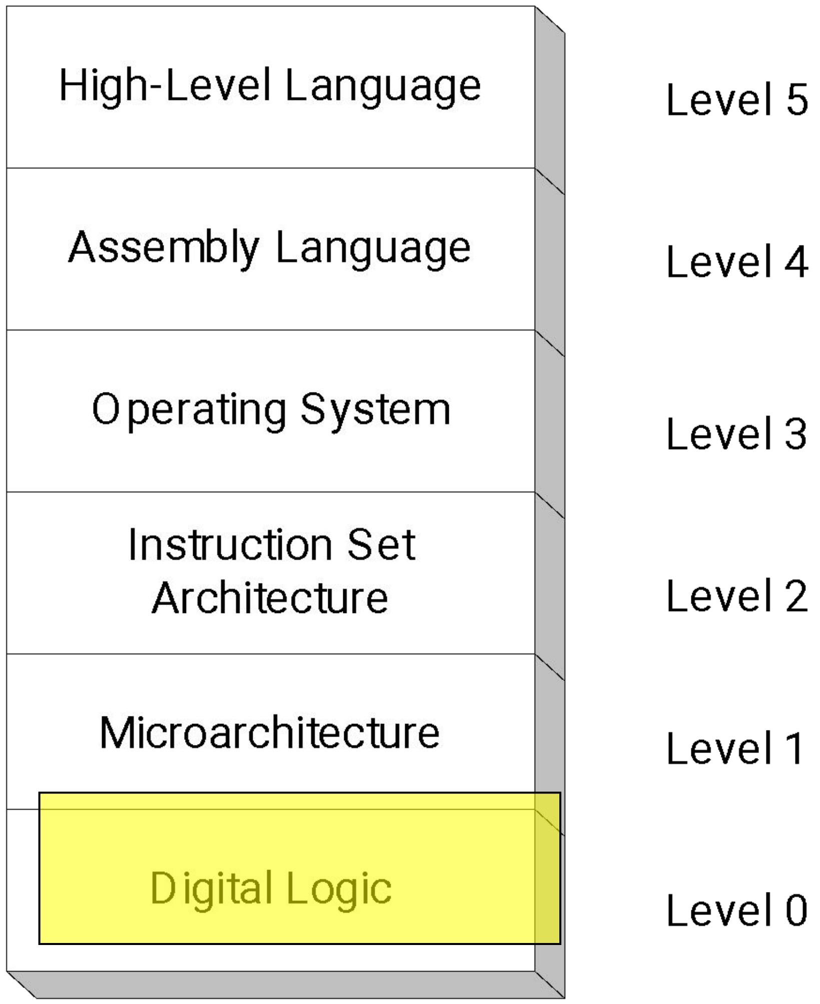
IA-32 Architecture

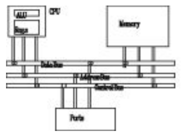
Computer Organization and Assembly Languages

with slides by Kip Irvine and Keith Van Rhein

Virtual machines

Abstractions for computers

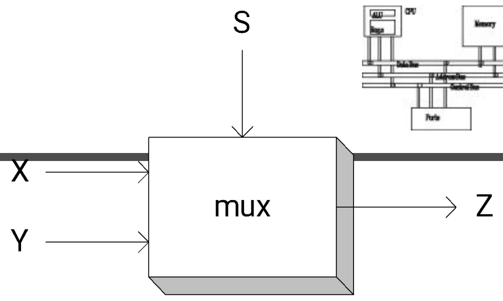




Truth tables

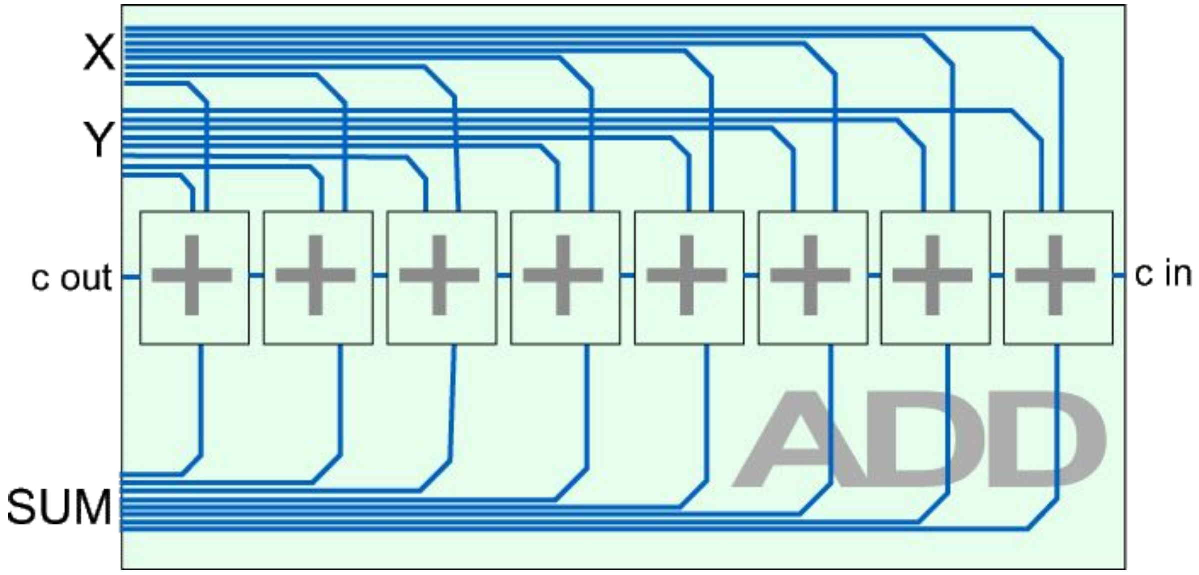
- Example: (Y ∧ S) ∨ (X ∧ ¬S)

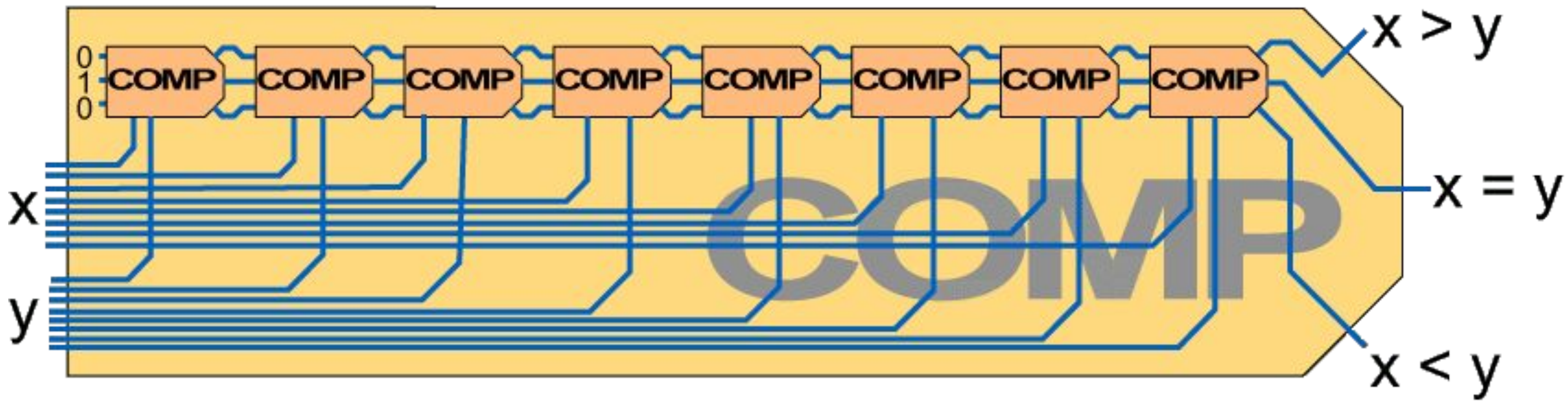
X	Y	S	(Y ∧ S) ∨ (X ∧ ¬S)
F	F	F	F
F	T	F	F
T	F	F	T
T	T	F	T
F	F	T	F
F	T	T	T
T	F	T	F
T	T	T	T



