1

Computer Fundamentals and Performance Evaluation

Welcome to read "Computer Principles and Design in Verilog HDL". This book starts from the very beginning – introducing the basic logic operations. You will learn that any digital circuit can be designed by using three kinds of logic gates: the AND gate, the OR gate, and the NOT gate. Then the methods of designing the basic components of the CPU (central processing unit), such as the adder, the subtracter, the shifter, the ALU (arithmetic logic unit), and register file, by using these gates, will be presented. After introducing the designs of various simple CPUs, this book describes how to design a multicore CPU with each core containing an IU (integer unit), an FPU (floating-point unit), an instruction cache, a data cache, an instruction TLB (translation lookaside buffer), a data TLB, and the mechanism of interrupt/exceptions.

The design of popular I/O (input/output) interfaces, such as the UART (universal asynchronous receiver transmitter), PS/2 keyboard/mouse and VGA (video graphics array) interfaces, the I2C (inter-integrated circuit) serial bus controller, and the PCI (peripheral component interconnect) parallel bus controller, will be also described. Except for the PS/2 mouse interface controller, all of the circuits of CPUs, I/O interfaces, and bus controllers were designed in Verilog HDL (hardware description language) and simulated with ModelSim. The Verilog HDL source codes and simulation waveforms are also included in the book. Finally, the book describes how to design an interconnection network for connecting multiple CPU and memory modules together to construct a high-performance supercomputer.

This chapter introduces some basic concepts, the organization of modern computers, and the evaluation method of computer performance.

1.1 Overview of Computer Systems

Suppose that you have a personal computer, which has 8 GB memory and the clock frequency is 4 GHz. Question: Are the "G" in 8 GB and "G" in 4 GHz same? The answer will be given in the last part of this section. Before that, we introduce the terminologies of "computer" and "computer system", a brief history of the computer, instruction set architectures (ISAs), and the differences between RISC (reduced instruction set computer) and CISC (complex instruction set computer).

1.1.1 Organization of Computer Systems

You may know what a "single-chip computer" is, or have heard about it (it does not matter even if you have never). It is an IC (integrated circuit) chip, in which there is a CPU (or a microprocessor), a small amount of memory, and some I/O interface controllers. Generally, any IC chip or printed circuit board that contains these three kinds of components is called a computer.

Computer Principles and Design in Verilog HDL, First Edition. Yamin Li.

Companion Website: www.wiley.com/go/li/verilog

Then, one more question: is it possible for an end user to use this computer directly? The answer is "no". The computer we often say is actually a "computer system". A computer system consists of not only a computer, but also the software, I/O devices, and power supply.

The essential software is an operating system (OS). It manages all the resources of the computer, I/O devices, and other software, and provides an interface for users to use the computer system. The MS-DOS (Microsoft disk operating system) was the early operating system for IBM PC. The various versions of Windows were then provided with graphical user interfaces (GUIs). Another OS is the open-source Linux kernel. The various versions of the operating systems that support GUI, such as Fedora, Ubuntu, OpenSUSE, CentOS, and Scientific Linux, were developed on the top of the Linux kernel. All of the above are called "system software".

Software is a collection of programs and related data. A program provides the instructions for telling a computer what to do and how to do it. The instructions are represented in binary format that a computer can understand. An executable program is usually generated by a compiler, based on the source codes of the program which are prepared by the programmers. Programmers develop source codes in a high-level programming language, C for instance, with a text editor. A debugger is often used by programmers to debug the program. The compiler, editor, debugger, and other libraries are also programs, sometimes we call them "utilities". All programs other than the system software and utilities are called "applications".

I/O devices, also known as peripheral devices, are like the wheels of a car. The keyboard, mouse, display, hard disk, and printer are typical examples of I/O devices. A network can also be considered as an I/O device. Other I/O devices include the scanner, video camera, microphone, speaker, CD/DVD drive, and so on.

As mentioned above, a computer consists of a CPU, memory, and I/O interfaces. An I/O interface is a hardware controller that makes the communication between an I/O device and the CPU (and memory) possible. Memory provides places where the programs and data can be stored. The CPU reads instructions and data from memory and executes the instructions that perform operations on data.

Inside a CPU, there are multiplexers, ALUs, FPUs, register files, and a control unit which decodes instructions and controls the operations of all other components. Because the speed of the memory is much smaller than that of the CPU, in modern CPUs there is an instruction cache and a data cache. For the purpose of the fast virtual address translation, an instruction TLB and a data TLB are also fabricated in the CPUs. This book focuses on the computer and computer design in Verilog HDL.

As a summary, Figure 1.1 illustrates the organization of a computer system. You can see that your personal computer is not a computer, but is a computer system. A high-end mobile phone can also be considered as a computer system.

The computer and I/O devices belong to computer hardware. Therefore, we can say that a computer system consists of the computer hardware and the computer software (and a power supply).

1.1.2 A Brief History of the Computer

One of the earliest machines designed to assist people in calculations was said to be the Abacus, which was invented about 4400 years ago and is still being used by merchants, traders, and clerks in Asia, Africa, and elsewhere. Since then, various mechanical and electrical analog computers were invented. From 1940, computers entered the electronic era. Electronic computers can be classified into four generations. Each generation used a new technology to build computers.

The first-generation computers (1946–1955) were distinguished primarily for their use of vacuum tubes as the main electronic components. Several special-purpose electronic computers were developed between 1940 and 1945. The electronic numerical integrator and computer (ENIAC) built in 1946 was said to be the first general-purpose digital computer. However, ENIAC had an architecture that required rewiring a plug-board to change its programming. Computers of this generation could only perform a single task, and they had no operating system.

