Recap

CPU Chips and Buses

CPU pins

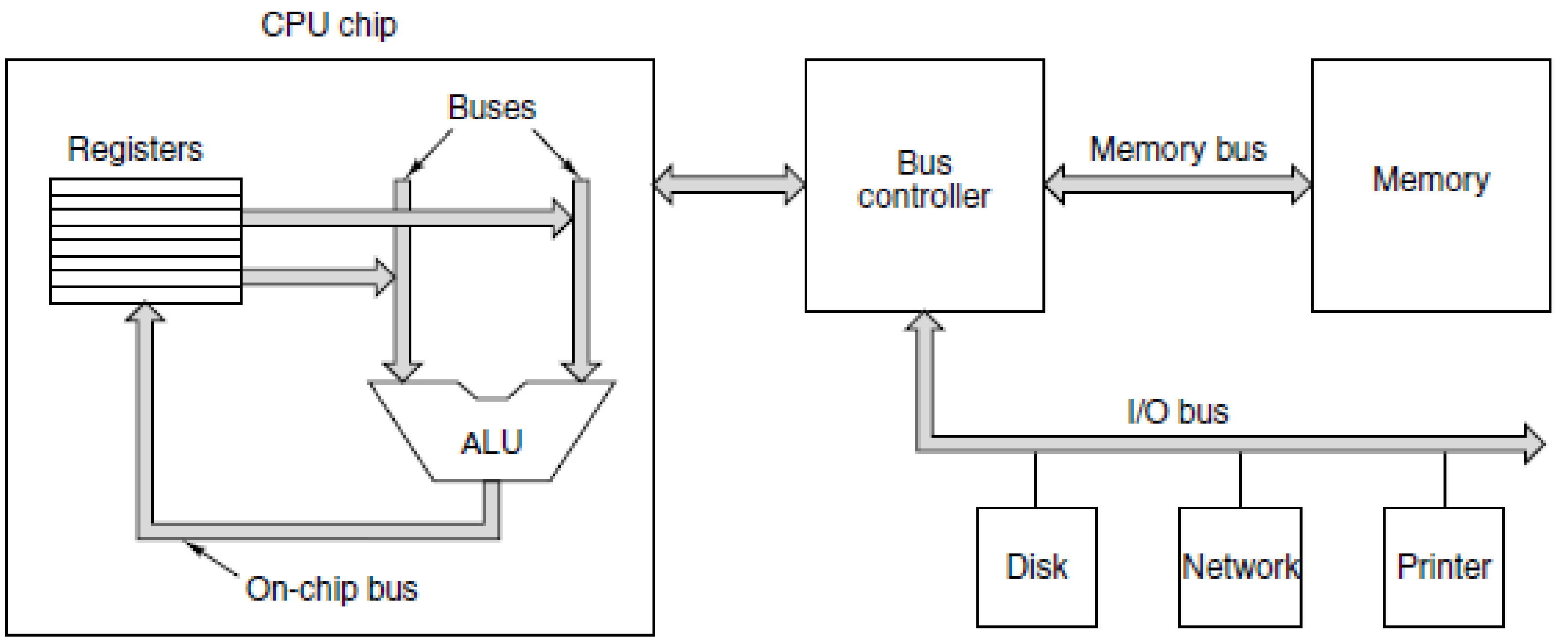
The pins on a CPU chip

1) Power (usually +1.2 to +1.5 volts)
2) Ground
3) Clock signal (a square wave at some well-defined frequency)
4) Address
5) Data
6) Control
    - A. Bus control
    - B. Interrupts
    - C. Bus arbitration
    - D. Coprocessor signaling
    - E. Status
    - F. Miscellaneous

Recap

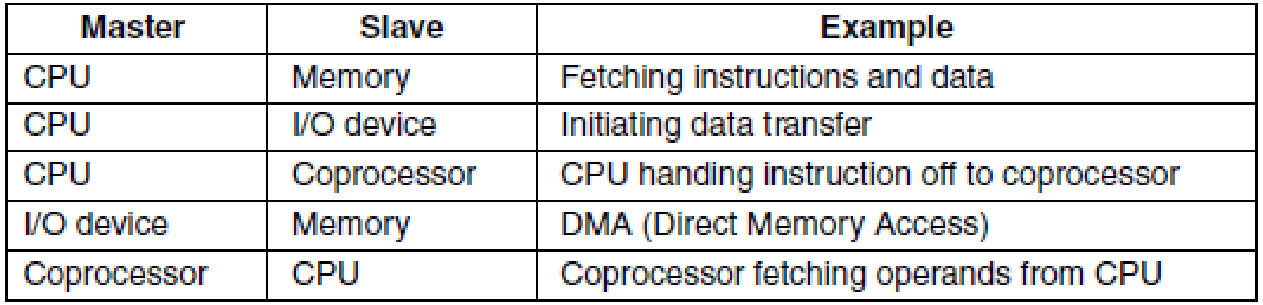
Internal to CPU

External to CPU



A computer system with multiple buses.

Recap



Master	Slave	Example
CPU	Memory	Fetching instructions and data
CPU	I/O device	Initiating data transfer
CPU	Coprocessor	CPU handing instruction off to coprocessor
I/O device	Memory	DMA (Direct Memory Access)
Coprocessor	CPU	Coprocessor fetching operands from CPU

Examples of bus masters and slaves.

Types:

Synchronous, has a master clock.
Asynchronous bus, does not have a master clock.

Recap

Bus Arbitration

Bus arbitration decides who goes next

1)Centralized (one-level and two-levels)
A bus arbiter is needed
2) Decentralized
No bus arbiter is needed

Examples:
PCI, PCI-X, PCI-Express, USB, and SCSI

Processors
Primary Memory
Secondary Memory
Input/Output

Processors

Central Processing Unit (**CPU**) is made of many distinct parts:

- **Local, high speed, fast memory or Registers**

  e.g.

  Program Counter (**PC**) points to the next instruction to be fetched for execution

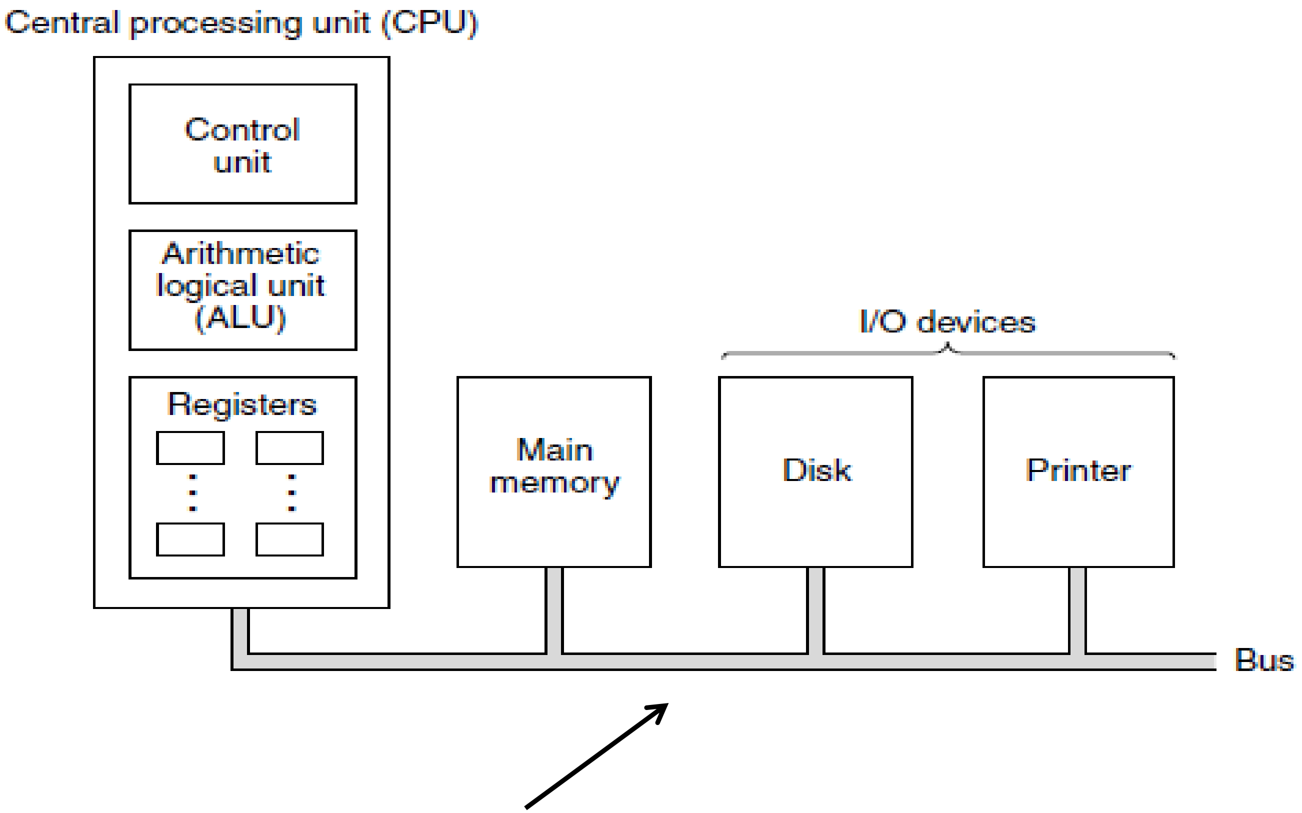
  Instruction Register (**IR**), holds the instruction currently being executed

- **Arithmetic Logic Unit (ALU)**

  Can perform addition, subtraction, logical operations(**NOT, OR, AND**), increment, and so on

- **Control unit**

Processors



External bus

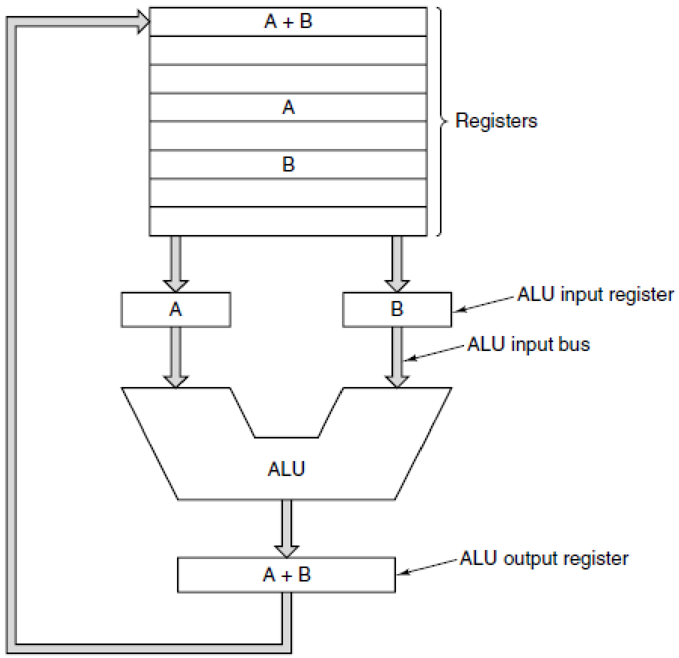
Processors

CPU Organization

- Type of instructions:

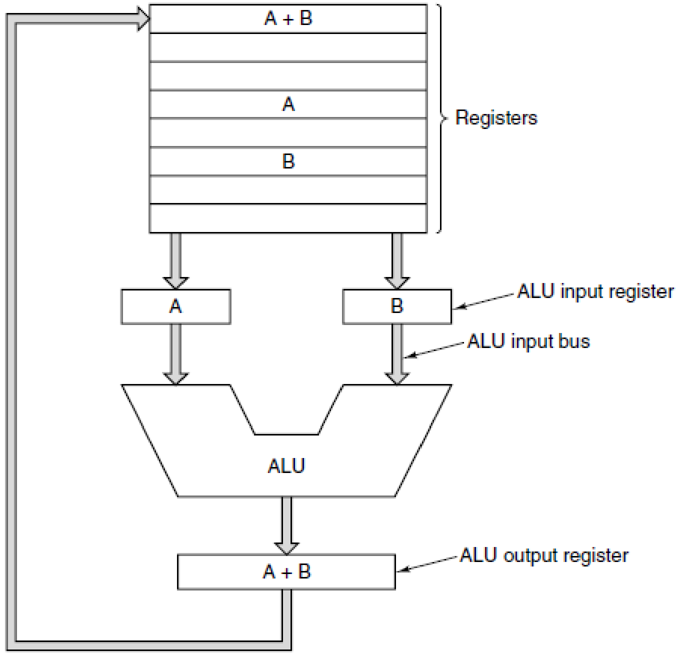
  - **Register-Memory**
    Words being fetched from memory into registers (they can be used as ALU inputs in subsequent instructions)

  - **Register-Register**
    two operands are fetched from the registers. brings them to the ALU input registers, performs some operation on them and stores the result back in one of the registers.



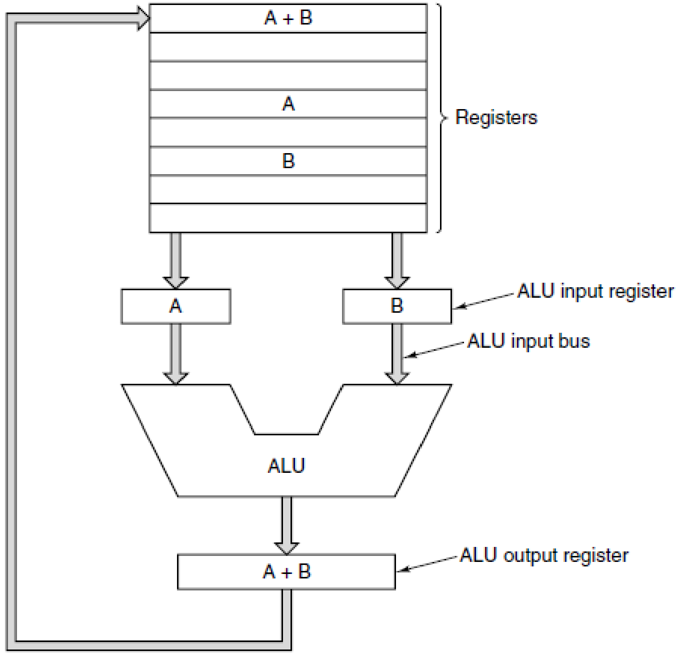
Data Path Cycle

- The process of running two operands through the ALU and storing results
- It defines what the machine can do
- The faster the data path cycles is, the faster the computer runs



Example:

Consider the operation of a machine with the data path of the following figure. Suppose that loading the ALU input registers takes 4 nsec, running the ALU takes 12 nsec, and storing the result back in the register scratchpad takes 4 nsec. What is the maximum number of cycle Per Second this machine is capable of in the absence of pipelining?



Example:

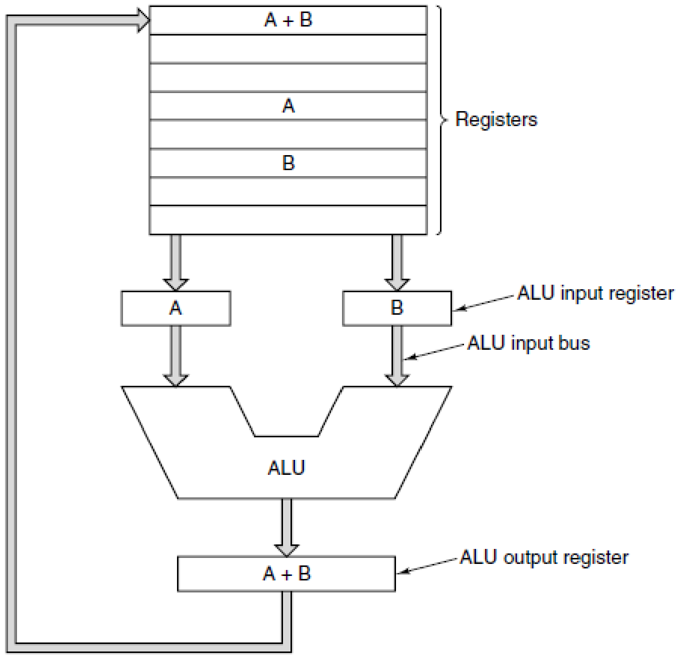
Consider the operation of a machine with the data path of the following figure. Suppose that loading the ALU input registers takes 4 nsec, running the ALU takes 12 nsec, and storing the result back in the register scratchpad takes 4 nsec. What is the maximum number of cycle Per Second this machine is capable of in the absence of pipelining?

Total time for one cycle T = 4 + 12 + 4 = 20 nsec

Number of cycles per second f = 1/T

$f = 1/20\text{x}10^{-9}$ (cycle/sec)

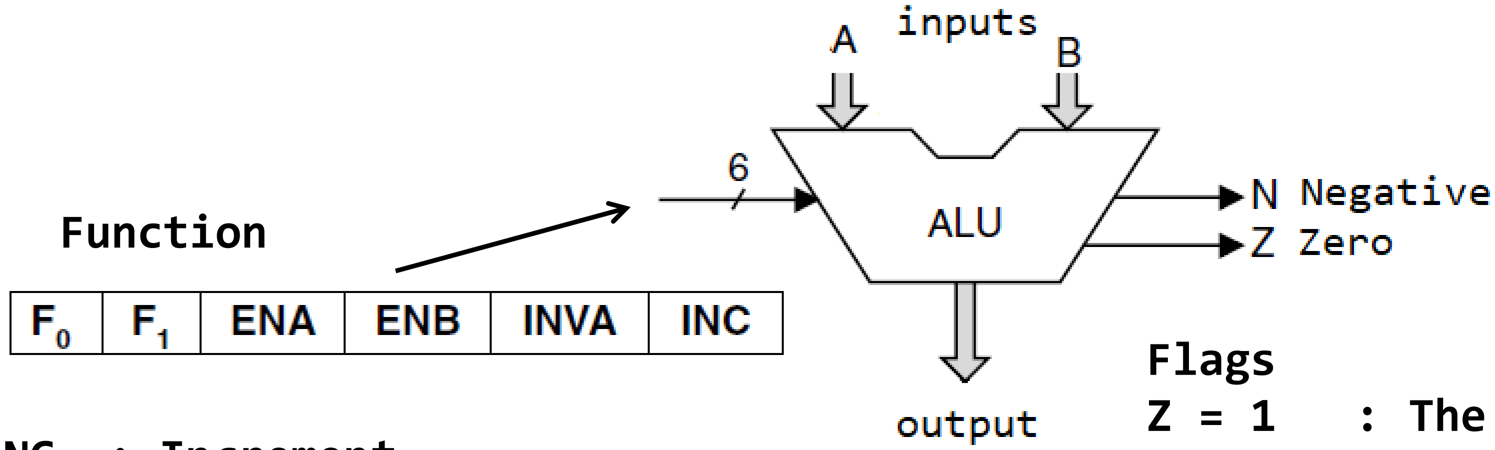
$f = 0.5 \text{ x } 10^{8}$ (cycle/sec) = 50 MHz



