Course – Computer Organization and Architecture

Course Instructor

Dr. Umadevi V

Department of CSE, BMSCE

Unit-2

Stacks, Subroutines, Additional Instructions
Basic Input/Output: Accessing I/O Devices, Interrupts, Bus Structure, Bus Operation, Arbitration

Stacks

Stack

- A stack is a special type of data structure where elements are inserted from one end and elements are deleted from the same end. This end is called the **top of the stack**
- The various operations performed on stack:
  - Insert: An element is inserted from top end. Insertion operation is called **push operation**.
  - Delete: An element is deleted from top end. Deletion operation is called **pop operation.**
- A processor-register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the **Stack Pointer**.
- If we assume a byte-addressable memory with 32-bit word length
  - The Push operation can be implemented as

```
Subtract SP, SP, #4
Store    Rj, (SP)
```

  - The Pop operation can be implemented as

```
Load Rj, (SP)
Add  SP, SP, #4
```

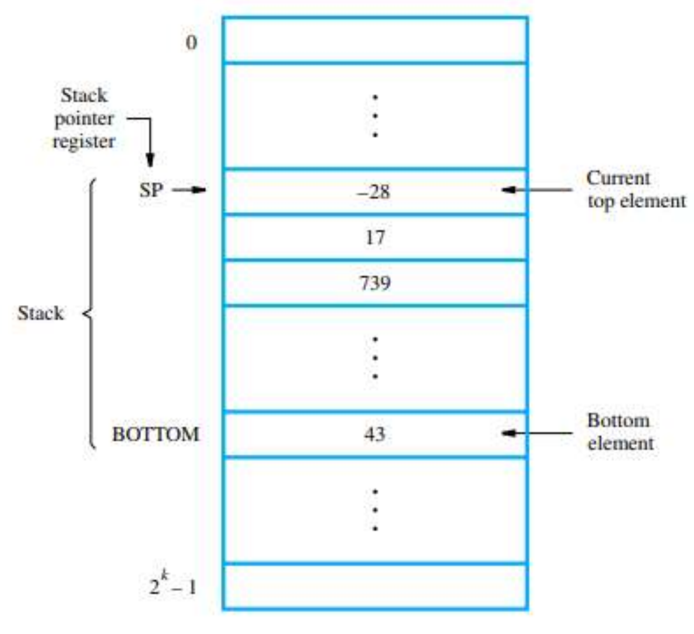
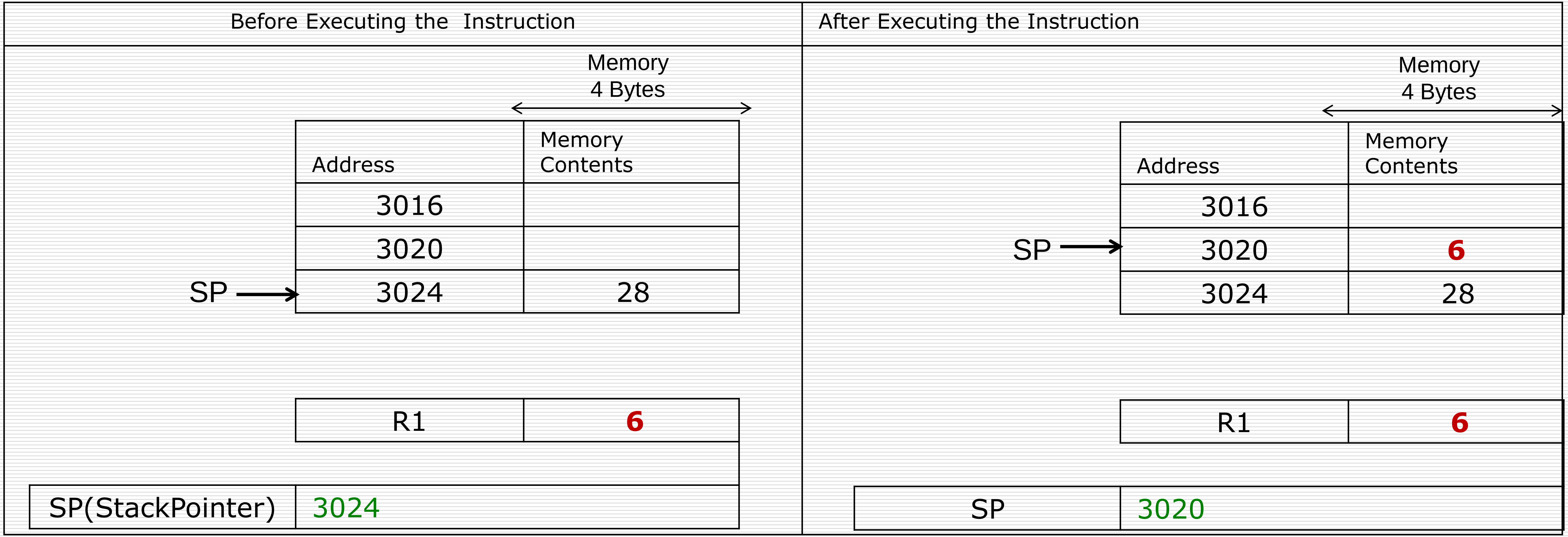


Figure A stack of words in the memory.

Stack: **Push** Operation Implementation

Subtract SP, SP, #4
Store R1, (SP)



Before Executing the Instruction

Memory
4 Bytes

	Address	Memory Contents
	3016	
	3020	
SP →	3024	28

R1	**6**

SP(StackPointer)	3024

After Executing the Instruction

Memory
4 Bytes

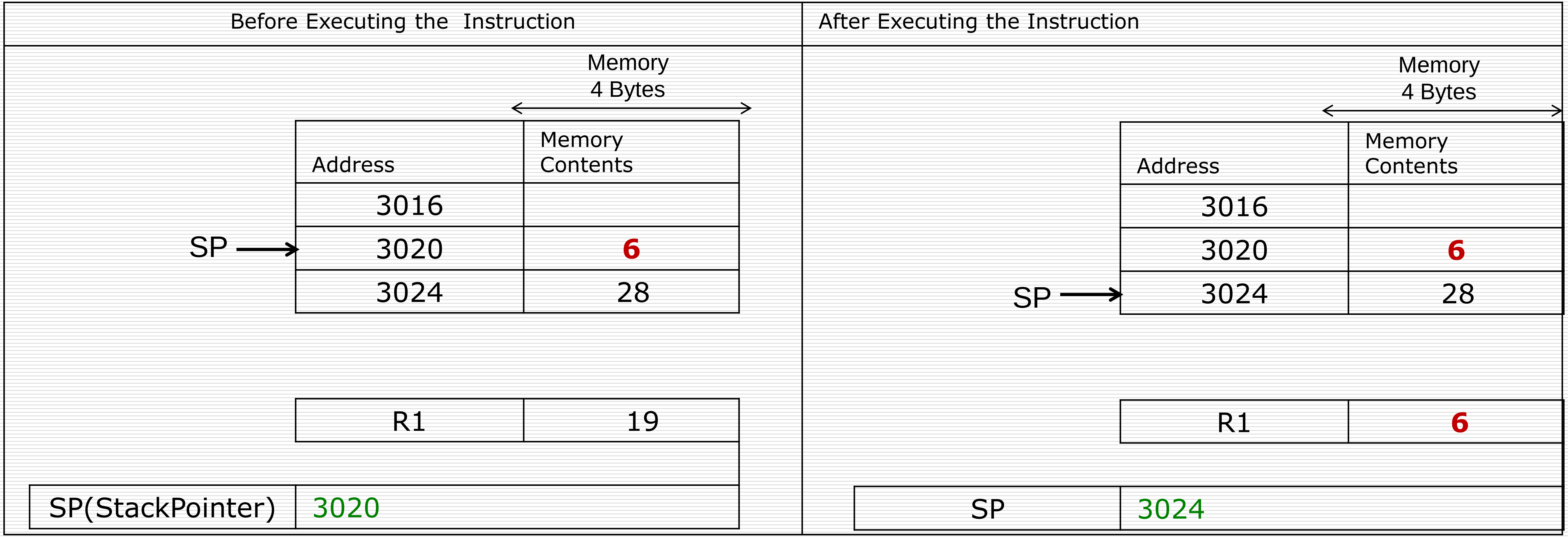
	Address	Memory Contents
	3016	
SP →	3020	**6**
	3024	28

R1	**6**

SP	3020

Stack: **Pop** Operation Implementation

Load R1, (SP)
Add SP, SP, #4



Before Executing the Instruction

Memory
4 Bytes

Address	Memory Contents
3016	
3020	**6**
3024	28

SP → 3020

R1	19

SP(StackPointer)	3020

After Executing the Instruction

Memory
4 Bytes

Address	Memory Contents
3016	
3020	**6**
3024	28

SP → 3024

R1	**6**

SP	3024

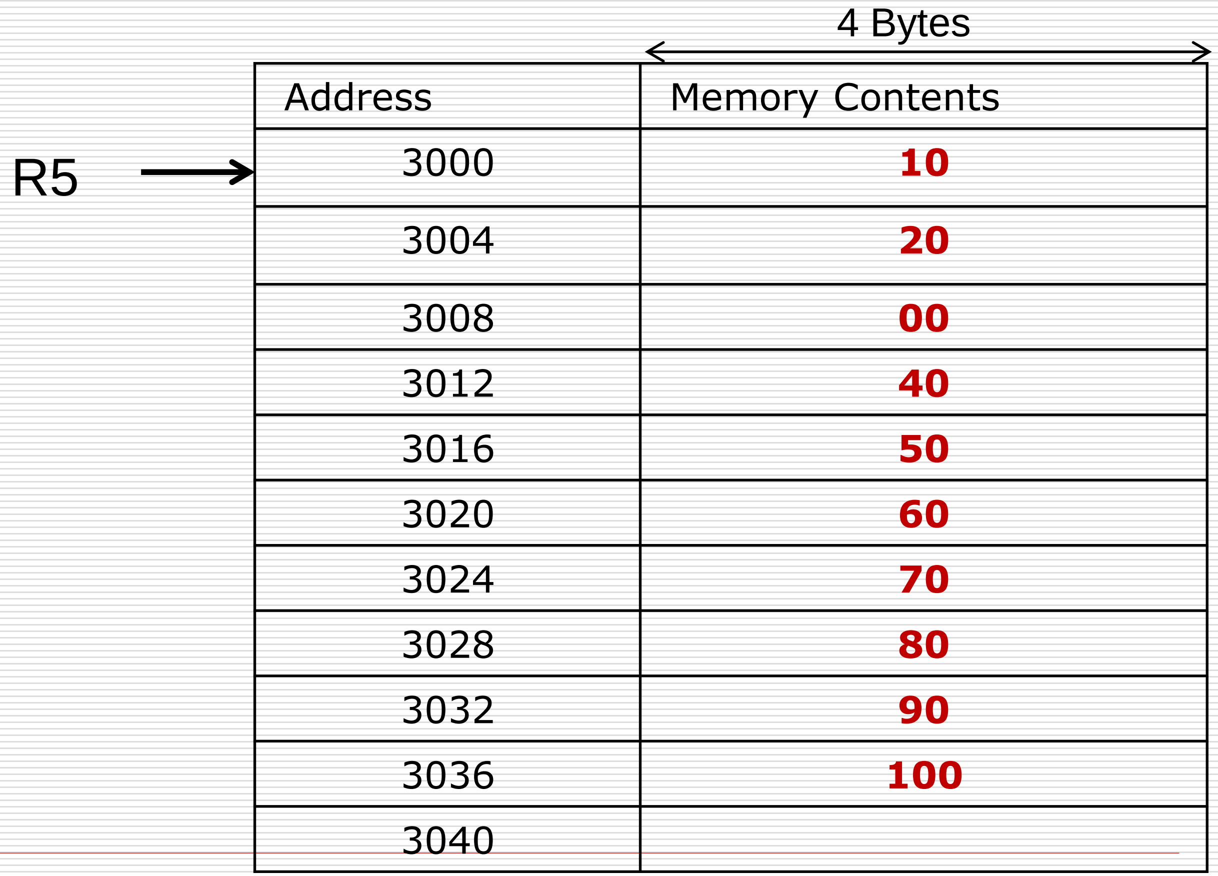
Question

Register R5 is used in a program to point to top of a stack. Consider each word length in stack is of 32-bits.Write a sequence of instructions to perform each of the following

a. Pop the top two items off the stack, add them and then push the result onto the stack.

b. Copy the fifth item from the top of the stack into register R3

c. Remove the top ten items from stack



Address	Memory Contents
3000	**10**
3004	**20**
3008	**00**
3012	**40**
3016	**50**
3020	**60**
3024	**70**
3028	**80**
3032	**90**
3036	**100**
3040	

Note:

Push operation can be implemented as

```
        Subtract        SP, SP, #4
        Store           Rj, (SP)
```

Pop operation can be implemented as

```
        Load Rj, (SP)
        Add  SP, SP, #4
```

Answer

Register R5 is used in a program to point to top of a stack. Consider each word length in stack is of 32-bits.Write a sequence of instructions to perform each of the following

a. Pop the top two items off the stack, add them and then push the result onto the stack.

b. Copy the fifth item from the top of the stack into register R3

c. Remove the top ten items from stack

a.

```
Load R1, (R5)        ; Pop
Add R5, R5, #4
Load R2, (R5)        ; Pop
Add R5, R5, #4
Add R3, R1, R2        ; Add
Subtract R5, R5, #4   ; Push
Store R3, (R5)
```

Note:

```
Push operation can be implemented as
        Subtract      SP, SP, #4
        Store         Rj, (SP)
Pop operation can be implemented as
        Load Rj, (SP)
        Add  SP, SP, #4
```

Contents of Memory after executing set of instruction

		← 4 Bytes →
	Address	Memory Contents
	3000	**10**
R5 →	3004	**30**
	3008	**00**
	3012	**40**
	3016	**50**
	3020	**60**
	3024	**70**
	3028	**80**
	3032	**90**
	3036	**100**
	3040	

Answer

Register R5 is used in a program to point to top of a stack. Consider each word length in stack is of 32-bits.Write a sequence of instructions to perform each of the following

a. Pop the top two items off the stack, add them and then push the result onto the stack.

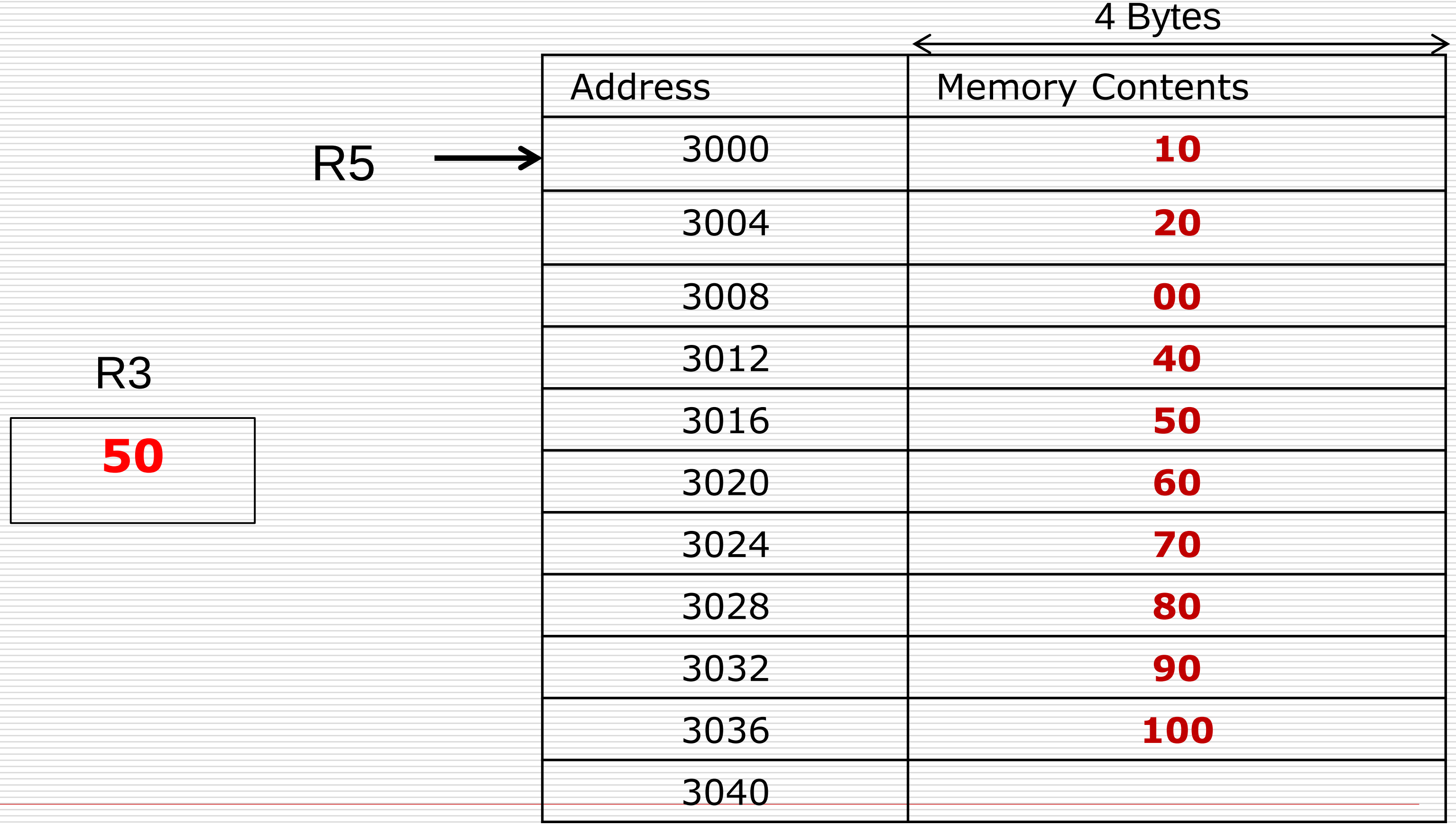
b. Copy the fifth item from the top of the stack into register R3

c. Remove the top ten items from stack

b.

Load R3, 16(R5)

Contents of Memory after executing set of instructions



Address	Memory Contents
3000	**10**
3004	**20**
3008	**00**
3012	**40**
3016	**50**
3020	**60**
3024	**70**
3028	**80**
3032	**90**
3036	**100**
3040	

Note:
Push operation can be implemented as

```
Subtract      SP, SP, #4
Store         Rj, (SP)
```

Pop operation can be implemented as

```
Load Rj, (SP)
Add  SP, SP, #4
```

Answer

Register R5 is used in a program to point to top of a stack. Consider each word length in stack is of 32-bits.Write a sequence of instructions to perform each of the following

a. Pop the top two items off the stack, add them and then push the result onto the stack.

b. Copy the fifth item from the top of the stack into register R3

c. Remove the top ten items from stack

c.

Add R5, R5, #40

Contents of Memory after executing set of instruction

4 Bytes

	Address	Memory Contents
	3000	**10**
	3004	**20**
	3008	**00**
	3012	**40**
	3016	**50**
	3020	**60**
	3024	**70**
	3028	**80**
	3032	**90**
	3036	**100**
R5 →	3040	

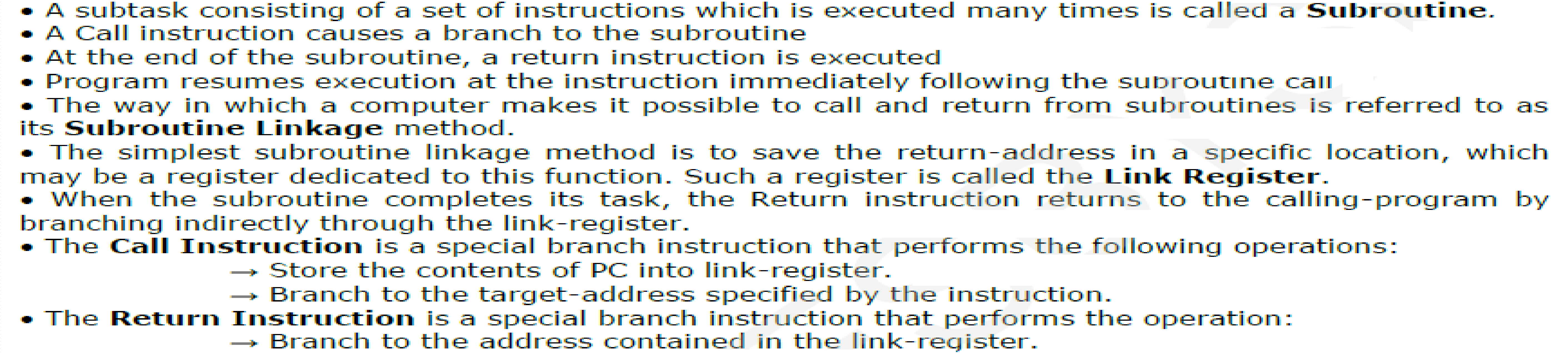
Note:
Push operation can be implemented as

```
    Subtract    SP, SP, #4
    Store       Rj, (SP)
```

Pop operation can be implemented as

```
    Load Rj, (SP)
    Add  SP, SP, #4
```

Subroutines

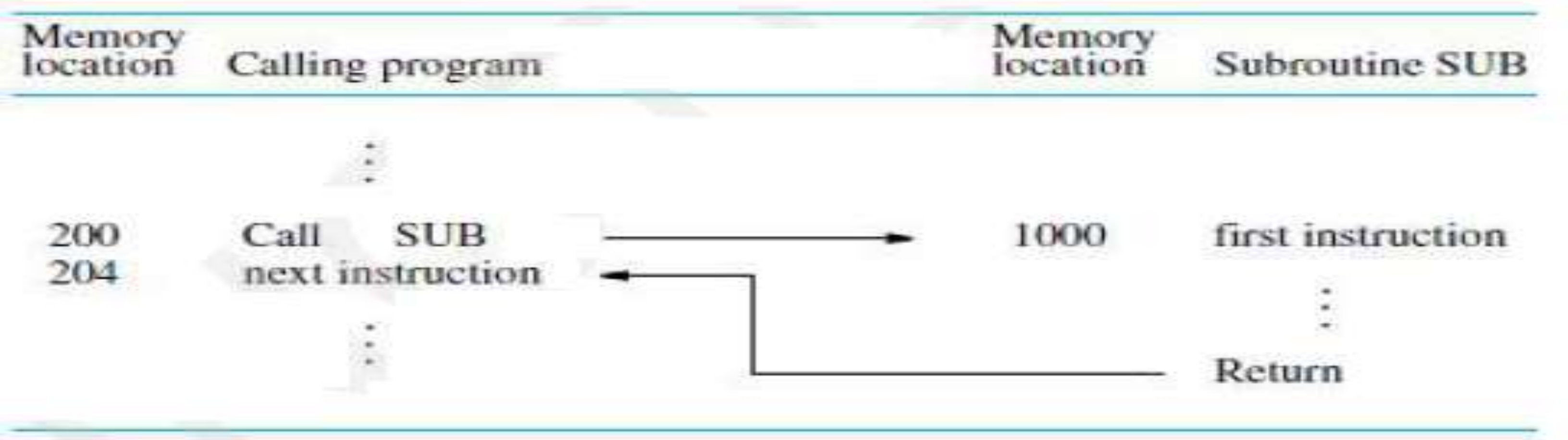


- A subtask consisting of a set of instructions which is executed many times is called a **Subroutine.**
- A Call instruction causes a branch to the subroutine
- At the end of the subroutine, a return instruction is executed
- Program resumes execution at the instruction immediately following the subroutine call
- The way in which a computer makes it possible to call and return from subroutines is referred to as its **Subroutine Linkage** method.
- The simplest subroutine linkage method is to save the return-address in a specific location, which may be a register dedicated to this function. Such a register is 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 a special branch instruction that performs the following operations:

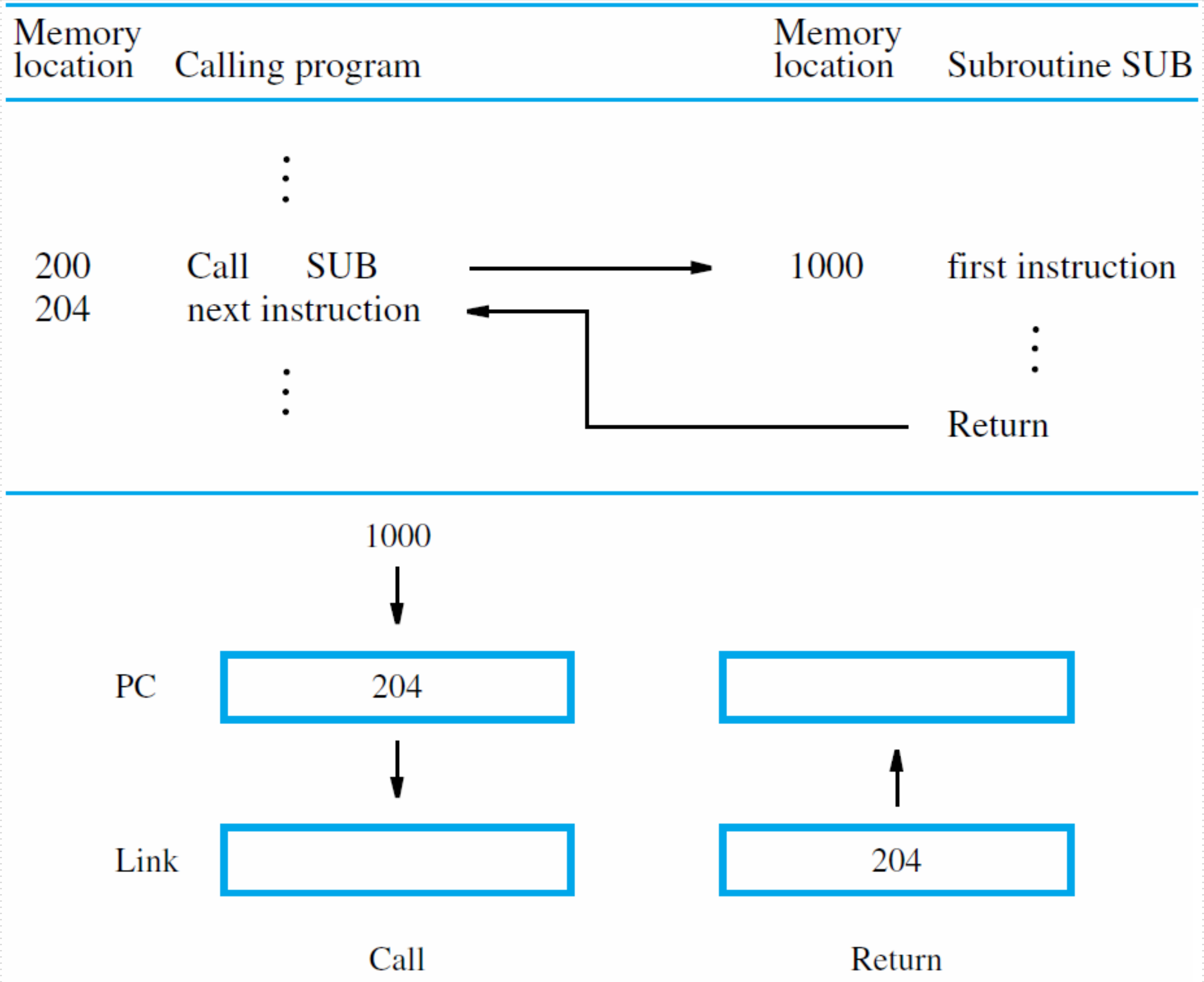
  → Store the contents of PC into link-register.

  → Branch to the target-address specified by the instruction.
- The **Return Instruction** is a special branch instruction that performs the operation:

  → Branch to the address contained in the link-register.

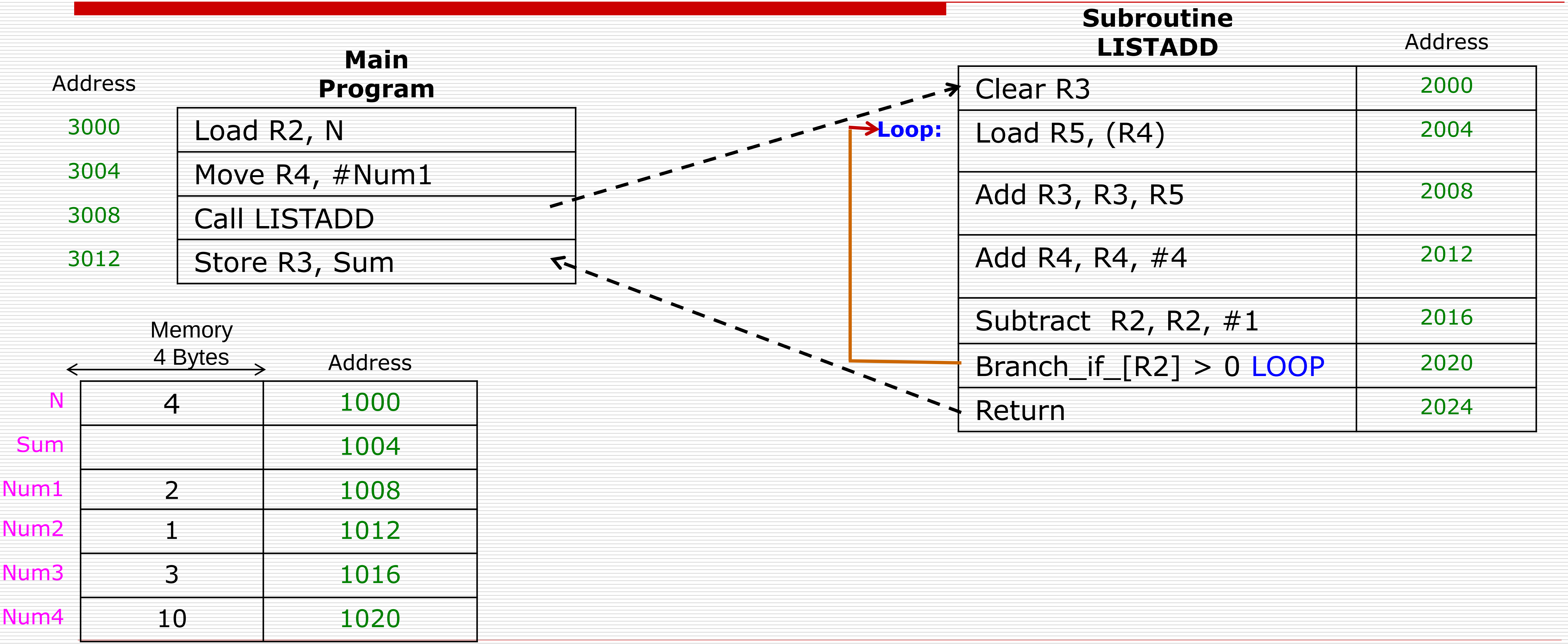


Subroutine Linkage using a Link register



Memory location	Calling program	Memory location	Subroutine SUB
	⋮		
200	Call SUB	1000	first instruction
204	next instruction		⋮
	⋮		Return

Subroutine: Program to add n numbers

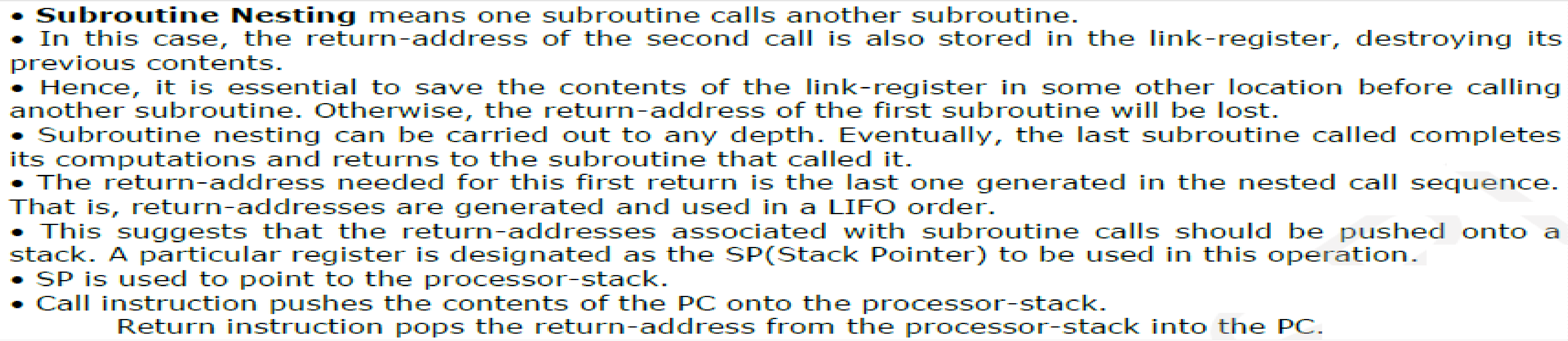


Address	Main Program
3000	Load R2, N
3004	Move R4, #Num1
3008	Call LISTADD
3012	Store R3, Sum

	Subroutine LISTADD	Address
	Clear R3	2000
Loop:	Load R5, (R4)	2004
	Add R3, R3, R5	2008
	Add R4, R4, #4	2012
	Subtract R2, R2, #1	2016
	Branch_if_[R2] > 0 LOOP	2020
	Return	2024

	Memory 4 Bytes	Address
N	4	1000
Sum		1004
Num1	2	1008
Num2	1	1012
Num3	3	1016
Num4	10	1020

Subroutine Nesting



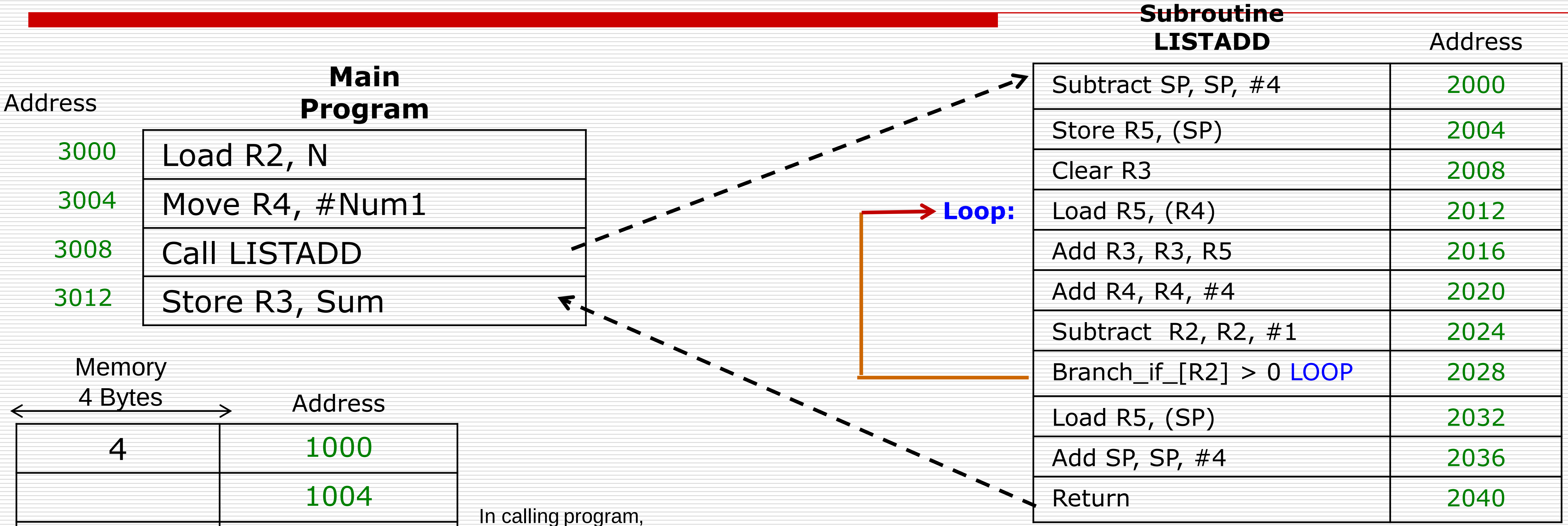
- **Subroutine Nesting** means one subroutine calls another subroutine.
- In this case, the return-address of the second call is also stored in the link-register, destroying its previous contents.
- Hence, it is essential to save the contents of the link-register in some other location before calling 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. That is, return-addresses are generated and used in a LIFO order.
- This suggests that the return-addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the SP(Stack Pointer) to be used in this operation.
- SP is used to point to the processor-stack.
- Call instruction pushes the contents of the PC onto the processor-stack.

  Return instruction pops the return-address from the processor-stack into the PC.