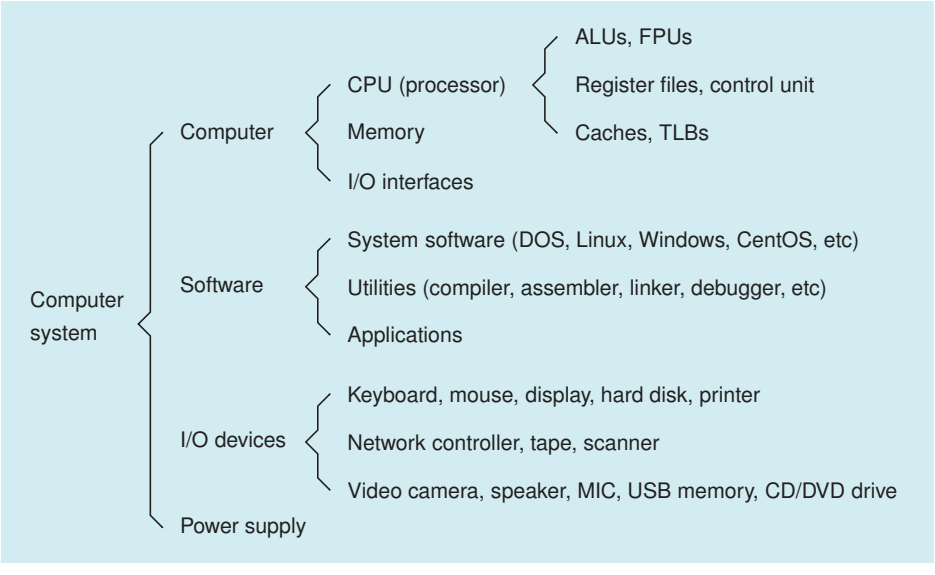

Figure 1.1 Computer system organization

The second-generation computers (1956–1963) used transistors as the processing elements and magnetic cores as their memory. Transistors were invented in 1947 as an alternative to vacuum tubes for use as electronic switches. The transistor was far superior to the vacuum tube, allowing computers to become smaller, faster, cheaper, more energy-efficient and more reliable than their first-generation predecessors. One of the most successful second-generation computers was the IBM 1401, which was introduced in 1959. By 1964, IBM had installed more than 100,000 units, capturing about one-third of the world market. Assembly programming languages became the major tool for software development, instead of the machine languages used in the first-generation computers. Operating systems and high-level programming languages were also being developed at this time.

The third-generation computers (1964–1970) were characterized by the transition from transistors to IC chips. The IC chip was invented in 1958 by Jack Kilby of Texas Instruments and Robert Noyce of Fairchild Semiconductor Corporation. In 1964, IBM announced System/360, one of the first computers to use ICs as its main processing technology. System/360 was designed to cover the complete range of applications, from small to large, both commercial and scientific. The design made a clear distinction between architecture and implementation, allowing IBM to release a suite of compatible designs at different prices. The number of transistors on an IC chip has been doubling approximately every 2 years, and the rate has held strong for more than half a century. The nature of this trend was first proposed by the Intel cofounder, Gordon Moore, in 1965.

The fourth-generation computers (1971 to the present) were distinguished primarily by their use of microprocessors and semiconductor memory. These were made possible by the improvements in IC design and manufacturing methods, which allowed engineers to create IC chips with tens of thousands of transistors (later hundreds of thousands, then millions, and now billions), a process now known as very large scale integration (VLSI). The personal computers first appeared during this time. In 1980, the MS-DOS was born, and in 1981 IBM introduced the personal computer for home and office use. Three years later, Apple gave us the Macintosh computers. The IBM PC used Intel 8086 as its microprocessor. The ubiquity of the PC platform has resulted in the Intel x86 becoming one of the most popular CPU architectures.

1.1.3 Instruction Set Architecture

The execution of a program is the job of the CPUs. A CPU can execute only the binary-format instructions it understands. A particular CPU has its own ISA. Generally, an instruction must consist of at least an operation code (opcode), which defines what will be done. Other parts that may be contained in an instruction include how to get the source operands and the place at which the execution result is stored.

A source operand may be an immediate or a data word in a register or in the memory. An immediate is a constant, which is given in the instruction directly. A data word in a register or memory is a variable. Of course, it is also allowed to put constants in memory. There may be several registers inside a CPU. All of the registers form a register file. Each register in the register file has a unique number. This register number is given in the instruction if a source operand is the register data.

The operation types of an ISA can be divided into the following: (i) integer arithmetic and logic calculations; (ii) data movement between register file and memory; (iii) conditional branches and unconditional jumps; (iv) subroutine call and return; (v) calculations on floating-point numbers; (vi) I/O accesses; and (vii) system controls, such as system calls, return from exceptions, and TLB manipulations.

Let's see some instruction examples of the Intel x86 and MIPS (microprocessor without interlocked pipeline stages) CPU. The following is a function written in C that calculates the 32-bit unsigned product of two 16-bit unsigned numbers.

```
unsigned int mul16 (unsigned int x, unsigned int y) { // mul by shift
    unsigned int a, b, c;                             // c = a * b
    unsigned int i;                                   // counter
    a = x;                                            // multiplicand
    b = y;                                            // multiplier
    c = 0;                                            // product
    for (i = 0; i < 16; i++) {                        // for 16 bits
        if ((b & 1) == 1) {                           // LSB of b is 1
            c += a;                                   // c = c + a
        }
        a = a << 1;                                   // shift a 1-bit left
        b = b >> 1;                                   // shift b 1-bit right
    }
    return(c);                                        // return product
}
```

Of course, you can use `c = a * b` to get the product, but we show here how to use the addition and shift operations to perform the multiplication. The following x86 assembly codes were generated by running the `gcc -O4 -S` command on an x86 machine under CentOS.

```
 1:  mul16:
 2:          pushl   %ebp
 3:          movl    %esp, %ebp
 4:          movl    8(%ebp), %ecx
 5:          pushl   %ebx
 6:          movl    12(%ebp), %edx
 7:          xorl    %ebx, %ebx
 8:          movl    $15, %eax
 9:          .p2align 2,,3
10:  .L6:
```

```
11:          testb   $1, %dl
12:          je      .L5
13:          addl    %ecx, %ebx
14:  .L5:
15:          sall    %ecx
16:          shrl    %edx
17:          decl    %eax
18:          jns     .L6
19:          movl    %ebx, %eax
20:          popl    %ebx
21:          leave
22:          ret
```

We do not explain all the instructions in the assembly codes above. The only instruction we examine is `addl %ecx, %ebx` in line 13. It performs the operation of `c += a` in the C codes. The `addl` (add long) points out that it is a 32-bit addition instruction; `%ecx` and `%ebx` are the registers of the two source operands; and `%ebx` is also the destination register. That is, `addl %ecx, %ebx` instruction adds the contents of the `ecx` and `ebx` registers, and places the sum in the `ebx` register. We can see that the x86 ISA has a two-operand format.

If we use the `gcc -O4 -S` command to compile the same C codes on a MIPS machine, we will get the following assembly codes:

```
 1:  mul16:
 2:          move    $6, $0
 3:          li      $3, 15
 4:  $L6:
 5:          andi    $2, $5, 0x1
 6:          addiu   $3, $3, -1
 7:          beq     $2, $0, $L5
 8:          srl     $5, $5, 1       # delay slot
 9:          addu    $6, $6, $4
10:  $L5:
11:          bgez    $3, $L6
12:          sll     $4, $4, 1       # delay slot
13:          j       $31
14:          move    $2, $6          # delay slot
```

The instruction in the ninth line, `addu $6, $6, $4`, performs the `c += a`. The `addu` (add unsigned) points out that it is an unsigned addition instruction; the first `$6` is the destination register and the others are the two source registers. That is, this instruction adds the contents of the `$6` and `$4` registers, and places the sum in the `$6` register. We can see that the MIPS ISA has a three-operand format. Note the instruction order in the MIPS assembly codes: The instructions `srl`, `sll`, and `move`, following-up the `beq`, `bgez`, and `j`, respectively, are always executed before the control is transferred to the target address. This feature is named delayed branch, which we will describe in detail in Chapter 8.

