decimal digits to 0 and 1. With only two choices, each step of the multiplication is simple:

1. Just place a copy of the multiplicand ($1 \times$ multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 ($0 \times$ multiplicand) in the proper place if the digit is 0.

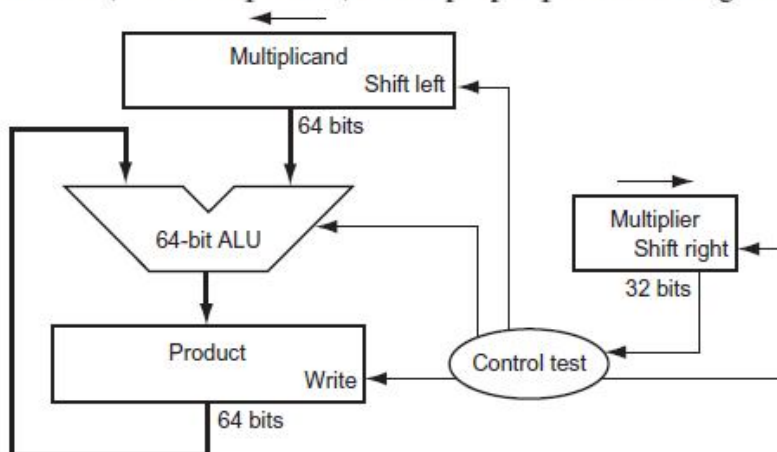


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix B describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

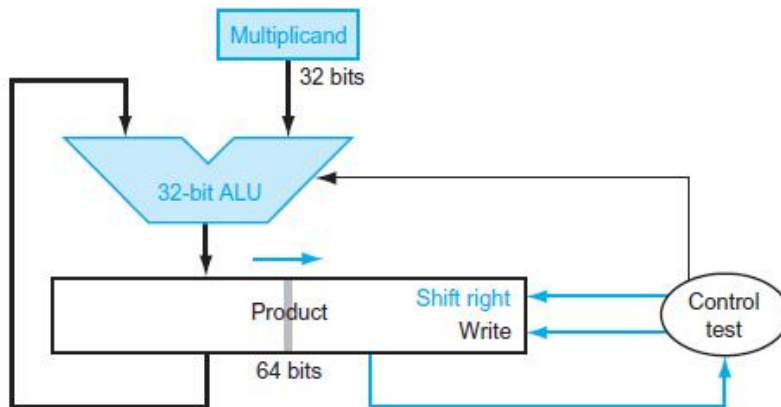


FIGURE 3.5 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it's shown here as 64 bits to highlight the evolution from Figure 3.3.)

Division

The reciprocal operation of multiply is divide, an operation that is even less frequent and even more quirky. It even offers the opportunity to perform a mathematically invalid operation: dividing by 0.

Let's start with an example of long division using decimal numbers to recall the names of the operands and the grammar school division algorithm. For reasons similar to those in the previous section, we limit the decimal digits to just 0 or 1. The example is dividing $1,001,010_{\text{ten}}$ by 1000_{ten} :

$$\begin{array}{r}
 \text{Quotient } 1001_{\text{ten}} \\
 \text{Divisor } 1000_{\text{ten}} \overline{) 1001010_{\text{ten}}} \\
 \underline{-1000} \\
 10 \\
 101 \\
 1010 \\
 \underline{-1000} \\
 10_{\text{ten}} \\
 \text{Remainder}
 \end{array}$$

Division's two operands, called the **dividend** and **divisor**, and the result, called the **quotient**, are accompanied by a second result, called the **remainder**. Here is another way to express the relationship between the components:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

A Division Algorithm and Hardware

Figure 3.8 shows hardware to mimic our grammar school algorithm. We start with the 32-bit Quotient register set to 0. Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

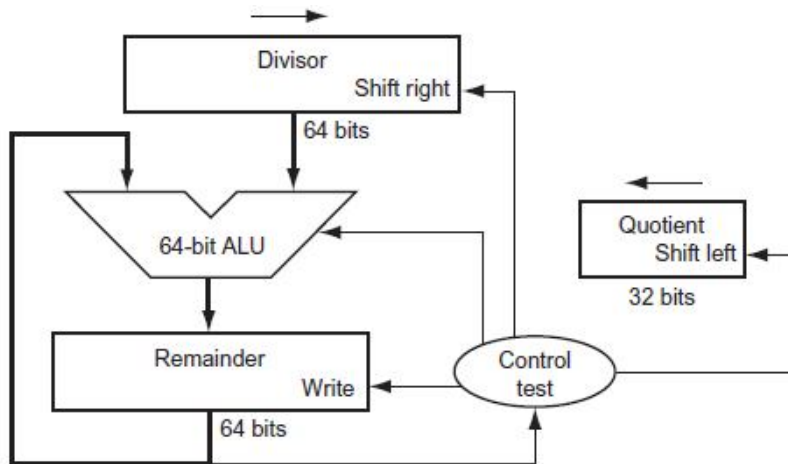


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

Figure 3.9 shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed the comparison in the set on less than instruction. If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a). If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b). The divisor is shifted right and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations are complete.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	⓪110 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem - Div	0000	0001 0000	⓪111 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	⓪111 1111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem - Div	0000	0000 0100	⓪000 0011
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem - Div	0001	0000 0010	⓪000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

Floating Point

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called *reals* in mathematics. Here are some examples of reals:

$3.14159265\dots_{\text{ten}}$ (π)

$2.71828\dots_{\text{ten}}$ (e)

0.000000001_{ten} or $1.0_{\text{ten}} \times 10^{-9}$ (seconds in a nanosecond)

$3,155,760,000_{\text{ten}}$ or $3.15576_{\text{ten}} \times 10^9$ (seconds in a typical century)

Notice that in the last case, the number didn't represent a small fraction, but it was bigger than we could represent with a 32-bit signed integer. The alternative notation for the last two numbers is called **scientific notation**, which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a **normalized** number, which is the usual way to write it. For example, $1.0_{\text{ten}} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{\text{ten}} \times 10^{-8}$ and $10.0_{\text{ten}} \times 10^{-10}$ are not.

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{\text{two}} \times 2^{-1}$$

Floating-Point Representation

A designer of a floating-point representation must find a compromise between the size of the **fraction** and the size of the **exponent**, because a fixed word size means you must take a bit from one to add a bit to the other. This tradeoff is between precision and range: increasing the size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented. As our design guideline from Chapter 2 reminds us, good design demands good compromise.

Floating-point numbers are usually a multiple of the size of a word. The representation of a MIPS floating-point number is shown below, where s is the sign of the floating-point number (1 meaning negative), $exponent$ is the value of the 8-bit exponent field (including the sign of the exponent), and $fraction$ is the 23-bit number. As we recall from Chapter 2, this representation is *sign and magnitude*, since the sign is a separate bit from the rest of the number.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent								fraction																						
1 bit	8 bits								23 bits																						

In general, floating-point numbers are of the form

$$(-1)^S \times F \times 2^E$$

F involves the value in the fraction field and E involves the value in the exponent field; the exact relationship to these fields will be spelled out soon. (We will shortly see that MIPS does something slightly more sophisticated.)

fraction The value, generally between 0 and 1, placed in the fraction field. The fraction is also called the *mantissa*.

exponent In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

FIGURE 3.13 IEEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the *Elaboration* on page 222. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Thus $00 \dots 00_{\text{two}}$ represents 0; the representation of the rest of the numbers uses the form from before with the hidden 1 added:

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$

where the bits of the fraction represent a number between 0 and 1 and E specifies the value in the exponent field, to be given in detail shortly. If we number the bits of the fraction from *left to right* s_1, s_2, s_3, \dots , then the value is

$$(-1)^S \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + (s_4 \times 2^{-4}) + \dots) \times 2^E$$

EXAMPLE

Converting Binary to Decimal Floating Point

What decimal number is represented by this single precision float?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

ANSWER

The sign bit is 1, the exponent field contains 129, and the fraction field contains $1 \times 2^{-2} = 1/4$, or 0.25. Using the basic equation,

$$\begin{aligned} (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129 - 127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= -5.0 \end{aligned}$$

Floating-Point Addition

Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{\text{ten}} \times 10^1 + 1.610_{\text{ten}} \times 10^{-1}$. Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

- Step 1. To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number, $1.610_{\text{ten}} \times 10^{-1}$, that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{\text{ten}} \times 10^{-1} = 0.1610_{\text{ten}} \times 10^0 = 0.01610_{\text{ten}} \times 10^1$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{\text{ten}} \times 10^1$. Thus, the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really

$$0.016 \times 10^1$$

- Step 2. Next comes the addition of the significands:

$$\begin{array}{r} 9.999_{\text{ten}} \\ + 0.016_{\text{ten}} \\ \hline 10.015_{\text{ten}} \end{array}$$

The sum is $10.015_{\text{ten}} \times 10^1$.

- Step 3. This sum is not in normalized scientific notation, so we need to adjust it:

$$10.015_{\text{ten}} \times 10^1 = 1.0015_{\text{ten}} \times 10^2$$

Thus, after the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

- Step 4. Since we assumed that the significand can be only four digits long (excluding the sign), we must round the number. In our grammar school algorithm, the rules truncate the number if the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number

$$1.0015_{\text{ten}} \times 10^2$$

Floating-Point Multiplication

Now that we have explained floating-point addition, let's try floating-point multiplication. We start by multiplying decimal numbers in scientific notation by hand: $1.110_{\text{ten}} \times 10^{10} \times 9.200_{\text{ten}} \times 10^{-5}$. Assume that we can store only four digits of the significand and two digits of the exponent.

- Step 1. Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together:

$$\text{New exponent} = 10 + (-5) = 5$$

Let's do this with the biased exponents as well to make sure we obtain the same result: $10 + 127 = 137$, and $-5 + 127 = 122$, so

$$\text{New exponent} = 137 + 122 = 259$$

This result is too large for the 8-bit exponent field, so something is amiss! The problem is with the bias because we are adding the biases as well as the exponents:

$$\text{New exponent} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$

Accordingly, to get the correct biased sum when we add biased numbers, we must subtract the bias from the sum:

$$\text{New exponent} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

and 5 is indeed the exponent we calculated initially.

- Step 2. Next comes the multiplication of the significands:

$$\begin{array}{r} 1.110_{\text{ten}} \\ \times 9.200_{\text{ten}} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{\text{ten}} \end{array}$$

There are three digits to the right of the decimal point for each operand, so the decimal point is placed six digits from the right in the product significand:

$$10.212000_{\text{ten}}$$

Assuming that we can keep only three digits to the right of the decimal point, the product is 10.212×10^5 .

Step 3. This product is unnormalized, so we need to normalize it:

$$10.212_{\text{ten}} \times 10^5 = 1.0212_{\text{ten}} \times 10^6$$

Thus, after the multiplication, the product can be shifted right one digit to put it in normalized form, adding 1 to the exponent. At this point, we can check for overflow and underflow. Underflow may occur if both operands are small—that is, if both have large negative exponents.

Step 4. We assumed that the significand is only four digits long (excluding the sign), so we must round the number. The number

$$1.0212_{\text{ten}} \times 10^6$$

is rounded to four digits in the significand to

$$1.021_{\text{ten}} \times 10^6$$

Step 5. The sign of the product depends on the signs of the original operands. If they are both the same, the sign is positive; otherwise, it's negative. Hence, the product is

$$+1.021_{\text{ten}} \times 10^6$$

The sign of the sum in the addition algorithm was determined by addition of the significands, but in multiplication, the sign of the product is determined by the signs of the operands.

Parallelism and Computer Arithmetic: Subword Parallelism

Since every desktop microprocessor by definition has its own graphical displays, as transistor budgets increased it was inevitable that support would be added for graphics operations.

Many graphics systems originally used 8 bits to represent each of the three primary colors plus 8 bits for a location of a pixel. The addition of speakers and microphones for teleconferencing and video games suggested support of sound as well. Audio samples need more than 8 bits of precision, but 16 bits are sufficient.

Every microprocessor has special support so that bytes and halfwords take up less space when stored in memory (see Section 2.9), but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there was little support beyond data transfers. Architects recognized that many graphics and audio applications would perform the same operation on vectors of this data. By partitioning the carry chains within a 128-bit adder, a processor could use **parallelism** to perform simultaneous operations on short vectors of sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands. The cost of such partitioned adders was small.

Given that the parallelism occurs within a wide word, the extensions are classified as *subword parallelism*. It is also classified under the more general name of *data level parallelism*. They have been also called vector or SIMD, for single instruction, multiple data (see Section 6.6). The rising popularity of multimedia

Data transfer	Arithmetic	Logical/Compare
VLDL.F32	VADD.F32, VADD{L,W}{S8,U8,S16,U16,S32,U32}	VAND.64, VAND.128
VSTR.F32	VSUB.F32, VSUB{L,W}{S8,U8,S16,U16,S32,U32}	VORR.64, VORR.128
VLD{1,2,3,4}.{i8,i16,i32}	VMUL.F32, VMULL{S8,U8,S16,U16,S32,U32}	VEOR.64, VEOR.128
VST{1,2,3,4}.{i8,i16,i32}	VMLA.F32, VMLAL{S8,U8,S16,U16,S32,U32}	VBIC.64, VBIC.128
VMOV.{i8,i16,i32,F32}, #imm	VMLS.F32, VMLSL{S8,U8,S16,U16,S32,U32}	VORN.64, VORN.128
VMVN.{i8,i16,i32,F32}, #imm	VMAX.{S8,U8,S16,U16,S32,U32,F32}	VCEQ.{i8,i16,i32,F32}
VMOV.{i64,i128}	VMIN.{S8,U8,S16,U16,S32,U32,F32}	VCGE.{S8,U8,S16,U16,S32,U32,F32}
VMVN.{i64,i128}	VABS.{S8,S16,S32,F32}	VCGT.{S8,U8,S16,U16,S32,U32,F32}
	VNEG.{S8,S16,S32,F32}	VCLE.{S8,U8,S16,U16,S32,U32,F32}
	VSHL.{S8,U8,S16,U16,S32,S64,U64}	VCLT.{S8,U8,S16,U16,S32,U32,F32}
	VSHR.{S8,U8,S16,U16,S32,S64,U64}	VTST.{i8,i16,i32}

FIGURE 3.19 Summary of ARM NEON instructions for subword parallelism. We use the curly brackets {} to show optional variations of the basic operations: {S8,U8,8} stand for signed and unsigned 8-bit integers or 8-bit data where type doesn't matter, of which 16 fit in a 128-bit register; {S16,U16,16} stand for signed and unsigned 16-bit integers or 16-bit type-less data, of which 8 fit in a 128-bit register; {S32,U32,32} stand for signed and unsigned 32-bit integers or 32-bit type-less data, of which 4 fit in a 128-bit register; {S64,U64,64} stand for signed and unsigned 64-bit integers or type-less 64-bit data, of which 2 fit in a 128-bit register; {F32} stand for signed and unsigned 32-bit floating point numbers, of which 4 fit in a 128-bit register. Vector Load reads one n-element structure from memory into 1, 2, 3, or 4 NEON registers. It loads a single n-element structure to one lane (See Section 6.6), and elements of the register that are not loaded are unchanged. Vector Store writes one n-element structure into memory from 1, 2, 3, or 4 NEON registers.

UNIT III PROCESSOR AND CONTROL UNIT

Basic MIPS implementation

A Basic MIPS Implementation

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

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

This subset does not include all the integer instructions (for example, shift, multiply, and divide are missing), nor does it include any floating-point instructions.

An Overview of the Implementation

In Chapter 2, we looked at the core MIPS instructions, including the integer arithmetic-logical instructions, the memory-reference instructions, and the branch instructions. Much of what needs to be done to implement these instructions is the same, independent of the exact class of instruction. For every instruction, the first two steps are identical:

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

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction. The simplicity and regularity of the MIPS instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and branches for comparison. After using the ALU, the actions required to complete various instruction classes differ. A memory-reference instruction will need to access the memory either to read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register. Lastly, for a branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by 4 to get the address of the next instruction.

