SECR2033
Computer Organization and Architecture

Lecture slides prepared by “Computer Organization and Architecture”, 9/e, by William Stallings, 2013.

Module 5
Central Processing Unit (CPU)

Objectives:

- To learn the components common to every CPU.
- Be able to explain how each component in CPU contributes to instruction cycle.
- To understand the concept of pipelining in CPU execution.
- Be able to understand micro-operations basis for the design and implementation of the control unit.

Module 5
Central Processing Unit (CPU)

5.1 Processor Organization
5.2 Register Organization
5.3 Instruction Cycles

5.4 Instruction Pipelining

5.5 Control Unit Operation
5.6 Microprogrammed Control

5.7 Summary

Module 5a
Central Processing Unit (CPU)

5.1 Processor Organization

- Overview

5.2 Register Organization

5.3 Instruction Cycles

5.4 Instruction Pipelining

5.5 Control Unit Operation

5.6 Microprogrammed Control

5.7 Summary

5 Overview

- To understand the organization of the processor, let us consider the requirements placed on the processor, the things that it must do:
  - *Fetch instruction*: reads an instruction from memory (register, cache, main memory).
  - *Interpret instruction*: The instruction is decoded to determine what action is required.
  - *Fetch data*: The execution of an instruction may require reading data from memory or an I/O module.

- *Process data*: The execution of an instruction may require performing some arithmetic or logical operation on data.
- *Write data*: The results of an execution may require writing data to memory or an I/O module.

- To do these things, the processor needs a small internal memory to store some data and instruction temporarily while an instruction is being executed.

Minimal internal memory, consisting of a set of storage locations, called registers.

ALU does the actual computation or processing of data

Control unit controls the movement of data and instructions into / out of the processor and controls the operation of the ALU

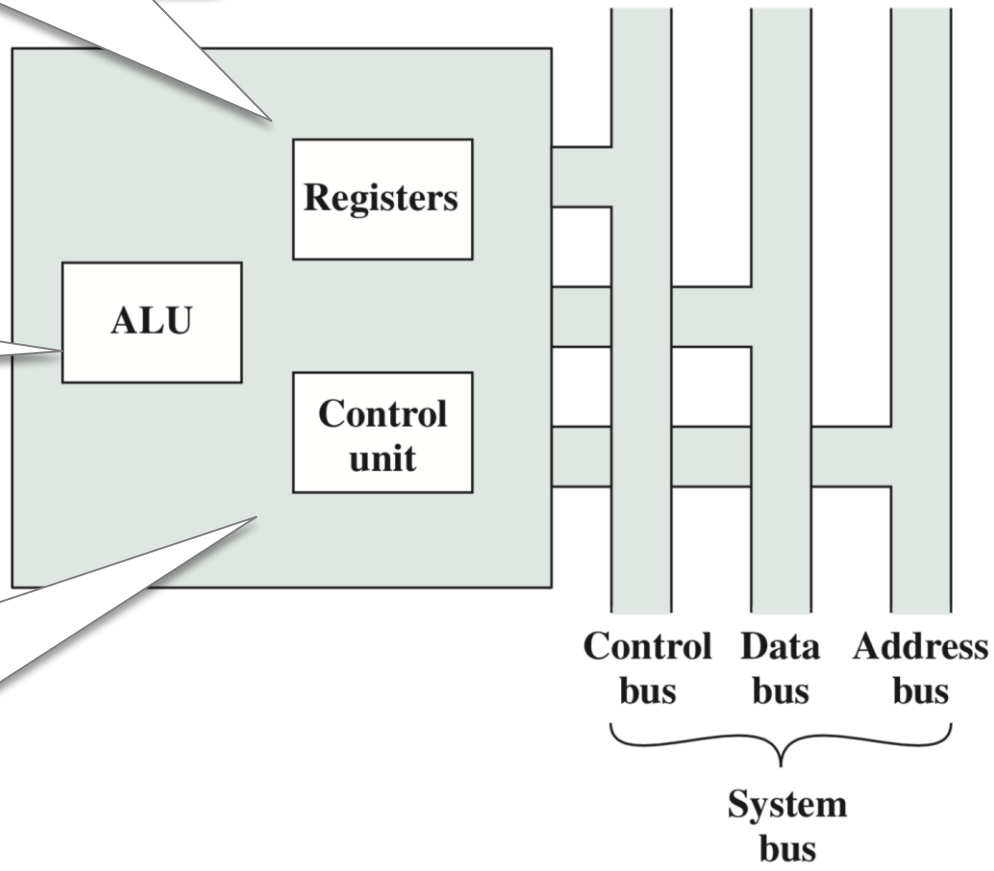


Figure: The CPU with the System Bus.

- The computer's CPU *fetches*, *decodes*, and *executes* program instructions.

- The two principal parts of the CPU:
  - data path
  - control unit

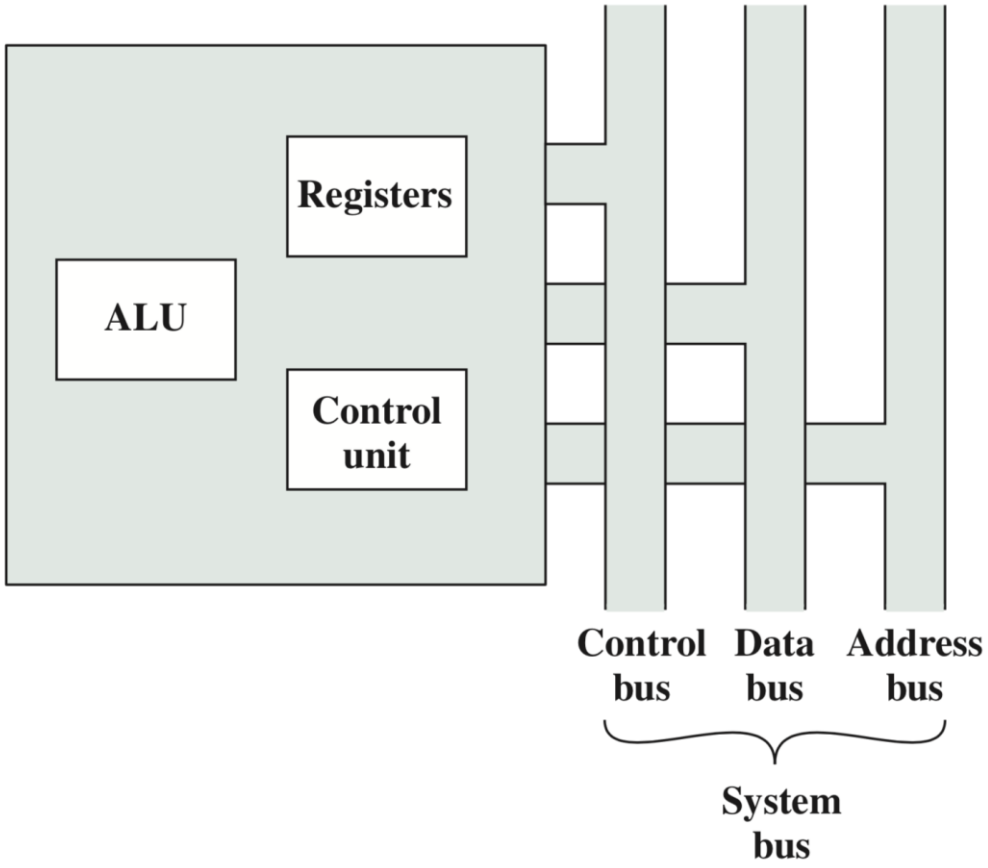


Figure: The CPU with the System Bus.

1) The **data path** consists of an *ALU* and storage units (*registers*) that are interconnected by a *data bus* that is also connected to *main memory*.

2) The **control unit** provides signals to tell the **data path**, *memory*, and *I/O* devices what to do according the instructions of the program.

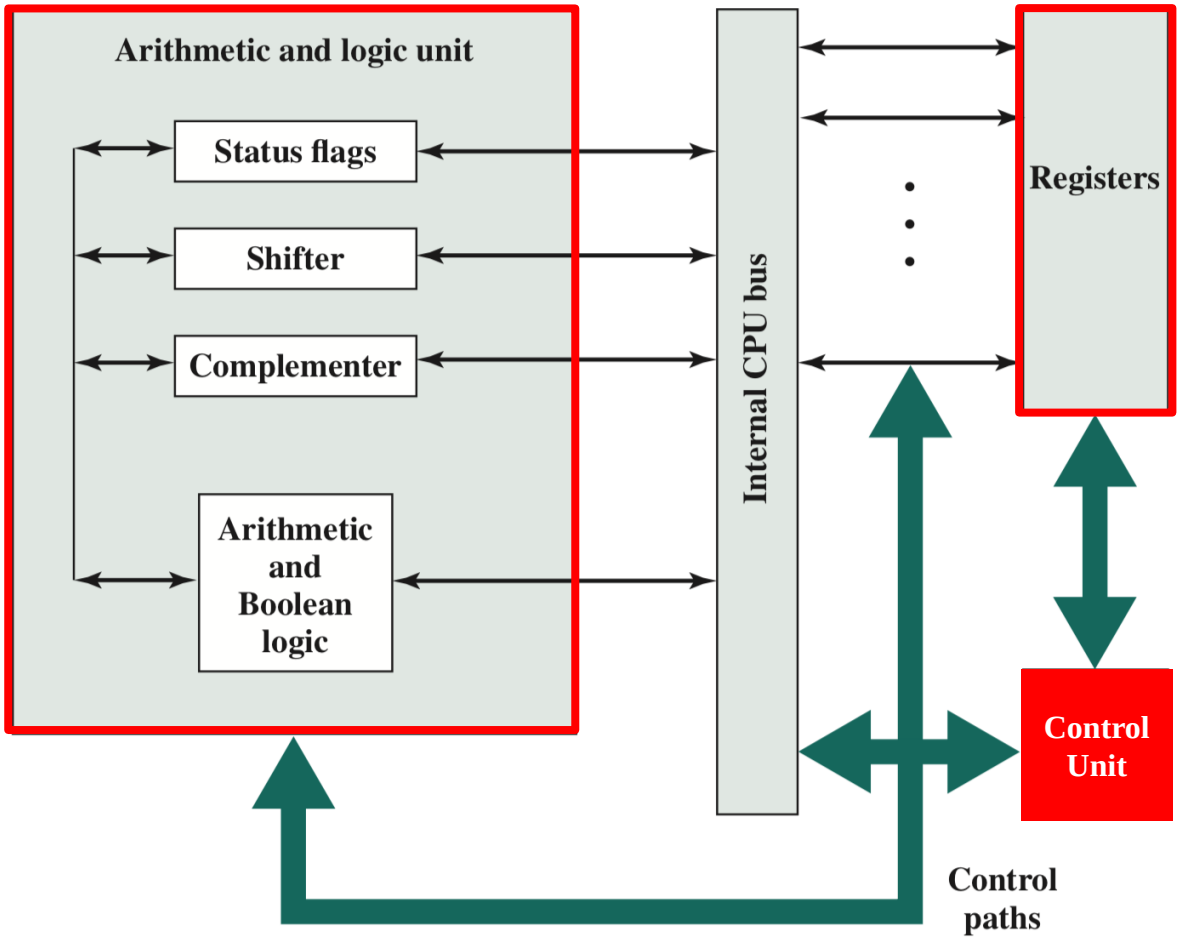


Figure: Internal Structure of the CPU.

Internal CPU Bus

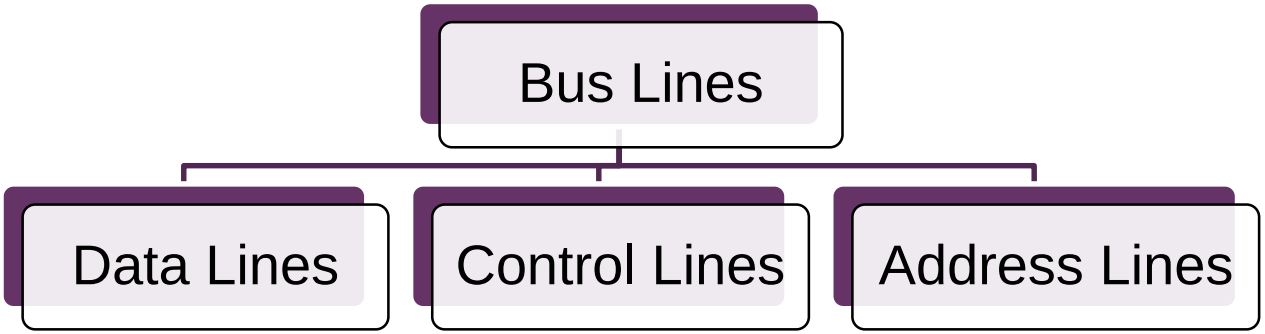
- The *registers* are interconnected, and connected with *main memory* through a common data bus.
- Each device on the bus is identified by a unique number that is set on the control lines whenever that device is required to carry out an operation.

This permits data transfer between these devices without use of the main data bus.

- Separate connections are also provided between:
  - the accumulator & the memory buffer register, and
  - the ALU & the accumulator & memory buffer register.

External Bus

- A bus is a set of wires that simultaneously convey a single bit along each line.



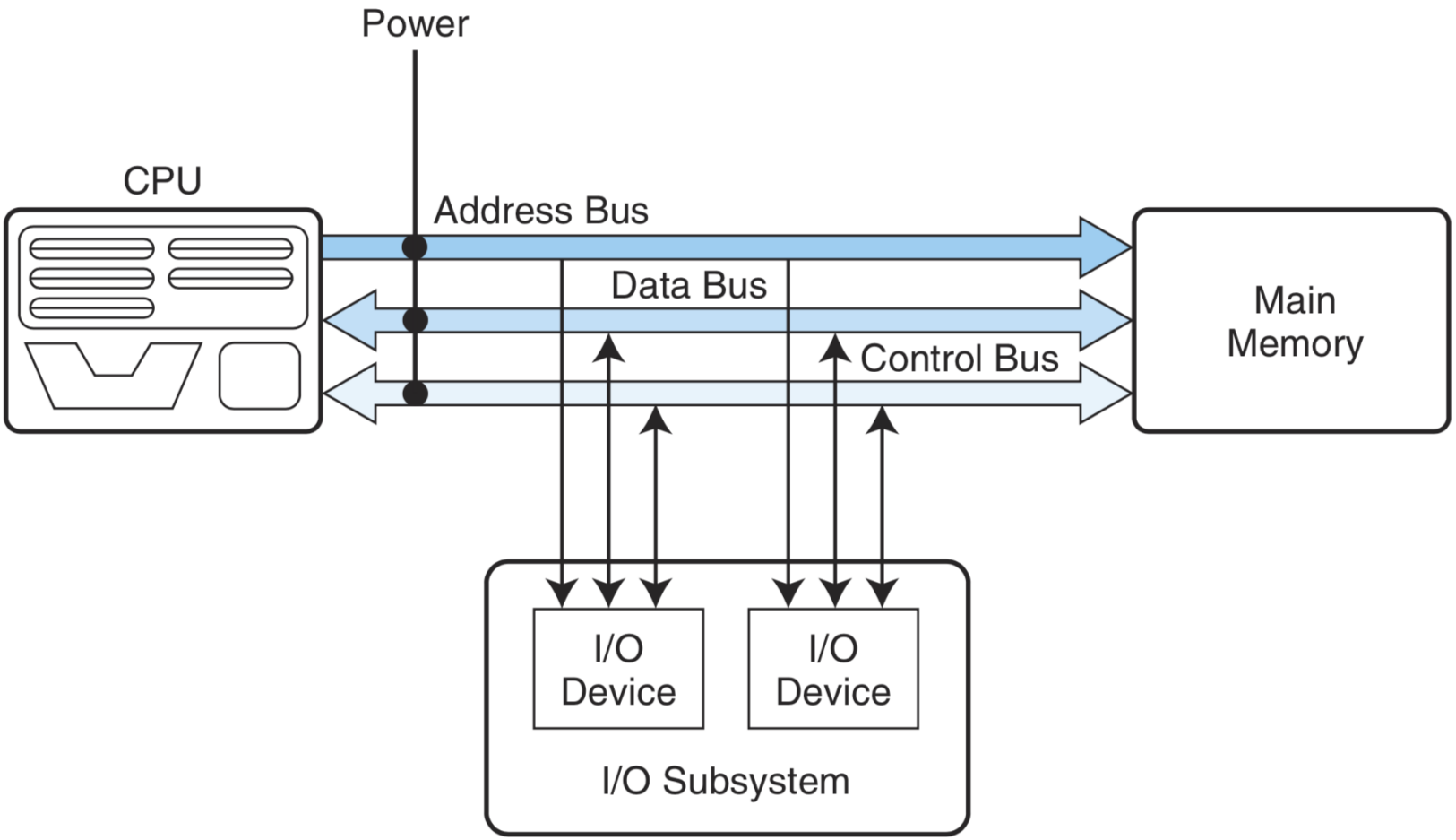


Figure: The Components of a Typical Bus

1) Data lines

- convey bits from one device to another.
- The number of lines (32, 64,128) gives the width of the data bus.
- **Example**: width = 32 bit, instruction length = 64 bits → this means the processor must access the memory modules twice during each instruction cycle.