There are ISAs that have a one-operand format. In the CPUs that implement the one-operand ISAs, a special register, called an accumulator, acts as a default source register and the default destination register. It doesn't appear in the instruction. The instruction needs only to give a register name or a memory address for the second source operand. Z80 and 6502 are examples of the one-operand ISAs.

Table 1.1 Category of instruction set architecture

Register/memory-oriented		Accumulator-oriented	Stack-oriented
Three-operand	Two-operand	One-operand	Zero-operand
`add x, y, z`	`add x, y`	`add x`	`add`

Are there ISAs that have no (zero) operand? The answer is "yes". Such ISAs are stack-oriented. The two source operands are popped from the top of the stack and the result is pushed onto the stack. The stack-top pointer is adjusted automatically according to the operation of the instruction. The Bytecode, an ISA of JVM (Java virtual machine), is a typical example of the zero-operand ISAs.

Table 1.1 summarizes the four `add` formats of ISAs where `x`, `y`, and `z` are the register numbers, or some of them are memory addresses.

All the instructions are represented in binary numbers. In x86 ISA, there are eight registers: `eax`, `ebx`, `ecx`, `edx`, `ebp`, `esp`, `esi`, and `edi`. Therefore, the encoding of a register has three bits ($\log_2 8 = 3$ or $2^3 = 8$). To shorten program codes, x86 uses a short opcode to encode the very commonly used instructions. The encodings of the x86 assembly program are shown below. From this we can see that the length of the instruction encodings is not fixed. If we do not decode the current instruction, we cannot know from where the next instruction starts. This feature makes the design of a pipelined x86 CPU difficult.

```
 1:  mul16:
 2:          pushl   %ebp                ; 01010101
 3:          movl    %esp, %ebp          ; 1000100111100101
 4:          movl    8(%ebp), %ecx       ; 100010000100110100001000
 5:          pushl   %ebx                ; 01010011
 6:          movl    12(%ebp), %edx      ; 100010110101010100001100
 7:          xorl    %ebx, %ebx          ; 0011000111011011
 8:          movl    $15, %eax           ; 1011100000001111
 9:          .p2align 2,,3               ; 00000000000000000000000000
                                         ; 10001101011011000000000
10:  .L6:
11:          testb   $1, %dl             ; 11110110110000010000000001
12:          je      .L5                 ; 0111010000000010
13:          addl    %ecx, %ebx          ; 0000000111001011
14:  .L5:
15:          sall    %ecx                ; 1101000111100001
16:          shrl    %edx                ; 1101000111101010
17:          decl    %eax                ; 01001000
18:          jns     .L6                 ; 0111100111110010
19:          movl    %ebx, %eax          ; 1000100111011000
20:          popl    %ebx                ; 01011011
21:          leave                       ; 11001001
22:          ret                         ; 11000011
```

The following shows the encodings of the MIPS assembly codes. MIPS32 ISA has a general-purpose register file that contains thirty-two 32-bit registers and hence a register number has five bits. We can see that the length of the MIPS instructions is fixed: All the instructions are represented with 32 bits. This feature makes the design of a pipelined MIPS CPU easy.

```
 1:  mul16:
 2:          move    $6, $0              # 00000000000000000011000000100001
 3:          li      $3, 15              # 00100100000000110000000000001111
 4:  $L6:
 5:          andi    $2, $5, 0x1         # 00110000101000100000000000000001
 6:          addiu   $3, $3, -1          # 00100100011000111111111111111111
 7:          beq     $2, $0, $L5         # 00010000010000000000000000000010
 8:          srl     $5, $5, 1           # 00000000000001010010100001000010
 9:          addu    $6, $6, $4          # 00000000110001000011000000100001
10:  $L5:
11:          bgez    $3, $L6             # 00000100011000011111111111111010
12:          sll     $4, $4, 1           # 00000000000001000010000001000000
13:          j       $31                 # 00000011111000000000000000001000
14:          move    $2, $6              # 00000000110000000001000000100001
```

Although the length of each MIPS instruction is longer than that of some x86 instructions, the operation of each MIPS instruction is very simple. We say that the MIPS belongs to RISC and the x86 belongs to CISC.

1.1.4 CISC and RISC

CISC is the general name for CPUs that have a complex instruction set. An instruction set is said to be complex if there are some instructions that perform complex operations or the instruction formats are not uniform. The Intel x86 family, Motorola 68000 series, PDP-11, and VAX are examples of CISC.

The CISC instruction set tries to enhance the code density so that a computer system can use a small amount of memory, including cache, to store as many instructions as possible for reducing the cost and improving the performance.

CISC adopts two measures to reduce the code size – it lets an instruction perform as many operations as possible and makes the encoding of each instruction as short as possible. The first measure results in using microcode to implement the complex instructions, and the second measure results in a variable length of the instruction formats. As a consequence, it becomes difficult to design and implement a pipelined CISC CPU to obtain substantial performance improvements.

Is every instruction in a CISC complex? No. There are some very simple instructions in a CISC. The analysis of the instruction mix generated by CISC compilers shows that about 80% of executed instructions in a typical program uses only 20% of an instruction set and these instructions perform the simple operations and use only the simple addressing modes. We can see that almost all instructions in the x86 codes listed in Section 1.1.3 are simple instructions.

RISC is the general name for CPUs that have a small number of simple instructions. In the 1980s, the team headed by David Patterson of the University of California at Berkeley investigated the existing ISAs, proposed the term of RISC, and made two CPU prototypes: RISC-I and RISC-II. This concept was adopted in the designs of Sun Microsystems' SPARC microprocessors. Meanwhile, the team headed by John Hennessy of Stanford University did similar research and created MIPS CPUs. Actually, John Cocke of IBM Research originated the RISC concept in the project IBM 801, initiated from 1974. The first computer to benefit from this project was IBM PC/RT (RISC technology), the ancestor of the RS/6000 series.

There are two main features in a RISC CPU. One is the fixed length of the instruction formats. This feature makes fetching an instruction in one clock cycle possible. The other feature is the so-called load/store architecture. It means that only the load and store instructions transfer data between the register file and memory, and other instructions perform operations on the register operands. This feature makes the operations of the RISC instructions simple. Both features make the design of the pipelined RISC CPUs easier than that of the CISC CPUs.