Subroutines: Parameter Passing

- The exchange of information between a calling program and a subroutine is referred to as Parameter passing.
- The parameters may be passed using
  - Registers
  - Memory Location
  - Stack

Subroutine: Passing of Parameters through Register and Memory Location



Address	Main Program
3000	Load R2, N
3004	Move R4, #Num1
3008	Call LISTADD
3012	Store R3, Sum

	Subroutine LISTADD	Address
	Subtract SP, SP, #4	2000
	Store R5, (SP)	2004
	Clear R3	2008
Loop:	Load R5, (R4)	2012
	Add R3, R3, R5	2016
	Add R4, R4, #4	2020
	Subtract R2, R2, #1	2024
	Branch_if_[R2] > 0 LOOP	2028
	Load R5, (SP)	2032
	Add SP, SP, #4	2036
	Return	2040

	Memory 4 Bytes	Address
N	4	1000
Sum		1004
Num1	2	1008
Num2	1	1012
Num3	3	1016
Num4	10	1020

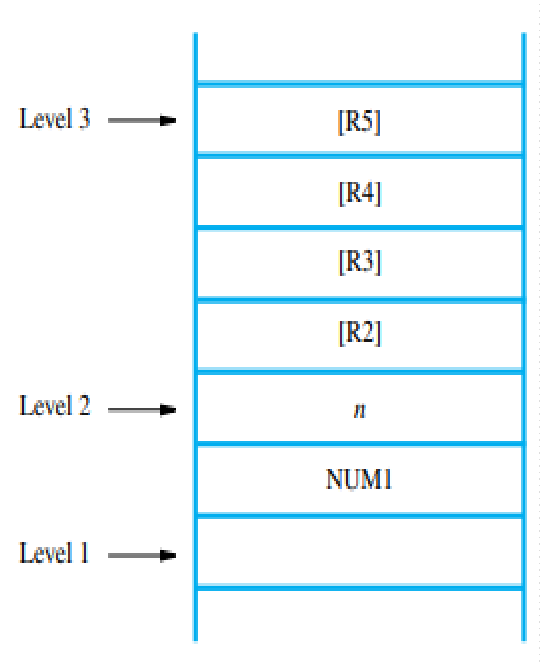
In calling program,
Register R2 and R4 is used to pass the parameters N and address of Num1 location

- N is passed using: **Pass by Value**
- Address of Num1 is passed using: **Pass by Reference.**

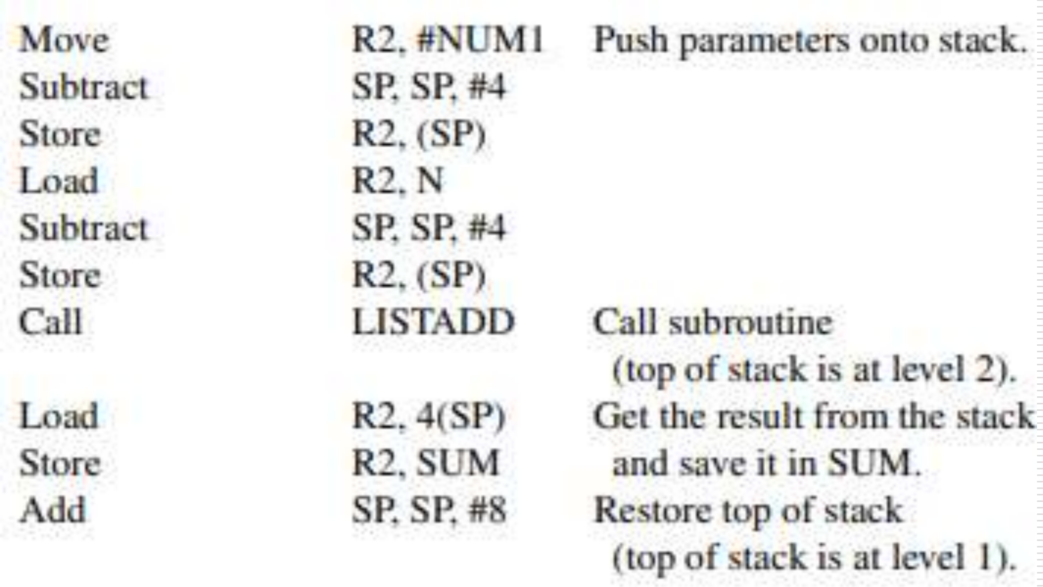
The called Subroutine: LISTADD, returns the parameter total value through the register R3

Subroutine: Passing of Parameters through Stack

		Address
N	3	1000
Sum		1004
NUM1	2	1008
NUM2	1	1012
NUM3	3	1016

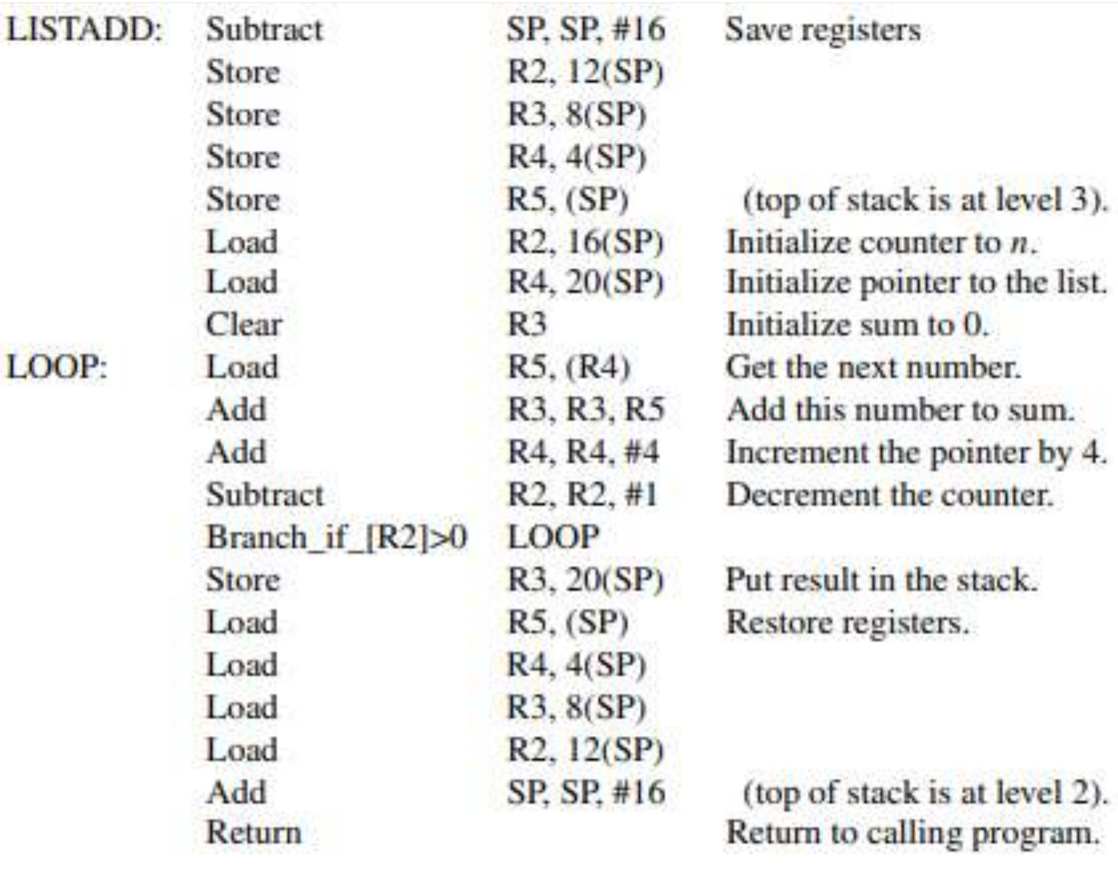


Main Program



```
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).
```

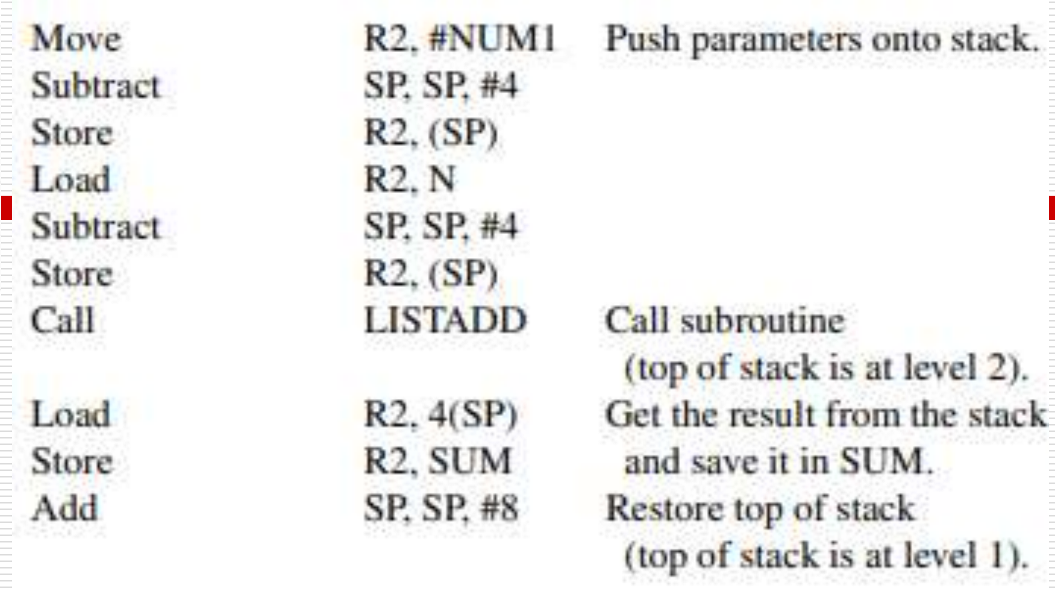
Subroutine LISTADD



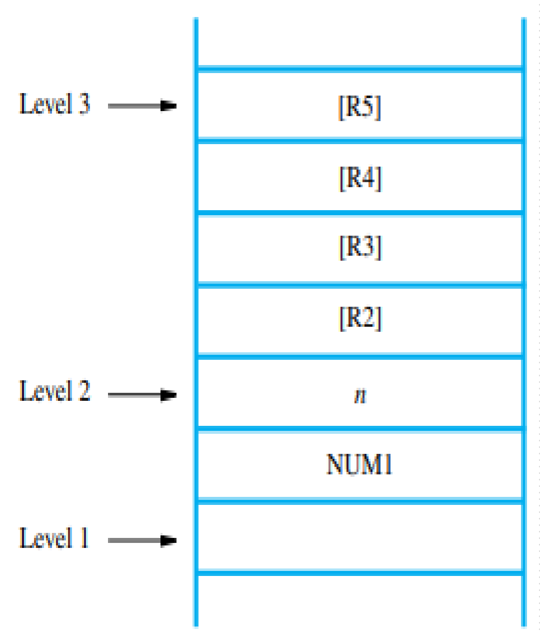
```
LISTADD:  Subtract           SP, SP, #16    Save registers
          Store              R2, 12(SP)
          Store              R3, 8(SP)
          Store              R4, 4(SP)
          Store              R5, (SP)         (top of stack is at level 3).
          Load               R2, 16(SP)     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.
```

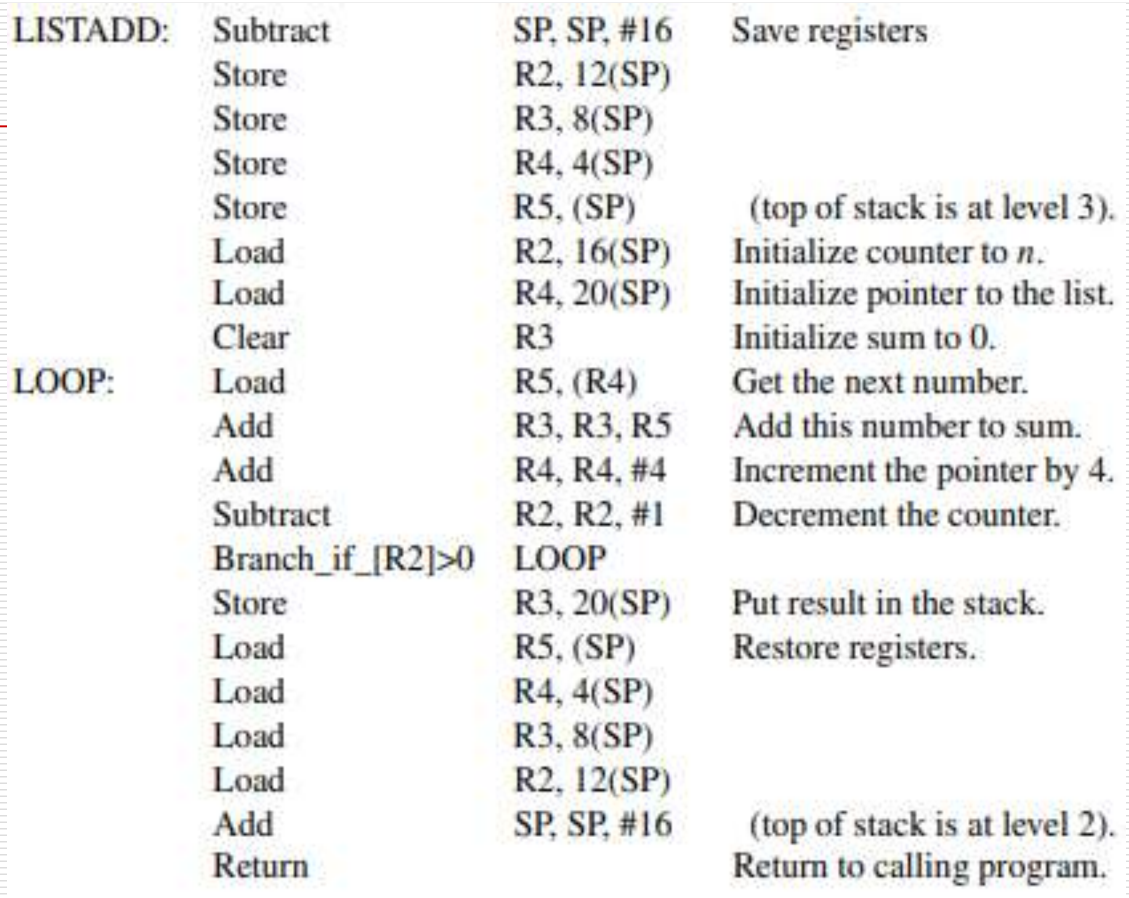
Subroutine LISTADD

		Address
N	3	1000
Sum		1004
NUM1	2	1008
NUM2	1	1012
NUM3	3	1016



```
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.
```

3012	
3016	
3020	
3024	
3028	
3032	
3036	
3040	

R2	
R3	100
R4	200
R5	300

SP
3040

Stack Frame

- **Stack Frame** refers to locations that constitute a private work-space for the subroutine.
- The work-space is
  - Created at the time the subroutine is entered and
  - Freed up when the subroutine returns the control to the calling program

Figure shows an example of a commonly used layout for information in a stack frame. In addition to the Stack Pointer(SP), it is useful to have another pointer register, called the Frame Pointer (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine. In the figure, we assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R2, R3, and R4 need to be saved because they will also be used within the Subroutine.

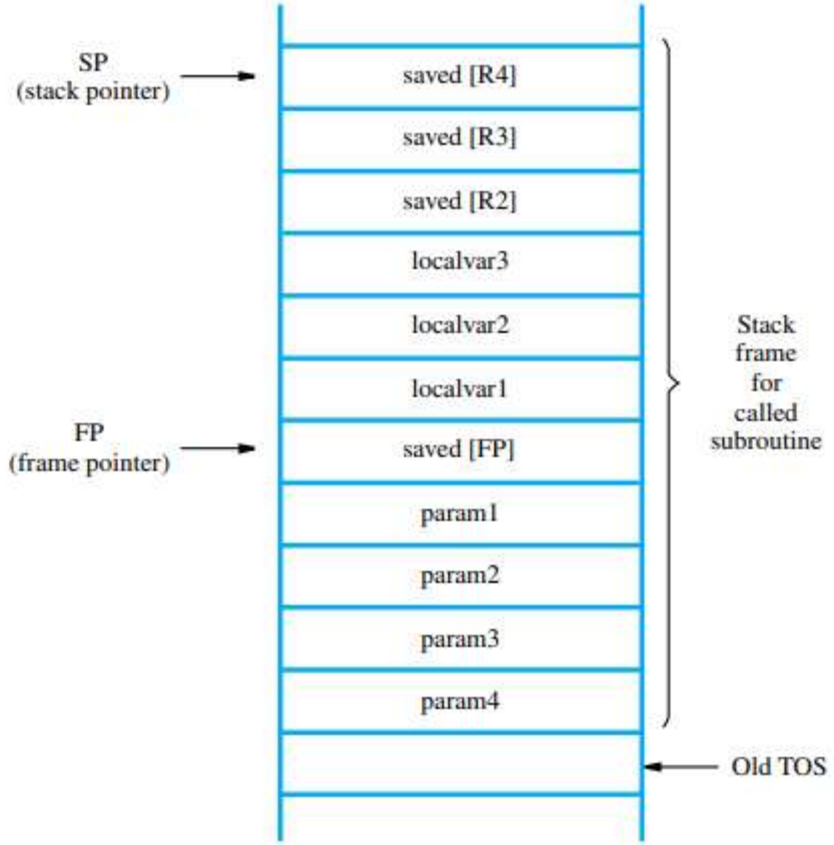


Figure A subroutine stack frame example.

Additional Instructions

So far, we have learned the following instructions:
Load, Store, Move, Clear, Add, Subtract, Branch, Call, and Return.

Next we will learn few more instructions that are found in most instruction sets.

Logic Instructions

- □ Logic operations such as AND, OR, and NOT are applied to individual bits.
- □ For example, the instruction

  And R4, R2, R3

  computes the bit-wise AND of operands in registers R2 and R3, and leaves the result in R4.

- □ An immediate form of this instruction may be

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.

Logic Instruction: Example

And R4, R2, R3

<table>
<tr><th colspan="8">Before Executing the Instruction</th><th colspan="8">After Executing the Instruction</th></tr>
<tr><td colspan="8">R3</td><td colspan="8">R3</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td colspan="8">R2</td><td colspan="8">R2</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">R4</td><td colspan="8">R4</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
</table>

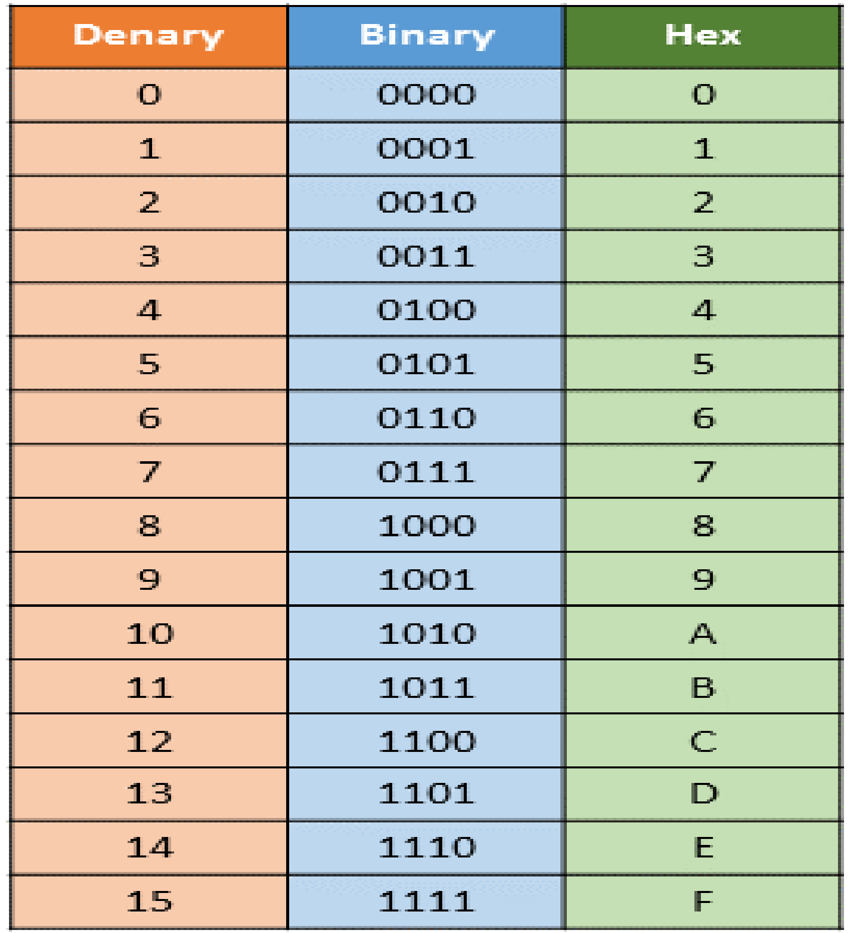
Question

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. Write sequence of instructions for this task

ASCII Hex value	Character	ASCII Hex value	Character
0x41	A	0x4A	J
0x42	B	…..	…..
0x43	C	0x59	X
…	…	0x5A	Z

Hexadecimal numbers

- □ A group of 4 bits can take any value between 0 (0000 binary) and 15 (1111 binary).
- □ In hexadecimal, we replace each group of 4 bits with a single digit to represent the value 0 to 15. Since we only have digits 0 to 9, we use letters A to E to represent values 10 to 15. Here is a table of binary, denary and hex values:



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

Note: To specify Hexadecimal numbers Prefix 0x will be used i.e.,

0x123

or

123**h**

Question

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. Write sequence of instructions for this task

ASCII Hex value	Character	ASCII Hex value	Character
0x41	A	0x4A	J
0x42	B	…..	…..
0x43	C	0x59	X
…	…	0x5A	Z

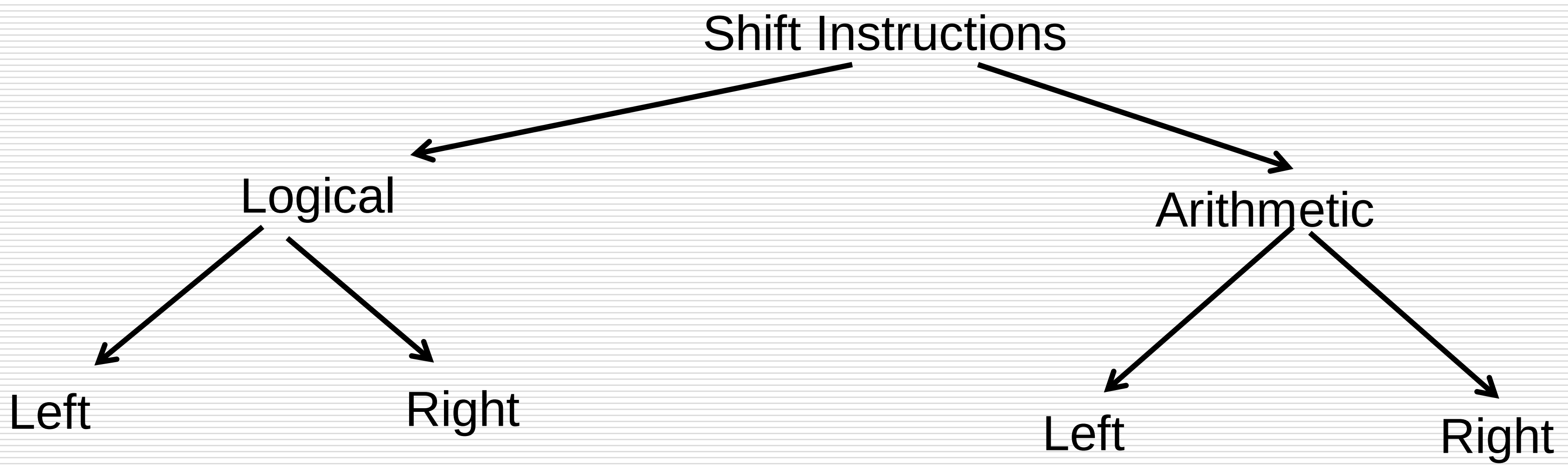
Answer

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. Write sequence of instructions for this task.

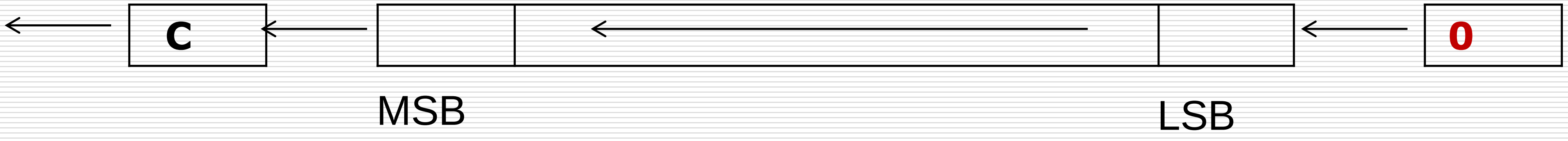
```
And R2, R2, #0xFF
Move R3, #0x5A
Branch_if_[R2]=[R3] FOUNDZ
```

ASCII Hex value	Character	ASCII Hex value	Character
0x41	A	0x4A	J
0x42	B	…..	…..
0x43	C	0x59	X
…	…	0x5A	Z

Shift Instructions



Logical Left Shift Instruction



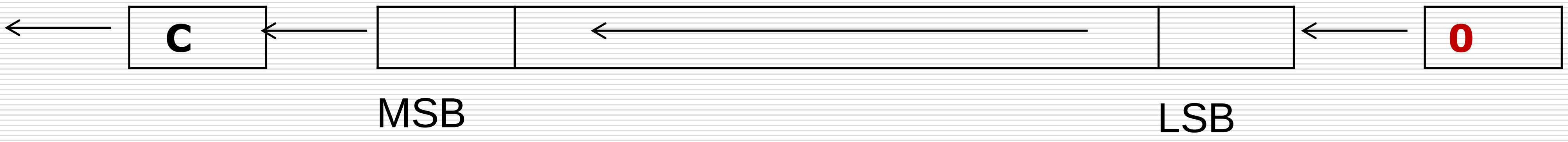
Logical Left Shift
General Syntax: LShiftL Ri, Rj, Count

LShiftL, 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. During shift, bits are passed through the Carry flag, C, and then dropped.

LShiftL R1, R2 , #2

	Before Executing the Instruction	After Executing the Instruction
R2	1 0 1 0 1 1 0 1	1 0 1 0 1 1 0 1
C	**0**	**0**
R1	0 0 0 0 0 0 0 0	1 0 1 1 0 1 **0** **0**

Logical Left Shift Instruction



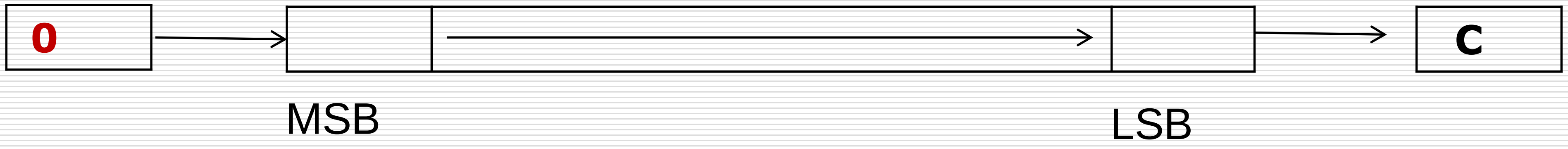
Logical Left Shift

General Syntax: LShiftL Ri, Rj, Count

LShiftL R1, R1 , #2

Before Executing the Instruction		After Executing the Instruction	
C	R1	C	R1
0	1 0 1 0 1 1 0 1	**1**	0 1 0 1 1 0 1 **0**
			After 1st Shift
		C	R1
		0	1 0 1 1 0 1 **0** **0**
			After 2nd Shift

Logical Right Shift Instruction



Logical Right Shift
General Syntax: LShiftR Ri, Rj, Count

LShiftR R1, R1, #2

Before Executing the Instruction	After Executing the Instruction
R1: 1 0 1 0 1 1 0 1 C: 0	R1: **0** 1 0 1 0 1 1 0 C: **1** After 1st Shift R1: **0** **0** 1 0 1 0 1 1 C: **0** After 2nd Shift

Question

The content of a 4-bit register is initially 1101. The register is logically shifted 2 times to the right. What is the content of the register after each shift?

a. 1110, 0111

b. 0001, 1000

c. 1101, 1011

d. 0110, 0011

Question

The content of a 4-bit register is initially 1101. The register is logically shifted 2 times to the right. What is the content of the register after each shift?

a. 1110, 0111

b. 0001, 1000

c. 1101, 1011

d. 0110, 0011

Question

If a register containing data 11001100 is subjected to logical shift left operation of 1 bit, then the content of the register after 'LshiftL' shall be

a. 01100110

b. 10011001

c. 11011001

d. 10011000

Answer

If a register containing data 11001100 is subjected to logical shift left operation of 1 bit, then the content of the register after 'LshiftL' shall be

a. 01100110

b. 10011001

c. 11011001

d. 10011000

Question

Suppose that two decimal digits represented in ASCII code are located in the memory at byte locations LOC and LOC + 1. We wish to represent each of these digits in the 4-bit BCD code and store both of them in a single byte location PACKED. The result is said to be in *packed-BCD* format. Write sequence of instructions for this task.

Hint: Use shift and logic instructions. Consider in memory, each memory location capacity is one of byte.

ASCII Hex value	Number
0x31	1
0x32	2
0x33	3
…	…
0x39	9

LOC	0x39
LOC+1	0x32
PACKED	92

Note: Binary Coded Decimal system(BCD): Each decimal digit is represented by a group of 4 bits.

Answer

Suppose that two decimal digits represented in ASCII code are located in the memory at byte locations LOC and LOC + 1. We wish to represent each of these digits in the 4-bit BCD code and store both of them in a single byte location PACKED. The result is said to be in *packed-BCD* format. Write sequence of instructions for this task.

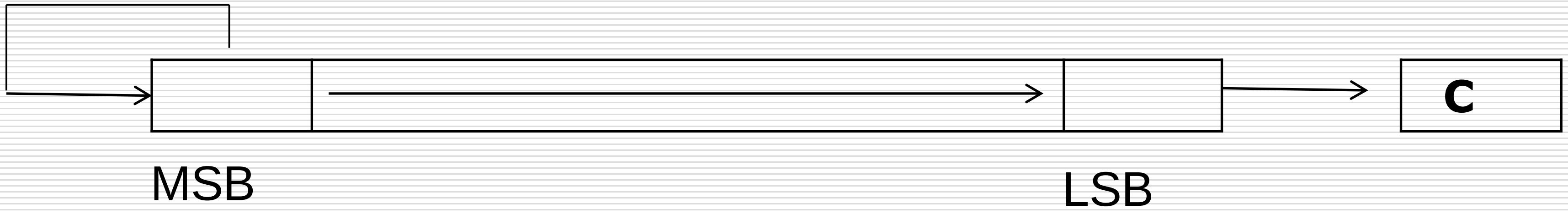
Hint: Use shift and logic instructions. Consider in memory, each memory location capacity is one of byte.

```
Move R2, #LOC
Load  R3, (R2)
LShiftL R3, R3, #4 ;Shift left by 4-bit position
Add R2, R2, #1
Load  R4, (R2)
And R4, R4, #0xF  ;Clear high-order bits to zero.
Or R3, R3, R4        ;Concatenate the BCD digits.
Store  R3, PACKED
```

ASCII Hex value	Number
0x31	1
0x32	2
0x33	3
…	…
0x39	9

LOC	0x39
LOC+1	0x32
PACKED	92

Arithmetic Right Shift Instruction



RShiftR R1, R1, #2

Before Executing the Instruction

R1								C
1	0	1	0	1	1	0	1	**0**

After Executing the Instruction

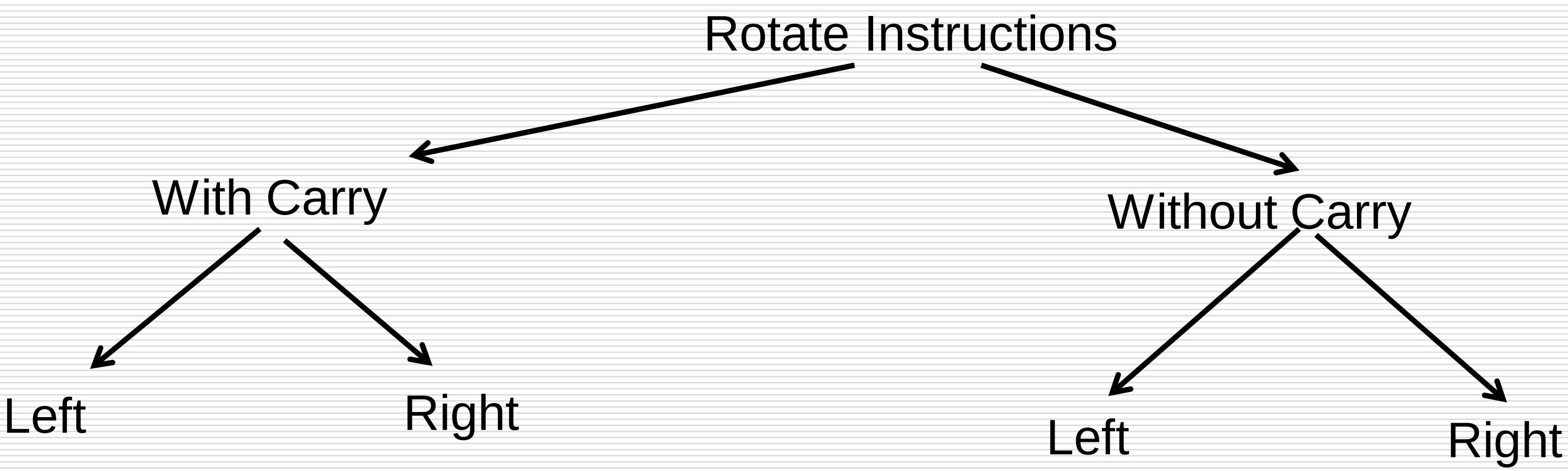
R1								C
1	1	0	1	0	1	1	0	**1**

After 1st Shift

R1								C
1	**1**	1	0	1	0	1	1	**0**

After 2nd Shift

Rotate Instructions


Rotate Instructions

With Carry

Without Carry

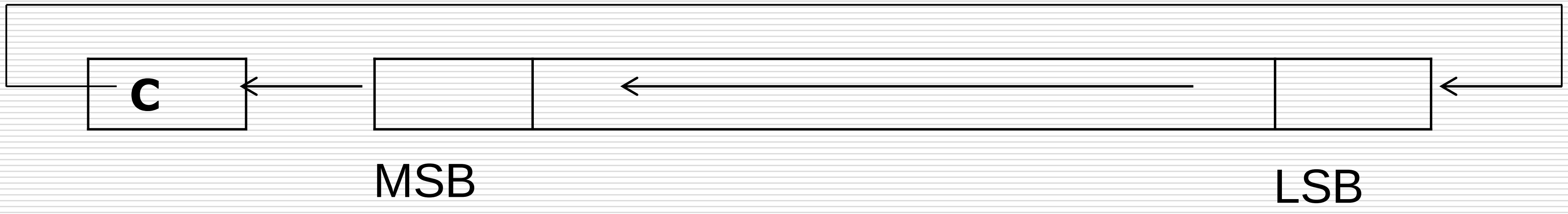
Left

Right

Left

Right

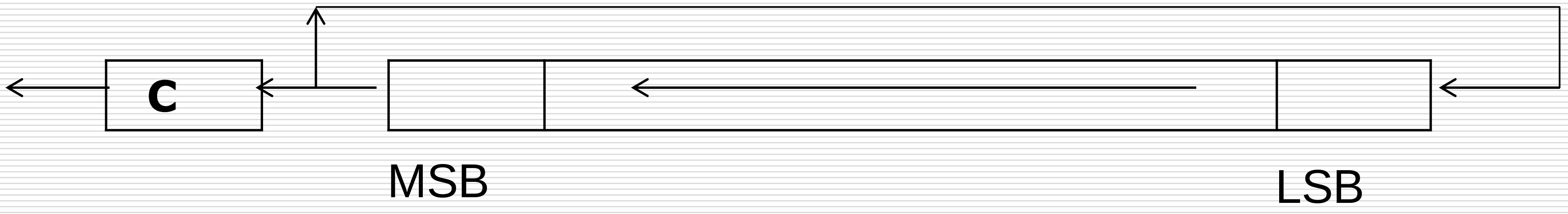
Rotate Left with Carry



RotateLC R1, R1, #1

Before Executing the Instruction									After Executing the Instruction								
C	R1								C	R1							
0	**1**	0	1	1	0	1	0	0	**1**	0	1	1	0	1	0	0	**0**