2) Control lines

- determine the direction of data flow, and when each device can access the bus.
- Uses control signals, timing signals, command signals → memory write, memory read, bus request and bus grant.

3) Address lines

- determine the location of the source or destination of the data.
- **Example**: want to read data in memory, put address on this bus, go to memory, get the content (*i.e.* the data) and put it on to data bus.

Module 5a
Central Processing Unit (CPU)

5.1 Processor Organization

5.2 Register Organization

- Overview
- User-Visible Registers
- Control & Status Registers

5.3 Instruction Cycles

5.4 Instruction Pipelining

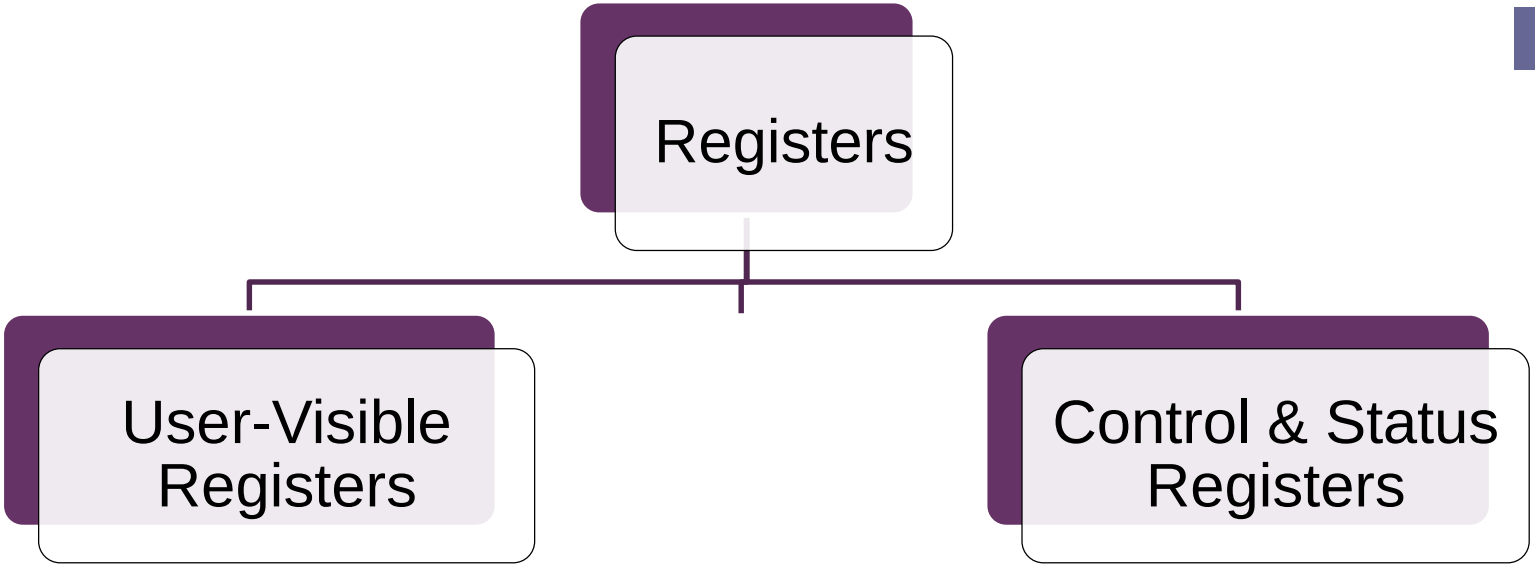
5.5 Control Unit Operation

5.6 Microprogrammed Control

5.7 Summary

5 Overview

- Registers form the highest level of the memory hierarchy.
  - Small set of high speed storage locations.
  - Temporary storage for data and control information.
- Registers hold data that can be readily accessed by the CPU.
- They can be implemented using D flip-flops.
  - A 32-bit register requires 32 D flip-flops.



(*General Purpose*)

May be referenced by assembly-level instructions and are thus "visible" to the user.

(*Special Purpose*)

Used to control the operation of the CPU. Most are not visible to the user.

5

(a) User-Visible Registers

- This register will be referenced by means of the machine language that the processor executes.

- Categories:
  - General purpose
  - Data
  - Address
  - Condition codes

5

General purpose registers:

- Can be assigned to a variety of functions.
- Defined to the operations within the instructions.
- Can be used for addressing functions.
- **Examples**: accumulator, base, count, data.

Data registers:

- Hold data and cannot be used in the calculation of an operand address.
- **Example**: accumulator.

Address registers:

- Hold address information.
- **Examples**: general purpose address registers, segment pointers, stack pointers, index registers.

Condition codes or Flags:

- Bits set by the processor hardware as a result of operations.
- Can be accessed by a program but not changed directly.
- **Examples**: *Sign Flag* (SF), *Zero Flag* (ZF), *Overflow Flag* (OF).
- Bit values are used as the basis for conditional jump instructions.

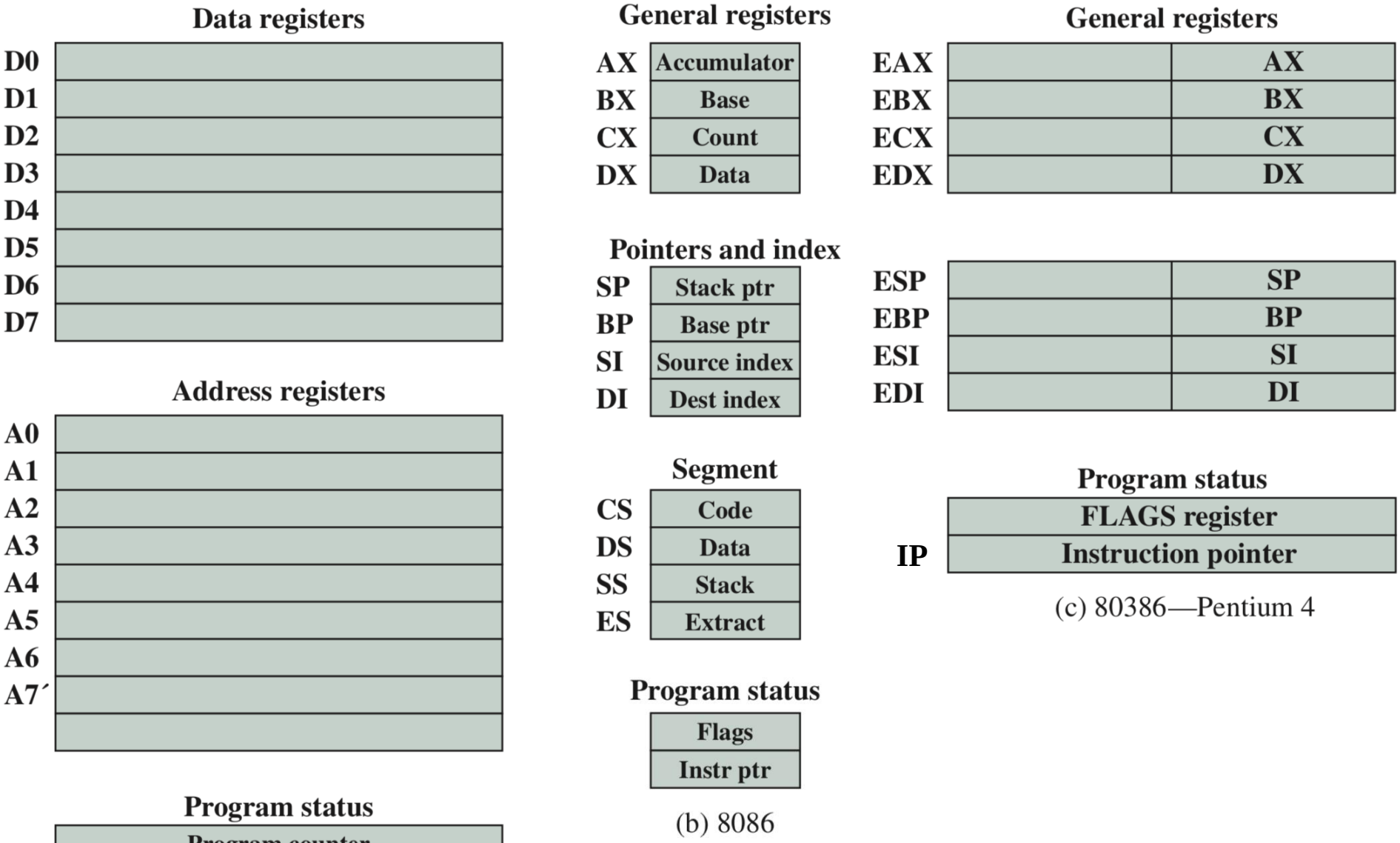
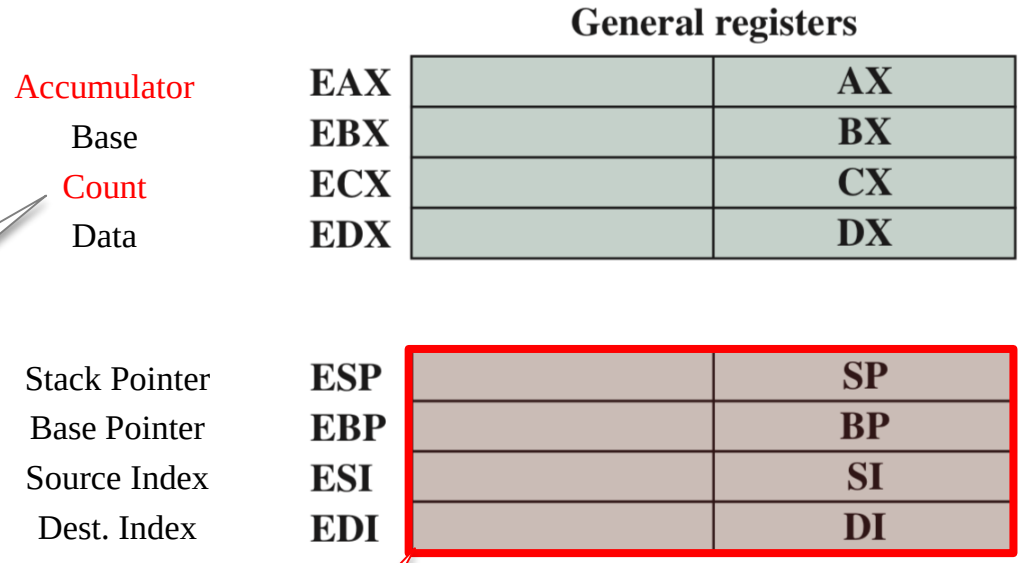


Figure: Example Microprocessor Register Organizations.

EAX *automatically used by* `MUL` *and* `DIV` *instructions*

ECX *automatically used by processor as a* `LOOP` *counter.*

Avoid usage for generic calculation !



- Address registers should be wide enough to hold the longest address!
- Data registers should be wide enough to hold most data types.
- Would not want to use 64-bit registers if the vast majority of data operations used 16 and 32-bit operands.
- Related to width of memory data bus.

- **Solution**: Concatenate registers together to store longer formats (DX:AX)

Example:

```
mul   ax
```

Assuming the following *DumpRegs*. Update the related register once the instruction executed:

```
EAX=770C0400  EBX=7FFD7098  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0012FF94  ESP=0012FF8C
EIP=00401025  EFL=00000A82  CF=0  SF=1  ZF=0  OF=1
```

Solutions:

```
mul   ax
```

→ ax = 0400h

→ ax * ax = 0400 * 0400 = 0010 0000h

DX : AX

```
EAX=770C0000  EBX=7FFD7098  ECX=00000000  EDX=00400010
ESI=00000000  EDI=00000000  EBP=0012FF94  ESP=0012FF8C
EIP=00401025  EFL=00000A82  CF=0  SF=1  ZF=0  OF=1
```

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX

Example: `div bx`

Assuming the following *DumpRegs*. Update the related register once the instruction executed:

```
EAX=770C0000  EBX=7FFD0020  ECX=00000000  EDX=00400010
ESI=00000000  EDI=00000000  EBP=0012FF94  ESP=0012FF8C
EIP=00401025  EFL=00000A82  CF=0  SF=1  ZF=0  OF=1
```

Solutions:

```
div  bx
```

→ dx : ax = 00100000h, bx = 0020h

→ [dx : ax] / bx = 100000 / 0020 = 0000 8000h

DX : AX

```
EAX=770C8000  EBX=7FFD0020  ECX=00000000  EDX=00400000
ESI=00000000  EDI=00000000  EBP=0012FF94  ESP=0012FF8C
EIP=00401025  EFL=00000A82  CF=0  SF=1  ZF=0  OF=1
```

Dividend	Divisor	Quotient	Remainder
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX

(b) Control & Status Registers

- There are a variety of processor registers that are employed to control the operation of the processor.
- Most of these, on most machines, are not visible to the user.
- Some of them may be visible to machine instructions executed in a control or operating system mode.

Different machines will have different register organizations and use different terminology.

PC = The address of the next instruction to be executed

Table: Four essential registers for instruction execution.