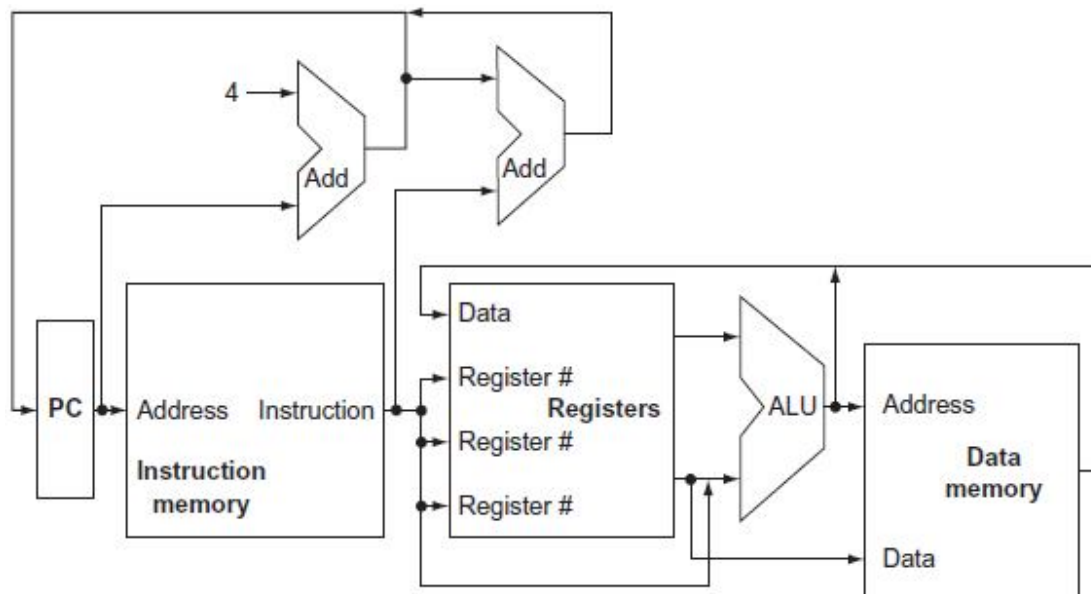
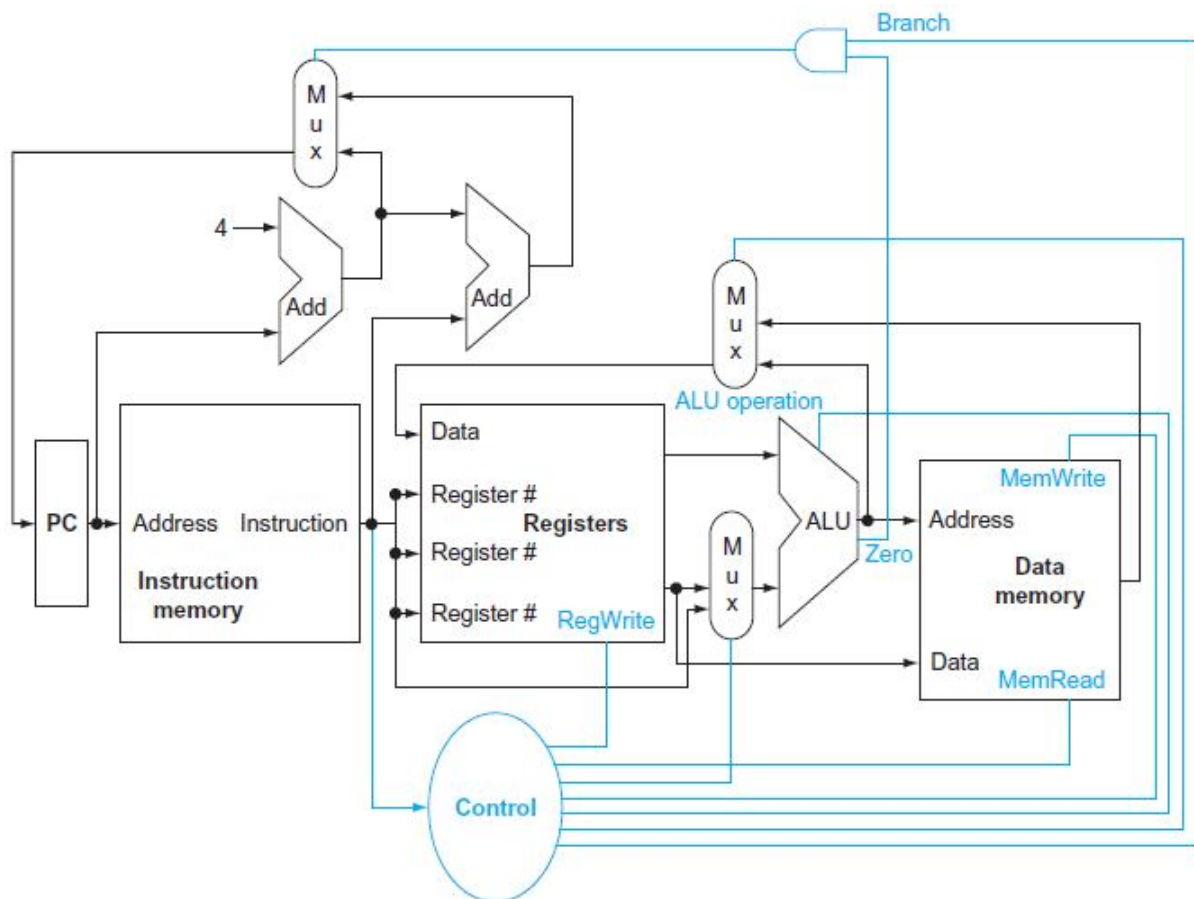


FIGURE 4.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

FIGURE 4.2 The basic implementation of the MIPS subset, including the necessary multiplexors and control lines.

The top multiplexor (“Mux”) controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that “ANDs” together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.



Building a Datapath

A reasonable way to start a datapath design is to examine the major components required to execute each class of MIPS instructions. Let’s start at the top by looking at which **datapath elements** each instruction needs, and then work our way down through the levels of **abstraction**. When we show the datapath elements, we will also show their control signals. We use abstraction in this explanation, starting from the bottom up.

Figure 4.5a shows the first element we need: a memory unit to store the instructions of a program and supply instructions given an address. Figure 4.5b also shows the **program counter (PC)**, which as we saw in Chapter 2 is a register that holds the address of the current instruction. Lastly, we will need an adder to increment the PC to the address of the next instruction. This adder, which is combinational, can be built from the ALU described in detail in Appendix B simply by wiring the control lines so that the control always specifies an add operation. We will draw such an ALU with the label *Add*, as in Figure 4.5, to indicate that it has been permanently made an adder and cannot perform the other ALU functions.

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later. Figure 4.6 shows how to combine the three elements from Figure 4.5 to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

Now let's consider the R-format instructions (see Figure 2.20 on page 120). They all read two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions either *R-type instructions* or *arithmetic-logical instructions* (since they perform arithmetic or logical operations). This instruction class includes `add`, `sub`, `AND`, `OR`, and `slt`,

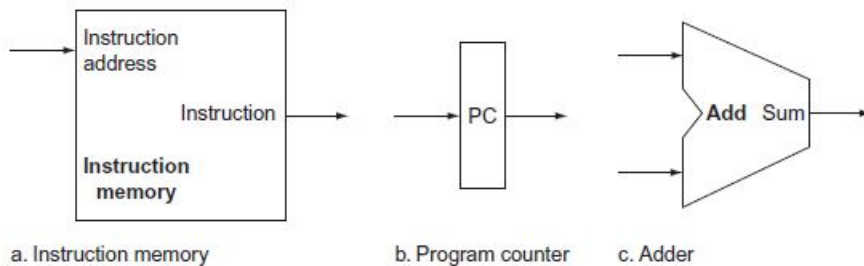


FIGURE 4.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.

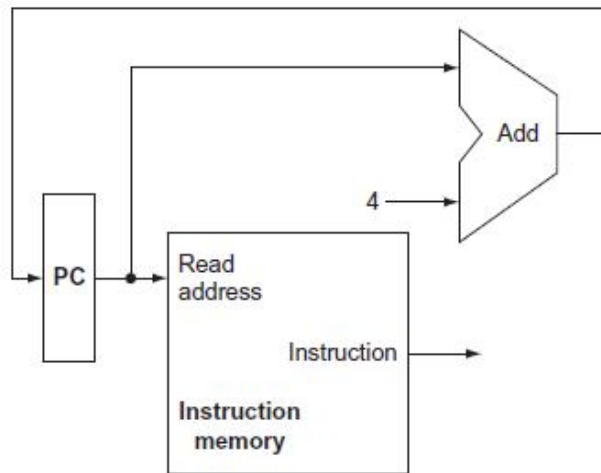


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

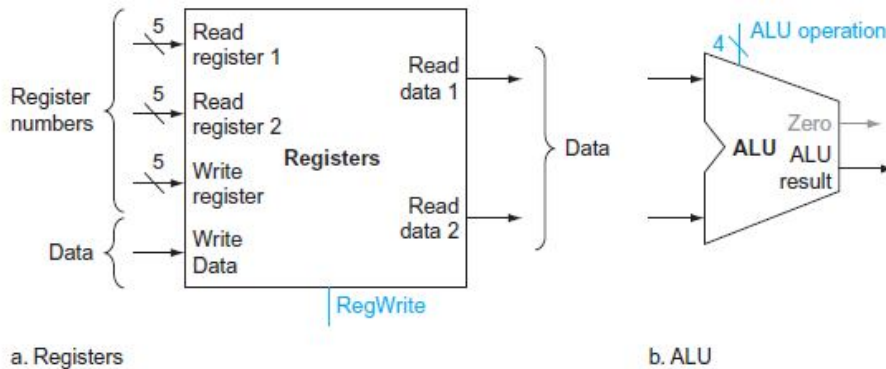


FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section B.8 of [Appendix B](#). The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in [Appendix B](#). We will use the Zero detection output of the ALU shortly to implement branches. The overflow output will not be needed until Section 4.9, when we discuss exceptions; we omit it until then.

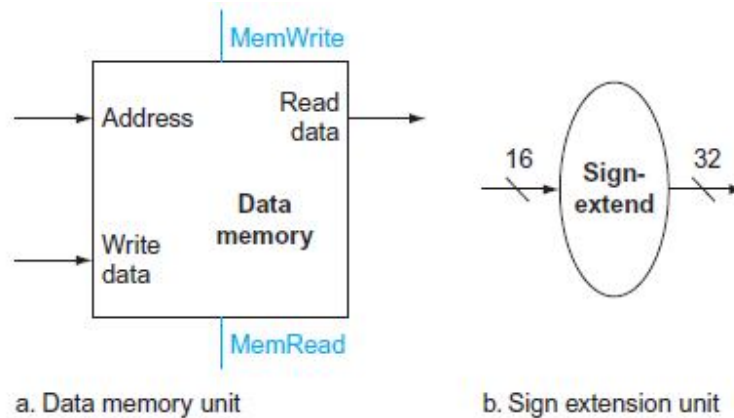


FIGURE 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the sign extension unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in Chapter 5. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output (see Chapter 2). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See Section B.8 of [Appendix B](#) for further discussion of how real memory chips work.

A Simple Implementation Scheme

The ALU Control

The MIPS ALU in [Appendix B](#) defines the 6 following combinations of four control inputs:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Depending on the instruction class, the ALU will need to perform one of these first five functions. (NOR is needed for other parts of the MIPS instruction set not found in the subset we are implementing.) For load word and store word instructions, we use the ALU to compute the memory address by addition. For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 4.12 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we “don’t care” about the value of the function code, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input. See [Appendix B](#).

Designing the Main Control Unit

Now that we have described how to design an ALU that uses the function code and a 2-bit signal as its control inputs, we can return to looking at the rest of the control. To start this process, let’s identify the fields of an instruction and the control lines that are needed for the datapath we constructed in [Figure 4.11](#). To understand how to connect the fields of an instruction to the datapath, it is useful to review

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don’t-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Note that when the function field is used, the first 2 bits (F5 and F4) of these instructions are always 10, so they are don’t-care terms and are replaced with XX in the truth table.

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

FIGURE 4.14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats. The jump instructions use another format, which we will discuss shortly. (a) Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. The ALU function is in the funct field and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are add, sub, AND, OR, and sllt. The shamt field is used only for shifts; we will ignore it in this chapter. (b) Instruction format for load (opcode = 35_{ten}) and store (opcode = 43_{ten}) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory. (c) Instruction format for branch equal (opcode = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC + 4 to compute the branch target address.

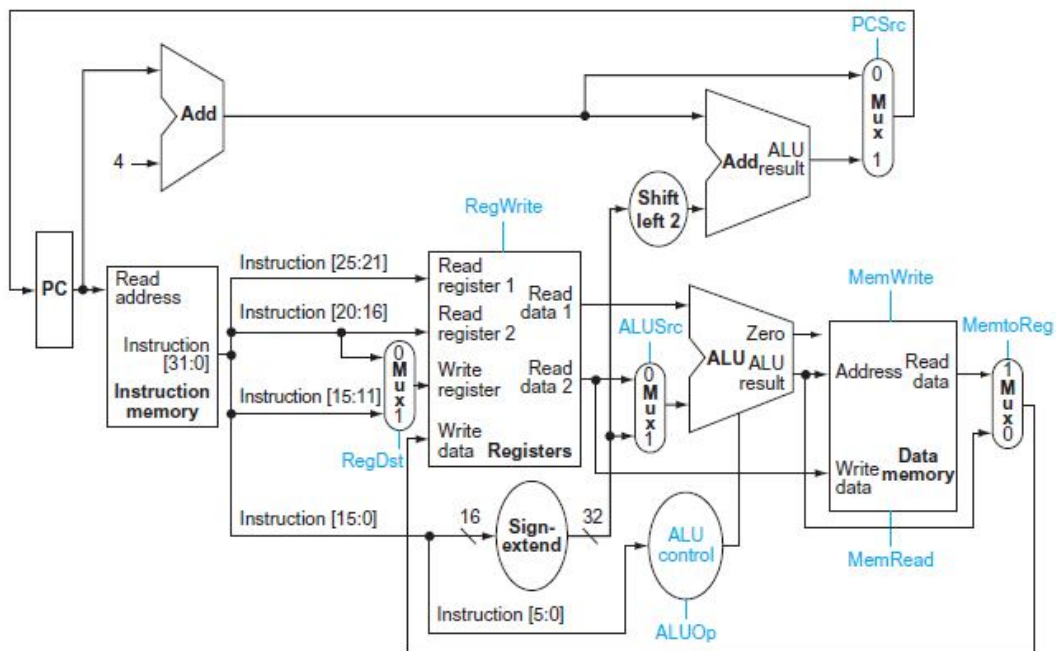


FIGURE 4.15 The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

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

FIGURE 4.16 The effect of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See [Appendix B](#) for further discussion of this problem.)

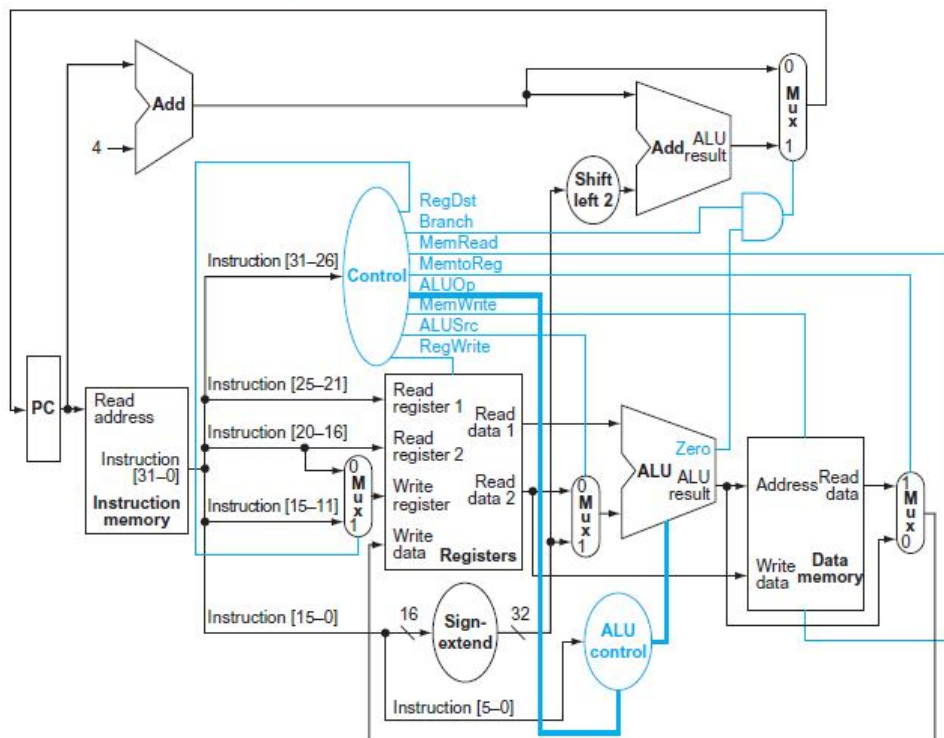


FIGURE 4.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures.

Finalizing Control

Now that we have seen how the instructions operate in steps, let's continue with the control implementation. The control function can be precisely defined using the contents of Figure 4.18. The outputs are the control lines, and the input is the 6-bit opcode field, Op [5:0]. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.

Figure 4.22 shows the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and we can implement it directly in gates in an automated fashion. We show this final step in Section D.2 in [Appendix D](#).

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

FIGURE 4.22 The control function for the simple single-cycle implementation is completely specified by this truth table. The top half of the table gives the combinations of input signals that correspond to the four opcodes, one per column, that determine the control output settings. (Remember that Op [5:0] corresponds to bits 31:26 of the instruction, which is the op field.) The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression $\overline{\text{Op5}} \cdot \overline{\text{Op2}}$, since this is sufficient to distinguish the R-format instructions from lw, sw, and beq. We do not take advantage of this simplification, since the rest of the MIPS opcodes are used in a full implementation.

An Overview of Pipelining

The same principles apply to processors where we pipeline instruction-execution. MIPS instructions classically take five steps:

1. Fetch instruction from memory.
2. Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.
3. Execute the operation or calculate an address.
4. Access an operand in data memory.
5. Write the result into a register.

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Under ideal conditions and with a large number of instructions, the speed-up from pipelining is approximately equal to the number of pipe stages; a five-stage pipeline is nearly five times faster.

The formula suggests that a five-stage pipeline should offer nearly a fivefold improvement over the 800 ps nonpipelined time, or a 160 ps clock cycle. The example shows, however, that the stages may be imperfectly balanced. Moreover, pipelining involves some overhead, the source of which will be clearer shortly. Thus, the time per instruction in the pipelined processor will exceed the minimum possible, and speed-up will be less than the number of pipeline stages.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

FIGURE 4.26 Total time for each instruction calculated from the time for each component. This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

Designing Instruction Sets for Pipelining

Even with this simple explanation of pipelining, we can get insight into the design of the MIPS instruction set, which was designed for pipelined execution.

First, all MIPS instructions are the same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage. In an instruction set like the x86, where instructions vary from 1 byte to 15 bytes, pipelining is considerably more challenging. Recent implementations of the x86 architecture actually translate x86 instructions into simple operations that look like MIPS instructions and then pipeline the simple operations rather than the native x86 instructions! (See Section 4.10.)

Second, MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction. This symmetry means that the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched. If MIPS instruction formats were not symmetric, we would need to split stage 2, resulting in six pipeline stages. We will shortly see the downside of longer pipelines.

Third, memory operands only appear in loads or stores in MIPS. This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage. If we could operate on the operands in memory, as in the x86, stages 3 and 4 would expand to an address stage, memory stage, and then execute stage.

Fourth, as discussed in Chapter 2, operands must be aligned in memory. Hence, we need not worry about a single data transfer instruction requiring two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage.

Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*, and there are three different types.

Hazards

The first hazard is called a **structural hazard**. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle. A structural hazard in the laundry room would occur if we used a washer-dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

Data Hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Suppose you found a sock at the folding station for which no match existed. One possible strategy is to run down to your room and search through your clothes bureau to see if you can find the match. Obviously, while you are doing the search, loads must wait that have completed drying and are ready to fold as well as those that have finished washing and are ready to dry.

In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry). For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum (\$s0):

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.

Although we could try to rely on compilers to remove all such hazards, the results would not be satisfactory. These dependences happen just too often and the delay is just too long to expect the compiler to rescue us from this dilemma.

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.

Forwarding with Two Instructions

For the two instructions above, show what pipeline stages would be connected by forwarding. Use the drawing in [Figure 4.28](#) to represent the datapath during the five stages of the pipeline. Align a copy of the datapath for each instruction, similar to the laundry pipeline in [Figure 4.25](#).

