Computer Organization & Architecture
UNIT II

Assembly Language and Input/output Organization

Overview

- Assembly Language Concepts
- Stacks
- Subroutines
- Additional Instructions
- Accessing I/O Devices
- Interrupts
- Bus Structure
- Bus Operation
- Arbitration

1.Assembly Language Concepts

- Machine instructions are represented by patterns of 0s and 1s
- Symbolic names to represent the patterns are Load, Store, Add, and Branch
- While writing programs for a specific computer, such words are normally replaced by acronyms called *mnemonics*, such as LD, ST, ADD, and BR.
- A complete set of such symbolic names and rules for their use constitutes a programming language, referred to as an *assembly language*.

- The set of rules for using the mnemonics and for specification of complete instructions and programs is called the *syntax* of the language.
- Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an *assembler*
- The assembler program is a part of the system software of a computer and stored as a sequence of machine instructions in the memory of the computer.

- When the assembler program is executed, it reads the user program, analyzes it, and then generates the desired machine language program.
- The user program in its original alphanumeric text format is called a *source program*, and the assembled machine-language program is called an *object program*.

Ex: a Store instruction as ST R2, SUM

- mnemonic ST represents the binary pattern, or *operation (OP) code*, for the operation performed by the instruction.
- The assembler translates this mnemonic into the binary OP code that the computer recognizes.
- OP-code mnemonic is followed by at least one blank space or tab character
- Then the information that specifies the operands is given.
- source operand is register R2, followed by the specification of the destination operand, separated from the source operand by a comma

- Since there are several possible addressing modes for specifying operand locations, an assembly-language instruction must indicate which mode is being used
- ADD R2, R3, #5
- LD R3, (R2)

Assembler Directives

- Assembly language allows the programmer to specify information needed to translate the source program into the object program
- Equate statement
- TWENTY EQU 20
- informs the assembler that the name TWENTY should be replaced by the value 20 wherever it appears in the program.
- *assembler directives* (or *commands*), are used by the assembler while it translates a source program into an object program

Assembler Directives

In addition to translating the instructions of the source program, the **assembler must process statements called assembler directives.**

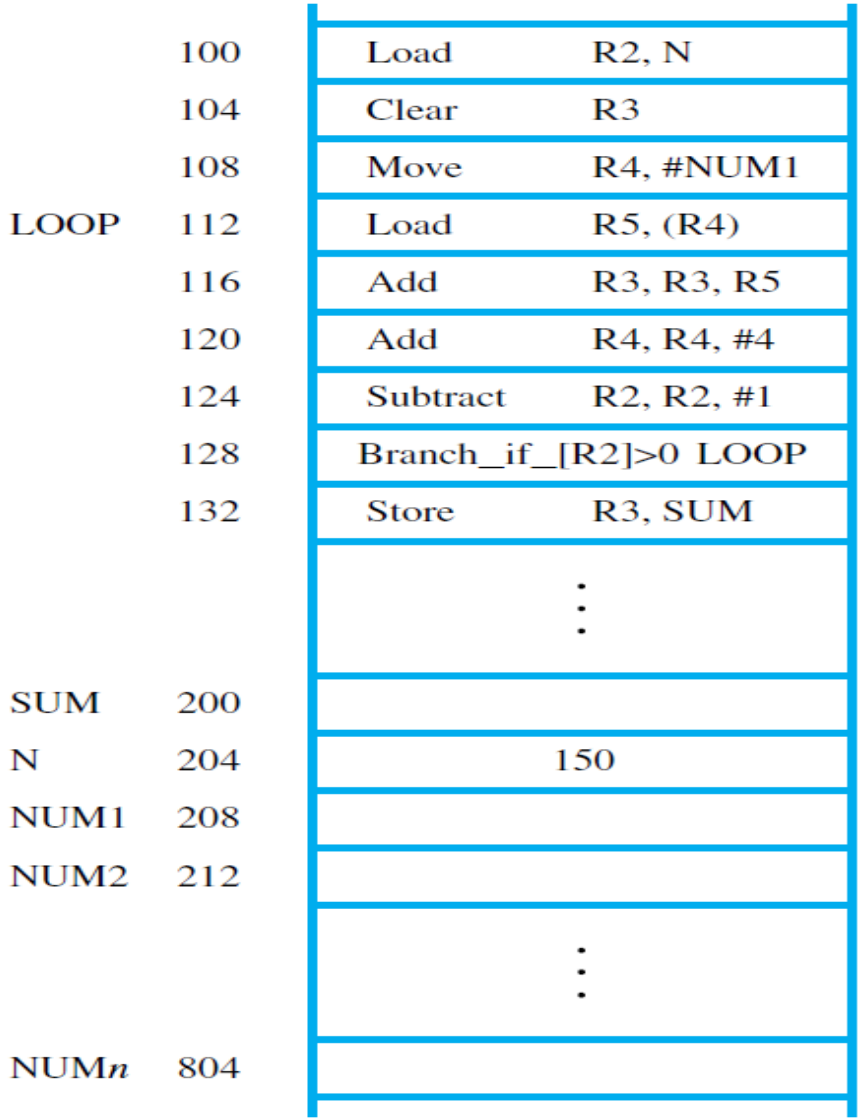
- These are pseudo instructions,
- They provide definition to the assembler itself.
- They are not translated into machine operation code.

Assembler Directives

START	Specify name and starting address for the program.
END	Indicate the end of the source program and (optionally) specify the first executable instruction in the program.
BYTE	Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.
WORD	Generate one-word integer constant.
RESB	Reserve the indicated number of bytes for a data area.
RESW	Reserve the indicated number of words for a data area.

- If the assembler is to produce an object program, it has to know
  - How to interpret the names
  - Where to place the instructions in the memory
  - Where to place the data operands in the memory
- To provide this information, the source program may be written as in Fig. b (next slide)

a. Memory arrangement



b. Assembly language representation

	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	LOOP:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, LOOP
		ST	R3, SUM
		next instruction	
Assembler directives		ORIGIN	200
	SUM:	RESERVE	4
	N:	DATAWORD	150
	NUM1:	RESERVE	600
		END	

- Assembler directive, ORIGIN, tells the assembler program where in the memory to place the instructions that follow
- Instructions of the object program are to be loaded in the memory starting at address 100
- second ORIGIN directive tells the assembler program where in the memory to place the data block that follows.
- In this case, the location specified has the address 200.
- This is intended to be the location in which the final sum will be stored.

- A 4-byte space for the sum is reserved by means of the assembler directive RESERVE.
- The next word, at address 204, has to contain the value 150 which is the number of entries in the list
  - The DATAWORD directive is used to inform the assembler of this requirement.
- The next RESERVE directive declares that a memory block of 600 bytes is to be reserved for data.
  - This directive does not cause any data to be loaded in these locations
- The last statement in the source program is the assembler directive END, which tells the assembler that this is the end of the source program text.

- Any statement that results in instructions or data being placed in a memory location may be given a memory address label.
- The assembler automatically assigns the address of that location to the label.
- For example, in the data block that follows the second ORIGIN directive, we used the labels SUM, N, and NUM1.
- Because the first RESERVE statement after the ORIGIN directive is given the label SUM, the name SUM is assigned the value 200.
- Whenever SUM is encountered in the program, it will be replaced with this value.
- Using SUM as a label in this manner is equivalent to using the assembler directive
- SUM EQU 200

- Most assembly languages require statements in a source program to be written in the form
- Label: Operation Operand(s) Comment
- These four *fields* are separated by an appropriate delimiter, perhaps one or more blank or tab characters.
- The Label is an optional name associated with the memory address where the machine-language instruction produced from the statement will be loaded.
- Labels may also be associated with addresses of data items.
- In Figure (slide 12) there are four labels: LOOP, SUM, N, and NUM1.

- The Operation field contains an assembler directive or the OP-code mnemonic of the desired instruction.
- The Operand field contains addressing information for accessing the operands.
- The Comment field is ignored by the assembler program.
- It is used for documentation purposes to make the program easier to understand.
- Assembly languages differ in detail and complexity from one computer to another.

Assembly and Execution of Programs

- A source program written in an assembly language must be assembled into a machine language object program before it can be executed.
- Assembler program replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.
- A key part of the assembly process is determining the values that replace the names (EQU directive)
- The assembler must keep track of addresses as it generates the machine code for successive instructions.

- In some cases, the assembler does not directly replace a name representing an address with the actual value of this address.
  - A branch instruction is implemented in machine code by specifying the branch target as the distance (in bytes) from the present address in the Program Counter to the target instruction.
- The assembler stores the object program on the secondary storage device available in the computer.
- The object program must be loaded into the main memory before it is executed.
  - For this, a utility program called a *loader* must already be in the memory.

- Having loaded the object code, the loader starts execution of the object program by branching to the first instruction to be executed, which may be identified by an address label such as START.
  - The assembler places that address in the header of the object code for the loader to use at execution time.
- When the object program begins executing, it proceeds to completion unless there are logical errors in the program.
- The assembler can detect and report syntax errors, the system software usually includes a *debugger* program.

Number Notation

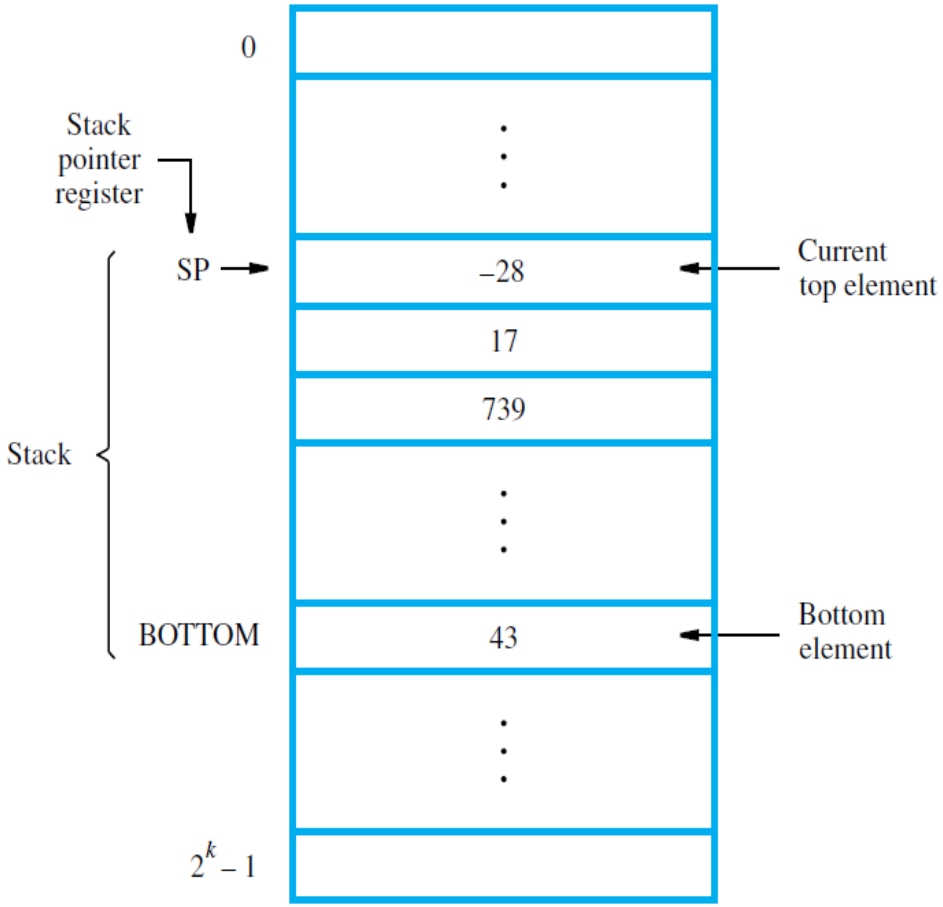
- Most assemblers allow numerical values to be specified in different ways, using conventions that are defined by the assembly-language syntax.
- ADDI R2, R3, 93
- ADDI R2, R3, %01011101
  - Percent: assembler-specific prefix symbol for binary
- Binary numbers can be written more compactly as *hexadecimal*(93 becomes 5D)
- hex representation is identified by the prefix 0x
  - ADDI R2, R3, 0x5D

2.Stacks

- A *stack* is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only.
- This end is called the top of the stack, and the other end is called the bottom.
- The structure is sometimes referred to as **a *pushdown* stack**.
- ***last-in–first-out* (LIFO) stack**, is a type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins.
- The terms ***push*** and ***pop*** are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

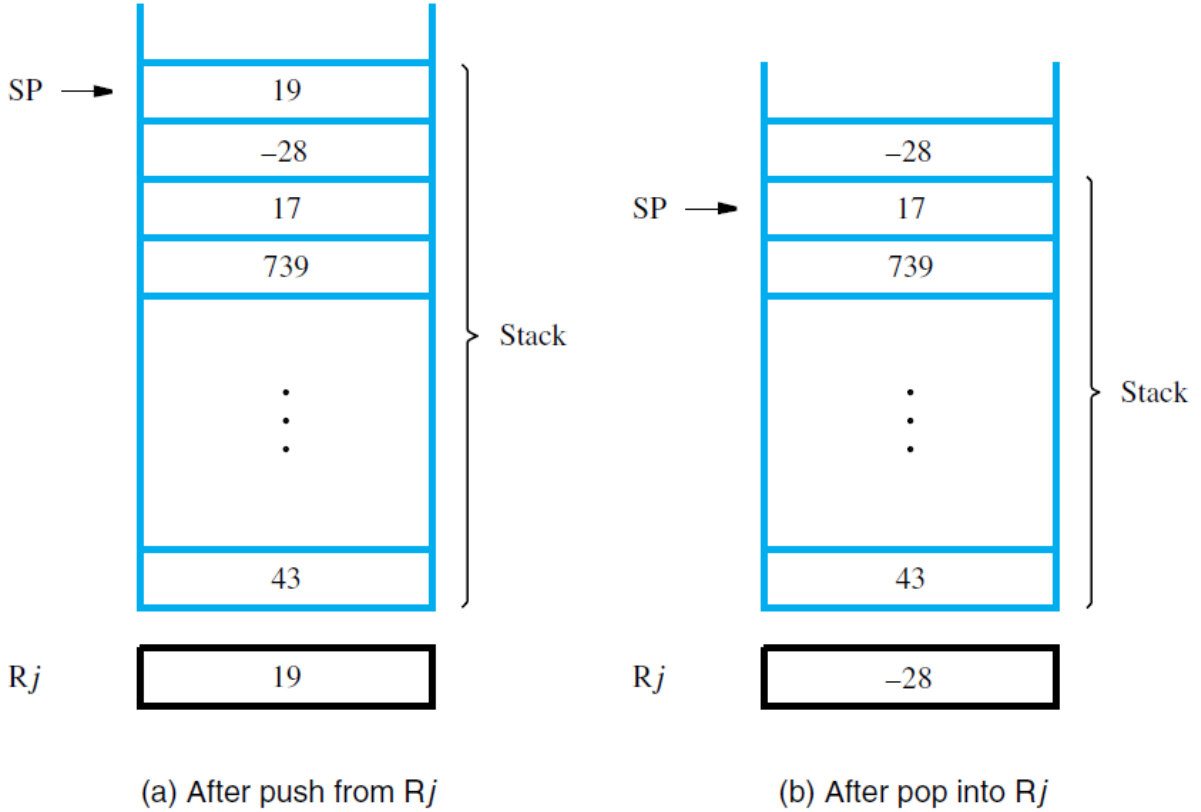
- processor register, called the ***stack pointer* (SP)**, is used to point to a particular stack structure called the *processor stack*

- Data can be stored in a stack with successive elements occupying successive memory locations.



- 43 at the bottom and −28 at the top
- stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time.