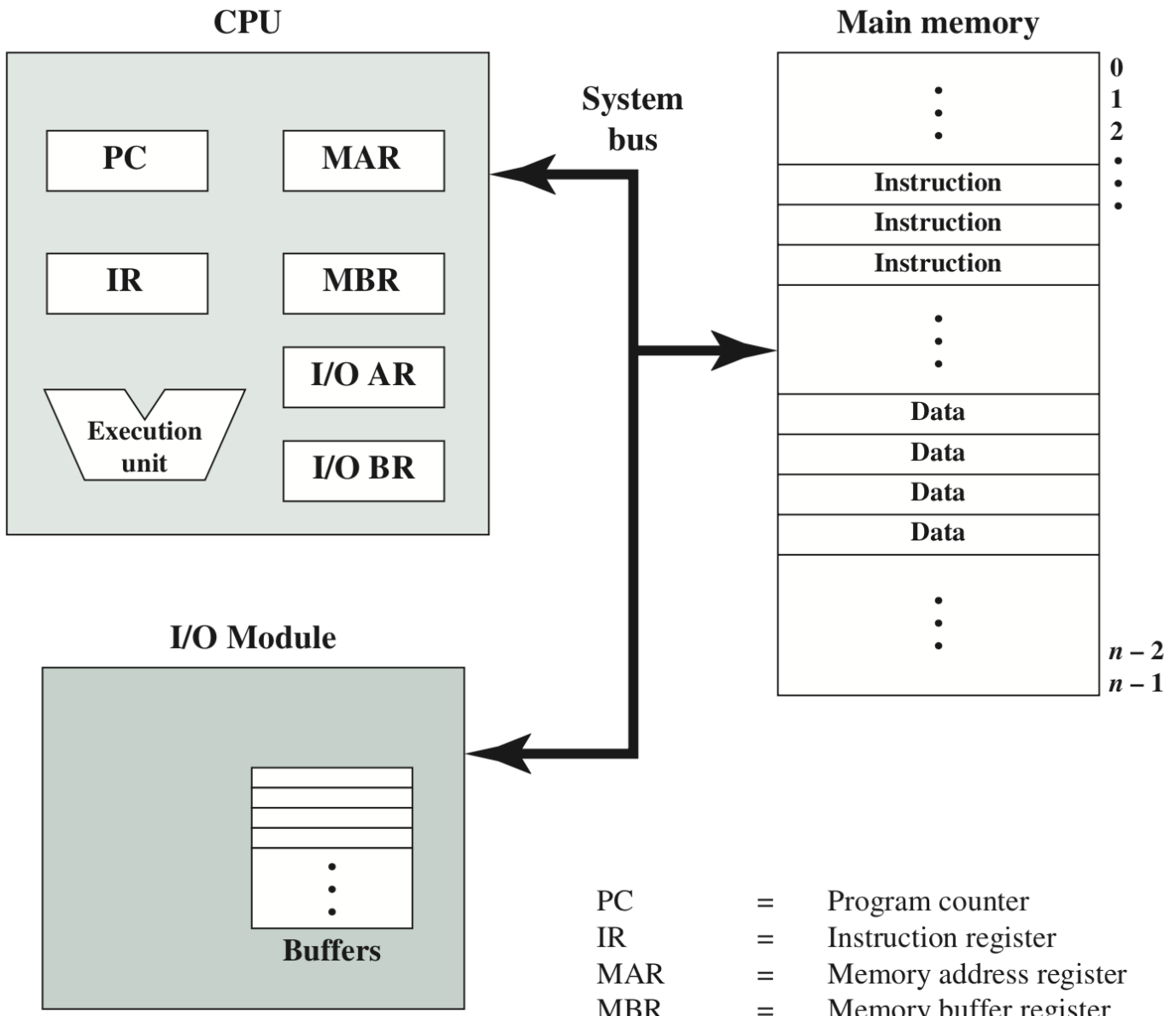
Registers	Function
Program Counter (PC)	The address of an instruction to be fetched.
Instruction Register (IR)	The instruction most recently fetched.
Memory Address Register (MAR)	The address of a location in memory.
Memory Buffer Register (MBR)	A word of data to be written to memory or the word most recently read.

5

- The four registers just mentioned are used for the movement of data between the processor and memory.
- Not all processors have internal registers designated as MAR and MBR → need some equivalent buffering mechanism to store the bits transferred / read bits to / from the data bus:

  - The ALU may have direct access to the MBR and user-visible registers.
  - There may be additional buffering registers at the boundary to the ALU serves as input and output registers for the ALU and exchange data with the MBR and user-visible registers.

Figure: Computer components: The CPU exchanges data with memory.



PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

Instruction Pointer

- The *Extended Instruction Pointer* (EIP) register holds the address of the next instruction to be executed.
- The EIP register corresponds to the *Program Counter* (PC) register in other architectures.
- EIP can be manipulated for certain instructions (*e.g.* call, `jmp`, `ret`) to branch to a new location.

Program Status

- Many processor designs include a register or set of registers, often known as the *Program Status Word* (PSW), that contain status information and condition codes.
- The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations.

Status flags that reflect the outcome of arithmetic (contents of the destination operand)

- Common field or flags:
  - *Zero flag* (ZF): set when destination equals zero.
  - *Sign flag* (SF): set when destination is negative.
  - *Carry flag* (CF): set when unsigned value is out of range.
  - *Overflow flag* (OF): set when signed value is out of range.
  - *Auxiliary flag* (AF): set when there is carry from lower nibble to higher nibble in the lower byte.
  - *Parity flag* (PF): set when there are odd or even number of 1's in the lower byte.

Example 1: Flags

Assuming that the system uses **even parity.**

→ 06h + 54h

```
     0000 0110 (06h)
+    0101 0100 (54h)
    ----------------
     0101 1010 (5Ah)
```

Number of 1*s* = 4 → *even* ;
so PF = 1

ZF : 0

SF : 0

CF : 0

OF : 0

AF : 0

PF : 1

Example 2: Flags

Assuming that the system uses **odd parity.**

→ 2Fh + DEh

```
    0010 1111 (2Fh)
+   1101 1110 (DEh)
  ----------------
 1 0000 1101
```

There is a carry from lower nibble to higher nibble in the lower byte → so AF = 1

There is a carry out → so CF = 1

Number of 1s = 3 → odd ; so PF = 1

ZF : 0

SF : 0

CF : 1

OF : 0

AF : 1

PF : 1

Example 3: Flags

Assuming that the system uses **even parity** for:

a) A word sized number.

b) A doubleword sized number.

→ 5433h + 7098h

ZF :
SF :
CF :
OF :
AF :
PF :

ZF :
SF :
CF :
OF :
AF :
PF :

Solutions (a):

If the carry into the *sign bit* is different from the carry out of the sign bit, overflow (and thus an error) has occurred.

ZF : 0
SF : 1
CF : 0
OF : 1
AF : 0
PF : 1

a) A word sized number.

→ 5433h + 7098h

```
    0101 0100 0011 0011
+   0111 0000 1001 1000
  ---------------------
    1100 0100 1100 1011
```

- *Sign flag* (SF): set when destination is negative.

Solution (b):

b) A doubleword sized number.

→ 00005433h + 00007098h

```
  0000 0000 0000 0000 0101 0100 0011 0011
+ 0000 0000 0000 0000 0111 0000 1001 1000
  ---------------------------------------
  0000 0000 0000 0000 1100 0100 1100 1011
```

ZF : 0

SF : 0

CF : 0

OF : 0

AF : 0

PF : 1

The *DumpRegs* if use `word` sized numbers:

```
EAX=770CC4CB  EBX=7FFD7098  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0012FF94  ESP=0012FF8C
EIP=00401025  EFL=00000A82  CF=0  SF=1  ZF=0  OF=1
```

The *DumpRegs* if use `dword` sized numbers:

```
EAX=0000C4CB  EBX=00007098  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0012FF94  ESP=0012FF8C
EIP=00401022  EFL=00000202  CF=0  SF=0  ZF=0  OF=0
```

Status flags that reflect the outcome of arithmetic (contents of the destination operand)

- Common field or flags:
  - *Zero flag* (ZF): set when destination equals zero.
  - *Sign flag* (SF): set when destination is negative.
  - *Carry flag* (CF): set when unsigned value is out of range.
  - *Overflow flag* (OF): set when signed value is out of range.
  - *Auxiliary flag* (AF): set when there is carry from lower nibble to higher nibble in the lower byte.
  - *Parity flag* (PF): set when there are odd or even number of 1's in the lower byte.

Remember how to do subtraction in Module2 ?

$$M - S = M + (-S)$$

$(-S) \rightarrow$ 2's Complement (negation)

Example 4 : Flags

Assuming that the system uses **even parity** for:

a) 3Ch – 28h

b) 5Ah – 7Fh

2's for 28h

```
a)     0011 1100
     + 1101 1000
       -----------
     1 0001 0100  (14h)
```

ZF :	0
SF :	0
CF :	1
OF :	0
AF :	1
PF :	1

2's for 7Fh

```
b)     0101 1010
     + 1000 0001
       -----------
       1101 1011  (DBh)
```

ZF :	0
SF :	1
CF :	0
OF :	0
AF :	0
PF :	1

Design Issues of the Control & Status Register Organization

5

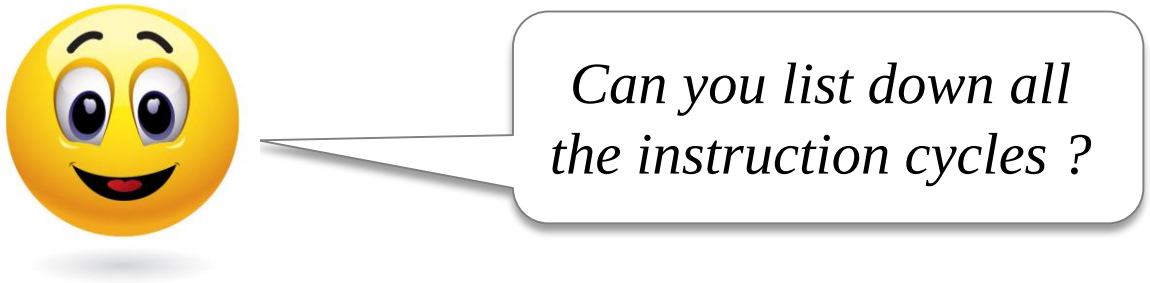
1) **Operating system support.**
   - Certain types of control information are of specific utility to the operating system.
   - The register organization need to some extent to be tailored to a particular operating system .

2) **The allocation of control information between registers and memory.**
   - The designer must decide how much control information should be in registers and how much in memory.
   - The usual trade-off of cost versus speed arises.

Module 5a
Central Processing Unit (CPU)

5 Overview

- The processing required for a single instruction is called an **instruction cycle**.



5 Overview

- The processing required for a single instruction is called an **instruction cycle**.
- The instruction processing consists of **three (3) stages** :

❑ *Fetch cycle*	- Fetch the instruction; - Decode it; - Fetch operands, if required (indirect).
❑ *Execute cycle*	- Perform the operation; - Store results, if required.
❑ *Interrupt cycle*	- Recognize pending interrupts.

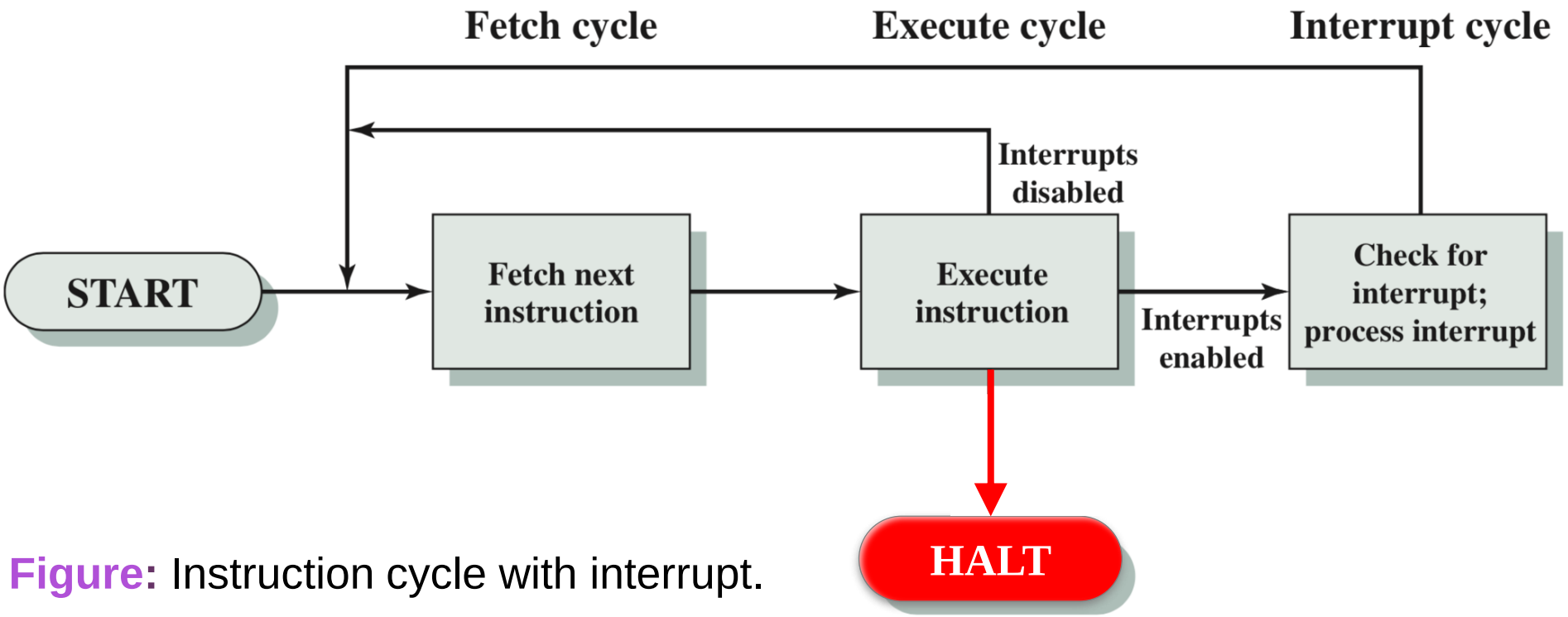


Figure: Instruction cycle with interrupt.

- Program execution halts only if :
  - the machine is turned off,
  - some sort of unrecoverable error occurs, or
  - a program instruction that halts the computer is encountered.

5 The Indirect Cycle

- We can think that the fetching of indirect addresses → another instruction stages.
- The main line of activity consists of alternating instruction fetch and instruction execution activities.
- Following execution, an interrupt may be processed before the next instruction fetch.

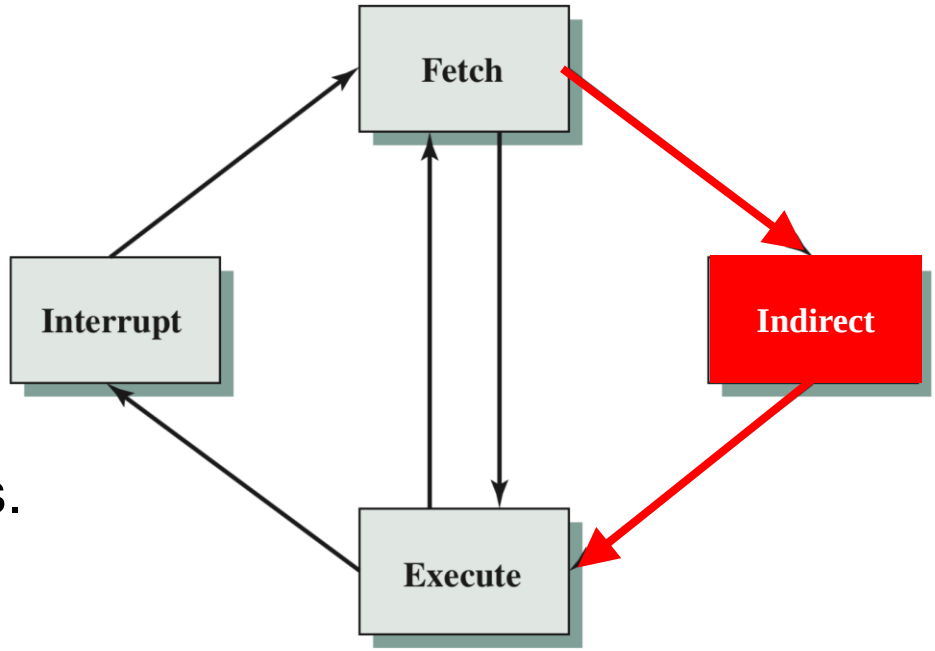


Figure: The instruction cycle.