Processors

CPU Organization

Function



F_0	F_1	ENA	ENB	INVA	INC

Flags

Z = 1 : The output is zero

N = 1 : The output is negative

INC : Increment

F : Function

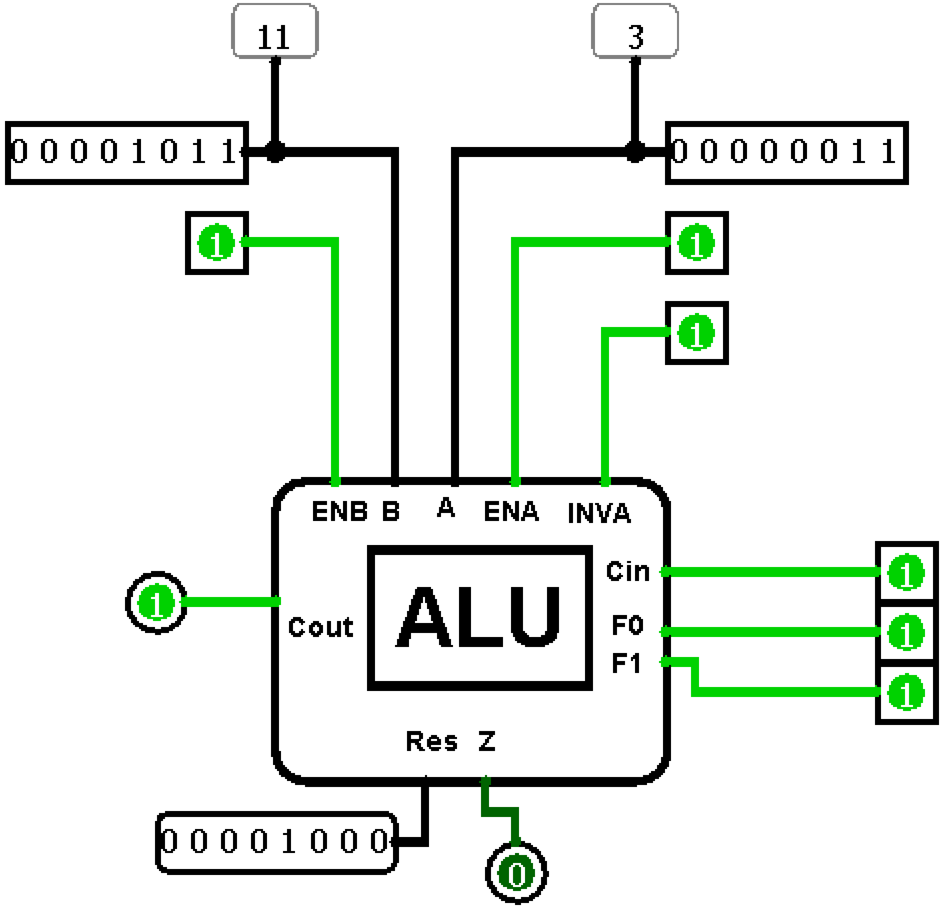
ENA : Enable A

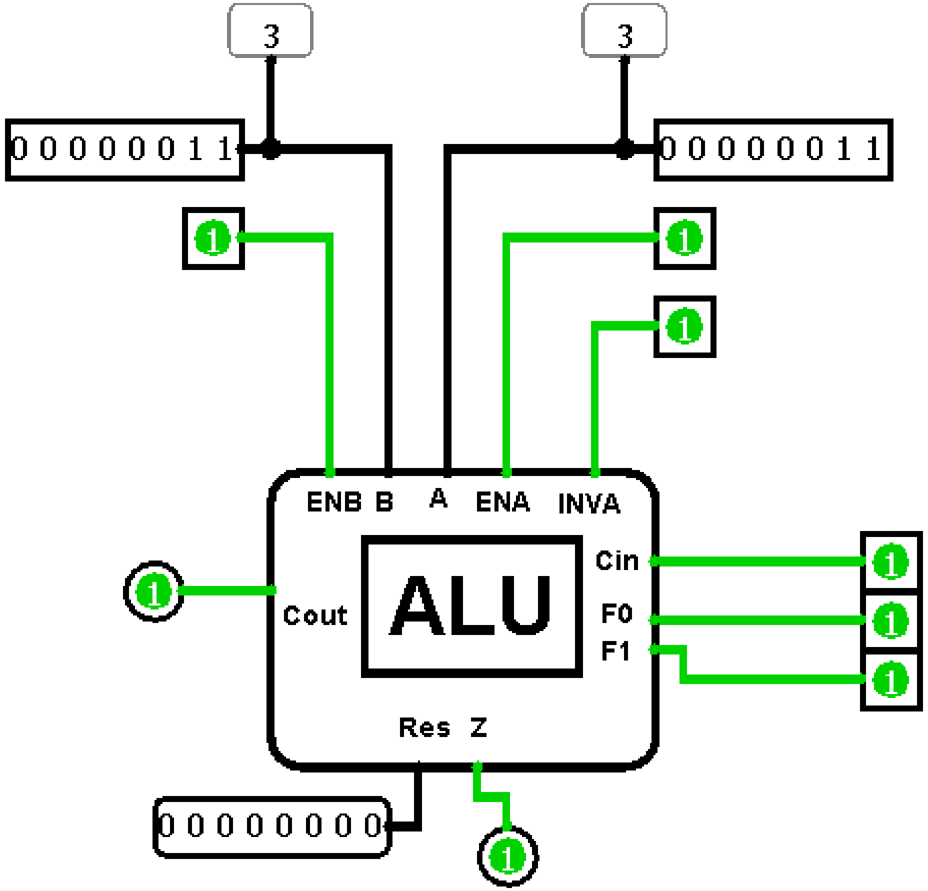
ENB : Enable B

INVA : Invert A

Processors

CPU Organization



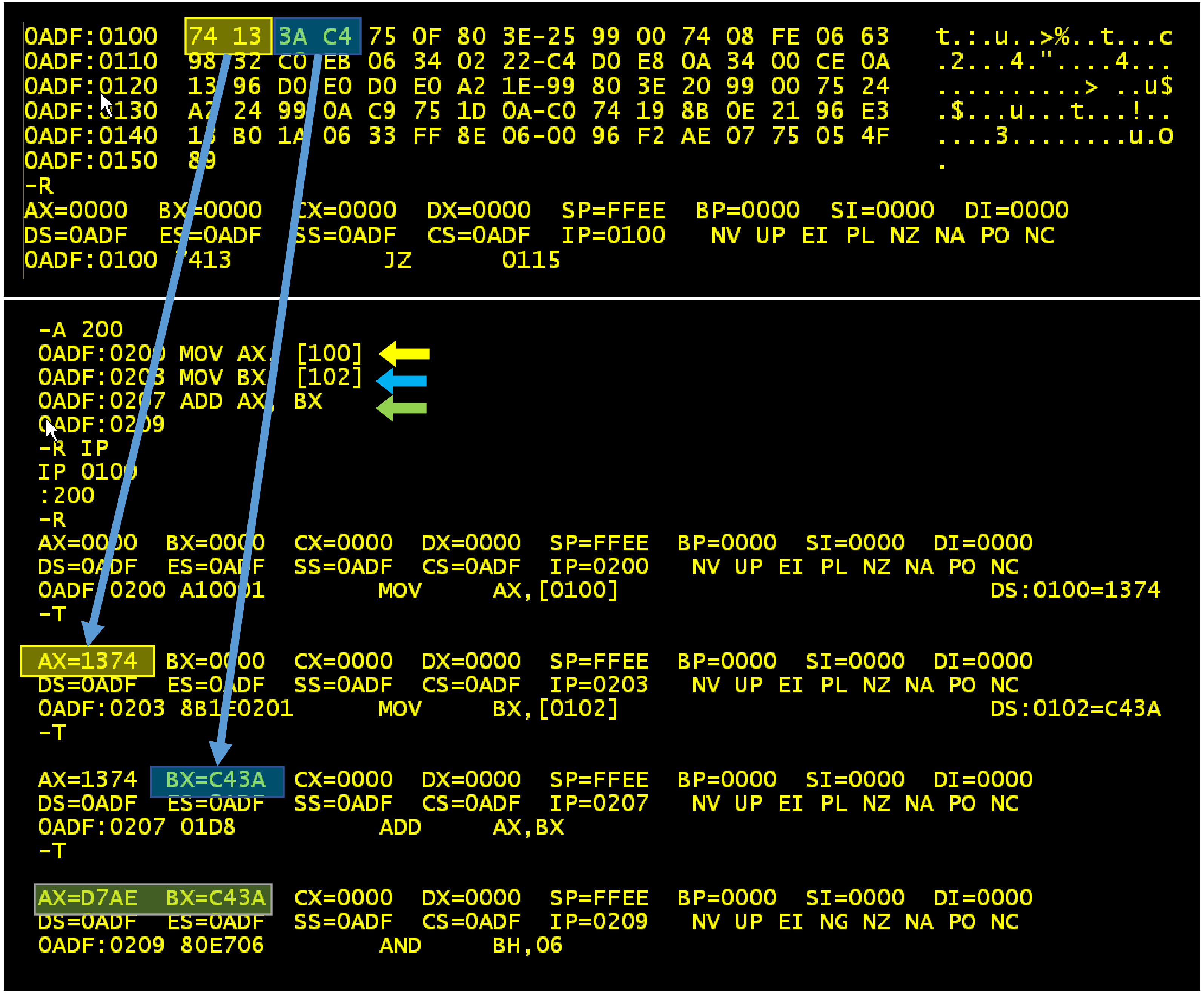


Register-Memory

Memory

Registers

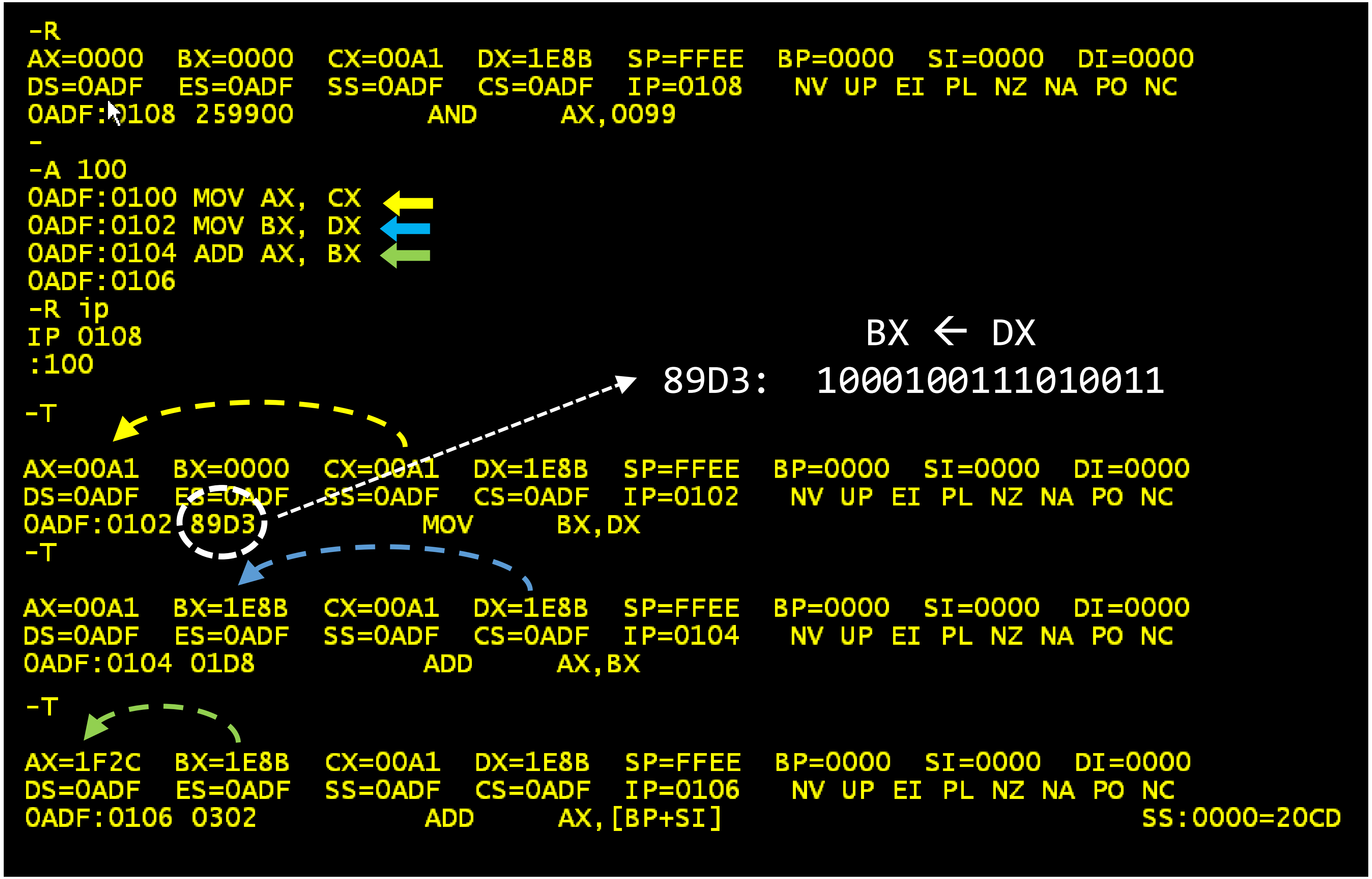
Assembly codes



```
0ADF:0100  74 13 3A C4 75 0F 80 3E-25 99 00 74 08 FE 06 63   t.:.u..>%..t...c
0ADF:0110  98 32 C0 EB 06 34 02 22-C4 D0 E8 0A 34 00 CE 0A   .2...4."....4...
0ADF:0120  13 96 D0 E0 D0 E0 A2 1E-99 80 3E 20 99 00 75 24   ..........> ..u$
0ADF:0130  A2 24 99 0A C9 75 1D 0A-C0 74 19 8B 0E 21 96 E3   .$...u...t...!..
0ADF:0140  13 B0 1A 06 33 FF 8E 06-00 96 F2 AE 07 75 05 4F   ....3........u.O
0ADF:0150  89                                                .
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0100   NV UP EI PL NZ NA PO NC
0ADF:0100 7413          JZ      0115
```

```
 -A 200
 0ADF:0200 MOV AX, [100]
 0ADF:0203 MOV BX, [102]
 0ADF:0207 ADD AX, BX
 0ADF:0209
 -R IP
 IP 0100
 :200
 -R
 AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
 DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0200   NV UP EI PL NZ NA PO NC
 0ADF:0200 A10001        MOV     AX,[0100]                       DS:0100=1374
 -T

 AX=1374  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
 DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0203   NV UP EI PL NZ NA PO NC
 0ADF:0203 8B1E0201      MOV     BX,[0102]                       DS:0102=C43A
 -T

 AX=1374  BX=C43A  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
 DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0207   NV UP EI PL NZ NA PO NC
 0ADF:0207 01D8          ADD     AX,BX
 -T

 AX=D7AE  BX=C43A  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
 DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0209   NV UP EI NG NZ NA PO NC
 0ADF:0209 80E706        AND     BH,06
```

Register-Register

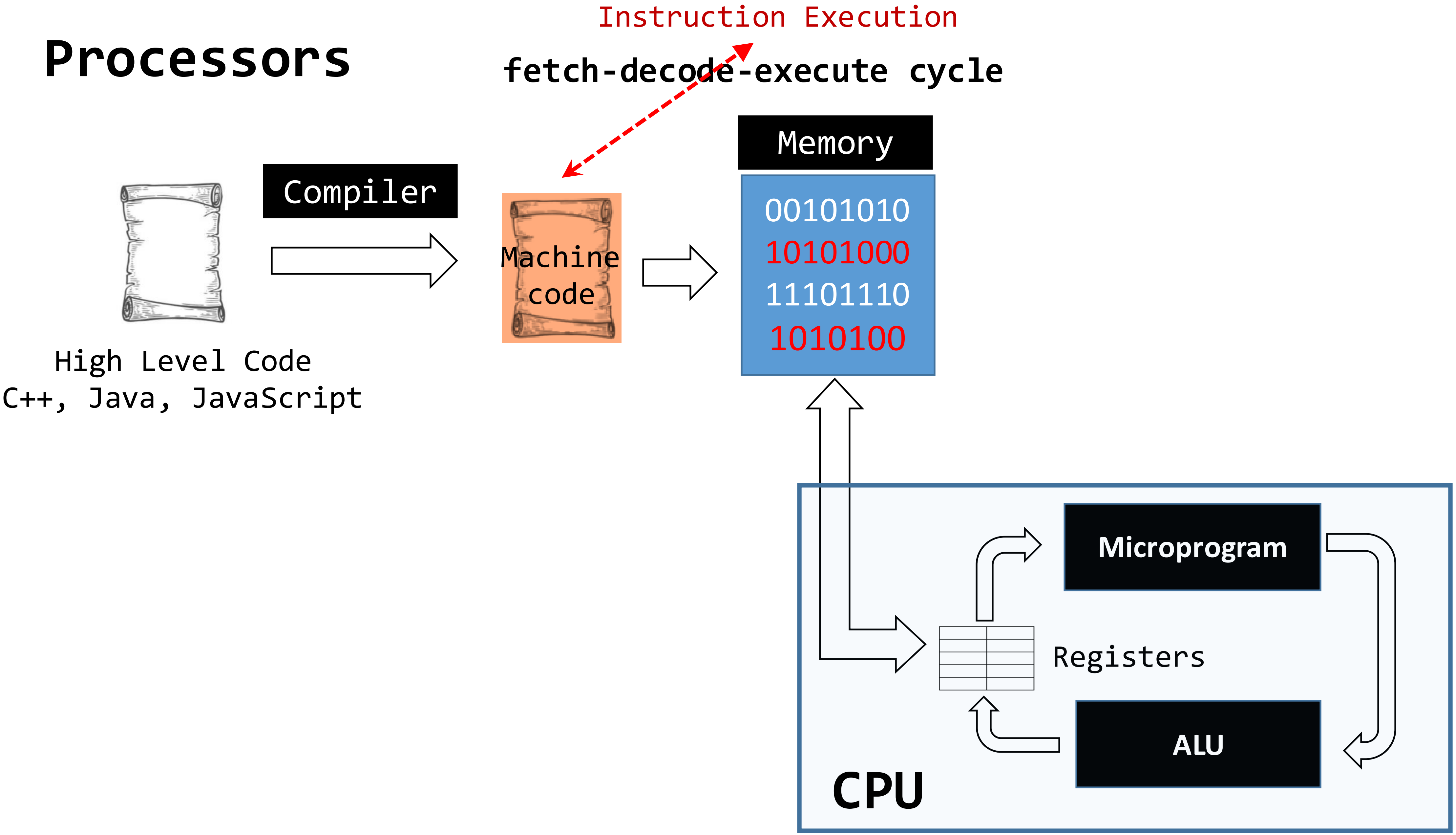


```
-R
AX=0000  BX=0000  CX=00A1  DX=1E8B  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0108   NV UP EI PL NZ NA PO NC
0ADF:0108 259900        AND     AX,0099
-
-A 100
0ADF:0100 MOV AX, CX
0ADF:0102 MOV BX, DX
0ADF:0104 ADD AX, BX
0ADF:0106
-R ip
IP 0108
:100

-T

AX=00A1  BX=0000  CX=00A1  DX=1E8B  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0102   NV UP EI PL NZ NA PO NC
0ADF:0102 89D3          MOV     BX,DX
-T

AX=00A1  BX=1E8B  CX=00A1  DX=1E8B  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0104   NV UP EI PL NZ NA PO NC
0ADF:0104 01D8          ADD     AX,BX

-T

AX=1F2C  BX=1E8B  CX=00A1  DX=1E8B  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0106   NV UP EI PL NZ NA PO NC
0ADF:0106 0302          ADD     AX,[BP+SI]                         SS:0000=20CD
```