- push operation
  - Subtract SP, SP, #4
  - Store R*j*, (SP)
- new item to be pushed on the stack is in processor register R*j*, the Store instruction will place this value on the stack
- copy the word from R*j* onto the top of the stack, decrementing the stack pointer by 4 before the store (push) operation.
- pop operation
  - Load R*j*, (SP)
  - Add SP, SP, #4
- These two instructions load (pop) the top value from the stack into register R*j* and then increment the stack pointer by 4 so that it points to the new top element


SP
19
–28
17
739
Stack
43
Rj
19
(a) After push from Rj
SP
–28
17
739
Stack
43
Rj
–28
(b) After pop into Rj

3.Subroutines

- block of instructions that is executed each time the task has to be performed.
- only one copy of this block is placed in the memory
- any program that requires the use of the subroutine simply branches to its starting location by using **Call instruction** (calling the subroutine)
- Subroutine resume execution
- After completing all instructions of subroutine **Return instruction** is executed to return back to the calling program

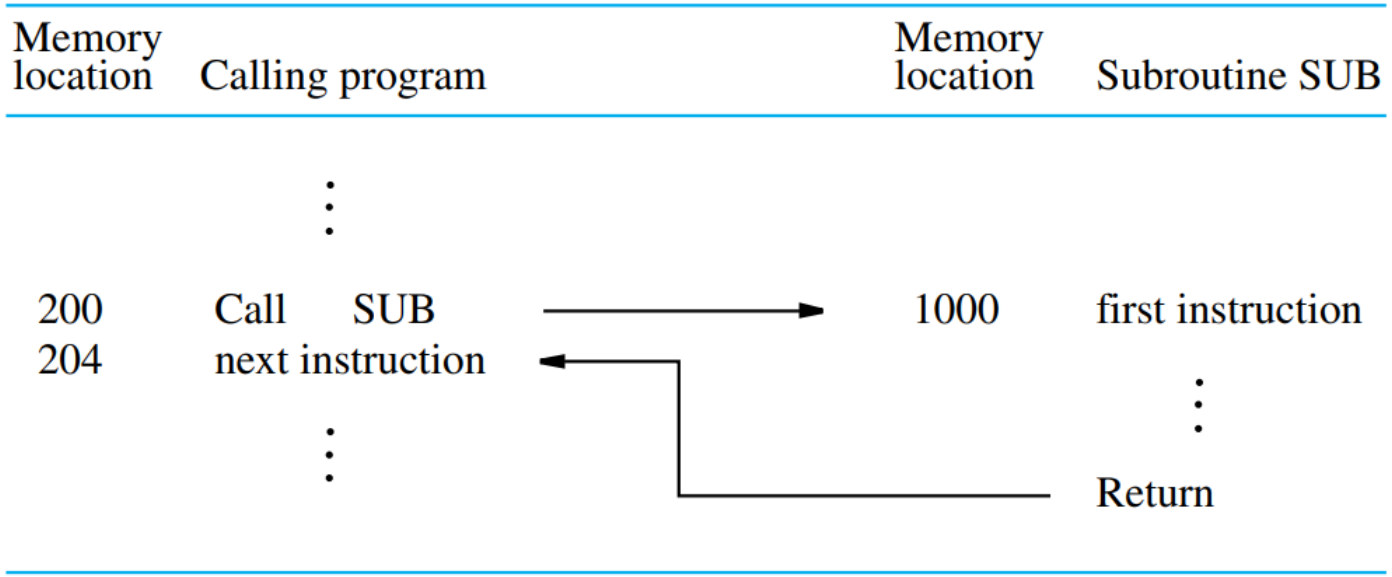
- **program counter (PC)** get updated during Call and Return instructions
- The way in which a computer makes it possible to call and return from subroutines is referred to as its **subroutine linkage method.**
- subroutine linkage method save the return address in a specific location - called the **link register.**
- When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register

The Call instruction is just a special branch instruction that performs the following operations:

- Store the contents of the PC in the link register
- Branch to the target address specified by the Call instruction

The Return instruction is a special branch instruction that performs the operation

- Branch to the address contained in the link register



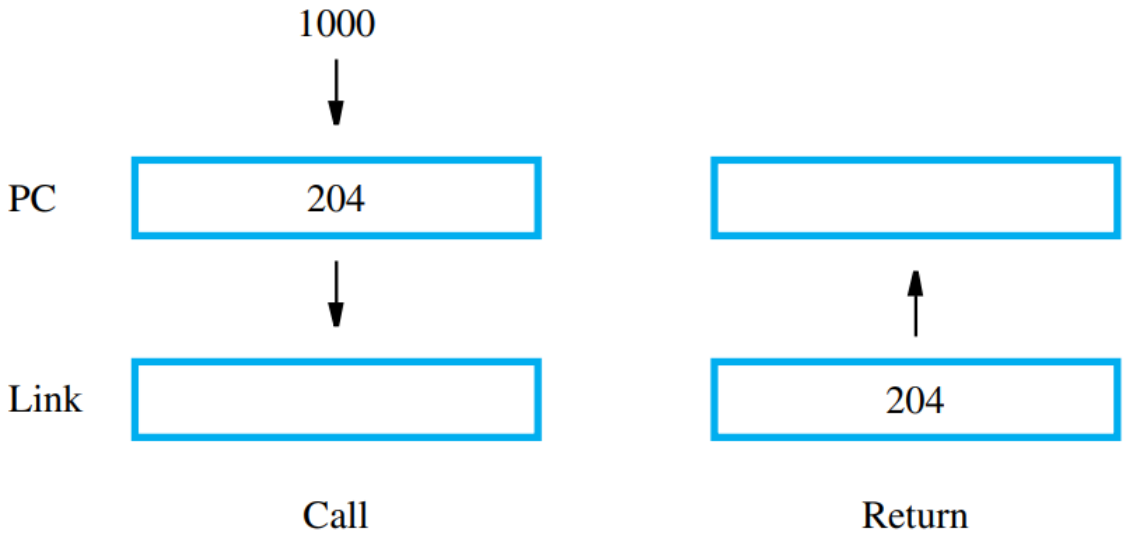


Figure 2.16 Subroutine linkage using a link register.

Subroutine Nesting and the Processor Stack

- A programming practice, called subroutine nesting, is to have one subroutine call another
- the return address of the second call is also stored in the link register, overwriting its previous contents.
- Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost.
- Subroutine nesting can be carried out to any depth

- Eventually, the last subroutine called completes its computations and returns to the subroutine that called it.
- The return address needed for this first return is the last one generated in the nested call sequence.
- return addresses are generated and used in a last-in–first-out order and should be pushed onto the processor stack.

Subroutine - Parameter Passing

- Subroutine Calls pass the operands or their addresses, to be used in the computation at subroutines
- subroutine returns other parameters, which are the results of the computation
- This exchange of information between a calling program and a subroutine is referred to as ***parameter passing***
- The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine.

Program for adding a list of numbers can be implemented as a subroutine, LISTADD, with the parameters passed through registers.

Calling program

```
                Load            R2, N           Parameter 1 is list size.
                Move            R4, #NUM1       Parameter 2 is list location.
                Call            LISTADD         Call subroutine.
                Store           R3, SUM         Save result.
                :
```

Subroutine

```
LISTADD:        Subtract        SP, SP, #4      Save the contents of
                Store           R5, (SP)          R5 on the stack.
                Clear           R3              Initialize sum to 0.
LOOP:           Load            R5, (R4)        Get the next number.
                Add             R3, R3, R5      Add this number to sum.
                Add             R4, R4, #4      Increment the pointer by 4.
                Subtract        R2, R2, #1      Decrement the counter.
                Branch_if_[R2]>0 LOOP
                Load            R5, (SP)        Restore the contents of R5.
                Add             SP, SP, #4
                Return                          Return to calling program.
```

- If many parameters are involved, there may not be enough general-purpose registers available for passing them to the subroutine.

- The processor stack provides a convenient and flexible mechanism for passing an arbitrary number of parameters.

Assume top of stack is at level 1 in Figure 2.19.

Stack Trace for program adding n numbers written as a subroutine;
parameters passed on the stack

```
Move      R2, #NUM1   Push parameters onto stack.
Subtract  SP, SP, #4
Store     R2, (SP)
Load      R2, N
Subtract  SP, SP, #4
Store     R2, (SP)
Call      LISTADD     Call subroutine
                        (top of stack is at level 2).
Load      R2, 4(SP)   Get the result from the stack
Store     R2, SUM       and save it in SUM.
Add       SP, SP, #8  Restore top of stack
                        (top of stack is at level 1).
```

```
LISTADD:    Subtract          SP, SP, #16     Save registers
            Store             R2, 12(SP)
            Store             R3, 8(SP)
            Store             R4, 4(SP)
            Store             R5, (SP)        (top of stack is at level 3).
            Load              R2, 16(SP)      Initialize counter to n.
            Load              R4, 20(SP)      Initialize pointer to the list.
            Clear             R3              Initialize sum to 0.
LOOP:       Load              R5, (R4)        Get the next number.
            Add               R3, R3, R5      Add this number to sum.
            Add               R4, R4, #4      Increment the pointer by 4.
            Subtract          R2, R2, #1      Decrement the counter.
            Branch_if_[R2]>0  LOOP
            Store             R3, 20(SP)      Put result in the stack.
            Load              R5, (SP)        Restore registers.
            Load              R4, 4(SP)
            Load              R3, 8(SP)
            Load              R2, 12(SP)
            Add               SP, SP, #16     (top of stack is at level 2).
            Return                            Return to calling program.
```

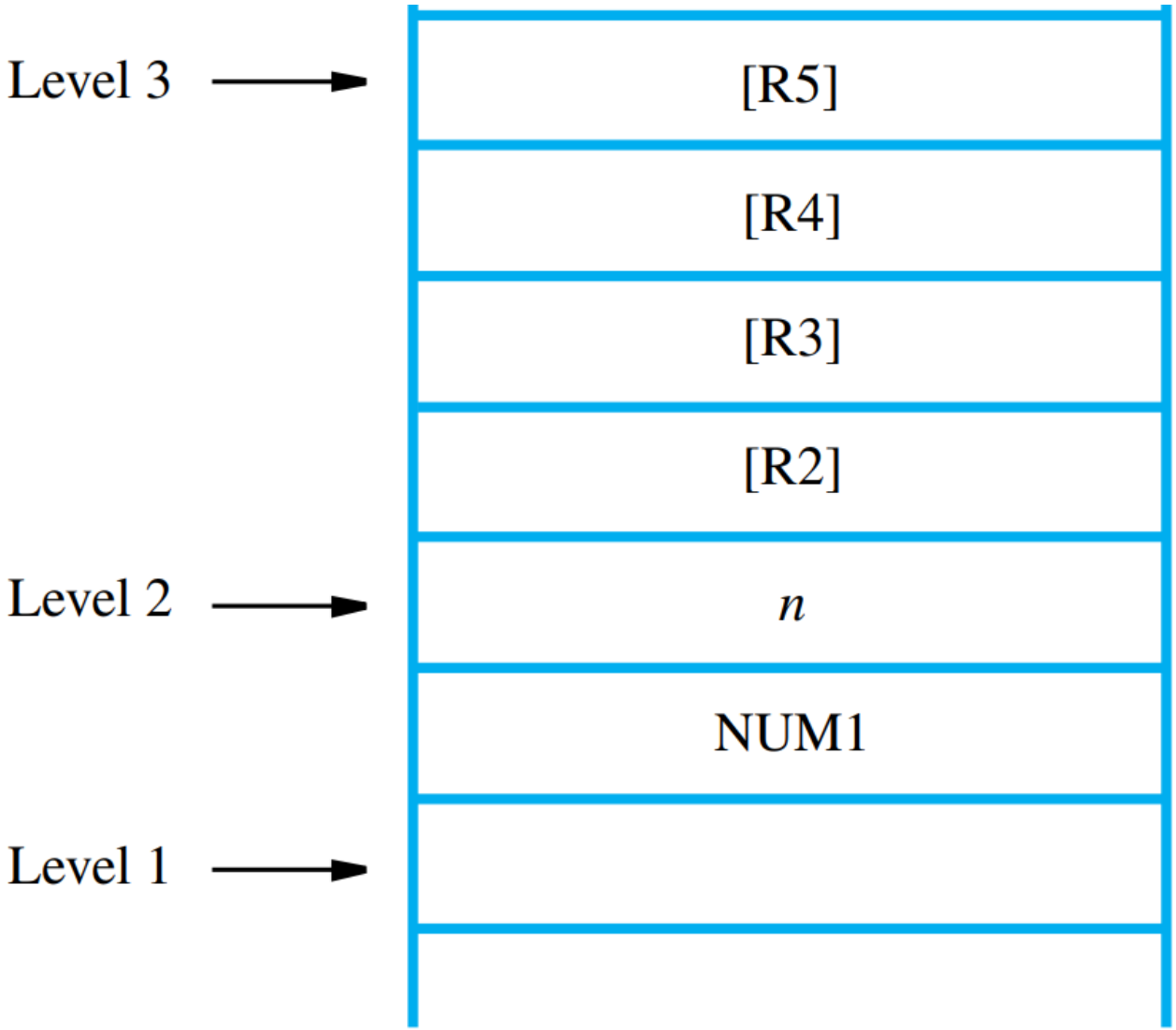


Figure 2.19 Stack contents for the program in Figure 2.18.

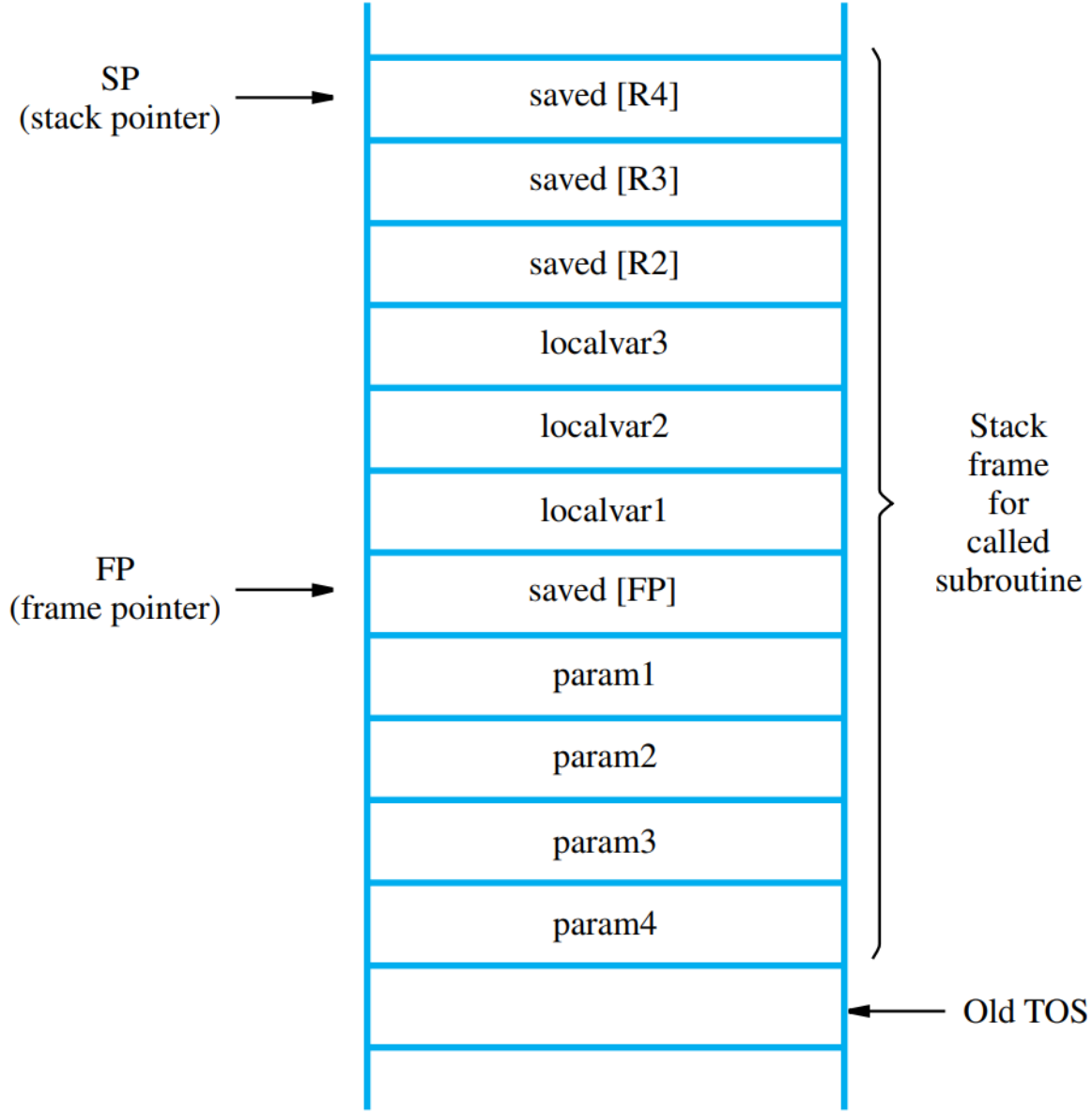


Figure 2.20 A subroutine stack frame example.