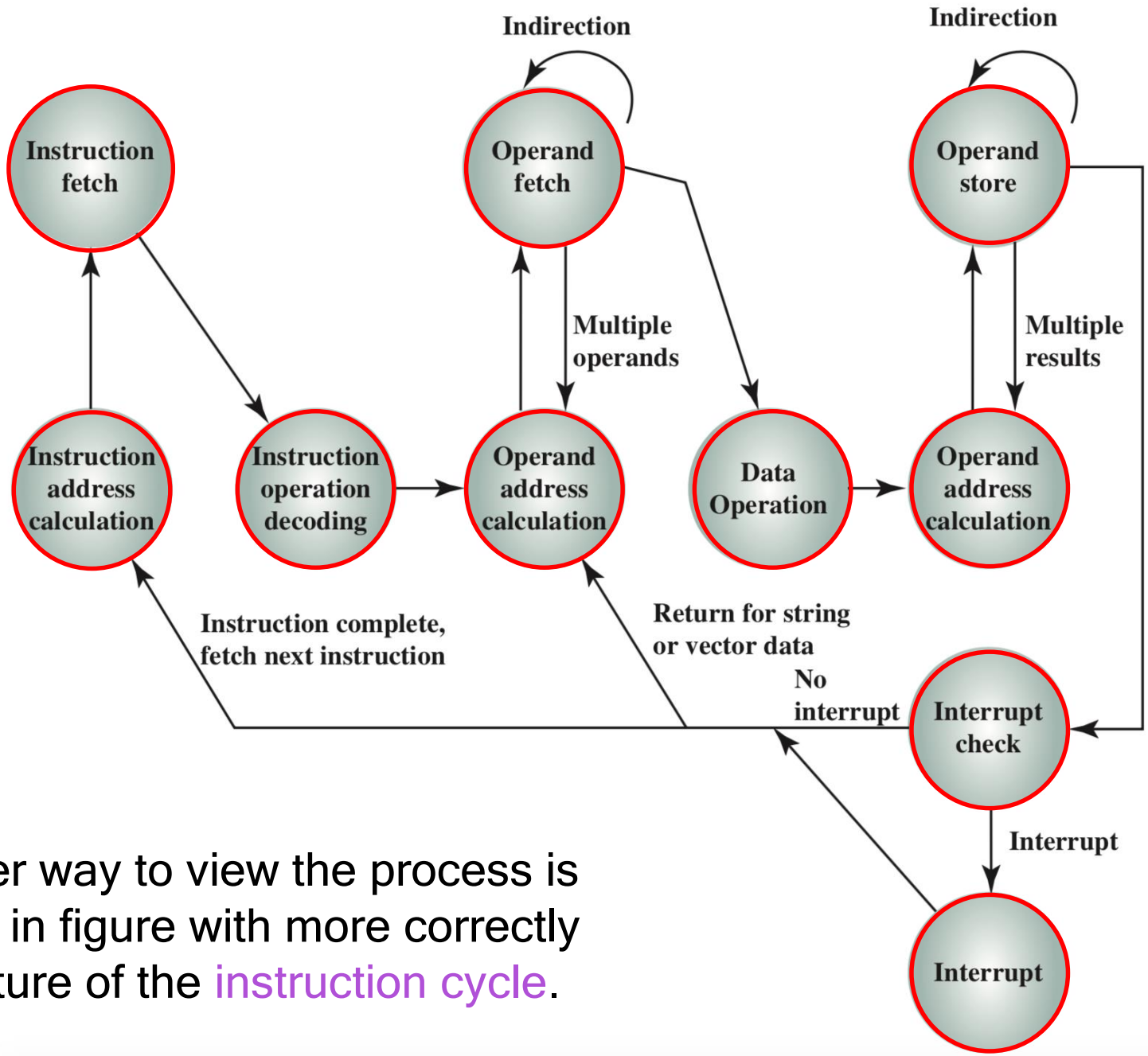


Figure: Instruction cycle state diagram.

- Another way to view the process is shown in figure with more correctly the nature of the instruction cycle.

5 Data Flow

- The exact sequence of events during an instruction cycle depends on the design of the processor.

Let us assume that a processor employs:

Memory Address Register (MAR)
Memory Buffer Register (MBR)
Program Counter (PC)
Instruction Register (IR).

- Each of the smaller cycles consists of a series of steps that involves the processor registers → micro-operations.

Detail in sub-topic: 5.5 and 5.6

- Conventional format using *Register Transfer Language* (RTL)
- **Example**: IP ← IP +1
  PC ← PC +1

(a) Fetch Cycle

- An instruction is read from memory.

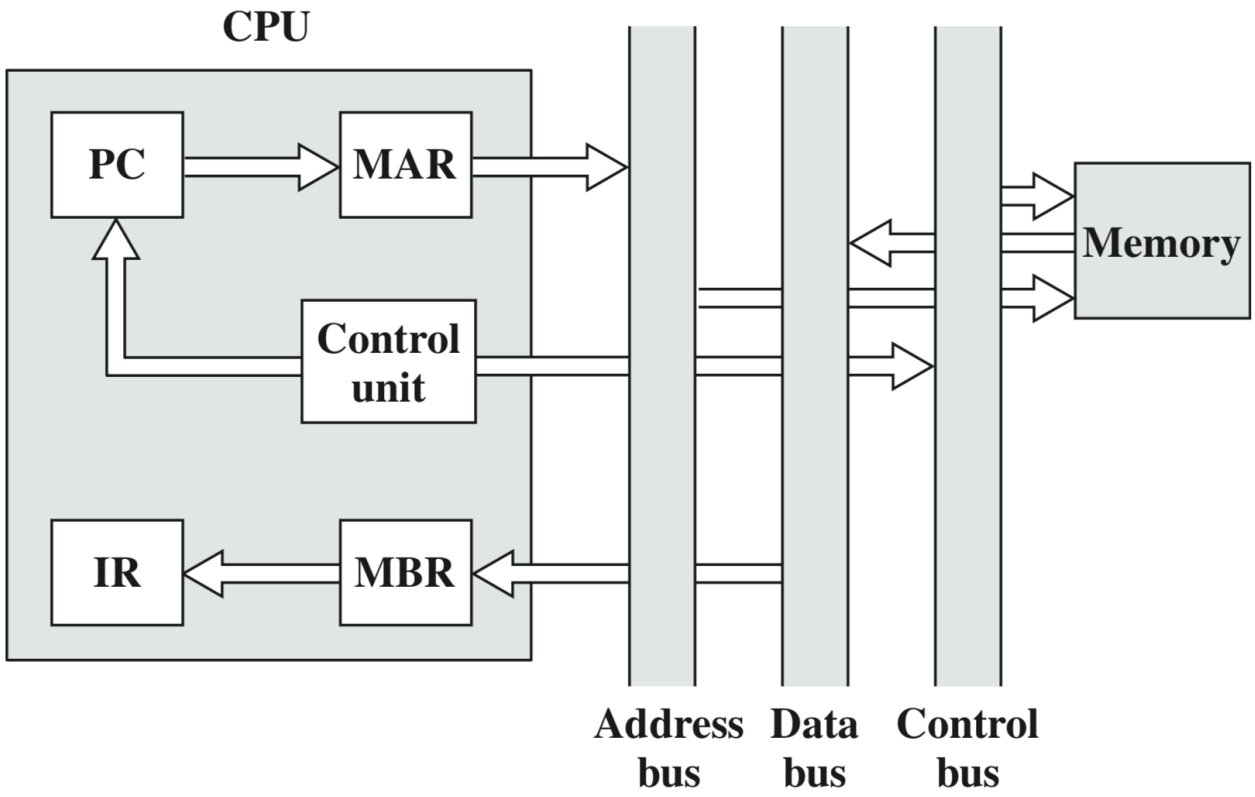


Figure: Data flow, fetch cycle.

- The *fetch cycle* can be written symbolically using RTL as:

t_1: MAR ← IP/PC

t_2: MBR ← [MAR];

IP/PC ← IP/PC + 1

t_3: IR ← MBR

t_1, t_2, t_3 are the timing sequence

CU (Control Unit)

MAR (Memory Address Register)
MBR (Memory Buffer Register)
PC (Program Counter)
IR (Instruction Register)

5

- The PC contains the address.
- Content of IP/PC is loaded to MAR.
- Then transferred to memory on the address bus.
- Next, CU will activate the read signal of the memory.

t_1: MAR ← IP/PC

t_2: MBR ← [MAR];
IP/PC ← IP/PC + 1

t_3: IR ← MBR

- Instruction is accessed from the memory and transfer to CPU on the data bus.
- This instruction address will be loaded into MBR.

- At the same time, IP/PC will be incremented (point to the next instruction).

- Instruction in MBR is loaded into IR.
- The opcode of the instruction will be decoded and translated so as to determine the *micro-operations* for the particular instruction.

MAR (Memory Address Register)
MBR (Memory Buffer Register)
IR (Instruction Register)

(a) Fetch Cycle: *Indirect Cycle*

- Once the *fetch cycle* is over, the control unit examines the contents of the IR to determine if it contains an operand specifier using indirect addressing.

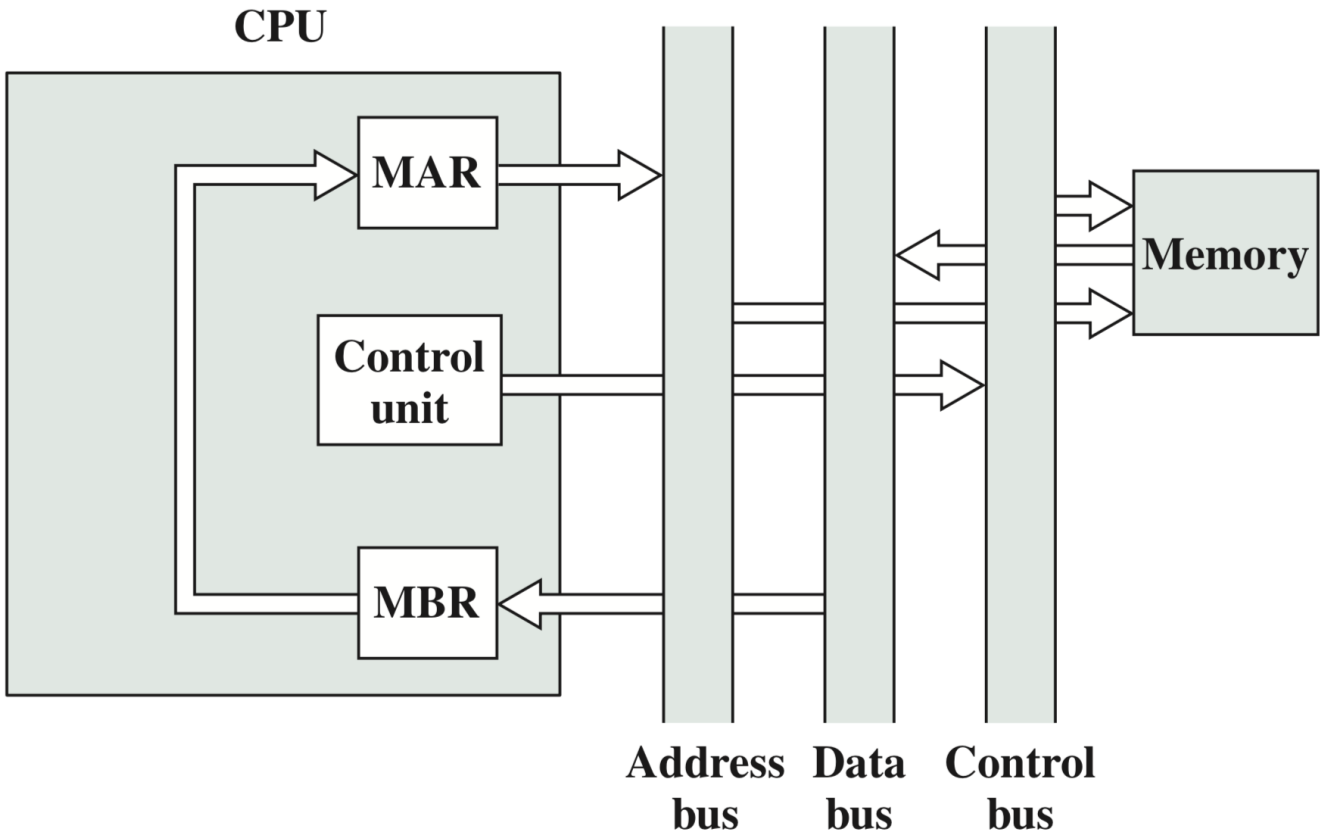


Figure: Data flow, indirect cycle.

- The *fetch cycle (Indirect addressing)* can be written symbolically using RTL as:

t_1: MAR ← IR[Operand/Address]

t_2: MBR ← Mem[MAR];

t_3: IR[operand] ← MBR

t_1, t_2 , t_3 are the timing sequence

CU (Control Unit)

MAR (Memory Address Register)
MBR (Memory Buffer Register)
IR (Instruction Register)

5

- The IR contains the indirect address that loaded to MAR.
- Then transferred to memory on the address bus.
- Next, CU will activate the read signal of the memory.

t_1: MAR ← IR[Operand/Address]

t_2: MBR ← Mem[MAR];

t_3: IR[operand] ← MBR

- Data (operand) is accessed from the memory and transfer to CPU on the data bus.
- This data will be loaded into MBR.

- Data in MBR is loaded into IR.
- The opcode of the instruction will be decoded and translated so as to determine the *micro-operations* for the particular instruction.

Example 5: Indirect cycle

`ADD AX,[2400]`

Involve with fetching data from memory before initiating the *execution cycle*.

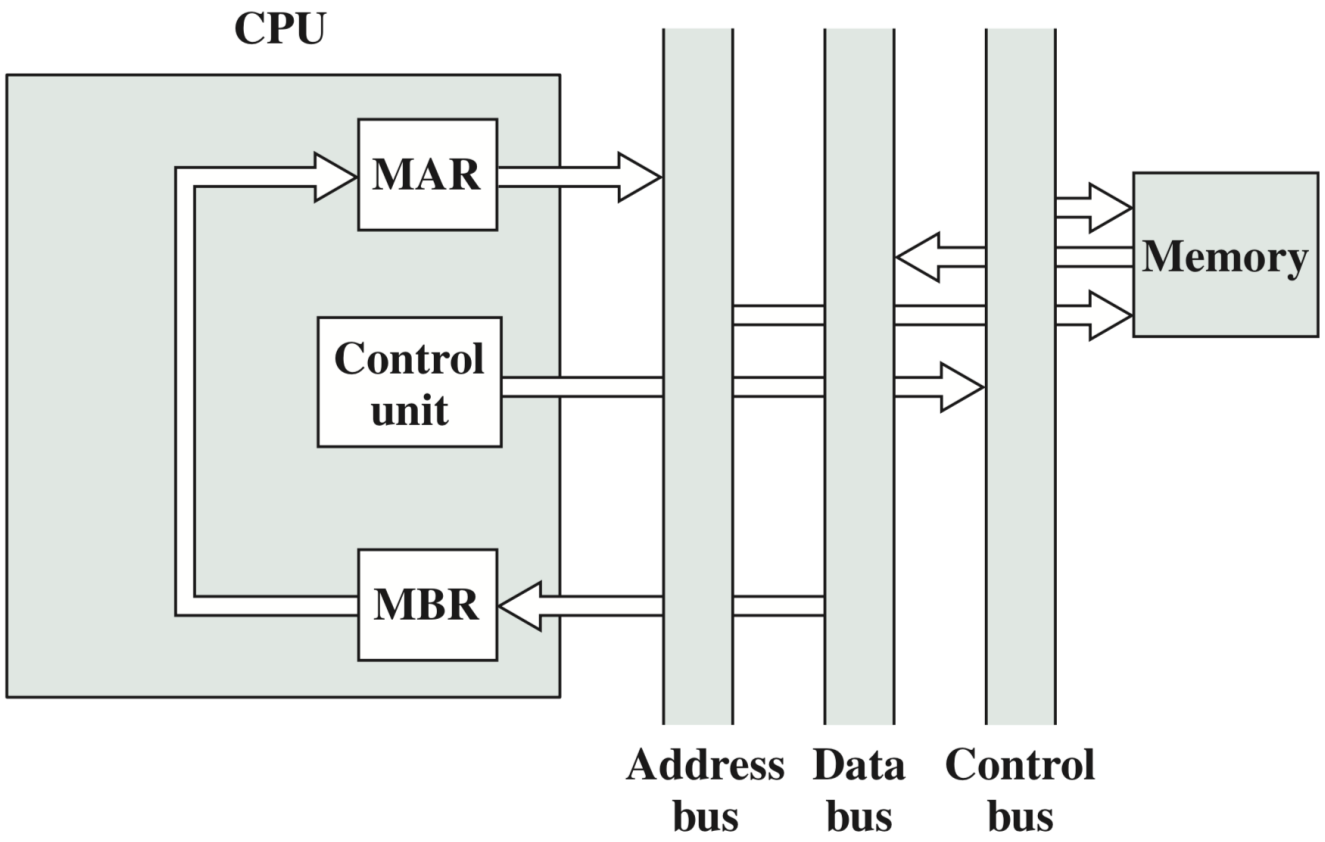
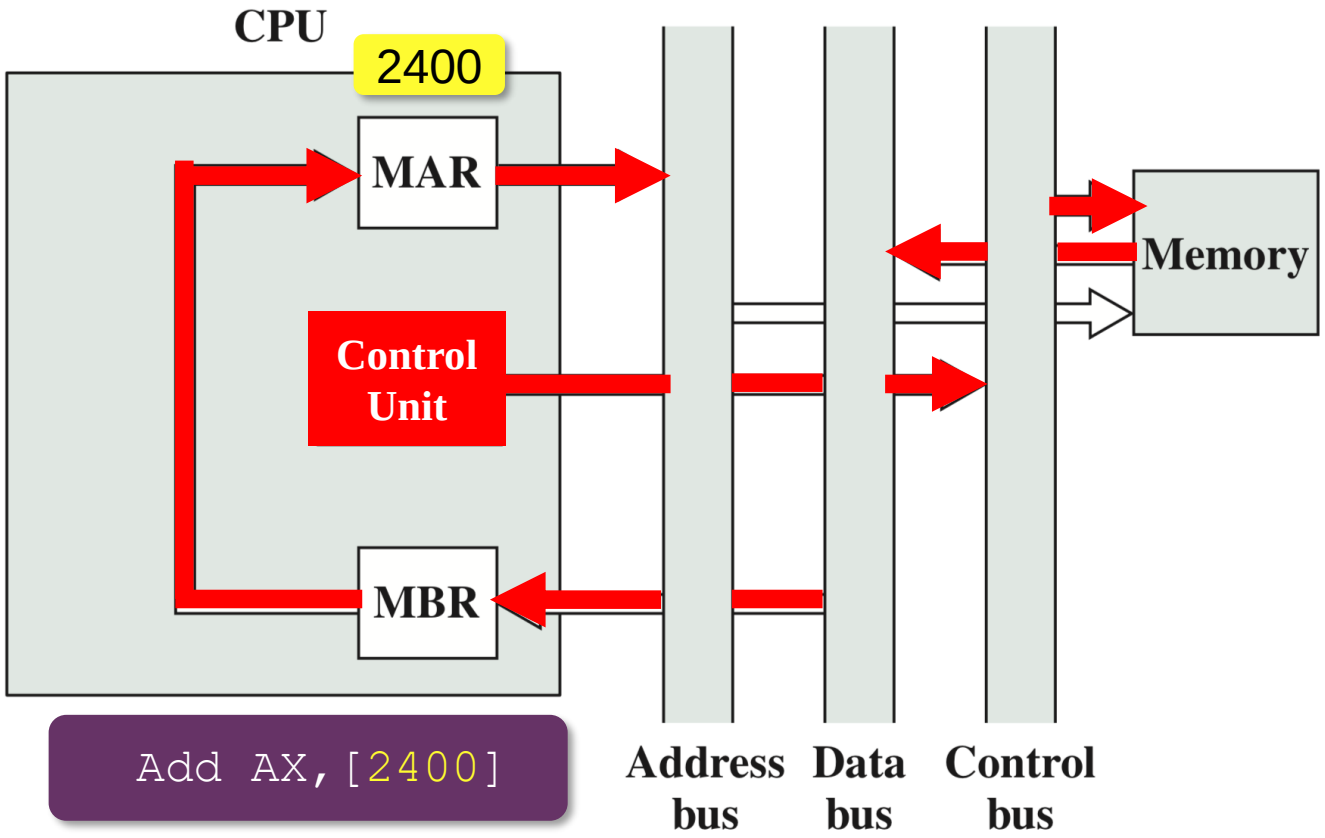