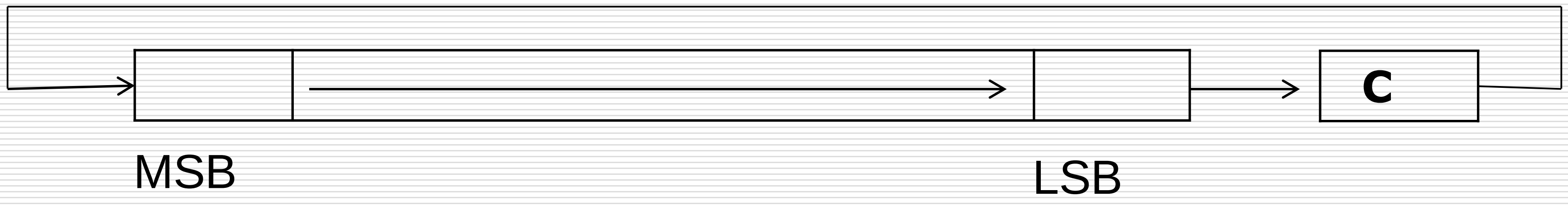
Rotate Left without Carry



RotateL R1, R1, #1

Before Executing the Instruction		After Executing the Instruction	
C	R1	C	R1
0	1 0 1 1 0 1 0 0	1	0 1 1 0 1 0 0 1

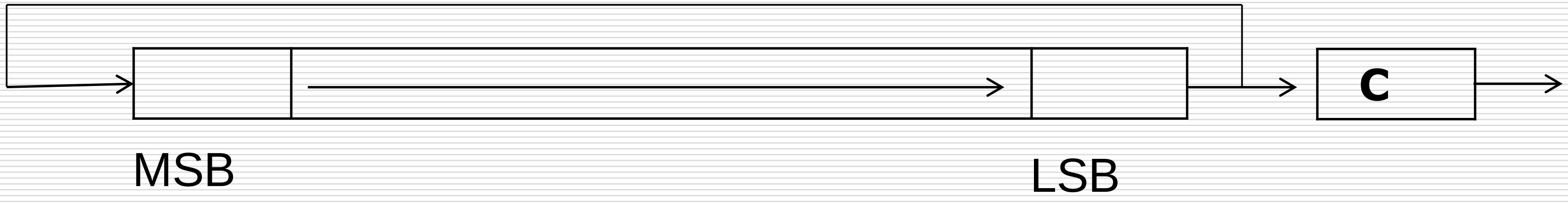
Rotate Right with Carry



RotateRC R1, R1, #1

Before Executing the Instruction									After Executing the Instruction								
R1								C	R1								C
1	0	1	1	0	1	0	**0**	**0**	**0**	1	0	1	1	0	1	0	**0**

Rotate Right without Carry



RotateL R1, R1, #1

Before Executing the Instruction		After Executing the Instruction	
R1	C	R1	C
1 0 1 1 0 1 0 **0**	0	**0** 1 0 1 1 0 1 0	**0**

Multiplication and Division Instructions

- Signed integer multiplication of *n*-bit numbers produces a product with up to 2*n* bits
- Processor truncates product to fit in a register:
  Multiply Rk, Ri, Rj (Rk ← [Ri] × [Rj])
- For general case, 2 registers may hold result
- Integer division produces quotient as result:
  Divide Rk, Ri, Rj (Rk ← [Ri] / [Rj])
- Remainder is discarded or placed in a register

Example for Multiplication instruction

Multiply R2 , R3, R4 ; R2 ← [R3] x [R4]

Before Executing the Instruction

R2	76
R3	10
R4	2

After Executing the Instruction

R2	**20**
R3	10
R4	2

Example for Divide instruction

Divide R2 , R3, R4 ; R2 ← [R3] / [R4] , Quotient in register R2 and Remainder in R1

Before Executing the Instruction

R1	38
R2	76
R3	10
R4	2

After Executing the Instruction

R1	**0**
R2	**5**
R3	10
R4	2

Basic Input/Output

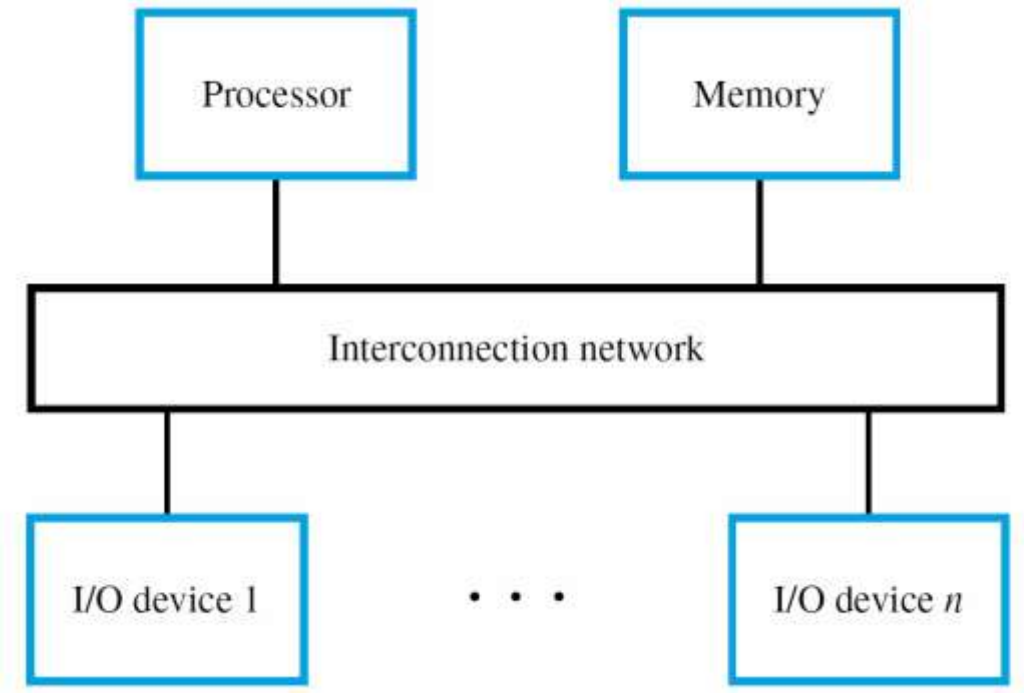
Accessing I/O Devices
Interrupts
Bus Structure
Bus Operation
Arbitration

Accessing Input/Output (I/O) Devices

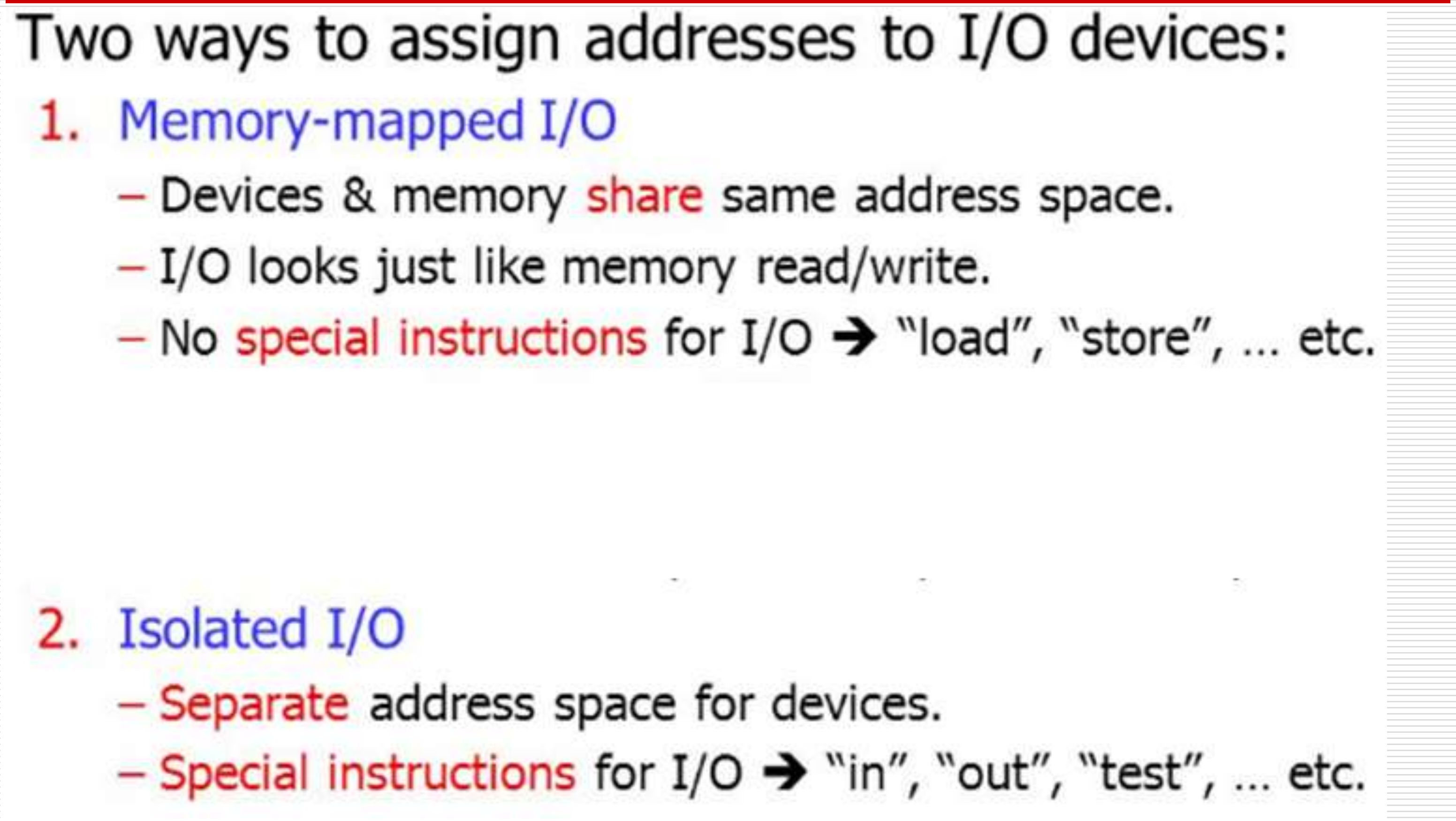
The components of a computer system communicate with each other through an interconnection network, as shown in Figure.

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

Each I/O device is assigned a unique set of addresses



Addressing Techniques of I/O Devices



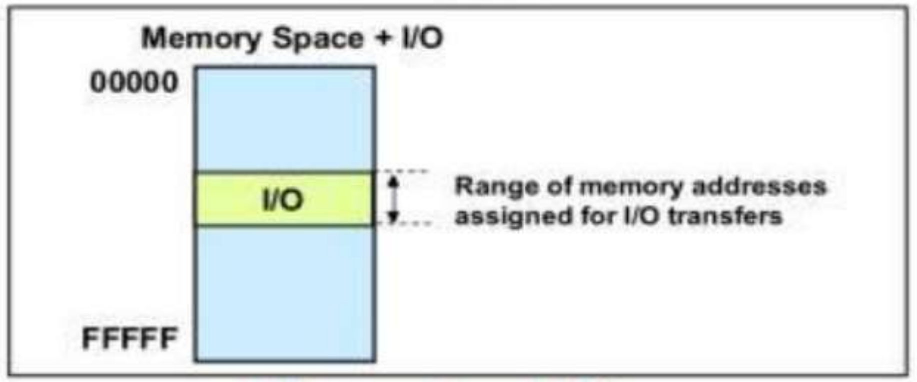
Two ways to assign addresses to I/O devices:

1. Memory-mapped I/O
   - Devices & memory share same address space.
   - I/O looks just like memory read/write.
   - No special instructions for I/O → "load", "store", ... etc.

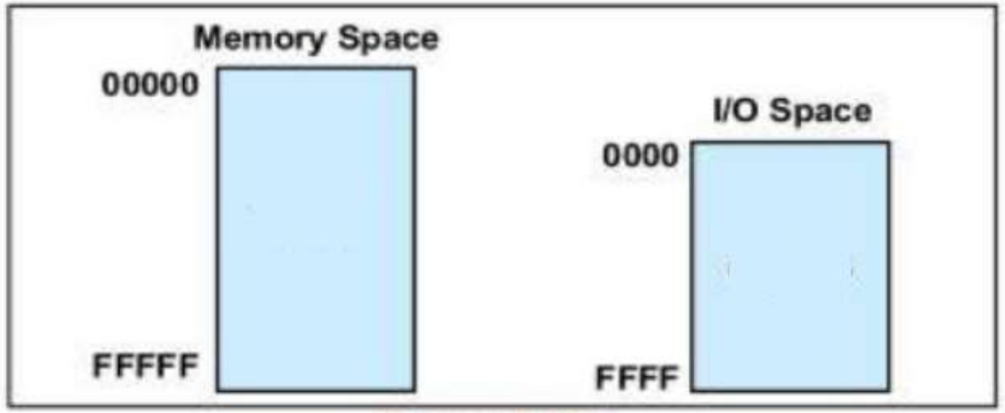
2. Isolated I/O
   - Separate address space for devices.
   - Special instructions for I/O → "in", "out", "test", ... etc.

Addressing Techniques of I/O Devices

1. Memory mapped I/O

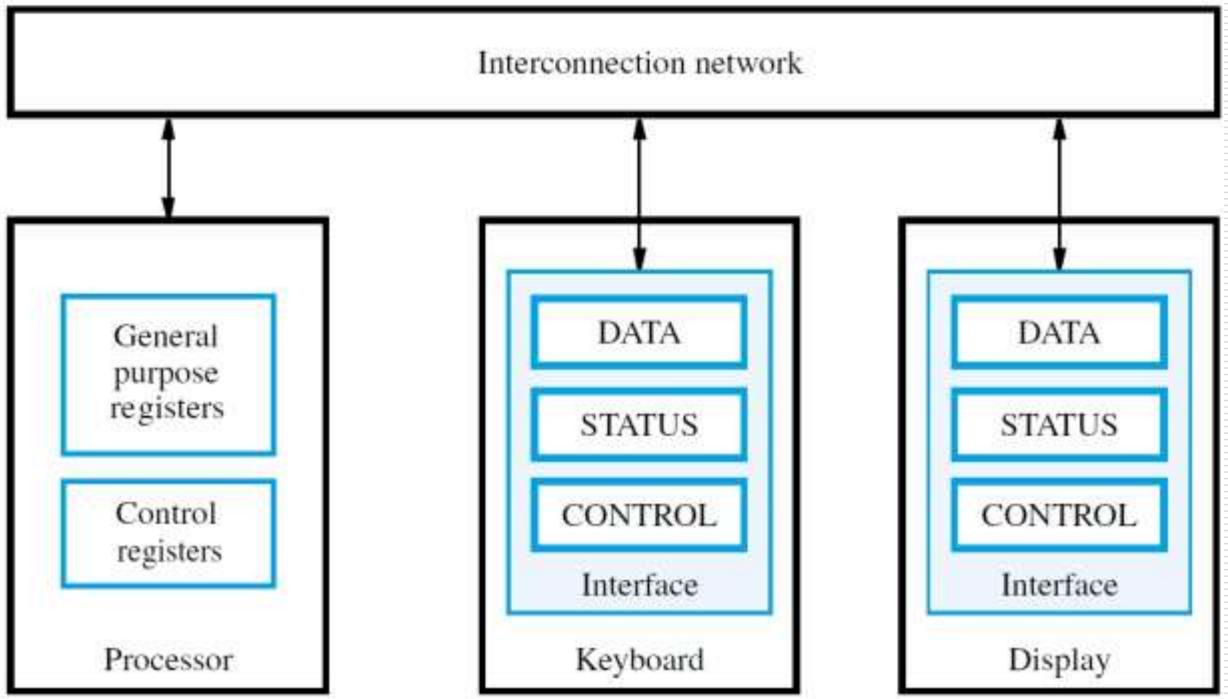


2. I/O (or Isolated) mapped I/O

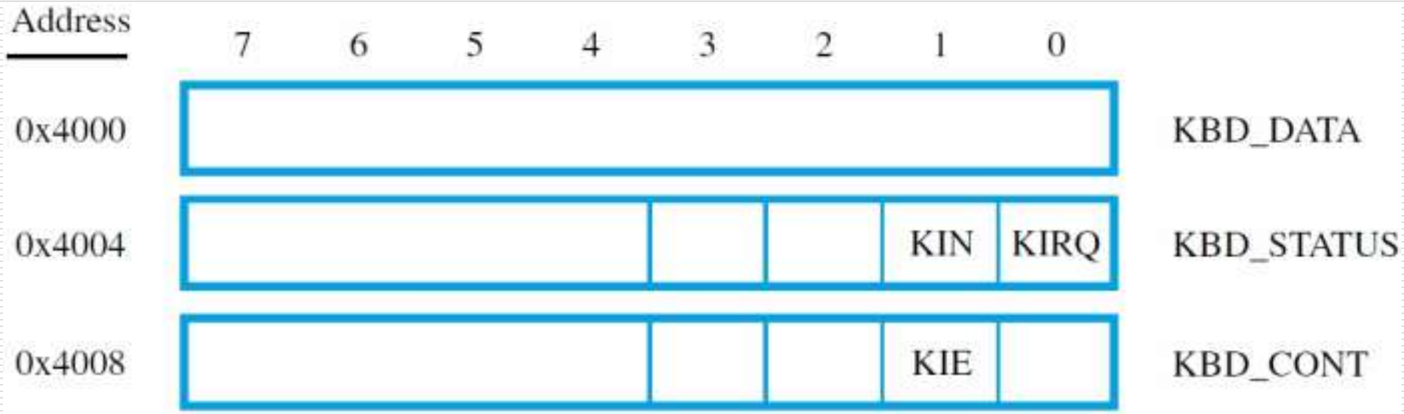


I/O Device Interface

- An I/O device interface is a circuit between a device and the interconnection network
- Provides the means for data transfer and exchange of status and control information
- Includes data, status, and control registers accessible with Load and Store instructions
- Memory-mapped I/O enables software to view these registers as locations in memory

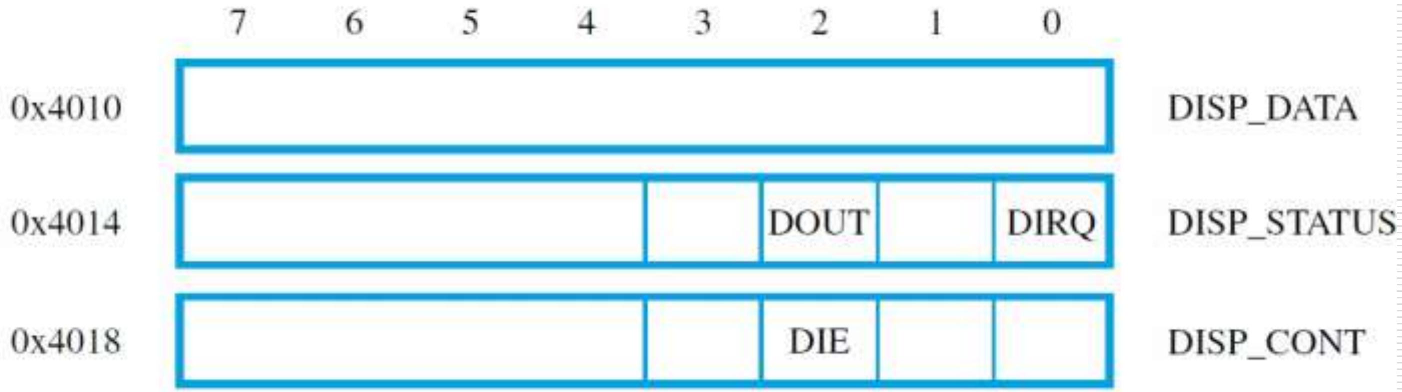


Registers in the Keyboard and Display interface



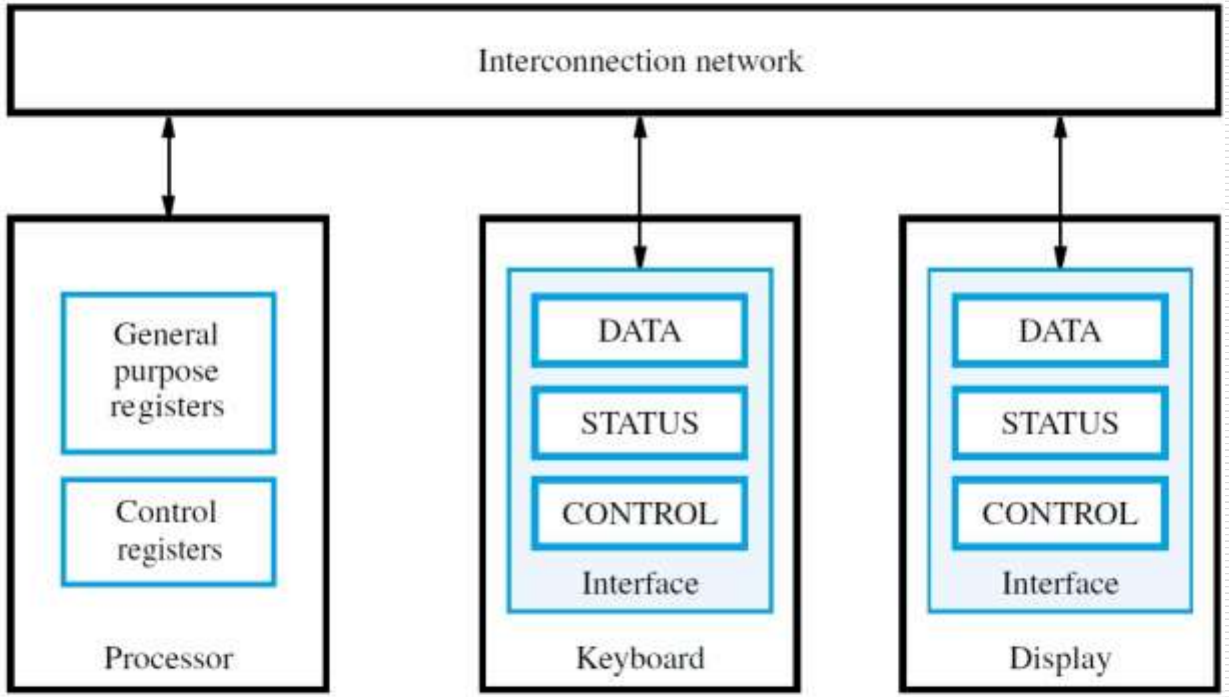
Address	7	6	5	4	3	2	1	0	
0x4000									KBD_DATA
0x4004							KIN	KIRQ	KBD_STATUS
0x4008							KIE		KBD_CONT

(a) Keyboard interface



Address	7	6	5	4	3	2	1	0	
0x4010									DISP_DATA
0x4014						DOUT		DIRQ	DISP_STATUS
0x4018						DIE			DISP_CONT

(b) Display interface



Basic Input/Output Operations

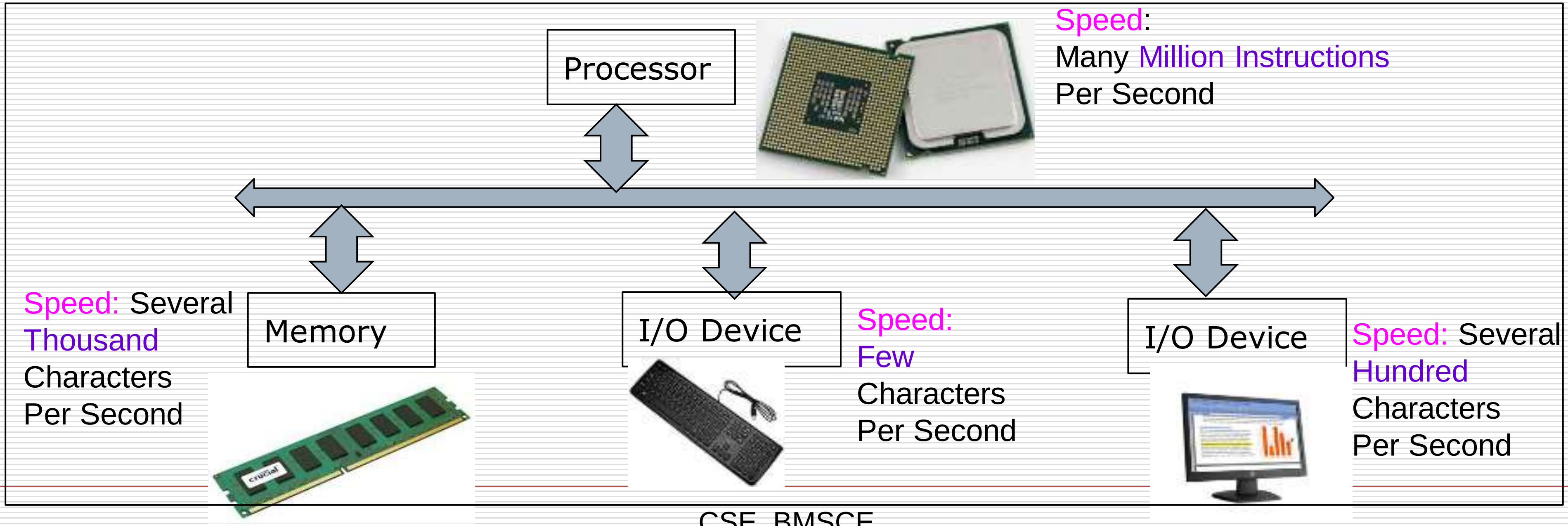
We have seen instructions to:

- Transfer information between the processor and the memory.
- Perform arithmetic and logic operations
- Program sequencing and flow control.

Input/Output operations which transfer data from the processor or memory to and from the real world are essential.

In general, *the rate of transfer from any input device to the processor, or from the processor to any output device is likely to the slower than the speed of a processor.*

The difference in speed makes it necessary to create mechanisms to synchronize the data transfer between them



Modes of I/O Data Transfer

Data transfer between the central unit and I/O devices can be handled in generally three types of modes which are given below:

1. Program Controlled I/O
2. Interrupt Initiated I/O
3. Direct Memory Access

Three modes of I/O data transfer

Programmed I/O

- □ Programmed I/O instructions are the result of I/O instructions written in computer program. Each data item transfer is initiated by the instruction in the program.
- □ Usually the program controls data transfer to and from CPU and peripheral (I/O devices). Transferring data under programmed I/O requires constant monitoring of the peripherals by the CPU.

Interrupt Initiated I/O

- □ In the programmed I/O method the CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This is time consuming process because it keeps the processor busy needlessly.
- □ This problem can be overcome by using **interrupt initiated I/O**. In this when the interface determines that the peripheral is ready for data transfer, it generates an interrupt. After receiving the interrupt signal, the CPU stops the task which it is processing and service the I/O transfer and then returns back to its previous processing task.

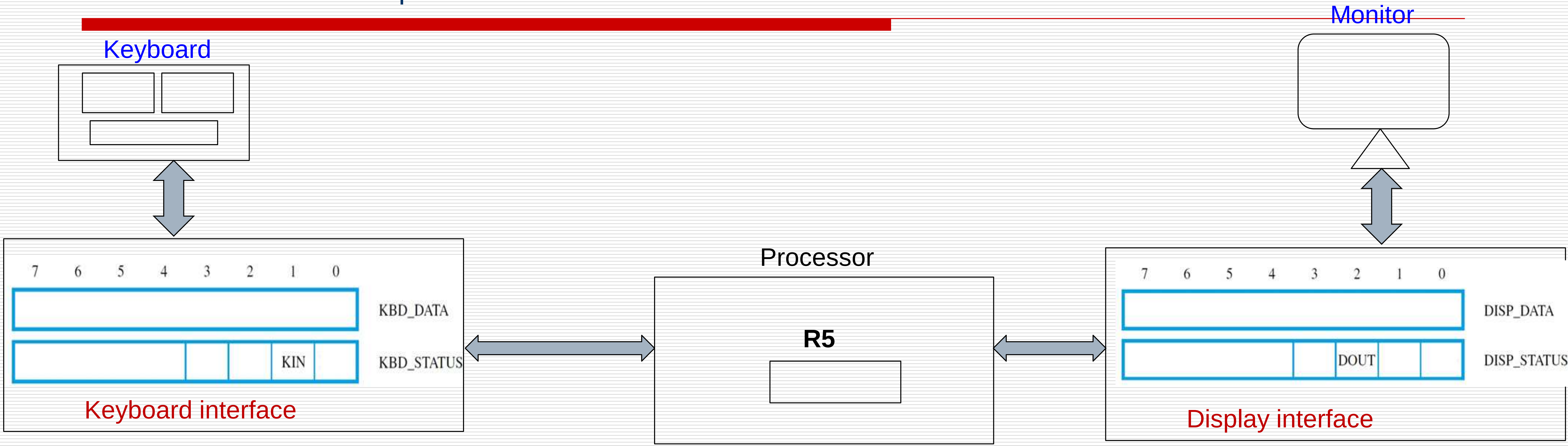
Direct Memory Access

- □ Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This technique is known as **DMA**.
- □ In this, the interface transfer data to and from the memory through memory bus. A DMA controller manages to transfer data between peripherals and memory unit.
- □ Many hardware systems use DMA such as disk drive controllers, graphic cards, network cards and sound cards etc. It is also used for intra chip data transfer in multicore processors. In DMA, CPU would initiate the transfer, do other operations while the transfer is in progress and receive an interrupt from the DMA controller when the transfer has been completed.

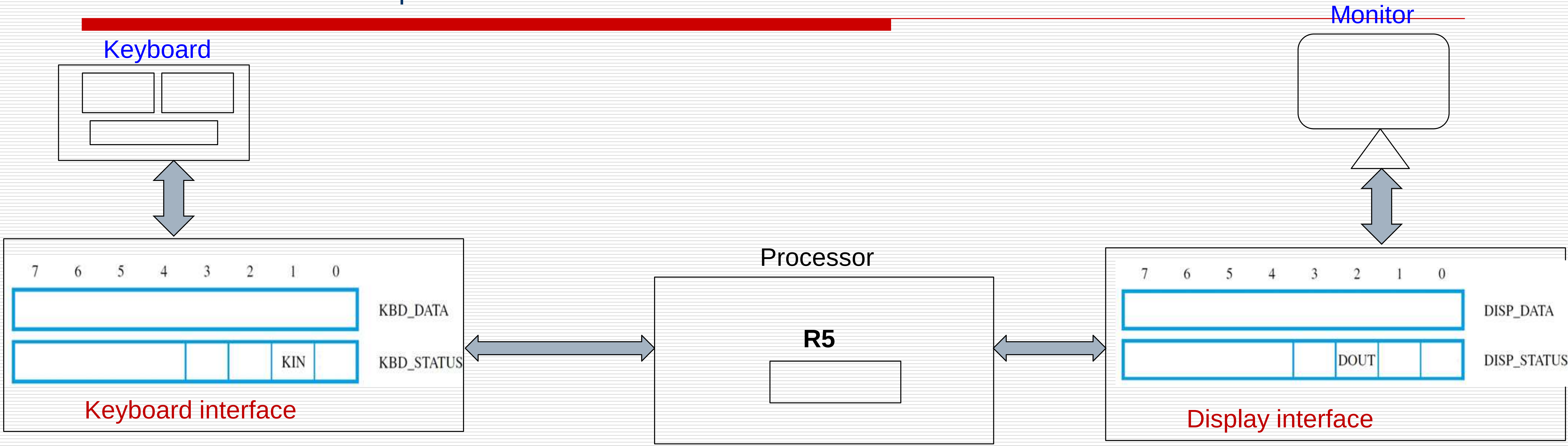
Program Controlled I/O

- Let us consider a simple task of reading a character from a keyboard and displaying that character on a display screen.
- A simple way of performing the task is called program-controlled I/O.
- There are two separate blocks of instructions in the I/O program that perform this task:
  - One block of instructions transfers the character into the processor.
  - Another block of instructions causes the character to be displayed.