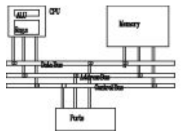
Two-input multiplexer

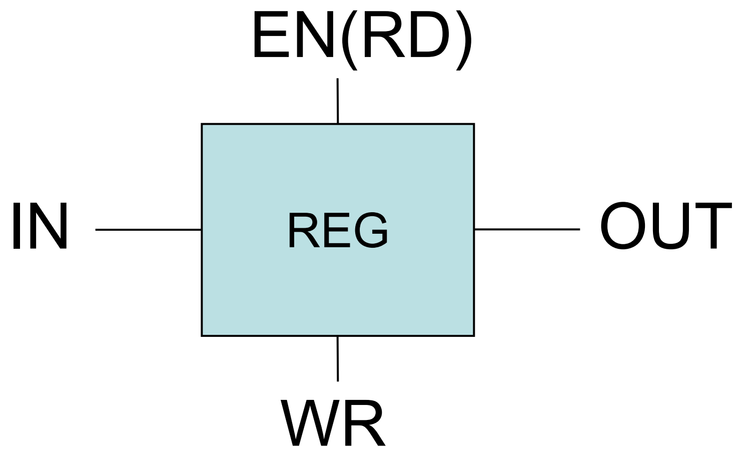
Combinational logic



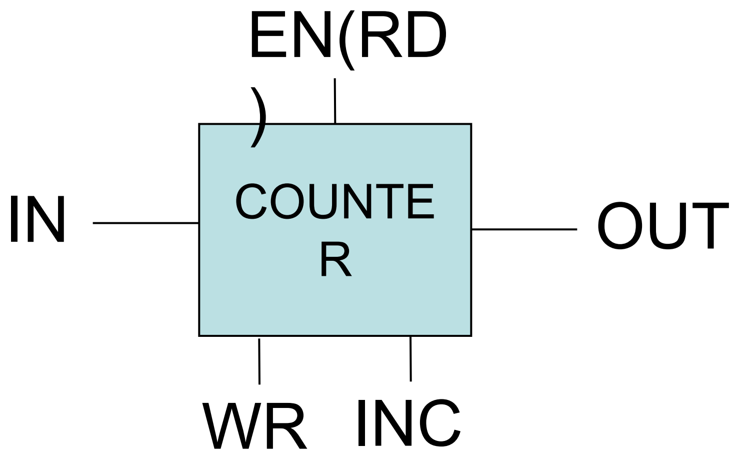


Sequential logic



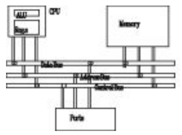


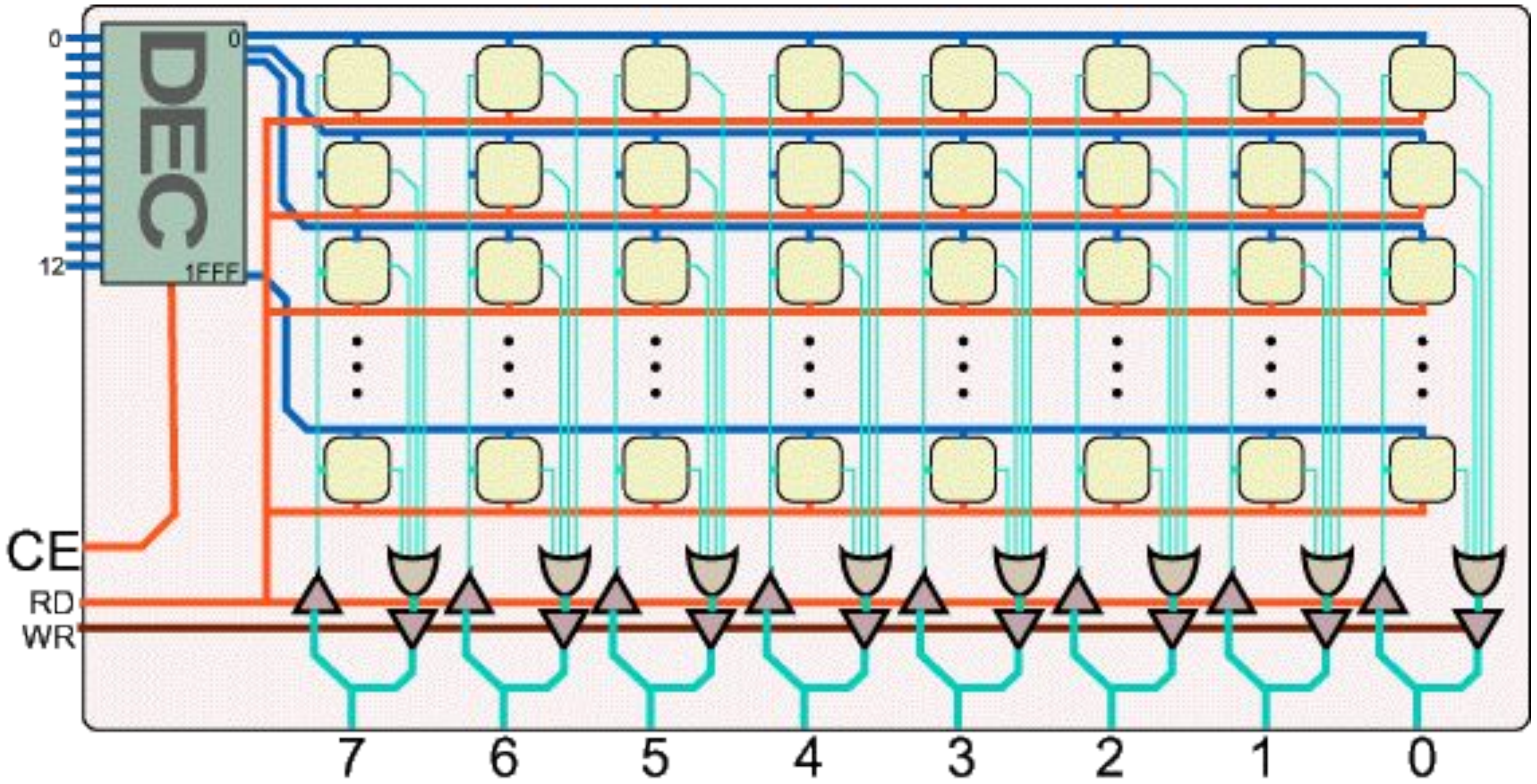
register



counter

Memory

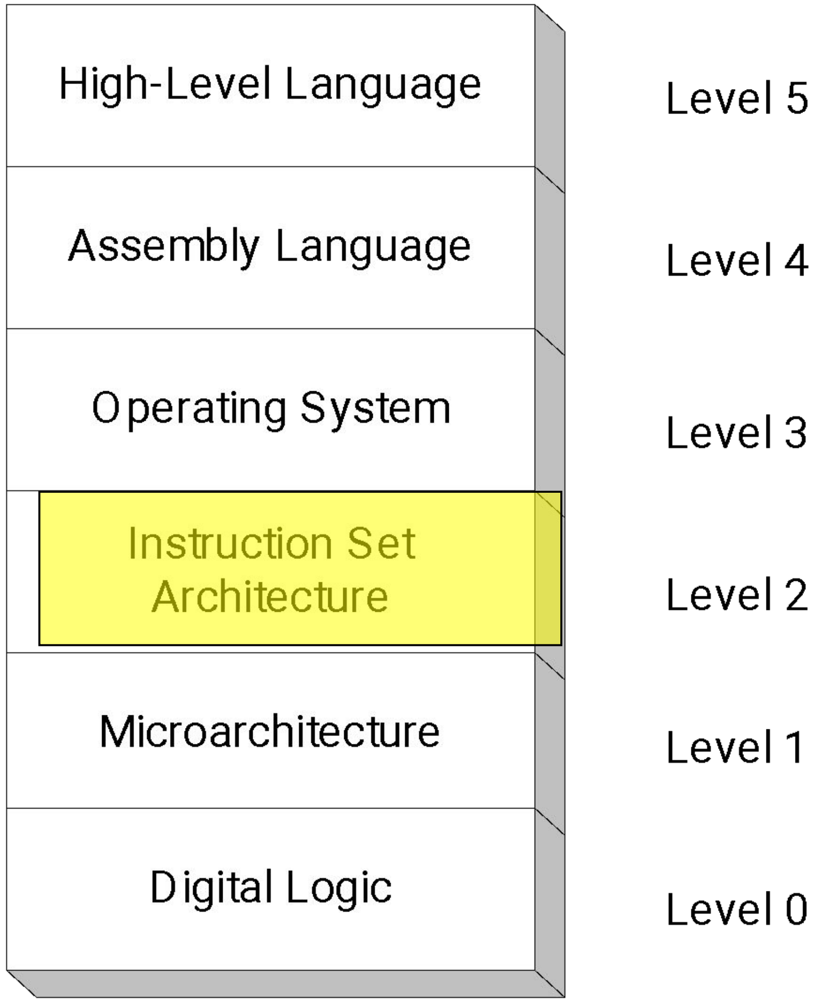




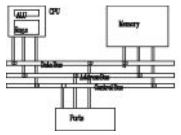
8K 8-bit memory

Virtual machines

Abstractions for computers



Instruction set

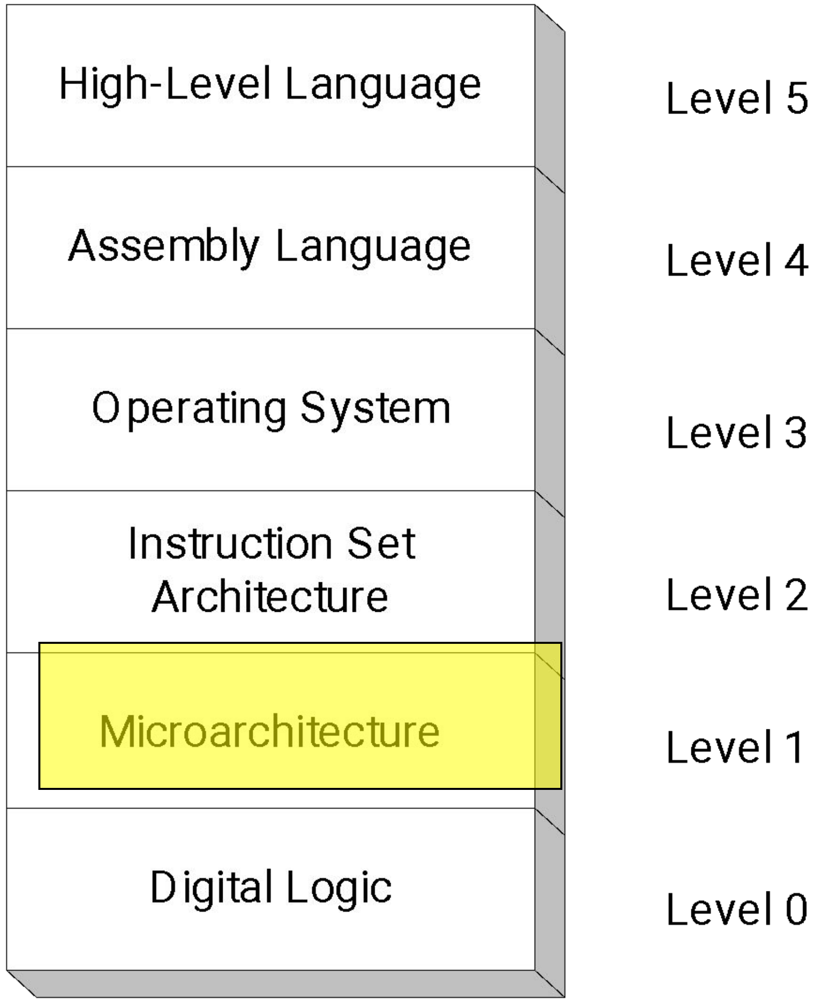


OPCODE	MNEMONIC	OPCODE	MNEMONIC
0	NOP	A	CMP addr
1	LDA addr	B	JG addr
2	STA addr	C	JE addr
3	ADD addr	D	JL addr
4	SUB addr		
5	IN port		
6	OUT port		
7	JMP addr		
8	JN addr		
9	HLT		