BX ← DX

89D3: 1000100111010011

Processors



Instruction Execution

fetch-decode-execute cycle

Compiler

Machine code

Memory

00101010
10101000
11101110
1010100

High Level Code
C++, Java, JavaScript

Microprogram

Registers

ALU

CPU

Processors

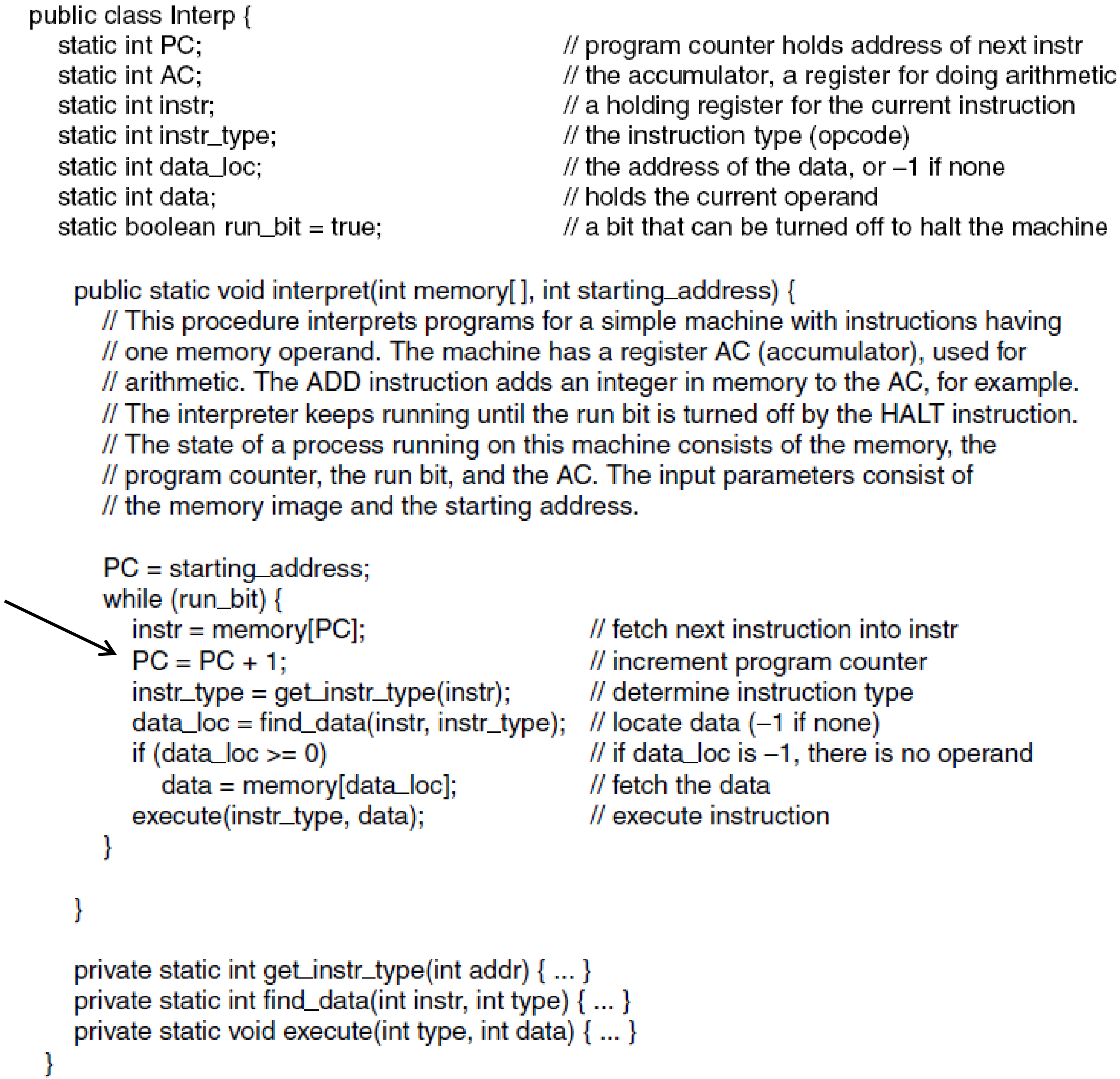
Instruction Execution

fetch-decode-execute cycle

Fetch-decode-execute cycle

1. Fetch the next instruction from memory into the instruction register.
2. Change the program counter to point to the following instruction.
3. Determine the type of instruction just fetched. E.g. ADD, SUB, XOR, NOT, and so on
4. If the instruction uses a word in memory, determine where it is.
5. Fetch the word, if needed, into a CPU register.
6. Execute the instruction.
7. Go to step 1 to begin executing the following instruction.

Used for next cycle



```
public class Interp {
   static int PC;                               // program counter holds address of next instr
   static int AC;                               // the accumulator, a register for doing arithmetic
   static int instr;                            // a holding register for the current instruction
   static int instr_type;                       // the instruction type (opcode)
   static int data_loc;                         // the address of the data, or –1 if none
   static int data;                             // holds the current operand
   static boolean run_bit = true;               // a bit that can be turned off to halt the machine

    public static void interpret(int memory[ ], int starting_address) {
       // This procedure interprets programs for a simple machine with instructions having
       // one memory operand. The machine has a register AC (accumulator), used for
       // arithmetic. The ADD instruction adds an integer in memory to the AC, for example.
       // The interpreter keeps running until the run bit is turned off by the HALT instruction.
       // The state of a process running on this machine consists of the memory, the
       // program counter, the run bit, and the AC. The input parameters consist of
       // the memory image and the starting address.

       PC = starting_address;
       while (run_bit) {
          instr = memory[PC];                       // fetch next instruction into instr
          PC = PC + 1;                              // increment program counter
          instr_type = get_instr_type(instr);       // determine instruction type
          data_loc = find_data(instr, instr_type);  // locate data (–1 if none)
          if (data_loc >= 0)                        // if data_loc is –1, there is no operand
             data = memory[data_loc];               // fetch the data
          execute(instr_type, data);                // execute instruction
       }

    }

    private static int get_instr_type(int addr) { ... }
    private static int find_data(int instr, int type) { ... }
    private static void execute(int type, int data) { ... }
 }
```

IP address vs Machine code

url

IP	Web address
72.30.35.9	yahoo.com
216.58.217.46	google.com
142.232.230.10	bcit.ca

Which one is easier to remember?
IP or web address

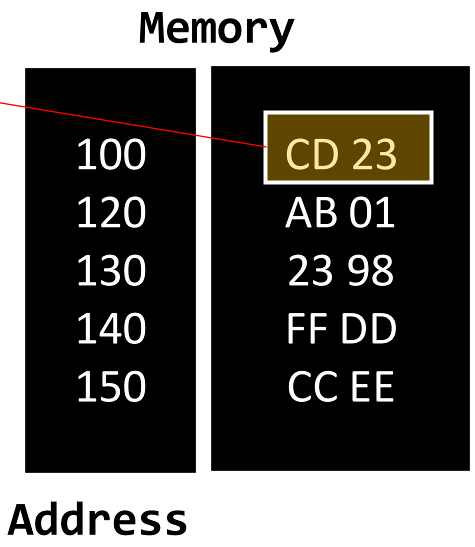
Which one is easier to remember?
machine code or operation name

Machine code	Operation name	Hig Level Language
01D8	ADD AX, BX	a+b
29C8	SUB AX, CX	a-b
31D8	XOR	a^b

Assembly Language (Human readable language)

Operation code (opcode) Destination Operand, Source Operand

```
MOV Destination, Source
MOV AX, [100] ⇒ AX = The value at memory location of 100
MOV AX, DX    ⇒ AX = DX

ADD Destination, Source
ADD AX, BX    ⇒ AX = AX + BX

SUB Destination, Source
SUB AX, BX    ⇒ AX = AX - BX

OR Destination, Source
OR AX, BX    ⇒ AX = AX OR BX

AND Destination, Source
AND AX, BX    ⇒ AX = AX AND BX

XOR Destination, Source
XOR AX, BX    ⇒ AX = AX XOR BX
```



Memory

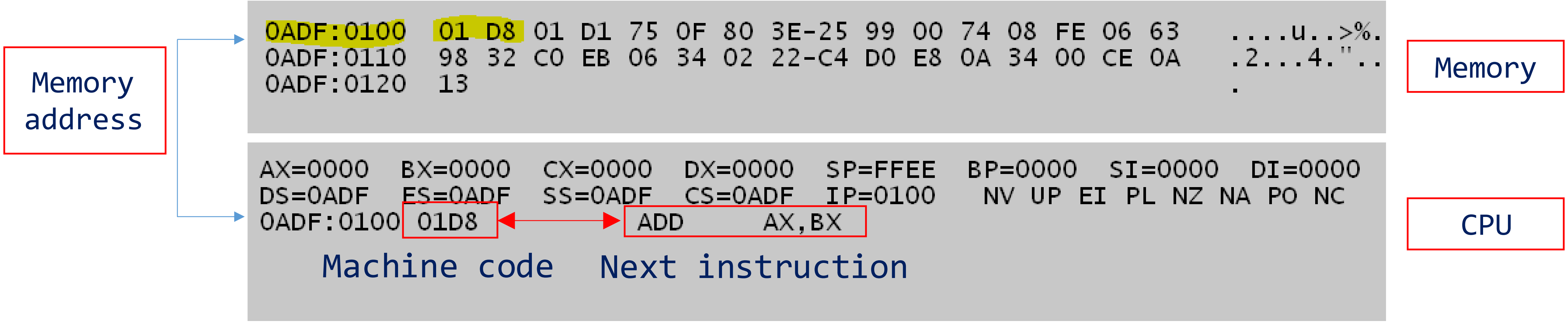
Address	Memory
100	CD 23
120	AB 01
130	23 98
140	FF DD
150	CC EE

Processors

Instruction Execution

fetch-decode-execute cycle

Example: 8088 Microprocessor (CPU)



```
0ADF:0100  01 D8 01 D1 75 0F 80 3E-25 99 00 74 08 FE 06 63   ....u..>%.
0ADF:0110  98 32 C0 EB 06 34 02 22-C4 D0 E8 0A 34 00 CE 0A   .2...4."..
0ADF:0120  13                                                .
```

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0100   NV UP EI PL NZ NA PO NC
0ADF:0100 01D8          ADD     AX,BX
```

AX, BX, CX, DX general purpose registers

IP (Instruction Pointer) the address of next instruction in the memory

CS (Code Segment)

SP (Stack Pointer)

BP (Base Pointer)

SI (Source Index)

DI (Destination Index)

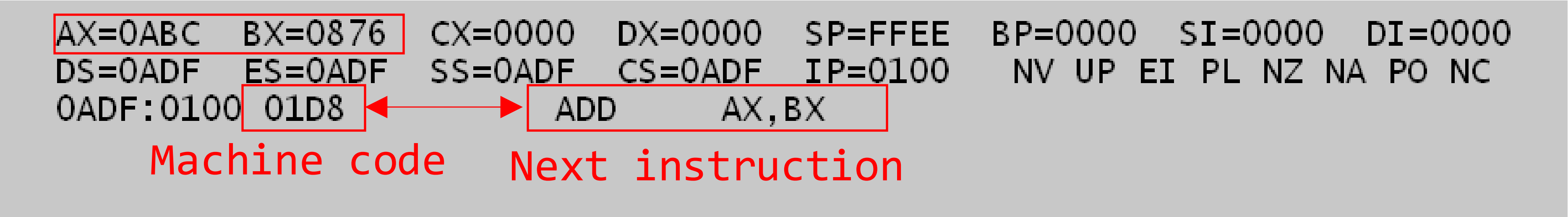
DS (Data Segment)

ES (Extra Segment)

SS (Stack Segment)

CS (Code Segment)

Example 1: Obtain next instruction and machine code ?
Stored values in AX, and BX before and after execution?



```
AX=0ABC  BX=0876  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0100   NV UP EI PL NZ NA PO NC
0ADF:0100 01D8          ADD     AX,BX
```

ADD Destination, Source

ADD AX, BX ⇒ AX = AX + BX

Before

AX = 0x0ABC

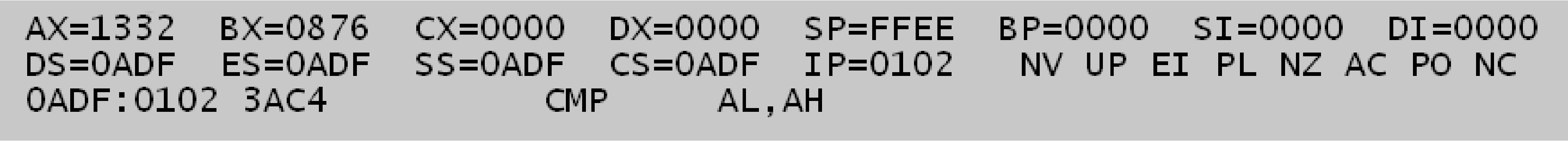
BX = 0x0876

After

BX = 0x0876

AX = 0x0ABC + 0x0876 = 0x1332

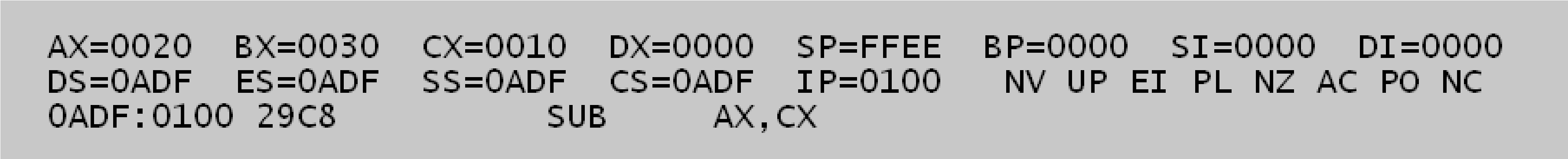
Execution



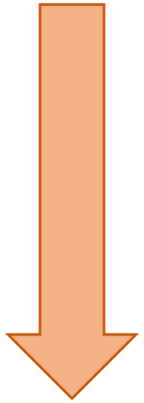
```
AX=1332  BX=0876  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0102   NV UP EI PL NZ AC PO NC
0ADF:0102 3AC4          CMP     AL,AH
```

Example 2: Obtain next instruction and machine code ?

Stored values in AX, BX, and CX before and after execution?



```
AX=0020  BX=0030  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0100   NV UP EI PL NZ AC PO NC
0ADF:0100 29C8          SUB     AX,CX
```

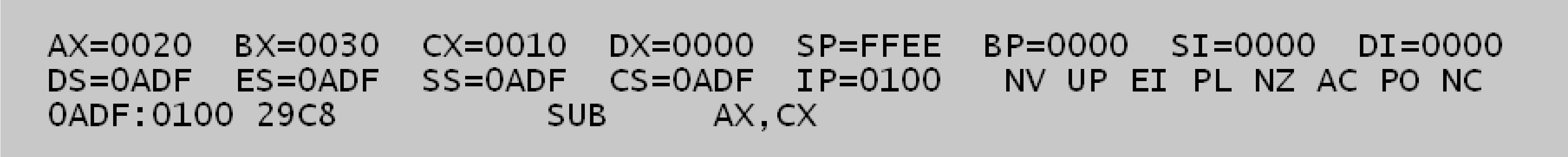


Execution

Before
AX = ?
BX = ?
CX = ?

After
AX = ?
BX = ?
CX = ?

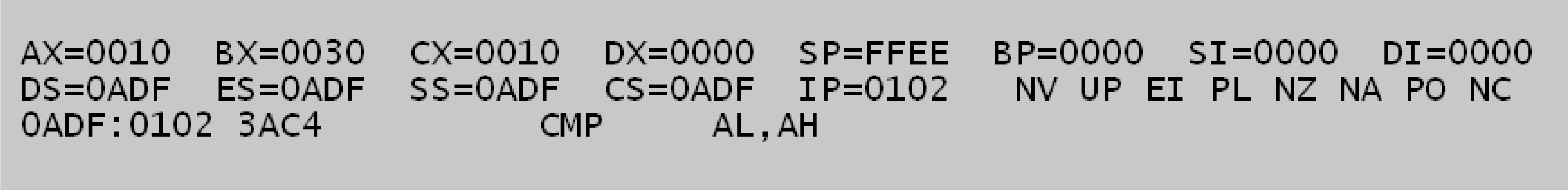
Example 2: Obtain next instruction and machine code ?
Stored values in AX, BX, and CX before and after execution?



```
AX=0020  BX=0030  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0100   NV UP EI PL NZ AC PO NC
0ADF:0100 29C8          SUB     AX,CX
```

Before
AX = 0x0020
BX = 0x0030
CX = 0x0010
Next execution = SUB AX, CX
Machine code   = 0x29C8

Execution

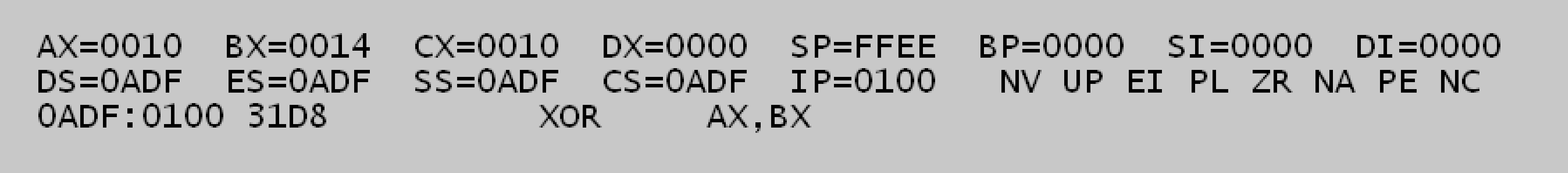


```
AX=0010  BX=0030  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0102   NV UP EI PL NZ NA PO NC
0ADF:0102 3AC4          CMP     AL,AH
```

After
AX = AX – CX = 0x0020 – 0x0010 = 0x0010
BX = 0x0030
CX = 0x0010

Example 3: Obtain next instruction and machine code ?

Stored values in AX, BX, and CX before and after execution?



```
AX=0010  BX=0014  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0100   NV UP EI PL ZR NA PE NC
0ADF:0100 31D8          XOR     AX,BX
```

Before

AX =

BX =

CX =

Next execution =

Machine code   =

Memory address =

Execution

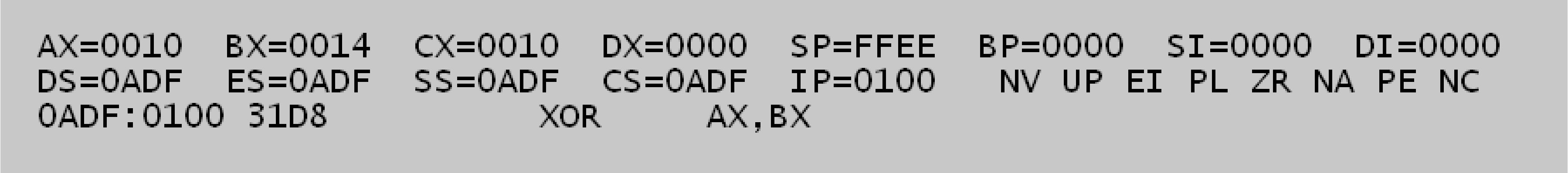
After

AX = ?

BX = ?

CX = ?

Example 3: Obtain next instruction and machine code ? Stored values in AX, BX, and CX before and after execution?