Figure: Data flow, indirect cycle.

Example : Indirect cycle

(1) The control unit request a memory read,

(2) and the result is placed onto the data bus



(5) then placed on the address bus

(4) The address is moved into MAR,

(3) Then copied into MBR

MAR (Memory Address Register)
MBR (Memory Buffer Register)

Figure

(b) Execute Cycle

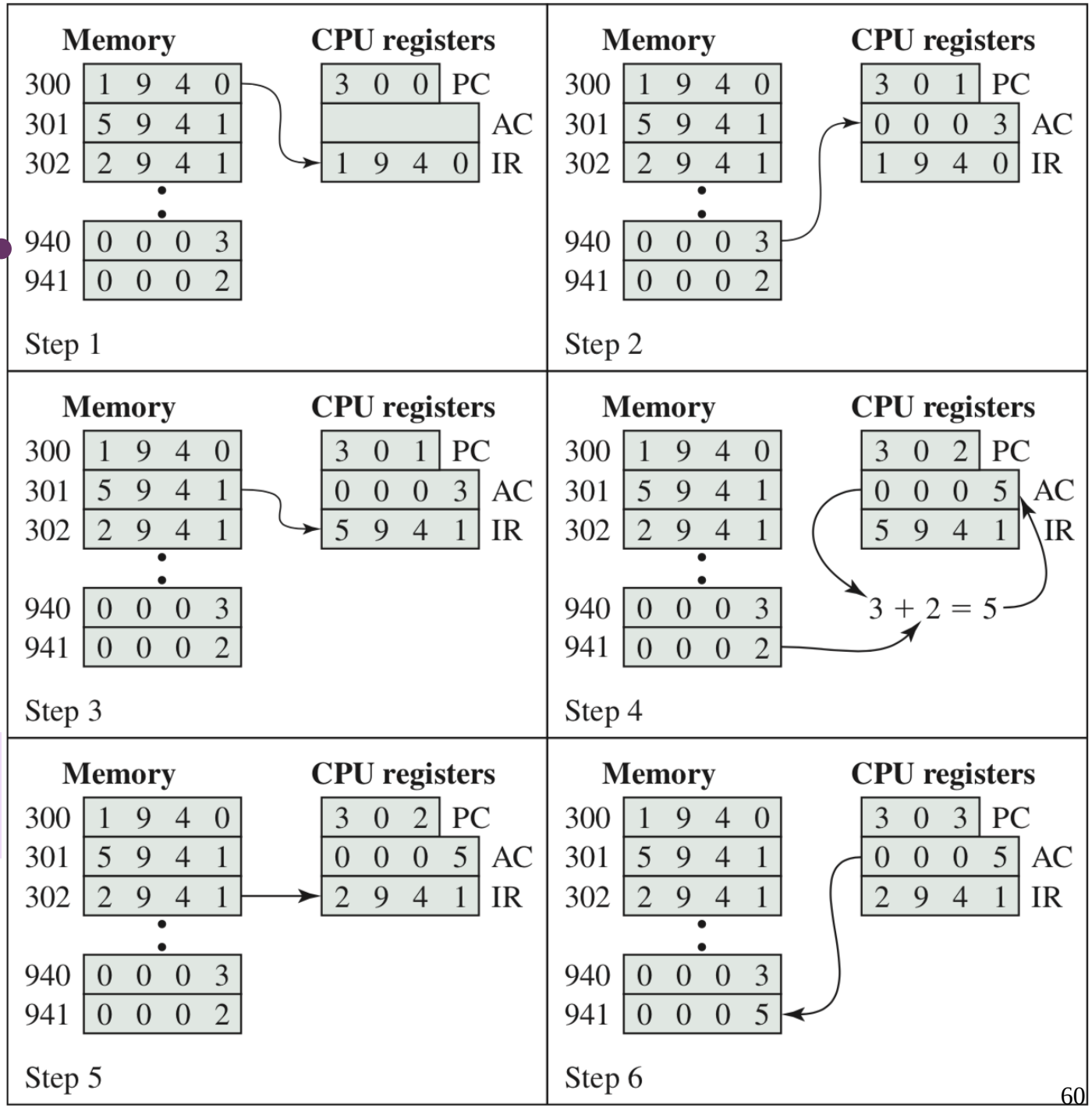
- The opcode of the instruction in IR will be decoded and related control signals will be generated.

Type of Opcode	Operation
Processor-memory	Data transfer between CPU and main memory.
Processor I/O	Data transfer between CPU and I/O module.
Data processing	Some arithmetic or logical operation on data.
Control	Alteration of sequence of operations. *e.g.* jump Combination of above

Example 6:

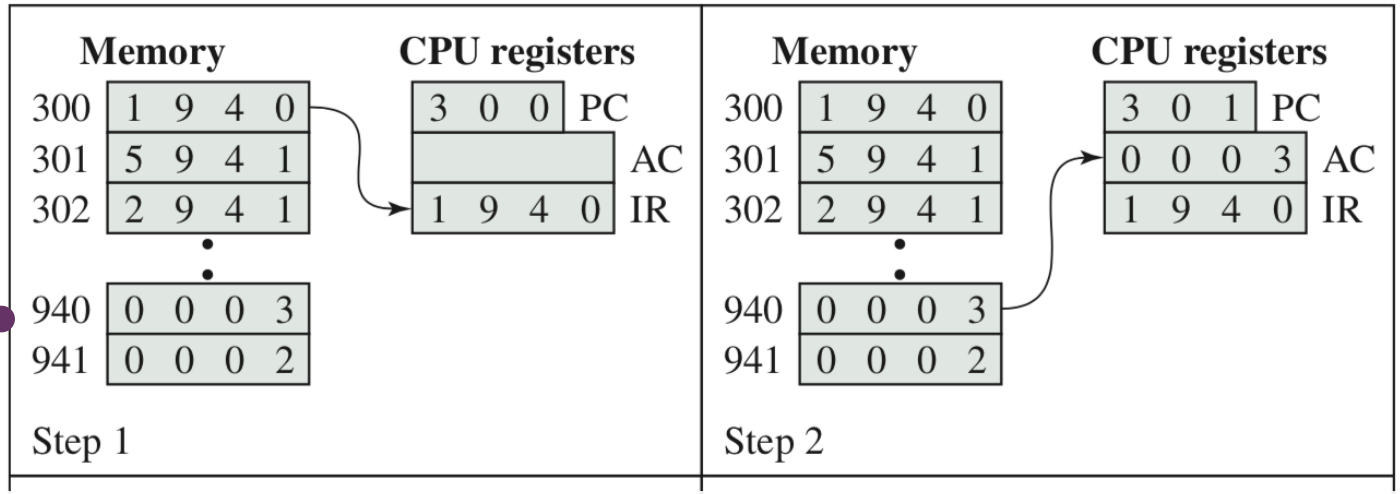
Example of program execution (contents of memory and registers in hexadecimal)

[940]+[941] → [941]



[940]+[941]
→ [941]

PC (Program Counter)
IR (Instruction Register)
AC (Accumulator)



Step 1:

- PC = 300 (address of 1st instruction.
- The value 1940 (in hex) is loaded into the IR.
- PC is incremented.

Step 2:

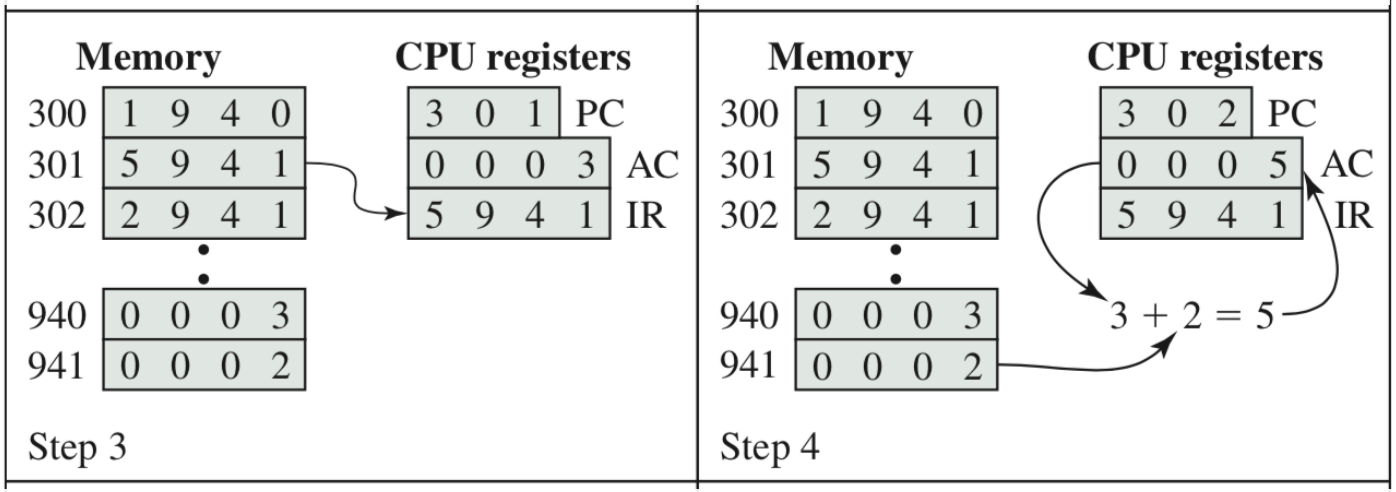
- The first 4 bits (1h) in the IR indicate that the AC is to be loaded.
- The remaining 12 bits specify the address (940) from which data is to be loaded. → go to 940 and get data (0003) to load into AC.

[940]+[941]
→ [941]

PC (Program Counter)
IR (Instruction Register)
AC (Accumulator)

Step 3:

- The next instruction (5941) is fetched from location 301.
- the PC is incremented.



Step 4:

- The old contents of the AC (0003) and the contents of location 941 (0002)are added.
- and the result (0005) is stored in the AC.

[940]+[941]
→ [941]

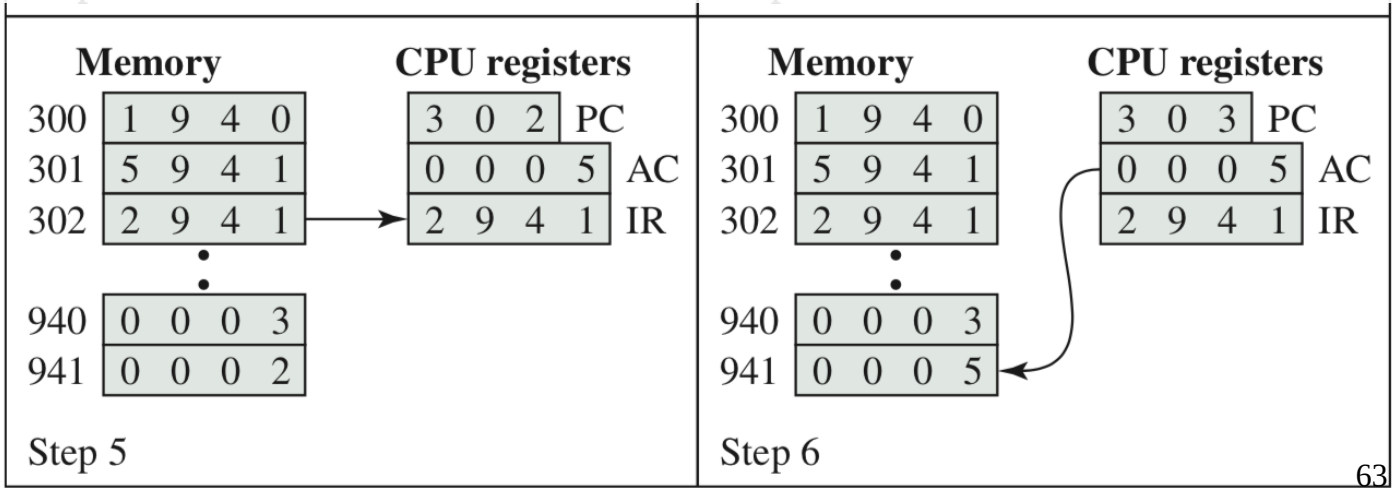
PC (Program Counter)
IR (Instruction Register)
AC (Accumulator)

Step 5:

- The next instruction (2941) is fetched from location 302.
- the PC is incremented.

Step 6:

- The contents of the AC are stored in location 941.