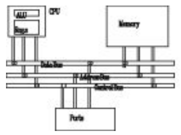
OPCODE	OPERAND
4	12

Virtual machines

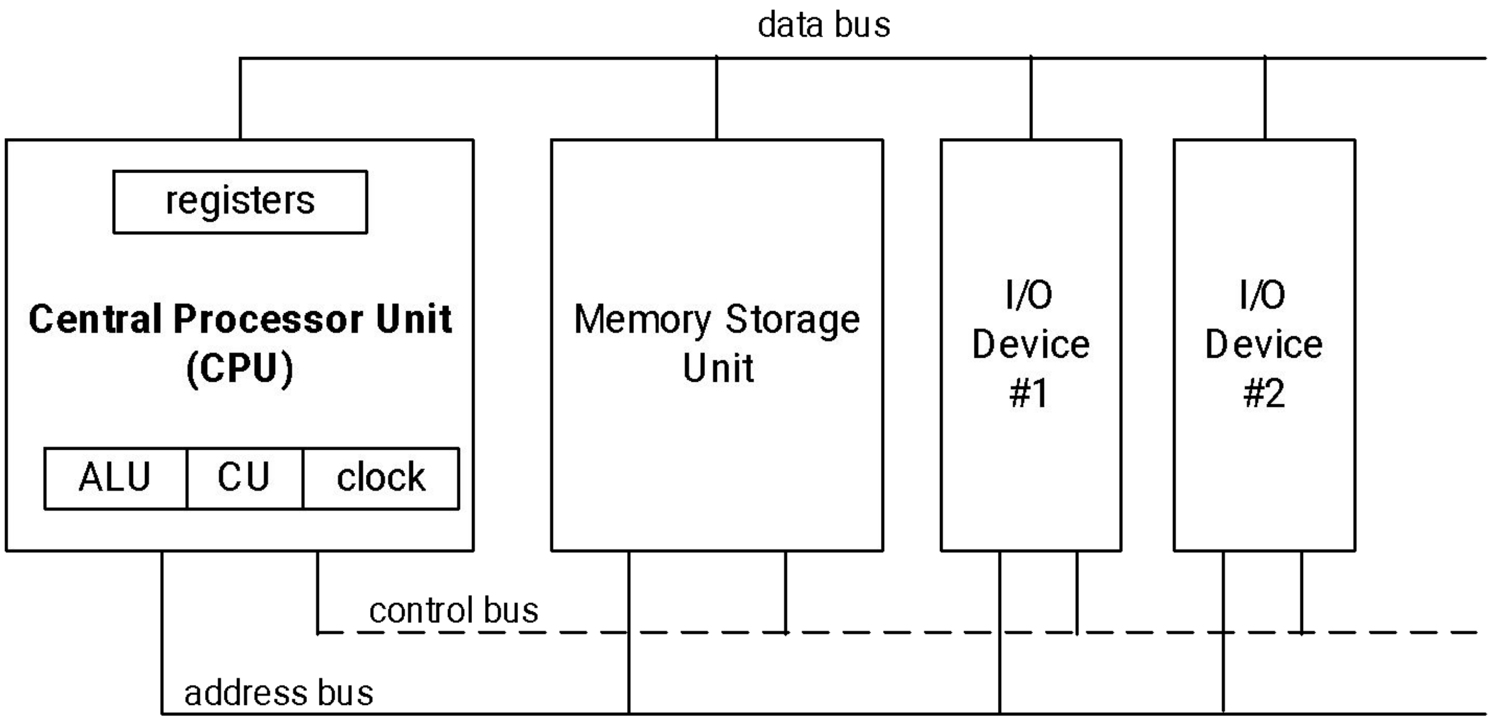
Abstractions for computers



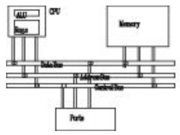
Basic microcomputer design



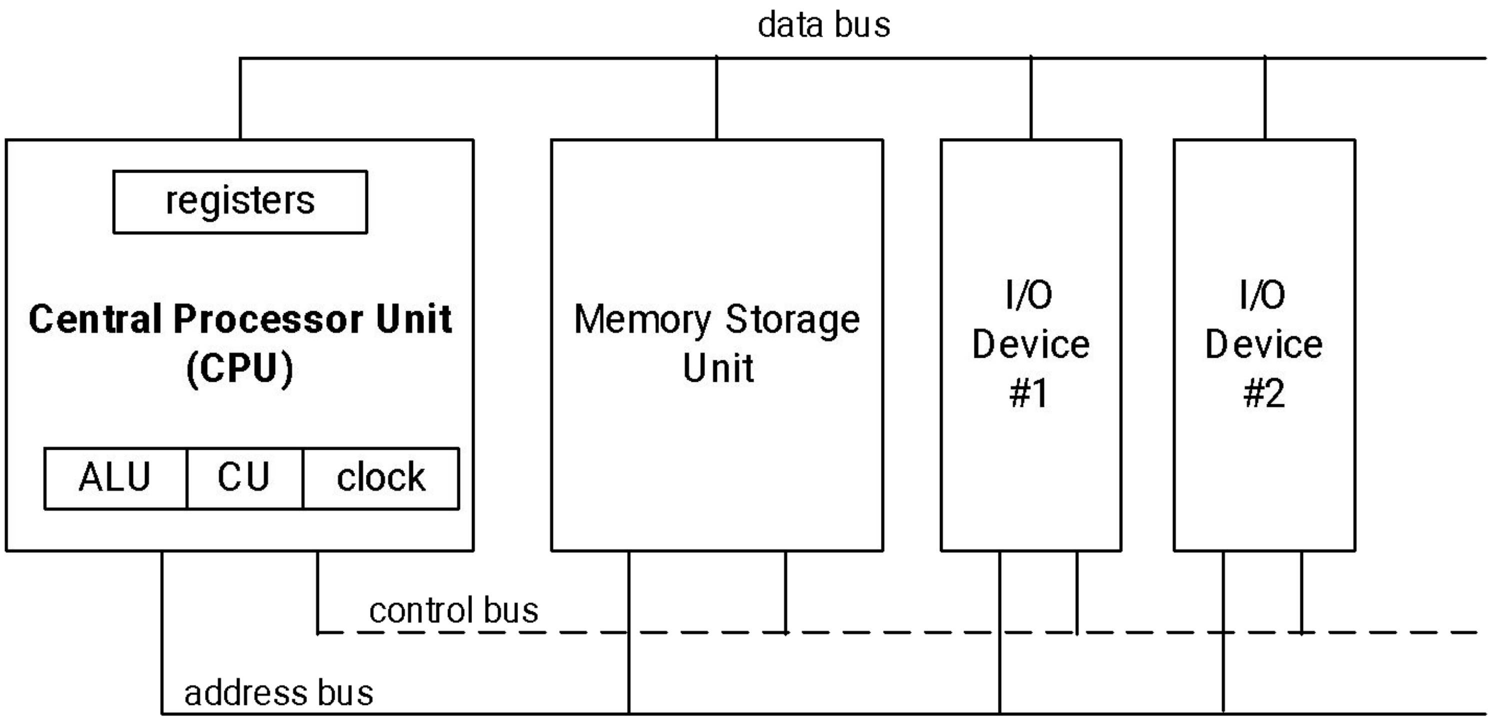
- clock synchronizes CPU operations
- control unit (CU) coordinates sequence of execution steps
- ALU performs arithmetic and logic operations



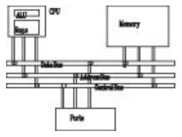
Basic microcomputer design



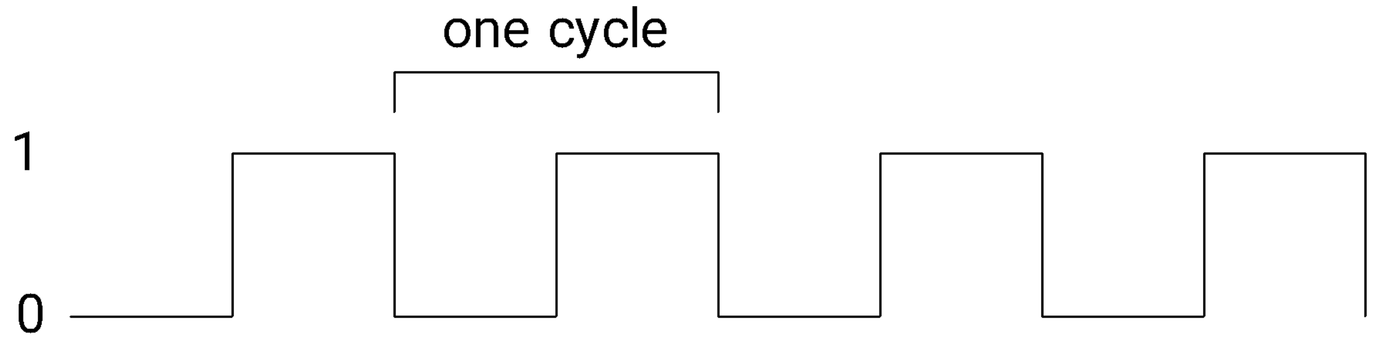
- The memory storage unit holds instructions and data for a running program
- A bus is a group of wires that transfer data from one part to another (data, address, control)



