CSE332

Computer Organization

and

Architecture

Instruction Set Architecture

“Instruction Set Architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine.”

The ISA defines:

Operations that the processor can execute

Data Transfer mechanisms + how to access data

Control Mechanisms (branch, jump, etc)

“Contract” between programmer/compiler + Hardware

What is Instruction Set Architecture?

- Refers to instruction types, instruction format in low level as well as in machine code and detailed descriptions of operand fields (addressing modes of instructions).
- **Instruction Set Architecture also defines:**
  - Operations that the processor can execute (add, sub, mult, …, how is it specified)
  - Data Transfer mechanisms
  - How to access data
  - Number of operands (0, 1, 2, 3)
  - Operand storage (where besides memory)
  - Memory address (how is memory location specified)
  - Type and size of operands (byte, int, float, …)
  - Control Mechanisms (branch, jump, etc)
  - “Contract” between programmer/compiler + Hardware

ISA: Seven Dimensions

- **Class of ISA**

General purpose register architectures,

80x86: register-memory ISA, MIPS: load-store ISA

- **Memory Addressing**

Byte addressing (usually), alignment (some)

- **Addressing modes**

Register, constants/immediate, displacement at least

- **Types and sizes of operands**

8bit (ASCII), 16 bit (Unicode, halfword), 32 bit (int, word), 64 bit

IEEE 754 floating point 32 bit single, 64 bit double precision

- **Operations**

Data transfer, arithmetic logical, control, floating point

- **Control flow instructions**

Jumps, cond. branches, procedure calls, returns, PC-relat. addressing

- **Encoding an ISA**

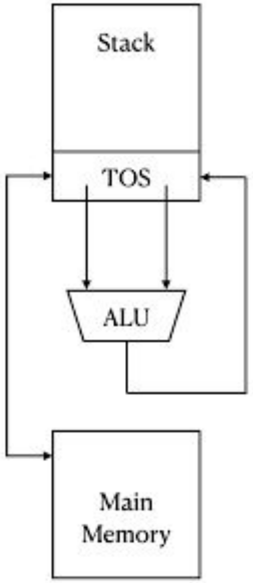
Fixed length vs variable length encoding

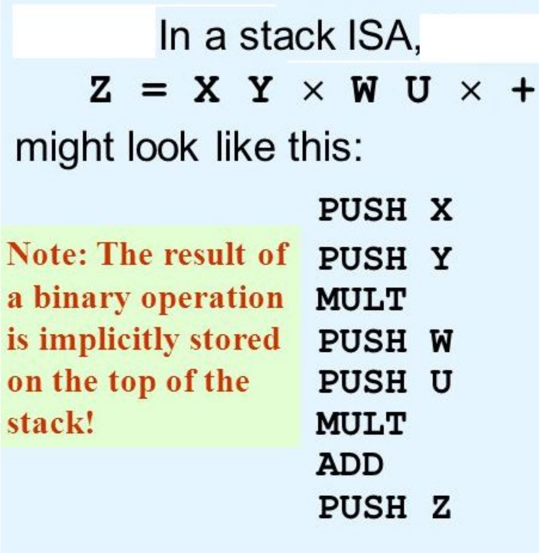
Instruction Set Architecture

- How many instructions and types of instructions processors can understand/decode/process/execute
- How to perform any operation: Accumulator based or General?
- Instruction Format?
- Number of binary bits used to form Instructions
- Same number of bits for all instructions or not?
- How arithmetic/logical/data processing/data transfer operations are encoded in instructions
- How data/main memory address/register names are indicated in instructions
- Data types/formats/number of bits used/positive/negative numbers/ranges of numbers etc
- Memory: Address format/addressing scheme/content of each addressable locations etc

Classification of processors based on ISA

- Stack-based CPU
- Accumulator-based CPU
- Register-Register CPU
- Register-Memory CPU




In a stack ISA,

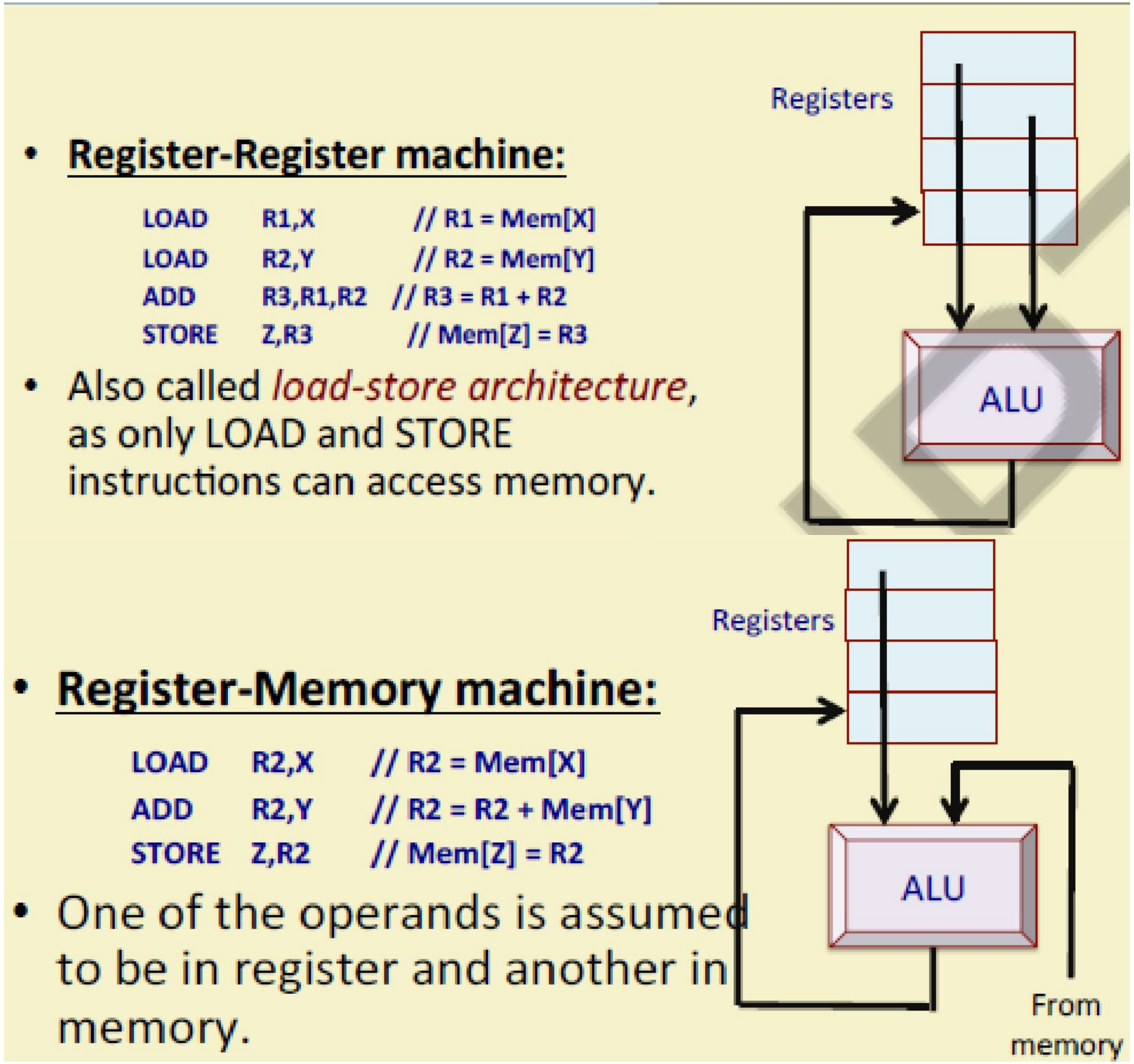
Z = X Y × W U × +

might look like this:

```
PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
PUSH Z
```

Note: The result of a binary operation is implicitly stored on the top of the stack!

Stack-based ISA: First of all push both operands onto the stack and then simply give an add instruction which will add the top two elements of the stack and then store the result in the stack.


- **Register-Register machine:**

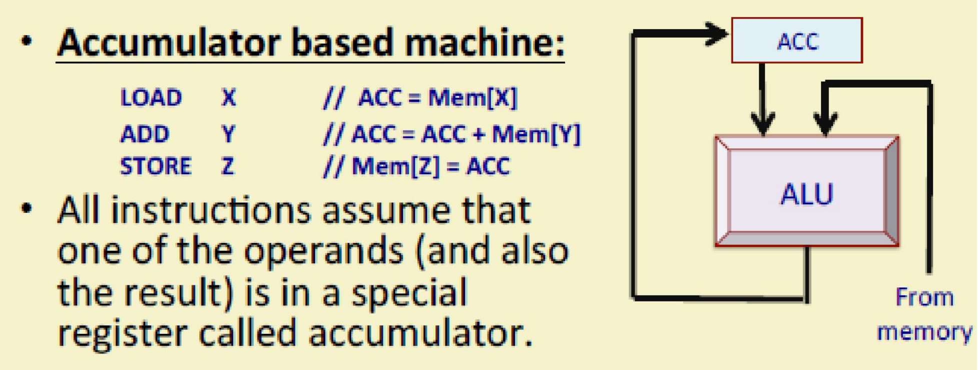
```
LOAD    R1,X        // R1 = Mem[X]
LOAD    R2,Y        // R2 = Mem[Y]
ADD     R3,R1,R2    // R3 = R1 + R2
STORE   Z,R3        // Mem[Z] = R3
```

- Also called *load-store architecture*, as only LOAD and STORE instructions can access memory.

- **Register-Memory machine:**

```
LOAD    R2,X     // R2 = Mem[X]
ADD     R2,Y     // R2 = R2 + Mem[Y]
STORE   Z,R2     // Mem[Z] = R2
```

- One of the operands is assumed to be in register and another in memory.


- **Accumulator based machine:**

```
LOAD    X    // ACC = Mem[X]
ADD     Y    // ACC = ACC + Mem[Y]
STORE   Z    // Mem[Z] = ACC
```

- All instructions assume that one of the operands (and also the result) is in a special register called accumulator.

Stack

Memory-Memory

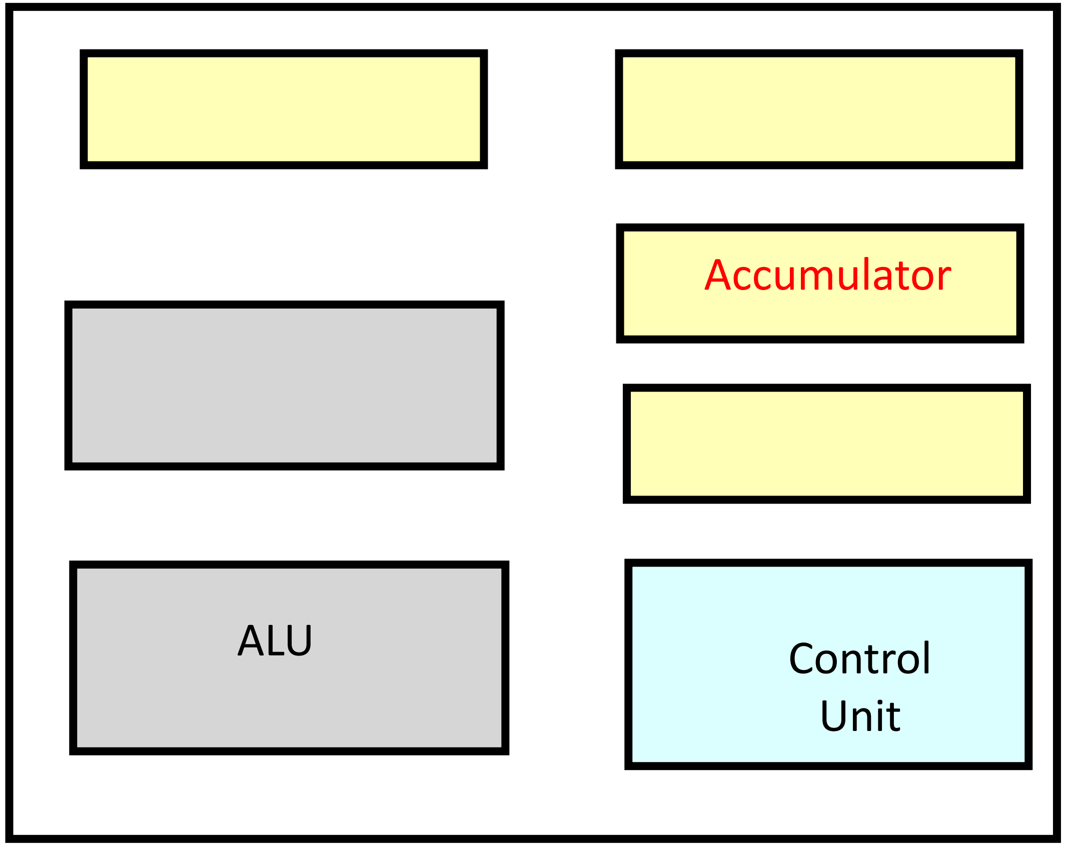
Accumulator Architecture

Extended Accumulator Architecture

General-Purpose Register Architecture

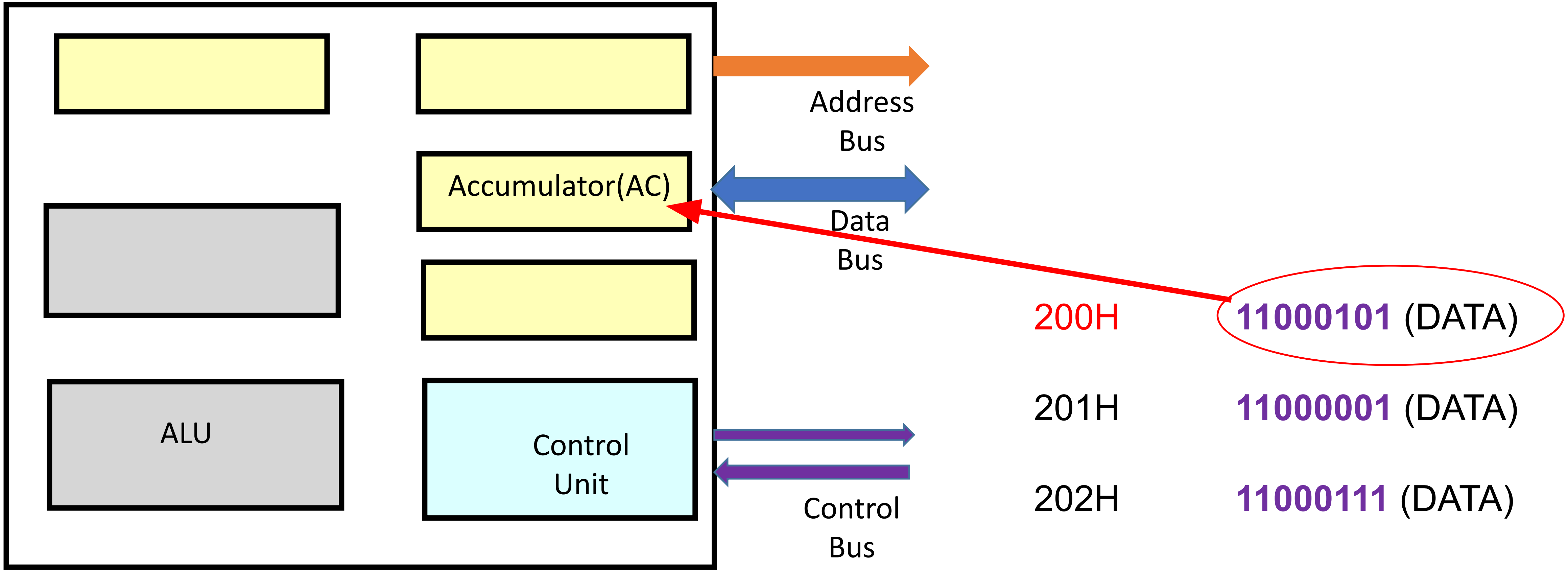
Machine	general-purpose registers	Architecture style	Year	
Motorola 6800	2	Accumulator	1974	
DEC VAX	16	Register-memory memory-memory	1977	
Intel 8086	1	Extended accumulator	1978	
Motorola 68000	16	Register-memory	1980	
Intel 80386	32	Register-memory	1985	
PowerPC	32	Load-store	1992	
DEC Alpha	32	Load-store	1992	

Accumulator-based CPU



- Accumulator is a special purpose register within CPU
- Commonly indicated by AC
- CPU uses this register in almost all instructions by default and it remains implicit (not written/indicated in instructions)
- Results/partial results are also stored in AC by default

Accumulator-based CPU



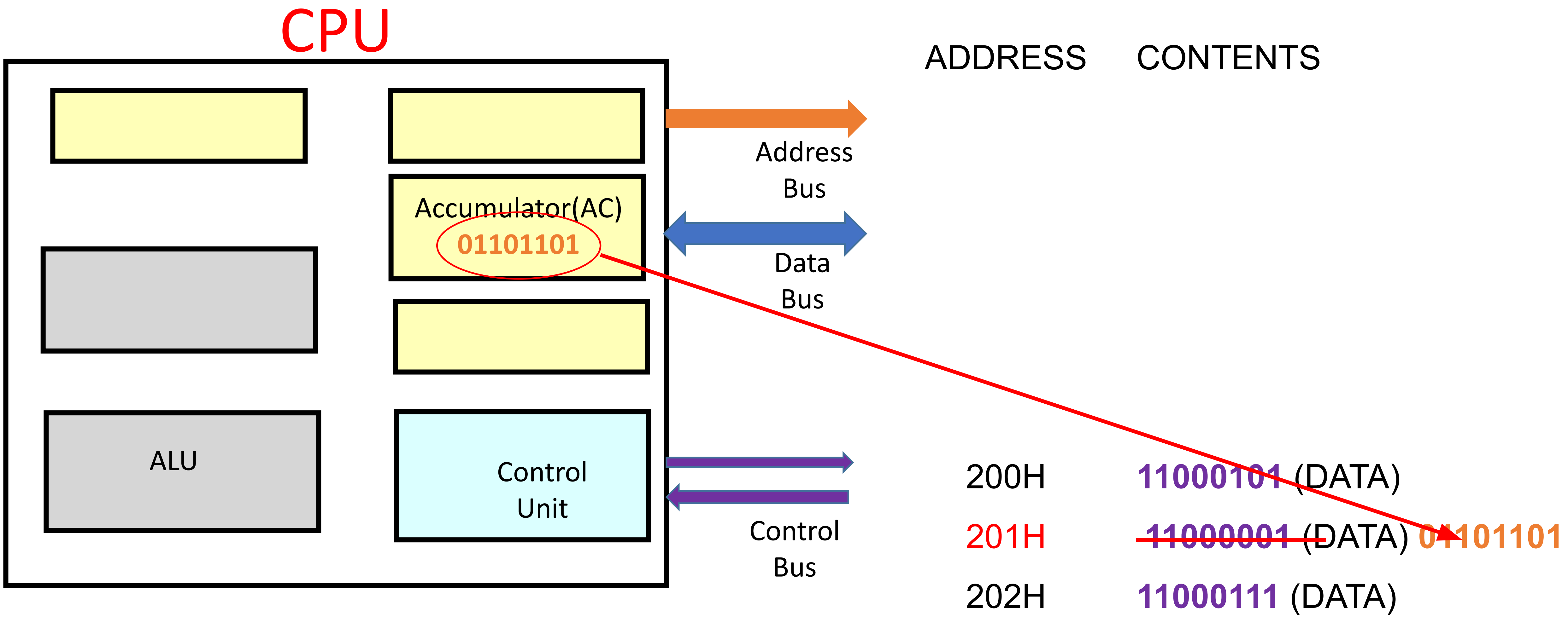
Memory

ADDRESS	CONTENTS
200H	11000101 (DATA)
201H	11000001 (DATA)
202H	11000111 (DATA)