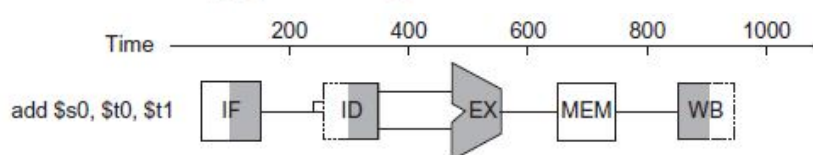


FIGURE 4.28 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in [Figure 4.25](#). Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write-back stage, with the drawing showing the register file being written. The shading indicates the element is used by the instruction. Hence, *MEM* has a white background because *add* does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of *ID* is shaded in the second stage because the register file is read, and the left half of *WB* is shaded in the fifth stage because the register file is written.

Pipelined Datapath and Control

Figure 4.33 shows the single-cycle datapath from Section 4.4 with the pipeline stages identified. The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus, we must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

In Figure 4.33, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the

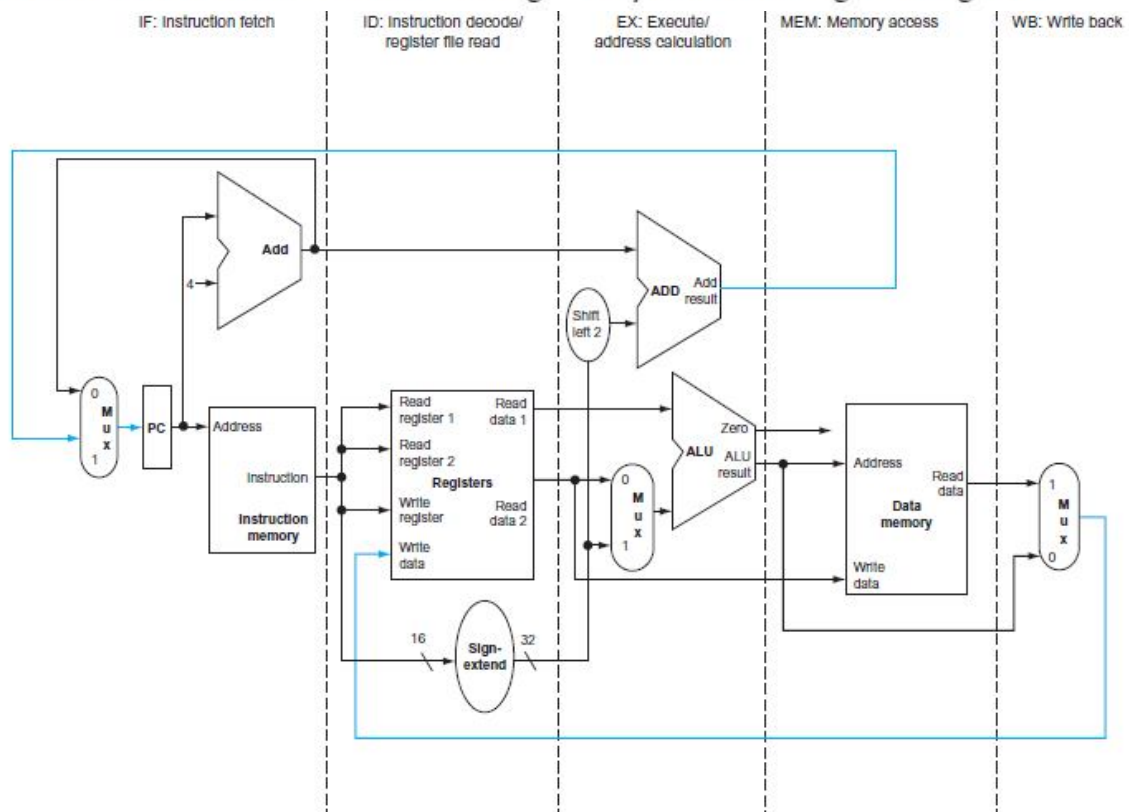


FIGURE 4.33 The single-cycle datapath from Section 4.4 (similar to Figure 4.17). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

five stages as they complete execution. Returning to our laundry analogy, clothes get cleaner, drier, and more organized as they move through the line, and they never move backward.

There are, however, two exceptions to this left-to-right flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the datapath
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage

Data flowing from right to left does not affect the current instruction; these reverse data movements influence only later instructions in the pipeline. Note that

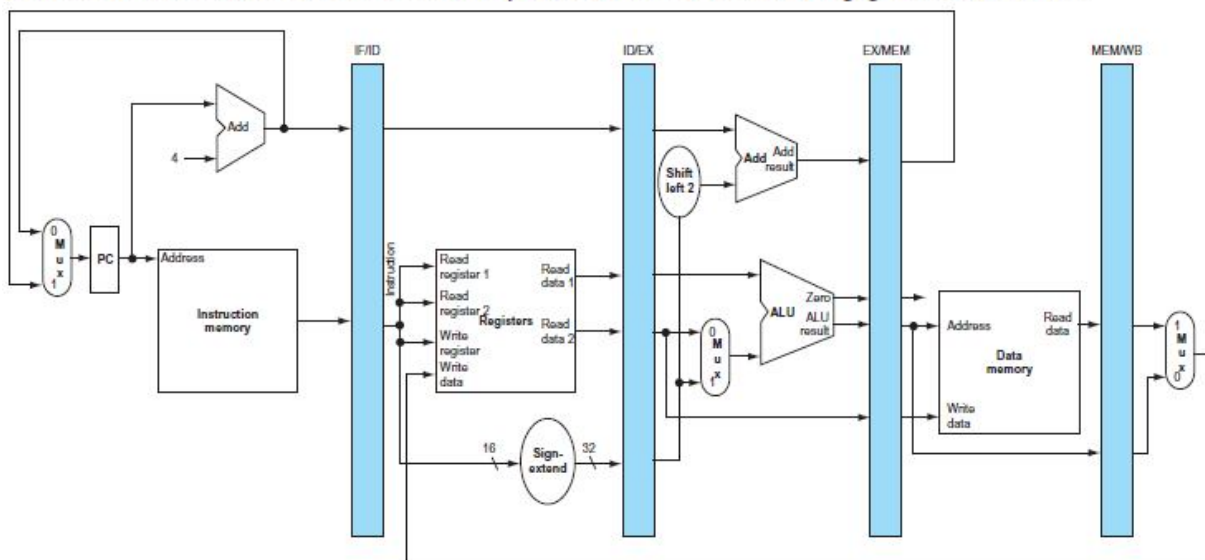


FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the *IF/ID* register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

We show the instruction abbreviation $\downarrow w$ with the name of the pipe stage that is active in each figure. The five stages are the following:

1. *Instruction fetch*: The top portion of Figure 4.36 shows the instruction being read from memory using the address in the PC and then being placed in the *IF/ID* pipeline register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the *IF/ID* pipeline register in case it is needed later for an instruction, such as *beq*. The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

2. *Instruction decode and register file read:* The bottom portion of Figure 4.36 shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.
3. *Execute or address calculation:* Figure 4.37 shows that the load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register.
4. *Memory access:* The top portion of Figure 4.38 shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.
5. *Write-back:* The bottom portion of Figure 4.38 shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

Data Hazards: Forwarding versus Stalling

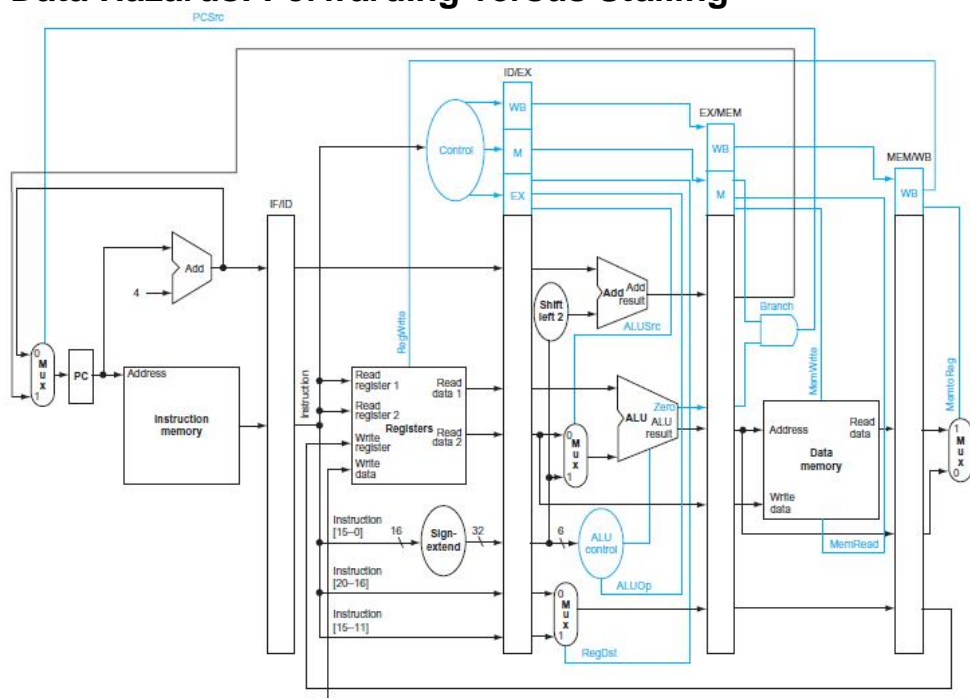


FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Let's look at a sequence with many dependences, shown in color:

```

sub   $2, $1, $3      # Register $2 written by sub
and   $12, $2, $5     # 1st operand($2) depends on sub
or    $13, $6, $2     # 2nd operand($2) depends on sub
add   $14, $2, $2     # 1st($2) & 2nd($2) depend on sub
sw    $15, 100($2)    # Base ($2) depends on sub

```

The last four instructions are all dependent on the result in register \$2 of the first instruction. If register \$2 had the value 10 before the subtract instruction and -20 afterwards, the programmer intends that -20 will be used in the following instructions that refer to register \$2.

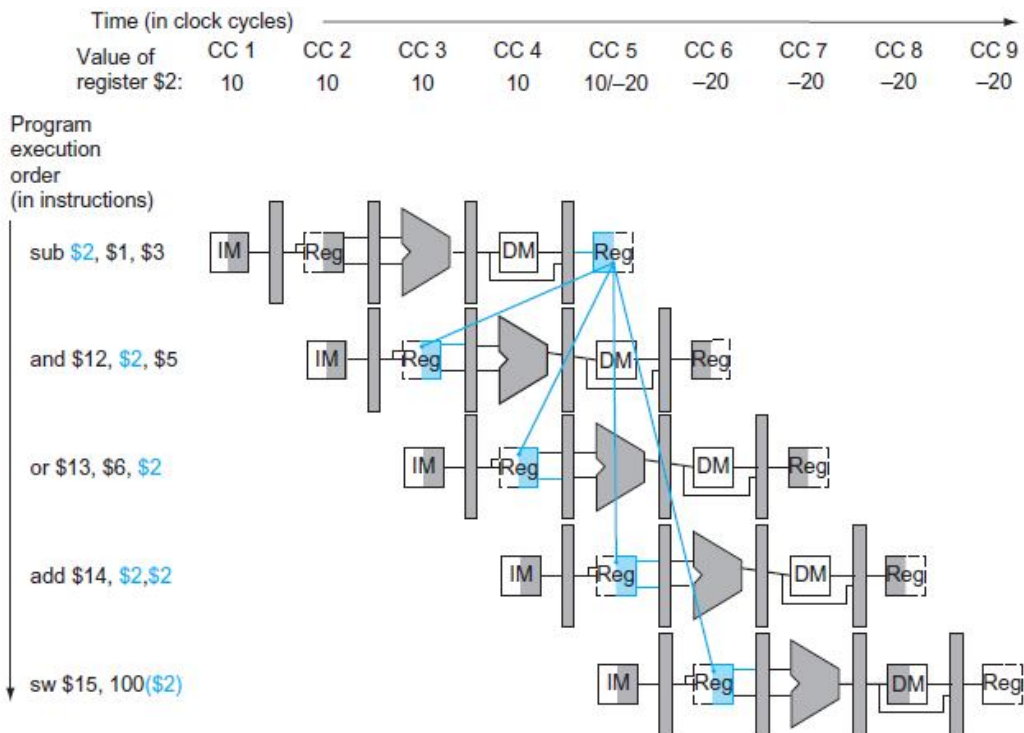


FIGURE 4.52 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and “CC 1” at the top of the figure means clock cycle 1. The first instruction writes into \$2, and all the following instructions read \$2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are *pipeline data hazards*.

Data Hazards and Stalls

As we said in Section 4.5, one case where forwarding cannot save the day is when an instruction tries to read a register following a load instruction that writes the same register. Figure 4.58 illustrates the problem. The data is still being read from memory in clock cycle 4 while the ALU is performing the operation for the following instruction. Something must stall the pipeline for the combination of load followed by an instruction that reads its result.

Hence, in addition to a forwarding unit, we need a *hazard detection unit*. It operates during the ID stage so that it can insert the stall between the load and its

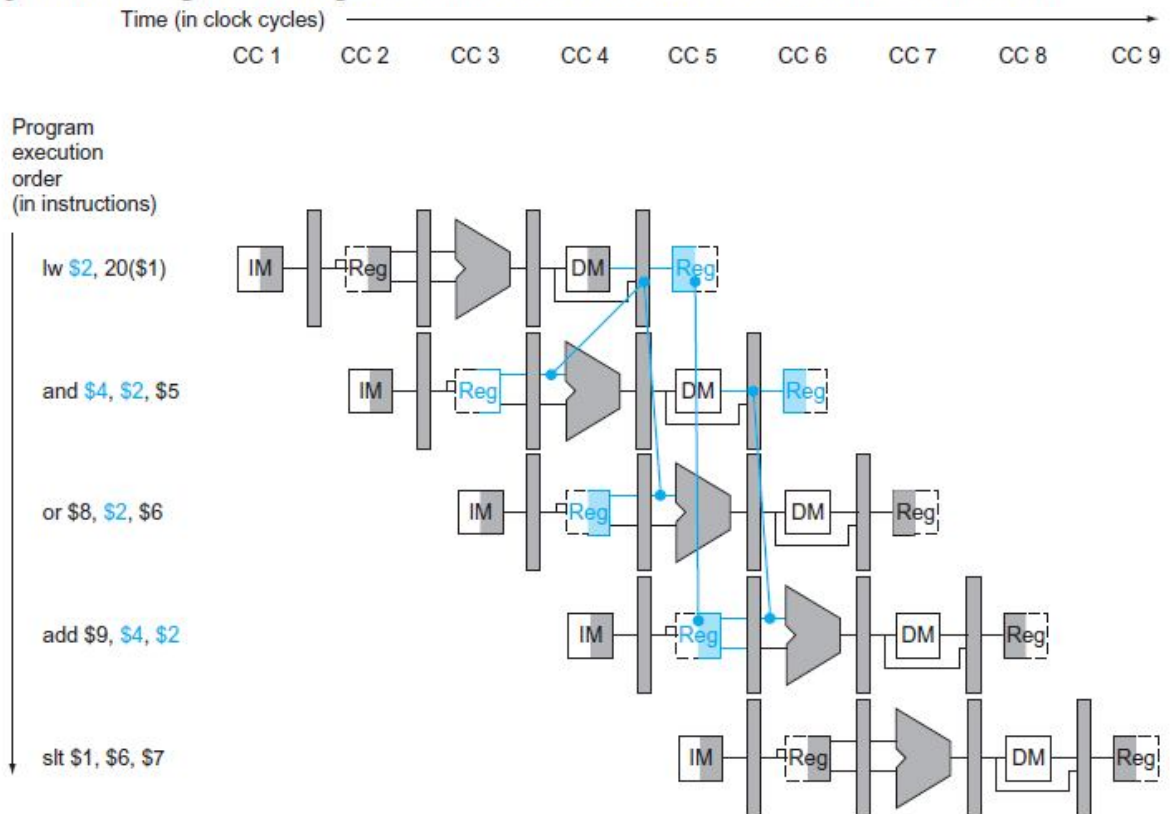


FIGURE 4.58 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (`and`) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

Control Hazards

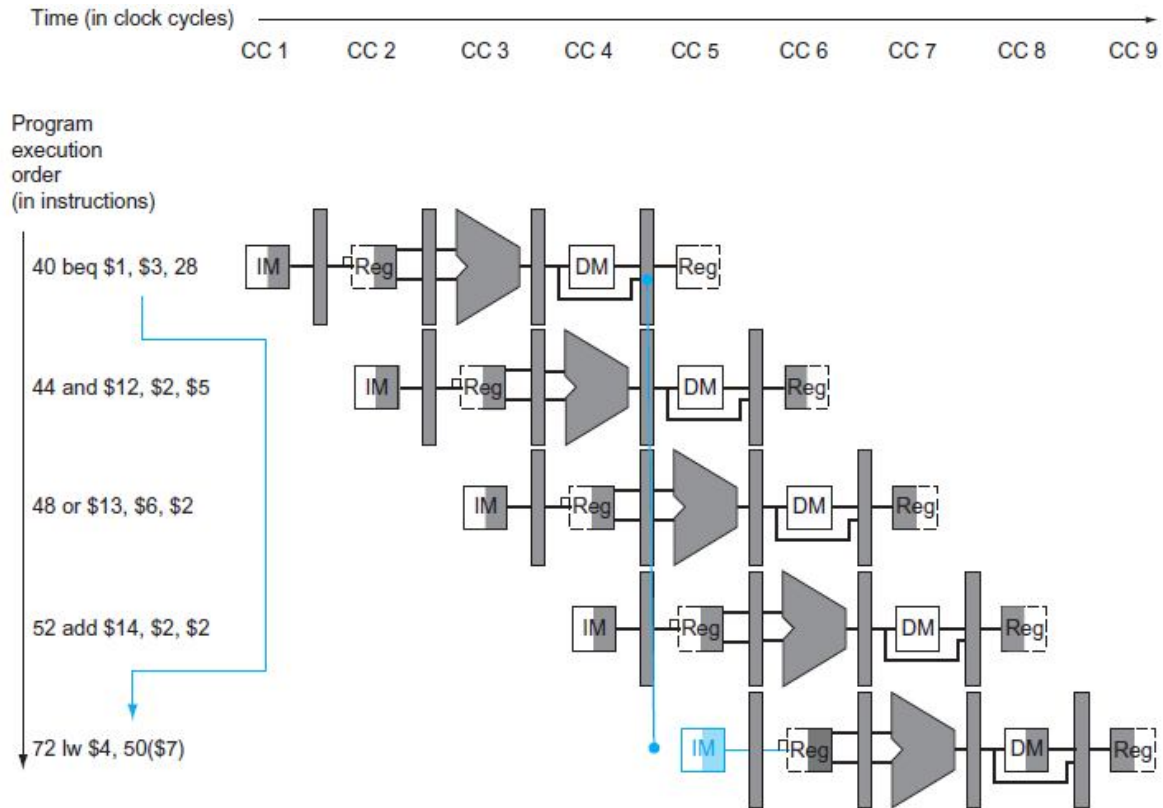


FIGURE 4.61 The impact of the pipeline on the branch instruction. The numbers to the left of the instruction (40, 44, ...) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the `beq` instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before `beq` branches to `lw` at location 72. (Figure 4.31 assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

Assume Branch Not Taken

As we saw in Section 4.5, stalling until the branch is complete is too slow. One improvement over branch stalling is to **predict** that the branch will not be taken and thus continue execution down the sequential instruction stream. If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

To discard instructions, we merely change the original control values to 0s, much as we did to stall for a load-use data hazard. The difference is that we must also change the three instructions in the IF, ID, and EX stages when the branch reaches the MEM stage; for load-use stalls, we just change control to 0 in the ID stage and let them percolate through the pipeline. Discarding instructions, then, means we must be able to **flush** instructions in the IF, ID, and EX stages of the pipeline.