4.Additional Instructions

- Load, Store, Move, Clear, Add, Subtract, Branch, Call, and Return

Logic Instructions

- AND, OR, and NOT, applied to individual bits of a word or byte independently and in parallel

AND operation:

Syntax: AND SRC,DEST

Ex. AND R1,R2

Usage : Can clear all bits of operand except some specified field

- building blocks of digital circuits
    - **And R4, R2, R3**
- computes the bit-wise AND of operands in registers R2 and R3, and leaves the result in R4

And R4, R2, #Value

where Value is a 16-bit logic value that is extended to 32 bits by placing zeros into the 16 most-significant bit positions.

Application for logic instruction :

- Suppose that four ASCII characters are contained in the 32-bit register R2.
- In some task, we wish to determine if the rightmost character is Z.
- If it is, then a conditional branch to FOUNDZ is to be made

```
And                  R2, R2, #0xFF
Move                 R3, #0x5A
Branch_if_[R2]=[R3]  FOUNDZ
```

0xff
= 11111111in binary
= 255 in decimal

Char Z = Ascii 0x5A in hexa decimal
= 01011010

- The And instruction clears all bits in the leftmost three character positions of R2 to zero,

- leaving the rightmost character unchanged.
- immediate operand that has eight 1s at its right end, and 0s in the 24 bits to the left.

- The Move instruction loads the hex value 5A into R3.

- Since both R2 and R3 have 0s in the leftmost 24 bits, the Branch instruction compares the remaining character at the right end of R2 with the binary representation for the character Z, and causes a branch to FOUNDZ if there is a match.

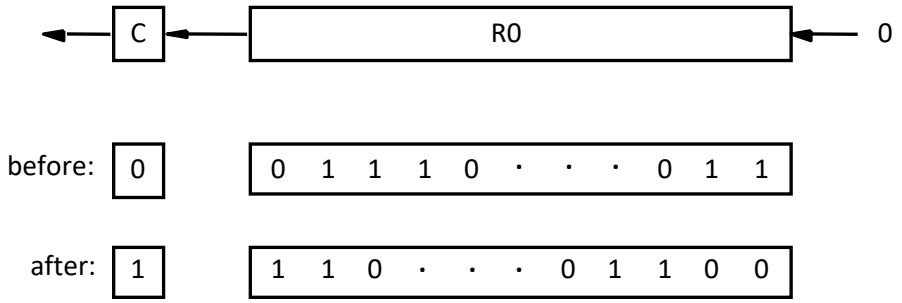
Shift and Rotate Instructions

- There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions.
- For general operands, we use a logical shift.
- For a signed number, we use an arithmetic shift, which preserves the sign of the number.

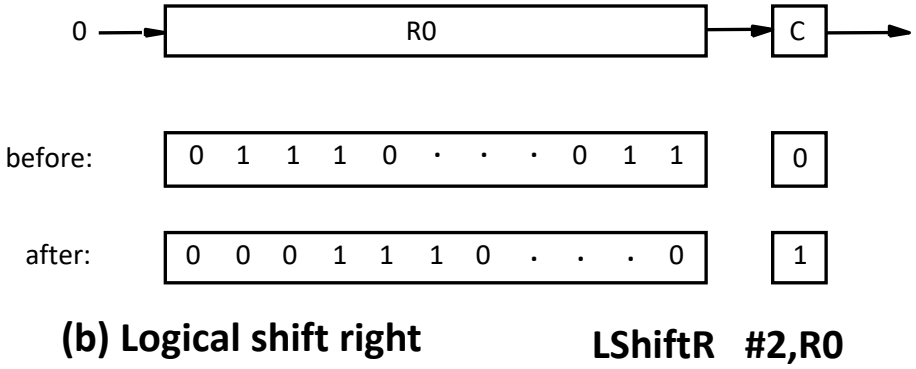
Logical Shifts

- 2 Logical shifts – shifting left (LShiftL) and shifting right (LShiftR)

LShiftL Ri, Rj, count



(a) Logical shift left LShiftL #2,R0

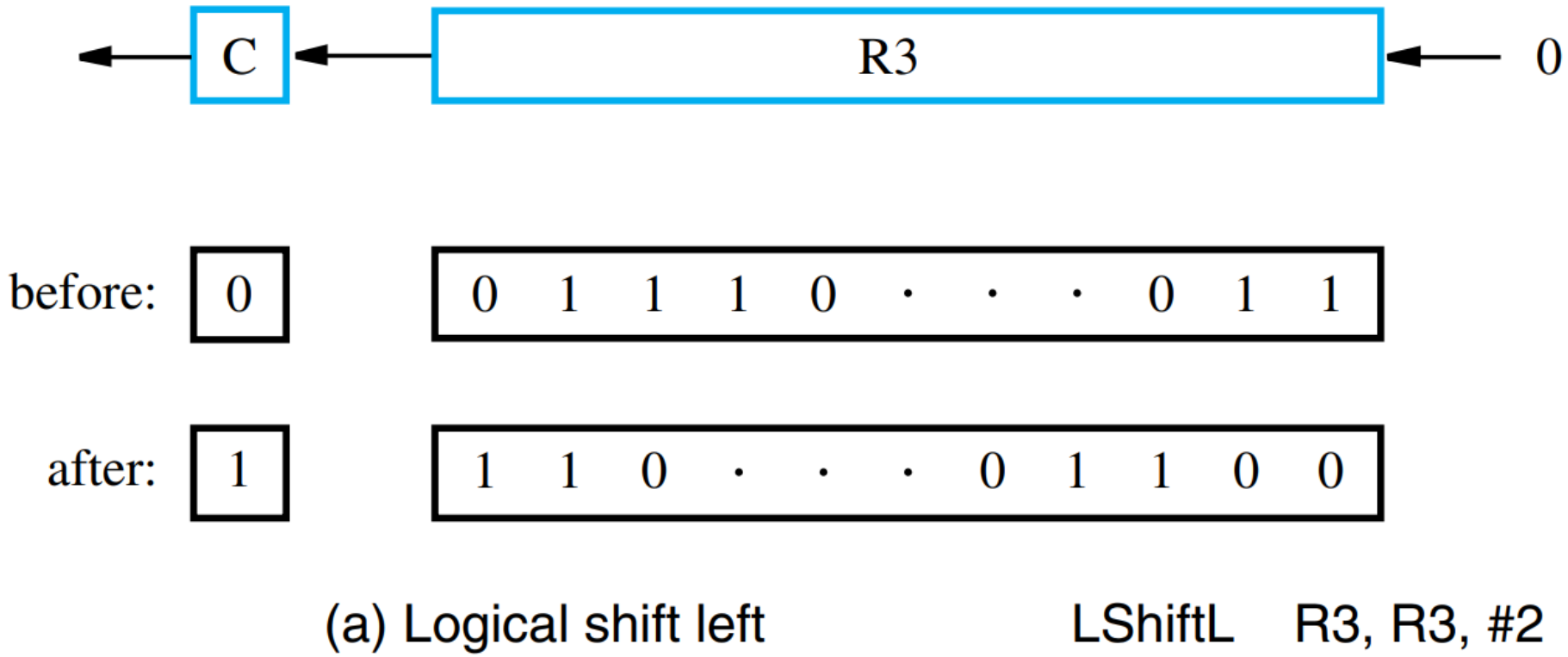


(b) Logical shift right LShiftR #2,R0

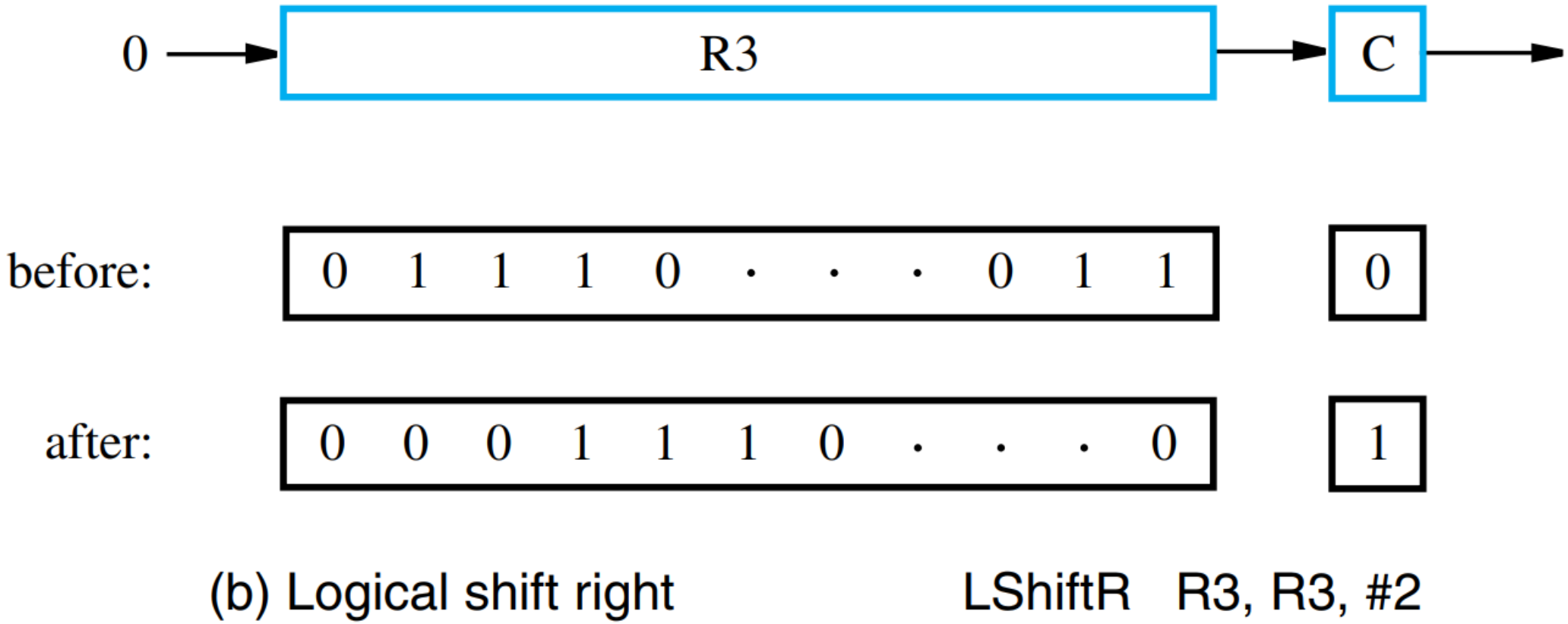
LShiftR count,dst

LShiftL Ri, Rj, count

shifts the contents of register Rj left by a number of bit positions given by the count operand, and places the result in register Ri, without changing the contents of Rj.



(a) Logical shift left LShiftL R3, R3, #2



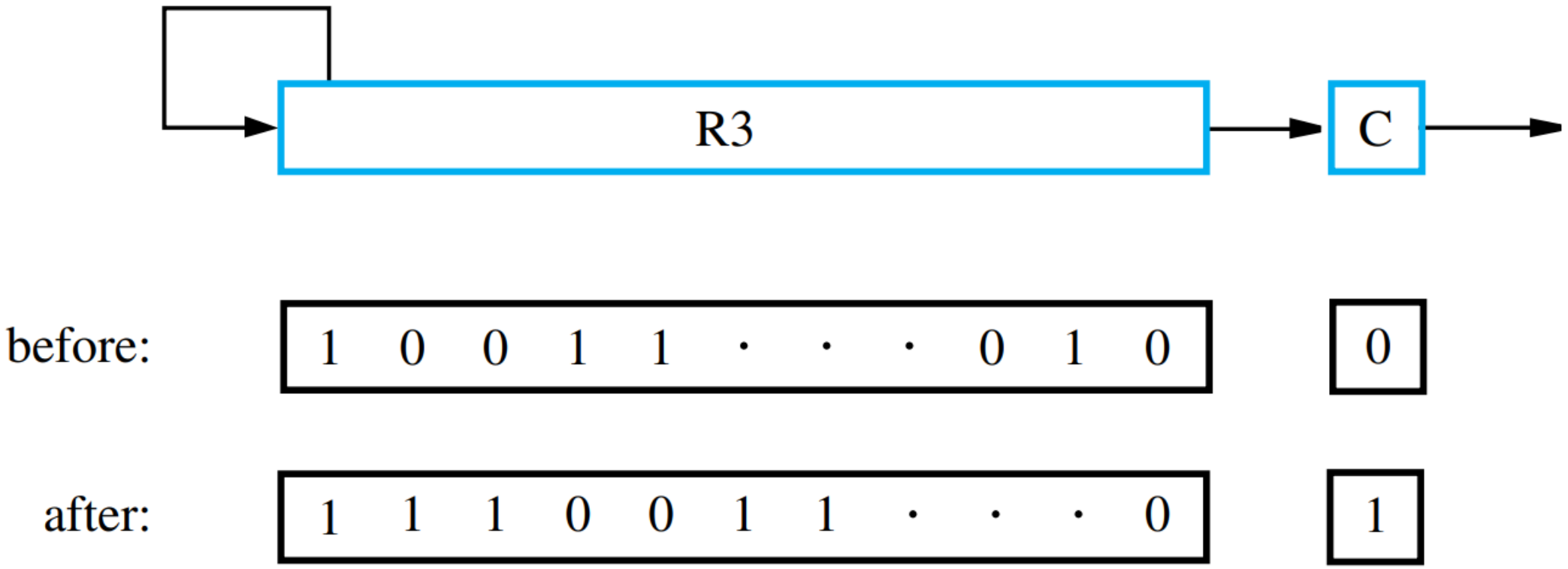
(b) Logical shift right LShiftR R3, R3, #2

Arithmetic Shifts

the bit pattern being shifted is interpreted as a signed number.

shifting a number one bit position to the left is equivalent to multiplying it by 2

shifting it to the right is equivalent to dividing it by 2



(c) Arithmetic shift right AShiftR R3, R3, #2

Not

Syntax: **Not destination**

It complements all bits contained in the destination operand, changing zero's to 1's and 1's to zero's

Ex. **Not R0 [R0 = 8] R0 = 1000**

outcome : R0 = 0 1 1 1 [R0 = 7]

For 2's Complement of R0

Not R0 & Add #1,R0

Some computers -> Negate R0

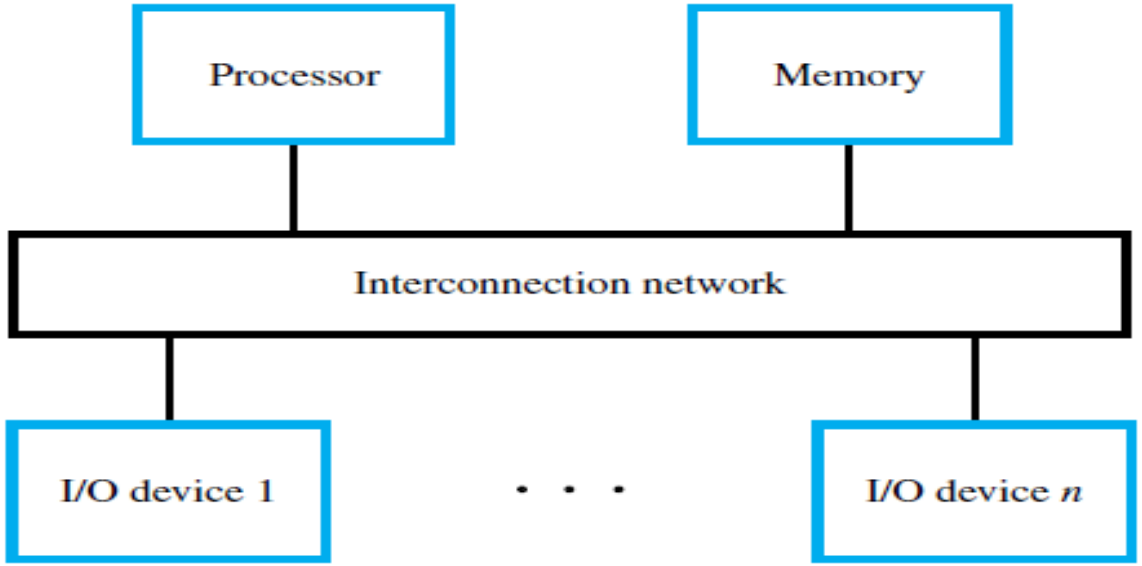
5.Accessing I/O Devices

- Basic features of a computer is its ability to exchange data with other devices.