Program Controlled I/O:
Data Transfer from Input device to Processor

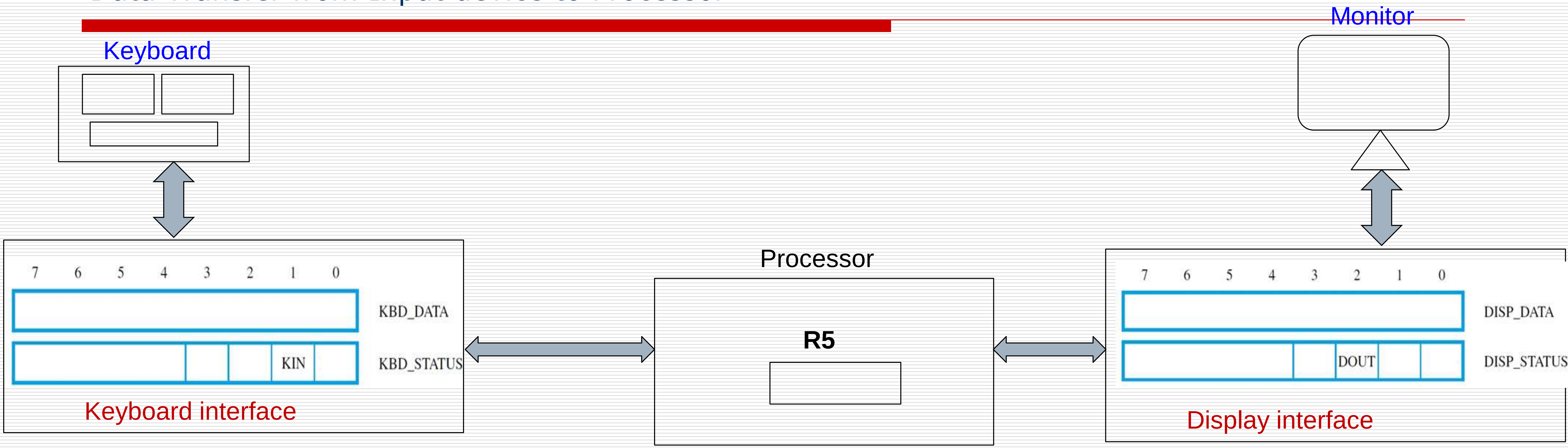


Program Controlled I/O:
Data Transfer from Input device to Processor



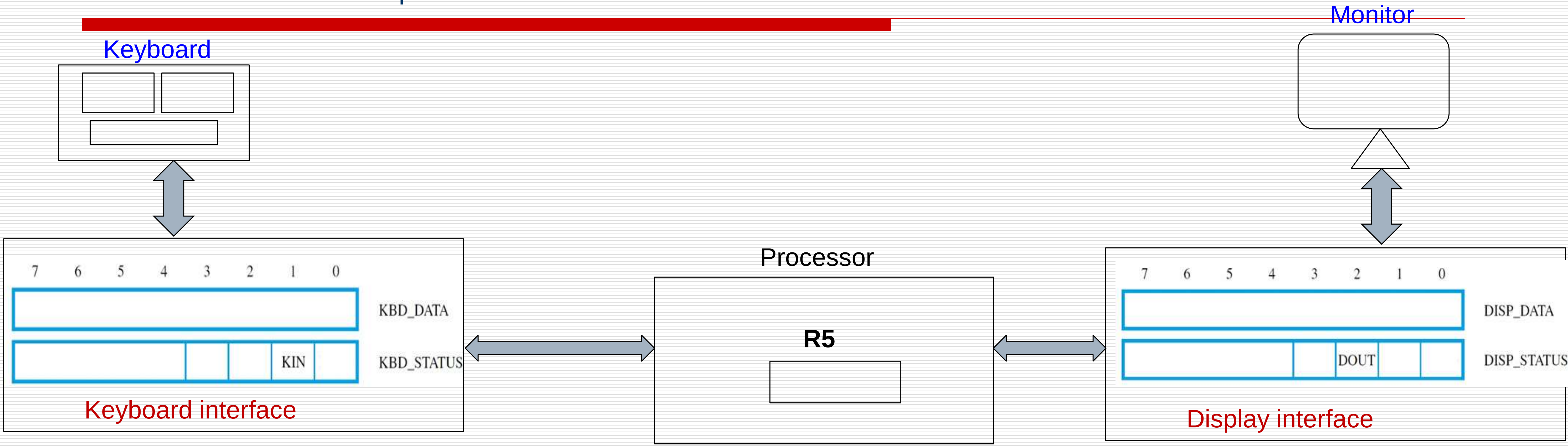
```
READWAIT Read the KIN flag
                  Branch to READWAIT if KIN = 0
Transfer data from KBD_DATA to R5
```

Program Controlled I/O:
Data Transfer from Input device to Processor



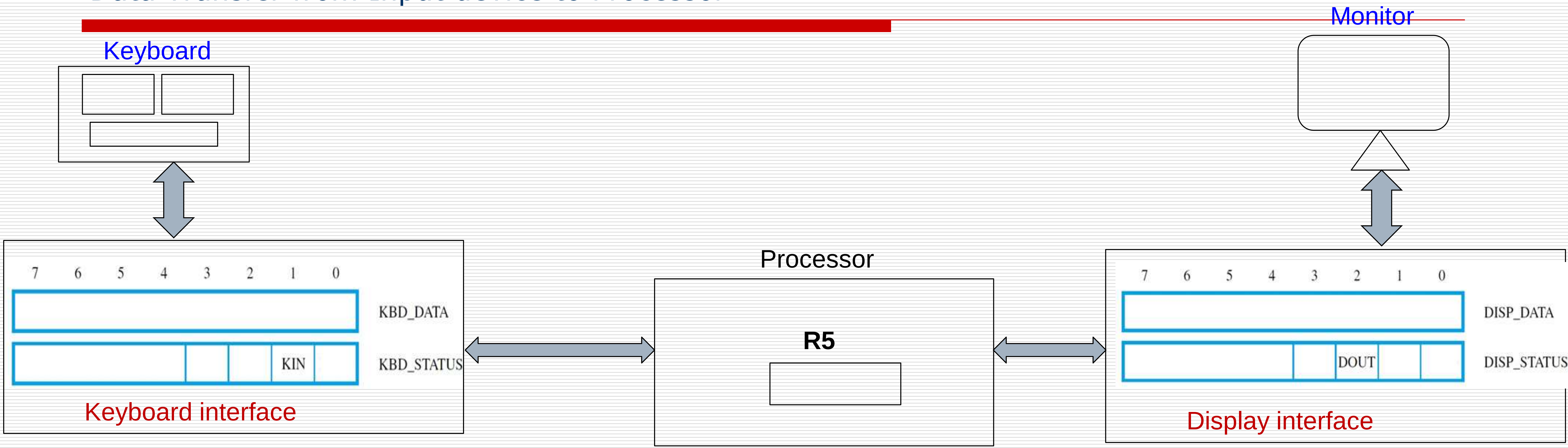
```
READWAIT: LoadByte R4, KBD_STATUS
          And R4, R4, #2
          Branch_if_[R4]=0 READWAIT
          LoadByte R5, KBD_DATA
```

Program Controlled I/O:
Data Transfer from Input device to Processor



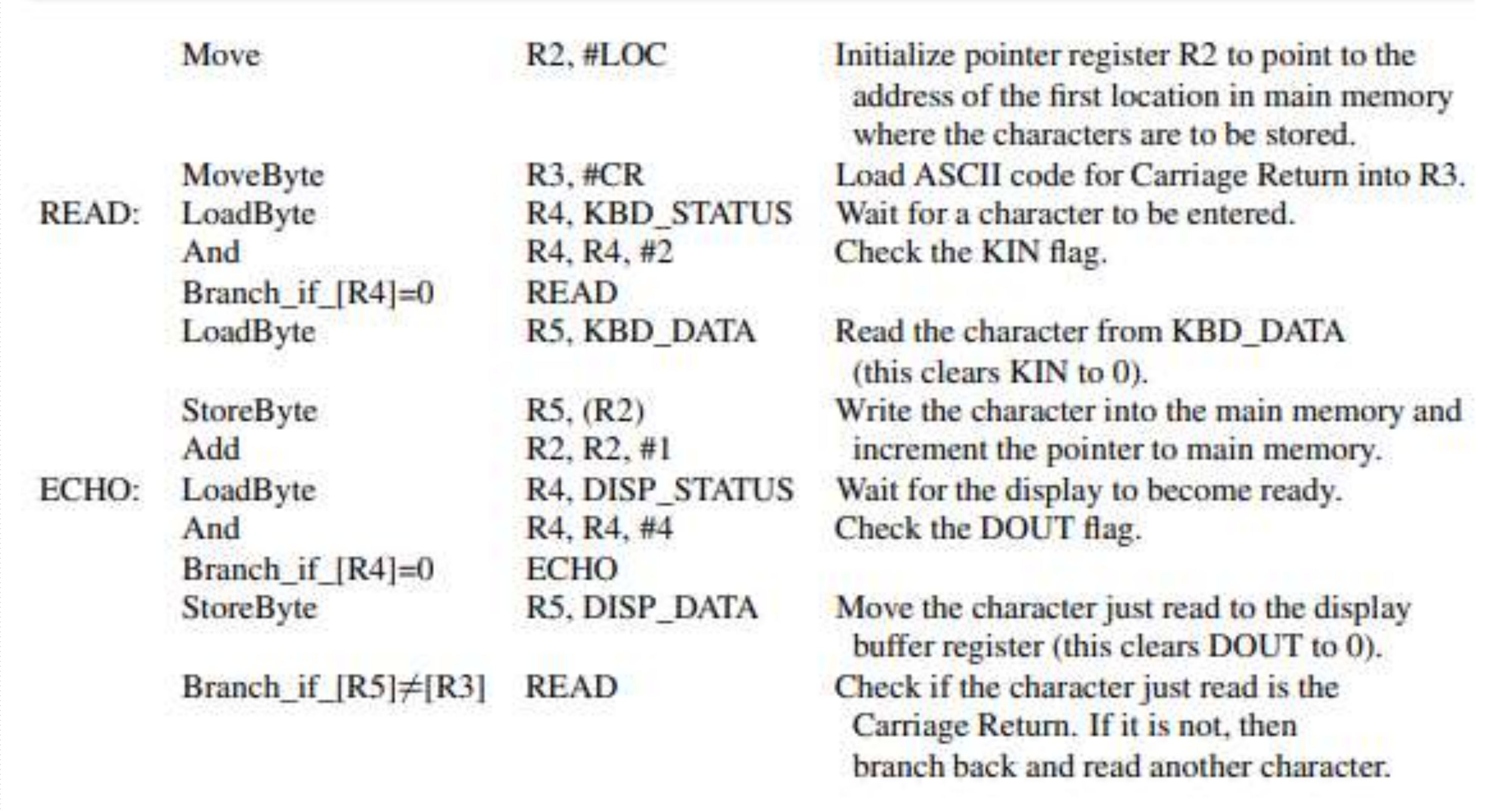
```
WRITEWAIT Read the DOUT flag
          Branch to WRITEWAIT if DOUT = 0
          Transfer data from R5 to DISP_DATA
```

Program Controlled I/O:
Data Transfer from Input device to Processor



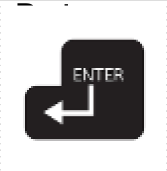
```
WRITEWAIT: LoadByte R4, DISP_STATUS
           And R4, R4, #4
           Branch_if_[R4]=0 WRITEWAIT
           StoreByte R5, DISP_DATA
```

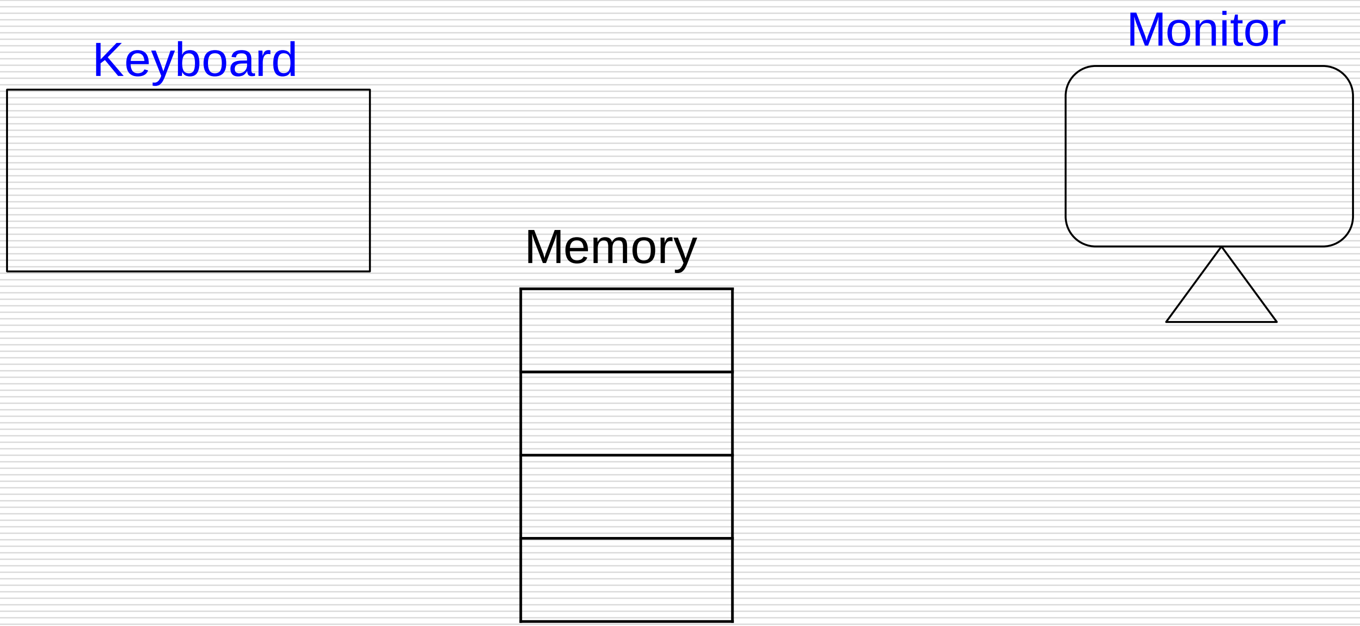
RISC-style program that reads a line of characters and displays it



```
        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.
```

Rough slide to explain: RISC-style program that reads a line of characters and displays it

Character	ASCII Hexacode
A	0x41
B	0x42
C	0x43
D	0x44
E	0x45
F	0x46
….	….
Enter Key or Carriage 	0x0D

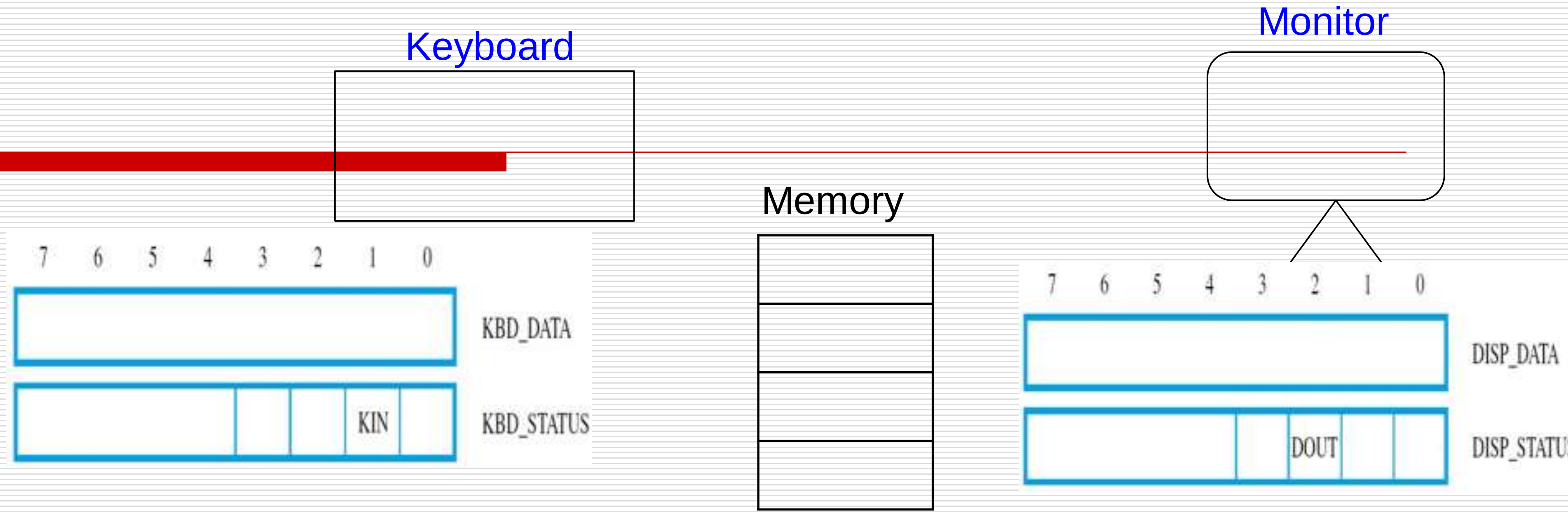


Keyboard

Memory

Monitor

Rough slide to explain:
RISC-style program that reads a line of characters and displays it



```
       Move R2, #LOC

       MoveByte R3, #0x0D
READ:  LoadByte R4, KBD_STATUS
       And R4, R4, #2
       Branch_if_[R4]=0 READ
       LoadByte R5, KBD_DATA

       StoreByte R5, (R2)
       Add R2, R2, #1
ECHO:  LoadByte R4, DISP_STATUS
       And R4, R4, #4
       Branch_if_[R4]=0 ECHO
       StoreByte R5, DISP_DATA
       Branch_if_[R5]≠[R3] READ
```

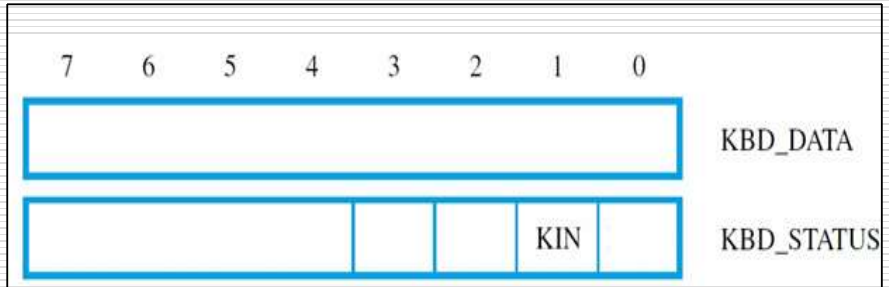
Modes of I/O Data Transfer

Data transfer between the central unit and I/O devices can be handled in generally three types of modes which are given below:

1. Program Controlled I/O
2. Interrupt Initiated I/O
3. Direct Memory Access

Disadvantage of Program Controlled I/O

□ Under Program Controlled I/O, the program enters a wait loop in which it repeatedly tests the device status. During this period, the processor is not performing any useful computation. There are many situations where other tasks can be performed while waiting for an I/O device to become ready.



Keyboard interface

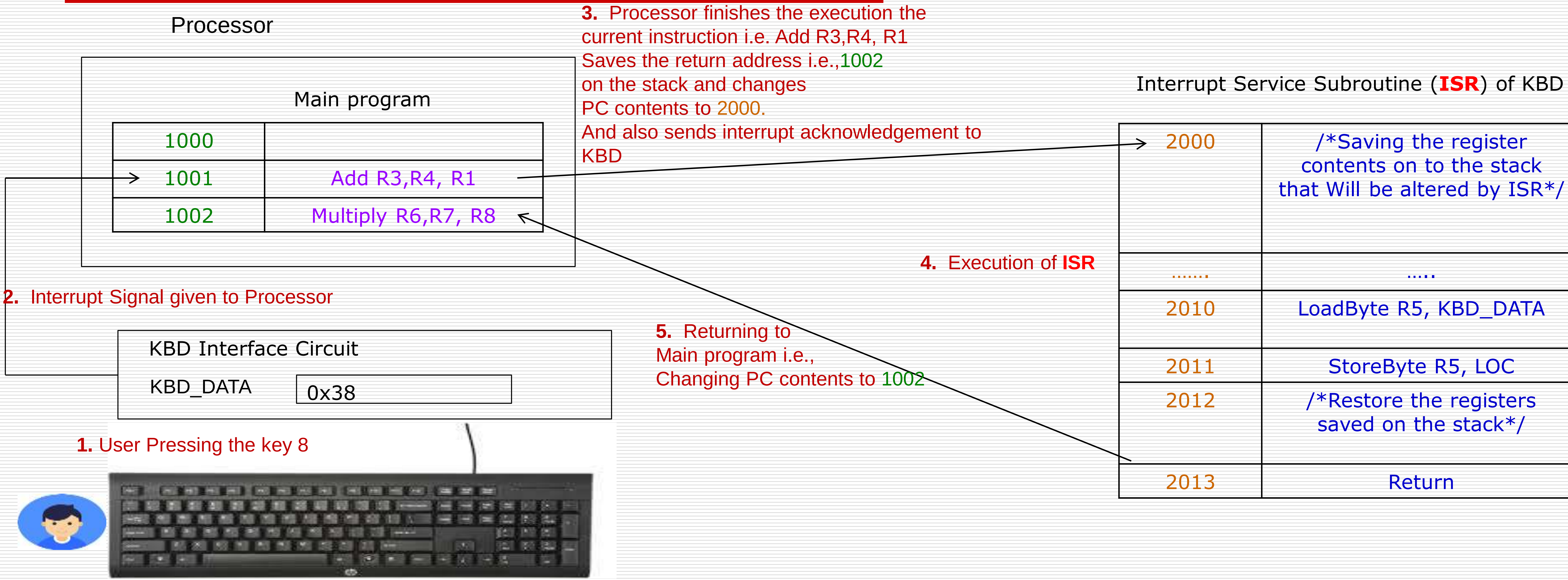
```
READWAIT: LoadByte R4, KBD_STATUS
          And R4, R4, #2
          Branch_if_[R4]=0 READWAIT
          LoadByte R5, KBD_DATA
```

Interrupt based Data transfer between I/O device and Processor

What is Interrupt ?

- Drawback of a Program controlled I/O: **Wait-loop**
- Instead of using wait-loop, let I/O device alert the processor when it is ready.
- Hardware sends an **interrupt-request signal** to the processor at the appropriate time, much like a **phone call**.
- Interrupt is a mechanism used by most processors to handle asynchronous type of events.
- Essentially, the interrupts allow devices to request that the processor stops what it is currently doing and executes software (called Interrupt Service Routine) to process the device’s request, much like a Subroutine call that is initiated by the external device rather than by the program running on the processor.
- Interrupts are also used when the processor needs to perform a long-running operation on some I/O device and wants to be able to do other work while waiting for the operation to complete.

Interrupt mechanism for data transfer between I/O device and processor



Processor

Main program

1000	
1001	Add R3,R4, R1
1002	Multiply R6,R7, R8

3. Processor finishes the execution the current instruction i.e. Add R3,R4, R1 Saves the return address i.e.,1002 on the stack and changes PC contents to 2000. And also sends interrupt acknowledgement to KBD

2. Interrupt Signal given to Processor

KBD Interface Circuit

KBD_DATA	0x38

1. User Pressing the key 8

4. Execution of **ISR**

5. Returning to Main program i.e., Changing PC contents to 1002

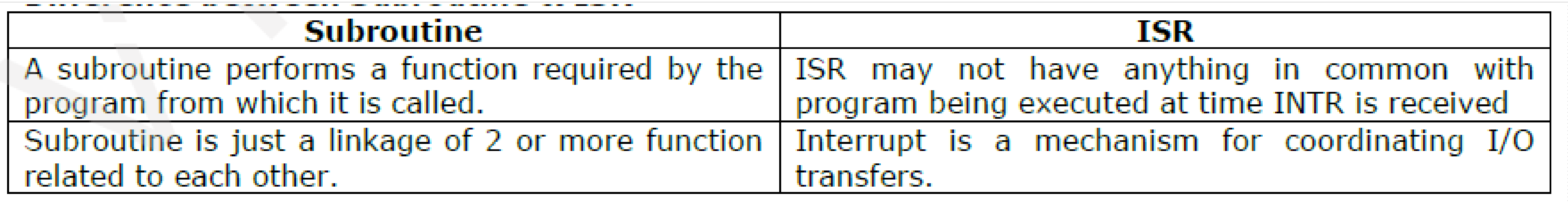
Interrupt Service Subroutine (**ISR**) of KBD

2000	/*Saving the register contents on to the stack that Will be altered by ISR*/
…….	…..
2010	LoadByte R5, KBD_DATA
2011	StoreByte R5, LOC
2012	/*Restore the registers saved on the stack*/
2013	Return

Interrupt Latency

- The information that needs to be saved and restored typically includes the condition code flags and the contents of any registers used by both the interrupted program and the interrupt-service routine.
- Saving registers also increases the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine. The delay is called **interrupt latency**.
- Typically, the processor saves only the contents of the program counter and the processor status register. Any additional information that needs to be saved must be saved by program instruction at the beginning of the interrupt-service routine and restored at the end of the routine

Difference between Subroutine and Interrupt Service Routine (ISR)

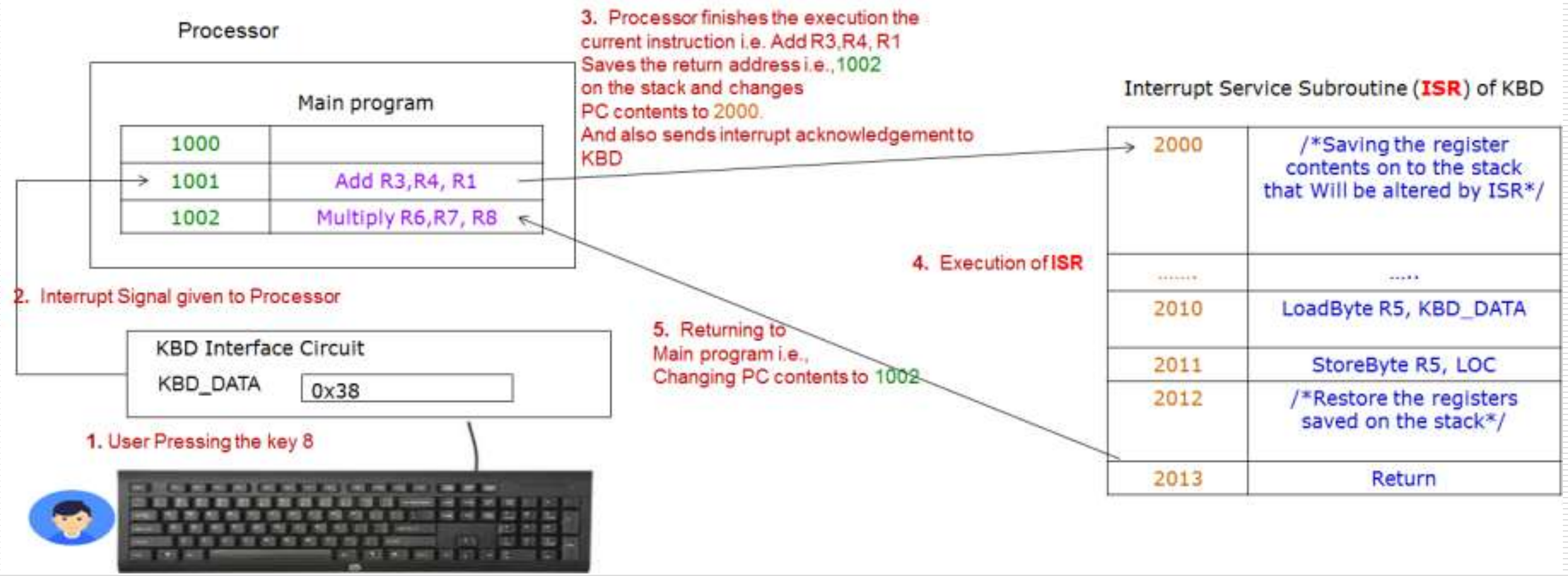


Subroutine	ISR
A subroutine performs a function required by the program from which it is called.	ISR may not have anything in common with program being executed at time INTR is received
Subroutine is just a linkage of 2 or more function related to each other.	Interrupt is a mechanism for coordinating I/O transfers.

Question

Consider the case of a single interrupt request from one device. The device keeps interrupt request signal activated until it is informed that the processor has accepted its request. This activated signal, if not deactivated, may lead to successive interruptions causing the system to enter infinite loop.

Think, what measures the processor can take to avoid “successive interruptions causing the system to enter infinite loop”



Processor

Main program

1000	
1001	Add R3,R4, R1
1002	Multiply R6,R7, R8

1. User Pressing the key 8

2. Interrupt Signal given to Processor

3. Processor finishes the execution the current instruction i.e. Add R3,R4, R1 Saves the return address i.e.,1002 on the stack and changes PC contents to 2000. And also sends interrupt acknowledgement to KBD

4. Execution of ISR

5. Returning to Main program i.e., Changing PC contents to 1002

KBD Interface Circuit

KBD_DATA	0x38

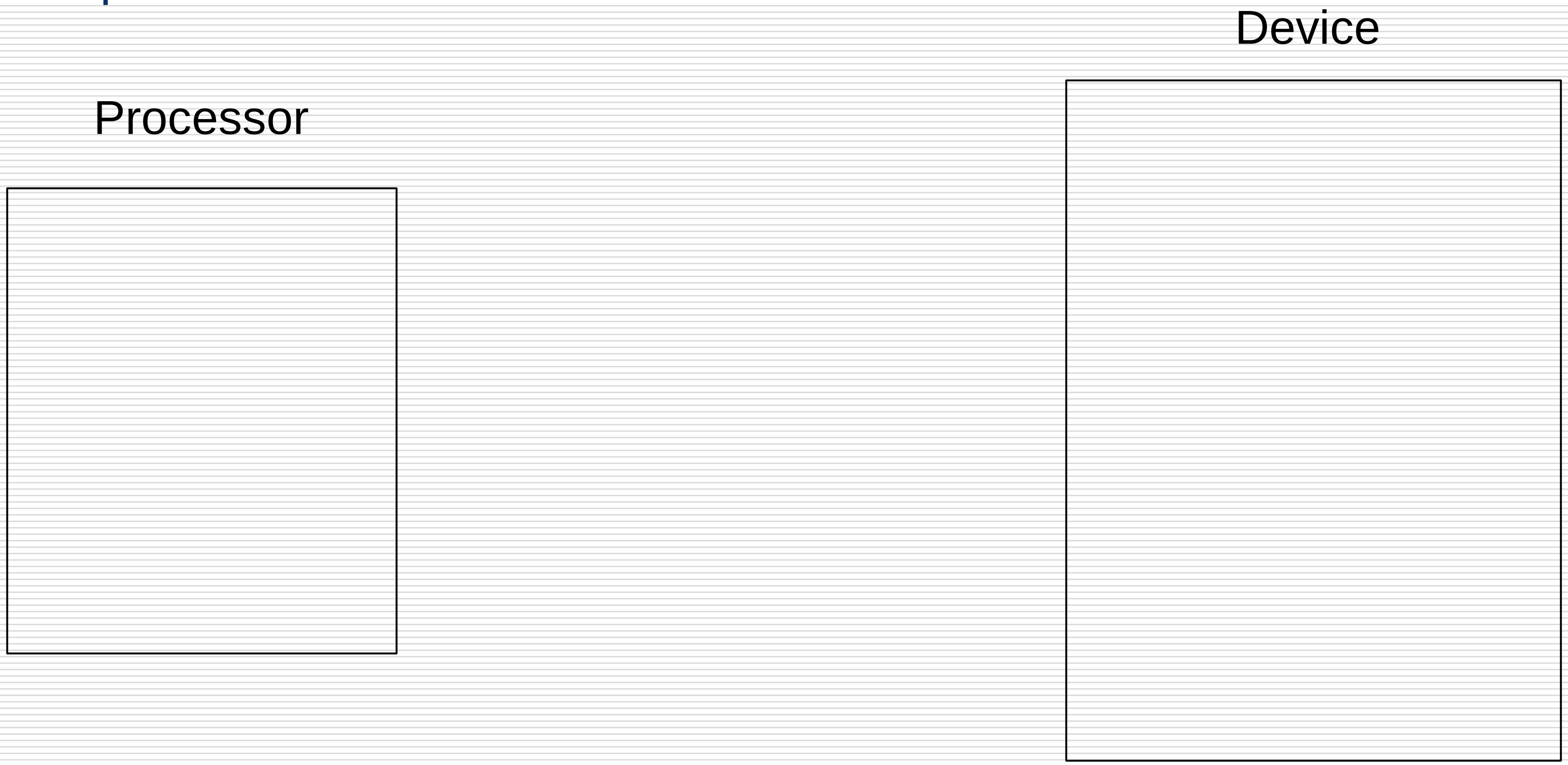
Interrupt Service Subroutine (**ISR**) of KBD

2000	/*Saving the register contents on to the stack that Will be altered by ISR*/
....	.....
2010	LoadByte R5, KBD_DATA
2011	StoreByte R5, LOC
2012	/*Restore the registers saved on the stack*/
2013	Return

Rough Slide to Explain the Question

Consider the case of a single interrupt request from one device. The device keeps interrupt request signal activated until it is informed that the processor has accepted its request. This activated signal, if not deactivated, may lead to successive interruptions causing the system to enter infinite loop.

Think, what measures the processor can take to avoid “successive interruptions causing the system to enter infinite loop”



Device

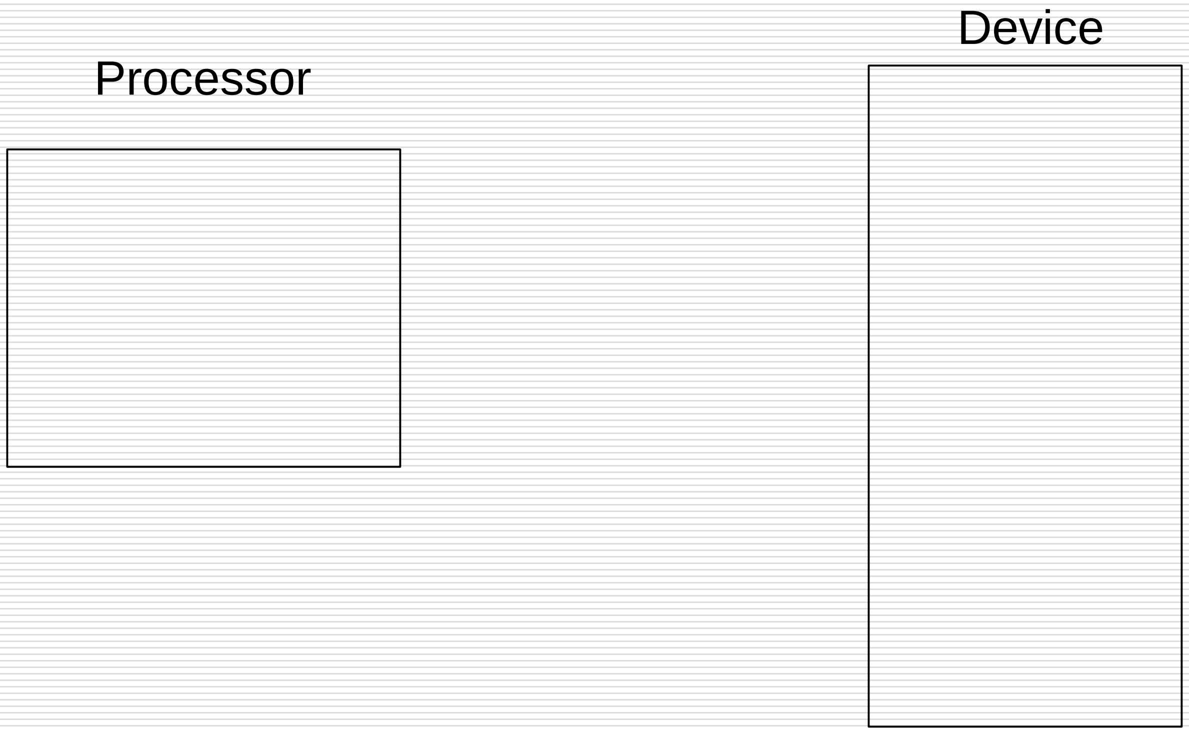
Processor

Rough slide to explain Answer:
Enabling and Disabling Interrupts

One mechanisms to avoid “successive interruptions causing the system to enter infinite loop” problem is as follows:

Processor automatically disabling the interrupts before staring the execution of ISR

The second option, which is suitable for a simple processor with only one interrupt-request line, is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine. After saving the contents of the PC and the processor status register (PS) on the stack, the processor performs the equivalent of executing an Interrupt-disable instruction. It is often the case that one bit in the PS register, called Interrupt-enable, indicates whether interrupts are enabled. An interrupt request is received while this bit is equal to 1 will be accepted. After saving the contents of the PS on the stack, with the Interrupt-enabled bit equal to 1, the processor clears the interrupt-enable bit in its PS register, thus disabling further interrupts. When a return from interrupt instruction is executed, the contents of the PS are restored from the stack, setting the interrupt enable bit to 1. Hence interrupts are again enabled.



Device

Processor

Enabling and Disabling Interrupts

The processor has a status register (PS), which contains information about its current state of operation. Let one bit, IE, of this register 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.

Let us now consider the specific case of a single interrupt request from one device. When a device activates the interrupt-request signal, it keeps this signal activated until it learns that the processor has accepted its request. This means that the interrupt-request signal will be active during execution of the interrupt-service routine, perhaps until an instruction is reached that accesses the device in question. It is essential to ensure that this active request signal does not lead to successive interruptions, causing the system to enter an infinite loop from which it cannot recover.

A good choice is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine. The processor saves the contents of the program counter and the processor status register. After saving the contents of the PS register, with the IE bit equal to 1, the processor clears the IE bit in the PS register, thus disabling further interrupts. Then, it begins execution of the interrupt-service routine. When a Return-from-interrupt instruction is executed, the saved contents of the PS register are restored, setting the IE bit back to 1. Hence, interrupts are again enabled.

Sequence of events involved in handling an interrupt request from a single device.

Assuming that interrupts are enabled, 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.

Question

The signal sent to the device from the processor to the device after receiving an interrupt is

a) Interrupt-acknowledge

b) Return signal

c) Service signal

d) Permission signal

Answer

The signal sent to the device from the processor to the device after receiving an interrupt is

a) **Interrupt-acknowledge**

b) Return signal

c) Service signal

d) Permission signal

Answer: a

Explanation: The Processor upon receiving the interrupt should let the device know that its request is received.