Example: **LOAD M1** **;M1= 200H**

The content of memory location 200H is transferred/copied into AC

Accumulator-based

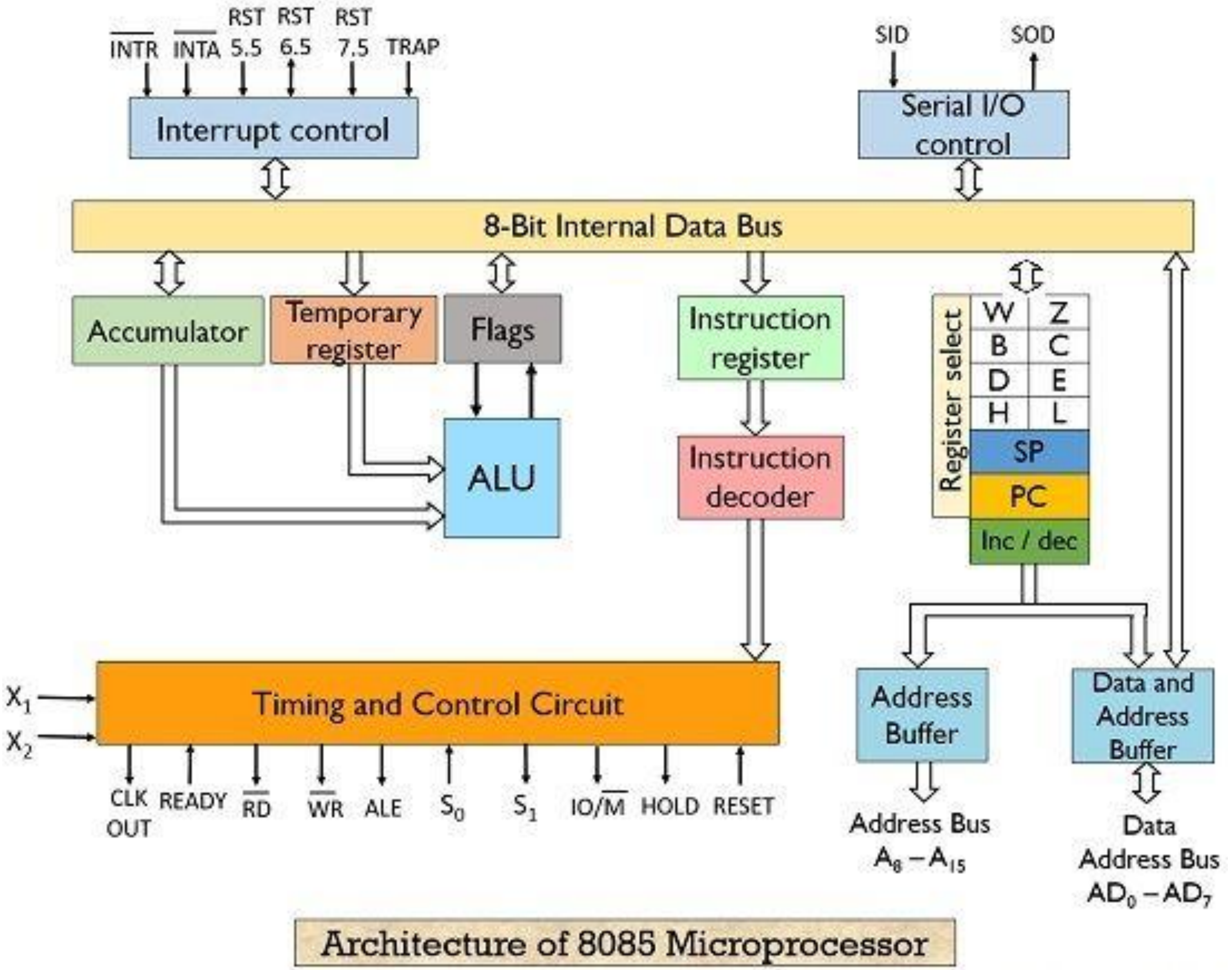


ADDRESS	CONTENTS
200H	11000101 (DATA)
201H	~~11000001~~ (DATA) 01101101
202H	11000111 (DATA)

Example: **STOR M2** ;M2 = 201H and [AC] = 01101101

Current content of AC is saved into RAM at 201H

Intel 8085: Accumulator based CPU



Architecture of 8085 Microprocessor

Electronics Desk

Instructions: Accumulator-based CPU

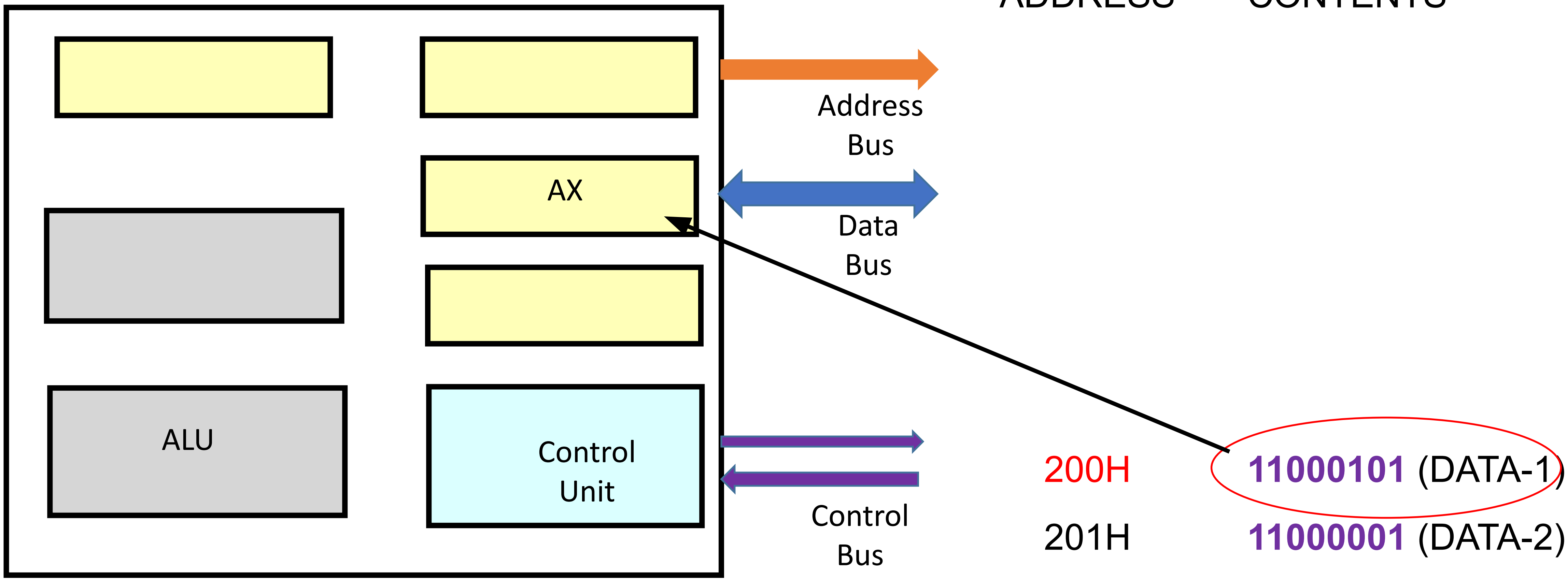
LDA 2034H; the contents of memory location 2034H will be copied into accumulator

ADI 45H ; add 45H to contents of accumulator and stores results to accumulator

ADD B; the contents of register B is added to accumulator and result is stored in accumulator

General purpose CPU

Memory



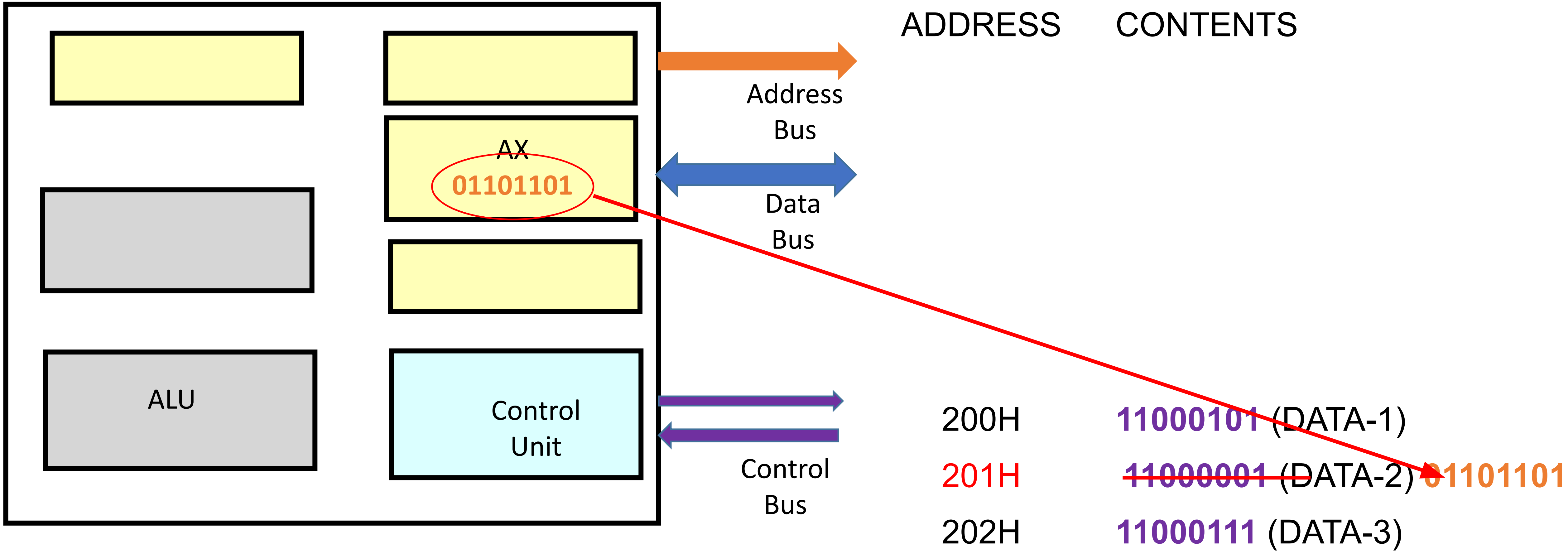
ADDRESS	CONTENTS
200H	11000101 (DATA-1)
201H	11000001 (DATA-2)
202H	11000111 (DATA-3)

Example: **MOV AX, M1** ;M1= 200H

The content of memory location 200H is copied into AX register

General purpose CPU

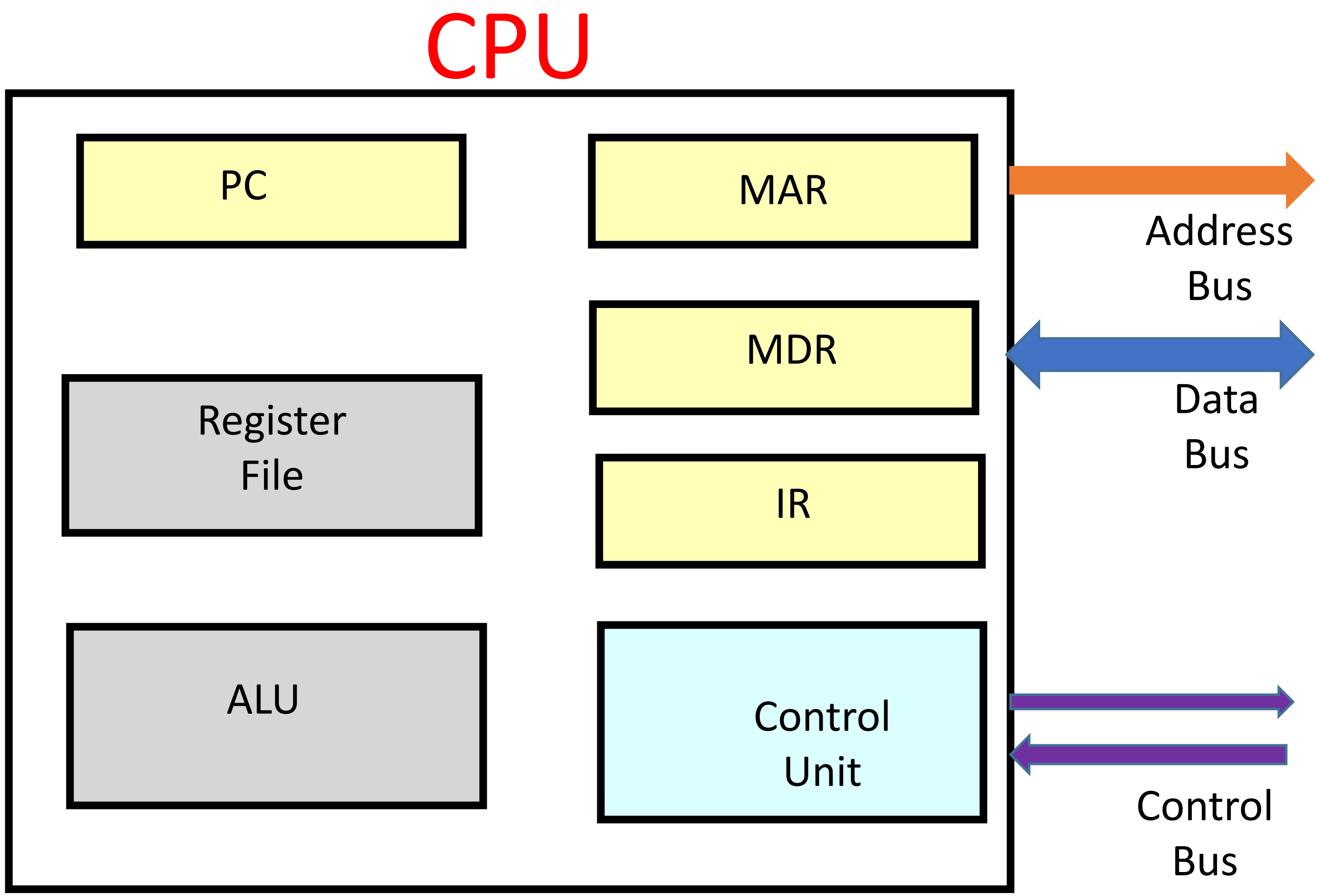
Memory



ADDRESS	CONTENTS
200H	11000101 (DATA-1)
201H	~~11000001~~ (DATA-2) 01101101
202H	11000111 (DATA-3)

```
MOV M2, AX  ;M2 = 201H
            ; [AX] = 01101101
```

Computer Components



Memory

ADDRESS	CONTENTS
000100100101 (125H)	**11001101** (Machine code of Instruction-1)
000100100110 (126H)	**10001101** (Machine code of Instruction-2)
000100100111 (127H)	**11101101** (Machine code of Instruction-3)
000100101000 (128H)	**11011101** (Machine code of Instruction-4)
200H	**11000101** (DATA-1)
201H	**11000001** (DATA-2)
202H	**11000111** (DATA-3)

PC-Program counter
MAR-Memory Address Register

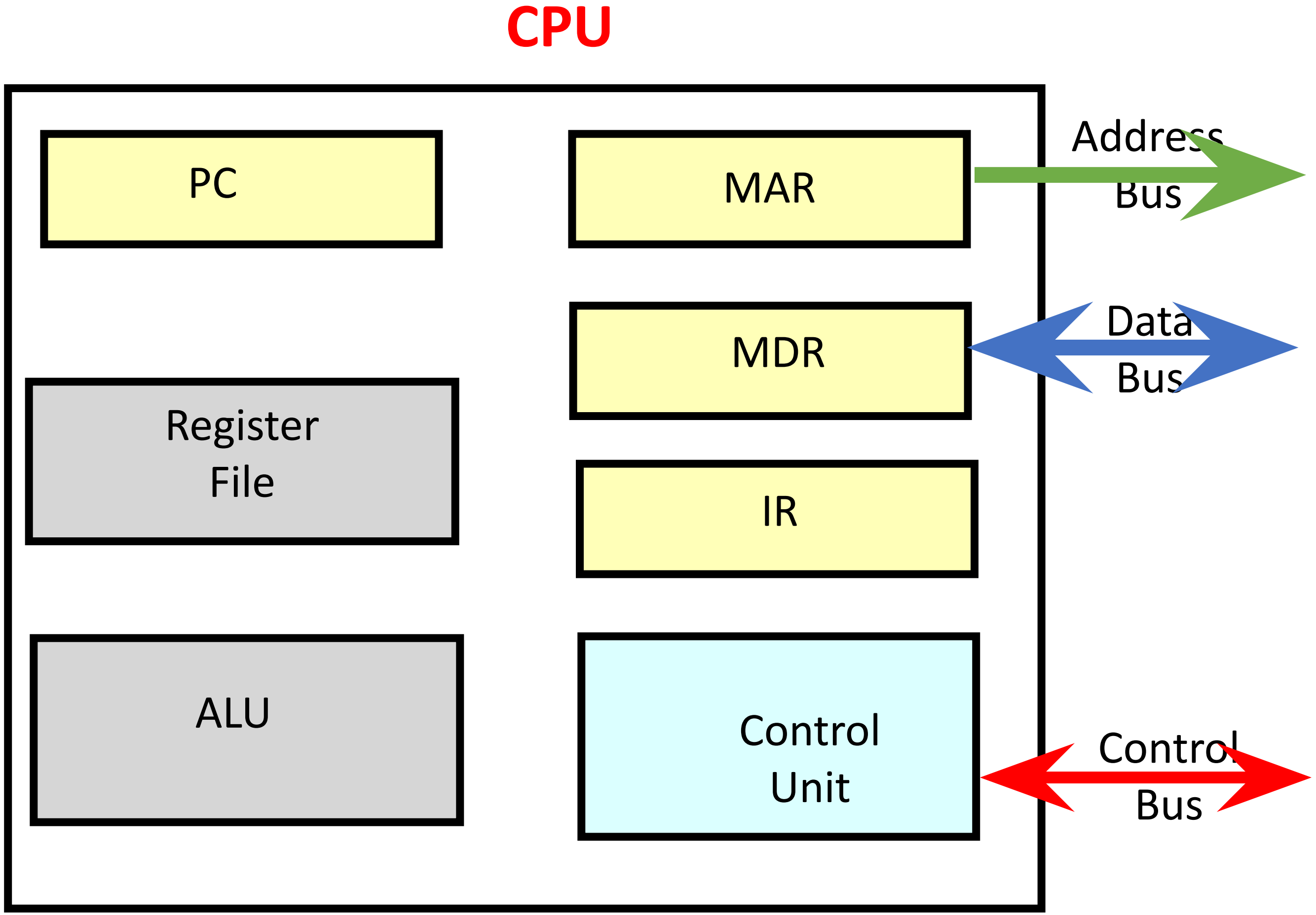
IR-Instruction Register
MDR-Memory Data Register

Control Unit decodes Instructions & Generates control signals

How does computer work?