- enables a human operator, for example, to use a keyboard and a display screen to process text and graphics.

Accessing I/O Devices

- The components of a computer system communicate with each other through an interconnection network



- The interconnection network consists of circuits needed to transfer information between the processor, the memory unit, and a number of I/O devices.

- Idea of using addresses to access various locations in the memory can be extended to deal with the I/O devices.

- Each I/O device must appear to the processor as consisting of some addressable locations, just like the memory.

- Some addresses in the address space of the processor are assigned to these I/O locations, rather than to the main memory

- These locations are usually implemented as bit storage circuits (flip-flops) organized in the form of registers called as *I/O registers*.

- Since the I/O devices and the memory share the same address space, this is called *memory-mapped I/O*.
  - any machine instruction that can access memory can be used to transfer data to or from an I/O device.

- For example, if DATAIN is the address of a register in an input device, the instruction

  Load R2, DATAIN

- reads the data from the DATAIN register and loads them into processor register R2.

  Store R2, DATAOUT

- sends the contents of register R2 to location DATAOUT, which is a register in an output device.

I/O Device Interface

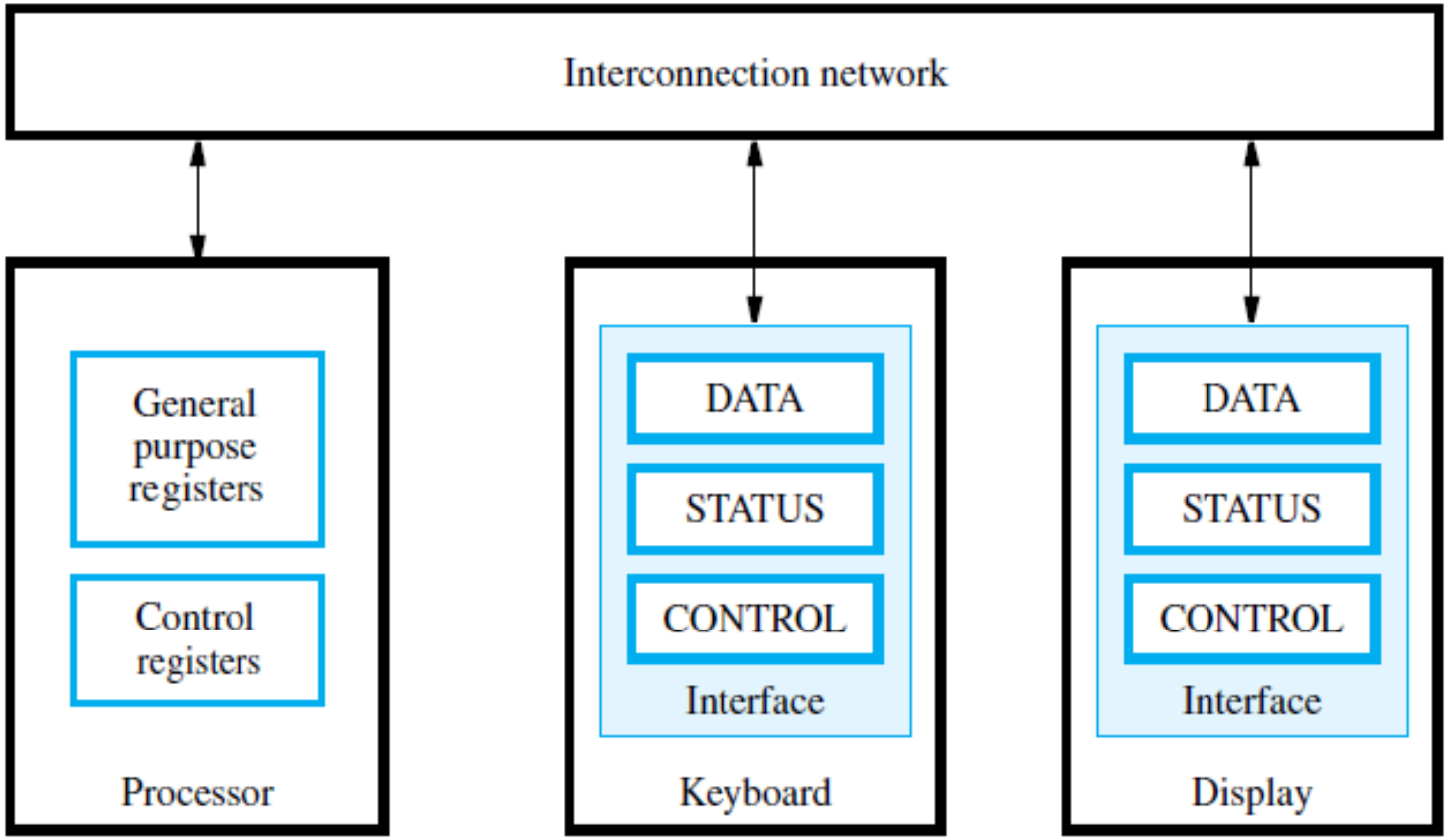
- An I/O device is connected to the interconnection network by using a circuit, called the *device interface*, which provides the means for
  1. data transfer and
  2. exchange of status and control information needed to facilitate the data transfers and
  3. govern the operation of the device.

- The interface includes some registers that can be accessed by the processor.
  a. serve as a buffer for data transfers,
  b. may hold information about the current status of the device
  c. may store the information that controls the operational behavior of the device.

- *data*, *status*, and *control* registers are accessed by program instructions as memory locations.

- Typical transfers of information are between I/O registers and the registers in the processor.

The connection for processor, keyboard, and display from software point-of-view



Program-Controlled I/O

- Consider a task that reads characters typed on a keyboard, stores these data in the memory, and displays the same characters on a display screen.
- A simple way of implementing this task is to *write a program* that performs all functions needed to realize the desired action. This method is known as *program-controlled I/O.*
- It is necessary to ensure that it happens at the right time.
- An input character must be read in response to a key being pressed.
- For output, a character must be sent to the display only when the display device is able to accept it.

- The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user
- The rate of output transfers from the computer to the display is much higher(typically several thousand characters per second).
- However, this is still much slower than the speed of a processor that can execute billions of instructions per second.
- The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.
- One solution to this problem involves a signaling protocol.

- On output, the processor sends the first character and then waits for a signal from the display that the next character can be sent. It then sends the second character, and so on.

- An input character is obtained from the keyboard in a similar way.

1. The processor waits for a signal indicating that a key has been pressed

2. a binary code that represents the corresponding character is available in an I/O register associated with the keyboard.

3. Then the processor proceeds to read that code.

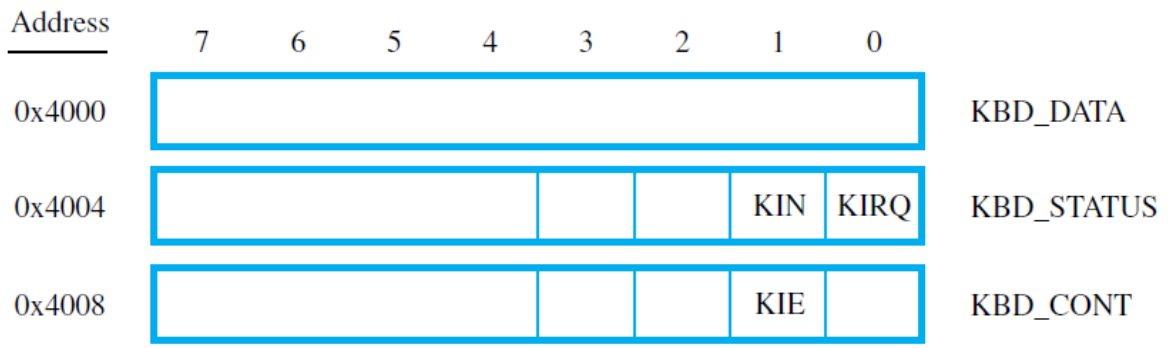
- The keyboard includes a circuit that responds to a key being pressed by producing the code for the corresponding character that can be used by the computer.
- Assume that ASCII code is used, in which each character code occupies one byte.
- Let *KBD_DATA* be the address label of an 8-bit register that *holds the generated character.*
- Also, let a signal indicating that a key has been pressed be provided by setting to 1 a flip-flop called KIN, which is a part of an eight-bit status register, KBD_STATUS.
- The processor can read the status flag KIN to determine when a character code has been placed in KBD_DATA.
- When the processor reads the status flag to determine its state, we say that the processor *polls* the I/O device.

- The display includes an 8-bit register, DISP_DATA, used to receive characters from the processor.

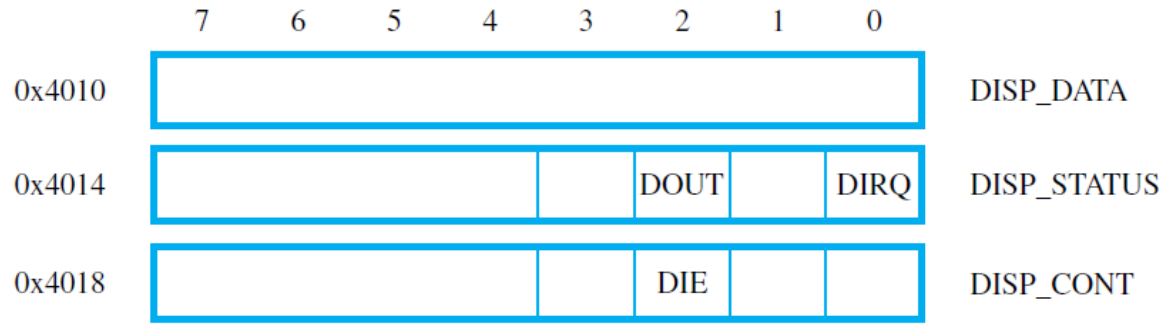
- It also must be able to indicate that it is ready to receive the next character; using a status flag called DOUT, which is one bit in a status register, DISP_STATUS.

- Figure (next slide) illustrates how these registers may be organized.

Registers in the keyboard and display interfaces.



(a) Keyboard interface



(b) Display interface

- If the registers in I/O interfaces are to be accessed as if they are memory locations, each register must be assigned a specific address that will be recognized by the interface circuit.
- In Figure (prev slide), we assigned hexadecimal numbers 4000 and 4010 as base addresses for the keyboard and display, respectively.
- These are the addresses of the data registers.
- The addresses of the status registers are four bytes higher, and the control registers are eight bytes higher.
- This makes all addresses word-aligned in a 32-bit word computer
- Assigning the addresses to registers in this manner makes the I/O registers accessible in a program executed by the processor.
- This is the programmer's view of the device

- A program is needed to perform the task of reading the characters produced by the keyboard, storing these characters in the memory, and sending them to the display.

- To perform I/O transfers, the processor must execute machine instructions that check the state of the status flags and transfer data between the processor and the I/O devices.

- Consider the details of the input process.

1. When a key is pressed, the keyboard circuit places the ASCII-encoded character into the KBD_DATA register.

2. At the same time, the circuit sets the KIN flag to 1.

3. Meanwhile, the processor is executing the I/O program which continuously checks the state of the KIN flag.

4. When it detects that KIN is set to 1, it transfers the contents of KBD_DATA into a processor register.

5. Once the contents of KBD_DATA are read, KIN must be cleared to 0

6. If a second character is entered at the keyboard, KIN is again set to 1 and the process repeats.

The desired action can be achieved by performing the operations:

- READWAIT Read the KIN flag
- Branch to READWAIT if KIN = 0
- Transfer data from KBD_DATA to R5

- Same process takes place when characters are transferred from the processor to the display.

1. When DOUT is equal to 1, the display is ready to receive a character.
2. Under program control, the processor monitors DOUT, and when DOUT is equal to 1, the processor transfers an ASCII-encoded character to DISP_DATA.
3. The transfer of a character to DISP_DATA clears DOUT to 0.
4. When the display device is ready to receive a second character, DOUT is again set to 1.

- This can be achieved by performing the operations:
- WRITEWAIT Read the DOUT flag
- Branch to WRITEWAIT if DOUT = 0
- Transfer data from R5 to DISP_DATA

- The wait loop is executed repeatedly until the status flag DOUT is set to 1 by the display when it is free to receive a character.
- Then, the character from R5 is transferred to DISP_DATA to be displayed, which also clears DOUT to 0.
- We assume that the initial state of KIN is 0 and the initial state of DOUT is 1.
- This initialization is normally performed by the device control circuits when power is turned on.

- In computers that use memory-mapped I/O, data can be transferred between registers and the processor using instructions such as Load, Store, and Move.

- *LoadByte R5, KBD_DATA*
- *StoreByte R5, DISP_DATA*

- The LoadByte and StoreByte operation codes signify that the operand size is a byte, to distinguish them from the Load and Store operation codes

- The Read operation can be implemented by the RISC-style instructions:

- READWAIT: LoadByte R4, KBD_STATUS
- And R4, R4, #2
- Branch_if_[R4]=0 READWAIT
- LoadByte R5, KBD_DATA

- The And instruction is used to test the KIN flag, which is bit *b*1 of the status information in R4 that was read from the KBD_STATUS register.
- As long as *b*1 = 0, the result of the AND operation leaves the value in R4 equal to zero, and the READWAIT loop continues to be executed.

- Similarly, the Write operation may be implemented as:

- WRITEWAIT: LoadByte R4, DISP_STATUS
- And R4, R4, #4
- Branch_if_[R4]=0 WRITEWAIT
- StoreByte R5, DISP_DATA

- And instruction in this case uses the immediate value 4 to test the display's status bit, *b*2.

An Example of a RISC-Style I/O Program

- A complete program for a typical I/O task, shown in Figure (next slide)
- The program uses the program-controlled I/O approach to read, store, and display a line of characters typed on the keyboard.
- As the characters are read in, one by one, they are stored in the memory and then *echoed* back to the display.
- The program finishes when the carriage return character, CR, is encountered.
- The address of the first-byte location of the memory where the line is to be stored is LOC.
- Register R2 is used to point to this part of the memory, and it is initially loaded with the address LOC by the first instruction in the program.
- R2 is incremented for each character read and displayed.

A RISC-style program that reads a line of characters and displays it.

```
        Move            R2, #LOC          Initialize pointer register R2 to point to the
                                            address of the first location in main memory
                                            where the characters are to be stored.
        MoveByte        R3, #CR           Load ASCII code for Carriage Return into R3.
READ:   LoadByte        R4, KBD_STATUS    Wait for a character to be entered.
        And             R4, R4, #2        Check the KIN flag.
        Branch_if_[R4]=0 READ
        LoadByte        R5, KBD_DATA      Read the character from KBD_DATA
                                            (this clears KIN to 0).
        StoreByte       R5, (R2)          Write the character into the main memory and
        Add             R2, R2, #1          increment the pointer to main memory.
ECHO:   LoadByte        R4, DISP_STATUS   Wait for the display to become ready.
        And             R4, R4, #4        Check the DOUT flag.
        Branch_if_[R4]=0 ECHO
        StoreByte       R5, DISP_DATA     Move the character just read to the display
                                            buffer register (this clears DOUT to 0).
        Branch_if_[R5]≠[R3] READ          Check if the character just read is the
                                            Carriage Return. If it is not, then
                                            branch back and read another character.
```

An Example of a CISC-Style I/O Program

- In CISC instruction sets it is possible to perform some arithmetic and logic operations directly on operands in the memory.
- So, it is possible to have the instruction
- TestBit destination, #k
- which tests the bit b_k of the destination operand and sets the condition flag Z (Zero) to 1 if $b_k = 0$ and to 0 otherwise.
- Since the operand can be in a memory location, we can use the instruction
- TestBit KBD_STATUS, #1
- to test the state of the KIN flag in the keyboard interface.
- A Branch instruction that checks the state of the Z flag can then be used to cause a branch to the beginning of the wait loop.