Table 1.2 Some base units

Powers of 2				Powers of 10					
Memory capacity				Clock frequency			Cycle length		
K	kilo	2^{10}	1,024	K	kilo	10^{3}	m	milli	10^{-3}
M	mega	2^{20}	1,048,576	M	mega	10^{6}	μ	micro	10^{-6}
G	giga	2^{30}	1,073,741,824	G	giga	10^{9}	n	nano	10^{-9}
T	tera	2^{40}	1,099,511,627,776	T	tera	10^{12}	p	pico	10^{-12}
P	peta	2^{50}	1,125,899,906,842,624	P	peta	10^{15}	f	femto	10^{-15}
E	exa	2^{60}	1,152,921,504,606,846,976	E	exa	10^{18}	a	atto	10^{-18}
Z	zetta	2^{70}	1,180,591,620,717,411,303,424	Z	zetta	10^{21}	z	zepto	10^{-21}
Y	yotta	2^{80}	1,208,925,819,614,629,174,706,176	Y	yotta	10^{24}	y	yocto	10^{-24}

The SUN Microsystems SPARC, AMD 29000 family, SGI MIPS, IBM PowerPC, HP PA-RISC, and ARM are examples of RISC.

Was the CISC replaced by RISC? No. The reason is simple – the market. The Intel x86 ISA is widely used in the IBM-compatible PC, which is the most common computer system in the world. There is a huge amount of software resources that we cannot throw away.

Today, most CISC CPUs use a decoder to convert CISC instructions into RISC instructions (micro-operations) and then use RISC cores to execute these instructions. Meanwhile, many RISC CPUs add more new instructions to support the complex operations, the multimedia operations for instance.

1.1.5 Definitions of Some Units

Now, we answer the question raised in the beginning of this section. The G in 8 GB and G in 4 GHz are different. The first G equals $2^{30} = 1{,}073{,}741{,}824$; the second G equals $10^{9} = 1{,}000{,}000{,}000$. A power of 10 is used for denoting the clock frequency; a power of 2 is used for denoting the memory capacity. Table 1.2 lists some of the base units.

The unit of byte is often used for counting the digital information, which commonly consists of 8 bits. Other units include half word (16 bits), word (32 bits), and long word (64 bits). Note that the x86 defines a word as 16 bits, a long word as 32 bits, and a quad word as 64 bits (only supported by x86-64).

1.2 Basic Structure of Computers

As we described in the previous section, a computer consists of a CPU, memory, and I/O interfaces. This section describes these three components briefly.

1.2.1 Basic Structure of RISC CPU

The job of a CPU is to execute instructions. Figure 1.2 shows a simplified structure of a RISC CPU. The instruction is fetched from the instruction memory. The content of PC (program counter) is used as the address of the instruction memory. In the register file, there are a certain number of registers which can store data. The ALU is responsible for calculations. Each of the two input operands of ALU can be either a register datum or an immediate provided in the instruction. The multiplexers (`mux` in the figure) are used for selecting an input from two inputs. The result calculated by ALU is saved into the register file. If the instruction is a load instruction, the data read from the data memory will be saved into the register file. If the instruction is a store instruction, the data read from the register file will be saved into the data memory. In both cases, the ALU output is used as the address of the data memory.

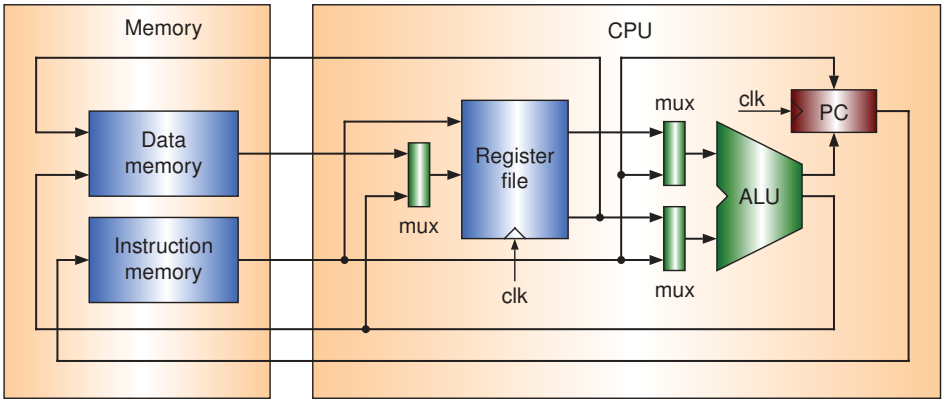

Figure 1.2 Simplified structure of RISC CPU

If the instruction is a conditional branch instruction, the flags, outputs of ALU, are used to determine whether jumping to the target address or not. The target address of the branch can be calculated by adding an immediate to the current PC. If the branch is not taken, the PC + 4 is saved into PC for fetching the next instruction (a 32-bit instruction has four bytes and the PC holds the byte address of the instruction memory). Figure 1.2 is actually a single-cycle CPU that executes an instruction in a single clock cycle.

In the single-cycle CPU, the execution of an instruction can be started only after the execution of the prior instruction has been completed. The pipelined CPU divides the execution of an instruction into several stages and allows overlapping execution of multiple instructions.

Figure 1.3 shows a simplified RISC pipelined CPU. There are five stages: (i) instruction fetch (IF), (ii) instruction decode (ID), (iii) execution (EXE), (iv) memory access (MEM), and (v) write back (WB). The pipeline registers are inserted in between the stages for storing the temporary results. Under ideal conditions, a new instruction can enter the pipeline during every clock cycle and five instructions are overlapped in execution. Therefore, the CPU throughput can be improved greatly.

From early 1980s, CPUs have been rapidly increasing in speed, much faster than the main memory. Referring to Figure 1.4, to hide the performance gap between the CPU and memory, instruction cache and data cache are fabricated on the CPU chips. A cache is a small amount of fast memory that is located in between the CPU and main memory and stores copies of the data or instructions from the most frequently used main memory locations. On a cache hit, the CPU can get data or instructions immediately without accessing the external main memory. Therefore, the average memory accessing time can be shortened.

In modern computer systems, the main memory is commonly implemented with DRAM (dynamic random access memory). DRAMs have large capacity and are less expensive, but the memory control circuit