User program

```
SUB AX, BX    ;0010101111000011
MOV CX, AX  ;1000101111001000
MOV DX, 0
```



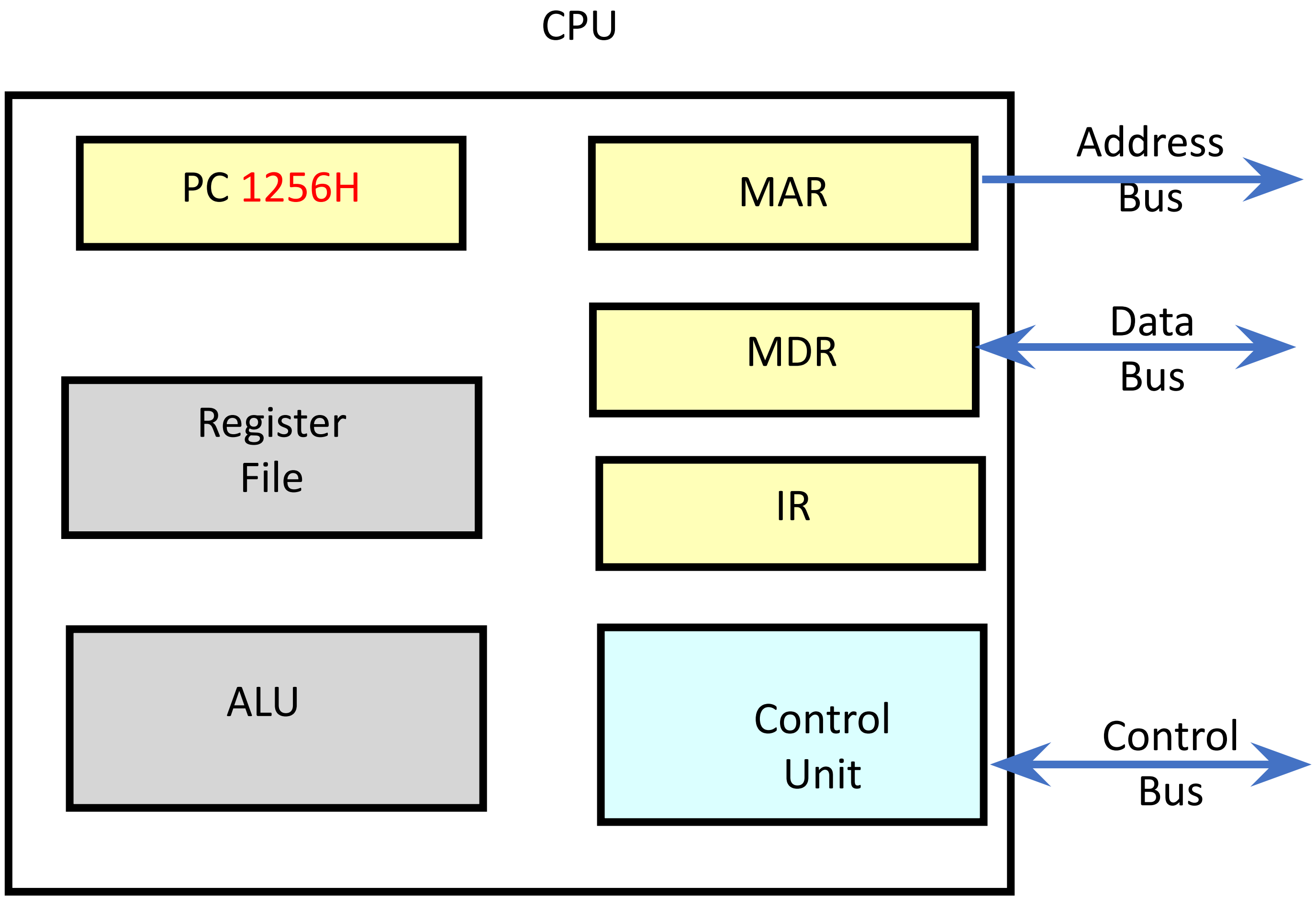
Stored in RAM(16 bits)

Address	Contents
0001001001010110 (1256H)	0010101111000011
0001001001010111 (1257H)	1000101111001000

STEP-1: PC is loaded with address of 1st instruction of program

User program

```
SUB AX, BX    ;0010101111000011
MOV CX, AX   ;1000101111001000
MOV DX, 0
```



RAM(16 bits)

Address	Contents
1256H	0010101111000011
1257H	1000101111001000
1258H	

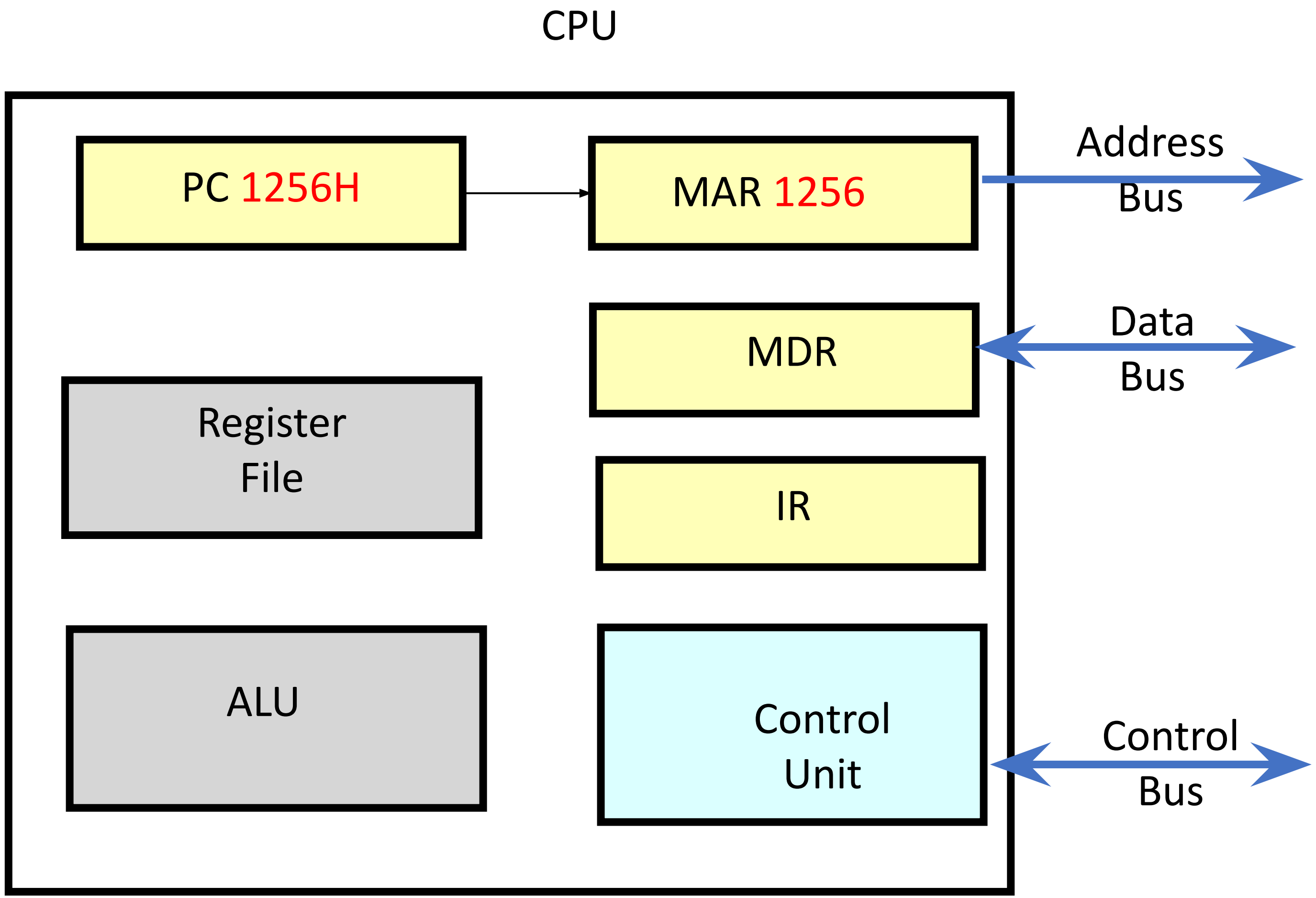
STEP-2: content of PC is loaded into MAR

User program

SUB AX, BX

MOV CX, AX

MOV DX, 0



RAM(16 bits)

Address	Contents
1256H	**0010101 11000011**
1257H	**1000101 11001000**
1258H	**1001111 11001011**

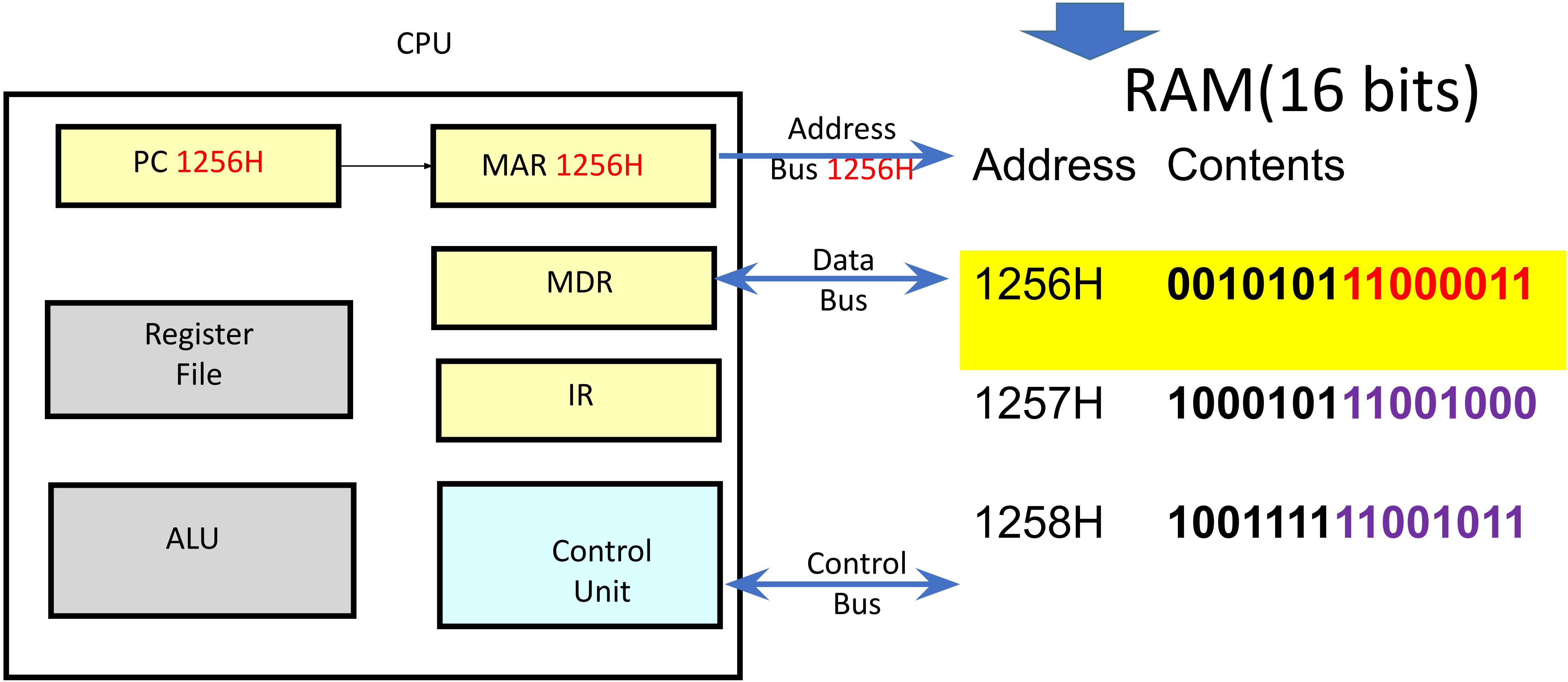
STEP-3: content of MAR is placed on Address bus and applied to Memory, as a result memory location 1256H is selected

User program

SUB AX, BX

MOV CX, AX

MOV DX, 0



Address	Contents
1256H	0010101**11000011**
1257H	1000101**11001000**
1258H	1001111**11001011**

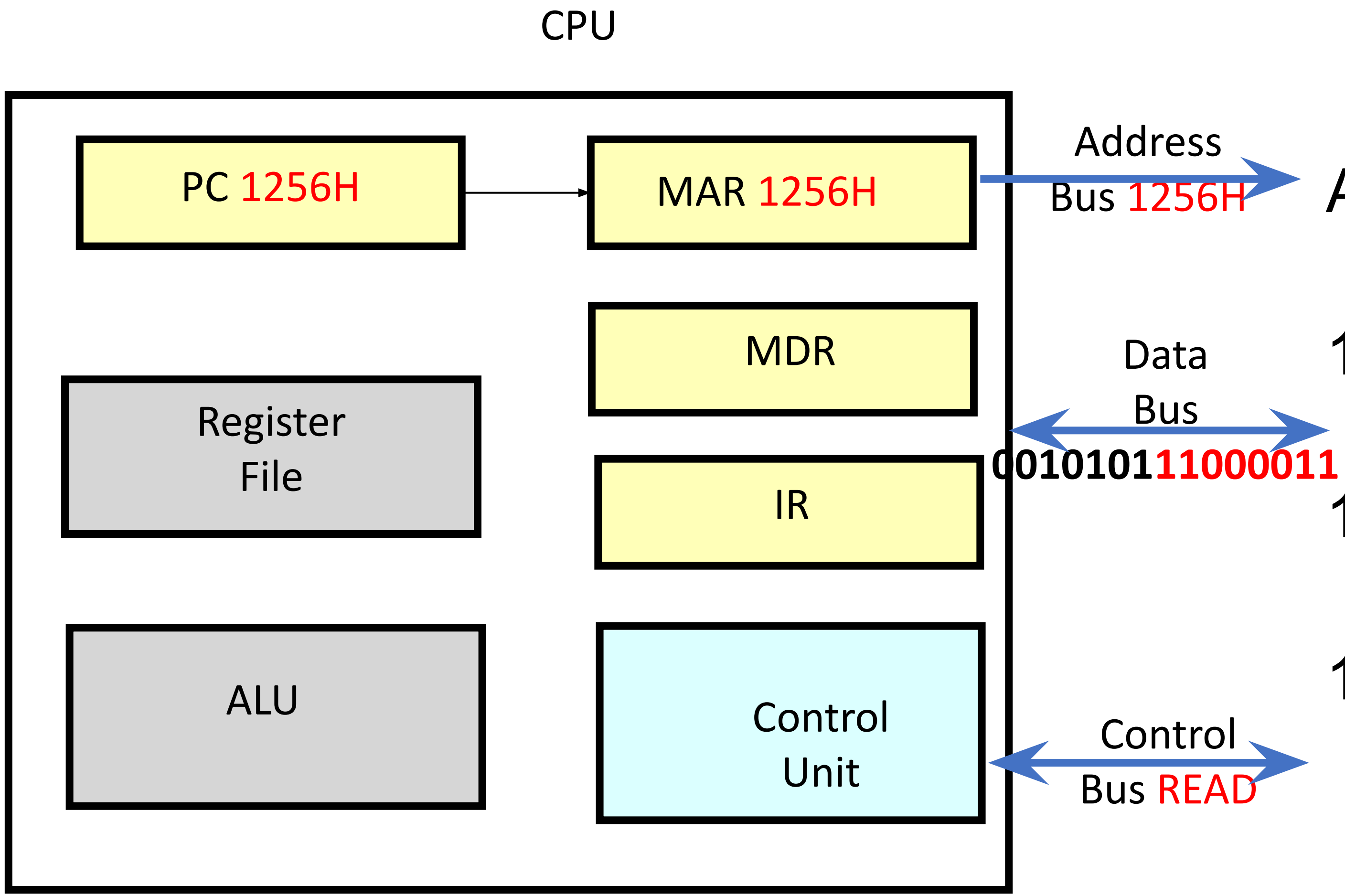
STEP-4: Control unit sends READ control signal to RAM, as a result machine code of 1st instruction is available on Data Bus