- Figure (next slide) gives a CISC-style program that reads and displays a line of characters.
- The first MoveByte instruction transfers each character directly from KBD_DATA to the memory location pointed to by R2.
- A Compare instruction
- Compare destination, source
- performs the comparison by subtracting the contents of the source from the contents of the destination, and then sets the condition flags based on the result.
- It does not change the contents of either the source or the destination.
- CompareByte instruction uses the auto increment addressing mode, which automatically increments the value of the pointer R2 after the comparison has been made.
- In the RISC-style program in Figure (slide 101) the pointer has to be incremented using a separate Add instruction.

A CISC-style program that reads a line of characters and displays it.

	Move	R2, #LOC	Initialize pointer register R2 to point to the address of the first location in main memory where the characters are to be stored.
READ:	TestBit	KBD_STATUS, #1	Wait for a character to be entered
	Branch=0	READ	in the keyboard buffer KBD_DATA.
	MoveByte	(R2), KBD_DATA	Transfer the character from KBD_DATA into the main memory (this clears KIN to 0).
ECHO:	TestBit	DISP_STATUS, #2	Wait for the display to become ready.
	Branch=0	ECHO	
	MoveByte	DISP_DATA, (R2)	Move the character just read to the display buffer register (this clears DOUT to 0).
	CompareByte	(R2)+, #CR	Check if the character just read is CR (carriage return). If it is not CR, then
	Branch≠0	READ	branch back and read another character. Also, increment the pointer to store the next character.

- Almost all of the execution time for the programs in RISC style and CISC style is spent in the two wait loops, while the processor waits for a key to be pressed or for the display to become available.

- Wasting the processor execution time in this manner can be avoided by using the concept of interrupts.

6.Interrupts

- Other tasks can be performed while waiting for an I/O device to become ready.
- The I/O device alerts the processor when it becomes ready by sending a hardware signal called an *interrupt request* to the processor.
- The processor is no longer required to continuously poll the status of I/O devices, it can use the waiting period to perform other useful tasks.
- Therefore, by using interrupts, waiting periods can be eliminated.

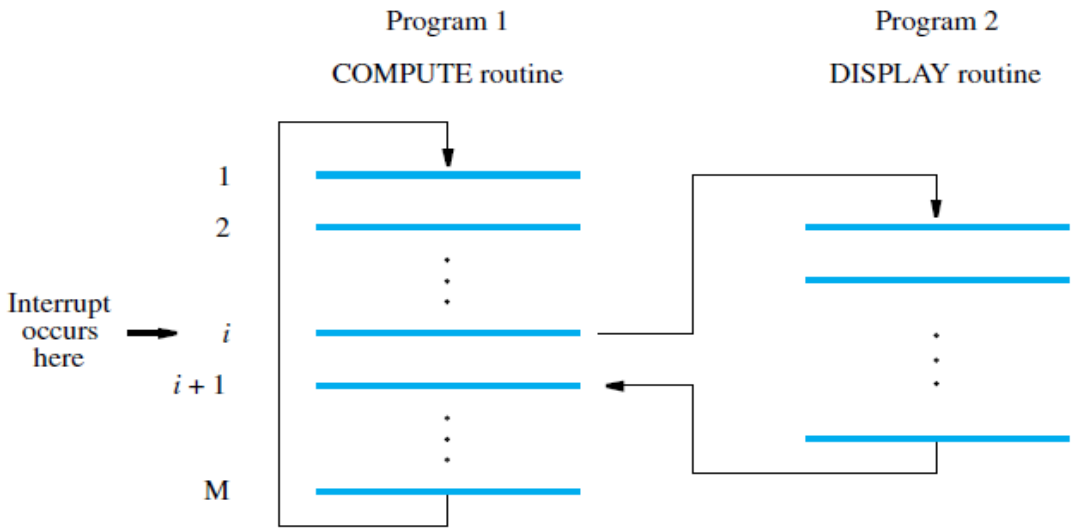
Example

- **C**onsider a task that requires continuous extensive computations to be performed and the results to be displayed on a display device.
- The displayed results must be updated every ten seconds.
- The ten-second intervals can be determined by a simple timer circuit, which generates an appropriate signal.
- The processor treats the timer circuit as an input device that produces a signal that can be interrogated.
- If this is done using polling, the processor will waste considerable time checking the state of the signal.
- A better solution is to have the timer circuit raise an interrupt request once every ten seconds.
- In response, the processor displays the latest results.

- The task can be implemented with a program that consists of two routines, COMPUTE and DISPLAY.
- The processor continuously executes the COMPUTE routine.
- When it receives an interrupt request from the timer, it suspends the execution of the COMPUTE routine and executes the DISPLAY routine which sends the latest results to the display device.
- Upon completion of the DISPLAY routine, the processor resumes the execution of the COMPUTE routine.
- Since the time needed to send the results to the display device is very small compared to the ten-second interval, the processor in effect spends almost all of its time executing the COMPUTE routine.

The routine executed in response to an interrupt request is called the *interrupt-service routine* (DISPLAY routine)

1. interrupt request arrives during execution of instruction
2. processor first completes execution of instruction
3. loads the program counter with the address of the first instruction of the interrupt-service routine
4. After execution of the interrupt-service routine, the processor returns to instruction *i* + 1.
5. when an interrupt occurs, the current contents of the PC, which point to instruction *i* + 1, must be put in temporary storage in a known location
6. A Return-from-interrupt instruction at the end of the interrupt service routine reloads the PC from that temporary storage location, causing execution to resume at instruction *i* + 1



7. The *return address* must be saved either in a designated general-purpose register or on the processor stack

- The processor must inform the device that its request has been recognized so that it may remove its interrupt-request signal using a special control signal, called *interrupt acknowledge.*
- An interrupt-service routine may not have any relation to the portion of the program being executed at the time the interrupt request is received.
- Therefore, before starting the execution of the interrupt service routine, status information and contents of processor registers that may be altered in unanticipated ways during the execution of that routine must be saved.
- This saved information must be restored before the execution of the interrupted program is resumed.

- Most modern processors save only the minimum amount of information needed to maintain the integrity of program execution.

- This is because the process of saving and restoring registers involves memory transfers that increase the total execution time, and hence represent execution overhead.

- Saving registers also increase the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine. This delay is called *interrupt latency*

- So, the processor saves only the contents of the program counter and the processor status register

- Some computers provide two types of interrupts.
  - One saves all register contents, and the other does not.
  - A particular I/O device may use either type, depending on its response time requirements.
- Another approach is to provide duplicate sets of processor registers.
  - a different set of registers can be used by the interrupt-service routine, thus eliminating the need to save and restore registers.
- The duplicate registers are called the *shadow registers*

- The concept of interrupts is used in operating systems and in many control applications where the processing of certain routines must be accurately timed relative to external events(*real-time processing)*.

Enabling and Disabling Interrupts

- A computer must give the programmer complete control over the events that take place during program execution.
- The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and start the execution of another.
- Because interrupts can arrive at any time, they may alter the sequence of events from that visualized by the programmer.
- Hence, the interruption of program execution must be carefully controlled. i. e. ability to enable and disable such interruptions as desired.

- It is convenient to be able to enable and disable interrupts at both the processor and I/O device ends.
- The processor can either accept or ignore interrupt requests.
- Control bits in registers can be used that can be accessed by program instructions for this purpose.

- The processor has a *status register* (PS), which contains information about its current state of operation.
- Let one bit, IE, be assigned for enabling/disabling interrupts.
- Then, the programmer can set or clear IE to cause the desired action.
    - When IE = 1, interrupt requests from I/O devices are accepted and serviced by the processor.
    - When IE = 0, the processor simply ignores all interrupt requests from I/O devices.
- The interface of an I/O device includes a control register that contains the information that governs the mode of operation of the device.
    - One bit in this register may be dedicated to interrupt control.
- The I/O device is allowed to raise interrupt requests only when this bit is set to 1.

Summarizing the sequence of events involved in handling an interrupt request from a single device.

- Assuming that interrupts are enabled in both the processor and the device, the following is a typical scenario:

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed and saves the contents of the PC and PS registers.
3. Interrupts are disabled by clearing the IE bit in the PS to 0.
4. The action requested by the interrupt is performed by the interrupt-service routine, during which time the device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. Upon completion of the interrupt-service routine, the saved contents of the PC and PS registers are restored (enabling interrupts by setting the IE bit to 1), and execution of the interrupted program is resumed.

Handling Multiple Devices

- When an interrupt request is received it is necessary to identify the particular device that raised the request.
- If two devices raise interrupt requests at the same time, it must be possible to break the tie and select one of the two requests for service.
- When the interrupt-service routine for the selected device has been completed, the second request can be serviced.

- The information needed to determine whether a device is requesting an interrupt is available in its status register.
- When the device raises an interrupt request, it sets to 1 bit in its status register, which is the IRQ bit.
- The simplest way to identify the interrupting device is to have the interrupt-service routine poll all I/O devices in the system.
- The first device encountered with its IRQ bit set to 1 is the device that should be serviced.
- The polling scheme is easy to implement. Its main disadvantage is the time spent interrogating the IRQ bits of devices that may not be requesting any service.

Vectored Interrupts

- To reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor.

- Then, the processor can immediately start executing the corresponding interrupt-service routine. This is *vectored interrupts.*

- A device requesting an interrupt identifies itself through
  1. an interrupt request signal, or
  2. send a special code to the processor through the interconnection network.

- Good to allocate permanently an area in the memory to hold the addresses of interrupt-service routines.

- These addresses are referred to as *interrupt vectors* and constitute the *interrupt-vector table*(in the lowest-address range).

- When an interrupt request arrives, the information provided by the requesting device is used as a pointer into the interrupt-vector table, and the address in the corresponding interrupt vector is automatically loaded into the program counter.

Interrupt Nesting

- Interrupt-service routines are usually short, and the delay they may cause is acceptable for most simple devices.
- A long delay in responding to an interrupt request may lead to erroneous operation.
- Consider, for example,
- a computer that keeps track of the time of day using a real-time clock.
- The processor executes a short interrupt-service routine to increment a set of counters in the memory that keep track of time in seconds, minutes, and so on.

- Proper operation requires that the delay in responding to an interrupt request from the real-time clock be small in comparison with the interval between two successive requests.

- To ensure that this requirement is satisfied in the presence of other interrupting devices, it may be necessary to accept an interrupt request from the clock during the execution of an interrupt-service routine for another device, i.e., to nest interrupts.

- An interrupt request from a high-priority device should be accepted while the processor is servicing a request from a lower-priority device.
- Assign a priority level to the processor that can be changed under program control.
- The priority level of the processor is the priority of the program that is currently being executed.
- The processor accepts interrupts only from devices that have priorities higher than its own.
- The processor automatically or with additional instructions raises its priority to that of the device when the interrupt-service routine for that device begins to run.
- This action disables interrupts from devices that have the same or lower level of priority.

- Interrupt requests from higher-priority devices will continue to be accepted.
- The processor's priority can be encoded in a few bits of the processor status register.
- Finally, we should point out that if nested interrupts are allowed, then each interrupt service routine must save on the stack the saved contents of the program counter and the status register.
- This has to be done before the interrupt-service routine enables nesting by setting the IE bit in the status register to 1.

Simultaneous Requests

- The processor must have some means of deciding which request to service first.
- Polling the status registers of the I/O devices is the simplest such mechanism.
- In this case, priority is determined by the order in which the devices are polled.
- When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code.

Controlling I/O Device Behavior

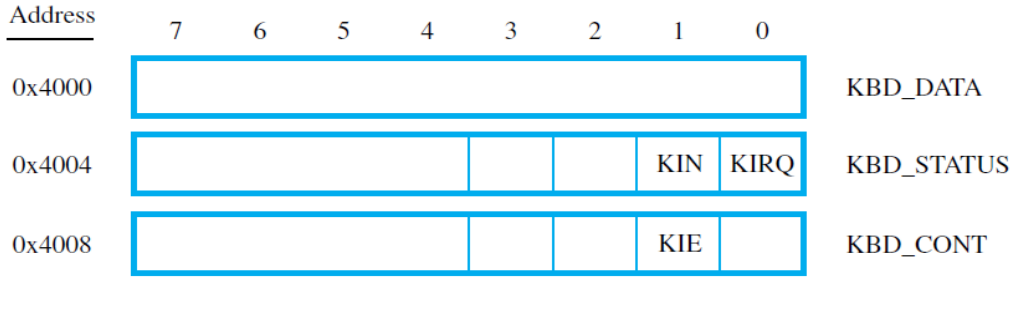
- Interrupt requests are generated only by those I/O devices that the processor is currently willing to recognize

- Hence, we need a mechanism in the interface circuits of individual devices to control whether a device is allowed to interrupt the processor.

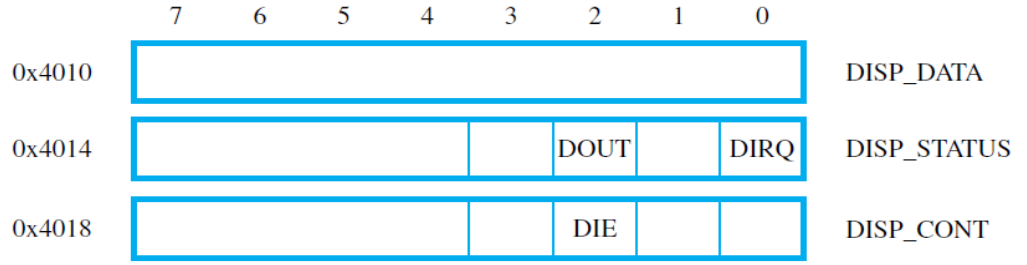
- The control needed is usually provided in the form of an *interrupt-enable* bit in the device's interface circuit.

- A commonly used approach is to provide a control register in the device interface, which holds the information needed to control the behavior of the device.
- This register is accessed as an addressable location
- One bit in the register serves as the interrupt-enable bit, IE.
- When it is set to 1 by an instruction that writes new information into the control register, the device is placed into a mode in which it is allowed to interrupt the processor whenever it is ready for an I/O transfer.

- Figure shows the registers that may be used in the interfaces of keyboard and display devices.
- Since these devices handle one character at a time, it is appropriate to use an eight-bit data register.
- The status and control registers are also eight bits long.
- Only one or two bits in these registers are needed to handle the I/O transfers.
- The remaining bits can be used to specify other aspects of the operation of the device, or ignored if they are not needed.
- The keyboard status register includes bits KIN and KIRQ.
- The KIRQ bit is set to 1 if an interrupt request has been raised, but not yet serviced.
- The keyboard may raise interrupt requests only when the interrupt-enable bit, KIE, in its control register, is set to 1.
- i.e. when both KIE and KIN bits are equal to 1, an interrupt request is raised and the KIRQ bit is set to 1.