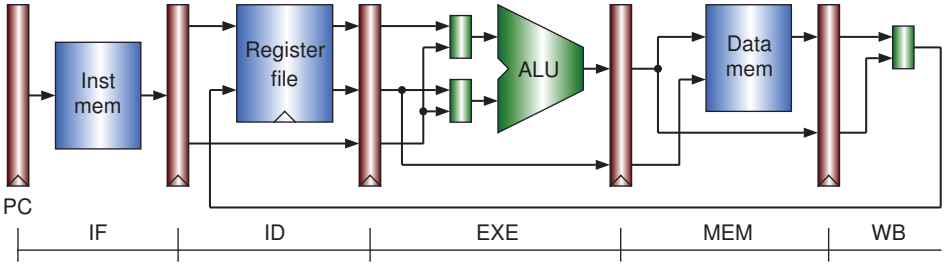

Figure 1.3 Simplified structure of pipelined CPU

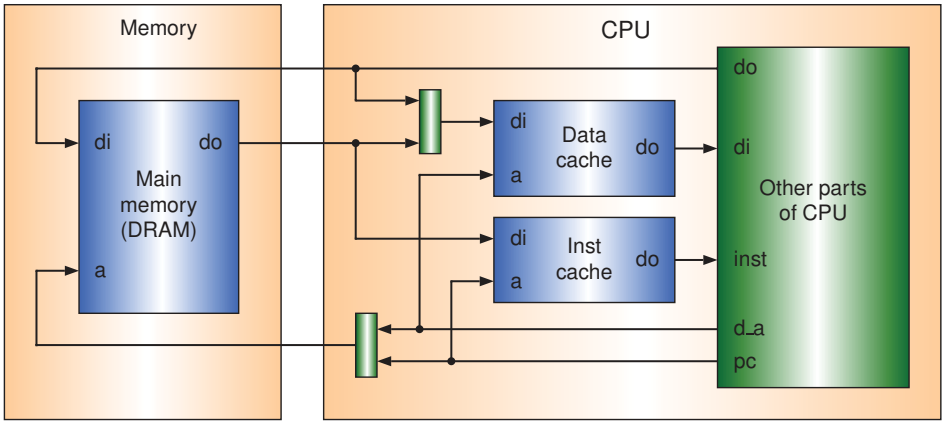

Figure 1.4 On-chip dedicated instruction cache and data cache

is complex. Some high-performance computer systems use the SRAM (static random access memory) as the main memory. The SRAM is faster but more expensive than DRAM.

1.2.2 Multithreading and Multicore CPUs

The pipelined CPU tries to produce a result on every clock cycle. The superscalar CPU tries to produce multiple results on every cycle by means of fetching and executing multiple instructions in a clock cycle. However, due to the control and data dependencies between instructions, the average number of instructions executed by a superscalar CPU per cycle is about 1.2. In order to achieve this 20% performance improvement, a superscalar must be designed with heavy extra circuits, such as register renaming, reservation stations, and reorder buffers, to support the parallel executions of multiple instructions. The superscalar CPUs cannot improve the performance further because of the lack of ILP (instruction level parallelism). Although the compilers can adopt the techniques of the loop-unrolling and static instruction scheduling, the improvement in performance is still limited.

Multithreading CPUs try to execute multiple threads in parallel. A thread is a sequential execution stream of instructions. Figure 1.5 shows the structure of a simplified multithreading CPU which can execute four threads simultaneously.

Each thread has a dedicated program counter and a register file, but the instruction cache, data cache, and functional units (FUs) are shared by all the threads. Sharing FUs and caches will increase

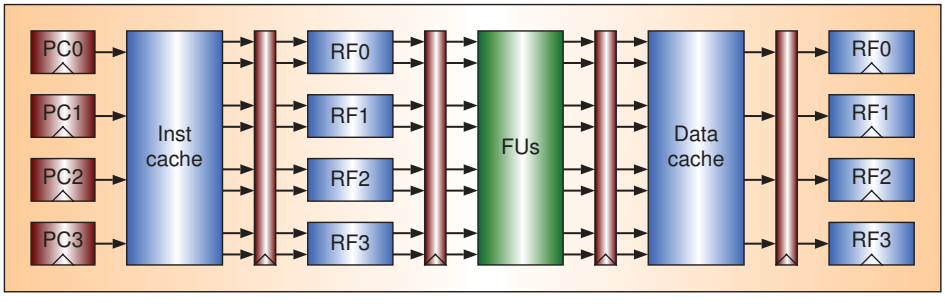

Figure 1.5 Simplified structure of a multithreading CPU

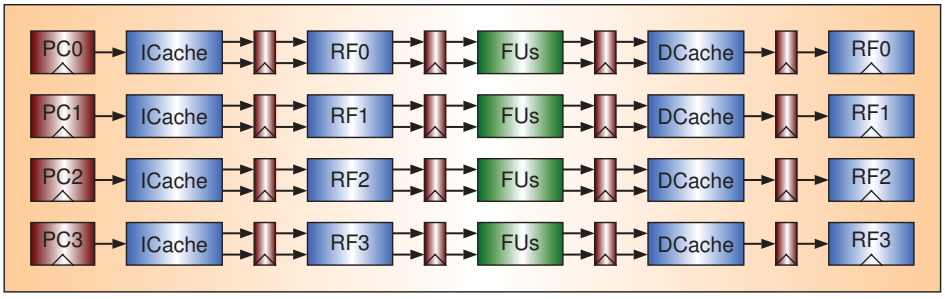

Figure 1.6 Simplified structure of a multicore CPU

the utilization of these components but make the control complex. Multithreading CPUs improve performance by exploiting the TLP (thread level parallelism).

A multicore CPU is an IC chip in which multiple ordinary CPUs (cores) are fabricated. An ordinary CPU may be a pipelined CPU, a superscalar CPU, or a multithreading CPU. Figure 1.6 shows the structure of a simplified multicore CPU in which there are four pipelined CPUs.

In a multicore CPU, each core has a dedicated level 1 (L1) instruction cache and a data cache. All the cores may share a level 2 (L2) cache, or each core has a dedicated L2 cache. Compared to the multithreading CPU, in the multicore CPU the FUs and L1 caches cannot be shared by all the cores. This decreases the utilization of the components but makes the design and implementation simple.

A multicore CPU can be considered as a small-scale multiprocessor system. The name "multicore" was given by the industry; it has almost the same meaning as the chip-multiprocessor, a name proposed by academia prior to the multicore.

1.2.3 Memory Hierarchy and Virtual Memory Management

Memory is a place in which the programs that are executing currently are stored temporarily. A program consists of instructions and data. The CPU reads instructions from memory and calculates on the data. As we have mentioned before, the speed of the memory is much smaller than that of the CPU. Therefore, high-speed caches are inserted in between the CPU and memory. Figure 1.7 shows a typical memory hierarchy. There are three levels of caches (L1, L2, and L3). An on-chip cache is one that is fabricated in the CPU chip; an off-chip cache is one that is implemented with the dedicated SRAM chip(s) outside the CPU. Commonly, the L1 on-chip cache consists of separated instruction cache and data cache, but the L2 on-chip cache is shared by instructions and data.