User program

SUB AX, BX

MOV CX, AX

MOV DX, 0



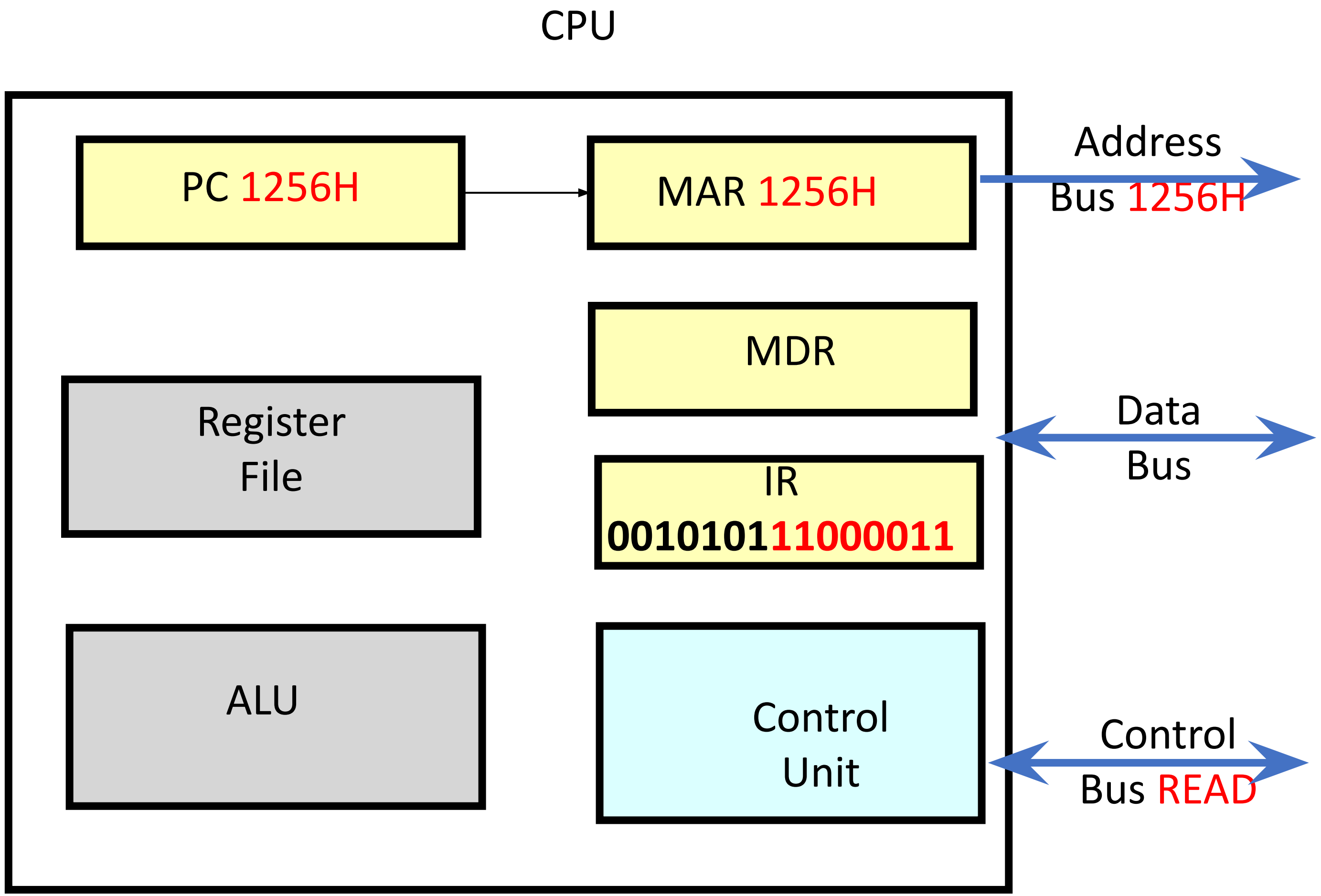
RAM(16 bits)

Address	Contents
1256H	1001010111100001
1257H	100010111001000
1258H	100111111001011

STEP-5: Machine code of 1st instruction is loaded into IR

User program

```
SUB AX, BX    ;0010101111000011
MOV CX, AX  ;1000101111001000
MOV DX, 0
```



RAM(16 bits)

Address	Contents
1256H	0010101111000011
1257H	1000101111001000
1258H	1001111111001011

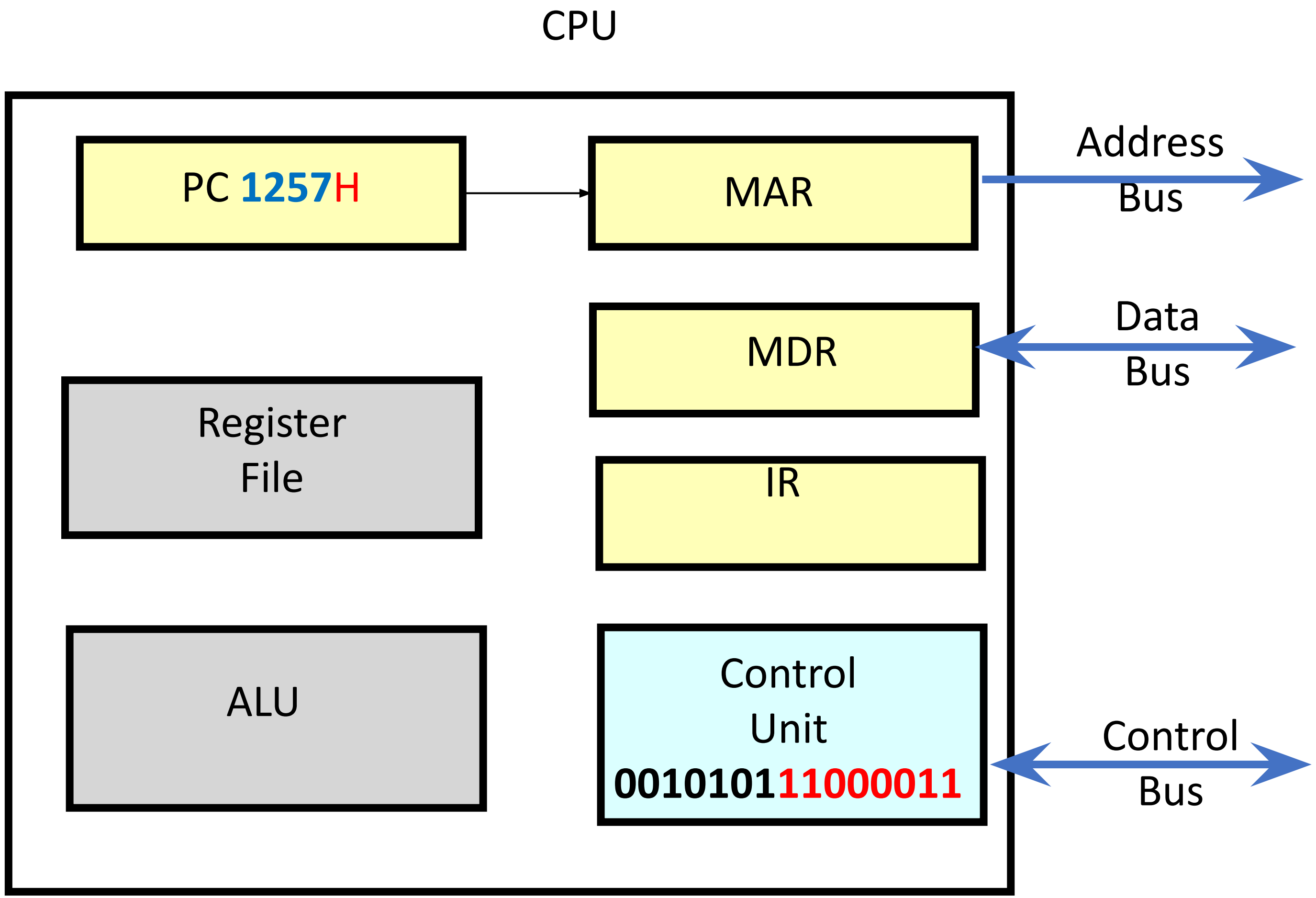
STEP-6: PC is incremented by 1 to point next instruction to be executed. Contents of IR is fed to Control Unit

User program

SUB AX, BX

MOV CX, AX

MOV DX, 0



RAM(16 bits)

Address	Contents
1256H	**0010101110000011**
1257H	**100010111001000**
1258H	**1001111111001011**

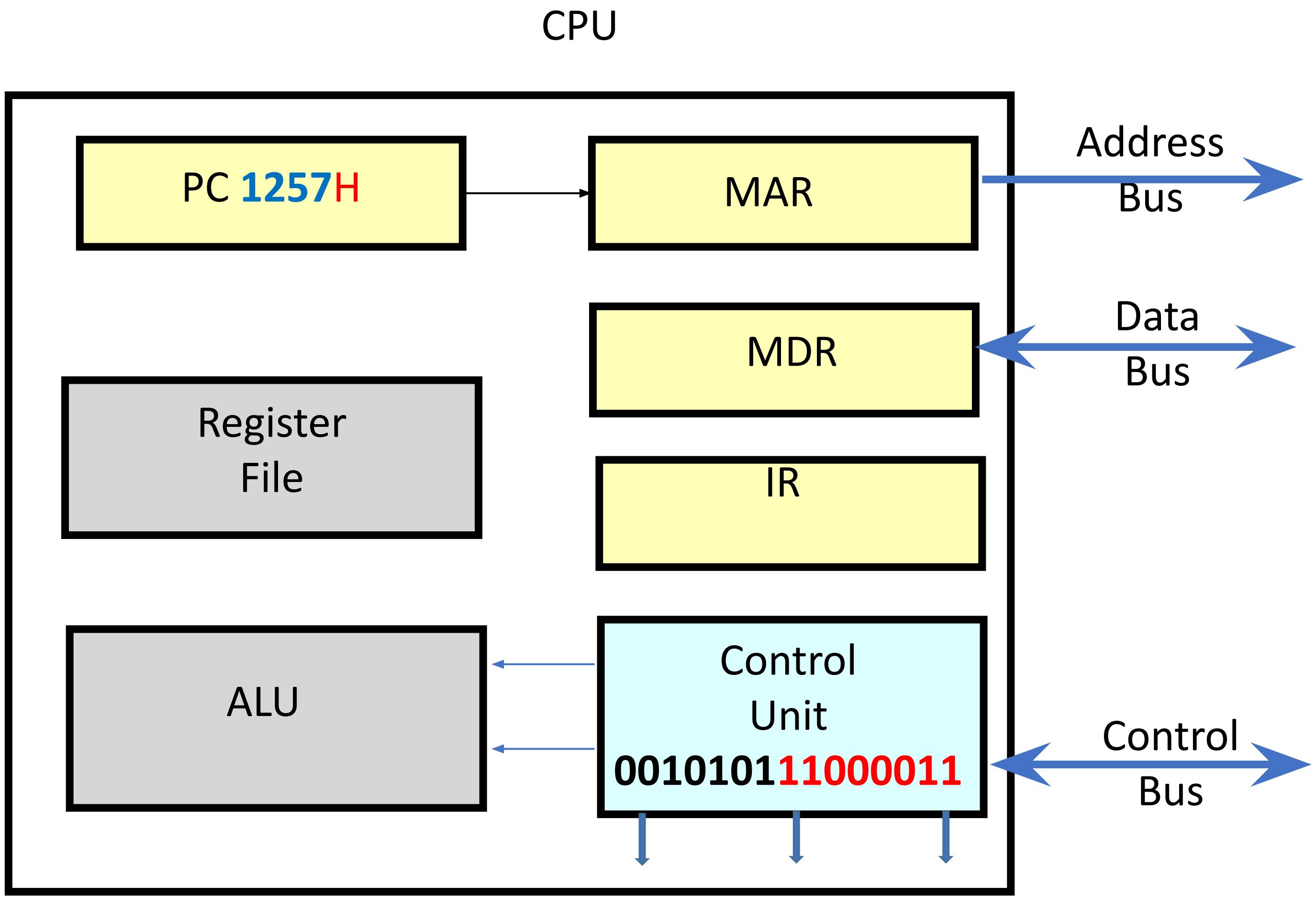
STEP-7: 1st instruction is decoded at control unit: control signals are generated to activate ALU for specific operation as per instruction

User program

SUB AX, BX

MOV CX, AX

MOV DX, 0



RAM(16 bits)

Address	Contents
1256H	001010111000011
1257H	100010111001000
1258H	100111111001011

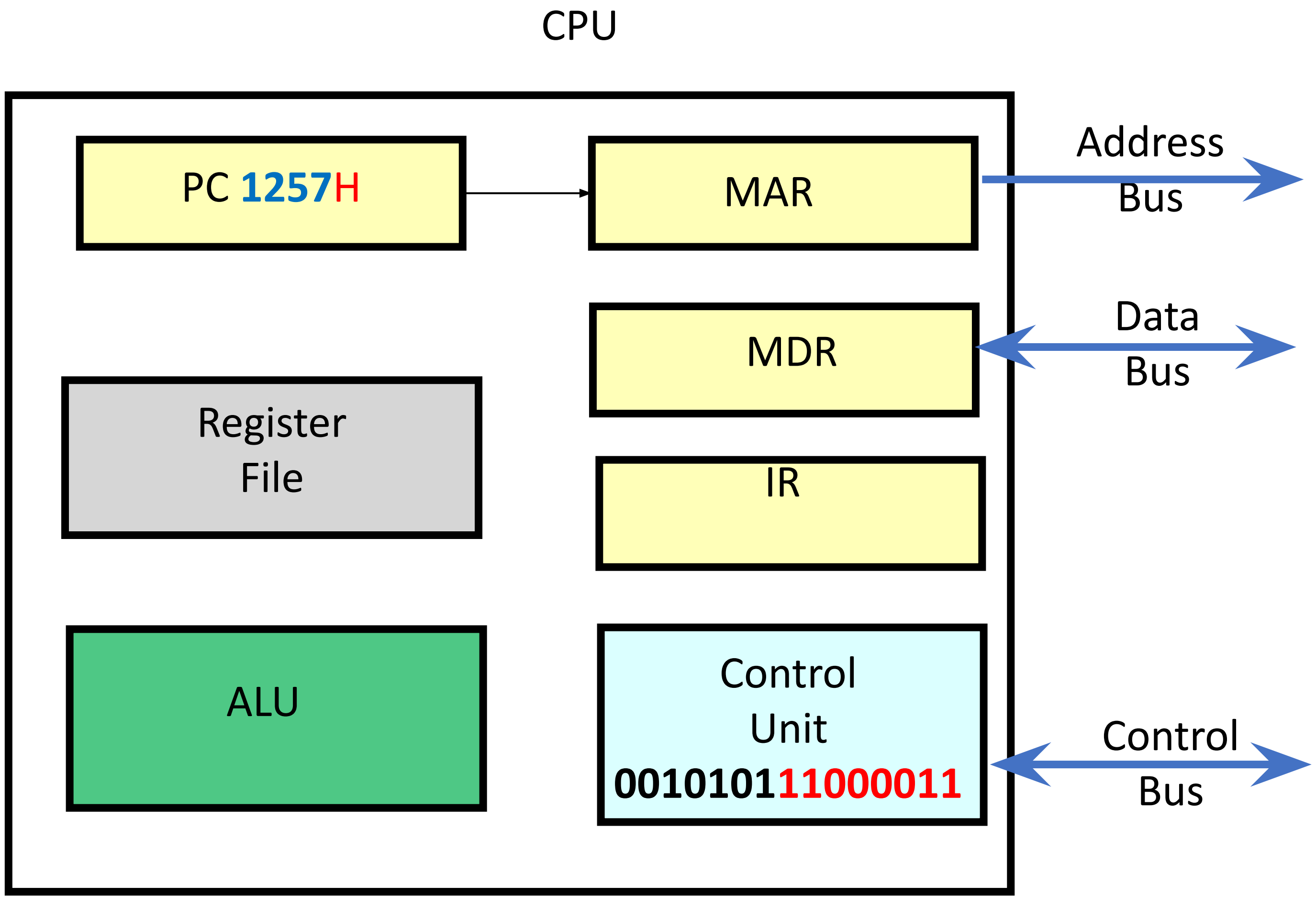
STEP-8: 1st instruction is Executed (that includes Data read from RAM followed by ALU operation, result stored as per instruction)

User program

SUB AX, BX

MOV CX, AX

MOV DX, 0



RAM(16 bits)

Address	Contents
1256H	**0010101 11000011**
1257H	**1000101 11001000**
1258H	**1001111 11001011**

STEPS-2:8 repeated for next instruction

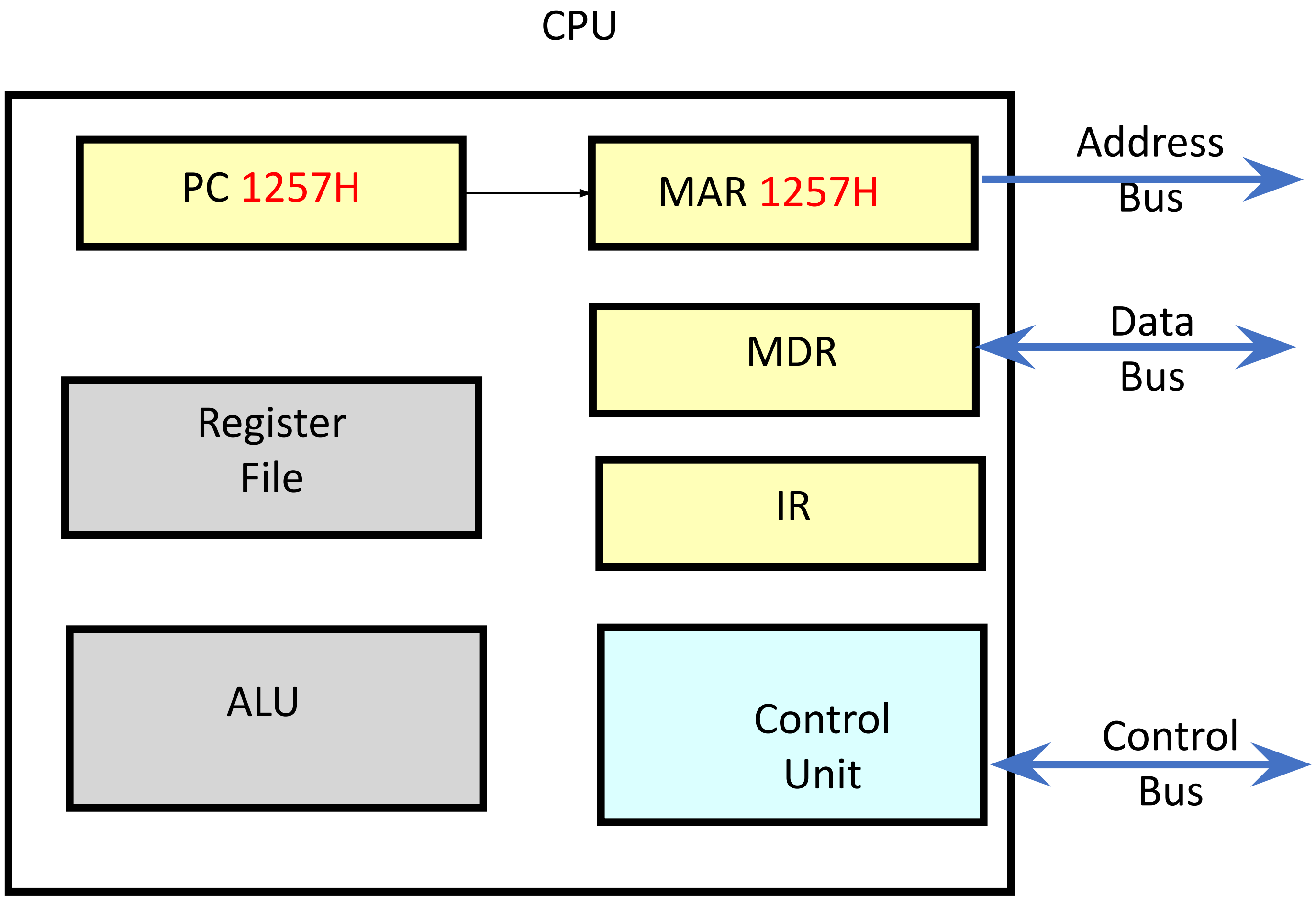
content of PC is loaded into MAR

User program

SUB AX, BX

MOV CX, AX

MOV DX, 0



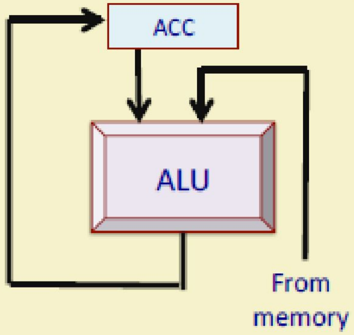
RAM(16 bits)

Address	Contents
1256H	0010101**11000011**
1257H	1000101**11001000**
1258H	1001111**11001011**

- **Accumulator based machine:**

```
LOAD    X       //  ACC = Mem[X]
ADD     Y       // ACC = ACC + Mem[Y]
STORE   Z       // Mem[Z] = ACC
```

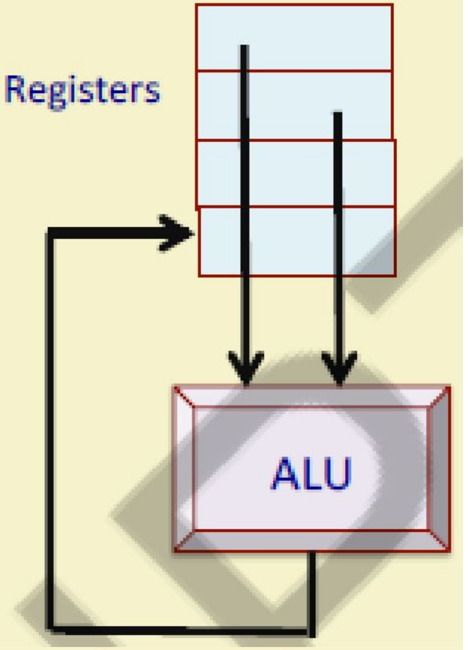
- All instructions assume that one of the operands (and also the result) is in a special register called accumulator.