Clock

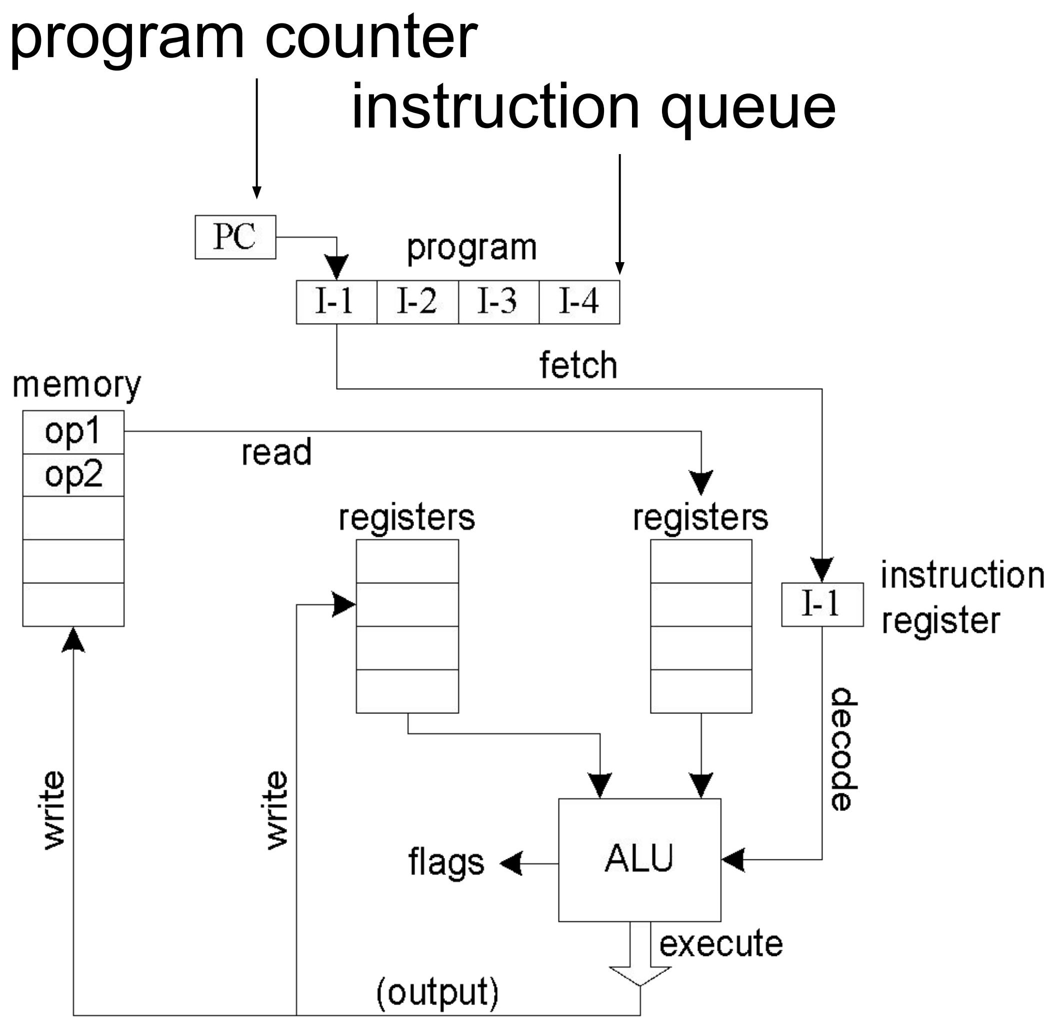


- synchronizes all CPU and BUS operations
- machine (clock) cycle measures time of a single operation
- clock is used to trigger events



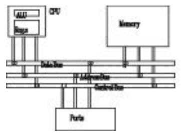
- Basic unit of time, 1GHz→clock cycle=1ns
- A instruction could take multiple cycles to complete, e.g. multiply in 8088 takes 50 cycles

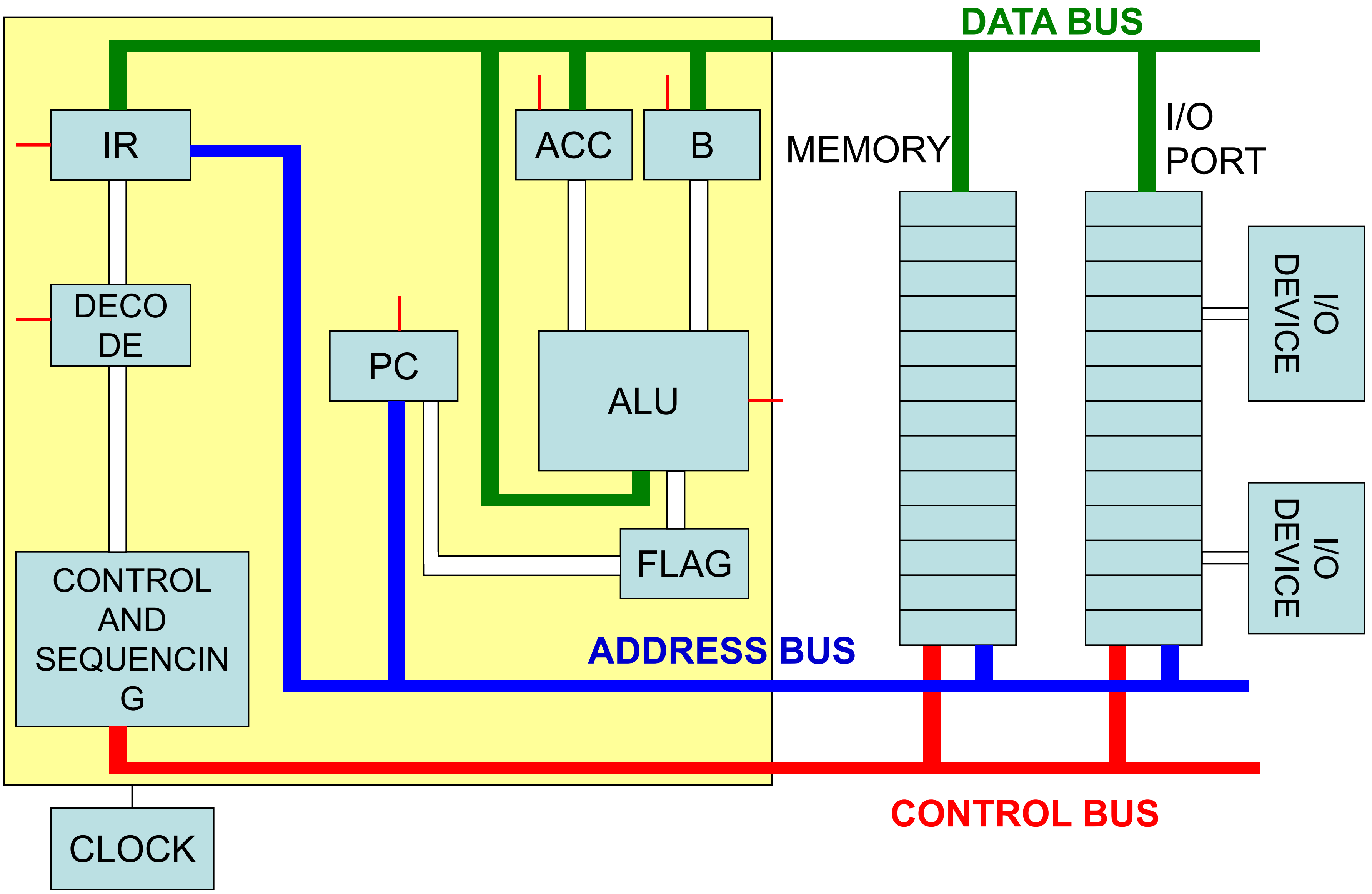
Instruction execution cycle



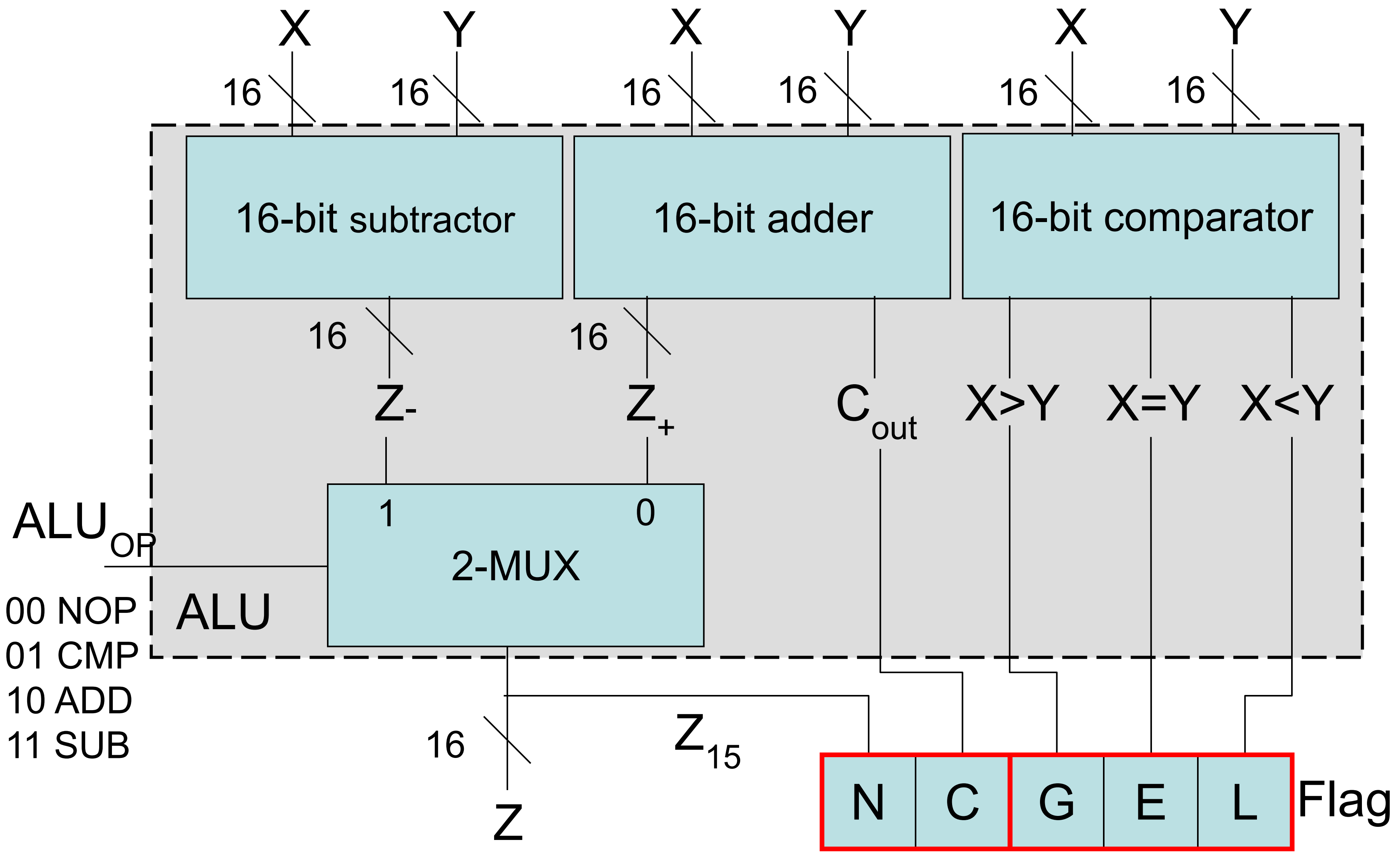
- **Fetch**
- **Decode**
- Fetch operands
- **Execute**
- Store output

