POST AND TELECOMMUICATIONS INSTITUTE OF TECHNOLOGY

Central Processing Unit (CPU)

Faculty: Computer science – IT1

Instructor: Viet Hoang Pham

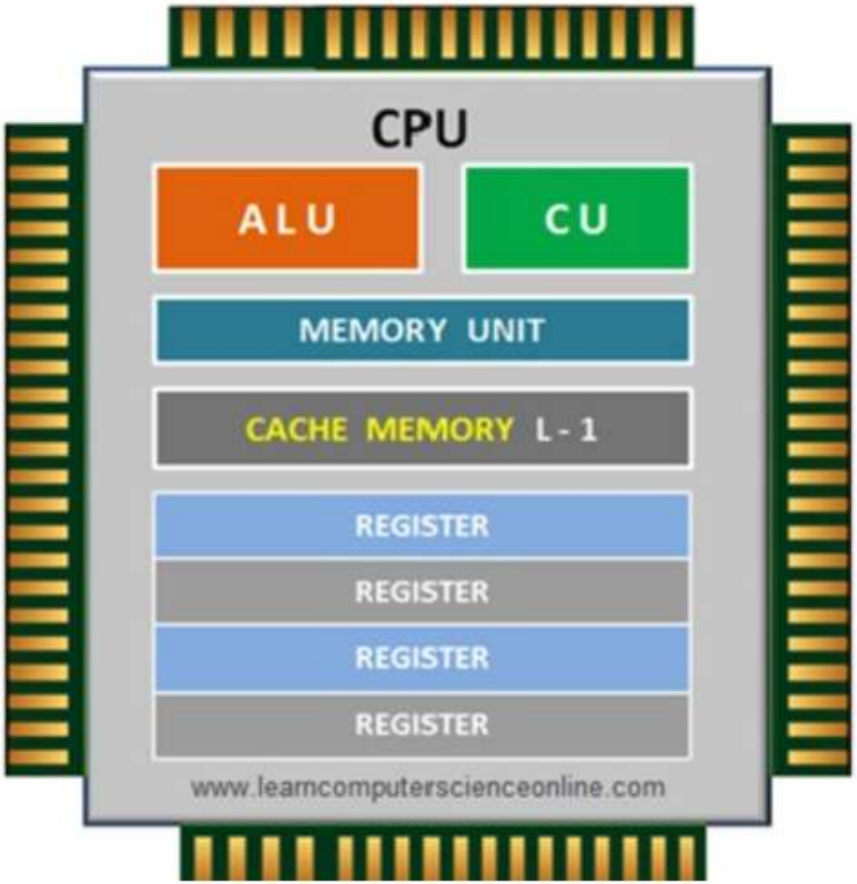
Email: vietph@ptit.edu.vn

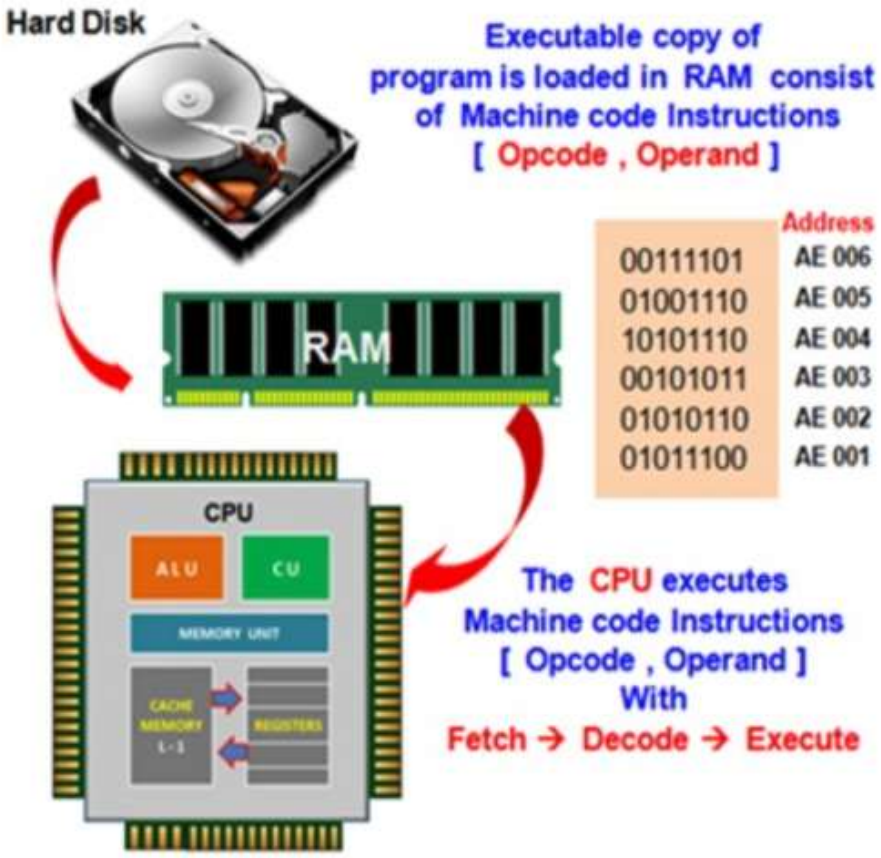
Phone: 0945296388

Contents

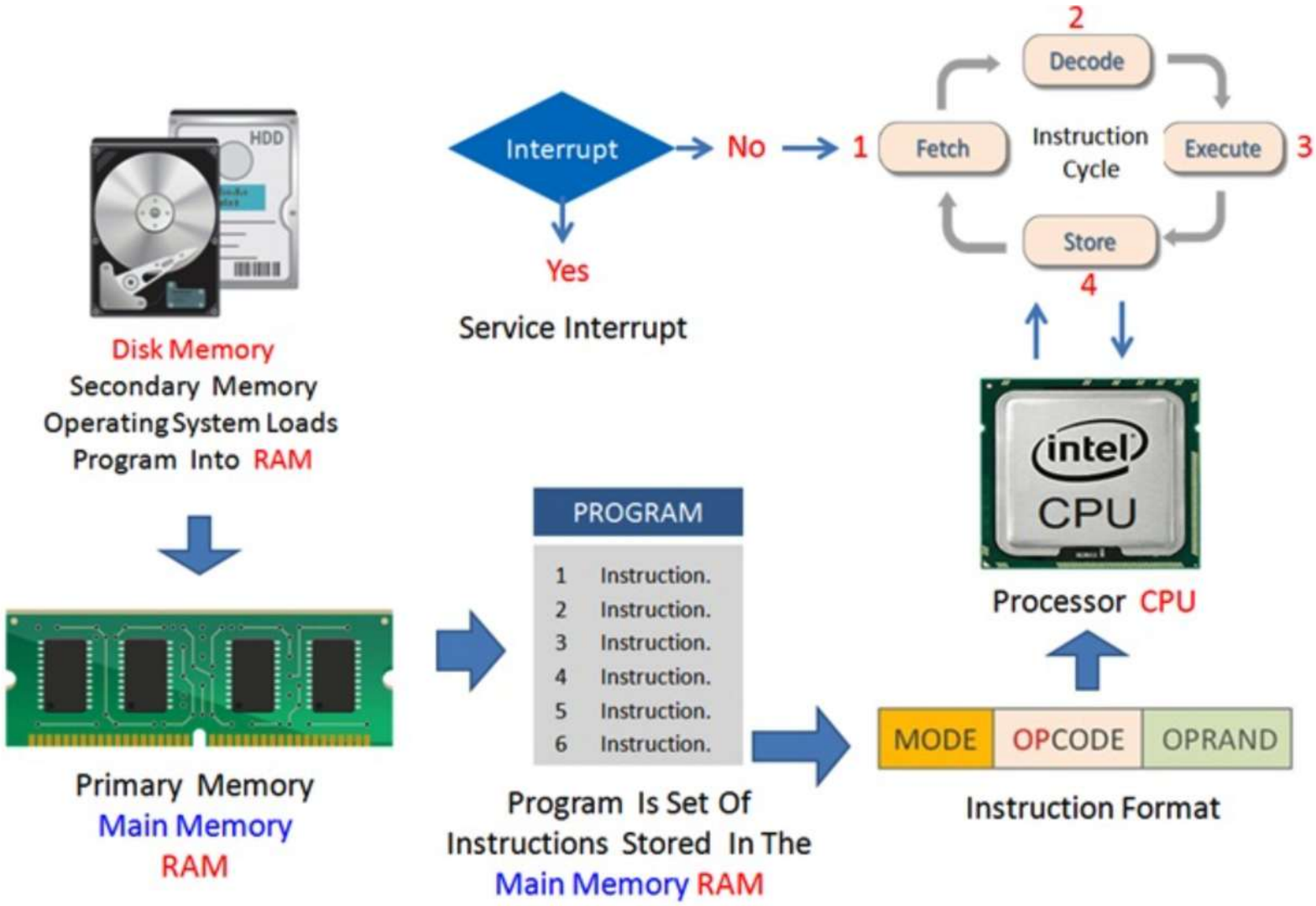
- General diagram and functions of CPU components.
  - Registers, Control Unit, Arithmetic and Logic Unit, Internal Bus System.
- Machine instruction set
  - Instruction format and components, Instruction operand types, Addressing modes, Some common instruction types
- CPU pipeline
  - Pipeline mechanism, resolution methods include handling data and resource conflicts, and branching.

Introduction



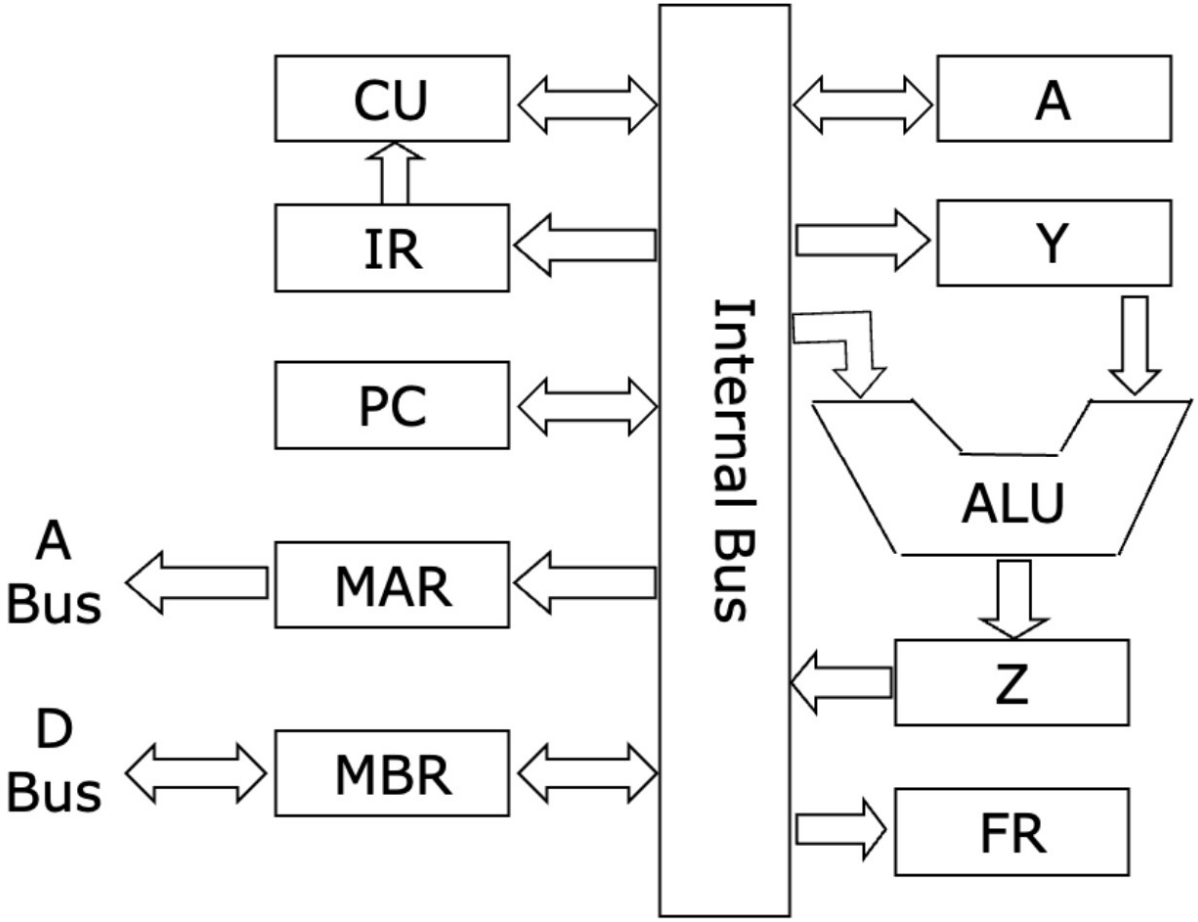


Introduction



CPU diagram

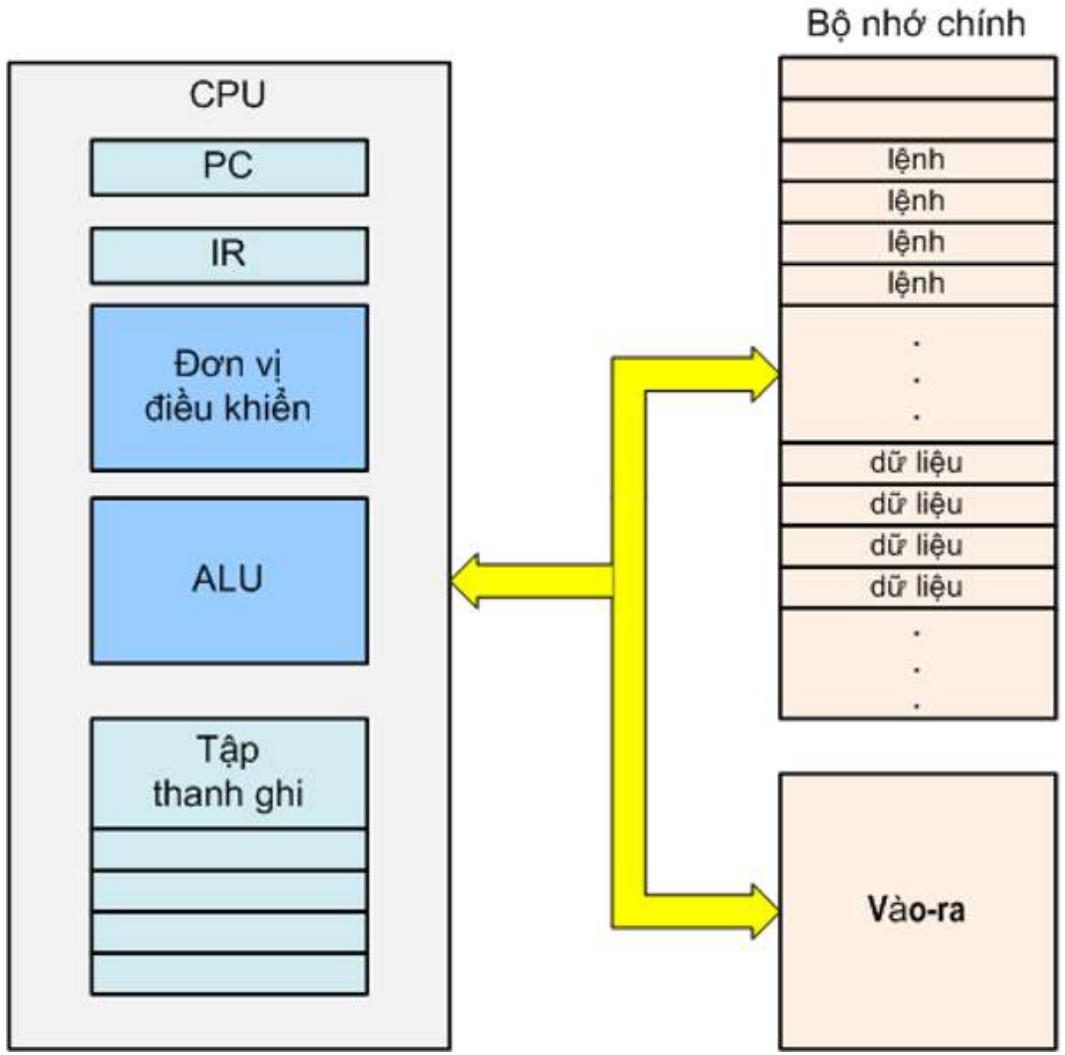
- CU: Control Unit
- IR: Instruction Register
- PC: Program Counter
- MAR: Memory Address Register
- MBR: Memory Buffer Register
- A: Accumulator Register
- Y, Z: Temporary Register
- FR: Flag Register
- ALU: Arithmetic and Logic Unit



Instruction Execution

❖CPU Instruction Cycle

- The instruction cycle is the time it takes for the CPU to execute an instruction, from the moment the CPU allocates the memory address for the instruction until it completes the execution of that instruction.



Instruction Execution

❖CPU Instruction Cycle

Each CPU instruction cycle is described by the following steps:

1. When a program is run, the operating system loads the program code into RAM.
2. The address of the first instruction in the program is placed in the PC register.
3. The address of the memory location containing the instruction is transferred to the A bus via the MAR register.
4. The A bus transmits the address to the Memory Management Unit (MMU).
5. The MMU selects the memory location and generates a READ signal.
6. The instruction contained in the memory location is transferred to the MBR register via the D bus.

Instruction Execution

❖CPU Instruction Cycle

Each CPU instruction cycle is described by the following steps:

7. The MBR transfers the instruction to the IR register. The IR then transfers the instruction to the CU.
8. The CU decodes the instruction and generates processing signals for other units, such as the ALU, to execute the addition instruction.
9. The address in the PC is incremented to point to the next instruction in the program to be executed.
10. Steps 3-9 are repeated to execute all the program instructions.

❖CPU Instruction Cycle

Example: Describe the CPU processing cycle for the following instruction and the CPU state (values of the CPU registers) when executing the instruction: z = x + y.

- Assuming
  - the encoded addition instruction has a value of 1001H,
  - the instruction is at address 1000H,
  - x has a value of 1
  - y has a value of 2.

Instruction Execution

❖CPU Instruction Cycle

Example: Describe the CPU processing cycle for the following instruction and the CPU state (values of the CPU registers) when executing the instruction: z = x + y.

Register	Before execution	After the Command Receipt phase	After the Execution phase
PC	1000H	1001H	1001H
MAR	(Empty)	1000H	1000H
MBR	(Empty)	1001H	(Value of z)
IR	(Empty)	1001H	1001H
A	(Determine later)	(Determine later)	3
Y	(Empty)	(Empty)	2
Z	(Empty)	(Empty)	3

CPU registers

- Act as memory cells, registers can:
  - Temporarily store instructions and data to be processed by CPU;
  - Small size;
  - High-speed access
- CPU can have tens (80x86) to hundreds of registers;
- Size can vary 8, 16, 32, 64, 128 and 256 bits:
  - 8086-80286: 8 and 6 bit
  - 80386-Pentium II: 16-32 bit
  - Pentium 4, Core Duo: 32, 64 and 128 bit.

CPU registers

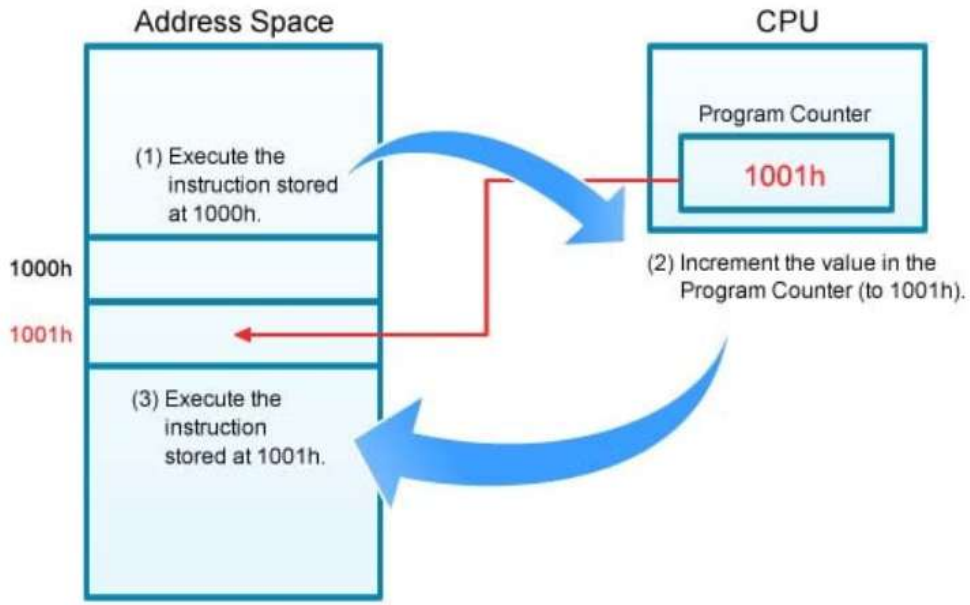
Accumulator (A)

- Present in most CPU, A can be used for:
  - Storing the input operands
  - Storing the results
  - Communicating between CPU and I/O devices
- Sized can be: 8, 16, 32 and 64 bit.
- Example: Performing the calculation x + y -> z
  - Operand x is placed in register A
  - Operand y is placed in register Y
  - The ALU performs the addition A+Y, the result is stored in Z
  - The result is then placed back into A

CPU registers

❖Program Counter (PC)

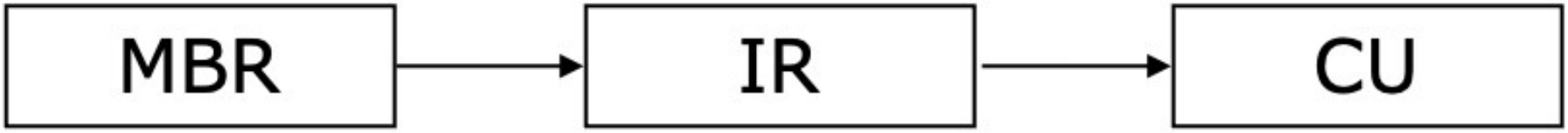
- The Program Counter/Instruction Pointer stores the address of the next instruction.
- The PC contains the memory address of the first instruction of the program when it is executed and loaded into memory.
- When the CPU finishes executing an instruction, the address of the next instruction is loaded into the PC.
- The size of the PC depends on the CPU design: 8, 16, 32, or 64 bits.



CPU registers

❖Instruction Register (IR)

- Stores the instruction being processed
- IR retrieves the instruction from the MBR and sends it to the CU for instruction decoding.



CPU registers

- MAR – (Memory address register):
  - Interface between CPU and bus A
  - Receive cell memory address from PC and pass it to bus A.
- MBR –(Memory buffer register):
  - Interface between CPU and bus D
  - Receive op-code from D bus and pass it to IR through internal bus in CPU.

CPU registers

❖Flag Register (FR)

- Stores the status of CPU in which each bit presents a "flag" derived from the computation output of ALU;
- Two types of flags:
  - Status flags: CF, OF, AF, ZF, PF, SF
  - Control flags: IF, TF, DF
- Flags often are used for branch or condition in the program;
- Size of FR relies on the design of CPU.

Flag	ZF	SF	CF	AF	IF	OF	PF	1
Bit No	7	6	5	4	3	2	1	0

CPU registers

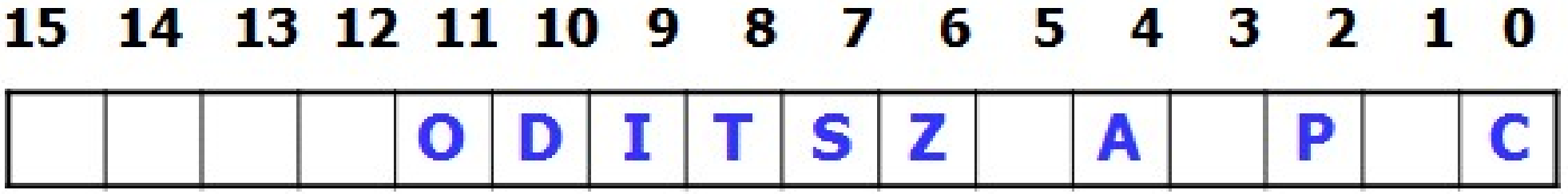
❖Flag Register (FR)

- ZF: Zero flag, ZF=1 if result (from ALU)=0 and ZF=0 if result <>0.
- SF: Sign flag, SF=1 if result is negative. SF=0 otherwise.
- CF: Carry flag, CF=1 if carried, CF=0 otherwise.
- AF: Auxilary flag, AF=1 if carried of an addition or borrowing of a subtraction to/from lower nibble (four lower bits).
- OF: Overflow flag, OF=1 if overflow, OF=0 otherwise.
- PF: parity flag, PF=1 if the sum of bits '1' is odd, and PF=0 if the sum of bits '1' is even.
- IF: Interrupt flag, IF=1: interrupt enable, IF=0: interrupt disable .