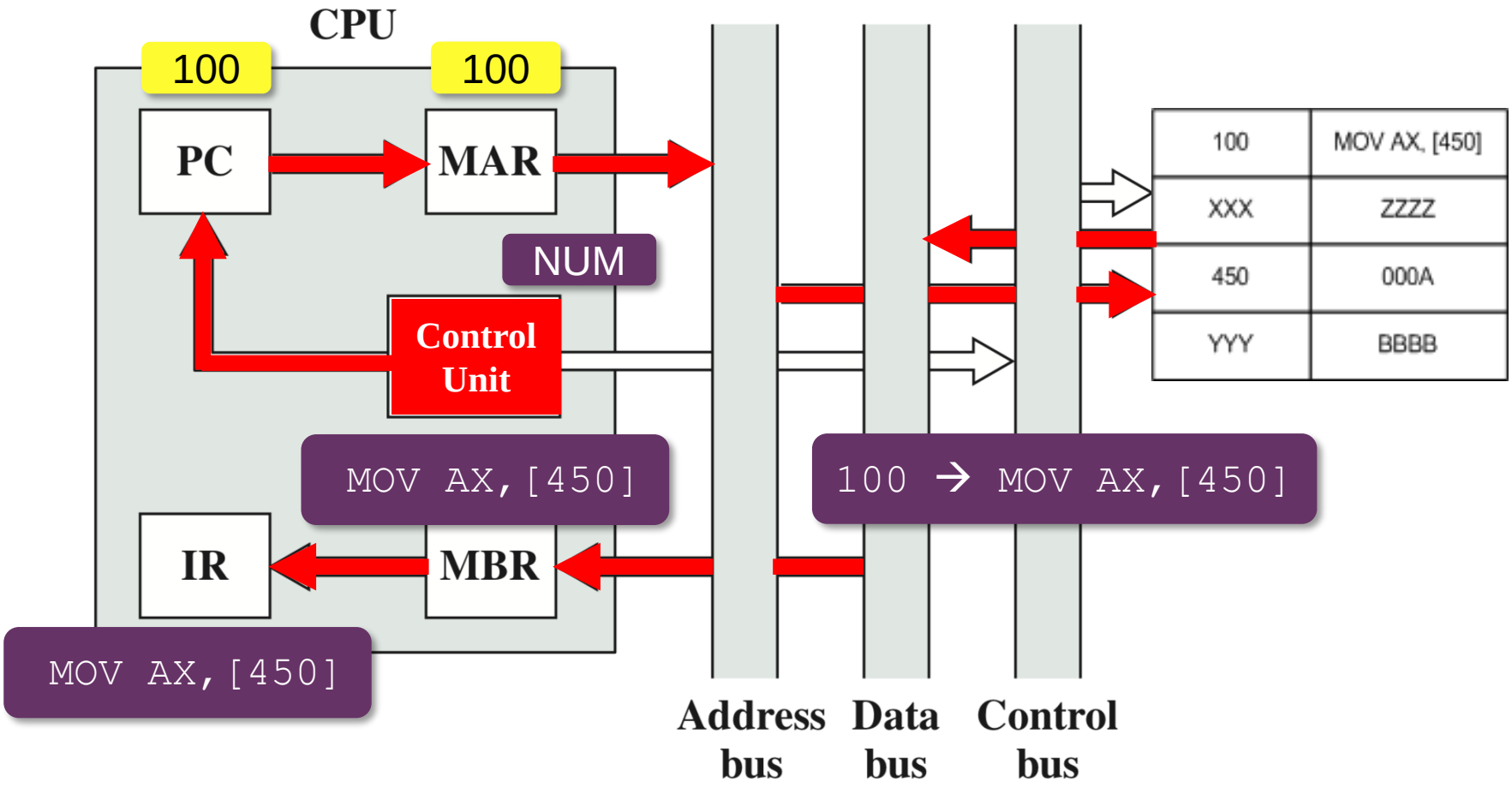
[940]+[941]
→ [941]

Summary

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the instruction register IP and the PC is incremented.
2. The first 4 bits in the IR indicate that the AC is to be loaded. The remaining 12 bits specify the address (940) from which data are to be loaded.
3. The next instruction (5941) is fetched from location 301 and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added and the result is stored in the AC.
5. The next instruction (2941) is fetched from location 302 and the PC is incremented.
6. The contents of the AC are stored in location 941.

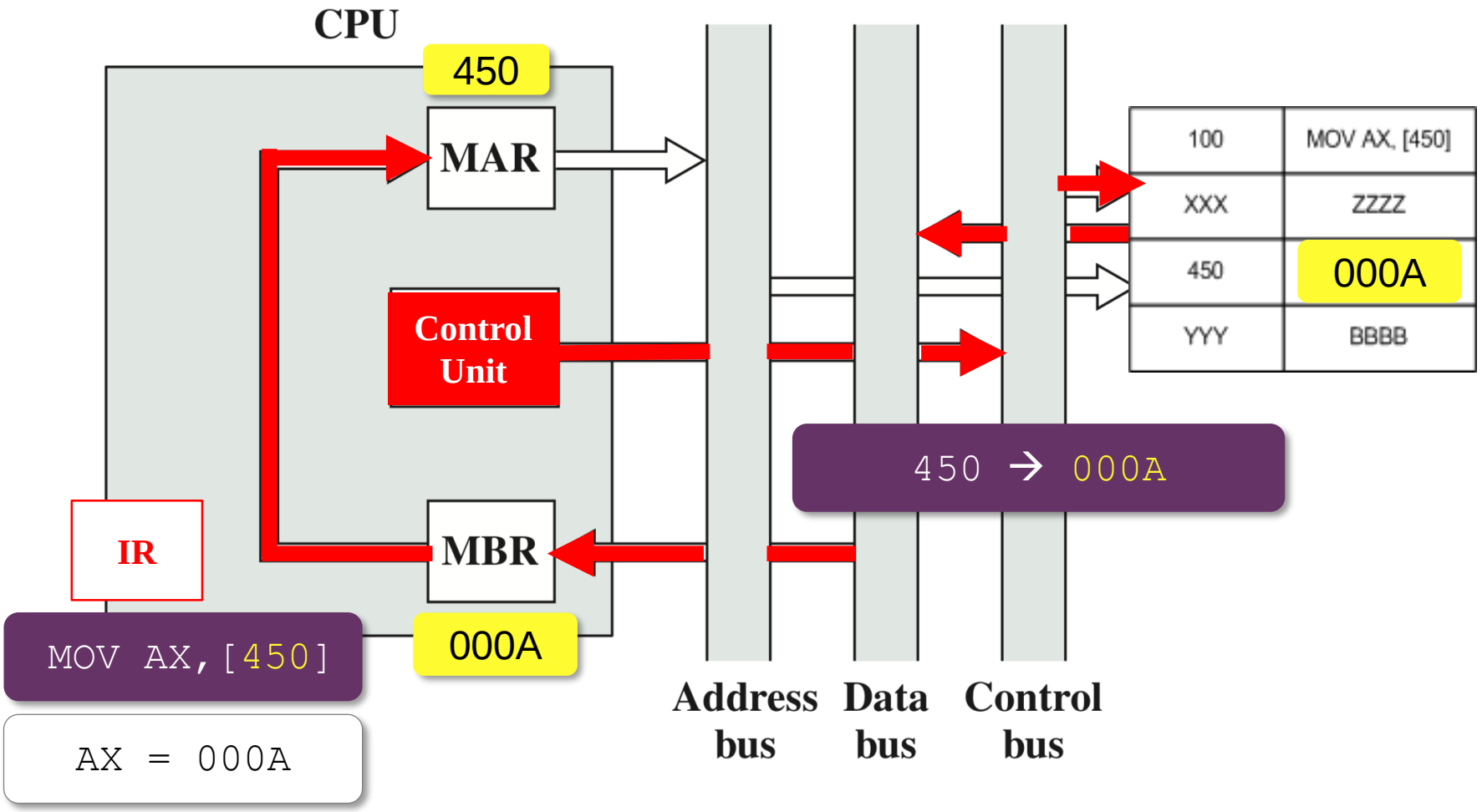
Example 7a: Fetch cycle

`MOV AX, NUM`



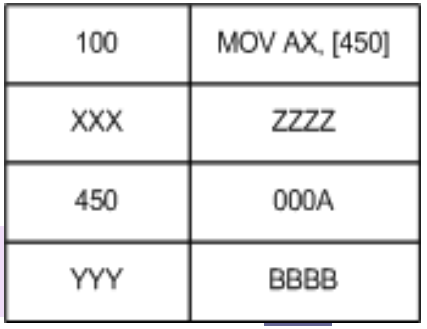
Example 7b: Indirect cycle

`MOV AX, NUM`



Example 7c:

`MOV AX, NUM`



100	MOV AX, [450]
XXX	ZZZZ
450	000A
YYY	BBBB

Clock	IP/PC	MAR	MBR	IR	AX	Micro-operation
t_0	100					IP/PC = 100
t_1		100				*Fetch cycle*: MAR ← IP/PC;
t_2	101		4450			MBR ← [MAR]; IP/PC ← IP/PC +1;
t_3				4450		IR ← MBR *Indirect cycle*;
t_4		450				MAR← IR[operand/address];
t_5			000A			MBR ← Mem[MAR]; *Execution cycle*:
t_6					000A	AX ← MBR;

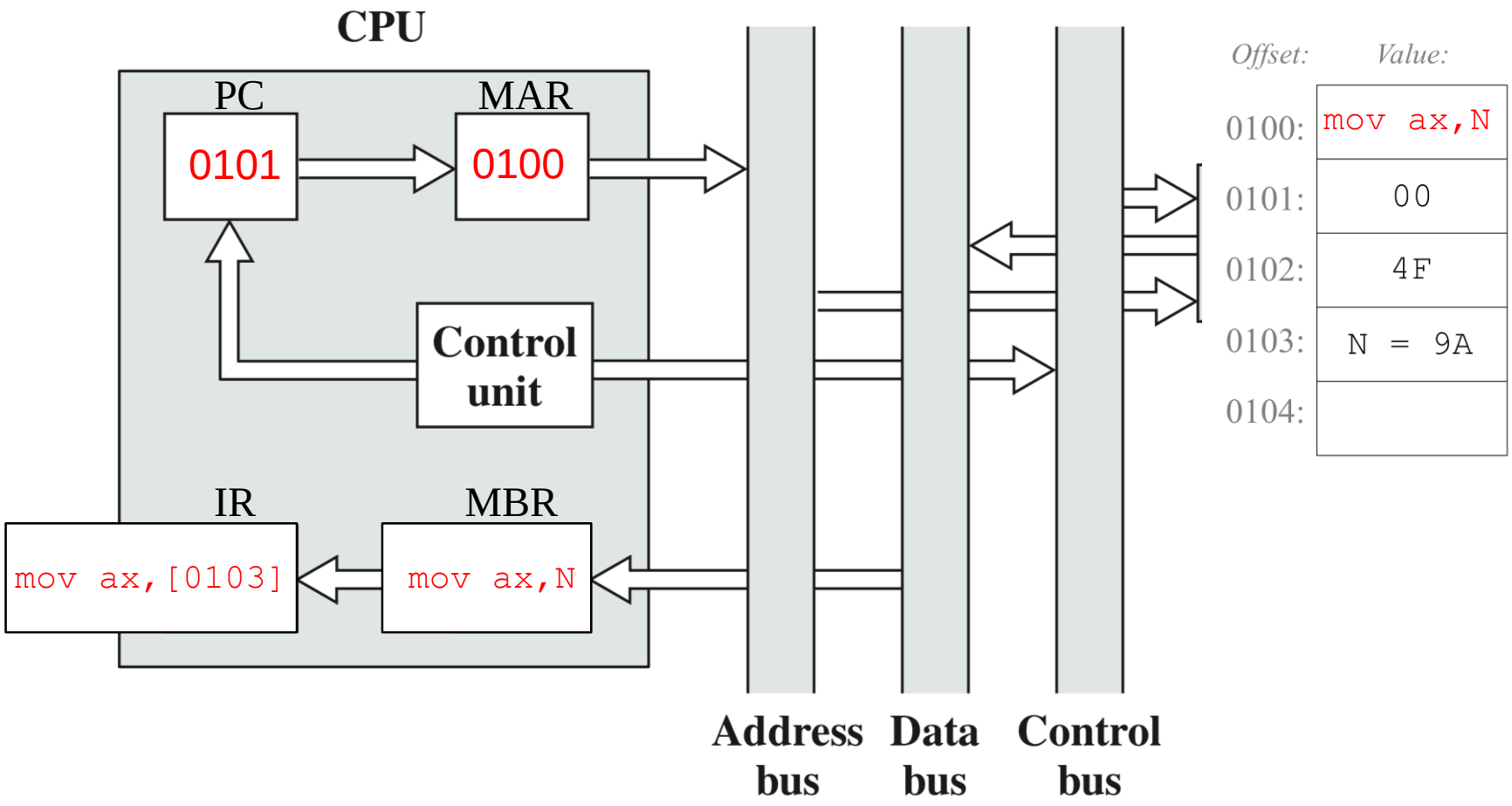
Example 7c:

MOV AX, NUM

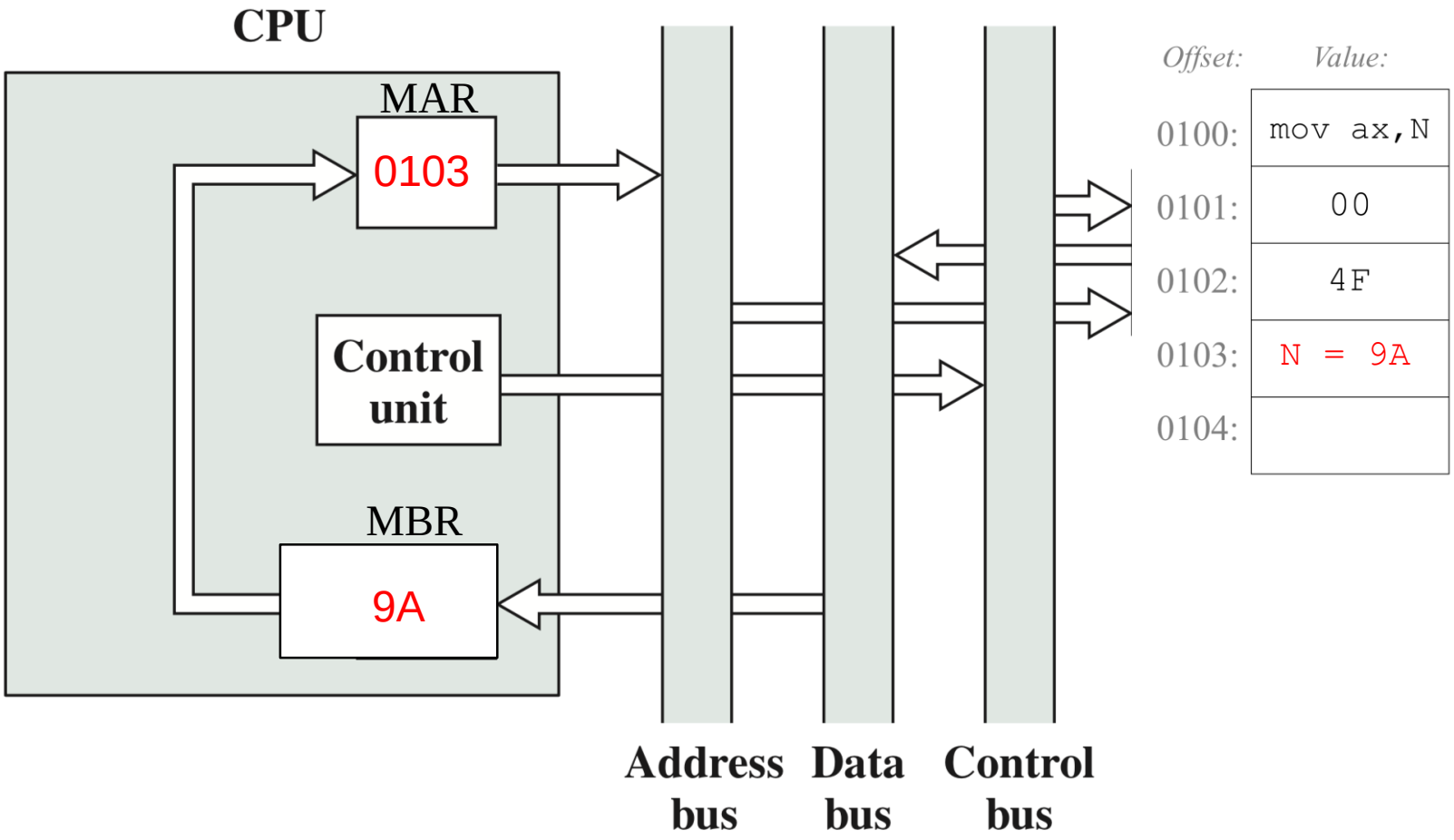
100	MOV AX, [450]
XXX	ZZZZ
450	000A
YYY	BBBB

Clock	IP/PC	MAR	MBR	IR	AX	Micro-operation
t_0	100					IP/PC = 100
t_1	100	100				*Fetch cycle*: MAR ← IP/PC;
t_2	101	100	4450			MBR ← [MAR]; IP/PC ← IP/PC +1;
t_3	101	100	4450	4450		IR ← MBR *Indirect cycle*;
t_4	101	450	4450	4450		MAR← IR[operand/address];
t_5	101	450	000A	4450		MBR ← Mem[MAR]; *Execution cycle*:
t_6	101	450	000A	4450	000A	AX ← MBR;

Example 8: Trace the execution of the instruction.