A simple microcomputer



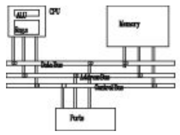


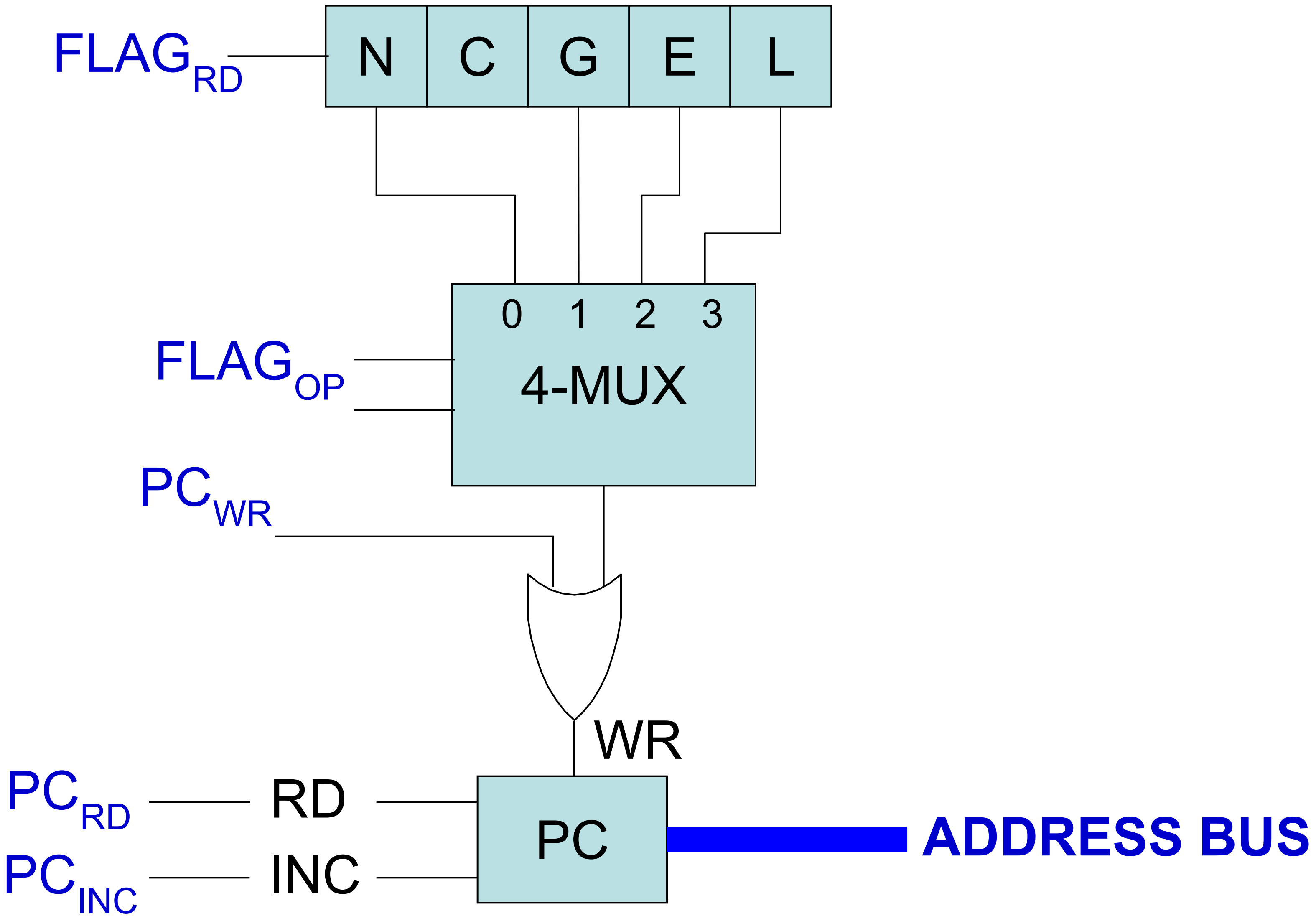
ALU and Flag



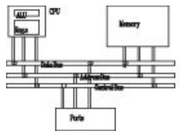
X
Y
X
Y
X
Y
16
16-bit subtractor
16-bit adder
16-bit comparator
Z-
Z_+
C_{out}
X>Y
X=Y
X<Y
1
0
2-MUX
ALU_{OP}
00 NOP
01 CMP
10 ADD
11 SUB
ALU
Z_{15}
Z
N
C
G
E
L
Flag

