Computer Organization and Architecture Unit 1

Computer Organization and Architecture

- Computer Organization and Architecture is used to design computer systems
- Computer Architecture addresses attributes visible to the user:
  - Addressing techniques
  - Instruction sets
  - Bits used for data

  These have direct impact on the logic execution of the program
- Computer Organization addresses:
  - The way in which a system has to structure
  - Operational units of a computer and interconnections between them

  Computer Architecture defines the system in an abstract manner and deals with what a system can do Whereas Computer organization realizes the abstract model and deals with how to implement the system.

Computer Types

- Embedded Computers
  - They are integrated into a larger device or system in order to automatically monitor and control a physical process or environment.
  - They are used for a specific purpose rather than for general processing tasks.
  - Used for industrial and home automation, appliances, telecommunication products, and vehicles.
- Personal Computers
  - have widespread use in homes, educational institutions, and business and engineering office settings, primarily for dedicated individual use.
  - They support general computation, document preparation, computer-aided design, audiovisual entertainment, interpersonal communication, and Internet browsing.
  - Ex: Desktop, Workstations, Portable and Notebooks

Computer Types

- Servers and Enterprise systems
  - These are shared by a potentially large number of users who access them from some form of personal computer over a public or private network.
  - They host large databases and provide information processing for a government or a commercial organization
- Super Computers and Grid systems
  - They are the most expensive and physically the largest category of computers.
  - They are used for the highly demanding computations needed in weather forecasting, engineering design and simulation, and scientific work.
  - They have a high cost.
  - Grid computers provide a more cost-effective alternative.
  - They combine a large number of personal computers and disk storage units in a physically distributed high-speed network, called a grid

Functional Units

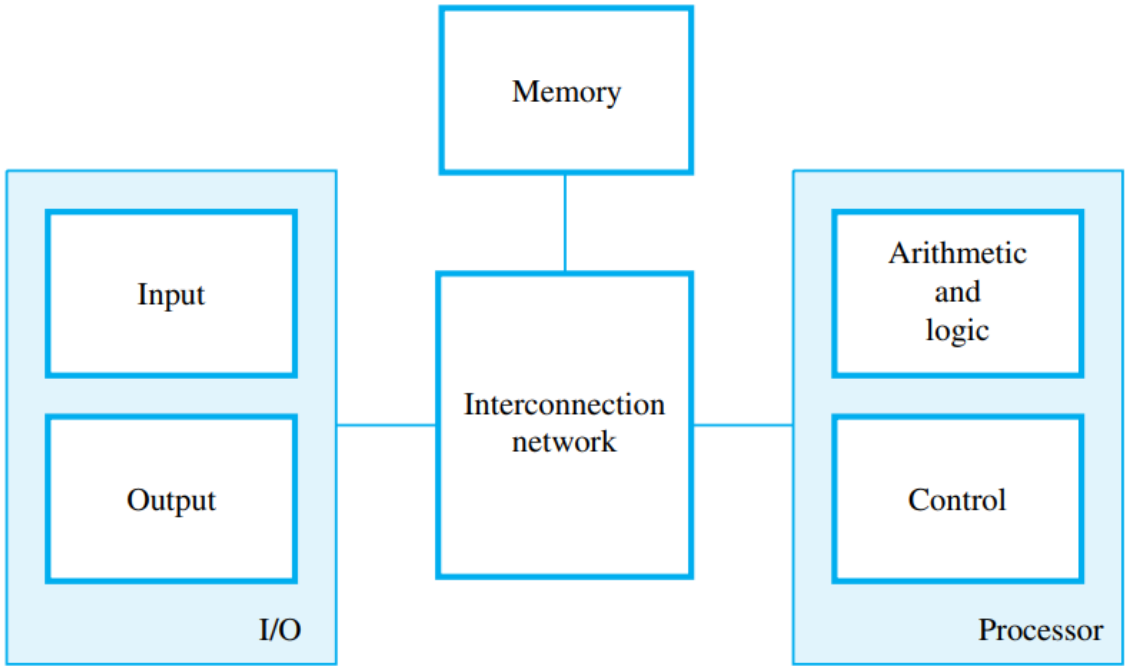


Figure 1.1 Basic functional units of a computer.

Functional Units

- The **input unit** accepts coded information from human operators using keyboards, or from other computers over the network.
- The information received is stored in the computer's **memory**
- Information is processed by the **ALU**. The processing steps are specified by a program stored in the memory.
- Finally, the results are sent back to the outside world through the **output unit.**
- All of these actions are coordinated by the **control unit.**
- An **interconnection network** provides the means for the functional units to exchange information and coordinate their actions.

Functional Units

- Input Unit:
  - Computers accept coded information through input units.
  - Ex: Keyboard, Touch pad, Mouse, Joy stick, Track ball
  - Microphones can provide audio input
- Memory Unit: The function of the memory unit is to store programs and data.
  - Primary Memory:
    - Primary memory, also called main memory, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed.
  - Cache:
    - As an adjunct to the main memory, a smaller, faster RAM unit, called a cache, is used to hold sections of a program that are currently being executed, along with any associated data.
    - The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip.
    - The purpose of the cache is to facilitate high instruction execution rates.
  - Secondary Memory:
    - less expensive, permanent secondary storage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently.
    - Access times for secondary storage are longer than for primary memory.
    - Ex: magnetic disks, optical disks (DVD and CD), and flash memory devices.

Functional Units

- ALU:
  - Any arithmetic or logic operation, such as addition, subtraction, multiplication division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU.
- Output Unit:
  - Its function is to send processed results to the outside world.
  - Ex: Monitor, Printer
- Control Unit:
  - The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units are coordinated by the control unit.

Basic Operational Concepts

- The activity in a computer is governed by instructions.
- To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory.
- Instructions are brought from the memory into the processor, which executes the specified operations.
- Data operands are also stored in the memory.
- A typical instruction might be Load R2, LOC
- This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2.
- The original contents of location LOC are preserved, whereas those of register R2 are overwritten.

Basic Operational Concepts

- First, the instruction is fetched from the memory into the processor.
- Next, the operation to be performed is determined by the control unit.
- The operand at LOC is then fetched from the memory into the processor.
- Finally, the operand is stored in register R2.
- After operands have been loaded from memory into processor registers, arithmetic or logic operations can be performed on them.
- For example, the instruction Add R4, R2, R3 adds the contents of registers R2 and R3, then places their sum into register R4.
- The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum.
- After completing the desired operations, the results are in processor registers.
- They can be transferred to the memory using instructions such as below
- Store R4, LOC This instruction copies the operand in register R4 to memory location LOC

Basic Operational Concepts

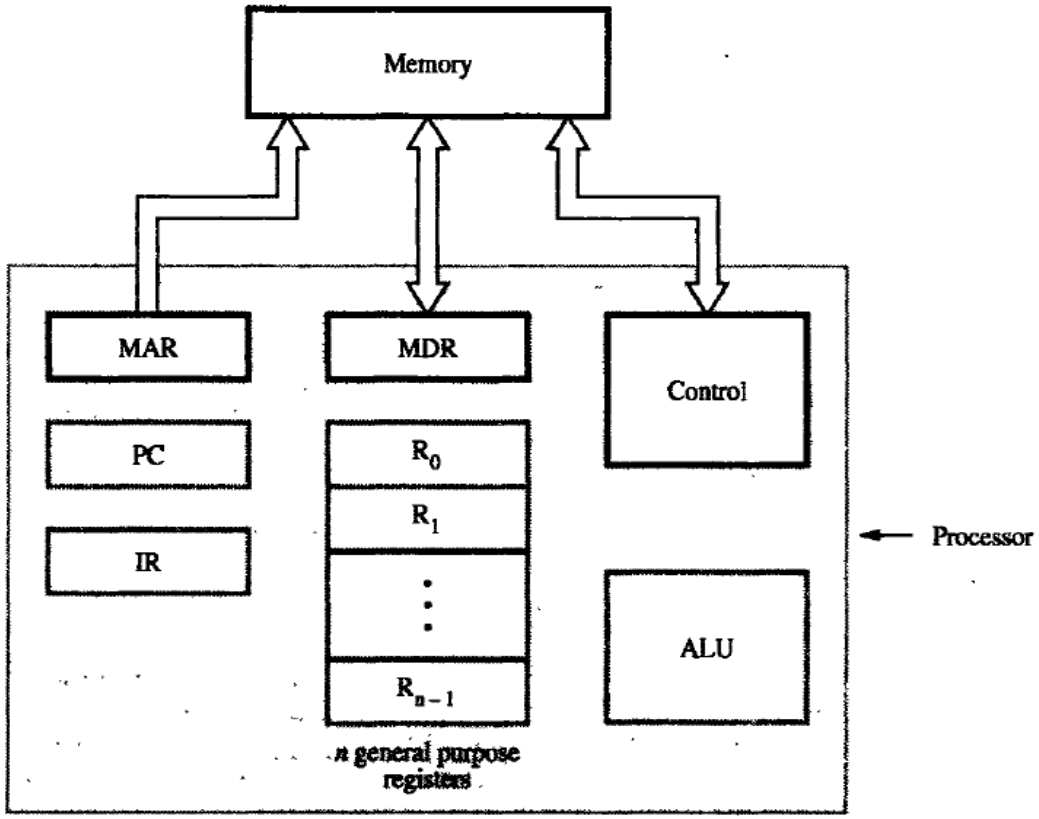


Figure 1.2 Connections between the processor and the memory.

Basic Operational Concepts

- In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes.
- The instruction register (IR) holds the instruction that is currently being executed.
- Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction.
- The program counter (PC) is another specialized register which the memory address of the next instruction to be fetched and executed.
- During the execution of an instruction, the contents of the PC are updated to point to the address of the next instruction to be executed.
- PC points to the next instruction that is to be fetched from the memory

Basic Operational Concepts

- In addition to the IR and PC, Figure 1.2 shows general-purpose registers R0 through Rn−1, often called processor registers.
- They serve a variety of functions, including holding operands that have been loaded from the memory for processing.
- The processor-memory interface is a circuit which manages the transfer of data between the main memory and the processor.
- If a word is to be read from the memory, the interface sends the address of that word to the memory along with a Read control signal.
- The interface waits for the word to be retrieved, then transfers it to the appropriate processor register.
- If a word is to be written into memory, the interface transfers both the address and the word to the memory along with a Write control signal.

Basic Operational Concepts

- A program must be in the main memory in order for it to be executed.
- Execution of the program begins when the PC is set to point to the first instruction of the program.
- The contents of the PC are transferred to the memory along with a Read control signal.
- When the addressed word (in this case, the first instruction of the program) has been fetched from the memory it is loaded into register IR.
- At this point, the instruction is ready to be interpreted and executed.