Flag	ZF	SF	CF	AF	IF	OF	PF	1
Bit No	7	6	5	4	3	2	1	0

CPU registers

- 8086's FR



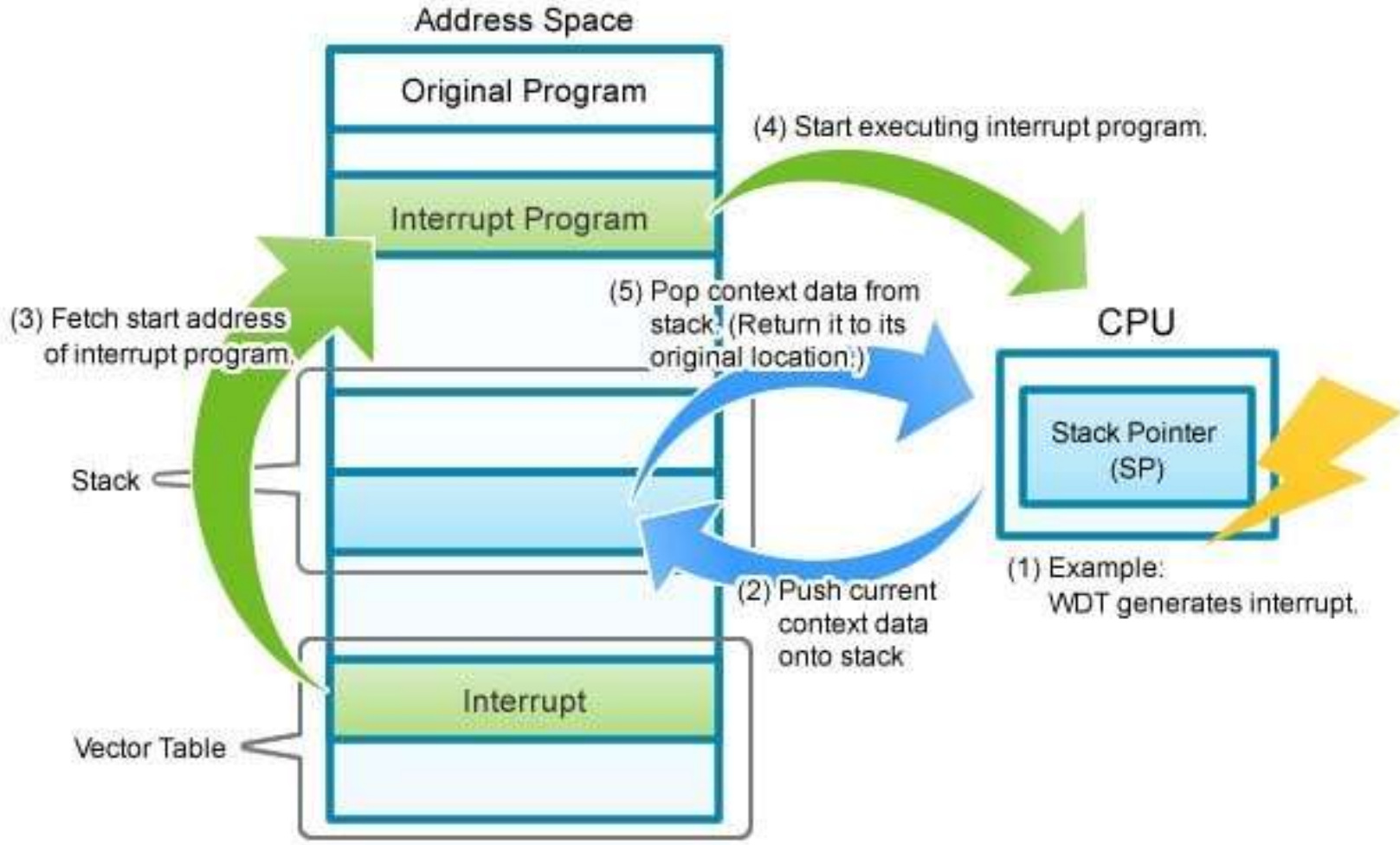
O OverFlow flag D : Direction flag I : Interrupt flag

T : Trap flag S : Sign flag Z : Zero flag

A : Auxiliary flag P : Parity flag C : Carry flag

CPU registers

- Illustrative



CPU registers

Stack Pointer (SP)

- Stack works under last in first out (LIFO) rule;
- SP contains the address of the top of the stack;
- Two operations:
  - Push:

    SP ← SP + 1

    {SP} ← Data
  - Pop:

    Register ← {SP}

    SP ← SP - 1

CPU registers

General Purpose Registers

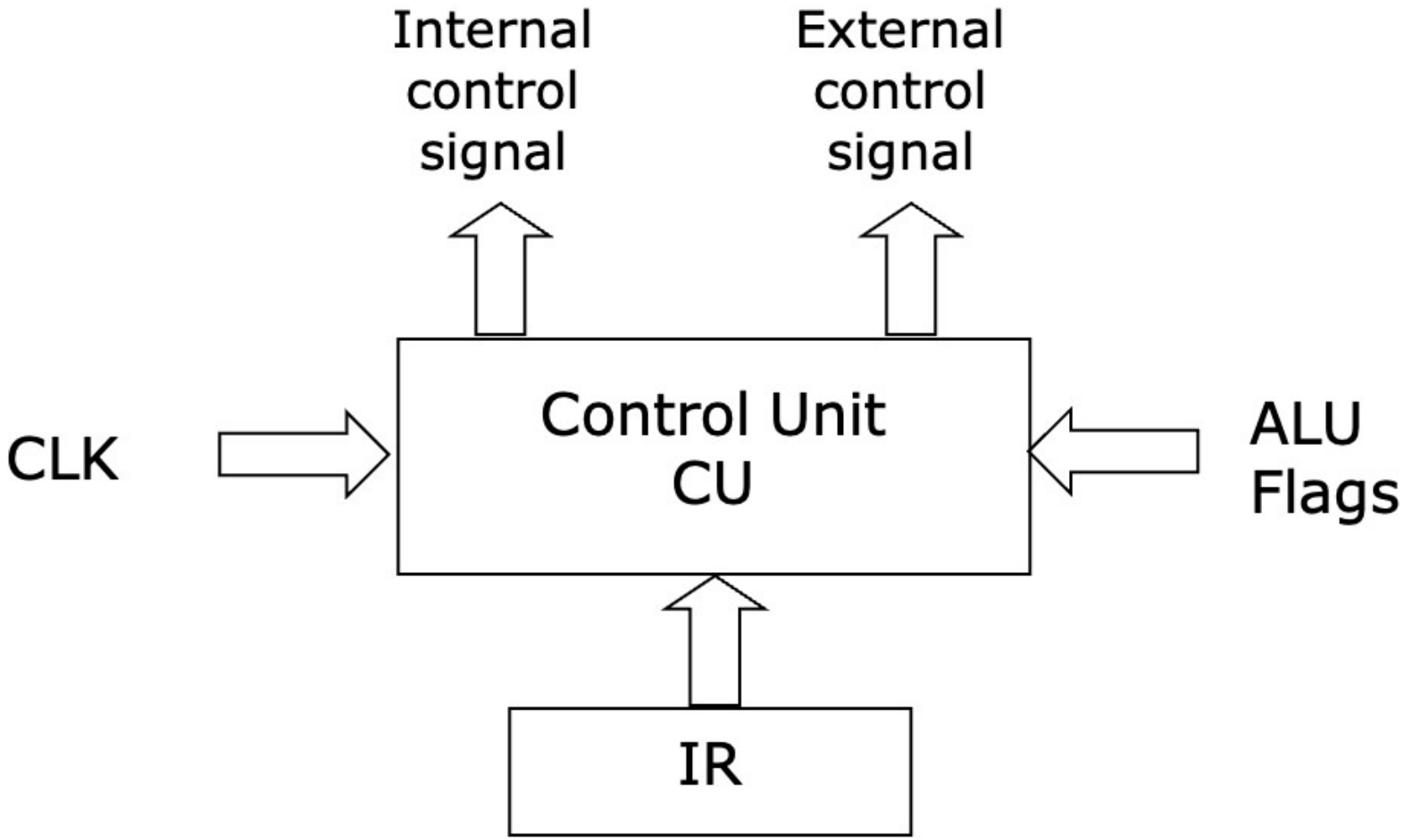
- General Purpose Registers are registers used for multi purposes:
- Examples: CPU Intel 8086 has four general purpose registers:
  - AX: Accumulator register
  - BX: Base register
  - CX: Counter register
  - DX: Data register

CPU registers

- Temporary registers can be used for:
  - Storing input operants
  - Storing output results
  - Concurrent execution
  - Out Of Order execution (OOO)

Control Unit (CU)

❖Control Unit

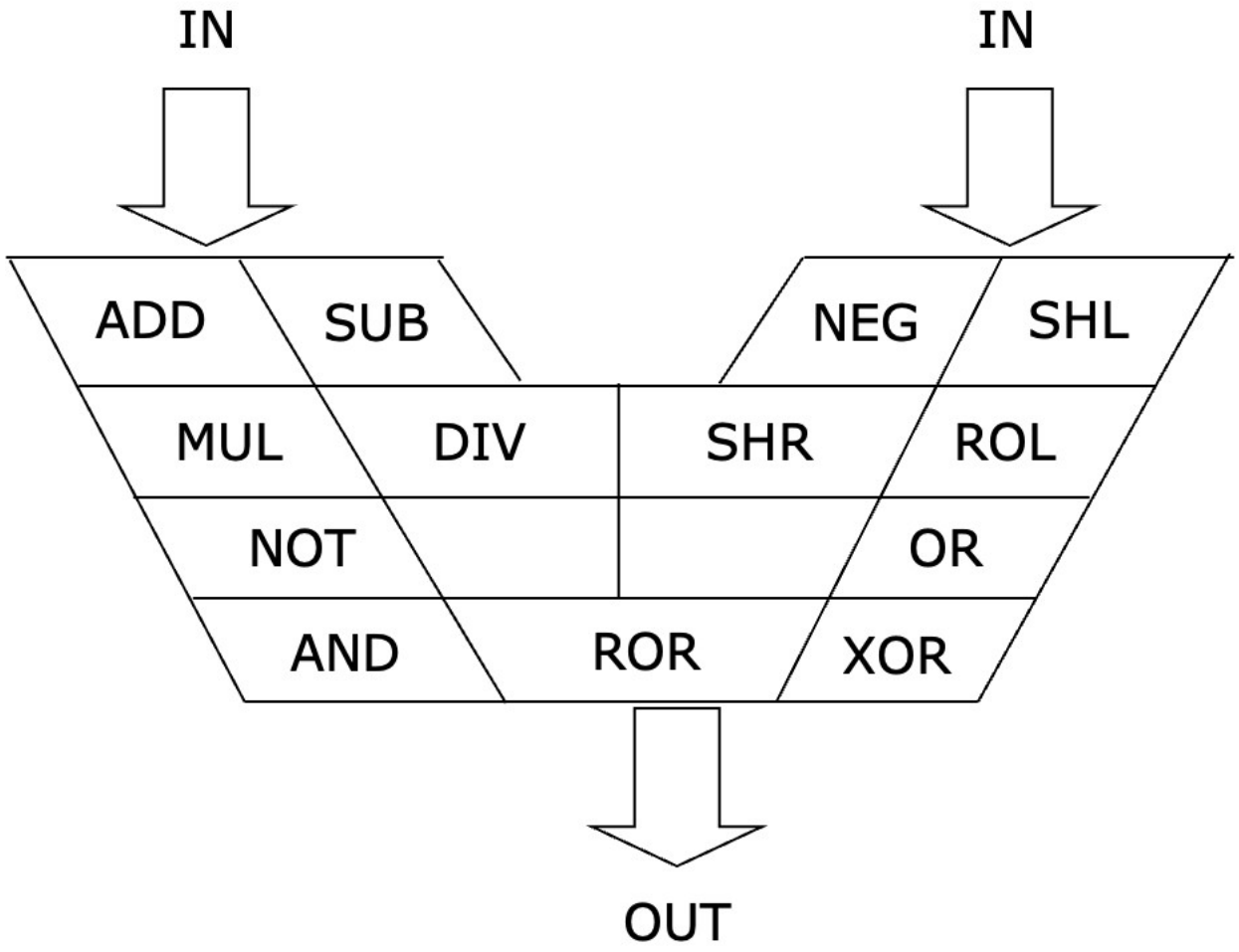


Control Unit (CU)

- Control Unit controlizes operations of CPU using CPU clock cycles;
  - CU's inputs:
    - Op-code stored in IR
    - FR
    - Clock cycles
  - CU's outputs:
    - Controlizes units inside CPU;
    - Controlizes units outside CPU
  - CU generizes clock cycles for synchronizing units inside and outside CPU.

Arithmetic logic unit (ALU)

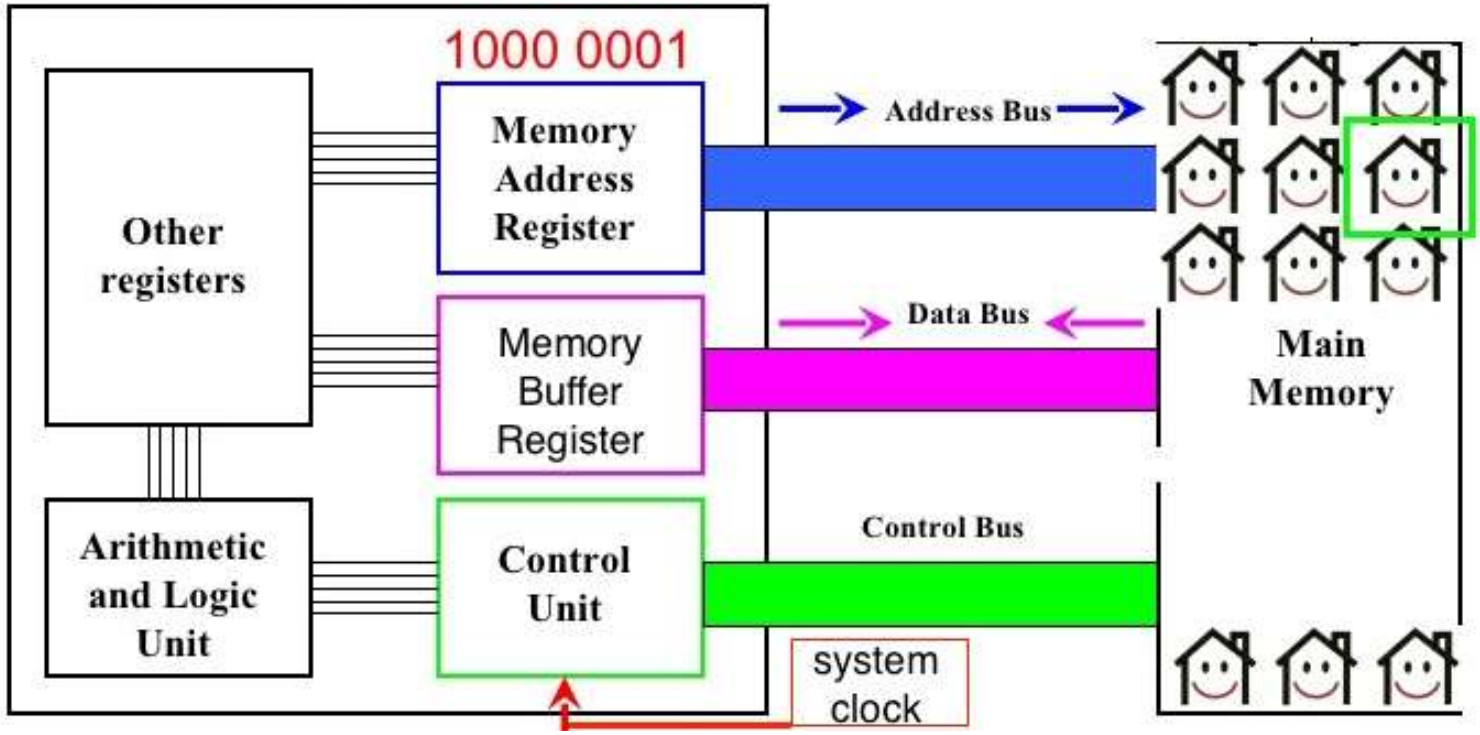
❖Arithmetic Logic Unit (ALU)



Arithmetic logic unit (ALU)

- ALU (Arithmetic and Logic Unit) comprises of computing sub-units:
  - ADD, SUB, MUL, DIV
  - SHIFT, ROTATE
  - NOT, AND, OR, XOR
  - ALU has
    - Two inputs (registers);
    - One output: to transfer computation results to registers.

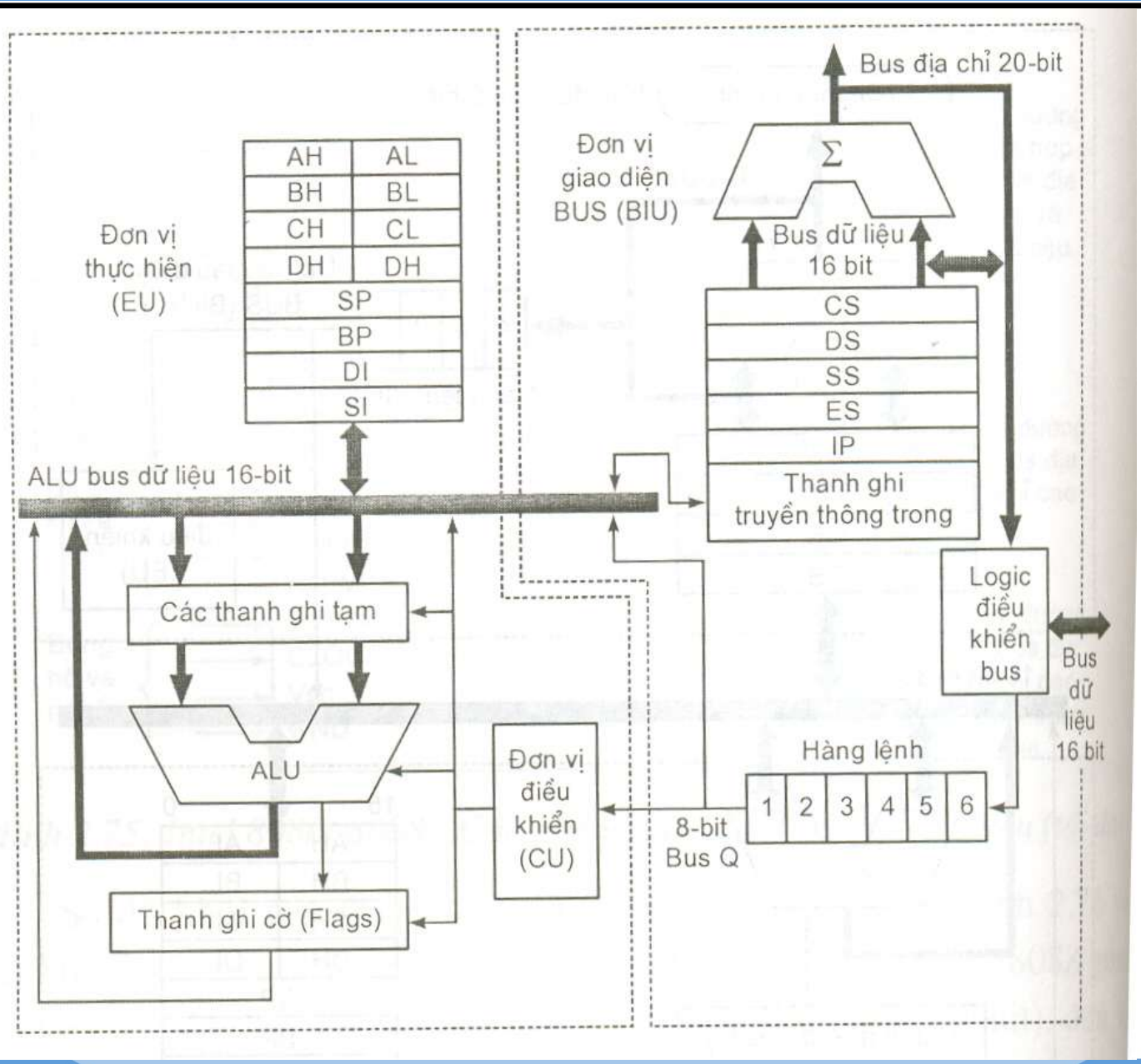
Internal Bus



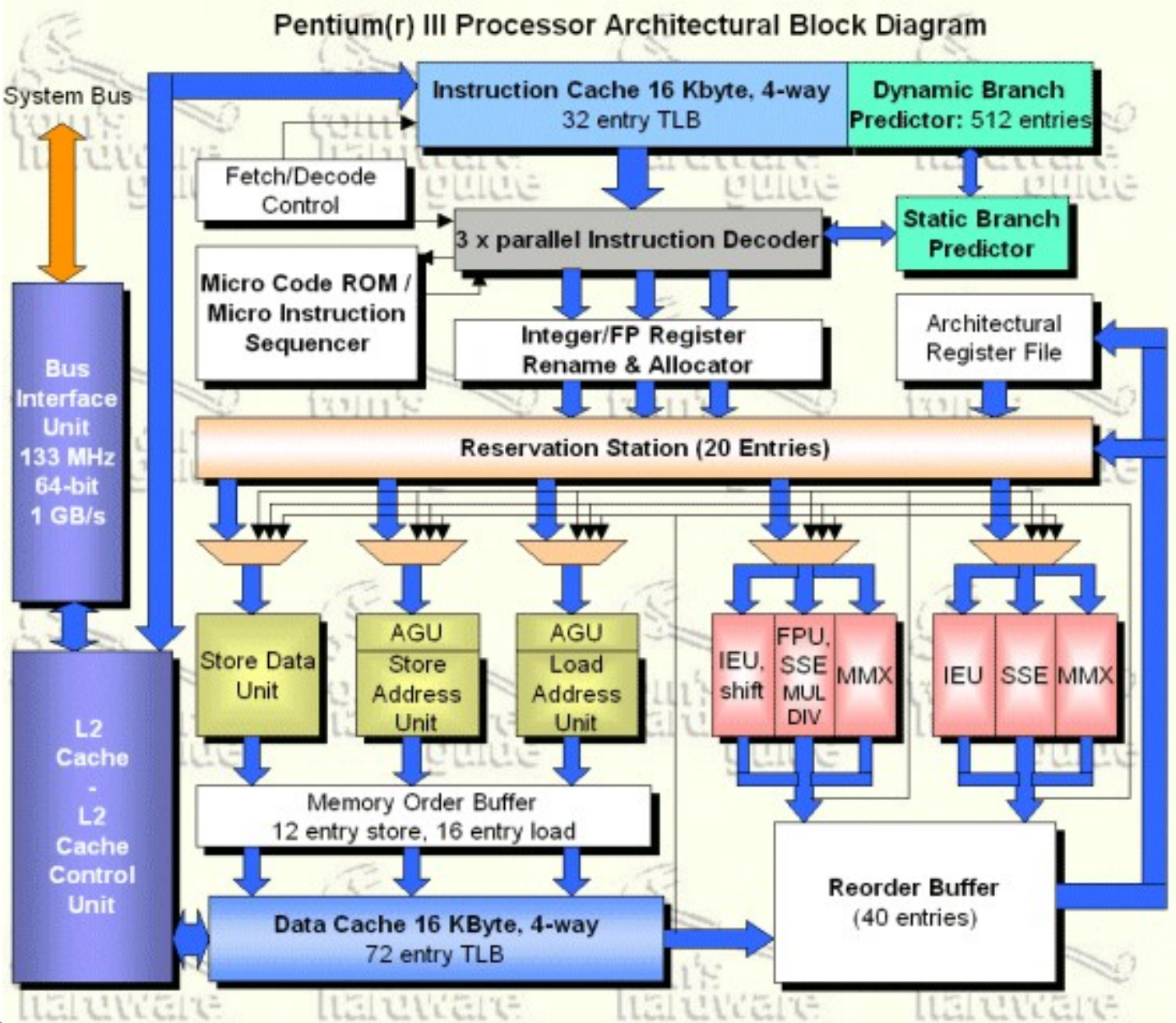
Internal Bus

- The internal bus is the communication channel for all components within the CPU.
- It supports bidirectional communication.
- The internal bus has an interface for exchanging information with the external bus.
- The internal bus has a larger bandwidth and faster speed compared to the external bus.
- The internal bus is the communication channel for all components within the CPU.
- It supports bidirectional communication.
- The internal bus has an interface for exchanging information with the external bus.
- The internal bus has a larger bandwidth and faster speed compared to the external bus.