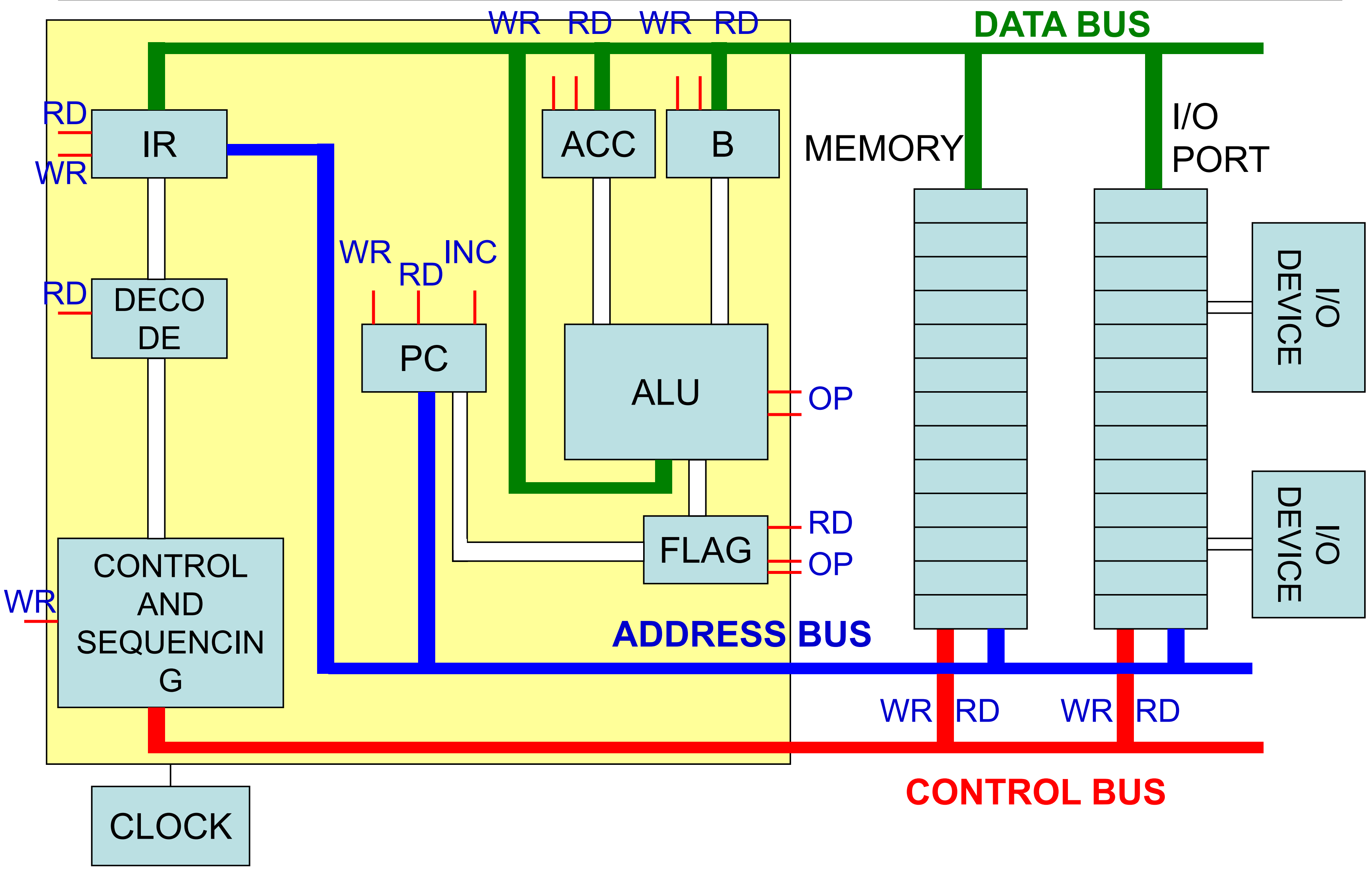
Flags



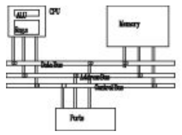

FLAG_RD
N
C
G
E
L
0
1
2
3
4-MUX
FLAG_OP
PC_WR
WR
PC_RD
RD
PC_INC
INC
PC
ADDRESS BUS

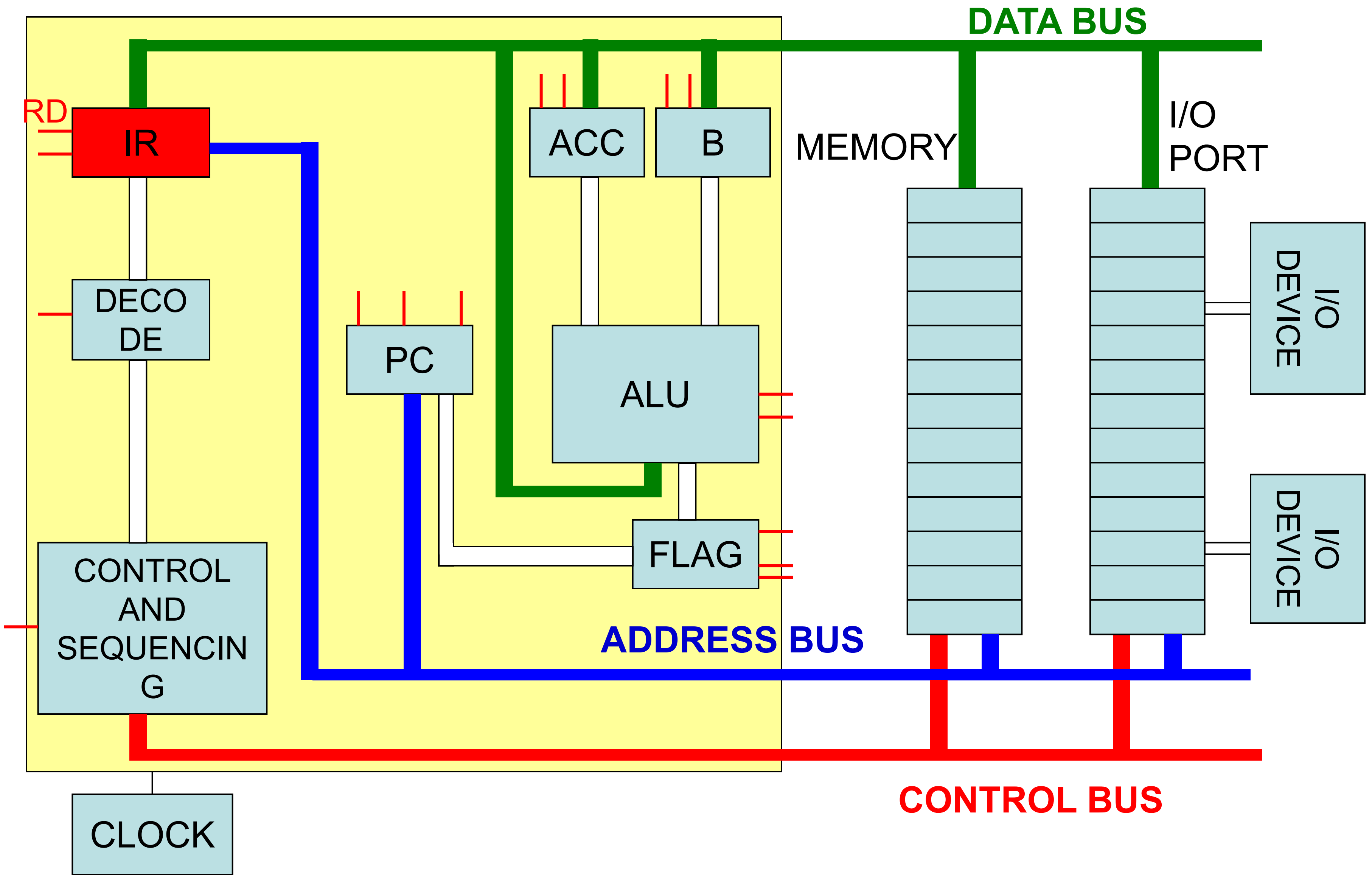
Control signals (20 in total)