Basic Operational Concepts

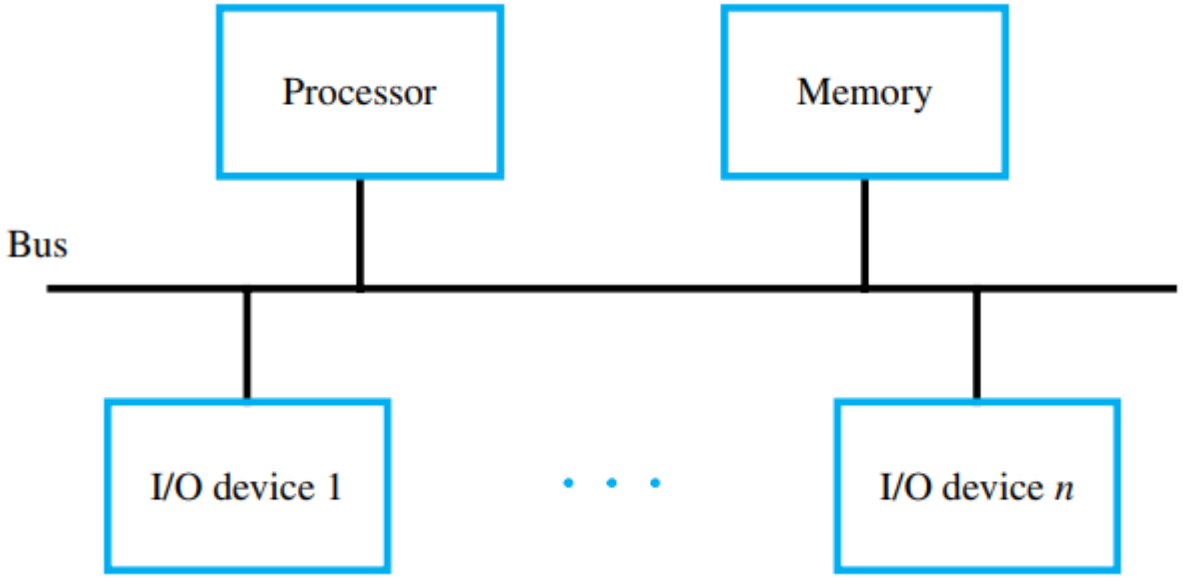
- Instructions such as Load, Store, and Add perform data transfer and arithmetic operations.
- If an operand resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation.
- When the operand has been fetched from the memory, it is transferred to a processor register.
- After operands have been fetched, the ALU can perform a desired arithmetic operation, such as Add, on the values in processor registers.
- The result is sent to a processor register.
- If the result is to be written into the memory with a Store instruction, it is transferred from the processor register to the memory, along with the address of the location where the result is to be stored, then a Write operation is initiated.
- In addition to transferring data between the memory and the processor, the computer accepts data from input devices and sends data to output devices. Thus, some machine instructions are provided for the purpose of handling I/O transfers.

Basic Operational Concepts

- Normal execution of a program may be preempted if some device requires urgent service.
- For example, a monitoring device in a computer-controlled industrial process may detect a dangerous condition.
- In order to respond immediately, execution of the current program must be suspended.
- To cause this, the device raises an interrupt signal, which is a request for service by the processor.
- The processor provides the requested service by executing a program called an interrupt-service routine.
- Because such diversions may alter the internal state of the processor, its state must be saved in the memory before servicing the interrupt request.
- Normally, the information that is saved includes the contents of the PC, the contents of the general-purpose registers, and some control information.
- When the interrupt-service routine is completed, the state of the processor is restored from the memory so that the interrupted program may continue

Bus Structure

- Bus structure implements the interconnection network of a computer
- Only one source/destination pair of units can use this bus to transfer data at any one time.



- The functional units have to be connected in an organized way to form an operational system
- The units have to be connected such a way that at a given point of time, one full word of data to be handled.
- Bits of full word are transferred parallelly/ simultaneously
- A group of wires that serves as a connecting path for several devises is called a bus
- Bus lines will carry data/address/control information
- Only two functional units can actively use a bus at a given time, as a bus can be used for only one transfer at a time
- Control lines arbitrate multiple requests for the use of the bus

- Single bus for data/address/control is cost effective and flexible for multiple devices but slows down the operation
- Multiple buses achieve concurrency in operation allowing 2 or more transfers in parallel. This leads to better performance but increased cost.
- Devices connected to the bus vary in operational speed. Devices like keyboard and printers are relatively slow compared to processor and memory which are at electronic speed
- Buffer registers are used to bridge the speed gaps

Number Representation and Arithmetic Operations- Integers

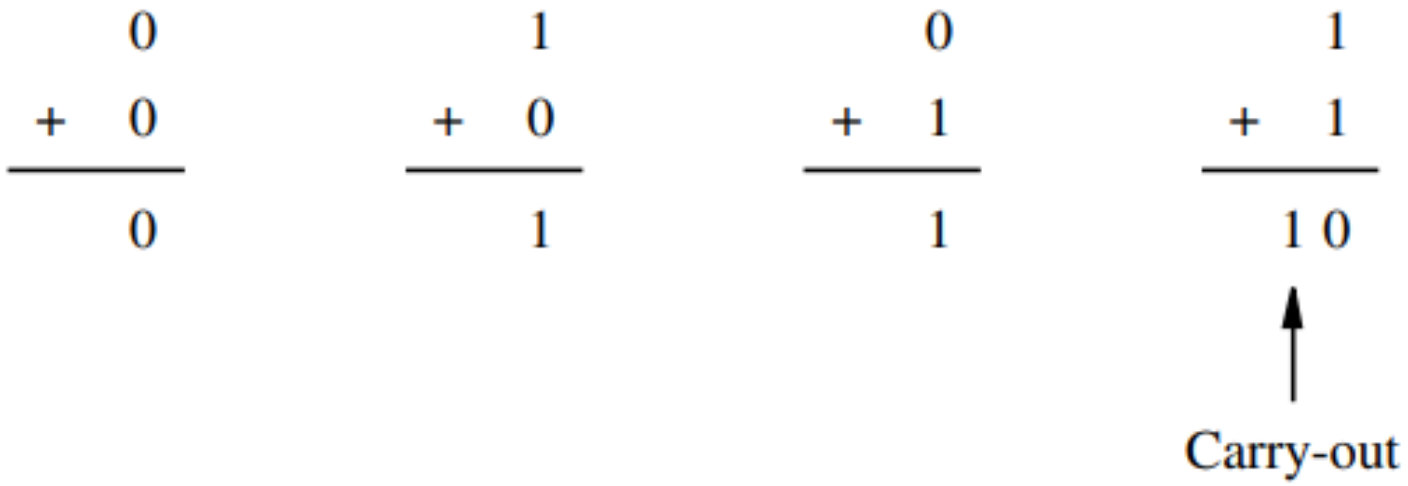
- Consider an n-bit vector B = $b_{n-1} \ldots b_1 b_0$
  - where $b_i = 0$ or 1 for $0 \leq i \leq n - 1$.
- This vector can represent an unsigned integer value V(B) in the range 0 to $2^n - 1$, where $V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$
- Three systems to represent both positive and negative numbers.
- In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers.
- Three systems are used for representing such numbers:
  - Sign-and-magnitude
  - 1's-complement
  - 2's-complement

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

Figure 1.3 Binary, signed-integer representations.

- Figure 1.3 illustrates all three representations using 4-bit numbers.
- Positive values have identical representations in all systems, but negative values have different representations.
- In the sign-and-magnitude system, negative values are represented by changing the most significant bit (b3 in Figure 1.3) from 0 to 1 in the B vector of the corresponding positive value.
- For example, +5 is represented by 0101, and −5 is represented by 1101.
- In 1’s-complement representation, negative values are obtained by complementing each bit of the corresponding positive number.
- Thus, the representation for −3 is obtained by complementing each bit in the vector 0011 to yield 1100.
- The same operation, bit complementing, is done to convert a negative number to the corresponding positive value.
- the 2’s-complement of a number is obtained by adding 1 to the 1’s-complement of that number
- 2’s-complement system leads to the most efficient way to carry out addition and subtraction operations. It is the one most often used in modern computers.

Addition of Unsigned Integers



Addition and Subtraction of Signed Integers

- Adding +7 and -3 is shown below
- If we ignore the carry-out from the fourth bit position in this addition, we obtain the correct answer.
- In fact, this is always the case. Ignoring this carry-out is a natural result of using mod N arithmetic.

```
    0  1  1  1
+   1  1  0  1
-------------
1   0  1  0  0
↑
Carry-out
```

2’s-complement Add and Subtract operations.

(a)	0 0 1 0	(+2)		(b)	0 1 0 0	(+4)	
	+ 0 0 1 1	(+3)			+ 1 0 1 0	(–6)	
	0 1 0 1	(+5)			1 1 1 0	(–2)	
(c)	1 0 1 1	(–5)		(d)	0 1 1 1	(+7)	
	+ 1 1 1 0	(–2)			+ 1 1 0 1	(–3)	
	1 0 0 1	(–7)			0 1 0 0	(+4)	
(e)	1 1 0 1	(–3)			1 1 0 1		
	– 1 0 0 1	(–7)	⇒		+ 0 1 1 1		
					0 1 0 0	(+4)	
(f)	0 0 1 0	(+2)			0 0 1 0		
	– 0 1 0 0	(+4)	⇒		+ 1 1 0 0		
					1 1 1 0	(–2)	

2’s-complement Add and Subtract operations

(g)	0 1 1 0	(+6)	⇒	0 1 1 0	
	− 0 0 1 1	(+3)		+ 1 1 0 1	
				0 0 1 1	(+3)
(h)	1 0 0 1	(−7)	⇒	1 0 0 1	
	− 1 0 1 1	(−5)		+ 0 1 0 1	
				1 1 1 0	(−2)
(i)	1 0 0 1	(−7)	⇒	1 0 0 1	
	− 0 0 0 1	(+1)		+ 1 1 1 1	
				1 0 0 0	(−8)
(j)	0 0 1 0	(+2)	⇒	0 0 1 0	
	− 1 1 0 1	(−3)		+ 0 0 1 1	
				0 1 0 1	(+5)

Sign Extension

- We often need to represent a value given in a certain number of bits by using a larger number of bits.
- For a positive number, this is achieved by adding 0s to the left.
- For a negative number in 2’s-complement representation, the leftmost bit, which indicates the sign of the number, is a 1.