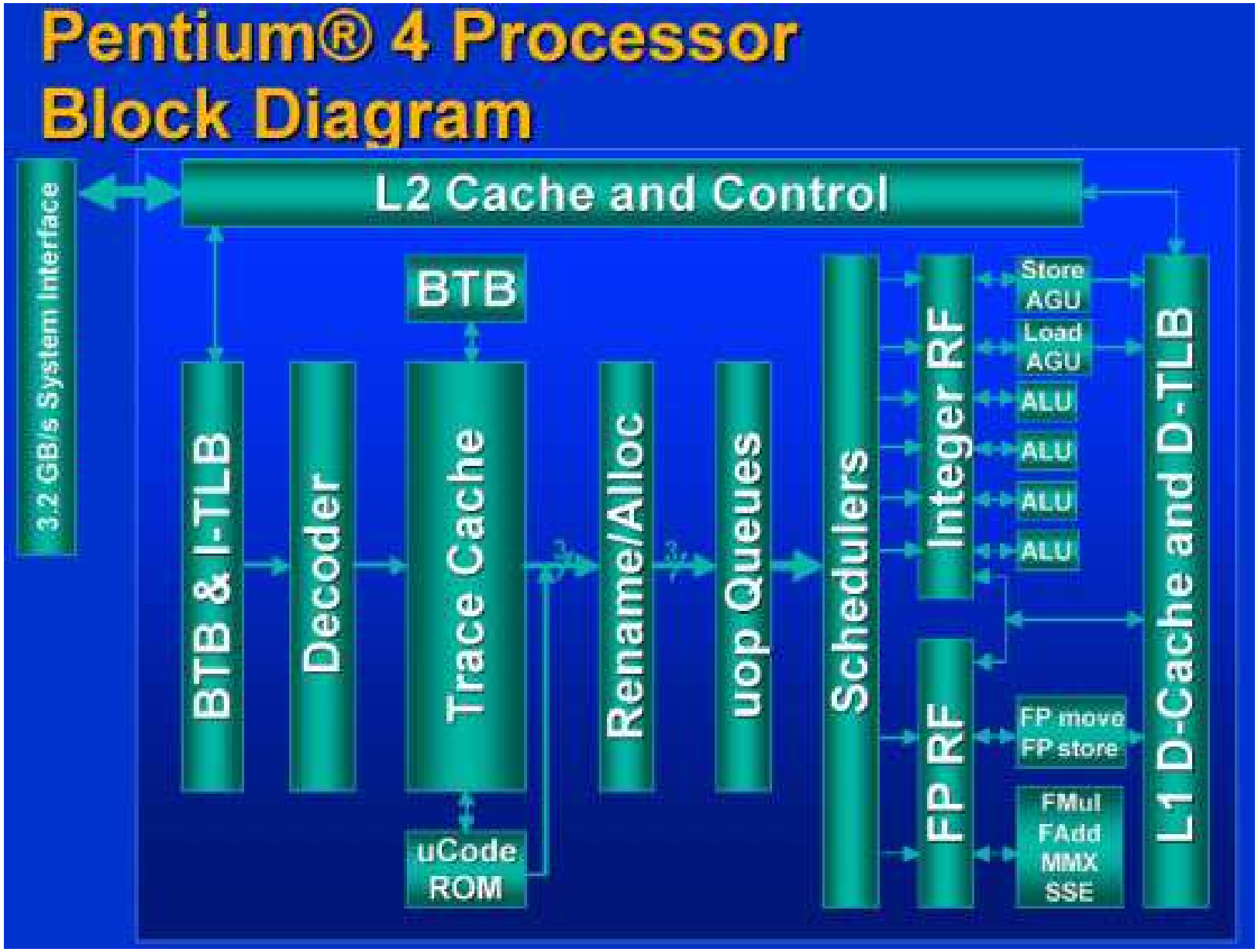
Intel 8086


Bus địa chỉ 20-bit
Đơn vị giao diện BUS (BIU)
Σ
Bus dữ liệu 16 bit
Đơn vị thực hiện (EU)
AH
AL
BH
BL
CH
CL
DH
DH
SP
BP
DI
SI
CS
DS
SS
ES
IP
Thanh ghi truyền thông trong
ALU bus dữ liệu 16-bit
Các thanh ghi tạm
Logic điều khiển bus
Bus dữ liệu 16 bit
ALU
Đơn vị điều khiển (CU)
Hàng lệnh
1
2
3
4
5
6
8-bit Bus Q
Thanh ghi cờ (Flags)

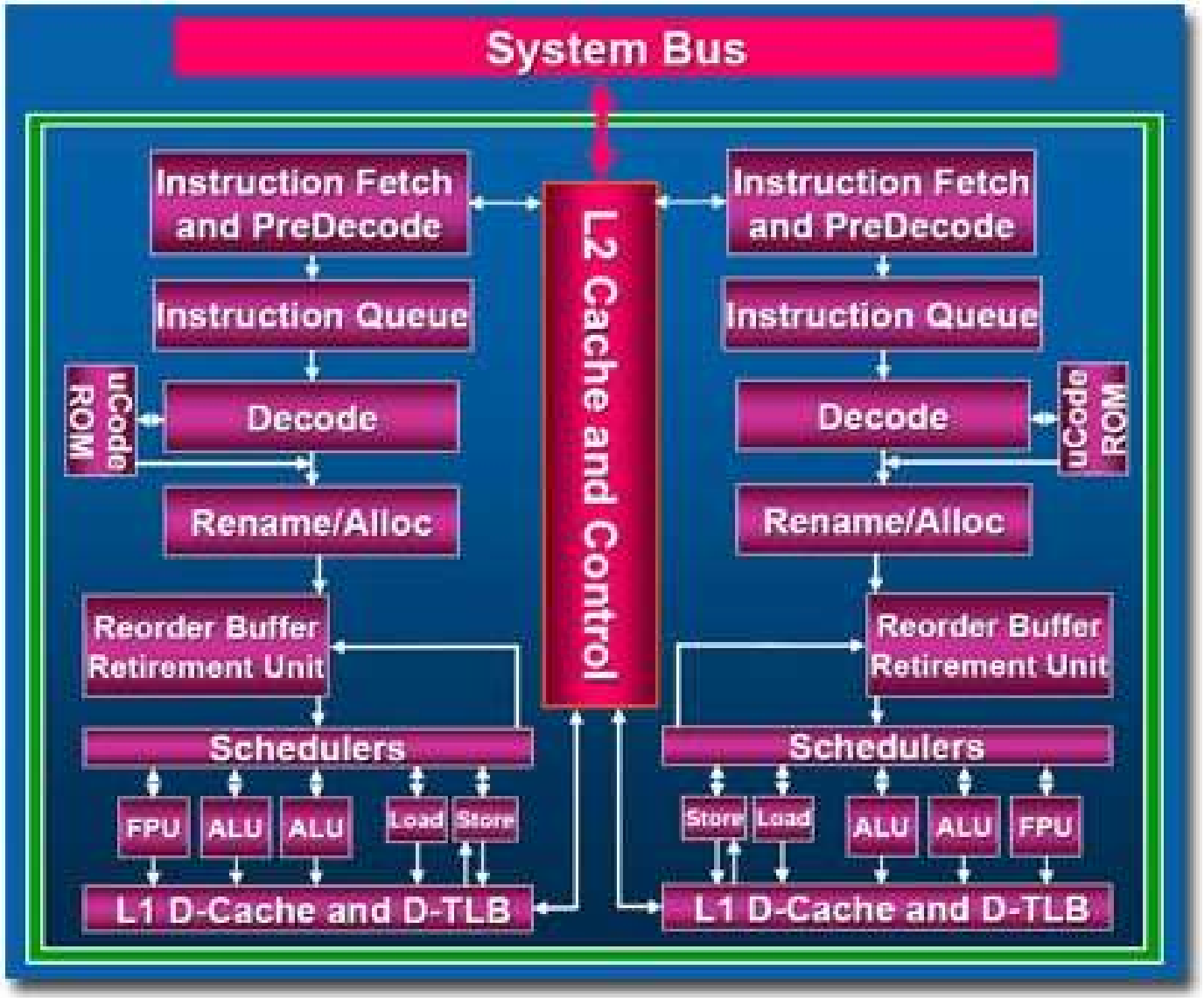
Intel Pen III



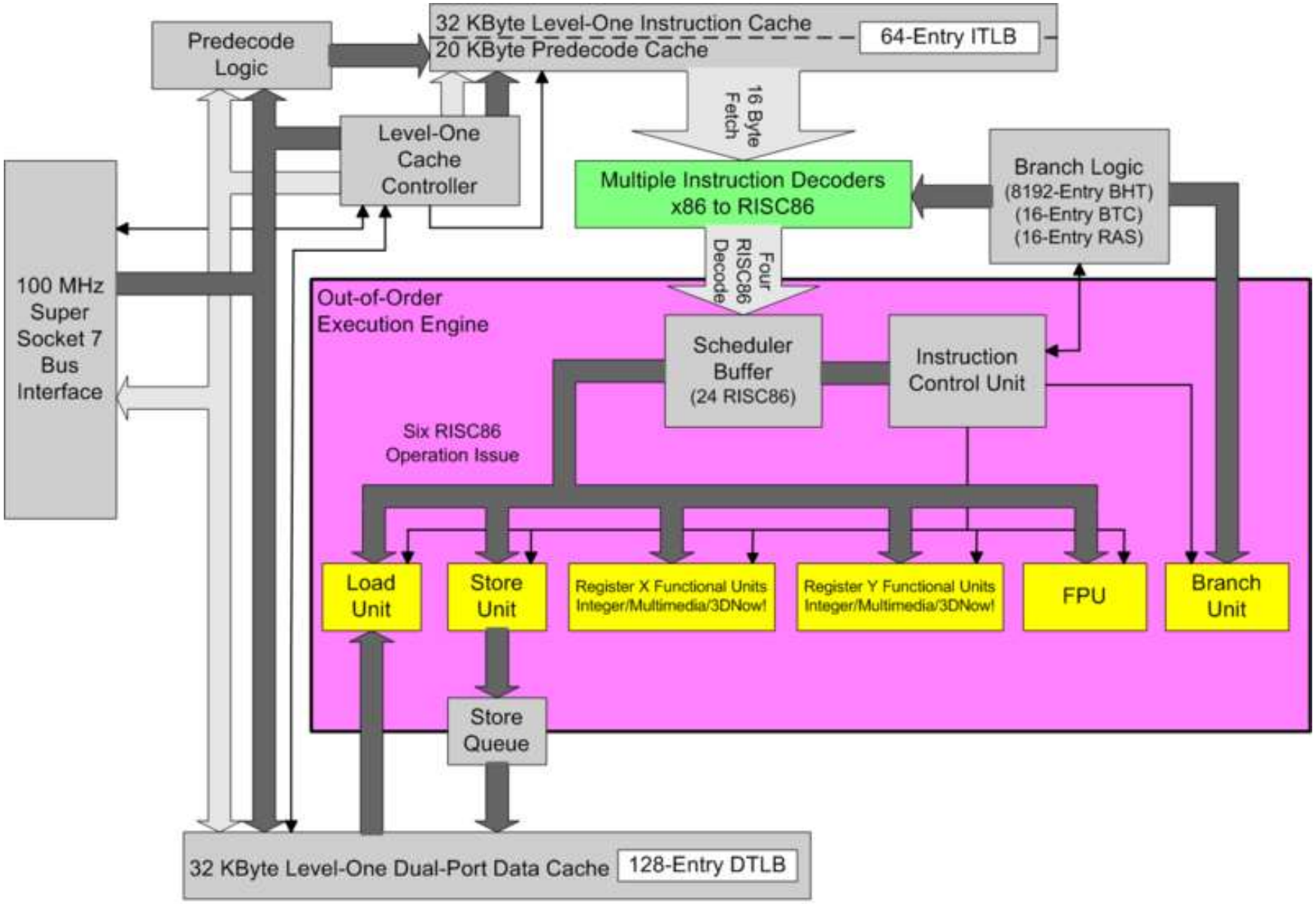
Intel Pen IV


Pentium® 4 Processor
Block Diagram
3.2 GB/s System Interface
L2 Cache and Control
BTB
BTB & I-TLB
Decoder
Trace Cache
uCode ROM
Rename/Alloc
uop Queues
Schedulers
Integer RF
FP RF
Store AGU
Load AGU
ALU
ALU
ALU
ALU
FP move
FP store
FMul
FAdd
MMX
SSE
L1 D-Cache and D-TLB

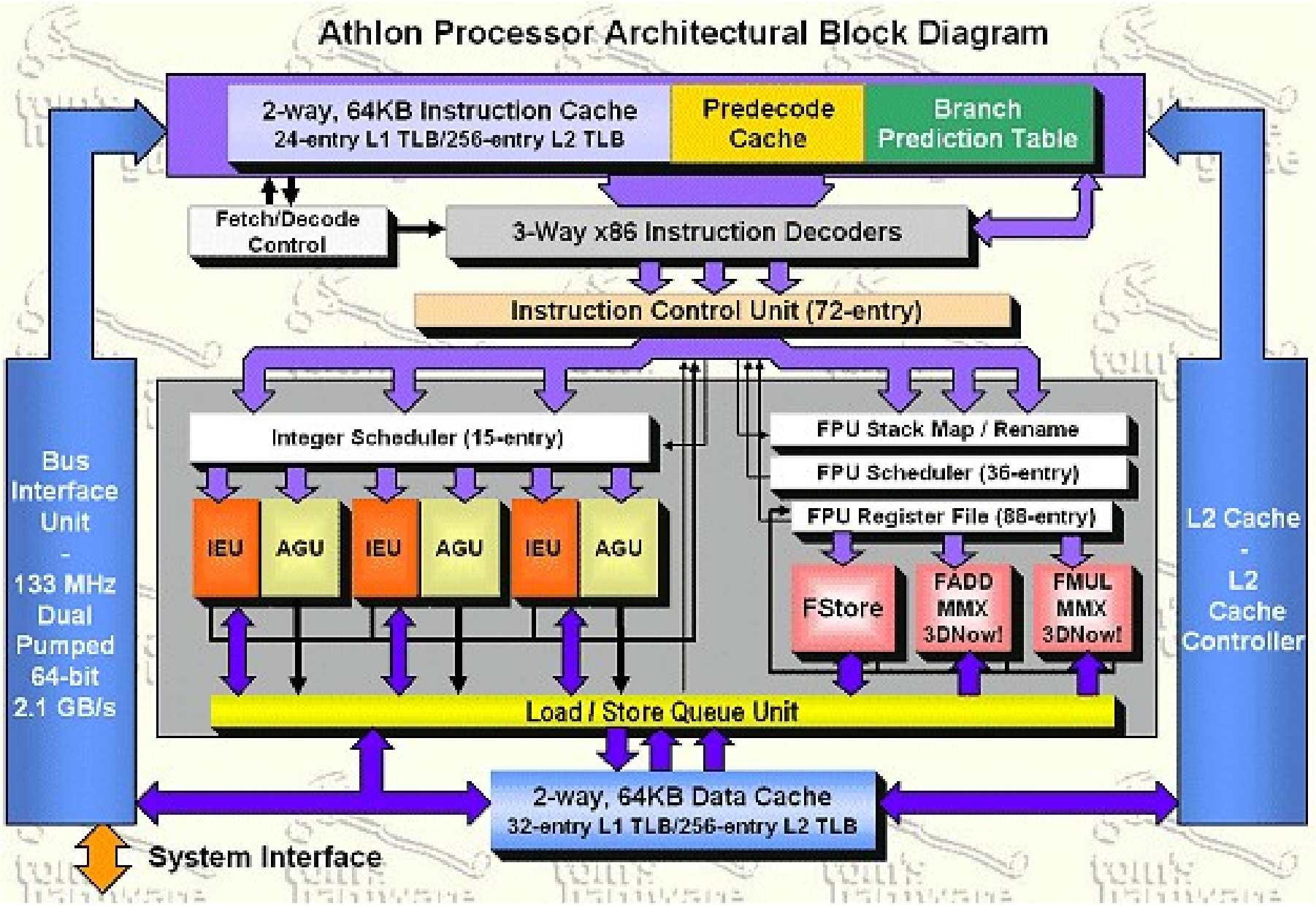
Intel Core 2 Duo



AMD K6-2


Predecode Logic
32 KByte Level-One Instruction Cache
20 KByte Predecode Cache
64-Entry ITLB
16 Byte Fetch
Level-One Cache Controller
Multiple Instruction Decoders x86 to RISC86
Branch Logic (8192-Entry BHT) (16-Entry BTC) (16-Entry RAS)
Four RISC86 Decode
100 MHz Super Socket 7 Bus Interface
Out-of-Order Execution Engine
Scheduler Buffer (24 RISC86)
Instruction Control Unit
Six RISC86 Operation Issue
Load Unit
Store Unit
Register X Functional Units Integer/Multimedia/3DNow!
Register Y Functional Units Integer/Multimedia/3DNow!
FPU
Branch Unit
Store Queue
32 KByte Level-One Dual-Port Data Cache
128-Entry DTLB

AMD Athlon XP



Review Questions

1. Draw a general block diagram of the CPU components. Explain the functions of the A, MAR, and MBR registers.
2. Explain the functions of the control unit (CU) and the arithmetic logic unit (ALU).
3. Explain the bits in the flag register; provide illustrative examples.
4. Explain the arithmetic logic unit (ALU); state the function of the program counter (PC); provide illustrative examples.
5. Explain the control unit (CU); state the function of the program counter (PC); provide illustrative examples.

Review Questions

6. Draw a general diagram of the central processing unit (CPU); state the functions of the components during the CPU's processing of the ADD R1, R2 instructions.
7. State the general block diagram and instruction processing cycle of the CPU.
8. What is the function of the flag register or status register of the microprocessor? Explain the meaning of the carry flag (C), zero flag (Z), and sign flag (S) and provide illustrative examples.

Instruction set

- A computer program is made up of a series of individual operations called instructions/computer commands. A computer instruction is a binary word assigned a specific task, telling the computer what to do.
  - Program instructions are stored in memory.
  - Instructions are read from memory into the CPU for decoding and execution.
  - Each instruction has a specific function.

8086 INSTRUCTION SET

OPCODE		DESCRIPTION
AAA		ASCII adjust addition
AAD		ASCII adjust division
AAM		ASCII adjust multiply
AAS		ASCII adjust subtraction
ADC	dt,sc	Add with carry
ADD	dt,sc	Add
AND	dt,sc	Logical AND
CALL	proc	Call a procedure
CBW		Convert byte to word
CLC		Clear carry flag
CDL		Clear direction flag
CLI		Clear interrupt flag
CMC		Complement carry flag
CMP	dt,sc	Compare
CMPS	[dt,sc]	Compare string
CMPSB	"	" bytes
CMPSW	"	" words
CWD		Convert word to double word

JNAE	slabel	Jump if not above or equal
JNB	slabel	Jump if not below
JNBE	slabel	Jump if below or equal
JNC	slabel	Jump if no carry
JNE	slabel	Jump if not equal
JNG	slabel	Jump if not greater
JNGE	slabel	Jump if not greater or equal
JNL	slabel	Jump if not less
JNLE	slabel	Jump if not less or equal
JNZ	slabel	Jump if not zero
JNO	slabel	Jump if not overflow
JNP	slabel	Jump if not parity
JNS	slabel	Jump if not sign
JO	slabel	Jump if overflow
JPO	slabel	Jump if parity odd
JP	slabel	Jump if parity
JPE	slabel	Jump if parity even
JS	slabel	Jump if sign
JZ	slabel	Jump if zero

PUSHF		Push flags onto stack
RCL	dt,cnt	Rotate left through carry
RCR	dt,cnt	Rotate right through carry
REP		Repeat string operation
REPE		Repeat while equal
REPZ		Repeat while zero
REPNE		Repeat while not equal
REPNZ		Repeat while not zero
RET	[pop]	Return from procedure
ROL	dt,cnt	Rotate left
ROR	dt,cnt	Rotate right
SAHF		Store AH into flags
SAL	dt,cnt	Shift arithmetic left
SHL	dt,cnt	Shift logical left
SAR	dt,cnt	Shift arithmetic right
SBB	dt,sc	Subtract with borrow
SCAS	[dt]	Scan string
SCASB	"	" byte
SCASW	"	" word

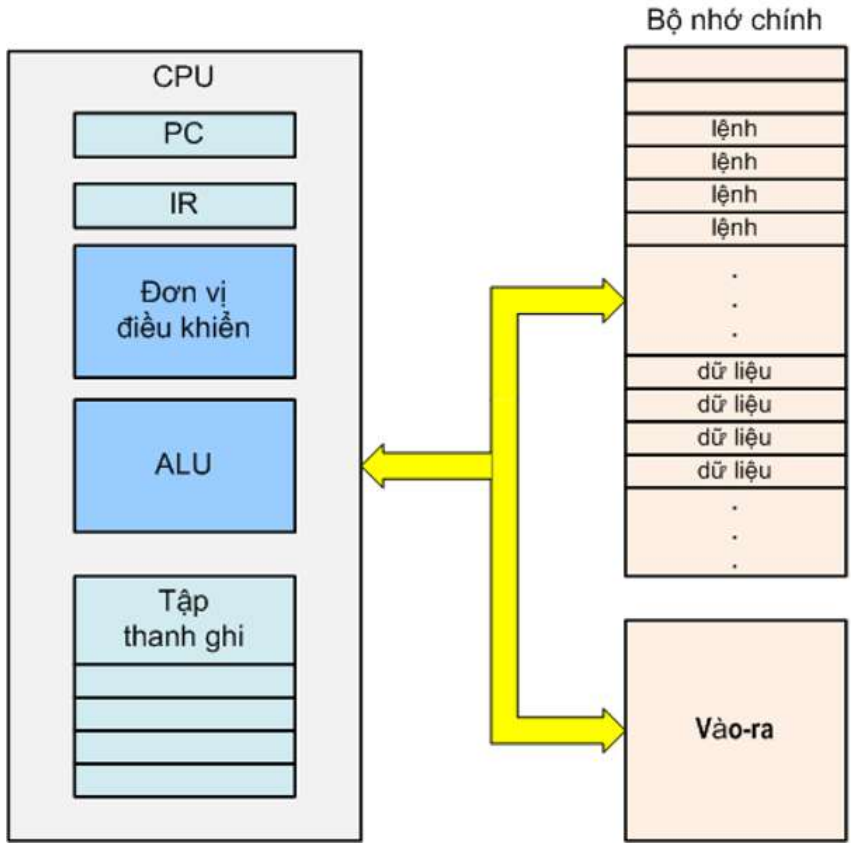
Instruction set

- Each processor has a defined instruction set:
  - The instruction set typically contains dozens to hundreds of instructions.
  - Each instruction is a sequence of binary numbers that the processor understands to perform a specific operation.
  - Instructions are described using mnemonic text symbols: these are assembly language instructions.

    For example: ADD A, 1.
- The instruction set consists of many instructions, divided into functional groups:
  - Data movement
  - Computational
  - Conditioning and branching
  - Other instructions

Instruction set

- The process of executing an instruction is divided into phases or stages. Each instruction can be executed in four stages:
  - Read instruction IF (Instruction Fetch): the instruction is read from memory into the CPU
  - Decode instruction ID (Instruction Decode): the CPU decodes the instruction
  - Execute instruction IE (Instruction Execution): the CPU executes the instruction
  - Write back WB (Write Back): the result (if any) is written to a register/memory



Instruction set

- An instruction cycle is the time it takes for a CPU to execute an instruction, from the moment the CPU allocates the memory address for the instruction until it completes the execution of that instruction.
  - An instruction execution cycle consists of several instruction execution phases.
  - An instruction execution phase can consist of several machine cycles.
  - A machine cycle can consist of several clock cycles.
  - The clock cycle (or clock pulse) is the most basic and smallest unit of time in a microprocessor.
- An instruction execution cycle can include the following components:
  - Instruction read cycle
  - Memory read/write cycle
  - Peripheral device read/write cycle (I/O read/write)
  - Bus idle cycle

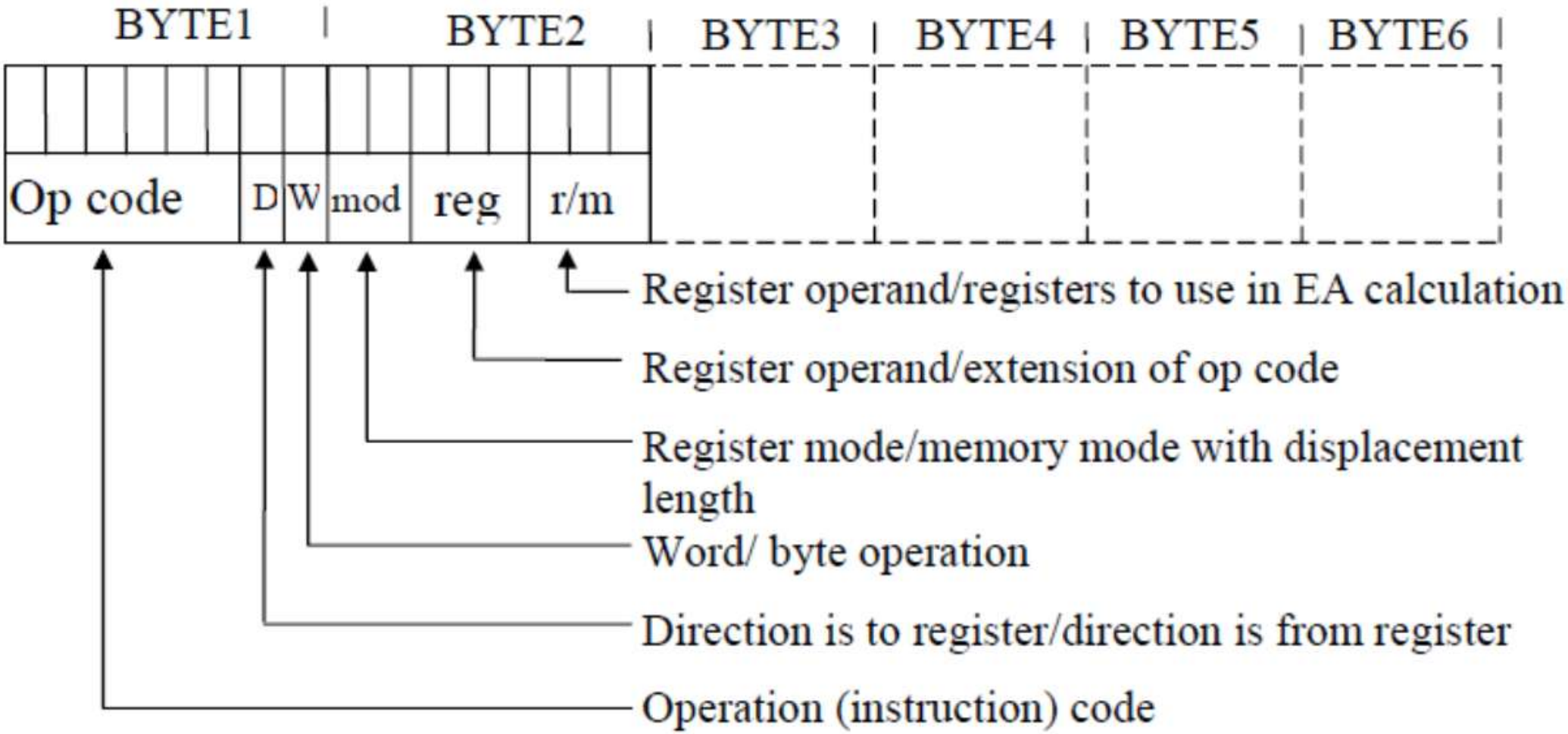
Form and component of instructions

- The general form of a computer instruction consists of two main parts:
  - Operation code: encodes the operation that the processor must
  - Addresses of operands: indicate where the operands that the operation will affect are located:
    - Source operand: input data of the operation
    - Destination operand: output data of the operation

Mã lệnh	Địa chỉ của các toán hạng	
Opcode	Addresses of Operands	
Opcode	Des addr.	Source addr.