```
AX=0010  BX=0014  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0100   NV UP EI PL ZR NA PE NC
0ADF:0100 31D8          XOR     AX,BX
```

Before

AX = 0x0010

BX = 0x0014

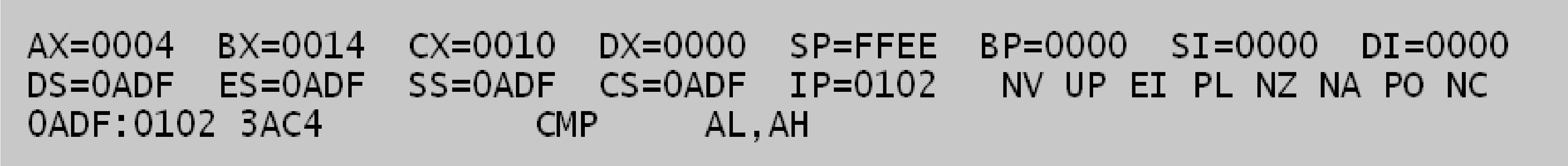
CX = 0x0010

Next execution = XOR AX, BX

Machine code   = 0x31D8

Memory address = 0ADF:100

Execution



```
AX=0004  BX=0014  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0102   NV UP EI PL NZ NA PO NC
0ADF:0102 3AC4          CMP     AL,AH
```

After

BX = 0x0014

CX = 0x0010

AX = AX XOR BX = 0000 0000 0001 0000 ⊕ 0000 0000 0001 0100

AX = 0000 0000 0000 0100 = 0x0004

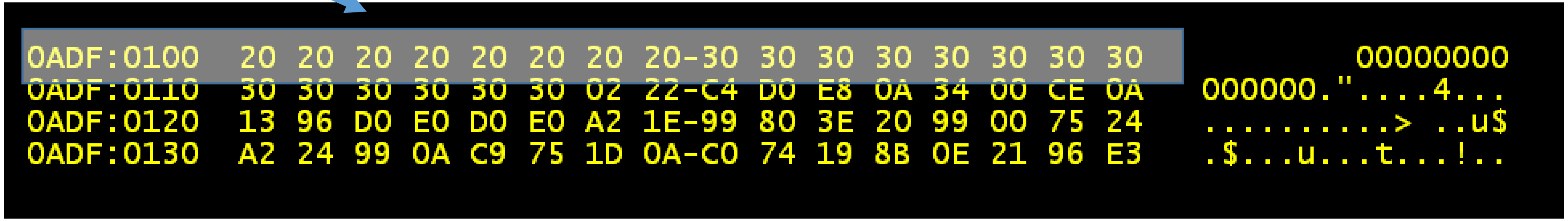
Processors

Instruction Execution

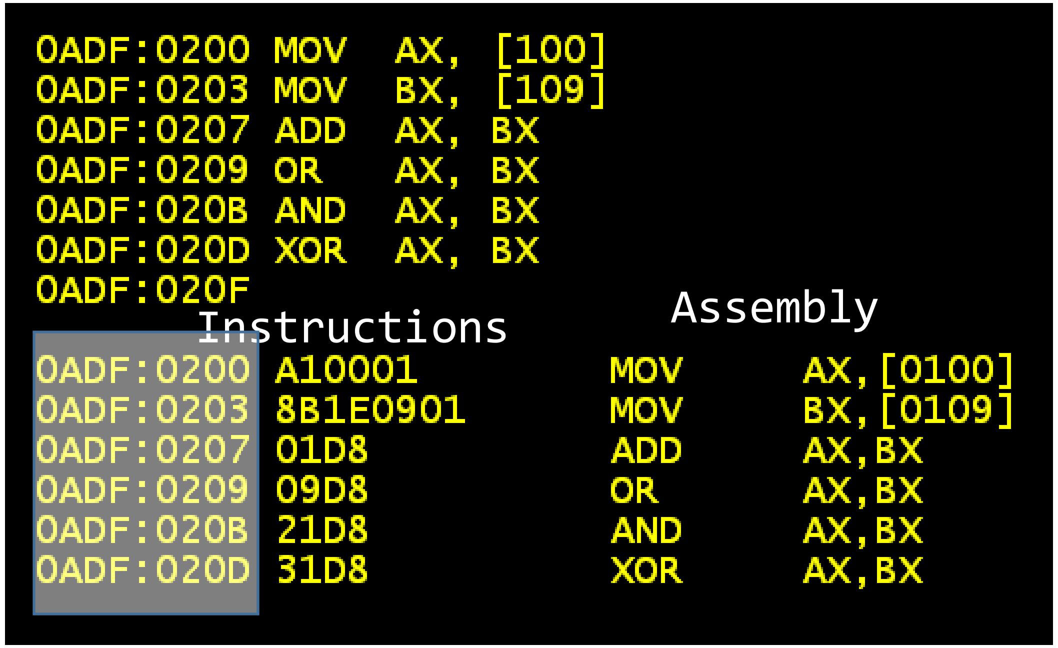
fetch-decode-execute cycle

The locations of data

Memory



```
0ADF:0100  20 20 20 20 20 20 20 20-30 30 30 30 30 30 30 30          00000000
0ADF:0110  30 30 30 30 30 30 02 22-C4 D0 E8 0A 34 00 CE 0A   000000."....4...
0ADF:0120  13 96 D0 E0 D0 E0 A2 1E-99 80 3E 20 99 00 75 24   ..........> ..u$
0ADF:0130  A2 24 99 0A C9 75 1D 0A-C0 74 19 8B 0E 21 96 E3   .$...u...t...!..
```



```
0ADF:0200 MOV   AX, [100]
0ADF:0203 MOV   BX, [109]
0ADF:0207 ADD   AX, BX
0ADF:0209 OR    AX, BX
0ADF:020B AND   AX, BX
0ADF:020D XOR   AX, BX
0ADF:020F
```

Instructions

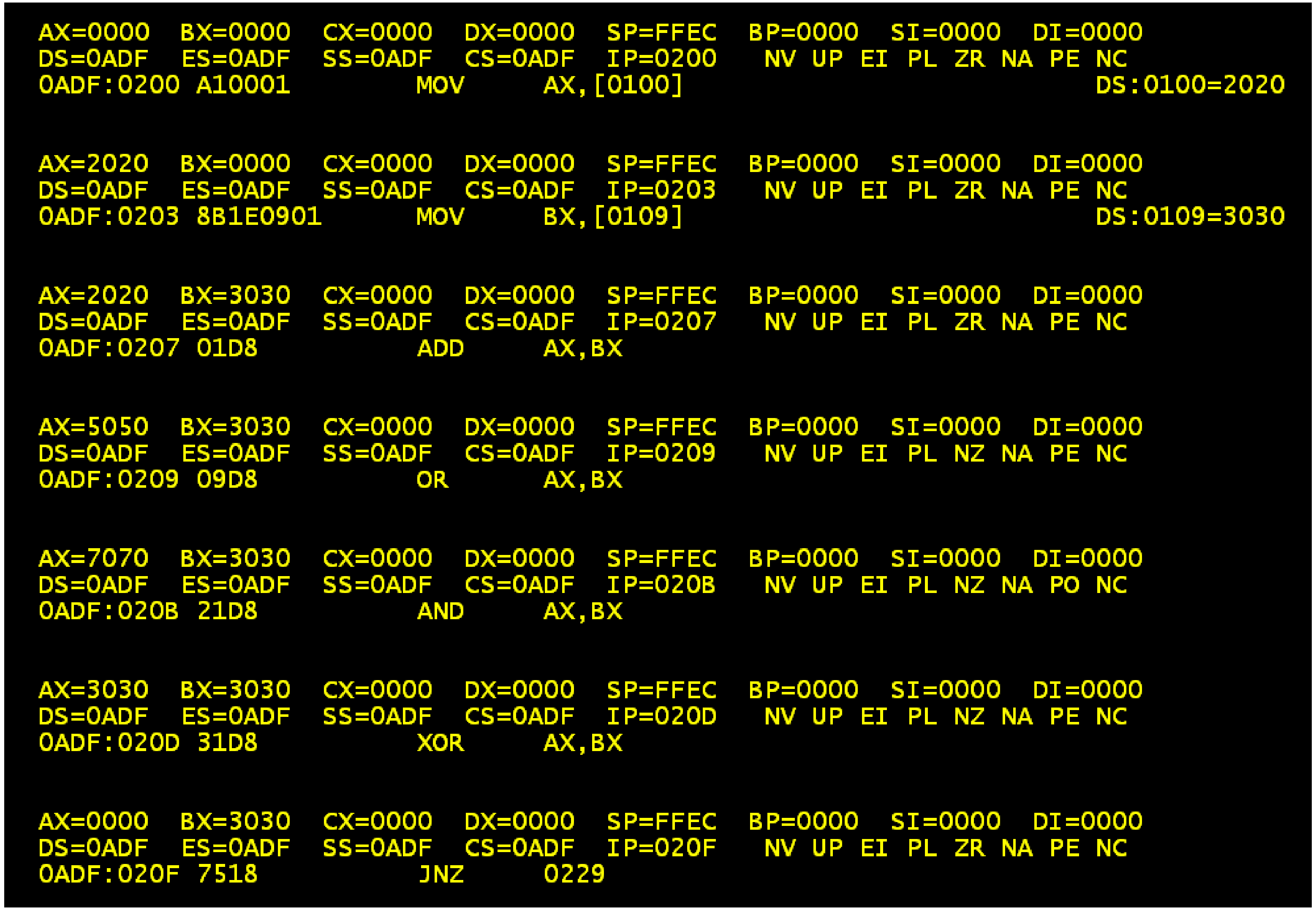
Assembly

```
0ADF:0200 A10001          MOV       AX,[0100]
0ADF:0203 8B1E0901        MOV       BX,[0109]
0ADF:0207 01D8            ADD       AX,BX
0ADF:0209 09D8            OR        AX,BX
0ADF:020B 21D8            AND       AX,BX
0ADF:020D 31D8            XOR       AX,BX
```

The locations of instructions

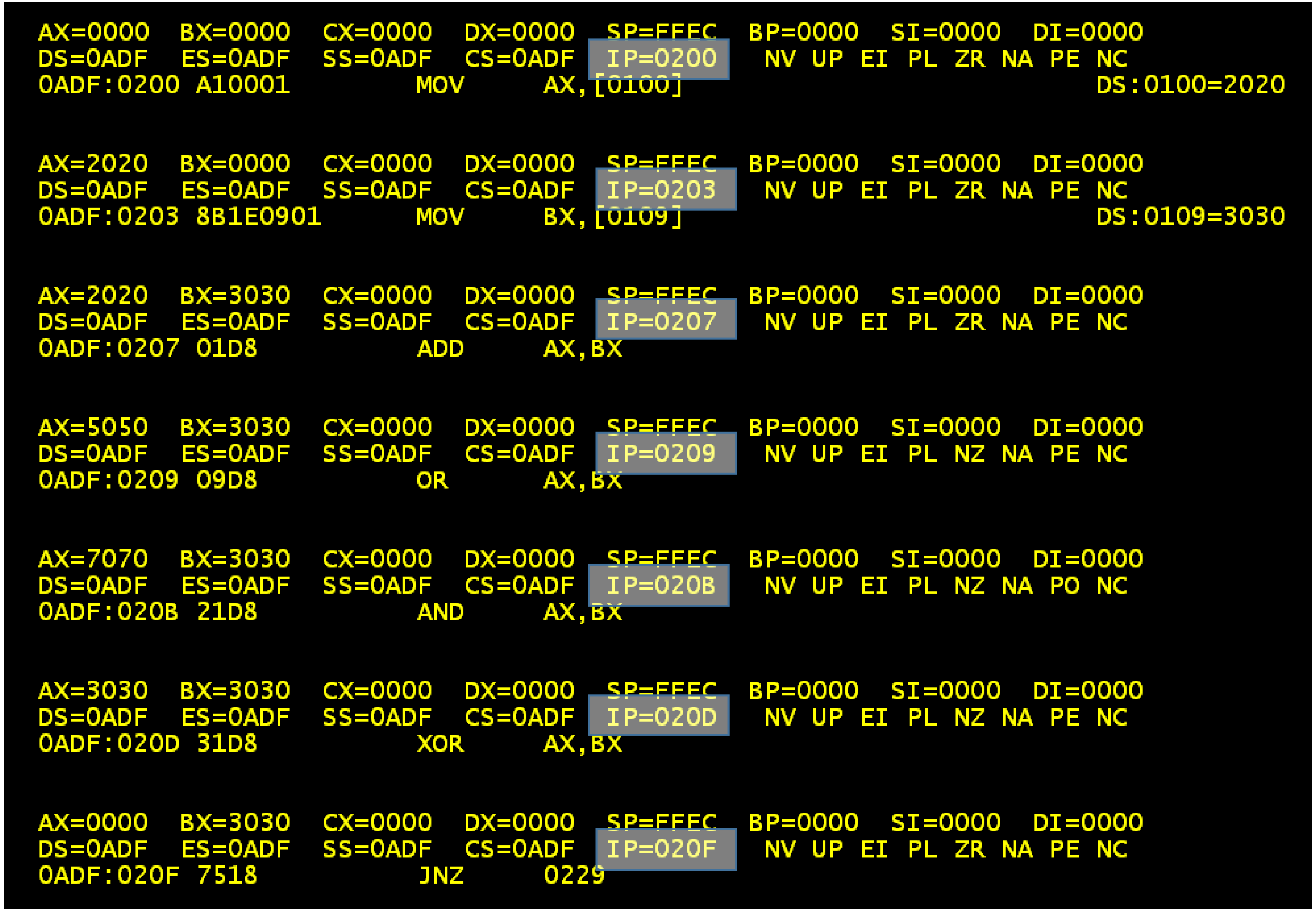
Instructions	After decodeing
A10001	AX = The value at memory location 0100
8B1E0901	BX = The value at memory location 0109
01D8	AX = AX + BX
09D8	AX OR BX
31D8	AX AND BX
31D8	AX⊕BX

After Execution



```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0200   NV UP EI PL ZR NA PE NC
0ADF:0200 A10001        MOV     AX,[0100]                      DS:0100=2020

AX=2020  BX=0000  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0203   NV UP EI PL ZR NA PE NC
0ADF:0203 8B1E0901      MOV     BX,[0109]                      DS:0109=3030

AX=2020  BX=3030  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0207   NV UP EI PL ZR NA PE NC
0ADF:0207 01D8          ADD     AX,BX

AX=5050  BX=3030  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0209   NV UP EI PL NZ NA PE NC
0ADF:0209 09D8          OR      AX,BX

AX=7070  BX=3030  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=020B   NV UP EI PL NZ NA PO NC
0ADF:020B 21D8          AND     AX,BX

AX=3030  BX=3030  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=020D   NV UP EI PL NZ NA PE NC
0ADF:020D 31D8          XOR     AX,BX

AX=0000  BX=3030  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=020F   NV UP EI PL ZR NA PE NC
0ADF:020F 7518          JNZ     0229
```

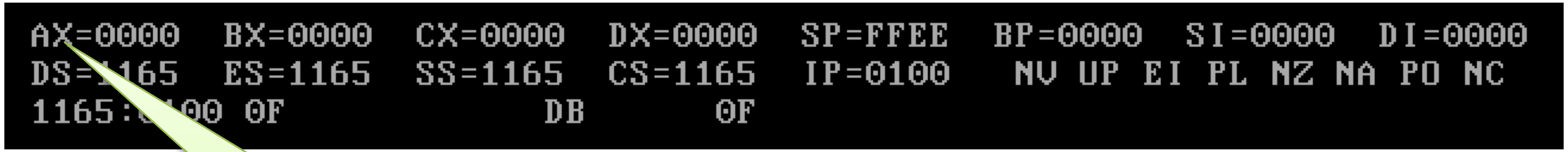
Instruction Pointer (**IP**) : points to the next instruction to be fetched for execution



```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0200   NV UP EI PL ZR NA PE NC
0ADF:0200 A10001        MOV     AX,[0100]                          DS:0100=2020


AX=2020  BX=0000  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0203   NV UP EI PL ZR NA PE NC
0ADF:0203 8B1E0901      MOV     BX,[0109]                          DS:0109=3030


AX=2020  BX=3030  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0207   NV UP EI PL ZR NA PE NC
0ADF:0207 01D8          ADD     AX,BX


AX=5050  BX=3030  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=0209   NV UP EI PL NZ NA PE NC
0ADF:0209 09D8          OR      AX,BX


AX=7070  BX=3030  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=020B   NV UP EI PL NZ NA PO NC
0ADF:020B 21D8          AND     AX,BX


AX=3030  BX=3030  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=020D   NV UP EI PL NZ NA PE NC
0ADF:020D 31D8          XOR     AX,BX


AX=0000  BX=3030  CX=0000  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0000
DS=0ADF  ES=0ADF  SS=0ADF  CS=0ADF  IP=020F   NV UP EI PL ZR NA PE NC
0ADF:020F 7518          JNZ     0229
```

Assembly Language(x86)

- 8 bit registers : A, B, C, D, ...
- 16 bit registers : AX, BX, CX, DX, . . .
- 32 bit registers : EAX, EBX, ECX, EDX, . . .
- 64 bit registers : RAX, RBX, RCX, RDX, . . .

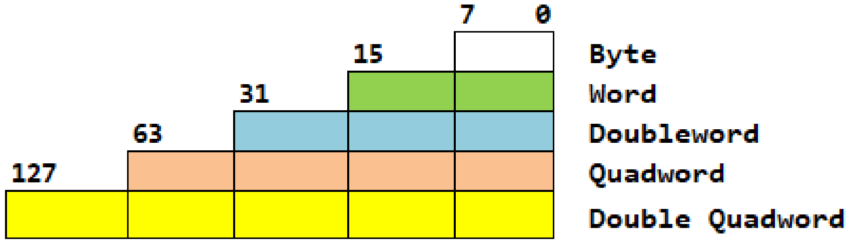