Overflow in Integer Arithmetic

- Using 2’s-complement representation, n bits can represent values in the range -2^{n-1} to $+2^{n-1} - 1$.
- For example, the range of numbers that can be represented by 4 bits is −8 through +7, as shown in Figure 1.3.
- When the actual result of an arithmetic operation is outside the representable range, an arithmetic overflow has occurred.
- When adding unsigned numbers, a carry-out of 1 from the most significant bit position indicates that an overflow has occurred

- However, this is not always true when adding signed numbers.
- For example, using 2’s-complement representation for 4-bit signed numbers, if we add +7 and +4, the sum vector is 1011, which is the representation for −5, an incorrect result.
- In this case, the carry-out bit from the MSB position is 0. If we add −4 and −6, we get 0110 = +6, also an incorrect result.
- In this case, the carry-out bit is 1.
- Hence, the value of the carry-out bit from the sign-bit position is not an indicator of overflow.
- Clearly, overflow may occur only if both summands have the same sign.
- The addition of numbers with different signs cannot cause overflow because the result is always within the representable range.
- . When both summands have the same sign, an overflow has occurred when the sign of the sum is not the same as the signs of the summands.

Memory Locations and Addresses

- The memory consists of many millions of storage cells, each of which can store a bit of information 0 or 1.
- Bits are dealt groups of fixed size
- The memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation.
- Each group of n bits is referred to as a word of information, and n is called the word length.
- The memory of a computer can be schematically represented as a collection of words, as shown in Figure 2.1.

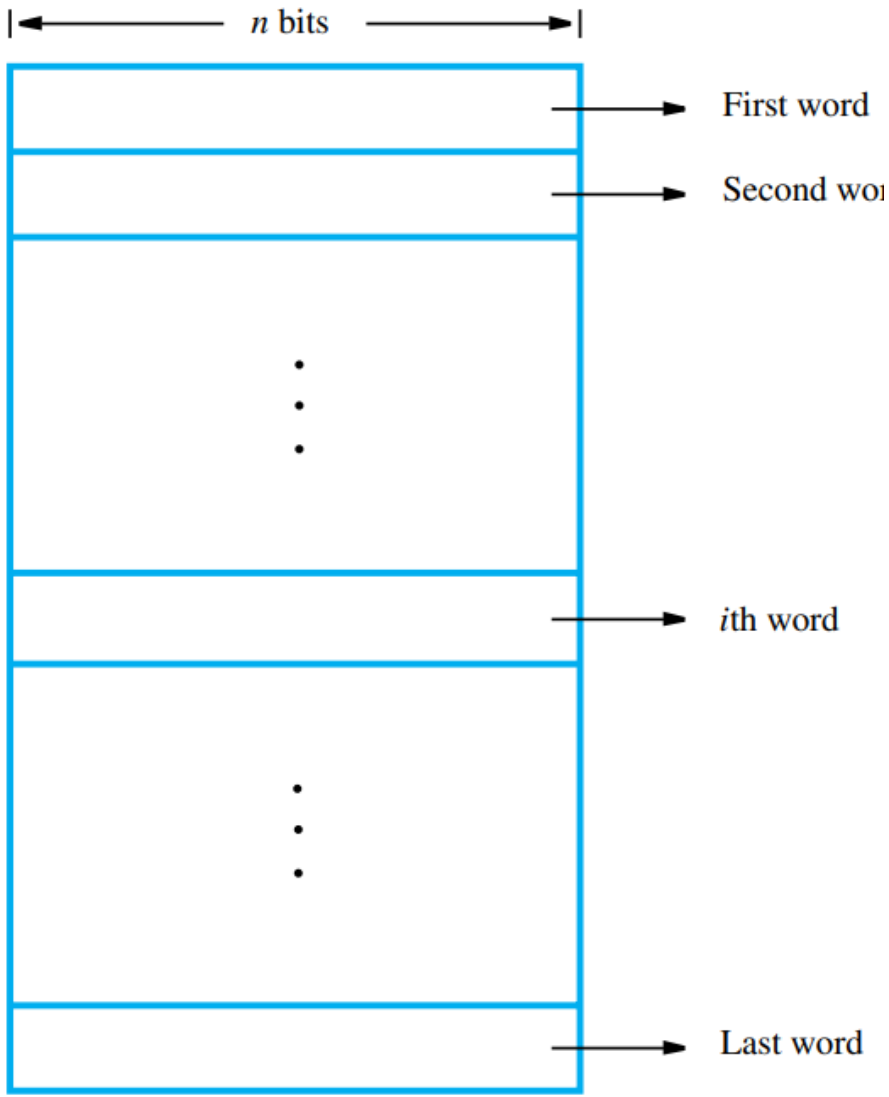
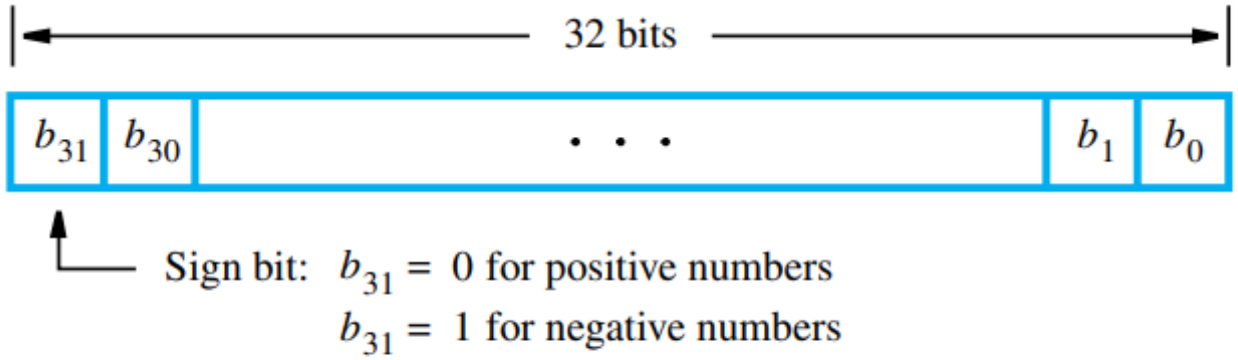
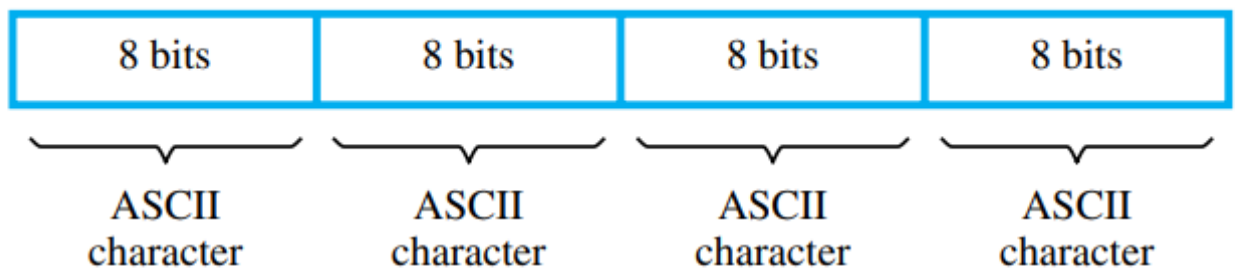


Figure 2.1 Memory words.

- Modern computers have word lengths that typically range from 16 to 64 bits.
- If the word length of a computer is 32 bits, a single word can store a 32-bit signed number
- or four ASCII-encoded characters, each occupying 8 bits, as shown in Figure 2.2.
- A unit of 8 bits is called a byte. Machine instructions may require one or more words for their representation.



(a) A signed integer



(b) Four characters

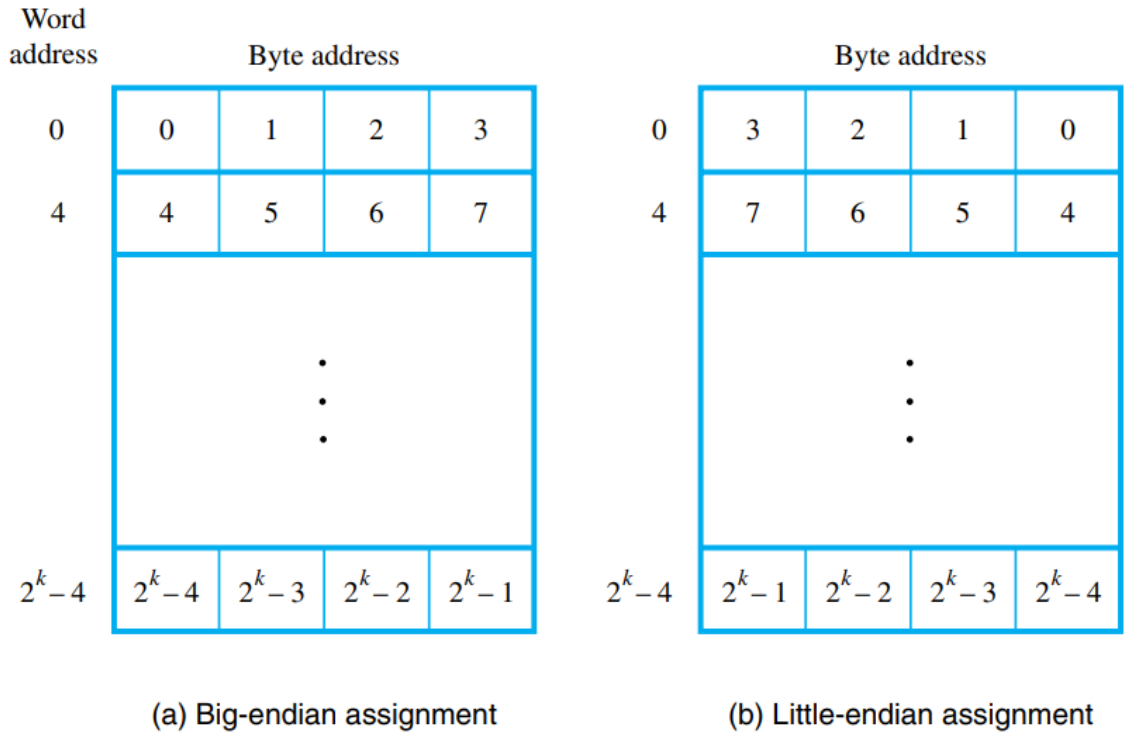
Figure 2.2 Examples of encoded information in a 32-bit word.

- Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or addresses for each location.
- It is customary to use numbers from 0 to 2k − 1, for some suitable value of k, as the addresses of successive locations in the memory. Thus, the memory can have up to 2k addressable locations.
- The 2k addresses constitute the address space of the computer. For example, a 24-bit address generates an address space of 2 power 24 (16,777,216) locations.
- This number is usually written as 16M (16 mega), where 1M is the number 2 power 20 (1,048,576). A 32-bit address creates an address space of 2 power 32 or 4G (4 giga) locations, where 1G is 2 power 30.
- Other notational conventions that are commonly used are K (kilo) for the number 2 power 10 (1,024), and T (tera) for the number 2 power 40.

Byte Addressability

- Three basic information quantities to deal with: bit, byte, and word.
- A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits.
- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory.
- This is the assignment used in most modern computers. The term byte-addressable memory is used for this assignment.
- Byte locations have addresses 0, 1, 2,....
- Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8,..., with each word consisting of four bytes.

Big-Endian and Little-Endian Assignments



(a) Big-endian assignment

(b) Little-endian assignment

Figure 2.3 Byte and word addressing.

Big-Endian and Little-Endian Assignments

- There are two ways that byte addresses can be assigned across words, as shown in Figure 2.3.
- The big-endian -when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word.
- The little-endian - when the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word.
- Both little-endian and big-endian assignments are used in commercial machines.
- In both cases, byte addresses 0, 4, 8,..., are taken as the addresses of successive words in the memory of a computer with a 32-bit word length.
- These are the addresses used when accessing the memory to store or retrieve a word.

Word Alignment

- In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8,..., as shown in Figure 2.3.
- We say that the word locations have aligned addresses if they begin at a byte address that is a multiple of the number of bytes in a word.
- For practical reasons associated with manipulating binary-coded addresses, the number of bytes in a word is a power of 2.
- Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0, 2, 4,..., and for a word length of 64 (23 bytes), aligned words begin at byte addresses 0, 8, 16,...
- There is no fundamental reason why words cannot begin at an arbitrary byte address.
- In that case, words are said to have unaligned addresses.
- But, the most common case is to use aligned addresses, which makes accessing of memory operands more efficient

Performance

- The most important measure of the performance of a computer is how quickly it can execute programs.
- The speed with which a computer executes programs is affected by the design of its instruction set, its hardware and its software, including the operating system, and the technology in which the hardware is implemented.
- Because programs are usually written in a high-level language, performance is also affected by the compiler that translates programs into machine language

Basic Performance Equation

- T – processor time required to execute a program that has been prepared in high-level language
- N – number of actual machine language instructions needed to complete the execution (note: loop)
- S – average number of basic steps needed to execute one machine instruction. Each step completes in one clock cycle
- R – clock rate
- Note: these are not independent to each other

$$T = \frac{N \times S}{R}$$

Let N be 100 instructions, S be 5, R be 2 GHZ Then T=

- T= (N*S)/R
- T=(100*5)/(2 *10 power 9)
- T=500/(2 *10 power 9)
- T=250 *10 power -9
- =250 ns

Pipeline and Superscalar Operation

- Instructions are not necessarily executed one after another.
- The value of S doesn’t have to be the number of clock cycles to execute one instruction.
- Pipelining – overlapping the execution of successive instructions.
- Add R1, R2, R3
- Superscalar operation – multiple instruction pipelines are implemented in the processor.
- Goal – reduce S (could become <1!)

Clock Rate

- Increase clock rate

➢Improve the integrated-circuit (IC) technology to make the circuits faster

➢Reduce the amount of processing done in one basic step (however, this may increase the number of basic steps needed)

- Increases in R that are entirely caused by improvements in IC technology affect all aspects of the processor's operation equally except the time to access the main memory.

CISC and RISC

- Tradeoff between N and S
- A key consideration is the use of pipelining

➢S is close to 1 even though the number of basic steps per instruction may be considerably larger

➢It is much easier to implement efficient pipelining in processor with simple instruction sets

- Reduced Instruction Set Computers (RISC)
- Complex Instruction Set Computers (CISC)

Compiler

- A compiler translates a high-level language program into a sequence of machine instructions.
- To reduce N, we need a suitable machine instruction set and a compiler that makes good use of it.
- Goal – reduce N×S
- A compiler may not be designed for a specific processor; however, a high-quality compiler is usually designed for, and with, a specific processor.

Performance Measurement

- T is difficult to compute.
- Measure computer performance using benchmark programs.
- System Performance Evaluation Corporation (SPEC) selects and publishes representative application programs for different application domains, together with test results for many commercially available computers.
- Compile and run (no simulation)
- Reference computer

$$SPEC\ rating = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

$$SPEC\ rating = \left(\prod_{i=1}^{n} SPEC_i\right)^{\frac{1}{n}}$$

Problem:
Suppose a program (or a program task) takes 1 billion instructions to execute on a processor running at 2 GHz. Suppose also that 50% of the instructions execute in 3 clock cycles, 30% execute in 4 clock cycles, and 20% execute in 5 clock cycles. What is the execution time for the program or task?

- **Solution**
- We have the instruction count: 10^9 instructions. The clock time can be computed quickly from the clock rate to be 0.5×10^{-9} seconds. So we only need to to compute clocks per instruction as an effective value:
- Then we have execution time = $1.0\times10^9 \times 3.7 \times 0.5\times10^{-9}$ sec = 1.85 sec.

Value	Frequency	Product
3	0.5	1.5
4	0.3	1.2
5	0.2	1.0
CPI =		3.7

1.5. (a) Let $T_R = (N_R \times S_R) / R_R$ and $T_C = (N_C \times S_C) / R_C$ be execution times on the RISC and CISC processors, respectively. Equating execution times and clock rates, we have

$$1.2\,N_R = 1.5\,N_C$$

Then

$$N_C / N_R = 1.2 / 1.5 = 0.8$$

Therefore, the largest allowable value for N_C is 80% of N_R.

(b) In this case

$$1.2\,N_R / 1.15 = 1.5\,N_C / 1.00$$

Then

$$N_C / N_R = 1.2 / (1.15 \times 1.5) = 0.696$$

Therefore, the largest allowable value for N_C is 69.6% of N_R.

1.6. (a) Let cache access time be 1 and main memory access time be 20. Every instruction that is executed must be fetched from the cache, and an additional fetch from the main memory must be performed for 4% of these cache accesses. Therefore,

$$\text{Speedup factor} = \frac{1.0 \times 20}{(1.0 \times 1) + (0.04 \times 20)} = 11.1$$

(b)

$$\text{Speedup factor} = \frac{1.0 \times 20}{(1.0 \times 1) + (0.02 \times 20)} = 16.7$$

Consider a non-pipelined processor with a clock rate of 2.5 gigahertz and average cycles per instruction of four. The same processor is upgraded to a pipelined processor with five stages; but due to the internal pipeline delay, the clock speed is reduced to 2 gigahertz. Assume that there are no stalls in the pipeline. The speed up achieved in this pipelined processor is_____.

Formula:

Speed up = $\frac{Execution\ time(non\ pipeline)}{Execution\ time\ (pipeline)}$

Execution time = CPI × Cycle time (CPI is cycles per instructi

$$\text{Cycle time } = \frac{1}{\text{clock rate}}$$

Calculation:

Execution time for non-pipeline = $4 \times \frac{1}{2.5} = 1.6\ ns$

Execution time for pipeline = $1 \times \frac{1}{2} = 0.5\ ns$

Speed up = $1.60.5 = 3.2$

Addition/subtraction of signed numbers

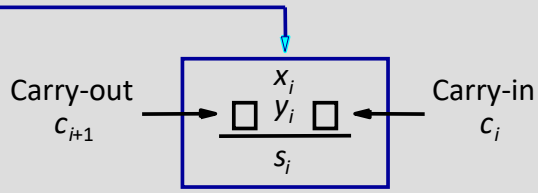
x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

E xample:

$$\begin{array}{rcl} X & 7 & 0\ 1\ 1\ 1 \\ +\ Y & +\ 6 & +\ {}_0 0\ {}_1 1\ {}_1 1\ {}_0 0 \\ \hline Z & 13 & 1\ 1\ 0\ 1 \end{array}$$



Legend for stage i

At the i^{th} stage:

Input:

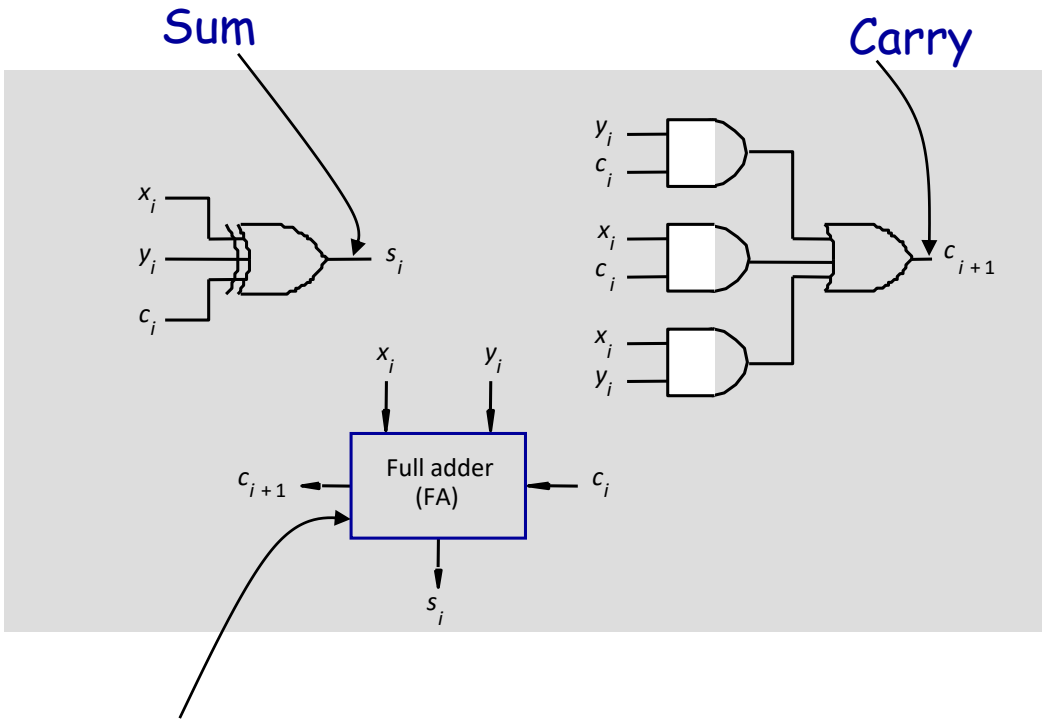
c_i is the carry-in

Output:

s_i is the sum

c_{i+1} carry-out to $(i+1)^{st}$ state

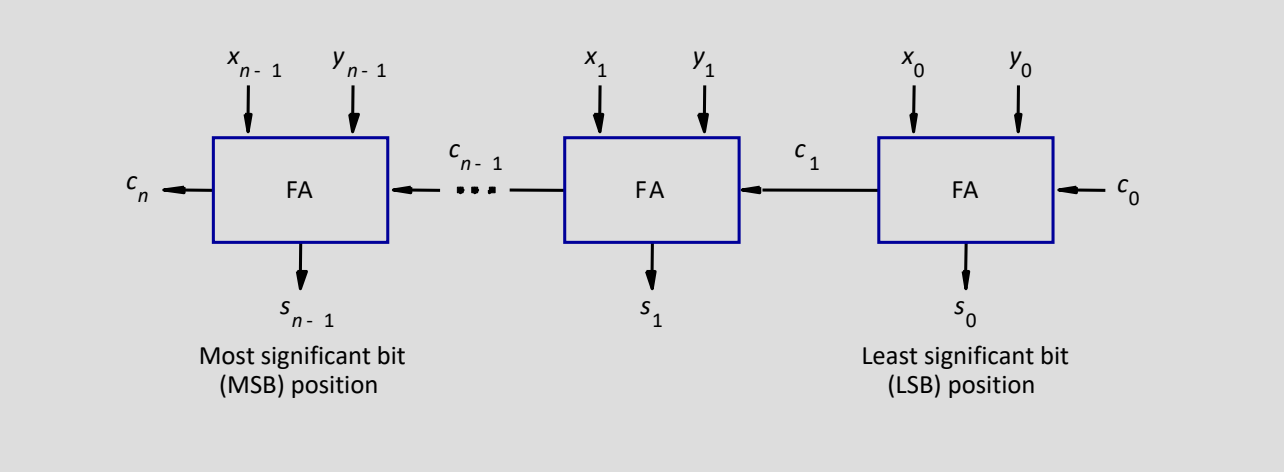
Addition logic for a single stage



Full Adder (FA): Symbol for the complete circuit for a single stage of addition.

n-bit adder

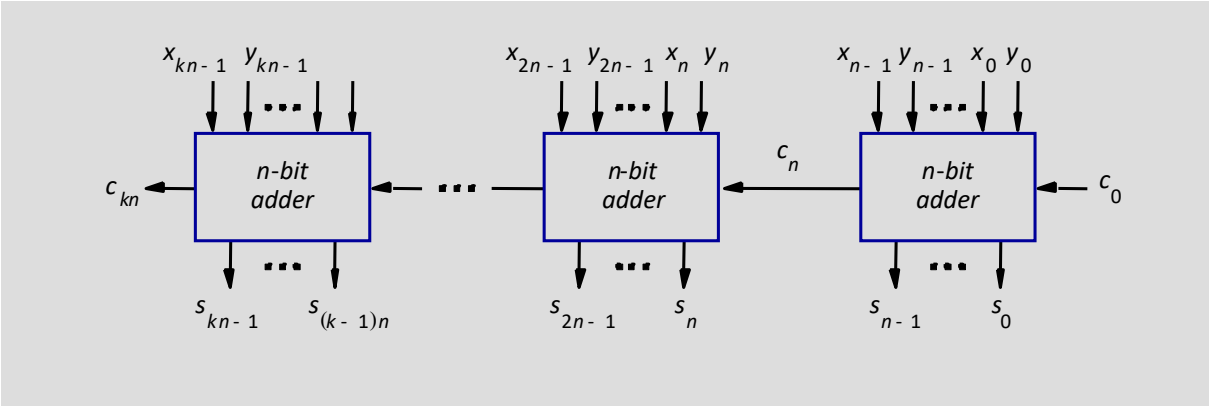
- Cascade *n* full adder (FA) blocks to form a *n*-bit adder.
- Carries propagate or ripple through this cascade, [*n*-bit ripple carry adder.]



Carry-in c_0 into the LSB position provides a convenient way to perform subtraction.

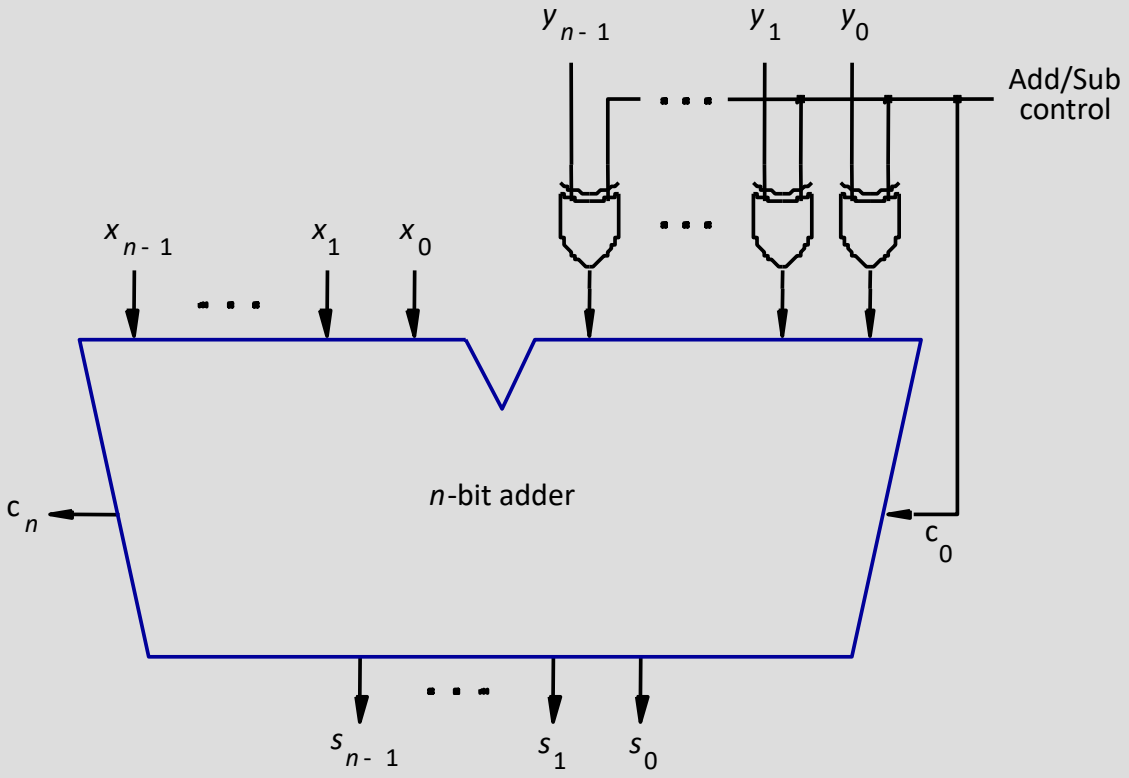
K n-bit adder

K n-bit numbers can be added by cascading k n-bit adders.



Each *n*-bit adder forms a block, so this is cascading of blocks.
Carries ripple or propagate through blocks, Blocked Ripple Carry Adder

n-bit adder/subtractor (contd..)



- Add/sub control = 0, addition.
- Add/sub control = 1, subtraction.

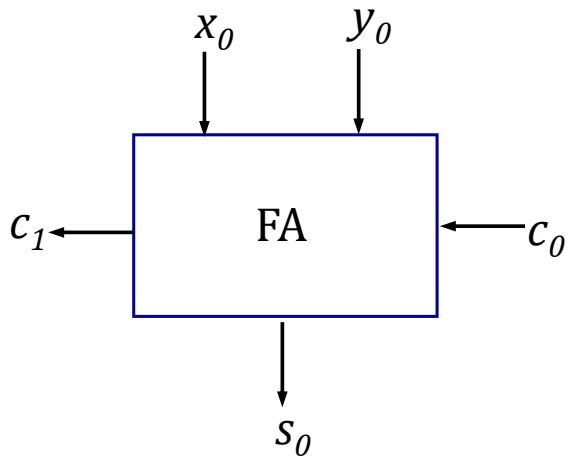
Detecting overflows

- Overflows can only occur when the sign of the two operands is the same.
- Overflow occurs if the sign of the result is different from the sign of the operands.
- Recall that the MSB represents the sign.
  - x_{n-1}, y_{n-1}, s_{n-1} represent the sign of operand *x,* operand *y* and result *s* respectively.
- Circuit to detect overflow can be implemented by the following logic expressions:

$$Overflow = x_{n-1} y_{n-1} \bar{s}_{n-1} + \bar{x}_{n-1} \bar{y}_{n-1} s_{n-1}$$

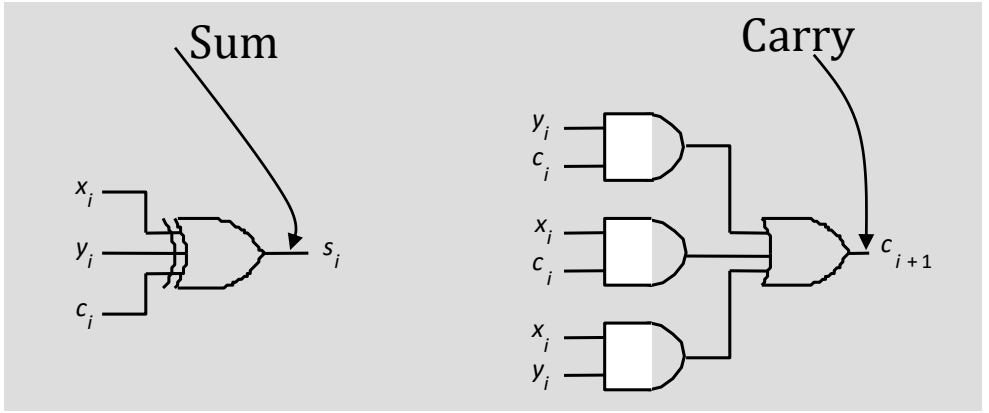
$$Overflow = c_n \oplus c_{n-1}$$

Computing the add time



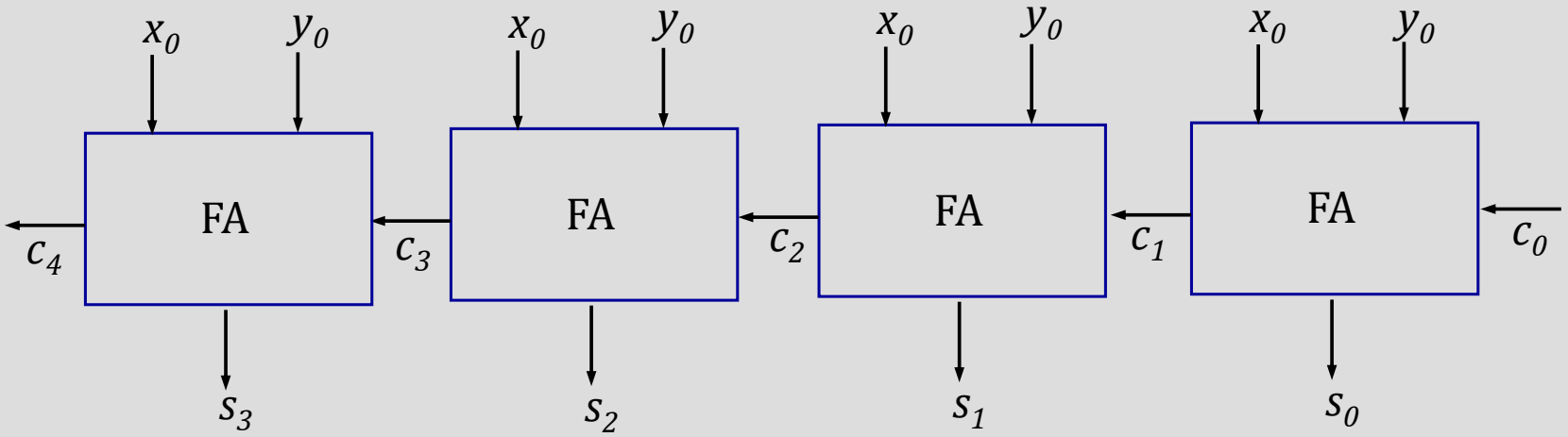
Consider 0^{th} stage:

- c_1 is available after 2 gate delays.
- s_1 is available after 1 gate delay.



Computing the add time (contd..)

Cascade of 4 Full Adders, or a 4-bit adder



- s_0 available after 1 gate delays, c_1 available after 2 gate delays.
- s_1 available after 3 gate delays, c_2 available after 4 gate delays.
- s_2 available after 5 gate delays, c_3 available after 6 gate delays.
- s_3 available after 7 gate delays, c_4 available after 8 gate delays.

For an n-bit adder, s_{n-1} is available after $2n-1$ gate delays
c_n is available after $2n$ gate delays.

Fast addition

Recall the equations:

$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Second equation can be written as:

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

We can write:

$$c_{i+1} = G_i + P_i c_i$$

$$where\ G_i = x_i y_i\ and\ P_i = x_i + y_i$$

- G_i is called generate function and P_i is called propagate function
- G_i and P_i are computed only from x_i and y_i and not c_i, thus they can be computed in one gate delay after X and Y are applied to the inputs of an n-bit adder.

Carry lookahead

$$c_{i+1} = G_i + P_i c_i$$

$$c_i = G_{i-1} + P_{i-1} c_{i-1}$$

$$\Rightarrow c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1} c_{i-1})$$

continuing

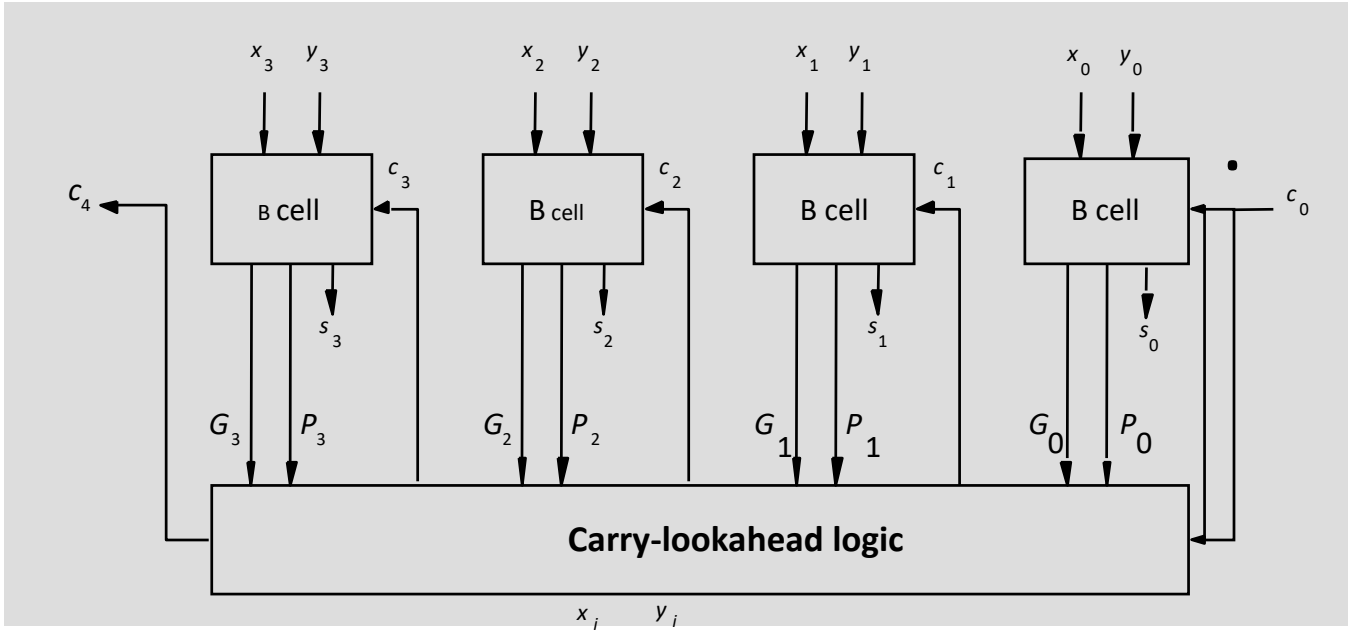
$$\Rightarrow c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1}(G_{i-2} + P_{i-2} c_{i-2}))$$

until

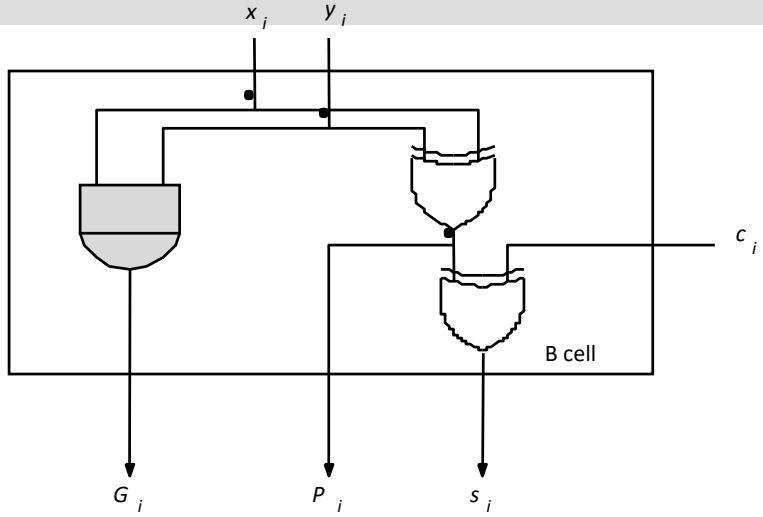
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + .. + P_i P_{i-1} .. P_1 G_0 + P_i P_{i-1} ... P_0 c_0$$

- All carries can be obtained 3 gate delays after X, Y and c_0 are applied.
  - -One gate delay for P_i and G_i
  - -Two gate delays in the AND-OR circuit for c_{i+1}
- All sums can be obtained 1 gate delay after the carries are computed.
- Independent of n, n-bit addition requires only 4 gate delays.
- This is called Carry Lookahead adder.

Carry-lookahead adder



4-bit carry-lookahead adder



B-cell for a single stage

Carry lookahead adder (contd..)

- Performing n-bit addition in 4 gate delays independent of n is good only theoretically because of fan-in constraints.

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + .. + P_i P_{i-1} .. P_1 G_0 + P_i P_{i-1} ... P_0 c_0$$

- Last AND gate and OR gate require a fan-in of (n+1) for a n-bit adder.
  - For a 4-bit adder ($n=4$) fan-in of 5 is required.
  - Practical limit for most gates.
- In order to add operands longer than 4 bits, we can cascade 4-bit Carry-Lookahead adders. Cascade of Carry-Lookahead adders is called Blocked Carry-Lookahead adder.

Blocked Carry-Lookahead adder

Carry-out from a 4-bit block can be given as:

$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

Rewrite this as:

$$P_0^I = P_3P_2P_1P_0$$

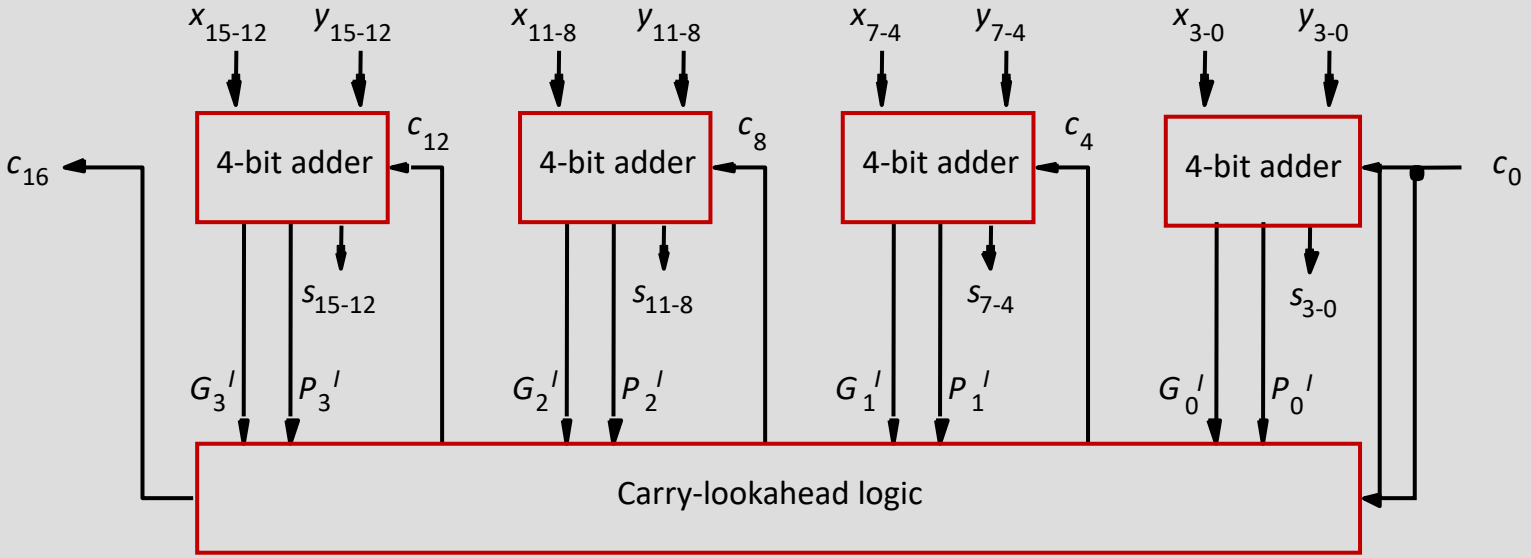
$$G_0^I = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

Subscript I denotes the blocked carry lookahead and identifies the block.