Reducing the Delay of Branches

One way to improve branch performance is to reduce the cost of the taken branch. Thus far, we have assumed the next PC for a branch is selected in the MEM stage, but if we move the branch execution earlier in the pipeline, then fewer instructions need be flushed. The MIPS architecture was designed to support fast single-cycle branches that could be pipelined with a small branch penalty. The designers observed that many branches rely only on simple tests (equality or sign, for example) and that such tests do not require a full ALU operation but can be done with at most a few gates. When a more complex branch decision is required, a separate instruction that uses an ALU to perform a comparison is required—a situation that is similar to the use of condition codes for branches (see Chapter 2).

Moving the branch decision up requires two actions to occur earlier: computing the branch target address and evaluating the branch decision. The easy part of this change is to move up the branch address calculation. We already have the PC value and the immediate field in the IF/ID pipeline register, so we just move the branch adder from the EX stage to the ID stage; of course, the branch target address calculation will be performed for all instructions, but only used when needed.

The harder part is the branch decision itself. For branch equal, we would compare the two registers read during the ID stage to see if they are equal. Equality can be tested by first exclusive ORing their respective bits and then ORing all the results. Moving the branch test to the ID stage implies additional forwarding and hazard detection hardware, since a branch dependent on a result still in the pipeline must still work properly with this optimization. For example, to implement branch on equal (and its inverse), we will need to forward results to the equality test logic that operates during ID. There are two complicating factors:

1. During ID, we must decode the instruction, decide whether a bypass to the equality unit is needed, and complete the equality comparison so that if the instruction is a branch, we can set the PC to the branch target address. Forwarding for the operands of branches was formerly handled by the ALU forwarding logic, but the introduction of the equality test unit in ID will require new forwarding logic. Note that the bypassed source operands of a branch can come from either the ALU/MEM or MEM/WB pipeline latches.
2. Because the values in a branch comparison are needed during ID but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed. For example, if an ALU instruction immediately preceding a branch produces one of the operands for the comparison in the branch, a stall will be required, since the EX stage for the ALU instruction will occur after the ID cycle of the branch. By extension, if a load is immediately followed by a conditional branch that is on the load result, two stall cycles will be needed, as the result from the load appears at the end of the MEM cycle but is needed at the beginning of ID for the branch.

Dynamic Branch Prediction

Assuming a branch is not taken is one simple form of *branch prediction*. In that case, we predict that branches are untaken, flushing the pipeline when we are wrong. For the simple five-stage pipeline, such an approach, possibly coupled with compiler-based prediction, is probably adequate. With deeper pipelines, the branch penalty increases when measured in clock cycles. Similarly, with multiple issue (see Section 4.10), the branch penalty increases in terms of instructions lost. This combination means that in an aggressive pipeline, a simple static prediction scheme will probably waste too much performance. As we mentioned in Section 4.5, with more hardware it is possible to try to **predict** branch behavior during program execution.

One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and, if so, to begin fetching new instructions from the same place as the last time. This technique is called **dynamic branch prediction**.

One implementation of that approach is a **branch prediction buffer** or **branch history table**. A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

This is the simplest sort of buffer; we don't know, in fact, if the prediction is the right one—it may have been put there by another branch that has the same low-order address bits. However, this doesn't affect correctness. Prediction is just a hint that we hope is correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken. The following example shows this dilemma.

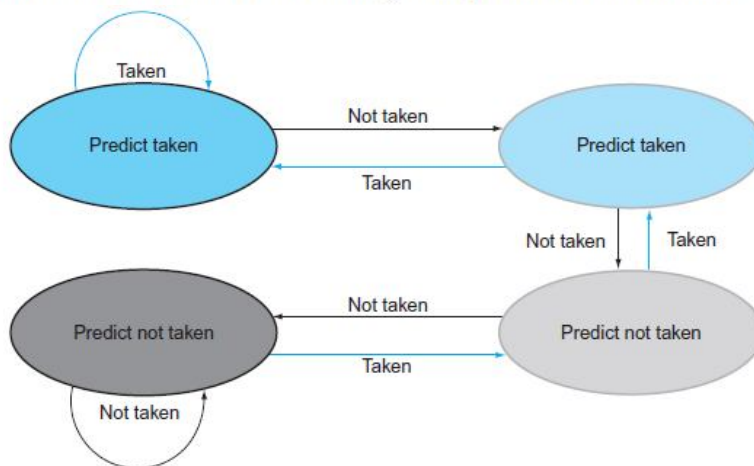


FIGURE 4.63 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The 2-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the mid-point of its range as the division between taken and not taken.

Exceptions

Many architectures and authors do not distinguish between interrupts and exceptions, often using the older name *interrupt* to refer to both types of events. For example, the Intel x86 uses *interrupt*. We follow the MIPS convention, using the term *exception* to refer to *any* unexpected change in control flow without distinguishing whether the cause is internal or external; we use the term *interrupt* only when the event is externally caused. Here are five examples showing whether the situation is internally generated by the processor or externally generated:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Many of the requirements to support exceptions come from the specific situation that causes an exception to occur. Accordingly, we will return to this topic in Chapter 5, when we will better understand the motivation for additional capabilities in the exception mechanism. In this section, we deal with the control implementation for detecting two types of exceptions that arise from the portions of the instruction set and implementation that we have already discussed.

Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a processor, which determines the clock cycle time and thus performance. Without proper attention to exceptions during design of the control unit, attempts to add exceptions to a complicated implementation can significantly reduce performance, as well as complicate the task of getting the design correct.

How Exceptions Are Handled in the MIPS Architecture

The two types of exceptions that our current implementation can generate are execution of an undefined instruction and an arithmetic overflow. We'll use arithmetic overflow in the instruction `add $1, $2, $1` as the example exception in the next few pages. The basic action that the processor must perform when an exception occurs is to save the address of the offending instruction in the *exception program counter* (EPC) and then transfer control to the operating system at some specified address.

A second method, is to use **vectored interrupts**. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception. For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:

Exception type	Exception vector address (in hex)
Undefined instruction	8000 0000 _{hex}
Arithmetic overflow	8000 0180 _{hex}

The operating system knows the reason for the exception by the address at which it is initiated. The addresses are separated by 32 bytes or eight instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence. When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause.

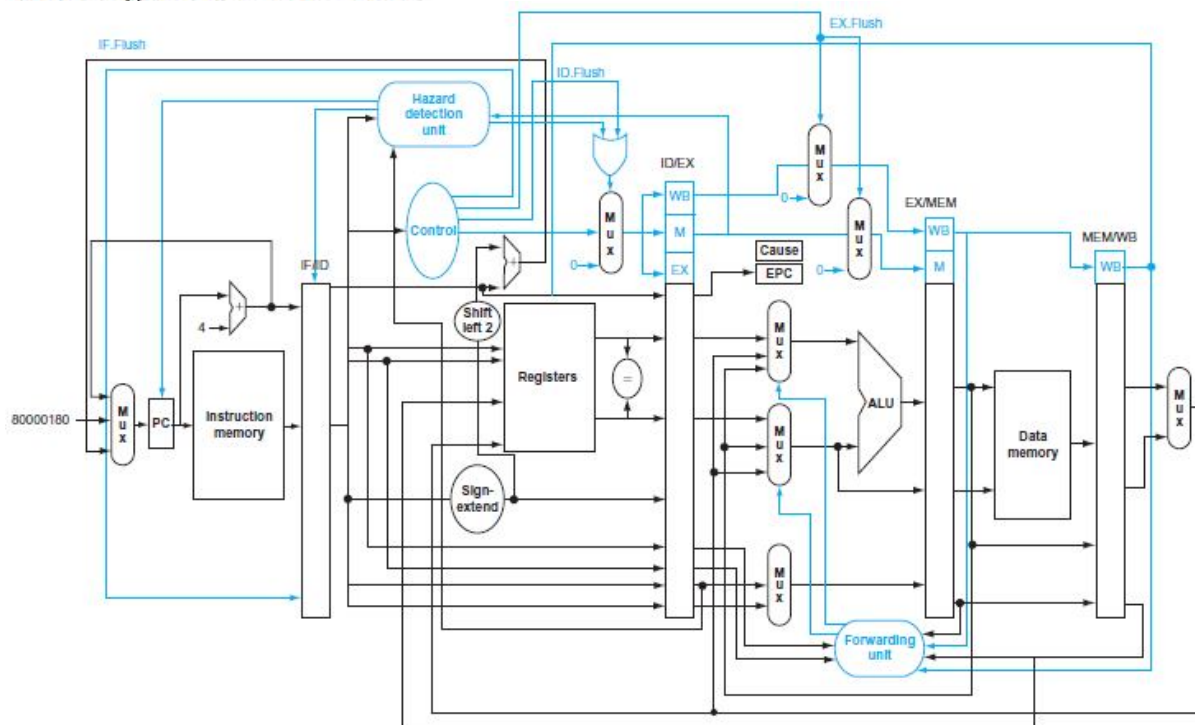


FIGURE 4.66 The datapath with controls to handle exceptions. The key additions include a new input with the value 8000 0180_{hex} in the multiplexor that supplies the new PC value; a Cause register to record the cause of the exception; and an Exception PC register to save the address of the instruction that caused the exception. The 8000 0180_{hex} input to the multiplexor is the initial address to begin fetching instructions in the event of an exception. Although not shown, the ALU overflow signal is an input to the control unit.

UNIT IV PARALLELISM

Instruction Level Parallelism

- **Instruction-Level Parallelism (ILP):** overlap the execution of instructions to improve performance
- **2 approaches to exploit ILP:**
 - 1) Rely on hardware to help discover and exploit the parallelism **dynamically** (e.g., Pentium 4, AMD Opteron, IBM Power)
 - 2) Rely on software technology to find parallelism, **statically** at compile-time (e.g., Itanium 2)
- **Basic Block (BB) ILP is quite small**
 - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
 - average dynamic branch frequency 15% to 25%
=> 4 to 7 instructions execute between a pair of branches
 - Plus instructions in BB likely to depend on each other
- **To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks**
- **Simplest: loop-level parallelism to exploit parallelism among iterations of a loop. E.g.,**

```
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```

- **Exploit loop-level parallelism to parallelism by “unrolling loop” either by**
 1. dynamic via branch prediction or
 2. static via loop unrolling by compiler

Determining instruction dependence is critical to Loop Level Parallelism
- **If 2 instructions are**
 - **parallel**, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
 - **dependent**, they are not parallel and must be executed in order, although they may often be partially overlapped


ILP and Data Dependancies, Hazards

- HW/SW must preserve **program order**:
order instructions would execute in if executed sequentially as determined by original source program
 - Dependences are a property of **programs**
- Presence of dependence indicates **potential** for a hazard, but actual hazard and length of any stall is property of the **pipeline**
- Importance of the data dependencies
 - 1) indicates the possibility of a hazard
 - 2) determines order in which results must be calculated
 - 3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order **only where it affects the outcome of the program**

Name Dependence #1: Anti-dependence

- **Name dependence**: when 2 instructions use same register or memory location, called a **name**, but no flow of data between the instructions associated with that name; **2 versions of name dependence**
- Instr_j writes operand **before** Instr_i reads it

```
    I: sub r4, r1, r3
    J: add r1, r2, r3
    K: mul r6, r1, r7
```



Called an “**anti-dependence**” by compiler writers.
This results from reuse of the name “r1”

- If anti-dependence caused a hazard in the pipeline, called a **Write After Read (WAR) hazard**

Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