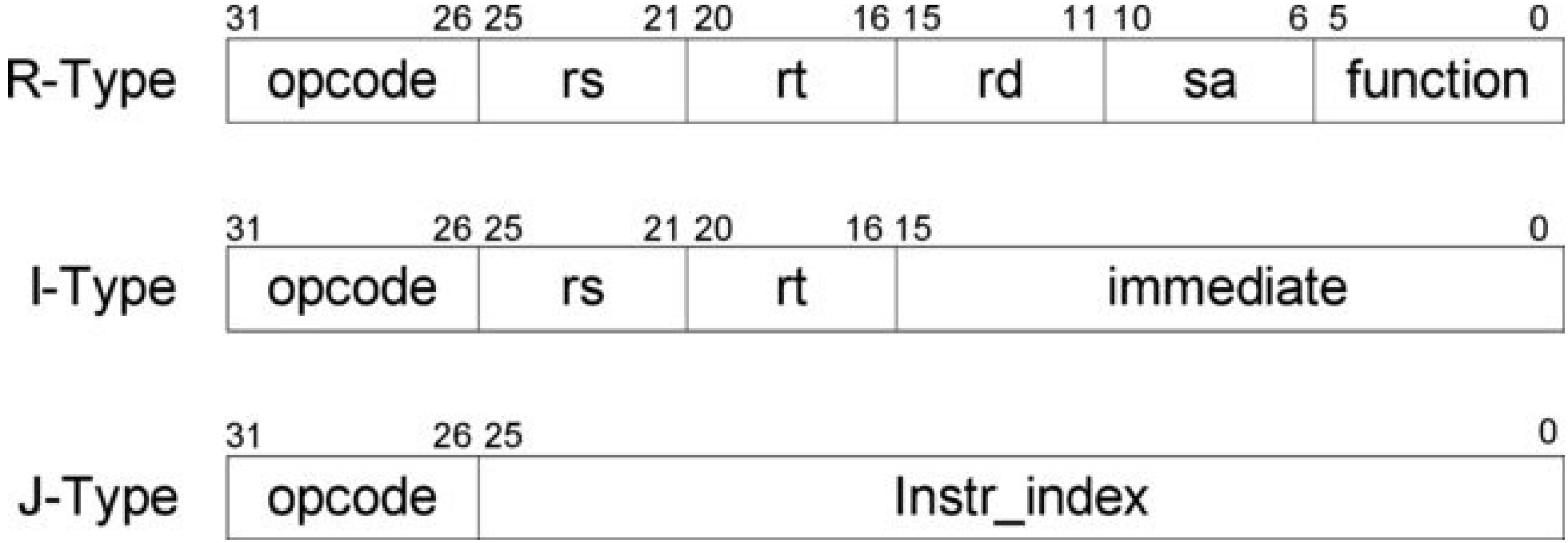
Form and component of instructions

- Example: 8086



Form and component of instructions

- Example: MIPS



Address types and operands

- Based on the number of operands, instructions are divided into 5 types:
  - 3-address operands;
  - 2-address operands;
  - 1.5-address operands;
  - 1-address operands
  - 0-address operands.

Address types and operands

- Some conventions for CPU instructions:
  - Ri: CPU registers
  - Racc: Accumulation/Sum register
  - (Ri): Content of memory location whose address is stored in register Ri
  - (100): Content of memory location whose address is stored in memory location 100
  - A, B, C, 1000 are memory location addresses
  - M[100]: Reference to the content of memory location 100
  - #100: Constant 100

Address types and operands

The 3-address operand

- Format: OPCODE Addr1, Addr2, Addr3.
- Each address Addr1, Addr2, Addr3 refers to a memory location or a register.
- For example:
  - ADD R1, R2, R3;
    - –R3 + R2–> R1; R2 plus R3 then the result is put into R1
    - –Ri are CPU registers
  - ADD A, B, C;
    - –M[B] + M[C] –> M[A];
    - –A, B, C are memory locations

Address types and operands

- Two-address operands
  - Format: OPCODE Addr1, Addr2
  - Each address Addr1, Addr2 refers to a memory location or a register.
  - For example:
    - ADD R1, R2;
      - –R1 + R2–> R1; R2 plus R1 then the result is put into R1
      - –Ri are CPU registers
    - ADD A, B;
      - –M[B] + M[A] –> M[A]
      - –A, B are memory locations

Address types and operands

❖Operand 1 Address

- Format: OPCODE Addr
- Each address Addr refers to a memory location or a register.
- This format uses Racc (accumulator register) by default for the second address.
- Example:
  - ADD R1;
    - –R1 + Racc –> Racc; Racc plus R1 then the result is put into Racc
    - –Ri which are CPU registers
  - ADD A;
    - –M[A] + Racc –> Racc
    - –A is the location in memory

Address types and operands

❖Operand 1.5 Address

- Format: OPCODE Addr1, Addr2
- One address refers to a memory location and the other to a register.
- It is a hybrid of register and memory location operands.
- Example:
  - ADD R1, B;
    - –M[B] + R1–> R1; R1 plus M[B] then the result is put into R1
    - –Ri which are CPU registers
    - –B is a memory location

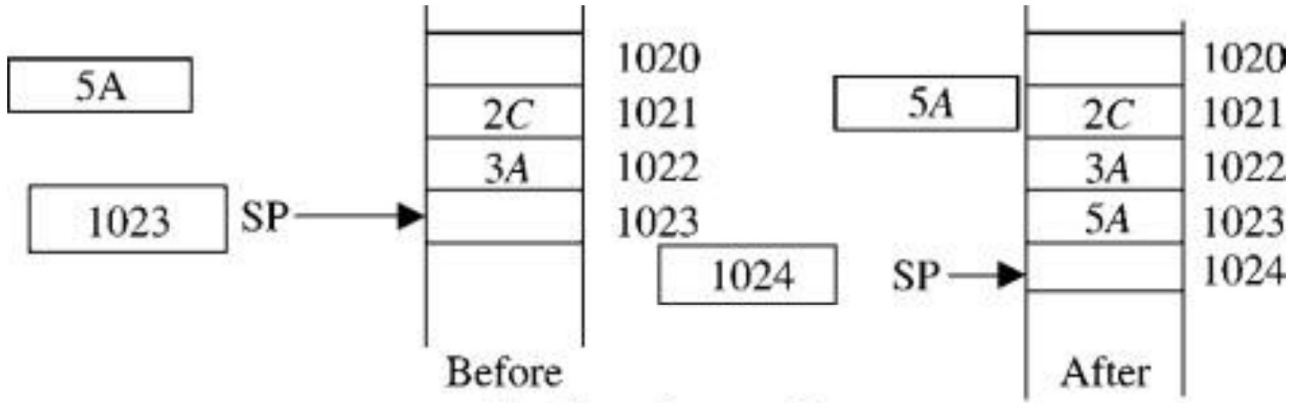
Address types and operands

- Operand 0 address
  - Format: OPCODE
  - Executed in instructions that perform stack operations
  - Examples: c = a + b
    - PUSH a
    - PUSH b
    - ADD
    - POP c

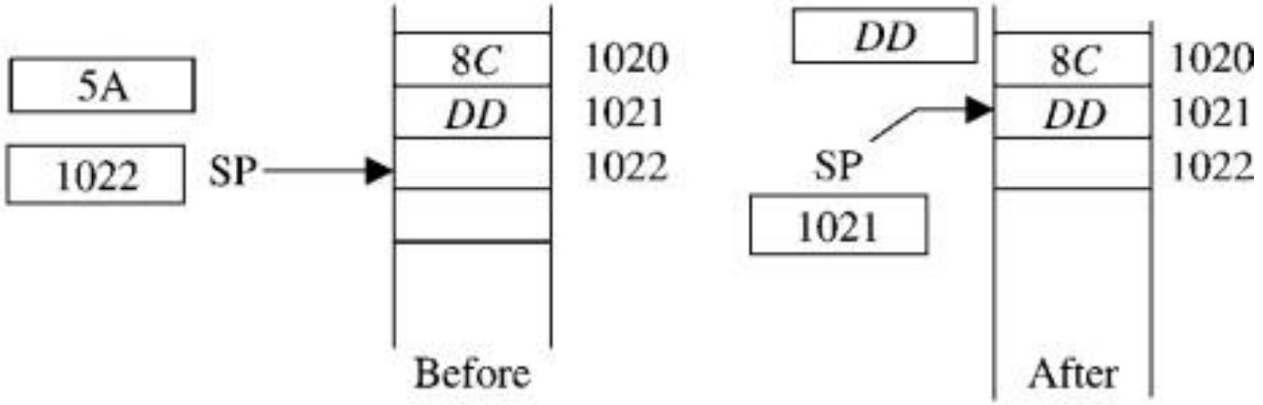
Address types and operands

❖Operand 0 address

- Examples: push & pop



Stack *push* operation



Stack *pop* operation

Addressing modes

Mã lệnh	Địa chỉ của các toán hạng	
Opcode	Addresses of Operands	
Opcode	Des addr.	Source addr.

- The operands of an instruction can be:
  - A specific value located directly within the instruction
  - The contents of a register
  - The contents of a memory location or input/output port.
- Addressing modes are the methods or ways in which the CPU organizes the operands of an instruction or how it addresses the address field of an instruction to determine where the operands are located.
  - Addressing modes allow the CPU to check the instruction format and find the operands of the instruction.
  - The number of addressing modes depends on the CPU design.

Addressing modes

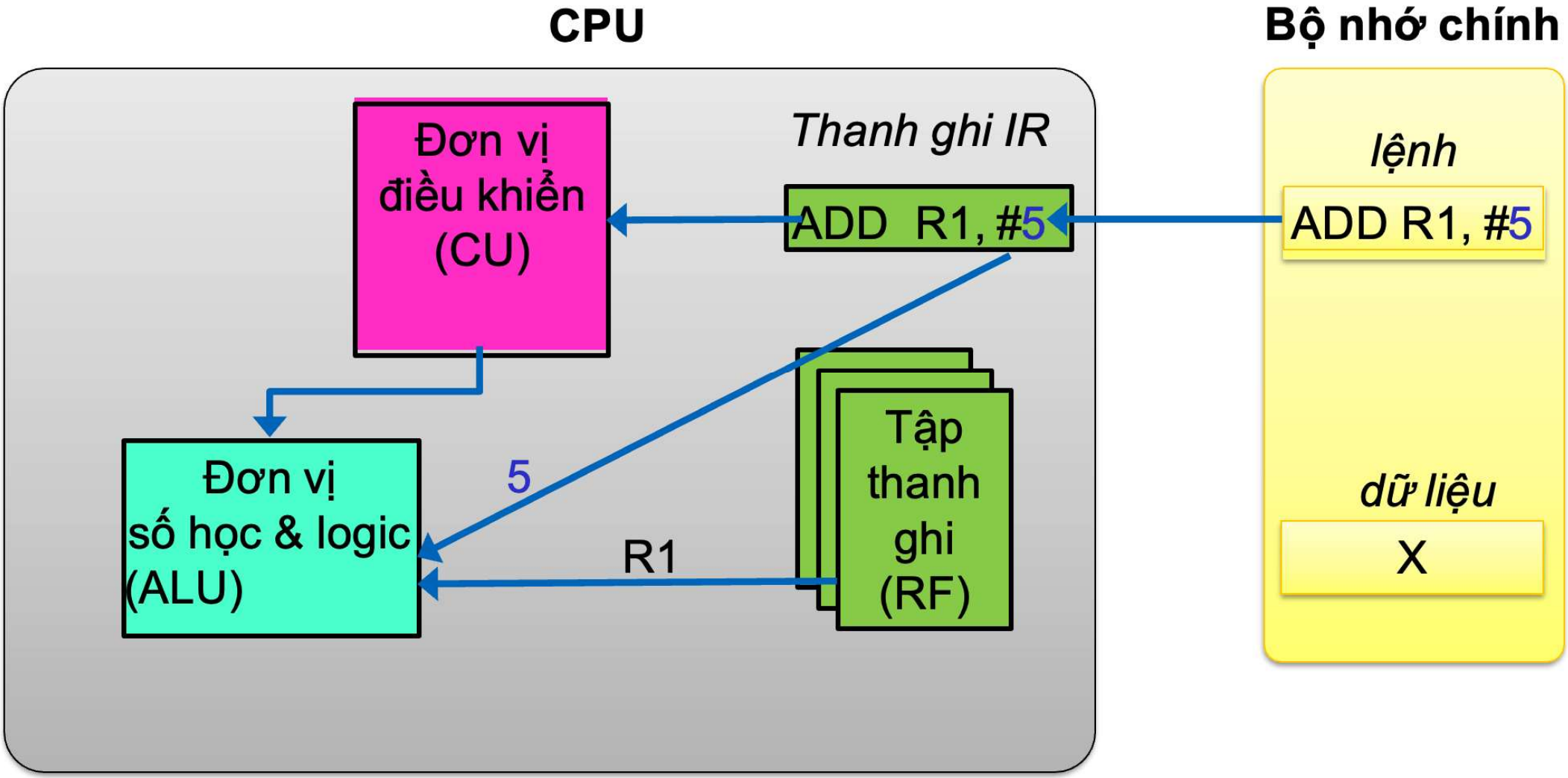
❖Addressing modes

Chế độ địa chỉ	Ý nghĩa	Ví dụ	Thực hiện
Tức thì	Giá trị của toán hạng được chứa trong lệnh	LOAD Ri, #1000	Ri ← 1000
Trực tiếp	Địa chỉ của toán hạng được chứa trong lệnh	LOAD Ri, 1000	Ri ← M[1000]
Gián tiếp thanh ghi	Giá trị của thanh ghi trong lệnh là địa chỉ bô nhớ chứa toán hạng	LOAD Ri, (Rj)	Ri ← M[Rj]
Gián tiếp bộ nhớ	Địa chỉ bộ nhớ trong lệnh chứa địa chỉ bộ nhớ của toán hạng	LOAD Ri, (1000)	Ri ← M[M[1000]]
Chỉ số	Địa chỉ của toán hạng là tổng của hằng số (trong lệnh) và giá trị của một thanh ghi chỉ số	LOAD Ri, X(Rind)	Ri ← M[X+ Rind]
Tương đối	Địa chỉ của toán hạng là tổng của hằng số và giá trị của thanh ghi con đếm chương trình	LOAD Ri, X(PC)	Ri ← M[X+ PC]

Addressing modes

❖Immediate Addressing Mode

ADD R1, #5



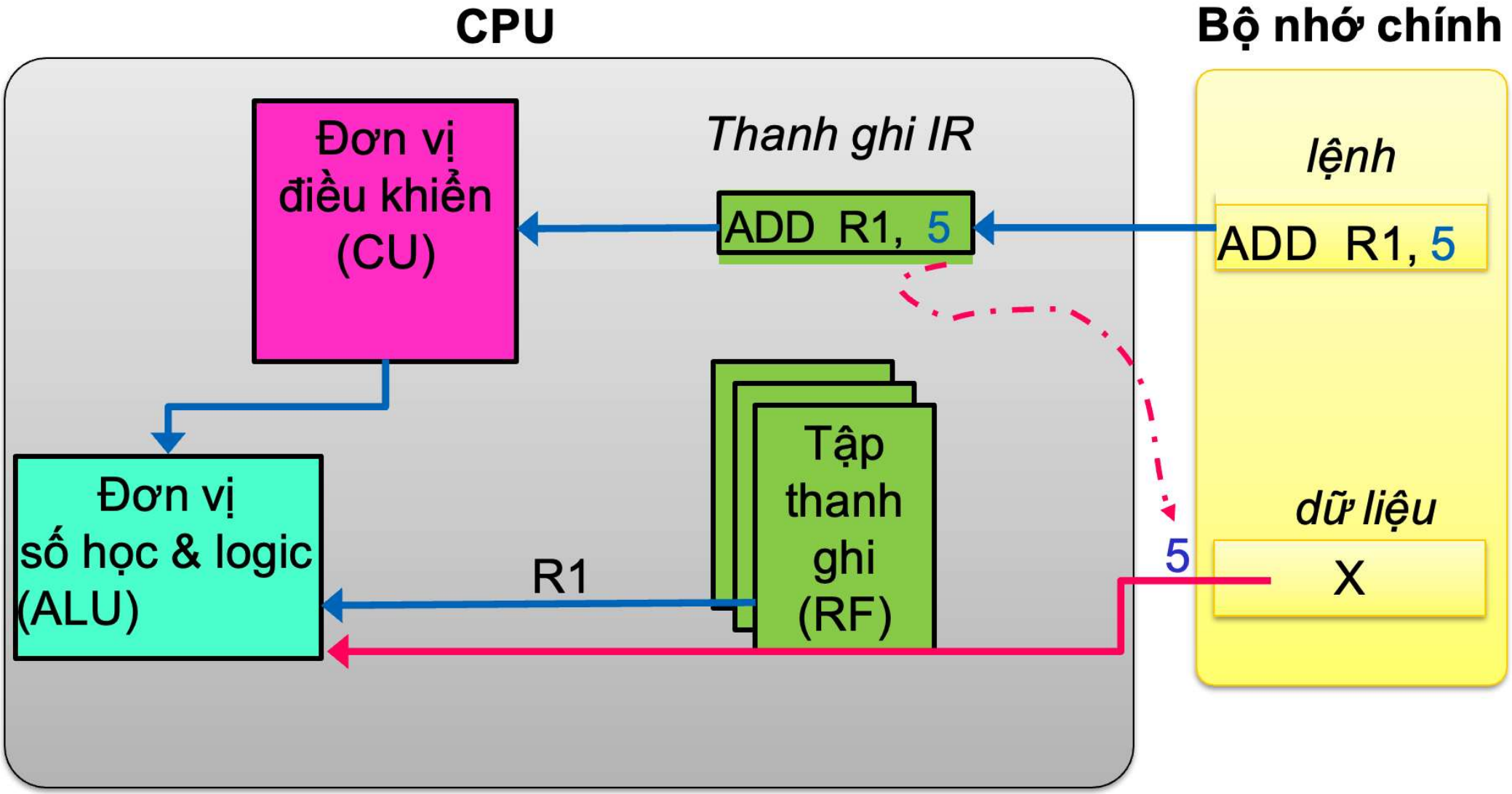
Addressing modes

❖ Instruction format in immediate addressing mode:

- The value of the source operand is available in the instruction (constant)
- The destination operand can be a register or a memory location
- Examples:
  - LOAD R1, #1000; 1000→R1 The value 1000 is loaded into register R1
  - ADD 100, #5; M[100] = M[100] + 5
  - Remarks on immediate addressing mode:
    - No memory reference
    - Very fast operand access
    - The range of operand values is limited

Addressing modes

❖Direct Addressing



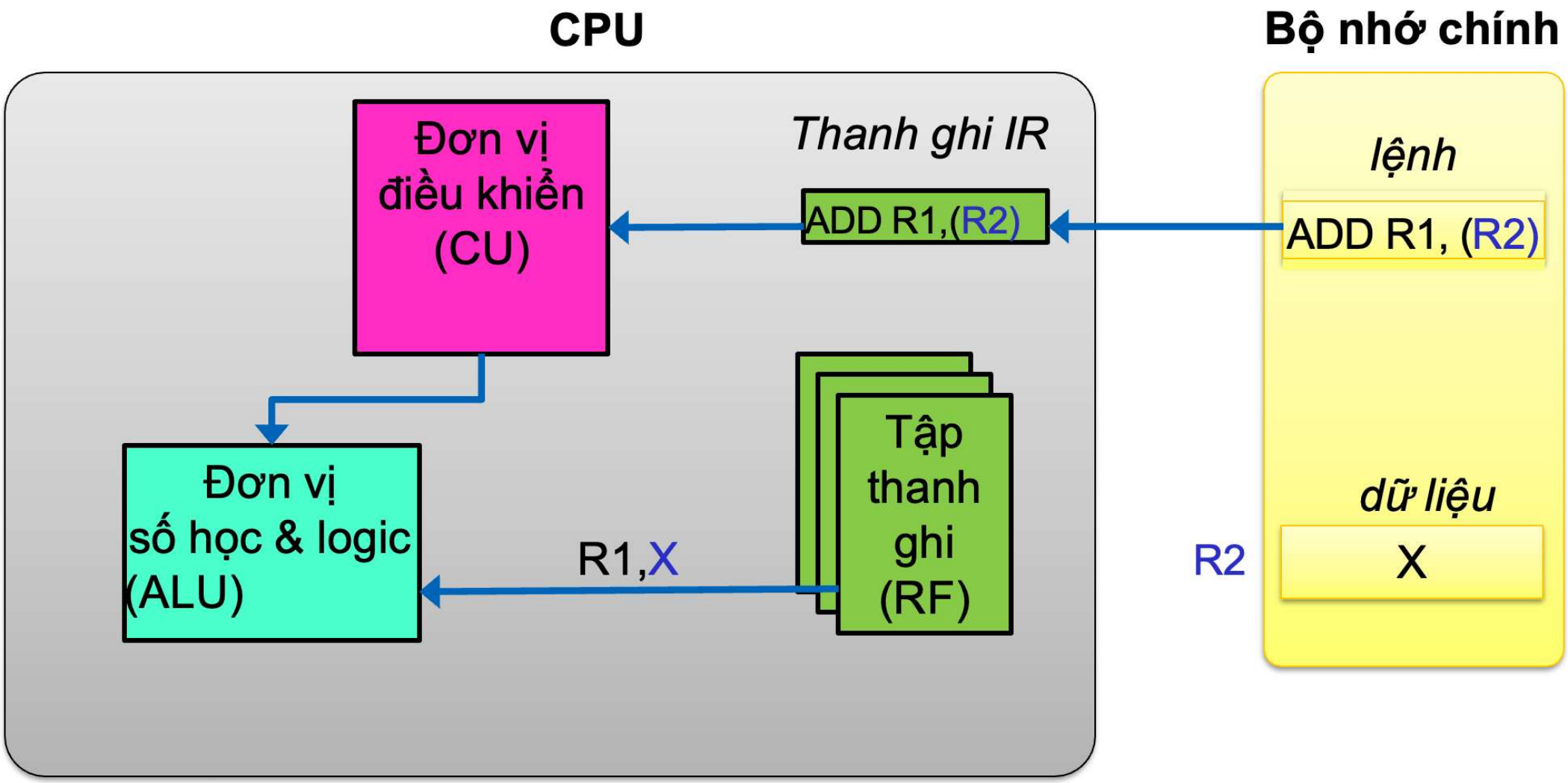
Addressing modes

❖Instruction format in direct addressing mode:

- One operand is a memory location with a direct address in the instruction.
- The other operand is a register or a memory address.
- Example:
  - LOAD R1, 1000; M[1000] → R1 The value stored in memory location 1000 is loaded into register R1.
  - ADD R1, 5; R1 = R1 + M[5] Finds the operand in memory at address 5 and adds the contents of register R1 to the contents of the memory location with address 5.
  - Remarks on direct addressing mode:
    - The CPU references memory only once to access data.

Addressing modes

❖Register Indirect



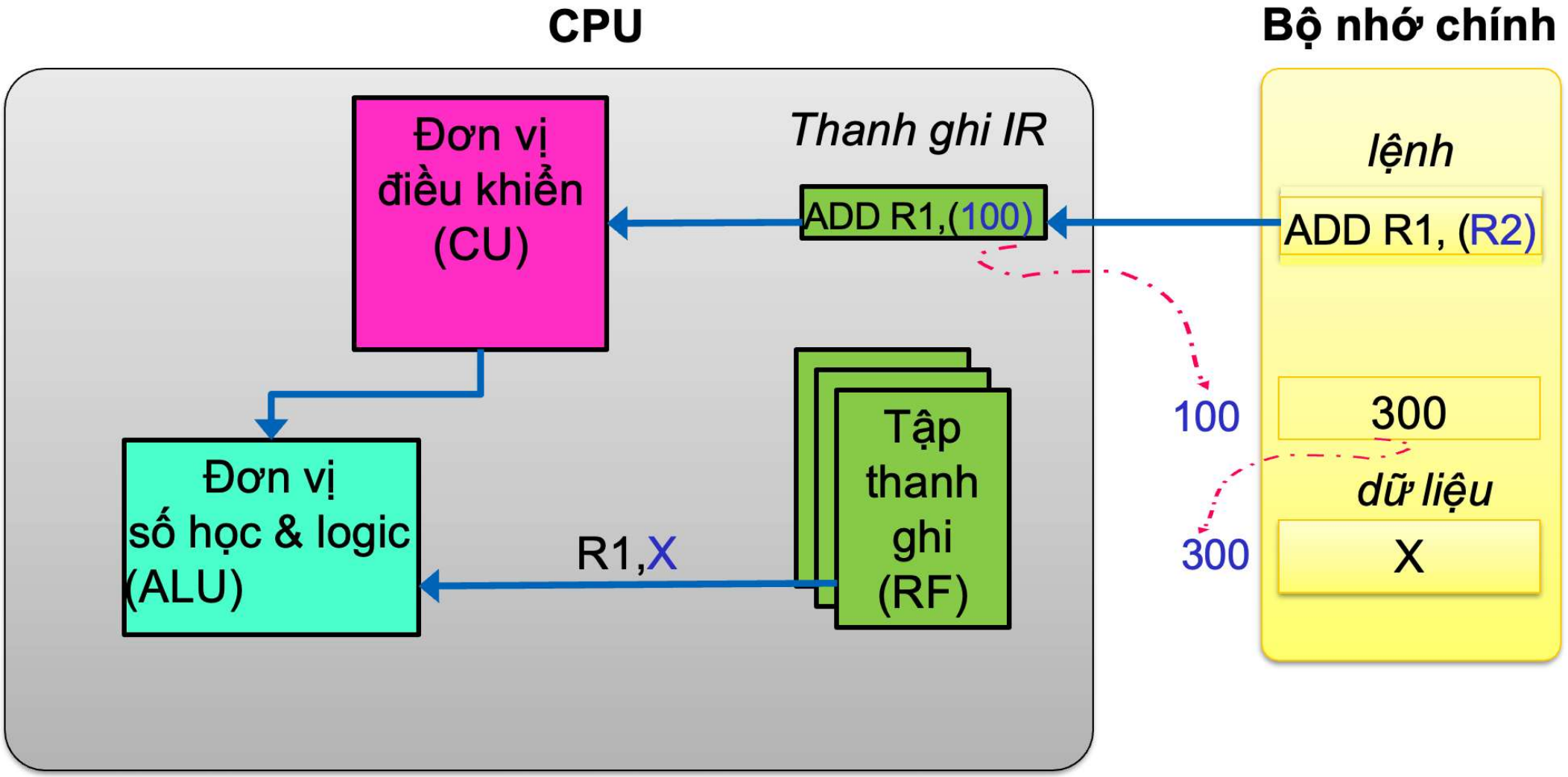
Addressing modes

❖Instruction format in register-indirect addressing mode:

- A register is used to store the address of the operand.
- Example:
  - LOAD Rj ,(Ri); M[Ri] → Rj; Load the value at the memory location whose address is stored in Ri into register Rj.
  - Remarks on register-indirect addressing mode:
    - –The CPU needs to make one memory reference for each access.