Cascade 4 *4*-bit adders, c_{16} can be expressed as:

$$c_{16} = G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^0 G_0^I + P_3^I P_2^I P_1^0 P_0^0 c_0$$

Blocked Carry-Lookahead adder



After x_i, y_i and c_0 are applied as inputs:

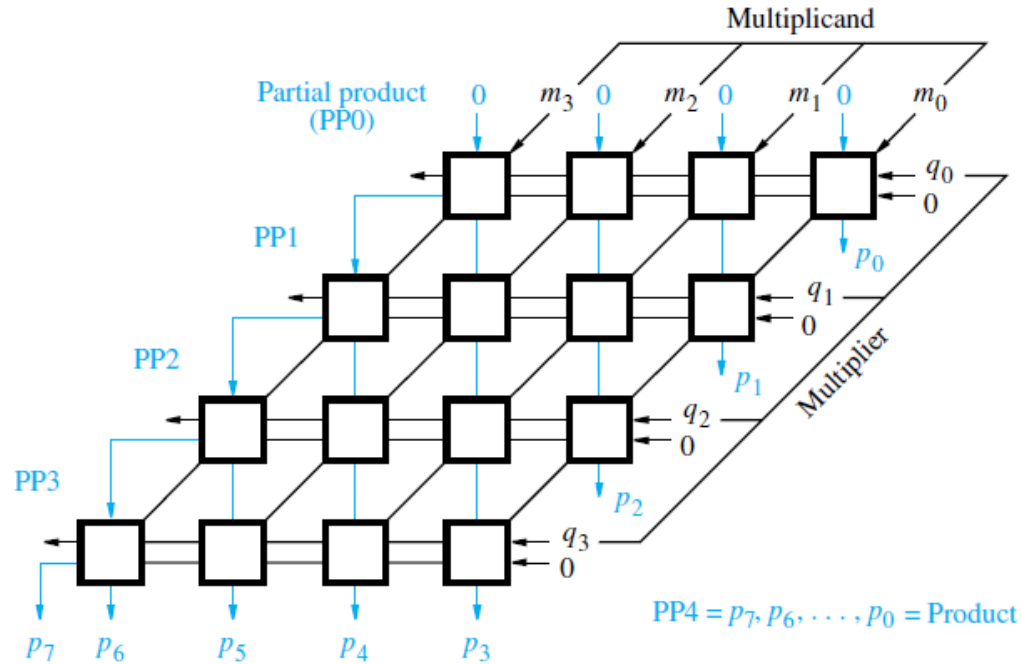
- G_i and P_i for each stage are available after 1 gate delay.
- P^I is available after 2 and G^I after 3 gate delays.
- All carries are available after 5 gate delays.
- c_{16} is available after 5 gate delays.
- s_{15} which depends on c_{12} is available after 8 (5+3)gate delays
  (Recall that for a 4-bit carry lookahead adder, the last sum bit is available 3 gate delays after all inputs are available)

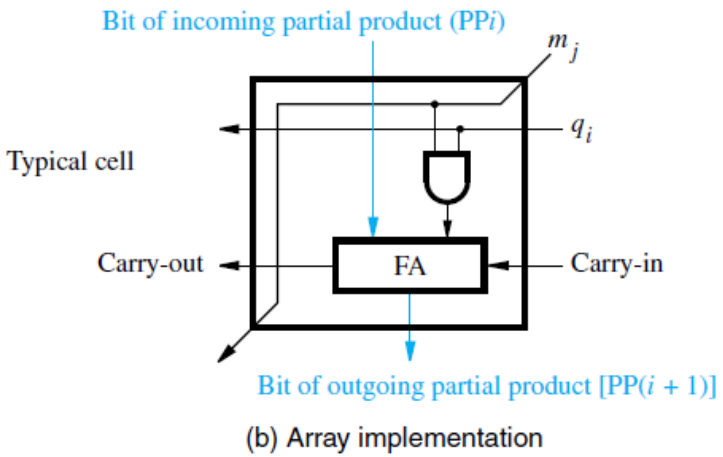
Multiplication of Unsigned Numbers

- The product of two, unsigned, *n*-digit numbers can be accommodated in 2*n* digits, so the product of the two 4-bit numbers in this example is accommodated in 8 bits, as shown.
- In the binary system, multiplication of the multiplicand by one bit of the multiplier is easy. If the multiplier bit is 1, the multiplicand is entered in the appropriate shifted position.
- If the multiplier bit is 0, then 0s are entered, as in the third row of the example.
- The product is computed one bit at a time by adding the bit columns from right to left and propagating carry values between columns.

			1	1	0	1		(13) Multiplicand M
		×	1	0	1	1		(11) Multiplier Q
			1	1	0	1		
		1	1	0	1			
	0	0	0	0				
1	1	0	1					
1	0	0	0	1	1	1	1	(143) Product P

(a) Manual multiplication algorithm





(b) Array implementation

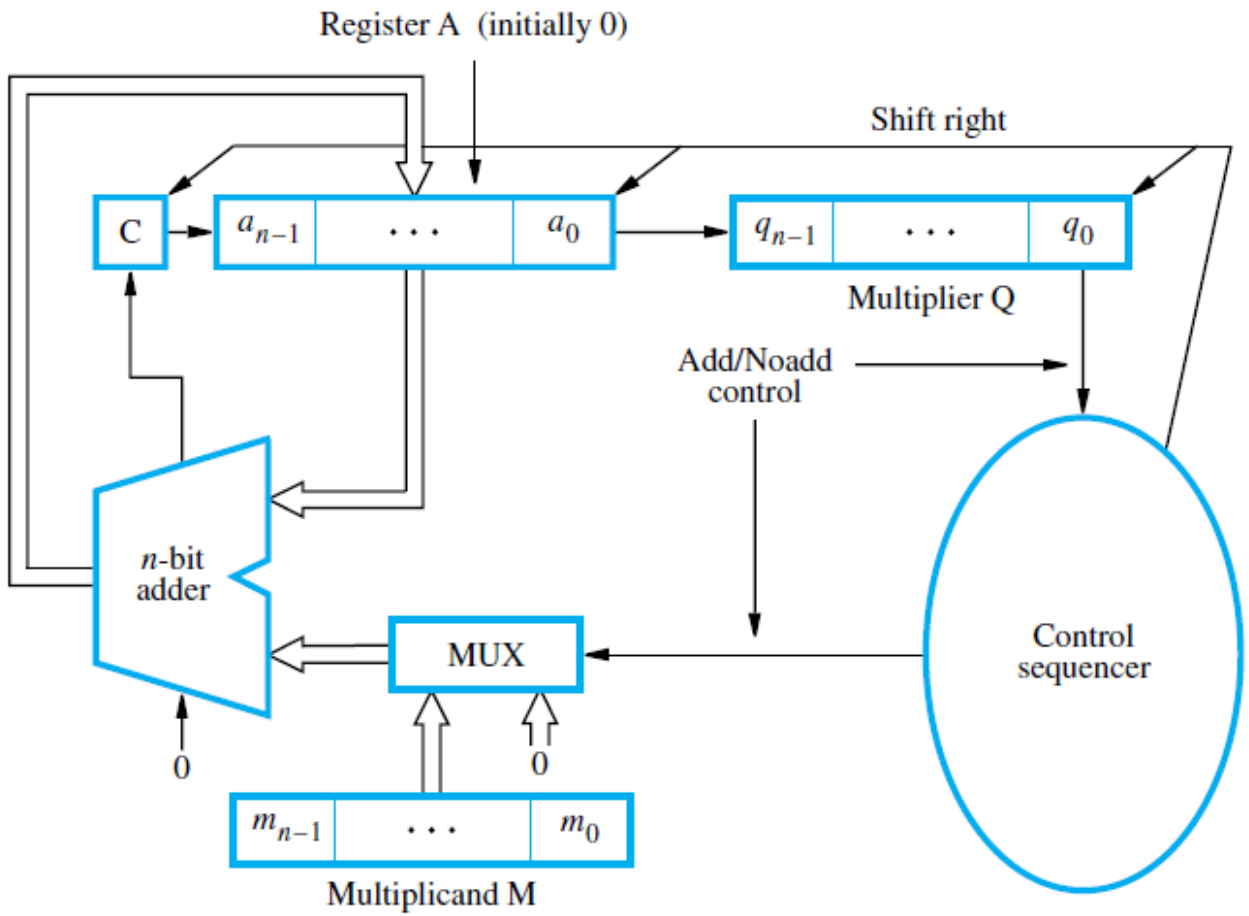
Array multiplication of unsigned binary operands.

Array Multiplier

- Binary multiplication of unsigned operands can be implemented in a combinational, two dimensional, logic array, as shown in Figure *b* for the 4-bit operand case.
- The main component in each cell is a full adder, FA.
- The AND gate in each cell determines whether a multiplicand bit, *mj* , is added to the incoming partial-product bit, based on the value of the multiplier bit, *qi* .
- Each row *i*, where $0 \leq i \leq 3$, adds the multiplicand (appropriately shifted) to the incoming partial product, PP*i*, to generate the outgoing partial product, PP($i + 1$), if $qi = 1$. If $qi = 0$, PP*i* is passed vertically downward unchanged.
- PP0 is all 0s, and PP4 is the desired product.
- The multiplicand is shifted left one position per row by the diagonal signal path.
- We note that the row-by-row addition done in the array circuit differs from the usual hand addition described previously, which is done column-by-column.