(Indirect addressing)



Exercise 5.1 :

Based on previous example, complete the following table in tracing the execution of the instruction.

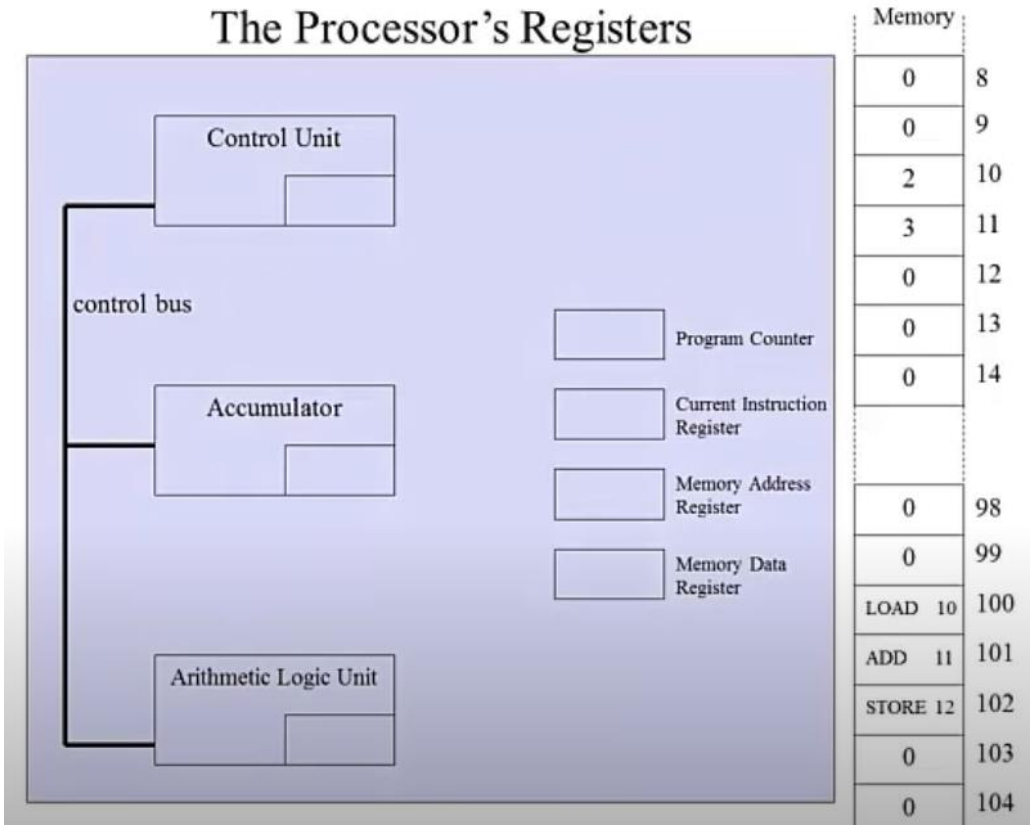
Clock	IP/PC	MAR	MBR	IR	_____	Micro-operation
t_0						IP/PC = _____
t_1						*Fetch cycle*: MAR ← IP/PC;
t_2						MBR ← [MAR]; IP/PC ← IP/PC +1;
t_3						IR ← MBR *Indirect cycle*;
t_4						MAR← IR[operand/address];
t_5						MBR ← Mem[MAR]; *Execution cycle*:
t_6						_____ ← MBR;

Solution 5.1 :

Clock	IP/PC	MAR	MBR	IR	AX	Micro-operation
t_0	0100					IP/PC = 0100
t_1		0100				*Fetch cycle*: MAR ← IP/PC;
t_2	0101		mov ax,N			MBR ← [MAR]; IP/PC ← IP/PC +1;
t_3				mov ax,N		IR ← MBR *Indirect cycle*;
t_4		0103				MAR← IR[operand/address];
t_5			9A			MBR ← Mem[MAR]; *Execution cycle*:
t_6					9A	AX ← MBR;

Fetch Control Cycle

- Watch this video that illustrates the fetch control cycle in details
- https://www.youtube.com/watch?v=jFDMZpkUWCw



Activity 1

Exercise 5.2 :

Trace the execution of the instruction by showing all the changes in CPU registers (control and general purpose registers) as well as the *micro-operations* related to the instructions. Use the given initial table.

Memory address	Memory Content	Instruction/Data
39D	A450	L1: XCHG CX, NUM
39E	B451	SUB VAL1,CX
39F	C39D	JMP L1:
450	100	NUM
451	500	VAL1

Clock	IP/PC	MAR	MBR	IR	CX	NUM	VAL1	Micro-operation
t_0					200	100		

MAR (Memory Address Register)
MBR (Memory Buffer Register)
IR (Instruction Register)

Memory address	Memory Content	Instruction/Data
39D	A450	L1: XCHG CX, NUM
39E	B451	SUB VAL1,CX
39F	C39D	JMP L1:
450	100	NUM
451	500	VAL1

Solution 5.2 :

Clock	IP/PC	MAR	MBR	IR	CX	NUM	VAL1	Micro-operation
t_0	39D				200	100		IP/PC = 39D
t_1		39D						MAR ← IP/PC
t_2								MBR ← [MAR] IP/PC ← IP/PC + 1
t_3								IR ← MBR
t_4								MAR ← IR[address]
t_5								MBR ← [MAR]
t_6								Exchange: CX ←→ MBR
t_7								
t_8								[MAR] ← MBR

MAR (Memory Address Register)
MBR (Memory Buffer Register)
IR (Instruction Register)

Memory address	Memory Content	Instruction/Data
39D	A450	L1: XCHG CX, NUM
39E	B451	SUB VAL1,CX
39F	C39D	JMP L1:
450	100	NUM
451	500	VAL1

Clock	IP/PC	MAR	MBR	IR	CX	NUM	VAL1	Micro-operation
t_9								IP/PC = 39E
t_{10}								MAR ← IP/PC
t_{11}								MBR ← [MAR] IP/PC ← IP/PC + 1
t_{12}								IR ← MBR
t_{13}								MAR ← IR[address]
t_{14}								MBR ← [MAR]
t_{15}								Subtract: VAL1=VAL1– CX
t_{16}								[MAR] ← MBR

MAR (Memory Address Register)
MBR (Memory Buffer Register)
IR (Instruction Register)

Memory address	Memory Content	Instruction/Data
39D	A450	L1: XCHG CX, NUM
39E	B451	SUB VAL1,CX
39F	C39D	JMP L1:
450	100	NUM
451	500	VAL1

Clock	IP/PC	MAR	MBR	IR	CX	NUM	VAL1	Micro-operation
t_{17}								IP/PC = 39F
t_{18}								MAR ← IP/PC
t_{19}								MBR ← [MAR] IP/PC ← IP/PC + 1
t_{20}								IR ← MBR
t_{21}								Execute JMP: IP/PC ← L1

(c) Interrupt Cycle

- All computers provide a way of interrupting the fetch-decode-execute cycle.
- Interrupts occur when:
  - A user break (*e.g.*, Control+C) is issued.
  - I/O is requested by the user or a program.
  - A critical error occurs.
- Interrupts can be caused by hardware or software.

→ Software interrupts are also called *traps*.

- Interrupt processing involves adding another step to the fetch-decode-execute cycle as shown below:

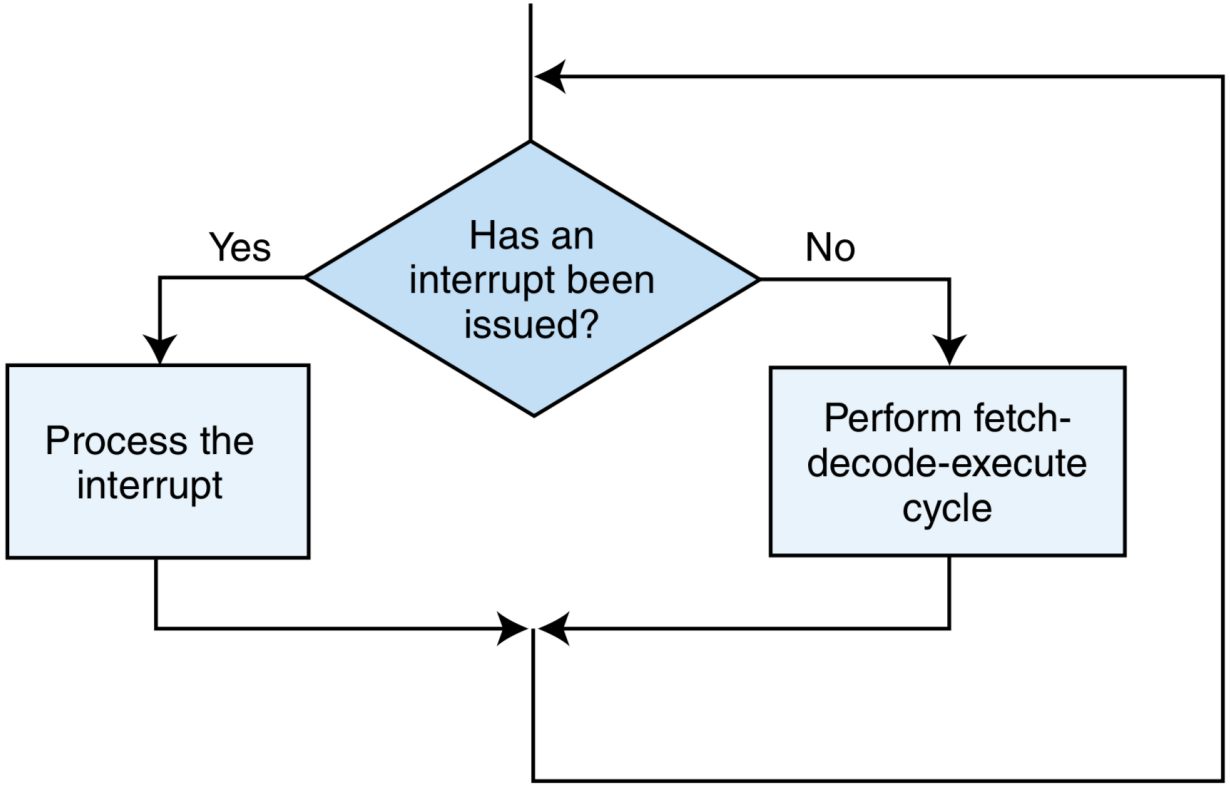


Figure: Modified Instruction Cycle to Check for Interrupt

- The current contents of the PC must be saved so that the processor can resume normal activity after the interrupt.

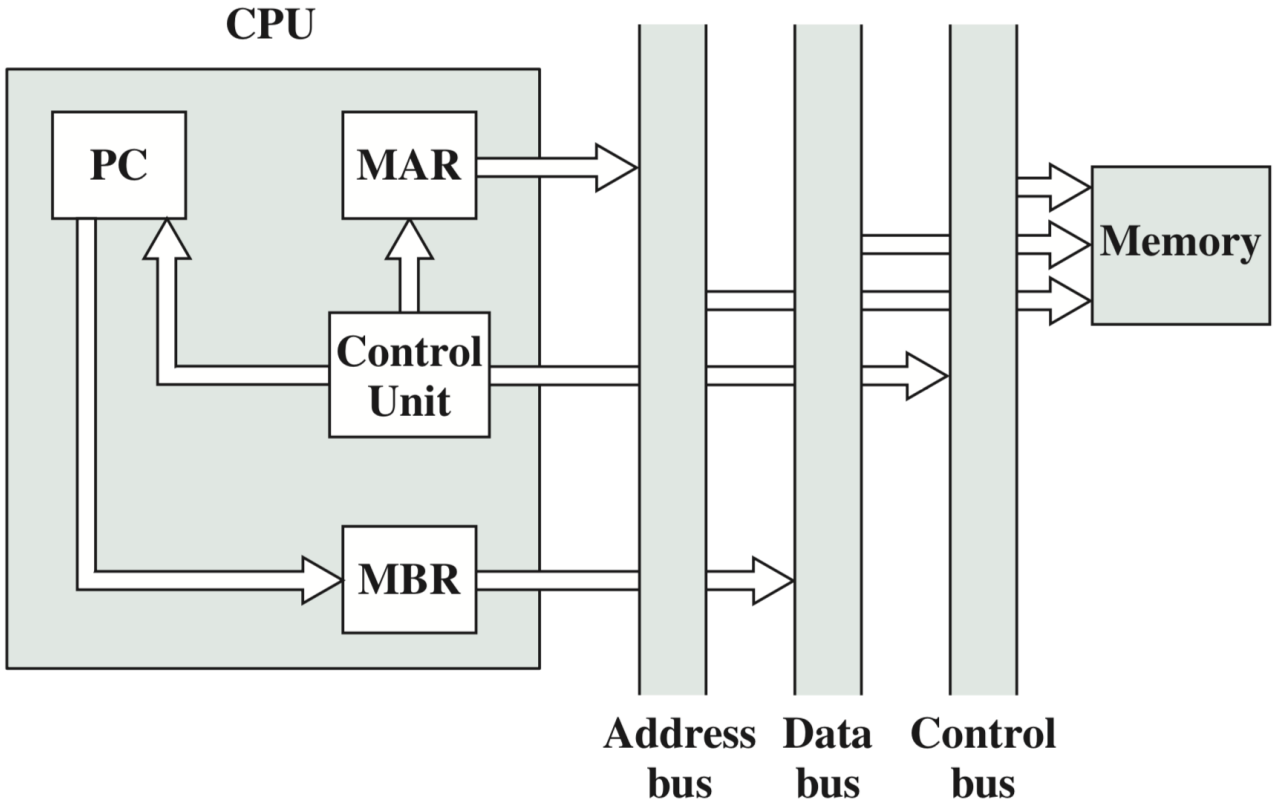
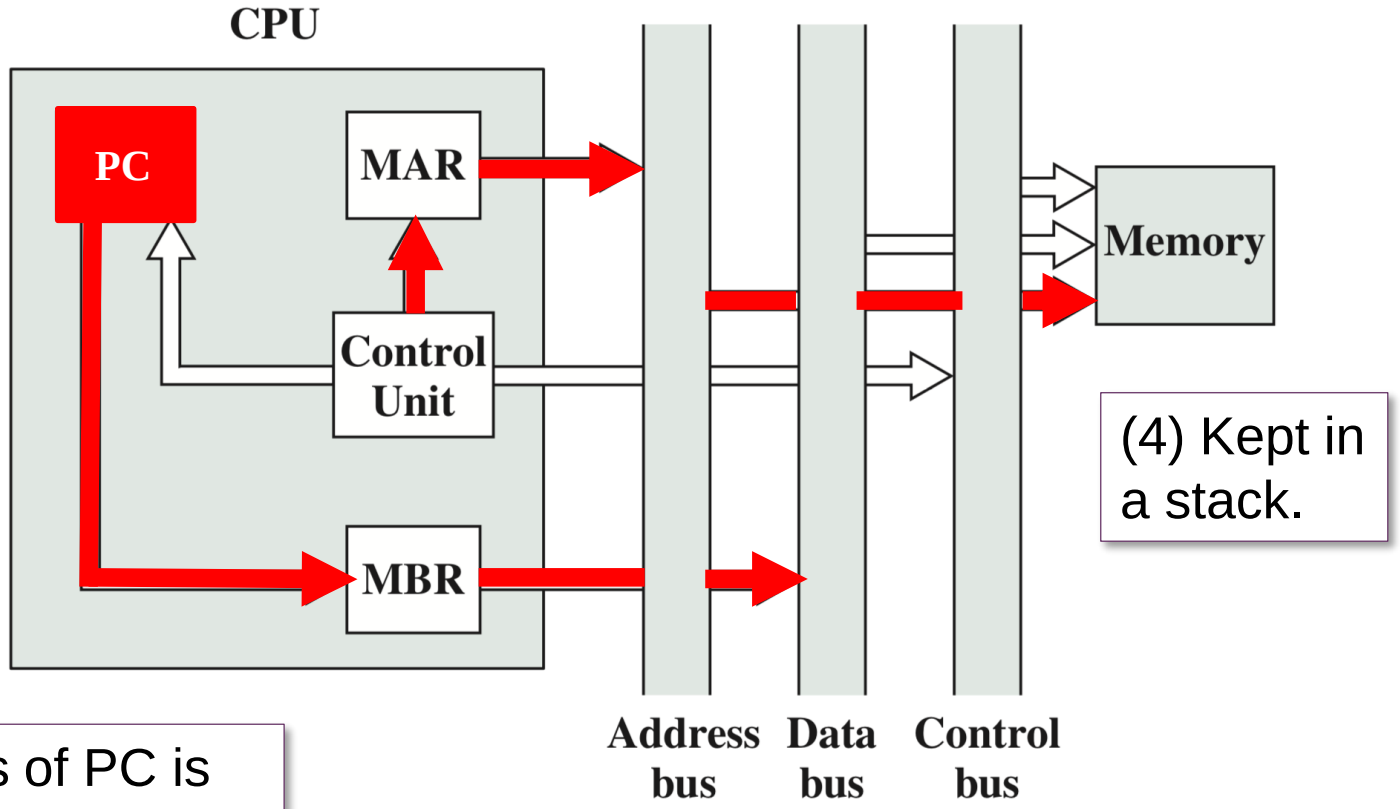