```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1165  ES=1165  SS=1165  CS=1165  IP=0100   NV UP EI PL NZ NA PO NC
1165:0100 0F             DB       0F
```

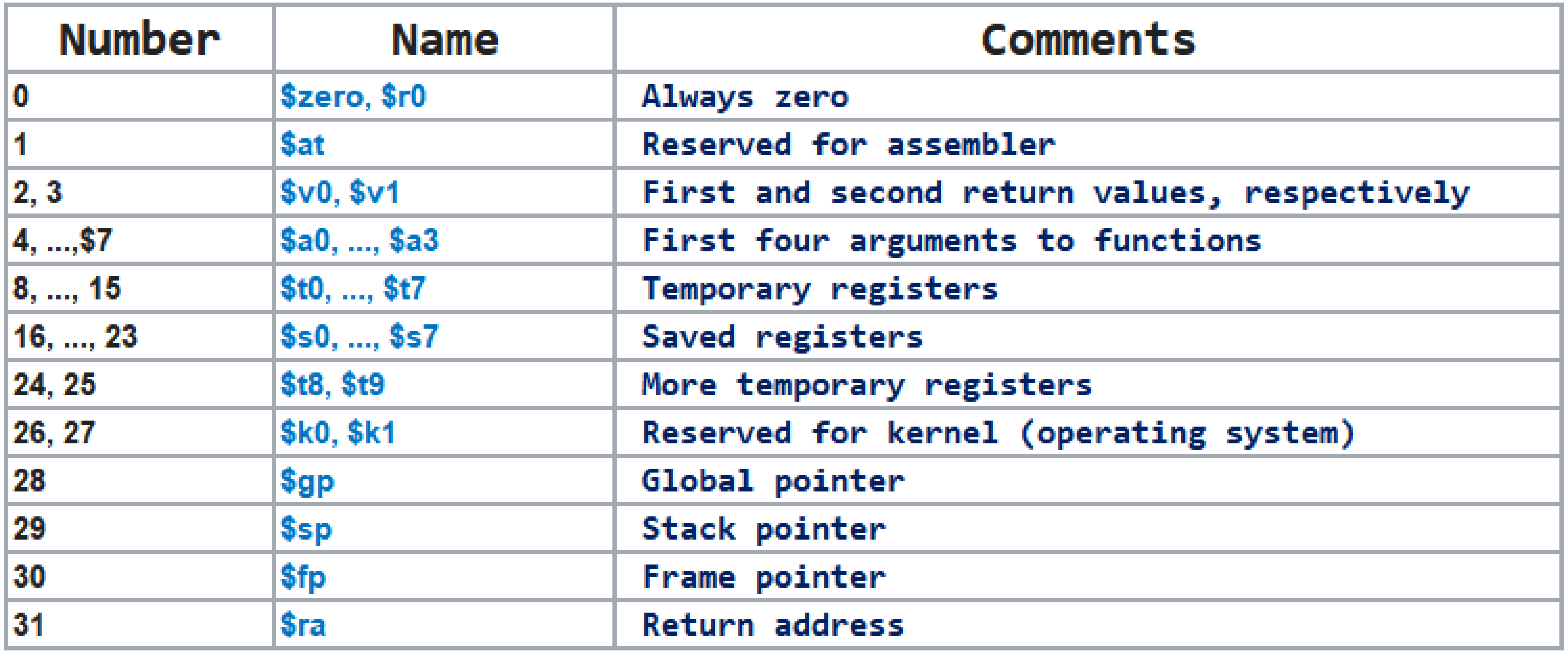
AX = 0000h(16 bits)

Assembly Language(x86) (x64)

64-bit	Lower 32-bit	Lower 16-bit	Lower 8-bit
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b



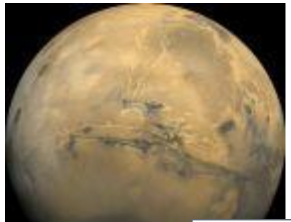
Assembly Language(MIPS)

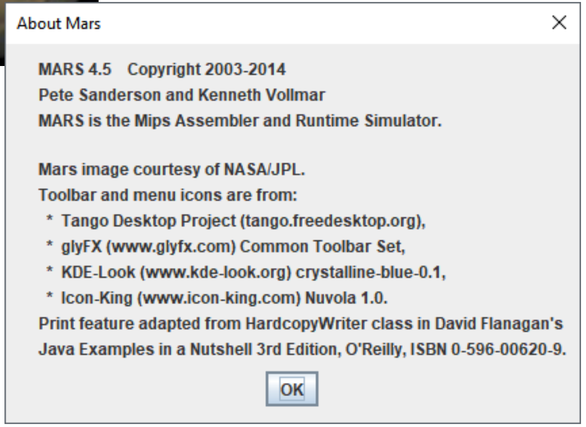


Number	Name	Comments
0	$zero, $r0	Always zero
1	$at	Reserved for assembler
2, 3	$v0, $v1	First and second return values, respectively
4, ...,$7	$a0, ..., $a3	First four arguments to functions
8, ..., 15	$t0, ..., $t7	Temporary registers
16, ..., 23	$s0, ..., $s7	Saved registers
24, 25	$t8, $t9	More temporary registers
26, 27	$k0, $k1	Reserved for kernel (operating system)
28	$gp	Global pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

Assembly Language(MIPS)

MARS



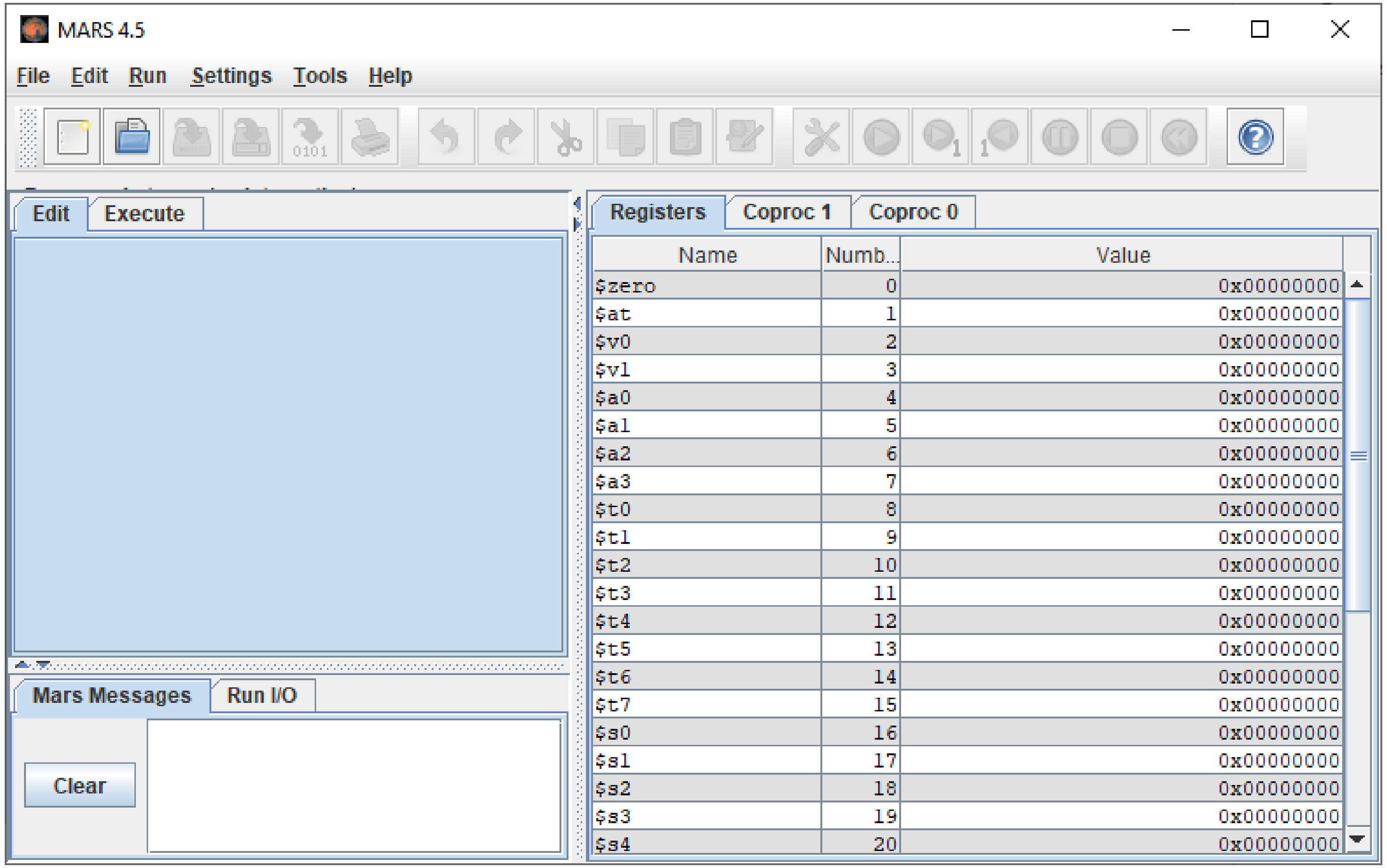


About Mars

MARS 4.5 Copyright 2003-2014
Pete Sanderson and Kenneth Vollmar
MARS is the Mips Assembler and Runtime Simulator.

Mars image courtesy of NASA/JPL.
Toolbar and menu icons are from:
- * Tango Desktop Project (tango.freedesktop.org),
- * glyFX (www.glyfx.com) Common Toolbar Set,
- * KDE-Look (www.kde-look.org) crystalline-blue-0.1,
- * Icon-King (www.icon-king.com) Nuvola 1.0.

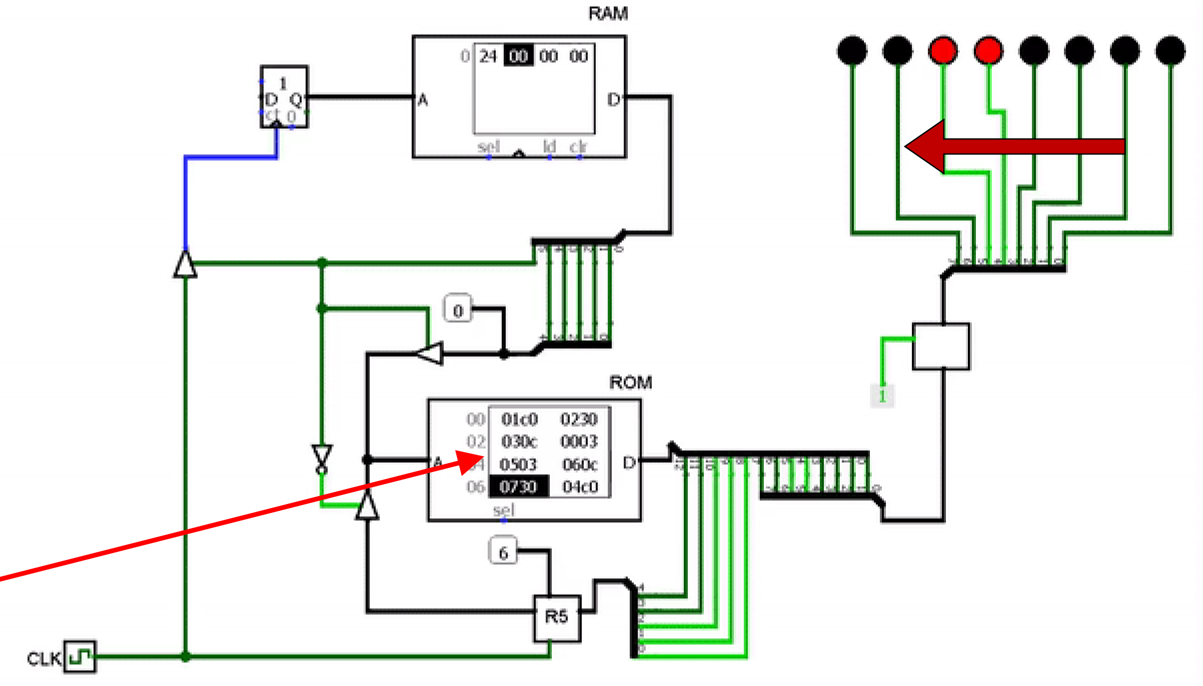
Print feature adapted from HardcopyWriter class in David Flanagan's Java Examples in a Nutshell 3rd Edition, O'Reilly, ISBN 0-596-00620-9.



Name	Numb...	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000

Hardwired vs Microprogrammed control

Microprogrammed Approach



