Notes for Digital Design and Computer Architecture ARM Edition

Creativity is just connecting things. (Steve Jobs)

CHAPTER 1: From Zero to One

1.1. THE GAME PLAN

1.2. THE ART OF MANAGING COMPLEXITY

One of the characteristics that separates an engineer or computer scientist from a layperson is a systematic approach to **managing complexity**. Modern digital systems are built from millions or billions of transistors. No human being could understand these systems by writing equations describing the movement of electrons in each transistor and solving all of the equations simultaneously. You will need to learn to manage complexity to understand how to build a microprocessor without getting mired in a morass of detail.

Abstraction

The critical technique for managing complexity is ***abstraction***: hiding details when they are not important. A system can be viewed from many different levels of abstraction.

Figure 1.1 illustrates levels of abstraction for an electronic computer system along with typical building blocks at each level.

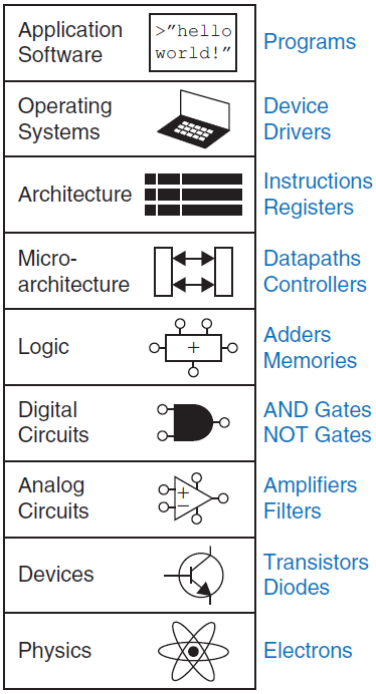


Figure 1. 1 Levels of abstraction for an electronic computing system

Microarchitecture links the logic and architecture levels of abstraction. The ***architecture*** level of abstraction describes a computer from the programmer's perspective.

A particular architecture can be implemented by one of many different microarchitectures with different price/performance/power trade-offs. For example, the Intel Core i7, the Intel 80486, and the AMD Athlon all implement the x86 architecture with **different** microarchitectures.

Moving into the software realm, the operating system handles low-level details such as accessing a hard drive or managing memory. Finally, the application software uses these facilities provided by the operating system to solve a problem for the user.

Discipline

Discipline is the act of intentionally restricting your design choices so that you can work more productively at a higher level of abstraction. Using interchangeable parts is a familiar application of discipline.

In the context of this book, the digital discipline will be very important. Digital circuits use discrete voltages, whereas analog circuits use continuous voltages. Therefore, digital circuits are a **subset** of analog circuits and in some sense must be capable of less than the broader class of analog circuits. However, digital circuits are much simpler to design.

The Three-Y's

In addition to abstraction and discipline, designers use the three "-y's" to manage complexity: *hierarchy, modularity, and regularity*. These principles apply to both software and hardware systems:

- *Hierarchy* involves dividing a system into modules, then further subdividing each of these modules until the pieces are easy to understand.
- *Modularity* states that the modules have **well-defined functions and interfaces**, so that they connect together easily without unanticipated side effects.
- *Regularity* seeks uniformity among the modules. Common modules are reused many times, reducing the number of distinct modules that must be designed.

CMOS (Complementary Metal Oxide Semiconductor Logic), TTL: Transistor-Transistor Logic.
Sign/Magnitude Numbers;

Finite state machines (FSMs): Moore machine and Mealy machine. Gray code.
Parallelism, latency, throughput.
Carry propagate adder (CPA): ripple-carry adders, carry-lookahead adders, and prefix adders.
Arithmetic/Logical Unit (ALU).
Random access memory (RAM): volatile, Read only memory (ROM): nonvolatile.
Programmable logic arrays (PLAs), Field programmable gate arrays (FPGAs)

1.3. THE DIGITAL ABSTRACTION

Unlike Babbage's machine, most electronic computers use a binary (two-valued) representation in which a high voltage indicates a '1' and a low voltage indicates a '0', because it is easier to distinguish between two voltages than ten.

The *amount of information* D in a discrete valued variable with N distinct states is measured in units of *bits* as

$$D = \log_2 N \text{ bits}$$

A binary variable conveys $\log_2 2 = 1$ bit of information.

This book focuses on digital circuits using binary variables: 1's and 0's. George Boole developed a system of logic operating on binary variables that is now known as *Boolean logic*. Each of Boole's variables could be TRUE or FALSE. Electronic computers commonly use a positive voltage to represent '1' and zero volts to represent '0'. In this book, we will use the terms '1', TRUE, and HIGH synonymously. Similarly, we will use '0', FALSE, and LOW interchangeably.

1.4. NUMBER SYSTEMS

You are accustomed to working with decimal numbers. In digital systems consisting of 1's and 0's, binary or hexadecimal numbers are often more convenient.

Decimal Numbers

Decimal numbers are referred to as *base* 10. The base is indicated by a subscript after the number to prevent confusion when working in more than one base. For example, Figure 1.2

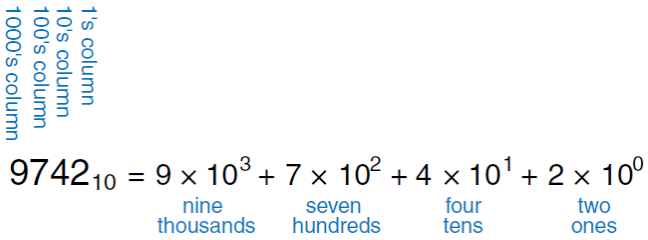


Figure 1. 2 Representation of a decimal number

An N-digit decimal number represents one of 10^N possibilities: $0, 1, 2, 3, \cdots, 10^N - 1$. This is called the *range* of the number. For example, a three-digit decimal number represents one of 1000 possibilities in the range of 0 to 999.

Binary Numbers

An N-bit binary number represents one of 2^N possibilities: $0, 1, 2, 3, \cdots, 2^N - 1$.

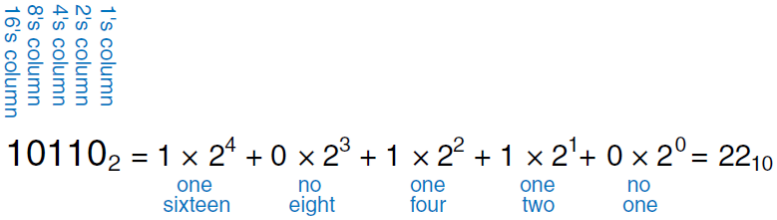


Figure 1. 3 Conversion of a binary number to decimal

Example 1.1: BINARY TO DECIMAL CONVERSION:

Convert the binary number 10110_2 to decimal. (Figure 1.3 shows the conversion)

1-Bit Binary Numbers	2-Bit Binary Numbers	3-Bit Binary Numbers	4-Bit Binary Numbers	Decimal Equivalents
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

Figure 1. 4 Binary numbers and their decimal equivalent

Example 1.2: DECIMAL TO BINARY CONVERSION:

Convert the decimal number 84_{10} to binary.

Solution: Determine whether each column of the binary result has a 1 or a 0. We can do this starting at either the left or the right column.

Working from the left, start with the largest power of 2 less than or equal to the number (in this case, 64). $84 \geq 64$, so there is a 1 in the 64's column, leaving $84 - 64 = 20$. $20 < 32$, so there is a 0 in the 32's column. $20 \geq 16$, so there is a 1 in the 16's column, leaving $20 -$

$16 = 4$. $4 < 8$, so there is a 0 in the 8's column. $4 \geq 4$, so there is a 1 in the 4's column, leaving $4 - 4 = 0$. Thus, there must be 0's in the 2's and 1's column. Putting this all together, $84_{10} = 1010100_2$.

Working from the right, repeatedly divide the number by 2. The remainder goes in each column. $84/2 = 42$, so 0 goes in the 1's column. $42/2 = 21$, so 0 goes in the 2's column. $21/2 = 10$ with a remainder of 1 going in the 4's column. $10/2 = 5$, so 0 goes in the 8's column. $5/2 = 2$ with a remainder of 1 going in the 16's column. $2/2 = 1$, so 0 goes in the 32's column. Finally, $1/2 = 0$ with a remainder of 1 going in the 64's column. Again, $84_{10} = 1010100_2$.

Hexadecimal Numbers

Writing long binary numbers becomes tedious and prone to error. A group of four bits represents one of $2^4 = 16$ possibilities. Hence, it is sometimes more convenient to work in base 16, called *hexadecimal*. Hexadecimal numbers use the digits 0 to 9 along with the letters A to F.

Example 1.3: HEXADECIMAL TO BINARY AND DECIMAL CONVERSION:

Convert the hexadecimal number $2ED_{16}$ to binary and to decimal.

Solution: Conversion between hexadecimal and binary is easy because each hexadecimal digit directly corresponds to four binary digits. $2_{16} = 0010_2$, $E_{16} = 1110_2$ and $D_{16} = 1101_2$ so $2ED_{16} = 001011101101_2$. Conversion to decimal requires the arithmetic shown in Figure 1.5.

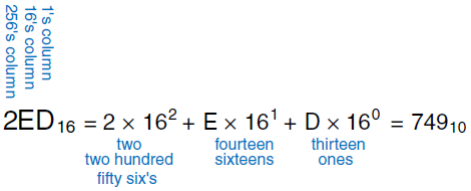


Figure 1. 5 Conversion of a hexadecimal number to decimal

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Figure 1. 6 Hexadecimal number system

Example 1.4: BINARY TO HEXADECIMAL CONVERSION:

Convert the binary number 1111010_2 to hexadecimal.

Solution: Again, conversion is easy. Start reading from the right. The four least significant bits are $1010_2 = A_{16}$. The next bits are $111_2 = 7_{16}$. Hence $1111010_2 = 7A_{16}$.

Example 1.5: DECIMAL TO HEXADECIMAL AND BINARY CONVERSION:

Convert the decimal number 333_{10} to hexadecimal and binary.

Solution: Like decimal to binary conversion, decimal to hexadecimal conversion can be done from the left or the right.

Working from the left, start with the largest power of 16 less than or equal to the number (in this case, 256). 256 goes into 333 once, so there is a 1 in the 256's column, leaving $333 - 256 = 77$. 16 goes into 77 four times, so there is a 4 in the 16's column, leaving $77 - 16 \times 4 = 13$. $13_{10} = D_{16}$, so there is a D in the 1's column. In summary, $333_{10} = 14D_{16}$. Now it is easy to convert from hexadecimal to binary, as in Example 1.3. $14D_{16} = 101001101_2$.

Working from the right, repeatedly divide the number by 16. The remainder goes in each column. $333/16 = 20$ with a remainder of $13_{10} = D_{16}$ going in the 1's column. $20/16 = 1$ with a remainder of 4 going in the 16's column. $1/16 = 0$ with a remainder of 1 going in the 256's column. Again, the result is $14D_{16}$.

Bytes, Nibbles, and All That Jazz

A group of **eight bits** is called a ***byte***. It represents one of $2^8 = 256$ possibilities.

A group of four bits, or half a byte, is called a *nibble*. It represents one of $2^4 = 16$ possibilities.

One hexadecimal digit stores one nibble and **two** hexadecimal digits store one full byte.

Microprocessors handle data **in chunks** called *words*. The size of a word depends on the architecture of the microprocessor. (A *microprocessor* is a processor built on **a single chip**)

Within a group of **bits**, the bit in the 1's column is called the *least significant bit* (*lsb*), and the bit at the other end is called the *most significant bit* (*msb*), as shown in Figure 1.7(a) for a 6-bit binary number. Similarly, within a **word**, the bytes are identified as *least significant byte* (*LSB*) through *most significant byte* (*MSB*), as shown in Figure 1.7(b) for a four-byte number written with eight hexadecimal digits.

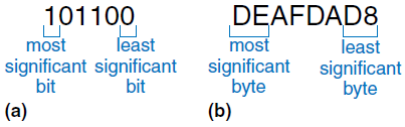


Figure 1. 7 Least and most significant bits and bytes

By handy coincidence, $2^{10} = 1024 \approx 10^3$. Hence, the term *kilo* (Greek for thousand) indicates 10^3. For example, 2^{10} bytes is one kilobyte (1 KB). Similarly, *mega* (million) indicates $2^{20} \approx 10^6$, and *giga* (billion) indicates $2^{30} \approx 10^9$.

1024 bytes is called a *kilobyte* (KB). 1024 bits is called a *kilobit* (Kb or Kbit). Similarly, MB, Mb, GB, and Gb are used for millions and billions of bytes and bits.

Binary Addition

Binary addition is much like decimal addition, but easier, as shown in Figure 1.8. As in decimal addition, if the sum of two numbers is greater than what fits in a single digit, we *carry* a 1 into the next column.

Figure 1. 8 Addition examples showing carries: (a) decimal (b) binary

For obvious reasons, the bit that is carried over to the neighboring column is called the *carry bit*. **Overflow** can be detected by checking for **a carry out** of the most significant column.

Digital systems usually operate on a fixed number of digits. Addition is said to *overflow* if the result is too big to fit in the available digits.

Signed Binary Numbers

So far, we have considered only *unsigned* binary numbers that represent positive quantities. We will often want to represent both positive and negative numbers, requiring a different binary number system. Several schemes exist to represent signed binary numbers; the **two most** widely employed are called sign/magnitude and two's complement.

Sign/Magnitude Numbers

An N-bit sign/magnitude number uses the most significant bit as the sign and the remaining $N-1$ bits as the magnitude (absolute value). A sign bit of 0 indicates positive and a sign bit of 1 indicates negative.

Unfortunately, ordinary binary addition **does not work** for sign/magnitude numbers. For example, using ordinary addition on $-5_{10}+5_{10}$ gives $1101_2+0101_2=10010_2$, which is nonsense.

An N-bit sign/magnitude number spans the range $[-2^{N-1}+1, 2^{N-1}-1]$. Sign/magnitude numbers are slightly odd in that both $+0$ and -0 exist. Both indicate zero. As you may expect, it can be **troublesome** to have two different representations for the same number.

Two's Complement Numbers

Two's complement numbers are identical to unsigned binary numbers except that the most significant bit position has a weight of -2^{N-1} instead of 2^{N-1}. They overcome the shortcomings of sign/magnitude numbers: zero has a single representation, and ordinary addition **works**.

In two's complement representation, zero is written as all zeros: $00\cdots000_2$. The most positive number has a 0 in the most significant position and 1's elsewhere: $01\cdots111_2=2^{N-1}-1$. The most negative number has a 1 in the most significant position and 0's elsewhere: $10\cdots000_2=-2^{N-1}$. And -1 is written as all ones: $11\cdots111_2$.

Notice that positive numbers have a 0 in the most significant position and negative numbers have a 1 in this position, so the most significant bit can be viewed as the sign bit. However, the overall number is interpreted differently for two's complement numbers and sign/magnitude numbers.

The sign of a two's complement number is reversed in a process called ***taking the two's complement***. The process consists of inverting all of the bits in the number, then adding 1 to the least significant bit position. This is useful to find the representation of a negative number or to determine the magnitude of a negative number.

Note: $-15_{10}=-01111_2=00000_2-01111_2(\text{or } 100000_2-01111_2)=10001_2=\overline{01111_2}+1$ (taking the two's complement). So $12_{10}-15_{10}=01100_2-01111_2=01100_2+10001_2$.

0	00000000	32	00100000	64	01000000	96	01100000	-128	10000000	-96	10100000	-64	11000000	-32	11100000
1	00000001	33	00100001	65	01000001	97	01100001	-127	10000001	-95	10100001	-63	11000001	-31	11100001
2	00000010	34	00100010	66	01000010	98	01100010	-126	10000010	-94	10100010	-62	11000010	-30	11100010
3	00000011	35	00100011	67	01000011	99	01100011	-125	10000011	-93	10100011	-61	11000011	-29	11100011
4	00000100	36	00100100	68	01000100	100	01100100	-124	10000100	-92	10100100	-60	11000100	-28	11100100
5	00000101	37	00100101	69	01000101	101	01100101	-123	10000101	-91	10100101	-59	11000101	-27	11100101
6	00000110	38	00100110	70	01000110	102	01100110	-122	10000110	-90	10100110	-58	11000110	-26	11100110
7	00000111	39	00100111	71	01000111	103	01100111	-121	10000111	-89	10100111	-57	11000111	-25	11100111
8	00001000	40	00101000	72	01001000	104	01101000	-120	10001000	-88	10101000	-56	11001000	-24	11101000
9	00001001	41	00101001	73	01001001	105	01101001	-119	10001001	-87	10101001	-55	11001001	-23	11101001
10	00001010	42	00101010	74	01001010	106	01101010	-118	10001010	-86	10101010	-54	11001010	-22	11101010
11	00001011	43	00101011	75	01001011	107	01101011	-117	10001011	-85	10101011	-53	11001011	-21	11101011
12	00001100	44	00101100	76	01001100	108	01101100	-116	10001100	-84	10101100	-52	11001100	-20	11101100
13	00001101	45	00101101	77	01001101	109	01101101	-115	10001101	-83	10101101	-51	11001101	-19	11101101
14	00001110	46	00101110	78	01001110	110	01101110	-114	10001110	-82	10101110	-50	11001110	-18	11101110
15	00001111	47	00101111	79	01001111	111	01101111	-113	10001111	-81	10101111	-49	11001111	-17	11101111
16	00010000	48	00110000	80	01010000	112	01110000	-112	10010000	-80	10110000	-48	11010000	-16	11110000
17	00010001	49	00110001	81	01010001	113	01110001	-111	10010001	-79	10110001	-47	11010001	-15	11110001
18	00010010	50	00110010	82	01010010	114	01110010	-110	10010010	-78	10110010	-46	11010010	-14	11110010
19	00010011	51	00110011	83	01010011	115	01110011	-109	10010011	-77	10110011	-45	11010011	-13	11110011
20	00010100	52	00110100	84	01010100	116	01110100	-108	10010100	-76	10110100	-44	11010100	-12	11110100
21	00010101	53	00110101	85	01010101	117	01110101	-107	10010101	-75	10110101	-43	11010101	-11	11110101
22	00010110	54	00110110	86	01010110	118	01110110	-106	10010110	-74	10110110	-42	11010110	-10	11110110
23	00010111	55	00110111	87	01010111	119	01110111	-105	10010111	-73	10110111	-41	11010111	-9	11110111
24	00011000	56	00111000	88	01011000	120	01111000	-104	10011000	-72	10111000	-40	11011000	-8	11111000
25	00011001	57	00111001	89	01011001	121	01111001	-103	10011001	-71	10111001	-39	11011001	-7	11111001
26	00011010	58	00111010	90	01011010	122	01111010	-102	10011010	-70	10111010	-38	11011010	-6	11111010
27	00011011	59	00111011	91	01011011	123	01111011	-101	10011011	-69	10111011	-37	11011011	-5	11111011
28	00011100	60	00111100	92	01011100	124	01111100	-100	10011100	-68	10111100	-36	11011100	-4	11111100
29	00011101	61	00111101	93	01011101	125	01111101	-99	10011101	-67	10111101	-35	11011101	-3	11111101
30	00011110	62	00111110	94	01011110	126	01111110	-98	10011110	-66	10111110	-34	11011110	-2	11111110
31	00011111	63	00111111	95	01011111	127	01111111	-97	10011111	-65	10111111	-33	11011111	-1	11111111

Example 1.10: TWO'S COMPLEMENT REPRESENTATION OF A NEGATIVE NUMBER:
Find the representation of -2_{10} as a 4-bit two's complement number.

Solution: Start with $+2_{10} = 0010_2$. To get -2_{10}, invert the bits and add 1. Inverting 0010_2 produces 1101_2. $1101_2 + 1 = 1110_2$. So -2_{10} is 1110_2.

Example 1.11: VALUE OF NEGATIVE TWO'S COMPLEMENT NUMBERS:
Find the decimal value of the two's complement number 1001_2.
Solution: 1001_2 has a leading 1, so it must be negative. To find its magnitude, invert the bits and add 1. Inverting $1001_2 = 0110_2$. $0110_2 + 1 = 0111_2 = 7_{10}$. Hence, $1001_2 = -7_{10}$.
Two's complement numbers have the compelling advantage that addition **works properly** for both positive and negative numbers. Recall that when adding N-bit numbers, the carry out of the Nth bit (i.e., the $N + 1^{\text{th}}$ result bit) is discarded.

Example 1.12: ADDING TWO'S COMPLEMENT NUMBERS:
Compute (a) $-2_{10} + 1_{10}$ and (b) $-7_{10} + 7_{10}$ using two's complement numbers.
Solution: (a) $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$. (b) $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$. The fifth bit is discarded, leaving the correct 4-bit result 0000_2.
Subtraction is performed by taking the two's complement of the second number, then adding.
Like unsigned numbers, N-bit two's complement numbers represent one of 2^N possible values. However, the values are split between positive and negative numbers. In general, the range of an N-bit two's complement number spans $[-2^{N-1}, 2^{N-1} - 1]$. The most negative number $10\cdots000_2 = -2^{N-1}$ is sometimes called the *weird number*.
Adding two N-bit positive numbers or negative numbers may cause overflow if the result is greater than $2^{N-1} - 1$ or less than -2^{N-1}. Adding a positive number to a negative number never causes overflow. **Unlike unsigned numbers**, a carry out of the most significant column does not indicate overflow (e.g., in 4-bit two's complement numbers: $-3 - 4 = 1101_2 + 1100_2 = 11001_2 = 1001_2 = -7$, or just $0 - 0 = 0000_2 + 10000_2 = 0000_2$). Instead, overflow occurs if the two numbers being added have the same sign bit and the result has the opposite sign bit.

When a two's complement number is **extended** to more bits, the sign bit must be copied into the most significant bit positions (or, "more" bits). This process is called *sign extension*. (e.g., 4-bit: $-3 = 1101_2$ → 5-bit: $-3 = 11101_2$ → 6-bit: $-3 = 111101_2$)

Comparison of Number Systems
The three most commonly used binary number systems are unsigned, two's complement, and sign/magnitude.

System	Range
Unsigned	$[0, 2^N - 1]$
Sign/Magnitude	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Two's Complement	$[-2^{N-1}, 2^{N-1} - 1]$

Figure 1. 9 Range of N-bit numbers

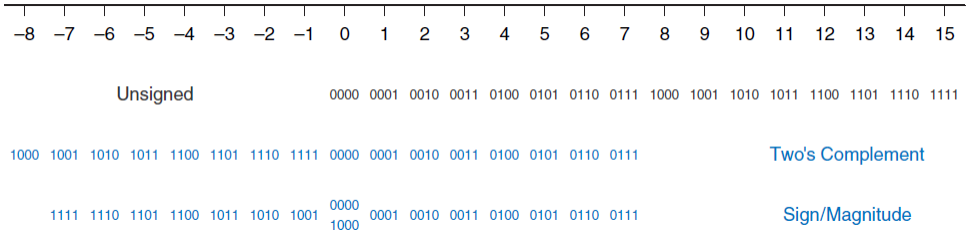


Figure 1.10 Number line and 4-bit binary encodings

1.5. LOGIC GATES
Now that we know how to use binary variables to represent information, we explore digital systems that perform operations on these binary variables. *Logic gates* are simple digital circuits

that take one or more binary inputs and produce a binary output.

The relationship between the inputs and the output can be **described** with a **truth table** or a **Boolean equation**. A ***truth table*** lists inputs on the left and the corresponding output on the right. It has one row for each possible combination of inputs. A ***Boolean equation*** is a mathematical expression using binary variables.

NOT Gate

A *NOT gate* has one input, A, and one output, Y, as shown in Figure 1.11. The line over A in the Boolean equation is pronounced NOT, so $Y = \bar{A}$ is read "Y equals NOT A." The NOT gate is also called an *inverter*.

Other texts use a variety of notations for NOT, including $Y = A'$, $Y = \neg A$, $Y = !A$ or $Y = \sim A$.

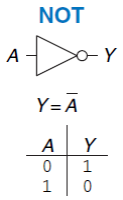


Figure 1. 11 NOT gate

Buffer

The other one-input logic gate is called a buffer and is shown in Figure 1.12. It simply copies the input to the output.

From the logical point of view, a buffer is no different from a wire, so it might seem useless. **However**, from the analog point of view, the buffer might have desirable characteristics such as the ability to deliver large amounts of current to a motor or the ability to quickly send its output to many gates.

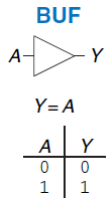


Figure 1. 12 Buffer

AND Gate

Two-input logic gates are more interesting. The AND gate shown in Figure 1.13. produces a TRUE output, Y, if and only if both A and B are TRUE. Otherwise, the output is FALSE. The Boolean equation for an AND gate can be written in several ways: $Y = A \bullet B$, $Y = AB$, or $Y = A \cap B$.

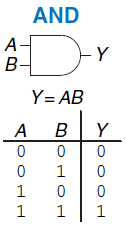


Figure 1. 13 AND gate

OR Gate

The OR gate shown in Figure 1.14 produces a TRUE output, Y, if either A or B (or both) are TRUE. The Boolean equation for an OR gate is written as $Y = A + B$ or $Y = A \cup B$.

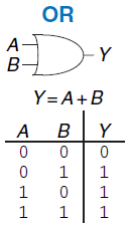


Figure 1. 14 OR gate

Other Two-Input Gates

Figure 1.15 shows other common two-input logic gates. XOR (exclusive OR, pronounced "ex-OR") is TRUE if A or B, but not both ($A \oplus B = A\bar{B} + \bar{A}B$), are TRUE. The XOR operation is indicated by $\oplus$, a plus sign with a circle around it. NAND gate performs NOT AND. Its output is TRUE unless both inputs are TRUE. The NOR gate performs NOT OR. Its output is TRUE if neither A nor B is TRUE. An N-input XOR gate is sometimes called a ***parity*** **gate** and produces a TRUE output if an **odd** number of inputs are TRUE.

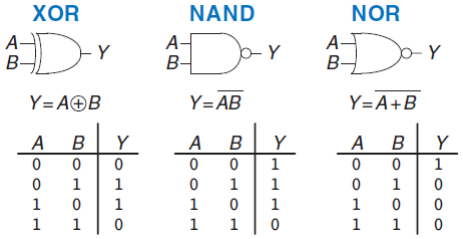


Figure 1. 15 More two-input logic gates

Example 1.15: XNOR GATE.

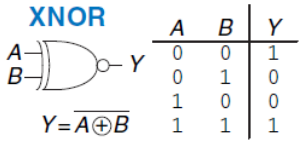


Figure 1. 16 XNOR gate

The XNOR output is TRUE if both inputs are FALSE or both inputs are TRUE. The two-input XNOR gate is sometimes called an ***equality*** **gate** because its output is TRUE when the inputs are equal. (Page 21)

Multiple-Input Gates

1.6. BENEATH THE DIGITAL ABSTRACTION

Supply Voltage

Suppose the lowest voltage in the system is 0 V, also called *ground* or GND. The highest voltage in the system comes from the power supply and is usually called V_{DD} (5/3.3/2.5/1.8/1.5/1.2 V).

The mapping of a continuous variable onto a discrete binary variable is done by defining *logic levels*, as shown below.

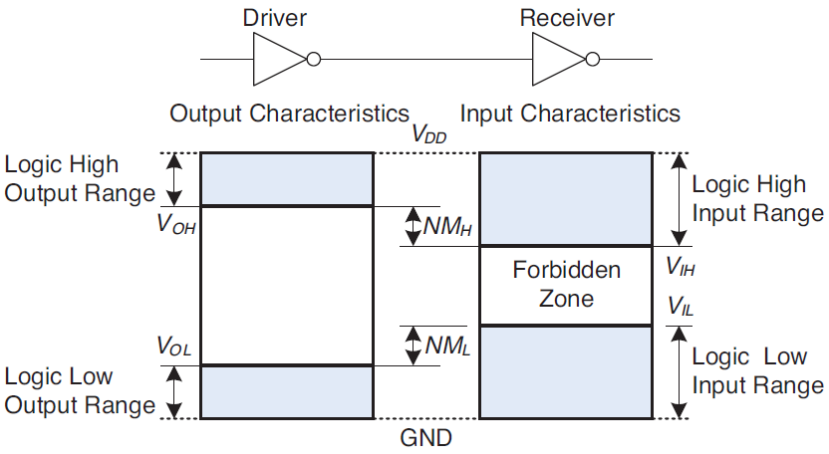


Figure 1. 17 Logic levels and noise margins

Noise Margins

The *noise margin* is the amount of noise that could be added to a worst-case output such that the signal can still be interpreted as a valid input. As can be seen in Figure 1.17, the low and high noise margins are, respectively

$$NM_L = V_{IL} - V_{OL}$$
$$NM_H = V_{OH} - V_{IH}$$

> V_{DD} stands for the voltage on the *drain* of a metal-oxide-semiconductor transistor, used to build most modern chips. The power supply voltage is also sometimes called V_{CC}, standing for the voltage on the collector of a bipolar junction transistor used to build chips in an older technology. Ground is sometimes called V_{SS} because it is the voltage on the *source* of a metal-oxide-semiconductor transistor.

DC Transfer Characteristics

The DC transfer characteristics of a gate describe the output voltage as a function of the input voltage when the input is changed slowly enough that the output can keep up. They are called transfer characteristics because they describe the relationship between input and output voltages.

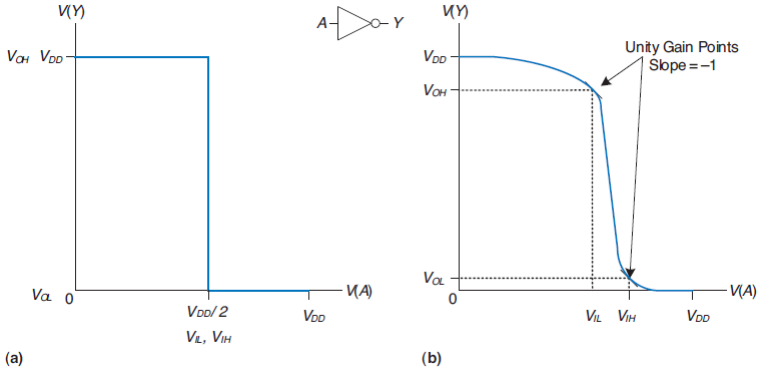


Figure 1.18 DC transfer characteristics and logic levels

> DC indicates behavior when an input voltage is held constant or changes slowly enough for the rest of the system to keep up. The term's historical root comes from *direct current*, a method of transmitting power across a line with a constant voltage. In contrast, the *transient response* of a circuit is the behavior when an input voltage changes rapidly.

The Static Discipline

Transistor-Transistor Logic (TTL), Complementary Metal Oxide Semiconductor Logic (CMOS, pronounced sea-moss), Low Voltage TTL Logic (LVTTL), and Low Voltage CMOS Logic (LVCMOS).

Logic Family	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4.75–5.25)	0.8	2.0	0.4	2.4
CMOS	5 (4.5–6)	1.35	3.15	0.33	3.84
LVTTL	3.3 (3–3.6)	0.8	2.0	0.4	2.4
LVCMOS	3.3 (3–3.6)	0.9	1.8	0.36	2.7

Figure 1. 19 Logic levels of 5 V and 3.3 V logic families

1.7. CMOS TRANSISTORS

Modern computers use transistors because they are cheap, small, and reliable. Transistors are electrically controlled switches that turn ON or OFF when a voltage or current is applied to a control terminal. The two main types of **transistors** are bipolar junction transistors and metal-oxide-semiconductor field effect transistors (MOSFETs or MOS transistors, pronounced "moss-fets" or "M-O-S", respectively).

Supplement materials

MOS transistor by combining

- Conductor (**M**etal)
- Insulator (**O**xide)
- Semiconductors

Semiconductors

MOS transistors are built from silicon, the predominant atom in rock and sand. Silicon (Si) is a group IV atom, so it has four electrons in its valence shell and forms bonds with four adjacent atoms, resulting in a crystalline lattice. *Figure 1.20*(a) shows the lattice in two dimensions for ease of drawing, but remember that **the lattice actually forms a cubic crystal**. In the figure, a line represents a covalent bond. By itself, silicon is a poor conductor because all the electrons are tied up in covalent bonds. However, **it becomes a better conductor when small amounts of impurities**, called **dopant atoms**, are carefully added. If a group V dopant such as arsenic (As) is added, the dopant atoms have an extra electron that is not involved in the bonds. The electron can easily move about the lattice, leaving an ionized dopant atom (As+) behind, as shown in *Figure 1.20*(b). The electron carries a negative charge, so we call arsenic an *n-type* dopant. On the other hand, if a group III dopant such as boron (B) is added, the dopant atoms are missing an electron, as shown in *Figure 1.20*(c). This missing electron is called a *hole*. An electron from a neighboring silicon atom may move over to fill the missing bond, forming an ionized dopant atom (B−) and leaving a hole at the neighboring silicon atom. In a similar fashion, the hole can migrate around the lattice. The hole is a lack of negative charge, so it acts like a positively charged particle. Hence, we call boron a *p-type dopant*. Because the conductivity of silicon changes over many orders of magnitude depending on the concentration of dopants, silicon is called a *semiconductor*.

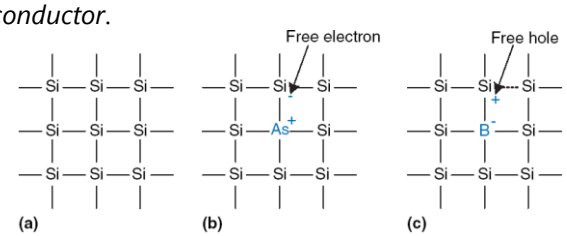


Figure 1.20 Silicon lattice and dopant atoms

Diodes

The junction between p-type and n-type silicon is called a *diode*. The p-type region is called the *anode* and the n-type region is called the *cathode*, as illustrated in *Figure 1.21*. When the voltage on the anode rises above the voltage on the cathode, the diode is forward biased, and current flows through the diode from the anode to the cathode. But when the anode voltage is lower than the voltage on the cathode, the diode is reverse biased, and **no current flows**. The diode symbol intuitively shows that current only flows in one direction.

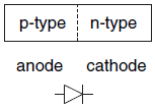


Figure 1.21 The p-n junction diode structure and symbol

Capacitors

Figure 1.22 Capacitor symbol

nMOS and pMOS Transistors

A MOSFET is a sandwich of several layers of conducting and insulating materials. MOSFETs are built on thin flat *wafers* of silicon of about 15 to 30 cm in diameter. The manufacturing process begins with a bare wafer. The process involves a sequence of steps in which dopants are implanted into the silicon, thin films of silicon dioxide and silicon are grown, and metal is deposited. Between each step, the wafer is *patterned* so that the materials appear only where they are desired. Because transistors are a fraction of a micron ($1\ \mu m = 1\ micron = 10^{-6}\ m$) in length and the entire wafer is processed at once, it is inexpensive to manufacture billions of transistors at a time. Once processing is complete, the wafer is cut into rectangles called chips or dice (die) that contain thousands, millions, or even billions of transistors. The chip is tested, then placed in a plastic or ceramic *package* with metal pins to connect it to a circuit board.

The MOSFET sandwich consists of a conducting layer called the gate on top of an insulating layer of silicon dioxide (SiO_2) on top of the silicon wafer, called the substrate. Historically, the gate was constructed from metal, hence the name metal-oxide-semiconductor. Modern manufacturing processes use polycrystalline silicon for the gate because it does not melt during subsequent high-temperature processing steps. Silicon dioxide is better known as glass and is often simply called oxide in the semiconductor industry. The metal-oxide-semiconductor sandwich forms a capacitor, in which a thin layer of insulating oxide called a dielectric separates the metal and semiconductor plates.

There are two flavors of **MOSFETs**: nMOS and pMOS (pronounced "n-moss" and "p-moss"). *Figure 1.23* shows cross-sections of each type, made by sawing through a wafer and looking at it from the side. The n-type transistors, called nMOS, have regions of n-type dopants adjacent to the gate called the source and the drain and are built on a p-type semiconductor substrate. The pMOS transistors are just the opposite, consisting of p-type source and drain regions in an n-type substrate.

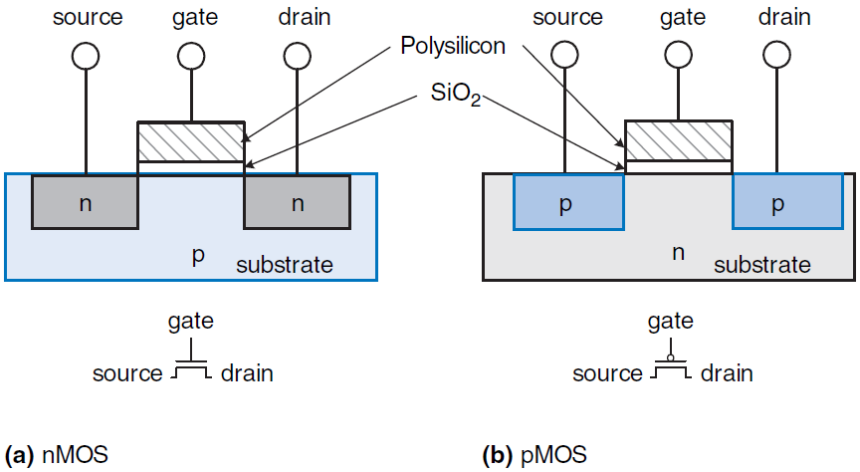


Figure 1.23 nMOS and pMOS transistors

A MOSFET behaves as a voltage-controlled switch in which the **gate voltage creates an electric field that turns ON or OFF a connection between the source and drain**. The term *field effect transistor* comes from this principle of operation. Let us start by exploring the operation of an nMOS transistor.

The substrate of an nMOS transistor is normally tied to GND, **the lowest voltage in the system**. First, consider the situation when the gate is also at 0 V, as shown in *Figure 1.24*(a). The diodes between the source or drain and the substrate are reverse biased because the source or drain voltage is nonnegative. Hence, there is no path for current to flow between the source and drain, so the transistor is OFF. Now, consider when the gate is raised to V_{DD}, as shown in *Figure 1.24* (b). When a positive voltage is applied to the top plate of a capacitor, it establishes an electric field that attracts positive charge on the top plate and negative charge to the bottom plate. If the voltage is sufficiently large, so much negative charge is attracted to the underside of the gate that the region *inverts* from p-type to effectively become n-type. This inverted region is called the channel. Now the transistor has a continuous path from the n-type source through the n-type channel to the n-type drain, so electrons can flow from source to drain. The transistor is ON. The gate voltage required to turn on a transistor is called the threshold voltage, V_t, and is typically 0.3 to 0.7 V.

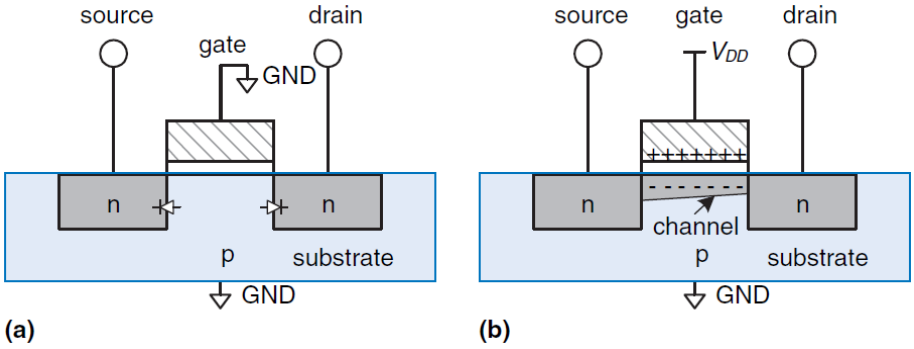


Figure 1.24 nMOS transistor operation

pMOS transistors are just the opposite fashion, as might be guessed from the bubble on their symbol shown in *Figure 1.25*. The substrate is tied to V_{DD}. When the gate is also at V_{DD}, the pMOS transistor is OFF. When the gate is at GND, the channel inverts to p-type and the pMOS transistor is ON.

Unfortunately, MOSFETs are not perfect switches. In particular, nMOS transistors pass 0's well

but pass 1's poorly. Specifically, when the gate of an nMOS transistor is at V_{DD}, the drain will only swing between 0 and $V_{DD} - V_t$. Similarly, pMOS transistors pass 1's well but 0's poorly. However, we will see that it is possible to build logic gates that use transistors only in their good mode.

nMOS transistors need a p-type substrate, and pMOS transistors need an n-type substrate. To build both flavors of transistors on the same chip, manufacturing processes typically start with a p-type wafer, then implant n-type regions called wells where the pMOS transistors should go. These processes that provide both flavors of transistors are called Complementary MOS or CMOS. CMOS processes are used to build the vast majority of all transistors fabricated today.

In summary, CMOS (Complementary MOS=nMOS+pMOS) processes give us two types of electrically controlled switches, as shown in *Figure 1.25*. The voltage at the gate (g) regulates the flow of current between the source (s) and drain (d). nMOS transistors are OFF when the gate is 0 and ON when the gate is 1. pMOS transistors are just the opposite: ON when the gate is 0 and OFF when the gate is 1.

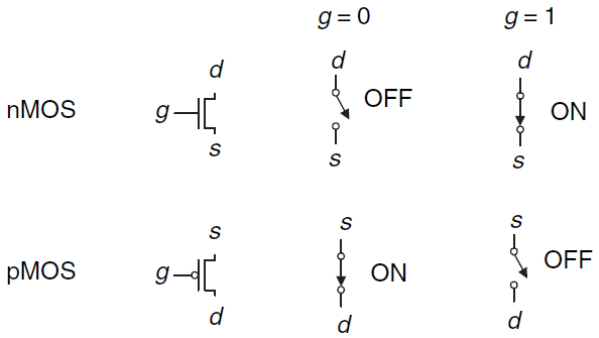


Figure 1.25 Switch models of MOSFETs

CMOS NOT Gate

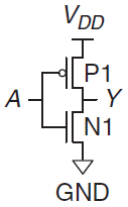


Figure 1.26 **NOT** gate schematic

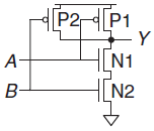


Figure 1.27 Two-input **NAND** gate schematic

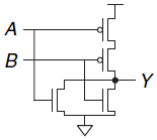


Figure 1.28 Two-input **NOR** gate schematic

Other CMOS Logic Gates

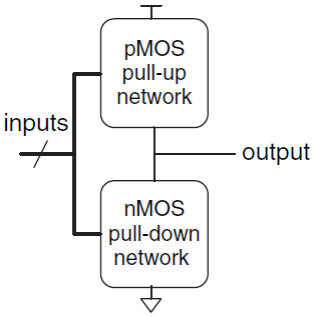


Figure 1.29 General form of an inverting logic gate

If both the pull-up and pull-down networks were ON simultaneously, a short circuit would exist between V_{DD} and GND. On the other hand, if both the pull-up and pull-down networks were OFF simultaneously, the output would be connected to neither V_{DD} nor GND. We say that the output floats.

Transmission Gates

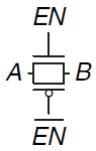


Figure 1.30 Transmission gate

Pseudo-nMOS Logic

Pseudo-nMOS logic replaces the slow stack of pMOS transistors with a single weak pMOS transistor that is always ON, as shown in Figure 1.30. This pMOS transistor is often called a *weak pull-up*. (Series connections are slower than parallel connections)

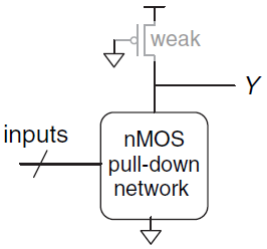


Figure 1.31 Generic pseudo-nMOS gate

1.8. POWER CONSUMPTION

Power consumption is the amount of energy used per unit time.

Example 1.23: POWER CONSUMPTION. (Page 34)

Exercises

Exercise 1.62: In a *biased* N-bit binary number system with bias B, positive and negative numbers are represented as their value plus the bias B. For example, for 5-bit numbers with a bias of 15, the number 0 is represented as 01111_2, 1 as 10000_2, and so forth. Biased number systems are sometimes used in floating point mathematics, which will be discussed in Chapter 5. Consider a biased 8-bit binary number system with a bias of 127_{10}. (not two's complement number)

(a) What decimal value does the binary number 10000010_2 represent?

$$10000010_2 - 01111111_2 = 00000011_2 = 3_{10}$$

(b) What binary number represents the value 0?

$$00000000_2 + 01111111_2 = 01111111_2$$

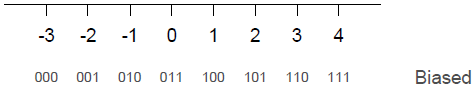
(c) What is the representation and value of the most negative number?

$$00000000_2 \text{ and } 00000000_2 - 01111111_2 = 1000001_2 = -127_{10}$$

(d) What is the representation and value of the most positive number?

$$11111111_2 \text{ and } 11111111_2 - 01111111_2 = 1000000_2 = 128_{10}$$

Exercise 1.63: Draw a number line for 3-bit biased numbers with a bias of 3.



Exercise 1.64: In a *binary coded decimal* (BCD) system, 4 bits are used to represent a decimal digit from 0 to 9. For example, 37_{10} is written as 00110111_{BCD}.

(a) Write 289_{10} in BCD

$$001010001001_{\text{BCD}}$$

(b) Convert $100101010001_{\text{BCD}}$ to decimal

$$951$$

(c) Convert 01101001_{BCD} to binary

$$01101001_{\text{BCD}} = 69_{10} = 1000101_2$$

(d) Explain why BCD might be a useful way to represent numbers

each 4-bit group represents one decimal digit, so conversion between binary and decimal is easy. BCD can also be used to represent decimal fractions exactly.

(e) Explain the disadvantages of BCD when compared to binary representations of numbers

Addition of BCD numbers doesn't work directly. Also, the representation doesn't maximize the amount of information that can be stored; for example, 2 BCD digits requires 8 bits and can store up to 100 values (0 – 99) - unsigned 8 bit binary can store 2^8 (256) values.

Exercise 1.73: A *majority gate* produces a TRUE output if and only if more than half of its inputs are TRUE. Complete a truth table for the three-input majority gate shown in Figure 1.30.

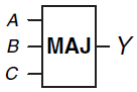


Figure 1. 32 Three-input majority gate

A	*B*	*C*	*Y*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

CHAPTER 2: Combinational Logic Design

2.1. INTRODUCTION

In digital electronics, a *circuit* is a network that processes discrete-valued variables. A circuit can be viewed as a black box, shown in Figure 2.1, with

- one or more discrete-valued input terminals
- one or more discrete-valued output terminals
- a functional specification describing the relationship between inputs and outputs
- a timing specification describing the delay between inputs changing and outputs responding.

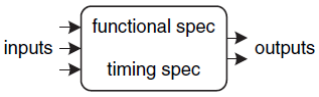
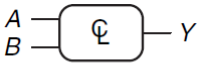


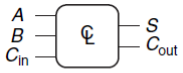
Figure 2. 1 Circuit as a black box with inputs, outputs, and specifications

Digital circuits are classified as *combinational* or *sequential*. A combinational circuit's outputs depend only on the current values of the inputs; in other words, it combines the current input values to compute the output. For example, a logic gate is a combinational circuit. A sequential circuit's outputs depend on both current and previous values of the inputs; in other words, it depends on the input sequence.



$$Y = F(A, B) = A + B$$

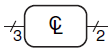
Figure 2. 2 Combinational logic circuit



$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Figure 2. 3 Multiple-output combinational circuit

To **simplify** drawings, we often use a single line with a slash through it and a number next to it to indicate a ***bus***, a bundle of multiple signals. The number specifies how many signals are in the bus.



(a)

(b)

Figure 2. 4 Slash notation for multiple signals

The rules of *combinational composition* tell us how we can build a large combinational circuit from smaller combinational circuit elements. A circuit is combinational if it consists of interconnected circuit elements such that

- Every circuit element is itself combinational.
- Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once.

Example 2.1: COMBINATIONAL CIRCUITS. (Page 57)

The functional specification of a combinational circuit is usually expressed as a truth table or a Boolean equation. In the next sections, we describe how to derive a Boolean equation from any truth table and how to use Boolean algebra and Karnaugh maps to simplify equations.

2.2. BOOLEAN EQUATIONS

Boolean equations deal with variables that are either TRUE or FALSE, so they are perfect for describing digital logic.

Terminology

The *complement* of a variable A is its inverse $\bar{A}$. The variable or its complement is called a *literal*. For example, A, $\bar{A}$, B, and $\bar{B}$ are literals.

The **AND** of one or more literals is called a *product* or an *implicant*. $\bar{A}B$, $A\bar{B}\bar{C}$, and B are all implicants for a function of three variables. A ***minterm*** is a product involving all of the inputs to the function. $A\bar{B}\bar{C}$ is a minterm for a function of the three variables A, B, and C. Similarly, the **OR** of one or more literals is called a *sum*. A ***maxterm*** is a sum involving all of the inputs to the function. $A + \bar{B} + C$ is a maxterm for a function of the three variables A, B, and C.

The ***order*** *of operations* is important when interpreting Boolean equations. In Boolean equations, NOT has the highest *precedence*, followed by AND, then OR. Just as in ordinary equations, products are performed before sums.

Sum-of-Products Form

A truth table of N inputs contains 2^N rows, one for each **possible** value of the inputs. Each row in a truth table is associated with **a minterm** that is TRUE for that row. Figure 2.5 shows a truth table of two inputs, A and B. The minterms are numbered starting with 0; the top row corresponds to minterm 0, m_0, the next row to minterm 1, m_1, and so on.

A	B	Y	minterm	minterm name
0	0	0	$\bar{A}\ \bar{B}$	m_0
0	1	1	$\bar{A}\ B$	m_1
1	0	0	$A\ \bar{B}$	m_2
1	1	0	$A\ B$	m_3

Figure 2. 5 Truth table and minterms

A	B	Y	minterm	minterm name
0	0	0	$\bar{A}\ \bar{B}$	m_0
0	1	1	$\bar{A}\ B$	m_1
1	0	0	$A\ \bar{B}$	m_2
1	1	1	$A\ B$	m_3

Figure 2. 6 Truth table with multiple TRUE minterms

We can write a Boolean equation for any truth table by summing each of the minterms for which the output, Y, is TRUE. Figure 2.6 shows $Y = \bar{A}B + AB$.

This is called the *sum-of-products canonical form* of a function because it is the sum (OR) of products (ANDs forming minterms).

The sum-of-products canonical form can **also** be written in sigma notation using the summation symbol, $\sum$. With this notation, the function from Figure 2.6 would be written as:

$$F(A,B) = \sum(m_1, m_3)$$

or

$$F(A,B) = \sum(1,3)$$

Unfortunately, sum-of-products form does not necessarily generate the simplest equation.

$$\begin{array}{ccc} 0 & \rightarrow & \bar{A} \\ 1 & \rightarrow & A \\ \hline \text{SOP Form} & & \end{array}$$

Product-of-Sums Form

An **alternative** way of expressing Boolean functions is the product-of-sums canonical form. Each row of a truth table corresponds to **a maxterm** that is FALSE for that row. We can write a Boolean equation for any circuit directly from the truth table as the AND of each of the maxterms for which the output is FALSE. The product-of-sums canonical form can **also** be written in pi notation using the product symbol, $\prod$. Figure 2.7 shows $Y = (A + B)(\bar{A} + B) = \prod(m_0, m_2)$. (Note: For A, B both are 0, we have $\bar{A}\bar{B}$. But here we focus on $Y = 0$, so $\overline{\bar{A}\bar{B}} = A + B$, and all of maxterms must be AND to be zero. Moreover, $Y = \bar{A}B + AB = (A + B)(\bar{A} + B) = \sum(m_1, m_3) = \prod(m_0, m_2)$.)

A	B	Y	maxterm	maxterm name
0	0	0	A + B	M_0
0	1	1	A + $\overline{B}$	M_1
1	0	0	$\overline{A}$ + B	M_2
1	1	1	$\overline{A}$ + $\overline{B}$	M_3

Figure 2. 7 Truth table with multiple FALSE maxterms

Sum-of-products produces a shorter equation when the output is TRUE on **only** a few rows of a truth table; **product-of-sums** is simpler when the output is FALSE on **only** a few rows of a truth table.

$$\begin{array}{rcl} 0 & \rightarrow & A \\ 1 & \rightarrow & \bar{A} \end{array}$$
POS Form

Example 1: How to get a POS form by using a SOP form, given the figure 2.6,

Solution:

By summing each of the minterms for which the output, Y, is False, like this

$$\bar{Y} = \bar{A}\bar{B} + A\bar{B}$$

Then

$$\bar{\bar{Y}} = Y = \overline{(\bar{A}\bar{B} + A\bar{B})} = (A + B)(\bar{A} + B)$$

Or just write,

$$Y = \overline{(\bar{A}\bar{B} + A\bar{B})} = (A + B)(\bar{A} + B)$$

So, the answer is same as shown in figure 2.7.

Complement materials

Useful Conversions:

1. Minterm to Maxterm conversion: rewrite minterm shorthand using maxterm shorthand replace minterm indices with the indices not already used

E.g., $\mathrm{F}(A,B,C) = \sum m(3,4,5,6,7) = \prod M(0,1,2)$

2. Maxterm to Minterm conversion: rewrite maxterm shorthand using minterm shorthand replace maxterm indices with the indices not already used

E.g., $\mathrm{F}(A,B,C) = \prod M(0,1,2) = \sum m(3,4,5,6,7)$

2.3. BOOLEAN ALGEBRA

Axioms and theorems of Boolean algebra obey the principle of duality. If the symbols 0 and 1 and the operators $\bullet$ (AND) and $+$ (OR) are **interchanged**, the statement will **still** be **correct**. We use the prime symbol (′) to denote the ***dual*** of a statement.

Axioms

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

Figure 2. 8 Axioms of Boolean algebra

Theorems of One Variable

	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

Figure 2. 9 Boolean theorems of one variable

Theorems of Several Variables

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D) = B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D) = (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots} = (\overline{B}_0 + \overline{B}_1 + \overline{B}_2 \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots} = (\overline{B}_0 \bullet \overline{B}_1 \bullet \overline{B}_2 \ldots)$	De Morgan's Theorem

Figure 2. 10 Boolean theorems of several variables

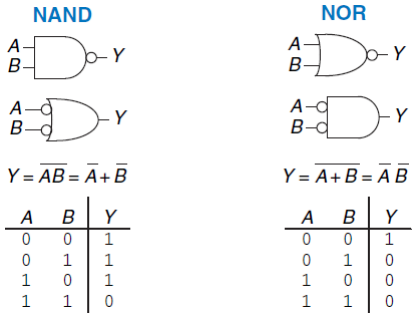


Figure 2. 11 De Morgan equivalent gates

Example 2.4: DERIVE THE PRODUCT-OF-SUMS FORM. (Page 64)

The Truth Behind It All

Simplifying Equations

How far can an equation be simplified? We define an equation in sum-of-products form to be *minimized* if it uses the fewest possible implicants. If there are several equations with the same number of implicants, the minimal one is the one with the fewest literals.

An implicant is called a *prime implicant* if it cannot be combined with any other implicants in the equation to form a new implicant with fewer literals.

Step	Equation	Justification
	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	
1	$\overline{B}\,\overline{C}(\overline{A} + A) + A\overline{B}C$	T8: Distributivity
2	$\overline{B}\,\overline{C}(1) + A\overline{B}C$	T5: Complements
3	$\overline{B}\,\overline{C} + A\overline{B}C$	T1: Identity

Figure 2. 12 Equation minimization

Step	Equation	Justification
	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	
1	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$	T3: Idempotency
2	$\overline{B}\,\overline{C}(\overline{A} + A) + A\overline{B}(\overline{C} + C)$	T8: Distributivity
3	$\overline{B}\,\overline{C}(1) + A\overline{B}(1)$	T5: Complements
4	$\overline{B}\,\overline{C} + A\overline{B}$	T1: Identity

Figure 2. 13 Improved equation minimization

You may have noticed that completely simplifying a Boolean equation with the theorems of Boolean algebra can take some trial and error. Section 2.7 describes a methodical technique called Karnaugh maps that makes the process easier.

2.4. FROM LOGIC TO GATES

A *schematic* is a diagram of a digital circuit showing the elements and the wires that connect them together. For example, the schematic in Figure 2.14 shows a logic function,

$$Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$$

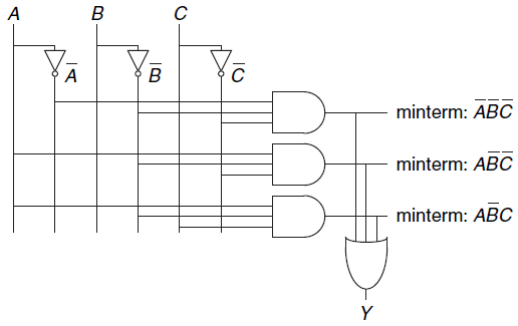


Figure 2. 14 Schematic of $Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$

By drawing schematics in a consistent fashion, we make them easier to read and debug. We will generally obey the following guidelines:

- Inputs are on the left (or top) side of a schematic.
- Outputs are on the right (or bottom) side of a schematic.
- Whenever possible, gates should flow from left to right.
- Straight wires are better to use than wires with multiple corners (jagged wires waste mental effort following the wire rather than thinking of what the circuit does).
- Wires always connect at a T junction.
- A dot where wires cross indicates a connection between the wires.
- Wires crossing *without* a dot make no connection.

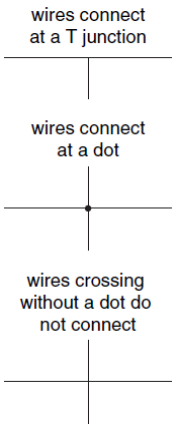
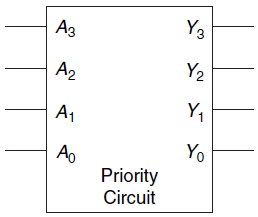


Figure 2. 15 Wire connections

Any Boolean equation in sum-of-products form can be drawn as a schematic in a systematic way similar to the above schematic. First, draw columns for the inputs. Place inverters in adjacent columns to provide the complementary inputs if necessary. Draw rows of AND gates for each of the minterms. Then, for each output, draw an OR gate connected to the minterms related to that output. This **style** is called a *programmable logic array (PLA)* because the inverters, AND gates, and OR gates are arrayed in a systematic fashion.

Example 2.7: MULTIPLE-OUTPUT CIRCUITS. (Page 68)



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Figure 2. 16 Priority circuit

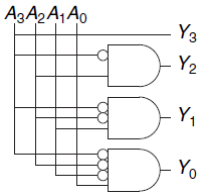


Figure 2. 17 Priority circuit schematic

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

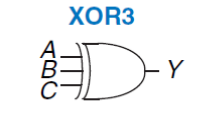
Figure 2. 18 Priority circuit truth table with don't cares (X's)

Notice that if A_3 is asserted in the priority circuit, the outputs don't care what the other inputs are. We use the symbol X to describe inputs that the output doesn't care about. Figure 2.18 shows that the four-input *priority circuit* truth table becomes much smaller with don't cares.

2.5. MULTILEVEL COMBINATIONAL LOGIC

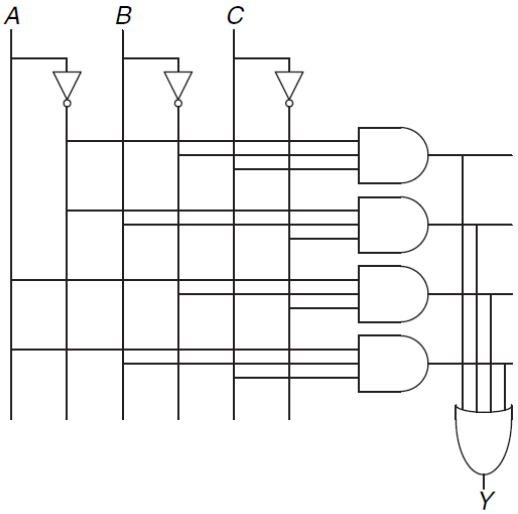
Logic in sum-of-products form is called *two-level logic* because it consists of literals connected to a level of AND gates connected to a level of OR gates. Designers often build circuits with more than two levels of logic gates. These multilevel combinational circuits may use less hardware than their two-level counterparts.

Hardware Reduction



A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(a)



(b)

Figure 2. 19 Three-input XOR: (a) functional specification and (b) two-level logic implementation

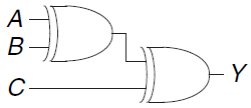


Figure 2. 20 Three-input XOR using two-input XORs

Bubble Pushing

You may recall from Section 1.7.6 that CMOS circuits **prefer** NANDs and NORs over ANDs and ORs. **But** reading the equation by inspection from a multilevel circuit with NANDs and NORs can get pretty hairy. Bubble pushing is a helpful way to redraw these circuits so that the bubbles cancel out and the function can be more easily determined. Building on the principles from Section 2.3.3, the **guidelines** for bubble pushing are as follows: (De Morgan's Theorem)

- **Begin** at the output of the circuit and work toward the inputs.
- Push any bubbles on the final output back toward the inputs so that you can read an equation in terms of the output (for example, Y) instead of the complement of the output ($\bar{Y}$).
- Working backward, draw each gate in a form so that bubbles cancel If the current gate has an input bubble, draw the preceding gate with an output bubble. If the current gate does not have an input bubble, draw the preceding gate without an output bubble.

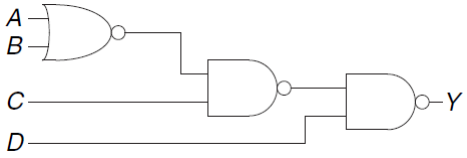
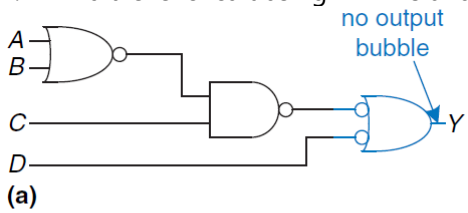
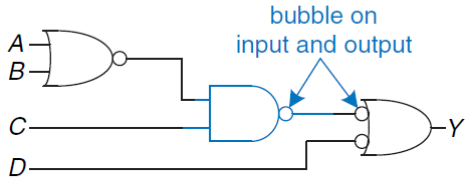


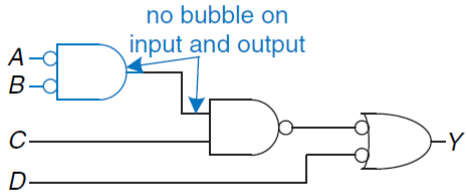
Figure 2. 21 Multilevel circuit using NANDs and NORs



(a)



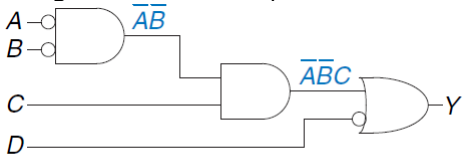
(b)



$Y = \bar{A}\bar{B}C + \bar{D}$

(c)

Figure 2. 22 Bubble-pushed circuit



$Y = \bar{A}\bar{B}C + \bar{D}$

Figure 2. 23 Logically equivalent bubble-pushed circuit

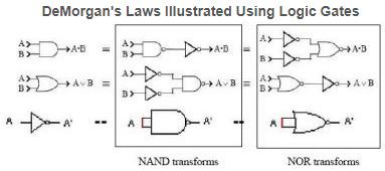
Example 2.8: BUBBLE PUSHING FOR CMOS LOGIC. (Page 73)

Supplement materials

NAND's and NOR's are the most common basic logic circuit element in use because they are simpler to build than AND and OR gates, and because each is **logically complete**.

OR to NAND	AND to NOR
$a \vee b \equiv \neg(\neg a \wedge \neg b)$	$a \wedge b \equiv \neg(\neg a \vee \neg b)$

DeMorgan's Law using Boolean Algebra



logically complete: A set of circuit gates or logical elements is logically complete if **any** boolean function representable by a truth table can be realized using **only** gates or elements from that set. (AND, OR, and NOT is a logically complete set. **NAND** is logically complete. **NOR** is logically complete.)

2.6. X'S AND Z'S, OH MY

Boolean algebra is limited to 0's and 1's. However, real circuits can also have illegal and floating values, represented symbolically by X and Z.

Illegal Value: X

The symbol X indicates that the circuit node has an *unknown* or *illegal* value. This commonly happens if it is being driven to both 0 and 1 at the same time. Figure 2.24 shows a case where node Y is driven both HIGH and LOW. This situation, called *contention*, is considered to be an error and must be avoided. The actual voltage on a node with contention may be somewhere between 0 and V_{DD}, depending on the relative strengths of the gates driving HIGH and LOW.

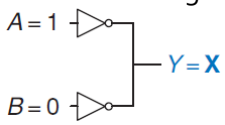


Figure 2. 24 Circuit with contention

X values are also sometimes used by circuit simulators to indicate an uninitialized value.

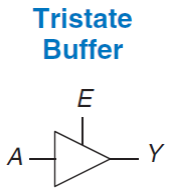
Be sure not to mix up the two meanings. When X appears in a truth table, it indicates that the value of the variable in the truth table is unimportant (can be either 0 or 1). When X appears in a circuit, it means that the circuit node has an unknown or illegal value.

Floating Value: Z

The symbol Z indicates that a node is being driven neither HIGH nor LOW. The node is said to be *floating*, *high impedance*, or *high* Z. A typical misconception is that a floating or undriven node is the same as a logic 0. In reality, a floating node might be 0, might be 1, or might be at some voltage in between, depending on the history of the system.

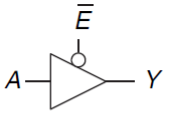
One common way to produce a floating node is to forget to connect a voltage to a circuit input, or to assume that an unconnected input is the same as an input with the value of 0. This mistake may cause the circuit to behave erratically as the floating input randomly changes from 0 to 1. Indeed, touching the circuit may be enough to trigger the change by means of static electricity from the body.

The *tristate* buffer, shown in Figure 2.25, has three possible output states: HIGH (1), LOW (0), and floating (Z). The tristate buffer has an input A, output Y, and enable E.



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2. 25 Tristate buffer



$\bar{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Figure 2. 26 Tristate buffer with active low enable

We often indicate an active low input by drawing a bar over its name, $\bar{E}$, or appending the letters "b" or "bar" after its name, *Eb* or *Ebar*.

Tristate buffers are commonly used on *busses* that connect multiple chips.

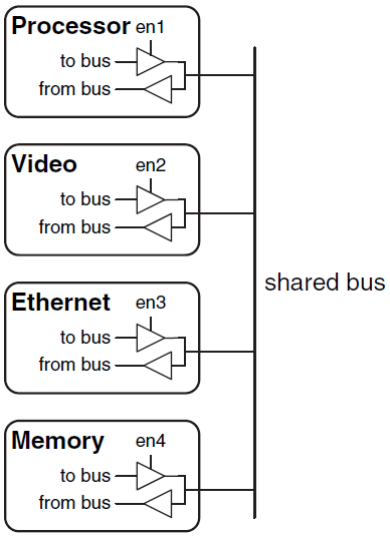


Figure 2. 27 Tristate bus connecting multiple chips

2.7. KARNAUGH MAPS

Karnaugh maps (*K-maps*) are a graphical method for simplifying Boolean equations. K-maps work well for problems with up to **four variables**.

Recall that logic minimization involves combining terms. Two terms containing an implicant P and the true and complementary forms of some variable A are combined to eliminate A: $PA + P\bar{A} = P$. Karnaugh maps make these combinable terms **easy to see** by putting them **next** to each other in a grid.

Figure 2.28 shows the truth table and K-map for a three-input function. Each square in the K-map corresponds to a row in the truth table and contains the value of the output Y for that row.

Each **square**, or **minterm**, differs from an adjacent square by a **change** in a **single** variable. You may have noticed that the A and B combinations in the top row are in a peculiar order: 00,

01, 11, 10. This order is called a *Gray code* (They are especially useful in mechanical encoders because a slight misalignment causes an error in only one bit).

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

(a)

Y, C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

(b)

Y, C \ AB	00	01	11	10
0	$\bar{A}\bar{B}\bar{C}$	$\bar{A}B\bar{C}$	$AB\bar{C}$	$A\bar{B}\bar{C}$
1	$\bar{A}\bar{B}C$	$\bar{A}BC$	ABC	$A\bar{B}C$

(c)

Figure 2. 28 Three-input function: (a) truth table, (b) K-map, (c) K-map showing minterms

Circular Thinking

Y, C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

Figure 2. 29 K-map minimization

Logic Minimization with K-Maps

K-maps provide an easy visual way to minimize logic. Simply circle all the rectangular blocks of 1's in the map, using the fewest possible number of circles. Each circle should be as **large** as possible. Then read off the implicants that were circled.

Rules for finding a minimized equation from a K-map are as follows:

- Use the fewest circles necessary to cover all the 1's.
- All the squares in each circle must contain 1's.
- Each circle must span a rectangular block that is a power of 2 (i.e., 1, 2, or 4) squares in each direction.
- Each circle should be as large as possible.
- A circle **may wrap** around the edges of the K-map.
- A 1 in a K-map may be circled **multiple times** if doing so allows **fewer** circles to be used.

Example 2.9: MINIMIZATION OF A THREE-VARIABLE FUNCTION USING A K-MAP. (Page 78)

Example 2.10: SEVEN-SEGMENT DISPLAY DECODER. (Page 79)

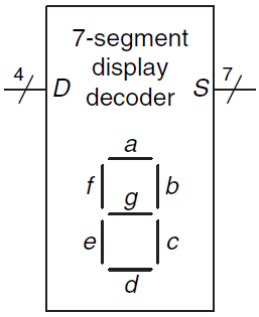


Figure 2. 30 Seven-segment display decoder icon

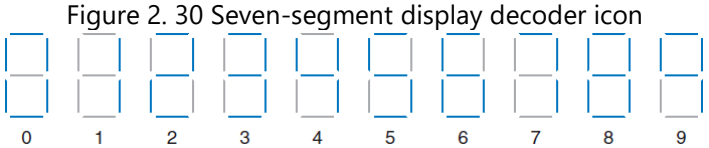


Figure 2. 31 Seven-segment display digits

$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0

Figure 2. 32 Seven-segment display decoder truth table

Don't Cares

Don't cares **also** appear in truth table outputs where the output value is **unimportant** or the corresponding input combination can **never** happen. Such outputs can be treated as either 0's or 1's at the designer's discretion.

In a K-map, X's **allow** for even more logic minimization. They can be circled if they help cover the 1's with fewer or larger circles, but they do not have to be circled if they are **not** helpful.

Example 2.11: SEVEN-SEGMENT DISPLAY DECODER WITH DON'T CARES. (Page 82)

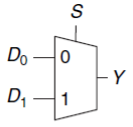
2.8. COMBINATIONAL BUILDING BLOCKS

Combinational logic is often grouped into larger building blocks to build more complex systems.

Multiplexers

Multiplexers are among the **most commonly** used combinational circuits. They choose an output from among several possible inputs based on the value of a select signal. A multiplexer is sometimes affectionately called a *mux*.

2:1 Multiplexer



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2. 33 2:1 multiplexer symbol and truth table

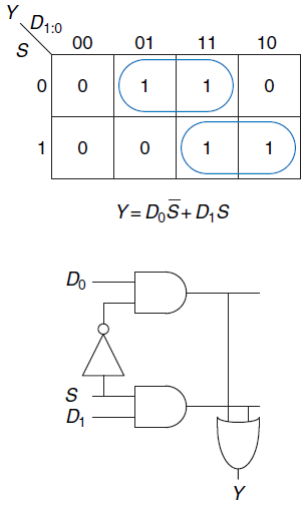


Figure 2. 34 2:1 multiplexer implementation using two-level logic

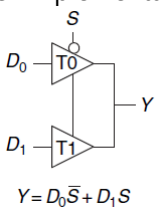


Figure 2. 35 Multiplexer using tristate buffers

Wider Multiplexers

Wider multiplexers, such as $8:1$ and $16:1$ multiplexers, can be built by expanding the methods shown in Figure 2.36. **In general**, an $N:1$ multiplexer needs $\log_2 N$ select lines.