Question

The time between the recieving of an interrupt and its service is _______

a) Interrupt delay

b) Interrupt latency

c) Cycle time

d) Switching time

Answer

The time between the recieving of an interrupt and its service is ______

a) Interrupt delay

b) Interrupt latency

c) Cycle time

d) Switching time

Answer: b

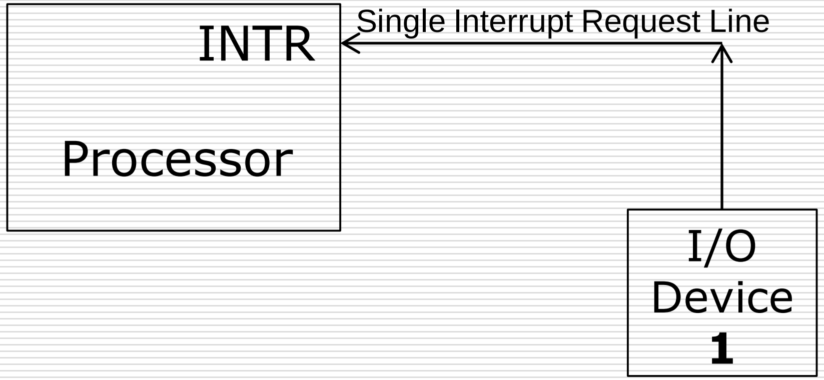
Explanation: The delay in servicing of an interrupt happens due to the time taken for transfer to take place from Main Program to ISR

Modes of I/O Data Transfer

Data transfer between the central unit and I/O devices can be handled in generally three types of modes which are given below:

1. Program Controlled I/O
2. Interrupt Initiated I/O
3. Direct Memory Access

Rough Slide to explain Interrupt request from single device

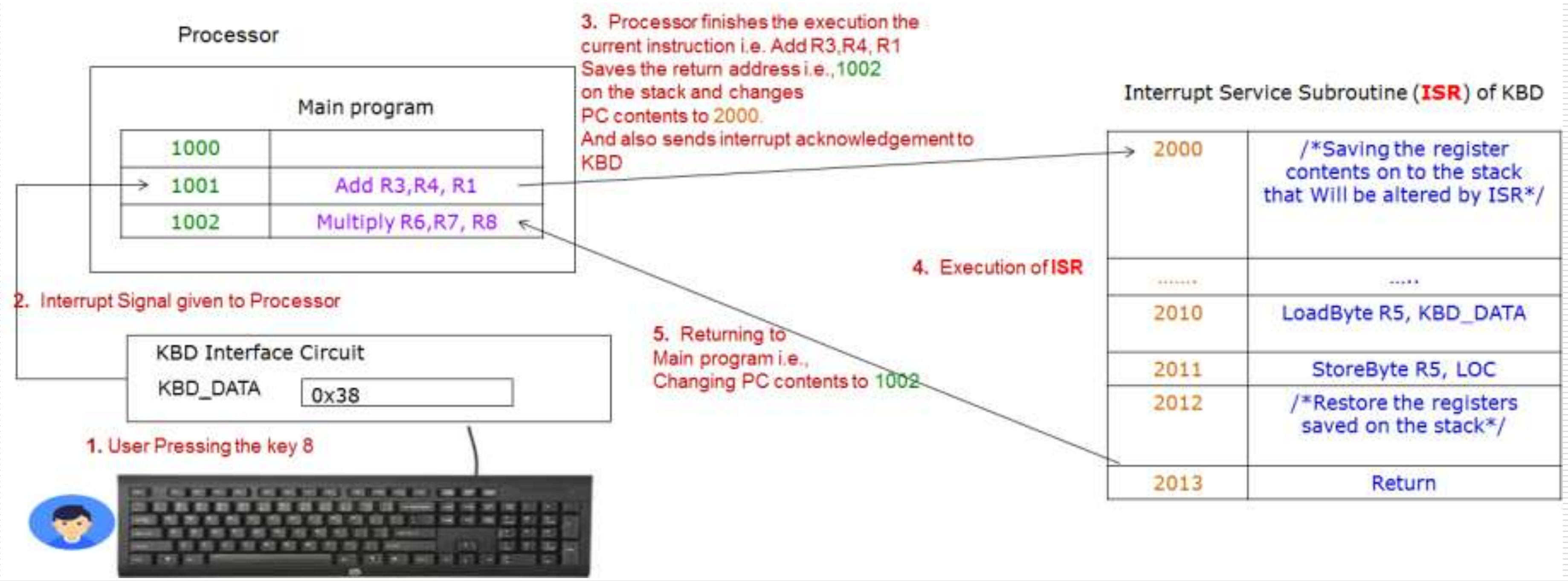


INTR

Processor

Single Interrupt Request Line

I/O Device **1**



Processor

Main program

1000	
1001	Add R3,R4, R1
1002	Multiply R6,R7, R8

3. Processor finishes the execution the current instruction i.e. Add R3,R4, R1 Saves the return address i.e.,1002 on the stack and changes PC contents to 2000. And also sends interrupt acknowledgement to KBD

Interrupt Service Subroutine (**ISR**) of KBD

2000	/*Saving the register contents on to the stack that Will be altered by ISR*/
....	.....
2010	LoadByte R5, KBD_DATA
2011	StoreByte R5, LOC
2012	/*Restore the registers saved on the stack*/
2013	Return

4. Execution of **ISR**

2. Interrupt Signal given to Processor

5. Returning to Main program i.e., Changing PC contents to 1002

KBD Interface Circuit

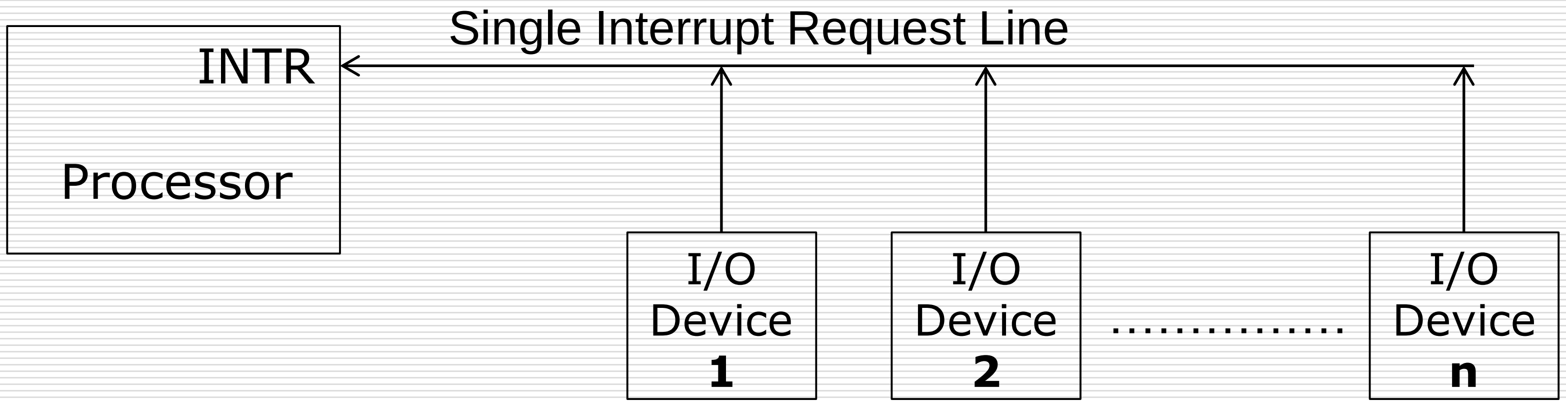
KBD_DATA 0x38

1. User Pressing the key 8

Handling Multiple Devices

Handling multiple devices gives rise to a number of questions:

- How can the processor recognize the device requesting an interrupt ?
- Given that different devices are likely to require different interrupt-service routines, how can the processor obtain the starting address of the appropriate routine in each case ?
- Should a device be allowed to interrupt the processor while another interrupt is being serviced ?
- How should two or more simultaneous interrupt request be handled ?



Under Interrupt Driven I/O data transfer: Handling Multiple Devices

I. Methods to **identify** the device over a single interrupt request line

   a. Polling

   b. Vectored Interrupt

II. **Interrupt Nesting**: A mechanism has to be developed such that even though a processor is executing an ISR, another interrupt request should be accepted and serviced

   a. Priority Structure

III. **Simultaneous Requests**: Multiple Requests are received over a single interrupt request line at the same time

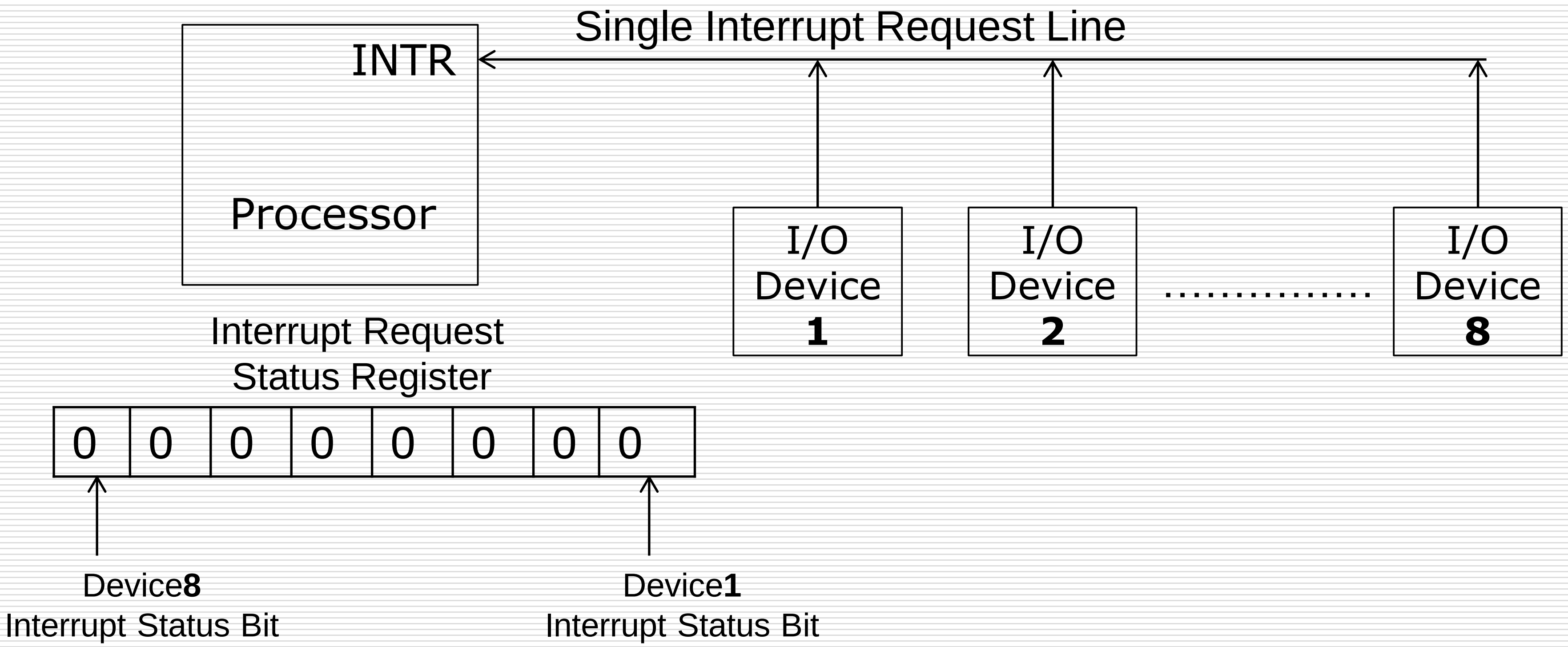
I. Methods to identify the device over a single interrupt request line:

a. **Polling**

- Consider a simple arrangement where all devices send their interrupt-requests over a single control line in the bus.
- When the processor receives an interrupt request over this control line, how does it know which device is requesting an interrupt?
- This information is available in the status register of the device requesting an interrupt:
  - The status register of each device has an IRQ bit which it sets to 1 when it requests an interrupt.
- Interrupt service routine can poll the I/O devices connected to the bus. The first device with IRQ equal to 1 is the one that is serviced.
- Polling mechanism is easy, but time consuming to query the status bits of all the I/O devices connected to the bus.

I. Methods to identify the device over a single interrupt request line: a.**Polling**

Illustration: Considering eight devices


Single Interrupt Request Line
INTR
Processor
I/O Device **1**
I/O Device **2**
....
I/O Device **8**
Interrupt Request Status Register
0 0 0 0 0 0 0 0
Device**8** Interrupt Status Bit
Device**1** Interrupt Status Bit

Question

- □ A single Interrupt line can be used to service *n* different devices?
  a) True
  b) False

Question

- A single Interrupt line can be used to service n different devices?
  a) **True**
  b) False

Question

The process that periodically checks the status of an I/O devices, is known as

a. Cold swapping

b. I/O instructions

c. Polling

d. Dealing

Question

The process that periodically checks the status of an I/O devices, is known as

a. Cold swapping

b. I/O instructions

c. Polling

d. Dealing

I. Methods to identify the device over a single interrupt request line:

b. **Vectored Interrupt**

- □ **Vectored interrupts** reduce service latency; no instructions executed to poll many devices.
- □ Let requesting device identify itself directly with a special signal or a **unique binary code** (like different ringing tones for different callers). Processor uses info to find address of correct routine in an interrupt-vector table
- □ Table lookup is performed. Vector table is located at fixed address, but routines can be located anywhere in memory

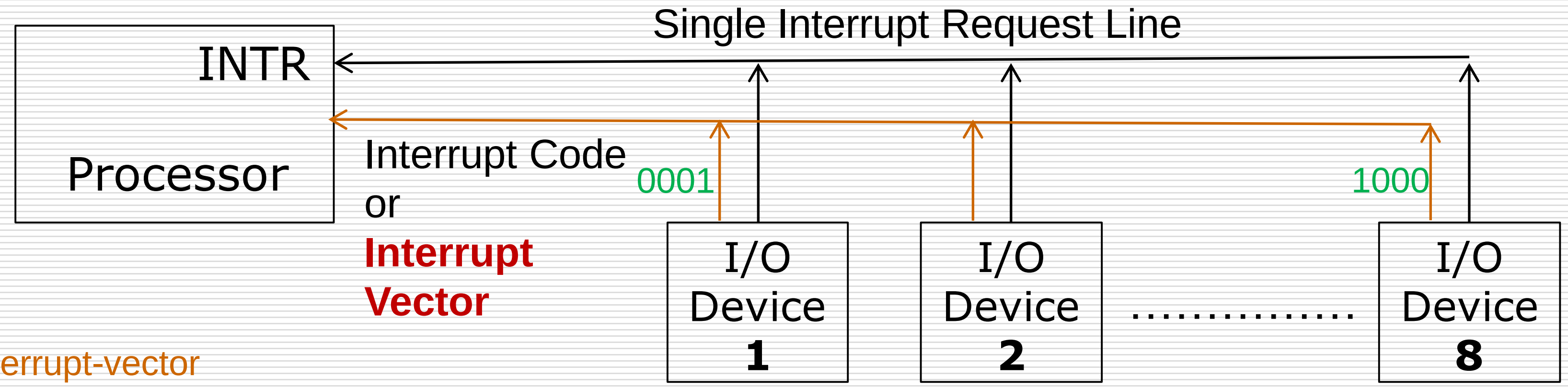
I. Methods to identify the device over a single interrupt request line:

b. **Vectored Interrupt**

- □ A device requesting an interrupt can identify itself if it has its own interrupt-request signal, or if it can send a special code to the processor through the interconnection network.
- □ The processor’s circuits determine the memory address of the required interrupt-service routine.
- □ A commonly used scheme is to allocate permanently an area in the memory to hold the addresses of interrupt-service routines. These addresses are usually referred to as **interrupt vectors**, and they are said to constitute the **interrupt-vector table**. For example, 128 bytes may be allocated to hold a table of 32 interrupt vectors.
- □ The interrupt-service routines may be located anywhere in the memory.
- □ 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.

I. Methods to identify the device over a single interrupt request line:
b. **Vectored Interrupt**

Illustration



Interrupt-vector Lookup Table

Addr	Memory
0001	1000
0010	2000

1000	8000

Address of **ISR** for **Device 1**

ISR for Device1

Addr	
1000	----
1001	----

Question

The starting address sent by the device in vectored interrupt is called as

a. Location ID

b. Interrupt vector

c. Service location

d. Service ID

Question

The starting address sent by the device in vectored interrupt is called as

a. Location ID

b. Interrupt vector

c. Service location

d. Service ID

Under Interrupt Driven I/O data transfer: Handling Multiple Devices

I. Methods to **identify** the device over a single interrupt request line

a. Polling

b. Vectored Interrupt

II. **Interrupt Nesting**: A mechanism has to be developed such that even though a processor is executing an ISR, another interrupt request should be accepted and serviced

a. Priority Structure

III. **Simultaneous Requests**: Multiple Requests are received over a single interrupt request line at the same time

Under Interrupt Driven I/O data transfer: Handling Multiple Devices
II. Interrupt Nesting: Priority structure

- Interrupt Nesting: A mechanism has to be developed such that even though a processor is executing an ISR, another interrupt request should be accepted and serviced
- I/O devices are organized in a priority structure: An interrupt request from a high-priority device is accepted while the processor is executing the interrupt service routine of a low priority device.
- A multiple-level priority organization means that during execution of an interrupt service routine, interrupt requests will be accepted from some devices but not from others, depending upon the device's priority.
- To implement this scheme, we can 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.
- At the time that execution of an interrupt-service routine for some device is started, the priority of the processor is raised to that of the device either automatically or with special instructions. This action disables interrupts from devices that have the same or lower level of priority.
- However, 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

Under Interrupt Driven I/O data transfer: Handling Multiple Devices
II. Interrupt Nesting: Priority structure

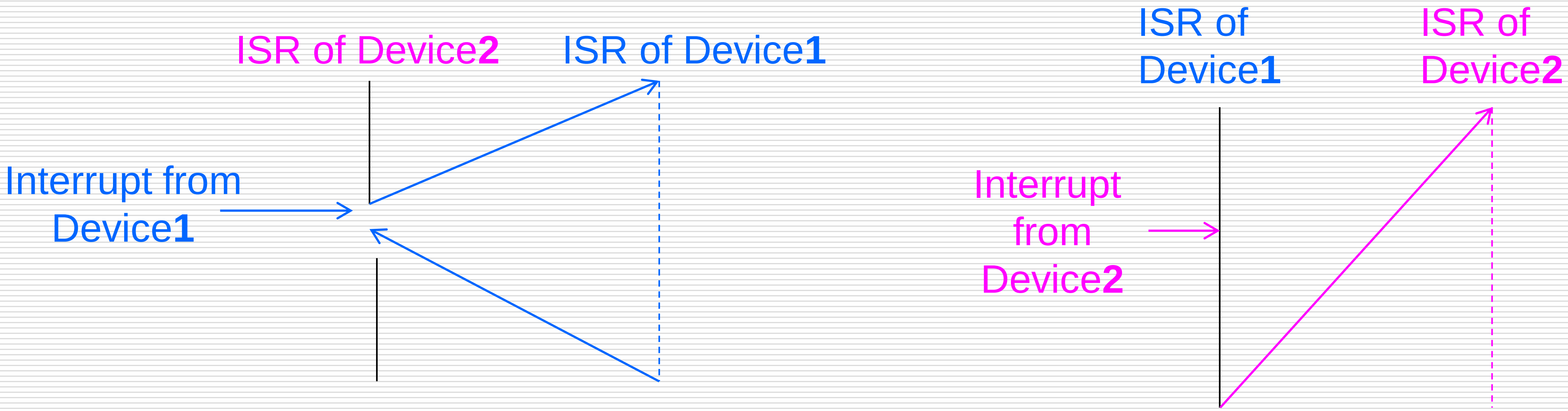
Rough Slide for explanation

Illustration:

Device**1**
Interrupt Priority

Greater Than
>

Device**2**
Interrupt Priority



Under Interrupt Driven I/O data transfer: Handling Multiple Devices

I. Methods to **identify** the device over a single interrupt request line

a. Polling

b. Vectored Interrupt

II. **Interrupt Nesting**: A mechanism has to be developed such that even though a processor is executing an ISR, another interrupt request should be accepted and serviced

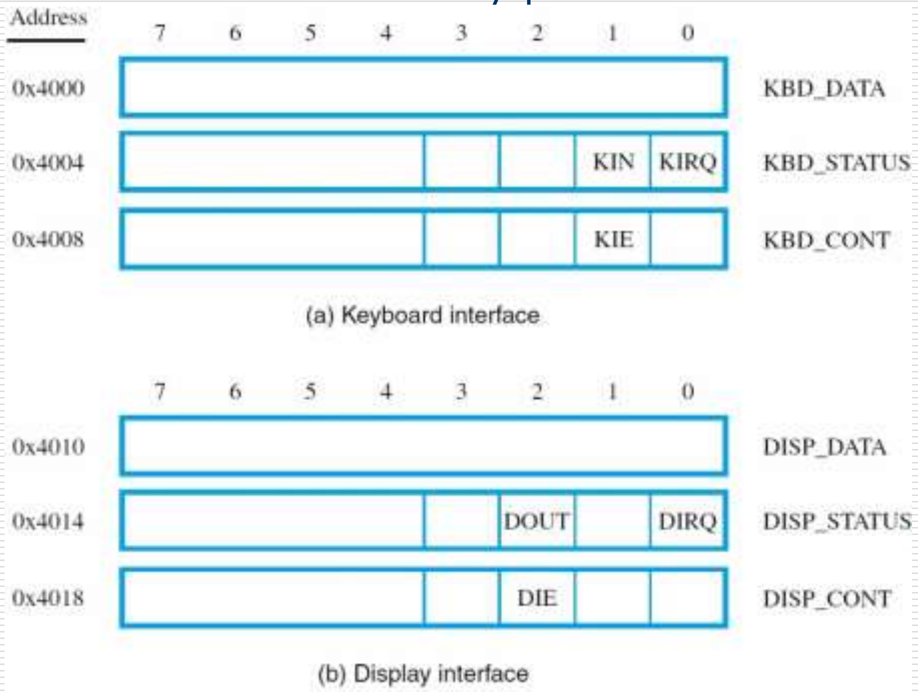
a. Priority Structure

III. **Simultaneous Requests**: Multiple Requests are received over a single interrupt request line at the same time

- □ We also need to consider the problem of simultaneous arrivals of interrupt requests from two or more devices.
- □ 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. This is done in hardware, by using arbitration circuits.

Under Interrupt Driven I/O data transfer: Controlling I/O Device Behavior

- It is important to ensure that 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.



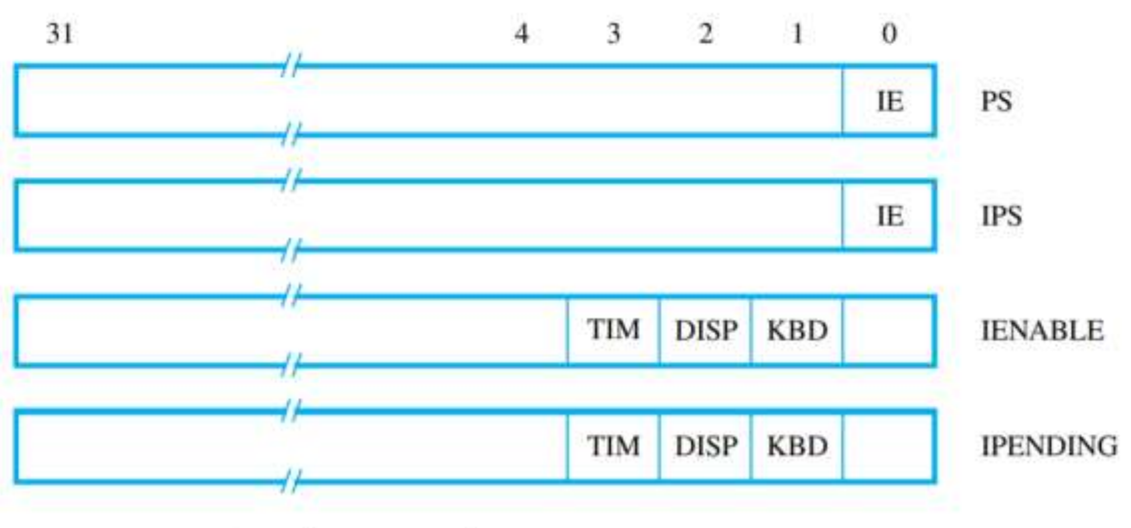
(a) Keyboard interface

(b) Display interface

- 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. Thus, when both KIE and KIN bits are equal to 1, an interrupt request is raised and the KIRQ bit is set to 1. Similarly, the 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.

Under Interrupt Driven I/O data transfer:
Processor Control Registers

- □ The status register, PS, includes the interrupt-enable bit, IE, in addition to other status information. Recall that the processor will accept interrupts only when this bit is set to 1. 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. Since there is only one register available for storing the previous status information, it becomes necessary to save the contents of IPS on the stack if nested interrupts are allowed.
- □ 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, and a timer circuit that we will use in a later example. 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. This is convenient when multiple devices may raise requests at the same time.



Control registers in the processor.

In a RISC-style processor, the special instructions may be of the type

MoveControl R2, PS

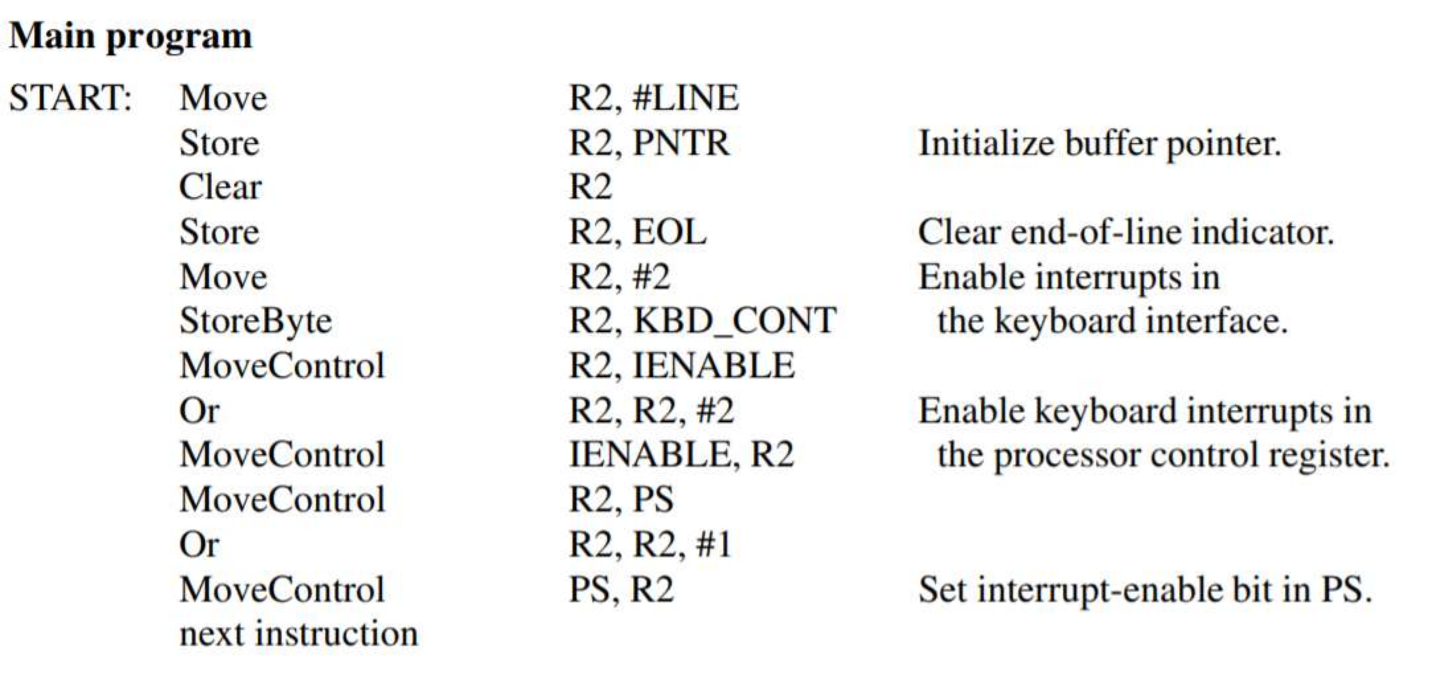
which loads the contents of the program status register into register R2, and

MoveControl IENABLE, R3

which places the contents of R3 into the IENABLE register.

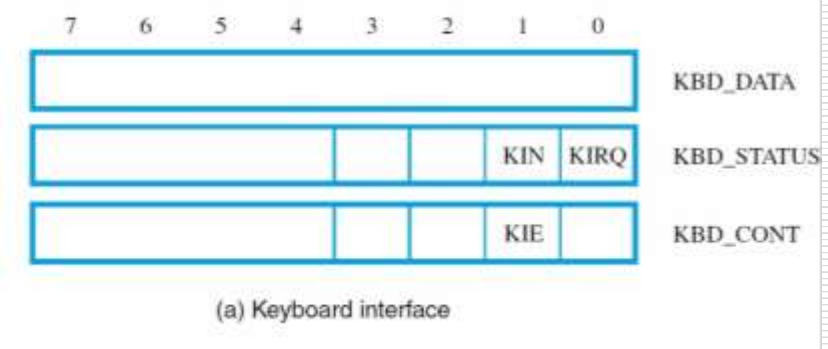
Under Interrupt Driven I/O data transfer:
Examples of Interrupt Programs

- Let us consider again the task of reading a line of characters typed on a keyboard, storing the characters in the main memory, and displaying them on a display device.
- We will use interrupts with the keyboard, but polling with the display.

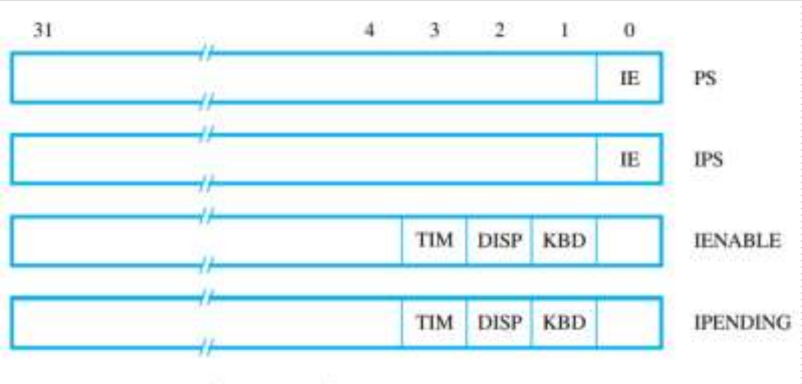


Main program

```
START:  Move            R2, #LINE
        Store           R2, PNTR        Initialize buffer pointer.
        Clear           R2
        Store           R2, EOL         Clear end-of-line indicator.
        Move            R2, #2          Enable interrupts in
        StoreByte       R2, KBD_CONT      the keyboard interface.
        MoveControl     R2, IENABLE
        Or              R2, R2, #2      Enable keyboard interrupts in
        MoveControl     IENABLE, R2       the processor control register.
        MoveControl     R2, PS
        Or              R2, R2, #1
        MoveControl     PS, R2          Set interrupt-enable bit in PS.
        next instruction
```

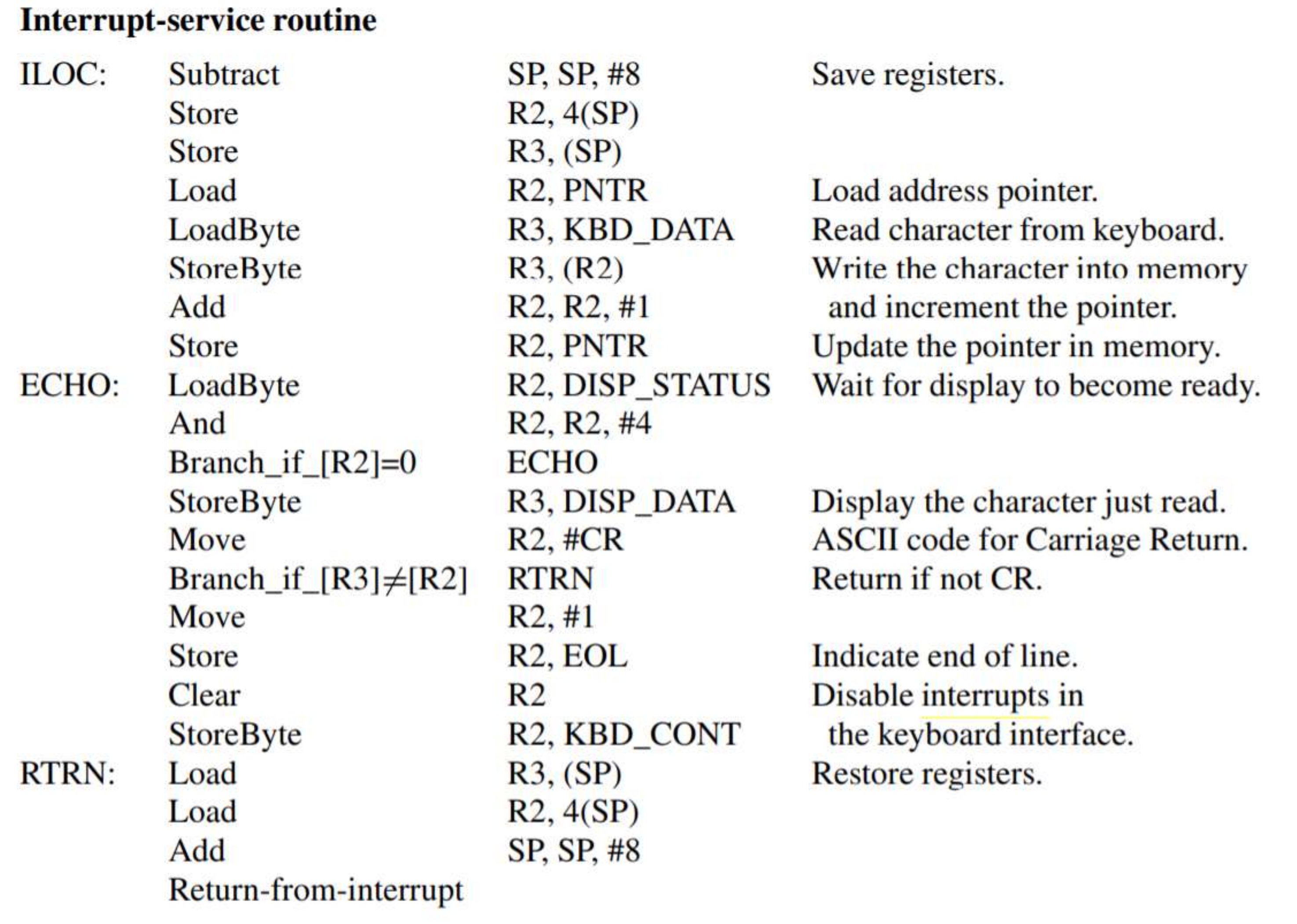


(a) Keyboard interface



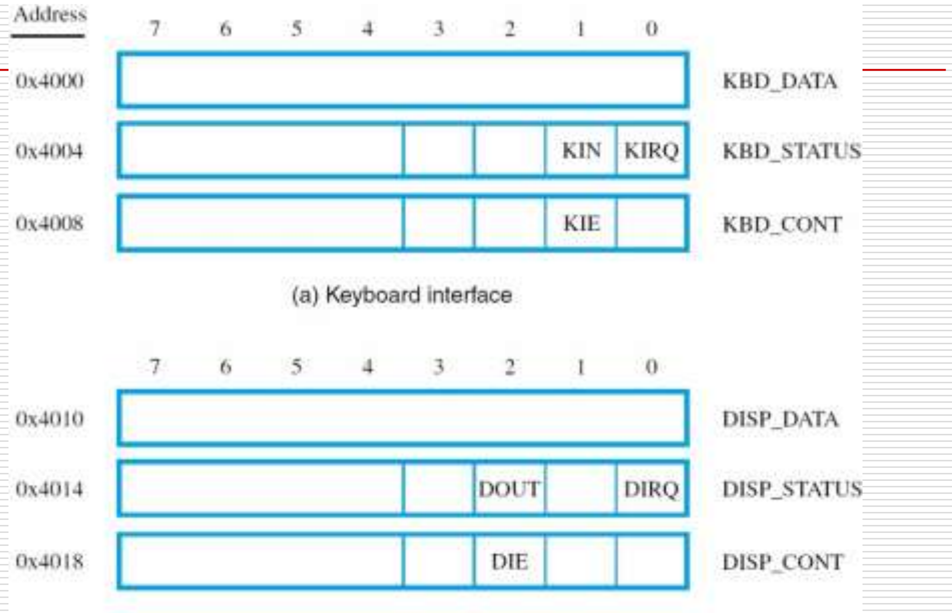
Control registers in the processor.

Under Interrupt Driven I/O data transfer: Examples of Interrupt Programs



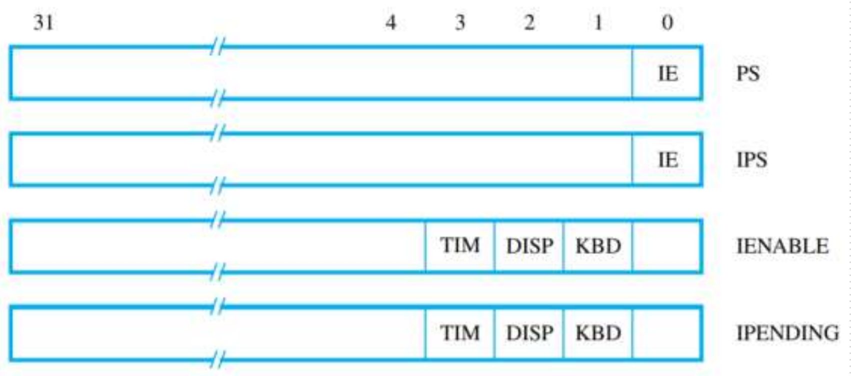
Interrupt-service routine

```
ILOC:   Subtract              SP, SP, #8          Save registers.
        Store                 R2, 4(SP)
        Store                 R3, (SP)
        Load                  R2, PNTR            Load address pointer.
        LoadByte              R3, KBD_DATA        Read character from keyboard.
        StoreByte             R3, (R2)            Write the character into memory
        Add                   R2, R2, #1            and increment the pointer.
        Store                 R2, PNTR            Update the pointer in memory.
ECHO:   LoadByte              R2, DISP_STATUS     Wait for display to become ready.
        And                   R2, R2, #4
        Branch_if_[R2]=0      ECHO
        StoreByte             R3, DISP_DATA       Display the character just read.
        Move                  R2, #CR             ASCII code for Carriage Return.
        Branch_if_[R3]≠[R2]   RTRN                Return if not CR.
        Move                  R2, #1
        Store                 R2, EOL             Indicate end of line.
        Clear                 R2                  Disable interrupts in
        StoreByte             R2, KBD_CONT          the keyboard interface.
RTRN:   Load                  R3, (SP)            Restore registers.
        Load                  R2, 4(SP)
        Add                   SP, SP, #8
        Return-from-interrupt
```



(a) Keyboard interface

(b) Display interface



Control registers in the processor.