```
if p1 {  
    s1;  
};  
if p2 {  
    s2;  
}
```

- `s1` is control dependent on `p1`, and `s2` is control dependent on `p2` but not on `p1`.

Control Dependence Ignored

- Control dependence need not be preserved
 - willing to execute instructions that should not have been executed, thereby violating the control dependences, `if` can do so without affecting correctness of the program
- Instead, 2 properties critical to program correctness are
 - 1) exception behavior and
 - 2) data flow

Exception Behavior

- **Preserving exception behavior**
⇒ any changes in instruction execution order must not change how exceptions are raised in program
(⇒ no new exceptions)
- **Example:**
 DADDU R2 , R3 , R4
 BEQZ R2 , L1
 LW R1 , 0 (R2)
L1 :
 – (Assume branches not delayed)
- **Problem with moving LW before BEQZ?**

FLYNN'S CLASSIFICATION

This classification was first studied and proposed by Michael Flynn in 1972. Flynn did not consider the machine architecture for classification of parallel computers; he introduced the concept of *instruction* and *data* streams for categorizing of computers. All the computers classified by Flynn are not parallel computers, but to grasp the concept of parallel computers, it is necessary to understand all types of Flynn's classification. Since, this classification is based on instruction and data streams, first we need to understand how the instruction cycle works.

2.3.1 Instruction Cycle

The instruction cycle consists of a sequence of steps needed for the execution of an instruction in a program. A typical instruction in a program is composed of two parts: Opcode and Operand. The Operand part specifies the data on which the specified operation is to be done. (See *Figure 1*). The Operand part is divided into two parts: addressing mode and the Operand. The addressing mode specifies the method of determining the addresses of the actual data on which the operation is to be performed and the operand part is used as an argument by the method in determining the actual address.

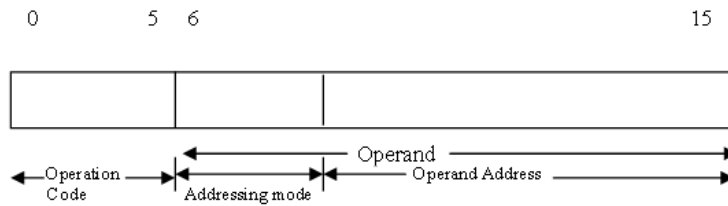


Figure 1: Opcode and Operand

The control unit of the CPU of the computer fetches instructions in the program, one at a time. The fetched instruction is then decoded by the decoder which is a part of the control unit and the processor executes the decoded instructions. The result of execution is temporarily stored in Memory Buffer Register (MBR) (also called Memory Data Register). The normal execution steps are shown in Figure 2.

2.3.2 Instruction Stream and Data Stream

The term 'stream' refers to a sequence or flow of either instructions or data operated on by the computer. In the complete cycle of instruction execution, a flow of instructions from main memory to the CPU is established. This flow of instructions is called **instruction stream**. Similarly, there is a flow of operands between processor and memory bi-directionally. This flow of operands is called **data stream**. These two types of streams are shown in Figure 3.

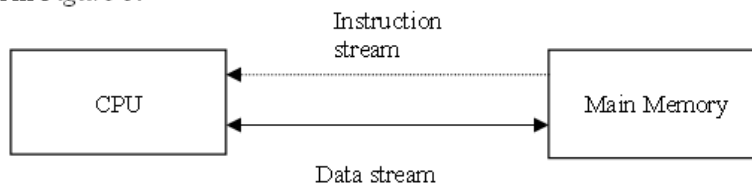


Figure 3: Instruction and data stream

2.3.3 Flynn's Classification

Flynn's classification is based on multiplicity of instruction streams and data streams observed by the CPU during program execution. Let I_s and D_s are minimum number of streams flowing at any point in the execution, then the computer organisation can be categorized as follows:

1) Single Instruction and Single Data stream (SISD)

In this organisation, sequential execution of instructions is performed by one CPU containing a single processing element (PE), i.e., ALU under one control unit as shown in Figure 4. Therefore, SISD machines are conventional serial computers that process only one stream of instructions and one stream of data. This type of computer organisation is depicted in the diagram:

$$I_s = D_s = 1$$

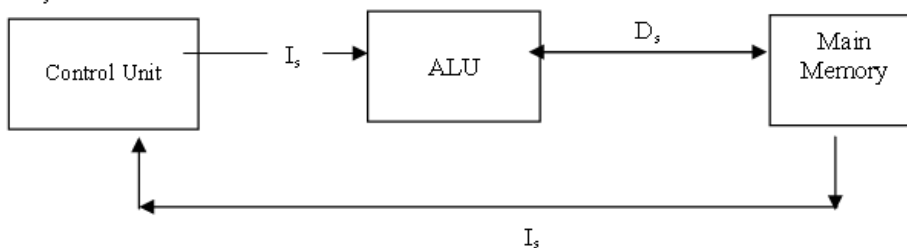


Figure 4: SISD Organisation

Examples of SISD machines include:

- CDC 6600 which is unpipelined but has multiple functional units.
- CDC 7600 which has a pipelined arithmetic unit.
- Amdhal 470/6 which has pipelined instruction processing.
- Cray-1 which supports vector processing.

2) Single Instruction and Multiple Data stream (SIMD)

In this organisation, multiple processing elements work under the control of a single control unit. It has one instruction and multiple data stream. All the processing elements of this organization receive the same instruction broadcast from the CU. Main memory can also be divided into modules for generating multiple data streams acting as a *distributed memory* as shown in *Figure 5*. Therefore, all the processing elements simultaneously execute the same instruction and are said to be 'lock-stepped' together. Each processor takes the data from its own memory and hence it has on distinct data streams. (Some systems also provide a shared global memory for communications.) Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Thus, the execution of instructions is synchronous. Examples of SIMD organisation are ILLIAC-IV, PEPE, BSP, STARAN, MPP, DAP and the Connection Machine (CM-1).

This type of computer organisation is denoted as:

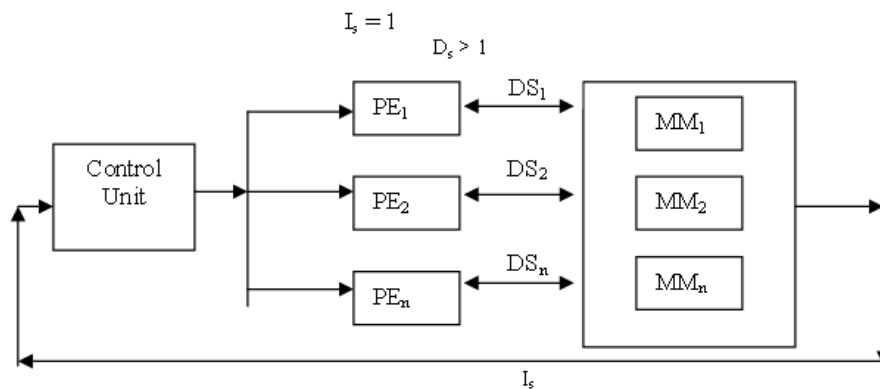


Figure 5: SIMD Organisation

3) Multiple Instruction and Single Data stream (MISD)

In this organization, multiple processing elements are organised under the control of multiple control units. Each control unit is handling one instruction stream and processed through its corresponding processing element. But each processing element is processing only a single data stream at a time. Therefore, for handling multiple instruction streams and single data stream, multiple control units and multiple processing elements are organised in this classification. All processing elements are interacting with the common shared memory for the organisation of single data stream as shown in *Figure 6*. The only known example of a computer capable of MISD operation is the C.mmp built by Carnegie-Mellon University.

This type of computer organisation is denoted as:

$$I_s > 1$$

$$D_s = 1$$

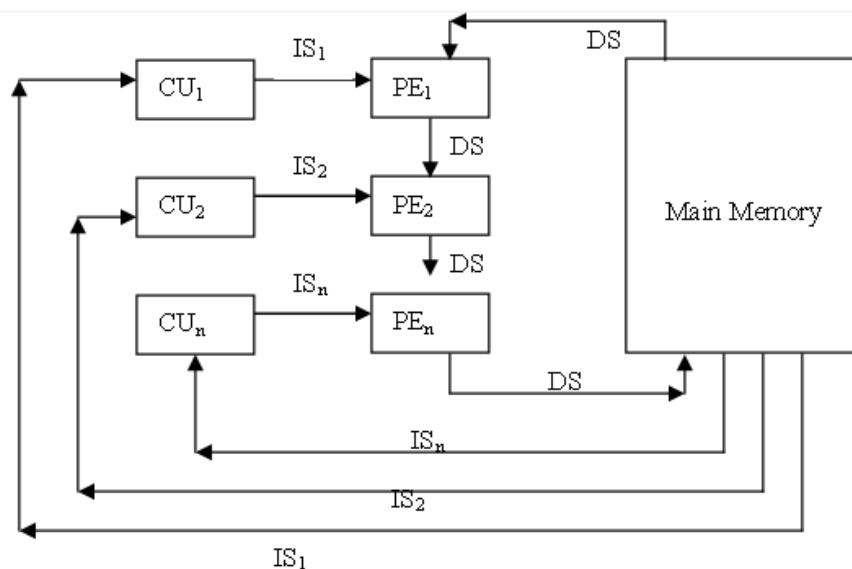


Figure 6: MISD Organisation

This classification is not popular in commercial machines as the concept of single data streams executing on multiple processors is rarely applied. But for the specialized applications, MISD organisation can be very helpful. For example, Real time computers need to be fault tolerant where several processors execute the same data for producing the redundant data. This is also known as N- version programming. All these redundant data

4) Multiple Instruction and Multiple Data stream (MIMD)

In this organization, multiple processing elements and multiple control units are organized as in MISD. But the difference is that now in this organization multiple instruction streams operate on multiple data streams. Therefore, for handling multiple instruction streams, multiple control units and multiple processing elements are organized such that multiple processing elements are handling multiple data streams from the Main memory as shown in *Figure 7*. The processors work on their own data with their own instructions. Tasks executed by different processors can start or finish at different times. They are not lock-stepped, as in SIMD computers, but run asynchronously. This classification actually recognizes the parallel computer. That means in the real sense MIMD organisation is said to be a Parallel computer. All multiprocessor systems fall under this classification.

Examples include; C.mmp, Burroughs D825, Cray-2, S1, Cray X-MP, HEP, Pluribus, IBM 370/168 MP, Univac 1100/80, Tandem/16, IBM 3081/3084, C.m*, BBN Butterfly, Meiko Computing Surface (CS-1), FPS T/40000, iPSC.

This type of computer organisation is denoted as:

$$I_s > 1$$

$$D_s > 1$$

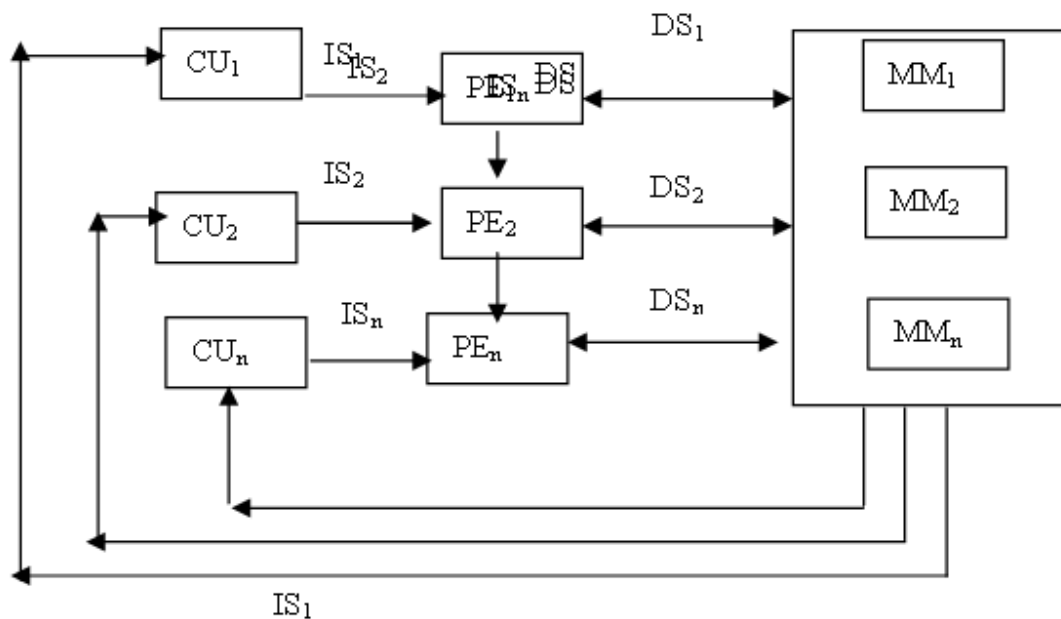


Figure 7: MIMD Organisation

Multithreading

Basics

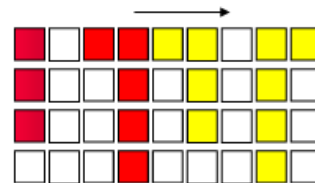
- ❑ **Process**
 - Each process has its unique address space
 - Can consist of several threads
- ❑ **Thread** – each thread has its unique execution context
 - Thread has its own PC (Sequencer) + registers + stack
 - All threads (within a process) share same address space
 - Private heap is optional
- ❑ **Multithreaded app's: process broken into threads**
 - #1 example: transactions (databases, web servers)
 - ✓ Increased concurrency
 - ✓ Partial blocking (your transaction blocks, mine doesn't have to)
 - ✓ Centralized (smarter) resource management (by process)

Multiple Hardware Threads

- ❑ **A thread can be viewed as a stream of instructions**
 - State is represented by PC, Stack pointer, GP Registers
- ❑ **Equip multithreaded processors with multiple hardware contexts (threads)**
 - OS views CPU as multiple logical processors
- ❑ **Execution of instructions from different threads are interleaved in the pipeline**
 - Interleaving policy is critical...

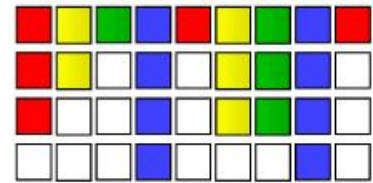
Blocked Multithreading

(SoE-MT- Switch on Event MT, aka' – “Poor Man MT”)



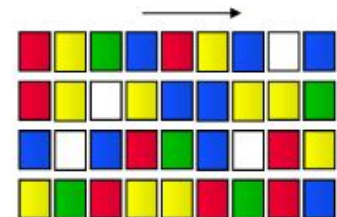
- ❑ **Critical decision: when to switch threads**
 - When current thread's utilization/throughput is about to drop (e.g. L2 cache miss)
- ❑ **Requirements for throughput:**
 - (Thread switch) + (pipe fill time) << blocking latency
 - ✓ Would like to get some work done before other thread comes back
 - **Fast thread-switch: multiple register banks**
 - **Fast pipe-fill: short pipe**
- ❑ **Advantage: small changes to existing hardware**
- ❑ **Drawback: high single thread performance requires long thread switch**
- ❑ **Examples**
 - **Macro-dataflow machine**
 - **MIT Alewife**
 - **IBM Northstar**

Interleaved (Fine grained) Multithreading (cont.)



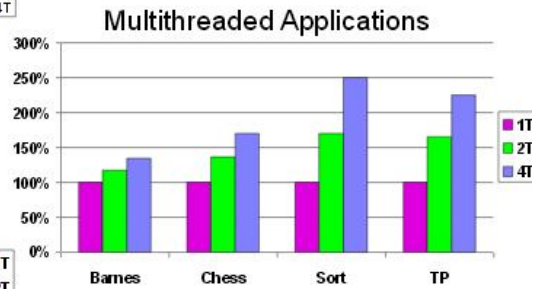
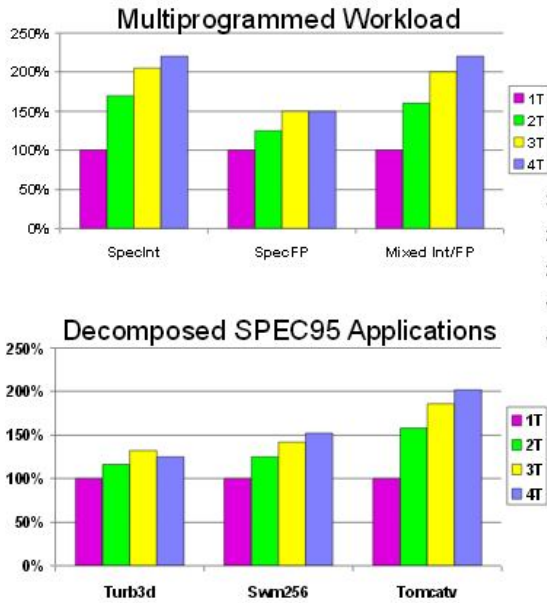
- Advantages:
 - (w/ flexible interleave:) Reasonable single thread performance
 - High processor utilization (esp. in case of many thread)
- Drawback:
 - Complicated hardware
 - Multiple contexts (states)
 - (w/ inflexible interleave:) limits single thread performance
- Examples:
 - HEP Denelcor: 8 threads (latencies were shorter then)
 - TERA: 128 threads
 - MicroUnity - 5 x 1GZ threads = 200 MHz like latency
- Became attractive for GPUs and network processors

Simultaneous Multi-threading (SMT)

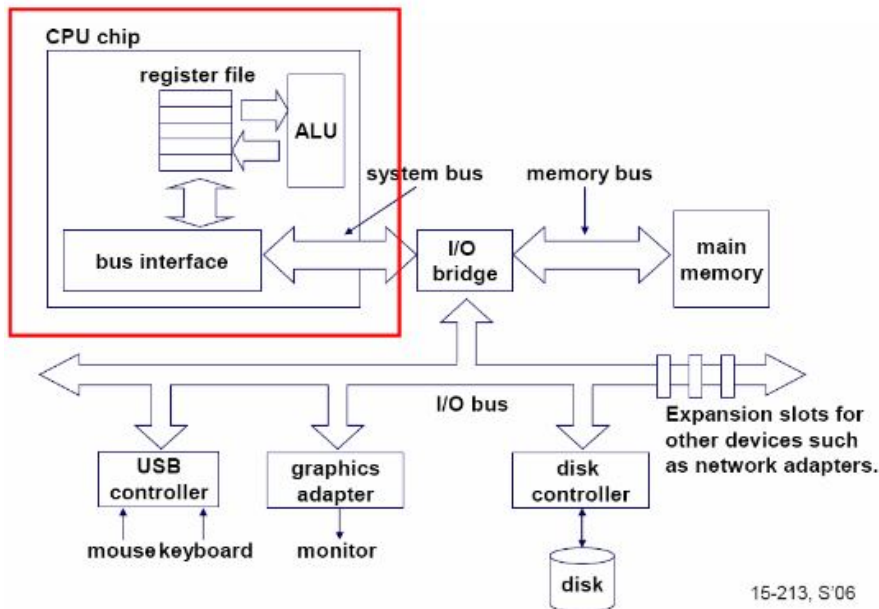


- Critical decision: fetch-interleaving policy
- Requirements for throughput:
 - Enough threads to utilize resources
 - ✓ Fewer than needed to stretch dependences
- Examples:
 - Compaq Alpha EV8 (cancelled)
 - Intel Pentium® 4 Hyper-Threading Technology

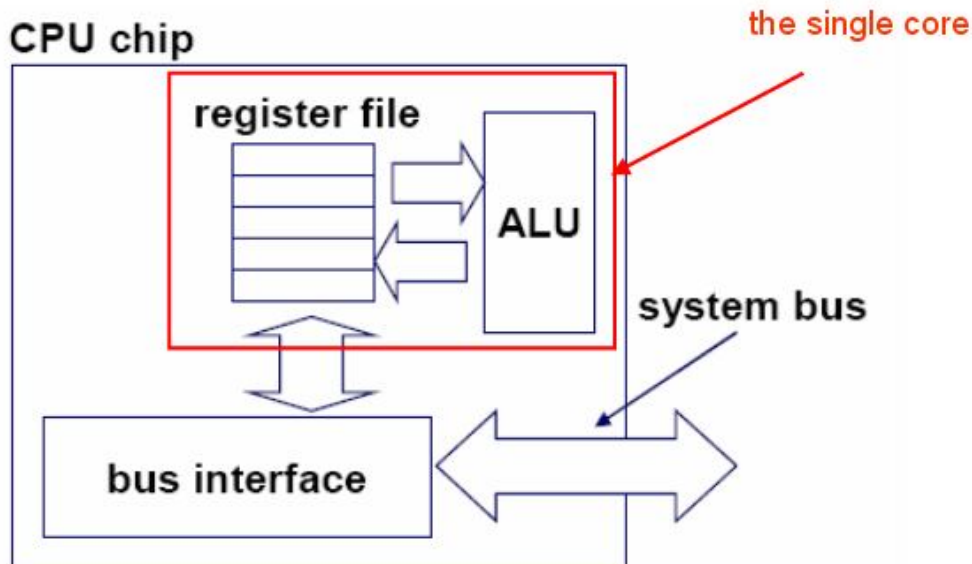
Performance Scalability



Single-core computer

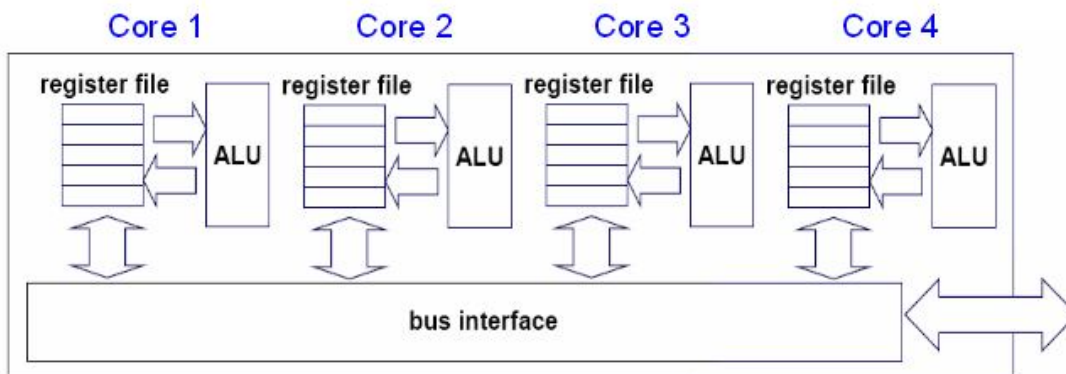


Single-core CPU chip



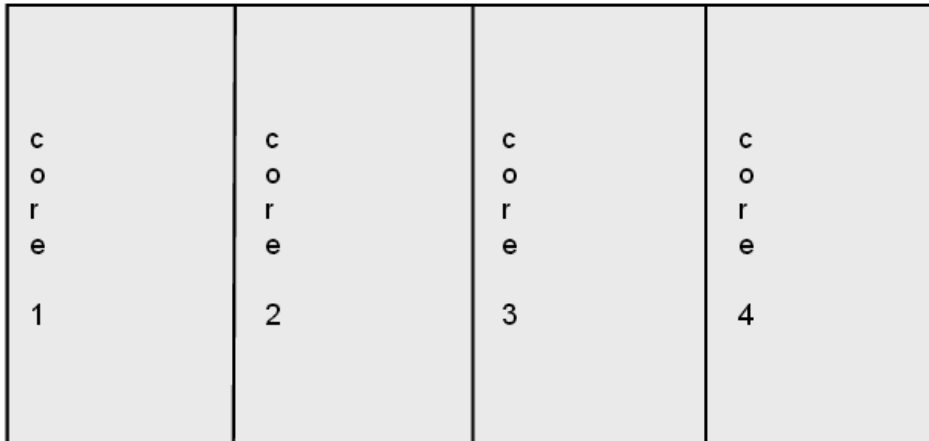
Multi-core architectures

- This lecture is about a new trend in computer architecture:
Replicate multiple processor cores on a single die.



Multi-core CPU chip

- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



UNIT V MEMORY AND I/O SYSTEMS

Memory hierarchy

This *principle of locality* underlies both the way in which you did your work in the library and the way that programs operate. The principle of locality states that programs access a relatively small portion of their address space at any instant of time, just as you accessed a very small portion of the library's collection. There are two different types of locality:

- **Temporal locality** (locality in time): if an item is referenced, it will tend to be referenced again soon. If you recently brought a book to your desk to look at, you will probably need to look at it again soon.
- **Spatial locality** (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon. For example, when you brought out the book on early English computers to find out about the EDSAC, you also noticed that there was another book shelved next to it about early mechanical computers, so you also brought back that book and, later on, found something useful in that book. Libraries put books on the same topic together on the same shelves to increase spatial locality. We'll see how memory hierarchies use spatial locality a little later in this chapter.

Speed	Processor	Size	Cost (\$/bit)	Current technology
Fastest	Memory	Smallest	Highest	SRAM
	Memory			DRAM
Slowest	Memory	Biggest	Lowest	Magnetic disk

FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many personal mobile devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 5.2.

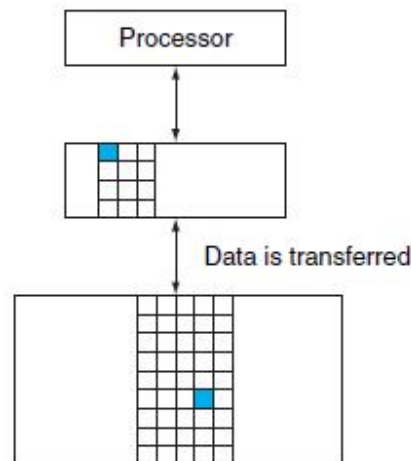


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a *block* or a *line*. Usually we transfer an entire block when we copy something between levels.

The upper level—the one closer to the processor—is smaller and faster than the lower level, since the upper level uses technology that is more expensive. Figure 5.2 shows that the minimum unit of information that can be either present or not present in the two-level hierarchy is called a **block** or a **line**; in our library analogy, a block of information is one book.

If the data requested by the processor appears in some block in the upper level, this is called a *hit* (analogous to your finding the information in one of the books on your desk). If the data is not found in the upper level, the request is called a *miss*. The lower level in the hierarchy is then accessed to retrieve the block containing the requested data. (Continuing our analogy, you go from your desk to the shelves to find the desired book.) The **hit rate**, or *hit ratio*, is the fraction of memory accesses found in the upper level; it is often used as a measure of the performance of the memory hierarchy. The **miss rate** (1–hit rate) is the fraction of memory accesses not found in the upper level.

Since performance is the major reason for having a memory hierarchy, the time to service hits and misses is important. **Hit time** is the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss (that is, the time needed to look through the books on the desk). The **miss penalty** is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor (or the time to get another book from the shelves and place it on the desk). Because the upper level is smaller and built using faster memory parts, the hit time will be much smaller than the time to access the next level in the hierarchy, which is the major component of the miss penalty. (The time to examine the books on the desk is much smaller than the time to get up and get a new book from the shelves.)