(a) Keyboard interface



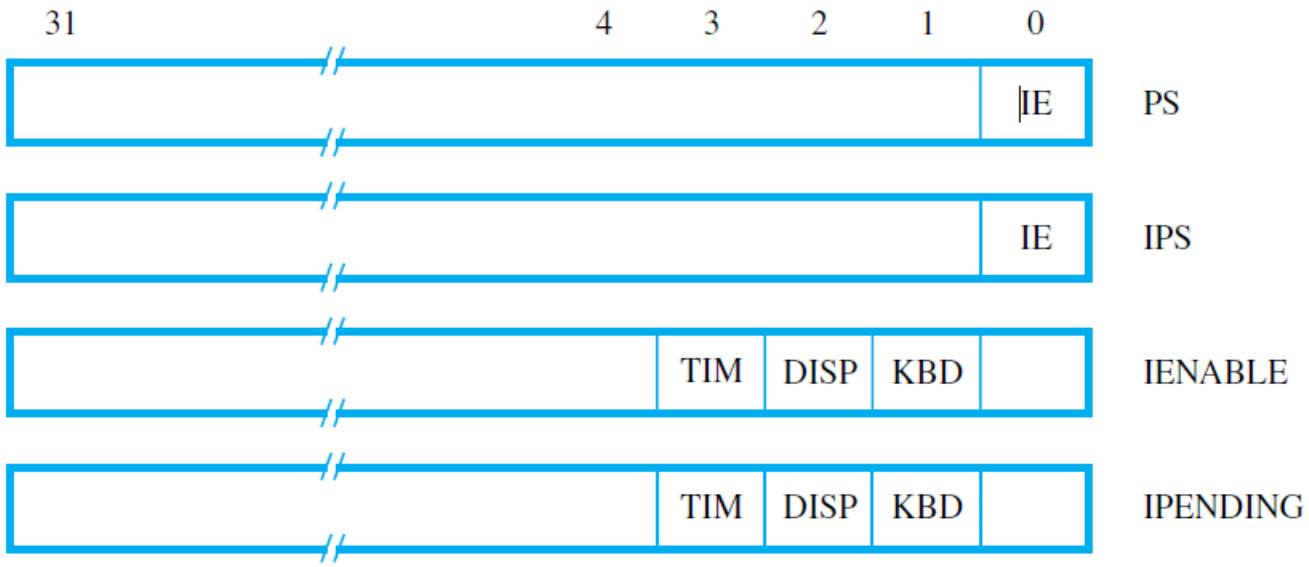
(b) Display interface

- DIRQ bit in the status register of the display interface indicates whether an interrupt request has been raised.
- Bit DIE in the control register of this interface is used to enable interrupts.
- KIN and KIE are in bit position 1, and DOUT and DIE in position 2.
- This is by choice.

Processor Control Registers

- To deal with interrupts it is useful to have some other control registers. (other than a status register)
- In figure (next slide) there are four processor control registers.
- The status register, PS, includes the interrupt-enable bit, IE, in addition to other status information.
- The IPS register is used to automatically save the contents of PS when an interrupt request is received and accepted.
- At the end of the interrupt-service routine, the previous state of the processor is automatically restored by transferring the contents of IPS into PS.
- save the contents of IPS(one register) on the stack if nested interrupts are allowed.

Control registers in the processor



- The IENABLE register allows the processor to selectively respond to individual I/O devices.
- A bit may be assigned for each device, as shown in the figure for the keyboard, display …
- When a bit is set to 1, the processor will accept interrupt requests from the corresponding device.
- The IPENDING register indicates the active interrupt requests.
- A program can decide which interrupt should be serviced first.

- In a 32-bit processor, the control registers are 32 bits long.
- Therefore, in Figure (slide 131), it is possible to accommodate 32 I/O devices
- Assembly-language instructions can refer to processor control registers by using names such as those in Figure
- But, these registers cannot

1. be accessed in the same way as the general-purpose registers.
2. be accessed by arithmetic and logic instructions.
3. be accessed by Load and Store instructions

- Special instructions or special addressing modes may be provided to access the processor control registers.

- In a RISC-style processor, the special instructions may be of the type
- MoveControl R2, PS
- which loads the contents of the program status register into register R2, and
- MoveControl IENABLE, R3
- which places the contents of R3 into the IENABLE register.
- These instructions perform transfers between control and general-purpose registers.

Exceptions

- An interrupt is an event that causes the execution of one program to be suspended and the execution of another program to begin.
  - Interrupts caused by events associated with I/O data transfers.
- Interrupt mechanism is used in a number of other situations.
- The term *exception* is often used to refer to any event that causes an interruption.
- Hence, I/O interrupts are one example of an exception.

Recovery from Errors

- Computers use a variety of techniques to ensure that all hardware components are operating properly.
- For example, many computers include an error-checking code in the main memory, which allows the detection of errors in the stored data.
- If an error occurs, the control hardware detects it and informs the processor by raising an interrupt.
- The processor may also interrupt a program if it detects an error or an unusual condition while executing the instructions of this program.
- For example, the OP-code field of an instruction may not correspond to any legal instruction, or an arithmetic instruction may attempt a division by zero.

- When exception processing is initiated as a result of such errors, the processor proceeds in exactly the same manner as in the case of an I/O interrupt request.

- It suspends the program being executed and starts an exception-service routine, which takes appropriate action to recover from the error, if possible, or to inform the user about it.

- When an interrupt is caused by an error associated with the current instruction, that instruction cannot usually be completed, and the processor begins exception processing immediately.

Debugging

- Another important type of exception is used as an aid in debugging programs.
- System software usually includes a program called a *debugger*, which helps the programmer find errors in a program.
- The debugger uses exceptions to provide two important facilities: trace mode and breakpoints.

Use of Exceptions in Operating Systems

- The operating system (OS) software coordinates the activities within a computer.
- It uses exceptions to communicate with and control the execution of user programs.
- It uses hardware interrupts to perform I/O operations

- Two basic approaches to I/O transfers.
- The simplest technique is programmed I/O, in which the processor performs all of the necessary functions under direct control of program instructions.
- The second approach is based on the use of interrupts; this mechanism makes it possible to interrupt the normal execution of programs in order to service higher-priority requests that require more urgent attention.
- The complexity and sophistication of interrupt-handling schemes vary from one computer to another.
- These are I/O issues from the programmer’s point of view.
- Next: the hardware aspects and some commonly used I/O standards.

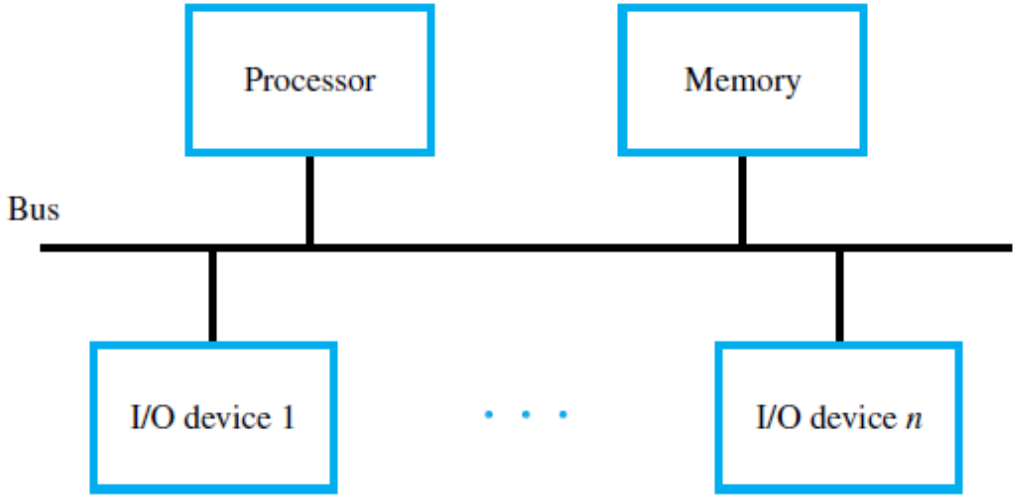
7.Bus Structure

- The basic feature of a computer is its ability to transfer data to and from I/O devices.
- A computer should have the ability to exchange information with a wide variety of devices(computers and embedded systems)
- The processor is fully involved in these exchanges.
- Data transfers may also take place directly between I/O devices, such as magnetic hard disks, and the main memory, with only minimal involvement of the processor

- An interconnection network is used to transfer data among the processor, memory, and I/O devices.

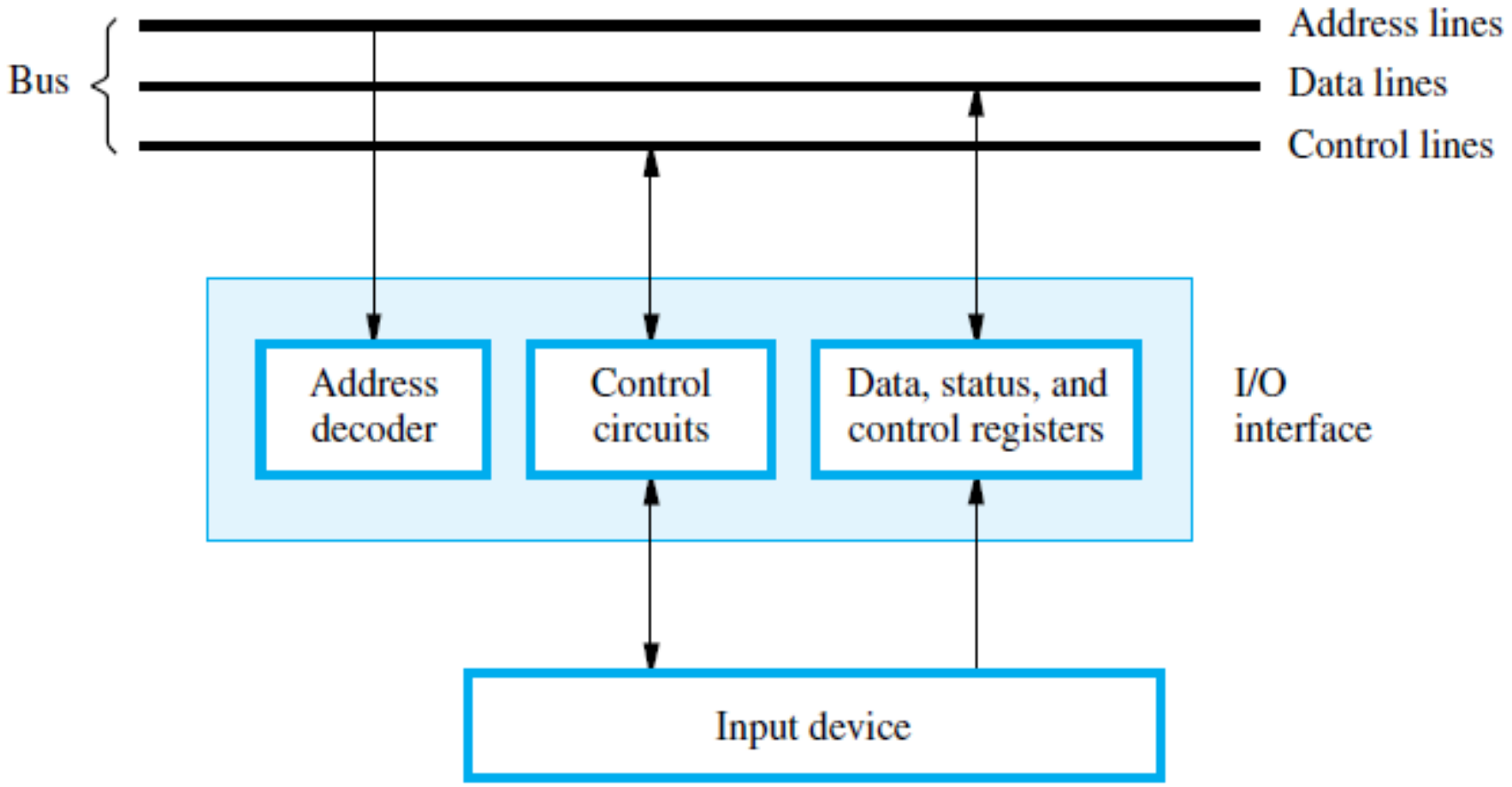
- A commonly used interconnection network is called a *bus* which is a simple structure that implements the interconnection network.

- Only one source/destination pair of units can use this bus to transfer data at any one time.



- The bus consists of three sets of lines used to carry address, data, and control signals.
- I/O device interfaces are connected to these lines
- Each I/O device is assigned a unique set of addresses for the registers in its interface.
- When the processor places a particular address on the address lines, it is examined by the address decoders of all devices on the bus.
- The device that recognizes this address responds to the commands issued on the control lines.
- The processor uses the control lines to request either a Read or a Write operation, and the requested data are transferred over the data lines.

I/O interface for an input device.



- When I/O devices and the memory share the same address space, the arrangement is called *memory-mapped I/O*

- Any machine instruction that can access memory can be used to transfer data to or from an I/O device.

- For example, if the input device is a keyboard and DATAIN is its data register, the instruction

- Load R2, DATAIN

- reads the data from DATAIN and stores them into processor register R2.

- Similarly, the instruction
- Store R2, DATAOUT
- sends the contents of register R2 to location DATAOUT, which may be the data register of a display device interface.

- The status and control registers contain information relevant to the operation of the I/O device.

- The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.

8.Bus Operation

- A bus requires a set of rules, called a *bus protocol*, that governs how the bus is used by various devices.
- The bus protocol determines

1. when a device may place information on the bus,
2. when it may load the data on the bus into one of its registers, and so on.

- These rules are implemented by control signals that indicate what and when actions are to be taken.

- One control line, labeled R/$\overline{W}$, specifies whether a Read or a Write operation is to be performed.
    1. specifies Read when set to 1 and
    2. Write when set to 0.

- When several data sizes are possible, such as byte, halfword, or word, the required size is indicated by other control lines.

- The bus control lines also carry timing information.

- They specify the times at which the processor and the I/O devices may place data on or receive data from the data lines.

- A variety of schemes have been devised for the timing of data transfers over a bus.

- These can be broadly classified as either synchronous or asynchronous schemes.

- In any data transfer operation, one device plays the role of a *master*.
- This is the device that initiates data transfers by issuing Read or Write commands on the bus.
- Normally, the processor acts as the master, but other devices may also become masters
- The device addressed by the master is referred to as a *slave*.

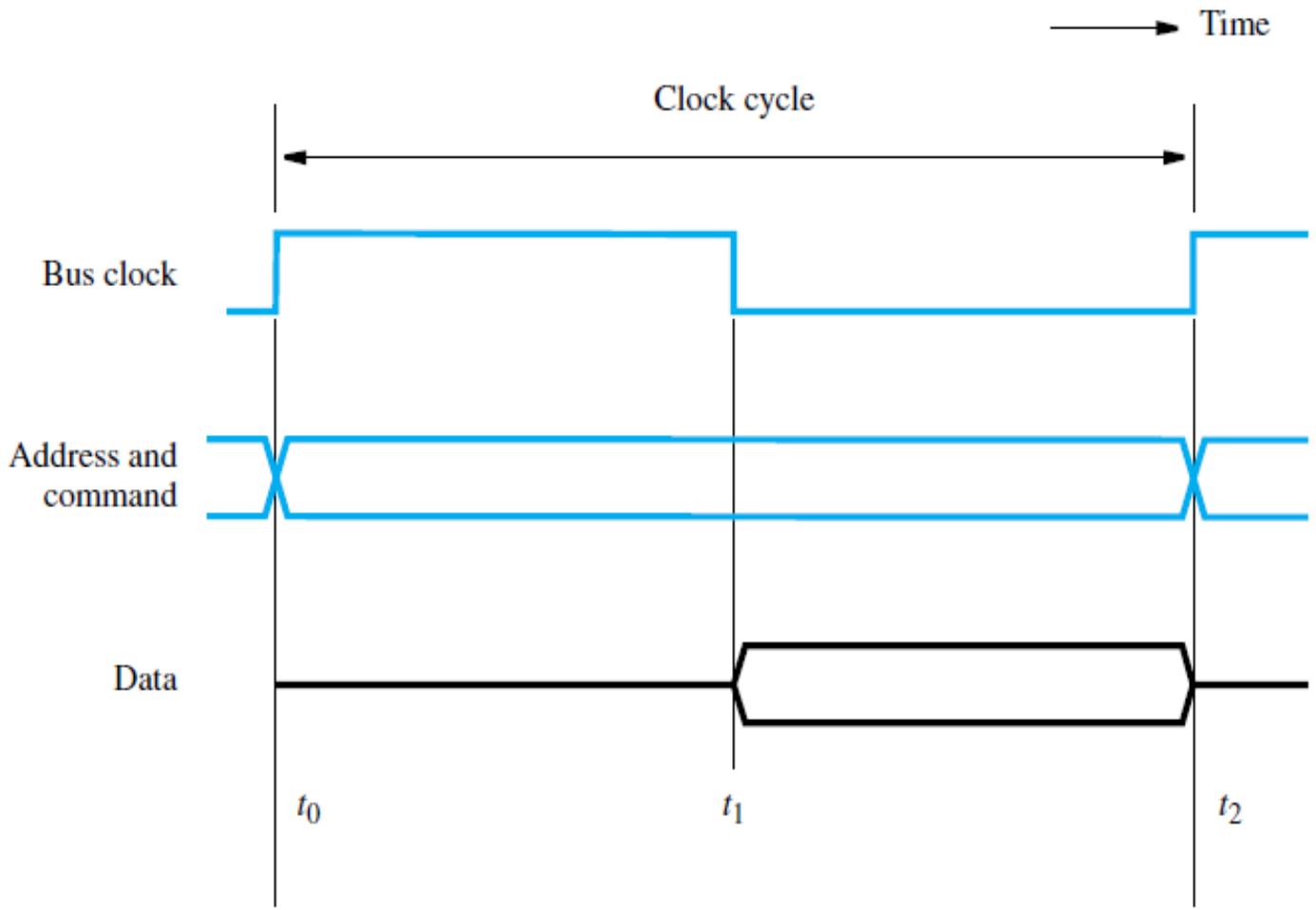
Synchronous Bus

- All devices derive timing information from a control line called the *bus clock*
- The signal on this line has two *phases*: a high level followed by a low level which together constitute a *clock cycle*.
- The first half of the cycle between the low-to-high and high-to-low transitions is referred to as a clock pulse.
- The address and data lines are shown as if they are carrying both high and low signal levels at the same time.
    - This is a common convention for indicating that some lines are high and some low, depending on the particular address or data values being transmitted.