Exceptions

- An exception is any interruption of execution.
- This includes interrupts for I/O transfers.
- But there are also other types of exceptions
  - *Recovery from errors*: Detect division by zero, or instruction with an invalid OP code
  - *Debugging*: Use of 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.

Concluding Remarks

- Two basic I/O-handling approaches: P*rogram-controlled* and *Interrupt-based*
  - 1st approach has direct control of I/O transfers
  - Drawback: wait loop to poll flag in status reg.
  - 2nd approach suspends program when needed to service I/O interrupt with separate routine
  - Until then, processor performs useful tasks
- Exceptions cover all interrupts including I/O

Unit-2

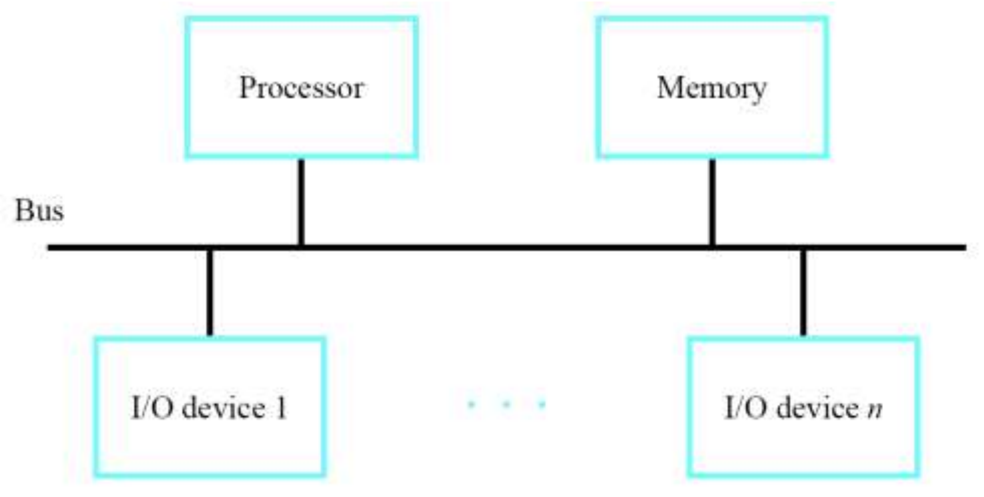
Bus Structure, Bus Operation, Arbitration

Bus

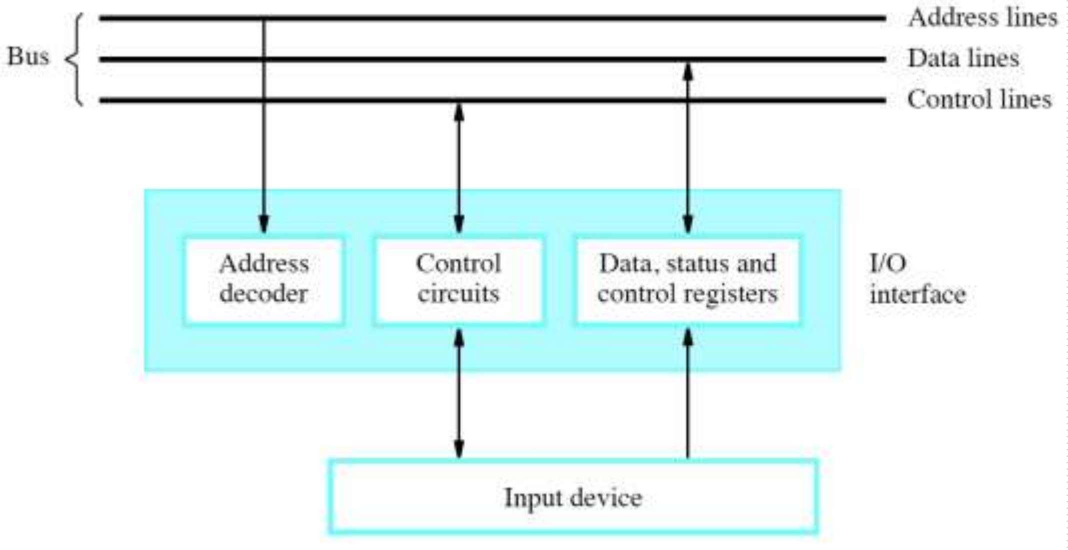
W.r.t to Computer System, a bus is a communication pathway connecting two or more devices

Bus Structure

- The bus as shown in Figure, 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, as shown in Figure for an input device. 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



A single-bus structure



I/O interface for an input device.

Bus Operation

- A bus requires a set of rules, often called a bus protocol, that govern how the bus is used by various devices.
- The bus protocol determines when a device may place information on the bus, 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, usually labelled R/W’, specifies whether a Read or a Write operation is to be performed. As the label suggests, it specifies Read when set to 1 and Write when set to 0
- 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.

Two different schemes for timing of data transfers over a bus

1. **Synchronous** scheme
2. **Asynchronous** scheme

1. Synchronous data transfer: sender and receiver use the same clock signal

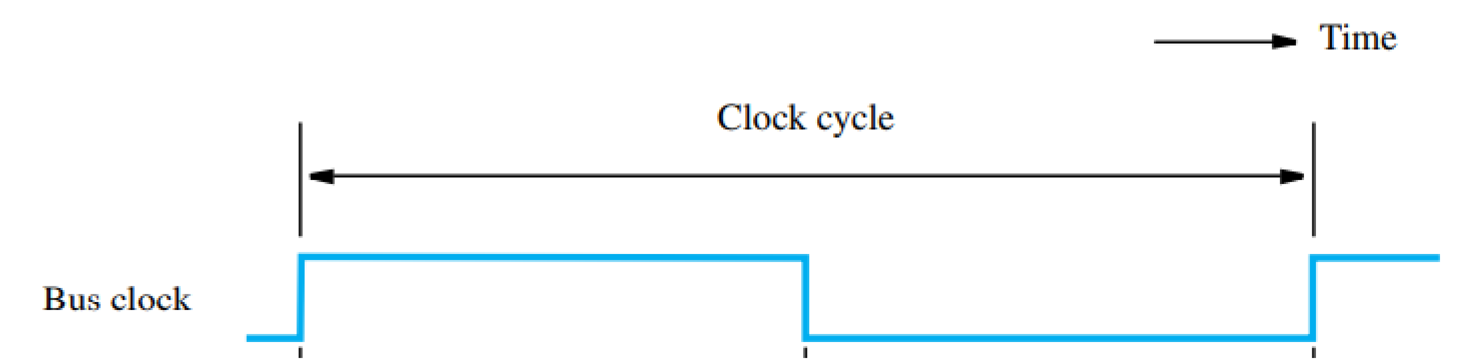
- Needs clock signal between the sender and the receiver
- Supports high data transfer rate

2. Asynchronous data transfer: sender provides a synchronization signal to the receiver before starting the transfer of each message

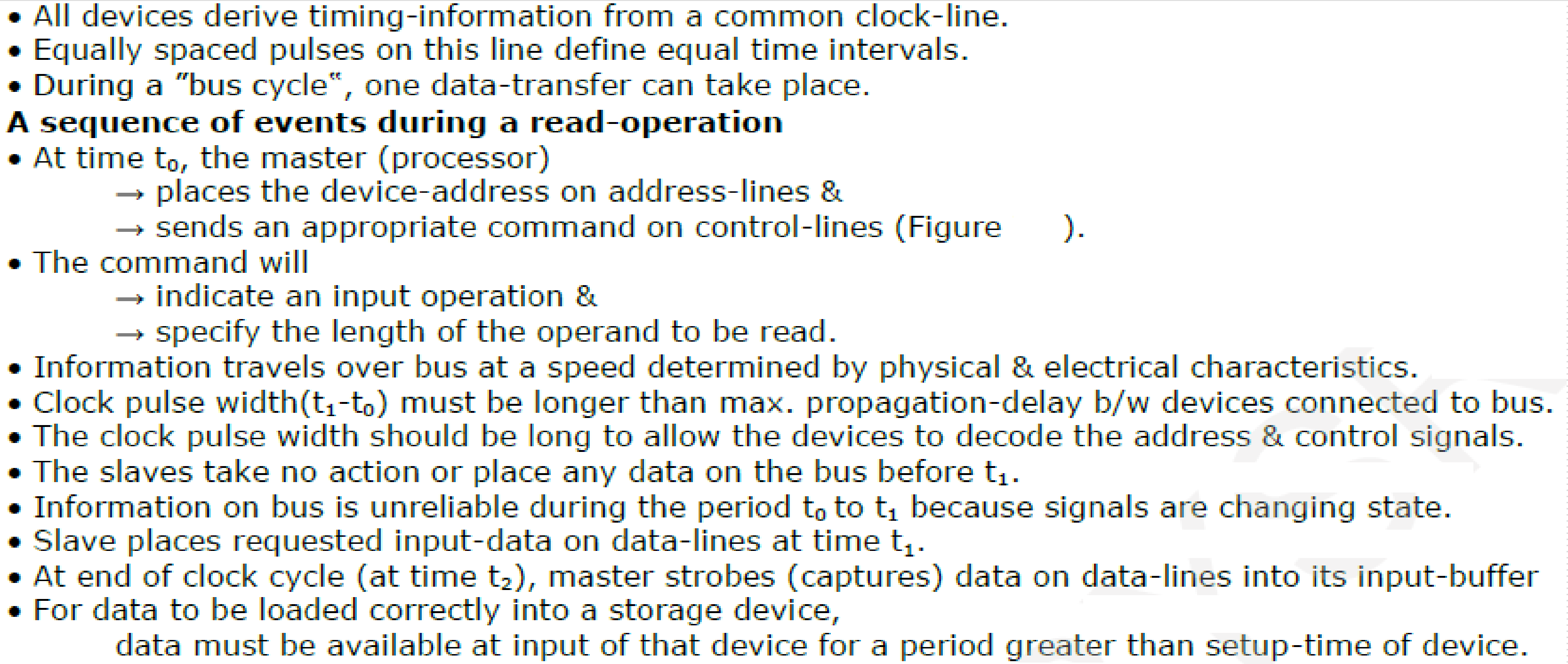
- Does not need clock signal between the sender and the receiver
- Slower data transfer rate

Synchronous bus

- On a synchronous bus, all devices derive timing information from a control line called the bus clock, as shown in the Figure below.
- The signal on this line has two phases: a high level followed by a low level. The two phases constitute a **clock cycle**.
- The first half of the cycle between the low-to-high and high-to-low transitions is often referred to as a **clock pulse**.



Synchronous Data Transfer or Synchronous Bus



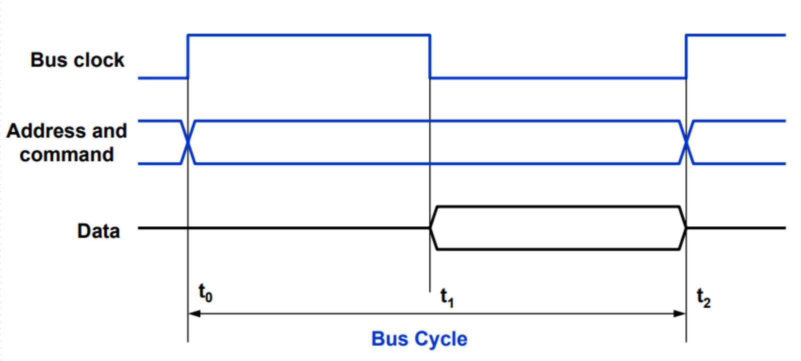
- All devices derive timing-information from a common clock-line.
- Equally spaced pulses on this line define equal time intervals.
- During a ″bus cycle‟, one data-transfer can take place.

A sequence of events during a read-operation

- At time t_0, the master (processor)
  - → places the device-address on address-lines &
  - → sends an appropriate command on control-lines (Figure).
- The command will
  - → indicate an input operation &
  - → specify the length of the operand to be read.
- Information travels over bus at a speed determined by physical & electrical characteristics.
- Clock pulse width(t_1-t_0) must be longer than max. propagation-delay b/w devices connected to bus.
- The clock pulse width should be long to allow the devices to decode the address & control signals.
- The slaves take no action or place any data on the bus before t_1.
- Information on bus is unreliable during the period t_0 to t_1 because signals are changing state.
- Slave places requested input-data on data-lines at time t_1.
- At end of clock cycle (at time t_2), master strobes (captures) data on data-lines into its input-buffer
- For data to be loaded correctly into a storage device,

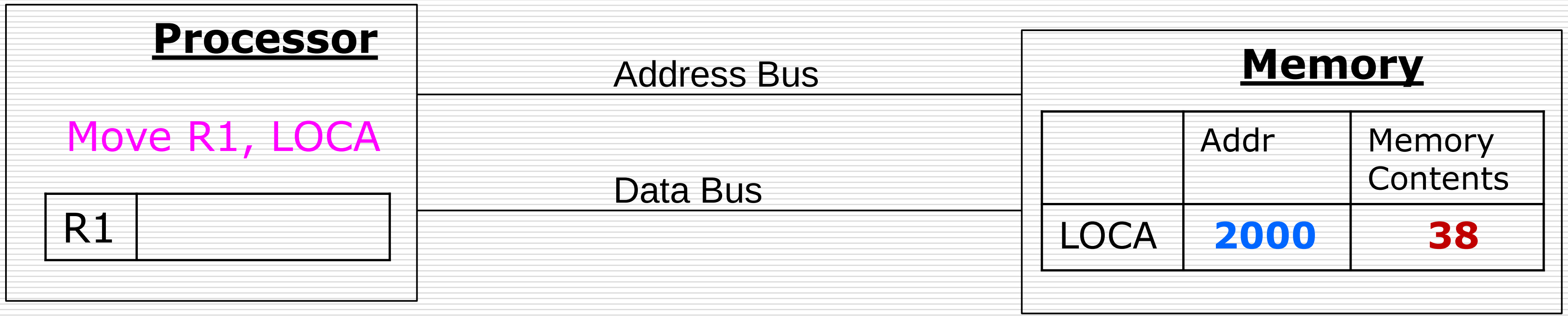
  data must be available at input of that device for a period greater than setup-time of device.

Timing of an input transfer on a synchronous bus



Synchronous Bus Example for Input Transfer or Read Operation

Before Executing the Instruction Move R1, LOCA



Processor

Move R1, LOCA

R1	

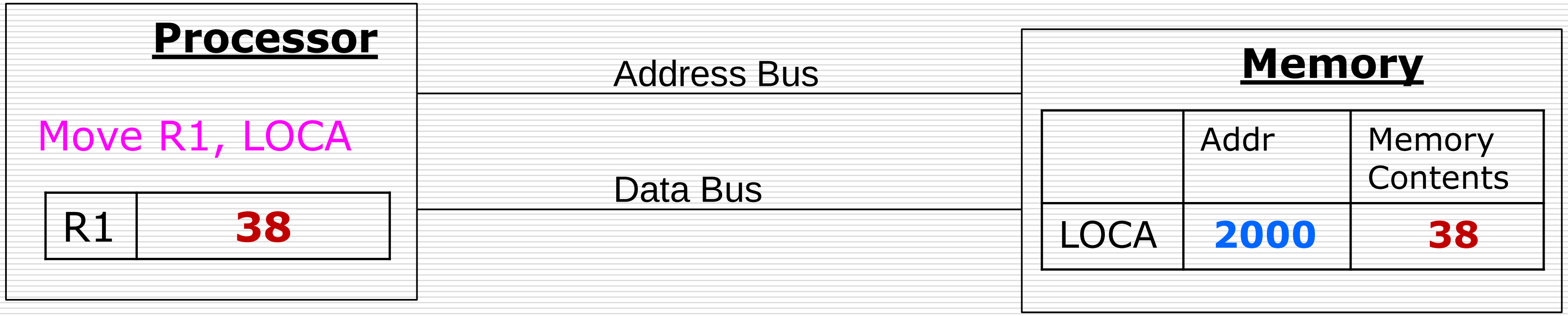
Address Bus

Data Bus

Memory

	Addr	Memory Contents
LOCA	**2000**	**38**

After Executing the Instruction Move R1, LOCA



Processor

Move R1, LOCA

R1	**38**

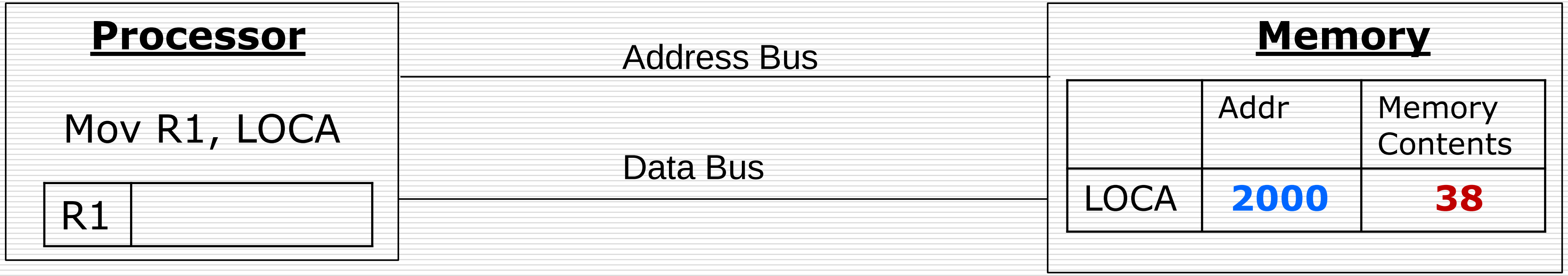
Address Bus

Data Bus

Memory

	Addr	Memory Contents
LOCA	**2000**	**38**

Synchronous Bus Example for Input Transfer or Read Operation



Processor

Mov R1, LOCA

R1

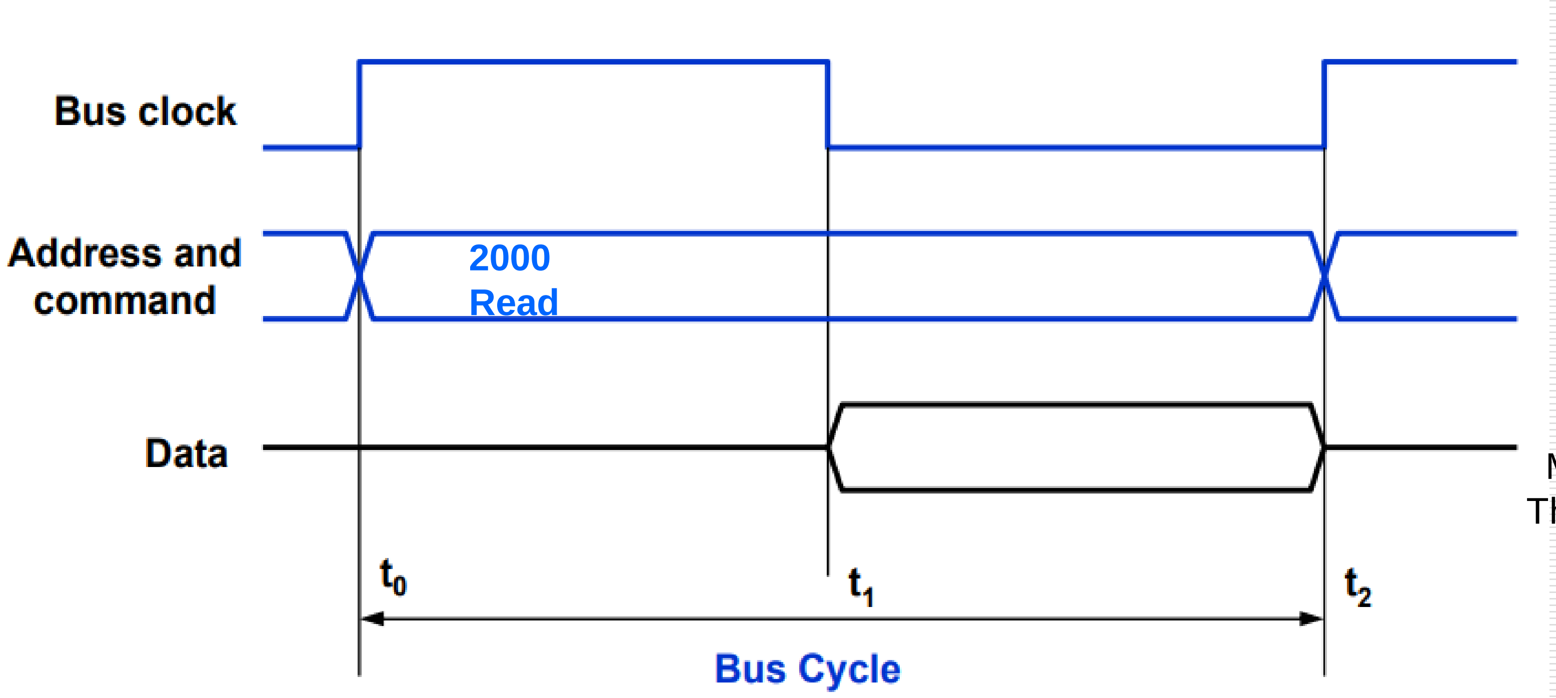
Address Bus

Data Bus

Memory

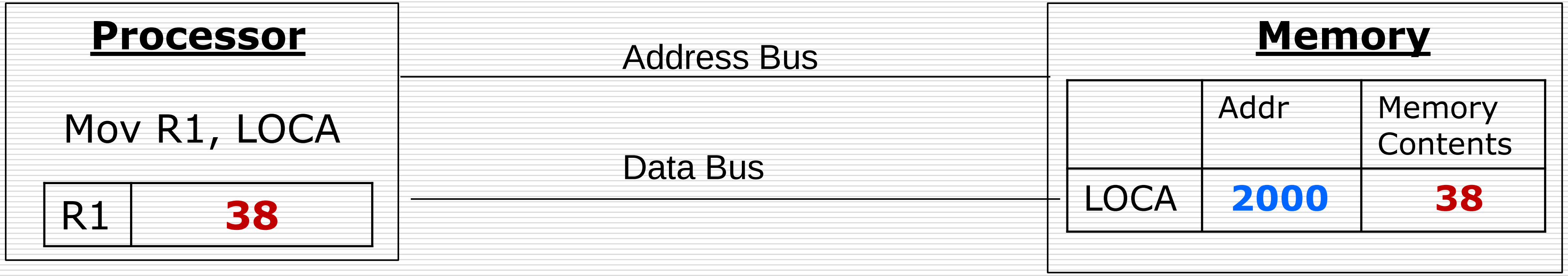
	Addr	Memory Contents
LOCA	**2000**	**38**

By
Processor
The **Master**



By
Memory
The **Slave**

Synchronous Bus Example for Input Transfer or Read Operation



Processor

Mov R1, LOCA

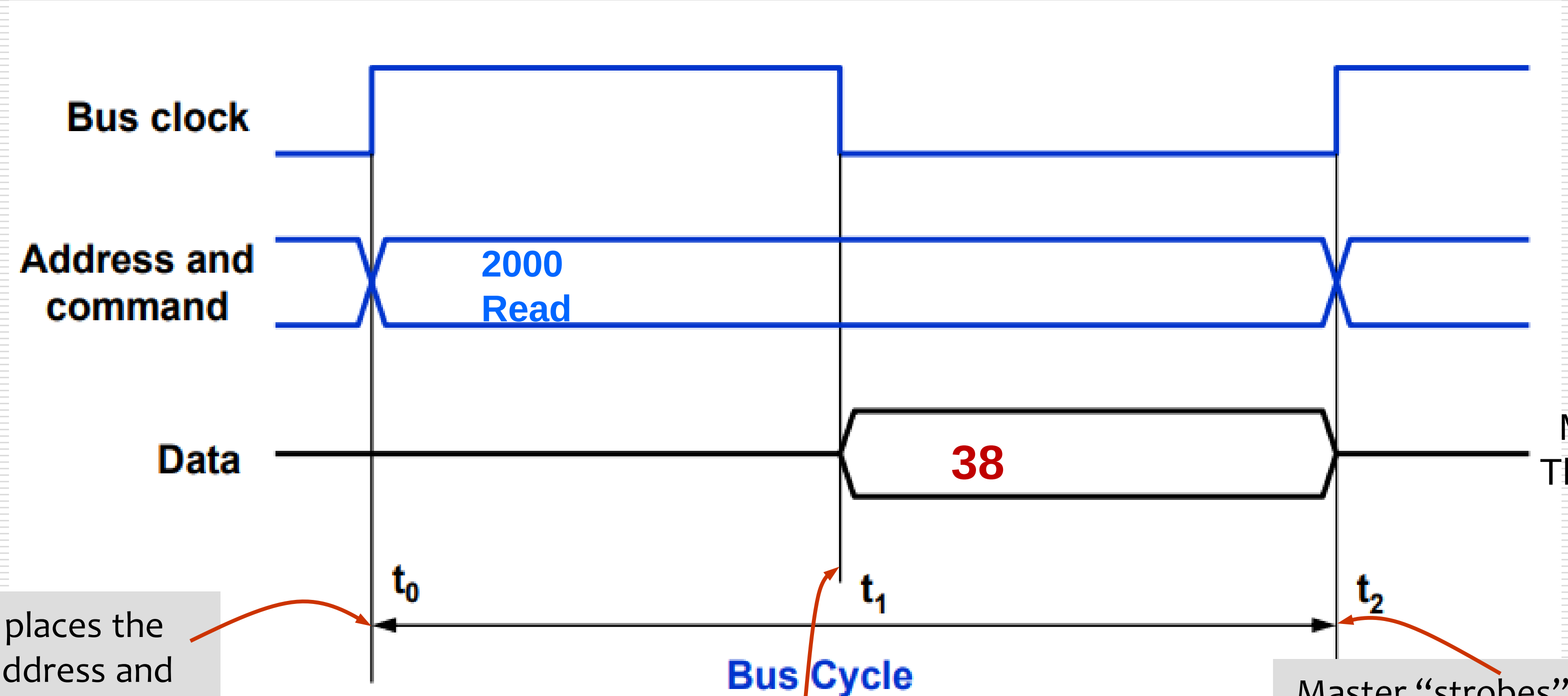
R1	38

Address Bus

Data Bus

Memory

	Addr	Memory Contents
LOCA	2000	38



By Processor The **Master**

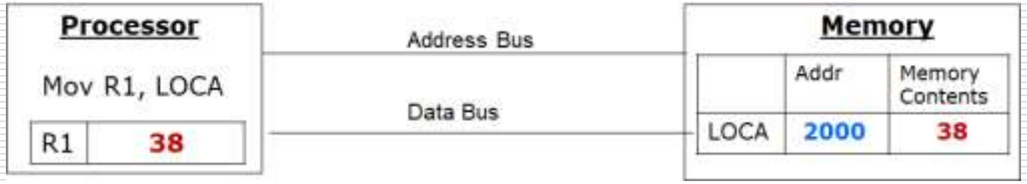
By Memory The **Slave**

Master places the device address and command on the bus, and indicates that it is a Read operation.

Addressed slave places data on the data lines

Master "strobes" the data on the data lines into its input buffer, for a Read operation.

Synchronous Bus Example for Input Transfer or Read Operation



Processor

Mov R1, LOCA

R1	38

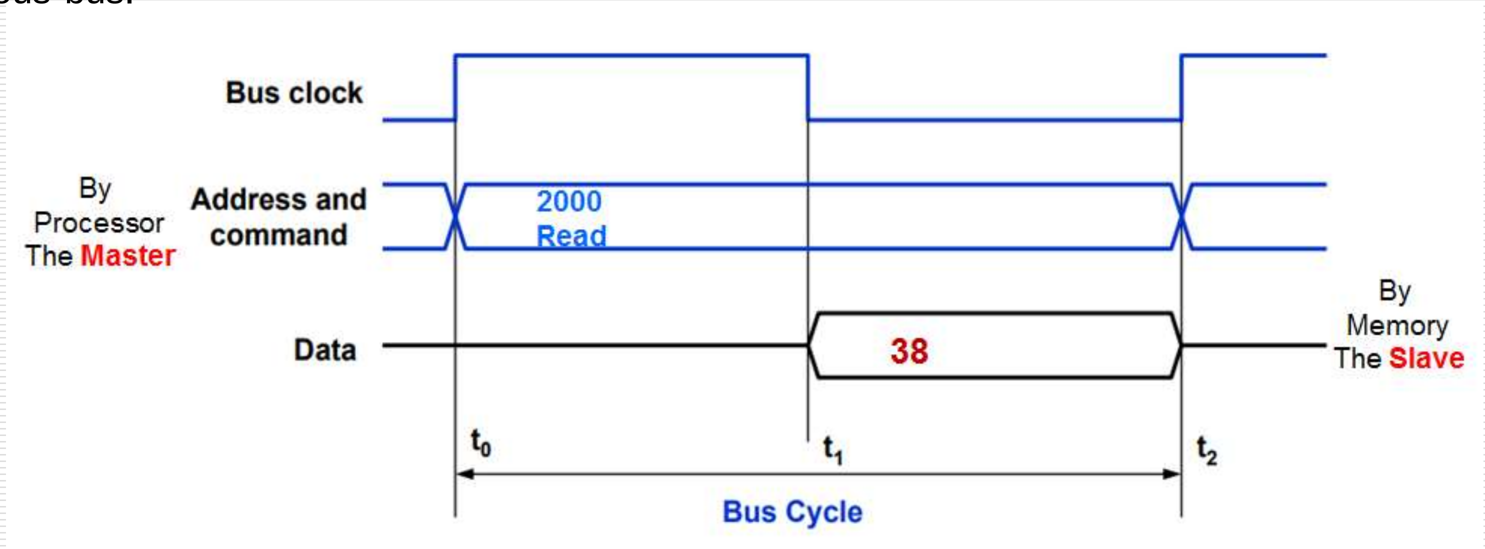
Address Bus

Data Bus

Memory

	Addr	Memory Contents
LOCA	2000	38

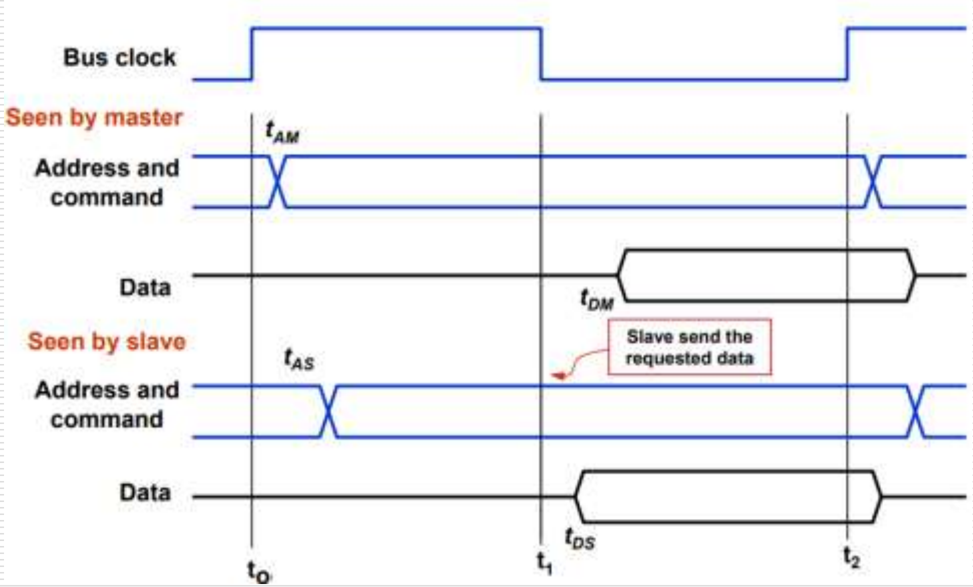
Timing Diagram of an input transfer on a synchronous bus.



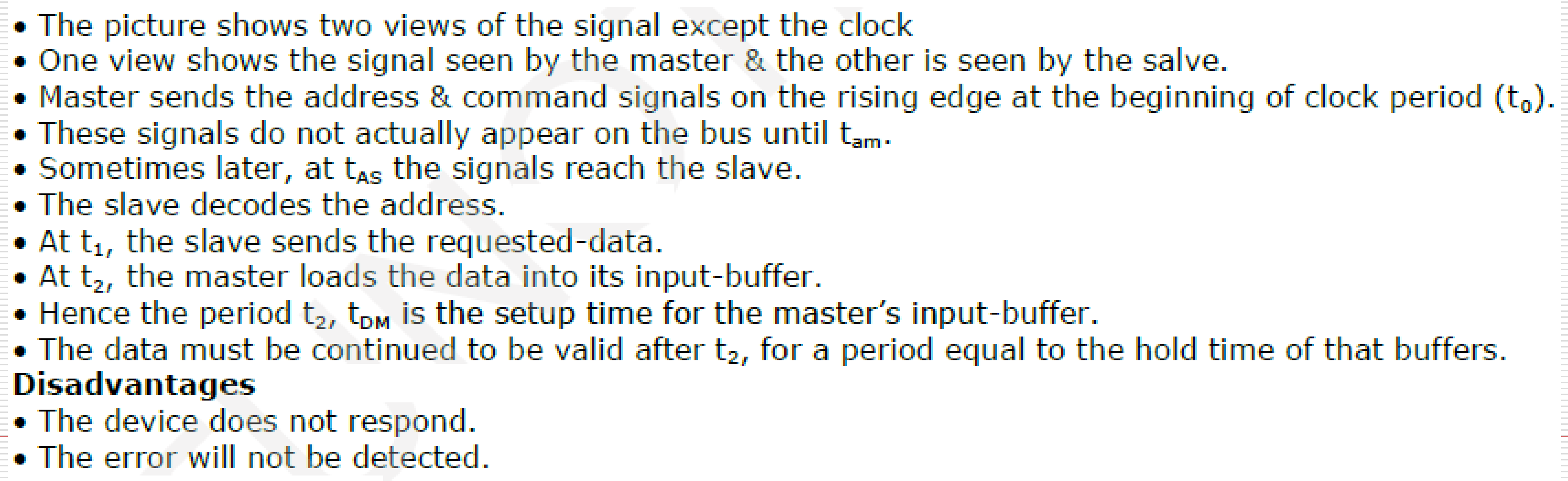
The clock pulse width, t1 – t0, must be longer than the maximum propagation delay between two devices connected to the bus. It also has to be long enough to allow all devices to decode the address and control signals so that the addressed device (the slave) can respond at time t1

It is important that slaves take no action or place any data on the bus before t1. The information on the bus is unreliable during the period t0 to t1 because signals are changing state. The addressed slave places the requested input data on the data lines at time t1. At the end of the clock cycle, at time t2, the master strobes the data on the data lines into its input buffer. In this context, “strobe” means to capture the values of the data at a given instant and store them into a buffer

Synchronous Bus: A detailed timing diagram for the input transfer



Slave send the requested data



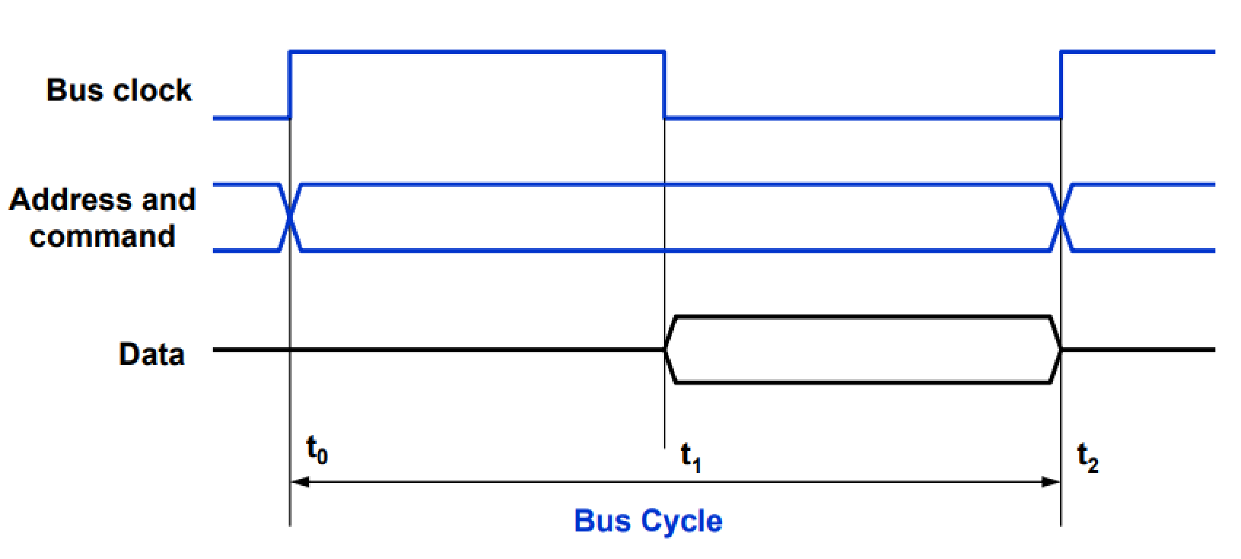
- The picture shows two views of the signal except the clock
- One view shows the signal seen by the master & the other is seen by the salve.
- Master sends the address & command signals on the rising edge at the beginning of clock period (t_0).
- These signals do not actually appear on the bus until t_{am}.
- Sometimes later, at t_{AS} the signals reach the slave.
- The slave decodes the address.
- At t_1, the slave sends the requested-data.
- At t_2, the master loads the data into its input-buffer.
- Hence the period t_2, t_{DM} is the setup time for the master's input-buffer.
- The data must be continued to be valid after t_2, for a period equal to the hold time of that buffers.