Memory technologies

There are four primary technologies used today in memory hierarchies. Main memory is implemented from DRAM (dynamic random access memory), while levels closer to the processor (caches) use SRAM (static random access memory). DRAM is less costly per bit than SRAM, although it is substantially slower. The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity for the same amount of silicon; the speed difference arises from several factors described in [Section B.9](#) of [Appendix B](#). The third technology is flash memory. This nonvolatile memory is the secondary memory in Personal Mobile Devices. The fourth technology, used to implement the largest and slowest level in the hierarchy in servers, is magnetic disk. The access time and price per bit vary widely among these technologies, as the table below shows, using typical values for 2012:

Memory technology	Typical access time	\$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	\$500–\$1000
DRAM semiconductor memory	50–70 ns	\$10–\$20
Flash semiconductor memory	5,000–50,000 ns	\$0.75–\$1.00
Magnetic disk	5,000,000–20,000,000 ns	\$0.05–\$0.10

We describe each memory technology in the remainder of this section.

SRAM Technology

SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write. SRAMs have a fixed access time to any datum, though the read and write access times may differ.

SRAMs don't need to refresh and so the access time is very close to the cycle time. SRAMs typically use six to eight transistors per bit to prevent the information from being disturbed when read. SRAM needs only minimal power to retain the charge in standby mode.

In the past, most PCs and server systems used separate SRAM chips for either their primary, secondary, or even tertiary caches. Today, thanks to Moore's Law, all levels of caches are integrated onto the processor chip, so the market for separate SRAM chips has nearly evaporated.

DRAM Technology

In a SRAM, as long as power is applied, the value can be kept indefinitely. In a dynamic RAM (DRAM), the value kept in a cell is stored as a charge in a capacitor. A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only a single transistor per bit of storage, they are much denser and cheaper per bit than SRAM. As DRAMs store the charge on a capacitor, it cannot be kept indefinitely and must periodically be refreshed. That is why this memory structure is called dynamic, as opposed to the static storage in an SRAM cell.

To refresh the cell, we merely read its contents and write it back. The charge can be kept for several milliseconds. If every bit had to be read out of the DRAM and then written back individually, we would constantly be refreshing the DRAM, leaving no time for accessing it. Fortunately, DRAMs use a two-level decoding structure, and this allows us to refresh an entire *row* (which shares a word line) with a read cycle followed immediately by a write cycle.

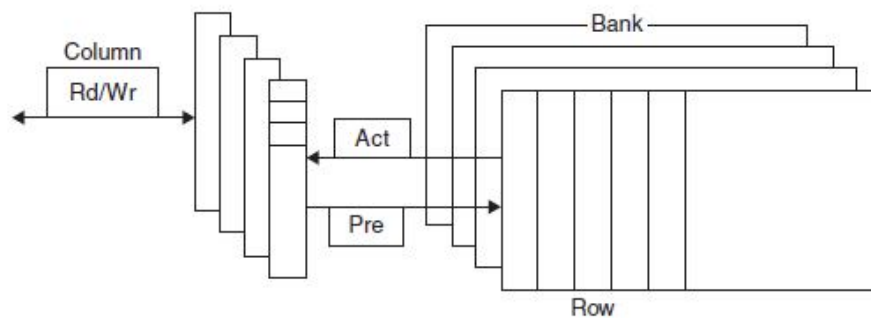


FIGURE 5.4 Internal organization of a DRAM. Modern DRAMs are organized in banks, typically four for DDR3. Each bank consists of a series of rows. Sending a PRE (precharge) command opens or closes a bank. A row address is sent with an Act (activate), which causes the row to transfer to a buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR3) or by specifying a block transfer and the starting address. Each command, as well as block transfers, is synchronized with a clock.

Year Introduced	Chip size	\$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	\$1,500,000	250 ns	150 ns
1983	256 Kibibit	\$500,000	185 ns	100 ns
1985	1 Mebibit	\$200,000	135 ns	40 ns
1989	4 Mebibit	\$50,000	110 ns	40 ns
1992	16 Mebibit	\$15,000	90 ns	30 ns
1996	64 Mebibit	\$10,000	60 ns	12 ns
1998	128 Mebibit	\$4,000	60 ns	10 ns
2000	256 Mebibit	\$1,000	55 ns	7 ns
2004	512 Mebibit	\$250	50 ns	5 ns
2007	1 Gibibit	\$50	45 ns	1.25 ns
2010	2 Gibibit	\$30	40 ns	1 ns
2012	4 Gibibit	\$1	35 ns	0.8 ns

FIGURE 5.5 DRAM size increased by multiples of four approximately once every three years until 1996, and thereafter considerably slower. The improvements in access time have been slower but continuous, and cost roughly tracks density improvements, although cost is often affected by other issues, such as availability and demand. The cost per gibibyte is not adjusted for inflation.

Flash Memory

Flash memory is a type of *electrically erasable programmable read-only memory* (EEPROM).

Unlike disks and DRAM, but like other EEPROM technologies, writes can wear out flash memory bits. To cope with such limits, most flash products include a controller to spread the writes by remapping blocks that have been written many times to less trodden blocks. This technique is called *wear leveling*. With wear leveling, personal mobile devices are very unlikely to exceed the write limits in the flash. Such wear leveling lowers the potential performance of flash, but it is needed unless higher-level software monitors block wear. Flash controllers that perform wear leveling can also improve yield by mapping out memory cells that were manufactured incorrectly.

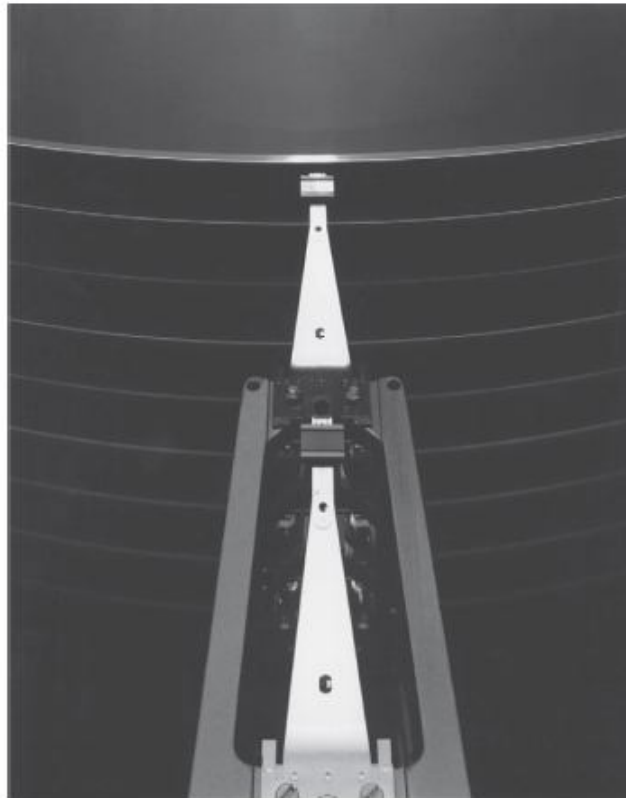


FIGURE 5.6 A disk showing 10 disk platters and the read/write heads. The diameter of today's disks is 2.5 or 3.5 inches, and there are typically one or two platters per drive today.

Once the head has reached the correct track, we must wait for the desired sector to rotate under the read/write head. This time is called the **rotational latency** or **rotational delay**. The average latency to the desired information is halfway around the disk. Disks rotate at 5400 RPM to 15,000 RPM. The average rotational latency at 5400 RPM is

$$\begin{aligned} \text{Average rotational latency} &= \frac{0.5 \text{ rotation}}{5400 \text{ RPM}} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM} / \left(60 \frac{\text{seconds}}{\text{minute}}\right)} \\ &= 0.0056 \text{ seconds} = 5.6 \text{ ms} \end{aligned}$$

The last component of a disk access, *transfer time*, is the time to transfer a block of bits. The transfer time is a function of the sector size, the rotation speed, and the recording density of a track. Transfer rates in 2012 were between 100 and 200 MB/sec.

One complication is that most disk controllers have a built-in cache that stores sectors as they are passed over; transfer rates from the cache are typically higher, and were up to 750 MB/sec (6 Gbit/sec) in 2012.

The Basics of Caches

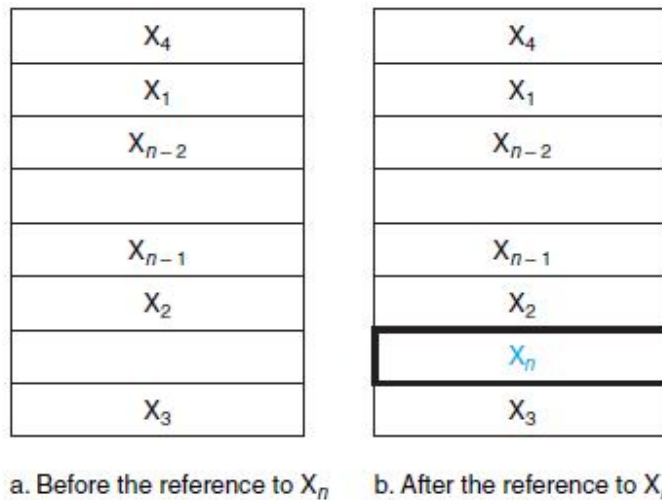


FIGURE 5.7 The cache just before and just after a reference to a word X_n that is not initially in the cache. This reference causes a miss that forces the cache to fetch X_n from memory and insert it into the cache.

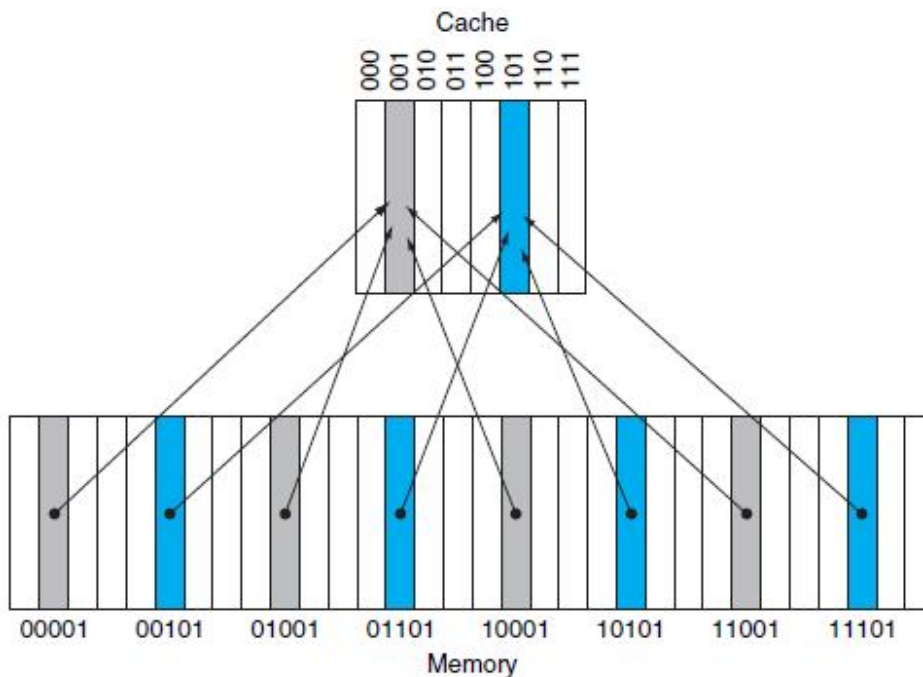


FIGURE 5.8 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the direct-mapped cache word X modulo 8. That is, the low-order $\log_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001_{two} , 01001_{two} , 10001_{two} , and 11001_{two} all map to entry 001_{two} of the cache, while addresses 00101_{two} , 01101_{two} , 10101_{two} , and 11101_{two} all map to entry 101_{two} of the cache.

In looking at the scenario in Figure 5.7, there are two questions to answer: How do we know if a data item is in the cache? Moreover, if it is, how do we find it? The answers are related. If each word can go in exactly one place in the cache, then it is straightforward to find the word if it is in the cache. The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the *address* of the word in memory. This cache structure is called **direct mapped**, since each memory location is mapped directly to exactly one location in the cache. The typical mapping between addresses and cache locations for a direct-mapped cache is usually simple. For example, almost all direct-mapped caches use this mapping to find a block:

$$(\text{Block address}) \bmod (\text{Number of blocks in the cache})$$

If the number of entries in the cache is a power of 2, then modulo can be computed simply by using the low-order \log_2 (cache size in blocks) bits of the address. Thus, an 8-block cache uses the three lowest bits ($8 = 2^3$) of the block address. For example, Figure 5.8 shows how the memory addresses between 1_{ten} (00001_{two}) and 29_{ten} (11101_{two}) map to locations 1_{ten} (001_{two}) and 5_{ten} (101_{two}) in a direct-mapped cache of eight words.

Because each cache location can contain the contents of a number of different memory locations, how do we know whether the data in the cache corresponds to a requested word? That is, how do we know whether a requested word is in the cache or not? We answer this question by adding a set of **tags** to the cache. The tags contain the address information required to identify whether a word in the cache corresponds to the requested word. The tag needs only to contain the upper portion of the address, corresponding to the bits that are not used as an index into the cache. For example, in Figure 5.8 we need only have the upper 2 of the 5 address bits in the tag, since the lower 3-bit index field of the address selects the block. Architects omit the index bits because they are redundant, since by definition the index field of any address of a cache block must be that block number.

Accessing a Cache

Below is a sequence of nine memory references to an empty eight-block cache, including the action for each reference. Figure 5.9 shows how the contents of the cache change on each miss. Since there are eight blocks in the cache, the low-order three bits of an address give the block number:

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.6b)	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss (5.6c)	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss (5.6d)	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss (5.6e)	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss (5.6f)	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	Y	11 _{two}	Memory (11010 _{two})
011	N		
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10 _{two}	Memory (10000 _{two})
001	N		
010	Y	11 _{two}	Memory (11010 _{two})
011	Y	00 _{two}	Memory (00011 _{two})
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	Y	10 _{two}	Memory (10000 _{two})
001	N		
010	Y	11 _{two}	Memory (11010 _{two})
011	N		
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10 _{two}	Memory (10000 _{two})
001	N		
010	Y	10 _{two}	Memory (10010 _{two})
011	Y	00 _{two}	Memory (00011 _{two})
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

f. After handling a miss of address (10010_{two})

FIGURE 5.9 The cache contents are shown after each reference request that misses, with the index and tag fields shown in binary for the sequence of addresses on page 386. The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), 10010_{two} (miss), and 10000_{two} (hit). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss. The tag field will contain only the upper portion of the address. The full address of a word contained in cache block i with tag field j for this cache is $j \times 8 + i$, or equivalently the concatenation of the tag field j and the index i . For example, in cache f above, index 010_{two} has tag 10_{two} and corresponds to address 10010_{two}.

Measuring and Improving Cache Performance

CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU spends waiting for the memory system. Normally, we assume that the costs of cache accesses that are hits are part of the normal CPU execution cycles. Thus,

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory-stall clock cycles}) \times \text{Clock cycle time}$$

The memory-stall clock cycles come primarily from cache misses, and we make that assumption here. We also restrict the discussion to a simplified model of the memory system. In real processors, the stalls generated by reads and writes can be quite complex, and accurate performance prediction usually requires very detailed simulations of the processor and memory system.

Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from writes:

$$\text{Memory-stall clock cycles} = (\text{Read-stall cycles} + \text{Write-stall cycles})$$

The read-stall cycles can be defined in terms of the number of read accesses per program, the miss penalty in clock cycles for a read, and the read miss rate:

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

Writes are more complicated. For a write-through scheme, we have two sources of stalls: write misses, which usually require that we fetch the block before continuing the write (see the *Elaboration* on page 394 for more details on dealing with writes), and write buffer stalls, which occur when the write buffer is full when a write occurs. Thus, the cycles stalled for writes equals the sum of these two:

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty} \right) + \text{Write buffer stalls}$$

Because the write buffer stalls depend on the proximity of writes, and not just the frequency, it is not possible to give a simple equation to compute such stalls. Fortunately, in systems with a reasonable write buffer depth (e.g., four or more words) and a memory capable of accepting writes at a rate that significantly exceeds the average write frequency in programs (e.g., by a factor of 2), the write buffer stalls will be small, and we can safely ignore them. If a system did not meet these criteria, it would not be well designed; instead, the designer should have used either a deeper write buffer or a write-back organization.

Write-back schemes also have potential additional stalls arising from the need to write a cache block back to memory when the block is replaced. We will discuss this more in Section 5.8.

In most write-through cache organizations, the read and write miss penalties are the same (the time to fetch the block from memory). If we assume that the write buffer stalls are negligible, we can combine the reads and writes by using a single miss rate and the miss penalty:

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

We can also factor this as

$$\text{Memory-stall clock cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Let's consider a simple example to help us understand the impact of cache performance on processor performance.

Calculating Cache Performance

Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.