- **Register-Register machine:**

```
LOAD    R1,X         // R1 = Mem[X]
LOAD    R2,Y         // R2 = Mem[Y]
ADD     R3,R1,R2   // R3 = R1 + R2
STORE   Z,R3         // Mem[Z] = R3
```

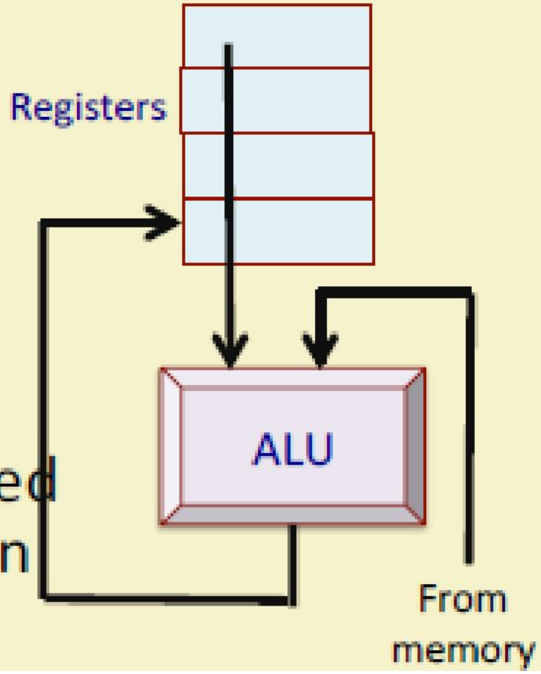
- Also called *load-store architecture*, as only LOAD and STORE instructions can access memory.



- **Register-Memory machine:**

```
LOAD    R2,X     // R2 = Mem[X]
ADD     R2,Y     // R2 = R2 + Mem[Y]
STORE   Z,R2     // Mem[Z] = R2
```

- One of the operands is assumed to be in register and another in memory.



Instruction Formats (3-operand fields)

ADD M, D1, D2

[M] ← D1+ D2

Binary code for ALU operation (Op Code)	Memory address to store result (Binary)	Data -1 (Binary)	Data-2 (Binary)

ADD M1, M2, M3

[M1] ← [M2]+ [M3]

Binary code for ALU operation (Op Code)	Memory address to store result (Binary)	Memory address of Data -1 (Binary)	Memory address of Data-2 (Binary)

ADD R1, R2, R3

R1 ← R2+ R3

Binary code for ALU operation (Op Code)	CPU Register to store result (Binary)	CPU register containing Data -1 (Binary)	CPU register containing Data -1 (Binary)

Instruction Formats(2-operand field)

ADD M1, M2

[M1] ← [M1]+ [M2]

Binary code for ALU operation (Op Code)

CPU register or memory address that contains Data-1 (Binary)
Result is stored in same CPU register or memory address after operation

CPU register or memory address that contains Data-2 (Binary)

ADD R1, R2

R1 ← R1 + R2

ADD M, R1

[M] ← R1 + [M]

Instruction Formats(one-operand field)

ADD M

$$AC \leftarrow [M] + AC$$

Binary code for ALU operation
(Op Code)

CPU register or memory address that contains Data-2
(Binary)

Here Data-1 should be loaded into a predefined register, namely Accumulator (AC) prior to use this instruction. Moreover, the result is stored into Accumulator (AC) as well.

Interestingly, the Accumulator (AC) is not explicitly indicated in the Instruction sub-field!

ADD R1

$$AC \leftarrow R1 + AC$$

Instruction Formats(Opcode only!)

CLC Clear carry Flag bit

Binary code for ALU /special operation

Data to be used should be initially loaded into default CPU register. Result is also stored there as well.

Format of an Instruction

OP CODE	OPERAND FIELD-1	OPERAND FIELD-2	OPERAND FIELD-3
4-BITS	4-BITS	4-BITS	4-BITS

OP CODE	OPERAND FIELD-1	OPERAND FIELD-2
4-BITS	4-BITS	4-BITS

OP CODE	OPERAND FIELD-1	OPERAND FIELD-2
8-BITS	16-BITS	16-BITS

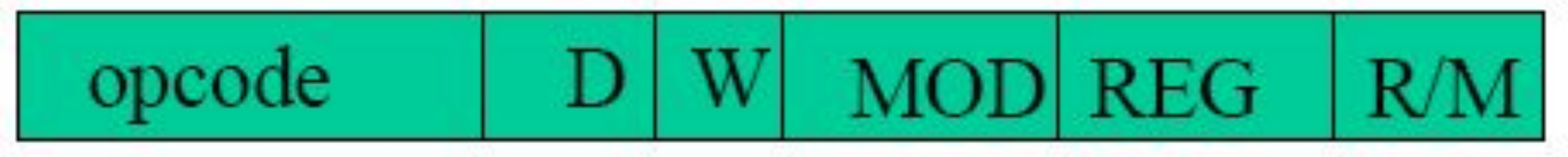
Machine Codes

- An instruction can be coded with 1 to 6 bytes
- Byte 1 contains three kinds of information
  - Opcode field (6 bits) specifies the operation (add, subtract, move)
  - Register Direction Bit (D bit) Tells the register operand in REG field in byte 2 is source or destination operand

    1: destination 0: source

  -Data Size Bit (W bit) Specifies whether the operation will be performed on 8-bit or 16-bit data

    0: 8 bits 1: 16 bits



opcode	D	W	MOD	REG	R/M
15 ------- 10	9	8	7 ---- 6	5 ---- 3	2 ---- 0

Example: MOV BL, AL

(machine code: 10001000 11000011)

(Hexcode: 88 C3H)

opcode	D	W	MOD	REG	R/M
1 0 0 0 1 0	0	0	1 1	0 0 0	0 1 1

Opcode = 100010 → MOV data transfer

D = 0 → AL is source operand

W = 0 → 8-bit data transfer

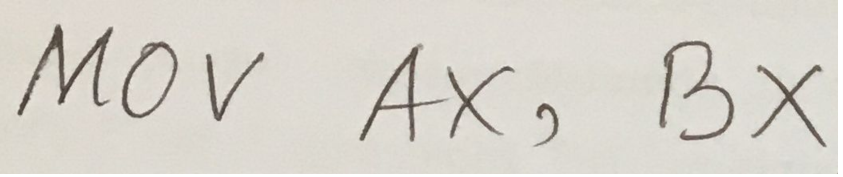
Therefore byte 1 is $10001000_2=88_{16}$

MOD = 11 → register mode

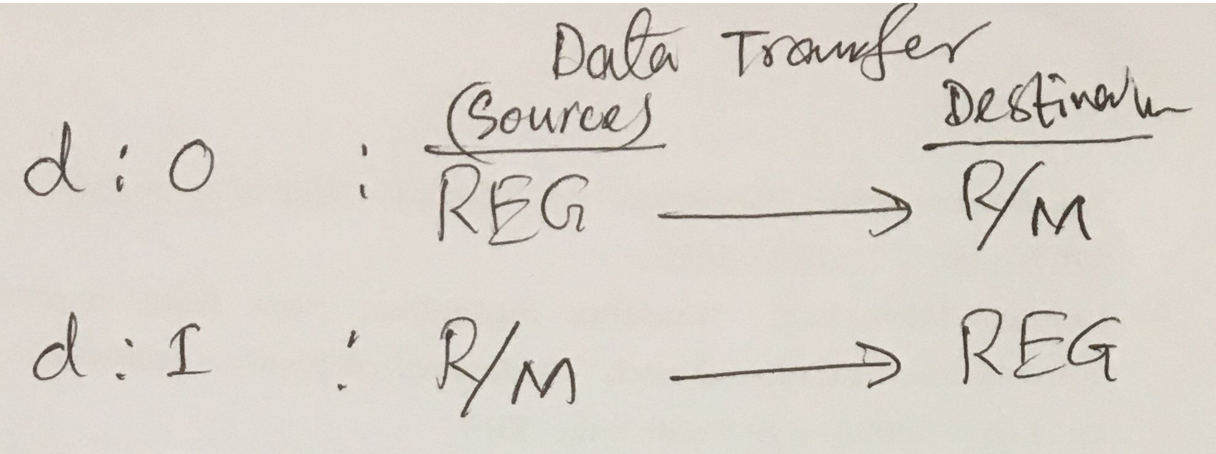
REG = 000 → code for AL

R/M = 011 → destination is BL

Therefore Byte 2 is $11000011_2=C3_{16}$

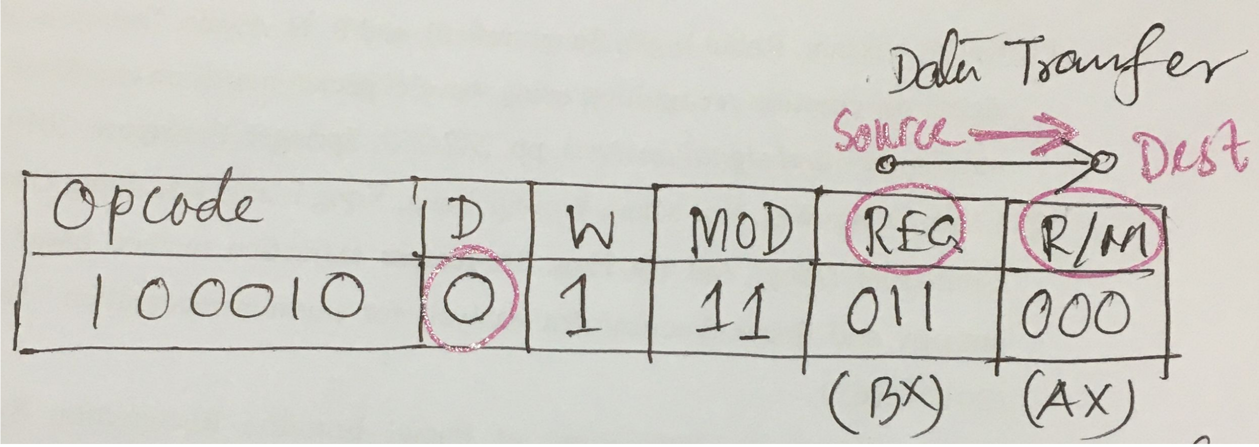


MOV AX, BX



Data Transfer

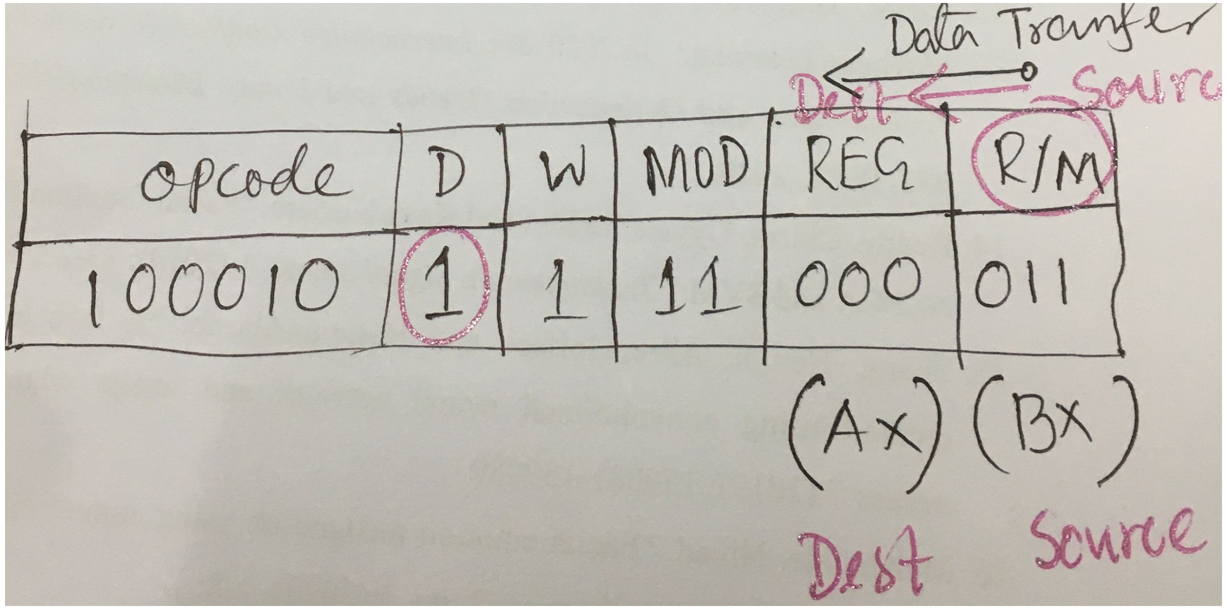
		(Source)		Destination
d : 0	:	REG	⟶	R/M
d : 1	:	R/M	⟶	REG



Data Transfer

Source → Dest

Opcode	D	W	MOD	REG	R/M
100010	0	1	11	011	000
				(BX)	(AX)



Data Transfer

Dest ← Source

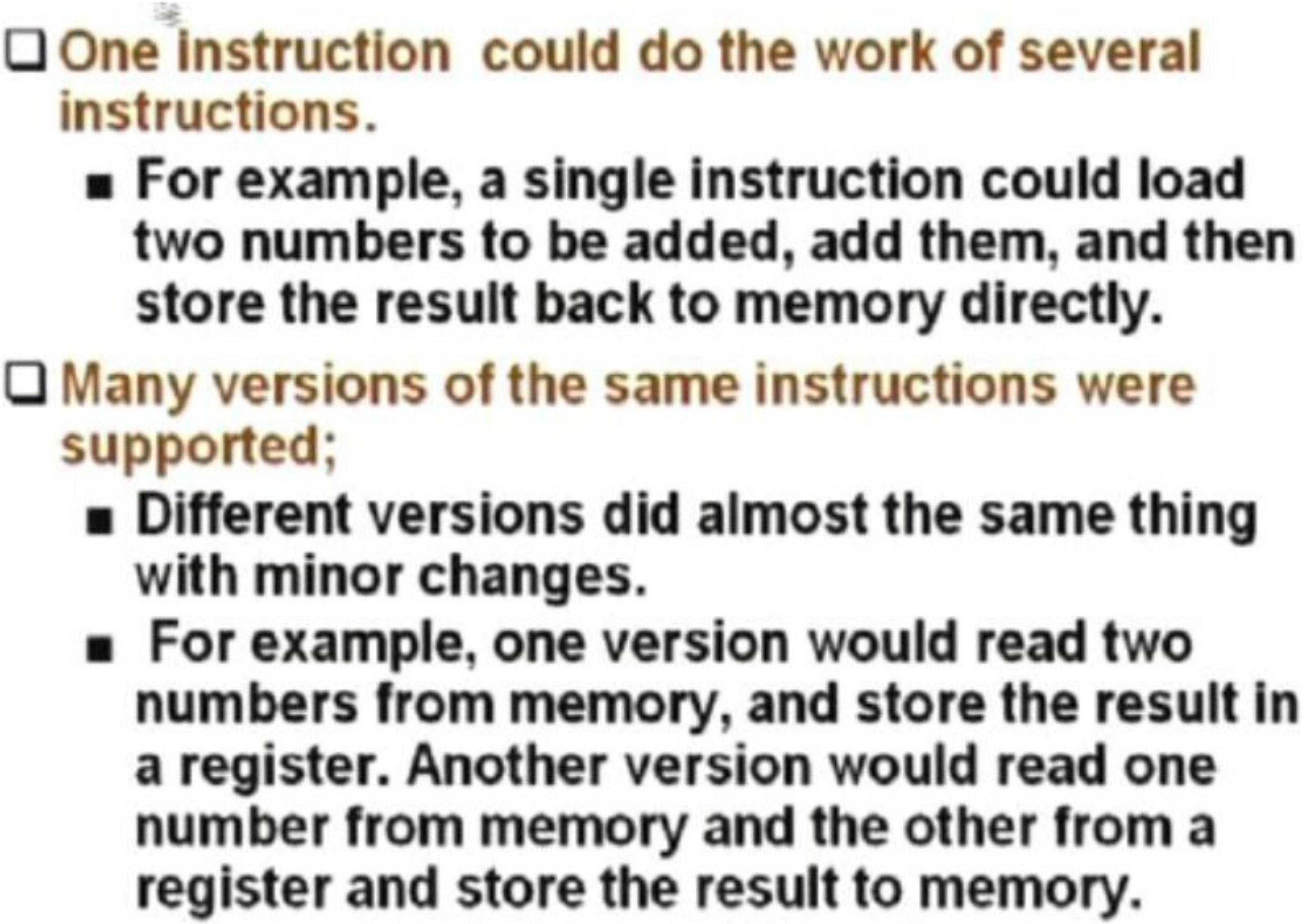
Opcode	D	W	MOD	REG	R/M
100010	1	1	11	000	011
				(AX)	(BX)
				Dest	Source

Instruction Set Architecture

- RISC (Reduced Instruction Set Computer) Architectures
  - **Memory accesses are restricted to load and store instruction, and data manipulation instructions are register to register.**
  - **Addressing modes are limited in number.**
  - **Instruction formats are all of the same length.**
  - **Instructions perform elementary operations**

- CISC (Complex Instruction Set Computer) Architectures
  - **Memory access is directly available to most types of instruction.**
  - **Addressing mode are substantial in number.**
  - **Instruction formats are of different lengths.**
  - **Instructions perform both elementary and complex operations.**

CISC features



- □ One instruction could do the work of several instructions.
  - ■ For example, a single instruction could load two numbers to be added, add them, and then store the result back to memory directly.
- □ Many versions of the same instructions were supported;
  - ■ Different versions did almost the same thing with minor changes.
  - ■ For example, one version would read two numbers from memory, and store the result in a register. Another version would read one number from memory and the other from a register and store the result to memory.