Addressing modes

❖Memory Indirect

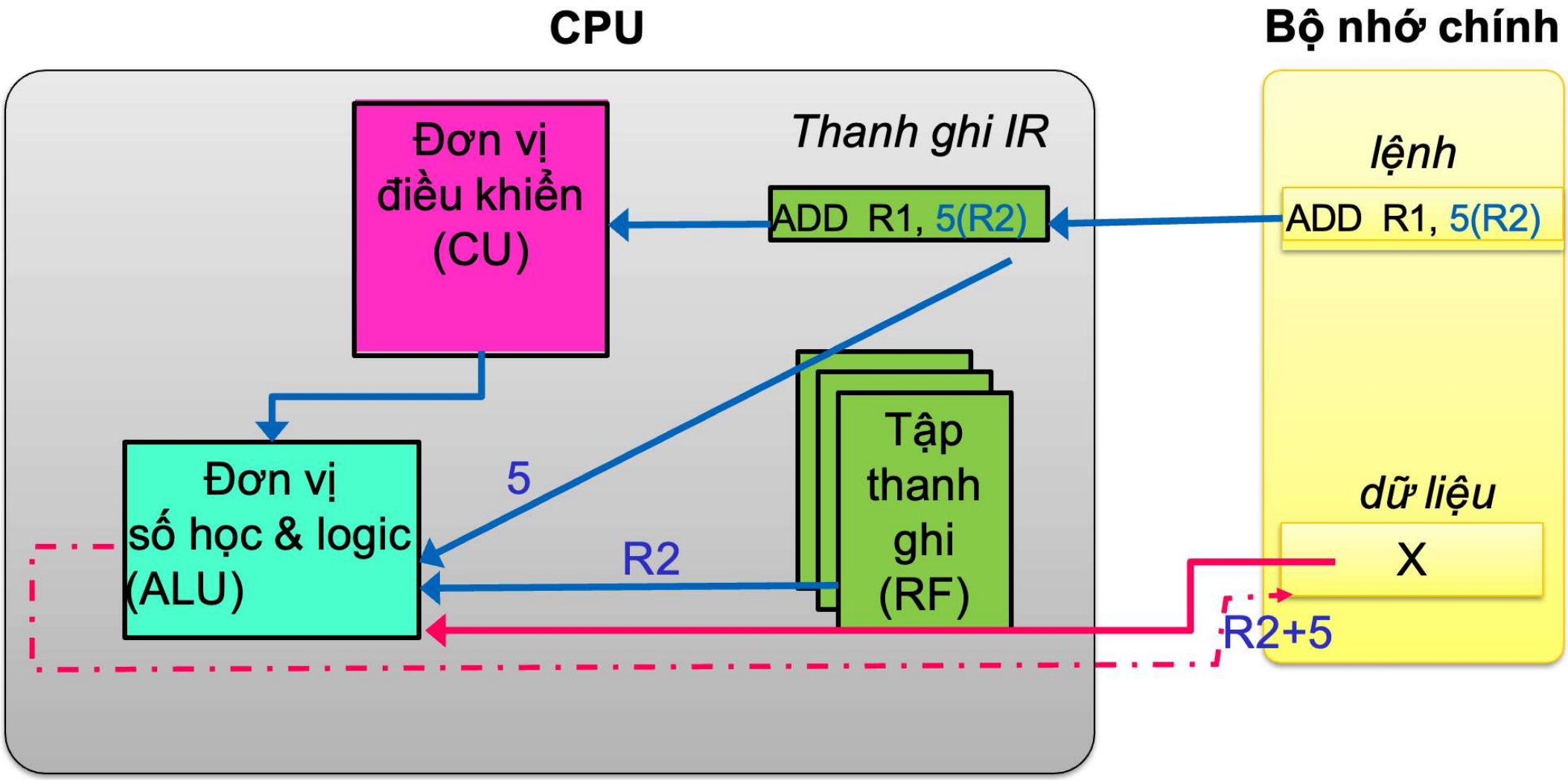


Addressing modes

❖Instruction format in memory-indirect addressing mode:

- A memory location is used to store the address of the operand
- Example:
  - LOAD Rj,v(100); M[M[100]] → Rj; Load the value at the memory location whose address is stored in memory cell 100 into register Rj
  - Remarks on memory-indirect addressing mode:
    - –The CPU references memory twice in each access.

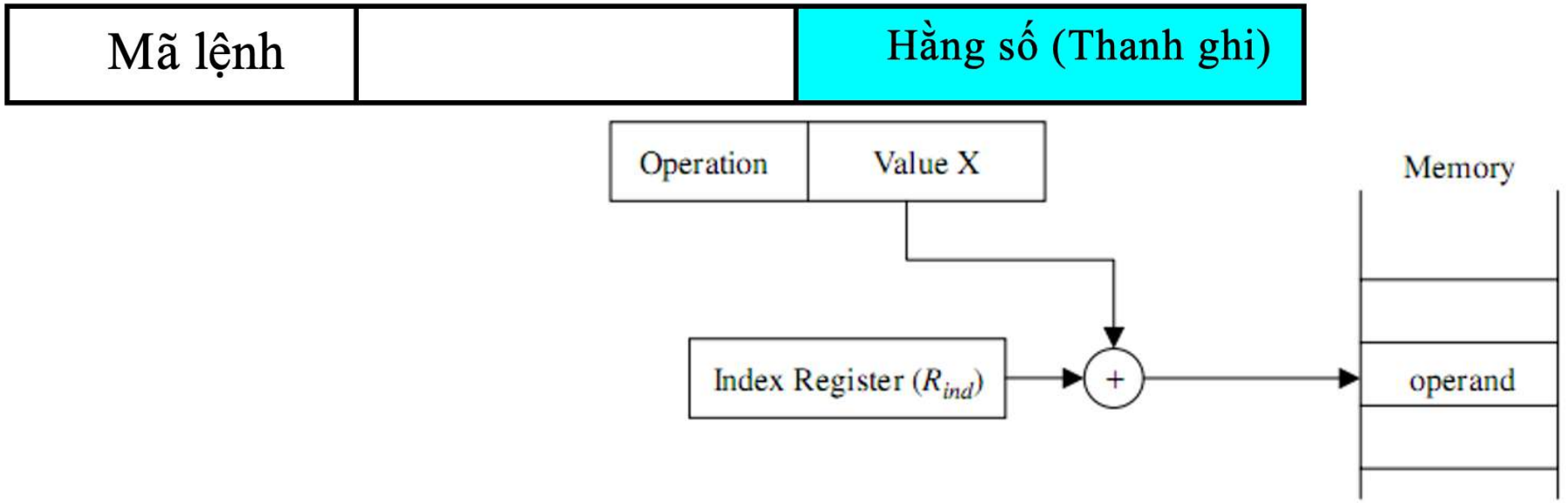
❖Indexed Addressing



Addressing modes

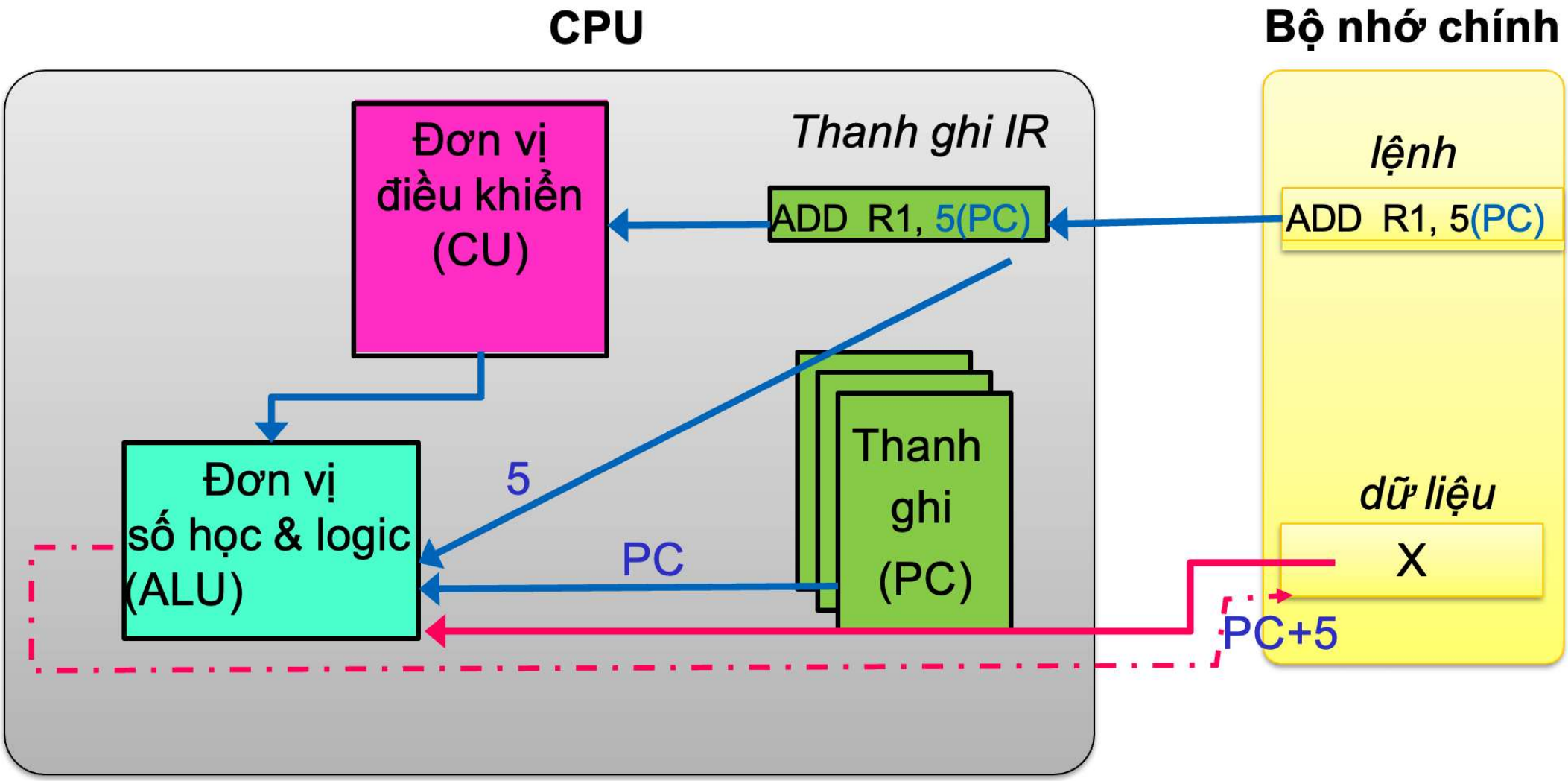
❖ Instruction format in indexed addressing mode:

- To specify the operand, the address field contains two parts: Register name and Constant (offset)
- Operand address = register content + constant
- Example:
  - LOAD Ri, X(Rind); M[X+Rind] → Ri; Load the value at the memory location whose address is stored in the Rind register plus the constant X into the R register



Addressing modes

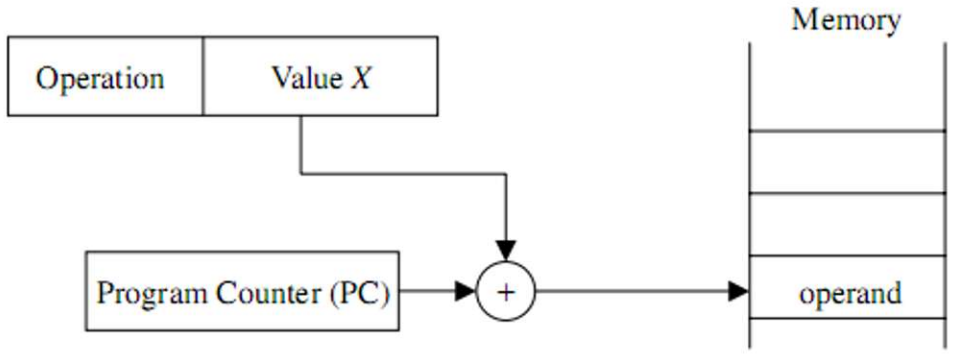
❖Relative Addressing



Addressing modes

❖Relative Addressing Mode Instruction format in relative addressing mode:

- To specify the operand, the address field contains two parts: PC register and constant (offset)
- Address of the operand = contents of PC register + constant
- Example: LOAD Ri, X(PC); M[PC+X] –> Ri; Load the value at the memory location whose address is stored in the PC register plus the constant X into the R register



Common Instructions

Depending on the CPU design, the CPU instruction set can have a very different number of instructions, including:

- Data transport instructions
- Arithmetic and logic instructions
- Program control instructions
- Input & output instructions

Common Instructions

- Data transfer instructions: Transferring data between computer memory locations
  - Between CPU registers
    - MOVE Ri, Rj ; Rj → Ri
  - Between a CPU register and a memory location
    - MOVE Ri, 1000 ; M[1000] → R
  - Between memory locations
  - MOVE 1000, (Rj) ; M[Rj] –> M[1000]
  - Some common data transfer instructions:
    - MOVE: Transfers data between registers and memory locations
    - LOAD: Loads the contents of a memory location into a register
    - STORE: Stores the contents of a register in a memory location
    - PUSH/POP: Pushes/pulls data onto/off the stack

Common Instructions

- Arithmetic and Logical Instructions: Performs arithmetic and logical operations between registers and memory contents.
  - Examples:
    - ADD R1, R2, R3; R2 + R3 → R1;
    - SUBSTRACT R1, R2, R3; R2 – R3 → R1.
  - Some common mathematical instructions:
    - ADD: adds two operands
    - SUBSTRACT: subtracts two operands
    - MULTIPLY: multiplies two operands
    - DIVIDE: divides
    - INCREMENT: increases by 1
    - DECREMENT: decreases by 1

Common Instructions

Arithmetic and Logical Commands

- Common logical commands:
  - NOT: negation
  - AND: and
  - OR: or
  - XOR: or exclusion
  - COMPARE: comparison
  - SHIFT: shift
  - ROTATE: rotation

Common Instructions

❖Control/Sequential Instructions

- Used to change the order in which instructions are executed:
  - Conditional branching/jump instructions
  - Unconditional branching/jump instructions
  - CALL and RETURN: call execution and return from a subroutine
  - A common characteristic of these instructions is that their execution changes the PC value
  - Uses ALU flags to define conditions

Common Instructions

❖Control/Sequential Instructions

- Some common data transfer instructions:
  - BRANCH – IF – CONDITION: moves to execute the instruction at the new address if the condition is true
  - JUMP: moves to execute the instruction at the new address
  - BRANCH-IF-EQUAL: moves to execute the instruction at the new address if the result of the operation equals 0
  - BRANCH-IF-GREATER-THAN: moves to execute the instruction at the new address if the result of the operation is greater than 0
  - CALL: moves to execute the subroutine
  - RETURN: returns (from the subroutine) to continue executing the program

Common Instructions

❖Control/Sequential Instructions

- Example: Add the contents of 100 adjacent memory locations, starting from address 1000. The result is stored in R0.

```
        LOAD R1, #100; R1 ← 100
        LOAD R2, #1000; R2 ←1000
        LOAD R0, #0; R0 ← 0
LOOP:   ADD R0, (R2); R0 ← R0 + M[R2]
        INCREMENT R2; R2 ← R2 + 1
        DECREMENT R1; R1 ← R1 – 1
        BRANCH-IF-GREATER-THAN Loop; Return to execute
    the instruction after the Loop label if R1 > 0.
```

Common Instructions

Input/Output Commands

- Used to transmit data between the computer and peripheral devices
- Peripheral devices communicate with the computer through ports. Each port has a dedicated address
- The two basic I/O commands used are INPUT and OUTPUT commands
  - INPUT commands are used to transfer data from a peripheral device to the microprocessor
  - OUTPUT commands are used to transfer data from the microprocessor to an output device

Common Instructions

- Example

```
        CLEAR R0; R0 ← 0
        CLEAR R2; R2 ← 0
        MOVE R1, #100; R1 ← 100
LAP:
        ADD R0, 1000(R2); R0 ← R0+ M[R2+1000]
        INCREMENT R2; R2 ← R2+1
        DECREMENT R1; R1 ← R1-1
        BRANCH_IF>0 LAP; go to LAP if R1>0
        STORE 2000, R0; M[2000] ← R0
```

Exercises

- Exercise 1: Given the following code snippet:

```
        MOVE R0, #100;
        CLEAR R1;
        CLEAR R2;
LAP:    ADD R1, 2000(R2);
        DECREMENT R2;
        DECREMENT R0;
        BRANCH_IF>0 LAP;
        STORE 3000, R1;
```

1. Explain the meaning of each instruction.
2. Indicate the addressing mode of each instruction (for instructions with two operands).
3. What does the above code snippet do?

Exercises

- Exercise 2: Given R0 = 1500, R1 = 4500, R2 = 1000, M[1500] = 3000, M[4500] = 500.

  ADD R2, (R0);

  SUBSTRACT R2, (R1);

  MOVE 500(R0), R2;

  LOAD R2, #5000;

  STORE 100(R2), R0;

1. Specify the addressing mode of each instruction.
2. Indicate the register value and its memory location for each executed instruction.

Exercises

- Problem 3: Given an array of 10 numbers, stored contiguously in memory, starting from memory location 1000. Write a program to calculate the sum of all numbers greater than 1 in the array and store the result in memory location 2000. Assume that each memory location stores one element in the array!

Exercises

- Exercise 4: Given an array of 10 numbers, each spaced one memory location apart, starting at memory location 1000. Write a program to calculate the sum of all numbers less than -1 in the array and store the result in memory location 2500.

Exercises

- Exercise 5: Given the following program segment (R1, R2 are registers and the instruction is conventionally in the form INSTRUCTION <DESTINATION> <ROOT>):

  (1) MOVE R0, #400

  (2) LOAD R1, #2000

  (3) STORE (R1), R0

  (4) SUBSTRACT R0, #20

  (5) ADD 2000, #10

  (6) ADD R0, (R1)

  State the meaning of each instruction and determine the value of R0 after the execution of instruction number (6)

Exercises

- Exercise 6: Given the following program segment (R1, R2 are registers and the instruction is conventionally in the form INSTRUCTION <DESTINATION> <ROOT>):

(1) STORE –100(R2), R1
(2) LOAD R1, (00FF)
(3) COMPARE R3, R4
(4) JUMP-IF-EQUAL Label
(5) ADD R3, R4
(6) ADD R2, 2
(7) Label:

1. Determine the addressing mode and meaning of each instruction;
2. State how to resolve data conflicts in the pipeline when executing the above program segment, knowing that each instruction is divided into 5 stages.
3. Assuming R3 != R4 and each instruction execution stage takes 0.1ns, compare the CPU execution time of the first 6 instructions in the case without using the pipeline mechanism and with the pipeline mechanism in part 2.

Exercises

- Exercise 7: Given the following program segment (R1, R2 are registers and the instruction is conventionally in the form INSTRUCTION <DESTINATION> <ROOT>):
  - (1) LOAD R2, #400
  - (2) LOAD R1, #1200
  - (3) STORE (R1), R2
  - (4) SUBSTRACT R2, #20
  - (5) ADD 1200, #10
  - (6) ADD R2, (R1)
  1. State the meaning of each instruction;
  2. Determine the value of register R2 after the execution of instruction number (6)
  3. State a way to resolve data conflicts in the pipeline when executing the above program segment knowing that each instruction is divided into 5 stages in the pipeline: Read instruction (IF), decode & read operand (ID), access memory (MEM), execute (EX) and store result (WB).

Exercises

- Exercise 8: Translate C code instructions into CPU-level machine code. Then, explain how to resolve data conflicts in the pipeline when executing the program segment.

Array in C: A is a 32-bit array of elements

```
g = g - A[8];
A[12] = h + A[8];
```

Knowing that g is stored in R1, h in R2, and R3 contains the base address of array A.

Exercises

- Exercise 9: Translate C code instructions into CPU-based machine code. Then, explain how to resolve data conflicts in the pipeline when executing the program segment.

  Branch and jump instructions: if (i==j) f = g+h; f = f-i;

  Knowing that f, g, h, i, j are currently stored in registers R1, R2, R3, R4, R5. Branch and jump instructions: if (i==j) f = g+h; else f = f-i;

  Knowing that f, g, h, i, j are currently stored in registers R1, R2, R3, R4, R5.

Exercises

- Exercise 10: Translate C code instructions into CPU-level machine code. Then, explain how to resolve data conflicts in the pipeline when executing the program segment.

Switch/case statement: For C code where amount is stored in R1 and fee is stored in R2:

```
switch (amount) {
        case 20: fee = 2; break;
        case 50: fee = 3; break;
        case 100: fee = 5; break;
        default: fee = 0;
        }
```

Exercises

- Exercise 11: Translate C code instructions into CPU machine code. Then, explain how to resolve data conflicts in the pipeline when executing the program segment.

  While loop statement: Knowing that i is stored in R1, k in R2, the address of array A is in R3 and A is an array of 32-bit elements.

  while (A[i] == k) i += 1;

Exercises

- Exercise 12: Translate C code instructions into CPU-level machine code. Then, explain how to resolve data conflicts in the pipeline when executing the program segment.

For loop:

```
int sum = 0;
int i;
for (i=0; i!=10; i = i+1) {
        sum = sum + i;
        }
```

Exercises

- Exercise 13: Translate C code instructions into CPU-level machine code. Then, explain how to resolve data conflicts in the pipeline when executing the program segment.

For loop:

```
int sum = 0;
int i;
for (i=1; i < 101; i = i*2) {
        sum = sum + i;
        }
```

Exercises

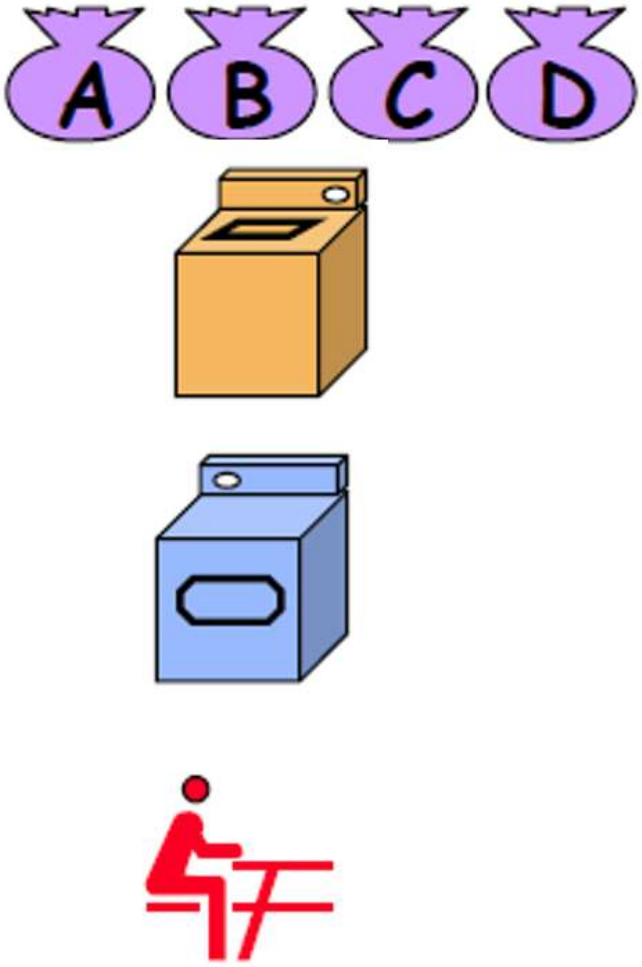
- Exercise 14: Translate C code instructions into CPU-level machine code. Then, explain how to resolve data conflicts in the pipeline when executing the program segment.