```
0  24 00 00 00
```

```
00  01c0  0230
02  030c  0003
04  0503  060c
06  0730  04c0
```

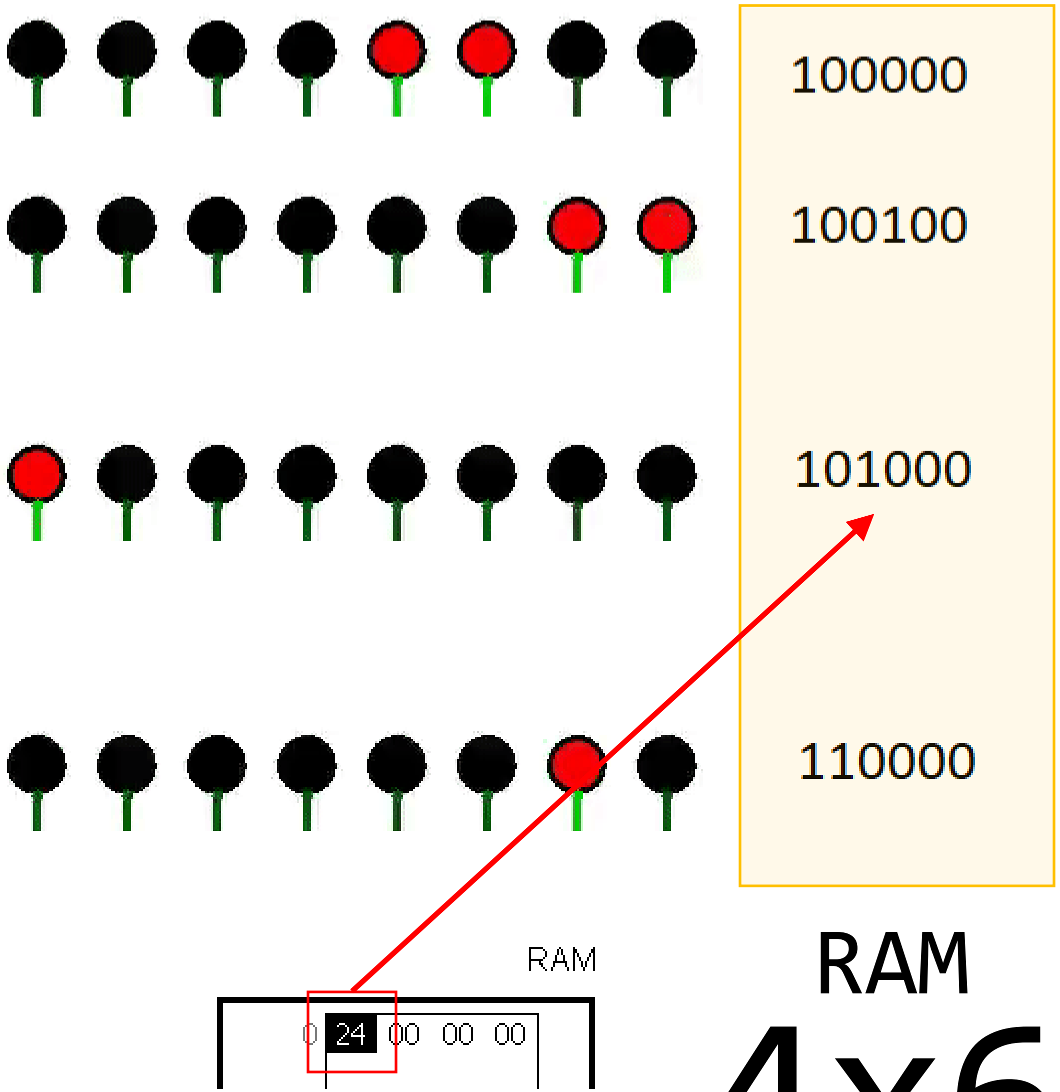
Microprogram

https://vimeo.com/414529620

Do we need to design a new circuit?

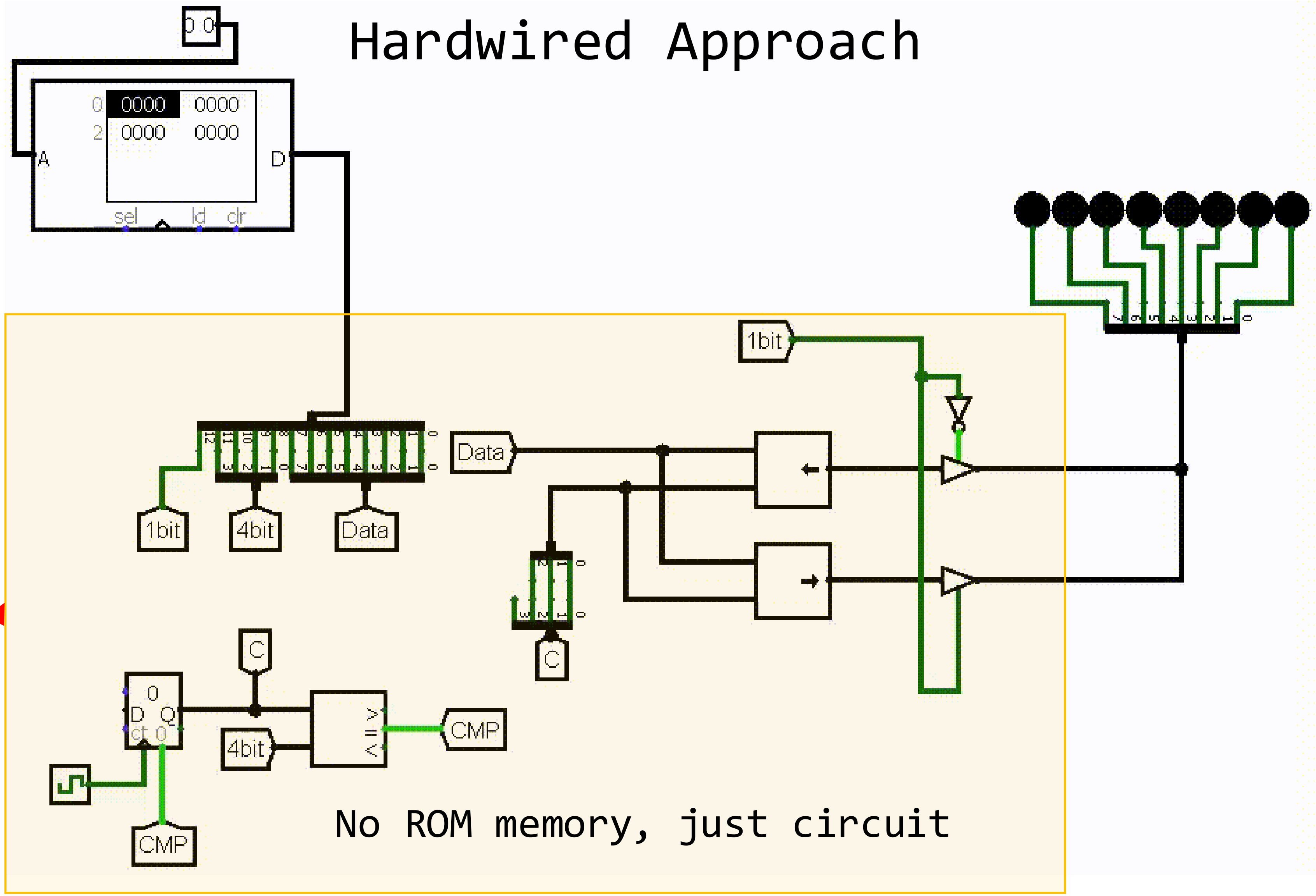
	0	1	2	3	4	5	6	7	8	9	10	11	12	0x
0	0	0	0	0	1	1	1	0	0	0	0	0	0	1c0
1	0	0	0	1	0	0	0	1	1	0	0	0	0	230
2	0	0	0	1	1	0	0	0	0	1	1	0	0	30c
3	0	0	0	0	0	0	0	0	0	0	0	1	1	3
4	0	0	1	0	1	0	0	0	0	0	0	1	1	503
5	0	0	1	1	0	0	0	0	0	1	1	0	0	60c
6	0	0	1	1	1	0	0	1	1	0	0	0	0	730
7	0	0	1	0	0	1	1	0	0	0	0	0	0	4C0
8	0	1	0	0	1	1	0	0	0	0	0	0	0	980
9	0	1	0	1	0	0	1	0	0	0	0	0	0	A40
10	0	1	0	1	1	0	0	1	0	0	0	0	0	B20
11	0	1	1	0	0	0	0	0	1	0	[illegible]	0	0	C10
12	0	1	1	0	1	0	0	0	[illegible]	[illegible]	[illegible]	0	0	D08
13	0	1	1	1	0	0	0	[illegible]	[illegible]	0	[illegible]	0	0	E04
14	0	1	1	1	1	0	0	0	0	0	0	1	0	F02
15	0	1	0	0	0	0	0	0	0	[illegible]	0	0	1	801
16	0	1	0	0	1	1	1	0	0	0	0	0	0	1101
17	0	1	0	1	0	0	1	0	0	0	0	0	0	1202
18	0	1	0	1	1	0	0	1	0	0	0	0	0	1304
19	0	1	1	0	0	0	0	0	1	0	0	0	0	1408
20	0	1	1	0	1	0	0	0	0	1	0	0	0	1510
21	0	1	1	1	0	0	0	0	0	0	1	0	0	1620
22	0	1	1	1	1	0	0	0	0	0	0	1	0	1740
23	0	1	0	0	0	0	0	0	0	0	0	0	1	1080
24	unused													
25														
26														
27														
28														
29														
30														
31														

ROM 32x13



RAM 4x6

Hardwired Approach



No ROM memory, just circuit

Hardwired
control

https://vimeo.com/414529276

Processors

Instruction Execution

Interpreter

It is possible to write a program that can imitate the function of a CPU.

A major advantage of the interpreter-based approach was the ability to design a simple processor with the complexity largely confined to the memory holding the interpreter. Thus a complex hardware design could be turned into a complex software design.

Nearly all new computers designed in the 1970s, from minicomputers to mainframes, were based on interpretation.

Processors

Instruction Execution
Interpreter

- **Benefits (simple computer with interpreted instructions)**
- The ability to fix incorrectly implemented instructions or make up for design deficiencies in the basic hardware
- The opportunity to add new instructions at minimal cost even after delivery of the machine
- Structured design that permitted efficient development, testing and documenting of complex instructions

Processors

RISC vs. CISC

During the late 70s there was experimentation with very complex instructions, made possible by the interpreter. Designers tried to close the semantic gap between what machines could do and what high-level programming languages required. Hardly anyone thought about designing simpler machines.

Reduced Instruction Set Computer (RISC)

- Did not use the interpretation
- Did not have to be backward compatible with existing products
- Small number of instructions, e.g. 50 (**DEC VAX** 200 to 300)

Complex Instruction Set Computer (CISC)

- Instructions, around 200-300, DEC VAX and IBM main-frames

Intel (486 up)

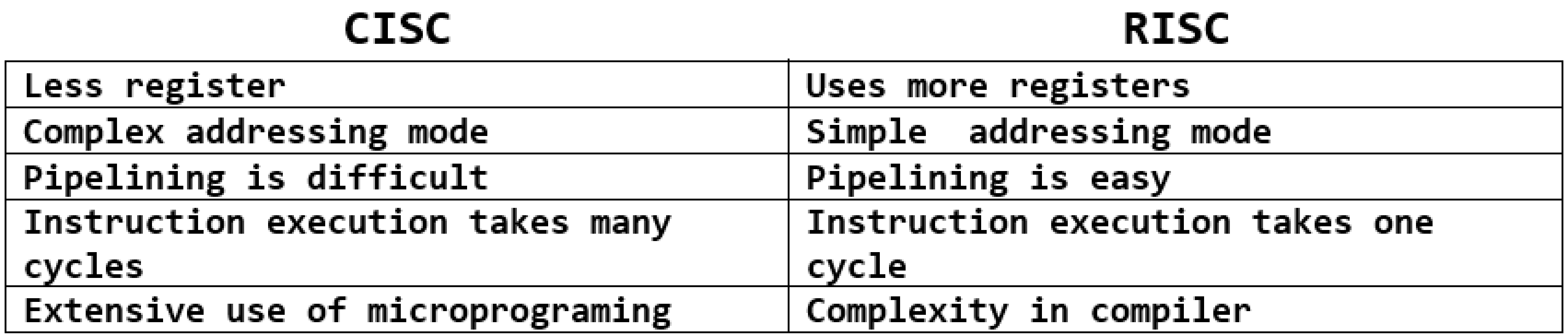
- A RISC core executes the simplest (most common) instructions
- Interpreting the more complicated instructions in the usual CISC way

RISC vs. CISC Architecture

The two instruction set architectures(CPU):

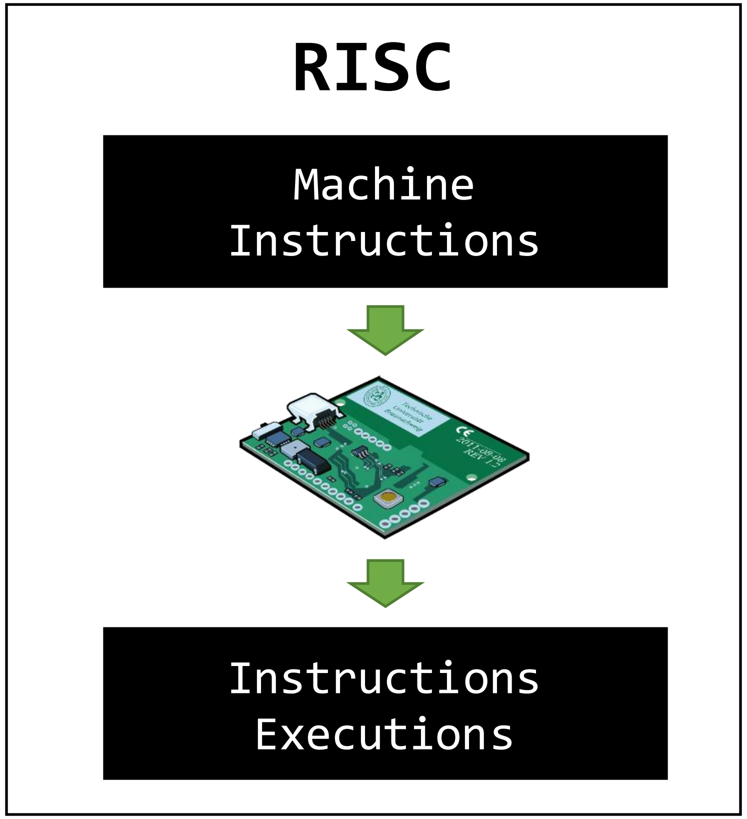
1) CISC (Complex Instruction Set Computer), **VAX, PDP-11, AMD, and Intel x86**

2) **RISC** (Reduced Instruction Set Computer), **MIPS, ARM, AVR**



CISC	RISC
Less register	Uses more registers
Complex addressing mode	Simple addressing mode
Pipelining is difficult	Pipelining is easy
Instruction execution takes many cycles	Instruction execution takes one cycle
Extensive use of microprograming	Complexity in compiler

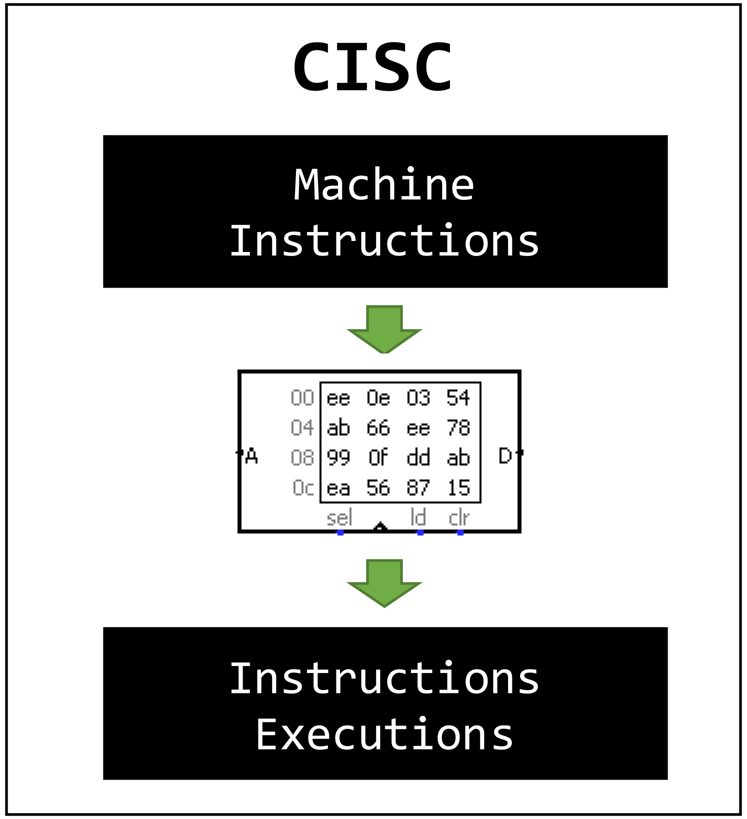
RISC vs. CISC Architecture



RISC

Machine Instructions

Instructions Executions



CISC

Machine Instructions