Disadvantages

- The device does not respond.
- The error will not be detected.

Question

Consider a synchronous bus that operates according to the timing diagram as shown in Figure below. The address transmitted by the processor appears on the bus after 4 ns. The propagation delay on the bus wires between the processor and different devices connected varies from 1 to 5 ns, address decoding takes 6 ns, and the addressed device takes between 5 and 10 ns to place the requested data on the bus. The input buffer needs 3 ns of setup time. What is the maximum clock period at which this bus can operate?

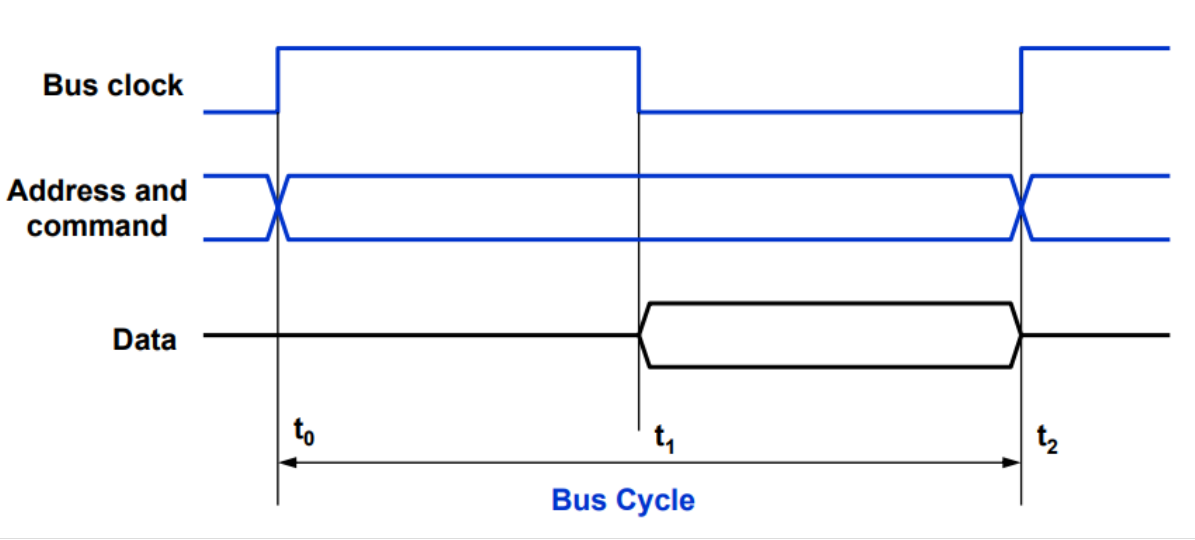


Answer

Consider a synchronous bus that operates according to the timing diagram as shown in Figure below. The address transmitted by the processor appears on the bus after 4 ns. The propagation delay on the bus wires between the processor and different devices connected varies from 1 to 5 ns, address decoding takes 6 ns, and the addressed device takes between 5 and 10 ns to place the requested data on the bus. The input buffer needs 3 ns of setup time. What is the maximum clock period at which this bus can operate?

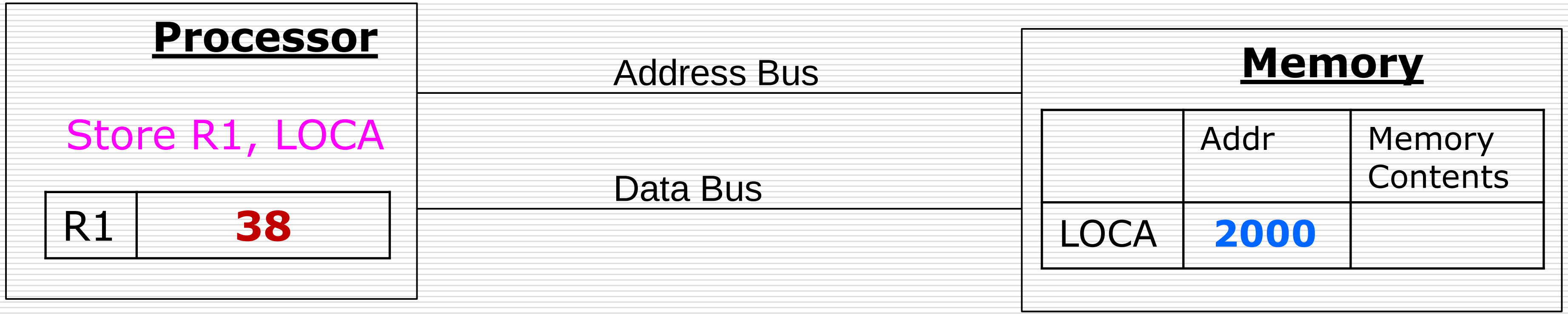
Answer:

Minimum clock period = 4+5+6+10+3 = 28 ns



A Synchronous Bus Example for Output Transfer or Write Operation

Before Executing the Instruction Store R1, LOCA



Processor

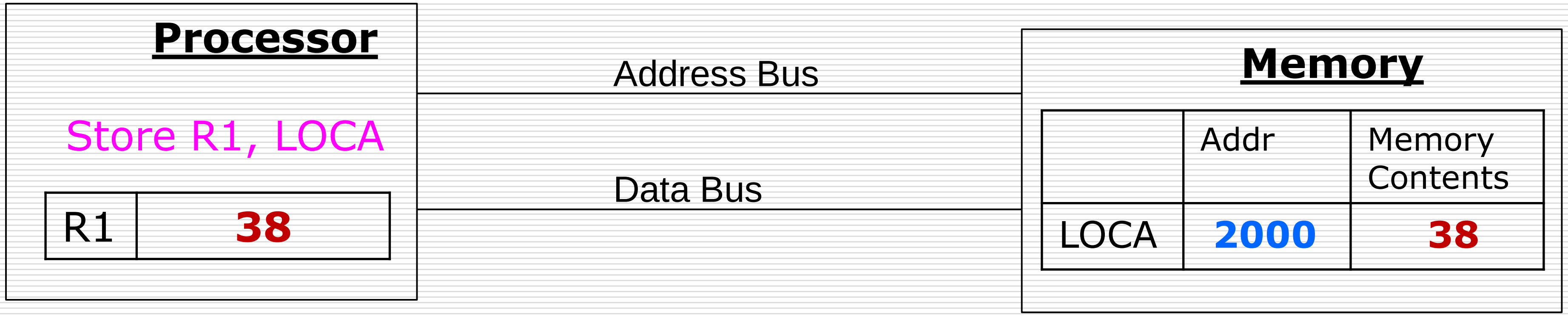
Store R1, LOCA

R1	38

Memory

	Addr	Memory Contents
LOCA	2000	

After Executing the Instruction Store R1, LOCA



Processor

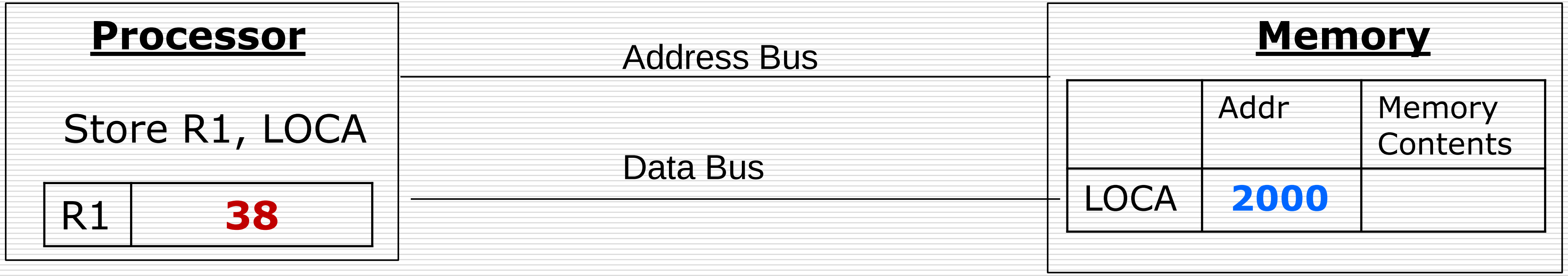
Store R1, LOCA

R1	38

Memory

	Addr	Memory Contents
LOCA	2000	38

A Synchronous Bus Example for Output Transfer or Write Operation



Processor

Store R1, LOCA

R1	38

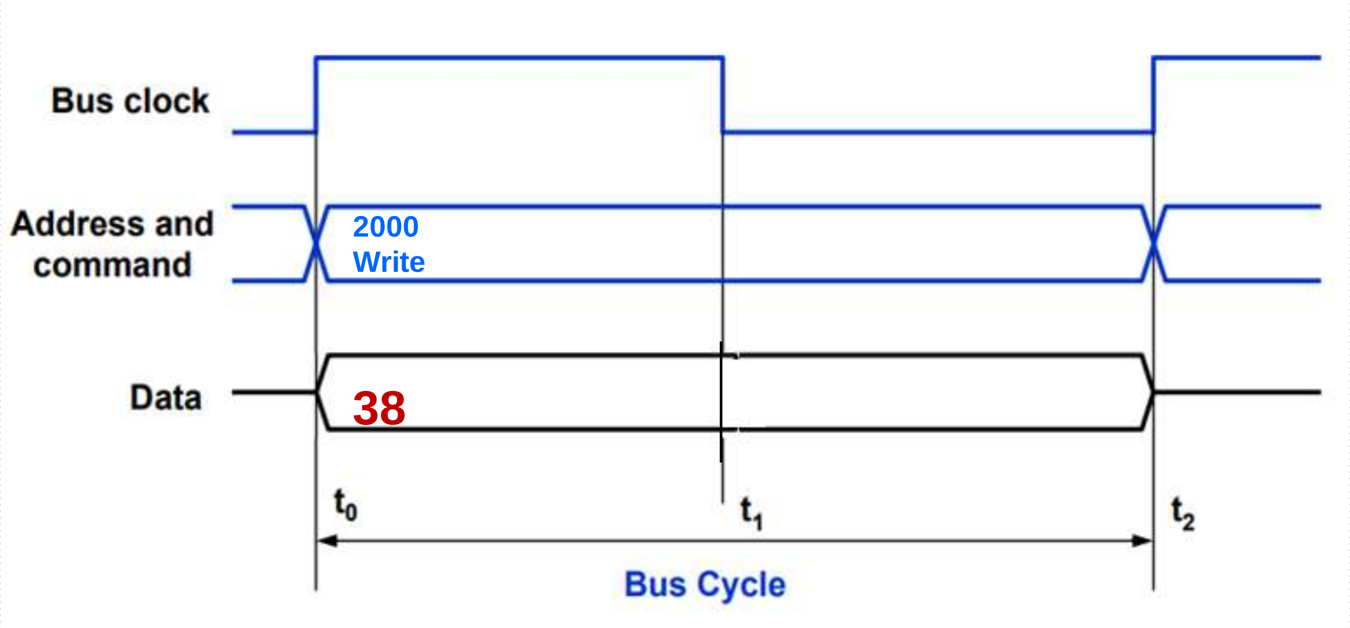
Address Bus

Data Bus

Memory

	Addr	Memory Contents
LOCA	2000	

By
Processor
The **Master**



By
Memory
The **Slave**

- *In case of a Write operation, the master places the data on the bus along with the address and commands at time t_o.*
- *The slave strobes the data into its input buffer at time t_2.*

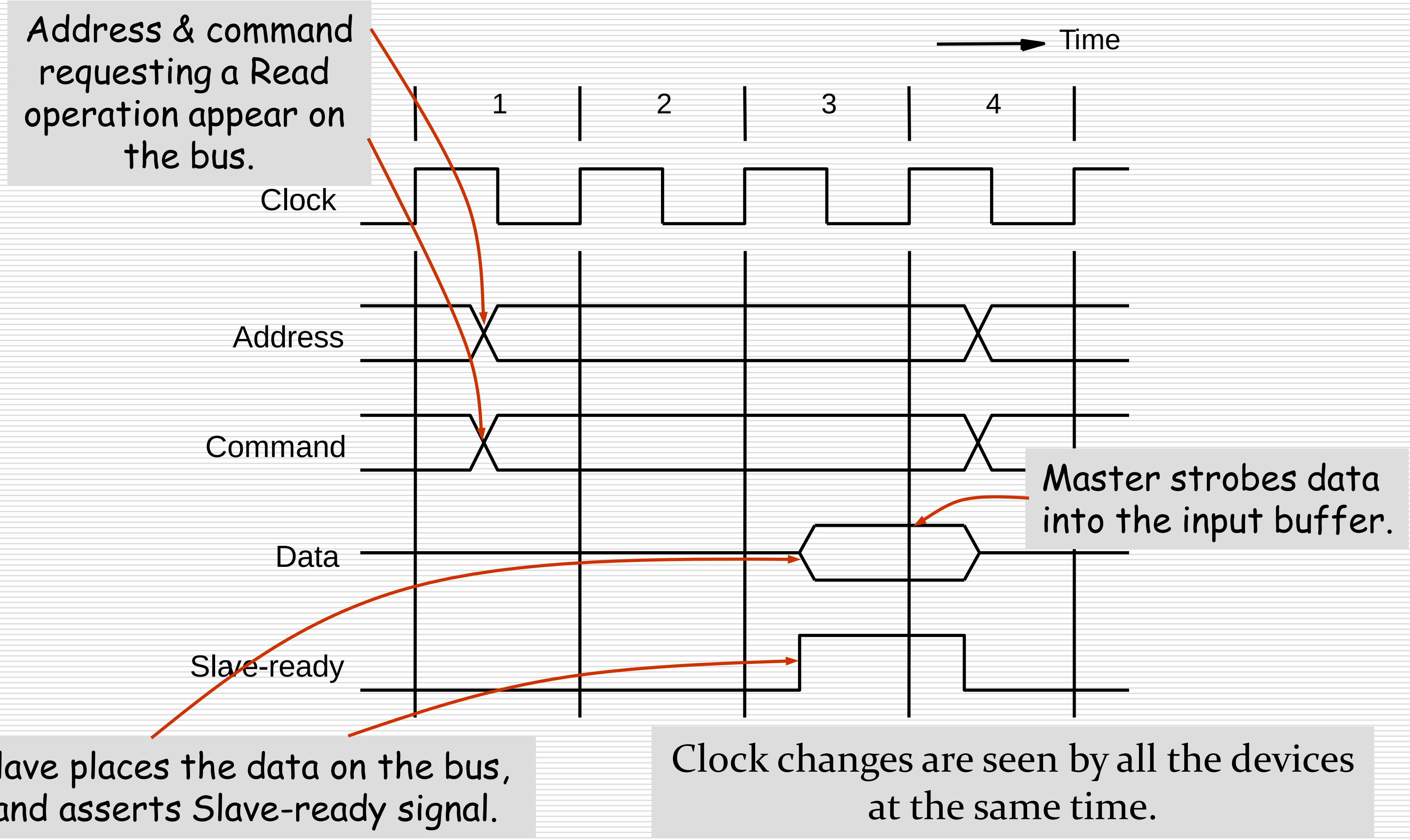
Synchronous bus

- Data transfer has to be completed within one clock cycle.
  - Clock period t_2 - t_0 must be such that the longest propagation delay on the bus and the slowest device interface must be accommodated.
  - Forces all the devices to operate at the speed of the slowest device.
- Processor just assumes that the data are available at t2 in case of a Read operation, or are read by the device in case of a Write operation.
  - What if the device is actually failed, and never really responded?

Synchronous bus (contd..)

- Most buses have control signals to represent a response from the slave.
- Control signals serve two purposes:
  - Inform the master that the slave has recognized the address, and is ready to participate in a data transfer operation.
  - Enable to adjust the duration of the data transfer operation based on the speed of the participating slaves.
- High-frequency bus clock is used:
  - Data transfer spans several clock cycles instead of just one clock cycle as in the earlier case.

Synchronous bus : Transfer by Multiple Clock Cycles



Address & command requesting a Read operation appear on the bus.

Master strobes data into the input buffer.

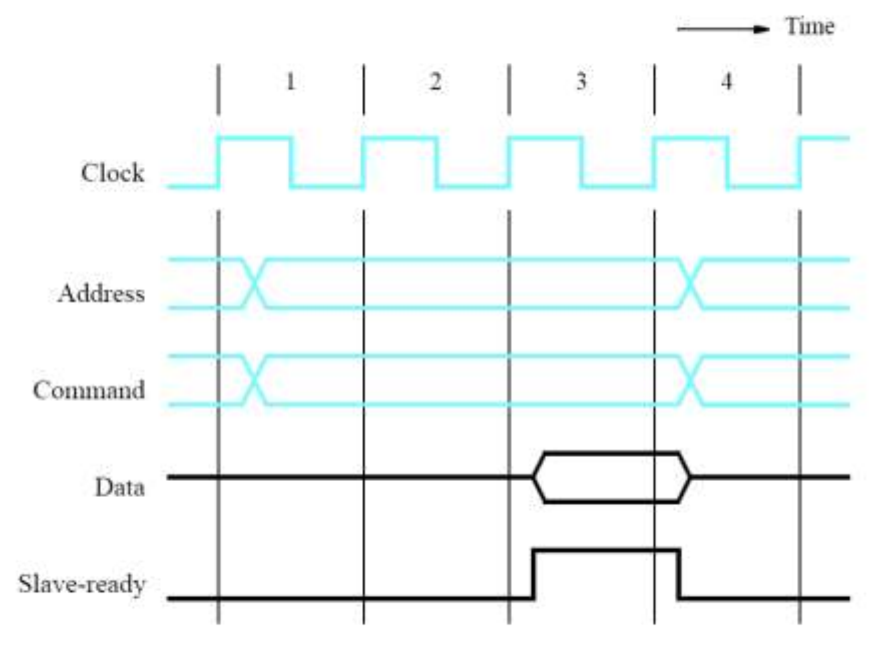
Slave places the data on the bus, and asserts Slave-ready signal.

Clock changes are seen by all the devices at the same time.

Question

Consider a synchronous bus that operates according to the timing diagram in figure as given. The bus and the interface circuitry connected to it have the following parameters:

Bus driver delay	2ns
Propagation delay on the bus	5 to 10 ns
Address decoder delay	6 ns
Time to fetch the requested data	0 to 25 ns
Setup time	1.5 ns



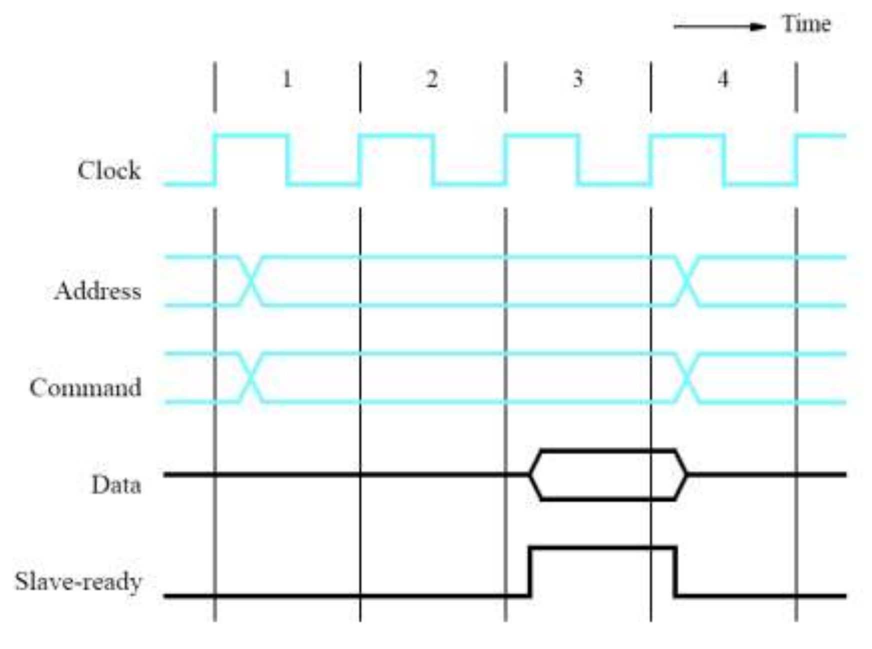
a. What is the maximum clock speed at which this bus can operate?

b. How many clock cycles are needed to complete an input operation?

Answer

Consider a synchronous bus that operates according to the timing diagram in figure as given. The bus and the interface circuitry connected to it have the following parameters:

Bus driver delay	2ns
Propagation delay on the bus	5 to 10 ns
Address decoder delay	6 ns
Time to fetch the requested data	0 to 25 ns
Setup time	1.5 ns



a. What is the maximum clock speed at which this bus can operate?

44.5 nano seconds

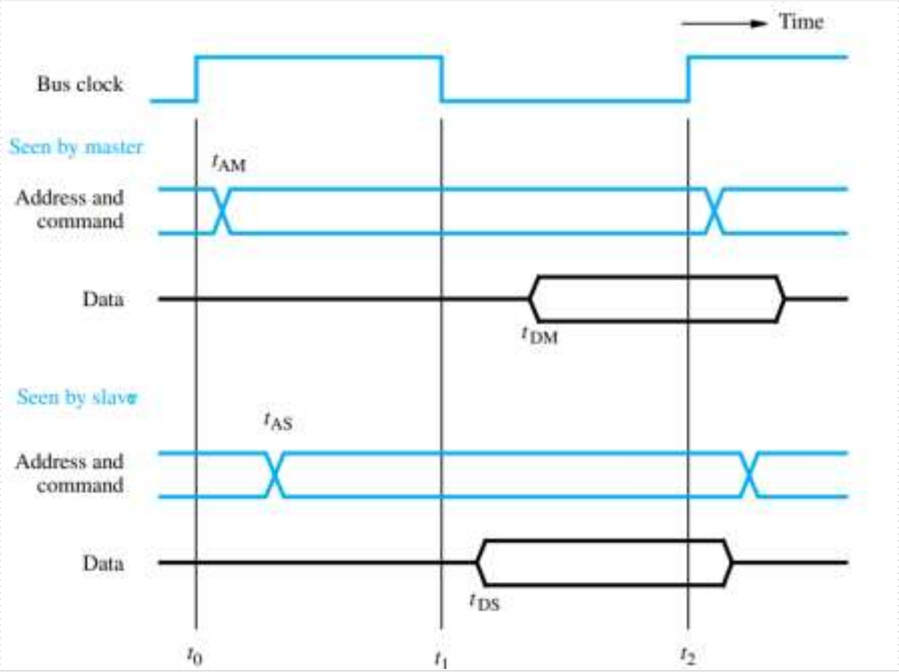
b. How many clock cycles are needed to complete an input operation?

4 clock cycles

Question

The I/O bus of a computer uses the synchronous protocol as shown in Figure. Maximum propagation delay on this bus is 4 ns. The bus master takes 1.5 ns to place an address on the address lines. Slave devices require 3 ns to decode the address and a maximum of 5 ns to place the requested data on the data lines. Input registers connected to the bus have a minimum setup time of 1 ns. Assume that the bus clock has a 50% duty cycle; that is, the high and low phases of the clock are of equal duration. What is the maximum clock frequency for this bus?.

Note: Let *clock period* or *cycle time*, T_c, is the time between rising edges of a repetitive clock signal. Its reciprocal, $f_c = 1/T_c$, is the *clock frequency*.



Question

The I/O bus of a computer uses the synchronous protocol as shown in Figure. Maximum propagation delay on this bus is 4 ns. The bus master takes 1.5 ns to place an address on the address lines. Slave devices require 3 ns to decode the address and a maximum of 5 ns to place the requested data on the data lines. Input registers connected to the bus have a minimum setup time of 1 ns. Assume that the bus clock has a 50% duty cycle; that is, the high and low phases of the clock are of equal duration. What is the maximum clock frequency for this bus?.

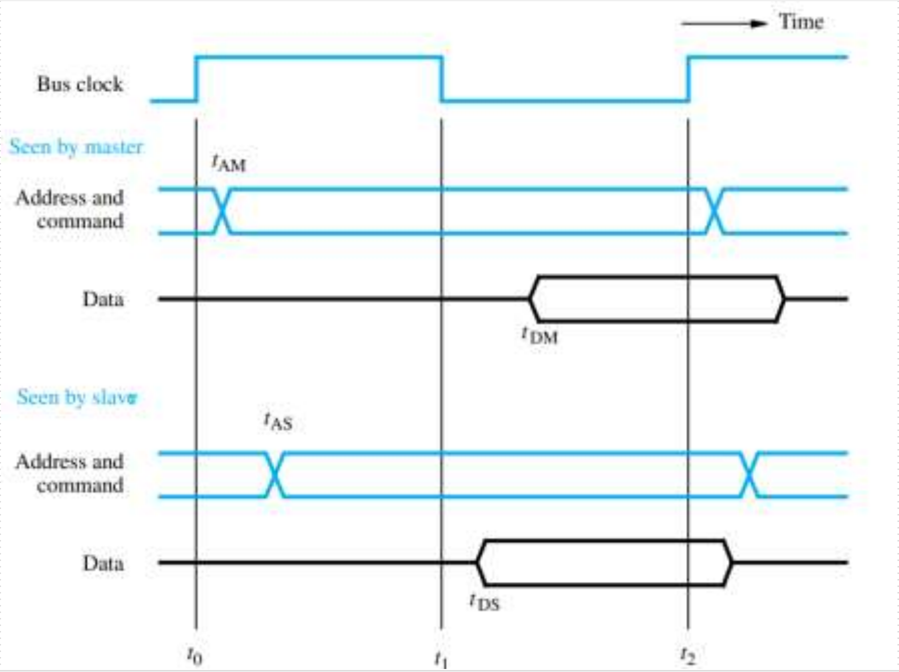
Note: Let *clock period* or *cycle time*, T_c, is the time between rising edges of a repetitive clock signal. Its reciprocal, $f_c = 1/T_c$, is the *clock frequency*.

Solution:

The minimum time for the high phase of the clock is the time for the address to arrive and be decoded by the slave, which is 1.5 + 4 + 3 = 8.5 ns.

The minimum time for the low phase of the clock is the time for the slave to place data on the bus and for the master to load the data into a register, which is 5 + 4 + 1 = 10 ns.

Then, the minimum clock period is 2 × 10 = 20 ns, and the maximum clock frequency is 50 MHz.



Two different schemes for timing of data transfers over a bus

1. **Synchronous** scheme
2. **Asynchronous** scheme

1. Synchronous data transfer: sender and receiver use the same clock signal

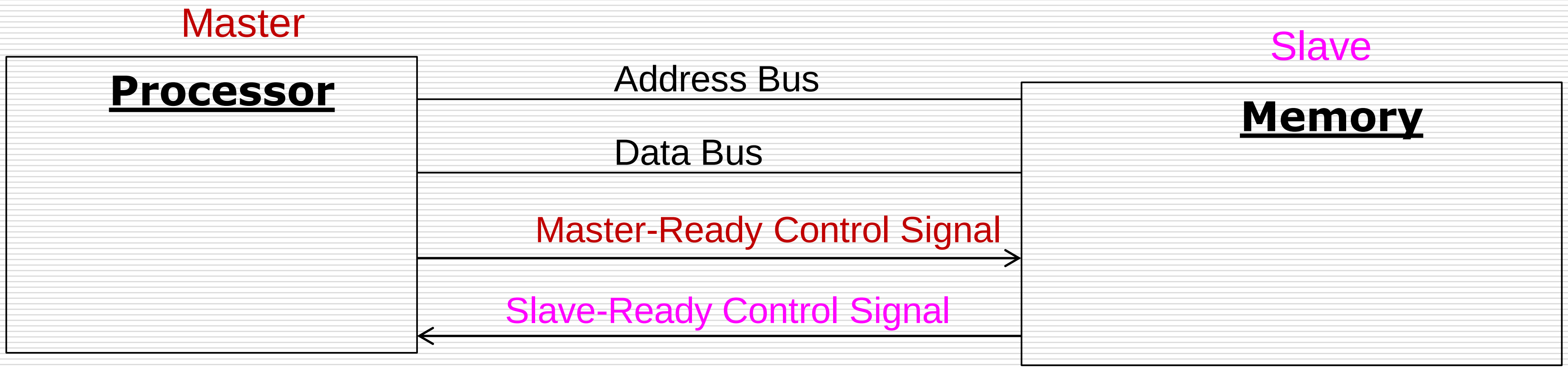
- Needs clock signal between the sender and the receiver
- Supports high data transfer rate

2. Asynchronous data transfer: sender provides a synchronization signal to the receiver before starting the transfer of each message

- Does not need clock signal between the sender and the receiver
- Slower data transfer rate

Asynchronous bus

- Under Asynchronous bus, Data transfers on the bus is controlled by a handshake between the master and the slave. A handshake is an exchange of command and response signals between the master and the slave.
- Common clock in the synchronous bus case is replaced by two timing control lines:
  - Master-ready,
  - Slave-ready.
- Master-ready signal is asserted by the master to indicate to the slave that it is ready to participate in a data transfer.
- Slave-ready signal is asserted by the slave in response to the master-ready from the master, and it indicates to the master that the slave is ready to participate in a data transfer.

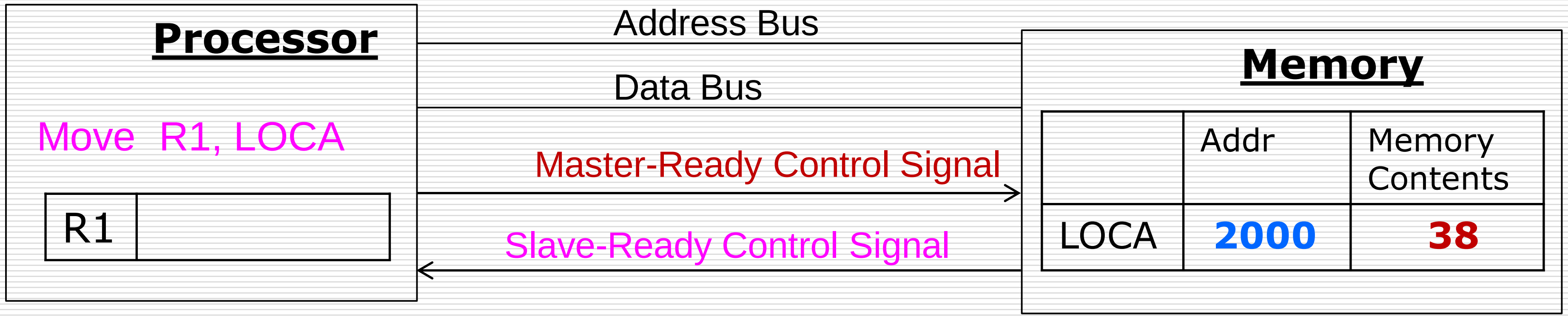


Asynchronous bus (contd..)

- Data transfer using the handshake protocol:
  - Master places the address and command information on the bus.
  - Asserts the Master-ready signal to indicate to the slave that the address and command information has been placed on the bus.
  - All devices on the bus decode the address.
  - Addressed slave performs the required operation, and informs the processor it has done so by asserting the Slave-ready signal.
  - Master removes all the signals from the bus, once Slave-ready is asserted.
  - If the operation is a Read operation, Master also strobes the data into its input buffer.

Asynchronous Bus Example for Input Transfer or Read Operation

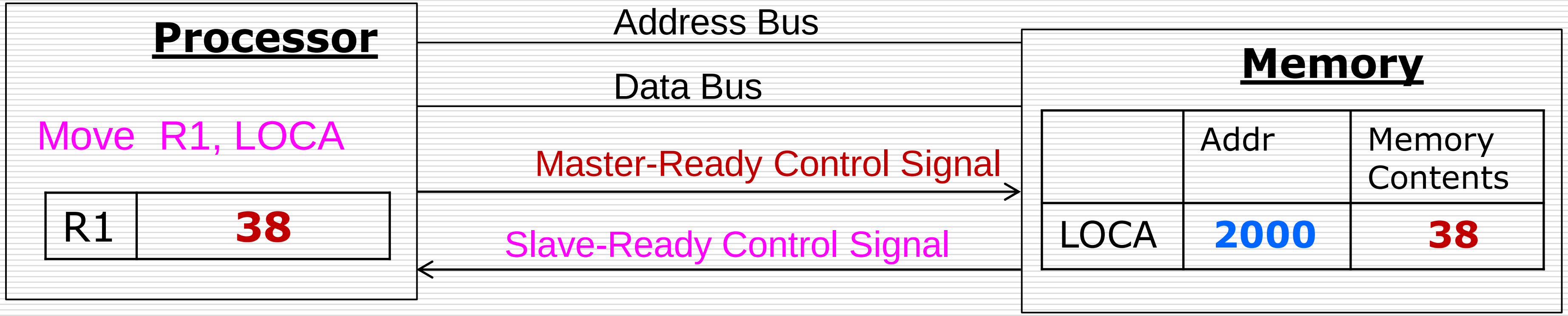
Before Executing the Instruction Move R1, LOCA



R1	

	Addr	Memory Contents
LOCA	**2000**	**38**

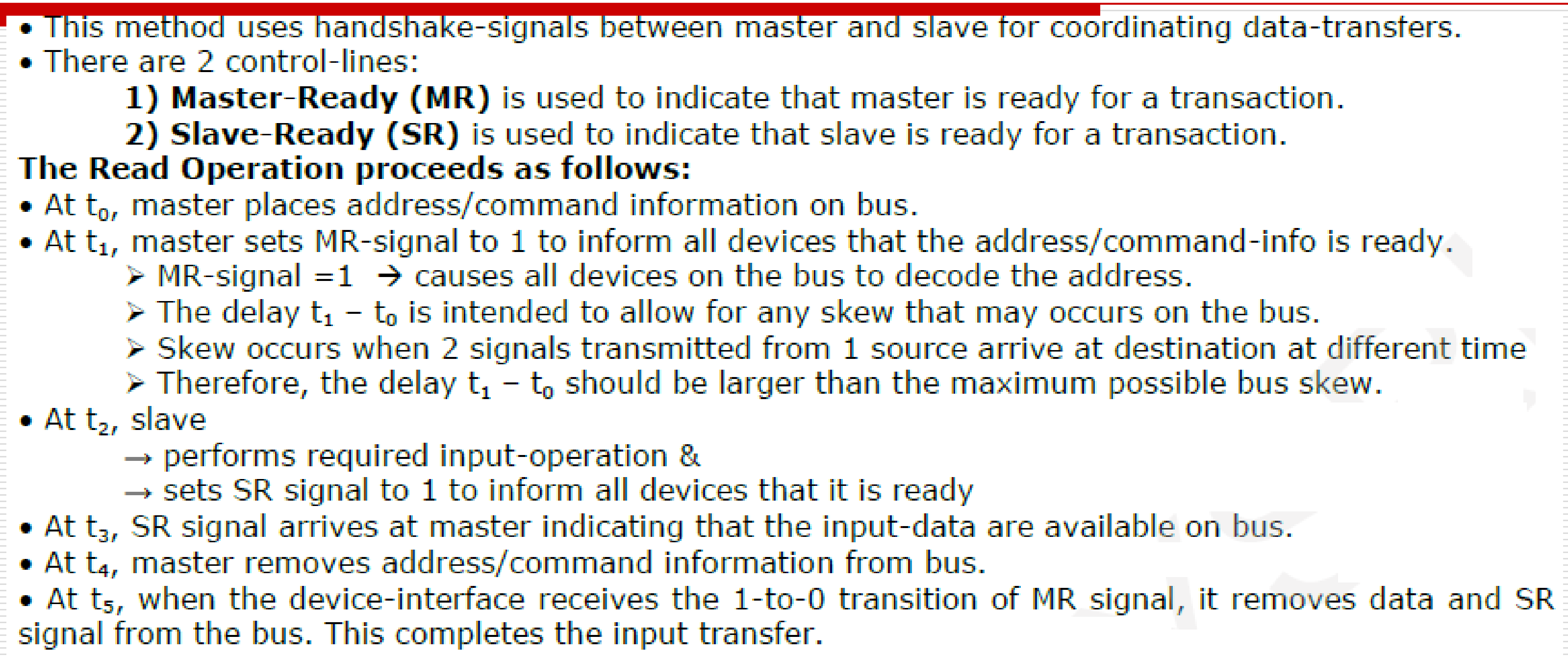
After Executing the Instruction Move R1, LOCA



R1	**38**

	Addr	Memory Contents
LOCA	**2000**	**38**

Asynchronous Bus: For Input Transfer or Read Operation



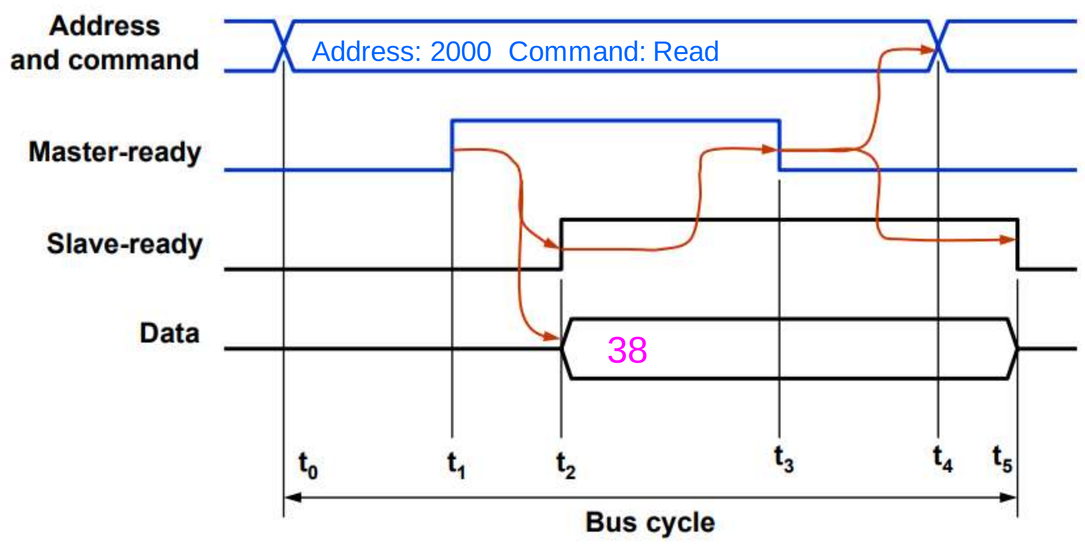
- This method uses handshake-signals between master and slave for coordinating data-transfers.
- There are 2 control-lines:

  1) Master-Ready (MR) is used to indicate that master is ready for a transaction.