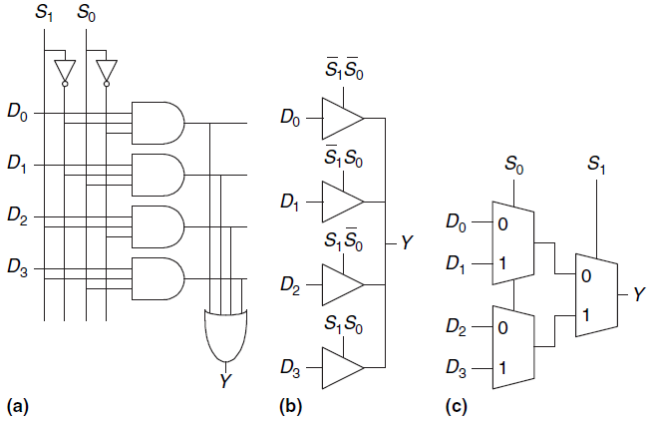


Figure 2. 36 $4:1$ multiplexer implementations: (a) two-level logic, (b) tristates, (c) hierarchical

Multiplexer Logic

Multiplexers can be used as *lookup tables* to perform logic functions. Figure 2.37 shows a $4:1$ multiplexer used to implement a two-input AND gate. **In general**, a 2^N input multiplexer can be programmed to perform **any** N-input logic function by applying 0's and 1's to the appropriate data inputs.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$

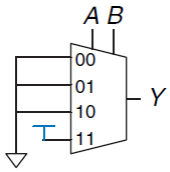


Figure 2. 37 4:1 multiplexer implementation of two-input AND function

With a little cleverness, we can **cut** the multiplexer size in half, using **only** a 2^{N-1} input multiplexer to perform any N-input logic function. The **strategy** is to provide one of the literals, as well as 0's and 1's, to the multiplexer data inputs.

To illustrate this principle, Figure 2.38 shows two-input AND and XOR functions implemented with $2:1$ multiplexers.

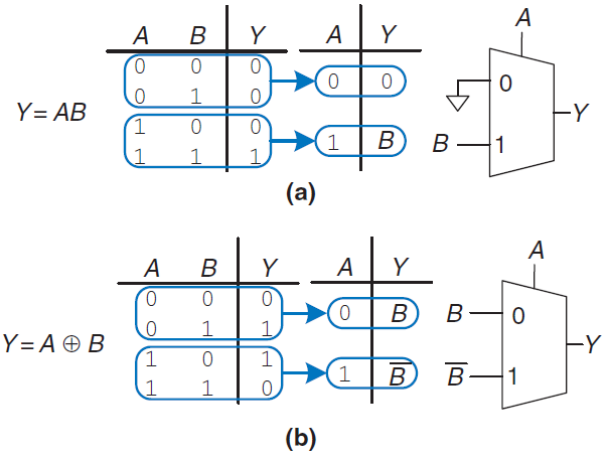
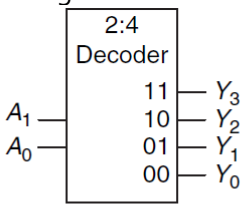


Figure 2. 38 Multiplexer logic using variable inputs

Decoders

A decoder has N inputs and 2^N outputs. It asserts exactly one of its outputs depending on the input combination. Figure 2.39 shows a $2:4$ decoder. The outputs are called *one-hot*, because exactly one is "hot" (HIGH) at a given time.



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Figure 2. 39 $2:4$ decoder

Example 2.14: DECODER IMPLEMENTATION.

In general, an $N:2^N$ decoder can be constructed from 2^N N-input **AND** gates that accept the various combinations of true or complementary inputs. (Page 87)

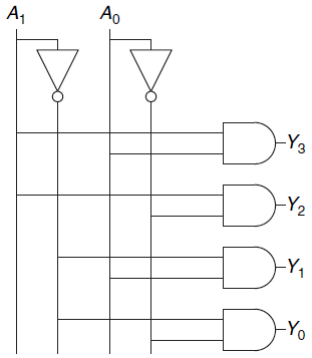


Figure 2. 40 2:4 decoder implementation

Decoder Logic

Decoders can be combined with **OR** gates to build logic functions. Figure 2.41 shows the two-input XNOR function using a $2:4$ decoder and a single OR gate.

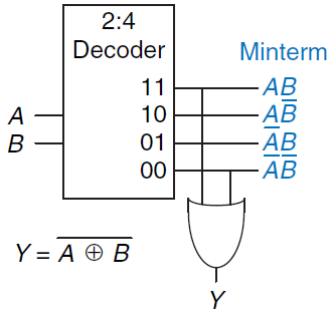


Figure 2. 41 Logic function using decoder

An N-input function with M 1's in the truth table can be built with an $N:2^N$ decoder and an M-input OR gate attached to all of the minterms containing 1's in the truth table.

2.9. TIMING

One of the most challenging issues in circuit design is ***timing***: making a circuit run **fast**.

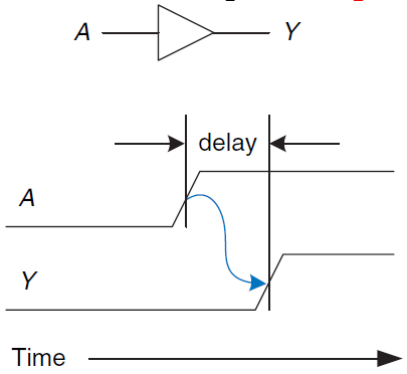


Figure 2. 42 Circuit delay

Propagation and Contamination Delay

Combinational logic is characterized by its ***propagation*** *delay* and ***contamination*** *delay*. The propagation delay t_{pd} is the **maximum** time from when an input changes until the output or outputs **reach** their final value. The contamination delay t_{cd} is the **minimum** time from when an input changes until any output **starts** to change its value.

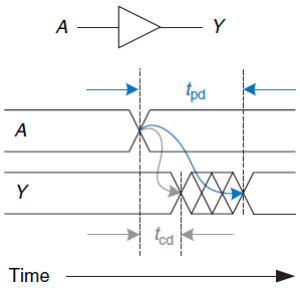


Figure 2. 43 Propagation and contamination delay

In Figure 2.43 the arcs indicate that Y may start to change t_{cd} after A transitions and that Y definitely settles to its new value within t_{pd}.

The underlying **causes** of **delay** in circuits include the time required to charge the capacitance in a circuit and the speed of light. t_{pd} and t_{cd} may be **different** for many reasons, including

- different rising and falling delays
- multiple inputs and outputs, some of which are faster than others
- circuits slowing down when hot and speeding up when cold

Circuit delays are ordinarily on the order of picoseconds (1 ps $= 10^{-12}$ seconds) to nanoseconds (1 ns $= 10^{-9}$ seconds).

Along with the factors already listed, propagation and contamination delays are also determined by the *path* a signal takes from input to output. Figure 2.44 shows a four-input logic circuit. The ***critical path***, is the **longest**, and therefore the **slowest**, path. The ***short path*** is the shortest, and therefore the fastest.

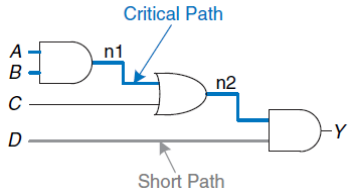


Figure 2. 44 Short path and critical path

The **propagation** delay of a combinational circuit is the sum of the propagation delays through each element on the **critical path**. The **contamination** delay is the sum of the contamination delays through each element on the **short path**. These delays are illustrated in Figure 2.45 and are described by the following equations:

$$t_{pd} = 2t_{pd\_\mathrm{AND}} + t_{pd\_\mathrm{OR}}$$

$$t_{cd} = t_{cd\ \mathrm{AND}}$$

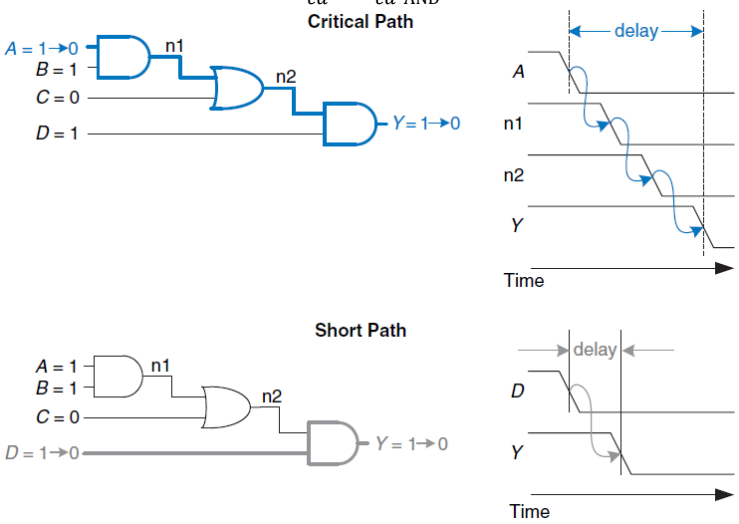


Figure 2. 45 Critical and short path waveforms

Although we are ignoring wire delay in this analysis, digital circuits are now so fast that the delay of long wires can be as important as the delay of the gates. The speed of light delay in

wires is covered in Appendix A.
Example 2.15: FINDING DELAYS. (Page 90)
Example 2.16: MULTIPLEXER TIMING: CONTROL-CRITICAL VS. DATA-CRITICAL. (Page 91)
Glitches
So far, we have discussed the case where a single input transition causes a single output transition. However, it is possible that a single input transition can cause *multiple* output transitions. These are called ***glitches*** or ***hazards***. Although glitches usually don't cause problems, it is **important** to realize that they exist and recognize them when looking at timing diagrams.

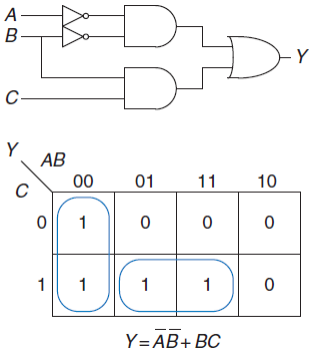


Figure 2. 46 Circuit with a glitch

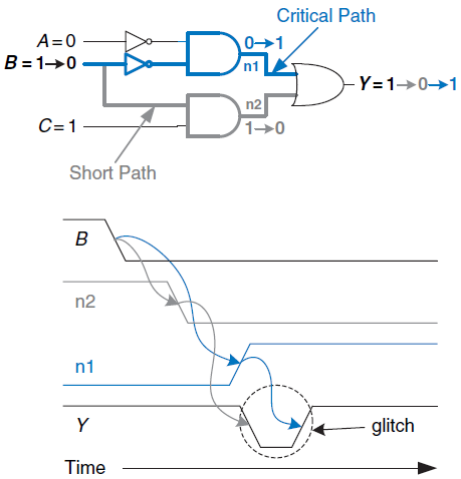


Figure 2. 47 Timing of a glitch

As long as we wait for the propagation delay to elapse before we depend on the output, glitches are **not** a problem, because the output **eventually** settles to the right answer.
If we choose to, we **can** avoid this glitch by adding another gate to the implementation. This is **easiest** to understand in terms of the K-map. Figure 2.48 shows the transition across the boundary of two prime implicants in the K-map indicates a **possible** glitch.
As we saw from the timing diagram in Figure 2.47, if the circuitry implementing one of the prime implicants turns *off* before the circuitry of the other prime implicant can turn *on*, there is a glitch. To fix this, we add another circle that *covers* that prime implicant boundary, as shown in Figure 2.49. You might recognize this as the consensus theorem, where the added term, $\bar{A}C$, is the **consensus** or **redundant** term.

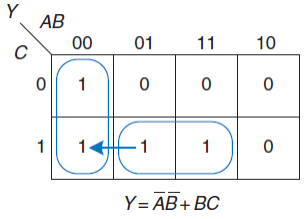


Figure 2. 48 Input change crosses implicant boundary

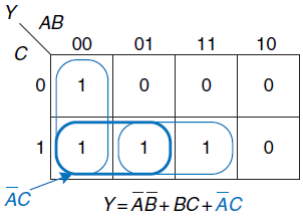


Figure 2. 49 K-map without glitch

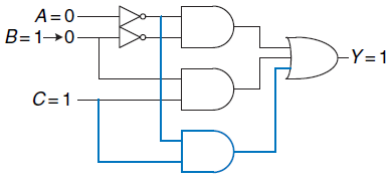


Figure 2. 50 Circuit without glitch

In general, a glitch **can** occur when a change in a single variable crosses the boundary between two prime implicants in a K-map. We can eliminate the glitch by adding **redundant** implicants to the K-map to **cover** these boundaries.

However, simultaneous transitions on multiple inputs can also cause glitches. These glitches cannot be fixed by adding hardware.

Exercises

Exercise 2.36: A ***priority encoder*** has 2^N inputs. It produces an N-bit binary output indicating the most significant bit of the input that is TRUE, or 0 if none of the inputs are TRUE. It also produces an output *NONE* that is TRUE if none of the inputs are TRUE. Design an eight-input priority encoder with inputs $A_{7:0}$ and outputs $Y_{2:0}$ and NONE. For example, if the input is 00100000, the output Y should be 101 and *NONE* should be 0. Give a simplified Boolean equation for each output, and sketch a schematic.

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Y_2	Y_1	Y_0	*NONE*
0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	X	0	0	1	0
0	0	0	0	0	1	X	X	0	1	0	0
0	0	0	0	1	X	X	X	0	1	1	0
0	0	0	1	X	X	X	X	1	0	0	0
0	0	1	X	X	X	X	X	1	0	1	0
0	1	X	X	X	X	X	X	1	1	0	0
1	X	X	X	X	X	X	X	1	1	1	0

$$Y_2 = \bar{A}_7\bar{A}_6\bar{A}_5A_4 + \bar{A}_7\bar{A}_6A_5 + \bar{A}_7A_6 + A_7 = A_7 + A_6 + A_5 + A_4 \quad \text{(k-map for 4 variables)}$$

$$Y_1 = \bar{A}_7\bar{A}_6\bar{A}_5\bar{A}_4\bar{A}_3A_2 + \bar{A}_7\bar{A}_6\bar{A}_5\bar{A}_4A_3 + \bar{A}_7A_6 + A_7$$

$$= \bar{A}_7\bar{A}_6\bar{A}_5\bar{A}_4\bar{A}_3A_2 + \bar{A}_7\bar{A}_6\bar{A}_5\bar{A}_4A_3 + A_7 + A_6 \quad \text{(k-map for: } \bar{A}_7A_6 + A_7\text{)}$$

$$= \bar{A}_7\bar{A}_6\bar{A}_5\bar{A}_4(\bar{A}_3A_2 + A_3) + A_7 + A_6 \quad \text{(k-map for: } \bar{A}_3A_2 + A_3\text{)}$$

$$= \bar{A}_7\bar{A}_6\bar{A}_5\bar{A}_4(A_2 + A_3) + A_7 + A_6 \quad \text{(treat } \bar{A}_6\bar{A}_5\bar{A}_4(A_2 + A_3) \text{ as } B\text{, k-map for: } \bar{A}_7B + A_7\text{)}$$

$$= \bar{A}_6\bar{A}_5\bar{A}_4(A_2 + A_3) + A_7 + A_6 \quad \text{(same as before)}$$

$$= \bar{A}_5\bar{A}_4(A_2 + A_3) + A_7 + A_6$$

$$Y_0 = A_7 + \bar{A}_6A_5 + \bar{A}_6\bar{A}_4A_3 + \bar{A}_6\bar{A}_4\bar{A}_2A_1$$

$$NONE = \bar{A}_7\bar{A}_6\bar{A}_5\bar{A}_4\bar{A}_3\bar{A}_2\bar{A}_1\bar{A}_0$$

3.2 LATCHES AND FLIP-FLOPS

The fundamental building block of **memory** is a ***bistable*** **element**, an element with two stable states. Figure 3.1 shows two same simple bistable element consisting of a pair of inverters connected in a loop. The inverters are *cross-coupled*, meaning that the input of I1 is the output of I2 and vice versa. The circuit has no inputs, but it does have two outputs, Q and $\bar{Q}$.

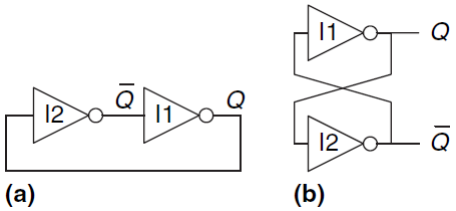


Figure 3. 1 Cross-coupled inverter pair

Just as Y is commonly used for the output of combinational logic, Q is commonly used for the output of sequential logic.

Because the cross-coupled inverters have two stable states, $Q = 0$ and $Q = 1$, the circuit is said to be bistable. A subtle point is that the circuit has a third possible state with both outputs approximately halfway between 0 and 1. This is called a *metastable* state and will be discussed in Section 3.5.4.

An element with N stable states conveys $\log_2 N$ bits of information, so a bistable element stores one bit.

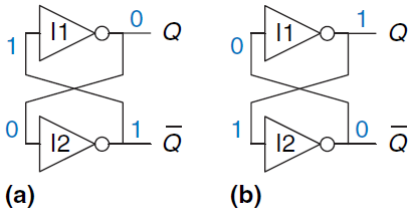


Figure 3. 2 Bistable operation of cross-coupled inverters

When power is first applied to a sequential circuit, the initial state is unknown and usually unpredictable. It may differ each time the circuit is turned on.

SR Latch

One of the simplest sequential circuits is the SR *latch*, which is composed of two cross-coupled NOR gates, as shown in Figure 3.3. The latch has two inputs, S and R, and two outputs, Q and $\bar{Q}$. The SR latch is similar to the cross-coupled inverters, but its state can be controlled through the S and R inputs, which set and reset the output Q.

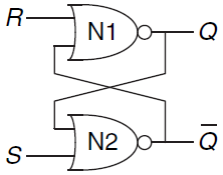


Figure 3. 3 SR latch schematic

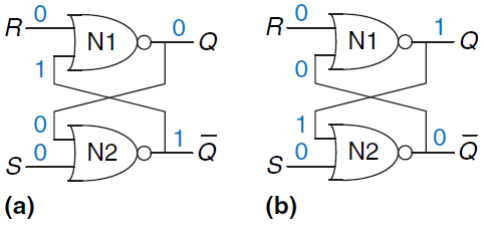


Figure 3. 4 Bistable states of SR latch

The truth table of SR latch is shown in Figure 3.5. The inputs S and R stand for *Set* and *Reset*. To set a bit **means** to make it **TRUE**. To **reset** a bit means to make it **FALSE**. The outputs, Q and $\bar{Q}$, are normally complementary. Asserting both S and R simultaneously doesn't make much

sense because it means the latch should be set and reset at the same time, which is impossible. The poor confused circuit responds by making both outputs 0.

Case	S	R	Q	$\bar{Q}$
IV	0	0	Q_{prev}	$\bar{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Figure 3. 5 SR latch truth table

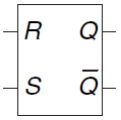


Figure 3. 6 SR latch symbol

Like the cross-coupled inverters, the SR latch is a bistable element with one bit of state stored in Q. However, the state can be controlled through the S and R inputs. When R is asserted, the state is reset to 0. When S is asserted, the state is set to 1. When neither is asserted, the state retains its **old value** Q_{prev}.

D Latch

The SR latch is awkward because it behaves strangely when both S and R are simultaneously asserted. Moreover, the S and R inputs conflate the issues of *what* and *when*. Asserting one of the inputs determines not only what the state should be but also when it should change. The D latch in Figure 3.7 solves these problems. It has two inputs. The data input, D, controls what the next state should be. The clock input, CLK, controls when the state should change.

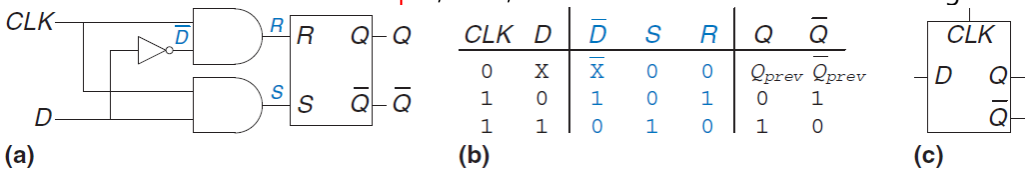


Figure 3. 7 D latch: (a) schematic, (b) truth table, (c) symbol

We see that the clock controls when data flows through the latch. When $CLK = 1$, the latch is transparent. The data at D flows through to Q as if the latch were just a buffer. When $CLK = 0$, the latch is *opaque*. It blocks the new data from flowing through to Q, and Q retains the old value. Hence, the D latch is sometimes called a *transparent latch* or a *level-sensitive* latch.

The D latch updates its state continuously while $CLK = 1$.

D Flip-Flop

A D *flip-flop* can be built from two back-to-back D latches controlled by complementary clocks, as shown in Figure 3.8. The first latch, L1, is called the *master*. The second latch, L2, is called the *slave*. When the $\bar{Q}$ output is not needed, the symbol is often condensed as in Figure 3.8 (c).

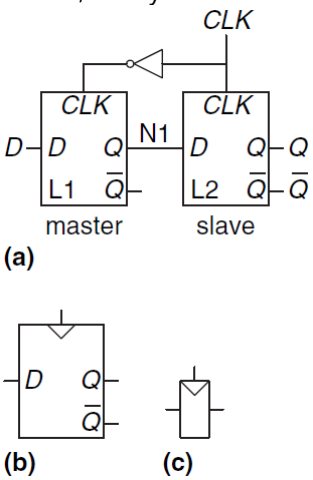


Figure 3. 8 D flip-flop: (a) schematic, (b) symbol, (c) condensed symbol

In other words, a D *flip-flop copies* D to Q on the *rising edge* of *the clock*, and *remembers its*

state at all other times. The rising edge of the clock is often just called the *clock edge* for brevity. The D input specifies what the new state will be. The clock edge indicates when the state should be updated.

A D flip-flop is also known as a *master-slave flip-flop*, an *edge-triggered flip-flop*, or a *positive edge-triggered flip-flop*. The triangle in the symbols denotes an edge-triggered clock input.

Example 3.1: FLIP-FLOP TRANSISTOR COUNT. (Page 114)

Register

An N-bit register is a bank of N flip-flops that share a common CLK input, so that all bits of the register are updated at the same time.

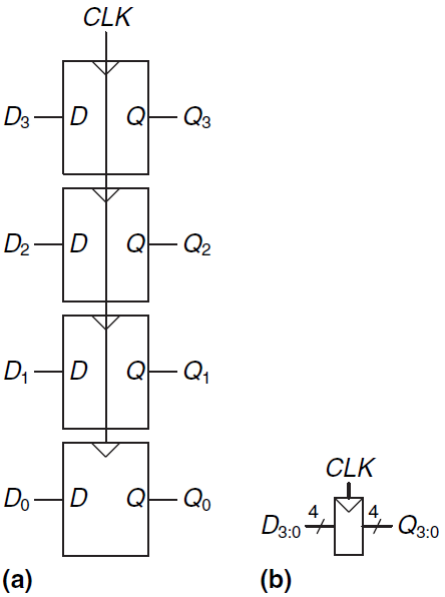


Figure 3. 9 A 4-bit register: (a) schematic and (b) symbol

Enabled Flip-Flop

An *enabled flip-flop* adds another input called *EN* or *ENABLE* to determine whether data is loaded on the clock edge. When *EN* is TRUE, the enabled flip-flop behaves like an ordinary D flip-flop. When *EN* is FALSE, the enabled flip-flop ignores the clock and retains its state. Enabled flip-flops are useful when we wish to load a new value into a flip-flop only some of the time, rather than on every clock edge.

Figure 3.10 shows two ways to construct an enabled flip-flop from a D flip-flop and an extra gate. In Figure 3.10 (a), an input multiplexer chooses whether to pass the value at D, if *EN* is TRUE, or to **recycle** the old state from Q, if *EN* is FALSE. Generally, performing logic on the clock is a bad idea. Clock gating delays the clock and can cause timing errors.

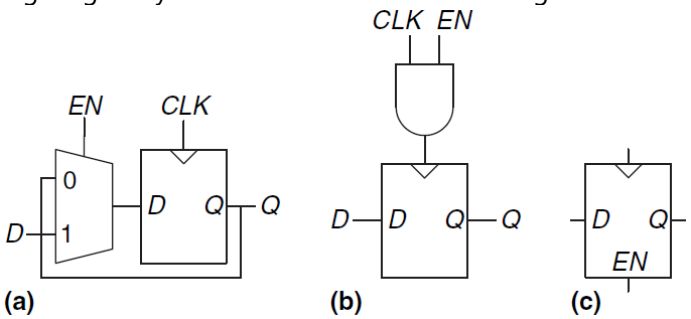


Figure 3. 10 Enabled flip-flop: (a, b) schematics, (c) symbol

Resettable Flip-Flop

A *resettable flip-flop* adds another input called *RESET*. When *RESET* is FALSE, the resettable flip-flop behaves like an ordinary D flip-flop. When *RESET* is TRUE, the resettable flip-flop ignores D and resets the output to 0. Resettable flip-flops are useful when we want to force a known state (i.e., 0) into all the flip-flops in a system when we first turn it on.

Such flip-flops may be *synchronously* or *asynchronously resettable*. Synchronously resettable flip-flops reset themselves only on the rising edge of *CLK*. Asynchronously resettable flip-flops reset themselves as soon as *RESET* becomes TRUE, independent of *CLK*.

Figure 3.11 shows how to construct a **synchronously** resettable flip-flop from an ordinary D flip-flop and an AND gate. $\overline{RESET}$ is an *active low* signal, meaning that the reset signal performs its function when it is 0, not 1. By adding an inverter, the circuit could have accepted an active high reset signal instead.

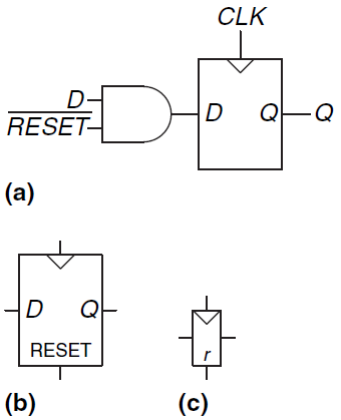


Figure 3. 11 Synchronously resettable flip-flop: (a) schematic, (b, c) symbols

As you might imagine, settable flip-flops are also occasionally used. They load an 1 into the flip-flop when SET is asserted, and they also come in synchronous and asynchronous flavors. Resettable and settable flip-flops may also have an enable input and may be grouped into N-bit registers.

Transistor-Level Latch and Flip-Flop Designs

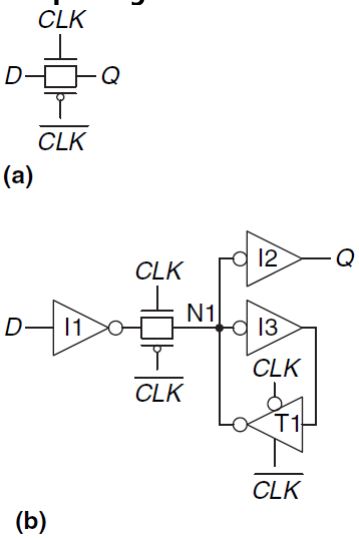


Figure 3. 12 D latch schematic

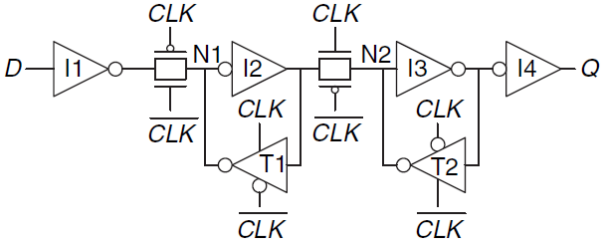


Figure 3. 13 D flip-flop schematic

Putting It All Together

Latches and flip-flops are the fundamental building blocks of sequential circuits.

Example 3.2: FLIP-FLOP AND LATCH COMPARISON

3.3 SYNCHRONOUS LOGIC DESIGN

In general, sequential circuits include all circuits that are not combinational- that is, those whose output cannot be determined simply by looking at the current inputs.

Some Problematic Circuits

Example 3.3: ASTABLE CIRCUITS

This circuit is called a ring oscillator. (Page 119)

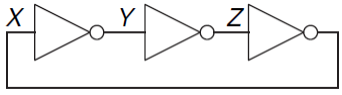


Figure 3. 14 Three-inverter loop

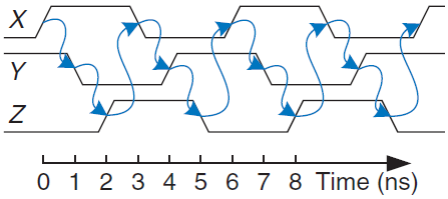


Figure 3. 15 Ring oscillator waveforms

The period of the ring oscillator depends on the propagation delay of each inverter. This delay depends on how the inverter was manufactured, the power supply voltage, and even the temperature. Therefore, the ring oscillator period is difficult to accurately predict. In short, the ring oscillator is a sequential circuit with zero inputs and one output that changes periodically.

Example 3.4: RACE CONDITIONS

This is an example of *asynchronous* circuit design in which outputs are directly fed back to inputs. Asynchronous circuits are infamous for having race conditions where the behavior of the circuit depends on which of two paths through logic gates is fastest. (Page 120)

Synchronous Sequential Circuits

The previous two examples contain loops called *cyclic paths*, in which outputs are fed directly back to inputs. They are sequential rather than combinational circuits. Combinational logic has no cyclic paths and no races. However, sequential circuits with cyclic paths can have undesirable races or unstable behavior.

To avoid these problems, designers break the cyclic paths by inserting registers somewhere in the path. This transforms the circuit into a collection of combinational logic and registers. The registers contain the state of the system, which changes only at the clock edge, so we say the state is *synchronized* to the clock. If the clock is sufficiently slow, so that the inputs to all registers settle before the next clock edge, all races are eliminated. Adopting this discipline of always using registers in the feedback path leads us to the formal definition of a synchronous sequential circuit.

A sequential circuit has a finite set of discrete *states* $\{S_0, S_1, \cdots, S_{k-1}\}$. A *synchronous sequential circuit* has a clock input, whose rising edges indicate a sequence of times at which state transitions occur. We often use the terms *current state* and *next state* to distinguish the state of the system at the present from the state to which it will enter on the next clock edge. The timing specification consists of an **upper bound**, t_{pcq}, and a **lower bound**, t_{ccq}, on the time from the rising edge of the clock until the output changes, as well as ***setup*** and ***hold*** times, t_{setup} and t_{hold}, that indicate when the *inputs* must be stable relative to the rising edge of the clock.

> t_{pcq} stands for the time of propagation from clock to Q, where Q indicates the output of a synchronous sequential circuit. t_{ccq} stands for the time of contamination from clock to Q.

The **rules** of *synchronous sequential circuit composition* teach us that a circuit is a synchronous sequential circuit if it consists of interconnected circuit elements such that

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register

- All registers receive the same clock signal
- Every cyclic path contains at least one register.

Sequential circuits that are not synchronous are called *asynchronous*.

A flip-flop is the **simplest** synchronous sequential circuit. It has one input, D, one clock, CLK, one output, Q, and two states, $\{0,1\}$. The functional specification for a flip-flop is that the next state is D and that the output, Q, is the current state, as shown in Figure 3.16.

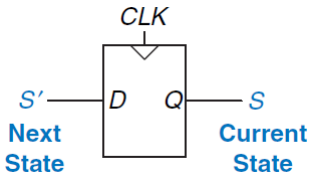


Figure 3. 16 Flip-flop current state and next state

We often call the current state variable S and the next state variable S'.

Synchronous and Asynchronous Circuits

Of course, **asynchronous** circuits are occasionally necessary when communicating between systems with different clocks or when receiving inputs at arbitrary times, just as analog circuits are necessary when communicating with the real world of continuous voltages.

3.4 FINITE STATE MACHINES

Synchronous sequential circuits can be drawn in the forms shown in Figure 3.17. These forms are called ***finite state machines*** (*FSMs*). They get their name because a circuit with k registers can be in one of a finite number (2^k) of unique states. An FSM has M inputs, N outputs, and k bits of state. It also receives a clock and, optionally, a reset signal. An FSM **consists** of two blocks of combinational logic, ***next state logic*** and ***output logic***, and a register that stores the state. On **each clock edge**, the FSM **advances** to the next state, which **was** computed based on the current state and inputs.

There are **two general** classes of finite state machines, characterized by their functional specifications. In *Moore machines*, the **outputs** depend only on the current state of the machine. In *Mealy machines*, the **outputs** depend on both the current state and the current inputs. Finite state machines provide a **systematic way** to **design** synchronous sequential circuits given a functional specification.

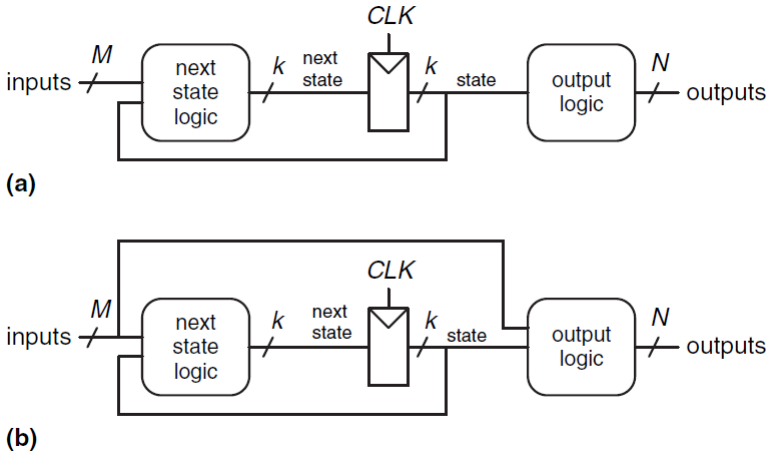


Figure 3. 17 Finite state machines: (a) Moore machine, (b) Mealy machine

FSM Design Example

To illustrate the design of FSMs, consider the problem of inventing a controller for a traffic light at a busy intersection on campus. Engineering students are moseying between their dorms and the labs on Academic Ave. Football players are hustling between the athletic fields and the dining hall on Bravado Boulevard. Several serious injuries have already occurred at the intersection of these two roads, and the Dean of Students asks Ben Bitdiddle to install **a traffic light** before there are fatalities.

Ben decides to solve the problem with an FSM. He installs **two traffic sensors**, T_A and T_B, on

Academic Ave. and Bravado Blvd., respectively. Each sensor indicates TRUE if students are present and FALSE if the street is empty. He also installs **two traffic lights**, L_A and L_B, to control traffic. Each light receives digital inputs specifying whether it should be green, yellow, or red. Hence, his FSM has two inputs, T_A and T_B, and two outputs, L_A and L_B. The intersection with lights and sensors is shown in Figure 3.18. Ben provides a clock with a 5-second period. On each clock tick (rising edge), the lights may change based on the traffic sensors. He also provides **a reset button** so that Physical Plant technicians can put the controller in a known initial state when they turn it on. Figure 3.19 shows a black box view of the state machine.

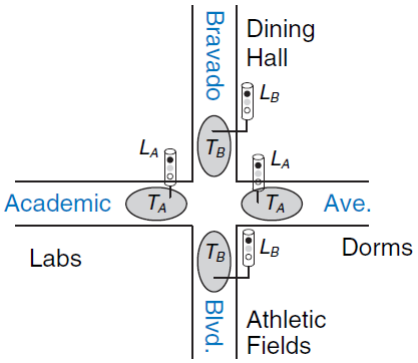


Figure 3. 18 Campus map

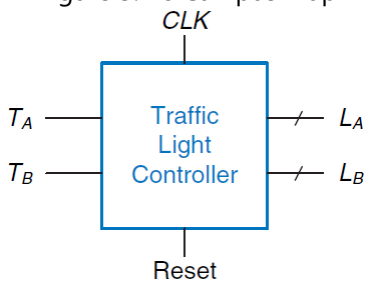


Figure 3. 19 Black box view of finite state machine

Ben's next step is to sketch the ***state transition diagram***, shown in Figure 3.20, to indicate all **the possible states** of the system and the **transitions** between these states.

In a state transition diagram, circles represent states and arcs represent transitions between states.

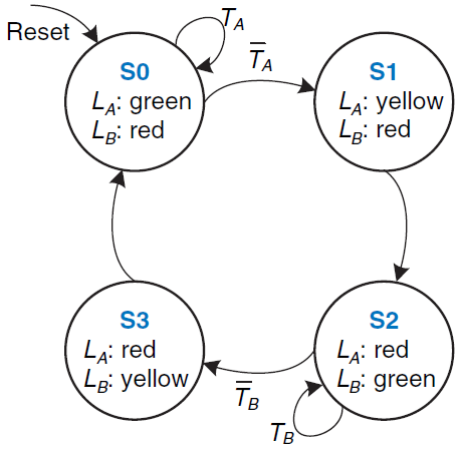


Figure 3. 20 State transition **diagram**

In a state transition diagram, **circles** represent **states** and **arcs** represent **transitions** between states. The transitions **take place** on the rising edge of the clock; we do not bother to show the clock on the diagram, because it is always present in a synchronous sequential circuit. Moreover, the **clock simply** controls **when** the transitions should occur, whereas the **diagram** indicates which transitions occur. The arc labeled Reset pointing from outer space into state S0 indicates

that the system should enter that state upon reset, regardless of what previous state it was in. If a state has **multiple arcs** leaving it, the arcs are **labeled** to show what **input** triggers each transition. If a state has a **single arc** leaving it, that transition **always** occurs regardless of the inputs. The value that the **outputs** have while in a particular state are **indicated** in the state.

Ben rewrites the state transition diagram as a ***state transition table*** (Figure 3.21), which indicates, **for each state and input**, what the **next** state, S', should be. Note that the table uses **don't care** symbols (X) whenever the next state does not depend on a particular input. Also note that Reset is omitted from the table. Instead, we use resettable flip-flops that always go to state S0 on reset, independent of the inputs.

Current State S	Inputs T_A	Inputs T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Figure 3. 21 State transition table

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Figure 3. 22 State **encoding**

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

Figure 3. 23 Output encoding

The state transition diagram is abstract in that it uses states labeled {S0,S1,S2,S3} and outputs labeled {red,yellow,green}. To build a real circuit, the states and outputs must be assigned ***binary encodings***.

Current State S_1	Current State S_0	Inputs T_A	Inputs T_B	Next State S'_1	Next State S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Figure 3. 24 State transition **table** with binary **encodings**

The **revised** state transition table is a truth table specifying the next state logic. It defines next state, S', as a **function** of the current state, S, and the inputs.

From this table, it is **straightforward** to read off the Boolean equations for the next state in **sum-of-products** form.

$$S'_1 = \bar{S}_1 S_0 + S_1 \bar{S}_0 \bar{T}_B + S_1 \bar{S}_0 T_B$$
$$S'_0 = \bar{S}_1 \bar{S}_0 \bar{T}_A + S_1 \bar{S}_0 \bar{T}_B$$

The equations can be simplified using Karnaugh maps, but often doing it **by inspection** is easier. To give the simplified next state equations:

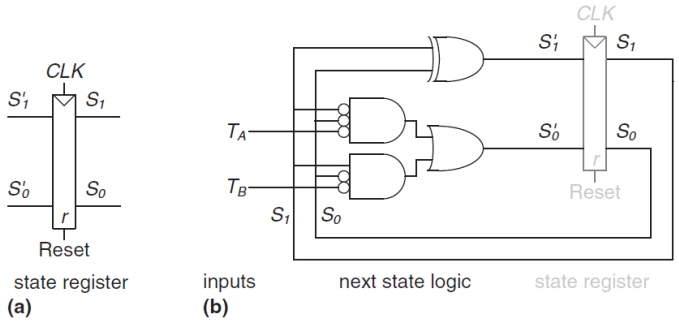
$$S'_1 = S_1 \oplus S_0$$
$$S'_0 = \bar{S}_1 \bar{S}_0 \bar{T}_A + S_1 \bar{S}_0 \bar{T}_B$$

Similarly, Ben writes an ***output table*** (Figure 3.25) indicating, for **each state**, what the **output** should be in that state. Again, it is **straightforward** to read off and simplify the Boolean equations for the outputs.

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Figure 3. 25 Output **table**

$$L_{A1} = S_1$$
$$L_{A0} = \bar{S}_1 S_0$$
$$L_{B1} = \bar{S}_1$$
$$L_{B0} = S_1 S_0$$



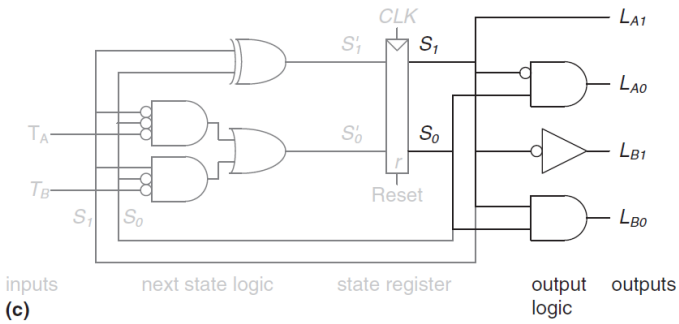


Figure 3. 26 State machine circuit for traffic light controller

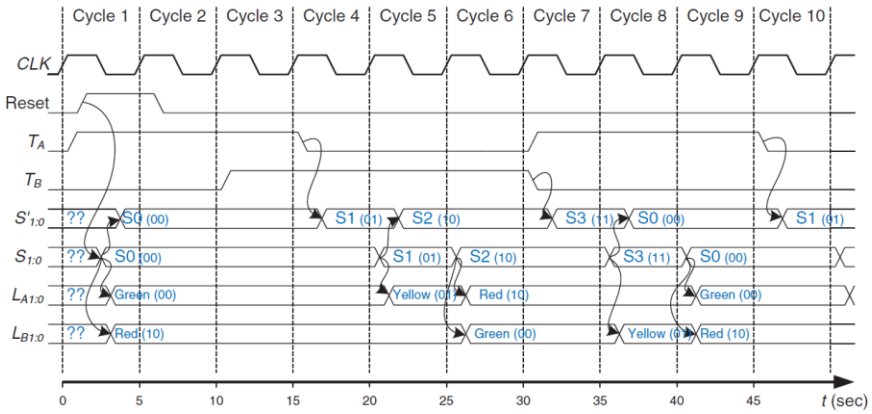


Figure 3. 27 Timing diagram for traffic light controller

State Encodings

In the previous example, the state and output encodings were selected arbitrarily. A different choice would have resulted in a different circuit. A natural question is how to determine the encoding that produces the circuit with the fewest logic gates or the shortest propagation delay. One important decision in state encoding is the choice between binary encoding (full encoding) and one-hot encoding. (and output encoding)

Supplement materials

Output encoding:
Outputs are directly accessible in the state encoding. For example, since we have 3 outputs (light color), encode state with 3 bits, where each bit represents a color.
Example states: 001, 010, 100, 110. Bit_0 encodes **green** light output, Bit_1 encodes **yellow** light output, Bit_2 and encodes **red** light output. This can minimize output logic, but it only works for Moore Machines (output function of state).
Example 3.6: FSM STATE ENCODING (Page 129)
A *divide-by-N counter* has one output and no inputs. The output Y is HIGH for one clock cycle out of every N. In other words, the output divides the frequency of the clock by N. The waveform and state transition diagram for a divide-by-3 counter is shown in Figure 3.28. Sketch circuit designs for such a counter using binary and **one-hot** state encodings. (A related encoding is the ***one-cold*** encoding, in which K states are represented with K bits, exactly one of which is FALSE.)

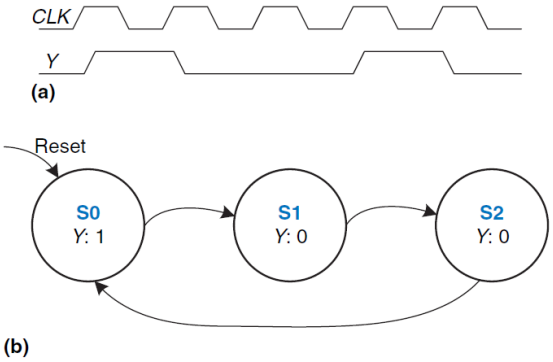


Figure 3. 28 Divide-by-3 counter (a) waveform and (b) state transition diagram

Current State	Next State
S0	S1
S1	S2
S2	S0

Figure 3. 29 Divide-by-3 counter state transition table

Current State	Output
S0	1
S1	0
S2	0

Figure 3. 30 Divide-by-3 counter output table

State	One-Hot Encoding			Binary Encoding	
	S_2	S_1	S_0	S_1	S_0
S0	0	0	1	0	0
S1	0	1	0	0	1
S2	1	0	0	1	0

Figure 3. 31 One-hot and binary encodings for divide-by-3 counter

Current State		Next State	
S_1	S_0	S_1'	S_0'
0	0	0	1
0	1	1	0
1	0	0	0

Figure 3. 32 State transition table with binary encoding

Current State			Next State		
S_2	S_1	S_0	S_2'	S_1'	S_0'
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1

Figure 3. 33 State transition table with one-hot encoding

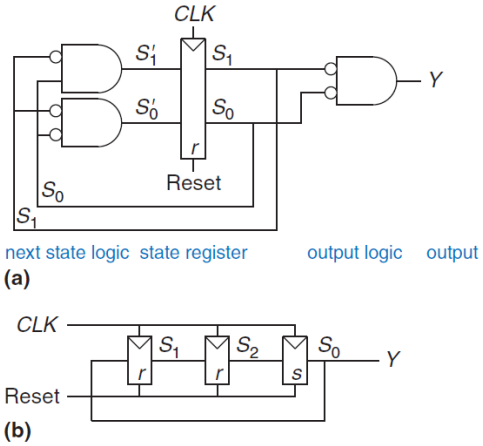


Figure 3. 34 Divide-by-3 circuits for (a) binary and (b) one-hot encodings

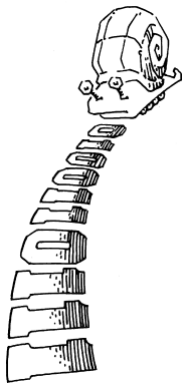
Observe that the one-hot encoding requires both settable (s) and resettable (r) flip-flops to initialize the machine to S0 on reset. The one-hot design is usually preferable for this specific example.

Moore and Mealy Machines

In state transition diagrams for Moore machines, the outputs are labeled in the circles. In state transition diagrams for Mealy machines, the **outputs** are labeled on the arcs instead of in the circles, since the outputs **depend** on inputs as well as the current state.

Example 3.7: MOORE VERSUS MEALY MACHINES

Alyssa P. Hacker owns a pet robotic snail with an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 1's and 0's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last two bits that it has crawled over are 01. Design the FSM to compute when the snail should smile. The input A is the bit underneath the snail's antenna. The output Y is TRUE when the snail smiles. Compare Moore and Mealy state machine designs. Sketch a timing diagram for each machine showing the input, states, and output as Alyssa's snail crawls along the sequence 0100110111.



Solution: The Moore machine requires three states, as shown in Figure 3.35(a). Convince yourself that the state transition diagram is correct. In particular, why is there an arc from S2 to S1 when the input is 0?

In comparison, the Mealy machine requires only two states, as shown in Figure 3.35(b). Each arc is labeled as A/Y. A is the value of the input that causes that transition, and Y is the corresponding output.

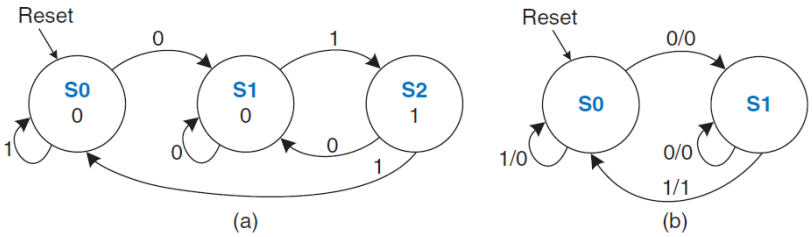


Figure 3. 35 FSM state transition diagrams: (a) Moore machine, (b) Mealy machine

Current State S	Input A	Next State S'
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

Figure 3. 36 Moore state transition table

Current State S	Output Y
S0	0
S1	0
S2	1

Figure 3. 37 Moore output table

Figures 3.36 and 3.37 show the state transition and output tables for the Moore machine. The Moore machine requires at least two bits of state. Consider using a binary state encoding: $S0 = 00$, $S1 = 01$, and $S2 = 10$. Figures 3.38 and 3.39 rewrite the state transition and output tables with these encodings.

Current State S_1	Current State S_0	Input A	Next State S_1'	Next State S_0'
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

Figure 3. 38 Moore state transition table with state encodings

Current State S_1	Current State S_0	Output Y
0	0	0
0	1	0
1	0	1

Figure 3. 39 Moore output table with state encodings

From these tables, we find the next state and output equations by inspection. Note that these equations are simplified using the fact that state 11 does not exist. Thus, the corresponding next state and output for the non-existent state are don't cares (not shown in the tables). We use the don't cares to minimize our equations.

$$S_1' = S_0 A$$
$$S_1' = \bar{A}$$
$$Y = S_1$$

Current State S	Input A	Next State S'	Output Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

Figure 3. 40 Mealy state transition and output table

Figure 3.40 shows the combined state transition and output table for the Mealy machine. The Mealy machine requires only one bit of state. Consider using a binary state encoding: $S0 = 0$ and $S1 = 1$. Figure 3.41 the state transition and output table with these encodings.

Current State S_0	Input A	Next State S_0'	Output Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

Figure 3. 41 Mealy state transition and output table with state encodings

From these tables, we find the next state and output equations by inspection.

$$S_1' = \bar{A}$$
$$Y = S_0 A$$

The two machines follow a different sequence of states. **Moreover**, the **Mealy machine's** output rises a cycle **sooner** because it **responds** to the **input** rather than waiting for the state change. If the Mealy output were delayed through a flip-flop, it would match the Moore output

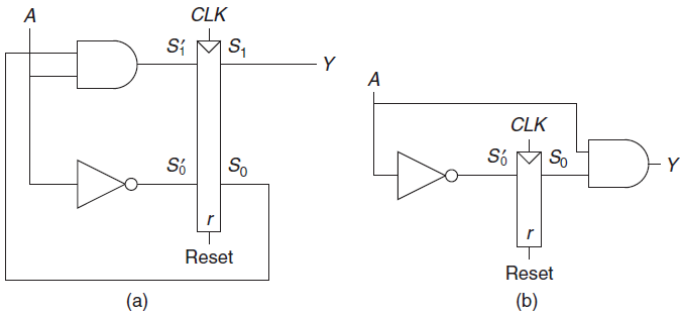


Figure 3. 42 FSM schematics for (a) Moore and (b) Mealy machines

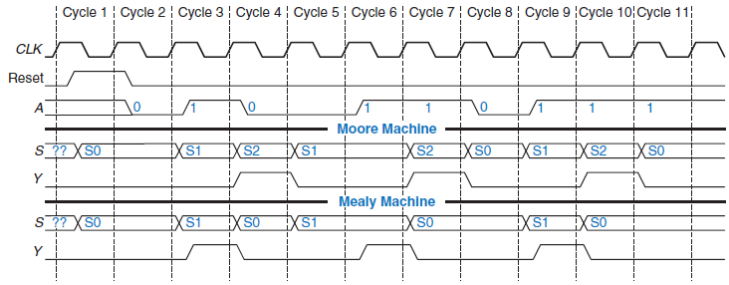


Figure 3. 43 Timing diagrams for Moore and Mealy machines

Factoring State Machines

Designing **complex** FSMs is often easier if they can be broken down into multiple interacting simpler state machines such that the output of some machines is the input of others. This application of hierarchy and modularity is called *factoring* of state machines.

Example 3.8: UNFACTORED AND FACTORED STATE MACHINES

Modify the traffic light controller from Section 3.4.1 to have a parade mode, which keeps the Bravado Boulevard light green while spectators and the band march to football games in scattered groups. The controller receives two more inputs: P and R. Asserting P for at least one cycle enters parade mode. Asserting R for at least one cycle leaves parade mode. When in parade mode, the controller proceeds through its usual sequence until L_B turns green, then remains in that state with L_B green until parade mode ends.

First, sketch a state transition diagram for a single FSM, as shown in Figure 3.44(a). Then, sketch the state transition diagrams for two interacting FSMs, as shown in Figure 3.44(b). The Mode FSM asserts the output M when it is in parade mode. The Lights FSM controls the lights based on M and the traffic sensors,

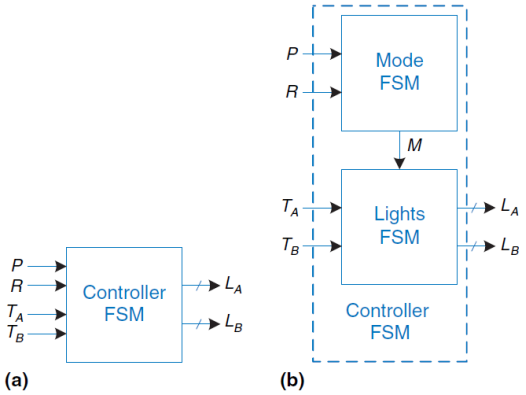


Figure 3. 44 (a) single and (b) factored designs for modified traffic light controller FSM

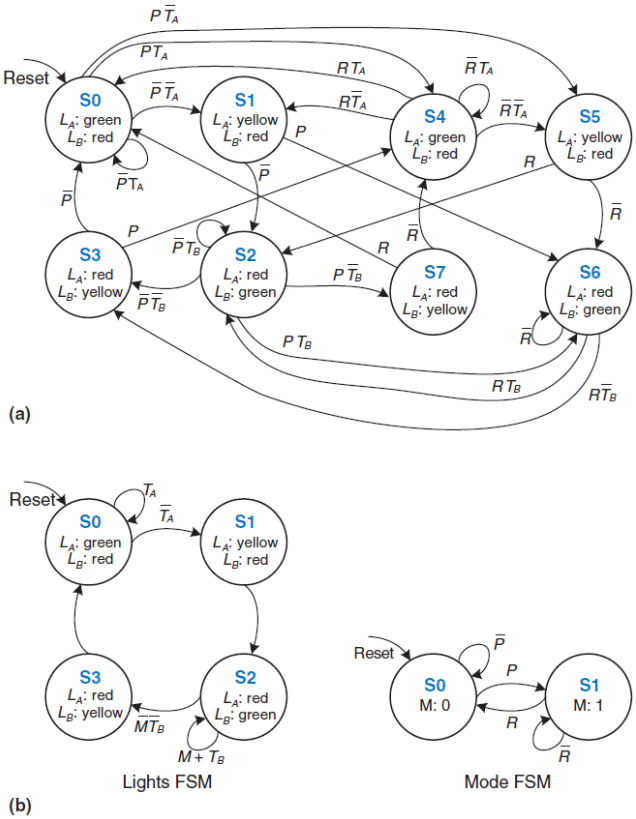


Figure 3. 45 State transition diagrams: (a) unfactored, (b) factored

Deriving an FSM from a Schematic

Deriving the state transition diagram from a schematic follows nearly the **reverse** process of FSM design. This process can be **necessary**, for example, when taking on an incompletely documented project or reverse engineering somebody else's system.

- Examine circuit, stating inputs, outputs, and state bits.
- Write next state and output equations.
- Create next state and output tables.
- Reduce the next state table to eliminate unreachable states.
- Assign each valid state bit combination a name.
- Rewrite next state and output tables with state names.
- Draw state transition diagram.
- State in words what the FSM does.

In the final step, be **careful** to succinctly describe the overall purpose and function of the FSM—do not simply restate each transition of the state transition diagram.

Example 3.9: DERIVING AN FSM FROM ITS CIRCUIT

Alyssa P. Hacker arrives home, but her keypad lock has been rewired and her old code no longer works. A piece of paper is taped to it showing the circuit diagram in Figure 3.46. Alyssa thinks the circuit could be a finite state machine and decides to derive the state transition diagram to see if it helps her get in the door.

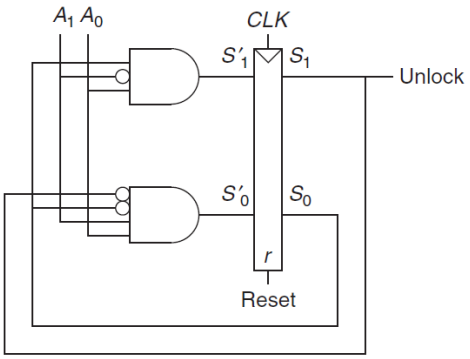


Figure 3. 46 Circuit of found FSM for Example 3.9

This is a Moore machine because the output depends only on the state bits. From the circuit, she writes down the next state and output equations directly:

$$S'_1 = S_0\bar{A}_1A_0$$
$$S'_0 = \bar{S}_1\bar{S}_0A_1A_0$$
$$unlock = S_1$$

Next, she writes down the next state and output tables from the equations, as shown below,

Current State S_1	Current State S_0	Input A_1	Input A_0	Next State S'_1	Next State S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0

Figure 3. 47 Next state table derived from circuit in Figure 3.46

Current State S_1	Current State S_0	Output *Unlock*
0	0	0
0	1	0
1	0	1
1	1	1

Figure 3. 48 Output table derived from circuit in Figure 3.46

Alyssa reduces the table by removing unused states and combining rows using don't cares. The $S_{1:0} = 11$ state is never listed as a possible next state in Figure 3.47, so rows with this current state are removed. For current state $S_{1:0} = 10$, the next state is always $S_{1:0} = 00$, independent of the inputs, so don't cares are inserted for the inputs.

Current State S_1	Current State S_0	Input A_1	Input A_0	Next State S_1'	Next State S_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	X	X	0	0

Current State S	Input A	Next State S'
S0	0	S0
S0	1	S0
S0	2	S0
S0	3	S1
S1	0	S0
S1	1	S2
S1	2	S0
S1	3	S0
S2	X	S0

Figure 3. 49 Reduced next state table

Current State S_1	Current State S_0	Output *Unlock*
0	0	0
0	1	0
1	0	1

Figure 3. 50 Reduced output table

Figure 3. 51 Symbolic next state table

Current State S	Output *Unlock*
S0	0
S1	0
S2	1

Figure 3. 52 Symbolic output table

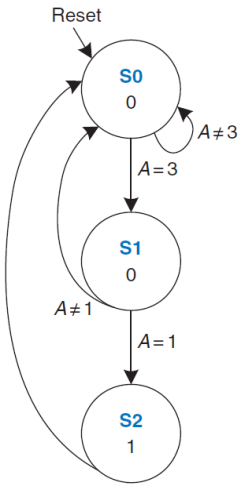


Figure 3. 53 State transition diagram of found FSM from Example 3.9

FSM Review

Finite state machines are a powerful way to **systematically** design sequential circuits from a written specification. Use the following **procedure** to design an FSM:

- Identify the inputs and outputs.
- Sketch a state transition diagram.
- For a **Moore** machine:

Write a state transition table.

Write an output table.

- For a **Mealy** machine:

Write a combined state transition and output table.

- Select state encodings—your selection affects the hardware design.
- Write Boolean equations for the next state and output logic.
- Sketch the circuit schematic.

3.5 TIMING OF SEQUENTIAL LOGIC

Recall that a flip-flop copies the input D to the output Q on the rising edge of the clock. This process is called *sampling* D on the clock edge. If D is *stable* at either 0 or 1 when the clock rises, this behavior is clearly defined.

A camera is characterized by its *aperture time*, during which the object must remain still for a sharp image to be captured. Similarly, a sequential element has an aperture time around the

clock edge, during which the input must be stable for the flip-flop to produce a well-defined output.

The aperture of a sequential element is defined by a ***setup*** time and a ***hold*** time, before and after the clock edge, respectively. Just as the static discipline limited us to using logic levels outside the forbidden zone, the *dynamic discipline* limits us to using signals that change outside the aperture time.

The clock period has to be long enough for all signals to settle. This sets a limit on the speed of the system. In real systems, the clock does not reach all flip-flops at precisely the same time. This variation in time, called clock skew, further increases the necessary clock period.

The Dynamic Discipline

Recall that a synchronous sequential circuit, such as a flip-flop or FSM, also has a timing specification, as illustrated in Figure 3.54. When the clock rises, the output (or outputs) may start to change after the clock-to-Q *contamination delay*, t_{ccq}, and must definitely settle to the final value within the clock to-Q *propagation delay*, t_{pcq}. These represent the fastest and slowest delays through the circuit, respectively. For the circuit to sample its input correctly, the input (or inputs) must have stabilized at least some *setup time*, t_{setup}, before the rising edge of the clock and must remain stable for at least some *hold time*, t_{hold}, after the rising edge of the clock. The sum of the setup and hold times is called the aperture time of the circuit, because it is the total time for which the input must remain stable.

The *dynamic discipline* states that the inputs of a synchronous sequential circuit must be stable during the setup and hold aperture time around the clock edge.

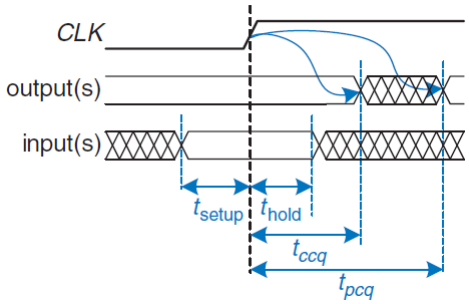


Figure 3. 54 Timing specification for synchronous sequential circuit

System Timing

The *clock period* or *cycle time*, T_c, is the time between rising edges of a repetitive clock signal. Its reciprocal, $f_c = 1/T_c$, is the *clock frequency*. All else being the same, increasing the clock frequency increases the work that a digital system can accomplish per unit time. Frequency is measured in units of Hertz (Hz), or cycles per second: 1 megahertz (MHz) $= 10^6$ Hz, and 1 gigahertz (GHz) $= 10^9$ Hz.

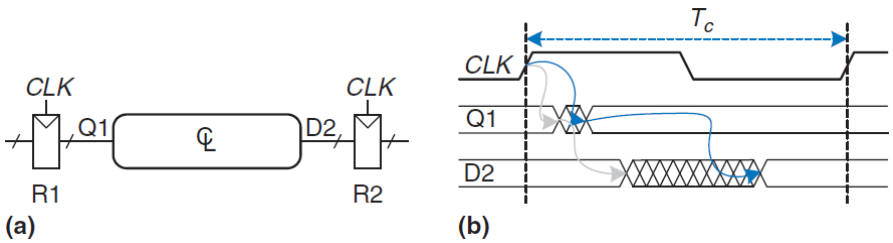


Figure 3. 55 Path between registers and timing diagram

Setup Time Constraint

Figure 3.56 is the timing diagram showing only the maximum delay through the path, indicated by the blue arrows. To satisfy the setup time of R2, $D2$ must settle no later than the setup time before the next clock edge. Hence, we find an equation for the minimum clock period:

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}}$$

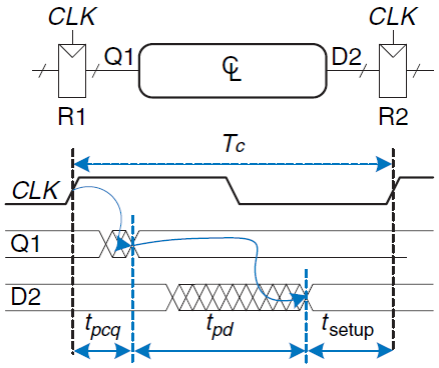


Figure 3. 56 Maximum delay for setup time constraint

In commercial designs, the clock period is often dictated by the Director of Engineering or by the marketing department. Moreover, the flip-flop clock-to-Q propagation delay and setup time, t_{pcq} and t_{setup} are specified by the manufacturer. Hence, the maximum propagation delay through the combinational logic is usually the only variable under the control of the individual designer. We rearrange the above equation:

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}})$$

The term in parentheses, $t_{pcq} + t_{\text{setup}}$, is called the *sequencing overhead*.

The above equation is called the *setup time constraint* or *max-delay constraint*, because it depends on the setup time and limits the maximum delay through combinational logic.

Hold Time Constraint

The register R2 in Figure 3.55 also has a *hold time constraint*. Its input, $D2$, must not change until some time, t_{hold}, after the rising edge of the clock. According to Figure 3.57, $D2$ might change as soon as $t_{ccq} + t_{cd}$ after the rising edge of the clock. Hence, we find

$$t_{ccq} + t_{cd} \geq t_{\text{hold}}$$

Again, t_{ccq} and t_{hold} are characteristics of the flip-flop that are usually outside the designer's control. Rearranging, we can solve for the minimum contamination delay through the combinational logic:

$$t_{cd} \geq t_{\text{hold}} - t_{ccq}$$

The above equation is also called the *hold time constraint* or *min-delay constraint* because it limits the minimum delay through combinational logic.

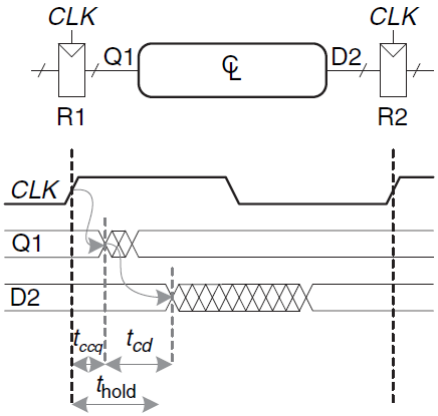


Figure 3. 57 Minimum delay for hold time constraint

We have assumed that any logic elements can be connected to each other without introducing timing problems. In particular, we would expect that two flip-flops may be directly cascaded as in Figure 3.58 without causing hold time problems.

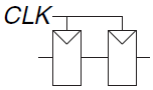


Figure 3. 58 Back-to-back flip-flops

In such a case, $t_{cd} = 0$ because there is no combinational logic between flip-flops. The hold time constraint yields the requirement that

$$t_{\text{hold}} \leq t_{ccq}$$

In other words, a reliable flip-flop must have a hold time shorter than its contamination delay. Often, flip-flops are designed with $t_{\text{hold}} = 0$.

Nevertheless, hold time constraints are critically important. If they are violated, the only solution is to increase the contamination delay through the logic, which requires redesigning the circuit.

Putting It All Together

Sequential circuits have setup and hold time constraints that dictate the maximum and minimum delays of the combinational logic between flip-flops.

Example 3.10: TIMING ANALYSIS (Page 145)

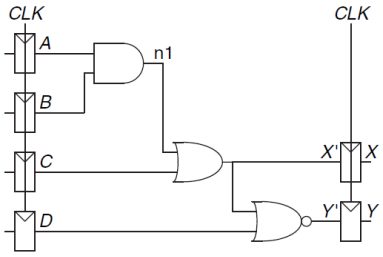


Figure 3. 59 Sample circuit for timing analysis

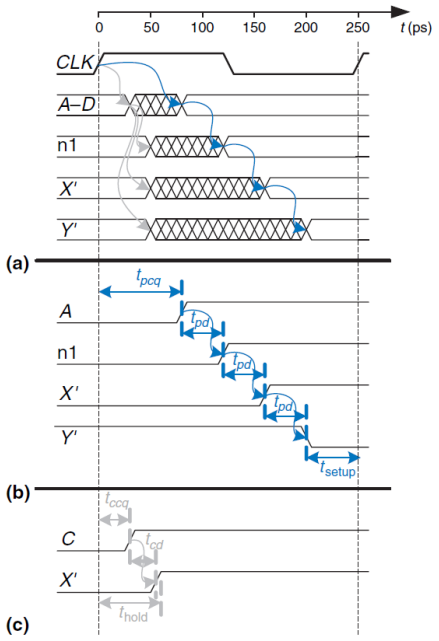


Figure 3. 60 Timing diagram: (a) general case, (b) critical path, (c) short path

Example 3.11: FIXING HOLD TIME VIOLATIONS (Page 147)

Alyssa P. Hacker proposes to fix Ben's circuit by adding buffers to slow down the short paths, as shown in Figure 3.61.

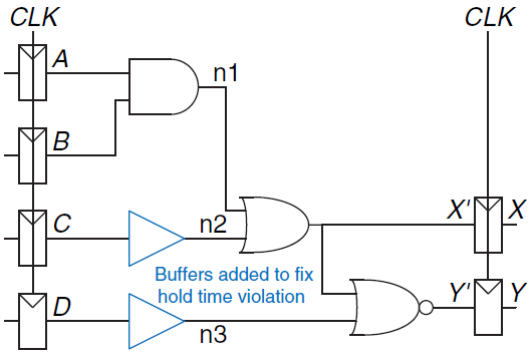


Figure 3. 61 Corrected circuit to fix hold time problem

Clock Skew

In the previous analysis, we assumed that the clock reaches all registers at exactly the same time. In reality, there is some variation in this time. This variation in clock edges is called *clock skew*. For example, the wires from the clock source to different registers may be of different lengths, resulting in slightly different delays, as shown in Figure 3.62. Noise also results in different delays. Clock gating, described in Section 3.2.5, further delays the clock. If some clocks are gated and others are not, there will be substantial skew between the gated and ungated clocks.

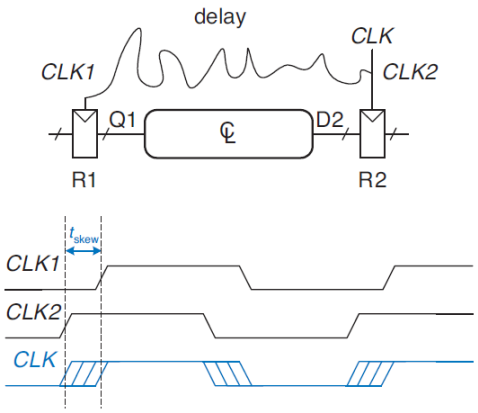


Figure 3. 62 Clock skew caused by wire delay

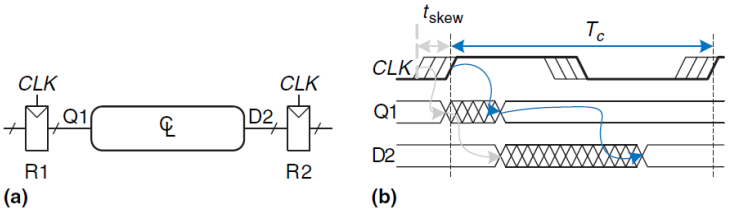


Figure 3. 63 Timing diagram with clock skew

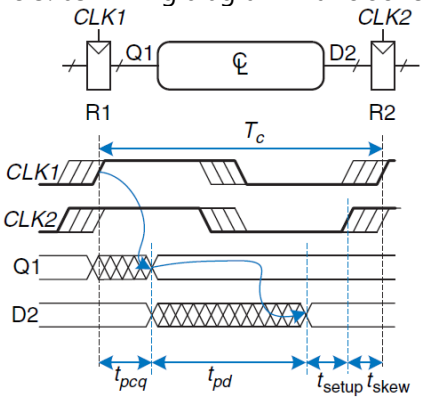


Figure 3. 64 Setup time constraint with clock skew

CLK arrives at $R2$ before $R1$, and this effectively increase t_{setup}:

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} + t_{\text{skew}}$$

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}} + t_{\text{skew}})$$

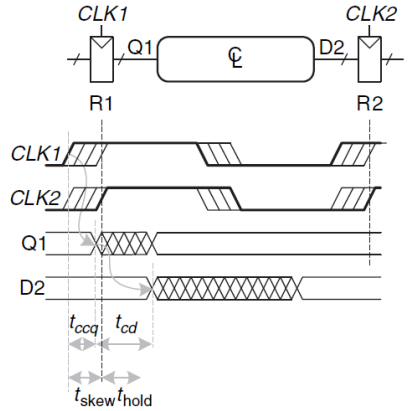


Figure 3. 65 Hold time constraint with clock skew

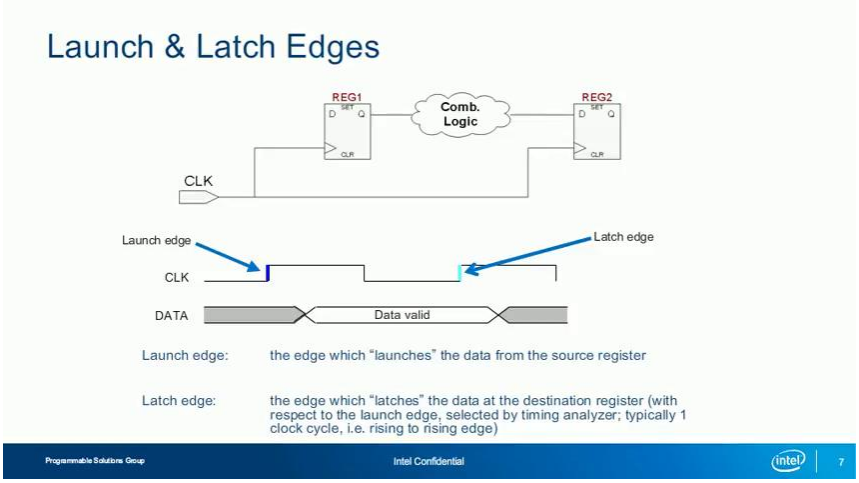
CLK arrives at $R2$ after $R1$, and this effectively increase t_{hold}:

$$t_{ccq} + t_{cd} \geq t_{\text{hold}} + t_{\text{skew}}$$

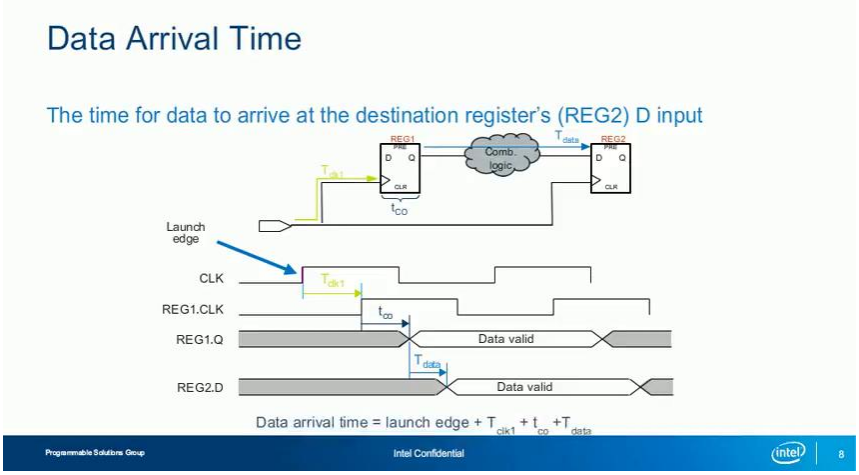
$$t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} - t_{ccq}$$

Example 3.12: TIMING ANALYSIS WITH CLOCK SKEW (Page 150)

Example 3.13: FIXING HOLD TIME VIOLATIONS (Page 150)
Addition:
(From Intel FPGA)
Timing Analyzer: Introduction to Timing Analysis
Firstly, we introduce some terminology,



Launch edge: the edge which "launches" the data from the source register.
Latch edge: the edge which "latches" the data at the destination register (with respect to the launch edge, selected by timing analyzer; typically 1 clock cycle, i.e. rising to rising edge).