RISC Processors

Background of RISC

- IBM RISC technology originated in 1974 in a project to design a large telephone-switching network capable of handing an average of three hundred calls per second. With an approximate 20 000 instructions per call and Stringent real-time response requirements, the performance target was 12 million instructions per second (MIPS).
- This specialized application required a very fast processor, but did not have to perform computed instructions and had little demand for floating-point calculations. Other than moving data between registers and memory, the machine had to be able to add, combine fields extracted from several registers, perform branches, and carry out input/output operations.

Background of RISC

- When the telephone project was terminated in 1975, the machine itself had not been built, but the design had progressed to the point where it seemed to be an excellent basis for a general-purpose, high-performance miniprocessor. The attractiveness of the processor design stemmed from projections that it would be able to compute at high speed relative to its cost in a variety of application areas.
- The most important features of the telephone switching machine which contributed to its low cost/performance ratio were 1) separate instruction and data caches, allowing a much higher bandwidth between memory and CPU; 2) no arithmetic operations to storage, which greatly simplified the pipeline; and 3) uniform instruction length and simplicity of design, making possible a very short cycle time: ten levels of logic. (For example, all register-to-register operations executed in one cycle.)

Background of RISC

- John Cocke and his colleagues developed simpler ISAs and compilers for minicomputers. As an experiment, they retargeted their research compilers to use only the simple register-register operations and load-store data transfers of the IBM 360 ISA, avoiding the more complicated instructions. They found that programs ran up to three times faster using the simple subset.

- Emer and Clark found 20% of the VAX instructions needed 60% of the microcode and represented only 0.2% of the execution time.

RISC: *Reduced Instruction Set Computer*

A type of microprocessor architecture that utilizes a small, highly-optimized set of instructions

History: The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy.

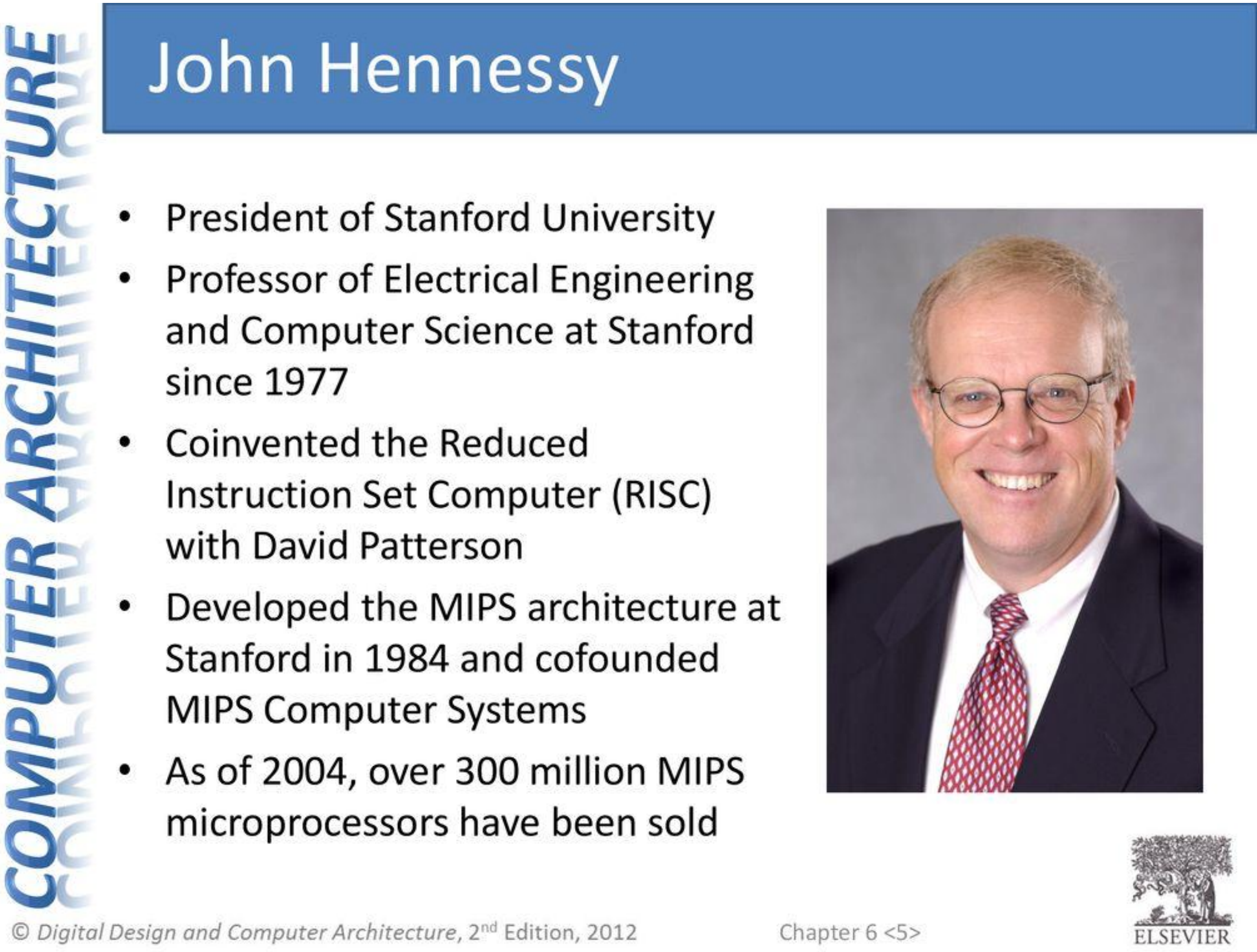
Design features of RISC processors:

- *one cycle execution time*: RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called pipelining;
- *pipelining*: a technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;
- *large number of registers*: RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

MIPS

The MIPS processor was developed as part of a VLSI research program at Stanford University in the early 80s.

Professor John Hennessy started the development of MIPS with a brainstorming class for graduate students. The readings and idea sessions helped launch the development of the processor which became one of the first RISC processors, with IBM and Berkeley developing processors at around the same time.



COMPUTER ARCHITECTURE

John Hennessy

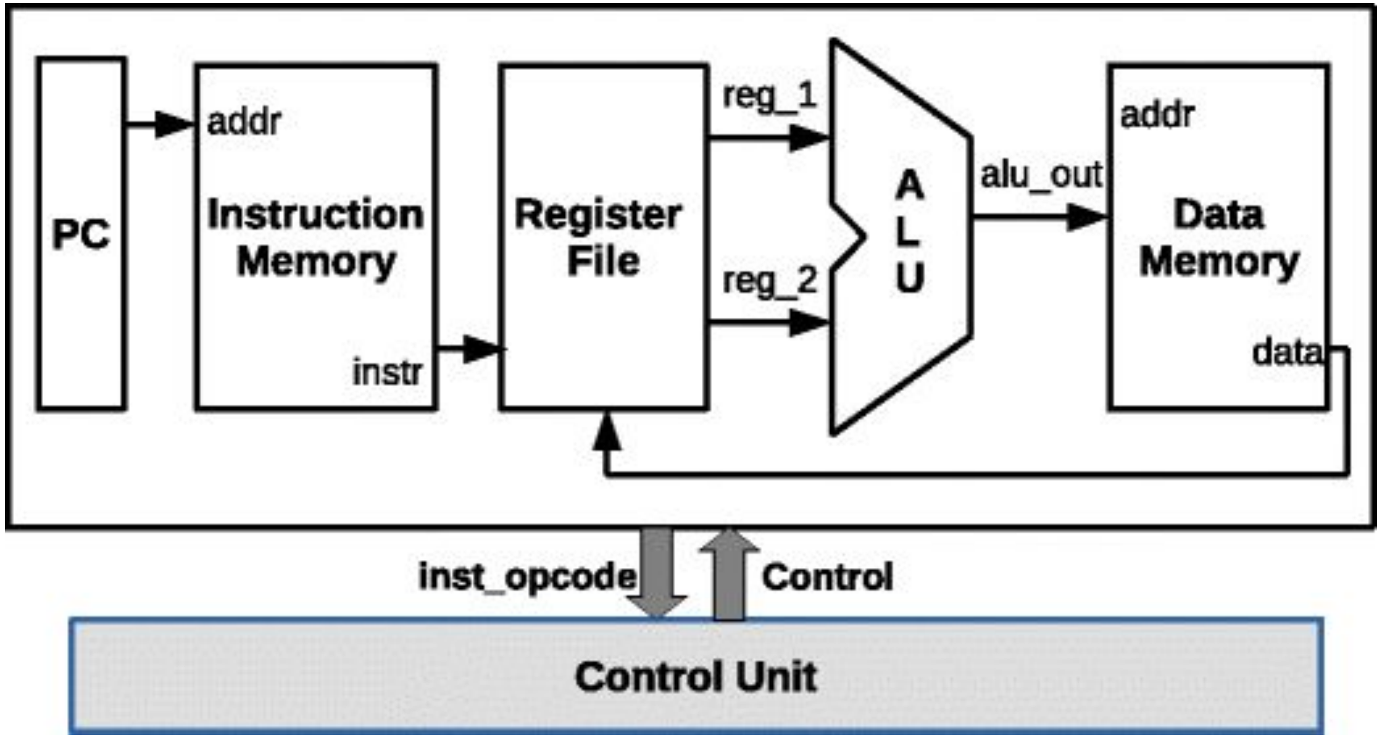
- President of Stanford University
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Coinvented the Reduced Instruction Set Computer (RISC) with David Patterson
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold

© *Digital Design and Computer Architecture*, 2nd Edition, 2012 Chapter 6 <5>

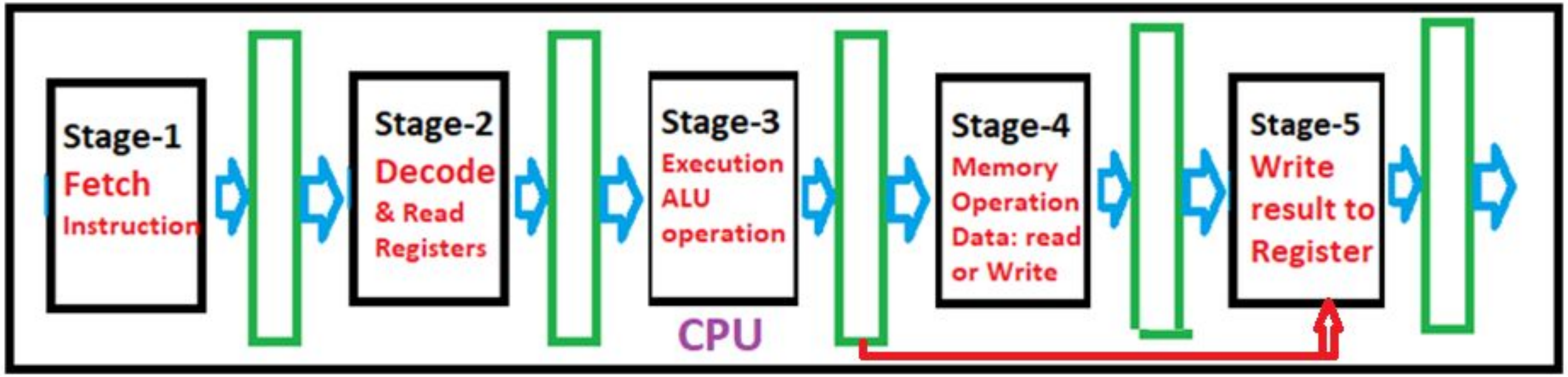
ELSEVIER

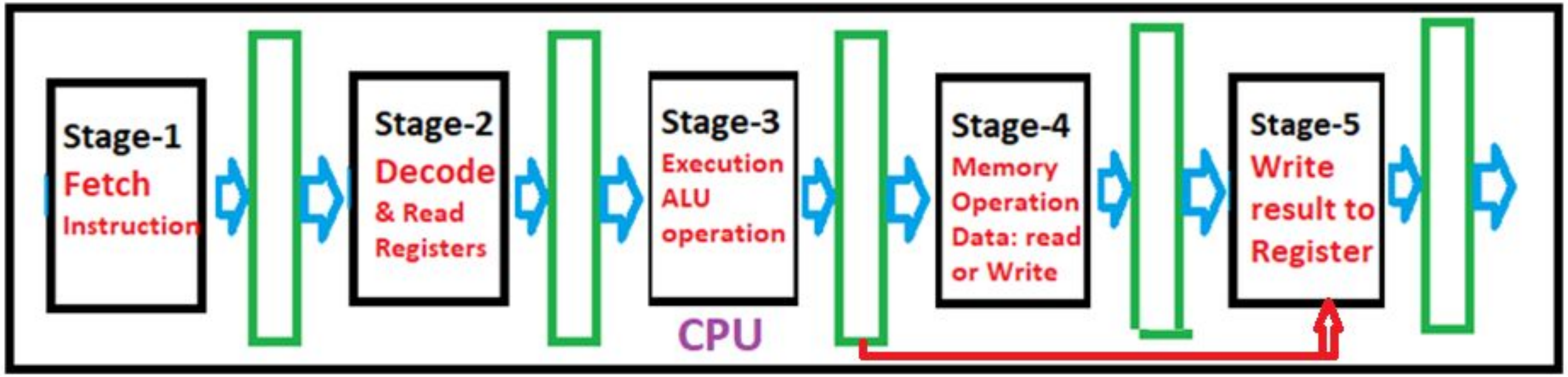
MIPS Architecture

- The Stanford research group had a strong background in compilers, which led them to develop a processor whose architecture would represent the lowering of the compiler to the hardware level, as opposed to the raising of hardware to the software level, which had been a long running design philosophy in the hardware industry.
- Thus, the MIPS processor implemented a smaller, simpler instruction set. Each of the instructions included in the chip design ran in a single clock cycle. The processor used a technique called pipelining to more efficiently process instructions.
- MIPS used 32 registers, each 32 bits wide (a bit pattern of this size is referred to as a *word*).



- Instruction cycle of MIPS processor was subdivided into **five stages**:
- **Instruction Fetch (IF)**
- **Instruction Decode (ID) and Register Read**
- **Execution (EXE)**
- **Memory read/write(MEM)**
- **Write Back** result **(WB)** to Registers





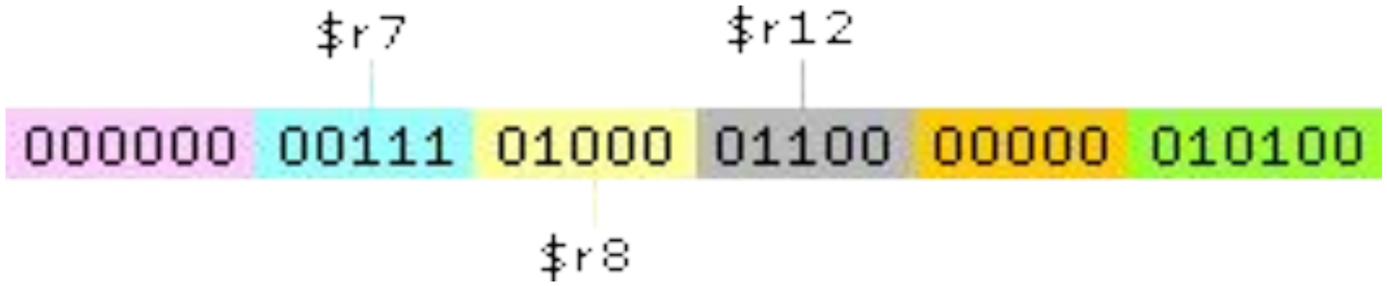
Instruction\ clock cycles	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WB	
Instruction-4				IF	ID	EX	MA	
Instruction-5					IF	ID	EX	
Instruction-6						IF	ID	
Instruction-7							IF	

- Single cycle implementation

Instruction Set

- The MIPS instruction set consists of about **111 total instructions**, each represented in 32 bits. An example of a MIPS instruction is below:

add $r12, $r7, $r8,



- Three-operand arithmetical and logical instructions and all instructions are 32-bit
- Arithmetical and logical instructions are Register based (operands in registers and result will be stored in register)
- 32 general-purpose registers of 32-bits each
- MIPS addition instruction. The instruction tells the processor to compute the sum of the values in registers 7 and 8 and store the result in register 12. The dollar signs are used to indicate an operation on a register. The colored binary representation on the right illustrates the 6 fields of a MIPS instruction. The processor identifies the type of instruction by the binary digits in the first and last fields. In this case, the processor recognizes that this instruction is an addition from the zero in its first field and the 20 in its last field.
- The operands are represented in the blue and yellow fields, and the desired result location is presented in the fourth (purple) field. The orange field represents the *shift amount*, something that is not used in an addition operation.