- The crossing points indicate the times at which these patterns change

- A signal line at a level halfway between the low and high signal levels indicates periods during which the signal is unreliable, and must be ignored by all devices.

Timing of an input transfer on a synchronous bus.



- Consider the sequence of signal events during an input (Read) operation.
- At time t_0, the master places the device address on the address lines and sends a command on the control lines indicating a Read operation.
  - The command may also specify the length of the operand to be read.
- Information travels over the bus at a speed determined by its physical and electrical characteristics.

- The clock pulse width, $t_1 - t_0$,

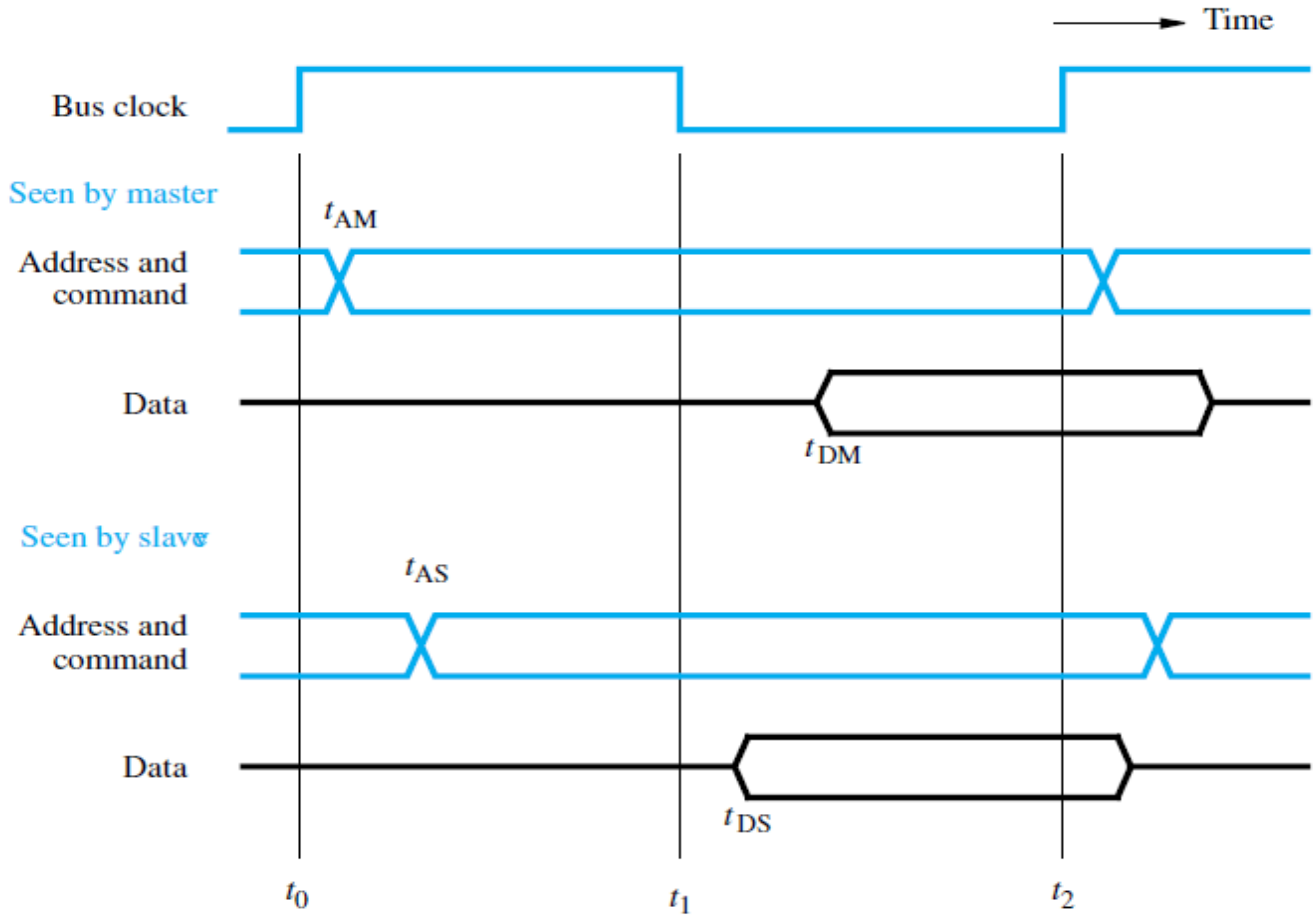
1. must be longer than the maximum propagation delay over the bus.
2. must be long enough to allow all devices to decode the address and control signals so that the addressed device (the slave) can respond at time t_1 by placing the requested input data on the data lines.

- At the end of the clock cycle, *t*2,
- the master loads the data on the data lines into one of its registers.
- To be loaded correctly into a register, data must be available for a period greater than the setup time of the register.
- The period *t*2 – *t*1 must be greater than the maximum propagation time on the bus plus the setup time of the master's register.

- A similar procedure is followed for a Write operation.
- The master places the output data on the data lines when it transmits the address and command information.
- At time t_2, the addressed device loads the data into its data register.
- The timing diagram in Figure is a representation of the actions that take place on the bus lines.

- Figure slide 130 gives a more realistic picture of what happens.
- Two views of each signal, except the clock.
- Signals take time to travel from one device to another, a given signal transition is seen by different devices at different times.
- The top view shows the signals as seen by the master and the bottom view as seen by the slave.
- The clock changes are seen at the same time by all devices connected to the bus.
- The master sends the address and command signals on the rising edge of the clock at the beginning of the clock cycle (at t_0).
- These signals do not appear on the bus until t_{AM}, due to the delay in the electronic circuit output from the master to the bus lines.

A detailed timing diagram for the input transfer (write operation)



- After a while, at t_{AS}, the signals reach the slave.
- The slave decodes the address, and at t_1 sends the requested data.
- The data signals do not appear on the bus until t_{DS}.
- They travel toward the master and arrive at t_{DM}.
- At $t2$, the master loads the data into its register.
  - $t_2 - t_{DM}$ must be greater than the setup time of that register.
- The data must continue to be valid after t_2 for a period equal to the hold time requirement of the register
- Actual signals will always involve delays as shown in Figure (prev slide)

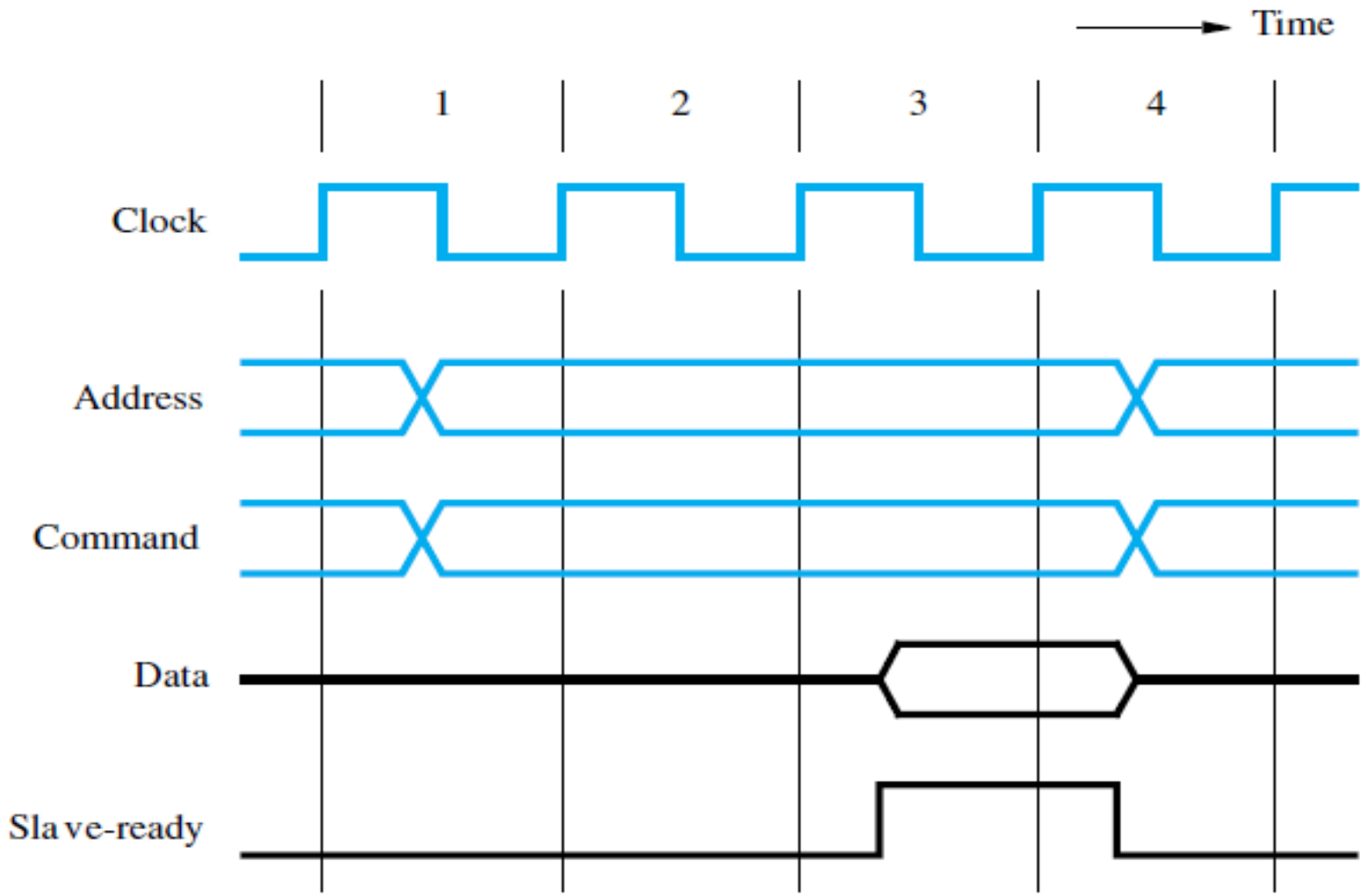
Multiple-Cycle Data Transfer

- Limitations of the scheme described:
- As a transfer has to be completed within one clock cycle, the clock period, $t_2 - t_0$, must be chosen to accommodate the longest delays on the bus and the slowest device interface.
- This forces all devices to operate at the speed of the slowest device.
- The processor has no way of determining whether the addressed device has responded.
- At t_2, it assumes that the input data are available on the data lines in a Read operation, or that the output data have been received by the I/O device in a Write operation.
- If, because of a malfunction, a device does not operate correctly, the error will not be detected.

To overcome these limitations, most buses incorporate control signals that represent a response from the device.

- These signals inform the master that the slave has recognized its address and that it is ready to participate in a data transfer operation.
- Also, makes it possible to adjust the duration of the data transfer period to match the response speeds of different devices.
- Accomplished by allowing a complete data transfer operation to span several clock cycles.
- Then, the number of clock cycles involved can vary from one device to another.

An input transfer using multiple clock cycles.



- An example of this approach is shown in Figure (slide 161)
- During clock cycle 1, the master sends address and command information on the bus, requesting a Read operation.
- The slave receives this information and decodes it.
- It begins to access the requested data on the active edge of the clock at the beginning of clock cycle 2.
- The data become ready and are placed on the bus during clock cycle 3.
- The slave asserts a control signal called Slave-ready at the same time.

- The master, which has been waiting for this signal, loads the data into its register at the end of the clock cycle.
- The slave removes its data signals from the bus and returns its Slave-ready signal to the low level at the end of cycle 3.
- The bus transfer operation is now complete, and the master may send a new address and command signals to start a new transfer in clock cycle 4.
- The Slave-ready signal is an acknowledgment from the slave to the master, confirming that the requested data have been placed on the bus.
  - Allows the duration of a bus transfer to change from one device to another.

- In the example in Figure (slide 161), the slave responds in cycle 3.
- A different device may respond in an earlier or a later cycle.
- If the addressed device does not respond at all, the master waits for some predefined maximum number of clock cycles, and then aborts the operation.
- This could be the result of an incorrect address or a device malfunction.
- A different approach that does not use a clock signal at all is the Asynchronous bus.

Asynchronous Bus

- An alternative scheme for controlling data transfers on a bus is based on the use of a *handshake* protocol between the master and the slave.
- A handshake is an exchange of command and response signals between the master and the slave.

- A control line called **Master-ready** is asserted by the master to indicate that it is ready to start a data transfer.

- The Slave responds by asserting **Slave-ready.**

- The master places the address and command information on the bus. Then it indicates to all devices that it has done so by activating the Master-ready line.
- This causes all devices to decode the address. The selected slave performs the required operation and informs the processor that it has done so by activating the Slave-ready line.
- The master waits for Slave-ready to become asserted before it removes its signals from the bus.
- In the case of a Read operation, it also loads the data into one of its registers.

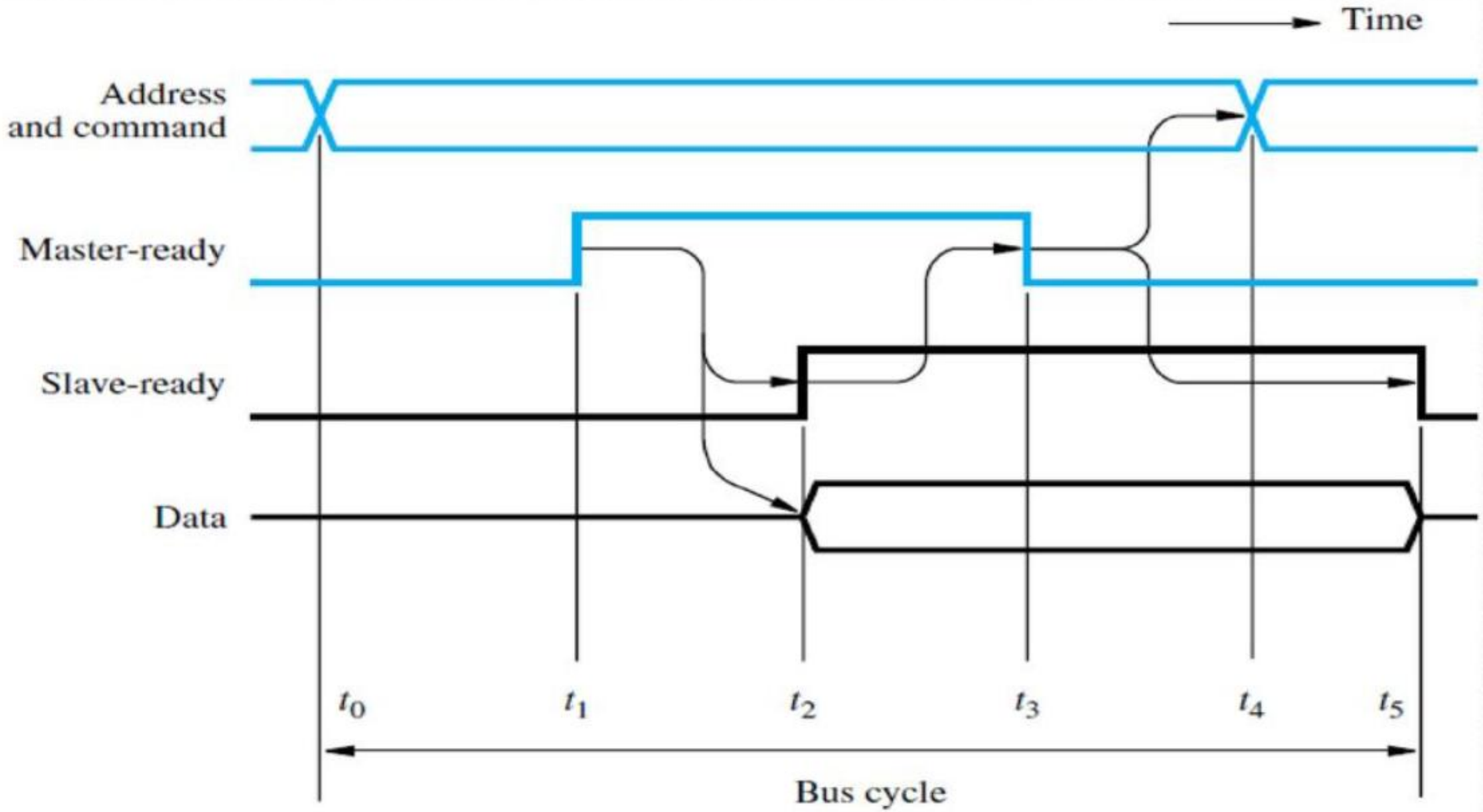


Figure 7.6 Handshake control of data transfer during an input operation.