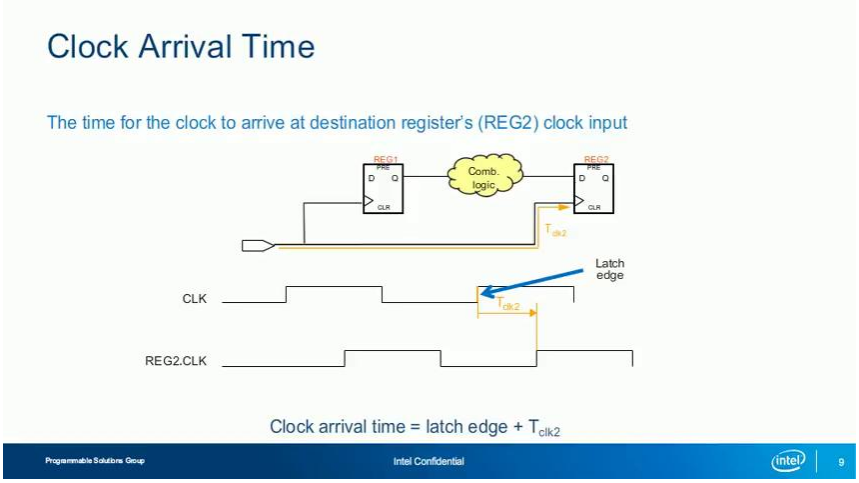


Data arrival time: the time for data to arrive at the destination register's (REG2) D input.

$$\text{Data arrival time} = \text{launch edge} + T_{\text{clk1}} + t_{\text{CO}} + T_{\text{data}}$$

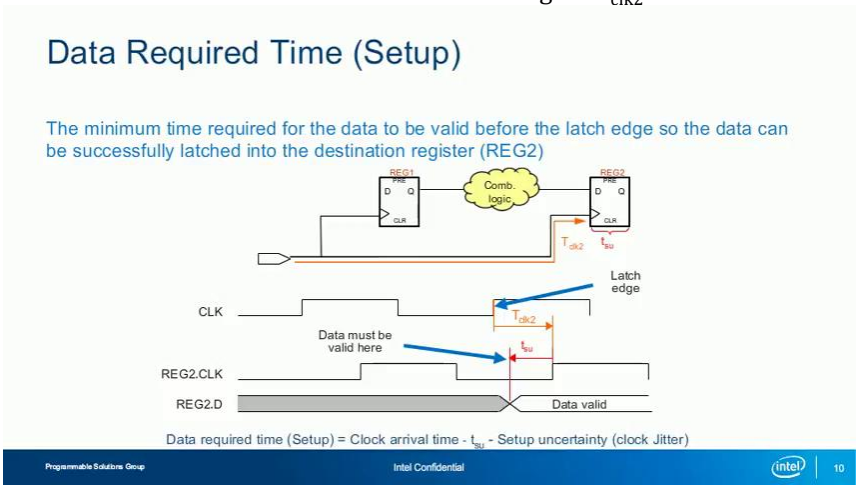
Since $t_{\text{CO}} = t_{pcq}, T_{\text{data}} = T_{pd}$, so

$$\text{Data arrival time} = \text{launch edge} + T_{\text{clk1}} + t_{pcq} + T_{pd}$$



Clock arrival time: the time for the clock to arrive at destination register's (REG2) clock input.

$$\text{Clock arrival time} = \text{latch edge} + T_{\text{clk2}}$$

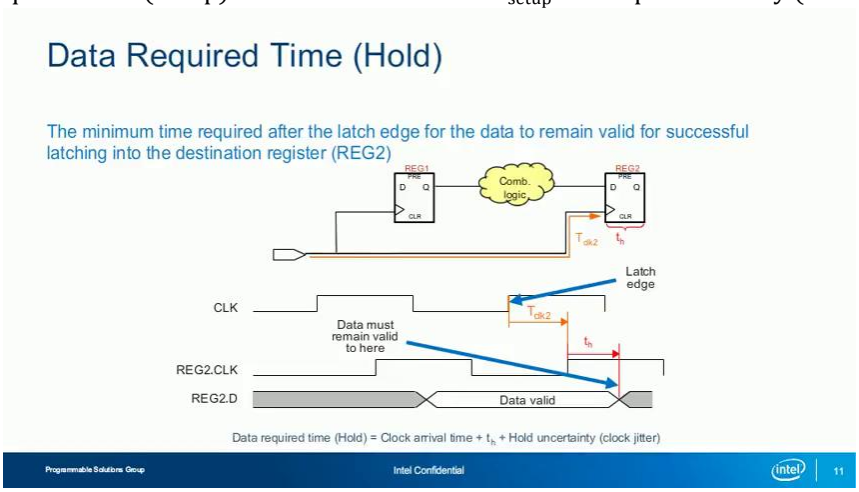


Data required time (Setup): the minimum time required for data to be valid before the latch edge so the data can be successfully latched into the destination register (REG2).

$$\text{Data required time (Setup)} = \text{Clock arrival time} - t_{\text{SU}} - \text{Setup uncertainty (clock Jitter)}$$

Since $t_{\text{SU}} = t_{\text{setup}}$,

$$\text{Data required time (Setup)} = \text{Clock arrival time} - t_{\text{setup}} - \text{Setup uncertainty (clock Jitter)}$$



Data required time (Hold): the minimum time required after the latch edge for the data to remain valid for successful latching into the destination register (REG2).

$$\text{Data required time (Hold)} = \text{Clock arrival time} + t_{\text{h}} + \text{Hold uncertainty (clock Jitter)}$$

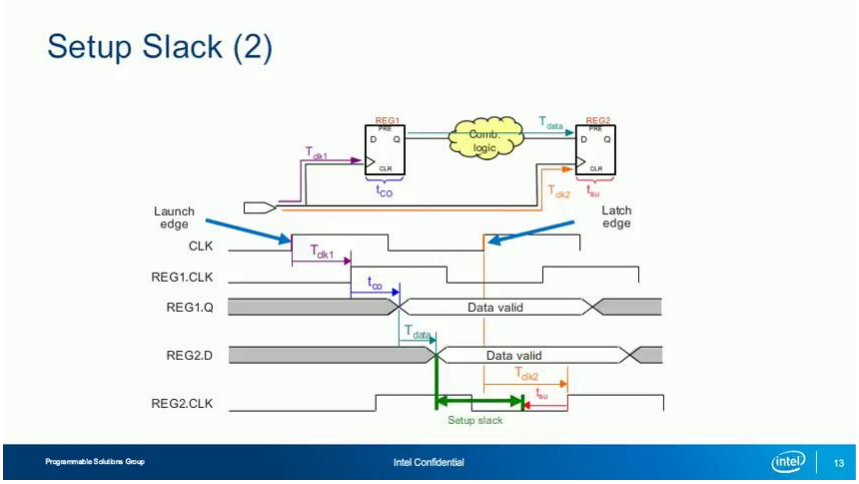
Since $t_{\text{h}} = t_{\text{hold}}$,

$$\text{Data required time (Hold)} = \text{Clock arrival time} + t_{\text{hold}} + \text{Hold uncertainty (clock Jitter)}$$

Our target is to make Setup Slack positive.

Setup Slack:

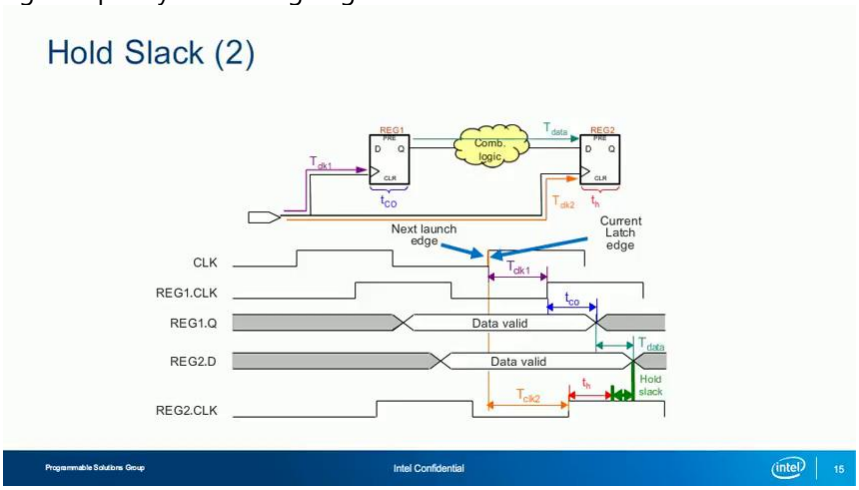
- Margin by which the setup timing requirement is met;
- Ensures launched data arrives in time to meet the latching requirement.



$$\text{Setup Slack} = \text{Data required time (Setup)} - \text{Data arrival time}$$
$$= \text{Clock arrival time} - t_{\text{setup}} - \text{Setup uncertainty (clock Jitter)}$$
$$- \left(\text{launch edge} + T_{\text{clk1}} + t_{pcq} + T_{pd}\right)$$

Hold Slack:

- Margin by which the hold timing requirement is met;
- Ensures latch data is not corrupted by data from the next launch edge i.e. the input data cannot change to quickly after rising edge of the clock.



$$\text{Hold Slack} = (T_{\text{clk1}} + t_{\text{CO}} + T_{\text{data}}) - (T_{\text{clk2}} + t_{\text{h}})$$
$$= \left(T_{\text{clk1}} + t_{ccq} + T_{cd}\right) - (T_{\text{clk2}} + t_{\text{hold}})$$

So, for each Slack,

Positive → Timing requirement met;

Negative → Timing requirement not met.

And equations need work for all timing paths. (Internal, I/O, & asynchronous control)

Metastability

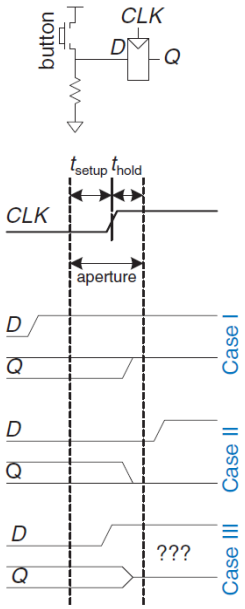


Figure 3. 66 Input changing before, after, or during aperture

Metastable State

When a flip-flop samples an input that is changing during its aperture, the output Q may momentarily take on a voltage between 0 and V_{DD} that is in the forbidden zone. This is called a *metastable state*. Eventually, the flip-flop will resolve the output to a *stable state* of either 0 or 1. However, the *resolution time* required to reach the stable state is unbounded.

Resolution Time

If a flip-flop input changes at a random time during the clock cycle, the resolution time, t_{res}, required to resolve to a stable state is also a random variable. If the input changes outside the aperture, then $t_{res} = t_{pcq}$. But if the input happens to change within the aperture, t_{res} can be substantially longer. Theoretical and experimental analyses (see Section 3.5.6) have shown that the probability that the resolution time, t_{res}, exceeds some arbitrary time, t, decreases exponentially with t: (Exponential Random Variables)

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-\frac{t}{\tau}}$$

where T_c is the clock period, and T_0 and τ are characteristic of the flip-flop. The equation is valid only for t substantially longer than t_{pcq}.

Synchronizers

Asynchronous inputs to digital systems from the real world are inevitable. The goal of a digital system designer should be to ensure that, given asynchronous inputs, the probability of encountering a metastable voltage is sufficiently small. To guarantee good logic levels, all asynchronous inputs should be passed through *synchronizers*.

A synchronizer, shown in Figure 3.67, is a device that receives an asynchronous input D and a clock CLK. It produces an output Q within a bounded amount of time; the output has a valid logic level with extremely high probability. If D is stable during the aperture, Q should take on the same value as D. If D changes during the aperture, Q may take on either a HIGH or LOW value but must not be metastable.

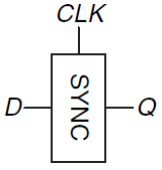


Figure 3. 67 Synchronizer symbol

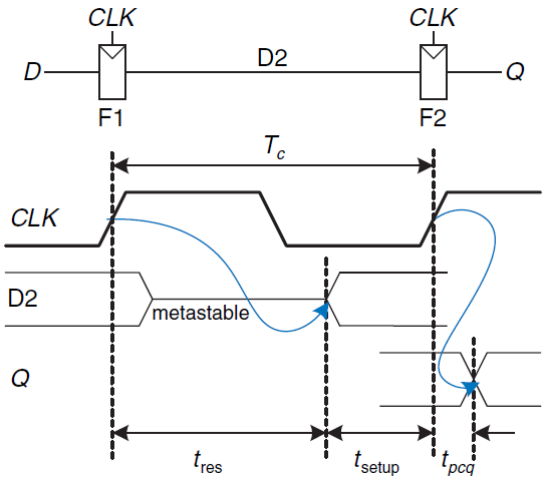


Figure 3. 68 Simple synchronizer

We say that a synchronizer *fails* if Q, the output of the synchronizer, becomes metastable. This may happen if $D2$ has not resolved to a valid level by the time it must setup at F2—that is, if $t_{res} > T_c - t_{\text{setup}}$. According to the equation of the resolution time, the probability of failure for a single input change at a random time is

$$P(\text{failure}) = \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}}$$

If D changes once per second, the probability of failure per second is just $P(\text{failure})$. However, if D changes N times per second, the probability of failure per second is N times as great:

$$P(\text{failure})/\text{sec} = N \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}}$$

System reliability is usually measured in *mean time between failures* (*MTBF*). As the name suggests, MTBF is the average amount of time between failures of the system. It is the reciprocal of the probability that the system will fail in any given second

$$MTBF = \frac{1}{P(\text{failure})/\text{sec}} = \frac{T_c e^{\frac{T_c - t_{\text{setup}}}{\tau}}}{NT_0}$$

The above equation shows that the MTBF improves exponentially as the synchronizer waits for a longer time, T_c.

Example 3.14: SYNCHRONIZER FOR FSM INPUT (Page 154)

Derivation of Resolution Time

(Unfinish, Need circuit knowledge)

3.6 PARALLELISM

The speed of a system is characterized by the latency and throughput of information moving through it. We define a *token* to be a group of inputs that are processed to produce a group of outputs (like an instruction). The *latency* of a system is the time required for one token to pass through the system from start to end (like cycle per instruction). The *throughput* is the number of tokens that can be produced per unit time.

Example 3.15: COOKIE THROUGHPUT AND LATENCY (157)

As you might imagine, the throughput can be improved by processing several tokens at the same time. This is called *parallelism*, and it comes in two forms: spatial and temporal. With *spatial parallelism*, multiple copies of the hardware are provided so that multiple tasks can be done at the same time. With temporal parallelism, a task is broken into stages, like an assembly line. Temporal parallelism is commonly called *pipelining*.

Example 3.16: COOKIE PARALLELISM (Page 158)

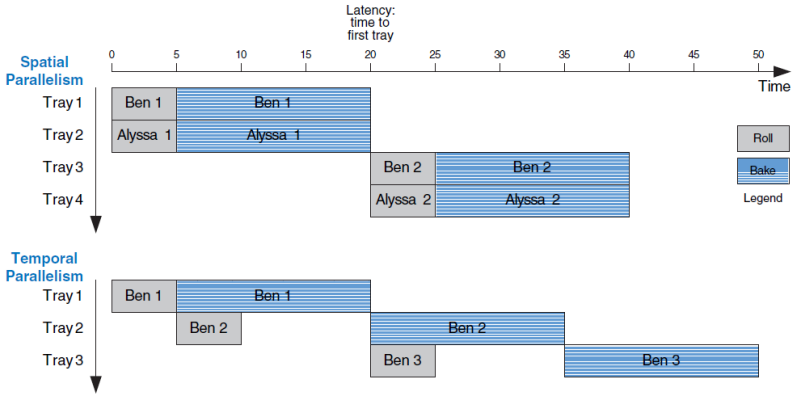


Figure 3. 69 Spatial and temporal parallelism in the cookie kitchen

Consider a task with latency L. In a system with no parallelism, the throughput is $1/L$. In a spatially parallel system with N copies of the hardware, the throughput is N/L. In a temporally parallel system, the task is ideally broken into N steps, or stages, of equal length (L/N). In such a case, the throughput is also N/L, and only one copy of the hardware is required.

Pipelining (temporal parallelism) is particularly attractive because it speeds up a circuit without duplicating the hardware. Instead, registers are placed between blocks of combinational logic to divide the logic into shorter stages that can run with a faster clock. The registers prevent a token in one pipeline stage from catching up with and corrupting the token in the next stage.

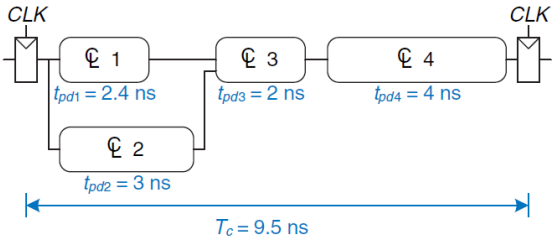


Figure 3. 70 Circuit with no pipelining

For figure 3.70, the latency is 9.5 ns and the throughput of $1/9.5$ ns$= 105$ MHz.

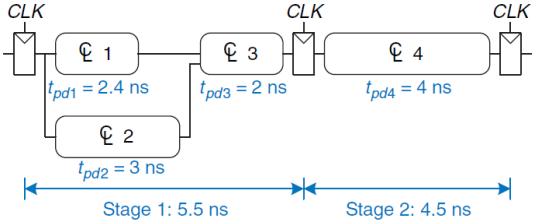


Figure 3. 71 Circuit with two-stage pipeline

For figure 3.71, the latency is 11 ns and the throughput of $1/5.5$ ns$= 182$ MHz.

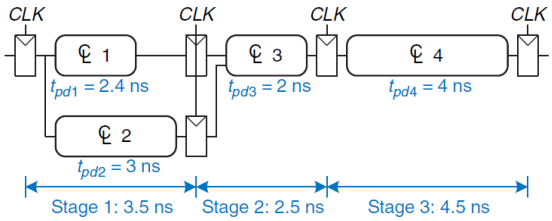


Figure 3. 72 Circuit with three-stage pipeline

For figure 3.72, the latency is 13.5 ns and the throughput of $1/4.5$ ns$= 222$ MHz.

Although these techniques are powerful, they do not apply to all situations. The bane of parallelism is *dependencies*.

Exercises

Exercise 3.7: Is the circuit in Figure 3.73 combinational logic or sequential logic? Explain in a simple fashion what the relationship is between the inputs and outputs. What would you call this circuit?

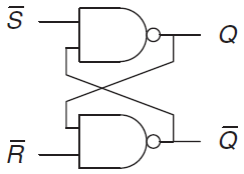


Figure 3. 73 Mystery circuit

The circuit is sequential because it involves feedback and the output depends on previous values of the inputs. This is a SR latch.

Exercise 3.8: Is the circuit in Figure 3.74 combinational logic or sequential logic? Explain in a simple fashion what the relationship is between the inputs and outputs. What would you call this circuit?

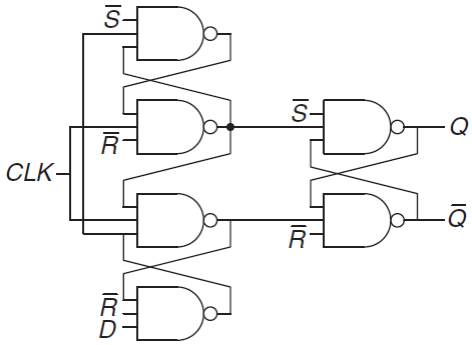
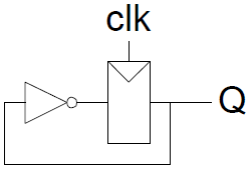


Figure 3. 74 Mystery circuit

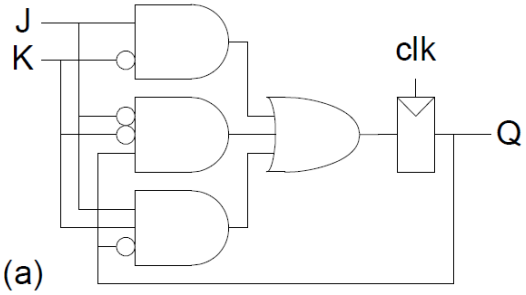
Sequential logic. This is a D flip-flop with active low asynchronous **set** and **reset** inputs. If $\bar{S}$ and $\bar{R}$ are both 1, the circuit behaves as an ordinary D flip-flop. If $\bar{S} = 0$, Q is immediately set to 0. If $\bar{R} = 0$, Q is immediately reset to 1. (This circuit is used in the commercial 7474 flip-flop.)

Exercise 3.9: The ***toggle*** *(T) flip-flop* has one input, CLK, and one output, Q. On each rising edge of CLK, Q toggles to the complement of its previous value. Draw a schematic for a T flip-flop using a D flip-flop and an inverter.



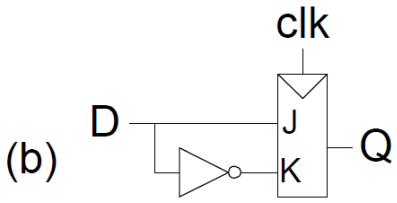
Exercise 3.10: A ***JK*** *flip-flop* receives a clock and two inputs, J and K. On the rising edge of the clock, it updates the output, Q. If J and K are both 0, Q retains its old value. If only J is 1, Q becomes 1. If only K is 1, Q becomes 0. If both J and K are 1, Q becomes the opposite of its present state.

(a) Construct a JK flip-flop using a D flip-flop and some combinational logic.

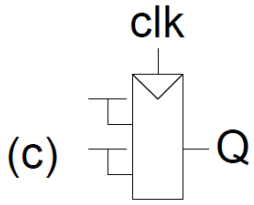


Note: using a $4{:}1$ multiplexer should also work. (or the internal structure of it: AND, OR and inverter)

(b) Construct a D flip-flop using a JK flip-flop and some combinational logic.



(c) Construct a T flip-flop (see Exercise 3.9) using a JK flip-flop.



Exercise 3.11: The circuit in Figure 3.75 is called a *Muller C-element*. Explain in a simple fashion what the relationship is between the inputs and output.

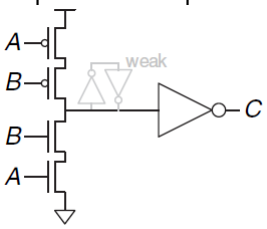
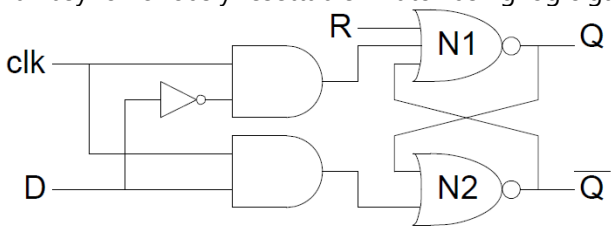


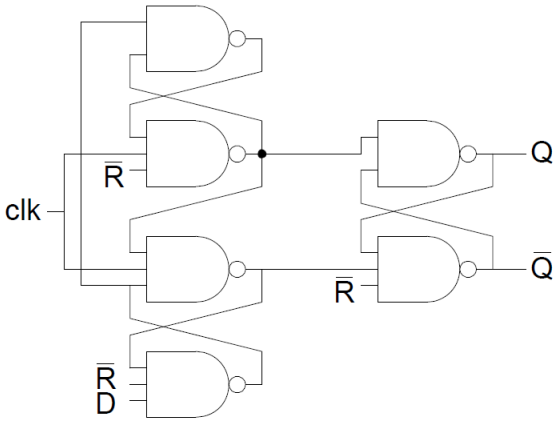
Figure 3. 75 Muller C-element

If A and B have the same value, C takes on that value. Otherwise, C retains its old value.

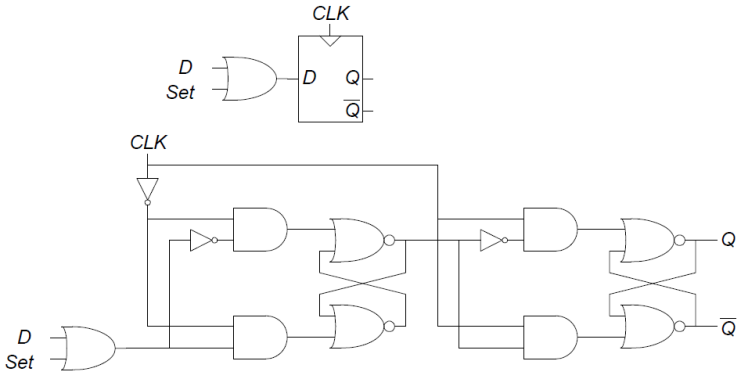
Exercise 3.12: Design an asynchronously resettable D latch using logic gates.



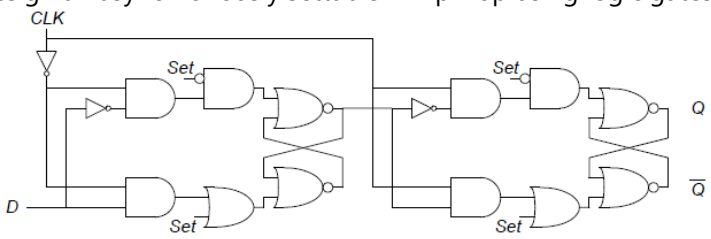
Exercise 3.13: Design an asynchronously resettable D flip-flop using logic gates.



Exercise 3.14: Design a synchronously settable D flip-flop using logic gates.



Exercise 3.15: Design an asynchronously settable D flip-flop using logic gates.



Exercise 3.16: Suppose a ring oscillator is built from N inverters connected in a loop. Each inverter has a minimum delay of t_{cd} and a maximum delay of t_{pd}. If N is odd, determine the range of frequencies at which the oscillator might operate.

From $\frac{1}{2Nt_{pd}}$ to $\frac{1}{2Nt_{cd}}$.

Exercise 3.17: Why must N be odd in Exercise 3.16?

If N is even, the circuit is stable and will not oscillate.

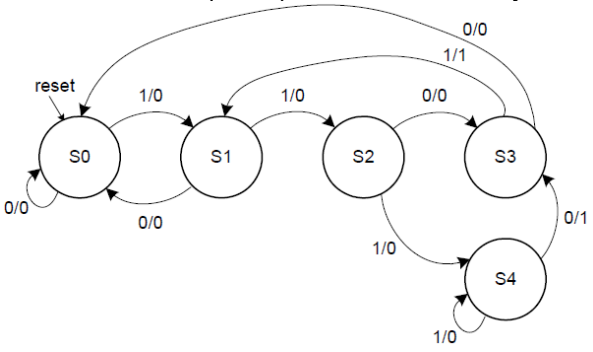
Exercise 3.20: You are designing an FSM to keep track of the mood of four students working in the digital design lab. Each student's mood is either HAPPY (the circuit works), SAD (the circuit blew up), BUSY (working on the circuit), CLUELESS (confused about the circuit), or ASLEEP (face down on the circuit board). How many states does the FSM have? What is the minimum number of bits necessary to represent these states?

The FSM has $5^4 = 625$ states. This requires at least 10 bits to represent all the states.

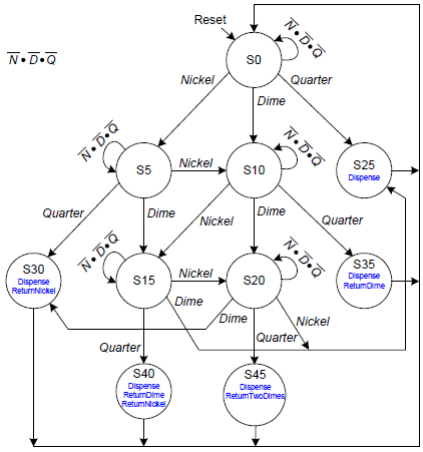
Exercise 3.21: How would you factor the FSM from Exercise 3.20 into multiple simpler machines? How many states does each simpler machine have? What is the minimum total number of bits necessary in this factored design?

The FSM could be factored into four independent state machines, one for each student. Each of these machines has five states and requires 3 bits, so at least 12 bits of state are required for the factored design.

Exercise 3.25: Alyssa P. Hacker's snail from Section 3.4.3 has a daughter with a Mealy machine FSM brain. The daughter snail smiles whenever she slides over the pattern 1101 or the pattern 1110. Sketch the state transition diagram for this happy snail using as few states as possible. Choose state encodings and write a combined state transition and output table using your encodings. Write the next state and output equations and sketch your FSM schematic.



Exercise 3.26: You have been enlisted to design a soda machine dispenser for your department lounge. Sodas are partially subsidized by the student chapter of the IEEE, so they cost only 25 cents. The machine accepts nickels, dimes, and quarters. When enough coins have been inserted, it dispenses the soda and returns any necessary change. Design an FSM controller for the soda machine. The FSM inputs are Nickel, Dime, and Quarter, indicating which coin was inserted. Assume that exactly one coin is inserted on each cycle. The outputs are Dispense, ReturnNickel, ReturnDime, and ReturnTwoDimes. When the FSM reaches 25 cents, it asserts Dispense and the necessary Return outputs required to deliver the appropriate change. Then it should be ready to start accepting coins for another soda.



Exercise 3.27: Gray codes have a useful property in that consecutive numbers differ in only a single bit position. Figure 3.76 lists a 3-bit Gray code representing the numbers 0 to 7. Design a 3-bit modulo 8 Gray code counter FSM with no inputs and three outputs. (A modulo N counter counts from 0 to $N-1$, then repeats. For example, a watch uses a modulo 60 counter for the minutes and seconds that counts from 0 to 59.) When reset, the output should be 000. On each clock edge, the output should advance to the next Gray code. After reaching 100, it should repeat with 000.

Number	Gray code		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

Figure 3. 76 3-bit Gray code

Note: a Gray code is an encoding of numbers so that adjacent numbers have a single digit differing by 1. The term Gray code is often used to refer to a "reflected" code, or more specifically still, the binary reflected Gray code.

The property of Gray code:

- Unweighted code.
- Unit distance code & Minimum error code.

Binary to Gray code conversion

Method 1

Step 1: Record the MSB as it is.

Step 2: Add the MSB to the next bit (binary bit), record the sum and neglect the carry.

Step 3: repeat the process (but a little different, the MSB is changing as shown below).

Example 1: Convert 1011_2 to Gray code.

$$\begin{array}{c} \text{MSB} \rightarrow 1011_2 \leftarrow \text{LSB} \\ \begin{array}{cccc} b_3 & b_2 & b_1 & b_0 \\ 1 & 0 & 1 & 1 \\ \downarrow & \searrow\downarrow & \searrow\downarrow & \searrow\downarrow \\ 1 & 1 & 1 & 0 \\ g_3 & g_2 & g_1 & g_0 \end{array} \end{array}$$

$$g_3 = b_3$$
$$g_2 = b_3 \oplus b_2$$
$$g_1 = b_2 \oplus b_1$$
$$g_0 = b_1 \oplus b_0.$$

Method 2

To convert a binary number $d_1 d_2 \cdots d_{n-1} d_n$ to its corresponding binary reflected Gray code, start at the right with the digit d_n (the nth, or last, digit). If the d_{n-1} is 1, replace d_n by $1 - d_n$; otherwise, leave it unchanged. Then proceed to d_{n-1}. Continue up to the first digit d_1, which is kept the same since d_0 is assumed to be a 0. The resulting number $g_1 g_2 \cdots g_{n-1} g_n$ is the reflected binary Gray code.

Gray code to Binary conversion

Method 1

Step 1: Record the MSB as it is.

Step 2: Add the MSB to the next bit, record the sum and neglect the carry.

Step 3: repeat the process (but a little different, the MSB is changing as shown below).

Example 1: Convert Gray code 1110_2 to binary.

$$\text{MSB} \rightarrow 1110_2 \leftarrow \text{LSB}$$

$$\begin{array}{cccc} g_3 & g_2 & g_1 & g_0 \\ 1 & 1 & 1 & 0 \\ \downarrow & \nearrow\downarrow & \nearrow\downarrow & \nearrow\downarrow \\ 1 & 0 & 1 & 1 \\ b_3 & b_2 & b_1 & b_0 \end{array}$$

$$b_3 = g_3$$
$$b_2 = b_3 \oplus g_2$$
$$b_1 = b_2 \oplus g_1$$
$$b_0 = b_1 \oplus g_0.$$

Method 2

To convert a binary reflected Gray code $g_1 g_2 \cdots g_{n-1} g_n$ to a binary number, start again with the nth digit, and compute

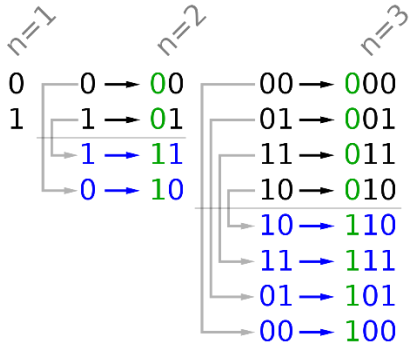
$$\sum\nolimits_n \equiv \sum_{i=1}^{n-1} g_i \pmod 2.$$

If $\sum_n$ is 1, replace g_n by $1 - g_n$; otherwise, leave it the unchanged. Next compute

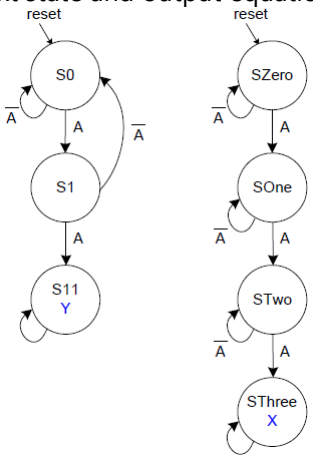
$$\sum\nolimits_{n-1} \equiv \sum_{i=1}^{n-2} g_i \pmod 2,$$

and so on. The resulting number $d_1 d_2 \cdots d_{n-1} d_n$ is the binary number corresponding to the initial binary reflected Gray code.

The first few steps of the reflect-and-prefix method:



Exercise 3.30: Design an FSM with one input, A, and two outputs, X and Y. X should be 1 if A has been 1 for at least three cycles altogether (not necessarily consecutively). Y should be 1 if A has been 1 for at least two consecutive cycles. Show your state transition diagram, encoded state transition table, next state and output equations, and schematic.



4.1 INTRODUCTION

Modules

A block of hardware with inputs and outputs is called a *module*. An AND gate, a multiplexer, and a priority circuit are all examples of hardware modules. The two general styles for describing module functionality are ***behavioral*** and ***structural***. **Behavioral** models describe **what** a module **does**. **Structural** models describe **how** a module is **built** from **simpler** pieces. The SystemVerilog and VHDL code in HDL Example 4.1 illustrate behavioral descriptions of a module that computes the Boolean function from Example 2.6.

Practical circuits use a combination of both.

HDL Example 4.1: COMBINATIONAL LOGIC

SystemVerilog

```
module sillyfunction(input  logic a, b, c,
                     output logic y);

  assign y = ~a & ~b & ~c |
             a & ~b & ~c |
             a & ~b &  c;

endmodule
```

A SystemVerilog module begins with the module name and a listing of the inputs and outputs. The `assign` statement describes combinational logic. `~` indicates NOT, `&` indicates AND, and `|` indicates OR.

`logic` signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating and undefined values, as discussed in Section 4.2.8.

The `logic` type was introduced in SystemVerilog. It supersedes the `reg` type, which was a perennial source of confusion in Verilog. `logic` should be used everywhere except on signals with multiple drivers. Signals with multiple drivers are called *nets* and will be explained in Section 4.7.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sillyfunction is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
  y <= (not a and not b and not c) or
       (a and not b and not c) or
       (a and not b and c);
end;
```

VHDL code has three parts: the `library` use clause, the `entity` declaration, and the `architecture` body. The `library use` clause will be discussed in Section 4.7.2. The `entity` declaration lists the module name and its inputs and outputs. The `architecture` body defines what the module does.

VHDL signals, such as inputs and outputs, must have a *type declaration*. Digital signals should be declared to be `STD_LOGIC` type. `STD_LOGIC` signals can have a value of `'0'` or `'1'`, as well as floating and undefined values that will be described in Section 4.2.8. The `STD_LOGIC` type is defined in the `IEEE.STD_LOGIC_1164` library, which is why the library must be used.

VHDL lacks a good default order of operations between AND and OR, so Boolean equations should be parenthesized.

Language Origins

Industry is trending toward SystemVerilog, but many companies still use VHDL and many designers need to be fluent in both.

Simulation and Synthesis

The two major purposes of HDLs are logic *simulation* and *synthesis*. During **simulation**, inputs are applied to a module, and the outputs are checked to verify that the module operates **correctly**. During **synthesis**, the textual description of a module is transformed into **logic gates**.

Simulation

Humans routinely make mistakes. Such errors in hardware designs are called *bugs*.

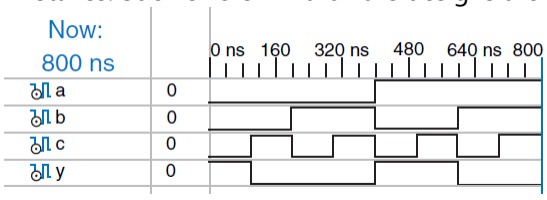


Figure 4. 1 Simulation waveforms

Synthesis

Logic synthesis transforms HDL code into a *netlist* describing the hardware (e.g., the logic gates and the wires connecting them).

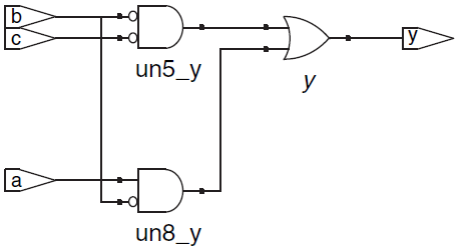


Figure 4. 2 Synthesized circuit

Circuit descriptions in HDL resemble code in a programming language. However, you must remember that the code is intended to represent hardware. SystemVerilog and VHDL are rich languages with many commands. Not all of these commands can be synthesized into hardware. We will emphasize a *synthesizable subset* of the languages. Specifically, we will divide HDL code into ***synthesizable*** modules and a ***testbench***.

4.2 COMBINATIONAL LOGIC

Bitwise Operators

Bitwise operators act on single-bit signals or on multi-bit busses.

HDL Example 4.2: INVERTERS

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

`a[3:0]` represents a 4-bit bus. The bits, from most significant to least significant, are `a[3]`, `a[2]`, `a[1]`, and `a[0]`. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus `a[4:1]`, in which case `a[4]` would have been the most significant. Or we could have used `a[0:3]`, in which case the bits, from most significant to least significant, would be `a[0]`, `a[1]`, `a[2]`, and `a[3]`. This is called *big-endian* order.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

VHDL uses `STD_LOGIC_VECTOR` to indicate busses of `STD_LOGIC`. `STD_LOGIC_VECTOR(3 downto 0)` represents a 4-bit bus. The bits, from most significant to least significant, are `a(3)`, `a(2)`, `a(1)`, and `a(0)`. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have declared the bus to be `STD_LOGIC_VECTOR(4 downto 1)`, in which case bit 4 would have been the most significant. Or we could have written `STD_LOGIC_VECTOR(0 to 3)`, in which case the bits, from most significant to least significant, would be `a(0)`, `a(1)`, `a(2)`, and `a(3)`. This is called *big-endian* order.

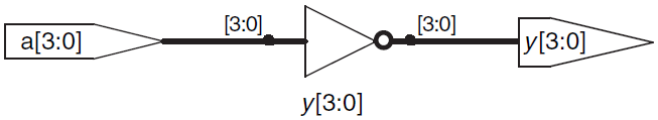


Figure 4. 3 inv synthesized circuit

The endianness of a bus is **purely** arbitrary. Endianness **matters only** for **operators**, such as addition, where the sum of one column carries over into the next. Either ordering is acceptable, as long as it is used consistently.

The gates module in HDL Example 4.3 demonstrates bitwise operations acting on 4-bit busses for other basic logic functions.

HDL Example 4.3: LOGIC GATES

SystemVerilog

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2,
                                y3, y4, y5);

  /* five different two-input logic
     gates acting on 4-bit busses */
  assign y1 = a & b;     // AND
  assign y2 = a | b;     // OR
  assign y3 = a ^ b;     // XOR
  assign y4 = ~(a & b);  // NAND
  assign y5 = ~(a | b);  // NOR
endmodule
```

~, ^, and | are examples of SystemVerilog *operators*, whereas a, b, and y1 are *operands*. A combination of operators and operands, such as a & b, or ~(a | b), is called an *expression*. A complete command such as assign y4 = ~(a & b); is called a *statement*.

assign out = in1 op in2; is called a *continuous assignment statement*. Continuous assignment statements end with a semicolon. Anytime the inputs on the right side of the = in a continuous assignment statement change, the output on the left side is recomputed. Thus, continuous assignment statements describe combinational logic.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
     y1, y2, y3, y4,
     y5:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
  -- five different two-input logic gates
  -- acting on 4-bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

not, xor, and or are examples of VHDL *operators*, whereas a, b, and y1 are *operands*. A combination of operators and operands, such as a and b, or a nor b, is called an *expression*. A complete command such as y4 <= a nand b; is called a *statement*.

out <= in1 op in2; is called a *concurrent signal assignment statement*. VHDL assignment statements end with a semicolon. Anytime the inputs on the right side of the <= in a concurrent signal assignment statement change, the output on the left side is recomputed. Thus, concurrent signal assignment statements describe combinational logic.

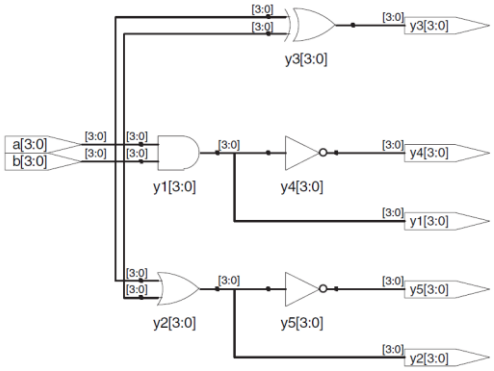


Figure 4. 4 gates synthesized circuit

Comments and White Space

SystemVerilog and VHDL are not picky about the use of white space (i.e., spaces, tabs, and line breaks). Module and signal names **must not** begin with a digit.

SystemVerilog

SystemVerilog comments are just like those in C or Java. Comments beginning with /* continue, possibly across multiple lines, to the next */. Comments beginning with // continue to the end of the line.

SystemVerilog is case-sensitive. y1 and Y1 are different signals in SystemVerilog. However, it is confusing to use multiple signals that differ only in case.

VHDL

Comments beginning with /* continue, possibly across multiple lines, to the next */. Comments beginning with -- continue to the end of the line.

VHDL is not case-sensitive. y1 and Y1 are the same signal in VHDL. However, other tools that may read your file might be case sensitive, leading to nasty bugs if you blithely mix upper and lower case.

Reduction Operators

Reduction operators imply a multiple-input gate acting on a single bus. HDL Example 4.4 describes an eight-input AND gate with inputs $a_7, a_6, \cdots, a_0$. Analogous reduction operators exist for OR, XOR, NAND, NOR, and XNOR gates. **Recall** that a multiple-input XOR performs parity, returning TRUE if an **odd** number of inputs are TRUE.

HDL Example 4.4: EIGHT-INPUT AND

SystemVerilog

```
module and8(input  logic [7:0] a,
            output logic       y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and8 is
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of and8 is
begin
  y <= and a;
  -- and a is much easier to write than
  -- y <= a(7) and a(6) and a(5) and a(4) and
  --      a(3) and a(2) and a(1) and a(0);
end;
```

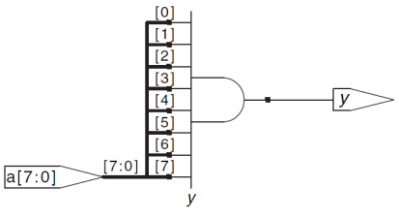


Figure 4. 5 and8 synthesized circuit

Conditional Assignment

Conditional assignments select the output from among alternatives based on an input called the ***condition***. HDL Example 4.5 illustrates a 2:1 multiplexer using conditional assignment.

HDL Example 4.5: 2:1 MULTIPLEXER

SystemVerilog

The *conditional operator* `?:` chooses, based on a first expression, between a second and third expression. The first expression is called the *condition*. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression.

`?:` is especially useful for describing a multiplexer because, based on the first input, it selects between two others. The following code demonstrates the idiom for a 2:1 multiplexer with 4-bit inputs and outputs using the conditional operator.

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```

If `s` is 1, then `y = d1`. If `s` is 0, then `y = d0`.

`?:` is also called a *ternary operator*, because it takes three inputs. It is used for the same purpose in the C and Java programming languages.

VHDL

Conditional signal assignments perform different operations depending on some condition. They are especially useful for describing a multiplexer. For example, a 2:1 multiplexer can use conditional signal assignment to select one of two 4-bit inputs.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

The conditional signal assignment sets `y` to `d1` if `s` is 1. Otherwise it sets `y` to `d0`. Note that prior to the 2008 revision of VHDL, one had to write `when s = '1'` rather than `when s`.

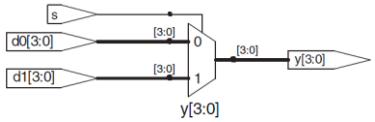


Figure 4. 6 mux2 synthesized circuit

HDL Example 4.6: 4:1 MULTIPLEXER

SystemVerilog

A 4:1 multiplexer can select one of four inputs using nested conditional operators.

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

If `s[1]` is 1, then the multiplexer chooses the first expression, `(s[0] ? d3 : d2)`. This expression in turn chooses either `d3` or `d2` based on `s[0]` (`y = d3` if `s[0]` is 1 and `d2` if `s[0]` is 0). If `s[1]` is 0, then the multiplexer similarly chooses the second expression, which gives either `d1` or `d0` based on `s[0]`.

VHDL

A 4:1 multiplexer can select one of four inputs using multiple `else` clauses in the conditional signal assignment.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
       d1 when s = "01" else
       d2 when s = "10" else
       d3;
end;
```

VHDL also supports *selected signal assignment statements* to provide a shorthand when selecting from one of several possibilities. This is analogous to using a `switch/case` statement in place of multiple `if/else` statements in some programming languages. The 4:1 multiplexer can be rewritten with selected signal assignment as follows:

```
architecture synth2 of mux4 is
begin
  with s select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```

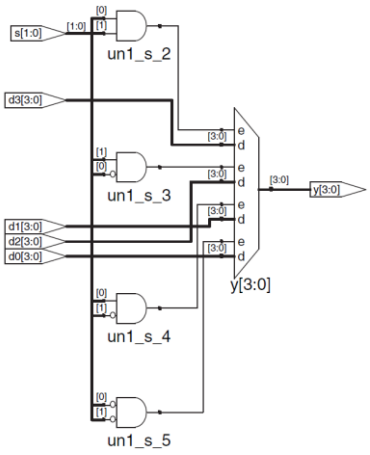


Figure 4. 7 mux4 synthesized circuit

Internal Variables

Often it is **convenient** to break a complex function into intermediate steps.

HDL Example 4.7: FULL ADDER

SystemVerilog

In SystemVerilog, internal signals are usually declared as `logic`.

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

VHDL

In VHDL, *signals* are used to represent internal variables whose values are defined by *concurrent signal assignment statements* such as `p <= a xor b;`

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;

  s <= p xor cin;
  cout <= g or (p and cin);
end;
```

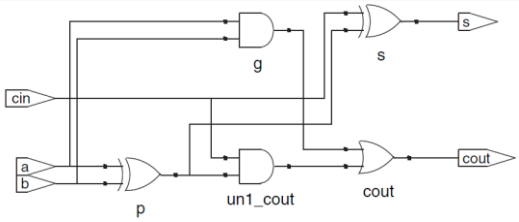


Figure 4. 8 fulladder synthesized circuit

HDL assignment statements (**assign** in SystemVerilog and <= in VHDL) take place concurrently. This is different from conventional programming languages such as C or Java, in which statements are evaluated in the order in which they are written.

Precedence

HDL Example 4.8: OPERATOR PRECEDENCE (Page 185)

	Op	Meaning
Highest	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
Lowest	|, ~|	OR, NOR
	?:	Conditional

Figure 4. 9 SystemVerilog operator precedence

	Op	Meaning
Highest	not	NOT
	*, /, mod, rem	MUL, DIV, MOD, REM
	+, -	PLUS, MINUS
	rol, ror, srl, sll	Rotate, Shift logical
	<, <=, >, >=	Relative Comparison
Lowest	=, /=	Equality Comparison
	and, or, nand, nor, xor, xnor	Logical Operations

Figure 4. 10 VHDL operator precedence

Numbers

Numbers can be specified in binary, octal, decimal, or hexadecimal (bases 2, 8, 10, and 16, respectively). The size, i.e., the number of bits, may optionally be given, and leading zeros are inserted to reach this size. Underscores in numbers are ignored and can be helpful in breaking long numbers into more readable chunks. HDL Example 4.9 explains how numbers are written in each language.

HDL Example 4.9: NUMBERS

SystemVerilog

The format for declaring constants is `N'Bvalue`, where `N` is the size in bits, `B` is a letter indicating the base, and `value` gives the value. For example, `9'h25` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. SystemVerilog supports `'b` for binary, `'o` for octal, `'d` for decimal, and `'h` for hexadecimal. If the base is omitted, it defaults to decimal.

If the size is not given, the number is assumed to have as many bits as the expression in which it is being used. Zeros are automatically padded on the front of the number to bring it up to full size. For example, if `w` is a 6-bit bus, `assign w = 'b11` gives `w` the value 000011. It is better practice to explicitly give the size. An exception is that `'0` and `'1` are SystemVerilog idioms for filling a bus with all 0s and all 1s, respectively.

Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000 ... 0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00 ... 0101010

Figure 4. 11 SystemVerilog numbers

Numbers	Bits	Base	Val	Stored
3B"101"	3	2	5	101
B"11"	2	2	3	11
8B"11"	8	2	3	00000011
8B"1010_1011"	8	2	171	10101011
3D"6"	3	10	6	110
6O"42"	6	8	34	100010
8X"AB"	8	16	171	10101011
"101"	3	2	5	101
B"101"	3	2	5	101
X"AB"	8	16	171	10101011

Figure 4. 12 VHDL numbers

Z's and X's

HDLs use **z** to indicate a **floating** value, z is particularly useful for describing a tristate buffer, whose output floats when the enable is 0.

Similarly, HDLs use **x** to indicate an **invalid** logic level.

At the start of simulation, state nodes such as flip-flop outputs are initialized to an unknown state (x in SystemVerilog and u in VHDL). This is helpful to track errors caused by forgetting to

reset a flip-flop before its output is used.

HDL Example 4.10: TRISTATE BUFFER

SystemVerilog

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

Notice that y is declared as tri rather than logic. logic signals can only have a single driver. Tristate busses can have multiple drivers, so they should be declared as a *net*. Two types of nets in SystemVerilog are called tri and trireg. Typically, exactly one driver on a net is active at a time, and the net takes on that value. If no driver is active, a tri floats (z), while a trireg retains the previous value. If no type is specified for an input or output, tri is assumed. Also note that a tri output from a module can be used as a logic input to another module. Section 4.7 further discusses nets with multiple drivers.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity tristate is
  port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
       en: in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of tristate is
begin
  y <= a when en else "ZZZZ";
end;
```

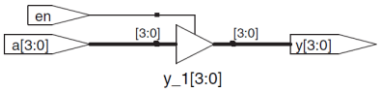


Figure 4. 13 tristate synthesized circuit

If a gate receives a floating input, it may produce an x output when it can't determine the correct output value. Similarly, if it receives an illegal or uninitialized input, it may produce an x output.

HDL Example 4.11: TRUTH TABLES WITH UNDEFINED AND FLOATING INPUTS (Page 187)

Bit Swizzling

Often it is necessary to operate on a subset of a bus or to concatenate (join together) signals to form busses. These operations are collectively known as *bit swizzling*. In HDL Example 4.12, y is given the 9-bit value $c_2c_1d_0d_0d_0c_0101$ using bit swizzling operations.

HDL Example 4.12: BIT SWIZZLING

SystemVerilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

The {} operator is used to concatenate busses. {3{d[0]}} indicates three copies of d[0].

Don't confuse the 3-bit binary constant 3'b101 with a bus named b. Note that it was critical to specify the length of 3 bits in the constant; otherwise, it would have had an unknown number of leading zeros that might appear in the middle of y.

If y were wider than 9 bits, zeros would be placed in the most significant bits.

VHDL

```
y <=(c(2 downto 1), d(0), d(0), d(0), c(0), 3B"101");
```

The () aggregate operator is used to concatenate busses. y must be a 9-bit STD_LOGIC_VECTOR.

Another example demonstrates the power of VHDL aggregations. Assuming z is an 8-bit STD_LOGIC_VECTOR, z is given the value 10010110 using the following command aggregation.

```
z <= ("10", 4 => '1', 2 downto 1 =>'1', others =>'0')
```

The "10" goes in the leading pair of bits. 1s are also placed into bit 4 and bits 2 and 1. The other bits are 0.

Delays

HDL statements may be associated with delays specified in arbitrary units. They are helpful during simulation to predict how fast a circuit will work (if you specify meaningful delays) and also for debugging purposes to understand cause and effect. These delays are ignored during synthesis; the delay of a gate produced by the synthesizer depends on its t_{pd} and t_{cd} specifications, not on numbers in HDL code.

HDL Example 4.13: LOGIC GATES WITH DELAYS

SystemVerilog

```
`timescale 1ns/1ps

module example(input  logic a, b, c,
               output logic y);

  logic ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

SystemVerilog files can include a timescale directive that indicates the value of each time unit. The statement is of the form `timescale unit/precision. In this file, each unit is 1 ns, and the simulation has 1 ps precision. If no timescale directive is given in the file, a default unit and precision (usually 1 ns for both) are used. In SystemVerilog, a # symbol is used to indicate the number of units of delay. It can be placed in assign statements, as well as non-blocking (<=) and blocking (=) assignments, which will be discussed in Section 4.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity example is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of example is
  signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
  n1 <= ab and bb and cb after 2 ns;
  n2 <= a and bb and cb after 2 ns;
  n3 <= a and bb and c after 2 ns;
  y  <= n1 or n2 or n3 after 4 ns;
end;
```

In VHDL, the after clause is used to indicate delay. The units, in this case, are specified as nanoseconds.

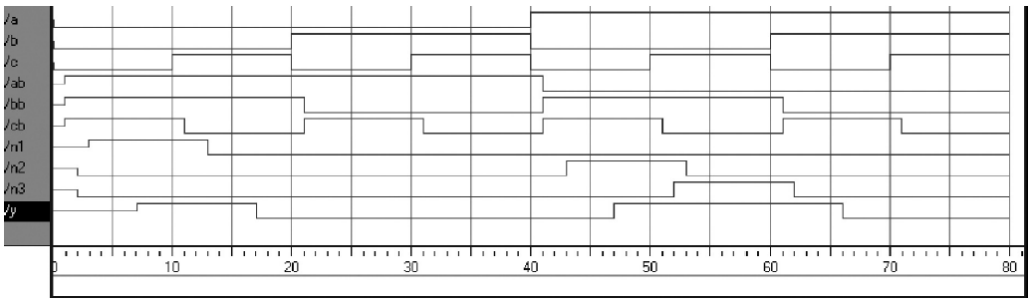


Figure 4. 14 Example simulation waveforms with delays (from the ModelSim simulator)

4.3 STRUCTURAL MODELING

The **previous** section discussed *behavioral* modeling, describing a module in terms of the **relationships** between inputs and outputs. This section examines structural modeling, describing a module in terms of **how** it is composed of simpler modules.

For example, HDL Example 4.14 shows how to assemble a 4:1 multiplexer from three 2:1 multiplexers. Each copy of the 2:1 multiplexer is called an *instance*. Multiple instances of the same module are distinguished by distinct names.

HDL Example 4.14: STRUCTURAL MODEL OF 4:1 MULTIPLEXER

SystemVerilog

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  logic [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

The three `mux2` instances are called `lowmux`, `highmux`, and `finalmux`. The `mux2` module must be defined elsewhere in the SystemVerilog code — see HDL Example 4.5, 4.15, or 4.34.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
  component mux2
    port(d0,
         d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
  lowmux:   mux2 port map(d0, d1, s(0), low);
  highmux:  mux2 port map(d2, d3, s(0), high);
  finalmux: mux2 port map(low, high, s(1), y);
end;
```

The architecture must first declare the `mux2` ports using the `component` declaration statement. This allows VHDL tools to check that the component you wish to use has the same ports as the entity that was declared somewhere else in another entity statement, preventing errors caused by changing the entity but not the instance. However, component declaration makes VHDL code rather cumbersome.

Note that this architecture of `mux4` was named `struct`, whereas architectures of modules with behavioral descriptions from Section 4.2 were named `synth`. VHDL allows multiple architectures (implementations) for the same entity; the architectures are distinguished by name. The names themselves have no significance to the CAD tools, but `struct` and `synth` are common. Synthesizable VHDL code generally contains only one architecture for each entity, so we will not discuss the VHDL syntax to configure which architecture is used when multiple architectures are defined.

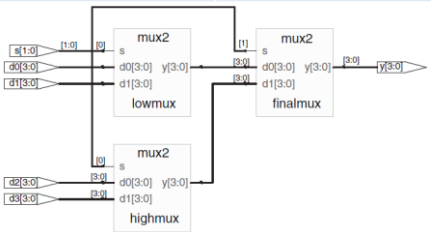


Figure 4. 15 mux4 synthesized circuit

HDL Example 4.15: STRUCTURAL MODEL OF 2:1 MULTIPLEXER

SystemVerilog

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

In SystemVerilog, expressions such as `~s` are permitted in the port list for an instance. Arbitrarily complicated expressions are legal but discouraged because they make the code difficult to read.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal sbar: STD_LOGIC;
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

In VHDL, expressions such as `not s` are not permitted in the port map for an instance. Thus, `sbar` must be defined as a separate signal.

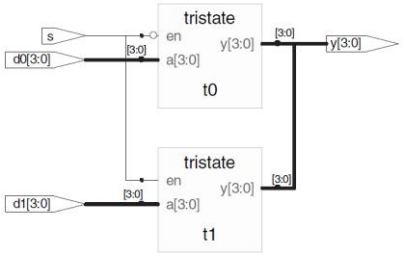


Figure 4. 16 mux2 synthesized circuit

In general, complex systems are designed *hierarchically*.

HDL Example 4.16: ACCESSING PARTS OF BUSSES

SystemVerilog

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2_8 is
  port(d0, d1: in  STD_LOGIC_VECTOR(7 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(3 downto 0));
  end component;
begin
  lsbmux: mux2
    port map(d0(3 downto 0), d1(3 downto 0),
             s, y(3 downto 0));
  msbmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
             s, y(7 downto 4));
end;
```

4.4 SEQUENTIAL LOGIC

HDL synthesizers recognize certain idioms and turn them into specific sequential circuits.

Registers

In SystemVerilog **always** statements and VHDL process statements, signals **keep** their old value until an event in the sensitivity list takes place that explicitly causes them to change. Hence, such code, with appropriate sensitivity lists, can be used to describe sequential circuits with memory.

HDL Example 4.17: REGISTER

SystemVerilog

```
module flop(input  logic      clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule
```

In general, a SystemVerilog `always` statement is written in the form

```
always @(sensitivity list)
  statement;
```

The `statement` is executed only when the event specified in the `sensitivity list` occurs. In this example, the statement is `q <= d` (pronounced "q gets d"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`. Note that sensitivity lists are also referred to as stimulus lists.

`<=` is called a *nonblocking assignment*. Think of it as a regular = sign for now; we'll return to the more subtle points in Section 4.5.4. Note that `<=` is used instead of `assign` inside an `always` statement.

As will be seen in subsequent sections, `always` statements can be used to imply flip-flops, latches, or combinational logic, depending on the sensitivity list and statement. Because of this flexibility, it is easy to produce the wrong hardware inadvertently. SystemVerilog introduces `always_ff`, `always_latch`, and `always_comb` to reduce the risk of common errors. `always_ff` behaves like `always` but is used exclusively to imply flip-flops and allows tools to produce a warning if anything else is implied.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flop is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of flop is
begin
  process(clk) begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;
```

A VHDL `process` is written in the form

```
process(sensitivity list) begin
  statement;
end process;
```

The `statement` is executed when any of the variables in the `sensitivity list` change. In this example, the `if` statement checks if the change was a rising edge on `clk`. If so, then `q <= d` (pronounced "q gets d"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`.

An alternative VHDL idiom for a flip-flop is

```
process(clk) begin
  if clk'event and clk = '1' then
    q <= d;
  end if;
end process;
```

`rising_edge(clk)` is synonymous with `clk'event and clk = '1'`.

Figure 4. 17 flop synthesized circuit

Resettable Registers

Note that distinguishing synchronous and asynchronous reset in a schematic can be difficult. The schematic produced by Synplify Premier places asynchronous reset at the bottom of a flip-flop and synchronous reset on the left side.

HDL Example 4.18: RESETTABLE REGISTER

SystemVerilog

```
module flopr(input  logic      clk,
             input  logic      reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr(input  logic      clk,
             input  logic      reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset)  q <= 4'b0;
    else        q <= d;
endmodule
```

Multiple signals in an `always` statement sensitivity list are separated with a comma or the word `or`. Notice that `posedge reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

Because the modules have the same name, `flopr`, you may include only one or the other in your design.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synchronous of flopr is
begin
  process(clk) begin
    if rising_edge(clk) then
      if reset then q <= "0000";
      else q <= d;
      end if;
    end if;
  end process;
end;
```

Multiple signals in a `process` sensitivity list are separated with a comma. Notice that `reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

Recall that the state of a flop is initialized to `'u'` at startup during VHDL simulation.

As mentioned earlier, the name of the architecture (`asynchronous` or `synchronous`, in this example) is ignored by the VHDL tools but may be helpful to the human reading the code. Because both architectures describe the entity `flopr`, you may include only one or the other in your design.

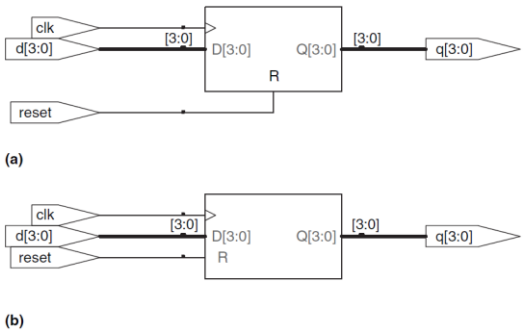


Figure 4. 18 flopr synthesized circuit (a) asynchronous reset, (b) synchronous reset

Enabled Registers

HDL Example 4.19: RESETTABLE ENABLED REGISTER

SystemVerilog

```
module flopenr(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopenr is
  port(clk,
       reset,
       en: in  STD_LOGIC;
       d:  in  STD_LOGIC_VECTOR(3 downto 0);
       q:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopenr is
-- asynchronous reset
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      if en then
        q <= d;
      end if;
    end if;
  end process;
end;
```

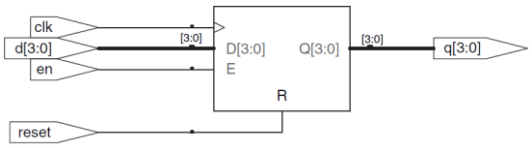


Figure 4. 19 flopenr synthesized circuit

Multiple Registers

A single always/process statement can be used to describe multiple pieces of hardware.

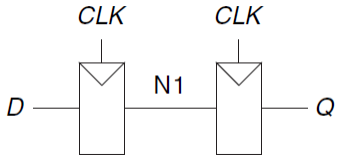


Figure 4. 20 Synchronizer circuit

HDL Example 4.20: SYNCHRONIZER

SystemVerilog

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d; // nonblocking
      q <= n1; // nonblocking
    end
endmodule
```

Notice that the `begin/end` construct is necessary because multiple statements appear in the `always` statement. This is analogous to `{}` in C or Java. The `begin/end` was not needed in the `flopr` example because `if/else` counts as a single statement.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sync is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture good of sync is
  signal n1: STD_LOGIC;
begin
  process(clk) begin
    if rising_edge(clk) then
      n1 <= d;
      q <= n1;
    end if;
  end process;
end;
```

`n1` must be declared as a `signal` because it is an internal signal used in the module.

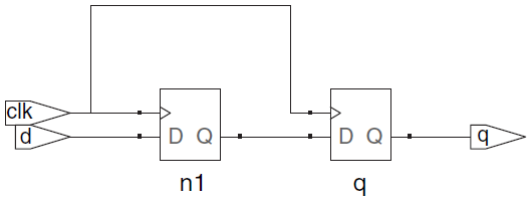


Figure 4. 21 sync synthesized circuit

Latches

Not all synthesis tools support latches well.

HDL Example 4.21: D LATCH

SystemVerilog

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;
endmodule
```

always_latch is equivalent to always @(clk, d) and is the preferred idiom for describing a latch in SystemVerilog. It evaluates any time clk or d changes. If clk is HIGH, d flows through to q, so this code describes a positive level sensitive latch. Otherwise, q keeps its old value. SystemVerilog can generate a warning if the always_latch block doesn't imply a latch.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity latch is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of latch is
begin
  process(clk, d) begin
    if clk='1' then
      q <= d;
    end if;
  end process;
end;
```

The sensitivity list contains both clk and d, so the process evaluates anytime clk or d changes. If clk is HIGH, d flows through to q.

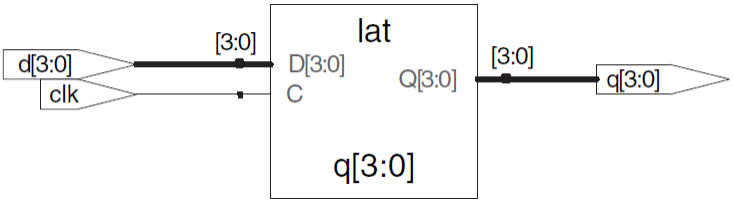


Figure 4. 22 latch synthesized circuit

4.5 MORE COMBINATIONAL LOGIC

In Section 4.2, we used assignment statements to describe combinational logic behaviorally. SystemVerilog **always** statements and VHDL **process** statements are used to describe sequential circuits, because they remember the **old** state when no new state is prescribed. However, **always/process** statements can also be used to describe combinational logic behaviorally if the sensitivity list is written to respond to changes in **all** of the inputs and the body prescribes the output value for **every** possible input combination.

HDLs support *blocking* and *nonblocking assignments* in an **always/process** statement. A **group** of blocking assignments are evaluated in the **order** in which they appear in the code, just as one would expect in a standard programming language. A **group** of nonblocking assignments are evaluated **concurrently**; all of the statements are evaluated before any of the signals on the left hand sides are updated.

HDL Example 4.22: INVERTER USING always/process

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  always_comb
    y=~a;
endmodule
```

always_comb reevaluates the statements inside the always statement any time any of the signals on the right hand side of <= or = in the always statement change. In this case, it is equivalent to always @(a), but is better because it avoids mistakes if signals in the always statement are renamed or added. If the code inside the always block is not combinational logic, SystemVerilog will report a warning. always_comb is equivalent to always @(*), but is preferred in SystemVerilog.

The = in the always statement is called a *blocking assignment*, in contrast to the <= nonblocking assignment. In SystemVerilog, it is good practice to use blocking assignments for combinational logic and nonblocking assignments for sequential logic. This will be discussed further in Section 4.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture proc of inv is
begin
  process(all) begin
    y <= not a;
  end process;
end;
```

process(all) reevaluates the statements inside the process any time any of the signals in the process change. It is equivalent to process(a) but is better because it avoids mistakes if signals in the process are renamed or added.

The begin and end process statements are required in VHDL even though the process contains only one assignment.

SystemVerilog

In a SystemVerilog `always` statement, = indicates a blocking assignment and <= indicates a nonblocking assignment (also called a concurrent assignment).

Do not confuse either type with continuous assignment using the `assign` statement. `assign` statements must be used outside `always` statements and are also evaluated concurrently.

VHDL

In a VHDL `process` statement, := indicates a blocking assignment and <= indicates a nonblocking assignment (also called a concurrent assignment). This is the first section where := is introduced.

Nonblocking assignments are made to outputs and to signals. Blocking assignments are made to variables, which are declared in `process` statements (see HDL Example 4.23). <= can also appear outside `process` statements, where it is also evaluated concurrently.

HDL Example 4.23: FULL ADDER USING always/process

SystemVerilog

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  always_comb
    begin
      p = a ^ b;              // blocking
      g = a & b;              // blocking

      s = p ^ cin;            // blocking
      cout = g | (p & cin);   // blocking
    end
endmodule
```

In this case, `always @(a, b, cin)` would have been equivalent to `always_comb`. However, `always_comb` is better because it avoids common mistakes of missing signals in the sensitivity list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for combinational logic. This example uses blocking assignments, first computing `p`, then `g`, then `s`, and finally `cout`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
begin
  process(all)
    variable p, g: STD_LOGIC;
  begin
    p := a xor b; -- blocking
    g := a and b; -- blocking
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

In this case, `process(a, b, cin)` would have been equivalent to `process(all)`. However, `process(all)` is better because it avoids common mistakes of missing signals in the sensitivity list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for intermediate variables in combinational logic. This example uses blocking assignments for `p` and `g` so that they get their new values before being used to compute `s` and `cout` that depend on them.

Because `p` and `g` appear on the left hand side of a blocking assignment (:=) in a `process` statement, they must be declared to be `variable` rather than `signal`. The variable declaration appears before the `begin` in the process where the variable is used.

case and if statements are convenient for modeling more complicated combinational logic. **case** and **if** statements must appear within **always/process** statements.

Case Statements

A better application of using the **always/process** statement for combinational logic is a seven-segment display decoder that takes advantage of the **case** statement that must appear inside an **always/process** statement.

The **case** statement performs different actions depending on the value of its input. A **case** statement implies **combinational** logic if all possible input combinations are defined; otherwise it implies **sequential** logic, because the output will keep its **old** value in the undefined cases.

HDL Example 4.24: SEVEN-SEGMENT DISPLAY DECODER

SystemVerilog

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case(data)
      //                    abc_defg
      0:       segments = 7'b111_1110;
      1:       segments = 7'b011_0000;
      2:       segments = 7'b110_1101;
      3:       segments = 7'b111_1001;
      4:       segments = 7'b011_0011;
      5:       segments = 7'b101_1011;
      6:       segments = 7'b101_1111;
      7:       segments = 7'b111_0000;
      8:       segments = 7'b111_1111;
      9:       segments = 7'b111_0011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the colon, setting `segments` to 1111110. The `case` statement similarly checks other data values up to 9 (note the use of the default base, base 10).

The `default` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

In SystemVerilog, `case` statements must appear inside `always` statements.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity seven_seg_decoder is
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
begin
  process(all) begin
    case data is
      --                        abcdefg
      when X"0" =>  segments <= "1111110";
      when X"1" =>  segments <= "0110000";
      when X"2" =>  segments <= "1101101";
      when X"3" =>  segments <= "1111001";
      when X"4" =>  segments <= "0110011";
      when X"5" =>  segments <= "1011011";
      when X"6" =>  segments <= "1011111";
      when X"7" =>  segments <= "1110000";
      when X"8" =>  segments <= "1111111";
      when X"9" =>  segments <= "1110011";
      when others => segments <= "0000000";
    end case;
  end process;
end;
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the =>, setting `segments` to 1111110. The `case` statement similarly checks other data values up to 9 (note the use of `X` for hexadecimal numbers). The `others` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

Unlike SystemVerilog, VHDL supports selected signal assignment statements (see HDL Example 4.6), which are much like `case` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic.

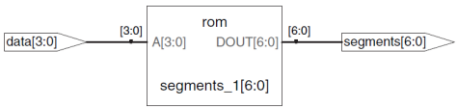


Figure 4. 23 sevenseg synthesized circuit

HDL Example 4.25: 3:8 DECODER

SystemVerilog

```
module decoder3_8(input  logic [2:0] a,
                  output logic [7:0] y);
  always_comb
    case(a)
      3'b000:  y = 8'b00000001;
      3'b001:  y = 8'b00000010;
      3'b010:  y = 8'b00000100;
      3'b011:  y = 8'b00001000;
      3'b100:  y = 8'b00010000;
      3'b101:  y = 8'b00100000;
      3'b110:  y = 8'b01000000;
      3'b111:  y = 8'b10000000;
      default: y = 8'bxxxxxxxx;
    endcase
endmodule
```

The `default` statement isn't strictly necessary for logic synthesis in this case because all possible input combinations are defined, but it is prudent for simulation in case one of the inputs is an `x` or `z`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity decoder3_8 is
  port(a: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of decoder3_8 is
begin
  process(all) begin
    case a is
      when "000" =>  y <= "00000001";
      when "001" =>  y <= "00000010";
      when "010" =>  y <= "00000100";
      when "011" =>  y <= "00001000";
      when "100" =>  y <= "00010000";
      when "101" =>  y <= "00100000";
      when "110" =>  y <= "01000000";
      when "111" =>  y <= "10000000";
      when others => y <= "XXXXXXXX";
    end case;
  end process;
end;
```

The `others` clause isn't strictly necessary for logic synthesis in this case because all possible input combinations are defined, but it is prudent for simulation in case one of the inputs is an `x`, `z`, or `u`.