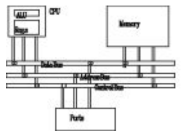


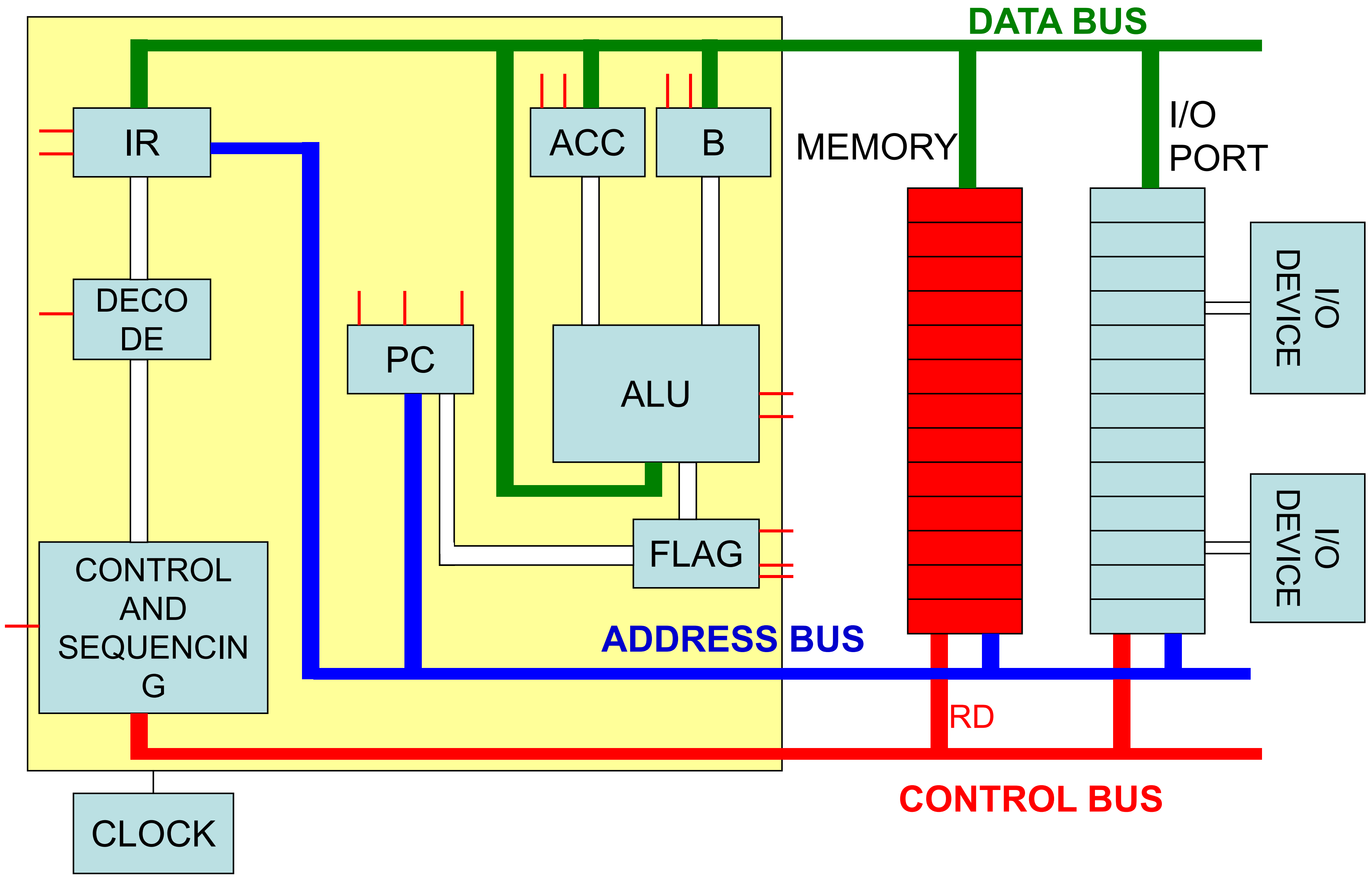
LDA (execution cycle 1): IR_{RD}



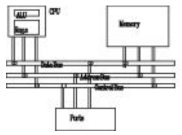


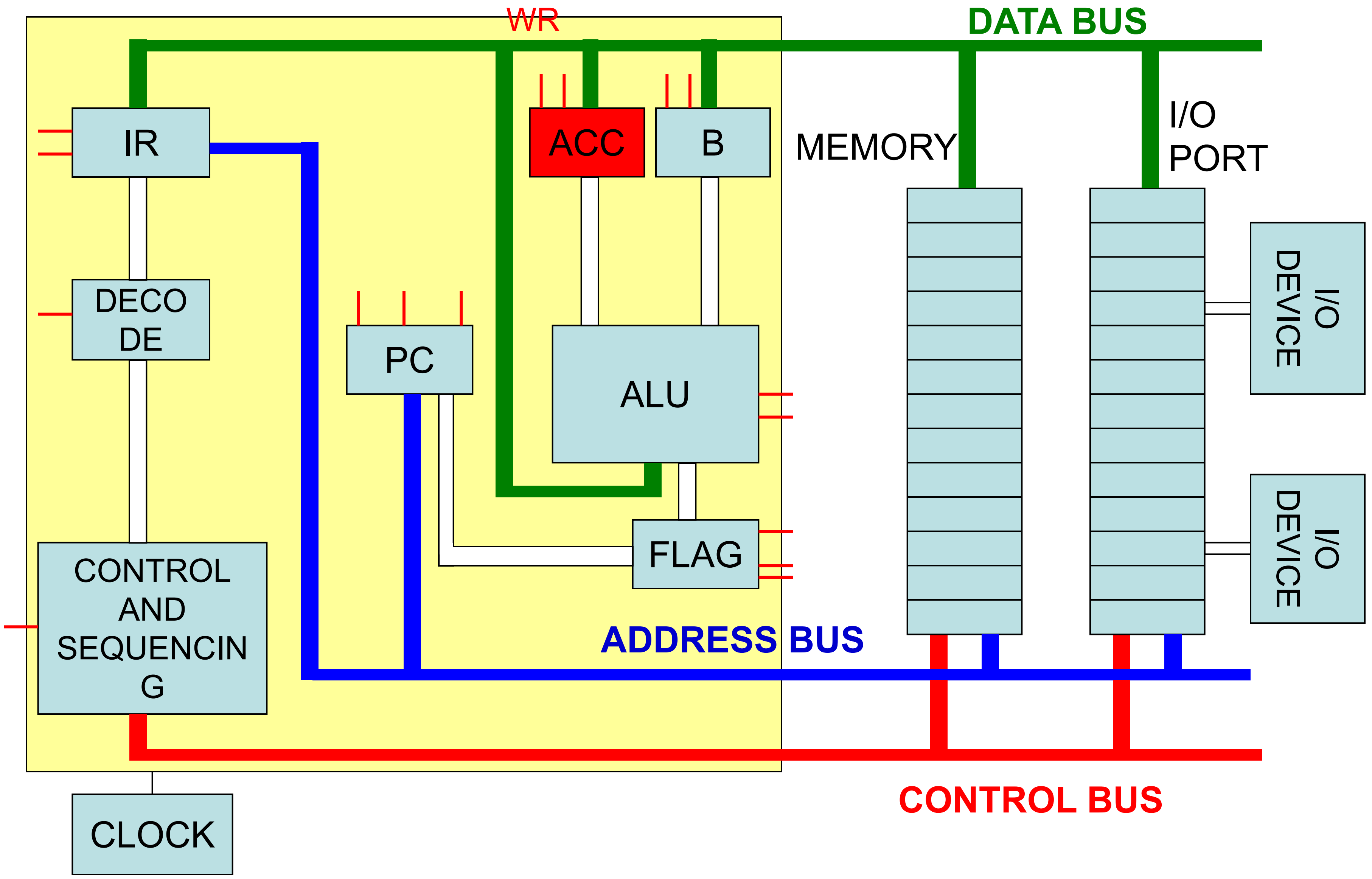
LDA (execution cycle 2): MEM_{RD}



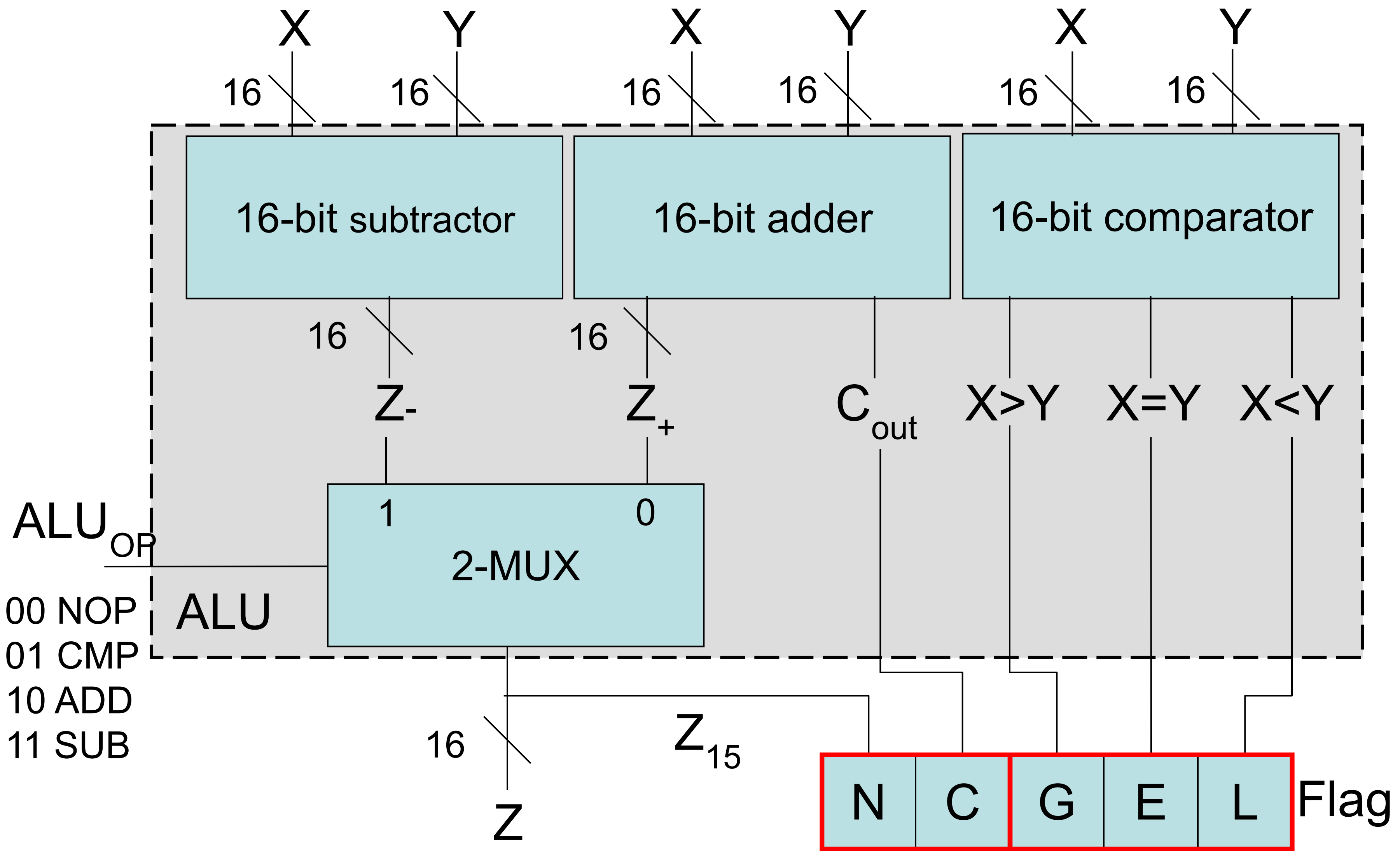


LDA (execution cycle 3): ACC_{WR}

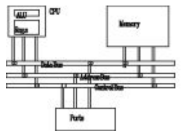


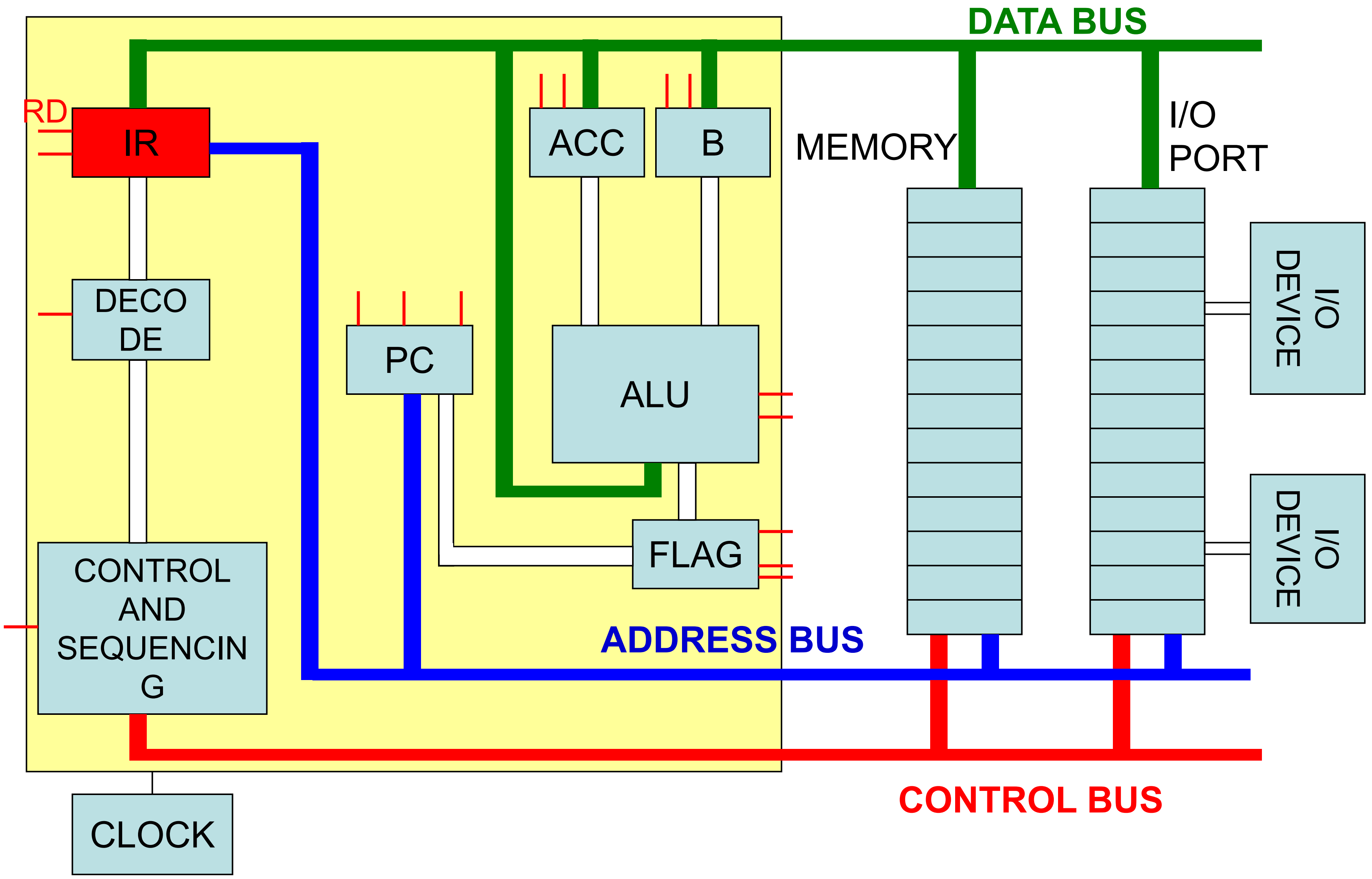


ALU and Flag

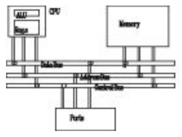

X
Y
16
16
X
Y
16
16
X
Y
16
16
16-bit subtractor
16-bit adder
16-bit comparator
16
16
Z-
Z+
Cout
X>Y
X=Y
X<Y
1
0
2-MUX
ALUOP
00 NOP
01 CMP
10 ADD
11 SUB
ALU
16
Z
Z15
N
C
G
E
L
Flag