The number of memory miss cycles for instructions in terms of the Instruction count (I) is

$$\text{Instruction miss cycles} = I \times 2\% \times 100 = 2.00 \times I$$

As the frequency of all loads and stores is 36%, we can find the number of memory miss cycles for data references:

$$\text{Data miss cycles} = I \times 36\% \times 4\% \times 100 = 1.44 \times I$$

The total number of memory-stall cycles is $2.00 I + 1.44 I = 3.44 I$. This is more than three cycles of memory stall per instruction. Accordingly, the total CPI including memory stalls is $2 + 3.44 = 5.44$. Since there is no change in instruction count or clock rate, the ratio of the CPU execution times is

$$\begin{aligned} \frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} &= \frac{I \times \text{CPI}_{\text{stall}} \times \text{Clock cycle}}{I \times \text{CPI}_{\text{perfect}} \times \text{Clock cycle}} \\ &= \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2} \end{aligned}$$

The performance with the perfect cache is better by $\frac{5.44}{2} = 2.72$.

What happens if the processor is made faster, but the memory system is not? The amount of time spent on memory stalls will take up an increasing fraction of the execution time; Amdahl's Law, which we examined in Chapter 1, reminds us of this fact. A few simple examples show how serious this problem can be. Suppose we speed-up the computer in the previous example by reducing its CPI from 2 to 1 without changing the clock rate, which might be done with an improved pipeline. The system with cache misses would then have a CPI of $1 + 3.44 = 4.44$, and the system with the perfect cache would be

$$\frac{4.44}{1} = 4.44 \text{ times as fast,}$$

The amount of execution time spent on memory stalls would have risen from

$$\frac{3.44}{5.44} = 63\%$$

to

$$\frac{3.44}{4.44} = 77\%$$

Similarly, increasing the clock rate without changing the memory system also increases the performance lost due to cache misses.

The previous examples and equations assume that the hit time is not a factor in determining cache performance. Clearly, if the hit time increases, the total time to access a word from the memory system will increase, possibly causing an increase in the processor cycle time. Although we will see additional examples of what can increase

Reducing Cache Misses by More Flexible Placement of Blocks

So far, when we place a block in the cache, we have used a simple placement scheme: A block can go in exactly one place in the cache. As mentioned earlier, it is called *direct mapped* because there is a direct mapping from any block address in memory to a single location in the upper level of the hierarchy. However, there is actually a whole range of schemes for placing blocks. Direct mapped, where a block can be placed in exactly one location, is at one extreme.

At the other extreme is a scheme where a block can be placed in *any* location in the cache. Such a scheme is called **fully associative**, because a block in memory may be associated with any entry in the cache. To find a given block in a fully associative cache, all the entries in the cache must be searched because a block can be placed in any one. To make the search practical, it is done in parallel with a comparator associated with each cache entry. These comparators significantly increase the hardware cost, effectively making fully associative placement practical only for caches with small numbers of blocks.

The middle range of designs between direct mapped and fully associative is called **set associative**. In a set-associative cache, there are a fixed number of locations where each block can be placed. A set-associative cache with n locations for a block is called an n -way set-associative cache. An n -way set-associative cache consists of a number of sets, each of which consists of n blocks. Each block in the memory maps to a unique *set* in the cache given by the index field, and a block can be placed in *any* element of that set. Thus, a set-associative placement combines direct-mapped placement and fully associative placement: a block is directly mapped into a set, and then all the blocks in the set are searched for a match. For example, Figure 5.14 shows where block 12 may be placed in a cache with eight blocks total, according to the three block placement policies.

Remember that in a direct-mapped cache, the position of a memory block is given by

$$(\text{Block number}) \bmod (\text{Number of blocks in the cache})$$



FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by $(12 \bmod 8) = 4$. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set $(12 \bmod 4) = 0$; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

Misses and Associativity in Caches

Assume there are three small caches, each consisting of four one-word blocks. One cache is fully associative, a second is two-way set-associative, and the third is direct-mapped. Find the number of misses for each cache organization given the following sequence of block addresses: 0, 8, 0, 6, and 8.

The direct-mapped case is easiest. First, let's determine to which cache block each block address maps:

Block address	Cache block
0	$(0 \text{ modulo } 4) = 0$
6	$(6 \text{ modulo } 4) = 2$
8	$(8 \text{ modulo } 4) = 0$

Now we can fill in the cache contents after each reference, using a blank entry to mean that the block is invalid, colored text to show a new entry added to the cache for the associated reference, and plain text to show an old entry in the cache:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

The direct-mapped cache generates five misses for the five accesses.

The set-associative cache has two sets (with indices 0 and 1) with two elements per set. Let's first determine to which set each block address maps:

Block address	Cache set
0	$(0 \text{ modulo } 2) = 0$
6	$(6 \text{ modulo } 2) = 0$
8	$(8 \text{ modulo } 2) = 0$

Because we have a choice of which entry in a set to replace on a miss, we need a replacement rule. Set-associative caches usually replace the least recently used block within a set; that is, the block that was used furthest in the past

is replaced. (We will discuss other replacement rules in more detail shortly.) Using this replacement rule, the contents of the set-associative cache after each reference looks like this:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Notice that when block 6 is referenced, it replaces block 8, since block 8 has been less recently referenced than block 0. The two-way set-associative cache has four misses, one less than the direct-mapped cache.

The fully associative cache has four cache blocks (in a single set); any memory block can be stored in any cache block. The fully associative cache has the best performance, with only three misses:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

For this series of references, three misses is the best we can do, because three unique block addresses are accessed. Notice that if we had eight blocks in the cache, there would be no replacements in the two-way set-associative cache (check this for yourself), and it would have the same number of misses as the fully associative cache. Similarly, if we had 16 blocks, all 3 caches would have the same number of misses. Even this trivial example shows that cache size and associativity are not independent in determining cache performance.

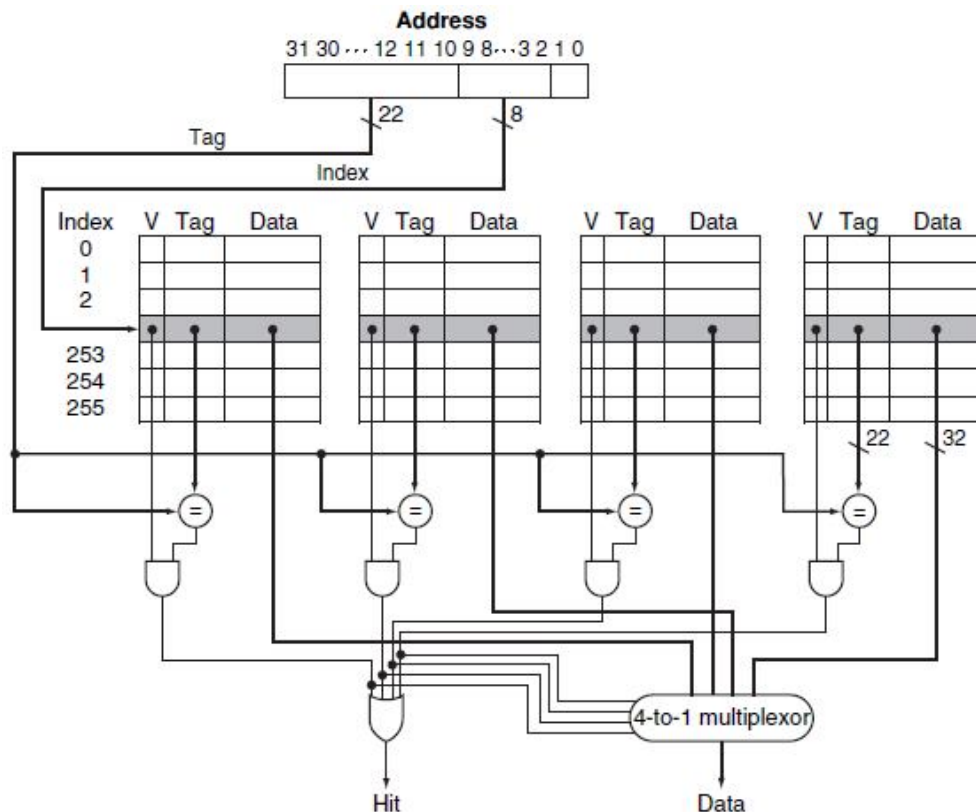


FIGURE 5.18 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor. The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

Choosing Which Block to Replace

When a miss occurs in a direct-mapped cache, the requested block can go in exactly one position, and the block occupying that position must be replaced. In an associative cache, we have a choice of where to place the requested block, and hence a choice of which block to replace. In a fully associative cache, all blocks are candidates for replacement. In a set-associative cache, we must choose among the blocks in the selected set.

The most commonly used scheme is **least recently used (LRU)**, which we used in the previous example. In an LRU scheme, the block replaced is the one that has been unused for the longest time. The set associative example on page 405 uses LRU, which is why we replaced Memory(0) instead of Memory(6).

LRU replacement is implemented by keeping track of when each element in a set was used relative to the other elements in the set. For a two-way set-associative cache, tracking when the two elements were used can be implemented by keeping a single bit in each set and setting the bit to indicate an element whenever that element is referenced. As associativity increases, implementing LRU gets harder; in Section 5.8, we will see an alternative scheme for replacement.

Virtual Memory

Of course, to allow multiple virtual machines to share the same memory, we must be able to protect the virtual machines from each other, ensuring that a program can only read and write the portions of main memory that have been assigned to it. Main memory need contain only the active portions of the many virtual machines, just as a cache contains only the active portion of one program. Thus, the principle of locality enables virtual memory as well as caches, and virtual memory allows us to efficiently share the processor as well as the main memory.

We cannot know which virtual machines will share the memory with other virtual machines when we compile them. In fact, the virtual machines sharing the memory change dynamically while the virtual machines are running. Because of this dynamic interaction, we would like to compile each program into its own *address space*—a separate range of memory locations accessible only to this program. Virtual memory implements the translation of a program's address space to **physical addresses**. This translation process enforces **protection** of a program's address space from other virtual machines.

The second motivation for virtual memory is to allow a single user program to exceed the size of primary memory. Formerly, if a program became too large for memory, it was up to the programmer to make it fit. Programmers divided programs into pieces and then identified the pieces that were mutually exclusive. These *overlays* were loaded or unloaded under user program control during execution, with the programmer ensuring that the program never tried to access an overlay that was not loaded and that the overlays loaded never exceeded the total size of the memory. Overlays were traditionally organized as modules, each containing both code and data. Calls between procedures in different modules would lead to overlaying of one module with another.

As you can well imagine, this responsibility was a substantial burden on programmers. Virtual memory, which was invented to relieve programmers of this difficulty, automatically manages the two levels of the memory hierarchy represented by main memory (sometimes called *physical memory* to distinguish it from virtual memory) and secondary storage.

In virtual memory, the address is broken into a *virtual page number* and a *page offset*. [Figure 5.26](#) shows the translation of the virtual page number to a *physical page number*. The physical page number constitutes the upper portion of the physical address, while the page offset, which is not changed, constitutes the lower portion. The number of bits in the page offset field determines the page size. The number of pages addressable with the virtual address need not match the number of pages addressable with the physical address. Having a larger number of virtual pages than physical pages is the basis for the illusion of an essentially unbounded amount of virtual memory.

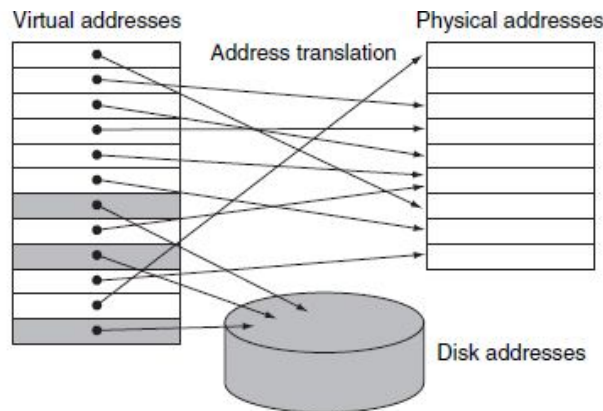


FIGURE 5.25 In virtual memory, blocks of memory (called *pages*) are mapped from one set of addresses (called *virtual addresses*) to another set (called *physical addresses*). The processor generates virtual addresses while the memory is accessed using physical addresses. Both the virtual memory and the physical memory are broken into pages, so that a virtual page is mapped to a physical page. Of course, it is also possible for a virtual page to be absent from main memory and not be mapped to a physical address; in that case, the page resides on disk. Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code.

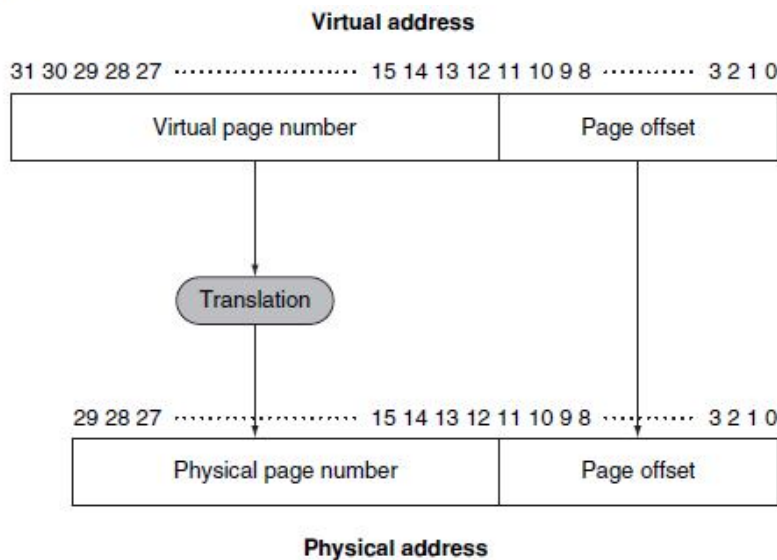


FIGURE 5.26 Mapping from a virtual to a physical address. The page size is $2^{12} = 4$ KiB. The number of physical pages allowed in memory is 2^{18} , since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GiB, while the virtual address space is 4 GiB.

Many design choices in virtual memory systems are motivated by the high cost of a page fault. A page fault to disk will take millions of clock cycles to process. (The table on page 378 shows that main memory latency is about 100,000 times quicker than disk.) This enormous miss penalty, dominated by the time to get the first word for typical page sizes, leads to several key decisions in designing virtual memory systems:

- Pages should be large enough to try to amortize the high access time. Sizes from 4 KiB to 16 KiB are typical today. New desktop and server systems are being developed to support 32 KiB and 64 KiB pages, but new embedded systems are going in the other direction, to 1 KiB pages.
- Organizations that reduce the page fault rate are attractive. The primary technique used here is to allow fully associative placement of pages in memory.
- Page faults can be handled in software because the overhead will be small compared to the disk access time. In addition, software can afford to use clever algorithms for choosing how to place pages because even small reductions in the miss rate will pay for the cost of such algorithms.
- Write-through will not work for virtual memory, since writes take too long. Instead, virtual memory systems use write-back.

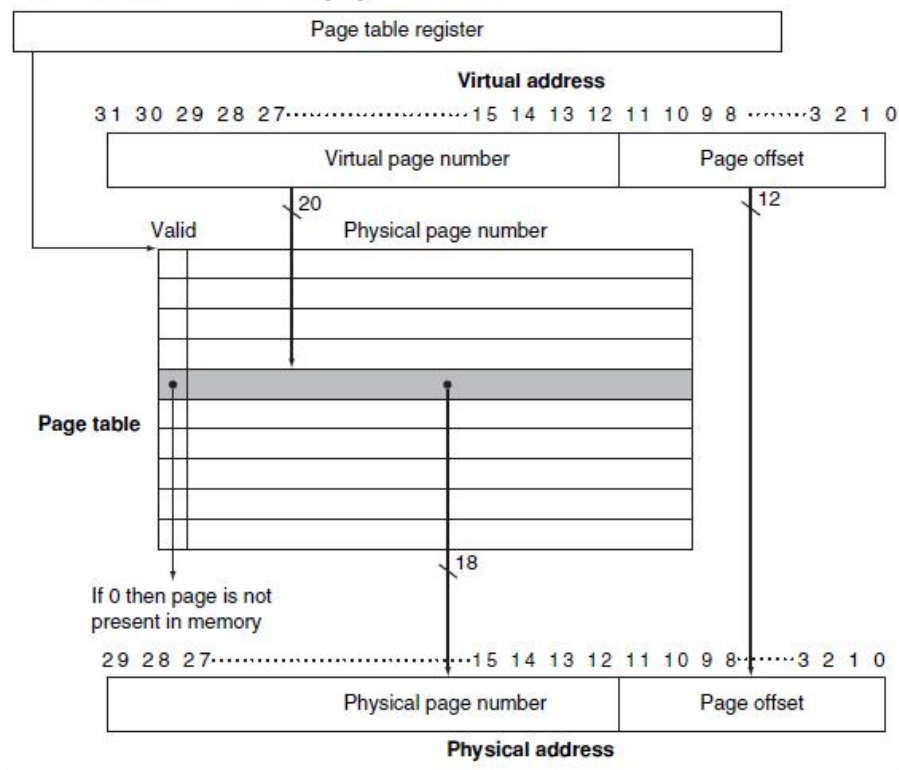


FIGURE 5.27 The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address. We assume a 32-bit address. The page table pointer gives the starting address of the page table. In this figure, the page size is 2^{12} bytes, or 4 KiB. The virtual address space is 2^{32} bytes, or 4 GiB, and the physical address space is 2^{30} bytes, which allows main memory of up to 1 GiB. The number of entries in the page table is 2^{20} , or 1 million entries. The valid bit for each entry indicates whether the mapping is legal. If it is off, then the page is not present in memory. Although the page table entry shown here need only be 19 bits wide, it would typically be rounded up to 32 bits for ease of indexing. The extra bits would be used to store additional information that needs to be kept on a per-page basis, such as protection.