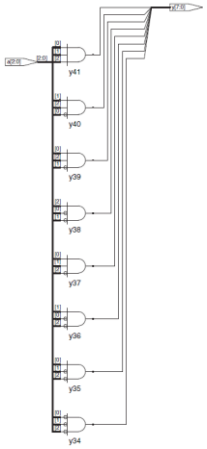


Figure 4. 24 decoder3_8 synthesized circuit

If Statements

always/process statements may also contain **if** statements. The **if** statement may be followed

by an **else** statement. If **all** possible input combinations are handled, the statement implies **combinational** logic; otherwise, it produces **sequential** logic.

HDL Example 4.26: PRIORITY CIRCUIT

SystemVerilog

```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

In SystemVerilog, `if` statements must appear inside of `always` statements.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priorityckt is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priorityckt is
begin
  process(all) begin
    if     a(3) then y <= "1000";
    elsif a(2) then y <= "0100";
    elsif a(1) then y <= "0010";
    elsif a(0) then y <= "0001";
    else            y <= "0000";
    end if;
  end process;
end;
```

Unlike SystemVerilog, VHDL supports conditional signal assignment statements (see HDL Example 4.6), which are much like `if` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic.

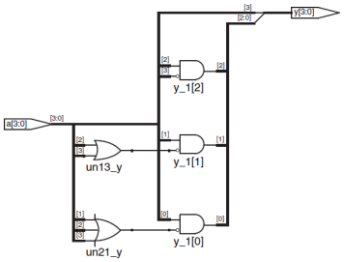


Figure 4. 25 priorityckt synthesized circuit

Truth Tables with Don't Cares

HDL Example 4.27 shows how to describe a priority circuit with **don't cares**.

HDL Example 4.27: PRIORITY CIRCUIT USING DON'T CARES

SystemVerilog

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y = 4'b1000;
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
    endcase
endmodule
```

The `casez` statement acts like a `case` statement except that it also recognizes `?` as don't care.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priority_casez is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture dontcare of priority_casez is
begin
  process(all) begin
    case? a is
      when "1---" => y <= "1000";
      when "01--" => y <= "0100";
      when "001-" => y <= "0010";
      when "0001" => y <= "0001";
      when others => y <= "0000";
    end case?;
  end process;
end;
```

The `case?` statement acts like a `case` statement except that it also recognizes `-` as don't care.

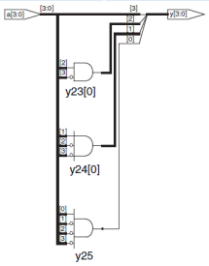


Figure 4. 26 priority_casez synthesized circuit

Blocking and Nonblocking Assignments

The guidelines below explain when and how to use each type of assignment. If these guidelines are not followed, it is possible to write code that appears to work in simulation but synthesizes to incorrect hardware.

BLOCKING AND NONBLOCKING ASSIGNMENT GUIDELINES

SystemVerilog

1. Use `always_ff @(posedge clk)` and nonblocking assignments to model synchronous sequential logic.

```
always_ff @(posedge clk)
  begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
  end
```

2. Use continuous assignments to model simple combinational logic.

```
assign y = s ? d1 : d0;
```

3. Use `always_comb` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.

```
always_comb
  begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

4. Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

VHDL

1. Use `process(clk)` and nonblocking assignments to model synchronous sequential logic.

```
process(clk) begin
  if rising_edge(clk) then
    n1 <= d; -- nonblocking
    q <= n1; -- nonblocking
  end if;
end process;
```

2. Use concurrent assignments outside `process` statements to model simple combinational logic.

```
y <= d0 when s = '0' else d1;
```

3. Use `process(all)` to model more complicated combinational logic where the `process` is helpful. Use blocking assignments for internal variables.

```
process(all)
  variable p, g: STD_LOGIC;
begin
  p := a xor b; -- blocking
  g := a and b; -- blocking
  s <= p xor cin;
  cout <= g or (p and cin);
end process;
```

4. Do not make assignments to the same variable in more than one `process` or concurrent assignment statement.

Combinational Logic

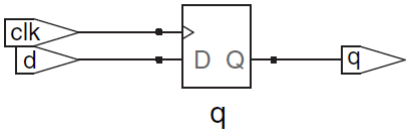
This section explores how it operates and how it would differ if nonblocking assignments had been used. (Page 207)

HDL Example 4.28: FULL ADDER USING NONBLOCKING ASSIGNMENTS (Page 207)

SystemVerilog	VHDL
If the sensitivity list of the `always` statement in HDL Example 4.28 were written as `always @(a, b, cin)` rather than `always_comb`, then the statement would not reevaluate when `p` or `g` changes. In that case, `s` would be incorrectly left at 0, not 1.	If the sensitivity list of the `process` statement in HDL Example 4.28 were written as `process(a, b, cin)` rather than `process(all)`, then the statement would not reevaluate when `p` or `g` changes. In that case, `s` would be incorrectly left at 0, not 1.

Sequential Logic

HDL Example 4.29: BAD SYNCHRONIZER WITH BLOCKING ASSIGNMENTS (Page 209)



q

Figure 4. 27 syncbad synthesized circuit

The moral of this illustration is to exclusively use nonblocking assignment in **always/process** statements when modeling sequential logic.

4.6 FINITE STATE MACHINES

HDL Example 4.30: DIVIDE-BY-3 FINITE STATE MACHINE

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0:      nextstate = S1;
      S1:      nextstate = S2;
      S2:      nextstate = S0;
      default: nextstate = S0;
    endcase

  // output logic
  assign y = (state == S0);
endmodule
```

The `typedef` statement defines `statetype` to be a two-bit `logic` value with three possibilities: `S0`, `S1`, or `S2`. `state` and `nextstate` are `statetype` signals.

The enumerated encodings default to numerical order: `S0` = 00, `S1` = 01, and `S2` = 10. The encodings can be explicitly set by the user; however, the synthesis tool views them as suggestions, not requirements. For example, the following snippet encodes the states as 3-bit one-hot values:

```
typedef enum logic [2:0] {S0=3'b001, S1=3'b010, S2=3'b100}
statetype;
```

Notice how a `case` statement is used to define the state transition table. Because the next state logic should be combinational, a `default` is necessary even though the state `2'b11` should never arise.

The output, `y`, is 1 when the state is `S0`. The *equality comparison* `a == b` evaluates to 1 if `a` equals `b` and 0 otherwise. The *inequality comparison* `a != b` does the inverse, evaluating to 1 if `a` does not equal `b`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity divideby3FSM is
  port(clk, reset: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  nextstate <= S1 when state = S0 else
               S2 when state = S1 else
               S0;

  -- output logic
  y <= '1' when state = S0 else '0';
end;
```

This example defines a new *enumeration* data type, `statetype`, with three possibilities: `S0`, `S1`, and `S2`. `state` and `nextstate` are `statetype` signals. By using an enumeration instead of choosing the state encoding, VHDL frees the synthesizer to explore various state encodings to choose the best one.

The output, `y`, is 1 when the `state` is `S0`. The inequality comparison uses `/=`. To produce an output of 1 when the state is anything but `S0`, change the comparison to `state /= S0`.

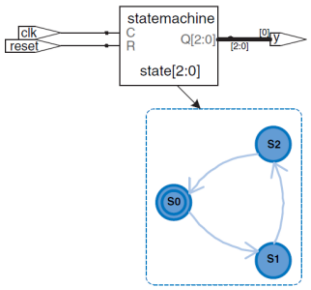


Figure 4. 28 divideby3fsm synthesized circuit

SystemVerilog

```
// output logic
assign y = (state == S0 | state == S1);
```

VHDL

```
-- output logic
y <= '1' when (state = S0 or state = S1) else '0';
```

HDL Example 4.31: PATTERN RECOGNIZER MOORE FSM

SystemVerilog

```
module patternMoore(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S2;
          else   nextstate = S1;
      S2: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (state == S2);
endmodule
```

Note how nonblocking assignments (<=) are used in the state register to describe sequential logic, whereas blocking assignments (=) are used in the next state logic to describe combinational logic.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMoore is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMoore is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then                state <= S0;
    elsif rising_edge(clk) then state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
        if a then nextstate <= S2;
        else      nextstate <= S1;
        end if;
      when S2 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when others =>
                  nextstate <= S0;
    end case;
  end process;

  --output logic
  y <= '1' when state = S2 else '0';
end;
```

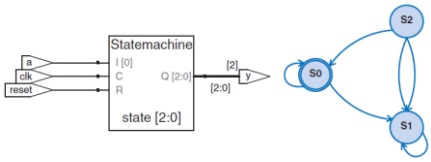


Figure 4. 29 patternMoore synthesized circuit

HDL Example 4.32: PATTERN RECOGNIZER MEALY FSM

SystemVerilog

```
module patternMealy(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic [S0, S1] statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (a & state == S1);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMealy is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMealy is
  type statetype is (S0, S1);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then                state <= S0;
    elsif rising_edge(clk) then state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when others =>
                  nextstate <= S0;
    end case;
  end process;

  -- output logic
  y <= '1' when (a = '1' and state = S1) else '0';
end;
```

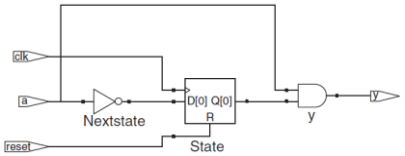


Figure 4. 30 patternMealy synthesized circuit

4.7 DATA TYPES

SystemVerilog

Prior to SystemVerilog, **Verilog** primarily used two types: **reg** and **wire**. Despite its name, a **reg** signal might or might not be associated with a register. This was a great source of **confusion**

for those learning the language. SystemVerilog introduced the **logic** type to eliminate the confusion; hence, this book emphasizes the **logic** type. This section explains the **reg** and **wire** types in more detail for those who need to read **old** Verilog code.

In Verilog, if a signal appears on the left hand side of `<=` or `=` in an always block, it must be declared as **reg**. Otherwise, it should be declared as **wire**. Hence, a **reg** signal might be the output of a flip-flop, a latch, or combinational logic, depending on the sensitivity list and statement of an **always** block.

Input and output ports default to the **wire** type unless their type is explicitly defined as **reg**. The following example shows how a flip-flop is described in conventional Verilog. Note that **clk** and **d** default to **wire**, while **q** is explicitly defined as **reg** because it appears on the left hand side of `<=` in the **always** block.

```
module flop(input            clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @(posedge clk)
    q <= d;
endmodule
```

SystemVerilog introduces the **logic** type. **logic** is a synonym for **reg** and avoids misleading users about whether it is actually a flip-flop. Moreover, SystemVerilog **relaxes** the rules on **assign** statements and hierarchical port instantiations so **logic** can be used outside **always** blocks where a **wire** traditionally would have been required. Thus, nearly all SystemVerilog signals can be **logic**. The exception is that signals with multiple drivers (e.g., a tristate bus) must be declared as a **net**, as described in HDL Example 4.10. This rule allows SystemVerilog to generate an error message rather than an `x` value when a **logic** signal is accidentally connected to multiple drivers.

The most common type of **net** is called a **wire** or **tri**. These two types are **synonymous**, but **wire** is conventionally used when a single driver is present and **tri** is used when multiple drivers are present. Thus, **wire** is obsolete in SystemVerilog because **logic** is preferred for signals with a single driver.

Net Type	No Driver	Conflicting Drivers
tri	z	x
trireg	previous value	x
triand	z	0 if there are any 0
trior	z	1 if there are any 1
tri0	0	x
tri1	1	x

Figure 4. 31 Net Resolution

VHDL

HDL Example 4.33: (a) UNSIGNED MULTIPLIER (b) SIGNED MULTIPLIER

SystemVerilog

```
// 4.33(a): unsigned multiplier
module multiplier(input  logic [3:0] a, b,
                  output logic [7:0] y);
  assign y = a * b;
endmodule

// 4.33(b): signed multiplier
module multiplier(input  logic signed [3:0] a, b,
                  output logic signed [7:0] y);
  assign y = a * b;
endmodule
```

In SystemVerilog, signals are considered unsigned by default. Adding the `signed` modifier (e.g., `logic signed [3:0] a`) causes the signal `a` to be treated as signed.

VHDL

```
-- 4.33(a): unsigned multiplier
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity multiplier is
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       y:    out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of multiplier is
begin
  y <= a * b;
end;
```

VHDL uses the `NUMERIC_STD_UNSIGNED` library to perform arithmetic and comparison operations on `STD_LOGIC_VECTOR`s. The vectors are treated as unsigned.

```
use IEEE.NUMERIC_STD_UNSIGNED.all;
```

VHDL also defines `UNSIGNED` and `SIGNED` data types in the `IEEE.NUMERIC_STD` library, but these involve type conversions beyond the scope of this chapter.

4.8 PARAMETERIZED MODULES

So far all of our modules have had **fixed**-width inputs and outputs. For example, we had to define separate modules for 4- and 8-bit wide 2:1 multiplexers. HDLs permit **variable bit** widths using parameterized modules.

HDL Example 4.34: PARAMETERIZED N-BIT 2:1 MULTIPLEXERS

SystemVerilog

```
module mux2
  #(parameter width=8)
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);

   assign y = s ? d1 : d0;
endmodule
```

SystemVerilog allows a `#(parameter . . . )` statement before the inputs and outputs to define parameters. The `parameter` statement includes a default value (8) of the parameter, in this case called `width`. The number of bits in the inputs and outputs can depend on this parameter.

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);

  logic [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

The 8-bit 4:1 multiplexer instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `#( )` before the instance name, as shown below.

```
module mux4_12(input logic  [11:0] d0, d1, d2, d3,
               input logic  [1:0]  s,
               output logic [11:0] y);

  logic [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

Do not confuse the use of the # sign indicating delays with the use of `#(. . .)` in defining and overriding parameters.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  generic(width: integer := 8);
  port(d0,
    d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
    s:  in  STD_LOGIC;
    y:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

The `generic` statement includes a default value (8) of `width`. The value is an integer.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4_8 is
  port(d0, d1, d2,
       d3: in  STD_LOGIC_VECTOR(7 downto 0);
       s:  in  STD_LOGIC_VECTOR(1 downto 0);
       y:  out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux4_8 is
  component mux2
    generic(width: integer := 8);
    port(d0,
         d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
  lowmux: mux2 port map(d0, d1, s(0), low);
  himux:  mux2 port map(d2, d3, s(0), hi);
  outmux: mux2 port map(low, hi, s(1), y);
end;
```

The 8-bit 4:1 multiplexer, `mux4_8`, instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `generic map`, as shown below.

```
lowmux: mux2 generic map(12)
             port map(d0, d1, s(0), low);
himux:  mux2 generic map(12)
             port map(d2, d3, s(0), hi);
outmux: mux2 generic map(12)
             port map(low, hi, s(1), y);
```

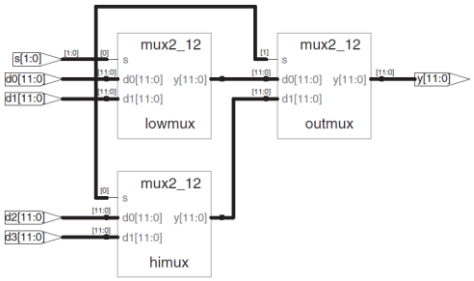


Figure 4. 32 mux4_12 synthesized circuit

HDL Example 4.35: PARAMETERIZED N: 2^N DECODER

SystemVerilog

```
module decoder
  #(parameter N=3)
   (input  logic [N-1:0]    a,
    output logic [2**N-1:0] y);

  always_comb
    begin
      y=0;
      y[a]=1;
    end
endmodule
```

2**N indicates 2^N.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE. NUMERIC_STD_UNSIGNED.all;

entity decoder is
  generic(N: integer := 3);
  port(a: in  STD_LOGIC_VECTOR(N-1 downto 0);
       y: out STD_LOGIC_VECTOR(2**N-1 downto 0));
end;

architecture synth of decoder is
begin
  process(all)
  begin
    y <= (OTHERS => '0');
    y(TO_INTEGER(a)) <= '1';
  end process;
end;
```

2**N indicates 2^N.

Specifically, the decoder uses blocking assignments to set **all** the bits to 0, **then** changes the appropriate bit to 1.

HDLs also provide **generate** statements to produce a **variable** amount of hardware depending on the value of a parameter. **generate** supports **for** loops and **if** statements to determine how many of what types of hardware to produce.

HDL Example 4.36: PARAMETERIZED N-INPUT AND GATE

SystemVerilog

```
module andN
  #(parameter width=8)
   (input  logic [width-1:0] a,
    output logic             y);

  genvar i;
  logic [width-1:0] x;

  generate
    assign x[0]=a[0];
    for(i=1; i<width; i=i+1) begin: forloop
      assign x[i]=a[i] & x[i-1];
    end

  endgenerate

  assign y=x[width-1];
endmodule
```

The `for` statement loops thrugh `i` = 1, 2, … , `width-1` to produce many consecutive AND gates. The `begin` in a `generate for` loop must be followed by a : and an arbitrary label (`forloop`, in this case).

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity andN is
  generic(width: integer := 8);
  port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of andN is
  signal x: STD_LOGIC_VECTOR(width-1 downto 0);
begin
  x(0) <= a(0);
  gen: for i in 1 to width-1 generate
    x(i) <= a(i) and x(i-1);
  end generate;
  y <= x(width-1);
end;
```

The generate loop variable `i` does not need to be declared.

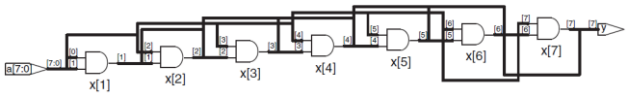


Figure 4. 33 andN synthesized circuit

4.9 TESTBENCHES

A ***testbench*** is an **HDL module** that is used to test another module, called the ***device under test*** (*DUT*). The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output **patterns** are called *test vectors*.

(Golden Models: A golden model represents the ideal circuit behavior)

HDL Example 4.37: TESTBENCH

SystemVerilog

```
module testbench1();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  initial begin
    a=0; b=0; c=0; #10;
    c=1;           #10;
    b=1; c=0;      #10;
    c=1;           #10;
    a=1; b=0; c=0; #10;
    c=1;           #10;
    b=1; c=0;      #10;
    c=1;           #10;
  end
endmodule
```

The `initial` statement executes the statements in its body at the start of simulation. In this case, it first applies the input pattern 000 and waits for 10 time units. It then applies 001 and waits 10 more units, and so forth until all eight possible inputs have been applied. `initial` statements should be used only in testbenches for simulation, not in modules intended to be synthesized into actual hardware. Hardware has no way of magically executing a sequence of special steps when it is first turned on.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench1 is -- no inputs or outputs
end;

architecture sim of testbench1 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    wait; -- wait forever
  end process;
end;
```

The `process` statement first applies the input pattern 000 and waits for 10 ns. It then applies 001 and waits 10 more ns, and so forth until all eight possible inputs have been applied.

At the end, the process waits indefinitely; otherwise, the process would begin again, repeatedly applying the pattern of test vectors.

A much better approach is to write a **self-checking** testbench, shown in HDL Example 4.38.

HDL Example 4.38: SELF-CHECKING TESTBENCH

SystemVerilog

```
module testbench2();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results
  initial begin
    a=0; b=0; c=0; #10;
    assert (y === 1) else $error("000 failed.");
    c=1;  #10;
    assert (y === 0) else $error("001 failed.");
    b=1; c=0;  #10;
    assert (y === 0) else $error("010 failed.");
    c=1;  #10;
    assert (y === 0) else $error("011 failed.");
    a=1; b=0; c=0;  #10;
    assert (y === 1) else $error("100 failed.");
    c=1;  #10;
    assert (y === 1) else $error("101 failed.");
    b=1; c=0;  #10;
    assert (y === 0) else $error("110 failed.");
    c=1;  #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

The SystemVerilog `assert` statement checks if a specified condition is true. If not, it executes the `else` statement. The `$error` system task in the `else` statement prints an error message describing the assertion failure. `assert` is ignored during synthesis.

In SystemVerilog, comparison using `==` or `!=` is effective between signals that do not take on the values of `x` and `z`. Testbenches use the `===` and `!==` operators for comparisons of equality and inequality, respectively, because these operators work correctly with operands that could be `x` or `z`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench2 is -- no inputs or outputs
end;

architecture sim of testbench2 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  -- checking results
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "000 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "001 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "010 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "011 failed.";
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "100 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '1' report "101 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "110 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "111 failed.";
    wait; -- wait forever
  end process;
end;
```

The `assert` statement checks a condition and prints the message given in the `report` clause if the condition is not satisfied. `assert` is meaningful only in simulation, not in synthesis.

An even better approach is to place the test vectors in a separate file. The testbench simply reads the test vectors from the file, applies the input test vector to the DUT, waits, checks that the output values from the DUT match the output vector, and repeats until reaching the end of the test vectors file.

HDL Example 4.39: TESTBENCH WITH TEST VECTOR FILE

SystemVerilog

```
module testbench3();
  logic         clk, reset;
  logic         a, b, c, y, yexpected;
  logic [31:0]  vectornum, errors;
  logic [3:0]   testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always
    begin
      clk=1; #5; clk=0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum=0; errors=0;
      reset=1; #27; reset=0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected}=testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin  // check result
        $display("Error: inputs=%b", {a, b, c});
        $display(" outputs=%b (%b expected)", y, yexpected);
        errors=errors+1;
      end
      vectornum=vectornum+1;
      if (testvectors[vectornum]=== 4'bx) begin
        $display("%d tests completed with %d errors",
                 vectornum, errors);
        $finish;
      end
    end
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL; use STD.TEXTIO.all;

entity testbench3 is -- no inputs or outputs
end;

architecture sim of testbench3 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y:  STD_LOGIC;
  signal y_expected: STD_LOGIC;
  signal clk, reset: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, pulse reset
  process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;

  -- run tests
  process is
    file tv: text;
    variable L: line;
    variable vector_in: std_logic_vector(2 downto 0);
    variable dummy: character;
    variable vector_out: std_logic;
    variable vectornum: integer := 0;
    variable errors: integer := 0;
  begin
    FILE_OPEN(tv, "example.tv", READ_MODE);
    while not endfile(tv) loop

      -- change vectors on rising edge
      wait until rising_edge(clk);

      -- read the next line of testvectors and split into pieces
      readline(tv, L);
      read(L, vector_in);
      read(L, dummy); -- skip over underscore
```

Exercises

Exercise 4.24: Sketch the state transition diagram for the FSM described by the following HDL code.

SystemVerilog

```
module fsm2(input  logic clk, reset,
            input  logic a, b,
            output logic y);

  logic [1:0] state, nextstate;

  parameter  S0 = 2'b00;
  parameter  S1 = 2'b01;
  parameter  S2 = 2'b10;
  parameter  S3 = 2'b11;

  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  always_comb
    case (state)
      S0: if (a ^ b) nextstate = S1;
          else       nextstate = S0;
      S1: if (a & b) nextstate = S2;
          else       nextstate = S0;
      S2: if (a | b) nextstate = S3;
          else       nextstate = S0;
      S3: if (a | b) nextstate = S3;
          else       nextstate = S0;
    endcase

  assign y = (state == S1) | (state == S2);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm2 is
  port(clk, reset: in  STD_LOGIC;
       a, b:       in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of fsm2 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  process(all) begin
    case state is
      when S0 => if (a xor b) then
                   nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if (a and b) then
                   nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if (a or b) then
                   nextstate <= S3;
                 else nextstate <= S0;
                 end if;
      when S3 => if (a or b) then
                   nextstate <= S3;
                 else nextstate <= S0;
                 end if;
    end case;
  end process;

  y <= '1' when ((state = S1) or (state = S2))
       else '0';
end;
```

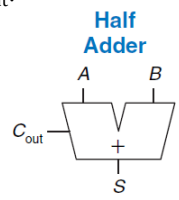
5.2 ARITHMETIC CIRCUITS

Arithmetic circuits are the central building blocks of computers. Computers and digital logic perform many arithmetic functions: addition, subtraction, comparisons, shifts, multiplication, and division.

Addition

Half Adder

We begin by building a 1-bit half adder. As shown in Figure 5.1, the half adder has two inputs, A and B, and two outputs, S and C_{out}.



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$
$$C_{\text{out}} = AB$$

Figure 5. 1 1-bit half adder

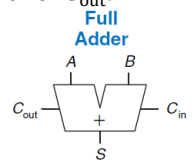
In a multi-bit adder, C_{out} is added or carried in to the next most significant bit. For example, in Figure 5.2.

$$\begin{array}{r} {\scriptstyle 1} \\ 0001 \\ +0101 \\ \hline 0110 \end{array}$$

Figure 5. 2 Carry bit

Full Adder

A full adder, introduced in Section 2.1, accepts the carry in C_{in} as shown in Figure 5.3. The figure also shows the output equations for S and C_{out}.



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{\text{in}}$$
$$C_{\text{out}} = AB + AC_{\text{in}} + BC_{\text{in}}$$

Figure 5. 3 1-bit full adder

Carry Propagate Adder

An N-bit adder sums **two** N-bit inputs, A and B, and a carry in C_{in} to produce **an** N-bit result S and a carry out C_{out}. It is commonly called a *carry propagate adder* (*CPA*) because the carry out of one bit propagates into the next bit. The symbol for a CPA is shown in Figure 5.4; it is drawn just like a full adder except that A, B, and S are **busses** rather than single bits. Three common CPA implementations are called ripple-carry adders, carry-lookahead adders, and

prefix adders.

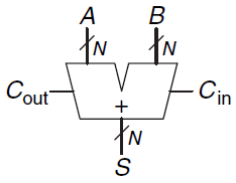


Figure 5. 4 Carry propagate adder

Ripple-Carry Adder

The simplest way to build an N-bit carry propagate adder is to **chain** together N full adders. The C_{out} of one stage acts as the C_{in} of the next stage, as shown in Figure 5.5 for 32-bit addition. This is called a *ripple-carry adder*. As shown in blue in Figure 5.5. We say that the carry *ripples* through the carry chain. The delay of the adder, t_{ripple}, grows directly with the number of bits,

$$t_{\text{ripple}} = Nt_{FA}$$

where t_{FA} is the delay of a full adder.

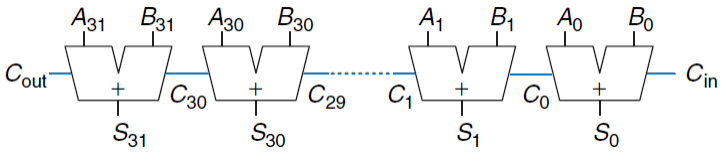


Figure 5. 5 32-bit ripple-carry adder

Carry-Lookahead Adder

The fundamental reason that large ripple-carry adders are slow is that the carry signals must propagate through every bit in the adder. A *carry-lookahead* adder (*CLA*) is another type of carry propagate adder that solves this problem by dividing the adder into blocks and providing circuitry to quickly determine the carry out of a block as soon as the carry in is known.

CLAs use ***generate*** (G) and ***propagate*** (P) signals that describe how a column or block determines the carry out. The ith column of an adder is said to *generate* a carry if it produces a carry out independent of the carry in. The column is said to *propagate* a carry if it produces a carry out whenever there is a carry in. The ith column of an adder will generate a carry out C_i if it either generates a carry, G_i, or propagates a carry in, P_iC_{i-1}. In equation form,

$$C_i = A_iB_i + (A_i + B_i)C_{i-1} = A_iB_i + (A_i \oplus B_i)C_{i-1} = G_i + P_iC_{i-1}$$

The generate and propagate definitions extend to multiple-bit blocks. A block is said to generate a carry if it produces a carry out independent of the carry in to the block. The block is said to propagate a carry if it produces a carry out whenever there is a carry in to the block. We define $G_{i:j}$ and $P_{i:j}$ as generate and propagate signals for blocks spanning columns i through j.

A block generates a carry if the most significant column generates a carry, or if the most significant column propagates a carry and the previous column generated a carry, and so forth. For example, the generate logic for a block spanning columns 3 through 0 is

$$G_{3:0} = G_3 + P_3\big(G_2 + P_2(G_1 + P_1G_0)\big)$$

A block propagates a carry if all the columns in the block propagate the carry. For example, the propagate logic for a block spanning columns 3 through 0 is

$$P_{3:0} = P_3P_2P_1P_0$$

Note from above:

$$C_{\text{out}} = G_3 + P_3\Big(G_2 + P_2\big(G_1 + P_1(G_0 + P_0C_{\text{in}})\big)\Big) = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_{\text{in}}$$

so,

$$G_{3:0} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 = G_3 + P_3\big(G_2 + P_2(G_1 + P_1G_0)\big)$$
$$P_{3:0} = P_3P_2P_1P_0$$

then,

$$C_{\text{out}} = C_3 = G_{3:0} + P_{3:0}C_{\text{in}} = G_{3:0} + P_{3:0}C_{-1}.$$

Using the block generate and propagate signals, we can quickly compute the carry out of the

block, C_i, using the carry in to the block, C_{j-1}.

$$C_i = G_{i:j} + P_{i:j}C_{j-1}$$

where $G_{i:j} = G_{i:i-1} + P_{i:i-1}G_{i-1:j}$, $P_{i:j} = P_iP_{i-1}\cdots P_j$.

Figure 5.6(a) shows a 32-bit carry-lookahead adder composed of eight 4-bit blocks. Each block contains a 4-bit ripple-carry adder and some lookahead logic to compute the carry out of the block given the carry in, as shown in Figure 5.6(b). The AND and OR gates needed to compute the single-bit generate and propagate signals, G_i and P_i, from A_i and B_i are left out for brevity. Again, the carry-lookahead adder demonstrates modularity and regularity.

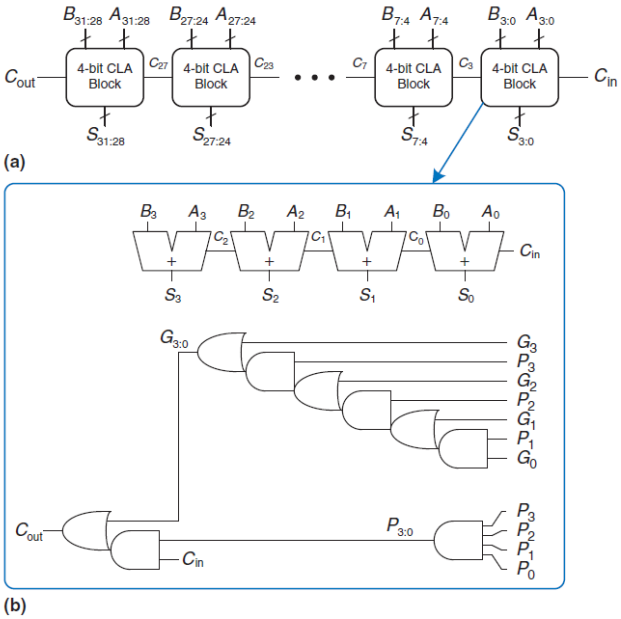


Figure 5. 6 (a) 32-bit carry-lookahead adder (CLA), (b) 4-bit CLA block

All of the CLA blocks compute the single-bit and block generate and propagate signals simultaneously. The critical path starts with computing G_0 and $G_{3:0}$ in the first CLA block. C_{in} then advances directly to C_{out} through the AND/OR gate in each block until the last. Finally, the critical path through the last block contains a short ripple-carry adder. Thus, an N-bit adder divided into k-bit blocks has a delay

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + \left(\frac{N}{k} - 1\right)t_{\text{AND\_OR}} + kt_{FA}$$

where t_{pg} is the delay of the individual generate/propagate gates (a single AND or OR gate) to generate G_i and P_i, t_{pg_block} is the delay to find the generate/propagate signals $P_{i:j}$ and $G_{i:j}$ for a k-bit block, and $t_{\text{AND\_OR}}$ is the delay from C_{in} to C_{out} through the final AND/OR logic of the k-bit CLA block.

Example 5.1: RIPPLE-CARRY ADDER AND CARRY-LOOKAHEAD ADDER DELAY (Page 243)

Prefix Adder

Prefix adders extend the generate and propagate logic of the carry-lookahead adder to perform addition even faster. They first compute G and P for pairs of columns, then for blocks of 4, then for blocks of 8, then 16, and so forth until the generate signal for every column is known. The sums are computed from these generate signals.

In other words, the strategy of a prefix adder is to compute the carry in C_{i-1} for each column i as quickly as possible, then to compute the sum, using

$$S_i = (A_i \oplus B_i) \oplus C_{i-1}$$

Define column $i = -1$ to hold C_{in}, so $G_{-1} = C_{\text{in}}$ and $P_{-1} = 0$.

Note:

$$G_{3:-1} = G_3 + P_3\left(G_2 + P_2\left(G_1 + P_1(G_0 + P_0G_{-1})\right)\right) = G_3 + P_3\left(G_2 + P_2\left(G_1 + P_1(G_0 + P_0C_{\text{in}})\right)\right)$$
$$= C_3$$

Then $C_{i-1} = G_{i-1:-1}$ because there will be a carry out of column $i-1$ if the block spanning

columns $i-1$ through -1 generates a carry. The generated carry is either generated in column $i-1$ or generated in a previous column and propagated. Thus, we rewrite Equation S_i as

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

Hence, the main challenge is to rapidly compute all the block generate signals $G_{-1:-1}$, $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $\cdots$, $G_{N-2:-1}$. ($N-1-1=N-2$) These signals, along $P_{-1:-1}$, $P_{0:-1}$, $P_{1:-1}$, $P_{2:-1}$, $\cdots$, $P_{N-2:-1}$, are called ***prefixes***.

Figure 5.7 shows an $N = 16$-bit prefix adder. The adder begins with a *precomputation* to form P_i and G_i for each column from A_i and B_i using OR and AND gates. It then uses $\log_2 N = 4$ levels of black cells to form the prefixes of $G_{i:j}$ and $P_{i:j}$. A black cell takes inputs from the upper part of a block spanning bits $i:k$ and from the lower part spanning bits $k-1:j$. It combines these parts to form generate and propagate signals for the entire block spanning bits $i:j$ using the equations

$$G_{i:j} = G_{i:k} + P_{i:k}G_{k-1:j}$$
$$P_{i:j} = P_{i:k}P_{k-1:j}$$

In other words, a block spanning bits $i:j$ will generate a carry if the upper part generates a carry or if the upper part propagates a carry generated in the lower part. The block will propagate a carry if both the upper and lower parts propagate the carry. Finally, the prefix adder computes the sums $S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$. (Note we can get: $C_{15} = G_{15:-1} = G_{15} + P_{15}G_{14:-1}$)

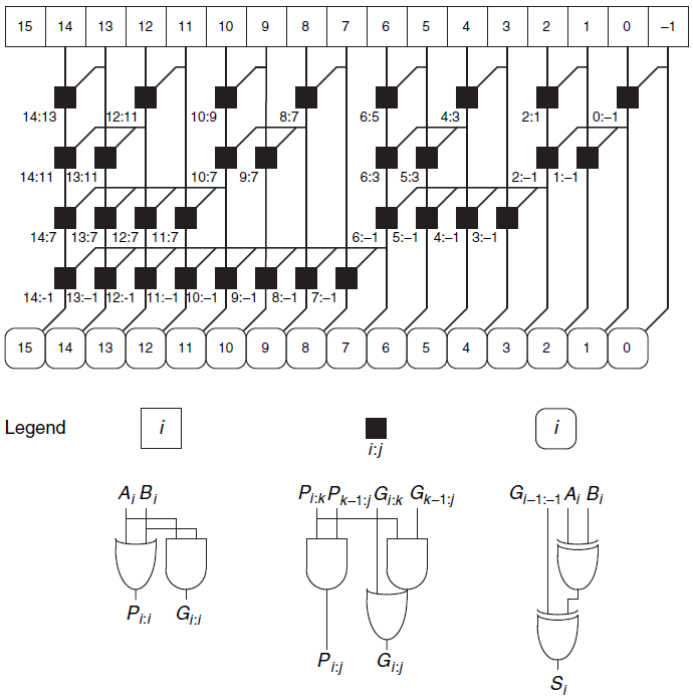


Figure 5. 7 16-bit prefix adder

The critical path for an N-bit prefix adder involves the precomputation of P_i and G_i followed by $\log_2 N$ stages of black prefix cells to obtain all the prefixes. $G_{i-1:-1}$ then proceeds through the final XOR gate at the bottom to compute S_i. Mathematically, the delay of an N-bit prefix adder is

$$t_{PA} = t_{pg} + \log_2 N\,(t_{pg\_\text{prefix}}) + t_{\text{XOR}}$$

where t_{pg_prefix} is the delay of a black prefix cell.

Example 5.2: PREFIX ADDER DELAY (Page 245)

Putting It All Together

HDL Example 5.1: ADDER

SystemVerilog

```
module adder #(parameter N = 8)
              (input  logic [N-1:0] a, b,
               input  logic         cin,
               output logic [N-1:0] s,
               output logic         cout);

  assign {cout, s} = a + b + cin;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity adder is
  generic(N: integer := 8);
  port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
       cin:  in  STD_LOGIC;
       s:    out STD_LOGIC_VECTOR(N-1 downto 0);
       cout: out STD_LOGIC);
end;

architecture synth of adder is
  signal result: STD_LOGIC_VECTOR(N downto 0);
begin
  result <= ("0" & a) + ("0" & b) + cin;
  s      <= result(N-1 downto 0);
  cout   <= result(N);
end;
```

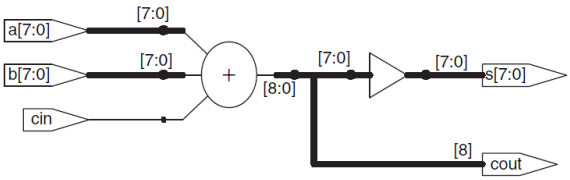


Figure 5. 8 Synthesized adder

Supplement materials

BCD Addition:

1) When Sum$\leq 9_{\text{BCD}}$, Final Carry$= 0_{\text{BCD}}$, the answer is correct;

2) When Sum$\leq 9_{\text{BCD}}$, Final Carry$= 1_{\text{BCD}}$, the answer is incorrect. We need add $6_{\text{BCD}}(0110_2)$, which is same as adding $-10_{\text{BCD}}(0110_2)$; (much trickier)

3) When Sum$> 9_{\text{BCD}}$, Final Carry$= 0_{\text{BCD}}$, the answer is incorrect. We need add $6_{\text{BCD}}(0110_2)$, which is same as adding $-10_{\text{BCD}}(0110_2)$;

Example 1: $(2)_{10} + (6)_{10}$ in BCD addition,

$$\begin{array}{cccc} 0 & 0 & 1 & 0 \\ + & 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 0 \end{array}$$

Sum$= 8_{\text{BCD}} \leq 9_{\text{BCD}}$, Final Carry$= 0_{\text{BCD}}$, the answer is correct.

Example 2: $(3)_{10} + (7)_{10}$ in BCD addition,

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ + & 0 & 1 & 1 & 1 \\ \hline 1 & 0 & 1 & 0 \end{array}$$

Since Sum$> 9_{\text{BCD}}$, Final Carry$= 0_{\text{BCD}}$, the answer is incorrect. The answer should be $1010_2 + 0110_2 = 10000_2 = 10_{\text{BCD}}$. Note here the carry out is used.

Example 3: $(8)_{10} + (9)_{10}$ in BCD addition,

$$\begin{array}{cccc} 1 & 0 & 0 & 0 \\ + & 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 \end{array}$$

Since Sum$\leq 9_{\text{BCD}}$, Final Carry$= 1_{\text{BCD}}$, the answer is incorrect. The answer should be, for last 4 bits, $0001_2 + 0110_2 = 0111_2$. For the first 4 bits, $0001_2 \leq 9_{\text{BCD}}$, Final Carry$= 0_{\text{BCD}}$, so the answer is correct. Finally, the answer is $00010111_2 = 17_{\text{BCD}}$.

Subtraction

Recall from Section 1.4.6 that adders can add positive and negative numbers using two's complement number representation. Subtraction is almost as easy: **flip** the sign of the **second** number, then add. Flipping the sign of a two's complement number is done by inverting the bits and adding 1. ($Y = A - B$, note here $-B = \bar{B} + 1$, then ($Y = A - B = A + \bar{B} + 1$)

This sum can be performed with a single CPA by adding $A + \bar{B}$ with $C_{\text{in}} = 1$.

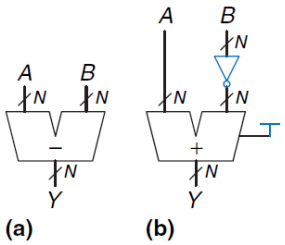


Figure 5. 9 Subtractor: (a) symbol, (b) implementation

HDL Example 5.2: SUBTRACTOR

SystemVerilog

```
module subtractor #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic [N-1:0] y);

  assign y = a - b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity subtractor is
  generic(N: integer := 8);
  port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
       y:    out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of subtractor is
begin
  y <= a - b;
end;
```

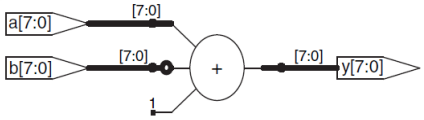


Figure 5. 10 Synthesized subtractor

Comparators

A *comparator* determines whether **two** binary numbers are equal or if one is greater or less than the other. A comparator receives two N-bit binary numbers A and B. There are two common types of comparators.

An *equality comparator* produces a single output indicating whether A is equal to B $(A == B)$. A *magnitude comparator* produces one or more outputs indicating the relative values of A and B.

The equality comparator is the simpler piece of hardware. In Figure 5.11, it first checks to determine whether the corresponding bits in each column of A and B are equal using XNOR gates. The numbers are equal if all of the columns are equal.

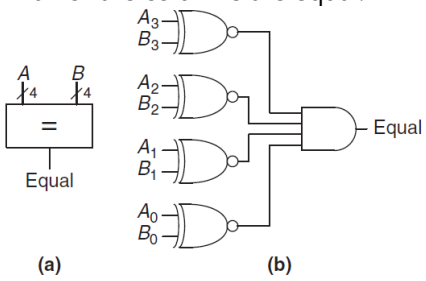


Figure 5. 11 4-bit equality comparator: (a) symbol, (b) implementation

Magnitude comparison of **signed** numbers is usually done by computing $A - B$ and looking at the sign (**most significant bit**) of the result as shown in Figure 5.12. If the result is negative (i.e., the sign bit is 1), then A is less than B. Otherwise A is greater than or equal to B. This comparator, however, functions **incorrectly** upon **overflow**.

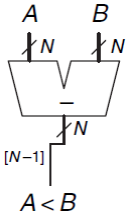


Figure 5. 12 N-bit signed comparator

HDL Example 5.3: COMPARATORS

SystemVerilog

```
module comparator #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic eq, neq, lt, lte, gt, gte);

  assign eq  = (a == b);
  assign neq = (a != b);
  assign lt  = (a < b);
  assign lte = (a <= b);
  assign gt  = (a > b);
  assign gte = (a >= b);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;

entity comparators is
  generic(N: integer := 8);
  port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
       eq, neq, lt, lte, gt, gte: out STD_LOGIC);
end;

architecture synth of comparator is
begin
  eq  <= '1' when (a = b)  else '0';
  neq <= '1' when (a /= b) else '0';
  lt  <= '1' when (a < b)  else '0';
  lte <= '1' when (a <= b) else '0';
  gt  <= '1' when (a > b)  else '0';
  gte <= '1' when (a >= b) else '0';
end;
```

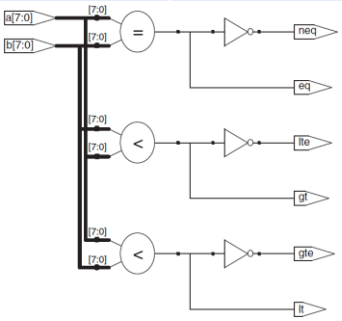


Figure 5. 13 Synthesized comparators

HDL Example 5.3 shows how to use various comparison operations for **unsigned** numbers.

ALU

An *Arithmetic/Logical Unit* (*ALU*) combines a variety of mathematical and logical operations into a **single** unit. For example, a typical ALU might perform addition, subtraction, AND, and OR operations. The ALU forms the heart of most computer systems.

Figure 5.14 shows the symbol for an N-bit ALU with N-bit inputs and outputs. The ALU receives a 2-bit **control** signal *ALUControl* that specifies which function to perform. Figure 5.15 lists typical functions that the ALU can perform.

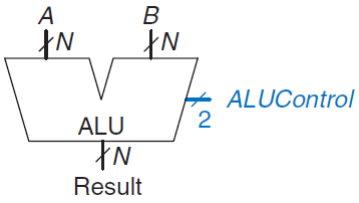


Figure 5. 14 ALU symbol

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Figure 5. 15 ALU operations

Figure 5.16 shows an implementation of the ALU. The ALU contains an N-bit adder and N two-input AND and OR gates. It also contains inverters and a multiplexer to invert input B when $ALUControl_0$ is asserted. A $4:1$ multiplexer chooses the desired function based on ALUControl.

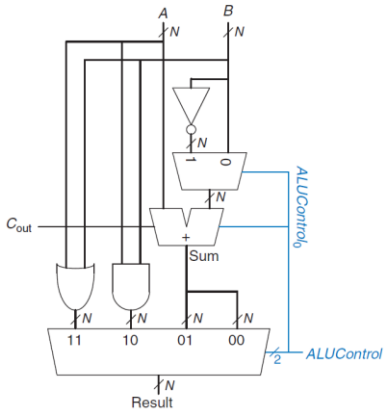


Figure 5. 16 N-bit ALU

Some ALUs produce extra outputs, called *flags*, that indicate information about the ALU output. Figure 5.17 shows the ALU symbol with a 4-bit *ALUFlags* output. As shown in the schematic of this ALU in Figure 5.18, the *ALUFlags* output is composed of the N, Z, C, and V flags that **indicate**, respectively, that the ALU output is **negative** or **zero** or that the adder produced a **carry out** or **overflowed**. Recall that the most significant bit of a two's complement number is 1 if it is negative and 0 otherwise. The Z flag is asserted when all of the bits of *Result* are 0. The C flag is asserted when the adder produces a carry out *and* the ALU is performing addition **or** subtraction. V (signed overflow) is asserted when all three of the following conditions are **true**: (1) the ALU is performing addition or subtraction, (2) A and Sum have opposite signs, (3) either A and B have the same sign and the adder is performing addition or A and B have opposite signs and the adder is performing subtraction.

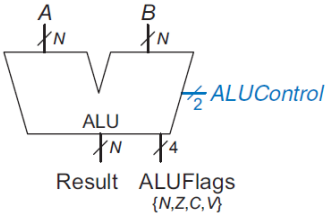


Figure 5. 17 ALU symbol with output flags

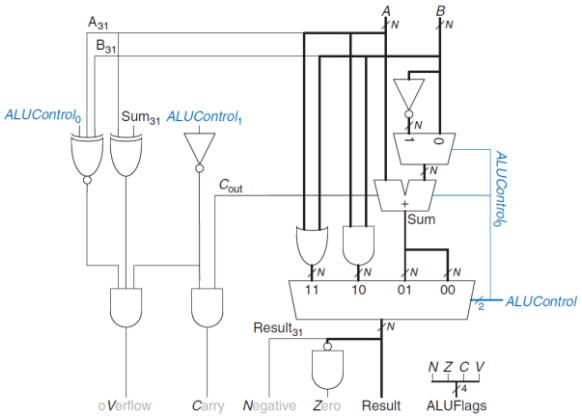


Figure 5. 18 N-bit ALU with output flags

Complement materials

Unsigned overflow: (1) for an addition, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise. (2) for a subtraction, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.

Shifters and Rotators

Shifters and *rotators* move bits and multiply or divide by powers of 2. As the name implies, a shifter shifts a binary number left or right by a specified number of positions. There are several kinds of commonly used shifters:

● **Logical shifter**—shifts the number to the left (LSL) or right (LSR) and fills empty spots with 0's.
Ex: 11001 LSR $2 = 00110$; 11001 LSL $2 = 00100$
(※ C language uses unsigned int for >>, >>=)
● **Arithmetic shifter**—is the same as a logical shifter, but on right shifts fills the most significant bits with a copy of the old most significant bit (msb). This is useful for multiplying and dividing signed numbers. Arithmetic shift left (ASL) is the same as logical shift left (LSL).
Ex: 11001 ASR $2 = 11110$; 11001 ASL $2 = 00100$
(※ C language uses signed int for >>, >>=)
● **Rotator**—rotates number in a circle such that empty spots are filled with bits shifted off the other end.
Ex: 11001 ROR $2 = 01110$; 11001 ROL $2 = 00111$
An N-bit shifter can be built from N N:1 multiplexers. The input is shifted by 0 to $N-1$ bits, depending on the value of the $\log_2 N$-bit select lines. Figure 5.19 shows the symbol and hardware of 4-bit shifters. The operators $\ll$, $\gg$, and $\ggg$ typically indicate shift left, logical shift right, and arithmetic shift right, respectively. Depending on the value of the 2-bit shift amount $shamt_{1:0}$, the output Y receives the input A shifted by 0 to 3 bits.

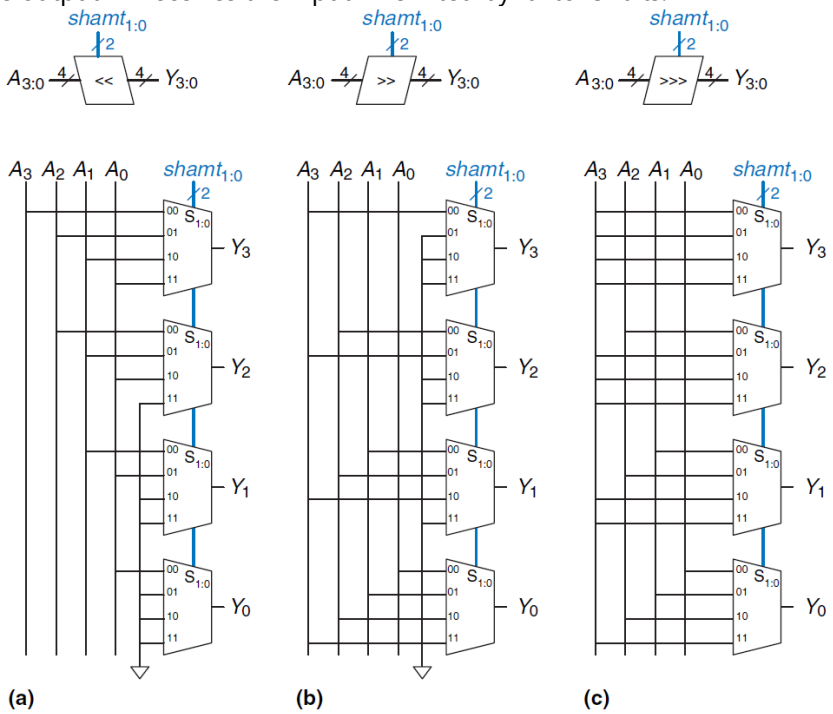


Figure 5. 19 4-bit shifters: (a) shift left, (b) logical shift right, (c) arithmetic shift right
A left shift is a special case of multiplication. A left shift by N bits multiplies the number by 2^N. An arithmetic right shift is a special case of division. An arithmetic right shift by N bits divides the number by 2^N.

Multiplication

Multiplication of **unsigned** binary numbers is similar to decimal multiplication but involves only 1's and 0's. Figure 5.20 compares multiplication in decimal and binary. In both cases, *partial products* are formed by multiplying a single digit of the multiplier with the entire multiplicand. The shifted partial products are summed to form the result.

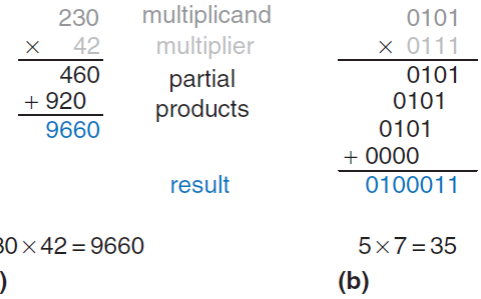


Figure 5. 20 Multiplication: (a) decimal, (b) binary

In general, an $N \times N$ multiplier multiplies two N-bit numbers and produces a $2N$-bit result. Signed and unsigned multiplication **differ**.

Figure 5.21 shows the symbol, function, and implementation of an unsigned 4×4 multiplier. With N-bit operands, there are N partial products and $N-1$ stages of 1-bit adders. (Note at each stage, the carry out of the CPA is connected to another CPA at same stage, unless the most left CPA)

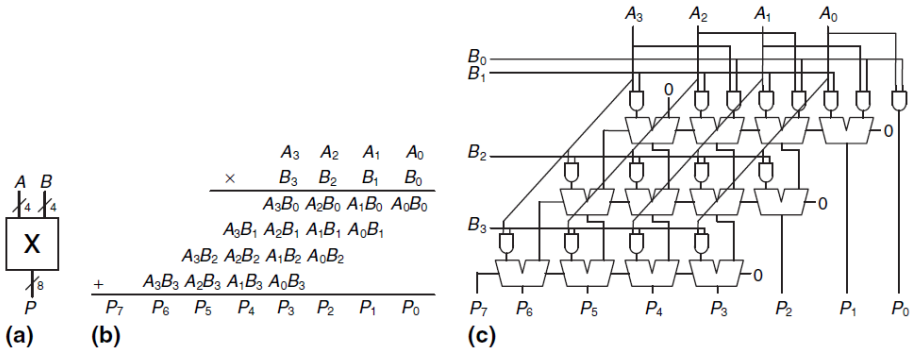


Figure 5. 21 4×4 multiplier: (a) symbol, (b) function, (c) implementation

A *multiply accumulate* operation multiplies two numbers and adds them to a third number, typically the accumulated value. These operations, also called MACs, are often used in *digital signal processing* (DSP) algorithms such as the Fourier transform, which requires a summation of products.

Division

Binary division can be performed using the following algorithm (Like normal decimal division) for N-bit **unsigned** numbers in the range $[0, 2^{N-1}]$: (Page 254)

```
R' = 0
for i = N–1 to 0
   R = {R' << 1, Ai}
   D = R – B
   if D < 0 then    Qi = 0, R' = R    // R < B
   else             Qi = 1, R' = D    // R ≥ B
R = R'
```

The *partial remainder* R is initialized to 0 ($R' = 0$), and the most significant bit of the dividend A becomes the least significant bit of R ($R = \{R' \ll 1, A_i\}$). The divisor B is subtracted from this partial remainder to determine whether it fits ($D = R - B$). If the difference D is negative (i.e., the sign bit of D is 1), then the quotient bit Q_i is 0 and the difference is discarded. Otherwise, Q_i is 1, and the partial remainder is updated to be the difference. In any event, the partial remainder is then doubled (left-shifted by one column), the next most significant bit of A becomes the least significant bit of R, and the process repeats. The result satisfies $\frac{A}{B} = Q + \frac{R}{B}$.

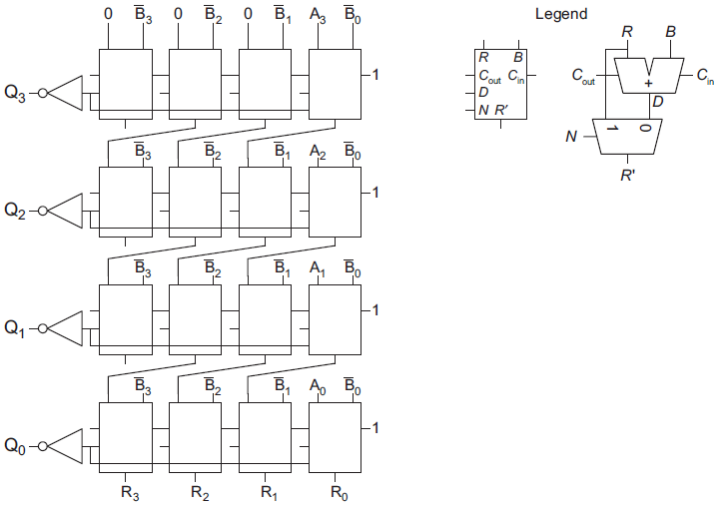
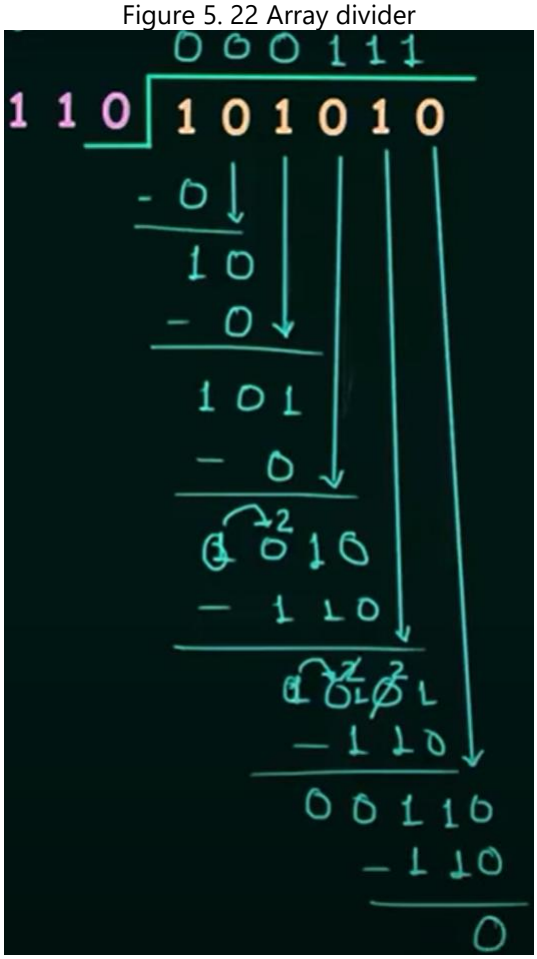


Figure 5. 22 Array divider



5.3 NUMBER SYSTEMS

Computers operate on both integers and fractions. So far, we have only considered representing signed or unsigned integers, as introduced in Section 1.4. This section introduces fixed- and floating-point number systems that can represent rational numbers. Fixed-point numbers are analogous to decimals; some of the bits represent the integer part, and the rest represent the fraction. Floating-point numbers are analogous to scientific notation, with a mantissa and an exponent.

Fixed-Point Number Systems

Fixed-point notation has an implied *binary point* between the integer and fraction bits, analogous to the decimal point between the integer and fraction digits of an ordinary decimal

number. Figure 5.23 shows a fixed-point number with four integer bits and four fraction bits. The integer bits are called the *high word* and the fraction bits are called the *low word*.

(a) 01101100

(b) 0110.1100

(c) $2^2+2^1+2^{-1}+2^{-2}=6.75$

Figure 5. 23 Fixed-point notation of 6.75 with four integer bits and four fraction bits

Signed fixed-point numbers can use either two's complement or sign/magnitude notation. Figure 5.24 shows the fixed-point representation of -2.375 using both notations with four integer and four fraction bits. The implicit binary point is shown in blue for clarity. In sign/magnitude form, the most significant bit is used to indicate the sign. The two's complement representation is formed by inverting the bits of the absolute value and adding a 1 to the least significant (rightmost) bit. In this case, the least significant bit position is in the 2^{-4} column.

(a) 0010.0110

(b) 1010.0110

(c) 1101.1010

Figure 5. 24 Fixed-point representation of -2.375: (a) absolute value, (b) sign and magnitude, (c) two's complement

Example 5.3: ARITHMETIC WITH FIXED-POINT NUMBERS

Compute $0.75 + -0.625$ using fixed-point numbers.

Solution: First convert 0.625, the magnitude of the second number, to fixed-point binary notation. $0.625 \geq 2^{-1}$, so there is a 1 in the 2^{-1} column, leaving $0.625 - 0.5 = 0.125$. Because $0.125 < 2^{-2}$, there is a 0 in the 2^{-2} column. Because $0.125 \geq 2^{-3}$, there is a 1 in the 2^{-3} column, leaving $0.125 - 0.125 = 0$. Thus, there must be a 0 in the 2^{-4} column. Putting this all together, $0.625_{10} = 0000.1010_2$. (multiplication can work: $\times 2$)

Use two's complement representation for signed numbers so that addition works correctly. Figure 5.25 shows the conversion of -0.625 to fixed-point two's complement notation.

	0000.1010	Binary Magnitude
	1111.0101	One's Complement
+	1	Add 1
	1111.0110	Two's Complement

Figure 5. 25 Fixed-point two's complement conversion

	0000.1100		0.75
+	1111.0110	+	(–0.625)
	10000.0010		0.125
	(a)		**(b)**

Figure 5. 26 Addition: (a) binary fixed-point, (b) decimal equivalent

Floating-Point Number Systems

Floating-point numbers are analogous to scientific notation. Like scientific notation, floating-point numbers have a *sign*, *mantissa* (M), *base* (B), and *exponent* (E), as shown in Figure 5.27. For example, the number 4.1×10^3 is the decimal scientific notation for 4100. It has a mantissa of 4.1, a base of 10, and an exponent of 3. Floating-point numbers are base 2 with a binary mantissa. 32 bits are used to represent 1 sign bit, 8 exponent bits, and 23 mantissa bits.

$$\pm M \times B^E$$

Figure 5. 27 Floating-point numbers

Example 5.4: 32-BIT FLOATING-POINT NUMBERS

Show the floating-point representation of the decimal number 228.

Solution: First convert the decimal number into binary: $228_{10} = 11100100_2 = 1.11001_2 \times 2^7$. Figure 5.28 shows the 32-bit encoding, which will be modified later for efficiency. The sign bit is positive (0), the 8 exponent bits give the value 7, and the remaining 23 bits are the mantissa.

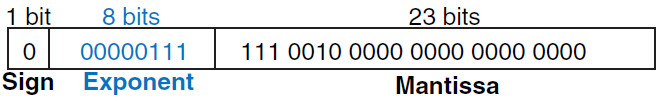


Figure 5. 28 32-bit floating-point version 1

In binary floating-point, the first bit of the mantissa (to the left of the binary point) is always 1 and therefore need not be stored. It is called the *implicit leading one*. Figure 5.29 shows the modified floating-point representation of $228_{10} = 11100100_2 \times 2^0 = 1.11001_2 \times 2^7$. The implicit leading one is not included in the 23-bit mantissa for efficiency. Only the fraction bits are stored. This frees up an extra bit for useful data.

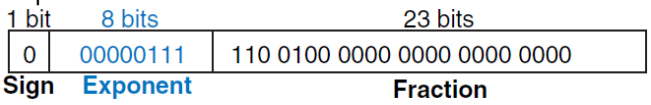


Figure 5. 29 32-bit floating-point version 2

We make one final modification to the exponent field. The exponent needs to represent both positive and negative exponents. To do so, floating-point uses a ***biased*** exponent, which is the original exponent plus a constant bias. 32-bit floating-point uses a bias of 127 ($0111\ 111_2$). Figure 5.30 shows $1.11001_2 \times 2^7$ represented in floating-point notation with an implicit leading one and a biased exponent of 134 ($7 + 127$). This notation conforms to the IEEE 754 floating-point standard.

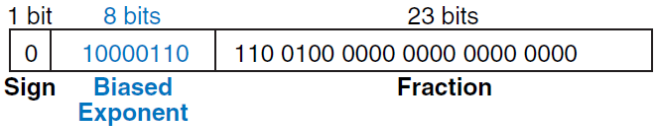


Figure 5. 30 IEEE 754 floating-point notation

Special Cases: 0, ±∞, and NaN

The IEEE floating-point standard has special cases to represent numbers such as zero, infinity, and illegal results. Special codes with exponents of all 0's or all 1's are reserved for these special cases. Figure 5.31 shows the floating-point representations of 0, ±∞, and NaN. As with sign/magnitude numbers, floating-point has both positive and negative 0. NaN is used for numbers that don't exist, such as $\sqrt{-1}$ or $\log_2(-5)$.

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

Figure 5. 31 IEEE 754 floating-point notations for 0, ±∞, and NaN

Single- and Double-Precision Formats

The 32-bit floating-point number is also called *single-precision*, *single*, or *float*. The IEEE 754 standard also defines 64-bit *double-precision* numbers (also called *doubles*) that provide greater precision and greater range.

Excluding the special cases mentioned earlier, normal single-precision numbers span a range of $\pm 1.175494 \times 10^{-38}$ (like $\pm 1.00 \cdots 0 \times 2^{-126}$; mantissa: $00 \cdots 00$; exponent: $0 + 1 - 127 = -126$) to $\pm 3.402824 \times 10^{-38}$ (like $\pm 1.999 \times 2^{127}$; mantissa: $11 \cdots 11$; exponent: $255 - 1 - 127 = 127$). They have a precision of about seven significant decimal digits (because $2^{-24} \approx 10^{-7}$). Similarly, normal double-precision numbers span a range of $\pm 2.22507385850720 \times 10^{-308}$ to $\pm 1.79769313486232 \times 10^{308}$ and have a precision of about 15 significant decimal digits.

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits
single	32	1	8	23
double	64	1	11	52

Figure 5. 32 Single- and double-precision floating-point formats

Floating-point cannot represent some numbers exactly, like 1.7, which is represented as $1.69999\cdots$. To handle this, some applications, such as calculators and financial software, use *binary coded decimal* (*BCD*) numbers or formats with a base 10 exponent. (Page 258)

Rounding

Arithmetic results that fall outside of the available precision must round to a neighboring number. The rounding modes are: round down, round up, round toward zero, and round to nearest. The default rounding mode is round to nearest. In the round to nearest mode, if two numbers are equally near, the one with a 0 in the least significant position of the fraction is chosen.

Recall that a number *overflows* when its magnitude is too large to be represented. Likewise, a number *underflows* when it is too tiny to be represented. In round to nearest mode, overflows are rounded up to $\pm\infty$ and underflows are rounded down to 0.

Floating-Point Addition

Addition with floating-point numbers is not as simple as addition with two's complement numbers. The steps for adding floating-point numbers with the same sign are as follows:

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. Add mantissas.
6. Normalize mantissa and adjust exponent if necessary.
7. Round result.
8. Assemble exponent and fraction back into floating-point number.

Figure 5.33 shows the floating-point addition of $7.875\ (1.11111 \times 2^2)$ and $0.1875\ (1.1 \times 2^{-3})$. The result is $8.0625\ (1.0000001 \times 2^3)$.

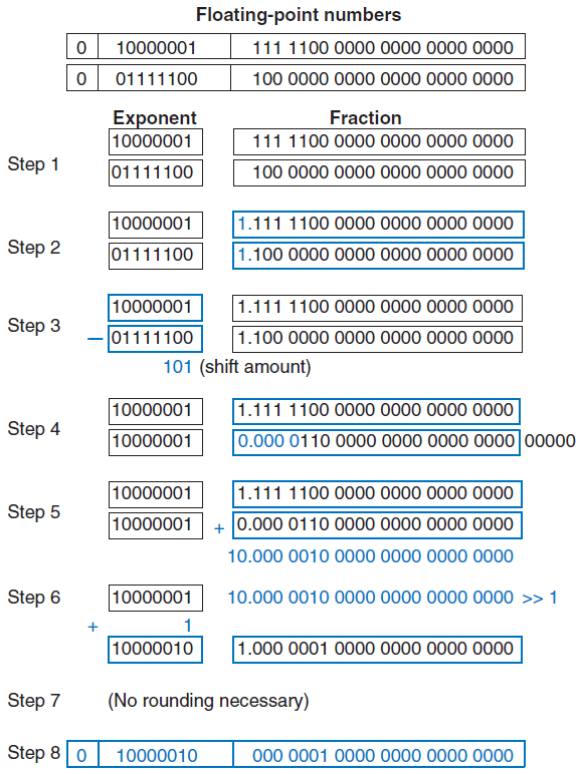


Figure 5. 33 Floating-point addition

5.4 SEQUENTIAL BUILDING BLOCKS

Counters

An N-bit *binary counter*, shown in Figure 5.34, is a sequential arithmetic circuit with clock and reset inputs and an N-bit output Q. *Reset* initializes the output to 0. The counter then advances through all 2^N possible outputs in binary order, incrementing on the rising edge of the clock.

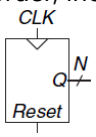


Figure 5. 34 Counter symbol

Figure 5.35 shows an N-bit counter composed of an adder and a resettable register.

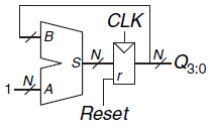


Figure 5. 35 N-bit counter

HDL Example 5.4: COUNTER (Page 261)

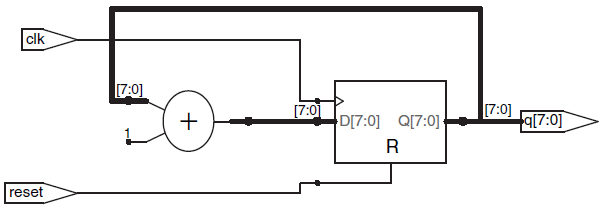


Figure 5. 36 Synthesized counter

Shift Registers

A *shift register* has a clock, a serial input S_{in}, a serial output S_{out}, and N parallel outputs $Q_{N-1:0}$, as shown in Figure 5.37. On each rising edge of the clock, a new bit is shifted in from S_{in} and all the subsequent contents are shifted forward. The last bit in the shift register is available at S_{out}. Shift registers can be viewed as *serial-to-parallel converters*. The input is provided serially (one bit at a time) at S_{in}. After N cycles, the past N inputs are available in parallel at Q.

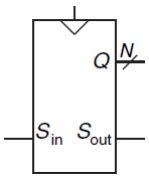


Figure 5. 37 Shift register symbol

A shift register can be constructed from N flip-flops connected in series, as shown in Figure 5.38. Some shift registers also have a reset signal to initialize all of the flip-flops.

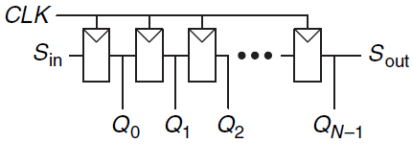


Figure 5. 38 Shift register schematic

A related circuit is a *parallel-to-serial* converter that loads N bits in parallel, then shifts them out one at a time. A shift register can be modified to perform both serial-to-parallel and parallel-to-serial operations by adding a parallel input $D_{N-1:0}$, and a control signal *Load*, as shown in Figure 3.39. When *Load* is asserted, the flip-flops are loaded in parallel from the D inputs. Otherwise, the shift register shifts normally.

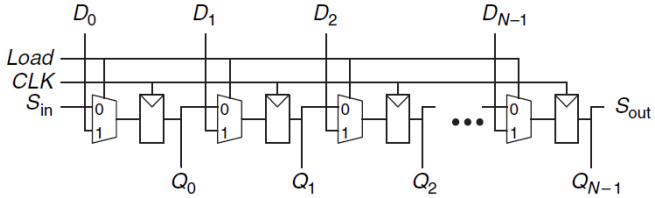


Figure 5. 39 Shift register with parallel load

HDL Example 5.5: SHIFT REGISTER WITH PARALLEL LOAD (Page 263)

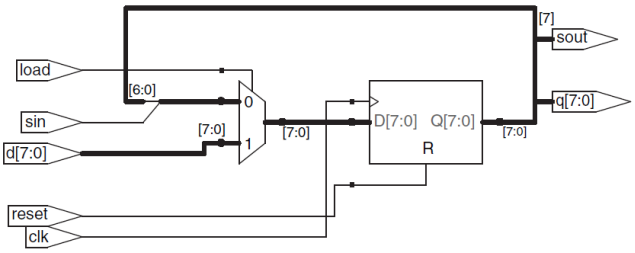


Figure 5. 40 Synthesized shiftreg

Scan Chains

Shift registers are often used to test sequential circuits using a technique called *scan chains*. To solve this problem, designers like to be able to directly observe and control all the state of the machine. All the flip-flops in the system are connected together into a shift register called a scan chain. The load multiplexer is usually integrated into the flip-flop to produce a *scannable flip-flop*.

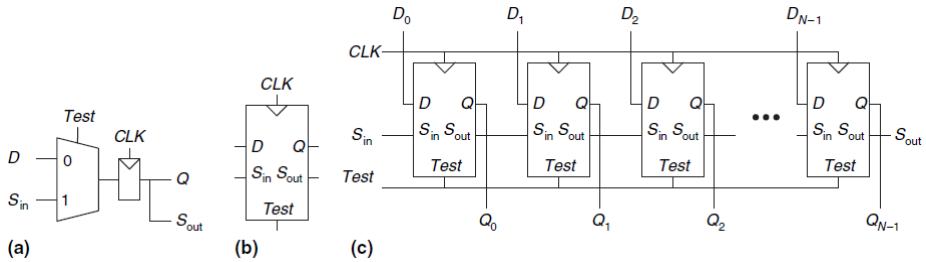


Figure 5. 41 Scannable flip-flop: (a) schematic, (b) symbol, and (c) N-bit scannable register

5.5 MEMORY ARRAYS

The previous sections introduced arithmetic and sequential circuits for manipulating data. Digital systems also require *memories* to store the data used and generated by such circuits.

Registers built from flip-flops are a kind of memory that stores small amounts of data. This section describes *memory arrays* that can **efficiently** store large amounts of data.

Overview

Figure 5.42 shows a generic symbol for a memory array. The memory is organized as a two-dimensional array of memory cells. The memory reads or writes the contents of one of the rows of the array. This row is specified by an ***Address***. The value read or written is called ***Data***. An array with N-bit addresses and M-bit data has 2^N rows and M columns. Each row of data is called a *word*. Thus, the array contains 2^N M-bit words.

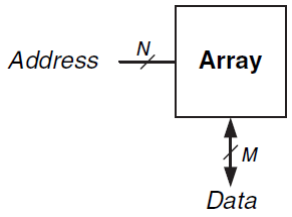


Figure 5. 42 Generic memory array symbol

Figure 5.43 shows a memory array with two address bits and three data bits. The two address bits specify one of the four rows (data words) in the array. Each data word is three bits wide.

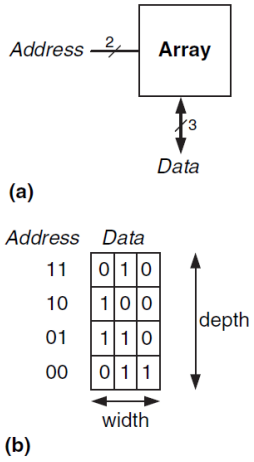


Figure 5. 43 4×3 memory array: (a) symbol, (b) function

The *depth* of an array is the number of rows, and the *width* is the number of columns, also called the word size. The size of an array is given as $depth \times width$. Figure 5.43 is a 4-word$\times$ 3-bit array, or simply 4×3 array. The symbol for an 1024-word$\times$ 32-bit array is shown in Figure 5.44. The total size of this array is 32 kilobits (Kb).

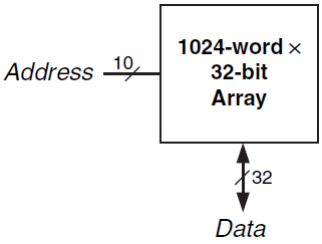


Figure 5. 44 32 Kb array: depth= $2^{10} = 1024$ words, width= 32 bits

Bit Cells

Memory arrays are built as an array of *bit cells*, each of which stores 1 bit of data. Figure 5.45 shows that each bit cell is connected to a *wordline* and a *bitline*. For each combination of address bits, the memory asserts a single wordline that activates the bit cells in that row. When the wordline is HIGH, the stored bit transfers to or from the bitline. Otherwise, the bitline is disconnected from the bit cell. The circuitry to store the bit varies with memory type.

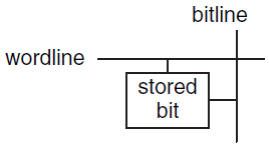


Figure 5. 45 Bit cell

To read a bit cell, the bitline is initially left floating (Z). Then the wordline is turned ON, allowing the stored value to drive the bitline to 0 or 1. To write a bit cell, the bitline is strongly driven to the desired value. Then the wordline is turned ON, connecting the bitline to the stored bit. The strongly driven bitline overpowers the contents of the bit cell, writing the desired value into the stored bit.

Organization

Figure 5.46 shows the internal organization of a 4×3 memory array.

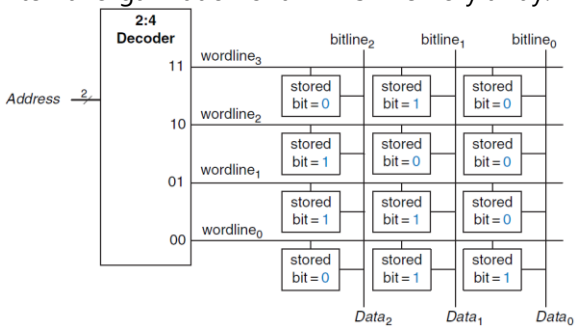


Figure 5. 46 4×3 memory array

Memory Ports

All memories have **one or more *ports***. Each port gives read and/or write access to one memory address. The previous examples were all single-ported memories.

Multiported memories can access several addresses **simultaneously**. Figure 5.47 shows a three-ported memory with two read ports and one write port. Port 1 reads the data from address $A1$ onto the read data output $RD1$. Port 2 reads the data from address $A2$ onto $RD2$. Port 3 writes the data from the write data input $WD3$ into address $A3$ on the rising edge of the clock if the write enable $WE3$ is asserted.

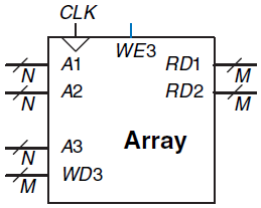


Figure 5. 47 Three-ported memory

Memory Types

Memory arrays are **specified** by their **size** (depth $\times$ width) and the number and type of **ports**. All memory arrays store data as an array of bit cells, but they **differ** in how they store bits.

Memories are **classified** based on how they store bits in the bit cell. The broadest classification is *random access memory* (*RAM*) versus *read only memory* (*ROM*). RAM is *volatile*, meaning that it loses its data when the power is turned off. ROM is *nonvolatile*, meaning that it retains its data indefinitely, even without a power source.

RAM is called ***random* access memory** because **any** data word is accessed with the **same** delay as any other. ROM is called ***read only* memory because**, historically, it could only be read but not written. These names are confusing, because ROMs are randomly accessed too. Worse yet, most modern ROMs can be written as well as read! The **important distinction** to remember is that RAMs are volatile and ROMs are nonvolatile.

The two major types of RAMs are *dynamic RAM* (*DRAM*) and *static RAM* (*SRAM*). Dynamic RAM stores data as a charge on a capacitor, whereas static RAM stores data using a pair of cross-coupled inverters.

Dynamic Random Access Memory (DRAM)

Dynamic RAM (*DRAM*, pronounced "dee-ram") stores a bit as the presence or absence of charge on a capacitor. Figure 5.48 shows a DRAM bit cell.

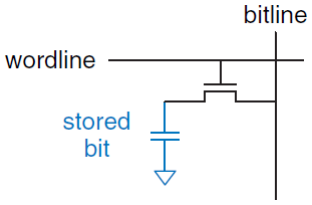


Figure 5. 48 DRAM bit cell

As shown in Figure 5.49(a), when the capacitor is charged to V_{DD}, the stored bit is 1; when it is discharged to GND (Figure 5.49(b)), the stored bit is 0. The capacitor node is *dynamic* because it is not actively driven HIGH or LOW by a transistor tied to V_{DD} or GND.

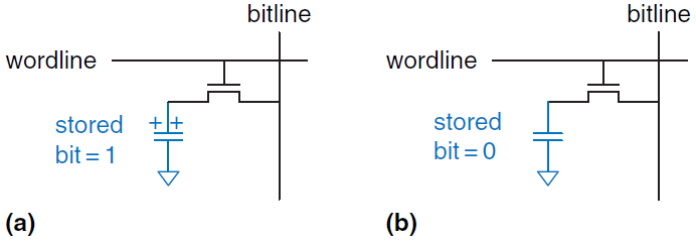


Figure 5. 49 DRAM stored values

Upon a read, data values are transferred from the capacitor to the bitline. Upon a write, data values are transferred from the bitline to the capacitor. Reading destroys the bit value stored on the capacitor, so the data word must be restored (rewritten) after each read. Even when DRAM is not read, the contents must be refreshed (read and rewritten) every few milliseconds, because the charge on the capacitor gradually leaks away.

Static Random Access Memory (SRAM)

Static RAM (*SRAM*, pronounced "es-ram") is *static* because stored bits do not need to be refreshed. Figure 5.50 shows an SRAM bit cell. Unlike DRAM, if noise degrades the value of the stored bit, the cross-coupled inverters restore the value.

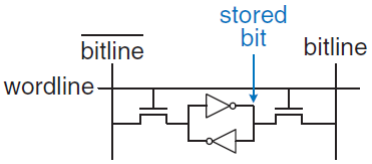


Figure 5. 50 SRAM bit cell

Area and Delay

Flip-flops, SRAMs, and DRAMs are all volatile memories, but each has different area and delay characteristics. Figure 5.51 shows a comparison of these three types of volatile memory.

Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

Figure 5. 51 Memory comparison

DRAM technologies such as *synchronous DRAM* (*SDRAM*) and *double data rate* (*DDR*) SDRAM have been developed to overcome the problem that, DRAM must refresh data periodically and after a read.

Memory latency and throughput **also depend** on memory size; larger memories tend to be slower than smaller ones if all else is the same.

Register Files

Digital systems often use a number of registers to store temporary variables. This group of

registers, called a *register file*, is usually built as a small, multiported **SRAM array**, because it is **more compact** than an array of **flip-flops**.

Figure 5.52 shows a 16-register × 32-bit three-ported register file built from a three-ported memory similar to that of Figure 5.47. The register file has two read ports ($A1/RD1$ and $A2/RD2$) and one write port ($A3/WD3$). The 4-bit addresses, $A1$, $A2$, and $A3$, can each access all $2^4 = 16$ registers. So, two registers can be read and one register written simultaneously.

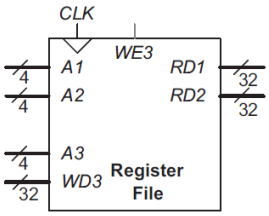


Figure 5. 52 16×32 register file with two read ports and one write port

Read Only Memory

Read only memory (*ROM*) stores a bit as the presence or absence of **a transistor**. Figure 5.53 shows a simple ROM bit cell. To read the cell, the bitline is weakly pulled HIGH. Then the wordline is turned ON. If the transistor is present, it pulls the bitline LOW. If it is absent, the bitline remains HIGH. Note that the ROM bit cell is **a combinational circuit** and has no state to "forget" if power is turned off.

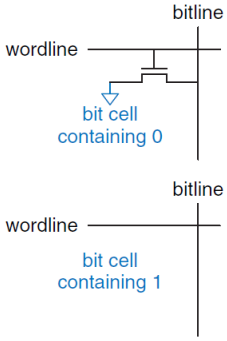


Figure 5. 53 ROM bit cells containing 0 and 1

The contents of a ROM can be indicated using *dot notation*. Figure 5.54 shows the dot notation for a 4-word × 3-bit ROM containing the data from Figure 5.43. A dot at the intersection of a row (wordline) and a column (bitline) indicates that the data bit is 1.

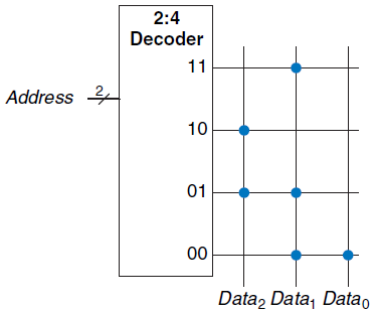


Figure 5. 54 4×3 ROM: dot notation

Conceptually, ROMs can be **built** using **two-level logic** with a group of **AND** gates followed by a group of **OR** gates. The AND gates produce all possible minterms and hence form a decoder. Figure 5.55 shows the ROM of Figure 5.54 built using a decoder and OR gates.

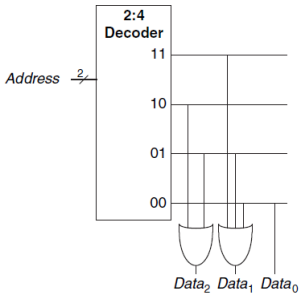


Figure 5. 55 4×3 ROM implementation using gates

The contents of the ROM bit cell in Figure 5.53 are specified during manufacturing by the presence or absence of a transistor in each bit cell. A *programmable ROM* (*PROM*, pronounced like the dance) places a transistor in every bit cell but provides a way to connect or disconnect the transistor to ground.

Figure 5.56 shows the bit cell for a *fuse-programmable ROM*. This is also called a one-time programmable ROM, because the fuse cannot be repaired once it is blown.

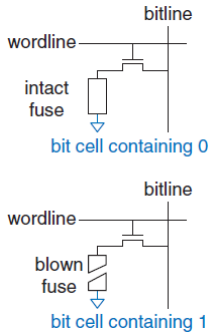


Figure 5. 56 Fuse-programmable ROM bit cell

Reprogrammable ROMs provide a reversible mechanism for connecting or disconnecting the transistor to GND. *Erasable PROMs* (*EPROMs*, pronounced "e-proms") replace the nMOS transistor and fuse with a *floating-gate transistor*. The floating gate is not physically attached to any other wires. When suitable high voltages are applied, electrons tunnel through an insulator onto the floating gate, turning on the transistor and connecting the bitline to the wordline (decoder output). When the EPROM is exposed to intense ultraviolet (UV) light for about half an hour, the electrons are knocked off the floating gate, turning the transistor off. These actions are called *programming* and *erasing*, respectively. *Electrically erasable PROMs* (*EEPROMs*, pronounced "e-e-proms" or "double-e proms") and *Flash* memory use similar principles but include circuitry on the chip for erasing as well as programming, so no UV light is necessary. EEPROM bit cells are individually erasable; Flash memory erases larger blocks of bits and is cheaper because fewer erasing circuits are needed.

In summary, modern ROMs are not really read only; they can be programmed (written) as well. The difference between RAM and ROM is that ROMs take a longer time to write but are nonvolatile.

Logic Using Memory Arrays

Although they are used primarily for data storage, memory arrays can also perform **combinational logic functions**.

Memory arrays used to perform logic are called ***lookup tables*** (***LUTs***) (Like multiplexer). Figure 5.57 shows a 4-word $\times$ 1-bit memory array used as a lookup table to perform the function $Y = AB$. Using memory to perform logic, the user can look up the output value for a given input combination (address). **Each address** corresponds to a row in the truth table, and **each data bit** corresponds to an output value.

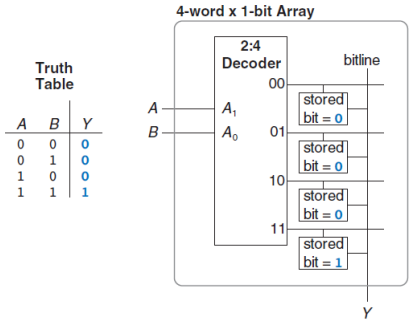


Figure 5. 57 4-word × 1-bit memory array used as a lookup table

Memory HDL

HDL Example 5.6 describes a 2^N-word × M-bit RAM. The RAM has a synchronous enabled write. In other words, writes occur on the **rising edge** of the clock if the write enable we is asserted. Reads occur immediately. When power is first applied, the contents of the RAM are **unpredictable**.

HDL Example 5.7 describes a 4-word × 3-bit ROM.

HDL Example 5.6: RAM

SystemVerilog