- $t0$—The master places the address and command information on the bus, and all devices on the bus decode this information.
- $t1$—The master sets the Master-ready line to 1 to inform the devices that the address and command information is ready. Sufficient time should be allowed for the device interface circuitry to decode the address. The delay needed can be included in the period $t1 - t0$.
- $t2$—The selected slave, having decoded the address and command information, performs the required input operation by placing its data on the data lines. At the same time, it sets the Slave-ready signal to 1. If extra delays are introduced by the interface circuitry before it places the data on the bus, the slave must delay the Slave-ready signal accordingly. The period $t2 - t1$ depends on the distance between the master and the slave and on the delays introduced by the slave's circuitry.

- $t3$—The Slave-ready signal arrives at the master, indicating that the input data are available on the bus. After a delay to the master loads the data into its register. Then, it drops the Master-ready signal, indicating that it has received the data.
- $t4$—The master removes the address and command information from the bus. The delay between $t3$ and $t4$ is again intended to allow for bus skew. Erroneous addressing may take place if the address, as seen by some device on the bus, starts to change while the Master-ready signal is still equal to 1.
- $t5$—When the device interface receives the 1-to-0 transition of the Master-ready signal, it removes the data and the Slave-ready signal from the bus. This completes the input transfer.

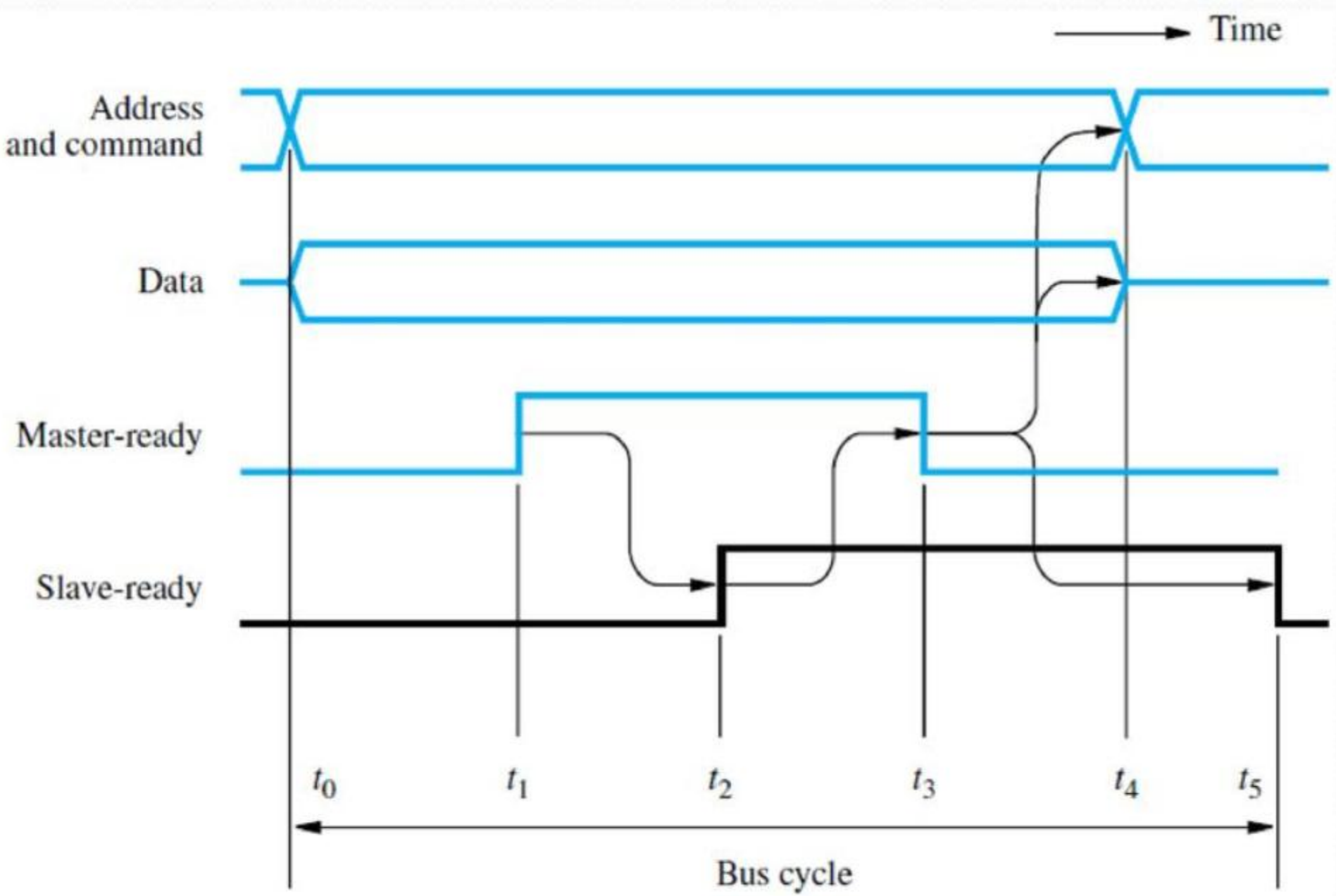


Figure 7.7 Handshake control of data transfer during an output operation.

Arbitration

- two or more entities contend for the use of a single resource in a computer system
- two devices may need to access a given slave at the same time
- necessary to decide which device will access the slave first
- decision is usually made in an arbitration process performed by anarbiter circuit
- each device sends a request to arbitrar for using the shared resource
- If it receives two requests at the same time, it grants the use of the slave to the device having the higher priority first.

- consider the case where a single bus is the shared resource.
- The device that initiates data transfer requests on the bus is the bus master (processor)

- It is possible that several devices in a computer system need to be bus masters to transfer data.
- For example, an I/O device needs to be a bus master to transfer data directly to or from the computer's memory. Since the bus is a single shared facility, it is essential to provide orderly access to it by the bus masters.
- A device that wishes to use the bus sends a request to the arbiter.
- When multiple requests arrive at the same time, the arbiter selects one request and grants the bus to the corresponding device.

- For some devices, a delay in gaining access to the bus may lead to an error.
- Such devices must be given high priority.

- If there is no particular urgency among requests, the arbiter may grant the bus using a simple round-robin scheme.

- Figure 7.8 illustrates an arrangement for bus arbitration involving two masters.

- There are two Bus-request lines, BR1 and BR2, and two Bus-grant lines, BG1 and BG2, connecting the arbiter to the masters

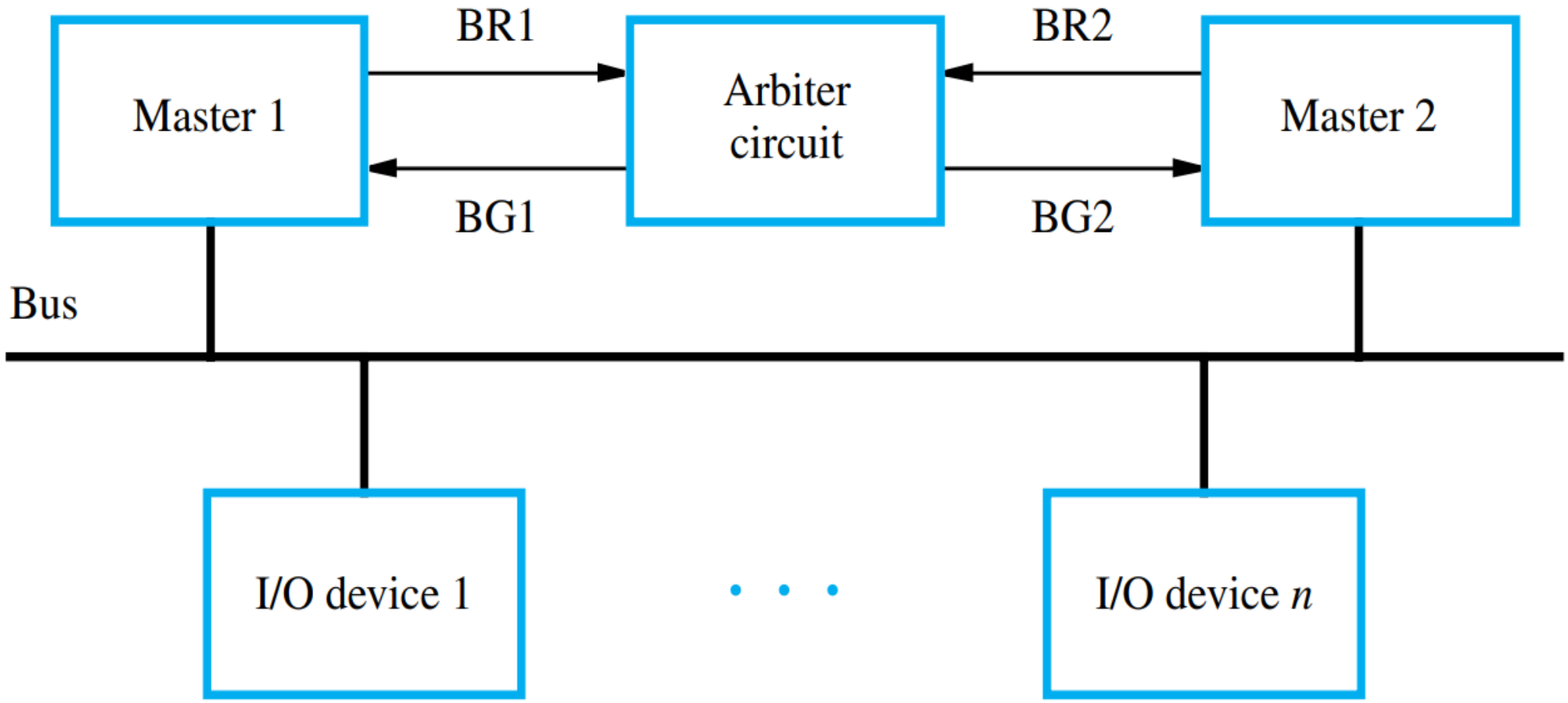


Figure 7.8 Bus arbitration.

- A master requests use of the bus by activating its Bus-request line.
- If a single Bus-request is activated, the arbiter activates the corresponding Bus-grant.

- This indicates to the selected master that it may now use the bus for transferring data.

- When the transfer is completed, that master deactivates its Bus-request, and the arbiter deactivates its Bus-grant

- Figure 7.9 illustrates a possible sequence of events for the case of three masters.
- Assume that master 1 has the highest priority, followed by the others in increasing numerical order.
- Master 2 sends a request to use the bus first. Since there are no other requests, the arbiter grants the bus to this master by asserting BG2.
- When master 2 completes its data transfer operation, it releases the bus by deactivating BR2. By that time, both masters 1 and 3 have activated their request lines.
- Since device 1 has a higher priority, the arbiter activates BG1 after it deactivates BG2, thus granting the bus to master 1.

- Later, when master 1 releases the bus by deactivating BR1, the arbiter deactivates BG1 and activates BG3 to grant the bus to master 3.

- Note that the bus is granted to master 1 before master 3 even though master 3 activated its request line before master 1.

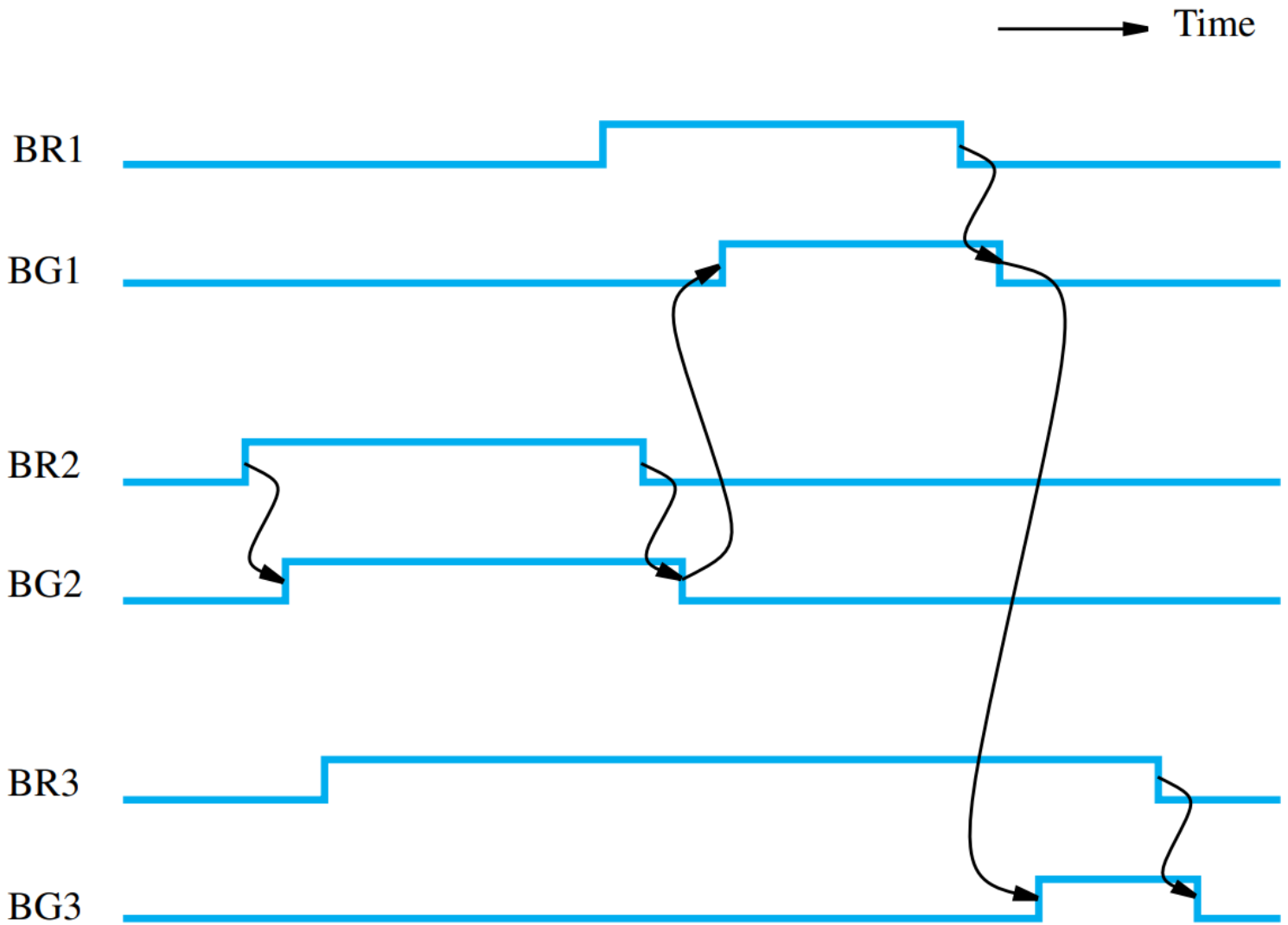


Figure 7.9 Granting use of the bus based on priorities.

End of Unit 2