```
00 ee 0e 03 54
04 ab 66 ee 78
08 99 0f dd ab
0c ea 56 87 15
```

Instructions Executions

Processors

Design Principles for Modern Computers

- **Instructions directly executed by hardware**

– Eliminating a level of interpretation provides high speed for most instructions;

– Less frequently occurring instructions are acceptable

- **Maximize rate at which instructions are issued**

– Parallelism can play a major role in improving performance

- **Instructions should be easy to decode**

– A critical limit on the rate of issue of instructions is decoding individual instructions to determine what resources they need

– Fewer different formats for instructions, the better

Processors

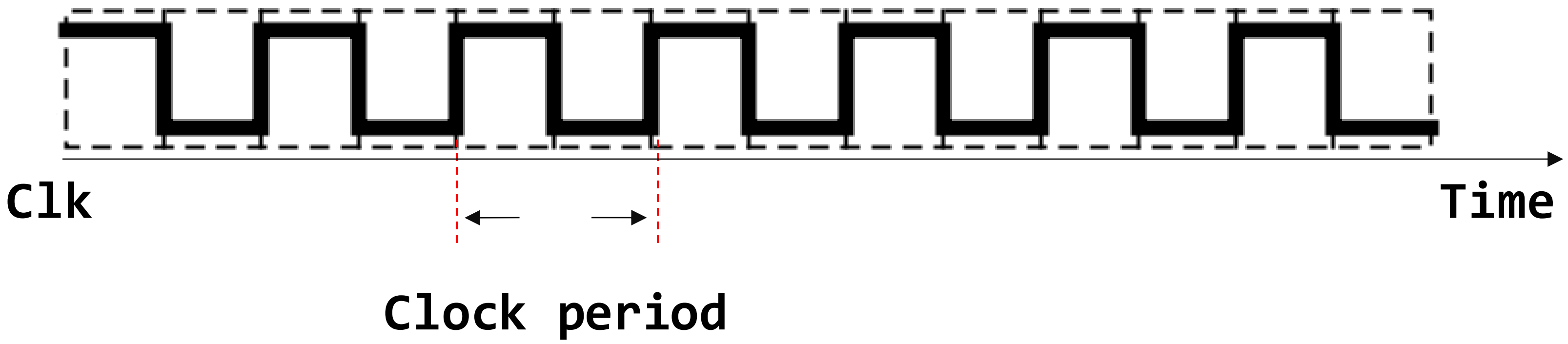
Design Principles for Modern Computers

- **Only loads, stores should reference memory**
- – Access to memory can take a long time
- – All other instructions should operate only on registers

- **Provide plenty of registers**
- – Running out of registers leads to flush them back to memory
- – Memory access leads to slow speed

Computer clock

Runs at a constant rate and determines when events take placed in hardware.



Cycle Time (CT) = Clock Cycle Time (clock period)

Frequency (f) = clock rate = 1/CT

For example, if a computer has a clock rate 200MHz, the cycle time is: $1/200x10^6$ = 5 ns

Performance measure

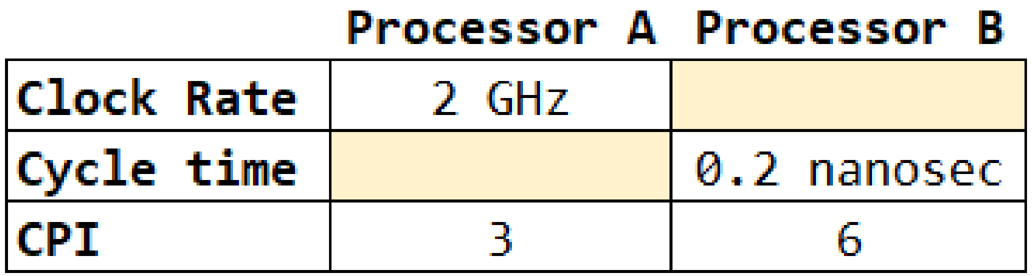
Execution time (CPU Time) = IC x CPI x CT

IC = Instruction Count
CPI = Cycle per instruction
CT = Clock Cycle Time

$$\text{Performance}_X = \frac{1}{\text{Execution time X}} \quad \text{, For program X}$$

$$\text{Relative Performance}_{XY} = \frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time Y}}{\text{Execution time X}}$$

Examples:



	Processor A	Processor B
Clock Rate	2 GHz	
Cycle time		0.2 nanosec
CPI	3	6

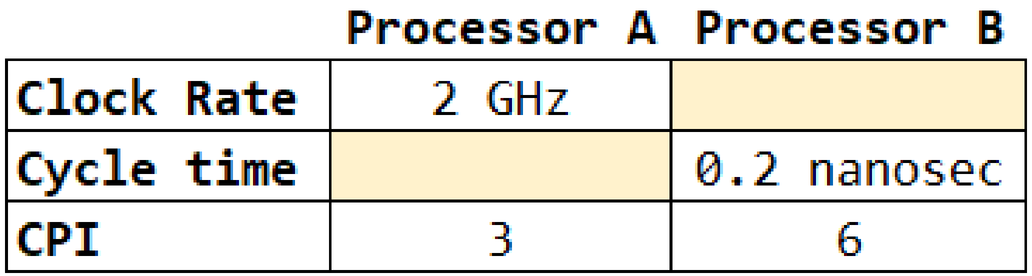
A. What is the Cycle Time for Processor A? (in nanoseconds)

B. What is the Clock Rate for Processor B? (in GHz)

C. Assume Processor A and B runs 1200 instructions each, what is the CPU time of Processor A and B? (in nanoseconds)

D. Which Processor is faster, and by how much? (Give the relative performance)

Solution:



	Processor A	Processor B
Clock Rate	2 GHz	
Cycle time		0.2 nanosec
CPI	3	6

What is the Cycle Time for Processor A? (in nanoseconds)

Cycle time A = $1/(2 \times 10^{9})$= 0.5 nanosecond

What is the Clock Rate for Processor B? (in GHz)

Clock rate B = $1/(0.2 \times 10^{-9})$= 5 GHz

Assume Processor A and B runs 1200 instructions each, what is the CPU time of Processor A and B? (in nanoseconds)

$T_A = IC_A \times CPI_A \times CT_A = 1200 \times 3 \times 0.5 \times 10^{-9} = 1800 \times 10^{-9}$ s = 1.8 microsecond

$T_B = IC_B \times CPI_B \times CT_B = 1200 \times 6 \times 0.2 \times 10^{-9} = 1440 \times 10^{-9}$ s = 1.44 microsecond

Which Processor is faster, and by how much? (Give the relative performance)

$Pf_B/Pf_A = T_A/T_B = 1.25$

Parallelism

Instruction-Level Parallelism
Pipelining

Processor-Level Parallelism
Multiprocessors

Processor-Level Parallelism
Multicomputers

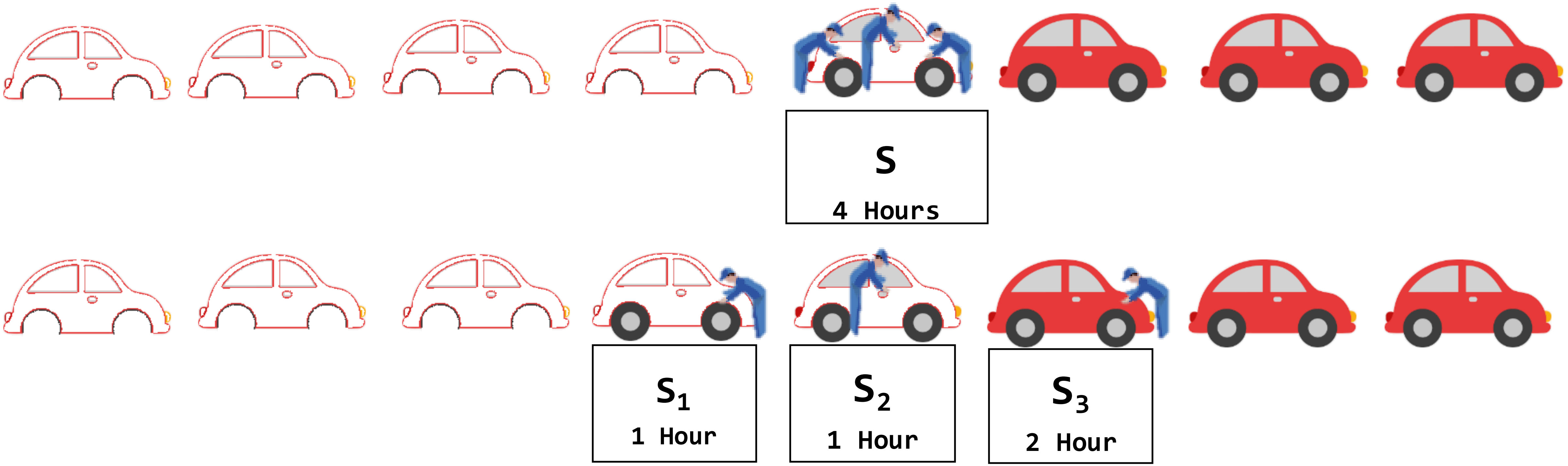
Processors

Instruction-Level Parallelism

Pipelining

Example: Car factory (the process of making a car)

Which assembly line is faster ?



S

4 Hours

S_1

1 Hour

S_2

1 Hour

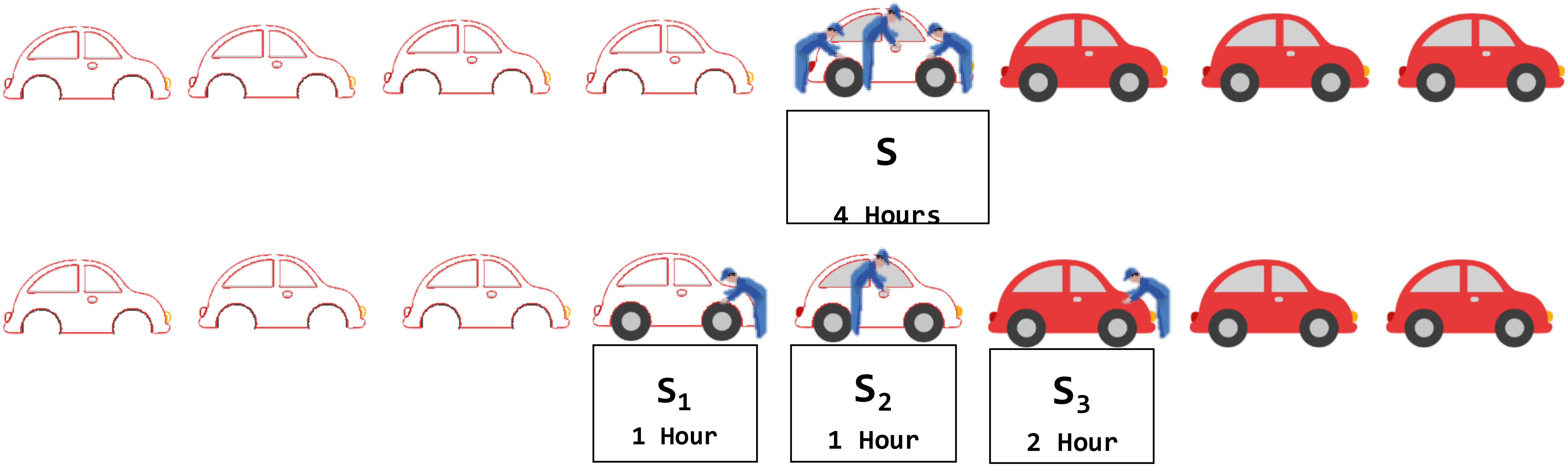
S_3

2 Hour

Example: Car factory (the process of making a car)
Which assembly line is faster ?

The number of cars made by the factory

Cycle Time (CT)
Every 4 hours 1 car



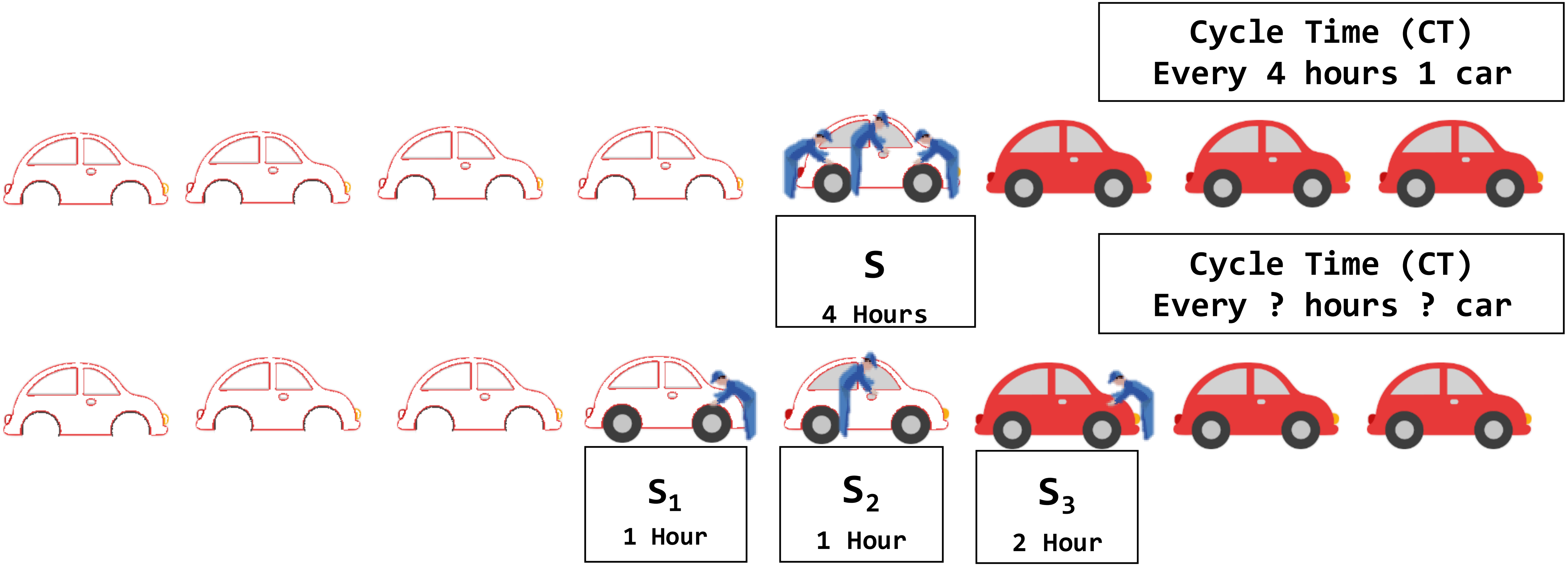
S
4 Hours

S_1
1 Hour

S_2
1 Hour

S_3
2 Hour

Example: Car factory (the process of making a car)
Which assembly line is faster ?



Cycle Time (CT)
Every 4 hours 1 car

S
4 Hours

Cycle Time (CT)
Every ? hours ? car

S_1
1 Hour

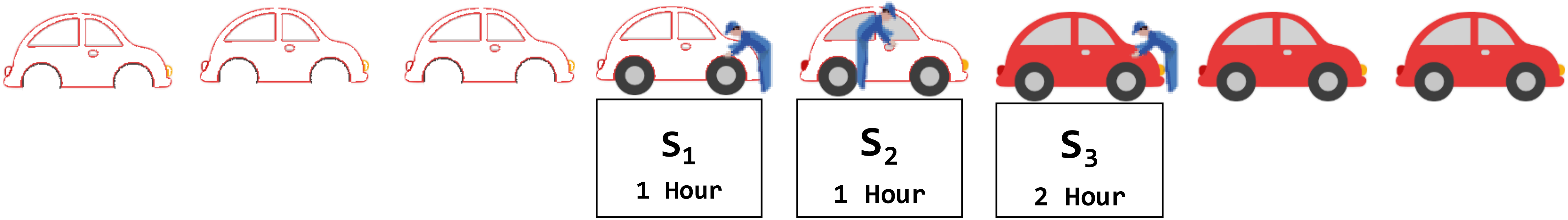
S_2
1 Hour

S_3
2 Hour

Example: Car factory

Is it possible to improve the cycle time ? How?

Cycle Time (CT)