For loop: This loop accesses a data array assuming array has a base address at R0 and array A is a 32-bit array.

```
int array[1000];
int i;
for (i=0; i < 1000; i = i + 1)
        array[i] = array[i] * 8;
```

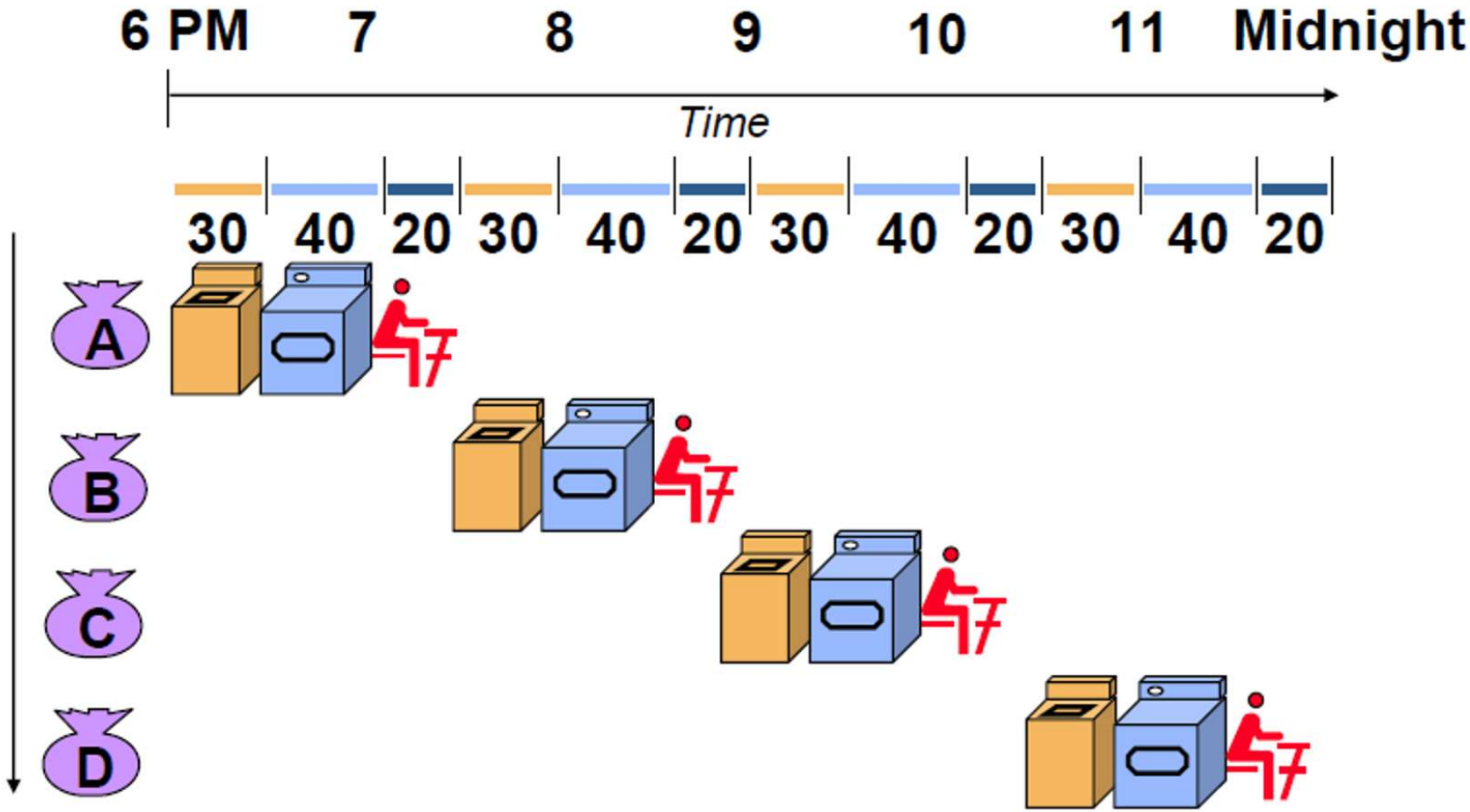
Introduction to pipeline

- Laundry problem: There are 4 bags of clothes A, B, C, D that need washing, drying, and folding.
  - Washing takes 30 minutes
  - Drying takes 40 minutes
  - Folding takes 20 minutes



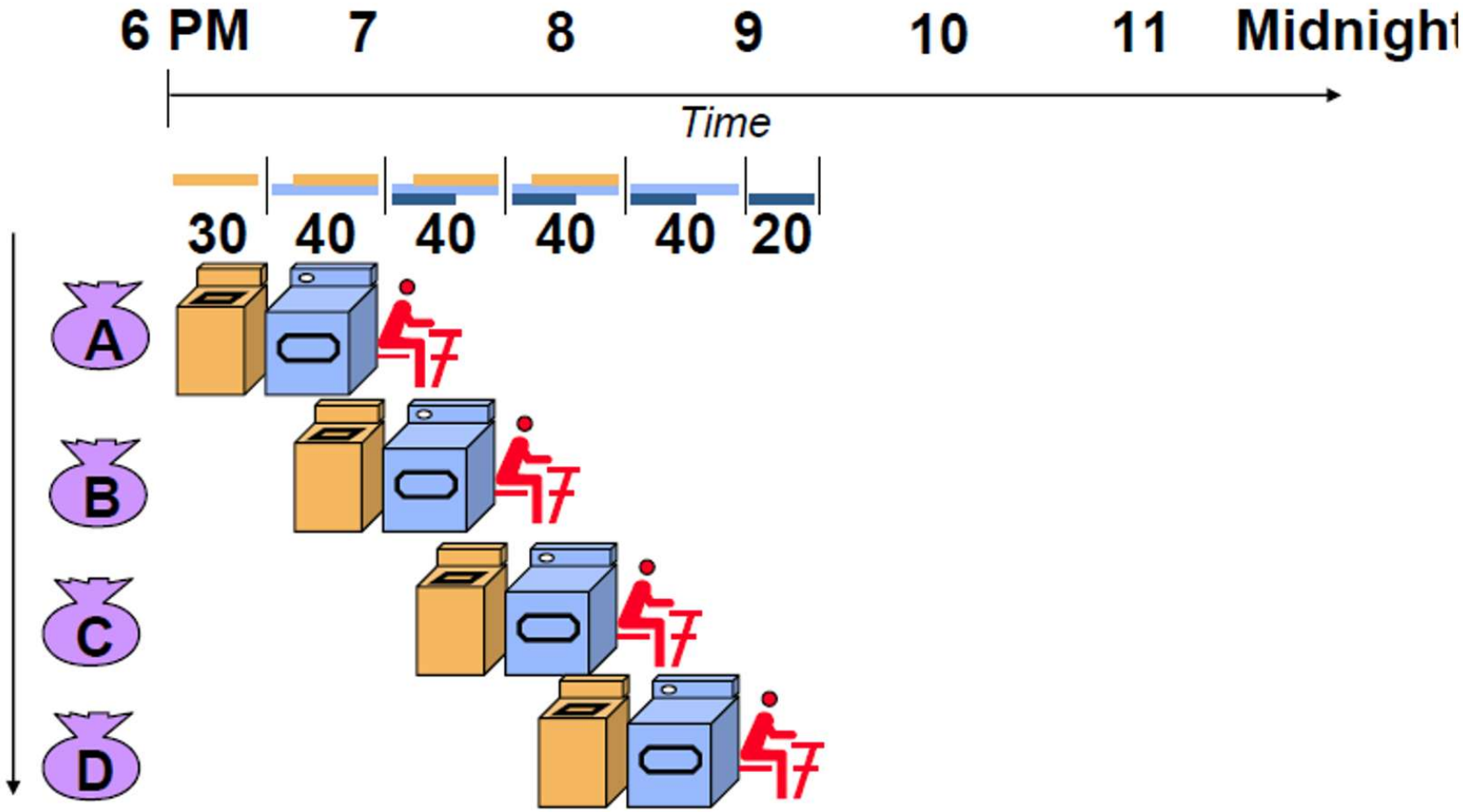
Introduction to pipeline

- Laundry problem:
  - Sequence performing



Introduction to pipeline

- Laundry problem:
  - Pipeline approach



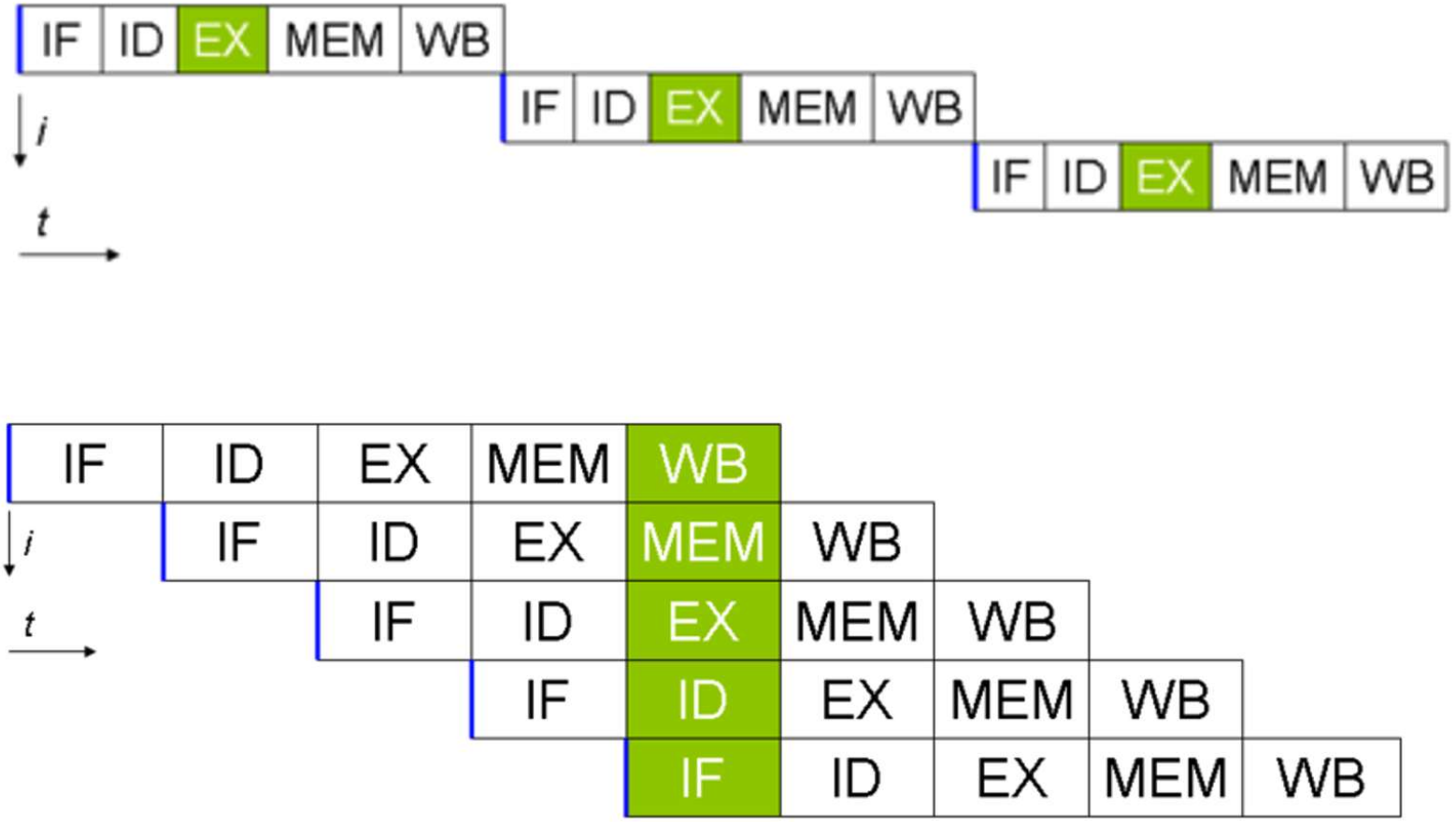
Introduction to pipeline

- Introduction to the CPU Pipeline Principle: The instruction execution process is divided into phases or stages. Each instruction can be executed in 5 stages of the load-store system:
  - IF (Instruction Fetch): Retrieves the instruction from memory (or cache)
  - ID (Instruction Decode): Decodes the instruction and retrieves the operands
  - IE (Instruction Execution): If it is a memory access instruction, calculates the memory address
  - Memory access (MEM): Reads and writes to memory; if there is no memory access, this stage is omitted
  - WB (Write Back): Writes the result (if any) processed by the CPU to a register/storage memory

Introduction to pipeline

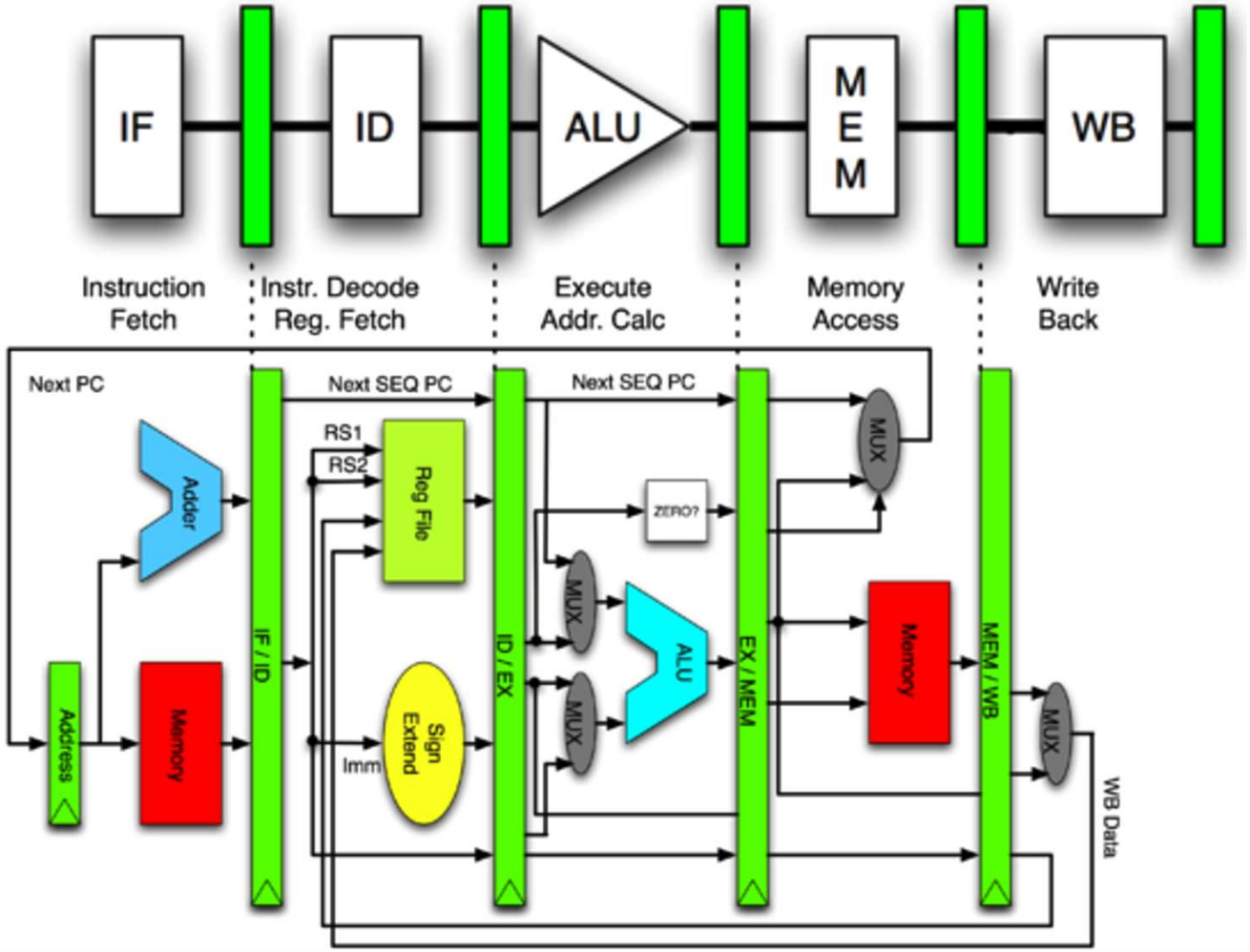
❖Introduction to the CPU Pipeline Principle:

- Improve performance by increasing the number of instructions to be processed.



❖Introduction to the CPU Pipeline Principle:

- Hardware components involved in the pipeline



Introduction to pipeline

❖Introduction to the CPU Pipeline Principle:

- Pipeline is a parallel technique at the instruction level (ILP: Instruction Level Parallelism)
- A full pipeline always receives one new instruction at each clock cycle
- An incomplete pipeline has delay stages in the processing
- The number of pipeline stages depends on the CPU design:
  - 2, 3, 5 stages: simple pipeline
  - 14 stages: Pen II, Pen III
  - 20 – 31 stages: Pen IV
  - 12 - 15 stages: Core

Introduction to pipeline

❖ Introduction to the CPU Pipeline Principle:

- Why not increase the number of pipeline stages to a larger number → increase performance?
  - Increase hardware components: More expensive
  - Larger chip: Consumes more power
- The current common number of stages ranges from 16 to 26:
  - Stage execution time:
    - All stages should have equal execution time
    - Slow stages should be divided
  - Choosing the number of stages:
    - Theoretically, the more stages, the higher the performance
    - If the pipeline is long and empty for some reason, it will take a long time to fill the pipeline

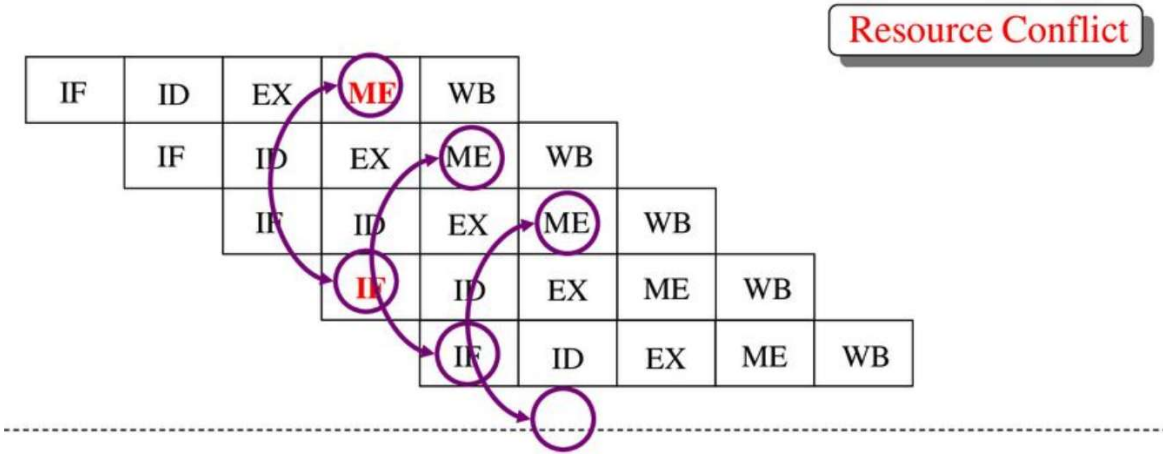
Pipeline problems

- Resource conflict issues:
  - Memory access conflicts
  - Register access conflicts
- Data hazards:
  - Mostly RAW or Read After Write Hazards
- Branch Instructions:
  - Unconditional
  - Conditional
  - Calls, execution, and return from subroutines

Pipeline problems

❖Resource Conflict

- Insufficient Resources
  - Example: If memory only supports one read/write operation at a time, the pipeline requires two memory accesses simultaneously (reading instructions at the IF stage and reading data at the ID stage) → a conflict arises
- Solution:
  - Improve resource capacity
  - Memory/cache: support multiple ead/write operations simultaneously
  - Divide the cache into instruction cache and data cache to improve access



❖Data Hazard

- Consider the following code:

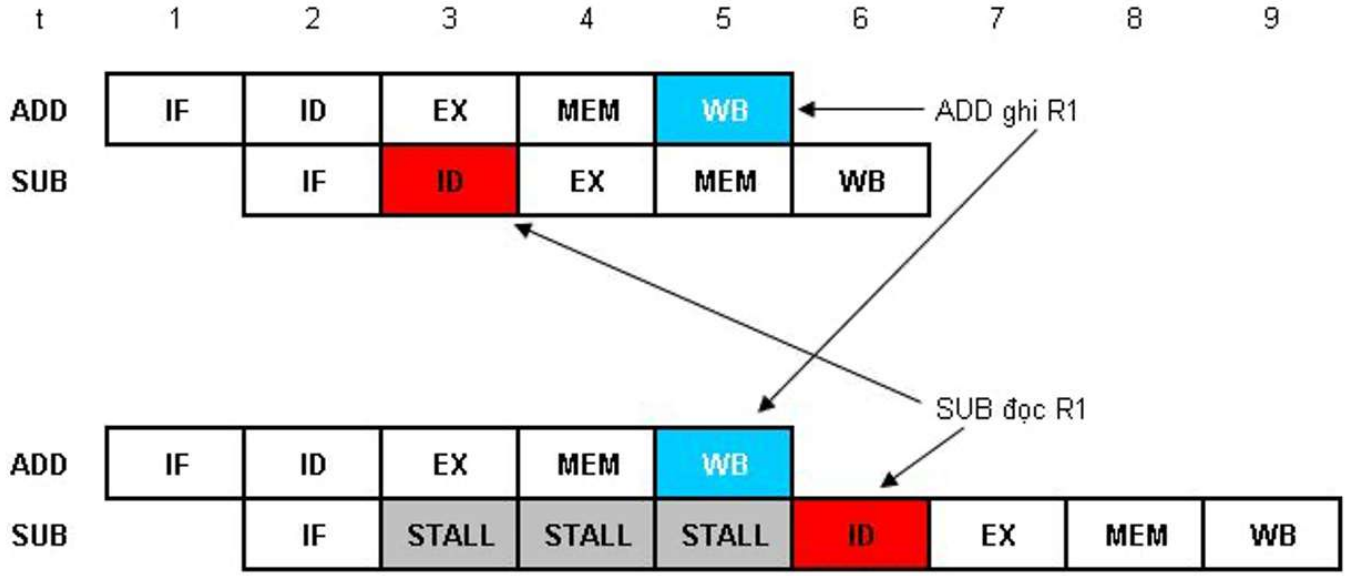
```
(1)  a = 1;
(2)  b = 2;
(3)  a = a + b;
(4)  c = a + 1;
```

- The problem of data not being ready for subsequent dependent commands is called a Data Hazard – a data conflict or data dependency between commands.

❖Data Hazard

- Consider the following code:

ADD R1, R1, R3;
SUB R4, R1, R2;



Pipeline problems

❖Data Hazard

- Data Conflict Resolution Approach
  - Identify data-dependent instructions
  - Dependent instructions need to execute (EX) after the instructions they depend on have finished executing →Stop the pipeline: delay or stop the pipeline using some method until the correct data is available
  - Use a compiler to identify RAW and Insert NO-OP instructions between instructions containing RAW
    - A NO-OP instruction does not change the CPU state but only takes away 1 instruction cycle - the time of 1 cycle
    - Change the sequence of instructions in the program and insert data-independent instructions between two instructions containing RAW
  - Use hardware to identify RAW (found in modern CPUs) and predict the value of dependent data

Handling data and resource conflicts

❖Data Hazard

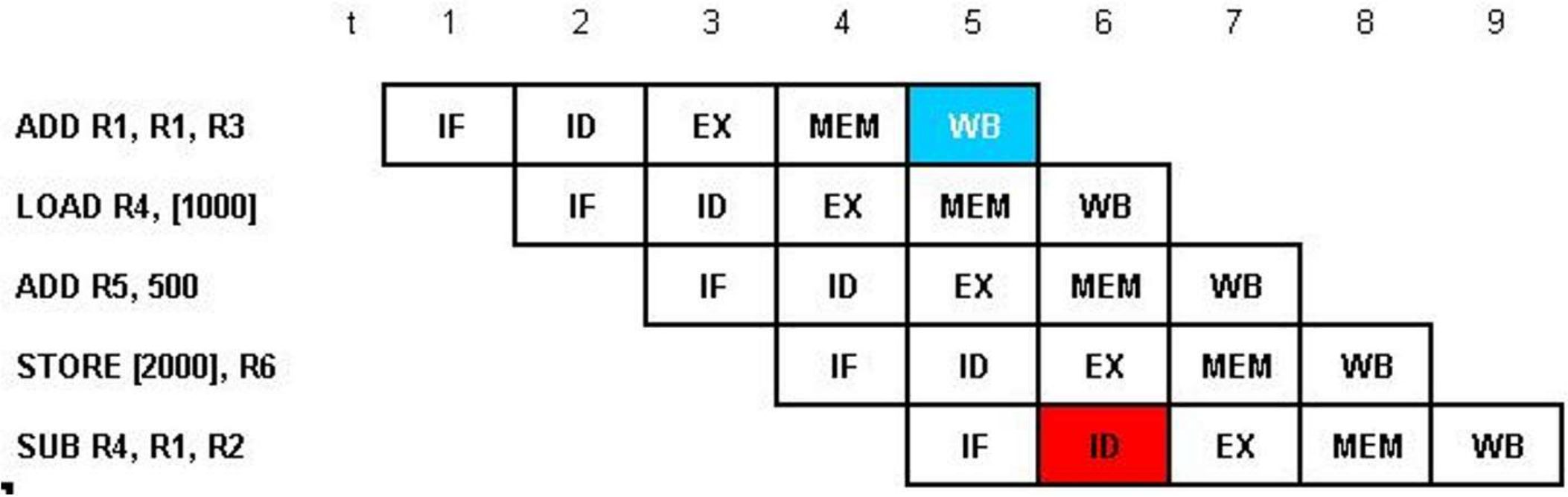
- Delay the execution of the SUB command by inserting 3 NO-OPs:

t	1	2	3	4	5	6	7	8	9
ADD	IF	ID	EX	MEM	WB				
NO-OP		NO-OP	NO-OP	NO-OP	NO-OP	NO-OP			
NO-OP			NO-OP	NO-OP	NO-OP	NO-OP	NO-OP		
NO-OP				NO-OP	NO-OP	NO-OP	NO-OP	NO-OP	
SUB					IF	ID	EX	MEM	WB

Handling data and resource conflicts

❖Data Hazard

- Insert three independent data commands between ADD and SUB:



Handling data and resource conflicts

- Exercise 1: Translate the following C code, where b, c, e, f are stored in R2, R3, R5, R6; a and d are stored in R1 and R4. Indicate how many NO-OP instructions need to be added in the case of CPU pipeline design:

```
a = b + c;
d = a - e + f;
```

Handling data and resource conflicts

- Exercise 2: Identify the errors and rearrange the statements to avoid delays when designing the pipeline for the following statements:

```
LOAD R1, 0(R0)
LOAD R2, 4(R0)
ADD R3, R1,R2
STORE R3, 12(R0)
LOAD R4, 8(R0)
ADD R5,R1,R4
STORE R5,16(R0)
```

Branch handling

❖Branching handling

- Consider the following code:

```
(1) if( a > b )
(2)         a = a - b ;
(3) Else
(4)         a = b - a;
```

- Remarks:
  - Pipeline execution: Command (1) was executed before checking whether a > b was completed.
  - Problems in pipeline techniques when checking conditions for branching or looping statements (e.g., while do)
  - This case is called Branch Hazard when encountering branching and looping statements