Figure: Data flow, interrupt cycle.

(1) The current content of PC must be saved → to come back after interrupt.

(3) The special place in memory to keep this is given by CU to MAR , then placed on address bus.



(5) PC is loaded with the address of the interrupt routine.

(4) Kept in a stack.

(2) The contents of PC is transferred to MBR, then placed on data bus, to be written into memory

Data flow, interrupt cycle.

Interrupt

- Virtually all computers provide a mechanism by which other modules (I/O, memory) may *interrupt* the normal processing of the processor *.
- Interrupts let the CPU execute its normal instruction sequence and pause to service the external devices **ONLY** when they signal (the interrupts) that they are ready for the CPU's attention.

* William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.89.

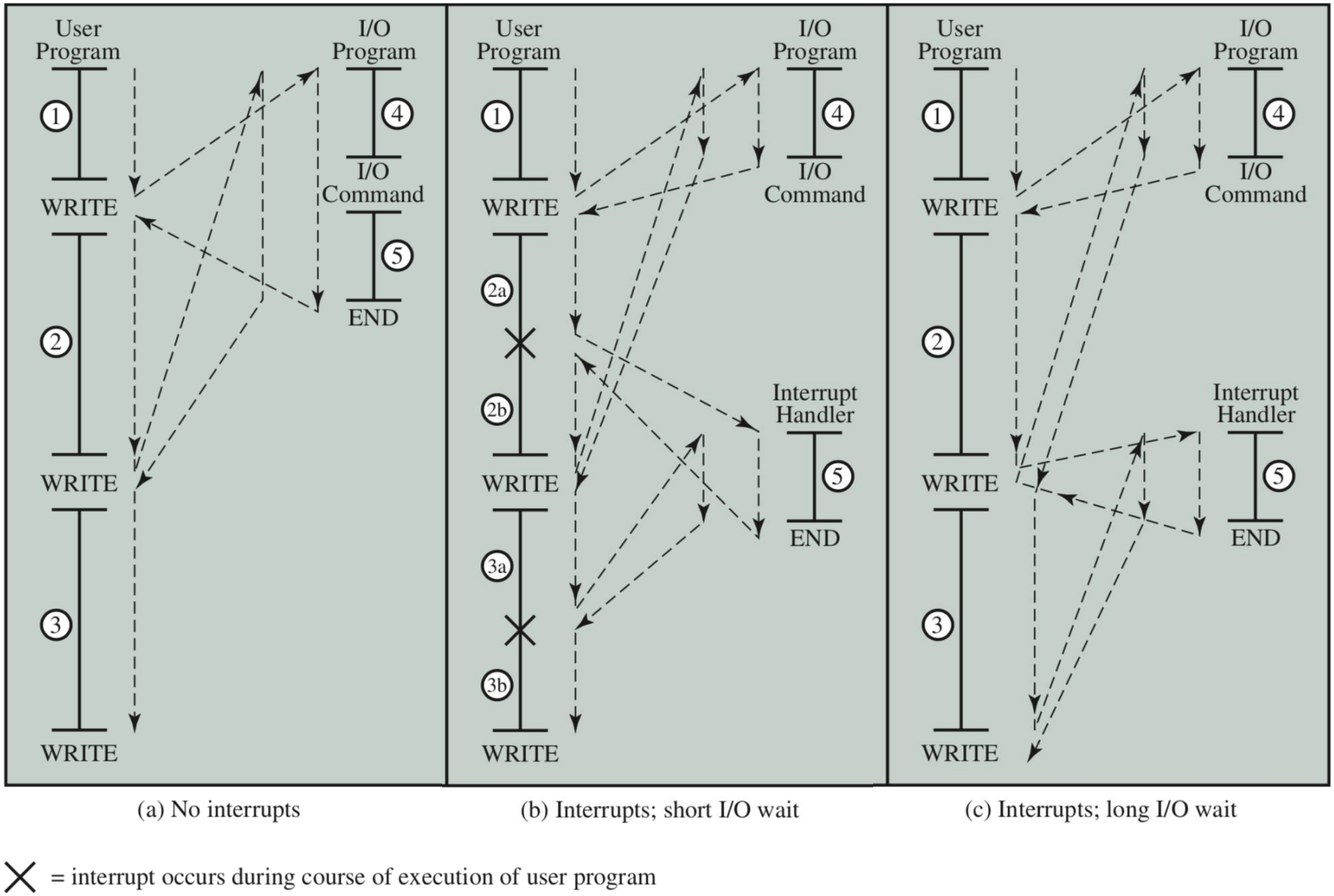
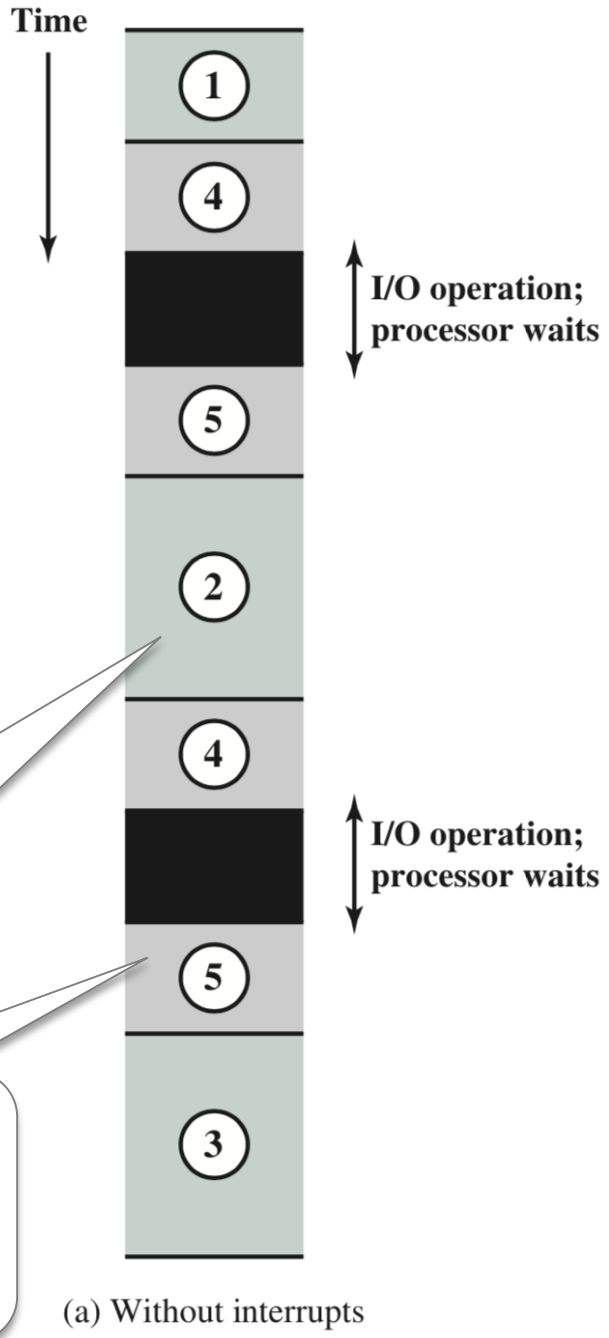


Figure: Program Flow of Control without and with Interrupts

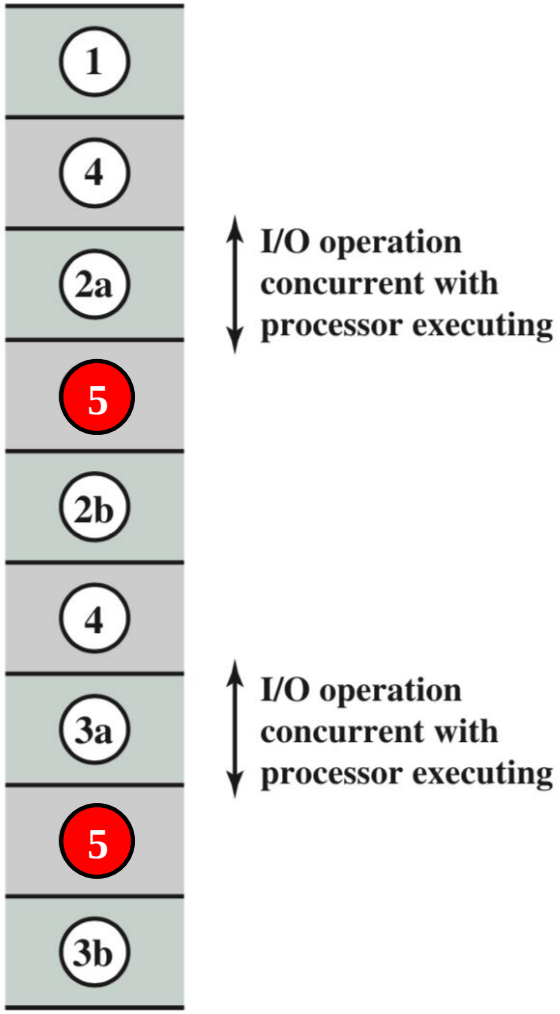
- To appreciate the gain in efficiency, consider the timing diagram based on the flow of control in previous figures.

User program *code segments* *are shaded green.*

I/O program *code segments* *are shaded gray.*



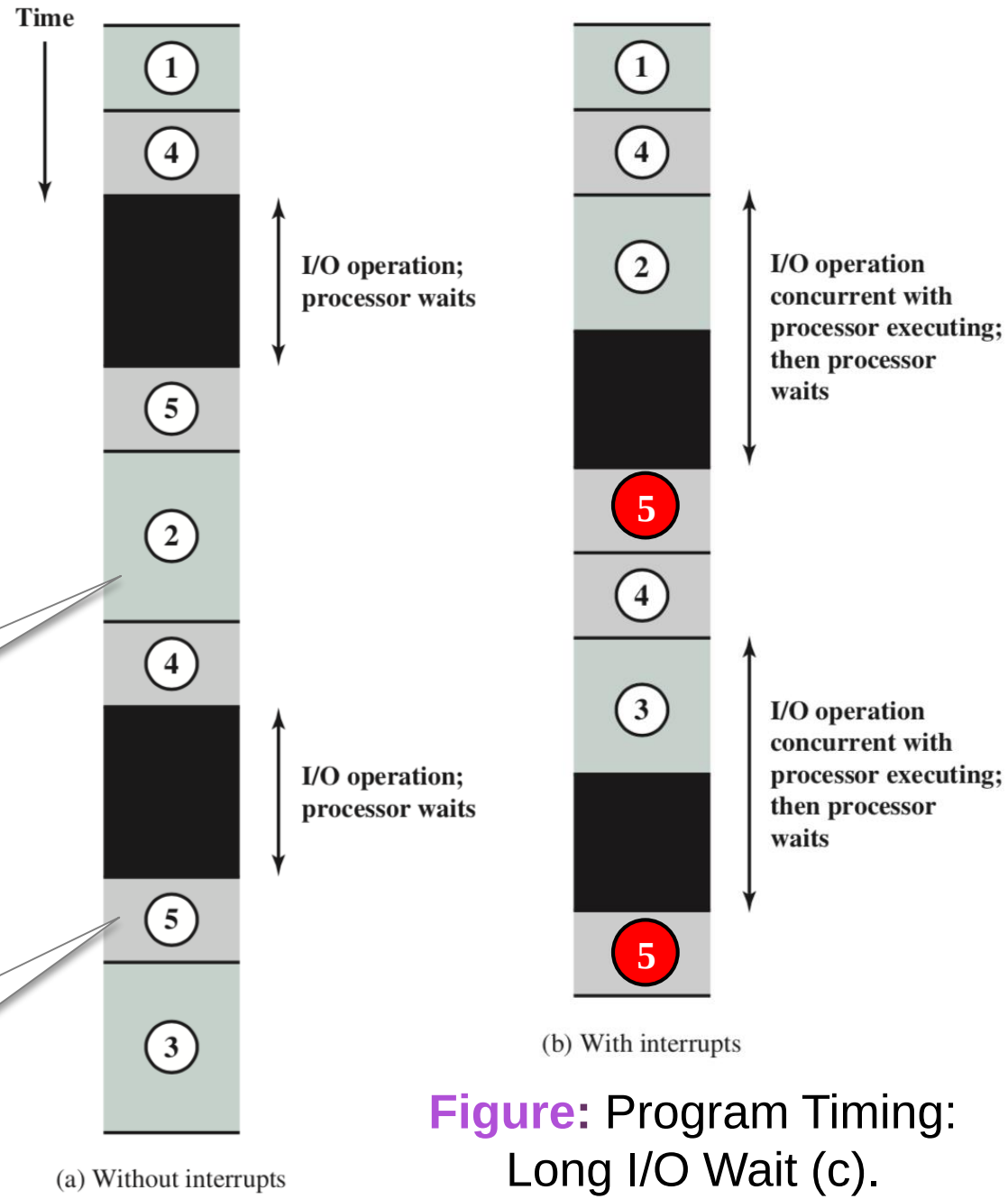
(a) Without interrupts



(b) With interrupts

Figure: Program Timing: Short I/O Wait (b).

- The more typical case, especially for a slow device such as a printer → the I/O operation will take much more time than executing a sequence of user instructions.



User program code segments

I/O program code segments

Figure: Program Timing: Long I/O Wait (c).

5

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, or reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation, request service from the processor, or to signal a variety of error conditions.
Hardware failure	Generated by a failure such as power failure or memory parity error.

Figure: Class of interrupts.

5 Summary 1

- The major components of a computer system are its *control unit*, *registers*, *memory*, *ALU*, and *data path*.
- Computers run programs through iterative *fetch-decode-execute* cycles → Summarize the instruction cycle.
- Distinguish between user-visible and control/status registers, and discuss the purposes of *registers* in each category.