```
module ram #(parameter N = 6, M = 32)
            (input  logic         clk,
             input  logic         we,
             input  logic [N-1:0] adr,
             input  logic [M-1:0] din,
             output logic [M-1:0] dout);

  logic [M-1:0] mem [2**N-1:0];

  always_ff @(posedge clk)
    if (we) mem [adr] <= din;

  assign dout = mem[adr];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity ram_array is
  generic(N: integer := 6; M: integer := 32);
  port(clk,
       we:  in  STD_LOGIC;
       adr: in  STD_LOGIC_VECTOR(N-1 downto 0);
       din: in  STD_LOGIC_VECTOR(M-1 downto 0);
       dout: out STD_LOGIC_VECTOR(M-1 downto 0));
end;

architecture synth of ram_array is
  type mem_array is array ((2**N-1) downto 0)
      of STD_LOGIC_VECTOR (M-1 downto 0);
  signal mem: mem_array;
begin
  process(clk) begin
    if rising_edge(clk) then
      if we then mem(TO_INTEGER(adr)) <= din;
      end if;
    end if;
  end process;

  dout <= mem(TO_INTEGER(adr));
end;
```

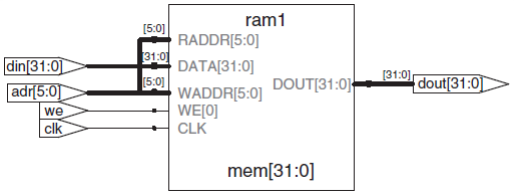


Figure 5. 58 Synthesized ram

HDL Example 5.7: ROM

SystemVerilog

```
module rom(input  logic [1:0] adr,
           output logic [2:0] dout);
  always_comb
    case(adr)
      2'b00: dout = 3'b011;
      2'b01: dout = 3'b110;
      2'b10: dout = 3'b100;
      2'b11: dout = 3'b010;
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity rom is
  port(adr:  in  STD_LOGIC_VECTOR(1 downto 0);
       dout: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of rom is
begin
  process(all) begin
    case adr is
      when "00" => dout <= "011";
      when "01" => dout <= "110";
      when "10" => dout <= "100";
      when "11" => dout <= "010";
    end case;
  end process;
end;
```

5.6 LOGIC ARRAYS

Like memory, gates can be organized into regular arrays. If the connections are made programmable, these logic arrays can be configured to perform **any function** without the user having to connect wires in specific ways.

This section introduces two types of logic arrays: programmable logic arrays (PLAs), and field

programmable gate arrays (FPGAs). PLAs, the older technology, perform only combinational logic functions. FPGAs can perform both combinational and sequential logic.

Programmable Logic Array

Programmable logic arrays (*PLAs*) implement two-level combinational logic in sum-of-products (SOP) form. PLAs are built from an AND array followed by an OR array, as shown in Figure 5.59. The inputs (in true and complementary form) drive an AND array, which produces implicants, which in turn are ORed together to form the outputs. An $M \times N \times P$-bit PLA has M inputs, N implicants, and P outputs.

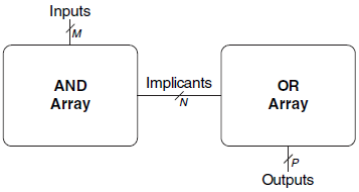


Figure 5. 59 $M \times N \times P$-bit PLA

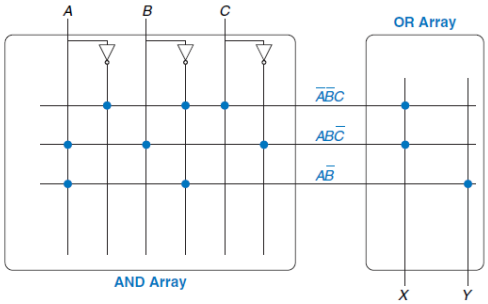


Figure 5. 60 $3 \times 3 \times 2$-bit PLA: dot notation

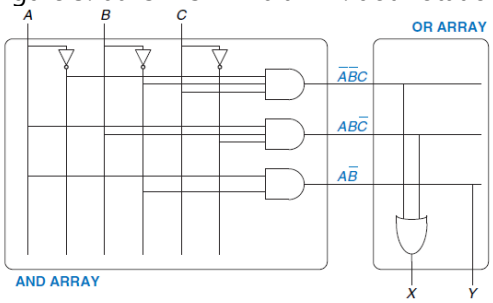


Figure 5. 61 $3 \times 3 \times 2$-bit PLA using two-level logic

Simple programmable logic devices (*SPLDs*) are souped-up PLAs that add registers and various other features to the basic AND/OR planes. However, SPLDs and PLAs have largely been displaced by FPGAs, which are more flexible and efficient for building large systems.

Field Programmable Gate Array

A *field programmable gate array* (FPGA) is an array of reconfigurable gates.

FPGAs are built as an array of configurable *logic elements* (*LEs*), also referred to as *configurable logic blocks* (*CLBs*). Each LE can be configured to perform combinational or sequential functions. Figure 5.62 shows a general block diagram of an FPGA. The LEs are surrounded by *input/output elements* (*IOEs*) for interfacing with the outside world. The IOEs connect LE inputs and outputs to pins on the chip package. LEs can connect to other LEs and IOEs through programmable routing channels.

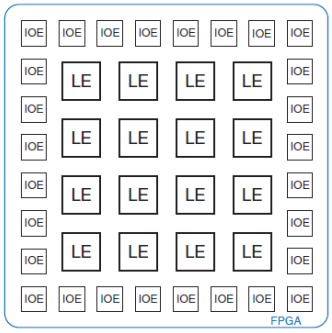


Figure 5. 62 General FPGA layout

Example 5.5: FUNCTIONS BUILT USING LEs (Page 277)

(A) data 1	(B) data 2	(C) data 3	data 4	(X) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0

(A) data 1	(B) data 2	data 3	data 4	(Y) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0

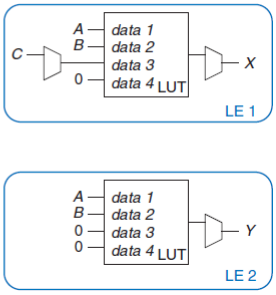


Figure 5. 63 LE configuration for two functions of up to four inputs each

(J) data 1	(K) data 2	(L) data 3	(M) data 4	(X) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(P) data 1	(Q) data 2	(R) data 3	(X) data 4	(Y) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

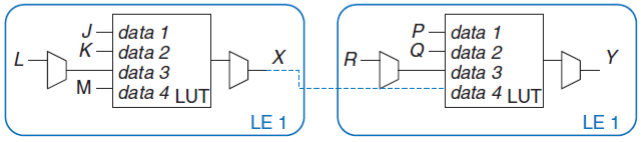


Figure 5. 64 LE configuration for one function of more than four inputs

data 1	data 2	(S_0) data 3	(S_1) data 4	(S_0') LUT output
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	0

data 1	data 2	(S_1) data 3	(S_0) data 4	(S_1') LUT output
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0

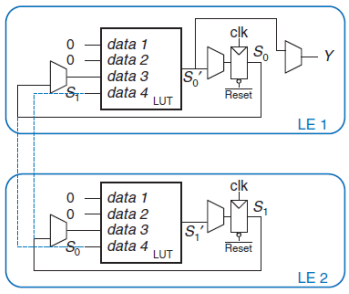


Figure 5. 65 LE configuration for FSM with two bits of state

Supplement materials

1) FPGA vs ASIC [Source]

FPGAs:

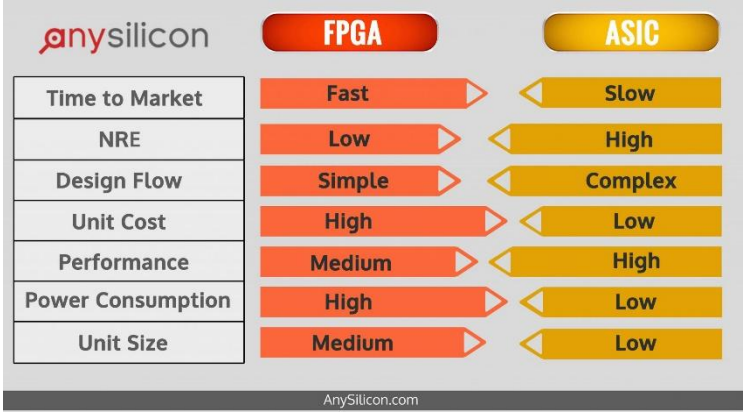
FPGAs (Field Programmable Gate Arrays) are chips created originally in 1985 to perform only digital functions but today they have already both analog and mixed signal blocks. It is

essentially a chip which can be programmed and reprogrammed to serve various purposes at any single point in time. A single chip is composed of thousands of units called logic blocks which are linked with programmable interconnects. Customers like to use FPGA because they are easy to use, and cost-effective reprogrammable devices. FPGA are known for their flexibility and their ability to be reprogrammed in the field. There is no need to have a full blown design flow and tooling, therefore the NRE (Non-Recurring Engineering) investment is very low and as consequence time to market is fast.

The **problem** is that FPGAs are not fully customable, for example, one cannot add a specific analog block or integrate RF capability into an FPGA, those functionalities need to be implemented by external ICs, thus making the product larger in size and more costly. Also, due to its flexibility it offers a few drawbacks such as higher internal delay, higher cost and limited analog functionality.

ASICs:
ASICs (Application Specific Integrated Circuits) are specific chips (as the name suggest) used to implement both analog and digital functionalities in high volume or high performance. It is a chip which serves the purpose for which it has been designed and cannot be reprogrammed or modified to perform another function or execute another application. ASICs are full custom therefore they require higher development costs in order to design and implement (NRE). Moreover, unlike the FPGAs chips, they are not reprogrammable and therefore a change requires again NRE payment.

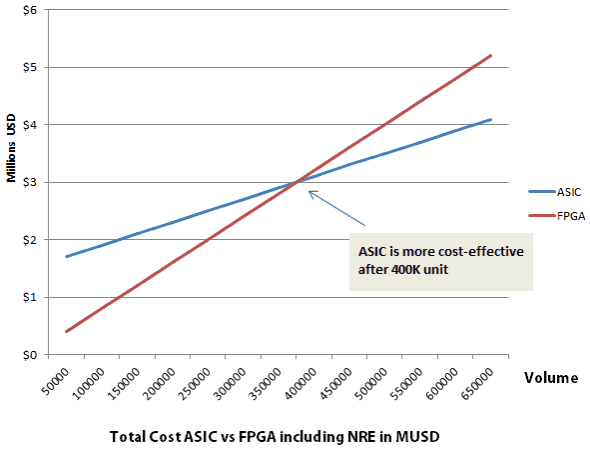
On the other side, however, ASICs are much denser, and one can integrate several different functionalities into one chip and therefore offer small size, low-power and low-cost solution.



Price Comparison FPGA vs ASIC:
An example,

ASIC NRE: $1.5M
FPGA NRE: $0

ASIC Unit Cost: $4
FPGA Unit Cost: $8



The graph clearly shows that after volume of 400K units, ASICs are starting to be **more** cost-effective. Therefore, despite the fact that the ASIC project requires $1.5M in NRE, after 400K unit the ASIC is starting to return the investment, compared to an FPGA.

Conclusion:

In conclusion, both ASIC and FPGA are technologies with different benefits, however their difference relies on costs, NRE, performance and flexibility. In general, we can say that for lower volumes' designs, FPGA flexibility allows to save costs and obtain better results; while ASICs chips are more efficient and cost effective on high volume applications.

Example 5.6: LE DELAY (Page 279)

Array Implementations

To minimize their size and cost, ROMs and PLAs commonly use pseudo-nMOS or dynamic circuits instead of conventional logic gates.

Figure 5.66 shows the dot notation for a 4×3-bit ROM that performs the functions of Figure 5.54, with the address inputs renamed A and B and the data outputs renamed X, Y, and Z. Remember that in pseudo-nMOS circuits, the weak pMOS transistor pulls the output HIGH *only if* there is no path to GND through the pulldown (nMOS) network.

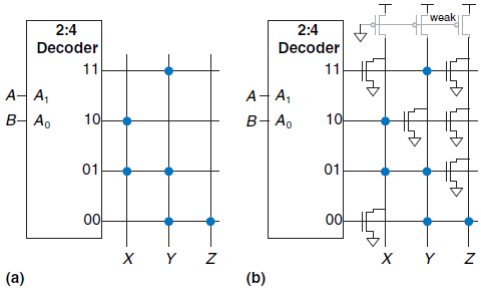


Figure 5. 66 ROM implementation: (a) dot notation, (b) pseudo-nMOS circuit

Pull-down transistors are placed at every junction without a dot.

PLAs can also be built using pseudo-nMOS circuits, as shown in Figure 5.67 for the PLA from Figure 5.60. Pull-down (nMOS) transistors are placed on the *complement* of dotted literals in the AND array and on dotted rows in the OR array. The columns in the OR array are sent through an inverter before they are fed to the output bits.

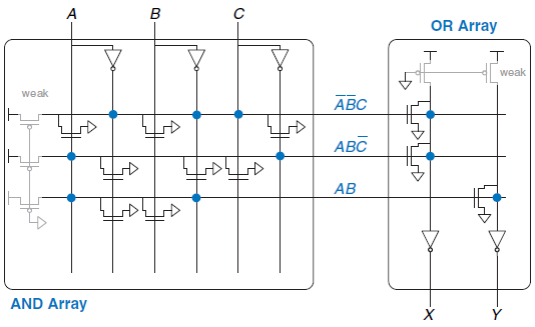


Figure 5. 67 $3 \times 3 \times 2$-bit PLA using pseudo-nMOS circuits

Exercises

Exercise 5.1: What is the delay for the following types of 64-bit adders? Assume that each two-input gate delay is 150 ps and that a full adder delay is 450 ps.

(a) a ripple-carry adder

$$t_{\text{ripple}} = Nt_{FA} = 64(450 \text{ ps}) = 28.8 \text{ ns}$$

(b) a carry-lookahead adder with 4-bit blocks

$$t_{\text{CLA}} = t_{\text{pg}} + t_{\text{pg\_block}} + \left(\frac{N}{k} - 1\right) t_{\text{AND\_OR}} + kt_{FA}$$

$$= 150 + 6 \times 150 + \left(\frac{64}{4} - 1\right) 300 + 4 \times 450 = 7.35 \text{ ns}$$

(c) a prefix adder

$$t_{\text{PA}} = t_{\text{pg}} + \log_2 N \left(t_{\text{pg\_prefix}}\right) + t_{\text{XOR}}$$

$$= 150 + 6(300) + 150 = 2.1 \text{ ns}$$

Exercise 5.5: The prefix network shown in Figure 5.7 uses black cells to compute all of the prefixes. Some of the block propagate signals are not actually necessary. Design a "gray cell" that receives G and P signals for bits $i{:}k$ and $k-1{:}j$ but produces only $G_{i:j}$, not $P_{i:j}$. Redraw the prefix network, replacing black cells with gray cells wherever possible.

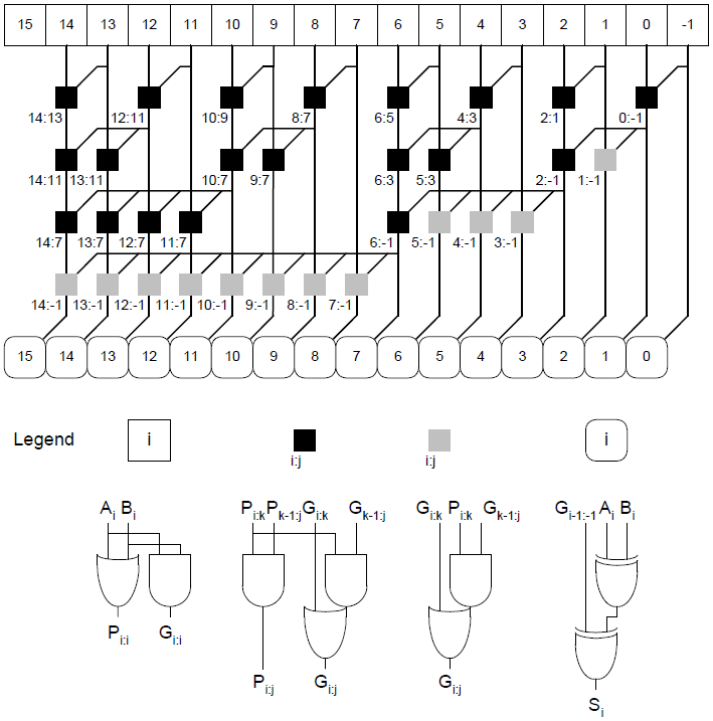


Figure 5. 68 16-bit prefix adder with "gray cells"

Exercise 5.6: The prefix network shown in Figure 5.7 is not the only way to calculate all of the prefixes in logarithmic time. The Kogge-Stone network is another common prefix network that performs the same function using a different connection of black cells. Research Kogge-Stone adders and draw a schematic similar to Figure 5.7 showing the connection of black cells in a

Kogge-Stone adder.

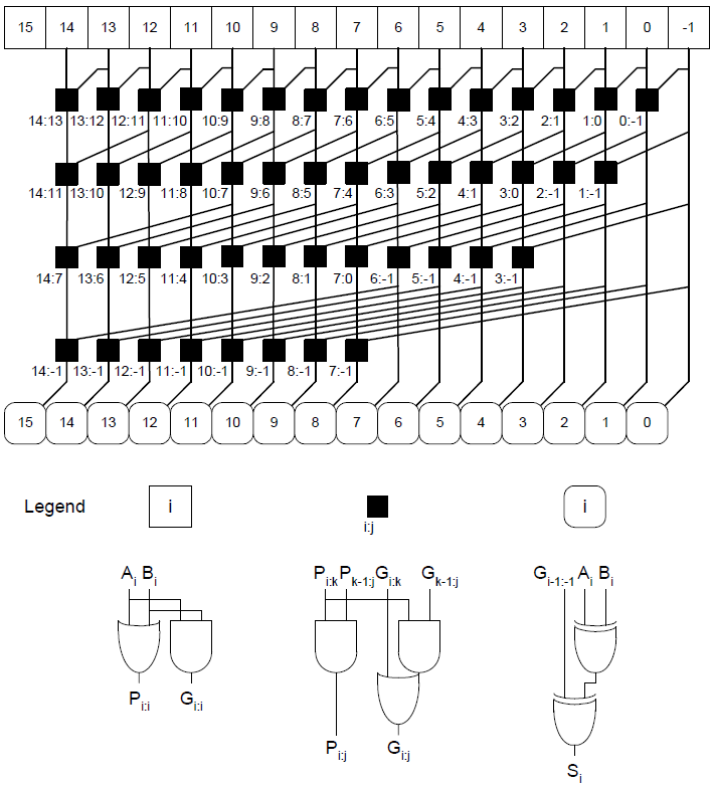
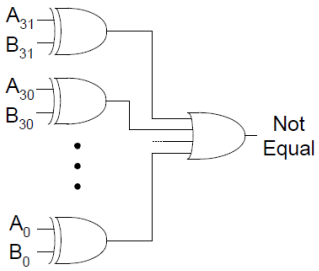


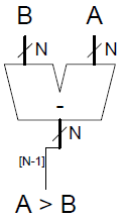
Figure 5. 69 Schematic of a 16-bit Kogge-Stone adder

Exercise 5.8: Design the following comparators for 32-bit unsigned numbers. Sketch the schematics.

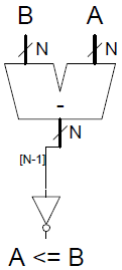
(a) not equal



(b) greater than or equal to (?)

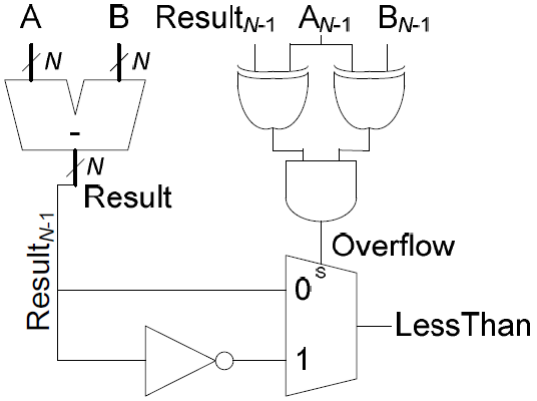


(c) less than (?)



Exercise 5.10: Modify the N-bit signed comparator of Figure 5.12 to correctly compute $A < B$ for all N-bit signed inputs A and B.

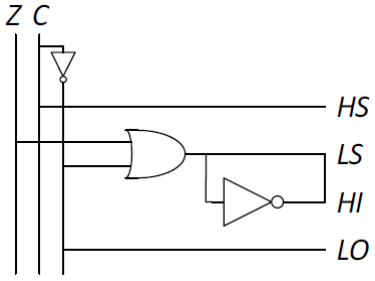
If no overflow occurs, connect the sign bit (i.e., most significant bit) of the result to the LessThan output.
If overflow occurs, invert the sign bit of the result and connect it to the LessThan output. Overflow occurs when (1) the two inputs have different signs, and (2) the sign of the subtraction result has a different sign than the A input, as shown in the figure below.



Exercise 5.15: Build an Unsigned Comparison Unit that compares two unsigned numbers A and B. The unit's input is the $ALUFlags$ signal (N, Z, C, V) from the ALU of Figure 5.16, with the ALU performing subtraction: $A - B$. The unit's outputs are HS, LS, HI, and LO, which indicate that A is higher than or the same as (HS), lower than or the same as (LS), higher (HI), or lower (LO) than B.
(a) Write minimal equations for HS, LS, HI, and LO in terms of N, Z, C, and V.

$$HS = C$$
$$LS = Z + \bar{C}$$
$$HI = \bar{Z}C = \overline{LS}$$
$$LO = \bar{C} = \overline{HS}$$

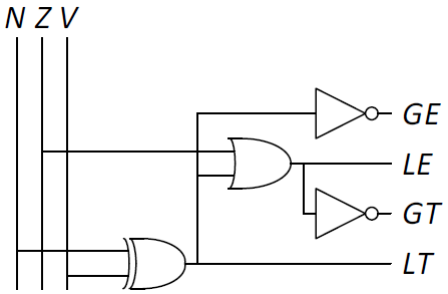
Proof:
For HS, we choose A and B like $(7,1),(6,6),(0,0)$ and we get $\bar{Z}C, ZC, ZC$, where N and V are unimportant. Then using K-map get $HS = C$.
For LS, we choose A and B like $(1,7),(6,6),(0,0)$ and we get $\bar{Z}\bar{C}, ZC, ZC$, where N and V are unimportant. Since we have HS, the LS can be written as $LS = \overline{HS} + ZC = \bar{C} + ZC = \bar{C} + Z$.
Others are similar.
(b) Sketch circuits for HS, LS, HI, and LO.



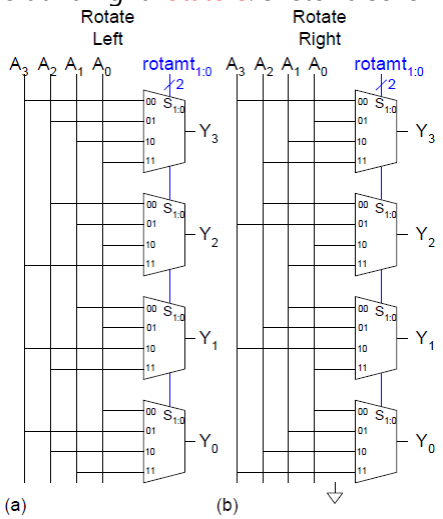
Exercise 5.16: Build a Signed Comparison Unit that compares two signed numbers A and B. The unit's input is the $ALUFlags$ signal (N, Z, C, V) from the ALU of Figure 5.16, with the ALU performing subtraction: $A - B$. The unit's outputs are GE, LE, GT, and LT, which indicate that A is greater than or equal to (GE), less than or equal to (LE), greater than (GT), or less than (LT) B.
(a) Write minimal equations for GE, LE, GT, and LT in terms of N, Z, C, and V.

$$GE = \overline{N \oplus V}$$
$$LE = Z + (N \oplus V)$$
$$GT = \overline{LE} = \bar{Z}\left(\overline{N \oplus V}\right)$$
$$LT = \overline{GE} = N \oplus V$$

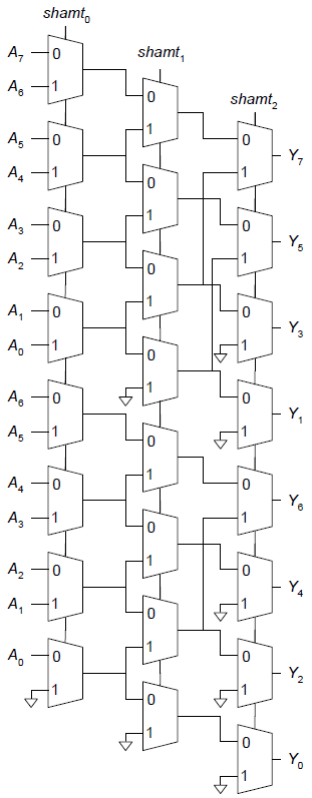
Proof:
For GE, we choose A and B like $(3,1),(2,2),(0,0),(-1,-4),(3,-4)$ and we get $\bar{N}\bar{Z}C\bar{V},\bar{N}ZC\bar{V},\bar{N}ZCV,\bar{N}\bar{Z}CV,N\bar{Z}\bar{C}V$. Then using K-map get $GE=\overline{N\oplus V}$. (use don't care)
For LE, we choose A and B like $(2,2),(0,0),(-4,-4)$ and we get $ZC\bar{N}\bar{V},ZC\bar{N}\bar{V},ZC\bar{N}\bar{V}$. Since we have GE, the LE can be written as $LE=\overline{GE}+Z=Z+(N\oplus V)$.
Others are similar.
(b) Sketch circuits for GE, LE, GT, and LT.



Exercise 5.18: Design 4-bit left and right rotators. Sketch a schematic of your design.



Exercise 5.19: Design an 8-bit left shifter using only 24 2:1 multiplexers. The shifter accepts an 8-bit input A and a 3-bit shift amount, $\mathrm{shamt}_{2:0}$. It produces an 8-bit output Y. Sketch the schematic.



Exercise 5.20: Explain how to build any N-bit shifter or rotator using only $N \log_2 N$ $2:1$ multiplexers.

Any N-bit shifter can be built by using $\log_2 N$ columns of 2-bit shifters. The first column of multiplexers shifts or rotates 0 to 1 bit, the second column shifts or rotates 0 to 3 bits, the following 0 to 7 bits, etc. until the final column shifts or rotates 0 to $N-1$ bits. The second column of multiplexers takes its inputs from the first column of multiplexers, the third column takes its input from the second column, and so forth. The 1-bit select input of each column is a single bit of the $shamt$ (shift amount) control signal, with the least significant bit for the leftmost column and the most significant bit for the right-most column.

Exercise 5.21: The ***funnel shifter*** in Figure 5.65 can perform any N-bit shift or rotate operation. It shifts a $2N$-bit input right by k bits. The output Y is the N least significant bits of the result. The most significant N bits of the input are called B and the least significant N bits are called C. By choosing appropriate values of B, C, and k, the funnel shifter can perform any type of shift or rotate. Explain what these values should be in terms of A, $shamt$, and N for

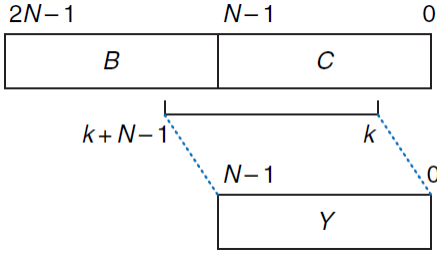


Figure 5. 70 Funnel shifter

(a) logical right shift of A by $shamt$

$$B = 0, C = A, k = shamt$$

(b) arithmetic right shift of A by $shamt$

$$B_i = A_{N-1}, i = 0,1,\cdots,N-1,, C = A, k = shamt$$

(c) left shift of A by $shamt$

$$B = A, C = 0, k = N - shamt$$

(d) right rotate of A by $shamt$

$$B = A, C = A, k = shamt$$

(e) left rotate of A by $shamt$

$$B = A, C = A, k = N - shamt$$

Exercise 5.22: Find the critical path for the 4×4 multiplier from Figure 5.21 in terms of an AND gate delay (t_{AND}) and an adder delay (t_{FA}) What is the delay of an $N \times N$ multiplier built in the same way?

$$t_{\text{pd\_MULT4}} = t_{\text{AND}} + 8t_{\text{FA}}$$

For $N = 1$, the delay is t_{AND}. For $N > 1$, an $N \times N$ multiplier has N-bit operands, N partial products, and $N - 1$ stages of 1-bit adders. The delay is through the AND gate, then through all N adders in the first stage, and finally through 2 adder delays for each of the remaining stages. So the propagation is:

$$t_{\text{pd\_MULT}N} = t_{\text{AND}} + [N + 2(N - 1 - 1)]t_{\text{FA}}$$

Exercise 5.25: A ***sign extension unit*** extends a two's complement number from M to N ($N > M$) bits by copying the most significant bit of the input into the upper bits of the output. It receives an M-bit input A and produces an N-bit output Y. Sketch a circuit for a sign extension unit with a 4-bit input and an 8-bit output.

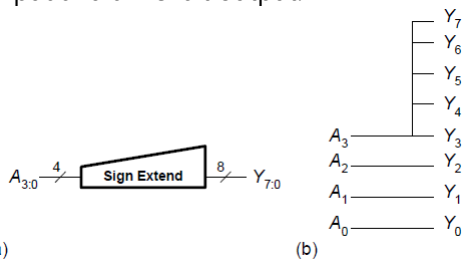


Figure 5. 71 Sign extension unit (a) symbol, (b) underlying hardware

Exercise 5.26: A ***zero extension unit*** extends an unsigned number from M to N bits ($N > M$) by putting zeros in the upper bits of the output. Sketch a circuit for a zero extension unit with a 4-bit input and an 8-bit output.

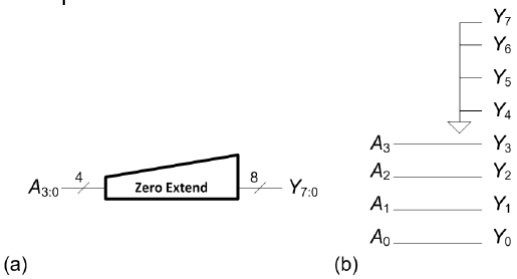


Figure 5. 72 Zero extension unit (a) symbol, (b) underlying hardware

Exercise 5.27: Compute $111001.000_2/001100.000_2$ in binary using the standard division algorithm from elementary school. Show your work.

```
          100.110
1100 | 111001.000
     -1100↓↓↓ |
      001001 0 |
     -   110 0↓
         11 00
     -   11 00
             0
```

Exercise 5.28: What is the range of numbers that can be represented by the following number systems?

(a) 24-bit unsigned fixed-point numbers with 12 integer bits and 12 fraction bits

$$\left[0, \left(2^{12} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$$

(b) 24-bit sign and magnitude fixed-point numbers with 12 integer bits and 12 fraction bits

$$\left[-\left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right), \left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$$

(c) 24-bit two's complement fixed-point numbers with 12 integer bits and 12 fraction bits

$$\left[-\left(2^{11}+\frac{2^{12}-1}{2^{12}}\right),\left(2^{11}-1+\frac{2^{12}-1}{2^{12}}\right)\right]$$

Exercise 5.30: Express the following base 10 numbers in 12-bit fixed-point sign/ magnitude format with six integer bits and six fraction bits. Express your answer in hexadecimal.

(a) -30.5

$$111110.100000 = 0\text{xFA0}$$

(b) 16.25

$$010000.010000 = 0\text{x}410$$

(c) -8.078125

$$101000.000101 = 0\text{xA05}$$

Exercise 5.34: Express the base 10 numbers in Exercise 5.30 in IEEE 754 single-precision floating-point format. Express your answer in hexadecimal.

(a) $-11110.1 = -1.11101 \times 2^4$

Thus, the biased exponent $= 127 + 4 = 131 = 1000\ 0011_2$

In IEEE 754 single-precision floating-point format:

$$1\ 1000\ 0011\ 1110\ 100\ 0000\ 0000\ 0000\ 0000 = 0\text{xC1F40000}$$

(b) $10000.01 = 1.000001 \times 2^4$

Thus, the biased exponent $= 127 + 4 = 131 = 1000\ 0011_2$

In IEEE 754 single-precision floating-point format:

$$0\ 1000\ 0011\ 0000\ 010\ 0000\ 0000\ 0000\ 0000 = 0\text{x}41820000$$

(c) $-1000.000101 = -1.000000101 \times 2^3$

Thus, the biased exponent $= 127 + 3 = 130 = 1000\ 0010_2$

In IEEE 754 single-precision floating-point format:

$$1\ 1000\ 0010\ 0000\ 0010\ 100\ 0000\ 0000\ 0000 = 0\text{xC}1014000$$

Exercise 5.37: When adding two floating-point numbers, the number with the smaller exponent is shifted. Why is this? Explain in words and give an example to justify your explanation.

When adding two floating point numbers, the number with the smaller exponent is shifted to preserve the most significant bits. For example, suppose we were adding the two floating point numbers 1.0×2^0 and 1.0×2^{-27}. We make the two exponents equal by shifting the second number right by 27 bits. Because the mantissa is limited to 24 bits, the second number ($1.000\ 0000\ 0000\ 0000\ 0000 \times 2^{-27}$) becomes $0.000\ 0000\ 0000\ 0000\ 0000 \times 2^0$, because the 1 is shifted off to the right. If we had shifted the number with the larger exponent (1.0×2^0) to the left, we would have shifted off the more significant bits (on the order of 2^0 instead of on the order of 2^{-27}).

Exercise 5.40: Expand the steps in section 5.3.2 for performing floating-point addition to work for negative as well as positive floating-point numbers.

We only need to change step 5.

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. If one number is negative: Subtract it from the other number. If the result is negative, take the absolute value of the result and make the sign bit 1.

If both numbers are negative: Add the numbers and make the sign bit 1.

If both numbers are positive: Add the numbers and make the sign bit 0.

6. Normalize mantissa and adjust exponent if necessary.
7. Round result
8. Assemble exponent and fraction back into floating-point number

Exercise 5.42: Consider the following decimal numbers: 245 and 0.0625.

(a) Write the two numbers using single-precision floating-point notation. Give your answers in hexadecimal.

$$245 = 1.1110101 \times 2^7$$
$$= 0\text{x}43750000$$

$$0.0625 = 1.0 \times 2^{-4}$$
$$= 0\text{x}3\text{D}800000$$

(b) Perform a magnitude comparison of the two 32-bit numbers from part (a). In other words, interpret the two 32-bit numbers as two's complement numbers and compare them. Does the integer comparison give the correct result?

$0\text{x}43750000$ is greater than $0\text{x}3\text{D}800000$, so magnitude comparison gives the correct result.

(c) You decide to come up with a new single-precision floating-point notation. Everything is the same as the IEEE 754 single-precision floating-point standard, except that you represent the exponent using two's complement instead of a bias. Write the two numbers using your new standard. Give your answers in hexadecimal.

$$1.1110101 \times 2^7 = 0\text{x}03\text{F}50000$$
$$1.0 \times 2^{-4} = 0\text{x}7\text{E}000000$$

(d) Does integer comparison work with your new floating-point notation from part (d)?

No, integer comparison no longer works. $0\text{x}7\text{E}000000 > 0\text{x}03\text{F}50000$ (indicating that 1.0×2^{-4} is greater than 1.1110101×2^7, which is incorrect.)

(e) Why is it convenient for integer comparison to work with floating-point numbers?

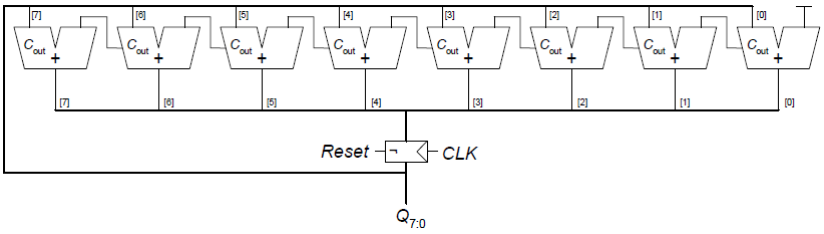
It is convenient for integer comparison to work with floating-point numbers because then the computer can compare numbers without needing to extract the mantissa, exponent, and sign.

Exercise 5.44: In this problem, you will explore the design of a 32-bit floating-point multiplier. The multiplier has two 32-bit floating-point inputs and produces a 32-bit floating-point output. You may consider positive numbers only and use round toward zero (truncate). You may also ignore the special cases given in Figure 5.31.

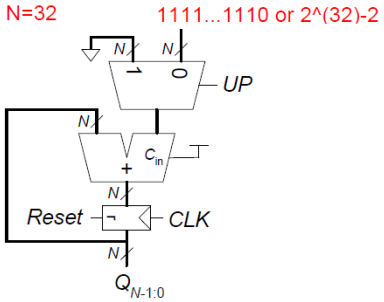
(a) Write the steps necessary to perform 32-bit floating-point multiplication.

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Add exponents.
4. Multiply mantissas.
5. Round result and truncate mantissa to 24 bits.
6. Assemble exponent and fraction back into floating-point number

Exercise 5.46: An incrementer adds 1 to an N-bit number. Build an 8-bit incrementer using half adders.



Exercise 5.47: Build a 32-bit synchronous ***Up/Down counter***. The inputs are *Reset* and *Up*. When *Reset* is 1, the outputs are all 0. Otherwise, when $Up = 1$, the circuit counts up, and when $Up = 0$, the circuit counts down.



Exercise 5.50: An N-bit ***Johnson counter*** consists of an N-bit shift register with a reset signal. The output of the shift register (S_{out}) is inverted and fed back to the input (S_{in}). When the counter is reset, all of the bits are cleared to 0.

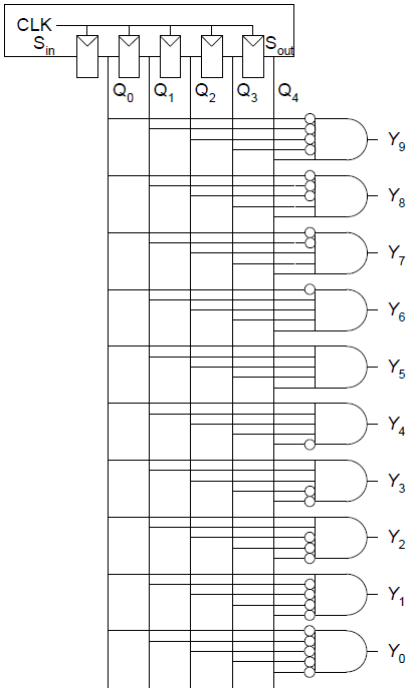
(a) Show the sequence of outputs, $Q_{3:0}$, produced by a 4-bit Johnson counter starting immediately after the counter is reset.

0000
1000
1100
1110
1111
0111
0011
0001
(repeat)

(b) How many cycles elapse until an N-bit Johnson counter repeats its sequence? Explain.

$2N$. 1's shift into the left-most bit for N cycles, then 0's shift into the left bit for N cycles. Then the process repeats.

(c) Design a decimal counter using a 5-bit Johnson counter, ten AND gates, and inverters. The decimal counter has a clock, a reset, and ten one-hot outputs $Y_{9:0}$. When the counter is reset, Y_0 is asserted. On each subsequent cycle, the next output should be asserted. After ten cycles, the counter should repeat. Sketch a schematic of the decimal counter.



(d) What advantages might a Johnson counter have over a conventional counter?

The counter uses less hardware and could be faster because it has a short critical path (a single inverter delay).

Exercise 5.52: The English language has a good deal of redundancy that allows us to reconstruct garbled transmissions. Binary data can also be transmitted in redundant form to allow error correction. For example, the number 0 could be coded as 00000 and the number 1 could be coded as 11111. The value could then be sent over a noisy channel that might flip up to two of the bits. The receiver could reconstruct the original data because a 0 will have at least three of the five received bits as 0's; similarly a 1 will have at least three 1's.

(a) Propose an encoding to send 00, 01, 10, or 11 encoded using five bits of information such that all errors that corrupt one bit of the encoded data can be corrected. Hint: the encodings 00000 and 11111 for 00 and 11, respectively, will not work.

value $a_{1:0}$	encoding $y_{4:0}$
00	00001
01	01010
10	10100
11	11111

The first two pairs of bits in the bit encoding repeat the value. The last bit is the XNOR of the two input values.

(b) Design a circuit that receives your five-bit encoded data and decodes it to 00, 01, 10, or 11, even if one bit of the transmitted data has been changed.

This circuit can be built using a 32×2-bit memory array, with the contents.

address $a_{4:0}$	data $d_{1:0}$
00001	00
00000	00
00011	00
00101	00
01001	00
10001	00
01010	01
01011	01
01000	01
01110	01
00010	01
11010	01
10100	10
10101	10
10110	10
10000	10
11100	10
00100	10
11111	11
11110	11
11101	11
11011	11
10111	11

address $a_{4:0}$	data $d_{1:0}$
01111	11
others	XX

(c) Suppose you wanted to change to an alternative 5-bit encoding. How might you implement your design to make it easy to change the encoding without having to use different hardware?

The implementation shown in part (b) allows the encoding to change easily. Each memory address corresponds to an encoding, so simply store different data values at each memory address to change the encoding.

Exercise 5.57: Specify the size of a ROM that you could use to program each of the following combinational circuits. Is using a ROM to implement these functions a good design choice? Explain why or why not.

(a) a 16-bit adder/subtractor with C_{in} and C_{out}

Number of inputs $= 2 \times 16 + 1 = 33$

Number of outputs $= 16 + 1 = 17$

Thus, this would require a $2^{33} \times 17$-bit ROM.

(b) an 8×8 multiplier

Number of inputs $= 16$

Number of outputs $= 16$

Thus, this would require a $2^{16} \times 16$-bit ROM.

(c) a 16-bit priority encoder

Number of inputs $= 16$
Number of outputs $= 4$
Thus, this would require a $2^{16} \times 4$-bit ROM.
All of these implementations are not good design choices. They could all be implemented in a smaller amount of hardware using discrete gates.

CHAPTER 6: Architecture

6.1 INTRODUCTION

The previous chapters introduced digital design principles and building blocks. In this chapter, we jump up a few levels of abstraction to define the architecture of a computer. The ***architecture*** is the **programmer's view** of a computer. It is defined by the instruction set (language) and operand locations (registers and memory). Many different architectures exist, such as ARM, x86, MIPS, SPARC, and PowerPC.

Supplement materials

1) The **Von Neumann** Model/Architecture:

Von Neumann model is also called stored program computer (instructions in memory). It has two key properties:

- Stored program

Instructions stored in a linear memory array;

Memory is unified between instructions and data, and the interpretation of a stored value **depends** on the control signals;

- Sequential instruction processing

One instruction processed (fetched, executed, completed) at a time;

Program counter (instruction pointer) identifies the current instruction;

Program counter is advanced sequentially except for control transfer instructions;

Note:

Computer Architecture is the science and art of designing computing platforms.

The first step in understanding any computer architecture is to learn its language. The words in a computer's language are called ***instructions***. The computer's vocabulary is called the *instruction set*. All programs running on a computer use the same instruction set. Computer instructions indicate both the operation to perform and the operands to use. The operands may come from memory, from registers, or from the instruction itself.

Computer hardware understands only 1's and 0's, so instructions are **encoded** as binary numbers in a format called **machine language**. The ARM architecture represents each instruction as a 32-bit word. Microprocessors are digital systems that read and execute machine language instructions. However, humans consider reading machine language to be **tedious**, so we prefer to represent the instructions in a symbolic format called **assembly language**.

The instruction sets of different architectures are more like different dialects than different languages. Almost all architectures define basic instructions, such as add, subtract, and branch, that operate on memory or registers.

A computer architecture does **not** define the underlying hardware implementation. Often, many different hardware implementations of a single architecture exist. For example, Intel and Advanced Micro Devices (AMD) both sell various microprocessors belonging to the same x86 architecture. The specific arrangement of registers, memories, ALUs, and other building blocks to form a microprocessor is called the **microarchitecture**.

6.2 ASSEMBLY LANGUAGE

Assembly language is the human-readable representation of the computer's native language. Each assembly language instruction specifies both the operation to perform and the operands on which to operate. We will then define the ARM instruction operands: registers, memory, and constants.

Instructions

The most common operation computers perform is addition. Code Example 6.1 shows code for adding variables b and c and writing the result to a.

Code Example 6.1: ADDITION (Page 297)

High-Level Code	ARM Assembly Code
`a = b + c;`	`ADD a, b, c`

The first part of the assembly instruction, ADD, is called the *mnemonic* and indicates what operation to perform. The operation is performed on b and c, the *source operands*, and the result is written to a, the *destination operand*.

Code Example 6.2: SUBTRACTION (Page 297)

High-Level Code	ARM Assembly Code
`a = b - c;`	`SUB a, b, c`

The instruction format is the same as the ADD instruction except for the operation specification, SUB. This consistent instruction format is an example of the first design principle:

Design Principle 1: Regularity supports simplicity.

More complex high-level code translates into multiple ARM instructions, as shown in Code Example 6.3.

Code Example 6.3: MORE COMPLEX CODE (Page 298)

High-Level Code	ARM Assembly Code

`a = b + c - d;  // single-line comment`
`/* multiple-line`
`comment */`
	`ADD t, b, c  ; t = b + c`
	`SUB a, t, d  ; a = t - d`

In the high-level language examples, single-line comments begin with // and continue until the end of the line. Multiline comments begin with /* and end with */. In ARM assembly language, only single-line comments are used. They begin with a semicolon (;) and continue until the end of the line. Using multiple assembly language instructions to perform more complex operations is an example of the second design principle of computer architecture:

Design Principle 2: Make the common case fast.

The ARM instruction set makes the common case fast by including only simple, commonly used instructions. The number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small, and fast. More elaborate operations that are less common are performed using sequences of multiple simple instructions. Thus, ARM is a *reduced instruction set computer* (RISC) architecture. Architectures with many complex instructions, such as Intel's x86 architecture, are *complex instruction set computers* (CISC).

A RISC architecture minimizes the hardware complexity and the necessary instruction encoding by keeping the set of distinct instructions small.

Operands: Registers, Memory, and Constants

An instruction operates on *operands*. But computers operate on 1's and 0's, not variable names. The instructions need a physical location from which to retrieve the binary data. Operands can be stored in registers or memory, or they may be constants stored in the instruction itself. Computers use various locations to hold operands in order to optimize for speed and data capacity. Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data. Additional data must be accessed from memory, which is large but slow. ARM (prior to ARMv8) is called a 32-bit architecture because it operates on 32-bit data.

Registers

The ARM architecture uses 16 registers, called the *register set* or *register file*. The fewer the registers, the faster they can be accessed. This leads to the third design principle:

Design Principle 3: Smaller is faster.

Code Example 6.4 shows the ADD instruction with register operands. ARM register names are preceded by the letter 'R'. The variables a, b, and c are arbitrarily placed in R0, R1, and R2. The name R1 is pronounced "register 1" or "R1" or "register R1".

Code Example 6.4: REGISTER OPERANDS (Page 299)

High-Level Code	ARM Assembly Code
`a = b + c;`	`; R0 = a, R1 = b, R2 = c` `ADD R0, R1, R2 ; a = b + c`

Code Example 6.5: TEMPORARY REGISTERS (Page 299)

High-Level Code	ARM Assembly Code

`a = b + c - d;`	`; R0 = a, R1 = b, R2 = c, R3 = d; R4 = t`
	`ADD R4, R1, R2 ; t = b + c`
	`SUB R0, R4, R3 ; a = t - d`

Example 6.1: TRANSLATING HIGH-LEVEL CODE TO ASSEMBLY LANGUAGE (Page 300)

The Register Set

Figure 6.1 lists the name and use for each of the 16 ARM registers. R0–R12 are used for storing variables; R0–R3 also have special uses during procedure calls. R13–R15 are also called SP, LR, and PC.

Complementary materials

(From ARM assembler user guide)

ARM processors provide general-purpose and special-purpose registers. Some additional registers are available in privileged execution modes.

In all ARM processors, the following registers (there are 17 registers) are available and accessible in any processor mode:

- 13 general-purpose registers R0-R12.
- One *Stack Pointer* (SP). (or R13)
- One *Link Register* (LR). (or R14)
- One *Program Counter* (PC). (or R15)
- One *Application Program Status Register* (APSR). (or R16)

Table 2-2 Predeclared core registers

Register names	Meaning
`r0-r15` and `R0-R15`	General purpose registers.
`a1-a4`	Argument, result or scratch registers. These are synonyms for `R0` to `R3`
`v1-v8`	Variable registers. These are synonyms for `R4` to `R11`.
`sb` and `SB`	Static base register. This is a synonym for `R9`.
`ip` and `IP`	Intra procedure call scratch register. This is a synonym for `R12`.
`sp` and `SP`	Stack pointer. This is a synonym for `R13`.
`lr` and `LR`	Link register. This is a synonym for `R14`.
`pc` and `PC`	Program counter. This is a synonym for `R15`.

Name	Use
R0	Argument / return value / temporary variable
R1–R3	Argument / temporary variables
R4–R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Figure 6. 1 ARM register set

Constants/Immediates

In addition to register operations, ARM instructions can use **constant** or ***immediate*** operands. These constants are called immediates, because their values are immediately available from the instruction and do not require a register or memory access. In assembly code, the immediate is

preceded by the # symbol and can be written in decimal or hexadecimal. Hexadecimal constants in ARM assembly language start with 0x, as they do in C. Immediates are unsigned 8- to 12-bit numbers with a peculiar encoding described in Section 6.4.

Code Example 6.6: IMMEDIATE OPERANDS (Page 301)

High-Level Code	ARM Assembly Code

`a = a + 4;`
`b = a – 12;`
	`; R7 = a, R8 = b`
	`ADD R7, R7, #4 ; a = a + 4`
	`SUB R8, R7, #0xC ; b = a – 12`

Code Example 6.7: INITIALIZING VALUES USING IMMEDIATES (Page 301)

High-Level Code	ARM Assembly Code

`i = 0;`
`x = 4080;`
	`; R4 = i, R5 = x`
	`MOV R4, #0 ; i = 0`
	`MOV R5, #0xFF0 ; x = 4080`

The move instruction (MOV) is a useful way to initialize register values. MOV can also take a register source operand. For example, MOV R1, R7 copies the contents of register R7 into R1.

Memory

In the ARM architecture, instructions operate **exclusively** on registers, so data stored in memory **must be moved** to a register before it can be processed. By using a **combination** of memory and registers, a program can access a large amount of data fairly quickly. The ARM architecture uses 32-bit memory addresses and 32-bit data words.

ARM uses a *byte-addressable* memory. That is, **each** byte in memory has a **unique** address, as shown in Figure 6.2. By convention, memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top.

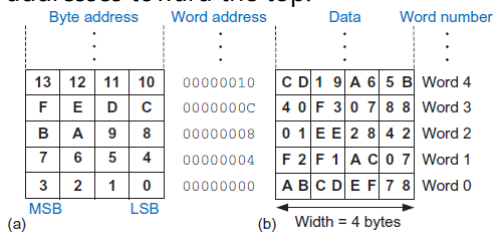


Figure 6. 2 ARM byte-addressable memory showing: (a) byte address and (b) data

ARM provides the *load register* instruction, LDR, to read a data word from memory into a register. Code Example 6.8 loads memory word 2 into a (R7). In C, the number inside the brackets is the *index* or word number, which we discuss further in Section 6.3.6. The LDR instruction specifies the memory address using a *base register* (R5) and an *offset* (8). The word address is four times the word number. The memory address is formed by adding the contents of the base register (R5) and the offset.

Code Example 6.8: READING MEMORY (Page 302)

High-Level Code	ARM Assembly Code

`a = mem[2];`	`; R7 = a`
	`MOV R5, #0 ; base address = 0`
	`LDR R7, [R5, #8] ; R7 <= data at memory address (R5+8)`

ARM uses the *store register* instruction, STR, to write a data word from a register into memory. Code Example 6.9 writes the value 42 from register R9 into memory word 5.

Code Example 6.9: WRITING MEMORY (Page 302)

High-Level Code	ARM Assembly Code

`mem[5] = 42;`	`MOV R1, #0 ; base address = 0`
	`MOV R9, #42`
	`STR R9, [R1, #0x14] ; value stored at memory address (R1+20) = 42`

Byte-addressable memories are organized in a big-endian or little-endian fashion, as shown in

Figure 6.3. In **both** formats, a 32-bit word's most significant byte (MSB) is on the **left** and the least significant byte (LSB) is on the **right**. Word addresses are the **same** in both formats and refer to the **same** four bytes. **Only** the addresses of **bytes** within a word differ. In *big-endian* machines, bytes are numbered **starting** with 0 at the big (most significant) end. In *little-endian* machines, bytes are numbered **starting** with 0 at the little (least significant) end.

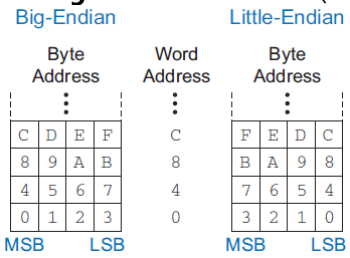
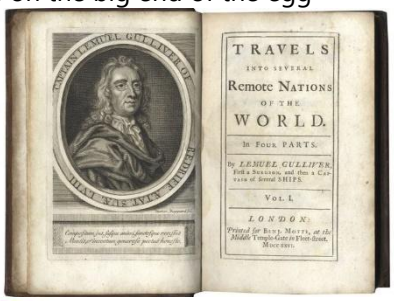


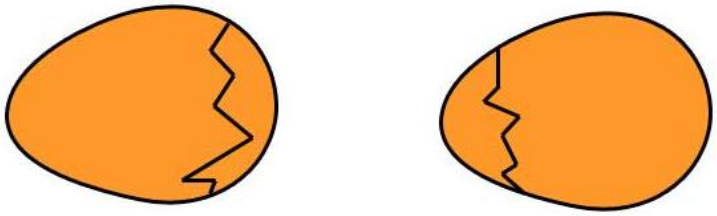
Figure 6. 3 Big-endian and little-endian memory addressing

Supplement material

Jonathan Swift's **Gulliver's Travels**:

- Little Endians broke their eggs on the little end of the egg
- Big Endians broke their eggs on the big end of the egg





BIG ENDIAN - The way people always broke their eggs in the Lilliput land

LITTLE ENDIAN - The way the king then ordered the people to break their eggs

The address of the current instruction is stored in the program counter (PC):

- If **word**-addressable memory, the processor increments the PC by 1 (in LC-3)
- If **byte**-addressable memory, the processor increments the PC by the word length (4 in MIPS)

Byte address is calculated as: word_address * bytes/word.

6.3 PROGRAMMING

Data-processing Instructions

The ARM architecture defines a variety of *data-processing* instruction (often called logical and arithmetic instructions in other architectures).

Logical Instructions

ARM *logical operations* include AND, ORR (OR), EOR (XOR), and BIC (bit clear). These each operate bitwise on two sources and write the result to a destination register. The first source is always a register and the second source is either an immediate or another register. Another logical operation, MVN (MoVe and Not), performs a bitwise NOT on the second source (an immediate or register) and writes the result to the destination register. Figure 6.4 shows examples of these operations on the two source values 0x46A1F1B7 and 0xFFFF0000.

	Source registers			
R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
AND R3, R1, R2	R3	0100 0110	1010 0001	0000 0000	0000 0000
ORR R4, R1, R2	R4	1111 1111	1111 1111	1111 0001	1011 0111
EOR R5, R1, R2	R5	1011 1001	0101 1110	1111 0001	1011 0111
BIC R6, R1, R2	R6	0000 0000	0000 0000	1111 0001	1011 0111
MVN R7, R2	R7	0000 0000	0000 0000	1111 1111	1111 1111

Figure 6. 4 Logical operations

The bit clear (BIC) instruction is useful for masking bits (i.e., forcing unwanted bits to 0). BIC R6, R1, R2 computes R1 AND NOT R2.

The ORR instruction is useful for combining bitfields from two registers.

Shift Instructions

Shift instructions shift the value in a register left or right, dropping bits off the end. The rotate instruction rotates the value in a register right by up to 31 bits. We refer to both shift and rotate generically as shift operations. ARM shift operations are LSL (logical shift left), LSR (logical shift right), ASR (arithmetic shift right), and ROR (rotate right). There is no ROL instruction because left rotation can be performed with a right rotation by a complementary amount ($n_{\text{complementary}} = 32 - n_{\text{right}}$).

The amount by which to shift can be an immediate or a register.

Figure 6.5 shows the assembly code and resulting register values for LSL, LSR, ASR, and ROR when shifting by an immediate value. Shifting a value left by N is equivalent to multiplying it by 2^N. Likewise, arithmetically shifting a value right by N is equivalent to dividing it by 2^N. Logical shifts are also used to extract or assemble bitfields.

	Source register			
R5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
LSL R0, R5, #7	R0	1000 1110	0000 1000	0111 0011	1000 0000
LSR R1, R5, #17	R1	0000 0000	0000 0000	0111 1111	1000 1110
ASR R2, R5, #3	R2	1111 1111	1110 0011	1000 0010	0001 1100
ROR R3, R5, #21	R3	1110 0000	1000 0111	0011 1111	1111 1000

Figure 6. 5 Shift instructions with immediate shift amounts

Figure 6.6 shows the assembly code and resulting register values for shift operations where the shift amount is held in a register, R6. This instruction uses the *register-shifted register* addressing mode, where one register (R8) is shifted by the amount (20) held in a second register (R6).

	Source registers			
R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code		Result			
LSL R4, R8, R6	R4	0110 1110	0111 0000	0000 0000	0000 0000
ROR R5, R8, R6	R5	1100 0001	0110 1110	0111 0000	1000 0001

Figure 6. 6 Shift instructions with register shift amounts

Multiply Instructions

Multiplying two 32-bit numbers produces a 64-bit product. The ARM architecture provides *multiply instructions* that result in a 32-bit or 64-bit product. Multiply (MUL) multiplies two 32-bit numbers and produces a 32-bit result. MUL R1, R2, R3 multiplies the values in R2 and R3 and places the least significant bits of the product in R1; the most significant 32 bits of the product are discarded. This instruction is useful for multiplying small numbers whose result fits in 32 bits. UMULL (unsigned multiply long) and SMULL (signed multiply long) multiply two 32-bit numbers and produce a 64-bit product. For example, UMULL R1, R2, R3, R4 performs an unsigned multiply of R3 and R4. The least significant 32 bits of the product is placed in R1 and the most significant 32 bits are placed in R2.

Each of these instructions also has a multiply-accumulate variant, MLA, SMLAL, and UMLAL, that adds the product to a running 32- or 64-bit sum.

Condition Flags

ARM instructions optionally set *condition flags* based on whether the result is negative, zero, etc. Subsequent instructions then execute *conditionally*, depending on the state of those condition flags. The ARM condition flags, also called *status flags*, are negative (N), zero (Z), carry (C), and overflow (V), as listed in Figure 6.7. These flags are set by the ALU and are held in the top 4 bits of the 32-bit *Current Program Status Register* (*CPSR*), as shown in Figure 6.8.

Flag	Name	Description
N	Negative	Instruction result is negative, i.e., bit 31 of the result is 1
Z	Zero	Instruction result is zero
C	Carry	Instruction causes a carry out
V	oVerflow	Instruction causes an overflow

Figure 6. 7 Condition flags

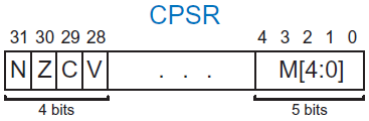


Figure 6. 8 Current Program Status Register (CPSR)

The most common way to set the status bits is with the compare (CMP) instruction, which subtracts the second source operand from the first and sets the condition flags based on the result. For example, if the numbers are equal, the result will be zero and the Z flag is set. If the first number is an unsigned value that is higher than or the same as the second, the subtraction will produce a carry out and the C flag is set.

Subsequent instructions can conditionally execute depending on the state of the flags. The instruction mnemonic is followed by a *condition mnemonic* that indicates when to execute. Figure 6.9 lists the 4-bit condition field (*cond*), the condition mnemonic, name, and the state of the condition flags that result in instruction execution (CondEx). For example, suppose a program performs CMP R4, R5, and then ADDEQ R1, R2, R3.

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

Figure 6. 9 Condition mnemonics

Other useful instructions for comparing two values are CMN (compare negative), TST (test), and TEQ (test if equal). Each instruction performs an operation, updates the condition flags, and discards the result. (Page 306)

Condition mnemonics differ for signed and unsigned comparison.

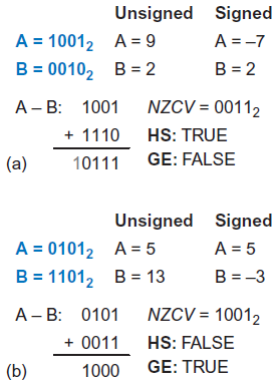


Figure 6. 10 Signed vs. unsigned comparison: HS vs. GE

Other data-processing instructions will set the condition flags when the instruction mnemonic is followed by "S." For example, SUBS R2, R3, R7 will subtract R7 from R3, put the result in R2, and set the condition flags.

Code Example 6.10 shows instructions that execute conditionally.

Code Example 6.10: CONDITIONAL EXECUTION (Page 308)

Branching

An advantage of a computer over a calculator is its ability to make decisions.

One way to make decisions is to use conditional execution to ignore certain instructions. This works well for simple if statements where a small number of instructions are ignored, but it is wasteful for if statements with many instructions in the body, and it is insufficient to handle loops. Thus, ARM and most other architectures use ***branch*** *instructions* to skip over sections of code or repeat code.

A program usually executes in sequence, with the program counter (PC) incrementing by 4 (or 2 for thumb instruction) after each instruction to point to the next instruction. Branch instructions change the program counter. ARM includes two types of branches: a simple *branch* (B) and *branch and link* (BL). Like other ARM instructions, branches can be unconditional or conditional. Branches are also called ***jumps*** in some architectures.

Code Example 6.11 shows unconditional branching using the branch instruction B. When the code reaches the B TARGET instruction, the branch is *taken*. That is, the next instruction executed is the SUB instruction just after the *label* called TARGET.

Code Example 6.11: UNCONDITIONAL BRANCHING (Page 309)

ARM Assembly Code

```
  ADD R1, R2, #17    ; R1 = R2 + 17
  B   TARGET         ; branch to TARGET
  ORR R1, R1, R3     ; not executed
  AND R3, R1, #0xFF  ; not executed

TARGET
  SUB R1, R1, #78    ; R1 = R1 – 78
```

Assembly code uses labels to indicate instruction locations in the program. When the assembly code is translated into machine code, these labels are translated into instruction addresses. ARM assembly labels cannot be reserved words, such as instruction mnemonics. Most programmers indent their instructions but not the labels, to help make labels stand out. The ARM compiler makes this a requirement: labels must not be indented, and instructions must be preceded by white space. Some compilers, including GCC, require a colon after the label.

Code Example 6.12: CONDITIONAL BRANCHING (Page 309)

ARM Assembly Code

```
  MOV R0, #4       ; R0 = 4
  ADD R1, R0, R0   ; R1 = R0 + R0 = 8
  CMP R0, R1       ; set flags based on R0–R1 = –4. NZCV = 1000
  BEQ THERE        ; branch not taken (Z != 1)
  ORR R1, R1, #1   ; R1 = R1 OR 1 = 9
THERE
  ADD R1, R1, #78  ; R1 = R1 + 78 = 87
```

Conditional Statements

if, if/else, and switch/case statements are conditional statements commonly used in high-level languages. They each conditionally execute a *block* of code consisting of one or more statements.

if Statements

An if statement executes a block of code, the *if block*, only when a condition is met.

Code Example 6.13: IF STATEMENT (Page 310)

High-Level Code

```
if (apples == oranges)
  f = i + 1;

f = f – i;
```

ARM Assembly Code

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
  CMP R0, R1       ; apples == oranges ?
  BNE L1           ; if not equal, skip if block
  ADD R2, R3, #1   ; if block: f = i + 1
L1
  SUB R2, R2, R3   ; f = f – i
```

The assembly code for the if statement tests the opposite condition of the one in the high-level code.