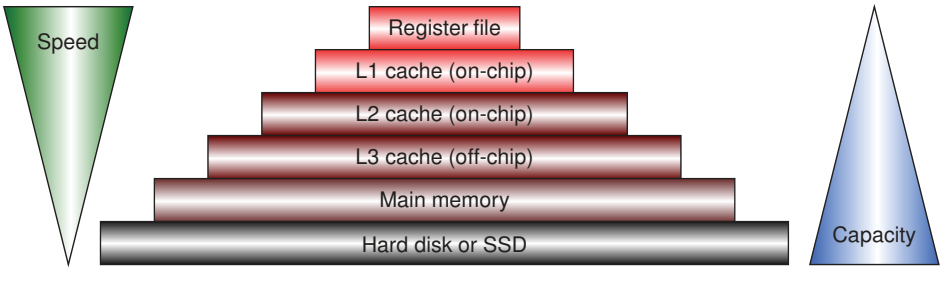

Figure 1.7 Memory hierarchy

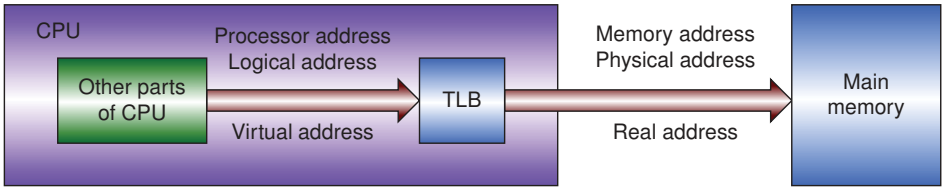

Figure 1.8 TLB maps virtual address to physical address

The register file can be considered as the topmost level of the memory hierarchy. It has a small capacity but the highest speed. The hard disk is at the bottom position of the memory hierarchy. In addition to storing files, the hard disk provides a large space for implementing the virtual memory. There is a trend that the SSD (solid-state drive) will replace the hard disk drive.

The virtual memory provides a large virtual address space for running programs (processes) to exist in. Each process uses its own virtual address space, usually starting from 0. The size of virtual address space is determined by the bits of the program counter. For example, a 32-bit program counter can access 4 GB of virtual memory. The virtual addresses cannot be used to access the main memory directly because multiple programs are allowed to be executed simultaneously. The virtual addresses must be mapped to physical memory locations. In a paging management mechanism, the virtual address space is divided into pages – blocks of contiguous virtual memory addresses. Thus, a virtual address consists of a virtual page number and an offset within a page. Only the virtual page number is needed to be mapped to a physical page number.

This mapping is handled by the operating system and the CPU. The operating system maintains a page table describing the mapping from virtual to physical pages for each process. Every virtual address issued by a running program must be mapped to the corresponding physical memory page containing the data for that location. If the operating system had to intervene in every memory access, the execution performance would be slow. Therefore, in the modern CPUs, the TLBs are fabricated for speeding up the mapping, as shown as in Figure 1.8.

The organization of the TLB is very similar to that of the cache. The TLB stores copies of the most frequently used page table entries. On a TLB hit, the physical page number can be obtained immediately without disturbing the operating system, so the actual mapping takes place very fast. Generally, the virtual address is also known as the logical address or processor address. And the physical address is also known as the real address or memory address. Note that for a particular architecture, these names may have different meanings.

1.2.4 Input/Output Interfaces and Buses

In a computer system, CPU communicates with I/O devices via I/O interfaces. There are multiple I/O devices in a computer system, and the multiple I/O interfaces are connected by a common bus, as shown as in Figure 1.9.

An I/O interface may have several I/O ports, and each I/O port has a unique I/O address. The CPU can use the I/O address to write control information to a control register and to read state information from a state register in an I/O interface. Although these two registers are different, but their I/O addresses can be the same. This is a significantly different feature from the memory, where the read information from a memory location is definitely the same as the information that was written into the same location before reading.

When an I/O device wants to communicate with the CPU, by pressing a key of the keyboard for instance, it sends an interrupt request to the CPU via the I/O interface. Once the CPU receives the request, it stops the current work and transfers control to an interrupt handler – a pre-prepared program. In the interrupt handler, the CPU can use I/O address to read or write I/O data. The Intel x86 has two special

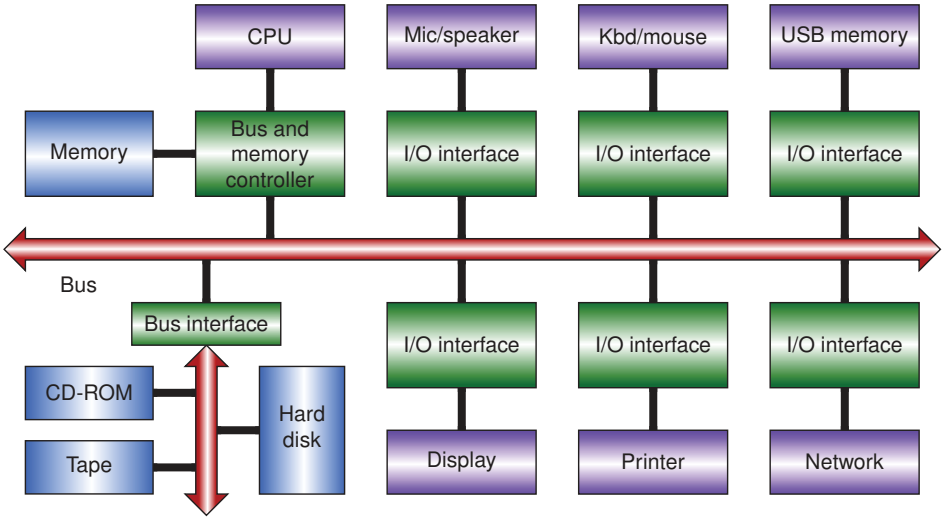

Figure 1.9 I/O interfaces in a computer system

instructions, `in` and `out`, for these operations. We say that the x86 has a dedicated I/O address space. But most of RISC CPUs use memory access instructions, `load` and `store` for example, to read and write I/O data. We say that these kinds of CPUs adopt a memory-mapped I/O address space.

1.3 Improving Computer Performance

Computer performance has improved incredibly since the first electronic computer was created. This rapid improvement came from the advances in IC technology used to build computers and the innovation in computer design. This section describes computer performance evaluation, trace-driven simulation, and high-performance parallel computers.

1.3.1 Computer Performance Evaluation

If we focus only on the execution time of real programs, then we can say that the shorter the execution time, the higher the performance. Therefore, we simply define the performance as the reciprocal of the time required. To calculate the execution time of a program, we have the following equation:

$$\text{Time} = \text{I} \times \text{CPI} \times \text{TPC} = \frac{\text{I} \times \text{CPI}}{\text{F}}$$

where I is the number of executed instructions of a program, CPI is the average clock cycles per instruction, and TPC is the time per clock cycle which is the reciprocal of the clock frequency (F).

Many researchers around the world are trying to improve computer performance by reducing the value of each of the three terms in the expression above. Architecture designers and compiler developers are trying to reduce the number of required instructions (I) of a program. Architecture and CPU designers are trying to decrease the CPI. And CPU designers and IC engineers are trying to increase the clock frequency (F).

Note that these three parameters are not independent. For example, CISC CPUs may reduce the instruction count I by providing complex instructions, but it may result in an increase of CPI; RISC CPUs may decrease CPI and TPC, but it may cause the increase of I.