The instruction set consists of about 111 total instructions. A variety of basic instructions, including:

21 arithmetic instructions (+, -, *, /, %)

8 logic instructions (&, |, ~)

8 bit manipulation instructions

12 comparison instructions (>, <, =, >=, <=, ¬)

25 branch/jump instructions

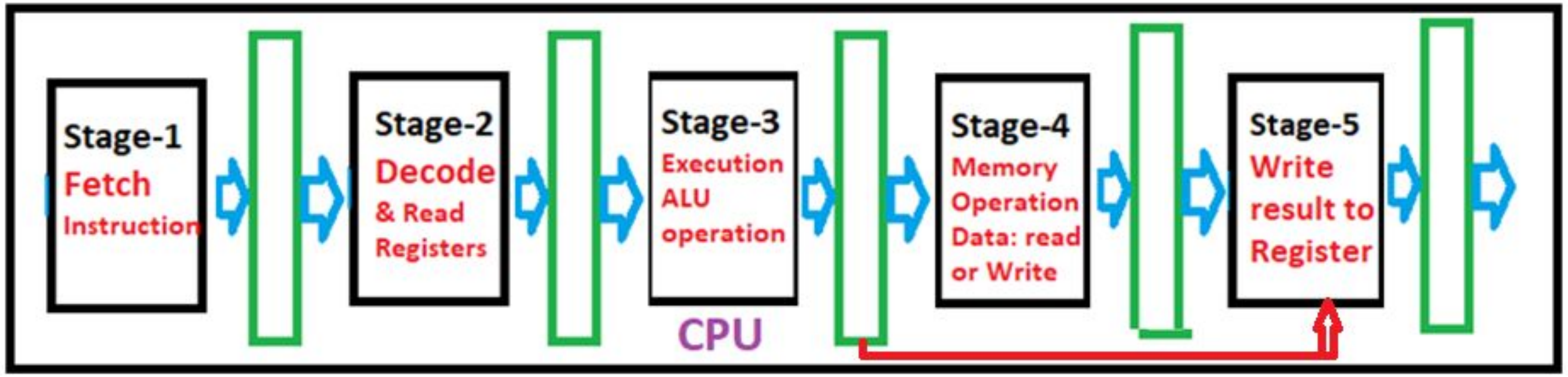
15 load instructions

10 store instructions

8 move instructions

4 miscellaneous instructions

How Pipelining Works


Stage-1
Fetch
Instruction
Stage-2
Decode
& Read
Registers
Stage-3
Execution
ALU
operation
CPU
Stage-4
Memory
Operation
Data: read
or Write
Stage-5
Write
result to
Register

Instruction\ clock cycles	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WB	
Instruction-4				IF	ID	EX	MA	
Instruction-5					IF	ID	EX	
Instruction-6						IF	ID	
Instruction-7							IF	

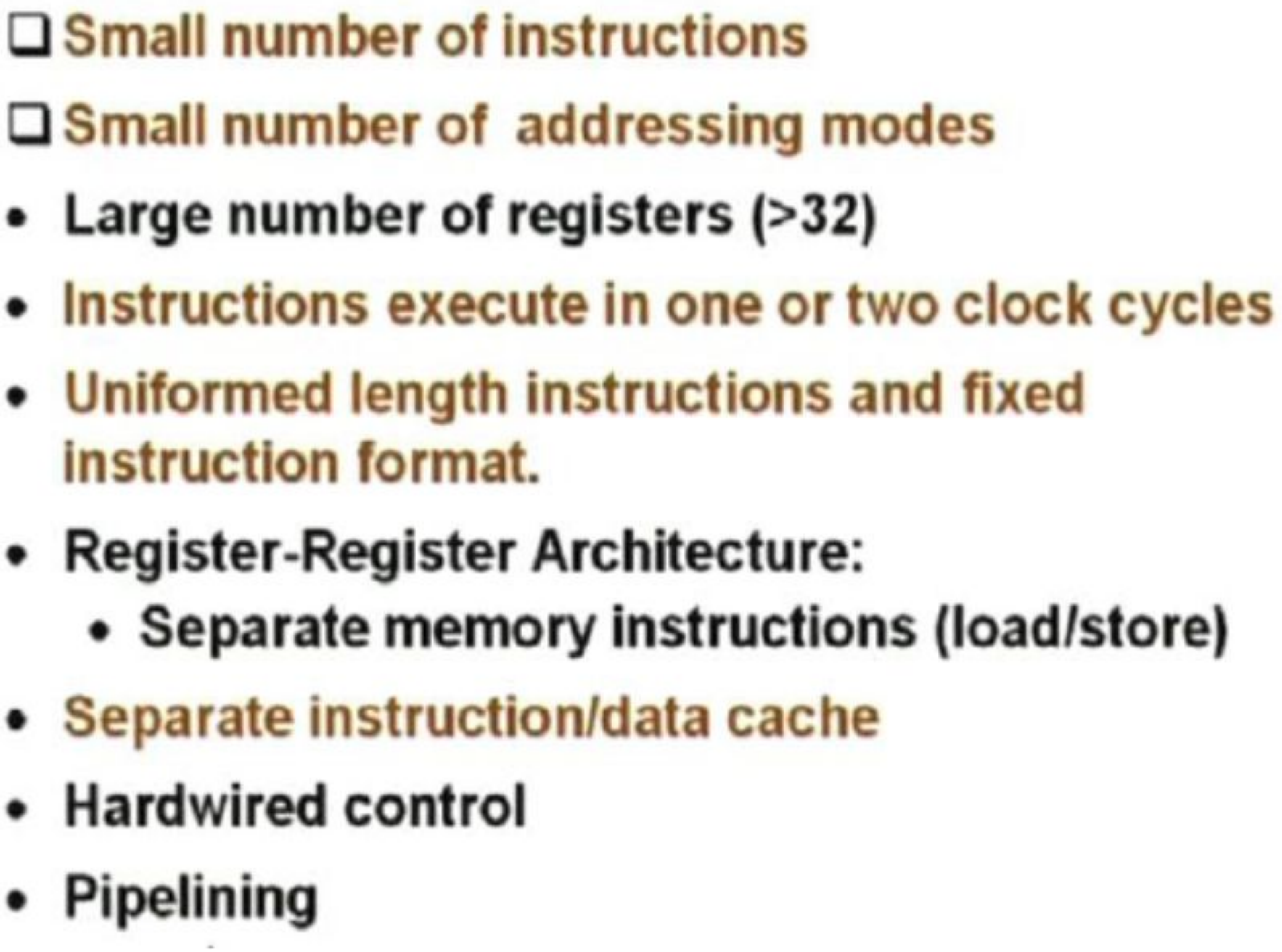
Pipelining, a standard feature in RISC processors, is much like an assembly line. Because the processor works on different steps of the instruction at the same time, more instructions can be executed in a shorter period of time.

CPI = 1

Non-pipelined processor (CPI = 5)

Instruction/Clock cycles	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Instruction-1	IF	ID	EX	MEM	WB										
Instruction-2						IF	ID	EX	MEM	WB					
Instruction-3											IF	ID	EX	MEM	WB

Features of RISC Processors

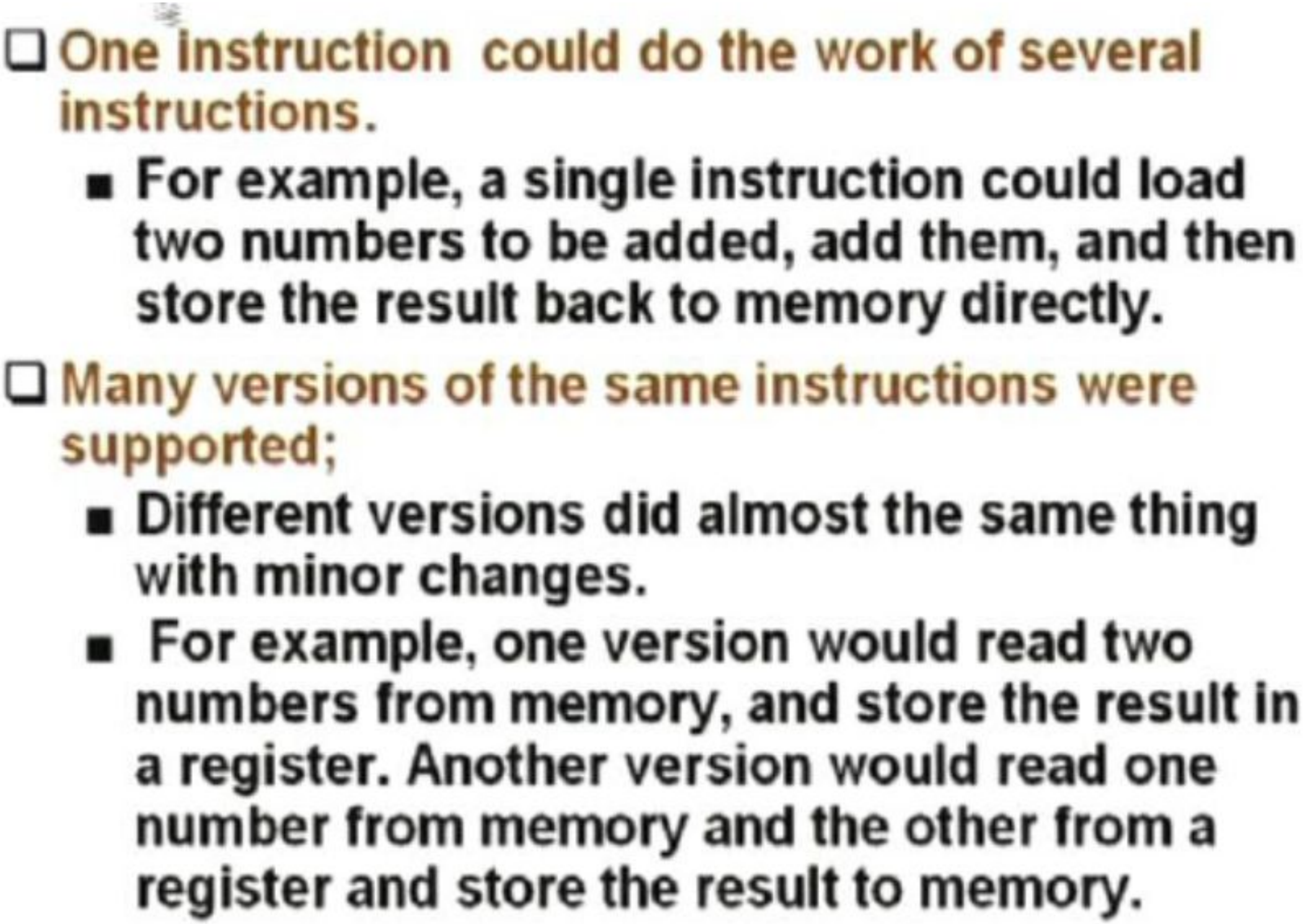


□ Small number of instructions

□ Small number of addressing modes

- Large number of registers (>32)
- Instructions execute in one or two clock cycles
- Uniformed length instructions and fixed instruction format.
- Register-Register Architecture:
  - Separate memory instructions (load/store)
- Separate instruction/data cache
- Hardwired control
- Pipelining

CISC features



□ One instruction could do the work of several instructions.

- ■ For example, a single instruction could load two numbers to be added, add them, and then store the result back to memory directly.

□ Many versions of the same instructions were supported;

- ■ Different versions did almost the same thing with minor changes.
- ■ For example, one version would read two numbers from memory, and store the result in a register. Another version would read one number from memory and the other from a register and store the result to memory.

RISC vs CISC

- The formula for processor performance:

Time/Program = Instructions /Program × (*Clock cycles*) */Instruction × Time/* (*Clock cycle*)

- DEC engineers later showed that the more complicated CISC ISA executed about 75% of the number instructions per program as RISC (the first term), but in a similar technology CISC executed about five to six more clock cycles per instruction (the second term), making RISC microprocessors approximately 4× faster.

RISC Pipelines

A RISC processor pipeline operates in much the same way, although the stages in the pipeline are different. While different processors have different numbers of steps, they are basically variations of these five, used in the MIPS R3000 processor:

- fetch instructions from memory
- read registers and decode the instruction
- execute the instruction or calculate an address
- access an operand in data memory
- write the result into a register

If you glance back at the diagram of the laundry pipeline, you'll notice that although the washer finishes in half an hour, the dryer takes an extra ten minutes, and thus the wet clothes must wait ten minutes for the dryer to free up. Thus, the length of the pipeline is dependent on the length of the longest step. Because RISC instructions are simpler than those used in pre-RISC processors (now called CISC, or Complex Instruction Set Computer), they are more conducive to pipelining. While CISC instructions varied in length, RISC instructions are all the same length and can be fetched in a single operation. Ideally, each of the stages in a RISC processor pipeline should take 1 clock cycle so that the processor finishes an instruction each clock cycle and averages one cycle per instruction (CPI).

Pipeline Problems

In practice, however, RISC processors operate at more than one cycle per instruction. The processor might occasionally stall a result of data dependencies and branch instructions.

- A data dependency occurs when an instruction depends on the results of a previous instruction. A particular instruction might need data in a register which has not yet been stored since that is the job of a preceding instruction which has not yet reached that step in the pipeline.

For example:

```
add $r3, $r2, $r1
add $r5, $r4, $r3
```

In this example, the first instruction tells the processor to add the contents of registers r1 and r2 and store the result in register r3.

The second instructs it to add r3 and r4 and store the sum in r5. We place this set of instructions in a pipeline. When the second instruction is in the second stage, the processor will be attempting to read r3 and r4 from the registers. Remember, though, that the first instruction is just one step ahead of the second, so the contents of r1 and r2 are being added, but the result has not yet been written into register r3. The second instruction therefore cannot read from the register r3 because it hasn't been written yet and must wait until the data it needs is stored. Consequently, the pipeline is stalled and a number of empty instructions (known as *bubbles* go into the pipeline. Data dependency affects long pipelines more than shorter ones since it takes a longer period of time for an instruction to reach the final register-writing stage of a long pipeline.

data dependency

- MIPS' solution to this problem is code reordering. If, as in the example above, the following instructions have nothing to do with the first two, the code could be rearranged so that those instructions are executed in between the two dependent instructions and the pipeline could flow efficiently. The task of code reordering is generally left to the compiler, which recognizes data dependencies and attempts to minimize performance stalls.

Branch instructions

- Branch instructions are those that tell the processor to make a decision about what the next instruction to be executed should be based on the results of another instruction. Branch instructions can be troublesome in a pipeline if a branch is conditional on the results of an instruction which has not yet finished its path through the pipeline.

```
Loop: add $r3, $r2, $r1
      sub $r6, $r5, $r4
      beq $r3, $r6, loop
```

The example above instructs the processor to add r1 and r2 and put the result in r3, then subtract r4 from r5, storing the difference in r6. In the third instruction, beq stands for branch if equal. If the contents of r3 and r6 are equal, the processor should execute the instruction labeled "Loop." Otherwise, it should continue to the next instruction. In this example, the processor cannot make a decision about which branch to take because neither the value of r3 or r6 have been written into the registers yet.

- The processor could stall, but a more sophisticated method of dealing with branch instructions is branch prediction. The processor makes a guess about which path to take - if the guess is wrong, anything written into the registers must be cleared, and the pipeline must be started again with the correct instruction. Some methods of branch prediction depend on stereotypical behavior. Branches pointing backward are taken about 90% of the time since backward-pointing branches are often found at the bottom of loops. On the other hand, branches pointing forward, are only taken approximately 50% of the time. Thus, it would be logical for processors to always follow the branch when it points backward, but not when it points forward. Other methods of branch prediction are less static: processors that use dynamic prediction keep a history for each branch and uses it to predict future branches. These processors are correct in their predictions 90% of the time.
- Still other processors forgo the entire branch prediction ordeal. The RISC System/6000 fetches and starts decoding instructions from both sides of the branch. When it determines which branch should be followed, it then sends the correct instructions down the pipeline to be executed.

CISC	**RISC**
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

The Performance Equation

- The following equation is commonly used for expressing a computer's performance ability:

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \text{ x } \frac{\text{cycles}}{\text{instruction}} \text{ x } \frac{\text{instructions}}{\text{program}}$$

- The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.

The Overall RISC Advantage

- Today, the Intel x86 is arguable the only chip which retains CISC architecture. This is primarily due to advancements in other areas of computer technology. The price of RAM has decreased dramatically. In 1977, 1MB of DRAM cost about $5,000. By 1994, the same amount of memory cost only $6 (when adjusted for inflation). Compiler technology has also become more sophisticated, so that the RISC use of RAM and emphasis on software has become ideal.

RISC vs CISC

- The formula for processor performance:

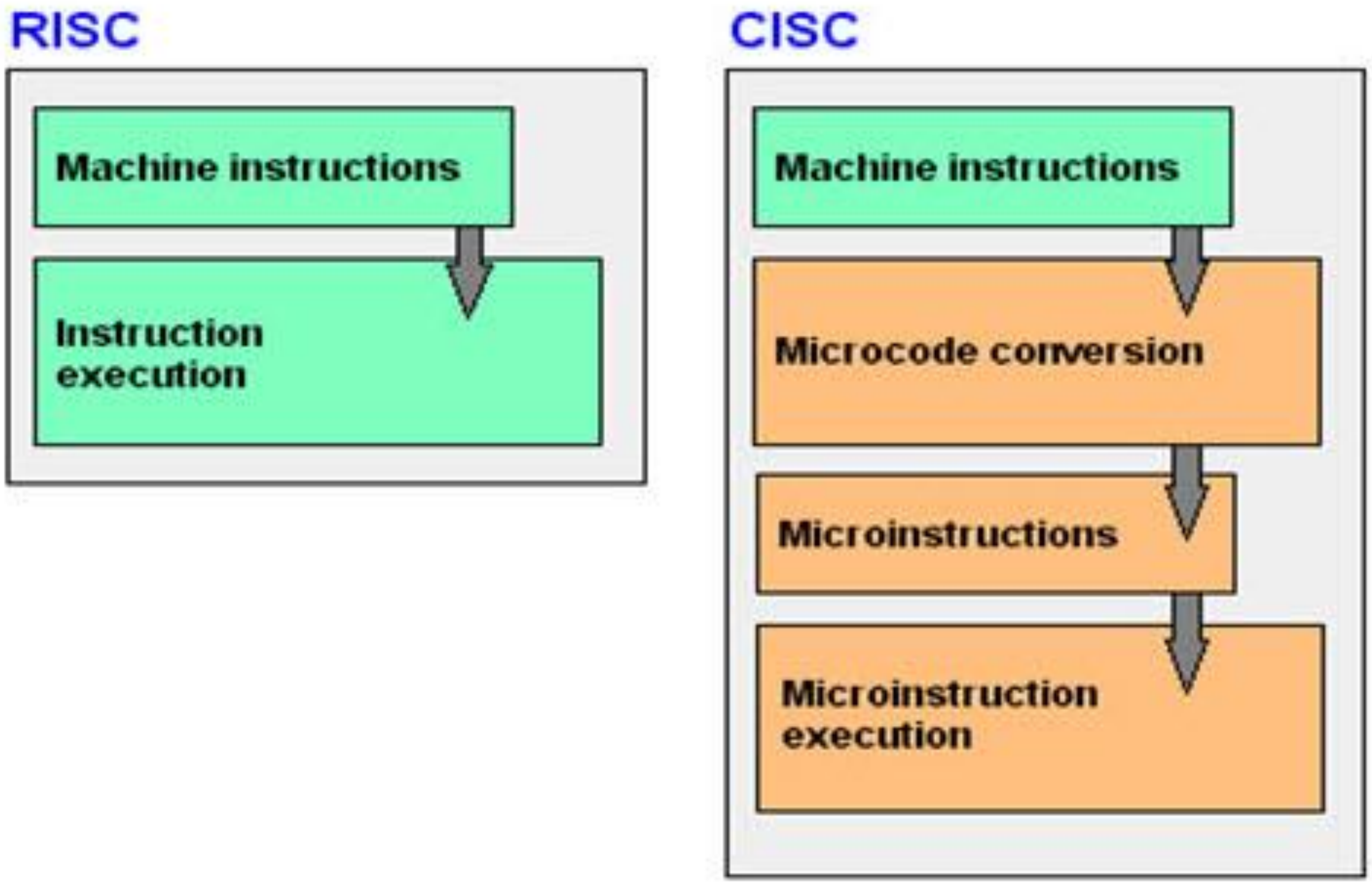
Time/Program = Instructions /Program × (Clock cycles) /Instruction × Time/ (Clock cycle)

- DEC engineers later showed that the more complicated CISC ISA executed about 75% of the number instructions per program as RISC (the first term), but in a similar technology CISC executed about five to six more clock cycles per instruction (the second term), making RISC microprocessors approximately 4× faster.