Page Faults

If the valid bit for a virtual page is off, a page fault occurs. The operating system must be given control. This transfer is done with the exception mechanism, which we saw in Chapter 4 and will discuss again later in this section. Once the operating system gets control, it must find the page in the next level of the hierarchy (usually flash memory or magnetic disk) and decide where to place the requested page in main memory.

The virtual address alone does not immediately tell us where the page is on disk. Returning to our library analogy, we cannot find the location of a library book on the shelves just by knowing its title. Instead, we go to the catalog and look up the book, obtaining an address for the location on the shelves, such as the Library of Congress call number. Likewise, in a virtual memory system, we must keep track of the location on disk of each page in virtual address space.

Because we do not know ahead of time when a page in memory will be replaced, the operating system usually creates the space on flash memory or disk for all the pages of a process when it creates the process. This space is called the **swap space**. At that time, it also creates a data structure to record where each virtual page is stored on disk. This data structure may be part of the page table or may be an auxiliary data structure indexed in the same way as the page table. [Figure 5.28](#) shows the organization when a single table holds either the physical page number or the disk address.

The operating system also creates a data structure that tracks which processes and which virtual addresses use each physical page. When a page fault occurs, if all the pages in main memory are in use, the operating system must choose a page to replace. Because we want to minimize the number of page faults, most operating systems try to choose a page that they hypothesize will not be needed in the near future. Using the past to predict the future, operating systems follow the *least recently used* (LRU) replacement scheme, which we mentioned in Section 5.4. The operating system searches for the least recently used page, assuming that a page that has not been used in a long time is less likely to be needed than a more recently accessed page. The replaced pages are written to swap space on the disk. In case you are wondering, the operating system is just another process, and these tables controlling memory are in memory; the details of this seeming contradiction will be explained shortly.

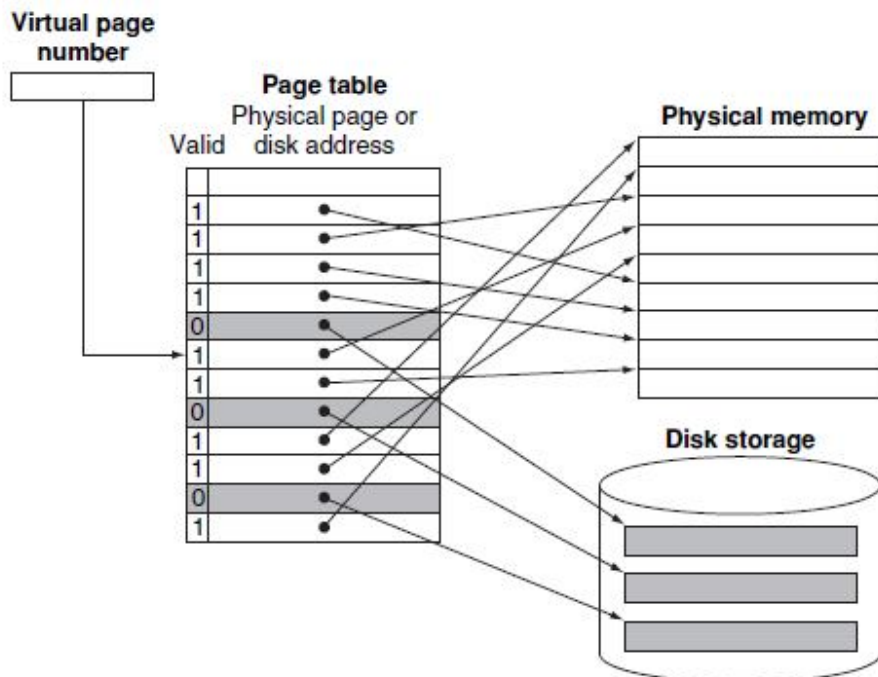


FIGURE 5.28 The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy. The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e., the starting address of the page in memory) corresponding to the virtual page. If the valid bit is off, the page currently resides only on disk, at a specified disk address. In many systems, the table of physical page addresses and disk page addresses, while logically one table, is stored in two separate data structures. Dual tables are justified in part because we must keep the disk addresses of all the pages, even if they are currently in main memory. Remember that the pages in main memory and the pages on disk are the same size.

Elaboration: With a 32-bit virtual address, 4 KiB pages, and 4 bytes per page table entry, we can compute the total page table size:

$$\text{Number of page table entries} = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$\text{Size of page table} = 2^{20} \text{ page table entries} \times 2^2 \frac{\text{bytes}}{\text{page table entry}} = 4 \text{ MiB}$$

That is, we would need to use 4 MiB of memory for each program in execution at any time. This amount is not so bad for a single process. What if there are hundreds of processes running, each with their own page table? And how should we handle 64-bit addresses, which by this calculation would need 2^{52} words?

Making Address Translation Fast: the TLB

Since the page tables are stored in main memory, every memory access by a program can take at least twice as long: one memory access to obtain the physical address and a second access to get the data. The key to improving access performance is to rely on locality of reference to the page table. When a translation for a virtual page number is used, it will probably be needed again in the near future, because the references to the words on that page have both temporal and spatial locality.

Accordingly, modern processors include a special cache that keeps track of recently used translations. This special address translation cache is traditionally referred to as a **translation-lookaside buffer (TLB)**, although it would be more accurate to call it a translation cache. The TLB corresponds to that little piece of paper we typically use to record the location of a set of books we look up in the card catalog; rather than continually searching the entire catalog, we record the location of several books and use the scrap of paper as a cache of Library of Congress call numbers.

Figure 5.29 shows that each tag entry in the TLB holds a portion of the virtual page number, and each data entry of the TLB holds a physical page number.

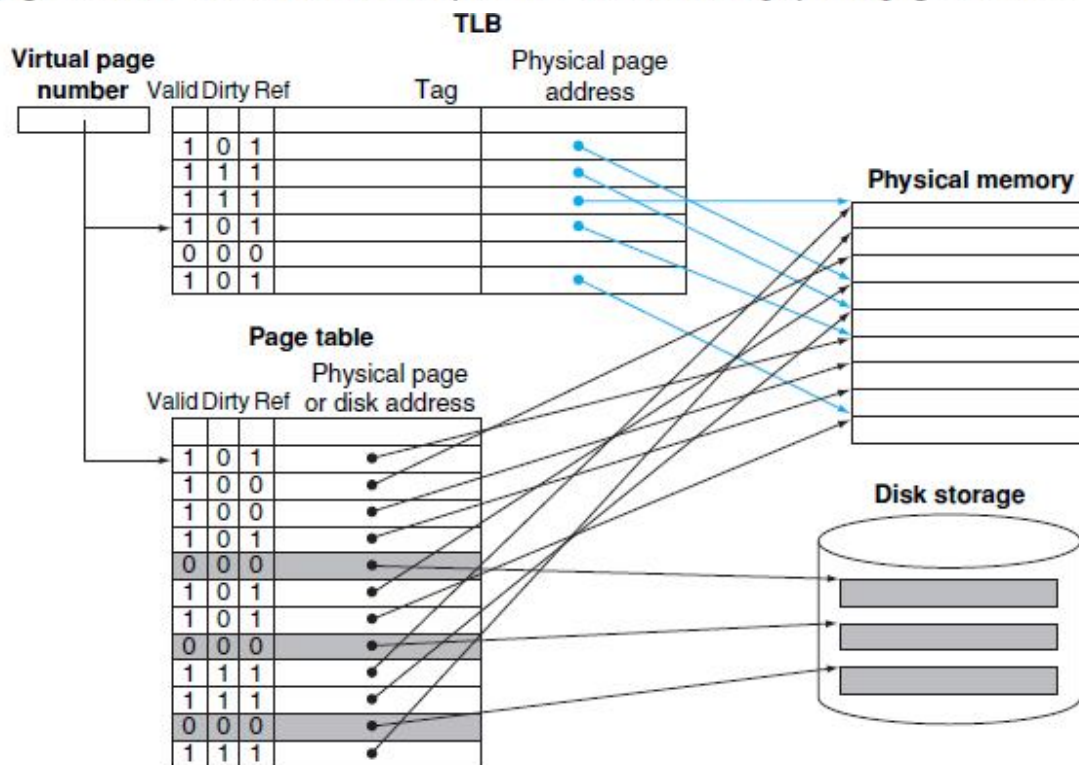


FIGURE 5.29 The TLB acts as a cache of the page table for the entries that map to physical pages only. The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. The TLB mappings are shown in color. Because the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. The page table either supplies a physical page number for the page (which can then be used to build a TLB entry) or indicates that the page resides on disk, in which case a page fault occurs. Since the page table has an entry for every virtual page, no tag field is needed; in other words, unlike a TLB, a page table is *not* a cache.

Some typical values for a TLB might be

- TLB size: 16–512 entries
- Block size: 1–2 page table entries (typically 4–8 bytes each)
- Hit time: 0.5–1 clock cycle
- Miss penalty: 10–100 clock cycles
- Miss rate: 0.01%–1%

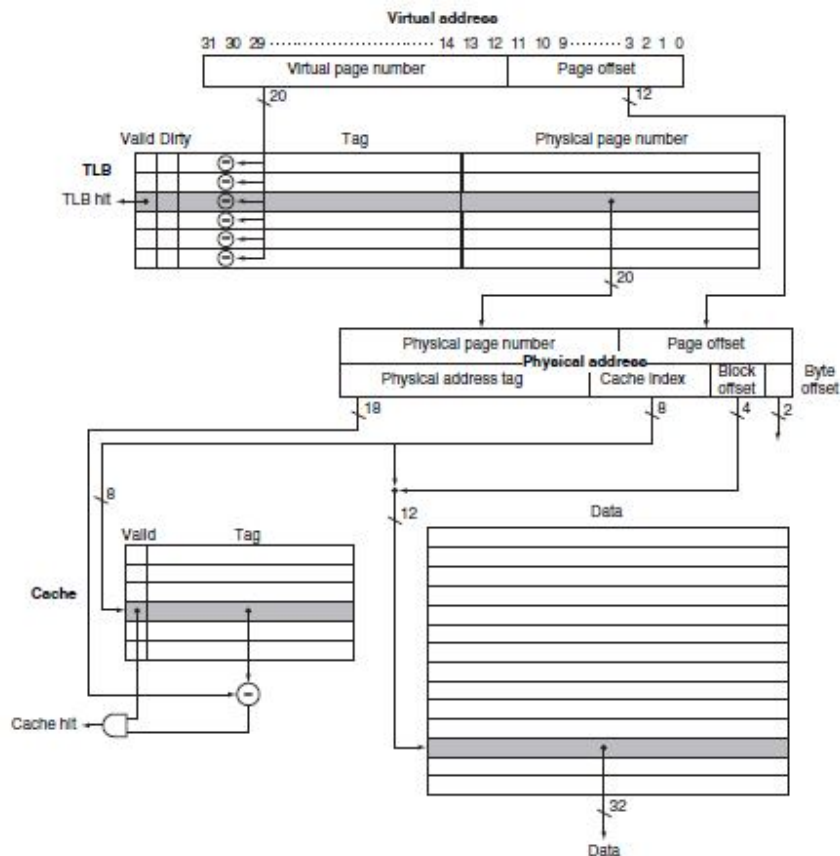
Designers have used a wide variety of associativities in TLBs. Some systems use small, fully associative TLBs because a fully associative mapping has a lower miss rate; furthermore, since the TLB is small, the cost of a fully associative mapping is not too high. Other systems use large TLBs, often with small associativity. With a fully associative mapping, choosing the entry to replace becomes tricky since implementing a hardware LRU scheme is too expensive. Furthermore, since TLB misses are much more frequent than page faults and thus must be handled more cheaply, we cannot afford an expensive software algorithm, as we can for page faults. As a result, many systems provide some support for randomly choosing an entry to replace. We'll examine replacement schemes in a little more detail in Section 5.8.

Integrating Virtual Memory, TLBs, and Caches

Our virtual memory and cache systems work together as a hierarchy, so that data cannot be in the cache unless it is present in main memory. The operating system helps maintain this hierarchy by flushing the contents of any page from the cache when it decides to migrate that page to disk. At the same time, the OS modifies the page tables and TLB, so that an attempt to access any data on the migrated page will generate a page fault.

Under the best of circumstances, a virtual address is translated by the TLB and sent to the cache where the appropriate data is found, retrieved, and sent back to the processor. In the worst case, a reference can miss in all three components of the memory hierarchy: the TLB, the page table, and the cache. The following example illustrates these interactions in more detail.

FIGURE 5.30 The TLB and cache implement the process of going from a virtual address to a data item in the Intrinsic FastMATH. This figure shows the organization of the TLB and the data cache, assuming a 4 KiB page size. This diagram focuses on a read; Figure 5.31 describes how to handle writes. Note that unlike Figure 5.12, the tag and data RAMs are split. By addressing the long but narrow data RAM with the cache index concatenated with the block offset, we select the desired word in the block without a 16:1 multiplexor. While the cache is direct mapped, the TLB is fully associative. Implementing a fully associative TLB requires that every TLB tag be compared against the virtual page number, since the entry of interest can be anywhere in the TLB. (See content addressable memories in the *Elaboration* on page 408.) If the valid bit of the matching entry is on, the access is a TLB hit, and bits from the physical page number together with bits from the page offset form the index that is used to access the cache.



Implementing Protection with Virtual Memory

Perhaps the most important function of virtual memory today is to allow sharing of a single main memory by multiple processes, while providing memory protection among these processes and the operating system. The protection mechanism must ensure that although multiple processes are sharing the same main memory, one renegade process cannot write into the address space of another user process or into the operating system either intentionally or unintentionally. The write access bit in the TLB can protect a page from being written. Without this level of protection, computer viruses would be even more widespread.

Interrupts

Defn: an event external to the currently executing process that causes a change in the normal flow of instruction execution; usually generated by hardware devices external to the CPU

- From "Design and Implementation of the FreeBSD Operating System", Glossary
- Key point is that interrupts are asynchronous w.r.t. current process
 - Typically indicate that some device needs service

Why Interrupts?

- People like connecting devices
 - A computer is much more than the CPU
 - Keyboard, mouse, screen, disk drives
 - Scanner, printer, sound card, camera, etc.
 - These devices occasionally need CPU service
 - But we can't predict when
 - External events typically occur on a macroscopic timescale
 - we want to keep the CPU busy between events
- 👉 **Need a way for CPU to find out devices need attention**

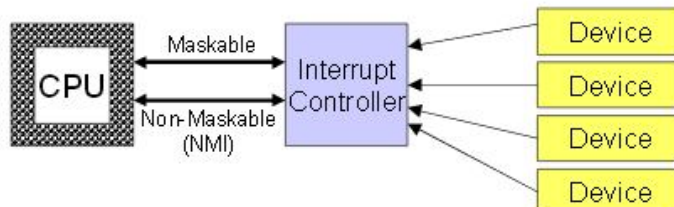
Possible Solution: Polling

- CPU periodically checks each device to see if it needs service
 - ✗ takes CPU time even when no requests pending
 - ✗ overhead may be reduced at expense of response time
 - ✓ can be efficient if events arrive rapidly

"Polling is like picking up your phone every few seconds to see if you have a call. ..."

Alternative: Interrupts

- Give each device a wire (interrupt line) that it can use to signal the processor
 - When interrupt signaled, processor executes a routine called an interrupt handler to deal with the interrupt
 - No overhead when no requests pending



Polling vs. Interrupts

"Polling is like picking up your phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring."

- Interrupts win if processor has other work to do and event response time is not critical
- Polling can be better if processor has to respond to an event ASAP
 - May be used in device controller that contains dedicated secondary processor

Hardware Interrupt Handling

- Details are architecture dependent!
- Interrupt controller signals CPU that interrupt has occurred, passes interrupt number
 - Interrupts are assigned priorities to handle simultaneous interrupts
 - Lower priority interrupts may be disabled during service
- CPU senses (checks) interrupt request line after every instruction; if raised, then:
 - uses interrupt number to determine which handler to start
 - interrupt vector associates handlers with interrupts
- Basic program state saved (as for system call)
- CPU jumps to interrupt handler
- When interrupt done, program state reloaded and program resumes

Software Interrupt Handling

- Typically two parts to interrupt handling
 - The part that has to be done immediately
 - So that device can continue working
 - The part that should be deferred for later
 - So that we can respond to the device faster
 - So that we have a more convenient execution context
 - What does that mean?

Interrupt Context

- Execution of first part of interrupt handler "borrows" the context of whatever was interrupted
 - Interrupted process state is saved in process structure
 - Handler uses interrupted thread's kernel stack
 - Have to be very careful about stack-allocated data
 - Handler is not allowed to block
 - Has no process structure of its own to save state or allow rescheduling
 - Can't call functions that might block (like kmalloc)
- Handler needs to be kept fast and simple
 - Typically sets up work for second part, flags that second part needs to execute, and re-enables interrupt

Software Interrupts

- The deferred parts of interrupt handling are sometimes referred to as "software interrupts"
 - In Linux, they are referred to as "bottom halves"
 - The terminology here is inconsistent and confusing
- What things can be deferred?
 - Networking
 - time-critical work → copy packet off hardware, respond to hardware
 - Deferred work → process packet, pass to correct application
 - Timers
 - Time-critical → increment current time-of-day
 - Deferred → recalculate process priorities

Signals

- Software equivalent of hardware interrupts
- Allows process to respond to asynchronous external events
 - Process may specify its own signal handlers or may use OS default action
 - Defaults include
 - Ignoring the signal
 - Terminating all threads in the process (with or without a core dump)
 - Stopping all threads in the process
 - Resuming all threads in the process
- Provide a simple form of inter-process communication (IPC)

Basics

- Process structure has flags for possible signals and actions to take
- When signal is posted to process, signal pending flag is marked
- When process is next scheduled to run, pending signals are checked and appropriate action is taken
 - Signal delivery is not instantaneous