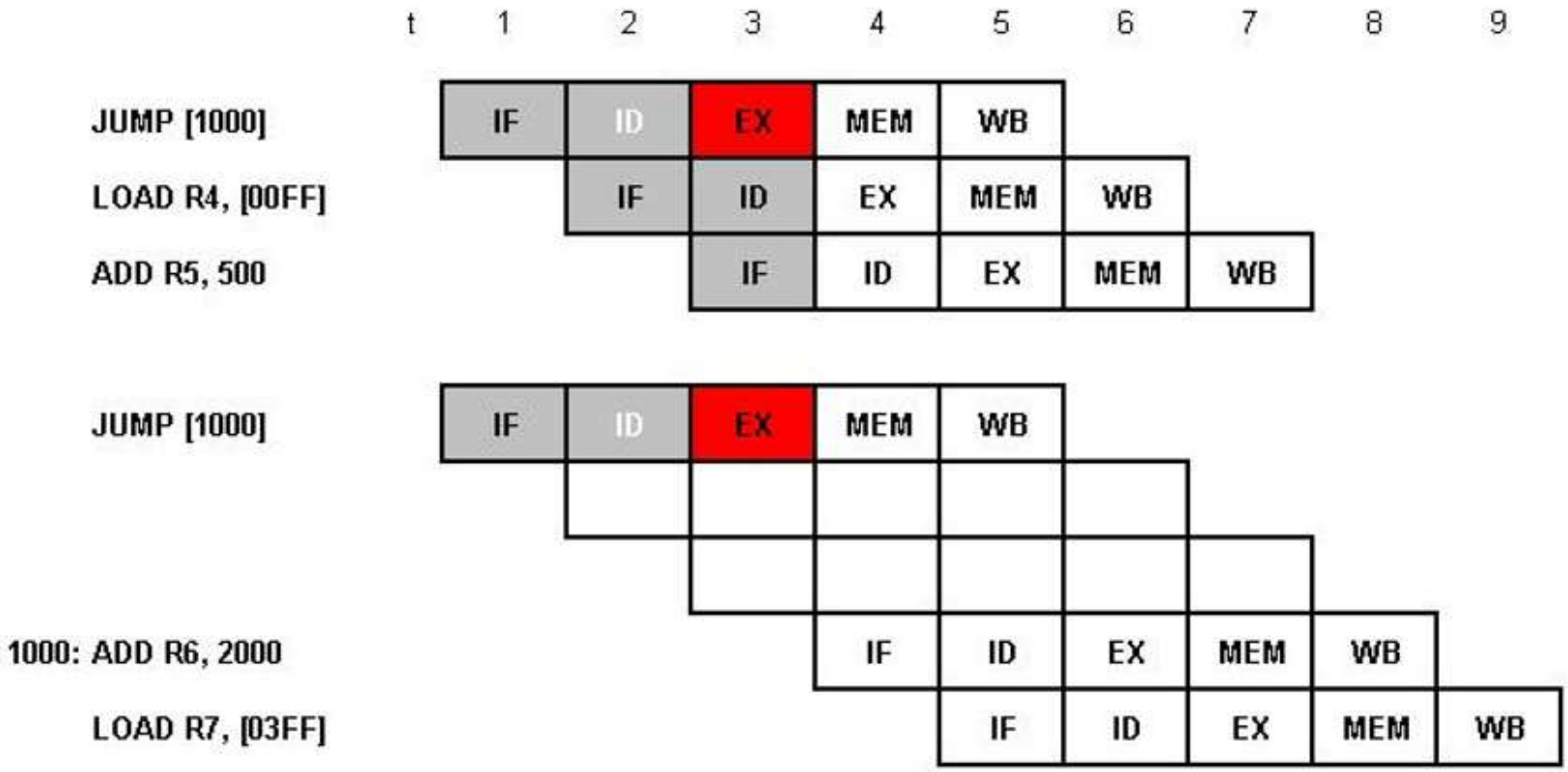
Branch handling

❖Branching handling

- Branch instructions account for approximately 10-30% of all instructions. Branch instructions can cause:
  - Interruptions during normal program execution
  - Emptying the pipeline if not effectively prevented
- With CPUs that have long pipelines (P4 with 31 stages) and multiple pipelines running in parallel, the branching problem becomes even more complex because:
  - All currently executing instructions must be pushed out of the pipeline when a branch instruction is encountered
  - New instructions from the branch address must be loaded into the pipeline. This consumes a lot of time to fill the pipeline.

❖Branching handling

- When a branch statement is executed, subsequent statements are pushed out of the pipeline and new statements are loaded ➔ insert NO-OP statements until the branch/conditional statement is executed.



Branch handling

❖Branching handling

- Consider the following code:

```
        JUMP <Address>
        ADD R1, R2
Address: SUB R3, R4
```

- Branch target
  - When a branch instruction is executed, the next instruction is taken from the branch target address, not the instruction at the next jump location.
  - Branch instructions are identified at the ID stage, so they can be known in advance by decoding them beforehand.
  - Use a branch target buffer (BTB) to track executed branch instructions.

Branch handling

❖Branching handling

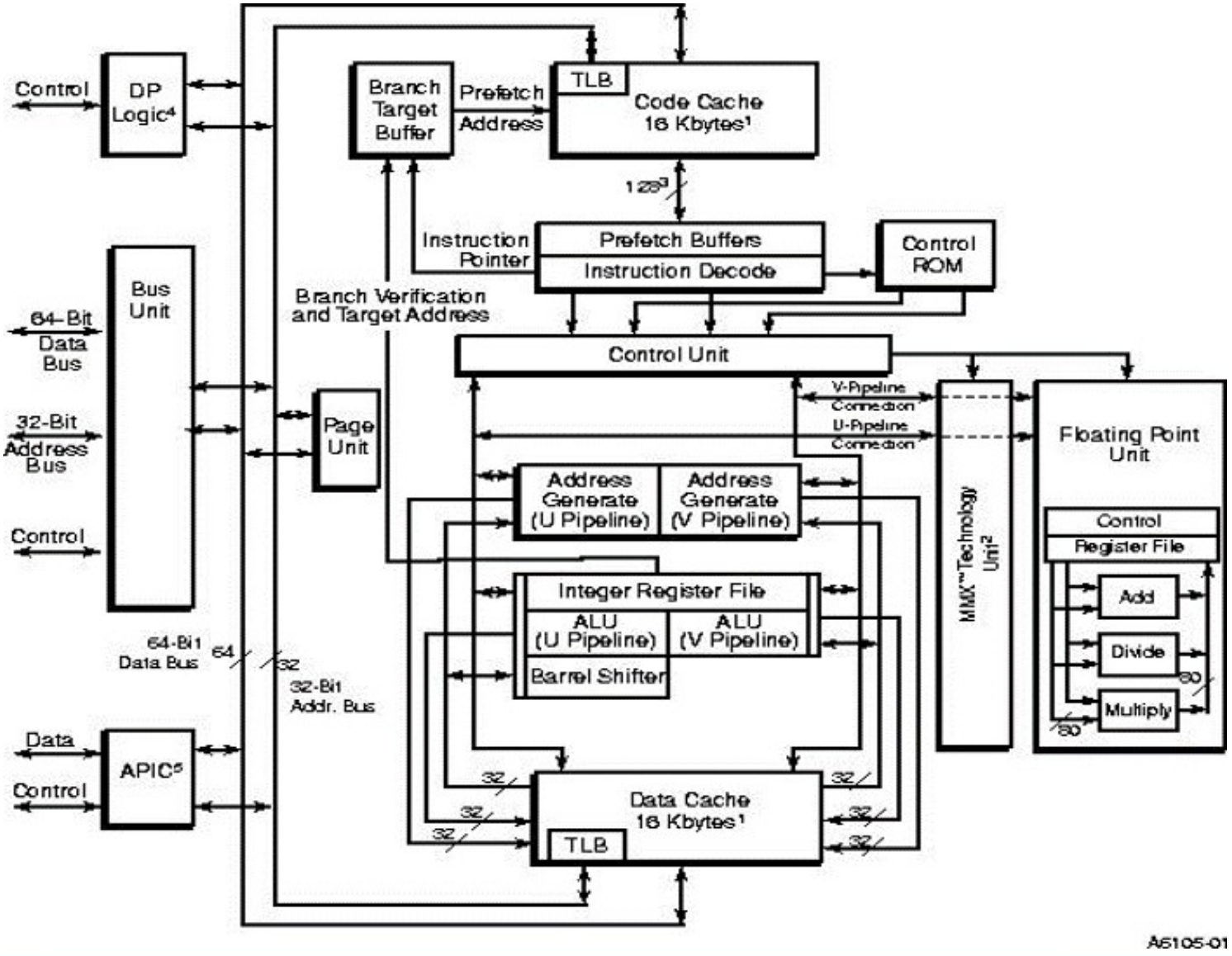
- Consider the following code:

```
        JUMP <Address>
        ADD R1, R2
Address: SUB R3, R4
```

- Branch target
  - Conditional branches
    - –Delayed branching
    - –Branch prediction

Branch handling

- PIII branching destination



Branch handling

❖Conditional Branching

- Conditional branching is more difficult to manage because:
  - There are two target orders to choose from
  - The target order cannot be determined until the branching is completed
    - –Using separate BTBs is inefficient because you have to wait until the target order can be determined.
  - Branching handling strategies:
    - –Slow down branching
    - –Predict branching.

Branch handling

❖Conditional Branching

- Slowing Branching
  - Idea: The branching instruction does not branch immediately • Instead, it is slowed down by a few clock cycles depending on the length of the pipeline
  - Characteristics:
    - –Works well on RISC microprocessors where instructions have equal processing times
    - –Short pipeline (typically 2 stages)
    - –The instruction after the jump instruction is always executed, regardless of the branching instruction's result

Branch handling

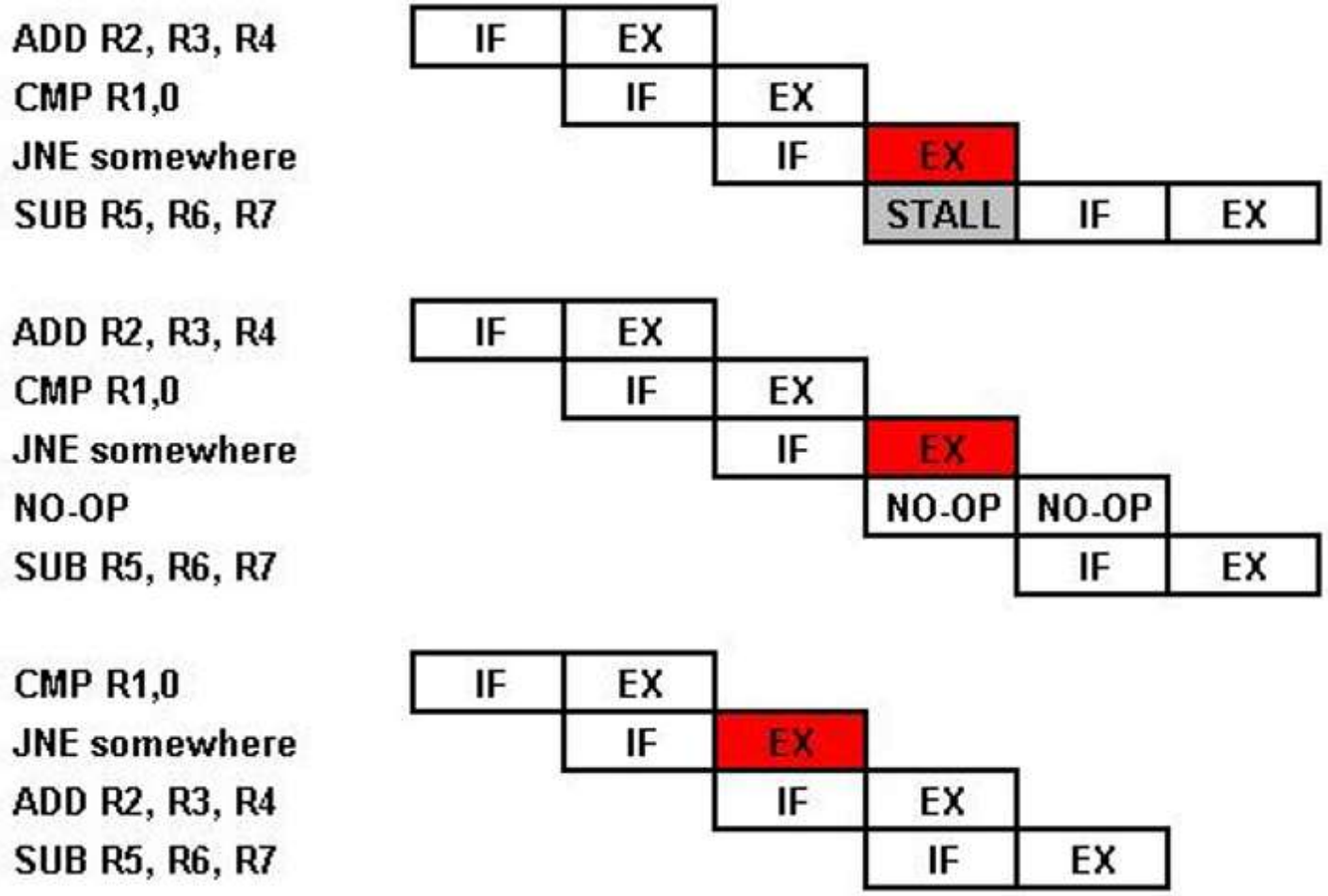
❖Conditional Branching

- Slowing Branching
  - Idea: The branching instruction does not branch immediately • Instead, it is slowed down by a few clock cycles depending on the length of the pipeline
  - Implementation:
    - –Use a compiler to insert a NO-OP at the position immediately after the branching instruction
    - –Move an independent instruction from before to immediately after the branching instruction

Branch handling

❖Conditional Branching

- Slowing Branching



Branch handling

❖Conditional Branching

- Slowing Branching
  - Advantages:
    - –Easy to implement thanks to compiler optimization
    - –No special hardware required
    - –Inserting only NO-OPs reduces performance in long pipelines
    - –Replacing NO-OP instructions with independent instructions can reduce the number of NO-OPs needed by up to 70%
  - Disadvantages:
    - –Increases code complexity
    - –Requires programmers and compiler builders with a deep understanding of the microprocessor pipeline: a major limitation
    - –Reduces code portability because programs must be rewritten or recompiled on new microprocessor platforms

Branch handling

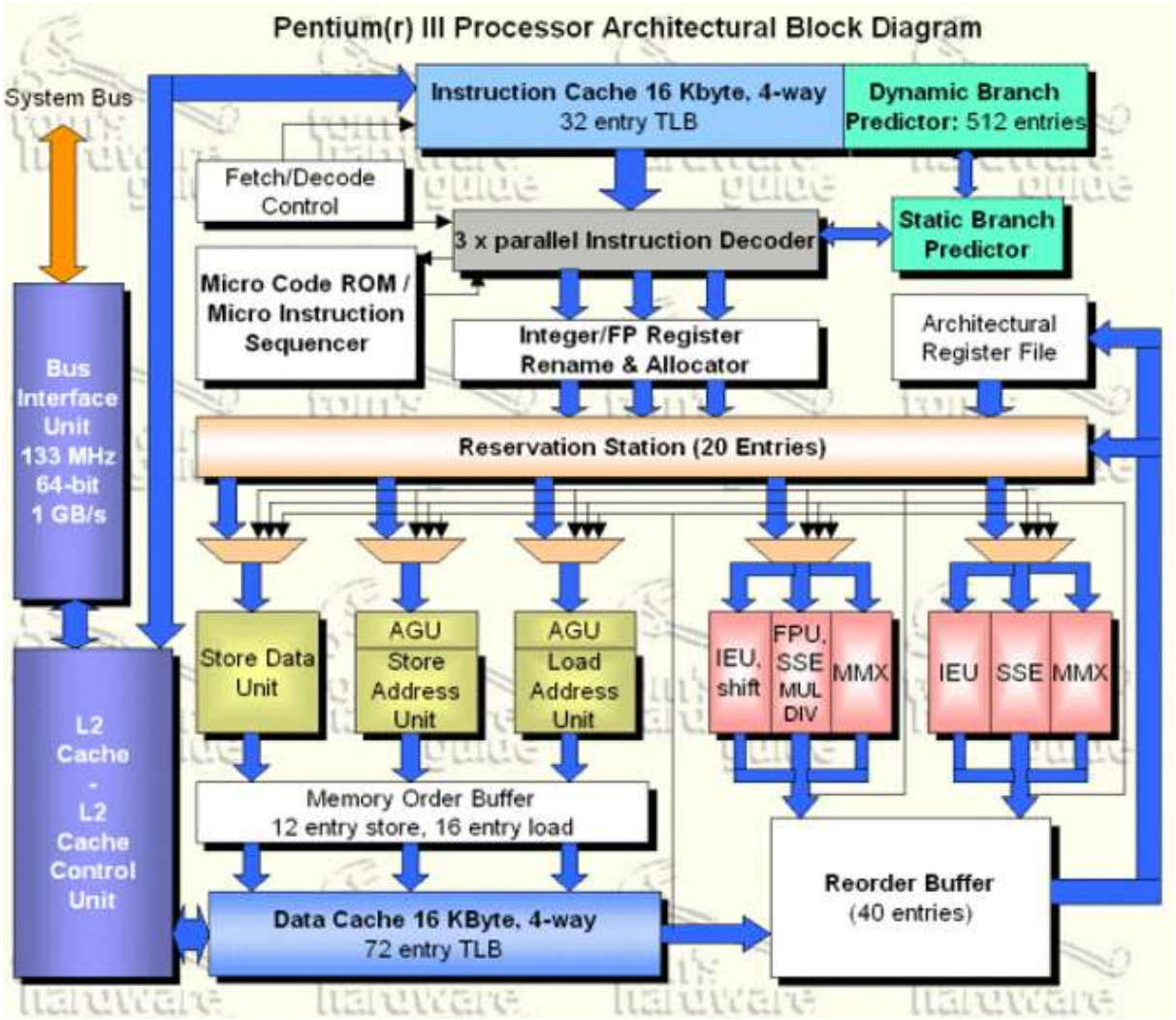
❖Conditional Branching

- Branch Prediction
  - It is possible to predict the destination of a branch instruction:
    - Correct prediction: improves performance
    - Incorrect prediction: pushes the next instructions that have already been loaded and requires reloading the instructions at the branch destination
    - The worst-case scenario for prediction is 50% correct and 50% incorrect
  - Basis for prediction:
    - For backward jump instructions:
    - Often part of a loop
    - Loops are often executed multiple times
    - For forward jump instructions, it is more difficult to predict
    - May be the end of a loop instruction • May be a conditional jump

Branch handling

❖Conditional Branching

- Branch Prediction



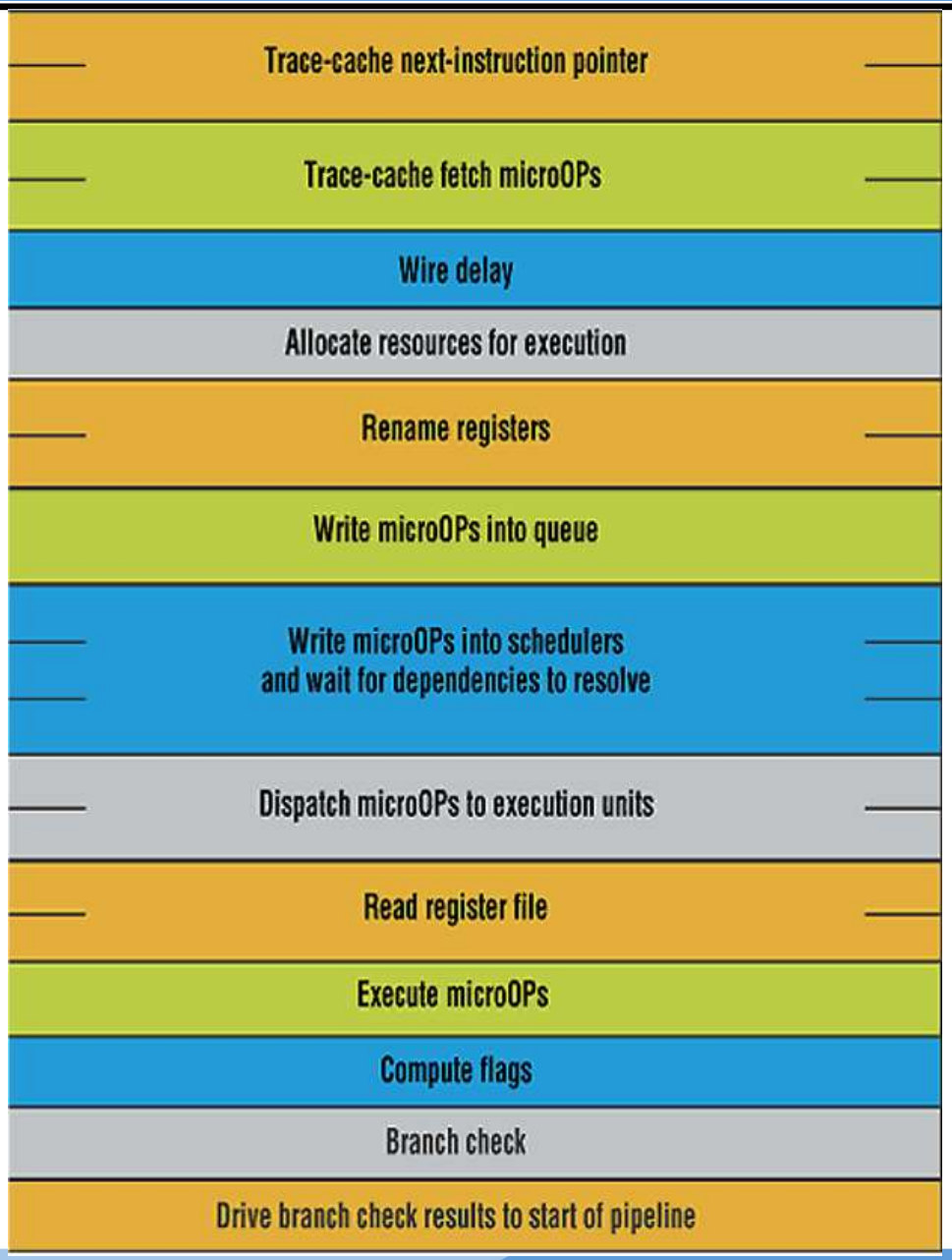
Superpipelining

- Superpipelines are a technique that allows for:
  - Increasing the depth of the instruction pipeline
  - Increasing the clock speed
  - Reducing the latency for each stage of instruction execution.
- For example, if the instruction execution stage by the ALU is long → it can be divided into several smaller stages → reducing the waiting time for those shorter stages.