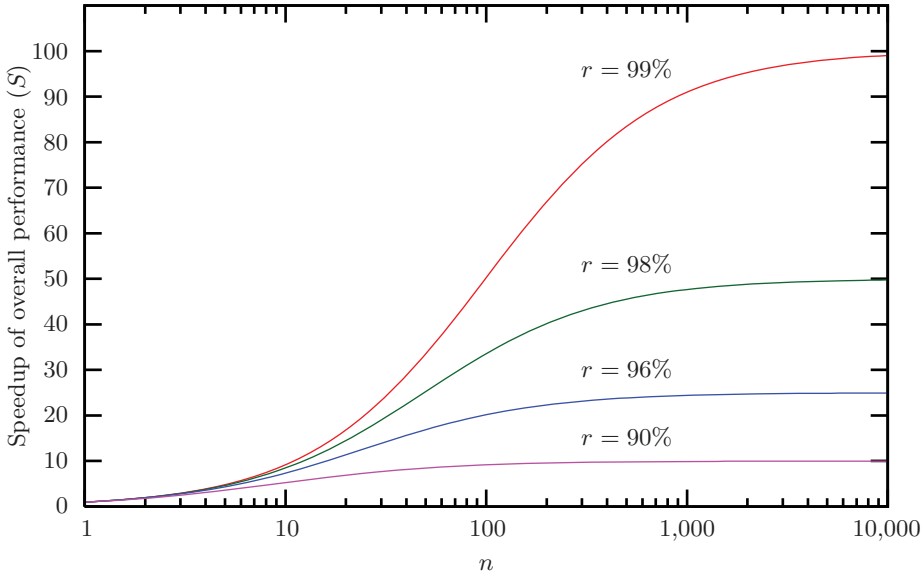

Figure 1.10 Amdahl's Law examples

Also, the clock frequency F cannot get higher unlimitedly. We know that electronic signals travel in a circuit at about two-thirds of the speed of light; this means that in a nanosecond (ns) the signal travels about 20 cm. If F = 20 GHz, in one clock cycle, the signal can travels only 1 cm. This will result in some parts of the circuit being in the "current" cycle and other parts in the "previous" cycle.

When we calculate the expected improvement to an overall system when only part of the system is improved, we often use Amdahl's Law. Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used. Let P_n be the performance with enhancement, P_o be the performance without enhancement, T_n be the execution time with enhancement, and T_o be the execution time without enhancement. Then the speedup is given by

$$S = \frac{P_n}{P_o} = \frac{T_o}{T_n} = \frac{T_o}{T_o \times r/n + T_o \times (1-r)} = \frac{1}{r/n + (1-r)}$$

where r is the fraction enhanced and n is the speedup of the enhanced section. Let $n \to \infty$; then we get the upper bound of the overall speedup (S), which is $1/(1-r)$. For example, if $r = 50\%$, no matter how big n is, the overall speedup S cannot be larger than 2. Figure 1.10 shows some examples of Amdahl's Law, from which we can see the upper bound of the speedup.

1.3.2 Trace-Driven Simulation and Execution-Driven Simulation

Trace-driven simulation is a method for estimating the performance of potential computer architectures by simulating their behavior in response to the instruction and data references contained in an input trace.

As shown in Figure 1.11, a real machine is used to execute a benchmark program and write the executed instruction information, such as the instruction address, instruction opcode, and data reference address, to a trace file. This trace is then fed into an architecture simulator for the performance study, such as cache performance, ILP, accuracy of branch prediction, TLB performance, and the utilization of FUs.

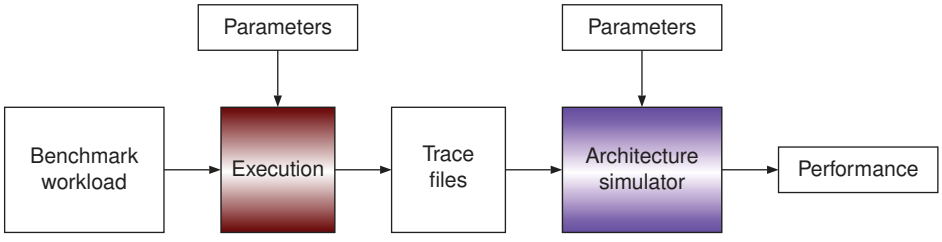

Figure 1.11 Trace-driven simulation

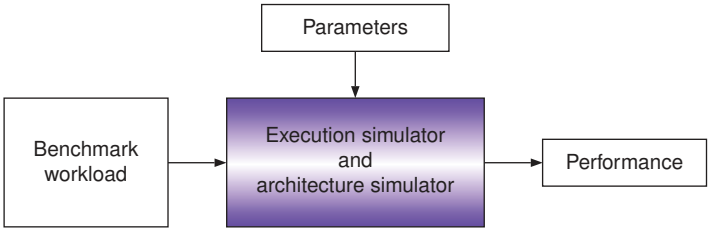

Figure 1.12 Execution-driven simulation

Through these simulations, we can find the performance bottleneck and change the architecture configurations to eliminate the bottleneck. The architecture simulator can run any machine as long as the trace format is known, but the trace files are very large in general for real benchmark programs. Execution-driven simulation is another method that does not require storing traces, as shown in Figure 1.12.

The execution-driven simulation requires the execution of the benchmark program: as the program is being executed, the performance study is also carried out at the same time. Execution-driven simulation must run the benchmark and simulator on the same machine and is much slower than trace-driven simulation. If a benchmark program will be used many times, it is better to generate the trace file once, and use the method of the trace-driven simulation.

1.3.3 High-Performance Computers and Interconnection Networks

A high-performance computer commonly means a computer system in which there are multiple CPUs or multiple computers. High-performance computers can be categorized into multiprocessors and multicomputers. In a multiprocessor system, there are multiple CPUs that communicate with each other through the shared memory. We also call it a parallel system. The shared memory can be centralized or distributed. Supercomputers are commonly parallel systems with the distributed shared memory (DSM). In a multicomputer system, there are multiple computers that communicate with each other via message-passing, over TCP/IP for example, and the memory in a computer is not shared by other computers. We call it a distributed system. Grid and cloud computer systems are usually distributed systems but in which there may be several supercomputers. Note that in some parallel systems, the distributed memories are shared by all CPUs by means of message-passing. An industry standard interface for this is the MPI (message-passing interface), which is installed in many popular parallel computing platforms.