Because any instruction can be conditionally executed, the ARM assembly code for Code Example 6.13 could also be written more compactly as shown below.

```
CMP   R0, R1        ; apples == oranges ?
ADDEQ R2, R3, #1    ; f = i + 1 on equality (i.e., Z = 1)
SUB   R2, R2, R3    ; f = f – i
```

This solution with conditional execution is shorter and also faster because it involves one fewer instruction.

In general, when a block of code has a single instruction, it is better to use conditional execution rather than branch around it. As the block becomes longer, the branch becomes valuable because it avoids wasting time fetching instructions that will not be executed.

if/else Statements

if/else statements execute one of two blocks of code depending on a condition. When the condition in the if statement is met, the *if block* is executed. Otherwise, the *else block* is executed.

Code Example 6.14: IF/ELSE STATEMENT (Page 311)

High-Level Code

```
if (apples == oranges)
  f = i + 1;

else
  f = f – i;
```

ARM Assembly Code

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
  CMP R0, R1       ; apples == oranges?
  BNE L1           ; if not equal, skip if block
  ADD R2, R3, #1   ; if block: f = i + 1
  B   L2           ; skip else block
L1
  SUB R2, R2, R3   ; else block: f = f – i
L2
```

Again, because any instruction can conditionally execute and because the instructions within the if block do not change the condition flags, the ARM assembly code for Code Example 6.14 could also be written much more succinctly as:

```
CMP   R0, R1        ; apples == oranges?
ADDEQ R2, R3, #1    ; f = i + 1 on equality (i.e., Z = 1)
SUBNE R2, R2, R3    ; f = f – i on not equal (i.e., Z = 0)
```

switch/case Statements

switch/case statements execute one of several blocks of code depending on the conditions. If no conditions are met, the *default block* is executed. A case statement is equivalent to a series of *nested* if/else statements.

Code Example 6.15: SWITCH/CASE STATEMENT (Page 311)

High-Level Code

```
switch (button) {
  case 1:  amt = 20;  break;

  case 2:  amt = 50;  break;

  case 3:  amt = 100; break;

  default: amt = 0;
}
// equivalent function using
// if/else statements
  if      (button == 1) amt = 20;
  else if (button == 2) amt = 50;
  else if (button == 3) amt = 100;
  else                  amt = 0;
```

ARM Assembly Code

```
; R0 = button, R1 = amt
  CMP   R0, #1          ; is button 1 ?
  MOVEQ R1, #20         ; amt = 20 if button is 1
  BEQ   DONE            ; break

  CMP   R0, #2          ; is button 2 ?
  MOVEQ R1, #50         ; amt = 50 if button is 2
  BEQ   DONE            ; break

  CMP   R0, #3          ; is button 3?
  MOVEQ R1, #100        ; amt = 100 if button is 3
  BEQ   DONE            ; break

  MOV   R1, #0          ; default amt = 0
DONE
```

Getting Loopy

Loops repeatedly execute a block of code depending on a condition. while loops and for loops are common loop constructs used by high-level languages.

while Loops

while loops repeatedly execute a block of code until a condition is *not* met.

Code Example 6.16: WHILE LOOP (Page 312)

High-Level Code

```
int pow = 1;
int x = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

```
; R0 = pow, R1 = x
  MOV R0, #1      ; pow = 1
  MOV R1, #0      ; x = 0

WHILE
  CMP R0, #128    ; pow != 128 ?
  BEQ DONE        ; if pow == 128, exit loop
  LSL R0, R0, #1  ; pow = pow * 2
  ADD R1, R1, #1  ; x = x + 1
  B   WHILE       ; repeat loop
DONE
```

for Loops

It is very common to initialize a variable before a while loop, check that variable in the loop condition, and change that variable each time through the while loop. for loops are a convenient shorthand that combines the initialization, condition check, and variable change in one place. The format of the for loop is:

```
for (initialization; condition; loop operation)
  statement
```

Code Example 6.17: FOR LOOP (Page 313)

High-Level Code

```
int i;
int sum = 0;


for (i = 0; i < 10; i = i + 1) {
   sum = sum + i;
}
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV R1, #0        ; sum = 0
  MOV R0, #0        ; i = 0          loop initialization

FOR
  CMP R0, #10       ; i < 10 ?       check condition
  BGE DONE          ; if (i >= 10) exit loop
  ADD R1, R1, R0    ; sum = sum + i  loop body
  ADD R0, R0, #1    ; i = i + 1      loop operation
  B   FOR           ; repeat loop
DONE
```

Memory

For ease of storage and access, similar data can be grouped together into an *array*. An array stores its contents at sequential data addresses in memory. Each array element is identified by a number called its *index*. The number of elements in the array is called the *length* of the array. ARM can *scale* (multiply) the index, add it to the base address, and load from memory in a single instruction. Instead of the LSL and LDR instruction sequence in Code Example 6.18, we can use a single instruction:

```
LDR R3, [R0, R1, LSL #2]
```

R1 is scaled (shifted left by two) then added to the base address (R0). Thus, the memory address is R0+(R1×4).

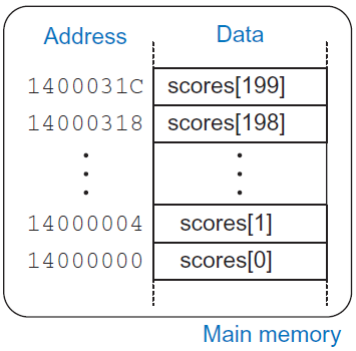


Figure 6. 11 Memory holding scores[200] starting at base address 0x14000000

Code Example 6.18: ACCESSING ARRAYS USING A FOR LOOP (Page 314)

High-Level Code

```
int i;
int scores[200];
...


for (i = 0; i < 200; i = i + 1)


   scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
; initialization code     ...
  MOV R0, #0x14000000     ; R0 = base address
  MOV R1, #0              ; i = 0

LOOP
  CMP R1, #200            ; i < 200?
  BGE L3                  ; if i ≥ 200, exit loop
  LSL R2, R1, #2          ; R2 = i * 4
  LDR R3, [R0, R2]        ; R3 = scores[i]
  ADD R3, R3, #10         ; R3 = scores[i] + 10
  STR R3, [R0, R2]        ; scores[i] = scores[i] + 10
  ADD R1, R1, #1          ; i = i + 1
  B   LOOP                ; repeat loop
L3
```

In addition to scaling the index register, ARM provides offset, pre-indexed, and post-indexed addressing to enable dense and efficient code for array accesses and function calls. Figure 6.12 gives examples of each indexing mode. In each case, the base register is R1 and the offset is R2. The offset can be subtracted by writing –R2. The offset may also be an immediate in the range of $0-4095$ that can be added (e.g., #20) or subtracted (e.g., #−20).

Mode	ARM Assembly	Address	Base Register
Offset	`LDR R0, [R1, R2]`	R1 + R2	Unchanged
Pre-index	`LDR R0, [R1, R2]!`	R1 + R2	R1 = R1 + R2
Post-index	`LDR R0, [R1], R2`	R1	R1 = R1 + R2

Figure 6. 12 ARM indexing modes

Offset addressing calculates the address as the base register ± the offset; the base register is unchanged. *Pre-indexed addressing* calculates the address as the base register ± the offset and updates the base register to this new address. *Post-indexed addressing* calculates the address as the base register only and then, after accessing memory, the base register is updated to the base register ± the offset.

Complement materials:

Many load and store instructions support different addressing modes.

Offset addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The base register is unchanged. The assembly language syntax for this mode is:

$$[Rn, offset]$$

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register. The assembly language syntax for this mode is:

$$[Rn, offset]!$$

Post-indexed addressing

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register. The assembly language syntax for this mode is:

$$[Rn], offset$$

In each case, Rn is the base register and $offset$ can be:

- An immediate constant.
- An index register, Rm.
- A shifted index register, such as Rm, LSL $\#shift$.

Code Example 6.19: FOR LOOP USING POST-INDEXING (Page 315)

High-Level Code

```
int i;
int scores[200];
...

for (i = 0; i < 200; i = i + 1)
        scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address
; initialization code    ...
  MOV R0, #0x14000000    ; R0 = base address
  ADD R1, R0, #800       ; R1 = base address + (200*4)

LOOP
  CMP R0, R1             ; reached end of array?
  BGE L3                 ; if yes, exit loop
  LDR R2, [R0]           ; R2 = scores[i]
  ADD R2, R2, #10        ; R2 = scores[i] + 10
  STR R2, [R0], #4       ; scores[i] = scores[i] + 10
                         ; then R0 = R0 + 4
  B   LOOP               ; repeat loop
  L3
```

Bytes and Characters

Numbers in the range $[-128,127]$ can be stored in a single byte rather than an entire word. Because there are much fewer than 256 characters on an English language keyboard, English characters are often represented by bytes. The C language uses the type char to represent a byte or character.

The *American Standard Code for Information Interchange* (ASCII) assigns each text character a unique byte value. Figure 6.13 shows these character encodings for printable characters. The ASCII values are given in hexadecimal. Lowercase and uppercase letters differ by 0x20 (32).

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Figure 6. 13 ASCII encodings

Other programming languages, such as Java, use different character encodings, most notably Unicode. Unicode uses 16 bits to represent each character.

ARM provides load byte (LDRB), load signed byte (LDRSB), and store byte (STRB) to access individual bytes in memory. LDRB zero-extends the byte, whereas LDRSB sign-extends the byte to fill the entire 32-bit register. STRB stores the least significant byte of the 32-bit register into the specified byte address in memory. All three are illustrated in Figure 6.14, with the base address R4 being 0. LDRB loads the byte at memory address 2 into the least significant byte of R1 and fills the remaining register bits with 0. LDRSB loads this byte into R2 and sign-extends the byte into the upper 24 bits of the register. STRB stores the least significant byte of R3 (0x9B) into memory byte 3; it replaces 0xF7 with 0x9B. The more significant bytes of R3 are ignored.

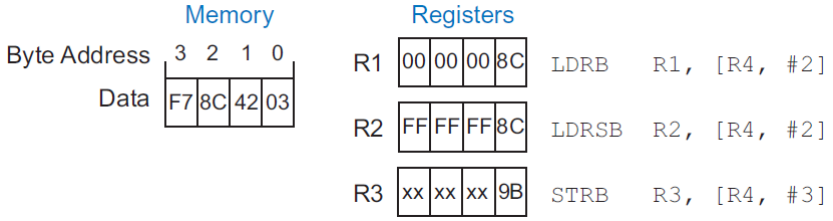


Figure 6. 14 Instructions for loading and storing bytes

LDRH, LDRSH, and STRH are similar, but access 16-bit halfwords.

A series of characters is called a *string*. Strings have a variable length, so programming languages must provide a way to determine the length or end of the string. In C, the null character (0x00) signifies the end of a string.

Example 6.2: USING LDRB AND STRB TO ACCESS A CHARACTER ARRAY (Page 317)

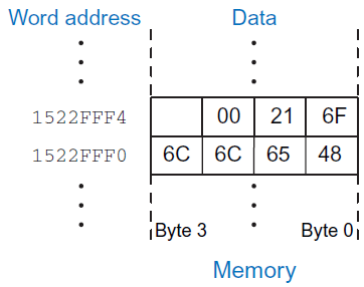


Figure 6. 15 The string “Hello!” stored in memory

Function Calls

High-level languages support *functions* (also called *procedures* or *subroutines*) to reuse common code and to make a program more modular and readable. Functions have inputs, called *arguments*, and an output, called the *return value*. Functions should calculate the return value and cause no other unintended side effects.

When one function calls another, the calling function, the *caller*, and the called function, the *callee*, must agree on where to put the arguments and the return value. In ARM, the caller conventionally places up to four arguments in registers R0–R3 before making the function call, and the callee places the return value in register R0 before finishing.

The callee must not interfere with the behavior of the caller. The caller stores the return address (next instruction) in the link register LR at the same time it jumps to the callee using the branch and link instruction (BL). The callee must not overwrite any architectural state or memory that the caller is depending on. Specifically, the callee must leave the *saved registers* (R4–R11, and LR) and the *stack*, a portion of memory used for temporary variables, unmodified.

Function Calls and Returns

ARM uses the branch and link instruction (BL) to call a function and moves the link register to the PC (MOV PC, LR) to return from a function. Code Example 6.20 shows the main function calling the simple function.

Code Example 6.20: simple FUNCTION CALL (Page 318)

High-Level Code

```
int main() {

  simple();
  ...
}

// void means the function returns no value
void simple() {
  return;
}
```

ARM Assembly Code

```
0x00008000 MAIN     ...
...                 ...
0x00008020          BL  SIMPLE    ; call the simple function
...


0x0000902C  SIMPLE  MOV PC, LR    ; return
```

BL (branch and link) and MOV PC, LR are the two essential instructions needed for a function call and return. BL performs two tasks: it stores the *return address* of the next instruction (the instruction after BL) in the link register (LR), and it branches to the target instruction.

These days, ARM compilers do a function return using BX LR. The BX branch and exchange instruction is like a branch, but it also can transition between the standard ARM instruction set and the Thumb instruction set.

Input Arguments and Return Values

Code Example 6.21: FUNCTION CALL WITH ARGUMENTS AND RETURN VALUES (Page 319)

High-Level Code

```
int main() {
  int y;
  . . .
  y = diffofsums(2, 3, 4, 5);
  . . .
}



int diffofsums(int f, int g, int h, int i) {
  int result;

  result = (f + g) - (h + i);
  return result;
}
```

ARM Assembly Code

```
; R4 = y
MAIN
  . . .
  MOV R0, #2       ; argument 0 = 2
  MOV R1, #3       ; argument 1 = 3
  MOV R2, #4       ; argument 2 = 4
  MOV R3, #5       ; argument 3 = 5
  BL  DIFFOFSUMS   ; call function
  MOV R4, R0       ; y = returned value
  . . .

; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1   ; R8 = f + g
  ADD R9, R2, R3   ; R9 = h + i
  SUB R4, R8, R9   ; result = (f + g) - (h + i)
  MOV R0, R4       ; put return value in R0
  MOV PC, LR       ; return to caller
```

According to ARM convention, the calling function, main, places the function arguments from left to right into the input registers, R0–R3. The called function, diffofsums, stores the return value in the return register, R0. When a function with more than four arguments is called, the additional input arguments are placed on the stack.

The Stack

The stack is memory that is used to save information within a function. The stack expands (uses more memory) as the processor needs more scratch space and contracts (uses less memory) when the processor no longer needs the variables stored there.

The stack is a last-in-first-out (LIFO) queue. Like a stack of dishes, the last item *pushed* onto the stack (the top dish) is the first one that can be *popped* off. Each function may allocate stack space to store local variables but must deallocate it before returning. The *top of the stack* is the most recently allocated space. Whereas a stack of dishes grows up in space, the ARM stack grows down in memory. The stack expands to lower memory addresses when a program needs more scratch space.

Figure 6.16 shows a picture of the stack. The stack pointer, SP (R13), is an ordinary ARM register that, by convention, *points to the top of the stack*. A pointer is a fancy name for a memory address. SP points to (gives the address of) data. For example, in Figure 6.16(a), the stack pointer, SP, holds the address value 0XBEFFFAE8 and points to the data value 0xAB000001.

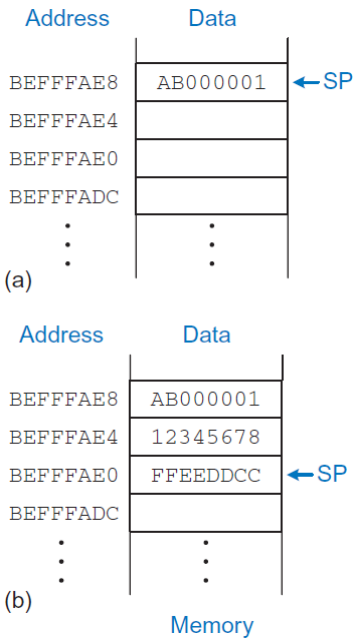


Figure 6. 16 The stack (a) before expansion and (b) after two-word expansion

The stack pointer (SP) starts at a high memory address and decrements to expand as needed. The stack is typically stored upside down in memory such that the top of the stack is actually the lowest address and the stack grows downward toward lower memory addresses. This is called a *descending stack*. ARM also allows for *ascending stacks* that grow up toward higher memory addresses. The stack pointer typically points to the topmost element on the stack; this is called a *full stack*. ARM also allows for *empty stacks* in which SP points one word beyond the top of the stack.

One of the important uses of the stack is to save and restore registers that are used by a function. Recall that a function should calculate a return value but have no other unintended side effects. In particular, it should not modify any registers besides R0, the one containing the return value. The diffofsums function in Code Example 6.21 violates this rule because it modifies R4, R8, and R9.

To solve this problem, a function saves registers on the stack before it modifies them, then restores them from the stack before it returns. Specifically, it performs the following steps:

1. Makes space on the stack to store the values of one or more registers
2. Stores the values of the registers on the stack
3. Executes the function using the registers
4. Restores the original values of the registers from the stack
5. Deallocates space on the stack

Code Example 6.22: FUNCTION SAVING REGISTERS ON THE STACK (Page 321)

ARM Assembly Code

```
;R4 = result
DIFFOFSUMS
  SUB   SP, SP, #12       ; make space on stack for 3 registers
  STR   R9, [SP, #8]      ; save R9 on stack
  STR   R8, [SP, #4]      ; save R8 on stack
  STR   R4, [SP]          ; save R4 on stack

  ADD  R8, R0, R1         ; R8 = f + g
  ADD  R9, R2, R3         ; R9 = h + i
  SUB  R4, R8, R9         ; result = (f + g) - (h + i)
  MOV  R0, R4             ; put return value in R0

  LDR   R4, [SP]          ; restore R4 from stack
  LDR   R8, [SP, #4]      ; restore R8 from stack
  LDR   R9, [SP, #8]      ; restore R9 from stack
  ADD   SP, SP, #12       ; deallocate stack space

  MOV   PC, LR            ; return to caller
```

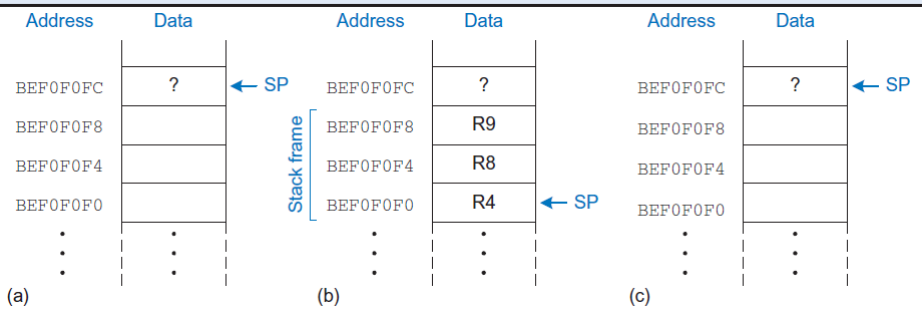


Figure 6. 17 The stack: (a) before, (b) during, and (c) after the diffofsums function call

The stack space that a function allocates for itself is called its *stack frame*. diffofsums's stack frame is three words deep. The principle of modularity tells us that each function should access only its own stack frame, not the frames belonging to other functions.

Loading and Storing Multiple Registers

Saving and restoring registers on the stack is such a common operation that ARM provides Load Multiple and Store Multiple instructions (LDM and STM) that are optimized to this purpose.

Code Example 6.23: SAVING AND RESTORING MULTIPLE REGISTERS (Page 322)

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
  STMFD  SP!, {R4, R8, R9}        ; push R4/8/9 on full descending stack

  ADD    R8, R0, R1               ; R8 = f + g
  ADD    R9, R2, R3               ; R9 = h + i
  SUB    R4, R8, R9               ; result = (f + g) - (h + i)
  MOV    R0, R4                   ; put return value in R0

  LDMFD  SP!, {R4, R8, R9}        ; pop R4/8/9 off full descending stack

  MOV    PC, LR                   ; return to caller
```

LDM and STM come in four flavors for full and empty descending and ascending stacks (FD, ED, FA, EA). The SP! in the instructions indicates to store the data relative to the stack pointer and to update the stack pointer after the store or load. PUSH and POP are synonyms for STMFD SP!, {regs} and LDMFD SP!, {regs}, respectively.

Preserved Registers

To avoid the waste of saving and restoring registers, ARM divides registers into *preserved* and *nonpreserved* categories. The preserved registers include R4–R11. The nonpreserved registers are R0–R3 and R12. SP and LR (R13 and R14) must also be preserved. A function must save and

restore any of the preserved registers that it wishes to use, but it can change the nonpreserved registers freely.

Code Example 6.24: REDUCING THE NUMBER OF PRESERVED REGISTERS (Page 323)

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
   PUSH {R4}                  ; save R4 on stack
   ADD  R1, R0, R1            ; R1 = f + g
   ADD  R3, R2, R3            ; R3 = h + i
   SUB  R4, R1, R3            ; result = (f + g) - (h + i)
   MOV  R0, R4                ; put return value in R0
   POP  {R4}                  ; pop R4 off stack
   MOV  PC, LR                ; return to caller
```

PUSH (and POP) save (and restore) registers on the stack in order of register number from low to high, with the lowest numbered register placed at the lowest memory address, regardless of the order listed in the assembly instruction.

Remember that when one function calls another, the former is the caller and the latter is the callee. The callee must save and restore any preserved registers that it wishes to use. The callee may change any of the nonpreserved registers. Hence, if the caller is holding active data in a nonpreserved register, the caller needs to save that nonpreserved register before making the function call and then needs to restore it afterward. For these reasons, preserved registers are also called *callee-save*, and nonpreserved registers are called *caller-save*.

Figure 6.18 summarizes which registers are preserved.

Preserved	Nonpreserved
Saved registers: R4–R11	Temporary register: R12
Stack pointer: SP (R13)	Argument registers: R0–R3
Return address: LR (R14)	Current Program Status Register
Stack above the stack pointer	Stack below the stack pointer

Figure 6. 18 Preserved and nonpreserved registers

Code Example 6.25: OPTIMIZED diffofsums FUNCTION CALL (Page 324)

ARM Assembly Code

```
DIFFOFSUMS
   ADD  R1, R0, R1     ; R1 = f + g
   ADD  R3, R2, R3     ; R3 = h + i
   SUB  R0, R1, R3     ; return (f + g) - (h + i)
   MOV  PC, LR         ; return to caller
```

Nonleaf Function Calls

A function that does not call others is called a *leaf function*; diffofsums is an example. A function that does call others is called a *nonleaf function*. As mentioned, nonleaf functions are somewhat more complicated because they may need to save nonpreserved registers on the stack before they call another function and then restore those registers afterward. Specifically:

Caller save rule: Before a function call, the caller must save any nonpreserved registers (R0–R3 and R12) that it needs after the call. After the call, it must restore these registers before using them.

Callee save rule: Before a callee disturbs any of the preserved registers (R4–R11 and LR), it must save the registers. Before it returns, it must restore these registers.

A nonleaf function overwrites LR when it calls another function using BL. Thus, a nonleaf function must always save LR on its stack and restore it before returning.

Code Example 6.26: NONLEAF FUNCTION CALL (Page 325)

High-Level Code

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}

int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; R0 = a, R1 = b, R4 = i, R5 = x
F1
   PUSH  {R4, R5, LR}    ; save preserved registers used by f1
   ADD   R5, R0, R1      ; x = (a + b)
   SUB   R12, R0, R1     ; temp = (a - b)
   MUL   R5, R5, R12     ; x = x * temp = (a + b) * (a - b)
   MOV   R4, #0          ; i = 0
FOR
   CMP   R4, R0          ; i < a?
   BGE   RETURN          ; no: exit loop
   PUSH  {R0, R1}        ; save nonpreserved registers
   ADD   R0, R1, R4      ; argument is b + i
   BL    F2              ; call f2(b+i)
   ADD   R5, R5, R0      ; x = x + f2(b+i)
   POP   {R0, R1}        ; restore nonpreserved registers
   ADD   R4, R4, #1      ; i++
   B     FOR             ; continue for loop
RETURN
   MOV   R0, R5          ; return value is x
   POP   {R4, R5, LR}    ; restore preserved registers
   MOV   PC, LR          ; return from f1

; R0 = p, R4 = r
F2
   PUSH  {R4}            ; save preserved registers used by f2
   ADD   R4, R0, 5       ; r = p + 5
   ADD   R0, R4, R0      ; return value is r + p
   POP   {R4}            ; restore preserved registers
   MOV   PC, LR          ; return from f2
```

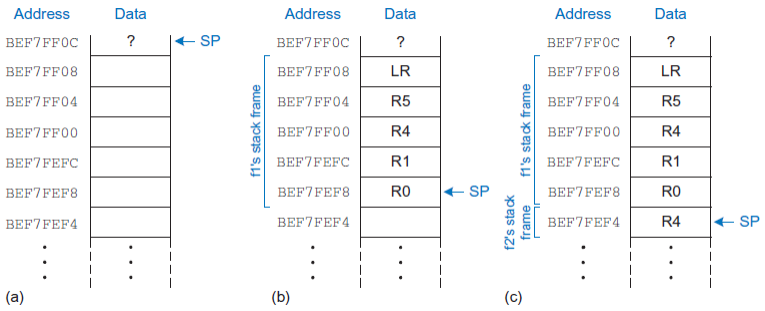


Figure 6. 19 The stack: (a) before function calls, (b) during f1, and (c) during f2

Recursive Function Calls

A *recursive function* is a nonleaf function that calls itself. Recursive functions behave as both caller and callee and must save both preserved and nonpreserved registers.

Code Example 6.27: factorial RECURSIVE FUNCTION CALL (Page 327)

High-Level Code

```
int factorial(int n) {
  if (n <= 1)
    return 1;