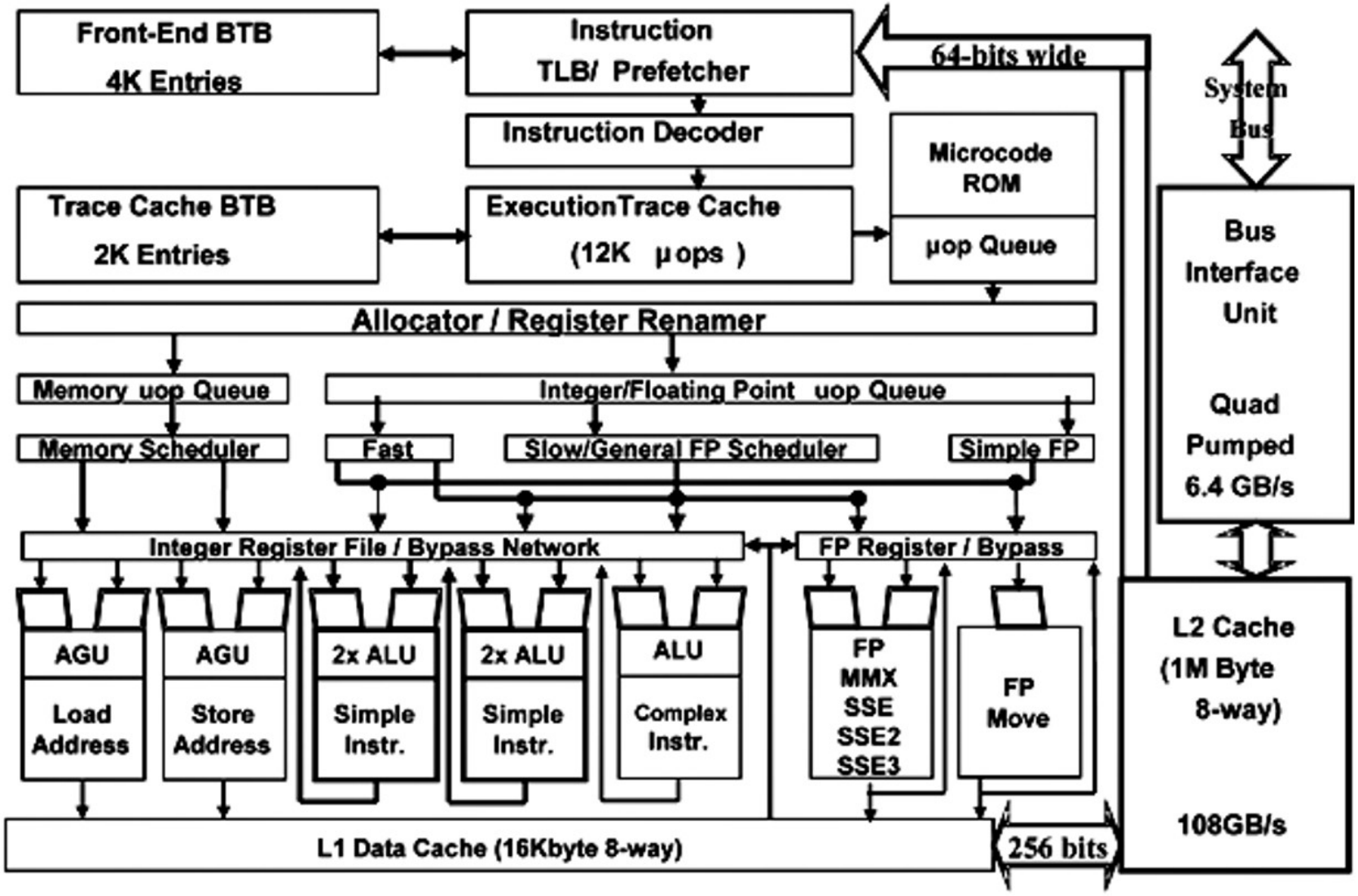
  Pentium 4 superpipelines with 20 stages:

Superpipelining

- Superpipelines are a technique that allows for:
  - Pentium 4 superpipelines with 20 stages:

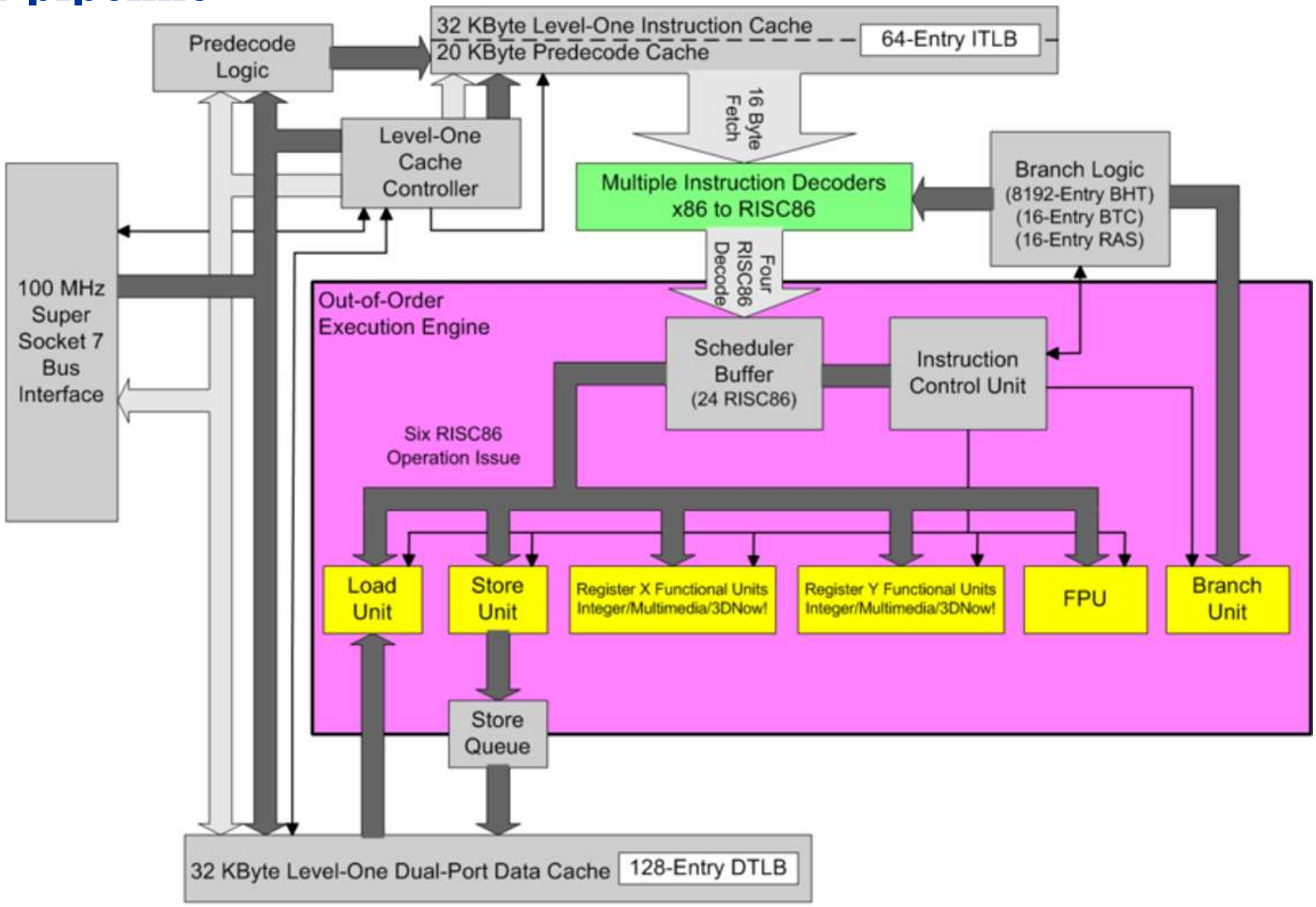


❑Branch Prediction – Intel P4:



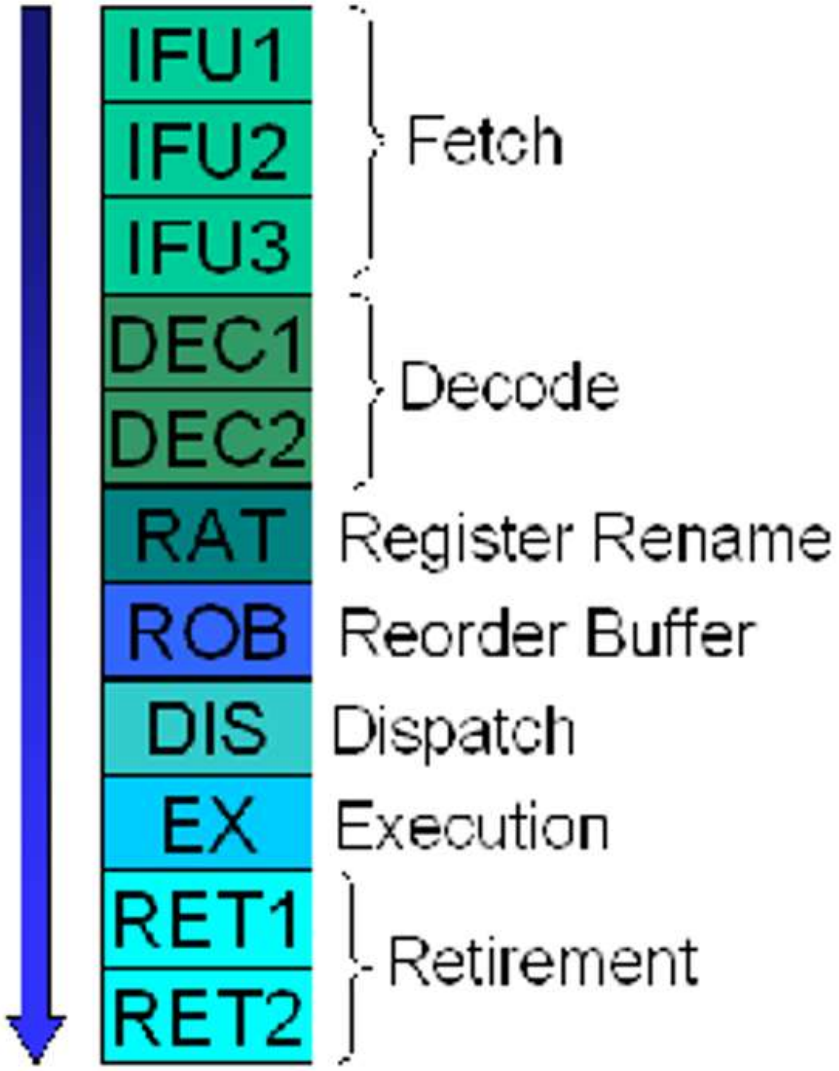
Superpipelining

❑AMD K6-2 pipeline

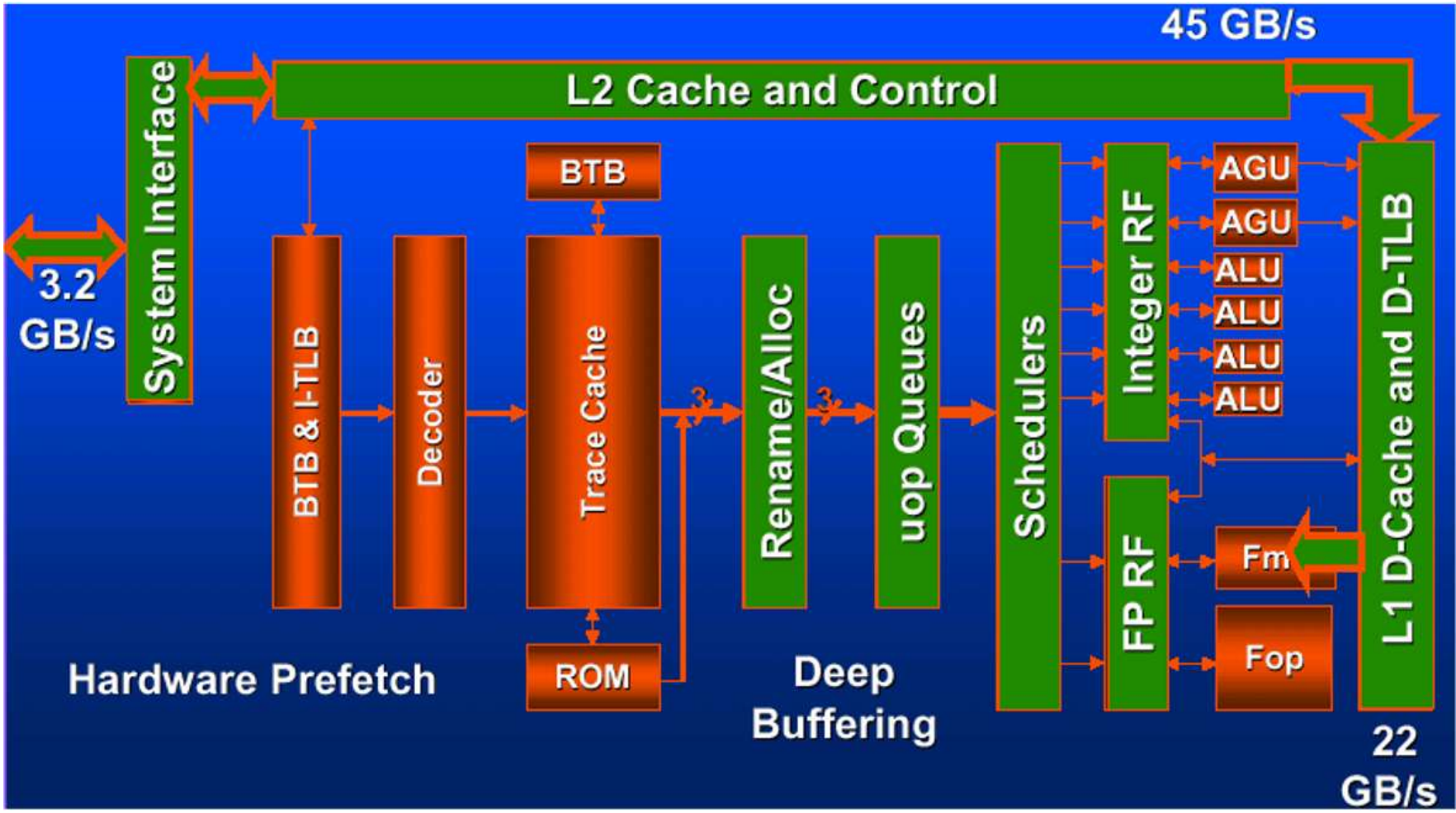


Superpipelining

❑Pipeline – Pen III, M

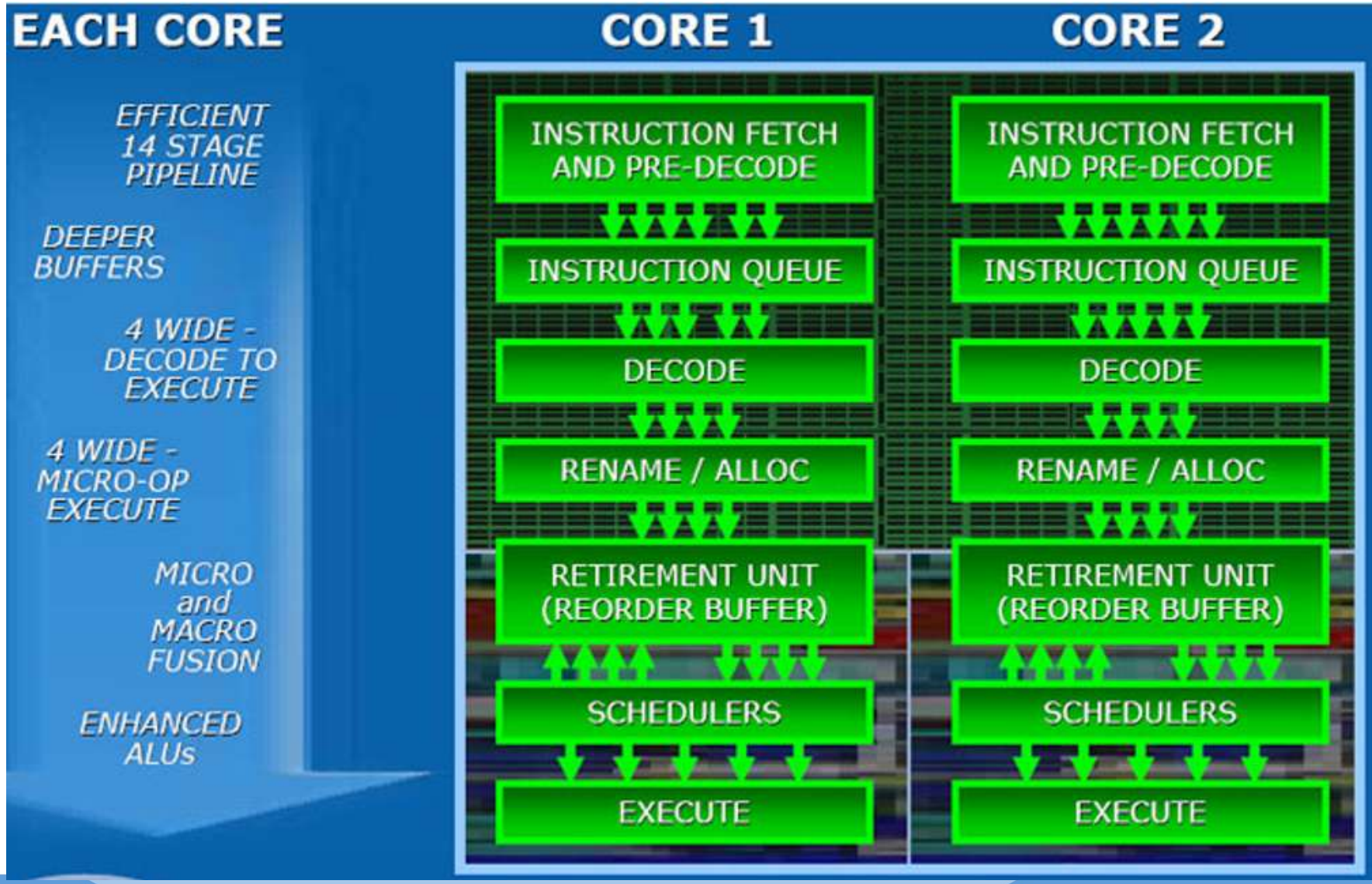


Intel Pen 4 Pipeline



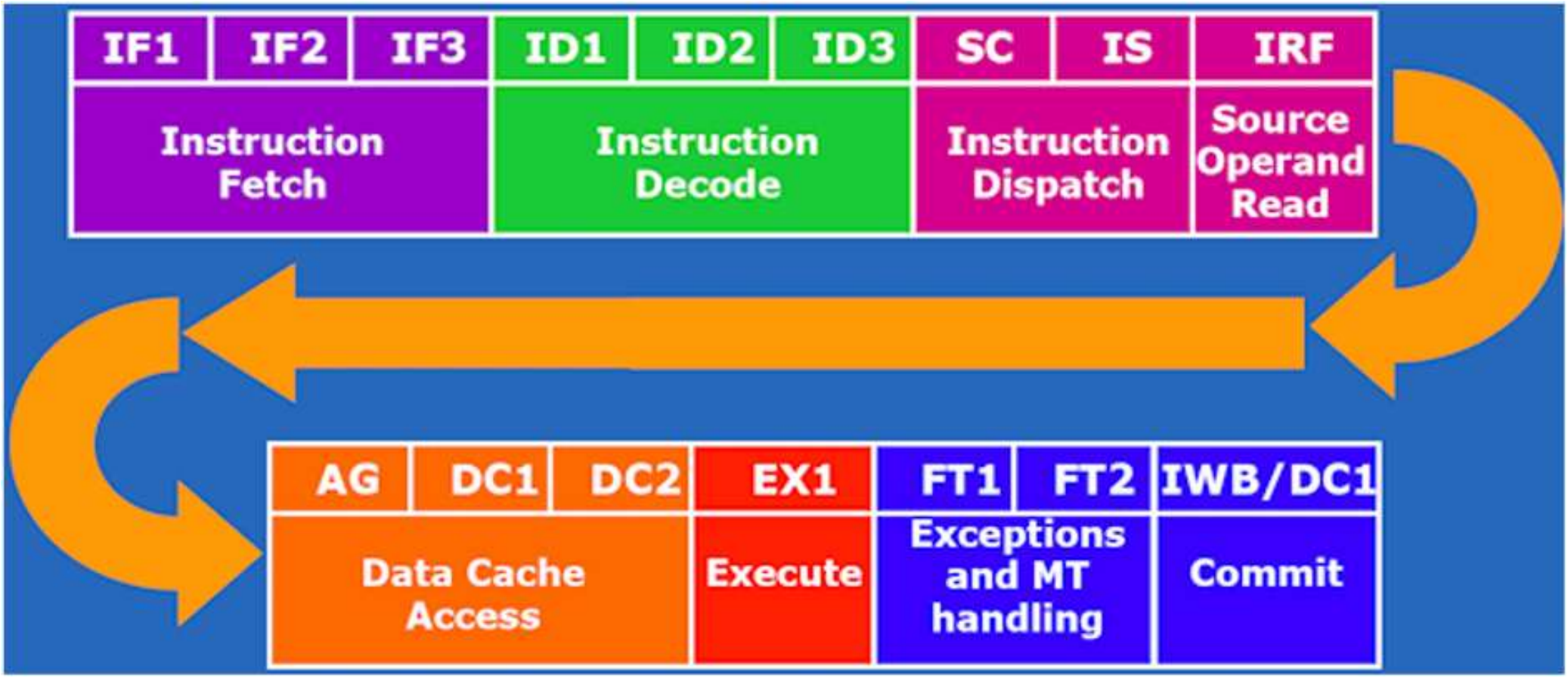
Superpipelining

Intel Core 2 Duo pipeline



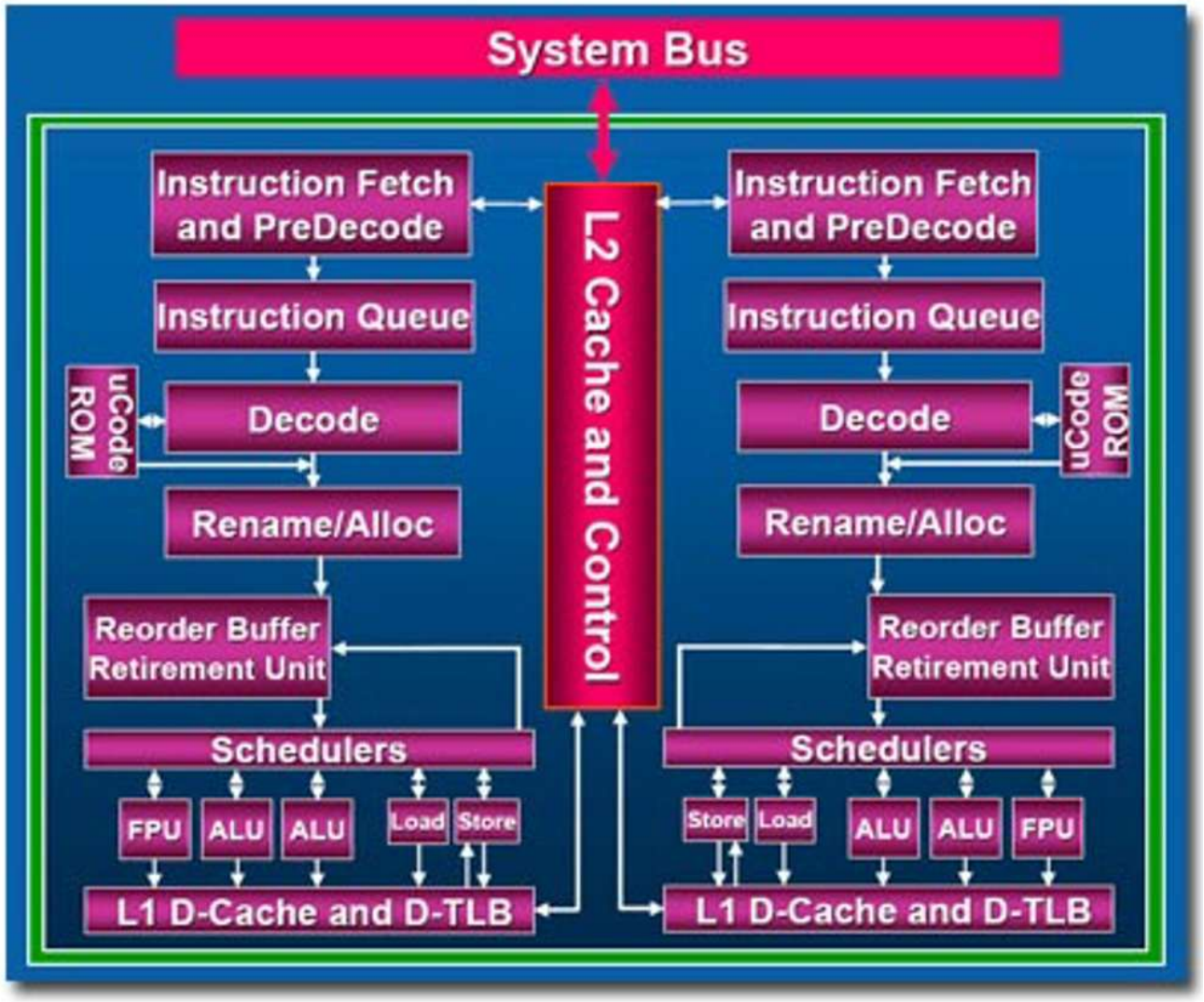
Superpipelining

Intel Atom 16-stage pipeline



Superpipelining

Intel Core 2 Duo – Super Pipeline



Exercises

- Exercise 1: What problems does the CPU pipeline mechanism often encounter? Suggest a solution for resolving data conflicts in the pipeline when executing the following program segment:

  ADD R1, R2, R3 ; R1 ← R2 + R3

  ADD R4, R4, #300 ; R4 ← R4 + 300

  CMP R1, #100 ; compare R1 with 100

  SUB R5, #2000 ; R5 ← R5 - 2000

  Assuming that each instruction is divided into 5 stages in the pipeline: Read instruction (IF), decode & read operand (ID), access memory (MEM), execute (EX), and store result (WB).

Exercises

- Exercise 2: What problems does the CPU pipeline mechanism often encounter? Suggest a solution to resolve data conflicts in the pipeline when executing the following program segment:

  ADD R4, R4, #300 ; R4 ← R4 + 300

  ADD R1, R1, R3 ; R1 ← R1 + R3

  SUB R5, #2000 ; R5 ← R5 - 2000

  SUB R1, R1, #100 ; R1 ← R1 - 100

  knowing that each instruction is divided into 5 stages in the pipeline: Read instruction (IF), decode & read operand (ID), access memory (MEM), execute (EX), and save result (WB).

Exercises

- Exercise 3: According to the command format <TARGET> <ROOT>):
  - (1) MOVE R0, #400
  - (2) LOAD R1, #2000
  - (3) STORE (R1), R0
  - (4) SUBSTRACT R0, #20
  - (5) ADD 2000, #10
  - (6) ADD R0, (R1)

1. State the meaning of each command and determine the value of R0 after the execution of command number (6)
2. State a way to resolve data conflicts in the pipeline when executing the above program segment knowing that each command is divided into 5 stages in the pipeline: Read command (IF), decode & read operand (ID), access memory (MEM), execute (EX) and save result (WB).

Exercises

- Exercise 4: Given the following program segment (R1, R2 are registers and the instruction is conventionally in the form

  INSTRUCTION <DESTINATION> <ROOT>):

  (1) STORE –100(R2), R1
  (2) LOAD R1, (00FF)
  (3) COMPARE R3, R4
  (4) JUMP-IF-EQUAL Label
  (5) ADD R3, R4
  (6) ADD R2, 2
  (7) Label:

  1. Determine the addressing mode and meaning of each instruction;
  2. State how to resolve data conflicts in the pipeline when executing the above program segment, knowing that each instruction is divided into 5 stages.
  3. Assume R3 != R4 and each instruction execution stage takes 0.1ns, compare the CPU time to complete the first 6 instructions in the case without using the pipeline mechanism and with the pipeline mechanism in part 2.

Exercises

- Exercise 5: Given the following program segment (R1, R2 are registers and the instruction is conventionally in the form

  INSTRUCTION <DESTINATION> <ROOT>):

  (1) LOAD R2, #400

  (2) LOAD R1, #1200

  (3) STORE (R1), R2

  (4) SUBSTRACT R2, #20 (5)

  (5) ADD 1200, #10

  (6) ADD R2, (R1)

  1. State the meaning of each instruction;
  2. Determine the value of register R2 after the execution of instruction number (6)
  3. State a way to resolve data conflicts in the pipeline when executing the above program segment knowing that each instruction is divided into 5 stages in the pipeline: Read instruction (IF), decode & read operand (ID), access memory (MEM), execute (EX) and save result (WB).