Instruction Set Architecture

- RISC (Reduced Instruction Set Computer) Architectures
  - Memory accesses are restricted to load and store instruction, and data manipulation instructions are register to register.
  - Addressing modes are limited in number.
  - Instruction formats are all of the same length.
  - Instructions perform elementary operations

- CISC (Complex Instruction Set Computer) Architectures
  - Memory access is directly available to most types of instruction.
  - Addressing mode are substantial in number.
  - Instruction formats are of different lengths.
  - Instructions perform both elementary and complex operations.



RISC

Machine instructions

Instruction execution

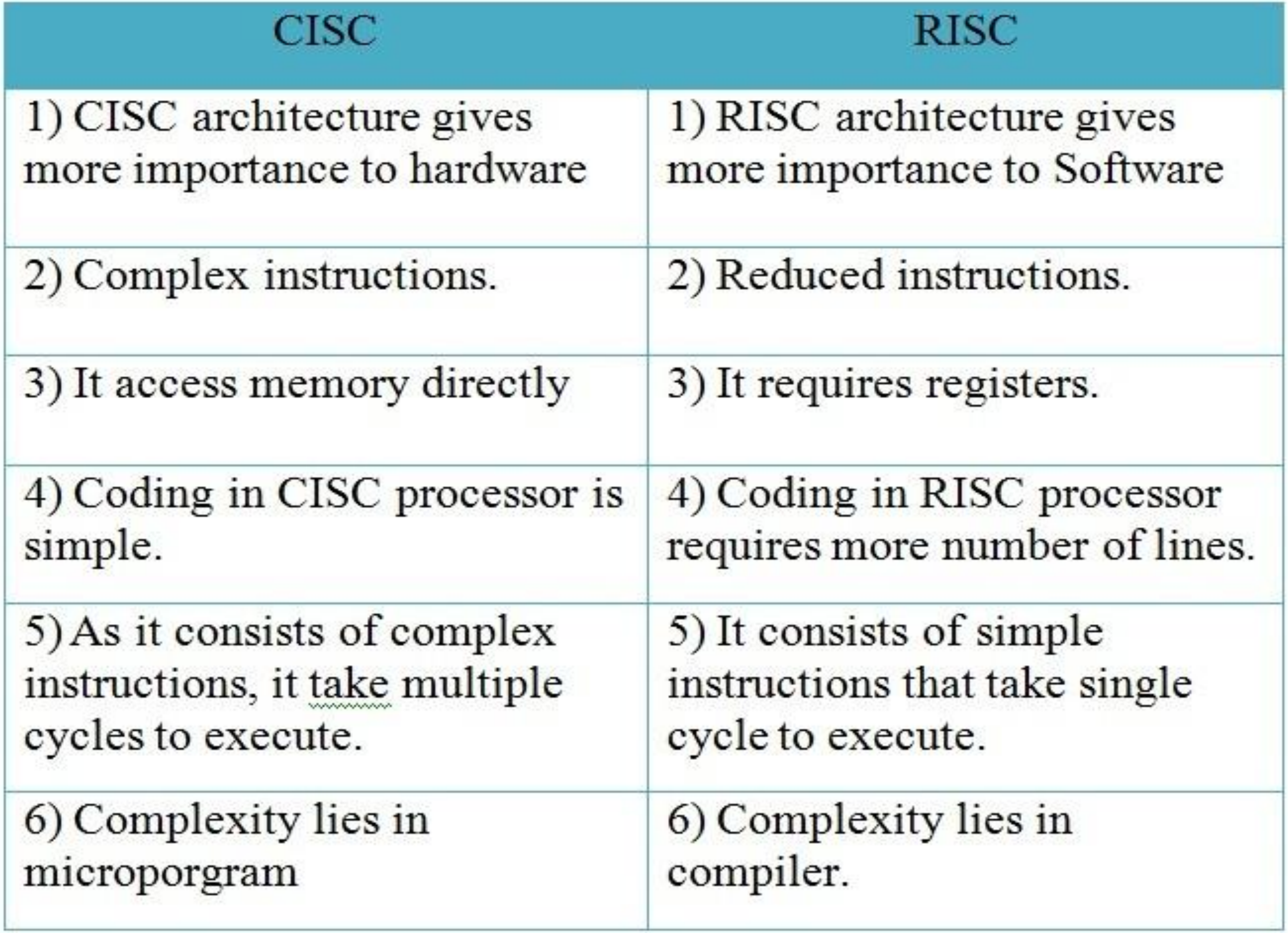
CISC

Machine instructions

Microcode conversion

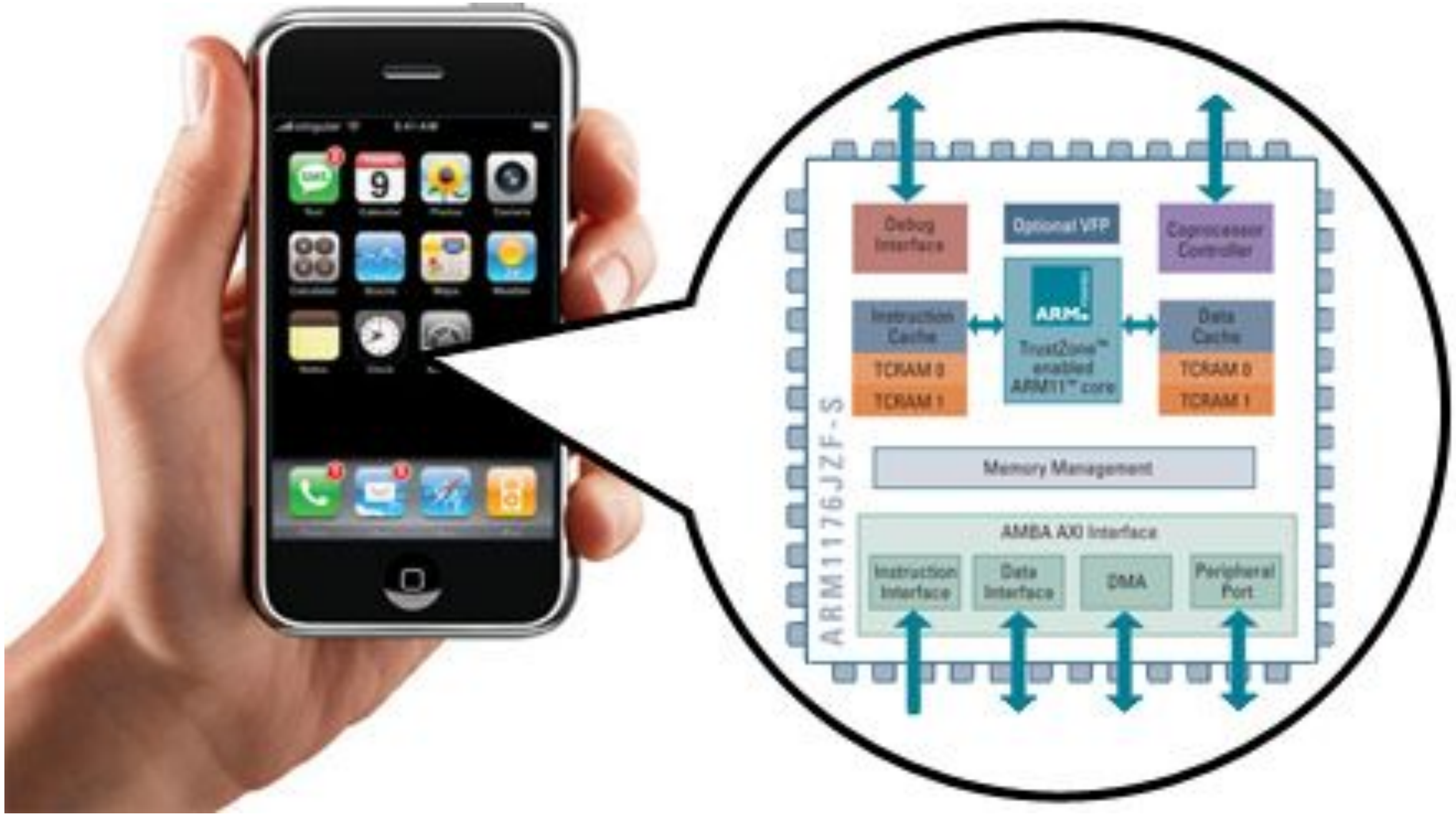
Microinstructions

Microinstruction execution



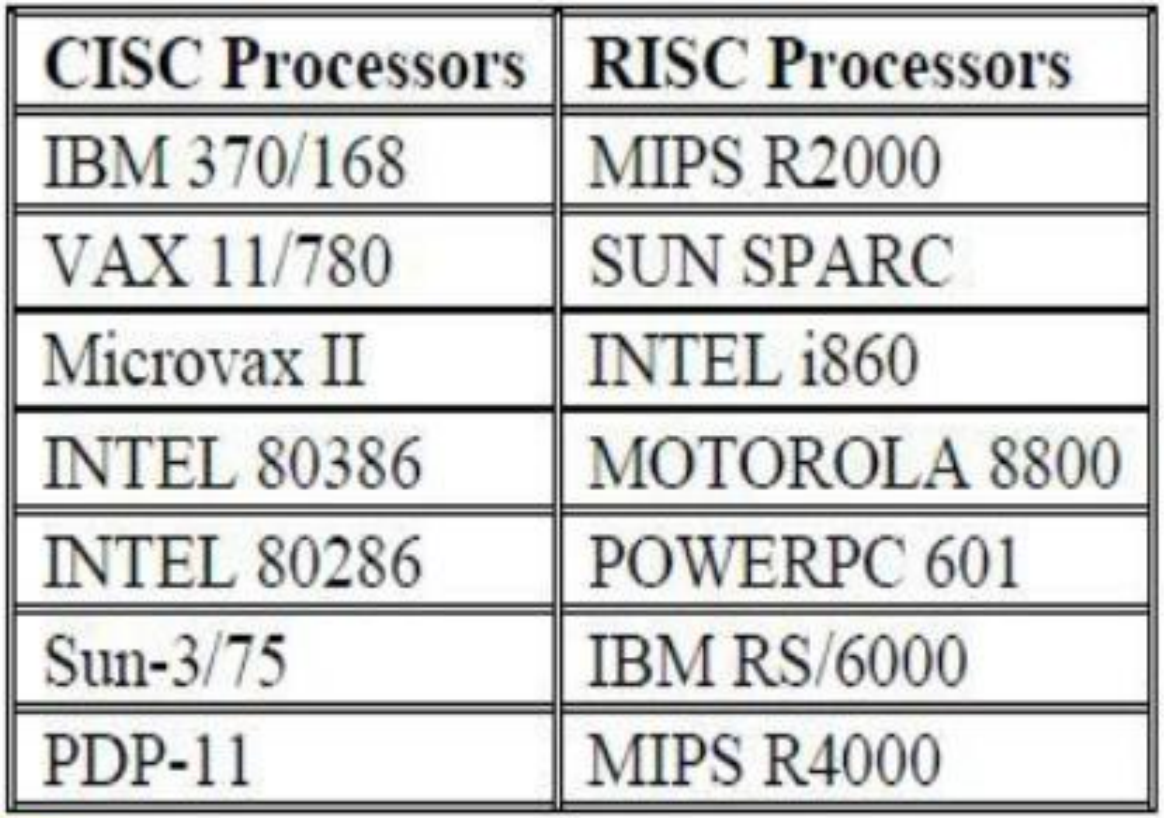
CISC	RISC
1) CISC architecture gives more importance to hardware	1) RISC architecture gives more importance to Software
2) Complex instructions.	2) Reduced instructions.
3) It access memory directly	3) It requires registers.
4) Coding in CISC processor is simple.	4) Coding in RISC processor requires more number of lines.
5) As it consists of complex instructions, it take multiple cycles to execute.	5) It consists of simple instructions that take single cycle to execute.
6) Complexity lies in microporgram	6) Complexity lies in compiler.

Advanced RISC Machine (ARM) instruction set inside the iPhone



You will how to design and program a related RISC computer: MIPS

Examples of CISC and RISC Processors



CISC Processors	RISC Processors
IBM 370/168	MIPS R2000
VAX 11/780	SUN SPARC
Microvax II	INTEL i860
INTEL 80386	MOTOROLA 8800
INTEL 80286	POWERPC 601
Sun-3/75	IBM RS/6000
PDP-11	MIPS R4000

The CISC Approach

The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations.

For this particular task, a CISC processor would come prepared with a specific instruction (we'll call it "MULT"). When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with one instruction:

MULT M1, M2

MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. It closely resembles a command in a higher level language. For instance, if we let "a" represent the value of memory M1 and "b" represent the value of memory M2, then this command is identical to the C statement “[M1] = [M1] x [M2]"

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

The RISC Approach

RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands: "LOAD," which moves data from the memory bank to a register, "PROD," which finds the product of two operands located within the registers, and "STORE," which moves data from a register to the memory banks. In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

```
LOAD A, M1

LOAD B, M2

PROD A, B

STORE M1, A
```

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

Background of RISC

- John Cocke and his colleagues developed simpler ISAs and compilers for minicomputers. As an experiment, they retargeted their research compilers to use only the simple register-register operations and load-store data transfers of the IBM 360 ISA, avoiding the more complicated instructions. They found that programs ran up to three times faster using the simple subset.

- Emer and Clark found 20% of the VAX instructions needed 60% of the microcode and represented only 0.2% of the execution time.

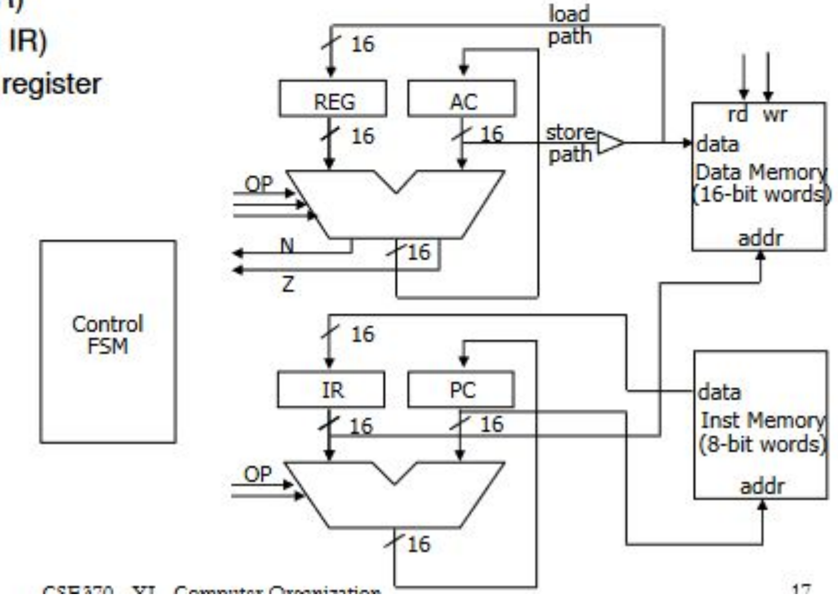
RISC

- First, the RISC instructions were simplified so there was no need for a microcoded interpreter. The RISC instructions were typically as simple as microinstructions and could be executed directly by the hardware.
- Second, the fast memory, formerly used for the microcode interpreter of a CISC ISA, was repurposed to be a cache of RISC instructions. (A cache is a small, fast memory that buffers recently executed instructions, as such instructions are likely to be reused soon.)
- Third, register allocators based on Gregory Chaitin’s graph-coloring scheme made it much easier for compilers to efficiently use registers, which benefited these register-register ISAs

- *Time/Program = Instructions /Program* × (*Clock cycles*) */Instruction* × *Time /* (*Clock cycle*)
- DEC engineers later showed2 that the more complicated CISC ISA executed about 75% of the number instructions per program as RISC (the first term), but in a similar technology CISC executed about five to six more clock cycles per instruction (the second term), making RISC microprocessors approximately 4× faster.

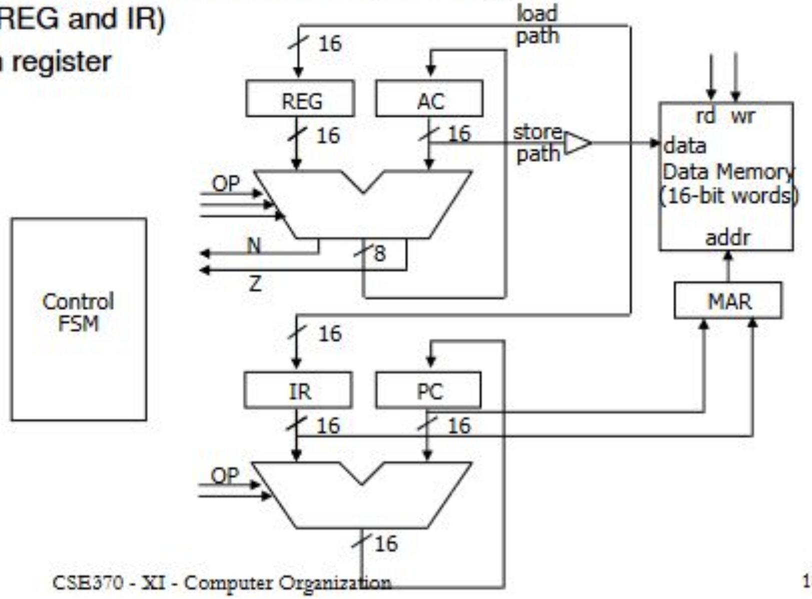
Block diagram of processor (Harvard)

- Register transfer view of Harvard architecture
  - which register outputs are connected to which register inputs
  - arrows represent data-flow, other are control signals from control FSM
  - two MARs (PC and IR)
  - two MBRs (REG and IR)
  - load control for each register



Block diagram of processor (Princeton)

- Register transfer view of Princeton architecture
  - which register outputs are connected to which register inputs
  - arrows represent data-flow, other are control signals from control FSM
  - MAR may be a simple multiplexer rather than separate register
  - MBR is split in two (REG and IR)
  - load control for each register


load path
16
REG
AC
16
16
store path
rd wr
data
Data Memory (16-bit words)
addr
MAR
OP
N
Z
8
Control FSM
16
IR
PC
16
16
OP
16