  else
    return (n * factorial(n - 1));
}
```

ARM Assembly Code

```
0x8500  FACTORIAL  PUSH  {R0, LR}   ; push n and LR on stack
0x8504             CMP   R0, #1     ; R0 <= 1?
0x8508             BGT   ELSE       ; no: branch to else
0x850C             MOV   R0, #1     ; otherwise, return 1
0x8510             ADD   SP, SP, #8 ; restore SP
0x8514             MOV   PC, LR     ; return
0x8518  ELSE       SUB   R0, R0, #1 ; n = n - 1
0x851C             BL    FACTORIAL  ; recursive call
0x8520             POP   {R1, LR}   ; pop n (into R1) and LR
0x8524             MUL   R0, R1, R0 ; R0 = n * factorial(n - 1)
0x8528             MOV   PC, LR     ; return
```

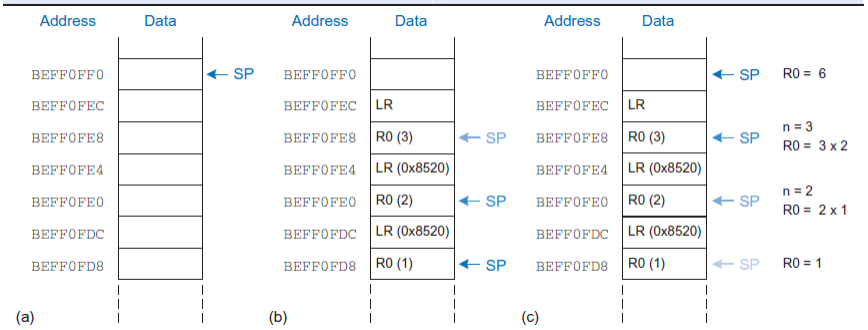


Figure 6. 20 Stack: (a) before, (b) during, and (c) after factorial function call with $n = 3$

Additional Arguments and Local Variables

Functions may have more than four input arguments and may have too many local variables to keep in preserved registers. The stack is used to store this information. By ARM convention, if a

function has more than four arguments, the first four are passed in the argument registers as usual. Additional arguments are passed on the stack, just above SP. The caller must expand its stack to make room for the additional arguments. Figure 6.21(a) shows the caller's stack for calling a function with more than four arguments.

A function can also declare local variables or arrays. Local variables are declared within a function and can be accessed only within that function. Local variables are stored in R4–R11; if there are too many local variables, they can also be stored in the function's stack frame. In particular, local arrays are stored on the stack.

Figure 6.21(b) shows the organization of a callee's stack frame. Accessing additional input arguments is the one exception in which a function can access stack data not in its own stack frame.

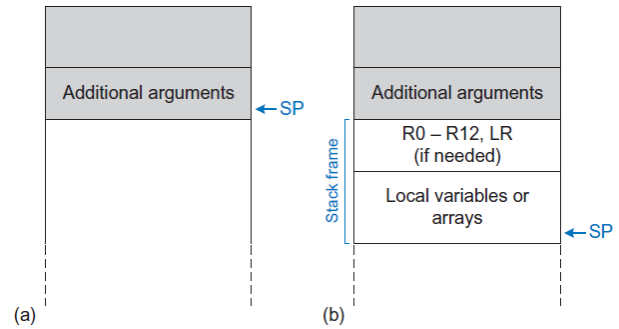


Figure 6. 21 Stack usage: (a) before and (b) after call

6.4 MACHINE LANGUAGE

Assembly language is convenient for humans to read. However, digital circuits understand only 1's and 0's. Therefore, a program written in assembly language is translated from mnemonics to a representation using only 1's and 0's called *machine language*.

ARM uses 32-bit instructions. Again, regularity supports simplicity, and the most regular choice is to encode all instructions as words that can be stored in memory. **Even though** some instructions may **not** require all 32 bits of encoding, variable-length instructions would add complexity. Simplicity would **also** encourage a single instruction format, but that is **too restrictive**. However, this issue allows us to introduce the last design principle:

Design Principle 4: Good design demands good compromises.

Data-processing Instructions

The data-processing instruction format is the most common.

The 32-bit instruction has six fields: $cond$, op, $funct$, Rn, Rd, and $Src2$.

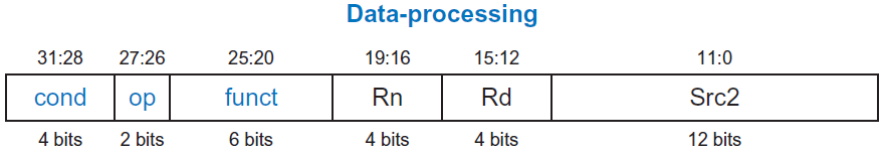


Figure 6. 22 Data-processing instruction format

Memory Instructions

Memory instructions use a format similar to that of data-processing instructions, with the same six overall fields: $cond$, op, $funct$, Rn, Rd, and $Src2$.

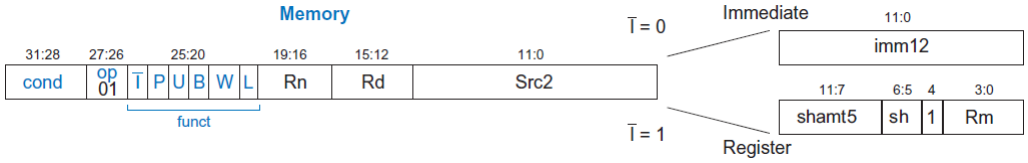


Figure 6. 23 Memory instruction format for LDR, STR, LDRB, and STRB

Branch Instructions

Branch instructions use a single 24-bit signed immediate operand, $imm24$.

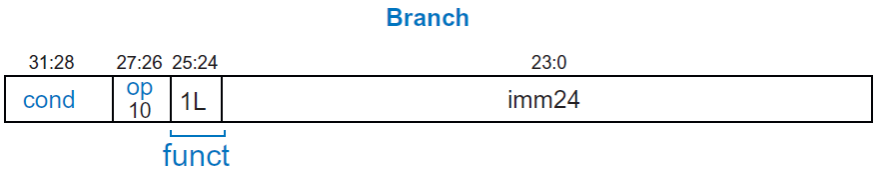


Figure 6. 24 Branch instruction format

The 24-bit immediate field gives the number of instructions between the BTA and PC+8 (two instructions past the branch).

Code Example 6.28: CALCULATING THE BRANCH TARGET ADDRESS

The processor calculates the BTA from the instruction by sign-extending the 24-bit immediate, shifting it left by 2 (to convert words to bytes), and adding it to PC+8.

(Page 335)

Addressing Modes

This section summarizes the modes used for addressing instruction operands. ARM uses four main modes: register, immediate, base, and PC-relative addressing.

Data-processing instructions use register or immediate addressing. Memory instructions use base addressing. Branches use PC-relative addressing in which the branch target address is computed by adding an offset to PC+8.

Complement materials

During execution, PC does not contain the address of the currently executing instruction. The address of the currently executing instruction is typically PC−8 for ARM, or PC−4 for Thumb.

Operand Addressing Mode	Example	Description
Register		
Register-only	ADD R3, R2, R1	R3 ← R2 + R1
Immediate-shifted register	SUB R4, R5, R9, LSR #2	R4 ← R5 − (R9 >> 2)
Register-shifted register	ORR R0, R10, R2, ROR R7	R0 ← R10 | (R2 ROR R7)
Immediate	SUB R3, R2, #25	R3 ← R2 − 25
Base		
Immediate offset	STR R6, [R11, #77]	mem[R11+77] ← R6
Register offset	LDR R12, [R1, −R5]	R12 ← mem[R1 − R5]
Immediate-shifted register offset	LDR R8, [R9, R2, LSL #2]	R8 ← mem[R9 + (R2 << 2)]
PC-Relative	B LABEL1	Branch to LABEL1

Figure 6. 25 ARM operand addressing modes

Interpreting Machine Language Code

Example 6.5: TRANSLATING MACHINE LANGUAGE TO ASSEMBLY LANGUAGE (Page 337)

The Power of the Stored Program

A program written in machine language is a series of 32-bit numbers representing the instructions. Like other binary numbers, these instructions can be stored in memory. This is called the *stored program* concept, and it is a key reason why computers are so powerful.

Instructions in a stored program are retrieved, or *fetched*, from memory and executed by the processor. Even large, complex programs are simply a series of memory reads and instruction executions.

Figure 6.26 shows how machine instructions are stored in memory. In ARM programs, the instructions are normally stored starting at low addresses, in this case 0x00008000.

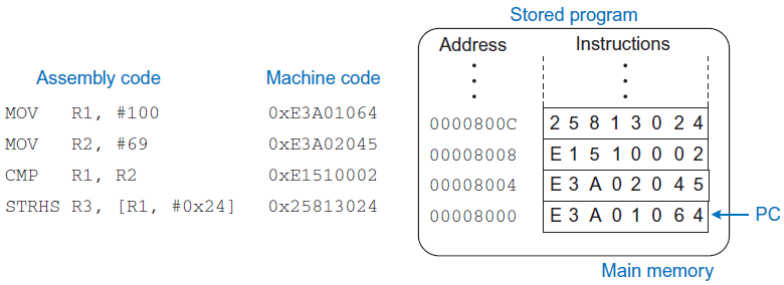


Figure 6. 26 Stored program

To run or execute the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in a 32-bit register called the program counter (PC), which is register R15. For historical reasons, a read to the PC returns the address of the current instruction plus 8.

To execute the code in Figure 6.26, the PC is initialized to address 0x00008000. The processor fetches the instruction at that memory address and executes the instruction, 0xE3A01064 (MOV R1, #100). The processor then increments the PC by 4 to 0x00008004, fetches and executes that instruction, and repeats.

The ***architectural state*** of a microprocessor holds the state of a program. For ARM, the architectural state includes the register file and status registers. If the operating system (OS) saves the architectural state at some point in the program, it can interrupt the program, do something else, and then restore the state such that the program continues properly, unaware that it was ever interrupted.

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

Figure 6.27 shows the steps required to translate a program from a high-level language into machine language and to start executing that program. First, a *compiler* translates the high-level code into assembly code. The *assembler* translates the assembly code into machine code and puts it in an object file. The *linker* combines the machine code with code from libraries and other files and determines the proper branch addresses and variable locations to produce an entire executable program. In practice, most compilers perform all three steps of compiling, assembling, and linking. Finally, the *loader* loads the program into memory and starts execution.

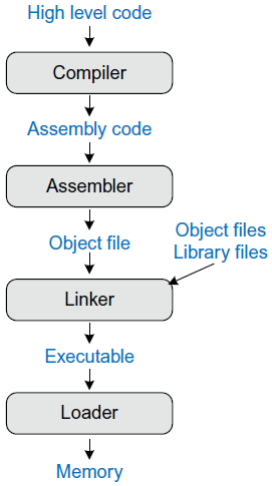


Figure 6. 27 Steps for translating and starting a program

The Memory Map

With 32-bit addresses, the ARM address space spans 2^{32} bytes (4 GB). Word addresses are multiples of 4 and range from 0 to 0xFFFFFFFC. Figure 6.28 shows an example *memory map*. The ARM architecture divides the address space into five parts or segments: the text segment, global data segment, dynamic data segment, and segments for exception handlers, the operating system (OS) and input/output (I/O).

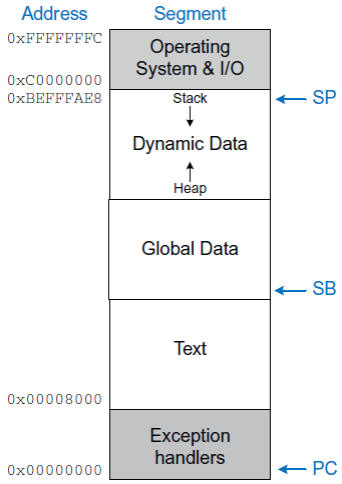


Figure 6. 28 Example ARM memory map

The Text Segment

The *text* segment stores the machine language program. ARM also calls this the *read-only* (RO) *segment*. In addition to code, it may include literals (constants) and read-only data.

The Global Data Segment

The *global data segment* stores global variables that, in contrast to local variables, can be accessed by all functions in a program. Global variables are allocated in memory before the program begins executing. ARM also calls this the *read/write* (RW) *segment*. Global variables are typically accessed using a *static base* register that points to the start of the global segment. ARM conventionally uses R9 as the static base pointer (SB).

The Dynamic Data Segment

The *dynamic data segment* holds the stack and the heap. The data in this segment is not known at start-up but is dynamically allocated and deallocated throughout the execution of the program.

Upon start-up, the operating system sets up the stack pointer (SP) to point to the top of the stack. The stack typically grows downward. The stack includes temporary storage and local variables, such as arrays, that do not fit in the registers.

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

The Exception Handler, OS, and I/O Segments

The lowest part of the ARM memory map is reserved for the exception vector table and exception handlers, starting at address 0x0. The highest part of the memory map is reserved for the operating system and memory-mapped I/O.

Compilation

A compiler translates high-level code into assembly language.

Code Example 6.29: COMPILING A HIGH-LEVEL PROGRAM (Page 341)

Assembling

An assembler turns the assembly language code into an object file containing machine language code.

Linking

The job of the linker is to combine all of the object files and the start startup code into one machine language file called the executable and assign addresses for global variables. The linker relocates the data and instructions in the object files so that they are not all on top of each other.

Loading

The operating system loads a program by reading the text segment of the executable file from

a storage device (usually the hard disk) into the text segment of memory.

6.6 ODDS AND ENDS

Loading Literals

Many programs need to load 32-bit literals, such as **constants** or **addresses**. MOV only accepts a 12-bit source, so the LDR instruction is used to load these numbers from a **literal pool** in the text segment. (Page 345)

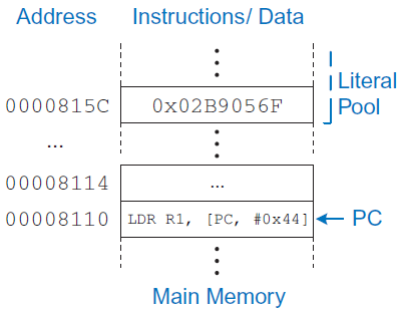


Figure 6. 29 Example literal pool

NOP

NOP is a mnemonic for "no operation" and is pronounced "no op." It is a *pseudoinstruction* that does nothing. The assembler translates it to MOV R0, R0 (0xE1A00000). NOPs are useful to, among other things, achieve some delay or align instructions.

Exceptions

An *exception* is like an unscheduled function call that branches to a new address. Exceptions may be caused by hardware or software. Such a hardware exception triggered by an input/output (I/O) device such as a keyboard is often called an *interrupt*. Software exceptions are sometimes called *traps*. A particularly important form of a trap is a *system call*, whereby the program invokes a function in the OS running at a higher privilege level.

Exceptions use a *vector* table to determine where to jump to the *exception handler* and use *banked registers* to maintain extra copies of key registers so that they will not corrupt the registers in the active program.

Execution Modes and Privilege Levels

An ARM processor can operate in one of several execution modes with different privilege levels. The different modes allow an exception to take place in an exception handler without corrupting state. The mode is specified in the bottom bits of the Current Program Status Register (CPSR), as was shown in Figure 6.8. Figure 6.30 lists execution modes and their encodings. User mode operates at privilege level PL0, which is unable to access protected portions of memory such as the operating system code. The other modes operate at privilege level PL1, which can access all system resources.

Mode	$CPSR_{4:0}$
User	10000
Supervisor	10011
Abort	10111
Undefined	11011
Interrupt (IRQ)	10010
Fast Interrupt (FIQ)	10001

Figure 6. 30 ARM execution modes

Exception Vector Table

When an exception occurs, the processor branches to an offset in the *exception vector table*, depending on the cause of the exception. Figure 6.31 describes the vector table, which is normally located starting at address 0x00000000 in memory.

Exception	Address	Mode
Reset	0x00	Supervisor
Undefined Instruction	0x04	Undefined
Supervisor Call	0x08	Supervisor
Prefetch Abort (instruction fetch error)	0x0C	Abort
Data Abort (data load or store error)	0x10	Abort
Reserved	0x14	N/A
Interrupt	0x18	IRQ
Fast Interrupt	0x1C	FIQ

Figure 6. 31 Exception vector table

Banked Registers

Before an exception changes the PC, it must save the return address in the LR so that the exception handler knows where to return. However, it must take care not to disturb the value already in the LR, which the program will need later. Therefore, the processor maintains a bank of different registers to use as LR during each of the execution modes. A bank of *saved program status registers* (SPSRs) is used to hold a copy of the CPSR during exceptions.

Exception Handling

Upon detecting an exception, the processor:

1. Stores the CPSR into the banked SPSR
2. Sets the execution mode and privilege level based on the type of exception
3. Sets *interrupt mask* bits in the CPSR so that the exception handler will not be interrupted
4. Stores the return address into the banked LR
5. Branches to the exception vector table based on the type of exception

The processor then executes the instruction in the exception vector table, typically a branch to the exception handler. The handler usually pushes other registers onto its stack, takes care of the exception, and pops the registers back off the stack. The exception handler returns using the MOVS PC, LR instruction, a special flavor of MOV that performs the following cleanup:

1. Copies the banked SPSR to the CPSR to restore the status register
2. Copies the banked LR (possibly adjusted for certain exceptions) to the PC to return to the program where the exception occurred
3. Restores the execution mode and privilege level

Exception-Related Instructions

Programs operate at a low privilege level, whereas the operating system has a higher privilege level. To transition between levels in a controlled way, the program places arguments in registers and issues a *supervisor call* (SVC) instruction, which generates an exception and raises the privilege level.

The OS and other code operating at PL1 can access the banked registers for the various execution modes using the MRS (move to register from special register) and MSR (move to special register from register) instructions.

Start-up

On start-up, the processor jumps to the reset vector and begins executing *boot loader* code in supervisor mode.

Supplement Materials

Exceptions vs. Interrupts:

- Cause
  - Exceptions: **internal** to the running thread
  - Interrupts: **external** to the running thread
- When to Handle
  - Exceptions: when detected (and known to be non-speculative)

- Interrupts: when convenient, except for very high priority ones like Power failure, Machine check (error)

6.7 EVOLUTION OF ARM ARCHITECTURE

ARM was an acronym for *Acorn RISC Machine*.

Thumb Instruction Set

Thumb instructions are 16 bits long to achieve higher code density; they are identical to regular ARM instructions but generally have limitations, including that they:

- Access only the bottom eight registers
- Reuse a register as both a source and destination
- Support shorter immediates
- Lack conditional execution
- Always write the status flags

ARM processors have an instruction set state register, ISETSTATE, that includes a T bit to indicate whether the processor is in normal mode (T= 0) or Thumb mode (T= 1).

DSP Instructions

Digital signal processors (DSPs) are designed to efficiently handle signal processing algorithms such as the Fast Fourier Transform (FFT) and Finite/Infinite Impulse Response filters (FIR/IIR). Common applications include audio and video encoding and decoding, motor control, and speech recognition. ARM provides a number of DSP instructions for these purposes. DSP instructions include multiply, add, and multiply-accumulate (MAC).

DSP instructions often operate on short (16-bit) data representing samples read from a sensor by an analog-to-digital converter. However, the intermediate results are held to greater precision (e.g., 32 or 64 bits) or saturated to prevent overflow. In *saturated arithmetic*, results larger than the most positive number are treated as the most positive, and results smaller than the most negative are treated as the most negative. For example, in 32-bit arithmetic, results greater than $2^{31}-1$ saturate at $2^{31}-1$, and results less than -2^{31} saturate at -2^{31}. Common DSP data types are given in Figure 6.32. Two's complement numbers are indicated as having one sign bit. The 16-, 32-, and 64-bit types are also known as *half*, *single*, and *double* precision, not to be confused with single and double-precision floating-point numbers.

Type	Sign Bit	Integer Bits	Fractional Bits
short	1	15	0
unsigned short	0	16	0
long	1	31	0
unsigned long	0	32	0
long long	1	63	0
unsigned long long	0	64	0
Q15	1	0	15
Q31	1	0	31

Figure 6. 32 DSP data types

Fractional types (Q15 and Q31) represent a signed fractional number; for example, Q31 spans the range $[-1,1-2^{-31}]$ with a step of 2^{-31} between consecutive numbers. These types are not defined in the C standard but are supported by some libraries. Q31 can be converted to Q15 by truncation or rounding. In truncation, the Q15 result is just the upper half. In rounding, 0x00008000 is added to the Q31 value and then the result is truncated.

ARM added a *Q* flag to the status registers to indicate that overflow or saturation has occurred in DSP instructions.

Addition and subtraction are performed identically no matter which format is used. However, multiplication depends on the type. For example, with 16-bit numbers, the number 0xFFFF is

interpreted as 65535 for unsigned short, -1 for short, and -2^{-15} for Q15 numbers. Hence, 0xFFFF×0xFFFF has a very different value for each representation ($4{,}294{,}836{,}225$; 1; and 2^{-30}, respectively). This leads to different instructions for signed and unsigned multiplication.

A Q15 number A can be viewed as $a \times 2^{-15}$, where a is its interpretation in the range $[-2^{15}, 2^{15}-1]$ as a signed 16-bit number. Hence, the product of two Q15 numbers is:

$$A \times B = a \times b \times 2^{-30} = 2 \times a \times b \times 2^{-31}$$

This means that to multiply two Q15 numbers and get a Q31 result, do ordinary signed multiplication and then double the product. The product can then be truncated or rounded to put it back into Q15 format if necessary.

Floating-Point Instructions

Power-Saving and Security Instructions

The wait for interrupt (WFI) instruction allows the processor to enter a low-power state until an interrupt occurs. The wait for event (WFE) instruction is similar but is helpful in multiprocessor systems so that a processor can go to sleep until notified by another processor.

SIMD Instructions

The term SIMD (pronounced "sim-dee") stands for *single instruction multiple data*, in which a single instruction acts on multiple pieces of data in parallel. A common application of SIMD is to perform many short arithmetic operations at once, especially for graphics processing. This is also called *packed* arithmetic.

64-bit Architecture

6.8 ANOTHER PERSPECTIVE: x86 ARCHITECTURE

x86 is an example of a Complex Instruction Set Computer (CISC) architecture. In contrast to RISC architectures such as ARM, each CISC instruction can do more work.

CHAPTER 7: Microarchitecture

7.1 INTRODUCTION

This chapter covers ***microarchitecture***, which is the connection between **logic** and **architecture**. Microarchitecture is the specific arrangement of registers, ALUs, finite state machines (FSMs), memories, and other logic building blocks needed to **implement** an architecture. (**Implementation** of the ISA under specific **design constraints and goals**)

Recall that a computer architecture is defined by its instruction set and architectural state. The *architectural state* (**AS**) for the ARM processor consists of 16 32-bit registers and the status register (or Memory, Register, and Program counter). Some microarchitectures contain additional *nonarchitectural state* to either simplify the logic or improve performance.

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

- Data-processing instructions: ADD, SUB, AND, ORR (with register and immediate addressing modes but no shifts)
- Memory instructions: LDR, STR (with positive immediate offset)
- Branches: B

These particular instructions were chosen because they are sufficient to write many interesting programs. Once you understand how to implement these instructions, you can expand the hardware to handle others.

Supplement materials

ISA vs. Microarchitecture Level Tradeoff:

- **ISA**: Specifies how the **programmer sees** the instructions to be executed

Programmer sees a sequential, control-flow execution order vs;

Programmer sees a data-flow execution order

Microarchitecture: How the **underlying implementation actually executes** instructions

Microarchitecture can execute instructions in **any** order as long as it **obeys** the **semantics** specified by the ISA when making the instruction results **visible** to software;

Programmer should see the order specified by the ISA;

Design Process

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

Supplement materials

Datapath: Consists of hardware elements that deal with and transform data signals;

Control logic: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data;

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

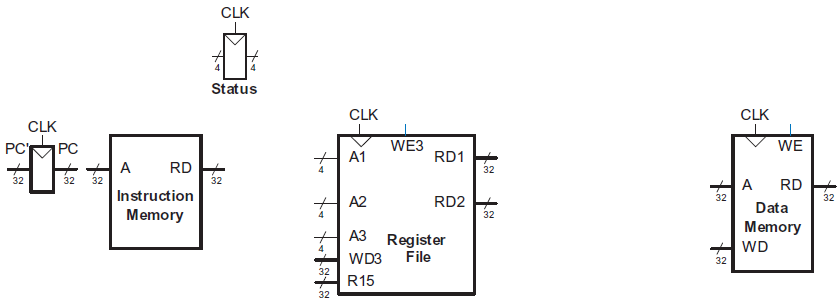


Figure 7. 1 State elements of ARM processor

Although the *program counter* (PC) is logically part of the register file, it is read and written on every cycle independent of the normal register file operation and is more naturally built as a stand-alone 32-bit register. Its output, PC, points to the current instruction. Its input, PC', indicates the address of the next instruction.

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

The 15-element × 32-bit register file holds registers R0–R14 and has an additional input to receive R15 from the PC. The register file has two read ports and one write port. The read ports take 4-bit address inputs, $A1$ and $A2$, each specifying one of $2^4 = 16$ registers as source operands. They read the 32-bit register values onto read data outputs $RD1$ and $RD2$, respectively. The write port takes a 4-bit address input, $A3$; a 32-bit write data input, $WD3$; a write enable input, $WE3$; and a clock. If the write enable is asserted, then the register file writes the data into the specified register on the rising edge of the clock. A read of R15 returns the value from the PC plus 8, and writes to R15 must be specially handled to update the PC because it is separate from the register file.

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

Resetting the PC

At the very least, the program counter must have a reset signal to initialize its value when the processor turns on. ARM processors normally initialize the PC to 0x00000000 on reset, and we start our programs there.

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

Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits. The microprocessor is built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit. Indeed, the processor can be viewed as a giant finite state machine, or as a collection of simpler interacting state machines.

Microarchitectures

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

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

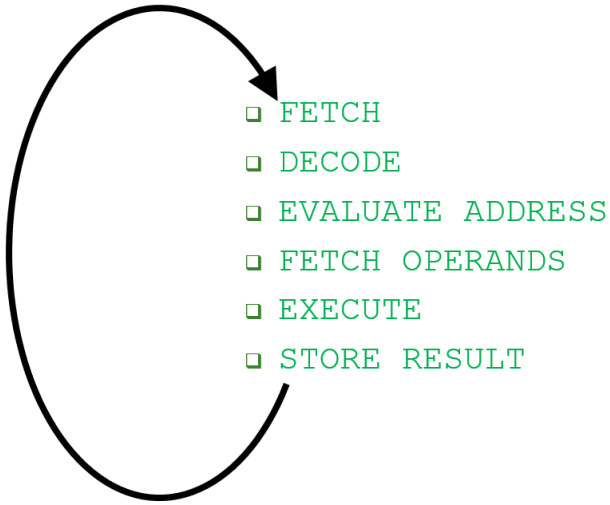
The ***multicycle microarchitecture*** executes instructions in a series of shorter cycles. Simpler instructions execute in fewer cycles than complicated ones. Moreover, the multicycle

microarchitecture reduces the hardware cost by reusing expensive hardware blocks such as adders and memories. For example, the adder may be used on different cycles for several purposes while carrying out a single instruction. The multicycle microprocessor accomplishes this by adding several nonarchitectural registers to hold intermediate results. The multicycle processor executes only one instruction at a time, but each instruction takes multiple clock cycles. The multicycle processor requires only a single memory, accessing it on one cycle to fetch the instruction and on another to read or write data. Therefore, multicycle processors were the historical choice for inexpensive systems.

The ***pipelined microarchitecture*** applies pipelining to the single-cycle microarchitecture. It therefore can execute several instructions simultaneously, improving the throughput significantly. Pipelining must add logic to handle dependencies between simultaneously executing instructions. It also requires nonarchitectural pipeline registers. Pipelined processors must access instructions and data in the same cycle; they generally use separate instruction and data caches for this purpose, as discussed in Chapter 8.

Supplement materials

The Instruction Cycle: (Not all instructions require all six steps)



7.2 PERFORMANCE ANALYSIS

Precise cost calculations require detailed knowledge of the implementation technology but, in general, more gates and more memory mean more dollars.

The only gimmick-free way to measure performance is by measuring the execution time of a program of interest to you. The next best choice is to measure the total execution time of a collection of programs that are similar to those you plan to run. Such collections of programs are called *benchmarks*, and the execution times of these programs are commonly published to give some indication of how a processor performs.

The formula of the execution time of **a program**, measured in seconds is shown:

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

$$Execution\ Time = \{\#\ \text{of instructions}\} \times \{\text{Average CPI}\} \times \{\text{clock cycle time}\}$$

The number of instruct ions ($\#\ instructions$) in a program depends on the processor architecture.

The ***cycles per instruction*** (*CPI*) is the number of clock cycles required to execute an average instruction. It is the reciprocal of the throughput (***instructions per cycle***, or *IPC*).

The number of seconds per cycle is the clock period, T_c. The clock period is determined by the critical path through the logic on the processor.

7.3 SINGLE-CYCLE PROCESSOR

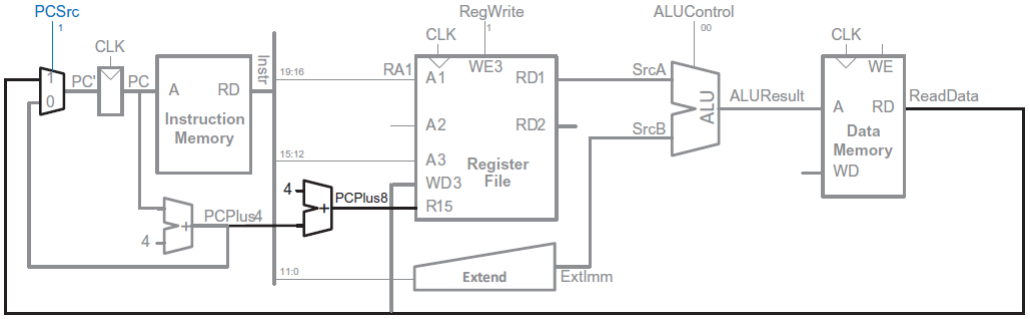
We begin constructing the datapath by connecting the state elements from Figure 7.1 with combinational logic that can execute the various instructions. Control signals determine which

specific instruction is performed by the datapath at any given time. The control unit contains combinational logic that generates the appropriate control signals based on the current instruction.

Single-Cycle Datapath

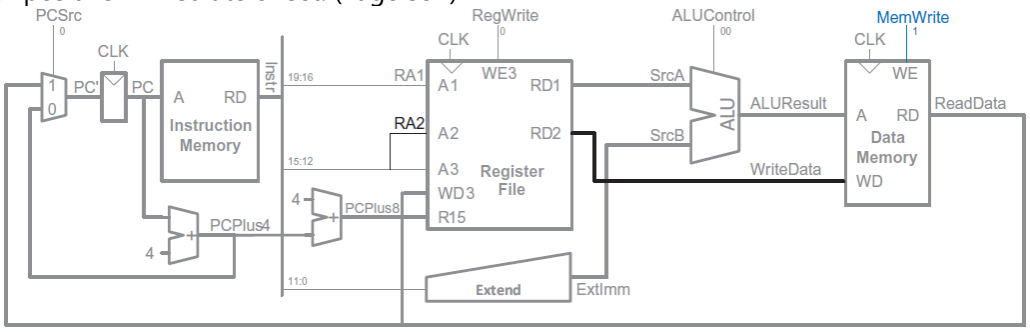
LDR

First, we will work out the datapath connections for the LDR instruction with positive immediate offset. (Page 391)



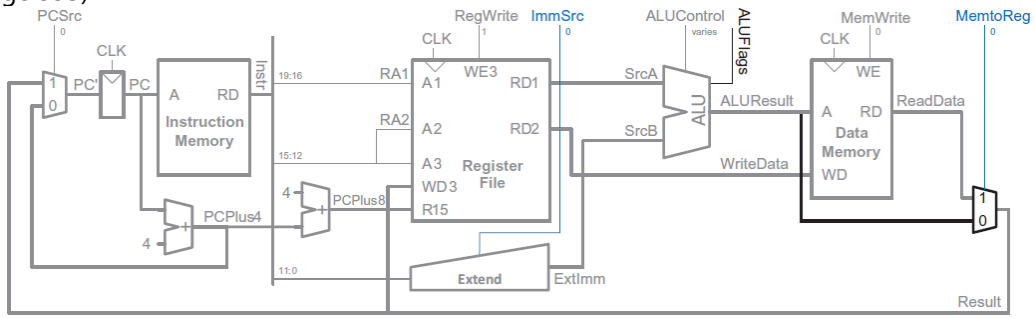
STR

With positive immediate offset. (Page 394)

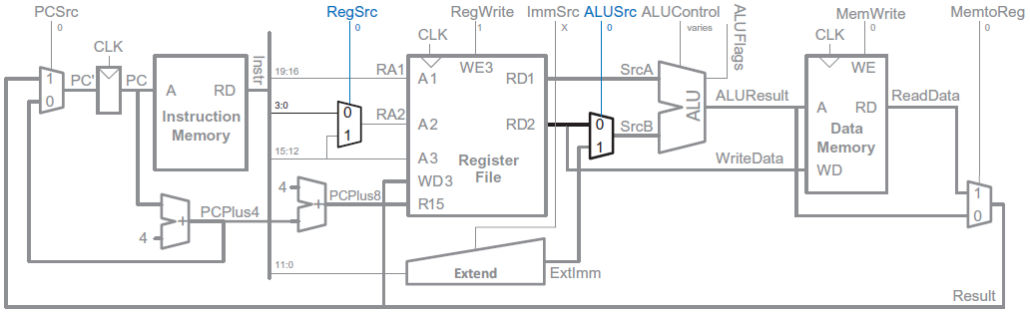


Data-Processing Instructions with Immediate Addressing

(Page 395)

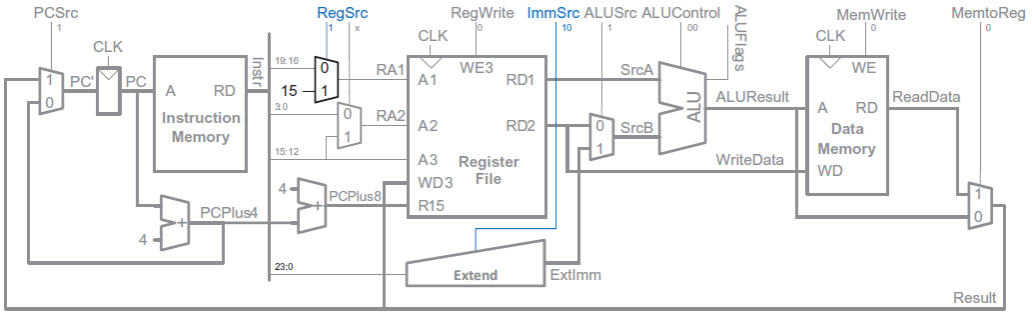


Data-Processing Instructions with Register Addressing



(Page 396)

B



(Page 396)

Single-Cycle Control

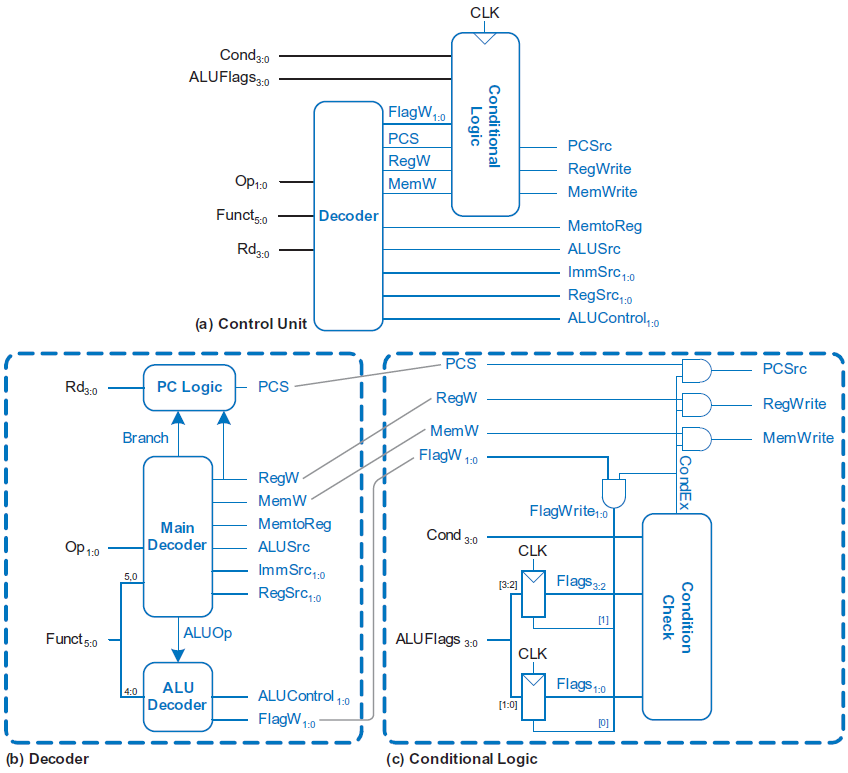


Figure 7. 2 Single-cycle control unit

Example 7.1: SINGLE-CYCLE PROCESSOR OPERATION (Page 401)

More Instructions

Example 7.2: CMP INSTRUCTION (Page 402)

Example 7.3: ENHANCED ADDRESSING MODE: REGISTERS WITH CONSTANT SHIFTS (Page 402)

Performance Analysis

Each instruction in the single-cycle processor takes one clock cycle, so the CPI is 1.

We are disciplining ourselves to **synchronous** sequential design, so the clock period is constant and must be **long enough** to accommodate the **slowest** instruction.

Example 7.4: SINGLE-CYCLE PROCESSOR PERFORMANCE (Page 405)

Ben Bitdiddle is contemplating building the single-cycle processor in a 16-nm CMOS manufacturing process. He has determined that the logic elements have the delays given in Table 7.5. Help him compute the execution time for a program with 100 billion instructions.

Table 7.5 Delay of circuit elements

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Supplement materials

All six phases of the instruction processing cycle take a ***single machine clock cycle*** to complete:

- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result

1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

Example Single-Cycle Datapath Analysis

- Assume (for the design in the previous slide)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

(Micro)architecture Design Principles

- Critical path design
  - Find and decrease the maximum combinational logic delay
  - Break a path into multiple cycles if it takes too long
- Bread and butter (common case) design
  - Spend time and resources on where it matters most
    - i.e., improve what the machine is really designed to do
  - Common case vs. uncommon case
- Balanced design
  - Balance instruction/data flow through hardware components
  - Design to eliminate bottlenecks: balance the hardware for the work

7.4 MULTICYCLE PROCESSOR

The single-cycle processor has three notable weaknesses. First, it requires separate memories for instructions and data. Second, it requires a clock cycle long enough to support the slowest instruction (LDR). Finally, it requires three adders (one in the ALU and two for the PC logic).

The multicycle processor addresses these weaknesses by breaking an instruction into multiple shorter steps.

Multicycle Datapath

In the single-cycle design, we used separate instruction and data memories because we needed to read the instruction memory and read or write the data memory all in one cycle.

LDR

(Page 407)

STR

(Page 411)

Data-Processing Instructions with Immediate Addressing

(Page 412)

Data-Processing Instructions with Register Addressing

(Page 412)

B

(Page 412)

Multicycle Control

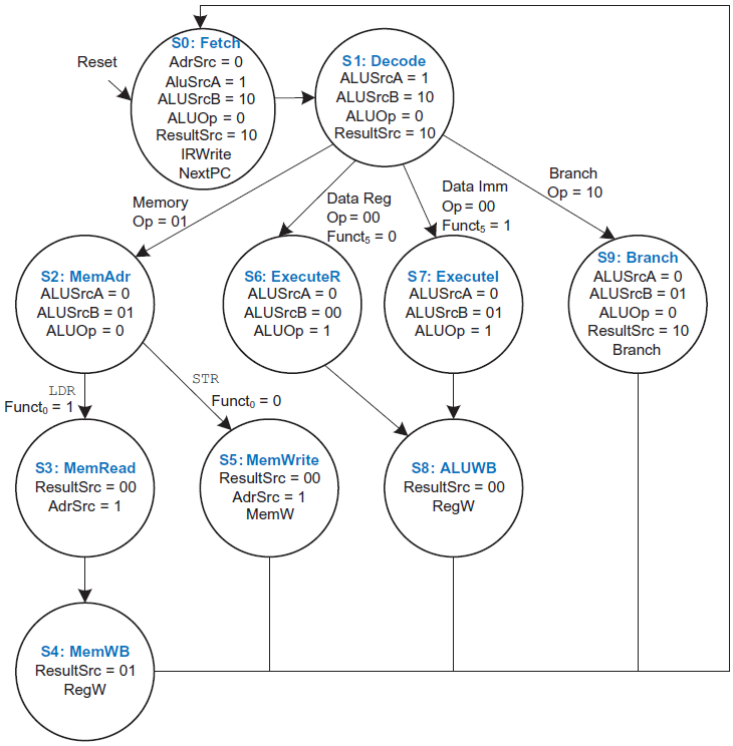


Figure 7. 3 Complete multicycle control FSM

Performance Analysis

Example 7.5: MULTICYCLE PROCESSOR CPI (Page 424)

Example 7.6: PROCESSOR PERFORMANCE COMPARISON (Page 424)

7.5 PIPELINED PROCESSOR

Pipelining, introduced in Section 3.6, is a powerful way to improve the throughput of a digital system. We design a pipelined processor by **subdividing** the **single-cycle processor** into **five pipeline stages**. Thus, five instructions can execute simultaneously, one in each stage. Because each stage has only **one-fifth** of the entire logic, the clock frequency is almost five times **faster**. Hence, the **latency** of each instruction is **ideally** unchanged, but the throughput is **ideally five**-times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. Pipelining introduces some overhead, so the throughput will **not** be quite as high as we might ideally desire, but pipelining nevertheless gives such great advantage for so little cost that all modern high-performance microprocessors are pipelined.

Reading and writing the memory and register file and using the ALU typically constitute the biggest **delays** in the processor. We choose five pipeline stages so that each stage involves exactly one of these slow steps. Specifically, we call the **five stages *Fetch***, ***Decode***, ***Execute***, ***Memory***, and ***Writeback***. They are **similar** to the five steps that the **multicycle processor** used to perform LDR. In the ***Fetch*** stage, the processor reads the instruction from instruction memory. In the ***Decode*** stage, the processor reads the source operands from the register file and decodes the instruction to produce the control signals. In the ***Execute*** stage, the processor performs a computation with the ALU. In the ***Memory*** stage, the processor reads or writes data memory. Finally, in the ***Writeback*** stage, the processor writes the result to the register file, when applicable.

Supplement material

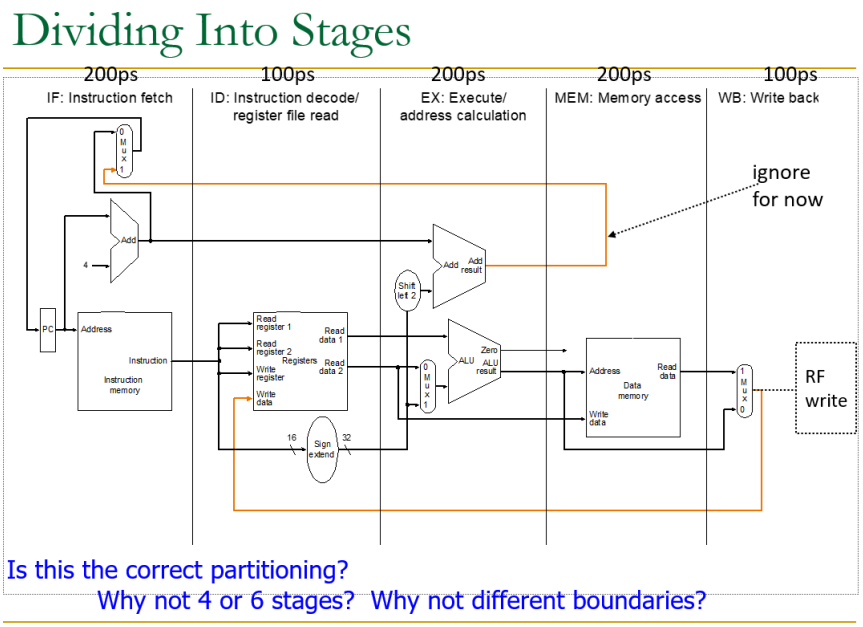


Figure 7.4 shows a timing diagram comparing the single-cycle and pipelined processors. Time is on the horizontal axis, and instructions are on the vertical axis.

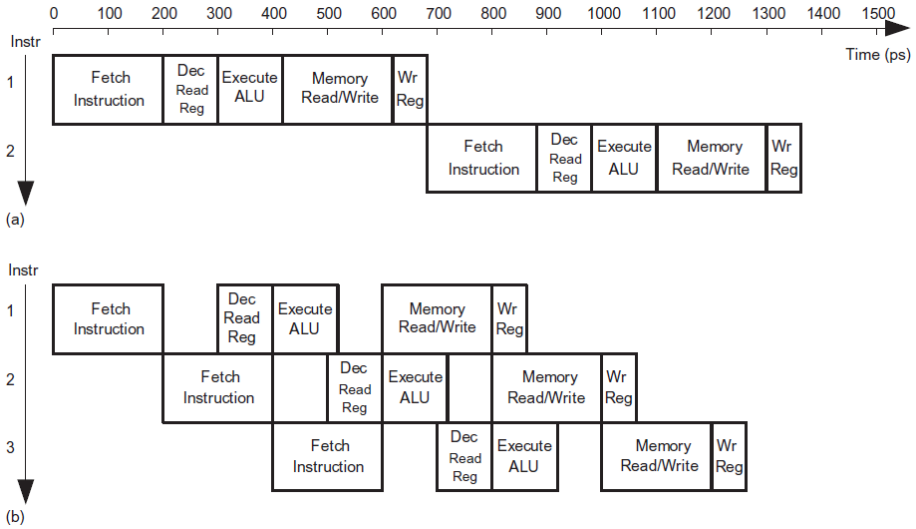


Figure 7. 4 Timing diagrams: (a) single-cycle processor and (b) pipelined processor

In this diagram, the single-cycle processor has an instruction **latency** of $200 + 100 + 120 + 200 + 60 = 680$ ps and a **throughput** of 1 instruction per 680 ps (1.47 billion instructions per second).

In the pipelined processor (Figure 7.4(b)), the length of a pipeline stage is set at 200 ps by the **slowest** stage, the memory access (in the Fetch or Memory stage). The instruction **latency** is $5 \times 200 = 1000$ ps. The **throughput** is 1 instruction per 200 ps (5 billion instructions per second). Because the stages are **not perfectly** balanced with equal amounts of logic, the latency is **longer** for the pipelined processor than for the single-cycle processor. Similarly, the throughput is **not** quite five-times as great for a five-stage pipeline as for the single-cycle processor. Nevertheless, the throughput advantage is **substantial**.

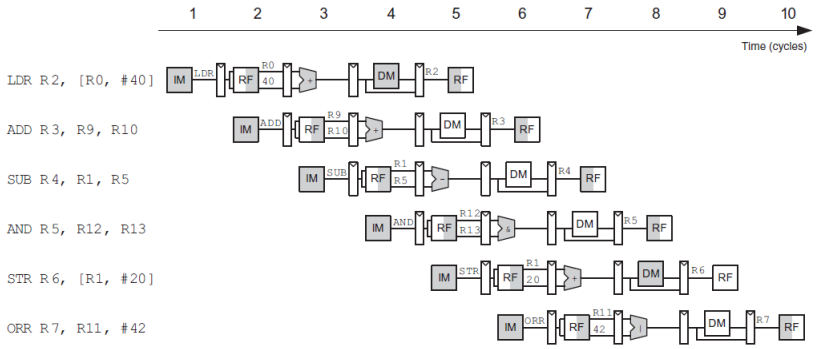


Figure 7. 5 Abstract view of pipeline in operation

Figure 7.5 shows an abstracted view of the pipeline in operation in which each stage is represented pictorially. Each pipeline stage is represented with its major component—instruction memory (IM), register file (RF) read, ALU execution, data memory (DM), and register file writeback— to illustrate the flow of instructions through the pipeline. Reading across a row shows the clock cycles in which a particular instruction is in each stage.

A central challenge in pipelined systems is handling hazards that occur when the results of one instruction are needed by a subsequent instruction before the former instruction has completed. For example, if the ADD in Figure 7.5 used R2 rather than R10, a hazard would occur because the R2 register has not been written by the LDR by the time it is read by the ADD. After designing the pipelined datapath and control, this section explores *forwarding*, *stalls*, and *flushes* as methods to resolve hazards.

Pipelined Datapath

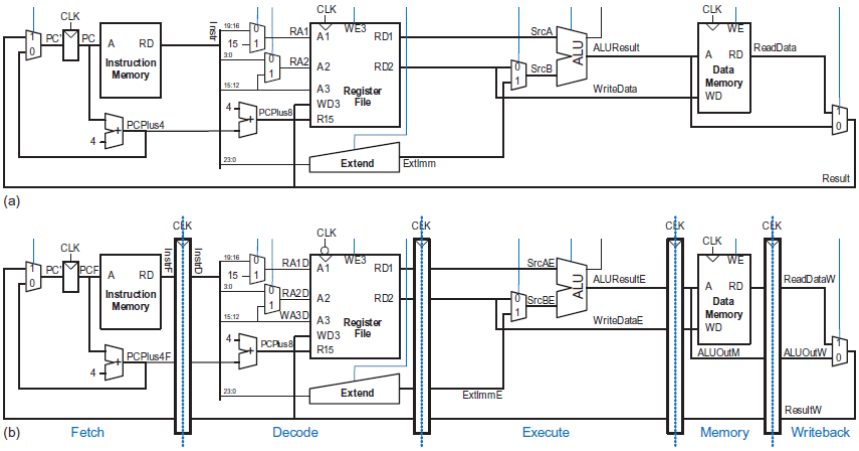


Figure 7. 6 Datapaths: (a) single-cycle and (b) pipelined

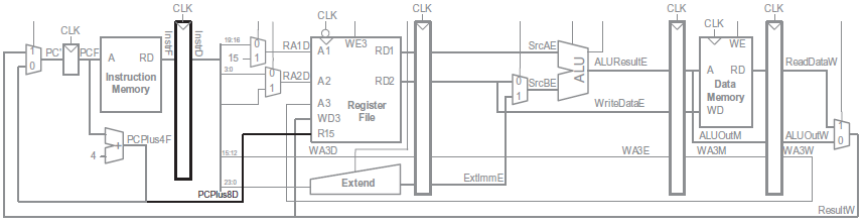


Figure 7. 7 Optimized PC logic eliminating a register and adder

Pipelined Control

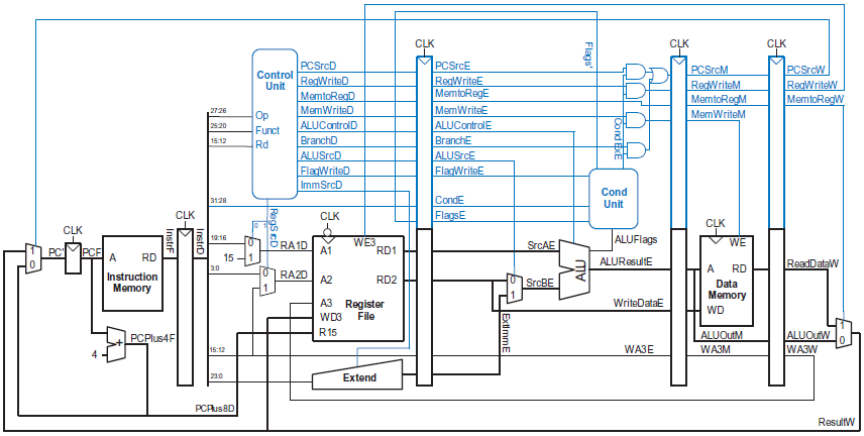


Figure 7. 8 Pipelined processor with control

Hazards

In a pipelined system, multiple instructions are handled **concurrently**. When one instruction is *dependent* on the results of another that has not yet completed, a ***hazard*** occurs. (Also called "**dependency**" or *less desirably* "hazard")

The register file can be **read and written** in the **same cycle**. The write takes place during the **first** half of the cycle and the read takes place during the **second** half of the cycle, so a register can be written and read back in the same cycle without introducing a hazard.

Figure 7.9 illustrates hazards that occur when one instruction writes a register (R1) and subsequent instructions read this register. This is called a ***read after write* (*RAW*) *hazard***.

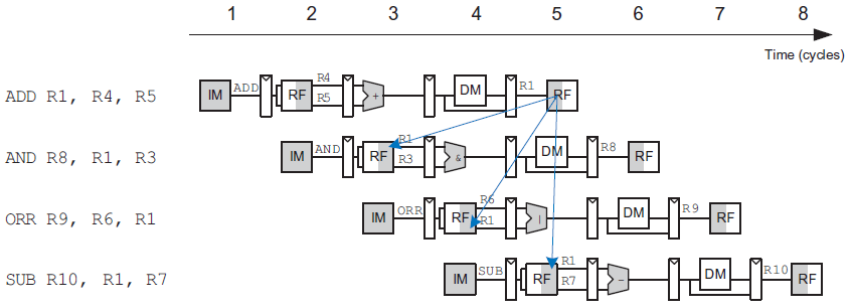


Figure 7. 9 Abstract pipeline diagram illustrating hazards

A software solution would be to require the programmer or compiler to insert NOP instructions between the ADD and AND instructions so that the dependent instruction does not read the result (R1) **until** it is available in the register file, as shown in Figure 7.10. Such a **software interlock complicates** programming as well as degrading performance, so it is not ideal.

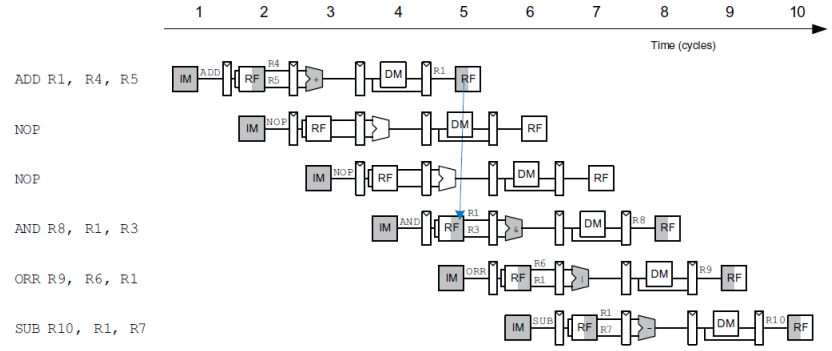


Figure 7. 10 Solving data hazard with NOP

Hazards are classified as data hazards or control hazards. A ***data hazard*** (**Data dependence**) occurs when an instruction tries to read a register that has not yet been written back by a **previous** instruction. A ***control hazard*** (**Control dependence**) occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place.

Solving Data Hazards with Forwarding

Some data hazards can be solved by ***forwarding*** (also called ***bypassing***) a result from the Memory or Writeback stage to a dependent instruction in the Execute stage. This requires adding multiplexers in front of the ALU to select the operand from either the register file or the Memory or Writeback stage. Figure 7.11 illustrates this principle. In cycle 4, R1 is forwarded from the Memory stage of the ADD instruction to the Execute stage of the dependent AND instruction. In cycle 5, R1 is forwarded from the Writeback stage of the ADD instruction to the Execute stage of the dependent ORR instruction.

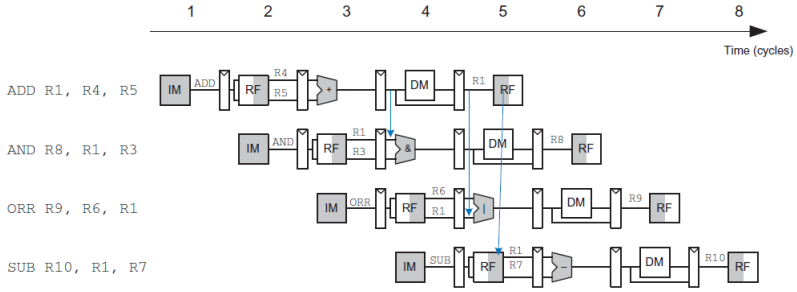


Figure 7. 11 Abstract pipeline diagram illustrating forwarding

Forwarding is **necessary** when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or Writeback stage.

Solving Data Hazards with Stalls

Forwarding is sufficient to solve RAW data hazards when the result is computed in the Execute stage of an instruction, because its result can then be forwarded to the Execute stage of the next instruction. Unfortunately, the LDR instruction does not finish reading data until the end of the Memory stage, so its result cannot be forwarded to the Execute stage of the next instruction. We say that the LDR instruction has a ***two-cycle*** latency, because a dependent instruction cannot use its result until two cycles later. Figure 7.12 shows this problem. The LDR instruction receives data from memory at the end of cycle 4. But the AND instruction needs that data as a source operand at the beginning of cycle 4. There is no way to solve this hazard with forwarding.

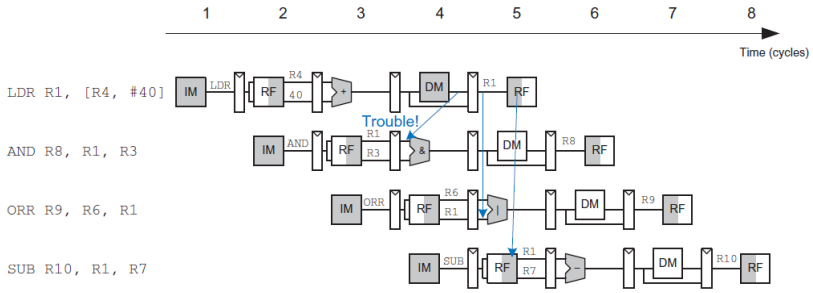


Figure 7. 12 Abstract pipeline diagram illustrating trouble forwarding from LDR

The alternative solution is to ***stall*** the pipeline, holding up operation until the data is available. Figure 7.13 shows stalling the dependent instruction (AND) in the Decode stage.

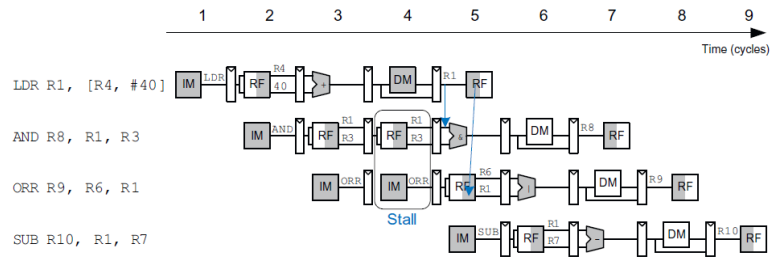


Figure 7. 13 Abstract pipeline diagram illustrating stall to solve hazards

Note that the Execute stage is **unused** in cycle 4. Likewise, Memory is **unused** in cycle 5 and Writeback is **unused** in cycle 6. This unused stage propagating through the pipeline is called a

bubble, and it behaves like a NOP instruction. The bubble is **introduced** by zeroing out the Execute stage control signals during a Decode stall so that the bubble performs no action and changes no architectural state.

In summary, stalling a stage is performed by disabling the **pipeline register**, so that the contents do not change. When a stage is stalled, **all previous** stages must also be stalled, so that no subsequent instructions are lost. The pipeline register **directly after** the stalled stage **must** be cleared (flushed) to prevent bogus information from propagating forward. Stalls degrade performance, so they should be used only when necessary.

Solving Control Hazards

The B instruction presents a control hazard: the pipelined processor does not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction is fetched. Writes to R15 (PC) present a **similar** control hazard.

One mechanism for dealing with the control hazard is to stall the pipeline until the branch decision is made (i.e., *PCSrcW* is computed). Because the decision is made in the Writeback stage, the pipeline would have to be stalled for four cycles at every branch. This would severely degrade the system performance if it occurs often.

An alternative is to **predict** whether the branch will be taken and begin executing instructions based on the prediction. Once the branch decision is available, the processor **can** throw out the instructions if the prediction was wrong. In the pipeline presented **so far**, the processor predicts that branches **are not taken** and **simply** continues executing the program in order until *PCSrcW* is asserted to select the next PC from *ResultW* instead. If the branch should have been taken, then the four instructions following the branch must be ***flushed*** (**discarded**) by clearing the pipeline registers for those instructions. These **wasted** instruction cycles are called the ***branch misprediction penalty***.

Figure 7.14 shows such a scheme in which a branch from address 0x20 to address 0x64 is taken.

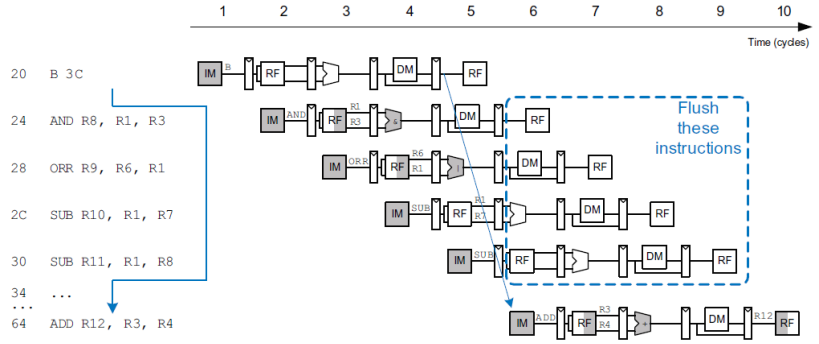


Figure 7. 14 Abstract pipeline diagram illustrating flushing when a branch is taken

We **could reduce** the branch misprediction penalty if the branch decision could be **made earlier**. Observe that the branch decision can be made in the Execute stage when the destination address has been computed and CondEx is known. Figure 7.15 shows the pipeline operation with the early branch decision being made in cycle 3.

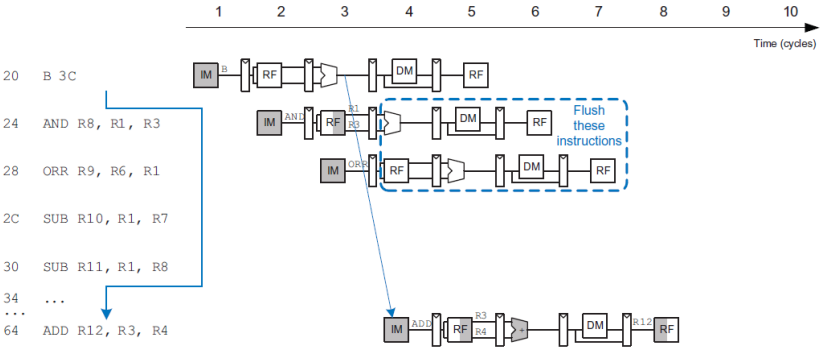


Figure 7. 15 Abstract pipeline diagram illustrating earlier branch decision

Supplement Materials

Types of data dependences:

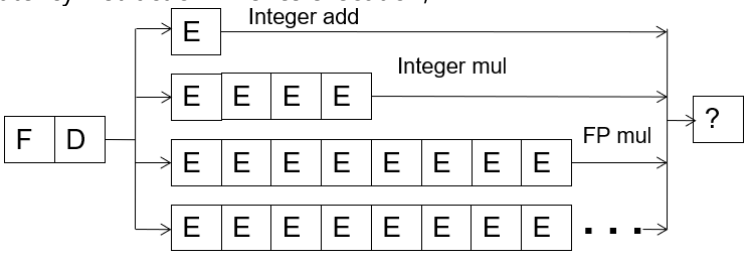
- **Flow dependence** (true data dependence – read after write);
- **Output dependence** (write after write);
- **Anti dependence** (write after read);

Multi-Cycle Execution:

Not all instructions take the **same** amount of time for "execution".

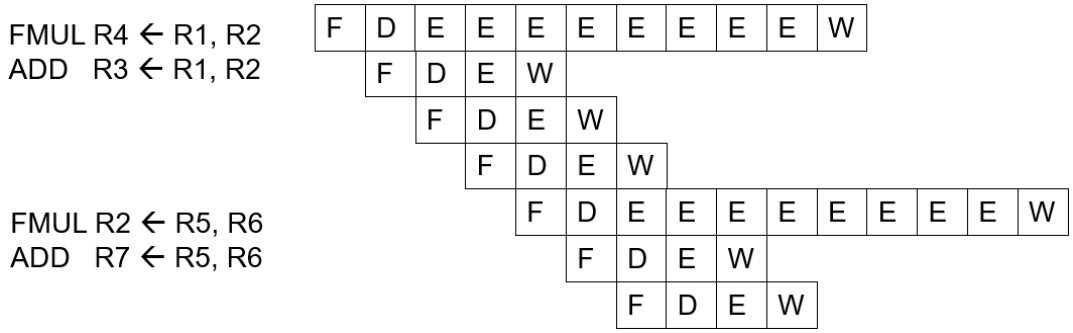
Idea: Have multiple different functional units that take different number of cycles.

- Can be pipelined or not pipelined;
- Can let **independent** instructions start execution on a different functional unit before a previous long-latency instruction finishes execution;



Issues in Pipelining: **Multi-Cycle** Execute (Multi-Cycle Pipeline):

Instructions can take different number of cycles in EXECUTE stage.



What is **wrong** with this picture in a Von Neumann architecture?

- Sequential semantics of the ISA NOT preserved!
- What if FMUL incurs an exception? (there won't have precise exceptions/interrupts)

What are Precise Exceptions/Interrupts?

The architectural state should be **consistent** (precise) when the exception/interrupt is ready to be handled:

1. All **previous** instructions should be completely retired.
2. No **later** instruction should be retired.

(Retire = commit = finish execution and update architectural state)

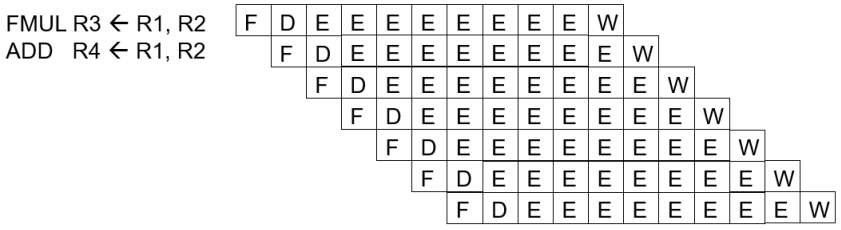
Checking for and Handling Exceptions in Pipelining

- When the oldest instruction ready-to-be-retired is detected to have caused an exception, the control logic
  - Ensures architectural state is precise (register file, PC, memory)
  - Flushes all younger instructions in the pipeline
  - Saves PC and registers (as specified by the ISA)
  - Redirects the fetch engine to the appropriate exception handling routine

Why Do We Want Precise Exceptions?

- Semantics of the von Neumann model ISA specifies it
  - Remember von Neumann vs. Dataflow
- Aids software debugging
- Enables (easy) recovery from exceptions
- Enables (easily) restartable processes
- Enables traps into software (e.g., software implemented opcodes)

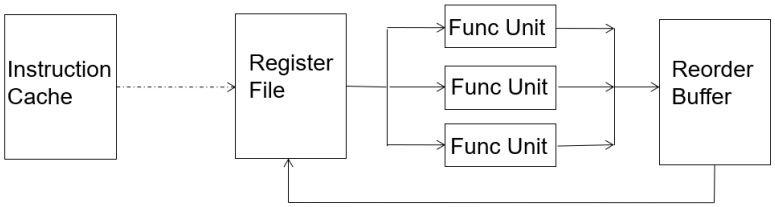
So, Solution I is making each operation take the **same** amount of time,



- Downside
  - Worst-case instruction latency determines all instructions' latency
  - What about memory operations?
  - Each functional unit takes worst-case number of cycles?

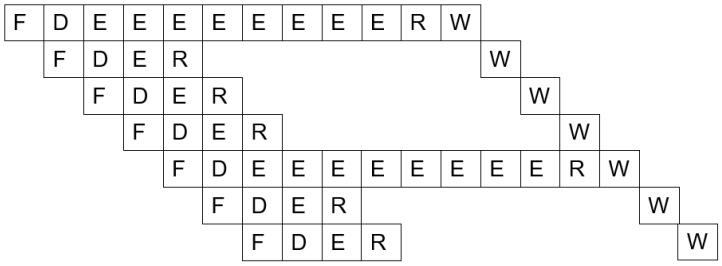
Solution II: **Reorder Buffer** (ROB)

- Idea: Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is decoded it reserves the next-sequential entry in the ROB
- When instruction completes, it writes result into ROB entry
- When instruction oldest in ROB and it has completed without exceptions, its result moved to reg. file or memory



Reorder Buffer: Independent Operations

- Result first written to ROB on instruction completion
- Result written to register file at commit time



- **Advantages**

Conceptually simple for supporting precise exceptions;

Can **eliminate** false dependences (Output and Anti dependence);

- **Disadvantages**

Reorder buffer needs to be accessed to get the results that are yet to be written to the register file; (and still needs stall if true dependence happens)

Performance Analysis

The pipelined processor **ideally** would have a CPI of 1, because a new instruction is issued **every cycle**. However, a stall or a flush **wastes** a cycle, so the CPI is **slightly higher** and depends on the specific program being executed.

Example 7.7: PIPELINED PROCESSOR CPI (Page 441)

Example 7.8: PROCESSOR PERFORMANCE COMPARISON (Page 442)

7.6 HDL REPRESENTATION

In this section, the instruction and data memories are separated from the datapath and connected by address and data busses. In practice, most processors pull instructions and data from separate caches. However, to handle literal pools, a more complete processor must also be able to read data from the instruction memory.

The processor is composed of a datapath and a controller. The controller, in turn, is composed of the Decoder and the Conditional Logic.

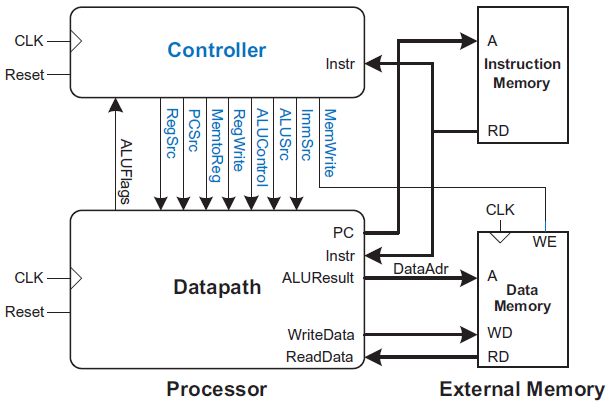


Figure 7. 16 Single-cycle processor interfaced to external memory

Single-Cycle Processor

Generic Building Blocks

Testbench

7.7 ADVANCED MICROARCHITECTURE

High-performance microprocessors use a wide variety of techniques to run programs faster.

Deep Pipelines

Aside from advances in manufacturing, the **easiest** way to speed up the clock is to chop the pipeline into **more** stages. Each stage contains less logic, so it can run faster. This chapter has considered a classic five-stage pipeline, but 10–20 stages are now commonly used.

The maximum number of pipeline stages is limited by pipeline hazards, sequencing overhead, and cost. Longer pipelines introduce **more dependencies**. Some of the dependencies can be solved by forwarding but others require stalls, which increase the CPI. The pipeline registers between each stage have **sequencing overhead** from their setup time and clk-to-Q delay (as well as clock skew). This sequencing overhead makes adding more pipeline stages give diminishing returns. Finally, adding more stages increases the **cost** because of the extra pipeline registers and hardware required to handle hazards.

Example 7.9 Consider building a pipelined processor by chopping up the single-cycle processor into N stages. (Page 457)

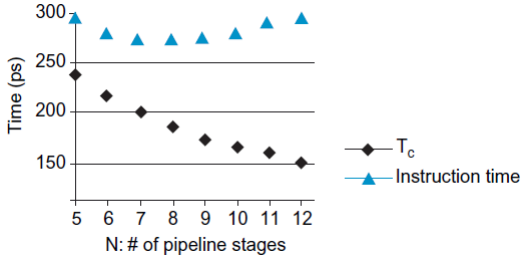


Figure 7. 17 Cycle time and instruction time vs. the number of pipeline stages

Micro-Operations

Recall our design principles of "regularity supports simplicity" and "make the common case fast." Pure reduced instruction set computer (RISC) architectures such as MIPS contain only simple instructions, typically those that can be executed in a single cycle on a simple, fast datapath with a three-ported register file, single ALU, and single data memory access like the ones we have developed in this chapter. Complex instruction set computer (CISC) architectures generally include instructions requiring more registers, more additions, or more than one memory access per instruction.

Computer architects make the common case **fast** by defining a set of simple ***micro-operations*** (also known as ***micro-ops*** or ***μops***) that can be executed on simple datapaths. **Each real** instruction is decoded into **one or more** micro-ops.

Although most ARM instructions are simple, some are decomposed into multiple micro-ops as well.

Although the programmer could have written the simpler instructions directly and the program may have run just as fast, a single complex instruction takes **less** memory than the pair of simpler instructions. Reading instructions from external memory can consume significant **power**, so the complex instruction also can save power.

Branch Prediction

An **ideal** pipelined processor would have a CPI of 1.0. The branch misprediction penalty is a **major reason** for increased CPI. As pipelines get deeper, branches are resolved later in the pipeline. Thus, the branch misprediction penalty gets larger because all the instructions issued after the mispredicted branch must be flushed. To address this problem, most pipelined processors use a ***branch predictor*** to guess whether the branch should be taken.

Some branches occur when a program reaches the **end** of a loop and branches back to repeat the loop (e.g., in a for or while loop). Loops tend to be executed many times, so these backward branches are usually taken. The **simplest** form of branch prediction checks the direction of the branch and predicts that **backward branches should be taken**. This is called *static branch prediction*, because it does **not** depend on the **history** of the program.

Forward branches are **difficult** to predict without knowing more about the specific program. Therefore, most processors use *dynamic branch predictors*, which use the **history** of program execution to **guess** whether a branch should be taken. Dynamic branch predictors maintain a **table** of the last several hundred (or thousand) branch instructions that the processor has executed. The table, called a *branch target buffer*, includes the **destination** of the branch and a **history** of whether the branch was taken.

A ***one-bit dynamic branch predictor*** remembers whether the branch was taken the last time and predicts that it will do the same thing the next time.

A ***two-bit dynamic branch predictor*** solves this problem by having **four** states: strongly taken, weakly taken, weakly not taken, and strongly not taken, as shown in Figure 7.18.

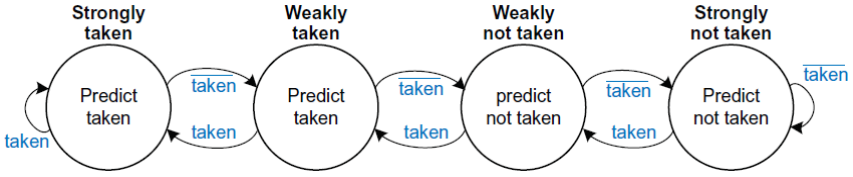


Figure 7. 18 Two-bit branch predictor state transition diagram

The branch predictor operates in the **Fetch stage of the pipeline** so that it can determine which instruction to execute on the next cycle. When it predicts that the branch should be taken, the processor fetches the next instruction from the branch destination stored in the branch target buffer.

Superscalar Processor

A *superscalar processor* contains multiple copies of the datapath hardware to execute multiple instructions simultaneously. Figure 7.19 shows a block diagram of a two-way superscalar processor that **fetches** and **executes two** instructions per cycle. The datapath fetches two instructions at a time from the instruction memory. It has a six-ported register file to read four source operands and write two results back in each cycle. It also contains two ALUs and a two-ported data memory to execute the two instructions at the same time.

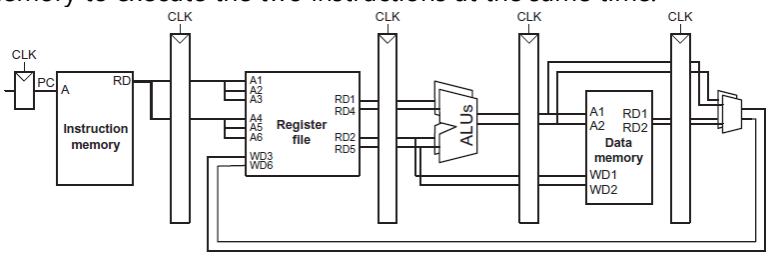


Figure 7. 19 Superscalar datapath

Figure 7.20 shows a pipeline diagram illustrating the two-way superscalar processor executing two instructions on each cycle. For this program, the processor has a CPI of 0.5. Designers commonly refer to the reciprocal of the CPI as the ***instructions per cycle***, or IPC. This processor has an IPC of 2 on this program.

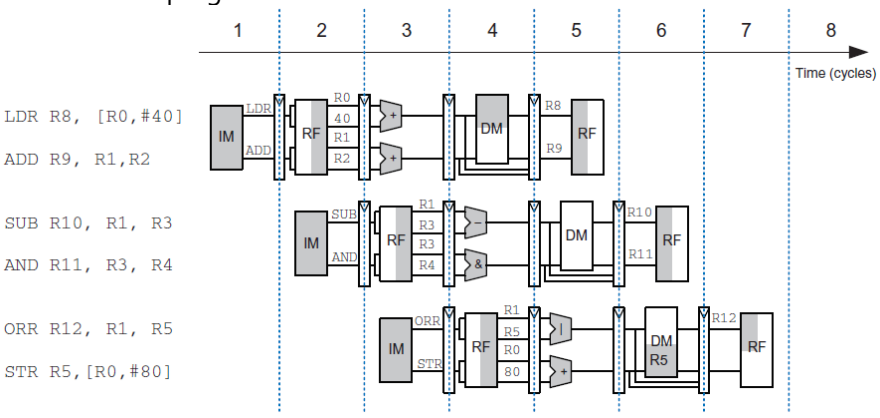


Figure 7. 20 Abstract view of a superscalar pipeline in operation

Executing many instructions simultaneously is **difficult** because of dependencies. For example, Figure 7.21 shows a pipeline diagram running a program with data dependencies.

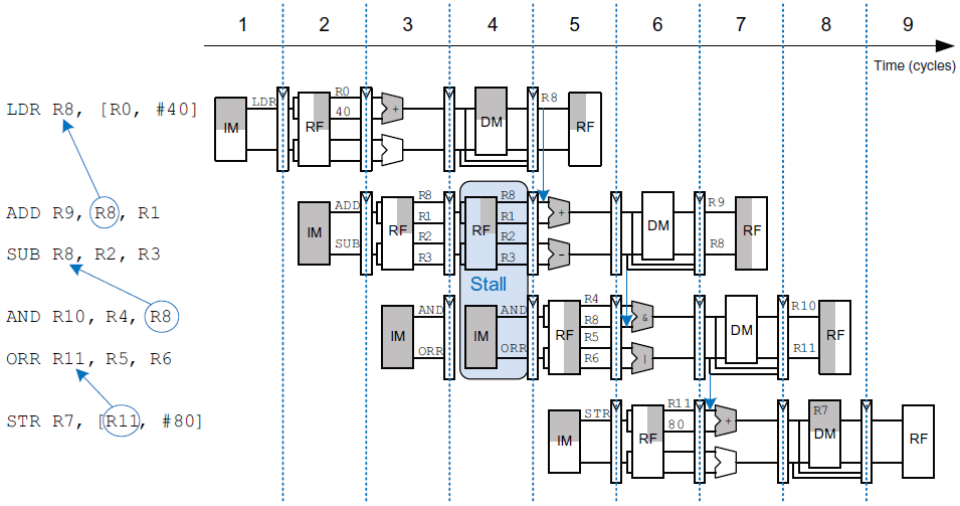


Figure 7. 21 Program with data dependencies

This program requires five cycles to issue six instructions, for an IPC of 1.2.

Recall that parallelism comes in temporal and spatial forms. Pipelining is a case of temporal parallelism. Multiple execution units is a case of spatial parallelism. Superscalar processors **exploit both** forms of parallelism to squeeze out performance far exceeding that of our single-cycle and multicycle processors.

A *scalar* processor acts on one piece of data at a time.

A *vector* processor acts on several pieces of data with a **single** instruction.

A *superscalar* processor issues several instructions at a time, each of which operates on one piece of data.

Our ARM pipelined processor is a scalar processor. Vector processors were popular for supercomputers, and they are heavily used now in *graphics processing units* (*GPUs*).

Modern high-performance microprocessors are **superscalar**. However, modern processors also include hardware to handle short **vectors** of data that are common in multimedia and graphics applications. These are called *single instruction multiple data* (*SIMD*) units.

Out-of-Order Processor

To cope with the problem of **dependencies**, an out-of-order processor **looks ahead across** many instructions to ***issue***, or begin executing, independent instructions as rapidly as possible.

The instructions can issue in a **different order** than that written by the programmer, as long as dependencies are honored so that the program produces the intended result.
Consider running the **same** program from Figure 7.21 on a two-way superscalar out-of-order processor. The processor can issue up to two instructions per cycle from anywhere in the program, as long as dependencies are observed. Figure 7.22 shows the data dependencies and the operation of the processor.

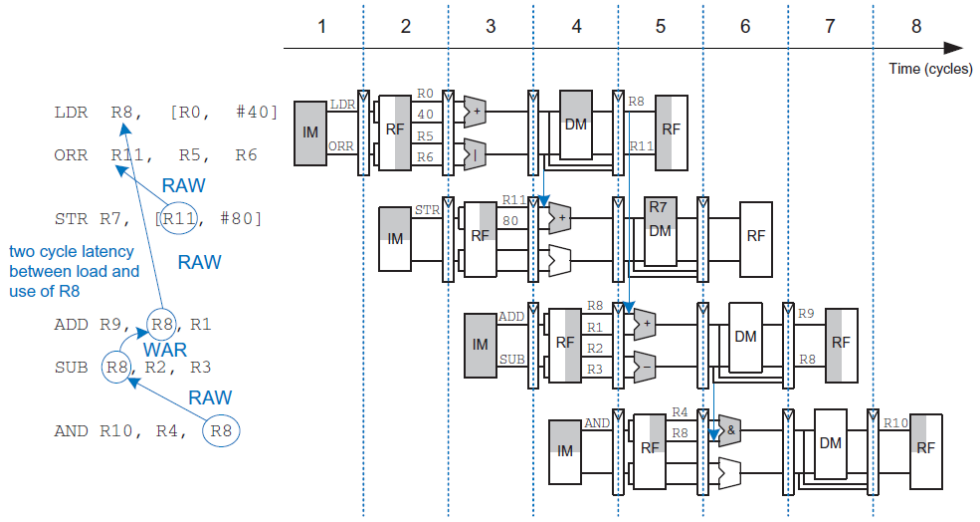


Figure 7. 22 Out-of-order execution of a program with dependencies

The out-of-order processor issues the six instructions in four cycles, for an IPC of 1.5.
The dependence of ADD on LDR by way of R8 is a ***read after write*** (RAW) hazard. ADD must not read R8 until after LDR has written it. This is the type of dependency we are **accustomed to** handling in the **pipelined** processor. It inherently limits the speed at which the program can run, even if infinitely many execution units are available. Similarly, the dependence of STR on ORR by way of R11 and of AND on SUB by way of R8 are RAW dependencies.
The dependence between SUB and ADD by way of R8 is called a ***write after read*** (WAR) hazard or an ***antidependence***. SUB must not write R8 before ADD reads R8, so that ADD receives the correct value according to the original order of the program. WAR hazards could not occur in the **simple** pipeline, but they may happen in an out-of-order processor if the dependent instruction (in this case, SUB) is moved too early.
A WAR hazard is **not essential** to the operation of the program. It is merely an **artifact** of the programmer's choice to use the same register for two unrelated instructions. If the SUB instruction had written R12 instead of R8, then the dependency would **disappear** and SUB could be issued before ADD. The ARM architecture **only** has 16 registers, so sometimes the programmer is **forced** to reuse a register and introduce a hazard just because all the other registers are in use.
A third type of hazard, not shown in the program, is called a ***write after write*** (WAW) hazard or an ***output dependence***. A WAW hazard occurs if an instruction attempts to write a register after a subsequent instruction has **already** written it. The hazard would result in the **wrong value** being written to the register. For example, in the following code, LDR and ADD both write R8. The final value in R8 should come from ADD according to the order of the program. If an out-of-order processor attempted to execute ADD first, then a WAW hazard would occur.