  2) Slave-Ready (SR) is used to indicate that slave is ready for a transaction.

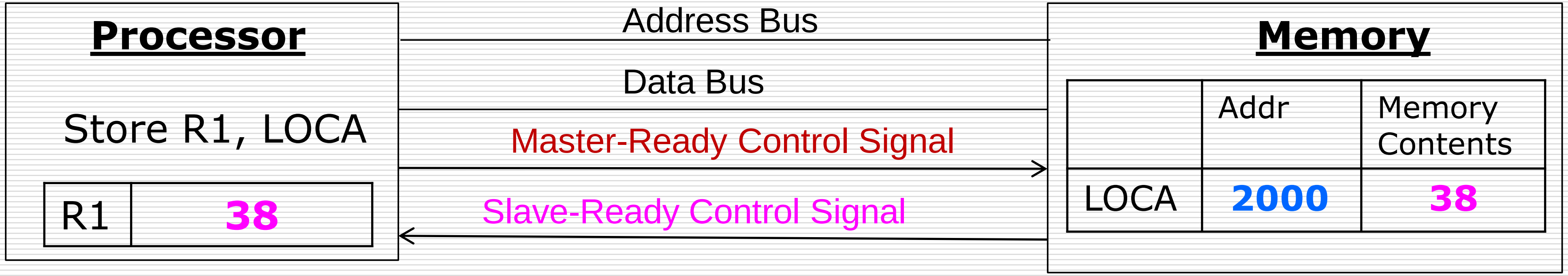
The Read Operation proceeds as follows:

- At t_0, master places address/command information on bus.
- At t_1, master sets MR-signal to 1 to inform all devices that the address/command-info is ready.
  - ➢ MR-signal =1 → causes all devices on the bus to decode the address.
  - ➢ The delay $t_1 - t_0$ is intended to allow for any skew that may occurs on the bus.
  - ➢ Skew occurs when 2 signals transmitted from 1 source arrive at destination at different time
  - ➢ Therefore, the delay $t_1 - t_0$ should be larger than the maximum possible bus skew.
- At t_2, slave
  - → performs required input-operation &
  - → sets SR signal to 1 to inform all devices that it is ready
- At t_3, SR signal arrives at master indicating that the input-data are available on bus.
- At t_4, master removes address/command information from bus.
- At t_5, when the device-interface receives the 1-to-0 transition of MR signal, it removes data and SR signal from the bus. This completes the input transfer.



Asynchronous bus Timing diagram:
Handshake control of data transfer
during Input operation

Asynchronous Bus Example for Output Transfer or Write Operation



Processor

Store R1, LOCA

R1	**38**

Address Bus

Data Bus

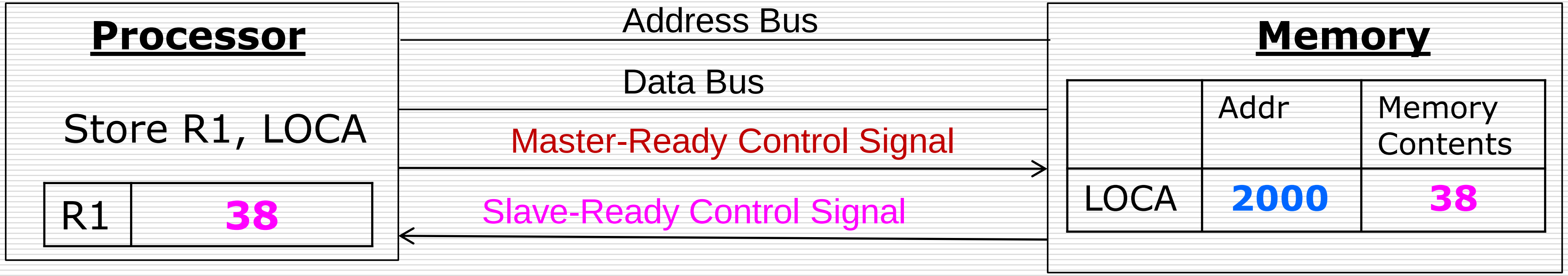
Master-Ready Control Signal

Slave-Ready Control Signal

Memory

	Addr	Memory Contents
LOCA	**2000**	**38**

Asynchronous Bus Example for Output Transfer or Write Operation



Processor

Store R1, LOCA

R1	38

Address Bus

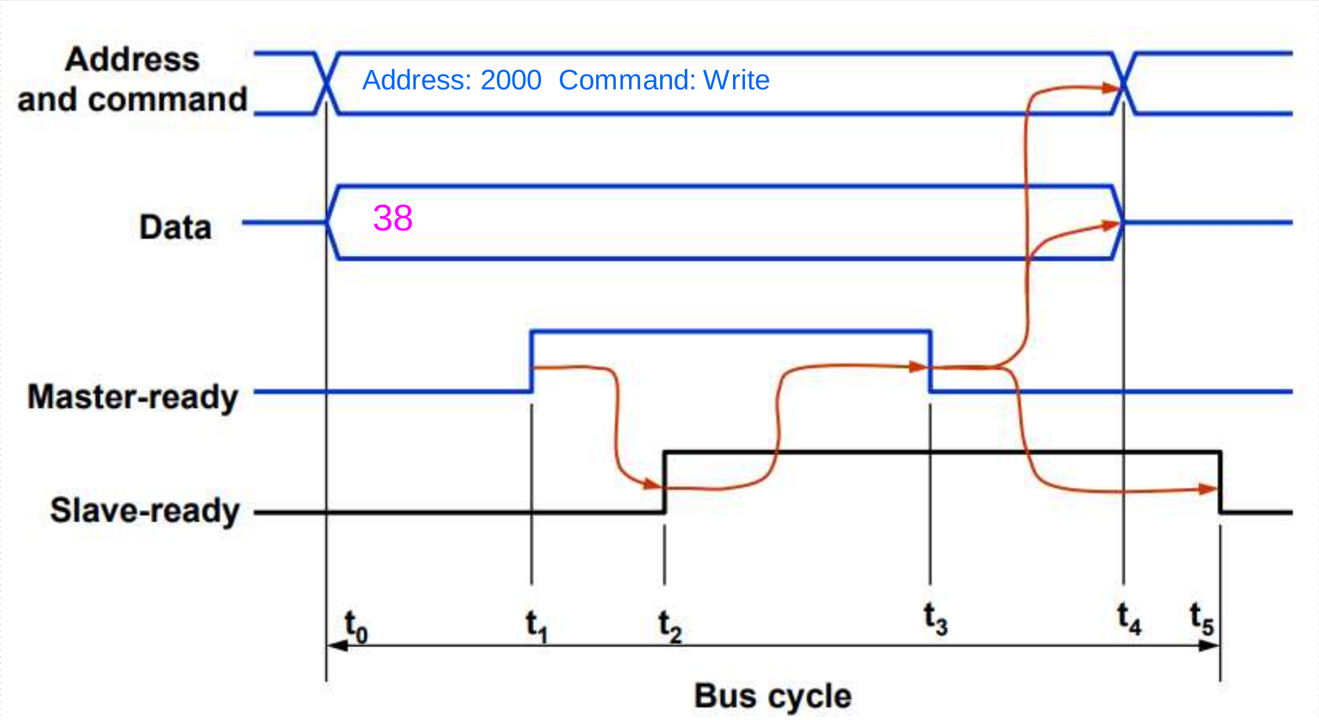
Data Bus

Master-Ready Control Signal

Slave-Ready Control Signal

Memory

	Addr	Memory Contents
LOCA	2000	38



Asynchronous bus Timing diagram:
Handshake control of data transfer
during Output operation

Asynchronous vs. Synchronous bus

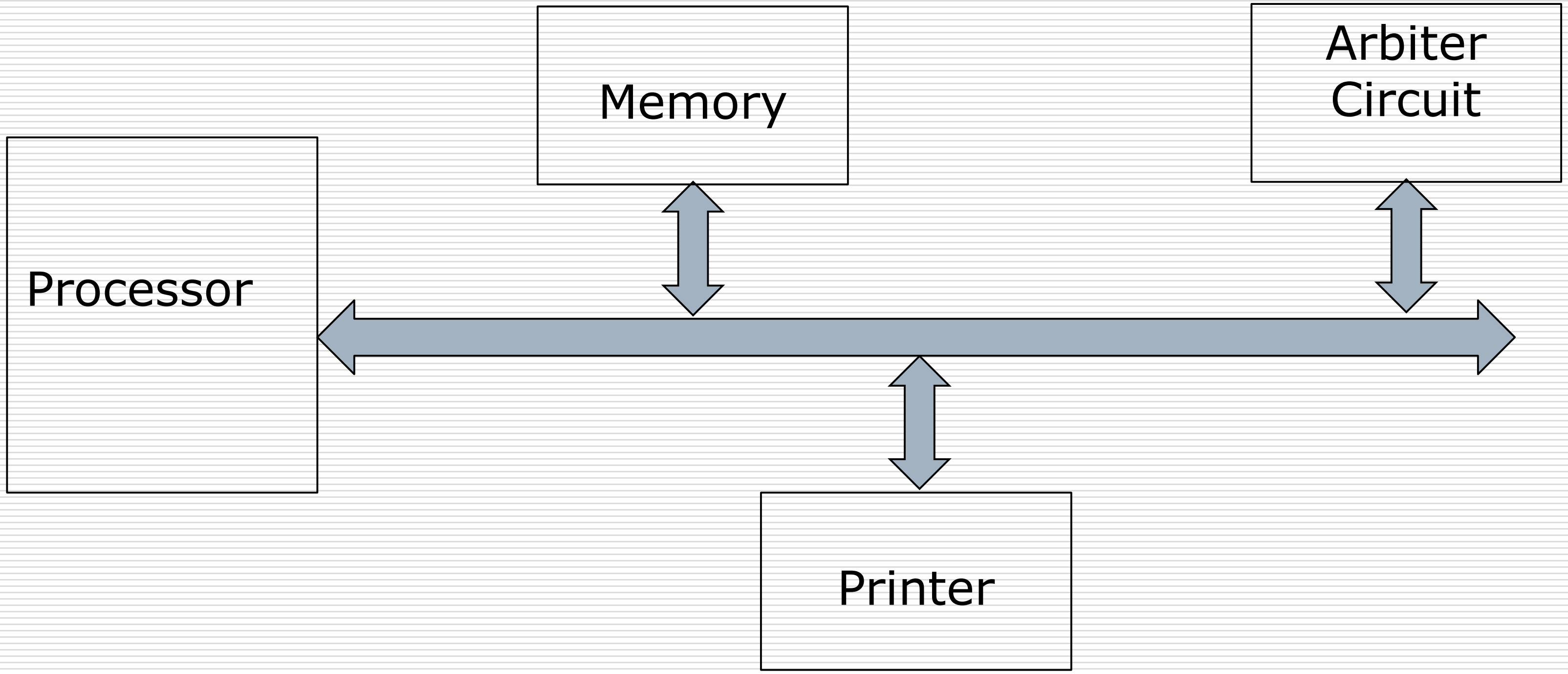
- Advantages of asynchronous bus:
  - Eliminates the need for synchronization between the sender and the receiver.
  - Can accommodate varying delays automatically, using the Slave-ready signal.
- Disadvantages of asynchronous bus:
  - Data transfer rate with full handshake is limited by two-round trip delays.
  - Data transfers using a synchronous bus involves only one round trip delay, and hence a synchronous bus can achieve faster rates.

Bus Arbitration

- There are occasions when two or more entities contend for the use of a single resource in a computer system. For example, two devices may need to access a given slave at the same time. In such cases, it is necessary to decide which device will access the slave first.
- The decision is usually made in an arbitration process performed by an arbiter circuit. The arbitration process starts by each device sending a request to use the shared resource. The arbiter associates **priorities** with individual requests. If it receives two requests at the same time, it **grants the use of the slave** to the device having the **higher priority first**.

Rough slide to explain Bus Arbitration

Example



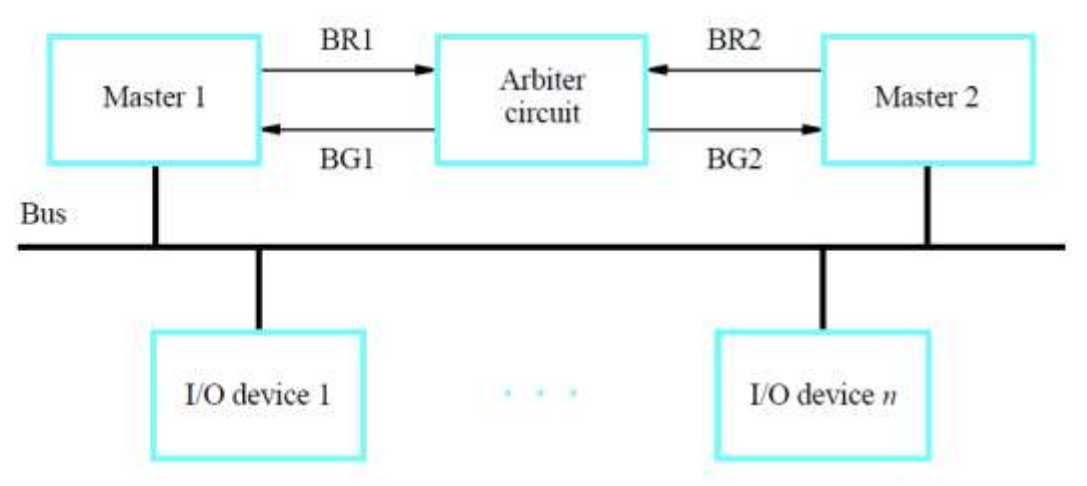
Bus Arbitration (Contd…)

To illustrate the arbitration process, we 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.

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 below, 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.

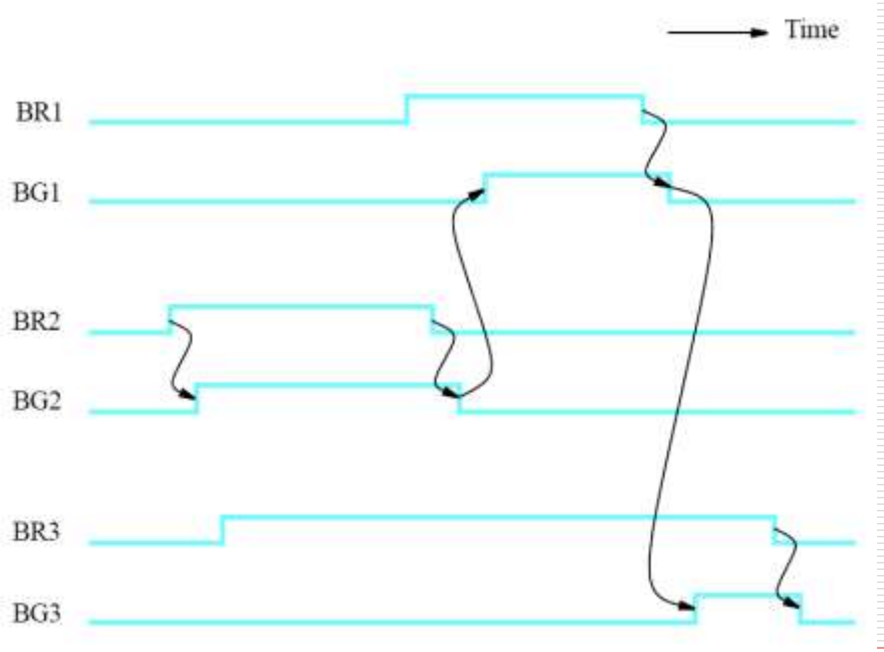
A master requests use of the bus by activating its Bus-request line. If a single Bus-request is activated, the arbiter activates the corresponding Bus-grant. This indicates to the selected master that it may now use the bus for transferring data. When the transfer is completed, that master deactivates its Bus-request, and the arbiter deactivates its Bus-grant.



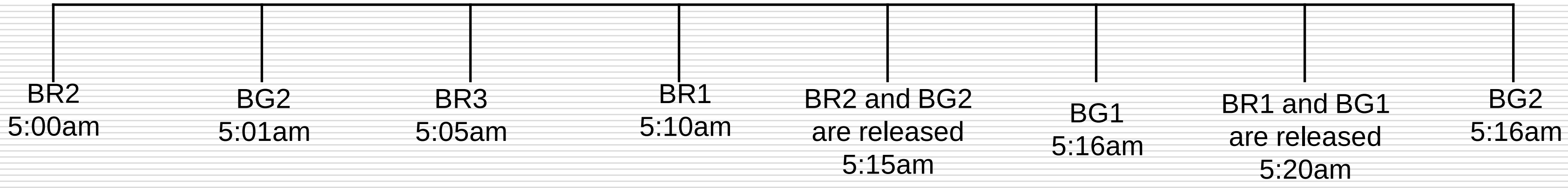
Bus Arbitration

Bus Arbitration (Contd…)

Figure below, 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.



Example



BR2 5:00am

BG2 5:01am

BR3 5:05am

BR1 5:10am

BR2 and BG2 are released 5:15am

BG1 5:16am

BR1 and BG1 are released 5:20am

BG2 5:16am

Granting use of the bus based on priorities:
BR1 > BR2 > BR3

Question

An arbiter controls access to a common resource. It uses a rotating-priority scheme in responding to requests on lines R1 through R4. Initially, R1 has the highest priority and R4 the lowest priority. After a request on one of the lines receives service, that line drops to the lowest priority, and the next line in sequence becomes the highest-priority line. For example, after R2 has been serviced, the priority order, starting with the highest, becomes R3, R4, R1, R2. What will be the sequence of grants for the following sequence of requests: R3, R1, R4, R2? Assume that the last three requests arrive while the first one is being serviced.

Answer

An arbiter controls access to a common resource. It uses a rotating-priority scheme in responding to requests on lines R1 through R4. Initially, R1 has the highest priority and R4 the lowest priority. After a request on one of the lines receives service, that line drops to the lowest priority, and the next line in sequence becomes the highest-priority line. For example, after R2 has been serviced, the priority order, starting with the highest, becomes R3, R4, R1, R2. What will be the sequence of grants for the following sequence of requests: R3, R1, R4, R2? Assume that the last three requests arrive while the first one is being serviced.

Answer:

R3 , R4 , R1 , R2

Question

The device which is allowed to initiate data transfers on the BUS at any time is called ______

a)Processor
b)BUS arbiter
c)Controller
d)BUS master

Question

The device which is allowed to initiate data transfers on the BUS at any time is called ______

a)Processor

b)BUS arbiter

c)Controller

d)BUS master

Answer : d

Thanks for Listening

The asynchronous bus protocol in Figure 7.6 uses a full-handshake, in which the master maintains an asserted signal on Master-ready until it receives Slave-ready, the slave keeps Slave-ready asserted until Master-ready becomes inactive, and so on. Consider an alternative protocol in which each of these signals is a pulse of a fixed width of 4 ns. Devices take action only on the rising edge of the pulse. Using the same parameters as in Problem 7.7, what is the minimum and maximum time to complete one bus transfer?

Bus driver delay 2 ns Propagation delay on the bus 5 to 10 ns Address decoder delay 6 ns Time to fetch the requested data 0 to 25 ns Setup time 1.5 ns

- minimum=2+5+4=11. we count the bus driver delay, propagation delay on the bus and the fixed with whcih is 4.
- maximum 2+10+4=16

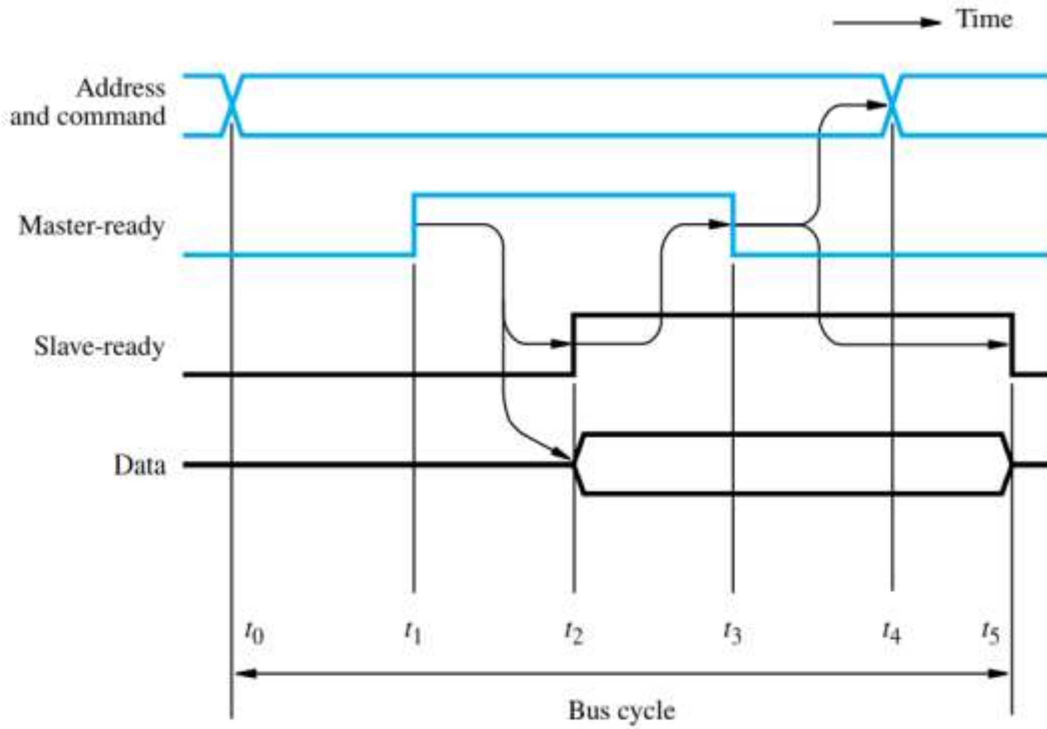


Figure 7.6 Handshake control of data transfer during an input operation.

Question

The asynchronous bus protocol as shown in figure uses a full-handshake, in which the master maintains an asserted signal on master-ready until it receives slave-ready, the slave keeps slave-ready asserted until master-ready becomes inactive, and so on. Consider an alternative protocol in which each of these signals is a pulse of a fixed width of 4 ns. Devices take action only on the rising edge of the pulse. Using the same parameters as given below:

Bus driver delay 2ns

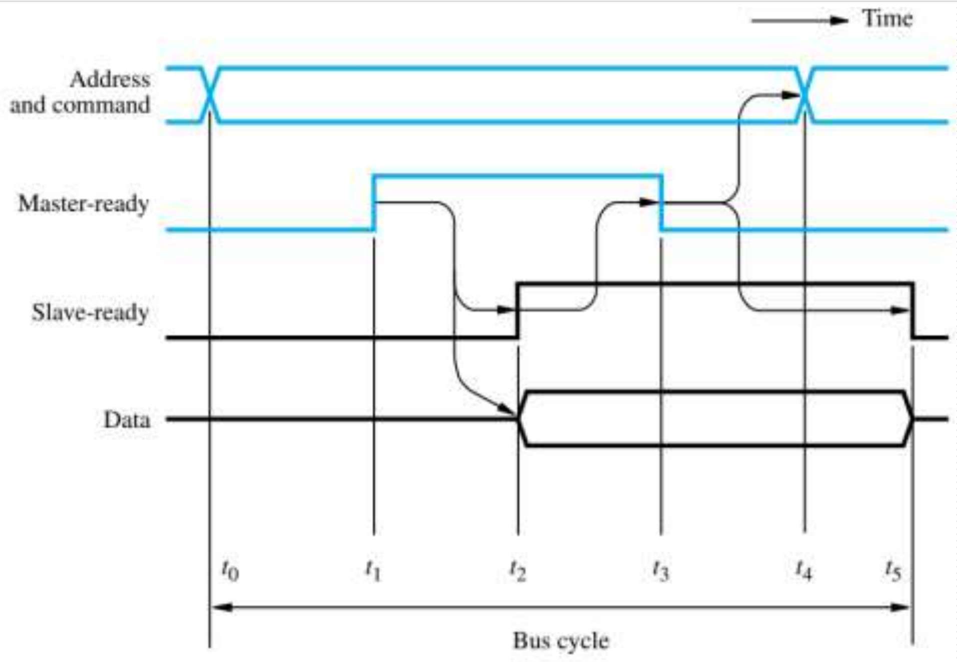
Propagation delay on the bus 5 to 10ns

Address decoder delay 6ns

Time to fetch the requested data 0 to 25ns

Setup time 1.5ns,

What is the minimum and maximum time to complete one bus transfer?



Wrong answer to cross check

Min Time

2+4+5+0=11 assuming no delay

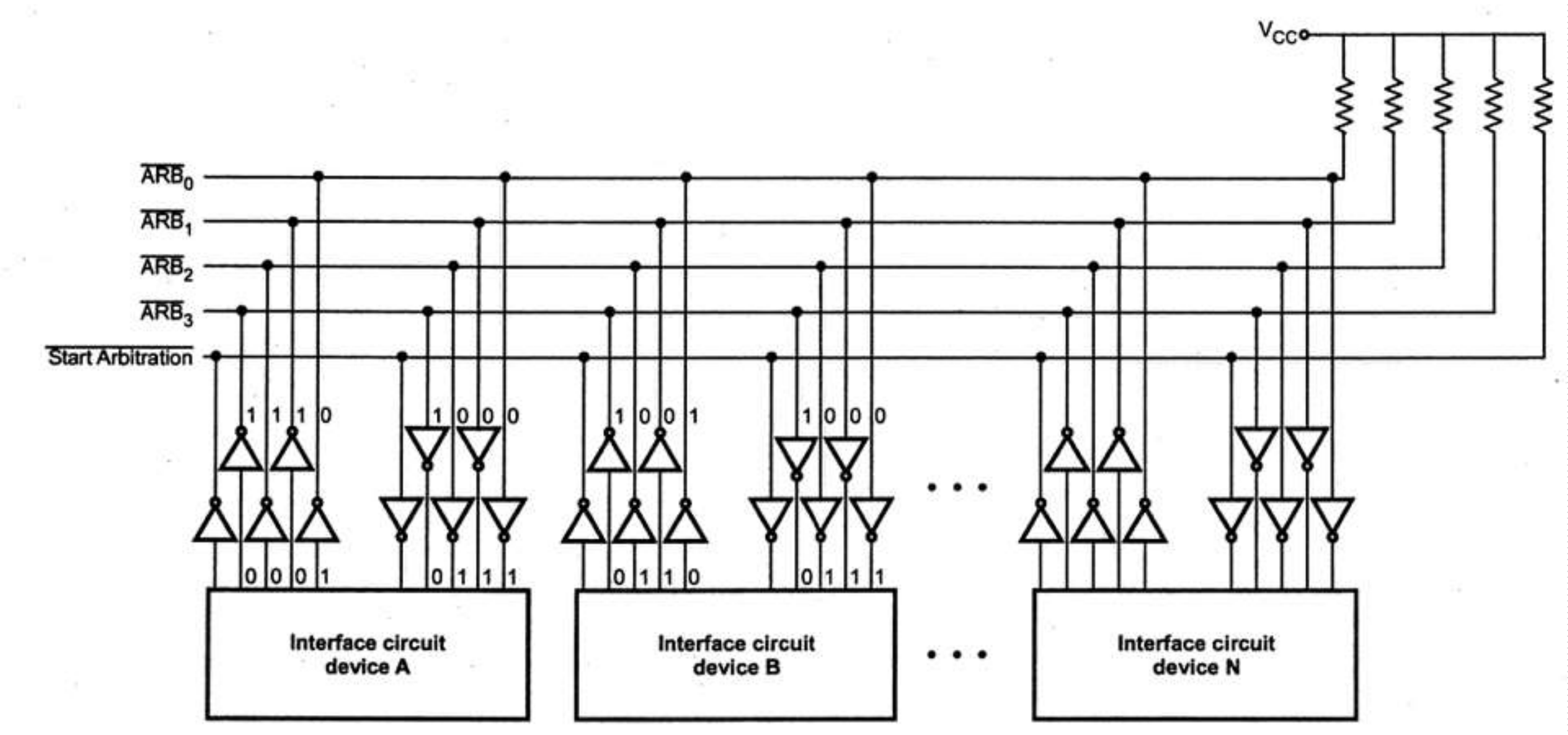
Max Time

2+4+5+25=36 (25 because this would be the longest data fetch would take for this problen

Distributed Arbitration

- More than one device can place their 4-bit ID number to indicate that they need to control of bus.
- If one device puts 1 on the bus line and another device puts 0 on the same bus line, the bus line status will be 0.
- Device reads the status of all lines through inverters buffers so device reads bus status 0 as logic 1.
- The device having highest ID number has highest priority.

Distributed Arbitration



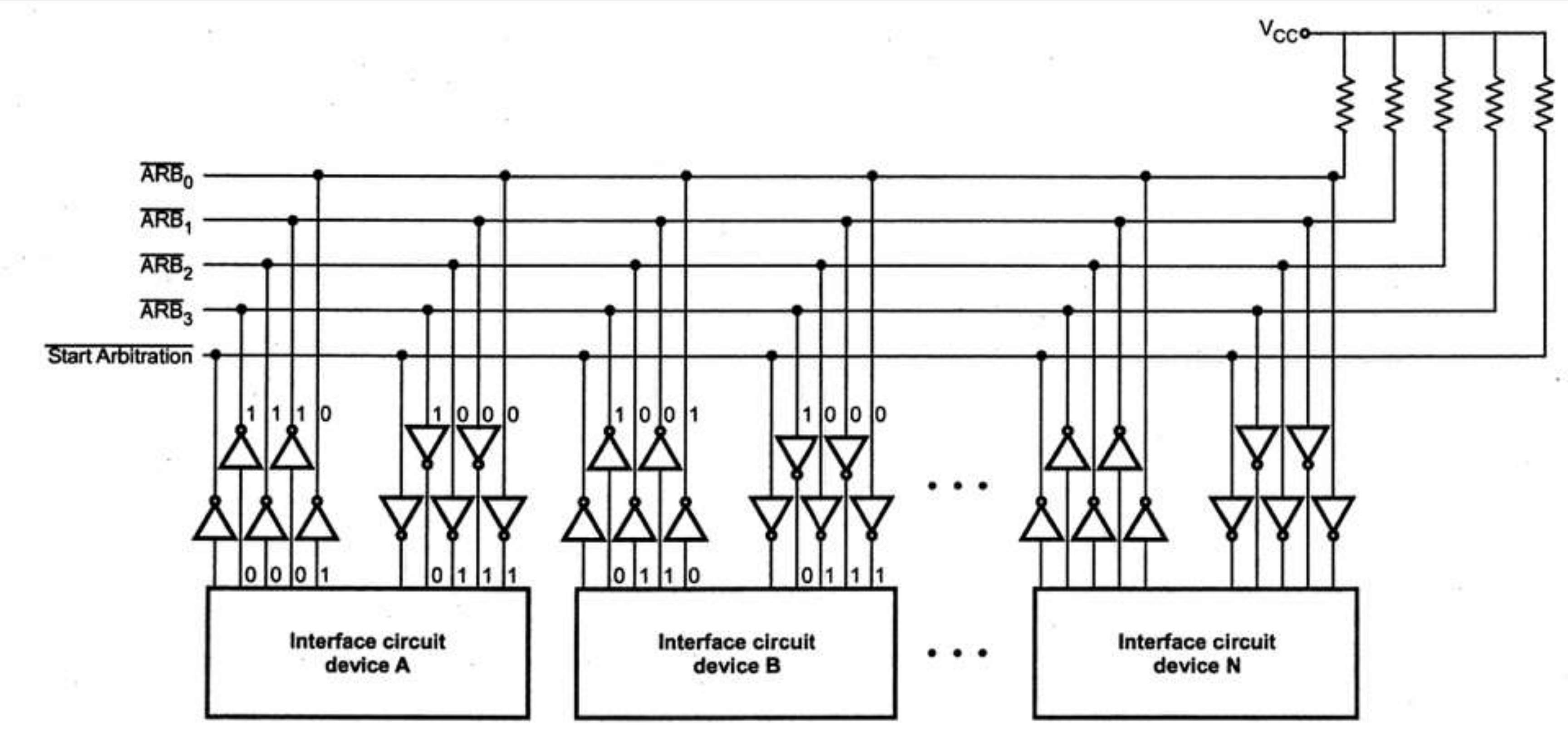
Distributed Arbitration

- Each device compares the code formed on the arbitration line to its own ID, starting from the most significant bit.
- If it finds the difference at any bit position, it disables its drives at that bit position and for all lower-order bits.
- It does so by placing a 0 at the input of their drive.

Distributed Arbitration

- In our example, device detects a different on line ARB2 and hence it disables its drives on line ARB2, ARB1 and ARB0.
- This causes the code on the arbitration lines to change to 0110. This means that device B has won the race.
- The decentralized arbitration offers high reliability because operation of the bus is not dependent on any single device.

Distributed Arbitration


V_{CC}
$\overline{ARB}_0$
$\overline{ARB}_1$
$\overline{ARB}_2$
$\overline{ARB}_3$
$\overline{\text{Start Arbitration}}$
1 1 1 0
0 0 0 1
1 0 0 0
0 1 1 1
1 0 0 1
0 1 1 0
1 0 0 0
0 1 1 1
Interface circuit device A
Interface circuit device B
Interface circuit device N
• • •
• • •