Every 2 hours 1 car

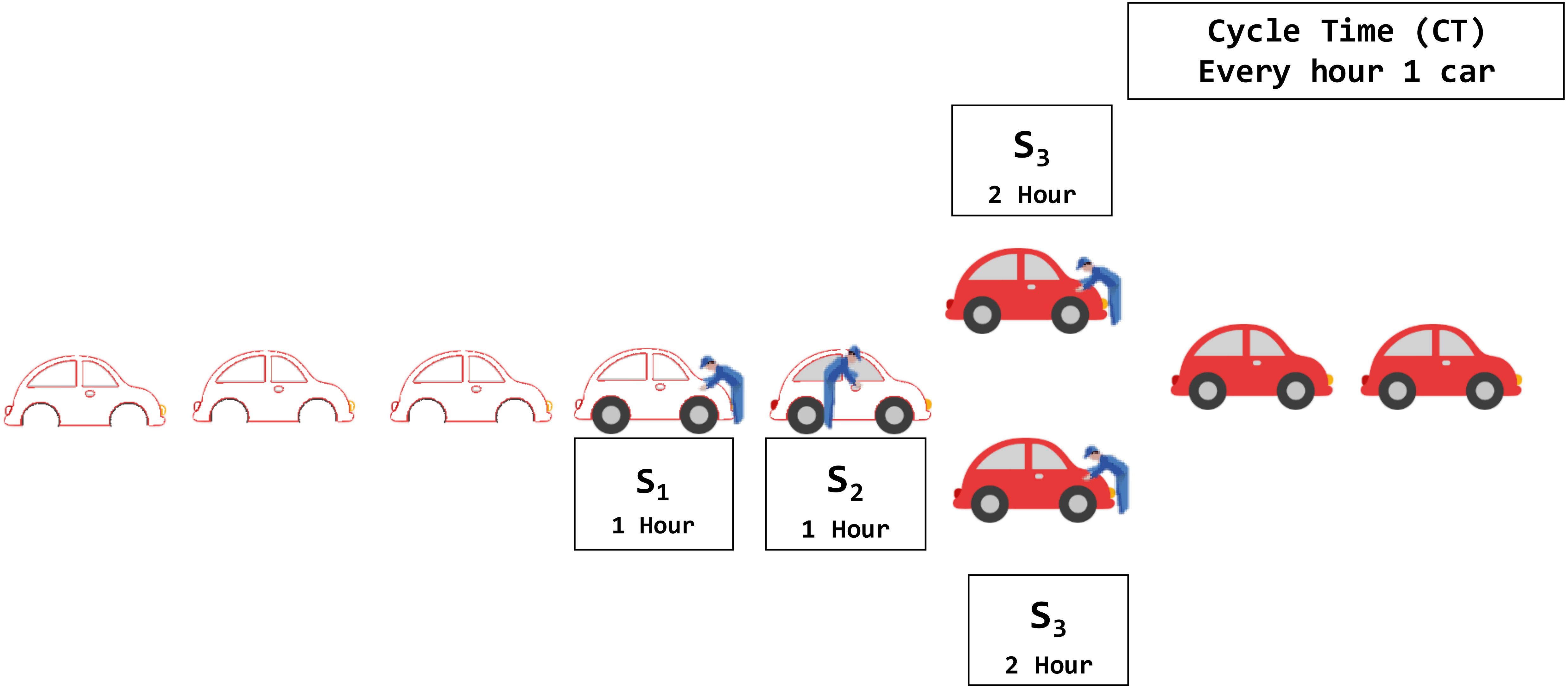


S_1
1 Hour

S_2
1 Hour

S_3
2 Hour

Example: Car factory
Is it possible to improve the cycle time ?



Cycle Time (CT)
Every hour 1 car

S_3
2 Hour

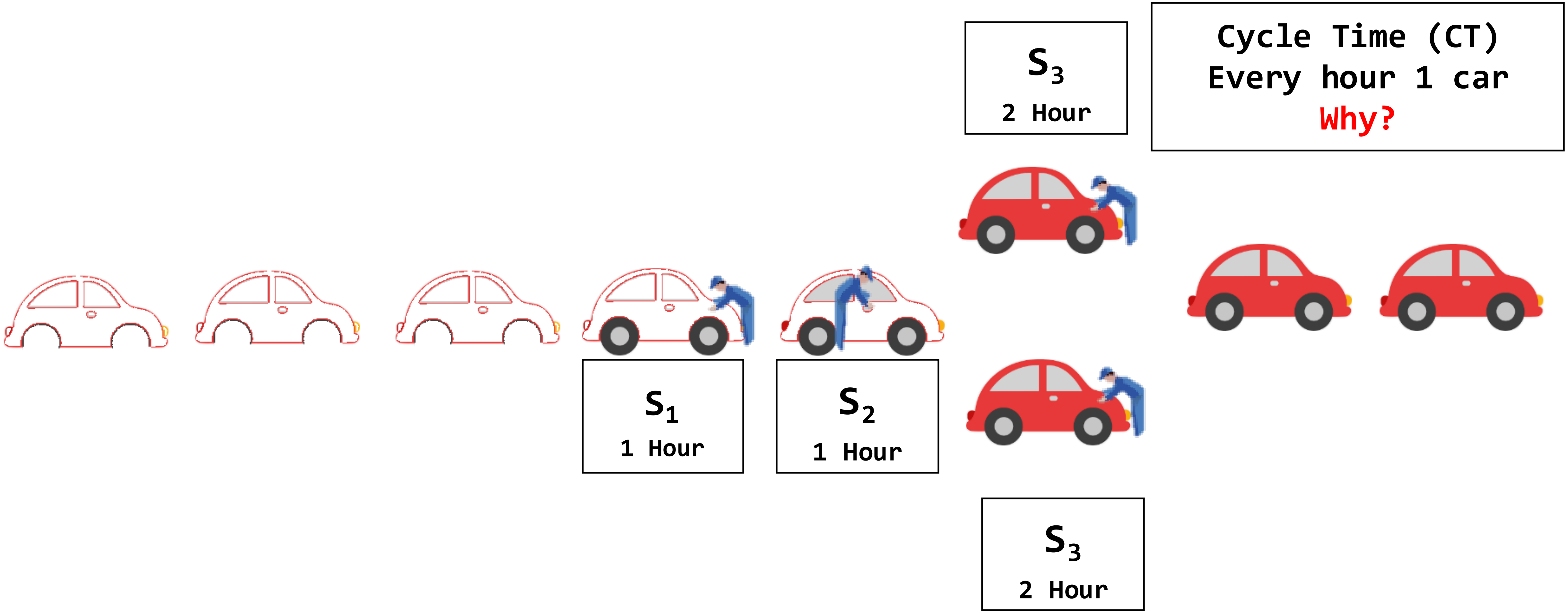
S_1
1 Hour

S_2
1 Hour

S_3
2 Hour

Example: Car factory

Is it possible to improve the cycle time ? yes



S_3

2 Hour

Cycle Time (CT)

Every hour 1 car

Why?

S_1

1 Hour

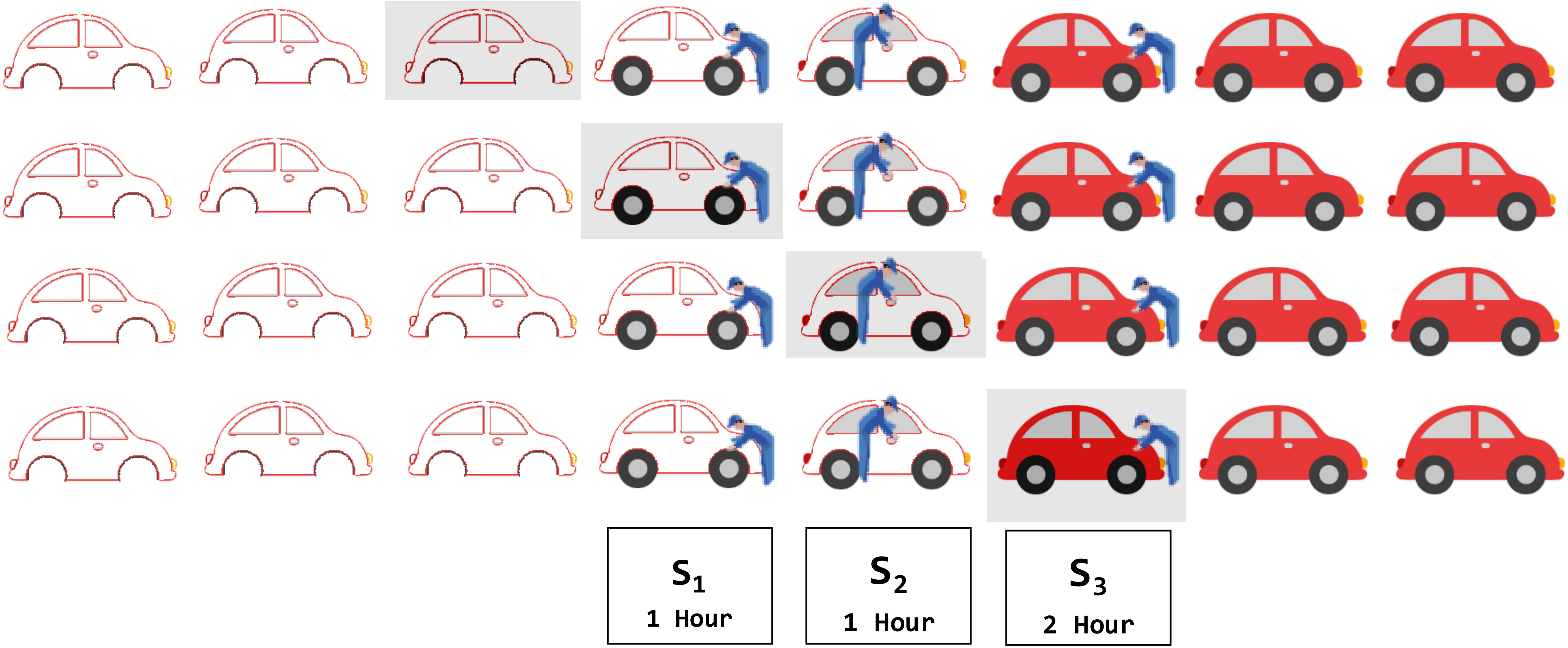
S_2

1 Hour

S_3

2 Hour

Example: Latency for each car

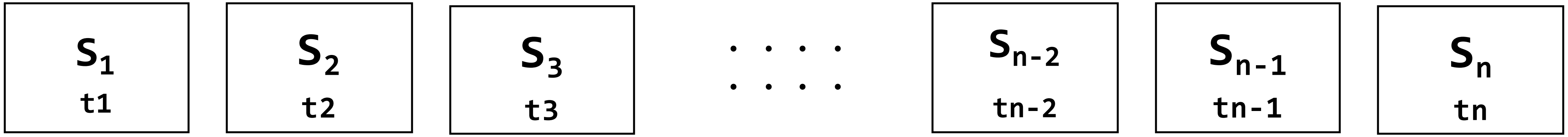


Latency for each car = 2 + 2 + 2 = 6 hours

Processors

Instruction-Level Parallelism

Pipelining



S_1 t1

S_2 t2

S_3 t3

. . . .

. . . .

S_{n-2} tn-2

S_{n-1} tn-1

S_n tn

$$\text{Clock period } \mathrm{T_p} = \max\{\mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_n\}$$

$$\text{Throughput} = \frac{1}{\mathrm{T_p}}$$

$$\text{Delay for each instruction} = \mathrm{nT_p}$$

Processors

Instruction-Level Parallelism

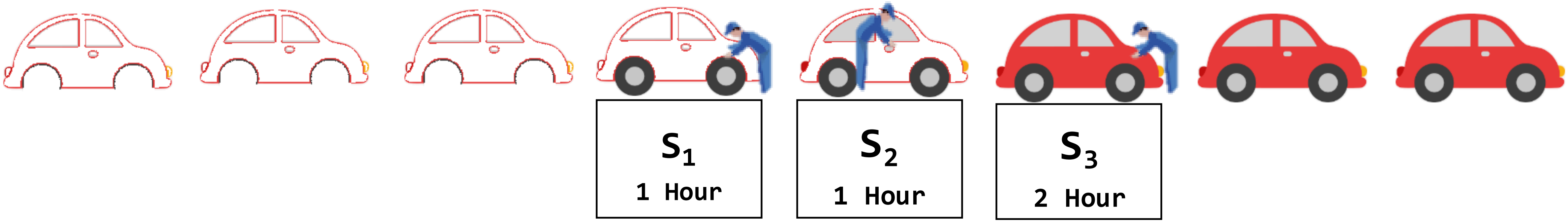
No Pipelining

$$\boxed{\begin{gathered}\text{Clock period } \mathrm{T_p} = \mathrm{t}_1 + \mathrm{t}_2 + \cdots + \mathrm{t}_n \\ \text{Throughput} = \frac{1}{\mathrm{T_p}}\end{gathered}}$$

$$\boxed{\text{Delay for each instruction} = \mathrm{T_p}}$$

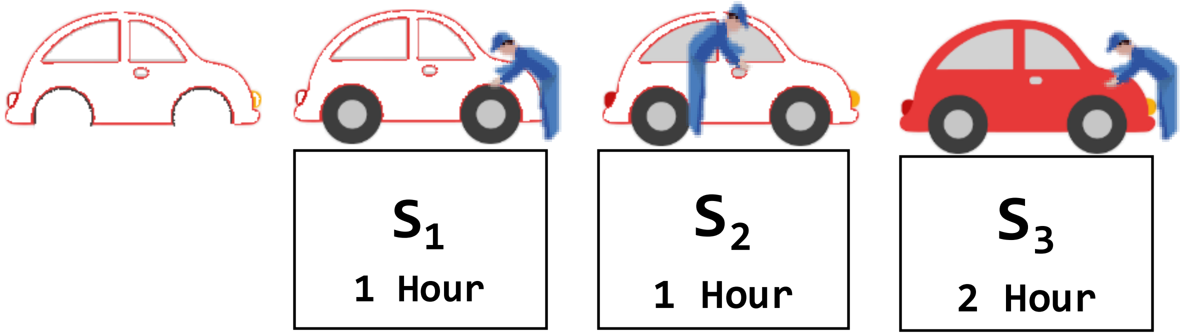
Example: Obtain the time taken to complete 10 cars on the assembly line.

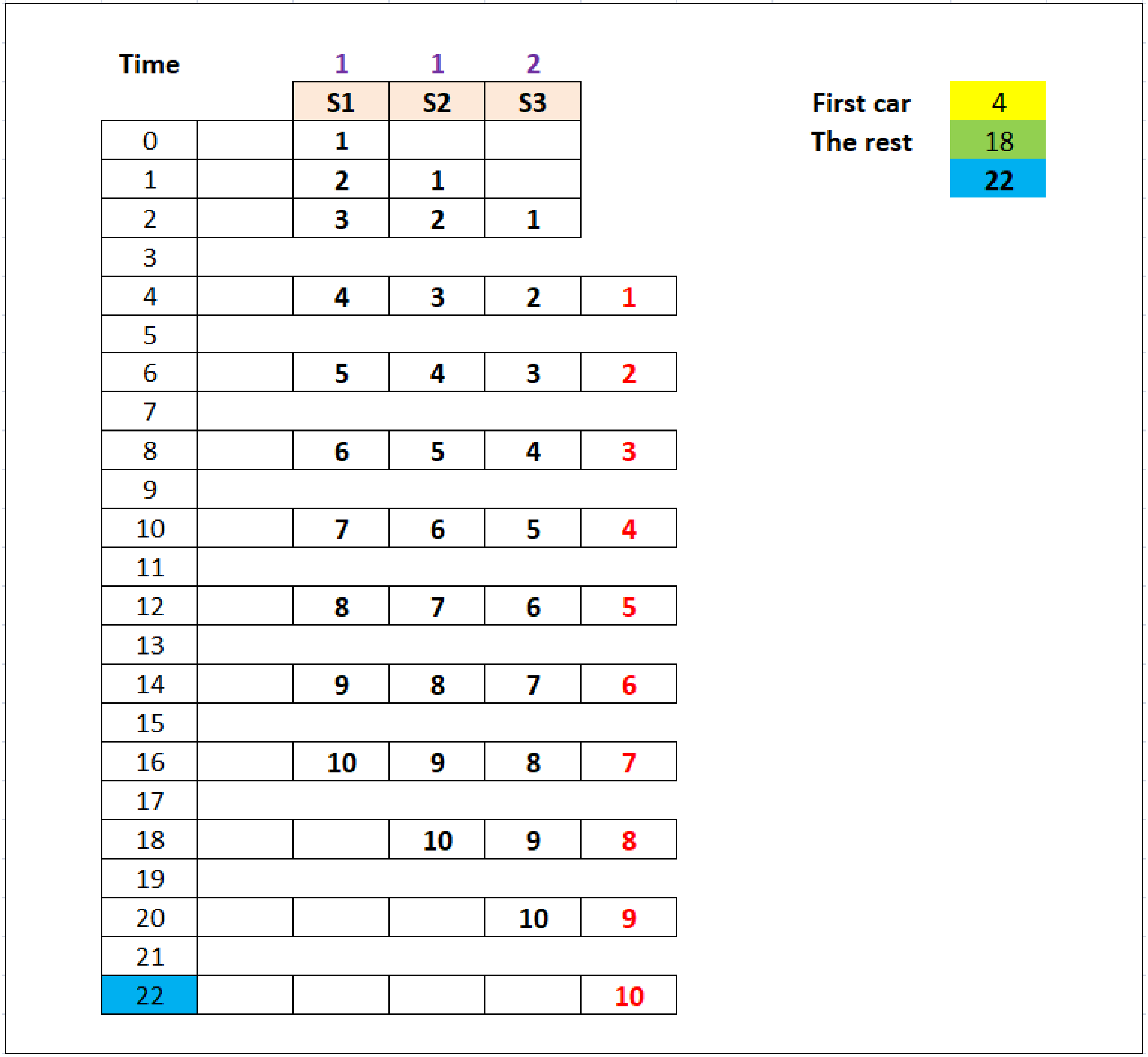
First car = 1 + 1 + 2 = 4
The rest = 9*2 = 18
Total time = 18 + 4 = 22



Example: Obtain the time taken to complete 10 cars on the assembly line.

First car = 1 + 1 + 2 = 4
The rest = 9*2 = 18
Total time = 18 + 4 = 22





Time		1	1	2	
		S1	S2	S3	
0		1			
1		2	1		
2		3	2	1	
3					
4		4	3	2	1
5					
6		5	4	3	2
7					
8		6	5	4	3
9					
10		7	6	5	4
11					
12		8	7	6	5
13					
14		9	8	7	6
15					
16		10	9	8	7
17					
18			10	9	8
19					
20				10	9
21					
22					10

First car	4
The rest	18
	22

Processors

Instruction-Level Parallelism

Pipelining

Parallelism is used to the improve performance of the machines

Pipelining

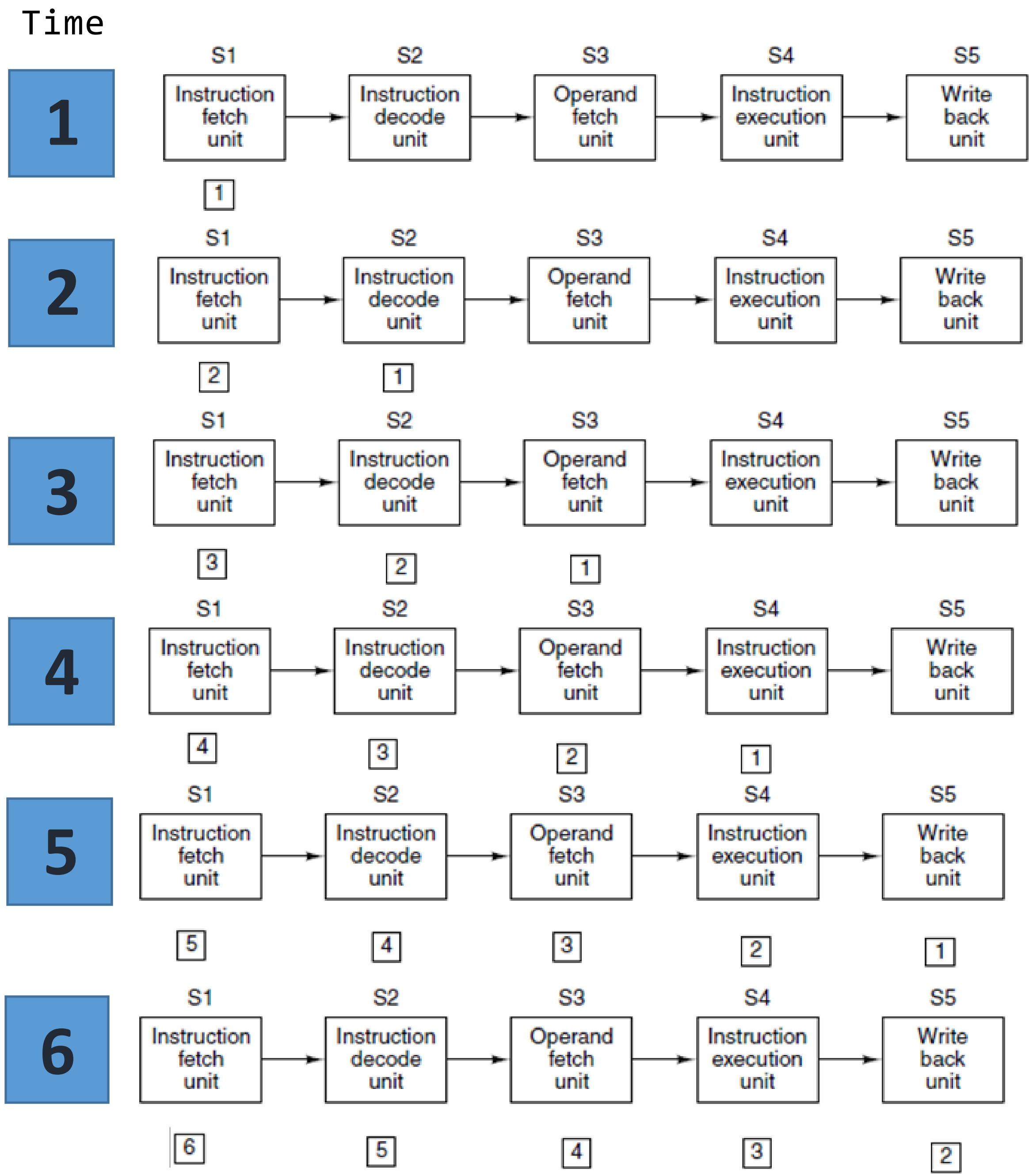
Instruction execution is often divided into many parts, each one handled by a dedicated piece of hardware, all of which can run in parallel.

Processors

Instruction-Level Parallelism

Pipelining

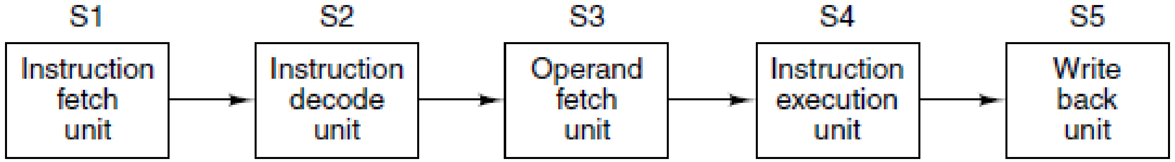
5-stage pipeline



Processors

Instruction-Level Parallelism
Pipelining

5-stage pipeline



(a)

S1:	1	2	3	4	5	6	7	8	9	
S2:		1	2	3	4	5	6	7	8	
S3:			1	2	3	4	5	6	7	. . .
S4:				1	2	3	4	5	6	
S5:					1	2	3	4	5	
	1	2	3	4	5	6	7	8	9	

Time →

(b)

Latency for each instruction with equal CT = number of stages (n) X cycle time (CT)

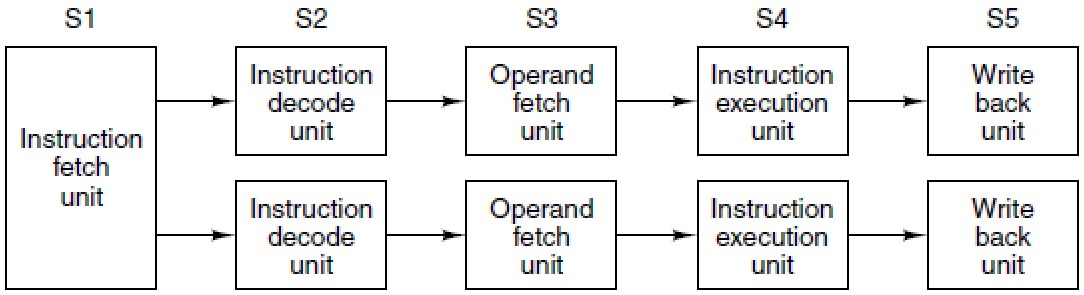
Processors

Instruction-Level Parallelism

Superscalar Architectures

- Single instruction fetch unit fetches pairs of instructions together and puts each one into its own pipeline,
- Not conflict over resource usage (e.g., registers)
- Neither must depend on the result of the other

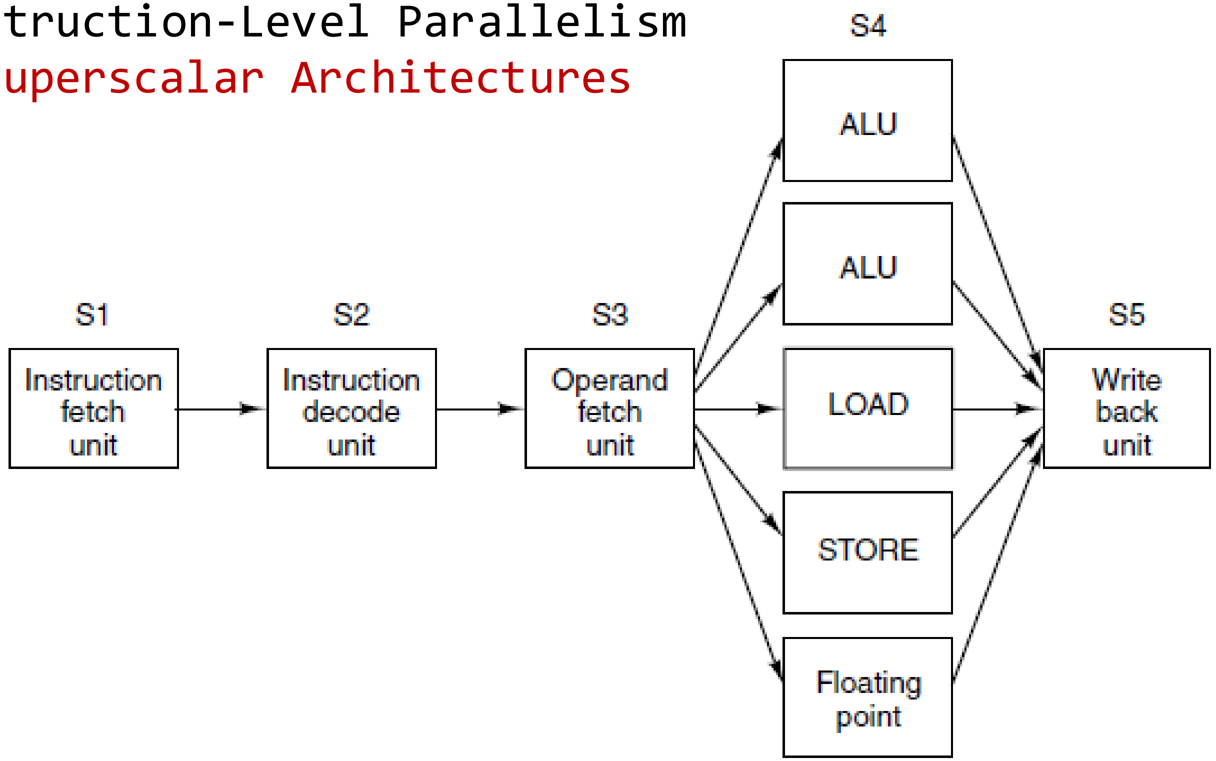
Dual five-stage pipeline



Processors

Instruction-Level Parallelism

Superscalar Architectures



It is possible to have multiple ALUs in stage (S4)

Implicit the idea of a superscalar processor is that:
– S3 stage can issue instructions considerably faster than the S4 stage is able to execute them

Processors

Processor-Level Parallelism

Multiprocessors

Tightly coupled (able to interact closely)

Easier to programming

Two types: Local memory and shared memory

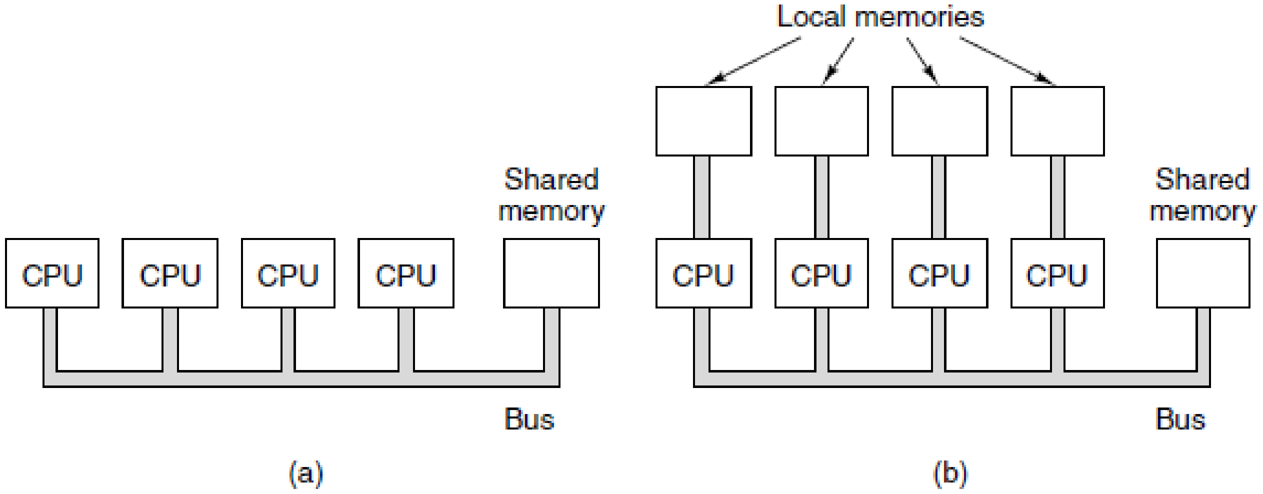


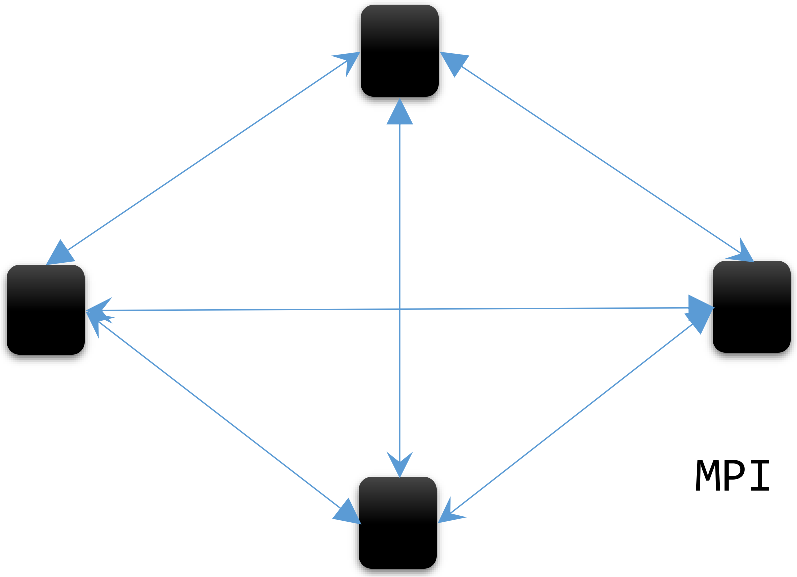
Figure 2-8. (a) A single-bus multiprocessor. (b) A multicomputer with local memories.

Processors

Processor-Level Parallelism

Multi-computers

- Multiple-Computers (Loosely coupled)
- Easier to build
- CPUs in a multicomputer communicate by sending each other messages, something like email, but much faster



MPI = message passing interface