```
LDR  R8,  [R3]
ADD  R8,  R1,  R2
```

WAW hazards are **not essential** either; again, they are **artifacts** caused by the programmer's using the same destination register for two unrelated instructions. If the ADD instruction were issued first, then the program **could eliminate** the WAW hazard by **discarding** the result of the LDR instead of writing it to R8. This is called ***squashing*** the LDR. (the LDR needs to be issued because the out-of-order processors must guarantee that all of the same exceptions occur that would have occurred if the program had been executed in its original order. The LDR potentially

may produce a Data Abort exception, so it must be issued to check for the exception, even though the result can be discarded.)
Out-of-order processors **use a table** to keep **track** of instructions **waiting** to **issue**. The table, sometimes called a ***scoreboard***, contains information about the dependencies. The **size** of the table determines how many instructions can be considered for issue. On **each** cycle, the processor examines the table and issues as many instructions as it can, **limited** by the dependencies and by the number of execution units (e.g., ALUs, memory ports) that are available.
The ***instruction level parallelism*** (ILP) is the number of instructions that can be executed **simultaneously** for a particular program and microarchitecture.

Supplement materials
● Superscalar execution and out-of-order execution are orthogonal concepts
- Can have all four combinations of processors:
[in-order, out-of-order] x [scalar, superscalar]
Superscalar execution is like working on y-axis, pipelining is like working on x-axis.

Register Renaming
Out-of-order processors use a technique called ***register renaming*** to eliminate WAR and WAW hazards. Register renaming adds **some** nonarchitectural renaming registers to the processor. For example, a processor might add 20 renaming registers, called T0-19. The programmer **cannot** use these registers **directly**, because they are not part of the architecture. However, the processor is **free** to use them to eliminate hazards.
For example, in the previous section (Out-of-order processors), a WAR hazard occurred between the SUB and ADD instructions based on reusing R8. The out-of-order processor could rename R8 to T0 for the SUB instruction. Then, SUB could be executed **sooner**, because T0 has no dependency on the ADD instruction. The processor **keeps a table** of which registers were renamed so that it can consistently rename registers in subsequent dependent instructions. In this example, R8 must also be renamed to T0 in the AND instruction, because it refers to the result of SUB.
Figure 7.23 shows the same program from Figure 7.21 executing on an **out-of-order** processor with **register renaming**. R8 is renamed to T0 in SUB and AND to eliminate the WAR hazard.

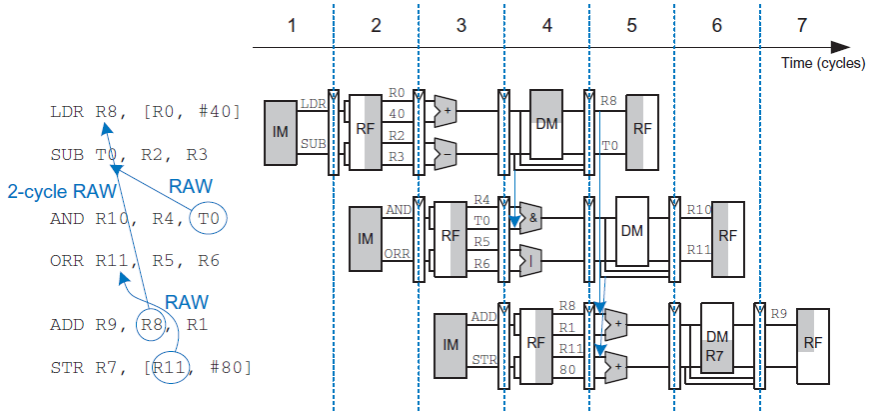


Figure 7. 23 Out-of-order execution of a program using register renaming
The out-of-order processor with register renaming issues the six instructions in three cycles, for an IPC of 2.

Multithreading
Most loads and stores access a smaller and faster memory, called a *cache*.
Multithreading is a technique that helps keep a processor with many execution units busy even if the ILP of a program is low or the program is stalled waiting for memory.

A program running on a computer is called a *process*. Computers can run multiple processes simultaneously. Each process consists of one or more *threads* that also run simultaneously. The degree to which a process can be split into multiple threads that can run simultaneously defines its level of *thread level parallelism* (TLP).

In a conventional processor, the threads only give the illusion of running simultaneously. The threads actually take turns being executed on the processor under control of the OS. When one thread's turn ends, the OS saves its architectural state, loads the architectural state of the next thread, and starts executing that next thread. This procedure is called *context switching*.

A multithreaded processor contains more than one copy of its architectural state, so that more than one thread can be active at a time.

Multithreading does not improve the performance of an individual thread, because it does not increase the ILP. However, it does improve the overall throughput of the processor, because multiple threads can use processor resources that would have been idle when executing a single thread.

Multiprocessors

Around the year 2005, computer architects made a major shift to building multiple copies of the processor on the same chip; these copies are called *cores*.

A *multiprocessor* system consists of multiple processors and a method for communication between the processors. Three common classes of multiprocessors include *symmetric* (or *homogeneous*) multiprocessors, *heterogeneous* multiprocessors, and *clusters*.

Symmetric Multiprocessors

Symmetric multiprocessors include two or more identical processors sharing a single main memory. The multiple processors may be separate chips or multiple cores on the same chip.

Heterogeneous Multiprocessors

Heterogeneous systems are good for cases that have more varying or special-purpose workloads.

Clusters

In clustered multiprocessors, each processor has its own local memory system. One type of cluster is a group of personal computers connected together on the network running software to jointly solve a large problem. Another type of cluster that has become very important is the *data center*.

7.8 REAL-WORLD PERSPECTIVE: EVOLUTION OF ARM MICROARCHITECTURE

CHAPTER 7: Memory Systems

8.1 INTRODUCTION

Computer system performance depends on the memory system as well as the processor microarchitecture. Chapter 7 assumed an ideal memory system that could be accessed in a single clock cycle. **However**, this would be true **only** for a very small memory—or a very slow processor! Early processors were relatively slow, so memory was able to keep up. But processor speed has increased at a faster rate than memory speeds. **DRAM** memories are currently 10 to 100 times **slower** than processors. The increasing gap between processor and DRAM memory speeds demands increasingly ingenious memory systems to try to approximate a memory that is as fast as the processor (memory wall problem). This chapter investigates memory systems and considers trade-offs of **speed, capacity, and cost**.

The processor communicates with the memory system over a *memory interface*. Figure 8.1 shows the simple memory interface used in our multicycle ARM processor. The processor sends an address over the *Address* bus to the memory system. For a read, *MemWrite* is 0 and the memory returns the data on the *ReadData* bus. For a write, *MemWrite* is 1 and the processor sends data to memory on the *WriteData* bus.

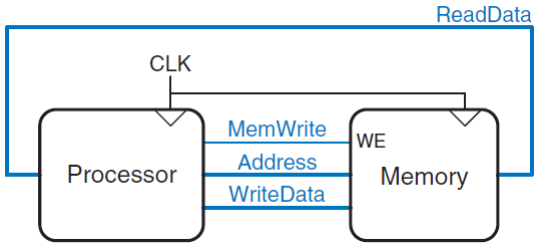


Figure 8. 1 The memory interface

The **major issues** in memory system design can be broadly explained using a metaphor of books in a library. (Page 487)

This metaphor **emphasizes** the **principle**, introduced in Section 6.2.1, of making the common case **fast**. By keeping books that you have recently used or might likely use in the future at your cubicle, you reduce the number of time-consuming trips to the stacks. In particular, you use the principles of *temporal* and *spatial locality*. Temporal locality means that if you have used a book recently, you are likely to use it again soon. Spatial locality means that when you use one particular book, you are likely to be interested in other books on the same shelf.

Memory subsystems used to build this hierarchy were introduced in Section 5.5. Computer memories are primarily built from dynamic RAM (**DRAM**) and static RAM (**SRAM**). Ideally, the computer memory system is fast, large, and cheap. In practice, a single memory only has two of these three attributes; it is either slow, small, or expensive. But computer systems can approximate the ideal by **combining** a fast small cheap memory and a slow large cheap memory.

In Figure 8.2, the plot shows memory (DRAM) and processor speeds with the 1980 speeds as a **baseline**. In about 1980, processor and memory speeds were the same. But performance has diverged since then, with memories badly lagging.

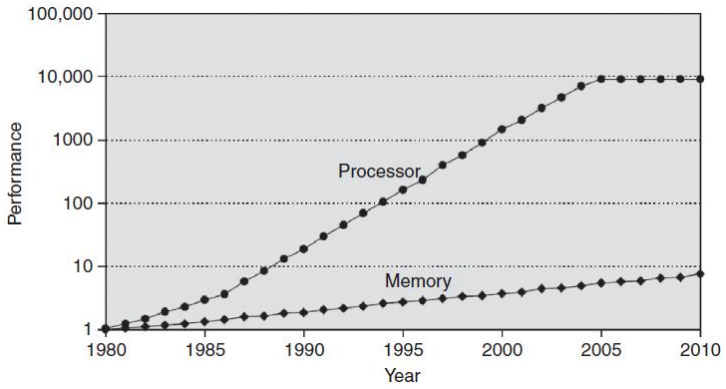


Figure 8. 2 Diverging processor and memory performance

The DRAM access time is one to two orders of magnitude longer than the processor cycle time (tens of nanoseconds, compared to less than one nanosecond).

To **counteract** this trend, computers store the **most commonly** used instructions and data in a faster but smaller memory, called a ***cache***. The cache is usually built out of SRAM on the **same chip** as the processor. The cache speed is **comparable** to the processor speed, because SRAM is inherently **faster** than DRAM, and because the on-chip memory **eliminates** lengthy delays caused by traveling to and from a separate chip. Caches can store both instructions and data, but we will refer to their contents generically as "**data**."

If the processor **requests data** that is available in the cache, it is returned quickly. This is called a **cache *hit***. Otherwise, the processor retrieves the data from **main memory** (DRAM). This is called a **cache *miss***. If the cache hits most of the time, then the processor seldom has to wait for the slow main memory, and the average access time is low.

The third level in the memory hierarchy is the hard drive. In the same way that a library uses the basement to store books that do not fit in the stacks, computer systems use the hard drive to store data that **does not** fit in main memory. In 2015, a **hard disk drive** (HDD), built using magnetic. **Solid state drives** (SSDs), built using flash memory technology, are an increasingly common alternative to HDDs.

The hard drive provides an **illusion** of more capacity than actually exists in the main memory. It is thus called **virtual memory**. Like books in the basement, data in virtual memory takes a **long time** to access. Main memory, also called **physical memory**, holds a subset of the virtual memory. **Hence**, the main memory can be viewed as a cache for the most commonly used data from the hard drive.

Figure 8.3 summarizes the memory hierarchy of the **computer system**. The processor first seeks data in a small but fast cache that is usually located on the same chip. If the data is not available in the cache, the processor then looks in main memory. If the data is not there either, the processor fetches the data from virtual memory on the large but slow hard disk. Figure 8.4 illustrates this capacity and speed trade-off in the memory hierarchy and lists typical **costs**, **access times**, and **bandwidth** in 2015 technology. As access time decreases, speed increases.

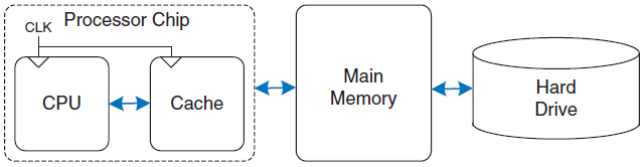
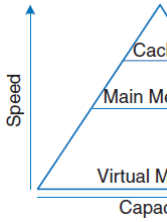


Figure 8. 3 A typical memory hierarchy



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$5,000	0.5	25+
DRAM	$7	10–50	10
SSD	$0.40	20,000	0.5
HDD	$0.05	5,000,000	0.75

Figure 8. 4 Memory hierarchy components, with typical characteristics in 2015

8.2 MEMORY SYSTEM PERFORMANCE ANALYSIS

Designers (and computer buyers) need **quantitative ways** to measure the performance of **memory systems** to evaluate the cost-benefit trade-offs of various alternatives. Memory system performance metrics are ***miss rate*** or ***hit rate*** and ***average memory access time***. Miss and hit rates are calculated as:

$$\text{Miss Rate} = \frac{\text{Number of misses}}{\text{Number of total memory accesses}} = 1 - \text{Hit Rate}$$

$$\text{Hit Rate} = \frac{\text{Number of hits}}{\text{Number of total memory accesses}} = 1 - \text{Miss Rate}$$

Example 8.1: CALCULATING CACHE PERFORMANCE (Page 491)

Average memory access time (*AMAT*) is the average time a processor **must wait** for memory per **load** or **store** instruction. In the typical computer system from Figure 8.3, the processor first looks for the data in the cache. If the cache misses, the processor then looks in main memory. If the main memory misses, the processor accesses virtual memory on the hard disk. Thus, *AMAT* is calculated as:

$$\text{AMAT} = t_{\text{cache}} + MR_{\text{cache}}(t_{MM} + MR_{MM}t_{VM})$$

where t_{cache}, t_{MM}, and t_{VM} are the access times of the cache, main memory, and virtual memory, and MR_{cache} and MR_{MM} are the cache and main memory miss rates, respectively.

Note here (AMAT) when the memory access is serial between cache and main memory, we don't need to multiply t_{cache} with cache's hit rate MH_{cache}. But when this memory access is parallel, we need to do it:

$$\begin{aligned}\text{AMAT} &= (1 - MR_{\text{cache}})t_{\text{cache}} + MR_{\text{cache}}(t_{MM} + MR_{MM}t_{VM}) \\ &= MH_{\text{cache}}t_{\text{cache}} + MR_{\text{cache}}(t_{MM} + MR_{MM}t_{VM})\end{aligned}$$

(This can be proved by a simple example: assuming there are 10 access, 5 hits and 5 misses with $t_{\text{cache}} = 1$ ns and $t_{MM} = 5$ ns)

Example 8.2: CALCULATING AVERAGE MEMORY ACCESS TIME (Page 492)

Example 8.3: IMPROVING ACCESS TIME (Page 492)

As a word of caution, performance improvements **might not always** be as good as they sound. For example, making the memory system ten times faster will **not** necessarily make a computer program run ten times as fast. If 50% of a program's performance is due to loads and stores, a tenfold memory system improvement only means a 1.82-fold improvement in program performance. This general principle is called ***Amdahl's Law***, which says that the effort spent on increasing the performance of a subsystem is worthwhile **only** if the subsystem affects a large percentage of the overall performance.

Supplement materials

- Amdahl's law

Amdahl's law can be formulated in the following way:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

where S_{latency} is the theoretical speedup of the execution of the whole task; s is the speedup of the part of the task that benefits from improved system resources; p is the proportion of execution time that the part benefiting from improved resources originally occupied.

8.3 CACHES

A cache holds **commonly used memory data**. The **number** of **data words** that it can hold is called the ***capacity***, C. Because the capacity of the cache is smaller than that of main memory, the computer system designer **must choose what subset** of the main memory is kept in the cache.

When the processor attempts to access data, it **first** checks the cache for the data. If the cache hits, the data is available immediately. If the cache misses, the processor fetches the data from main memory **and** places it in the cache for future use. To accommodate the **new** data, the cache must *replace* old data. This section investigates these issues in cache design by answering the following questions: (1) What data is **held** in the cache? (2) How is data **found**? and (3) What data is **replaced** to make room for **new** data when the cache is **full**?

Caches use spatial and temporal locality to predict what data will be needed next. If a program accesses data in a random order, it would not benefit from a cache (Can I improve it?).

As we explain in the following sections, caches are specified by their capacity (C), number of sets (S), block size (b), number of blocks (B), and degree of associativity (N). ($B = C/b, S = B/N$)

What Data is Held in the Cache?

An **ideal** cache would **anticipate all** of the data needed by the processor and **fetch** it from main memory ahead of time so that the cache has a **zero** miss rate. Because it is **impossible** to predict the future with perfect accuracy, the cache **must guess** what data will be needed based on the **past pattern** of **memory accesses**. In particular, the cache exploits temporal and spatial locality to achieve a low miss rate.

Recall that **temporal locality** means that the processor is likely to access a piece of data again soon if it has accessed that data recently. Therefore, when the processor loads or stores data that is not in the cache, the data is **copied** from main memory into the cache. Subsequent requests for that data hit in the cache.

Recall that **spatial locality** means that, when the processor accesses a piece of data, it is also likely to access data in nearby memory locations. Therefore, when the cache fetches one word from memory, it may also fetch **several** adjacent words. **This group** of **words** is called a ***cache block*** or ***cache line***. The **number** of **words** in the cache block, b, is called the **block size**. A cache of capacity C contains $B = C/b$ **blocks**. (e.g., 64 bits)

How is Data Found?

A cache is organized into S ***sets***, each of which holds one or more **blocks** of data. The **relationship** between the **address** of **data** in main memory and the **location** of **that data** in the cache is called the ***mapping***. **Each** memory address maps to **exactly one set** in the cache. Some of the **address bits** are used to determine which cache set contains the data. If the set contains **more than one** block, the data may be kept in **any** of the blocks in the set.

Caches are **categorized** based on **the number of blocks in a set**. In a ***direct mapped*** cache, each set contains exactly **one block**, so the cache has $S = B$ sets. Thus, a particular main memory address maps to a **unique block** in the cache. In an ***N-way set associative*** cache, each set contains N **blocks**. The address **still** maps to a **unique set**, with $S = B/N$ sets. But the data from that address can go in **any** of the N blocks in that set. A ***fully associative*** cache has only $S = 1$ set ($N = B$). Data can go in **any** of the B blocks in the set. Hence, a fully associative cache is another name for a ***B*-way** set associative cache.

To illustrate these cache organizations, we will consider an ARM memory system with 32-bit addresses and 32-bit words. The memory is byte-addressable, and each word is **four** bytes, so the memory consists of 2^{30} words aligned on **word boundaries**. ($2^{32}/4 = 2^{30}$) We analyze caches with an **eight**-word capacity (C) for the sake of simplicity. We begin with a **one**-word block size (b), then generalize later to larger blocks.

Direct Mapped Cache

A *direct mapped* cache has **one** block in **each** set ($N = 1$), so it is organized into $S = B$ sets. To **understand** the **mapping** of memory addresses onto cache blocks, imagine main memory as

being mapped into b-word blocks, **just** as the cache is. An address in block 0 of main memory maps to set 0 of the cache. An address in block 1 of main memory maps to set 1 of the cache, and so forth until an address in block $B-1$ of main memory maps to block $B-1$ of the cache. There are **no more** blocks of the cache, so the mapping **wraps** around, such that block B of main memory maps to block 0 of the cache.

This mapping is illustrated in Figure 8.5 for a direct mapped cache with a capacity of **eight** words and a block size of **one** word. The cache has **eight** sets, each of which contains a one-word block. The bottom **two bits** of the address are always 00, because they are **word aligned**. The **next** $\log_2 8 = 3$ bits indicate the **set** onto which the memory address maps. Thus, the data at addresses 0x00000004, 0x00000024, . . ., 0xFFFFFFE4 all map to set 1, as shown in blue. Likewise, data at addresses 0x00000010, . . . , 0xFFFFFFF0 all map to set 4, and so forth. Each main memory address maps to **exactly one set** in the cache. (Note here $b = 1, N = 1$, so $B = C/1, S = B/1$)

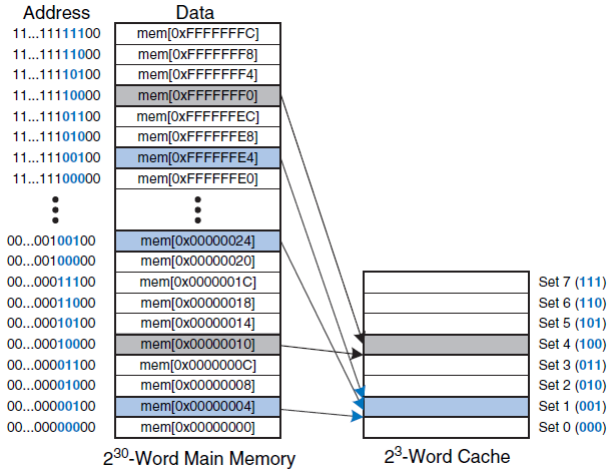


Figure 8. 5 Mapping of main memory to a direct mapped cache

Because many addresses map to a **single** set, the cache must also **keep track** of the **address** of the data actually contained in each set. The least significant bits (here is last 3 bits from the aligned bits) of the address specify which set holds the data. The **remaining most significant** bits (here is first 27 bits) are called the ***tag*** and indicate which of the many possible addresses is held in that set.

In our previous examples, the two least significant bits of the 32-bit address are called the ***byte offset***, because they **indicate the byte within the word**. The **next** three bits are called the ***set bits***, because they indicate the set to which the address maps. (In general, the number of set bits is $\log_2 S$.) The **remaining** 27 tag bits indicate the **memory address** of the data stored in a given cache set. Figure 8.6 shows the cache fields for **address** 0xFFFFFFE4. It maps to set 1 and its tag is all 1's.

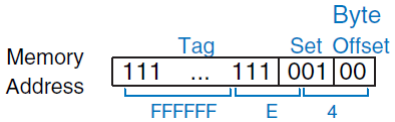


Figure 8. 6 **Cache fields** for address 0xFFFFFFE4 when mapping to the cache in Figure 8.5

Example 8.5: CACHE FIELDS (Page 496)

Sometimes, such as when the computer first starts up, the cache sets contain **no** data at all. The cache uses a ***valid bit*** for each set to indicate **whether the set holds meaningful data**. If the valid bit is 0, the contents are meaningless.

Figure 8.7 shows the hardware for the direct mapped cache of Figure 8.5. The cache is constructed as an eight-entry **SRAM**. Each entry, or **set**, contains **one line** consisting of 32 bits of data, 27 bits of tag, and 1 valid bit. The cache is **accessed** using the 32-bit address. The two least significant bits, the byte offset bits, are **ignored** for word accesses. The next three bits, the set bits, specify the entry or set in the cache. A **load** instruction reads the specified entry

from the cache and **checks** the tag and valid bits. If the tag matches the most significant 27 bits of the address and the valid bit is 1, the cache hits and the data is returned to the processor. Otherwise, the cache misses and the memory system must fetch the data from main memory.

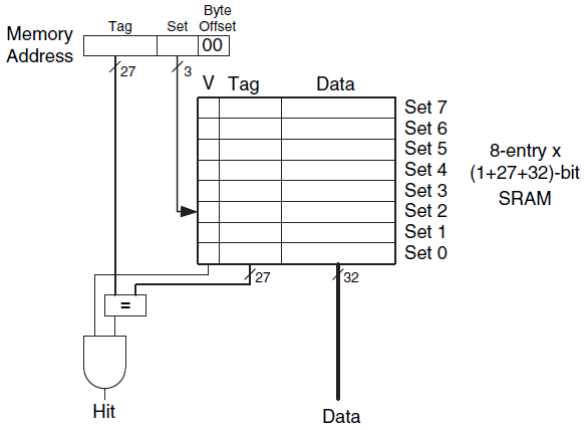


Figure 8. 7 Direct mapped cache with 8 sets

Example 8.6: TEMPORAL LOCALITY WITH A DIRECT MAPPED CACHE (Page 497)

When **two recently** accessed addresses map to the **same** cache block, a ***conflict*** occurs, and the most recently accessed address ***evicts*** the previous one from the block. Direct mapped caches have only one block in each set, so two addresses that map to the same set **always** cause a conflict.

Example 8.7: CACHE BLOCK CONFLICT (Page 498)

Multi-way Set Associative Cache

An ***N-way set associative*** cache **reduces conflicts** by providing N blocks in **each** set where data mapping to that set might be found. Each memory address **still** maps to a specific set, but it can map to any one of the N blocks in **the** set. Hence, a direct mapped cache is another name for a one-way set associative cache. N is also called the ***degree of associativity*** of the cache.

Figure 8.8 shows the hardware for a $C = 8$-word, $N = 2$-way set associative cache. The cache now has only $S = 4$ sets rather than 8. Thus, only $\log_2 4 = 2$ set bits rather than 3 are used to select the set. The tag increases from 27 to 28 bits. Each set contains two *ways* or degrees of associativity. Each way consists of a data block and the valid and tag bits. The cache reads blocks from **both ways** in the selected set and checks the tags and valid bits for a hit. If a hit occurs in one of the ways, a **multiplexer** selects data from that way. (Note here $b = 1$, so $B = C/1$)

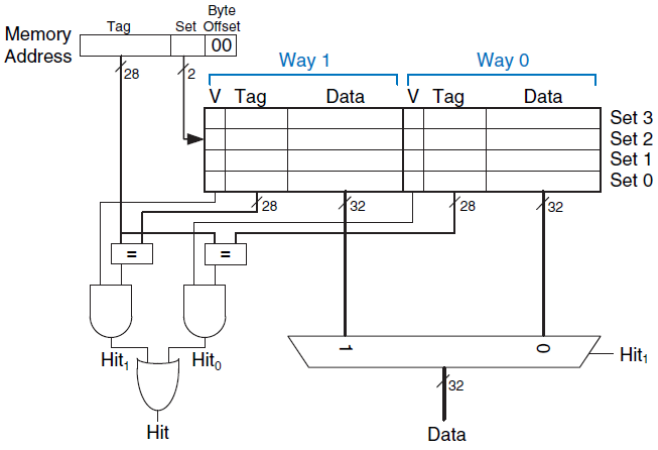


Figure 8. 8 Two-way set associative cache

Set associative caches **generally** have **lower miss rates** than direct mapped caches of the same capacity, because they have fewer conflicts. However, set associative caches are usually **slower** and somewhat more **expensive** to build because of the output multiplexer and additional

comparators. They also raise the question of **which way** to replace when both ways are **full**; this is addressed further in Section 8.3.3. Most commercial systems use set associative caches.

Example 8.8: SET ASSOCIATIVE CACHE MISS RATE (Page 499)

Fully Associative Cache

A ***fully associative*** cache contains a single set with B ways, where B is the number of blocks. A memory address can map to a block in any of these ways. A fully associative cache is another name for a B-way set associative cache with one set.

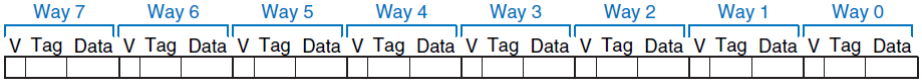


Figure 8. 9 Eight-block fully associative cache

Figure 8.11 shows the SRAM array of a fully associative cache with eight blocks. Upon a data request, eight tag **comparisons** (not shown) **must be made**, because the data could be in any block. Similarly, an $8:1$ **multiplexer** chooses the proper data if a hit occurs. Fully associative caches tend to have the **fewest conflict misses** for a given cache capacity, but they require **more** hardware for additional tag comparisons. They are **best** suited to relatively small caches because of the large number of comparators.

Block Size

The previous examples were able to take advantage **only** of **temporal locality**, because the block size was **one word**. To exploit **spatial locality**, a cache uses larger blocks to hold several consecutive words (b).

The advantage of a block size greater than one is that when a miss occurs and the word is fetched into the cache, the **adjacent words** in the block are **also** fetched. Therefore, subsequent accesses are more likely to hit because of spatial locality. However, a large block size means that a fixed-size cache will have fewer blocks. This may lead to **more** conflicts, increasing the miss rate. Moreover, it takes **more time** to fetch the missing cache block after a miss, because **more than** one data word is fetched from main memory. The time required to load the missing block into the cache is called the ***miss penalty***. If the adjacent words in the block are **not** accessed later, the effort of fetching them is **wasted**. Nevertheless, most real programs **benefit** from larger block sizes.

Figure 8.10 shows the hardware for a $C = 8$-word direct mapped cache with a $b = 4$-word block size. The cache now has only $B = C/b = 2$ blocks. A direct mapped cache has one block in each set, so this cache is organized as two sets. Thus, only $\log_2 2 = 1$ bit is used to select the set. A multiplexer is **now needed** to select the word within the block. The multiplexer is controlled by the $\log_2 4 = 2$ ***block offset*** *bits* of the **address**. The most significant 27 address bits form the tag. Only **one** tag is needed for the entire block, because the words in the block are at consecutive addresses. (**Note**: if more than one degree of associativity is used, one **more** layer of multiplexer is need.)

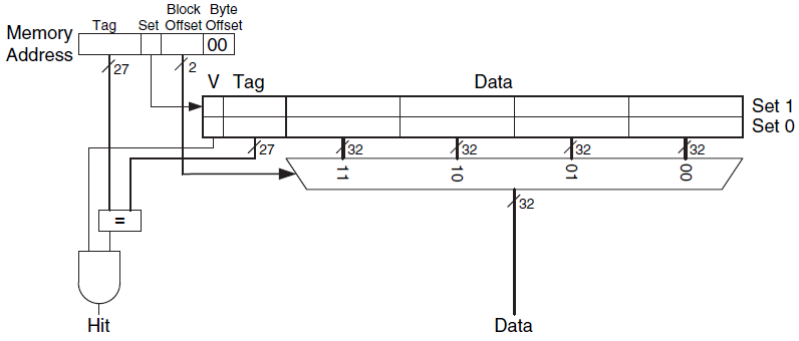


Figure 8. 10 Direct mapped cache with two sets and a four-word block size

Figure 8.11 shows the cache fields for address 0x8000009C when it maps to the direct mapped cache of Figure 8.10. The byte offset bits are always 0 for word accesses. The next $\log_2 b = 2$ block offset bits indicate the word within the block. And the next bit indicates the set. The

remaining 27 bits are the tag. Therefore, word 0x8000009C maps to set 1, word 3 in the cache. The principle of using larger block sizes to exploit spatial locality also applies to associative caches.

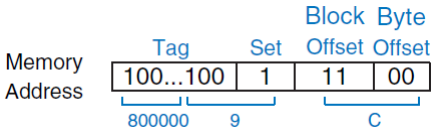


Figure 8. 11 Cache fields for address 0x8000009C when mapping to the cache of Figure 8.11

Example 8.9: SPATIAL LOCALITY WITH A DIRECT MAPPED CACHE (Page 501)

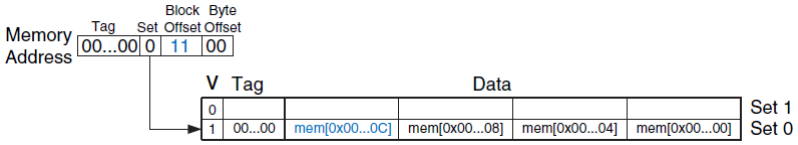


Figure 8. 12 Cache contents with a block size b of four words

Putting it All Together

Caches are organized as **two-dimensional** arrays. The **rows** are called **sets**, and the **columns** are called **ways**. Each **entry** in the array consists of a data block and its associated valid and tag bits. Caches are characterized by

- capacity C
- block size b (and number of blocks, $B = C/b$)
- number of blocks in a set (N)

Figure 8.13 summarizes the various cache organizations. Each address in memory maps to only one set **but** can be stored in **any** of the ways.

Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapped	1	B
Set Associative	$1 < N < B$	B/N
Fully Associative	B	1

Figure 8. 13 Cache organizations

Cache capacity, associativity, set size, and block size are **typically** powers of 2. This makes the cache fields (tag, set, and block offset bits) **subsets** of the address bits.

Increasing the associativity N **usually** reduces the miss rate caused by conflicts. **But** higher associativity requires more tag comparators. Increasing the block size b takes **advantage** of spatial locality to reduce the miss rate. However, it decreases the number of sets in a fixed sized cache and therefore could lead to more conflicts. It also increases the miss penalty.

What Data is Replaced?

In a direct mapped cache, each address maps to a unique block and set. If a set is full when new data must be loaded, the block in that set is replaced with the new data. In **set associative and fully associative** caches, the cache must choose which block to evict when a cache set is full. The principle of temporal locality suggests that the best choice is to evict the least recently used block, because it is least likely to be used again soon. Hence, **most** associative caches have a ***least recently used*** **(*LRU*)** replacement **policy**. (Note, only for set associative and fully associative caches)

In a two-way set associative cache, a ***use bit***, U, indicates which way within **a set** was least recently used. For set associative caches with **more than two** ways, tracking the least recently used way becomes **complicated**. To simplify the problem, the ways are often **divided** into **two groups** and U indicates which *group* **of ways** was least recently used. Upon replacement, the **new** block replaces a **random** block **within** the least recently used group. Such a policy is called ***pseudo-LRU*** and is good enough in practice.

Example 8.10: LRU REPLACEMENT (Page 503)

Advanced Cache Design

Modern systems use **multiple** levels of caches to **decrease** memory access time. This section explores the performance of a two-level caching system and examines how block size, associativity, and cache capacity affect miss rate. The section also describes how caches handle stores, or writes, by using a write-through or write-back policy.

Multiple-Level Caches

Large caches are beneficial because they are more likely to hold data of interest and therefore have lower miss rates. **However**, **large** caches tend to be **slower** than small ones. Modern systems often use **at least two levels** of caches, as shown in Figure 8.14. The first-level (L1) cache is small enough to provide a one- or two-cycle access time. The second-level (L2) cache is also built from SRAM but is larger, and therefore slower, than the L1 cache. The processor first looks for the data in the L1 cache. If the L1 cache misses, the processor looks in the L2 cache. If the L2 cache misses, the processor fetches the data from main memory. Many modern systems add even more levels of cache to the memory hierarchy, because accessing main memory is so slow.

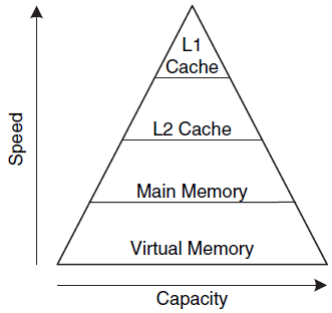


Figure 8. 14 Memory hierarchy with two levels of cache

Example 8.11: SYSTEM WITH AN L2 CACHE (Page 504)

Reducing Miss Rate

Cache misses can be reduced by changing **capacity**, **block size**, and/or **associativity**. The **first step** to reducing the miss rate is to **understand** the **causes** of the misses. The **misses** can be classified as **compulsory, capacity, and conflict**. The first request to a cache block is called a *compulsory miss*, because the block must be read from memory regardless of the cache design. *Capacity misses* occur when the cache is too small to hold all concurrently used data. *Conflict misses* are caused when several addresses map to the same set and evict blocks that are still needed.

Changing cache parameters can affect one or more type of cache miss. For example, increasing cache **capacity** can reduce conflict and capacity misses, but it does not affect compulsory misses. On the other hand, increasing **block size** could reduce compulsory misses (due to spatial locality) **but** might actually *increase* conflict misses (because more addresses would map to the same set and could conflict).

Memory systems are **complicated enough** that the best way to evaluate their performance is by running **benchmarks** while **varying** cache parameters. Figure 8.15 plots miss rate versus cache size and degree of associativity for the **SPEC2000** benchmark. This benchmark has a small number of compulsory misses, shown by the dark region near the x-axis. As expected, when cache size increases, capacity misses decrease. Increased associativity, especially for small caches, decreases the number of conflict misses shown along the top of the curve. Increasing associativity beyond four or eight ways provides **only** small decreases in miss rate.

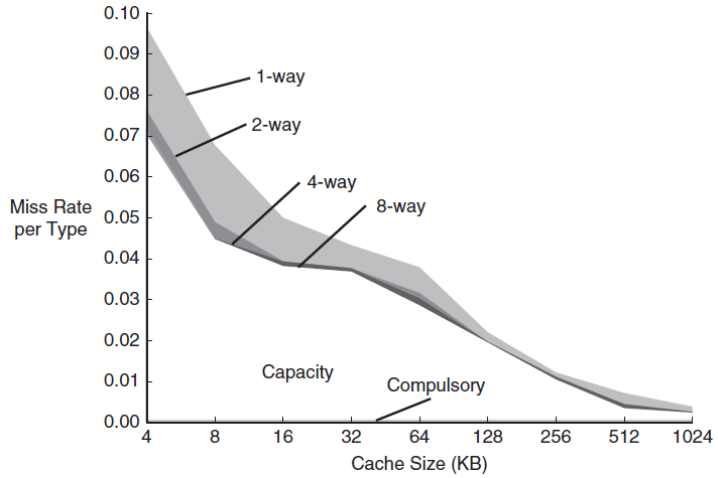


Figure 8. 15 Miss rate versus cache size and associativity on SPEC2000 benchmark

As mentioned, miss rate can also be decreased by using larger block sizes that take advantage of spatial locality. **But** as block size increases, the number of sets in a fixed-size cache decreases, increasing the probability of conflicts. Figure 8.16 plots miss rate versus block size (in number of bytes) for caches of varying capacity. For small caches, such as the 4-KB cache, increasing the block size beyond 64 bytes *increases* the miss rate because of conflicts. For larger caches, increasing the block size beyond 64 bytes does not change the miss rate. However, large block sizes might still increase execution time because of the **larger** miss penalty, the time required to fetch the missing cache block from main memory.

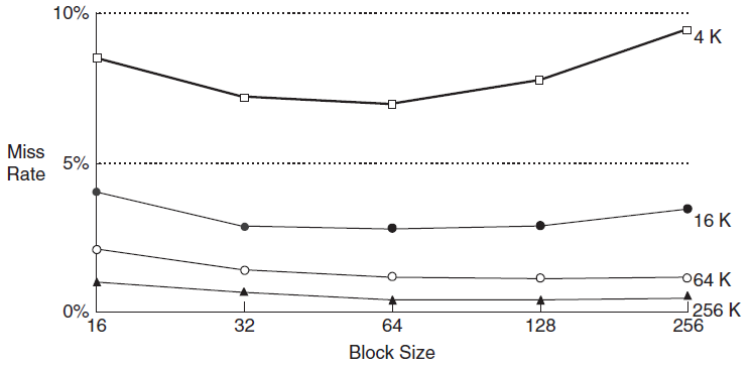


Figure 8. 16 Miss rate versus block size and cache size on SPEC92 benchmark

Write Policy

The previous sections focused on memory loads. Memory stores, or writes, follow a similar procedure as loads. Upon a memory store, the processor checks the cache. If the cache misses, the cache block is **fetched from** main memory into the cache, and then the appropriate word in the cache block is written. If the cache hits, the word is **simply written** to the cache block.

Caches are classified as **either** write-through or write-back. In a ***write-through*** cache, the data written to a cache block is **simultaneously** written to main memory. In a ***write-back*** cache, a ***dirty bit*** (D) is associated with **each cache block**. D is 1 when the cache block has been written and 0 otherwise. Dirty cache blocks are written back to main memory **only** when they are **evicted** from the cache. A write-through cache requires no dirty bit but **usually** requires more main memory writes than a write-back cache. Modern caches are **usually** write-back, because main memory access time is so large.

Example 8.12: WRITE-THROUGH VERSUS WRITE-BACK (Page 507)

The Evolution of ARM Caches

Supplement materials

- Write Allocate and Write No Allocate Cache (need update)

● Cache Prefetching
Prefetching is an attempt to fetch lines from memory into the cache before their instructions are required by the execution unit.

The goal of cache prefetching is to reduce cache misses.

The behavior of cache prefetching:
1. Guess which cache lines will soon be needed by the fetch unit;
2. Initiate the prefetch requests long before the instructions are needed (miss latencies can be reduced or eliminated completely);

● Instruction Prefetching
The goal of instruction prefetching is to assist in instruction decode or to increase the instruction issue rate.

● Data Prefetching
The goal of data prefetching is to reduce data cache misses by exploiting a program's data access patterns in order to prefetch data from memory.

● Non-blocking Caches (need update, refer to "Computer architecture a quantitative approach")
Non-blocking cache (lockup-free cache) allows the CPU to continue executing instructions while a miss is handled.

Non-blocking cache has the miss status holding registers (MSHR), which are the hardware structure for tracking outstanding misses.

8.4 VIRTUAL MEMORY

Most modern computer systems use a *hard drive* made of magnetic or solid state storage as the lowest level in the memory hierarchy. Compared with the ideal large, fast, cheap memory, a hard drive is large and cheap but terribly slow. It provides a much larger capacity than is possible with a **cost-effective** main memory (DRAM). However, if a significant fraction of memory accesses involve the hard drive, performance is dismal. You may have encountered this on a PC when running **too many programs** at once.
The **objective** of adding a hard drive to the memory hierarchy is to **inexpensively** give the illusion of a very large memory while still providing the speed of faster memory for most accesses. A computer with only 128 MB of DRAM, for example, could effectively provide 2 GB of memory using the hard drive. This larger 2-GB memory is called ***virtual memory***, and the smaller 128-MB main memory is called ***physical memory***. We will use the term physical memory to **refer to** main memory throughout this section. (2-GB = 128-MB + 1920-MB)
Programs can access data anywhere in virtual memory, so they **must** use ***virtual addresses*** that specify the location in virtual memory. The physical memory holds a subset of **most recently** accessed virtual memory. In this way, physical memory acts **as a cache** for virtual memory. Thus, most accesses hit in physical memory at the speed of DRAM, yet the program enjoys the capacity of the larger virtual memory.
Virtual memory systems use different **terminologies** for the **same caching principles** discussed in Section 8.3. Figure 8.17 summarizes the analogous terms. **Virtual memory** is divided into ***virtual pages***, typically 4 KB in size. **Physical memory** is likewise divided into ***physical pages*** of the **same size**. A virtual page may **be located in** physical memory (DRAM) or on the hard drive. Figure 8.18 shows a virtual memory that is larger than physical memory. The **rectangles** indicate **pages**. Some virtual pages are present in physical memory, and some

are located on the hard drive. The **process of determining** the physical address from the virtual address is called ***address translation***. If the processor attempts to access a virtual address that is **not** in physical memory, a ***page fault*** occurs, **and** the **operating system** loads the page from the hard drive into physical memory.

Cache	Virtual Memory
Block	Page
Block size	Page size
Block offset	Page offset
Miss	Page fault
Tag	Virtual page number

Figure 8. 17 Analogous cache and virtual memory terms

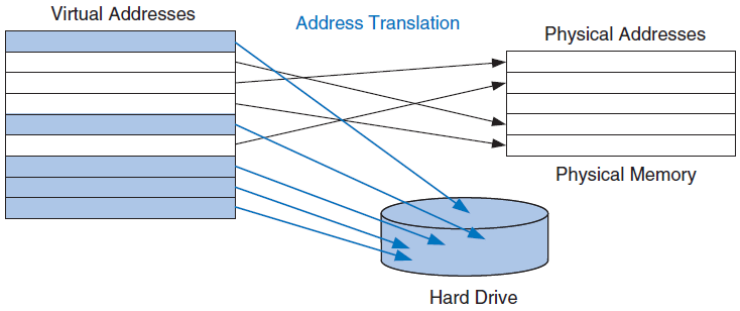


Figure 8. 18 Virtual and physical pages

To **avoid** page faults caused by conflicts, **any** virtual page can map to **any** physical page. In other words, physical memory behaves as a **fully associative cache** for virtual memory. In a conventional fully associative cache, **every** cache block has a comparator that checks the most significant address bits against a tag to determine whether the request hits in the block. In an **analogous** virtual memory system, **each** physical page would need a comparator to check the most significant virtual address bits against a tag to determine whether the virtual page maps to that physical page.

A **realistic virtual memory system** has so many physical pages that providing a comparator for each page would be excessively expensive. **Instead**, the virtual memory system uses **a page table** to perform address translation. A page table contains **an entry** for each virtual page, indicating its **location** in physical memory **or** that it is on the hard drive. Each load or store instruction requires a page table access **followed** by a physical memory access. The page table access translates the virtual address used by the program to a physical address. The physical address is then used to actually read or write the data.

The page table is usually so large that it is **located** in physical memory. Hence, each load or store involves **two** physical memory accesses: a page table access, and a data access. To **speed** up address translation, **a translation lookaside buffer (TLB)** caches the most commonly used page table entries.

Address Translation

In a system with virtual memory, **programs** use virtual addresses so that they can access a **large** memory. The **computer must** translate these virtual addresses to either find the address in physical memory or take a page fault and fetch the data from the hard drive.

Recall that virtual memory and physical memory are **divided** into pages. The most significant bits of the virtual or physical address specify the virtual or physical ***page number***. The least significant bits specify the word within the page and are called the ***page offset***.

Figure 8.19 illustrates the page organization of a virtual memory system with 2 GB of virtual memory and 128 MB of physical memory divided into 4-KB pages. MIPS accommodates 32-bit addresses. With a 2-GB= 2^{31}-byte virtual memory, only the least significant 31 virtual address bits are used; the 32nd bit is always 0. Similarly, with a 128-MB= 2^{27}-byte physical

memory, only the least significant 27 physical address bits are used; the upper 5 bits are always 0.

Because the page size is 4 KB = 2^{12} bytes, there are $2^{31}/2^{12} = 2^{19}$ virtual pages and $2^{27}/2^{12} = 2^{15}$ physical pages. Thus, the virtual and physical page numbers are 19 and 15 bits, respectively. Physical memory can only hold up to 1/16th ($19/15 = 4$ bits->$2^4 = 16$ choices) of the virtual pages at any given time. The rest of the virtual pages are kept on the hard drive.

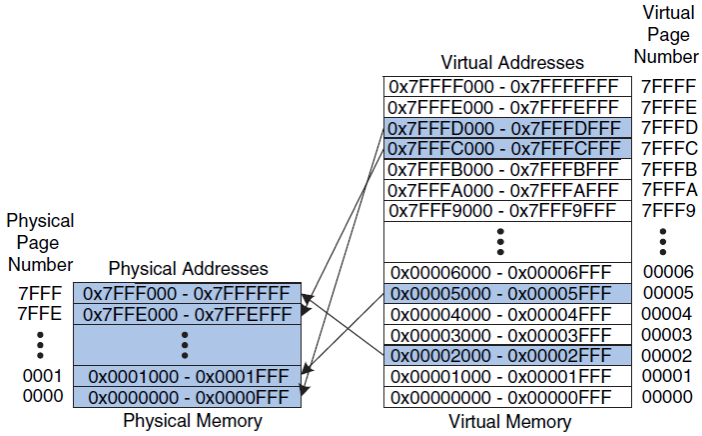


Figure 8. 19 Physical and virtual pages

Figure 8.19 shows virtual page 5 mapping to physical page 1, virtual page 0x7FFFC mapping to physical page 0x7FFE, and so forth. For example, virtual address 0x53F8 (an offset of 0x3F8 within virtual page 5) maps to physical address 0x13F8 (an offset of 0x3F8 within physical page 1). The least significant 12 bits of the virtual and physical addresses are the same (0x3F8) and specify the page offset within the virtual and physical pages. Only the page number needs to be translated to obtain the physical address from the virtual address.

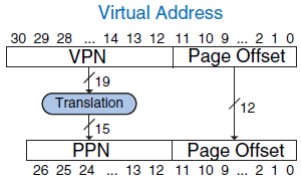


Figure 8. 20 Translation from virtual address to physical address

Figure 8.20 illustrates the translation of a virtual address to a physical address. The least significant 12 bits indicate the page offset and require no translation. The upper 19 bits of the virtual address specify the ***virtual page number*** **(*VPN*)** and are translated to a 15-bit ***physical page number*** **(*PPN*)**. The next two sections describe how page tables and TLBs are used to perform this address translation.

Example 8.13: VIRTUAL ADDRESS TO PHYSICAL ADDRESS TRANSLATION (Page 512)

The Page Table

The processor uses a *page table* to translate virtual addresses to physical addresses. The page table contains an entry for each virtual page. This entry **contains** a physical page number and a valid bit. If the valid bit is 1, the virtual page maps to the physical page specified in the entry. Otherwise, the virtual page is found on the hard drive.

Because the page table is so large, it is stored in physical memory. Let us assume for now that it is stored as a contiguous array, as shown in Figure 8.21. This page table contains the mapping of the memory system of Figure 8.19. The page table is **indexed** with the virtual page number (VPN). For example, entry 5 specifies that virtual page 5 maps to physical page 1. Entry 6 is invalid ($V = 0$), **so** virtual page 6 is located on the hard drive.

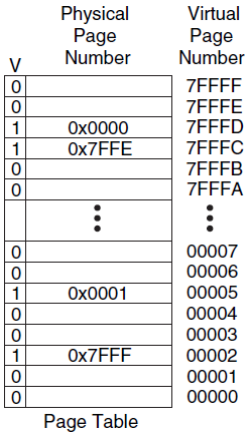


Figure 8. 21 The page table for Figure 8.19

Example 8.14: USING THE PAGE TABLE TO PERFORM ADDRESS TRANSLATION (Page 512)

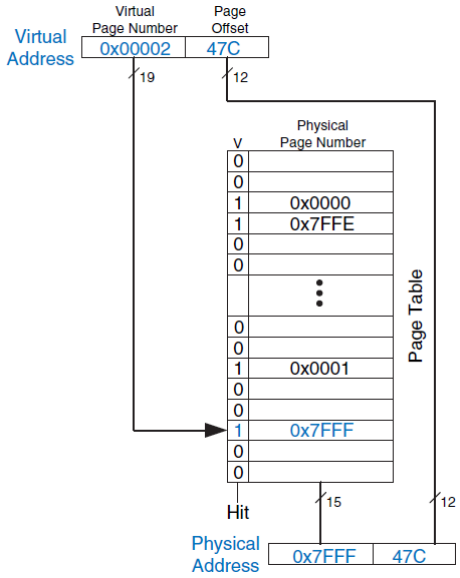


Figure 8. 22 Address translation using the page table

The page table can be stored **anywhere** in physical memory, at the discretion of the OS. The processor typically uses a **dedicated register**, called the ***page table register***, to store the **base address** of the page table in physical memory.

To perform a load or store, the processor must first translate the virtual address to a physical address and then access the data at that physical address. The processor extracts the virtual page number from the virtual address and **adds** it to the page table register to find the physical address of the page table entry. The processor then reads this page table entry from physical memory to obtain the physical page number. If the entry is valid, it merges this physical page number with the page offset to create the physical address. Finally, it reads or writes data at this physical address. **Because** the page table is stored in physical memory, each load or store involves **two** physical memory **accesses**.

The Translation Lookaside Buffer

Virtual memory would have a **severe** performance impact if it required a page table read on **every** load or store, doubling the delay of loads and stores. Fortunately, page table accesses have **great** temporal locality. The **temporal** and **spatial locality** of data accesses and the large page size mean that many consecutive loads or stores are likely to reference the same page. Therefore, if the processor **remembers** the last page table entry that it read, it can probably reuse this translation **without** rereading the page table. In general, the processor can keep the **last several page table entries** in a small **cache** called a ***translation lookaside buffer*** **(*TLB*)**. The processor "looks aside" to find the translation in the TLB **before** having to access the page

table in physical memory. In real programs, the vast majority of accesses hit in the TLB, **avoiding** the time-consuming page table reads from physical memory.

A TLB is organized as a **fully associative** cache and **typically** holds 16 to 512 entries. **Each** TLB entry **holds** a virtual page number and its corresponding physical page number. The TLB is accessed using the virtual page number. If the TLB hits, it returns the corresponding physical page number. Otherwise, the processor **must** read the page table in physical memory. The TLB is designed to be small enough that it can be accessed in **less** than one cycle. Even so, TLBs typically have a hit rate of **greater** than 99%. The TLB decreases the number of memory accesses required for most load or store instructions from two to **one**.

Example 8.15: USING THE TLB TO PERFORM ADDRESS TRANSLATION (Page 514)

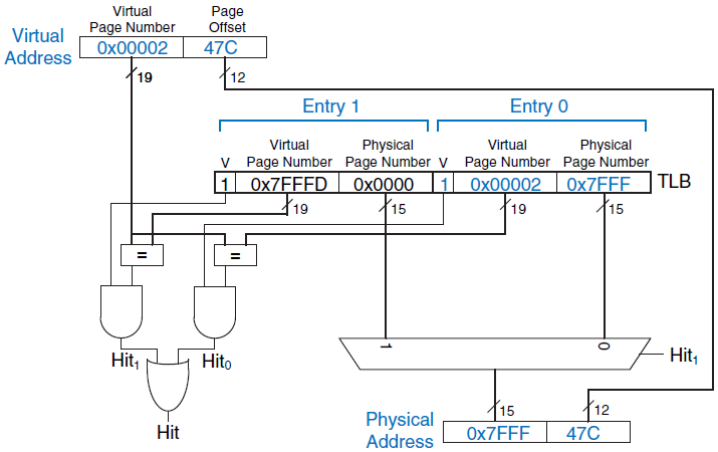


Figure 8. 23 Address translation using a two-entry TLB

Memory Protection

So far, this section has focused on using virtual memory to provide a fast, inexpensive, large memory. An equally **important** reason to use virtual memory is to provide protection between **concurrently running programs**.

As you probably know, modern computers **typically** run several programs or ***processes*** at the same time. All of the programs are simultaneously present in physical memory. In a well-designed computer system, the programs should be protected from each other so that no program can **crash** or **hijack** another program. Specifically, no program should be able to access another program's memory without **permission**. This is called ***memory protection***.

Virtual memory systems provide memory protection by **giving each** program its **own *virtual address space***. Each program can use as much memory as it wants in that virtual address space, but **only a portion** of the virtual address space is in physical memory at any given time. Each program can use **its entire** virtual address space without having to worry about where other programs are physically located. However, a program can access only those physical pages that are mapped in **its** page table. In this way, a program **cannot** accidentally or maliciously access another program's physical pages, because they are **not** mapped in **its** page table. In some cases, multiple programs access **common instructions or data**. The **operating system** adds control bits to each page table entry to deter- mine which programs, if any, can write to the shared physical pages.

Replacement Policies

Virtual memory systems use write-back and an **approximate least recently used (LRU)** replacement **policy**. A write-through policy, where each write to physical memory initiates a write to the hard drive, would be **impractical**. Store instructions would operate at the speed of the hard drive instead of the speed of the processor (milliseconds instead of nanoseconds). Under the writeback policy, the physical page is written back to the hard drive **only** when it is **evicted** from physical memory. **Writing** the physical page back to the hard drive and **reloading** it with a different virtual page is called ***paging***, and the hard drive in a virtual memory system

is sometimes called ***swap space***. The processor pages out one of the least recently used physical pages when a **page fault** occurs, then **replaces** that page with the **missing** virtual page. To support these replacement policies, **each** page table **entry** contains **two** additional status bits: a dirty bit D and a use bit U. (Note here the differences with cache)
The dirty bit is 1 if **any** store instructions have **changed** the physical page since it was read from the hard drive. When a physical page is paged out, it **needs** to be written back to the hard drive **only** if its dirty bit is 1; otherwise, the hard drive **already** holds an exact copy of the page. The use bit is 1 if the physical page has been **accessed recently**. As in a cache system, exact LRU replacement would be impractically complicated. Instead, the OS **approximates** LRU replacement by **periodically** resetting all the use bits in the page table. When a page is accessed, its use bit is set to 1. Upon a page fault, the OS **finds** a page with $U = 0$ to page out of physical memory. Thus, it **does not** necessarily replace the least recently used page, just **one** of the least recently used page**s**. (Note here the differences with cache)

Multilevel Page Tables
Page tables can occupy a large amount of physical memory. For example, the page table from the previous sections for a 2 GB virtual memory with 4 KB pages would need 2^{19} entries. If each entry is 4 bytes, the page table is $2^{19} \times 2^2$ bytes $= 2^{21}$ bytes $= 2$ MB.
To **conserve** physical memory, page tables can be broken up into multiple (usually two) levels. The first-level page table is always kept in physical memory. It indicates where small second-level page tables are stored in virtual memory. The second-level page tables each contain the actual translations for a range of virtual pages. If a particular range of translations is **not** actively used, the corresponding second-level page table can be paged **out** to the hard drive so it **does not** waste physical memory.
In a two-level page table, the virtual page number is split into two parts: the ***page table number*** and the ***page table offset***, as shown in Figure 8.24. The page table number indexes the first-level page table, which **must** reside in physical memory. The first-level page table entry gives the base address of the second-level page table **or** indicates that it must be fetched from the hard drive when V is 0. The page table offset indexes the second-level page table. The remaining 12 bits of the virtual address are the page offset, as before, for a page size of $2^{12} = 4$ KB.
In Figure 8.24 the 19-bit virtual page number is broken into 9 and 10 bits, to indicate the page table number and the page table offset, respectively. Thus, the first-level page table has $2^9 = 512$ entries. Each of these 512 second-level page tables has $2^{10} = 1$ K entries. If each of the first- and second-level page table entries is 32 bits (4 bytes) and only two second-level page tables are present in physical memory at once, the hierarchical page table uses only (512×4 bytes) $+2 \times$ (1 K $\times 4$ bytes) $=$ 10 KB of physical memory. The two-level page table requires a **fraction** of the physical memory needed to store the entire page table (2 MB). The drawback of a two-level page table is that it adds yet another memory access for translation when the TLB misses.

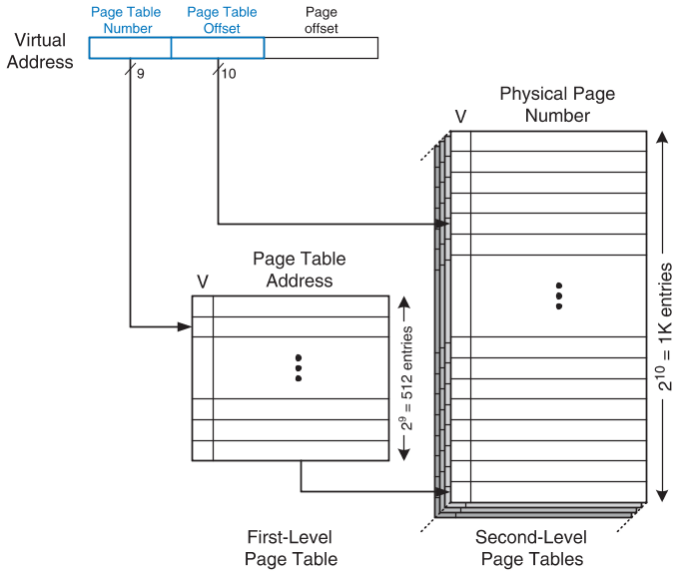


Figure 8. 24 Hierarchical page tables

Example 8.16: USING A MULTILEVEL PAGE TABLE FOR ADDRESS TRANSLATION (Page 517)

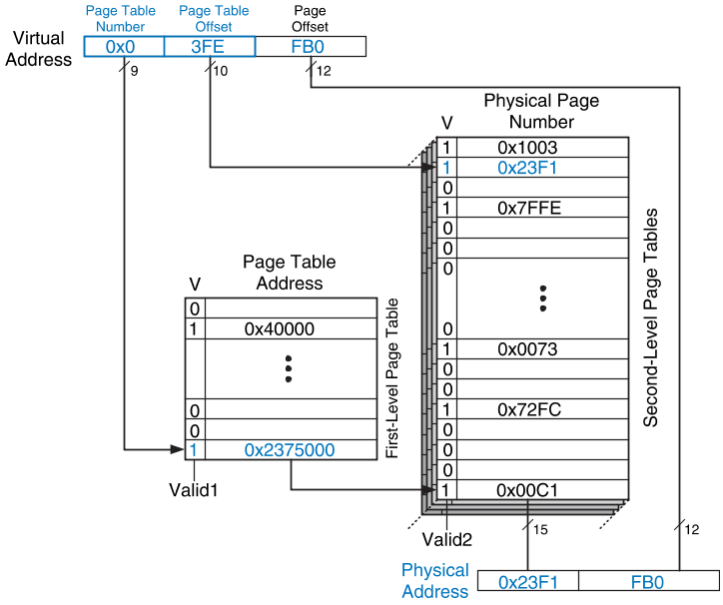


Figure 8. 25 Address translation using a two-level page table

8.5 SUMMARY

Memory system organization is a major factor in determining computer performance. Different memory technologies, such as DRAM, SRAM, and hard drives, offer trade-offs in capacity, speed, and cost. This chapter introduced cache and virtual memory organizations that use a hierarchy of memories to approximate an ideal large, fast, inexpensive memory. Main memory is typically built from DRAM, which is significantly slower than the processor. A cache reduces access time by keeping commonly used data in fast SRAM. Virtual memory increases the memory capacity by using a hard drive to store data that does not fit in the main memory. Caches and virtual memory add complexity and hardware to a computer system, but the benefits usually outweigh the costs. All modern personal computers use caches and virtual memory.

8.6 Supplement materials

Non-volatile Memories

Semiconductor memories can be divided into two major categories: RAM (*Random Access Memories*) and ROM (*Read Only Memories*): RAMs lose their content when power supply is switched off, while ROMs virtually hold it forever. A third category lies in between, i.e. NVM

(*Non-Volatile Memories*), whose content can be electrically altered but it is also preserved when the power supply is switched off. NVMs are more flexible than the original ROM, whose content is defined during manufacturing and cannot be changed by the user anymore.

NAND Flash cell is based on the Floating Gate (FG) technology, whose cross section is shown in Figure 8.26. A MOS transistor is built with two overlapping gates rather than a single one: the first one is completely surrounded by oxide, while the second one is contacted to form the gate terminal. The isolated gate constitutes an excellent trap for electrons, which guarantees charge retention for years. **The operations performed to inject and remove electrons from the isolated gate are called** program and erase**, respectively**. These operations modify the threshold voltage V_{TH} of the memory cell, which is **a special type of MOS transistor**. Applying a fixed voltage to cell's terminals, it is then possible to discriminate two storage levels: when the gate voltage is higher than the cell's V_{TH}, the cell is on ("1"), otherwise it is off ("0").

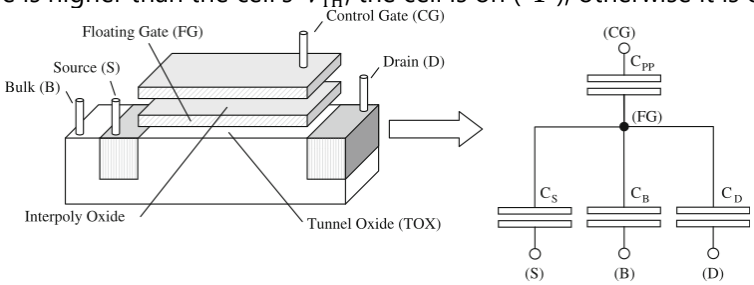


Figure 8.26 Schematic representation of a floating gate memory cell (*left*) and the corresponding capacitive model (*right*)

NAND flash-based Solid-state drive (SSD)

NAND flash memory is the de facto standard for architecting a storage device in modern computing systems. As modern computing systems process a large amount of data at an unprecedented scale, ***a storage device needs to meet high requirements on storage capacity and I/O performance***. A NAND flash-based solid-state drive (SSD) can provide orders-of-magnitude higher I/O performance compared to traditional hard-disk drives (HDDs), with a much lower cost-per bit value over ***SSDs based on emerging non-volatile memory (NVM) technologies***.

NAND flash memory has several unique characteristics, such as the erase-before write property (i.e., a flash cell needs to be first erased before programming it, why?), limited lifetime (i.e., a cell cannot reliably store data after experiencing a certain number of ***program/erase (P/E)*** cycles), and large operation units (e.g., modern NAND flash memory typically reads/writes data in a page (e.g., 16 KiB) granularity). ***To achieve high performance and large capacity of the storage system while hiding the unique characteristics of NAND flash memory***,(target) it is critical to design efficient SSD firmware, commonly called Flash-Translation Layer (FTL). An FTL is responsible for many critical management tasks, such as ***address translation, garbage collection, wear leveling, and I/O scheduling***, which significantly affect the performance, reliability, and lifetime of the SSD.

CHAPTER A: Digital System Implementation

A.1 INTRODUCTION

A.2 74xx LOGIC

A.3 PROGRAMMABLE LOGIC

A.4 APPLICATION-SPECIFIC INTEGRATED CIRCUITS

Application-specific integrated circuits (*ASICs*) are chips designed for a particular purpose. Graphics accelerators, network interface chips, and cell phone chips are common examples of ASICs.

CHAPTER C: C Programming

C.1 INTRODUCTION

Language	Description
Matlab	Designed to facilitate heavy use of math functions
Perl	Designed for scripting
Python	Designed to emphasize code readability
Java	Designed to run securely on any computer
C	Designed for flexibility and overall system access, including device drivers
Assembly Language	Human-readable machine language
Machine Language	Binary representation of a program

Figure C. 1 Languages at roughly decreasing levels of abstraction

C.2 WELCOME TO C

A C program is a text file that describes operations for the computer to perform. C programs are generally contained in one or more text files that end in ".c". Good programming style requires a file name that indicates the contents of the program.

C Code Example eC.1: SIMPLE C PROGRAM (Page 541.e3)

```
// Write "Hello world!" to the console
#include <stdio.h>

int main(void){
  printf("Hello world!\n");
}
```

Console Output

```
Hello world!
```

● C Program Dissection

In general, a C program is organized into one or more functions. Every program must include the main function, which is where the program starts executing.

Header: #include <stdio.h>

The header includes the *library functions* needed by the program.

Main function: int main(void)

All C programs must include exactly one main function. Execution of the program occurs by running the code inside main, called the body of main. The *body* of a function contains a sequence of *statements*. Each statement ends with a semicolon. int denotes that the main function outputs, or *returns*, an integer result that indicates whether the program ran successfully.

Body: printf("Hello world!¥n");

The body of this main function contains one statement, a call to the printf function.

● Running a C Program

Slightly different versions of the C compiler exist, including *cc* (C compiler), or *gcc* (GNU C compiler).

C.3 COMPILATION

● Comments

Programmers use comments to describe code at a high-level and clarify code function. C programs use two types of comments: Single-line comments begin with // and terminate at the end of the line; multiple-line comments begin with /* and end with */.

A comment at the top of each C file is useful to describe the file's author, creation and modification dates, and purpose.

● #define

Constants are named using the #define directive and then used by name throughout the program. These globally defined constants are also called *macros*. For example, suppose you

write a program that allows at most 5 user guesses, you can use #define to identify that number.

#define MAXGUESSES 5

The # indicates that this line in the program will be handled by the *preprocessor*. By convention, #define lines are located at the top of the file and identifiers are written in all capital letters.

C Code Example eC.2: USING #define TO DECLARE CONSTANTS (Page 541.e6)

```
// Convert inches to centimeters
#include <stdio.h>

#define INCH2CM 2.54

int main(void) {
  float inch = 5.5;     // 5.5 inches
  float cm;

  cm = inch * INCH2CM;
  printf("%f inches = %f cm\n", inch, cm);
}
```

Console Output

```
5.500000 inches = 13.970000 cm
```

● #include

Variable declarations, defined values, and function definitions located in a *header file* can be used by another file by adding the #include preprocesser directive.

The ".h" postfix of the include file indicates it is a header file. While #include directives can be placed anywhere in the file before the included functions, variables, or identifiers are needed, they are conventionally located at the top of a C file.

Programmer-created header files can also be included by using quotation marks (" ") around the file name instead of brackets (< >). For example:

#include "myfunctions.h"

At compile time, files specified in brackets are searched for in system directories. Files specified in quotes are searched for in the same local directory where the C file is found. If the user-created header file is located in a different directory, the path of the file relative to the current directory must be included, #include "other/myFuncs.h".

C.4 VARIABLES

Variables in C programs have a type, name, value, and memory location. A variable declaration states the type and name of the variable. For example:

char x;

C views memory as a group of consecutive bytes, where each byte of memory is assigned a unique number indicating its location or *address*, as shown in Figure C.2. A variable occupies one or more bytes of memory, and the address of multiple-byte variables is indicated by the lowest numbered byte.

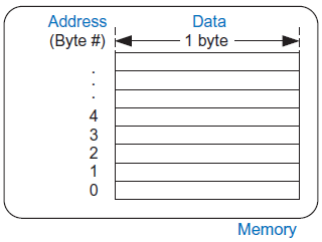


Figure C. 2 C's view of memory

Variable names are case sensitive and can be of your choosing. However, the name may not be any of C's reserved words (i.e., int, while, etc.), may not start with a number (i.e., int 1x; is not a valid declaration), and may not include special characters such as \, *, ?, or -. Underscores (_) are allowed.

● Primitive Data Types

C has a number of primitive, or built-in, data types available. They can be broadly characterized

as integers, floating-point variables, and characters. Figure C.3 lists the size and range of each primitive type.

Type	Size (bits)	Minimum	Maximum
char	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
unsigned char	8	0	$2^8 - 1 = 255$
short	16	$-2^{15} = -32{,}768$	$2^{15} - 1 = 32{,}767$
unsigned short	16	0	$2^{16} - 1 = 65{,}535$
long	32	$-2^{31} = -2{,}147{,}483{,}648$	$2^{31} - 1 = 2{,}147{,}483{,}647$
unsigned long	32	0	$2^{32} - 1 = 4{,}294{,}967{,}295$
long long	64	-2^{63}	$2^{63} - 1$
unsigned long	64	0	$2^{64} - 1$
int	machine-dependent		
unsigned int	machine-dependent		
float	32	$\pm 2^{-126}$	$\pm 2^{127}$
double	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

Figure C. 3 Primitive data types and sizes

C Code Example eC.3: EXAMPLE DATA TYPES (Page 541.e9)

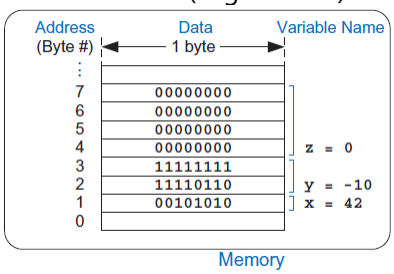


Figure C. 4 Variable storage in memory for C Code Example eC.3

● Global and Local Variables

A global variable is declared outside of all functions, typically at the top of a program, and can be accessed by all functions.

A local variable is declared inside a function and can only be used by that function.

The *scope* of a variable is the context in which it can be used.

C Code Example eC.4: GLOBAL VARIABLES (Page 541.e10)

C Code Example eC.5: LOCAL VARIABLES (Page 541.e10)

Using a local variable in C Code Example eC.5 is the preferred style because it preserves the well-defined interface of modularity.

● Initializing Variables

A variable needs to be *initialized* – assigned a value – before it is read. When a variable is declared, the correct number of bytes is reserved for that variable in memory. However, the memory at those locations retains whatever value it had last time it was used, essentially a random value. Global and local variables can be initialized either when they are declared or within the body of the program.

C.5 OPERATORS

The most common type of statement in a C program is an *expression*, such as

y = a + 3;

An expression involves operators (such as + or *) acting on one or more operands, such as variables or constants.

Category	Operator	Description	Example
Unary	++	post-increment	`a++; // a = a+1`
	--	post-decrement	`x--; // x = x-1`
	&	memory address of a variable	`x = &y; // x = the memory // address of y`
	~	bitwise NOT	`z = ~a;`
	!	Boolean NOT	`!x`
	-	negation	`y = -a;`
	++	pre-increment	`++a; // a = a+1`
	--	pre-decrement	`--x; // x = x-1`
	(type)	casts a variable to (type)	`x = (int)c; // cast c to an // int and assign it to x`
	sizeof()	size of a variable or type in bytes	`long int y; x = sizeof(y); // x = 4`
Multiplicative	*	multiplication	`y = x * 12;`
	/	division	`z = 9 / 3; // z = 3`
	%	modulo	`z = 5 % 2; // z = 1`
Additive	+	addition	`y = a + 2;`
	-	subtraction	`y = a - 2;`
Bitwise Shift	<<	bitshift left	`z = 5 << 2; // z = 0b00010100`
	>>	bitshift right	`x = 9 >> 3; // x = 0b00000001`
Relational	==	equals	`y == 2`
	!=	not equals	`x != 7`
	<	less than	`y < 12`
	>	greater than	`val > max`
	<=	less than or equal	`z <= 2`
	>=	greater than or equal	`y >= 10`

(*continued*)

Category	Operator	Description	Example
Bitwise	&	bitwise AND	`y = a & 15;`
	^	bitwise XOR	`y = 2 ^ 3;`
	|	bitwise OR	`y = a | b;`
Logical	&&	Boolean AND	`x && y`
	||	Boolean OR	`x || y`
Ternary	? :	ternary operator	`y = x ? a : b; // if x is TRUE, // y=a, else y=b`
Assignment	=	assignment	`x = 22;`
	+=	addition and assignment	`y += 3; // y = y + 3`
	-=	subtraction and assignment	`z -= 10; // z = z - 10`
	*=	multiplication and assignment	`x *= 4; // x = x * 4`
	/=	division and assignment	`y /= 10; // y = y / 10`
	%=	modulo and assignment	`x %= 4; // x = x % 4`
	>>=	bitwise right-shift and assignment	`x >>= 5; // x = x>>5`
	<<=	bitwise left-shift and assignment	`x <<= 2; // x = x<<2`
	&=	bitwise AND and assignment	`y &= 15; // y = y & 15`
	|=	bitwise OR and assignment	`x |= y; // x = x | y`
	^=	bitwise XOR and assignment	`x ^= y; // x = x ^ y`

Figure C. 5 Operators listed by decreasing precedence

C considers a variable to be TRUE if it is nonzero and FALSE if it is zero. Logical and ternary operators, as well as control-flow statements such as if and while, depend on the truth of a variable. Relational and logical operators produce a result that is 1 when TRUE or 0 when FALSE.

Expression		Result	Notes
`44 / 14`		`3`	Integer division truncates
`44 % 14`		`2`	44 mod 14
`0x2C && 0xE`	`//0b101100 && 0b1110`	`1`	Logical AND
`0x2C || 0xE`	`//0b101100 || 0b1110`	`1`	Logical OR
`0x2C & 0xE`	`//0b101100 & 0b1110`	`0xC (0b001100)`	Bitwise AND
`0x2C | 0xE`	`//0b101100 | 0b1110`	`0x2E (0b101110)`	Bitwise OR
`0x2C ^ 0xE`	`//0b101100 ^ 0b1110`	`0x22 (0b100010)`	Bitwise XOR
`0xE << 2`	`//0b1110 << 2`	`0x38 (0b111000)`	Left shift by 2
`0x2C >> 3`	`//0b101100 >> 3`	`0x5 (0b101)`	Right shift by 3
`x = 14;` `x += 2;`		`x=16`	
`y = 0x2C;` `y &= 0xF;`	`// y = 0b101100` `// y &= 0b1111`	`y=0xC (0b001100)`	
`x = 14; y = 44;` `y = y + x++;`		`x=15, y=58`	Increment x after using it
`x = 14; y = 44;` `y = y + ++x;`		`x=15, y=59`	Increment x before using it

Figure C. 6 OPERATOR EXAMPLES

C.6 FUNCTION CALLS

Modularity is key to good programming. A large program is divided into smaller parts called functions that, similar to hardware modules, have well-defined inputs, outputs, and behavior. C Code Example eC.8 shows the sum3 function. The function declaration begins with the return type, int, followed by the name, sum3, and the inputs enclosed within parentheses (int a, int b, int c). Curly braces { } enclose the body of the function. The return statement indicates the value that the function should return to its caller; this can be viewed as the output of the function. A function can only return a single value.

C Code Example eC.8: sum3 FUNCTION (Page 541.e15)

```
// Return the sum of the three input variables
int sum3(int a, int b, int c) {
  int result = a + b + c;
  return result;
}
```

Although a function may have inputs and outputs, neither is required.

C Code Example eC.9: FUNCTION printPrompt WITH NO INPUTS OR OUTPUTS (Page 541.e15)

```
// Print a prompt to the console
void printPrompt(void)
{
  printf("Please enter a number from 1-3:\n");
}
```

A function must be declared in the code before it is called. Alternatively, a function *prototype* can be placed in the program before the function is defined. The function prototype is the first line of the function, declaring the return type, function name, and function inputs.

C Code Example eC.10: FUNCTION PROTOTYPES (Page 541.e16)

```c
#include <stdio.h>

// function prototypes
int sum3(int a, int b, int c);
void printPrompt(void);

int main(void)
{
  int y = sum3(10, 15, 20);

  printf("sum3 result: %d\n", y);
  printPrompt();
}

int sum3(int a, int b, int c) {
  int result = a+b+c;
  return result;
}

void printPrompt(void) {
  printf("Please enter a number from 1-3:\n");
}
```

Console Output

```
sum3 result: 45
Please enter a number from 1-3:
```

With careful ordering of functions, prototypes may be unnecessary. However, they are unavoidable in certain cases, such as when function f1 calls f2 and f2 calls f1. It is good style to place prototypes for all of a program's functions near the beginning of the C file or in a header file.

The main function is always declared to return an int, which conveys to the operating system the reason for program termination. A zero indicates normal completion, while a nonzero value signals an error condition. If main reaches the end without encountering a return statement, it will automatically return 0.

As with variable names, function names are case sensitive, cannot be any of C's reserved words, may not contain special characters (except underscore _), and cannot start with a number. Typically function names include a verb to indicate what they do.

Be consistent in how you capitalize your function and variable names. Two common styles are to camelCase, in which the initial letter of each word after the first is capitalized like the humps of a camel (e.g., printPrompt), or to use underscores between words (e.g., print_prompt).

C.7 CONTROL-FLOW STATEMENTS

C provides control-flow statements for conditionals and loops. Conditionals execute a statement only if a condition is met. A loop repeatedly executes a statement as long as a condition is met.

- Conditional Statements

if, if/else, and switch/case statements are conditional statements.

if Statements

An if statement executes the statement immediately following it when the expression in parentheses is TRUE (i.e., nonzero). The general format is:

```
if (expression)
    statement
```

C Code Example eC.11: if STATEMENT (Page 541.e17)

```c
int dontFix = 0;

if (aintBroke == 1)
  dontFix = 1;
```

Curly braces, { }, are used to group one or more statements into a *compound statement* or *block*.

C Code Example eC.12: if STATEMENT WITH A BLOCK OF CODE (Page 541.e17)

```c
// If amt >= $2, prompt user and dispense candy
if (amt >= 2) {
  printf("Select candy.\n");
  dispenseCandy = 1;
}
```

if/else Statements

if/else statements execute one of two statements depending on a condition. As shown in the general format below.

if (expression)
    statement1
else
    statement2

switch/case Statements

switch/case statements execute one of several statements depending on the conditions, as shown in the general format below.

switch (variable) {
    case (expression1): statement1 break;
    case (expression2): statement2 break;
    case (expression3): statement3 break;
    default: statement4
}

C Code Example eC.13: switch/case STATEMENT (Page 541.e18)

```
// Assign amt depending on the value of option
switch (option) {
  case 1:  amt = 100; break;
  case 2:  amt = 50;  break;
  case 3:  amt = 20;  break;
  case 4:  amt = 10;  break;
  default: printf("Error: unknown option.\n");
}
```

● Loops

while, do/while, and for loops are common loop constructs.

while Loops

while loops repeatedly execute a statement until a condition is not met, as shown in the general format below.

while (condition)
    statement

C Code Example eC.15: while LOOP (Page 541.e19)

do/while Loops

do/while loops are like while loops but the condition is checked only after the statement is executed once. The general format is shown below. The condition is followed by a semi-colon.

do
    statement
while (condition);

C Code Example eC.16: do/while LOOP (Page 541.e19)

```
// Query user to guess a number and check it against the correct number.
#define MAXGUESSES 3
#define CORRECTNUM 7
int guess, numGuesses = 0;
```

```
do {
  printf("Guess a number between 0 and 9. You have %d more guesses.\n",
         (MAXGUESSES-numGuesses));
  scanf("%d", &guess);     // read user input
  numGuesses++;
} while ( (numGuesses < MAXGUESSES) & (guess != CORRECTNUM) );
// do loop checks the condition after the first iteration

if (guess == CORRECTNUM)
  printf("You guessed the correct number!\n");
```

for Loops

for loops, like while and do/while loops, repeatedly execute a statement until a condition is not satisfied. However, for loops add support for a *loop variable*, which typically keeps track of the number of loop executions. The general format of the for loop is

for (initialization; condition; loop operation)
    statement

The initialization code executes only once, before the for loop begins. The condition is tested at the beginning of each iteration of the loop. If the condition is not TRUE, the loop exits. The loop operation executes at the end of each iteration.

C Code Example eC.17: for LOOP (Page 541.e20)

```
// Compute 9!
int i;    // loop variable
int fact = 1;

for (i=1; i<10; i++)
  fact *= i;
```

C.8 MORE DATA TYPES

- Pointers

A pointer is the address of a variable.

C Code Example eC.18: POINTERS (Page 541.e22)

```
// Example pointer manipulations
int salary1, salary2;  // 32-bit numbers
int *ptr;              // a pointer specifying the address of an int variable

salary1 = 67500;       // salary1 = $67,500 = 0x000107AC
ptr = &salary1;        // ptr = 0x0070, the address of salary1
salary2 = *ptr + 1000; /* dereference ptr to give the contents of address 70 = $67,500,
                          then add $1,000 and set salary2 to $68,500 */
```

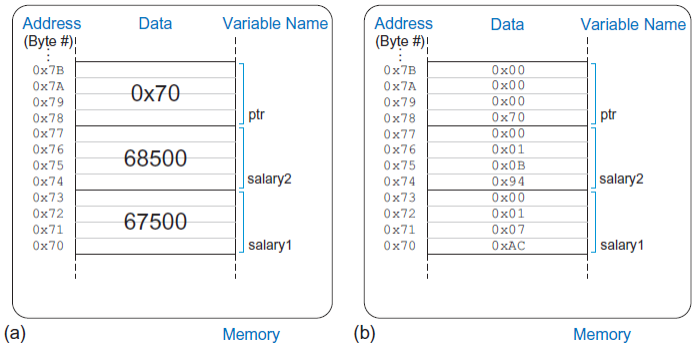


Figure C. 7 Contents of memory after C Code Example eC.18 executes shown (a) by value and (b) by byte using little-endian memory

In a variable declaration, a star (*) before a variable name indicates that the variable is a pointer to the declared type. In using a pointer variable, the * operator dereferences a pointer, returning the value stored at the indicated memory address contained in the pointer. The & operator is pronounced "address of," and it produces the memory address of the variable being referenced.

Dereferencing a pointer to a non-existent memory location or an address outside of the range accessible by the program will usually cause a program to crash. The crash is often called a *segmentation fault*.

A function can make the input a pointer to the variable to modify their inputs directly. This is called passing an input variable *by reference* instead of *by value*,

C Code Example eC.19: PASSING AN INPUT VARIABLE BY REFERENCE (Page 541.e22)

```c
// Quadruple the value pointed to by a
#include <stdio.h>

void quadruple(int *a)
{
  *a = *a * 4;
}

int main(void)
{
  int x = 5;

  printf("x before: %d\n", x);
  quadruple(&x);
  printf("x after: %d\n", x);
  return 0;
}
```

Console Output

```
x before: 5
x after: 20
```

A pointer to address 0 is called a *null pointer* and indicates that the pointer is not actually pointing to meaningful data. It is written as NULL in a program.

- Arrays

An array is a group of similar variables stored in consecutive addresses in memory. The elements are numbered from 0 to $N-1$, where N is the length of the array. C Code Example eC.20 declares an array variable called scores that holds the final exam scores for three students. In C, the array variable, in this case scores, is a pointer to the 1^{st} element.

C Code Example eC.20: ARRAY DECLARATION (Page 541.e23)

```c
long scores[3];    // array of three 4-byte numbers
```

C Code Example eC.21: ARRAY INITIALIZATION AT DECLARATION USING { } (Page 541.e23)

```c
long scores[3]={93, 81, 97}; // scores[0]=93; scores[1]=81; scores[2]=97;
```

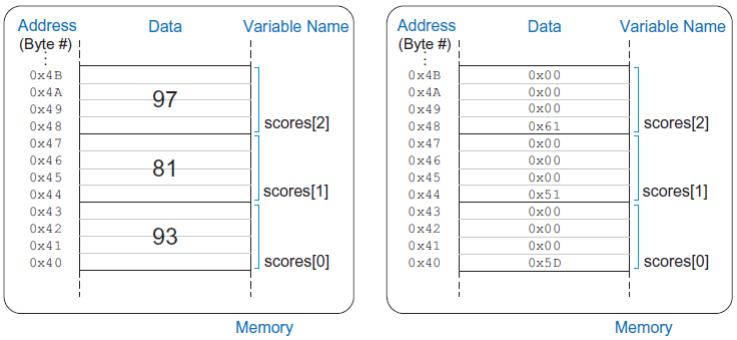


Figure C. 8 scores array stored in memory

The elements of an array can be initialized either at declaration using curly braces { },or individually in the body of the code. Each element of an array is accessed using brackets [ ]. Array initialization using curly braces { } can only be performed at declaration, and not afterward. for loops are commonly used to assign and read array data.

C Code Example eC.22: ARRAY INITIALIZATION USING ASSIGNMENT (Page 541.e24)

```c
long scores[3];

scores[0] = 93;
scores[1] = 81;
scores[2] = 97;
```

C Code Example eC.23: ARRAY INITIALIZATION USING A for LOOP (Page 541.e24)

```c
// User enters 3 student scores into an array
long scores[3];
int i, entered;

printf("Please enter the student's 3 scores.\n");
for (i=0; i<3; i++) {
  printf("Enter a score and press enter.\n");
  scanf("%d", &entered);
  scores[i] = entered;
}
printf("Scores: %d %d %d\n", scores[0], scores[1], scores[2]);
```

When an array is declared, the length must be constant so that the compiler can allocate the

proper amount of memory. However, when the array is passed to a function as an input argument, the length need not be defined because the function only needs to know the address of the beginning of the array. C Code Example eC.24 shows how an array is passed to a function. In a function, an input argument of type int[] indicates that it is an array of integers.

C Code Example eC.24: PASSING AN ARRAY AS AN INPUT ARGUMENT (Page 541.e24)

```
// Initialize a 5-element array, compute the mean, and print the result.
#include <stdio.h>

// Returns the mean value of an array (arr) of length len
float getMean(int arr[], int len) {
  int i;
  float mean, total = 0;

  for (i=0; i < len; i++)
    total += arr[i];

  mean = total / len;
  return mean;
}
```

```
int main(void) {
  int data[4] = {78, 14, 99, 27};
  float avg;

  avg = getMean(data, 4);

  printf("The average value is: %f.\n", avg);
}
```

Console Output

```
The average value is: 54.500000.
```

An array argument is equivalent to a pointer to the beginning of the array. Thus, getMean could also have been declared as

float getMean(int *arr, int len);

Although functionally equivalent, datatype[] is the preferred method for passing arrays as input arguments because it more clearly indicates that the argument is an array.

The length of an array in a function declaration (i.e., int vals[100]) is ignored.

C Code Example eC.25: PASSING AN ARRAY AND ITS LENGTH AS INPUTS (Page 541.e25)

```
// Sort the elements of the array vals of length len from lowest to highest
void sort(int vals[], int len)
{
  int i, j, temp;

  for (i=0; i<len; i++) {
    for (j=i+1; j<len; j++) {
      if (vals[i] > vals[j]) {
        temp = vals[i];
        vals[i] = vals[j];
        vals[j] = temp;
      }
    }
  }
}
```

C Code Example eC.26: TWO-DIMENSIONAL ARRAY INITIALIZATION (Page 541.e26)

```
// Initialize 2-D array at declaration
 int grades[10][8] = { {100, 107, 99,  101, 100, 104, 109, 117},
                       {103, 101, 94,  101, 102, 106, 105, 110},
                       {101, 102, 92,  101, 100, 107, 109, 110},
                       {114, 106, 95,  101, 100, 102, 102, 100},
                       {98,  105, 97,  101, 103, 104, 109, 109},
                       {105, 103, 99,  101, 105, 104, 101, 105},
                       {103, 101, 100, 101, 108, 105, 109, 100},
                       {100, 102, 102, 101, 102, 101, 105, 102},
                       {102, 106, 110, 101, 100, 102, 120, 103},
                       {99,  107, 98,  101, 109, 104, 110, 108} };
```

Multi-dimensional arrays used as input arguments to a function must define all but the first dimension. Thus, the following two function prototypes are acceptable:

void print2dArray(int arr[10][8]);

void print2dArray(int arr[][8]);

C Code Example eC.27: OPERATING ON MULTI-DIMENSIONAL ARRAYS (Page 541.e26)

```c
#include <stdio.h>

// Print the contents of a 10×8 array
void print2dArray(int arr[10][8])
{
  int i, j;

  for (i=0; i<10; i++) {          // for each of the 10 students
    printf("Row %d\n", i);
    for (j=0; j<8; j++) {
      printf("%d ", arr[i][j]); // print scores for all 8 problem sets
    }
    printf("\n");
  }
}

// Calculate the mean score of a 10×8 array
float getMean(int arr[10][8])
{
  int i, j;
  float mean, total = 0;

  // get the mean value across a 2D array
  for (i=0; i<10; i++) {
    for (j=0; j<8; j++) {
      total += arr[i][j];       // sum array values
    }
  }
  mean = total/(10*8);
  printf("Mean is: %f\n", mean);

  return mean;
}
```

● Characters

A character (char) is an 8-bit variable. It can be viewed either as a two's complement number between -128 and 127 or as an ASCII code for a letter, digit, or symbol. ASCII characters can be specified as a numeric value (in decimal, hexadecimal, etc.) or as a printable character enclosed in single quotes. Figure C.9 lists characters used to indicate formatting or special characters. Formatting codes include carriage return (\r), newline (\n), horizontal tab (\t), and the end of a string (\0). \r is shown for completeness but is rarely used in C programs. \r returns the carriage (location of typing) to the beginning (left) of the line, but any text that was there is overwritten. \n, instead, moves the location of typing to the beginning of a new line. The NULL character ('\0') indicates the end of a text string.

Special Character	Hexadecimal Encoding	Description
\r	0x0D	carriage return
\n	0x0A	new line
\t	0x09	tab
\0	0x00	terminates a string
\\	0x5C	backslash
\"	0x22	double quote
\'	0x27	single quote
\a	0x07	bell

Figure C. 9 Special characters

● Strings

A string is an array of characters used to store a piece of text of bounded but variable length. Each character is a byte representing the ASCII code for that letter, number, or symbol. The size of the array determines the maximum length of the string, but the actual length of the string could be shorter. In C, the length of the string is determined by looking for the null terminator (ASCII value 0x00) at the end of the string.

C Code Example eC.28 shows the declaration of a 10-element character array called greeting that holds the string "Hello!". Note that the string only uses the first seven elements of the array,

even though ten elements are allocated in memory.

C Code Example eC.28: STRING DECLARATION (Page 541.e28)

```
char greeting[10] = "Hello!";
```

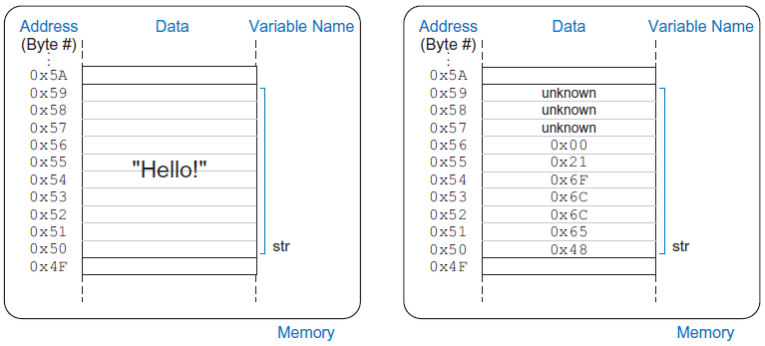


Figure C. 10 The string "Hello!" stored in memory

C Code Example eC.29 shows an alternate declaration of the string greeting. The pointer greeting holds the address of the 1^{st} element of a 7-element array comprised of each of the characters in "Hello!" followed by the null terminator.

C Code Example eC.29: ALTERNATE STRING DECLARATION (Page 541.e28)

```
char *greeting = "Hello!";
printf("greeting: %s", greeting);
```

Console Output

```
greeting: Hello!
```

C Code Example eC.30: COPYING STRINGS (Page 541.e28)

```
// Copy the source string, src, to the destination string, dst
void strcpy(char *dst, char *src)
{
```

```
  int i = 0;

  do {
    dst[i] = src[i];        // copy characters one byte at a time
  } while (src[i++]);       // until the null terminator is found
}
```

● Structures

In C, structures are used to store a collection of data of various types. The general format of a structure declaration is

struct name {

    type1 element1;

    type2 element2;

    …

};

where struct is a keyword indicating that it is a structure, name is the structure tag name, and element1 and element2 are members of the structure.

C Code Example eC.31: STRUCTURE DECLARATION (Page 541.e29)

```
struct contact {
  char name[30];
  int phone;
  float height; // in meters
};

struct contact c1;
```

```
strcpy(c1.name, "Ben Bitdiddle");
c1.phone = 7226993;
c1.height = 1.82;
```

Just like built-in C types, you can create arrays of structures and pointers to structures.

C Code Example eC.32: ARRAY OF STRUCTURES (Page 541.e30)

```
struct contact classlist[200];
classlist[0].phone = 9642025;
```

It is common to use pointers to structures. C provides the *member access operator* -> to dereference a pointer to a structure and access a member of the structure.

C Code Example eC.33: ACCESSING STRUCTURE MEMBERS USING POINTERS AND -> (Page 541.e30)

```
struct contact *cptr;
cptr = &classlist[42];
cptr->height = 1.9;  // equivalent to: (*cptr).height = 1.9;
```

Structures can be passed as function inputs or outputs by value or by reference.

C Code Example eC.34: PASSING STRUCTURES BY VALUE OR BY NAME (Page 541.e30)

```
struct contact stretchByValue(struct contact c)
{
  c.height += 0.02;
  return c;
}
```

```
void stretchByReference(struct contact *cptr)
{
  cptr->height += 0.02;
}

int main(void)
{
  struct contact George;

  George.height = 1.4; // poor fellow has been stooped over
  George = stretchByValue(George); // stretch for the stars
  stretchByReference(&George);     // and stretch some more
}
```

- typedef

C also allows you to define your own names for data types using the typedef statement.

C Code Example eC.35: CREATING A CUSTOM TYPE USING typedef (Page 541.e31)

```
typedef struct contact {
  char name[30];
  int phone;
  float height; // in meters
} contact;        // defines contact as shorthand for "struct contact"

contact c1;       // now we can declare the variable as type contact
```

typedef can be used to create a new type occupying the same amount of memory as a primitive type.

C Code Example eC.36: typedef byte AND bool (Page 541.e31)

```
typedef unsigned char byte;
typedef char bool;
#define TRUE 1
#define FALSE 0

byte pos = 0x45;
bool loveC = TRUE;
```

These types make a program easier to read than if one simply used char everywhere.

C Code Example eC.37: typedef vector AND matrix (Page 541.e32)

```
typedef double vector[3];
typedef double matrix[3][3];

vector a = {4.5, 2.3, 7.0};
matrix b = {{3.3, 4.7, 9.2}, {2.5, 4, 9}, {3.1, 99.2, 88}};
```

- Dynamic Memory Allocation

In all the examples thus far, variables have been declared *statically*; that is, their size is known at compile time. An alternative is to *dynamically* allocate memory at run time when the actual size is known.

The malloc function from stdlib.h allocates a block of memory of a specified size and returns a pointer to it. If not enough memory is available, it returns a NULL pointer instead. For example,

// dynamically allocate 20 bytes of memory

short *data = malloc(10*sizeof(short));

C Code Example eC.38: DYNAMIC MEMORY ALLOCATION AND DE-ALLOCATION (Page 541.e32)

```
// Dynamically allocate and de-allocate an array using malloc and free
#include <stdlib.h>

// Insert getMean function from C Code Example eC.24.

int main(void) {
  int len, i;

  int *nums;

  printf("How many numbers would you like to enter?    ");
  scanf("%d", &len);
  nums = malloc(len*sizeof(int));
  if (nums == NULL) printf("ERROR: out of memory.\n");
  else {
    for (i=0; i<len; i++) {
      printf("Enter number:    ");
      scanf("%d", &nums[i]);
    }
    printf("The average is %f\n", getMean(nums, len));
  }
  free(nums);
}
```

● Linked Lists

A *linked list* is a common data structure used to store a variable number of elements. Each element in the list is a structure containing one or more data fields and a link to the next element. The first element in the list is called the *head*.

C Code Example eC.39: LINKED LIST (Page 541.e33)

```
#include <stdlib.h>
#include <string.h>

typedef struct userL {
```

```
  char uname[80];     // user name
  char passwd[80];    // password
  int uid;            // user identification number
  int admin;          // 1 indicates administrator privileges
  struct userL *next;
} userL;

userL *users = NULL;

void insertUser(char *uname, char *passwd, int uid, int admin) {
  userL *newUser;

  newUser = malloc(sizeof(userL));  // create space for new user
  strcpy(newUser->uname, uname);    // copy values into user fields
  strcpy(newUser->passwd, passwd);
  newUser->uid = uid;
  newUser->admin = admin;
  newUser->next = users;            // insert at start of linked list
  users = newUser;
}

void deleteUser(int uid) {  // delete first user with given uid
  userL *cur = users;
  userL *prev = NULL;

  while (cur != NULL) {
    if (cur->uid == uid) { // found the user to delete
      if (prev == NULL) users = cur->next;
      else prev->next = cur->next;
      free(cur);
      return; // done
    }
    prev = cur;       // otherwise, keep scanning through list
    cur = cur->next;
  }
}

userL *findUser(int uid) {
  userL *cur = users;

  while (cur != NULL) {
    if (cur->uid == uid) return cur;
    else cur = cur->next;
  }
  return NULL;
}

int numUsers(void) {
  userL *cur = users;
  int count = 0;

  while (cur != NULL) {
    count++;
    cur = cur->next;
  }
  return count;
}
```

C.9 STANDARD LIBRARIES

C Library Header File	Description
stdio.h	Standard input/output library. Includes functions for printing or reading to/from the screen or a file (printf, fprintf and scanf, fscanf) and to open and close files (fopen and fclose).
stdlib.h	Standard library. Includes functions for random number generation (rand and srand), for dynamically allocating or freeing memory (malloc and free), terminating the program early (exit), and for conversion between strings and numbers (atoi, atol, and atof).
math.h	Math library. Includes standard math functions such as sin, cos, asin, acos, sqrt, log, log10, exp, floor, and ceil.
string.h	String library. Includes functions to compare, copy, concatenate, and determine the length of strings.

Figure C. 11 Frequently used C libraries

● stdio

The standard input/output library stdio.h contains commands for printing to a console, reading keyboard input, and reading and writing files. To use these functions, the library must be included at the top of the C file:

#include <stdio.h>

printf

The *print formatted* statement printf displays text to the console.

Code	Format
%d	Decimal
%u	Unsigned decimal
%x	Hexadecimal
%o	Octal
%f	Floating point number (float or double)
%e	Floating point number (float or double) in scientific notation (e.g., 1.56e7)
%c	Character (char)
%s	String (null-terminated array of characters)

Figure C. 12 printf format codes for printing variables

C Code Example eC.40: PRINTING TO THE CONSOLE USING printf (Page 541.e36)

C Code Example eC.41: FLOATING POINT NUMBER FORMATS FOR PRINTING (Page 541.e37)

C Code Example eC.42: PRINTING % AND \ USING printf (Page 541.e37)

scanf

The scanf function reads text typed on the keyboard.

C Code Example eC.43: READING USER INPUT FROM THE KEYBOARD WITH scanf (Page 541.e38)

File Manipulation

C Code Example eC.44: PRINTING TO A FILE USING fprintf (Page 541.e39)

C Code Example eC.45: READING INPUT FROM A FILE USING fscanf (Page 541.e39)

Other Handy stdio Functions

● stdlib

#include <stdlib.h>

rand **and** srand

C Code Example eC.46: RANDOM NUMBER GENERATION USING rand (Page 541.e41)

```
#include <stdlib.h>
int x, y;

x = rand();        // x = a random integer
y = rand() % 10;   // y = a random number from 0 to 9
printf("x = %d, y = %d\n", x, y);
```

Console Output:

```
x = 1481765933, y = 3
```

C Code Example eC.47: SEEDING THE RANDOM NUMBER GENERATOR USING srand (Page 541.e41)

```
// Produce a different random number each run
#include <stdlib.h>
#include <time.h>    // needed to call time()

int main(void)
{
  int x;

  srand(time(NULL));    // seed the random number generator
  x = rand() % 10;      // random number from 0 to 9
  printf("x = %d\n", x);
}
```

exit

The exit function terminates a program early.

Format Conversion: atoi, atol, atof

C Code Example eC.48: FORMAT CONVERSION (Page 541.e42)

● math

#include <math.h>

C Code Example eC.49: MATH FUNCTIONS (Page 541.e42)

● string

The string library string.h provides commonly used string manipulation functions. Key functions include:

```
// copy src into dst and return dst
char *strcpy(char *dst, char *src);
// concatenate (append) src to the end of dst and return dst
char *strcat(char *dst, char *src);
// compare two strings. Return 0 if equal, nonzero otherwise
int strcmp(char *s1, char *s2);
// return the length of str, not including the null termination
int strlen(char *str);
```

C.10 COMPILER AND COMMAND LINE OPTIONS

Compiler Option	Description	Example
`-o` outfile	specifies output file name	`gcc -o hello hello.c`
`-S`	create assembly language output file (not executable)	`gcc -S hello.c` this produces `hello.s`
`-v`	verbose mode – prints the compiler results and processes as compilation completes	`gcc -v hello.c`
`-O`level	specify the optimization level (level is typically 0 through 3), producing faster and/or smaller code at the expense of longer compile time	`gcc -O3 hello.c`
`--version`	list the version of the compiler	`gcc -version`
`--help`	list all command line options	`gcc --help`
`-Wall`	print all warnings	`gcc -Wall hello.c`

Figure C. 13 Compiler options

C.11 COMMON MISTAKES