Sequential Circuit Multiplier

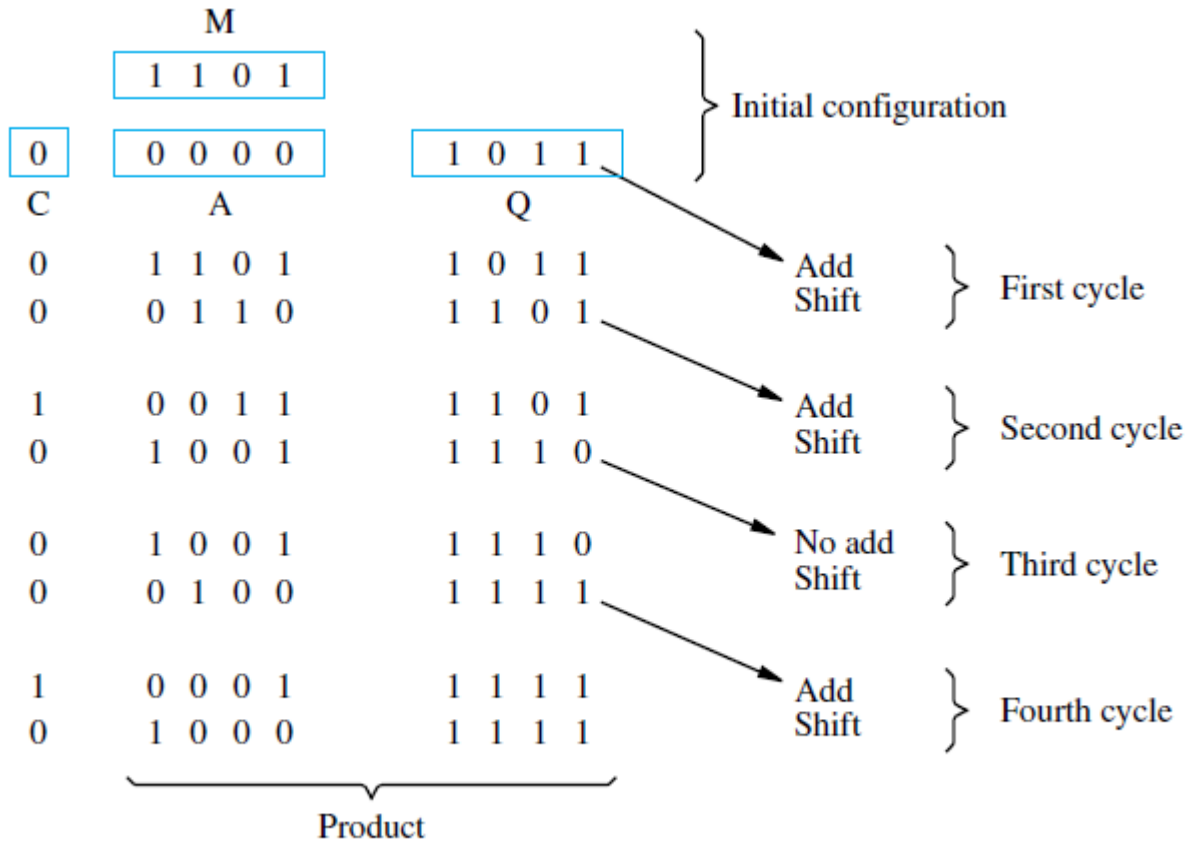
- The combinational array multiplier just described uses a large number of logic gates for multiplying numbers of practical size, such as 32- or 64-bit numbers.
- Multiplication of two *n*-bit numbers can also be performed in a sequential circuit that uses a single *n*-bit adder.
- The block diagram in next slide shows the hardware arrangement for sequential multiplication.
- This circuit performs multiplication by using a single *n*-bit adder *n* times to implement the spatial addition performed by the *n* rows of ripple-carry adders in Figure
- Registers A and Q are shift registers, concatenated as shown. Together, they hold partial product PP*i* while multiplier bit *qi* generates the signal Add/No add.



(a) Register configuration

- This signal causes the multiplexer MUX to select 0 when *qi* = 0, or to select the multiplicand M when *qi* = 1, to be added to PP*i* to generate PP(*i* + 1).
- The product is computed in *n* cycles.
- The partial product grows in length by one bit per cycle from the initial vector, PP0, of *n* 0s in register A.
- The carry-out from the adder is stored in flip-flop C, shown at the left end of register A.
- At the start, the multiplier is loaded into register Q, the multiplicand into register M, and C and A are cleared to 0.
- At the end of each cycle, C, A, and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q.
- Because of this shifting, multiplier bit *qi* appears at the LSB position of Q to generate the Add/Noadd signal at the correct time, starting with *q*0 during the first cycle, *q*1 during the second cycle, and so on.
- After they are used, the multiplier bits are discarded by the right-shift operation.
- Note that the carry-out from the adder is the leftmost bit of PP(*i* + 1), and it must be held in the C flip-flop to be shifted right with the contents of A and Q.
- After *n* cycles, the high-order half of the product is held in register A and the low-order half is in register Q.

Sequential circuit binary multiplier.



(b) Multiplication example

Multiplication of Signed Numbers

- We now discuss multiplication of 2’s-complement operands, generating a double-length product.
- The general strategy is still to accumulate partial products by adding versions of the multiplicand as selected by the multiplier bits.
- First, consider the case of a positive multiplier and a negative multiplicand.
- When we add a negative multiplicand to a partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend.
- Figure shows an example in which a 5-bit signed operand, −13, is the multiplicand.
- It is multiplied by +11 to get the 10-bit product, −143. The sign extension of the multiplicand is shown in blue.
- The hardware discussed earlier can be used for negative multiplicands if it is augmented to provide for sign extension of the partial products

Sign extension of negative multiplicand.

						1	0	0	1	1	(–13)
					×	0	1	0	1	1	(+11)
	1	1	1	1	1	1	0	0	1	1	
	1	1	1	1	1	0	0	1	1		
Sign extension is shown in blue	0	0	0	0	0	0	0	0			
	1	1	1	0	0	1	1				
	0	0	0	0	0	0					
	1	1	0	1	1	1	0	0	0	1	(–143)

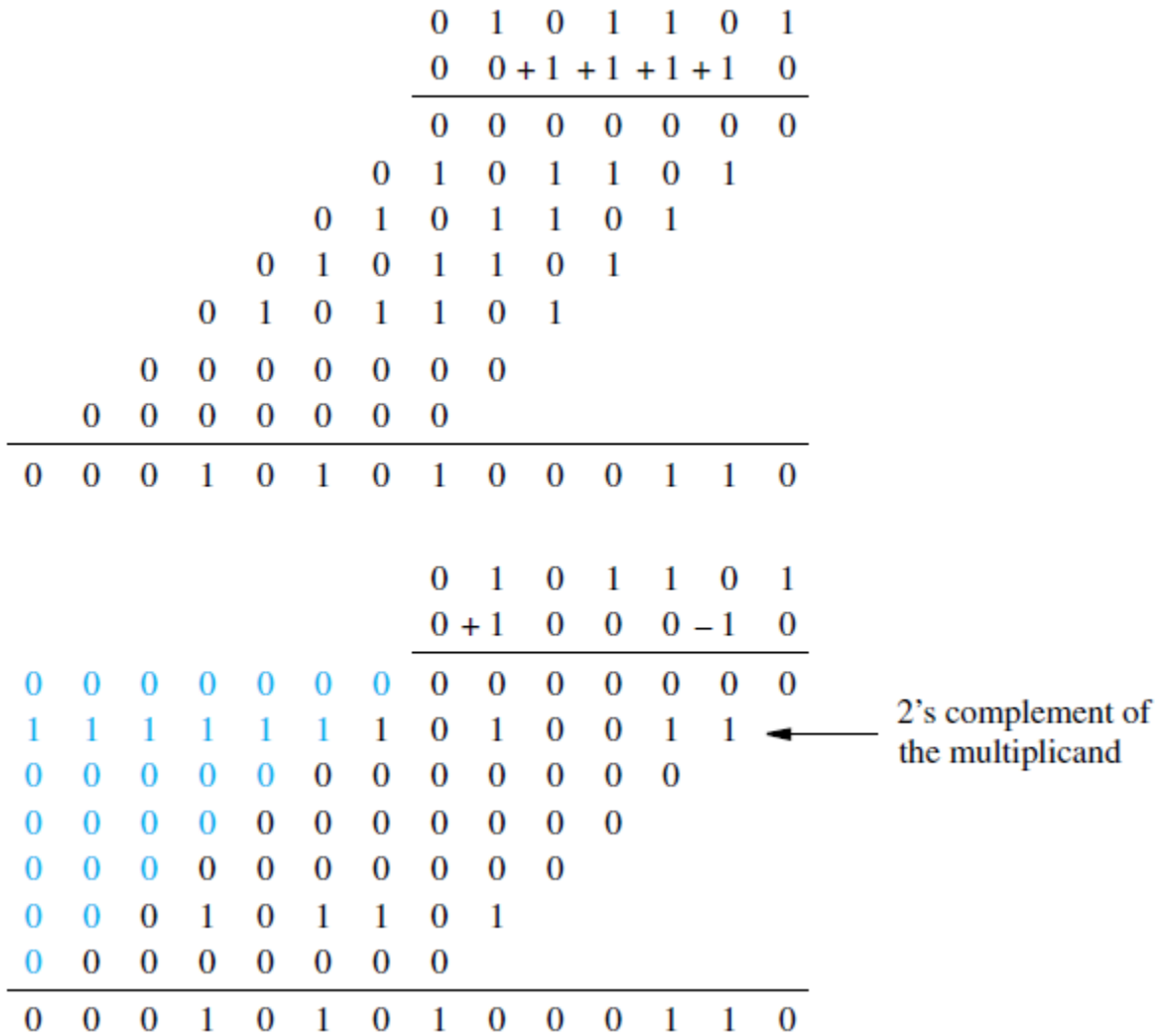
- For a negative multiplier, a straightforward solution is to form the 2’s-complement of
- both the multiplier and the multiplicand and proceed as in the case of a positive multiplier.
- This is possible because complementation of both operands does not change the value or
- the sign of the product. A technique that works equally well for both negative and positive
- multipliers, called the Booth algorithm, is described next.

The Booth Algorithm

- In general, in the Booth algorithm,
- −1 times the shifted multiplicand is selected when moving from 0 to 1, and
- +1 times the shifted multiplicand is selected when moving from1 to 0, as the multiplier is scanned from right to left.
- Ex1: 0011110 is coded as 0 +1 0 0 0 −1 0
- Ex2: 0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 1 0 0 is coded as

0 + 1 – 1 + 1 0 – 1 0 + 1 0 0 – 1 + 1 – 1 + 1 0 – 1 0 0

- Figure next slide illustrates the normal and the Booth algorithms for the example just discussed.
- The Booth algorithm clearly extends to any number of blocks of 1s in a multiplier, including the situation in which a single 1 is considered a block.



Normal and Booth multiplication schemes.

- The Booth algorithm can also be used directly for negative multipliers

as shown in Figure below.

- To demonstrate the correctness of the Booth algorithm for negative multipliers, we use the following property of negative-number representations in the 2's-complement system.