Figure 1.13 illustrates the organization of a supercomputer in which the interconnection network provides the communication paths between CPUs and remote memories. A memory is said to be remote if

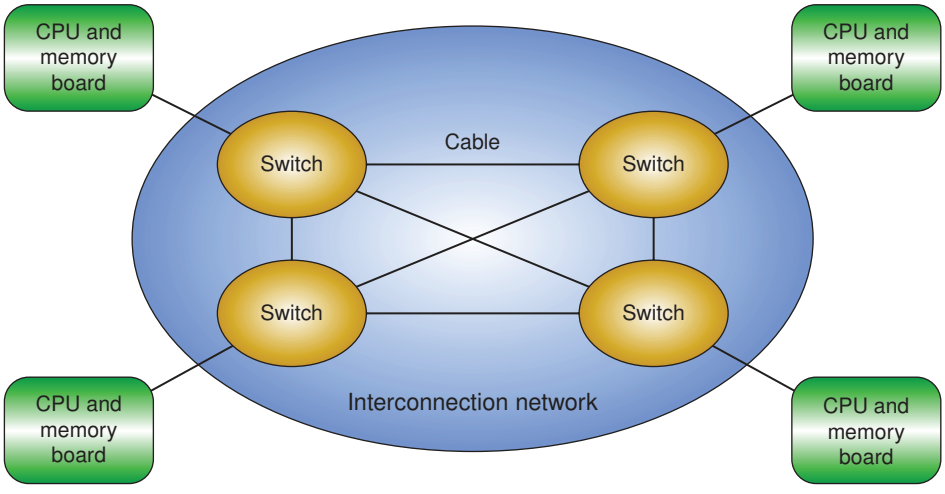

Figure 1.13 Supercomputer and interconnection network

the memory is not located on the same board with the CPU. If the memory is located on the same board with the CPU, we call it local memory. A CPU can access the local memory directly without disturbing the interconnection network. Access to the local memory is faster than access to the remote memory; we call this feature nonuniform memory access (NUMA).

The supercomputer shown in Figure 1.13 is a DSM parallel system. A DSM system can be very large. When we want to build a small parallel system, a server for instance, we can use the architecture of symmetric multiprocessors (SMPs). In an SMP system, there is a common bus connecting CPUs and memory modules. The memory accesses in an SMP have uniform memory access (UMA).

The www.top500.org supercomputer site announces the rank of supercomputers in the world in June and November every year. Currently, there are millions of CPU cores in top supercomputers. The number of cores in top supercomputers will increase year by year.

Amdahl's Law is also suitable for calculating the speedup of a program using a supercomputer. If the sequential fraction of a program is 1%, the theoretical speedup using a supercomputer cannot exceed 100 no matter how many cores are used, as plotted for $r = 99\%$ in Figure 1.10.

1.4 Hardware Description Languages

HDLs are languages used for the hardware design, just like C and Java are languages for software development. HDLs are easier to use than the schematic capture, especially for designing large-scale hardware circuits. The most popular HDLs are Verilog HDL and VHDL (very high speed integrated circuit HDL). Both are covered by IEEE standards. Other high-level HDLs include SystemVerilog and SystemC.

Below is the Verilog HDL code that implements a 4-bit counter. The file name is `time_counter_verilog`, the same as the module name; the extension name is `.v`. There are two input signals: `enable` and `clk` (clock). The output signal is a 4-bit `my_counter` of `reg` (register) type. If `enable` is a 1, the counter is increased at the rising edge of the `clk`. `always` is a keyword. `posedge` (positive edge) is also a keyword, standing for the rising edge. `4'h1` is a 4-bit constant 1, denoted with the hexadecimal number. The symbol of "<=" can be also a "=". You see that, even if you do not understand the low level of the circuit design, you can design hardware, just as you can develop software in C or Java although you do not know the assembly programming languages.

```
module time_counter_verilog (enable, clk, my_counter);  // a counter example
    input        enable;                                // input,  1 bit
    input        clk;                                   // input,  1 bit
    output [3:0] my_counter;                            // output, 4 bits
    reg    [3:0] my_counter = 0;                        // register type
    always @ (posedge clk) begin                        // positive edge
        if (enable)                                     // if (enable == 1)
            my_counter <= my_counter + 4'h1;            //     my_counter++
    end
endmodule
```

Figure 1.14 shows the simulation waveform of the counter using the ModelSim, a powerful HDL simulator. The output signal, `my_counter`, is denoted in hexadecimal numbers. Counting happens on the clock rising edges when `enable` is a 1.

The following is the VHDL code (`time_counter_vhdl.vhdl`) that does the same work above. The Verilog HDL is said to be like the programming language C and VHDL is like C++, but actually it is more like Ada.

```
LIBRARY IEEE;                                              - a counter example
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY time_counter_vhdl IS
  PORT (clk        : IN  STD_LOGIC;                        - input,  1 bit
        enable     : IN  STD_LOGIC;                        - input,  1 bit
        my_counter : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)     - output, 4 bits
  );
END time_counter_vhdl;
ARCHITECTURE a_counter OF time_counter_vhdl IS
  SIGNAL cnt : STD_LOGIC_VECTOR (3 DOWNTO 0) := "0000"; - initialize cnt
BEGIN
  my_counter <= cnt;                                       - assign to output
  PROCESS (clk) BEGIN
    IF (clk'EVENT AND clk = '1') THEN                      - positive edge
      IF (enable = '1') THEN                               - if (enable == 1)
        cnt <= cnt + '1';                                  -     cnt++
      END IF;
    END IF;
  END PROCESS;
END a_counter;
```

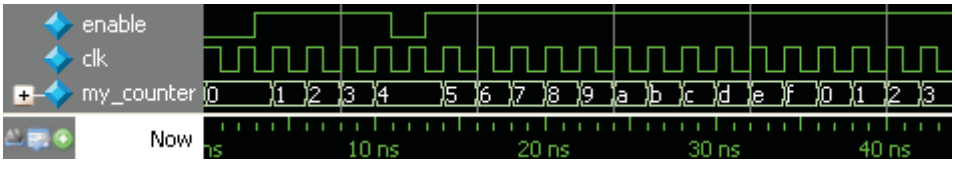

Figure 1.14 Waveform of time counter

Both examples use the high-level style to implement the counter. There are also low-level styles that we can use to design the circuits. This book uses only Verilog HDL for the designs of CPUs and I/O interfaces/buses.

Exercises

1.1 What are the main differences between RISC and CISC?

1.2 Investigate the meaning of each instruction of x86 and MIPS in the code examples of `mul16` given in this chapter.

1.3 Why is the microcode difficult to be pipelined?

1.4 Suppose that we have two machines: Machine A has a clock rate of 1 GHz and machine B has a clock rate of 2 GHz. We have made the measurements for these two machines as listed in the following table. Calculate the execution time and the MIPS (million instructions per second) of each machine.

Machine	Clock frequency	CPI				Executed instructions
		1	2	3	4	
A	1 GHz	50%	35%	10%	5%	20,200,000
B	2 GHz	10%	10%	30%	50%	22,000,000

1.5 Let $n = 10$ and $r = 75\%$. Calculate the overall speedup S by using Amdahl's Law and give the upper bound of S.

1.6 Let $n = 1{,}000{,}000$ and $S = 500{,}000$. Calculate the fractions r and $1 - r$ by using Amdahl's Law. What are the meanings of these numbers when we compare the performance of a supercomputer to that of a uniprocessor computer?