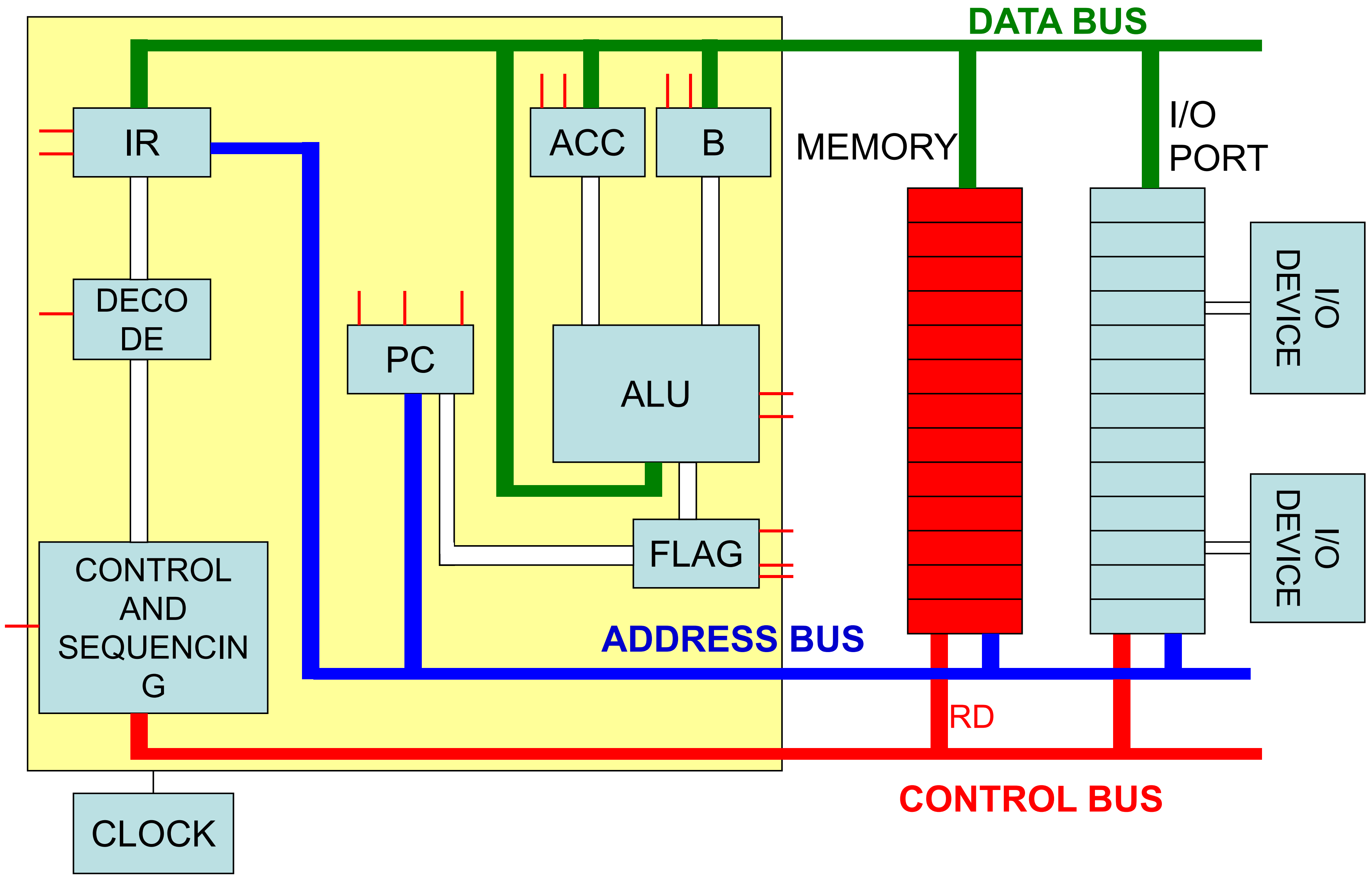
ADD (execution cycle 1): IR_{RD}



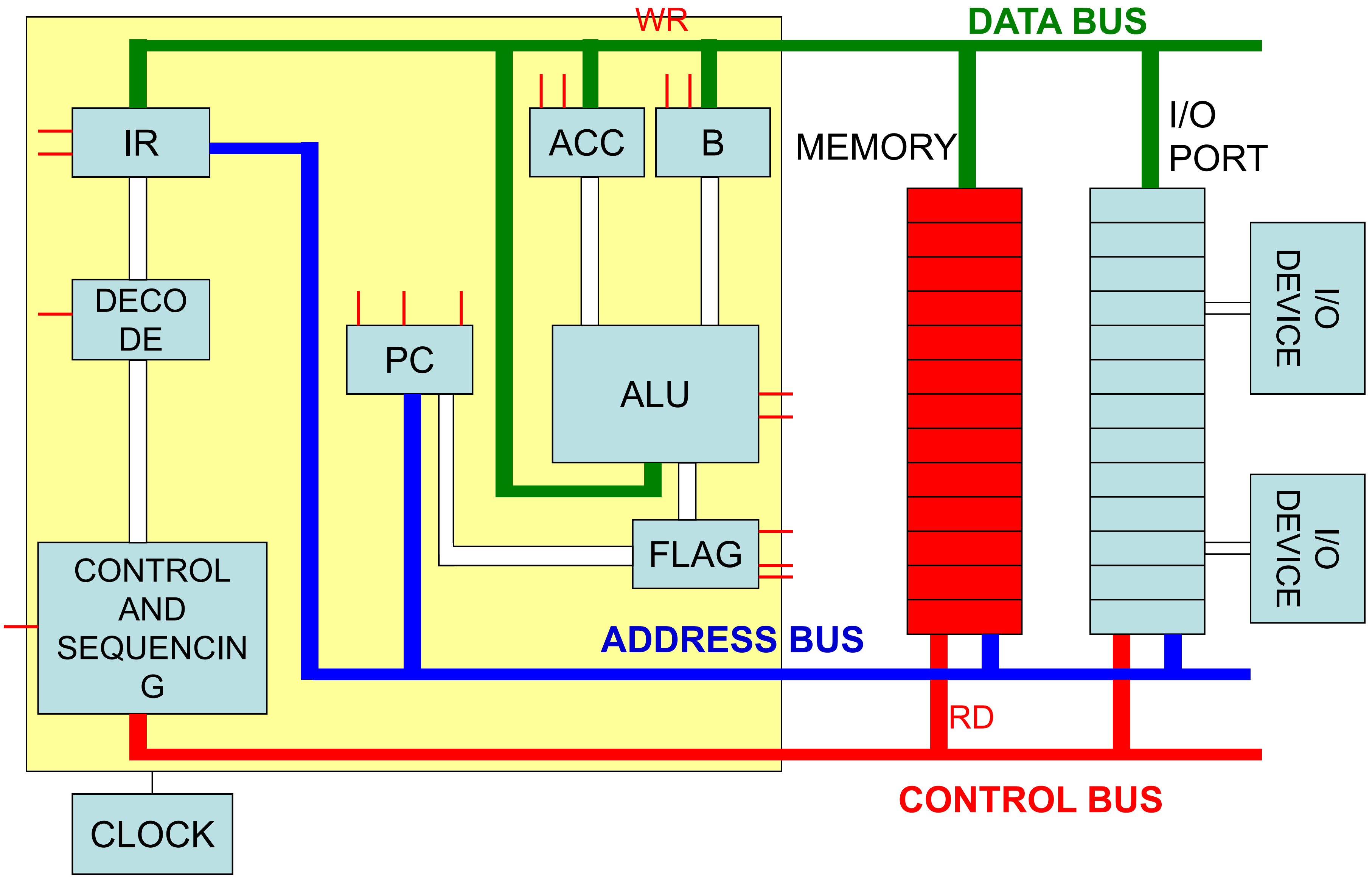


ADD (execution cycle 2): MEM_{RD}

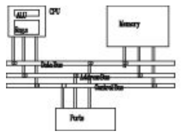


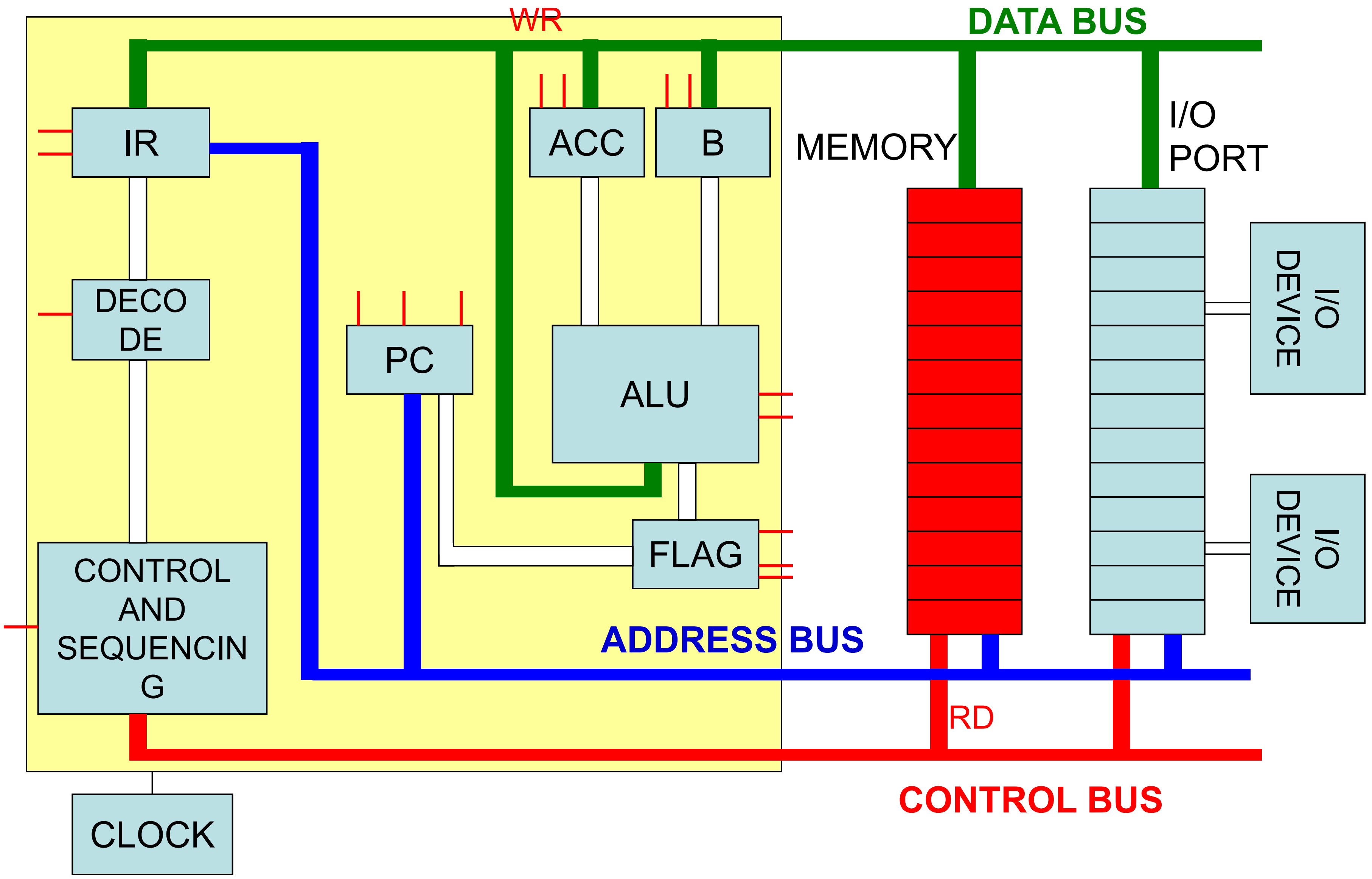


ADD (execution cycle 3): B_{WR}

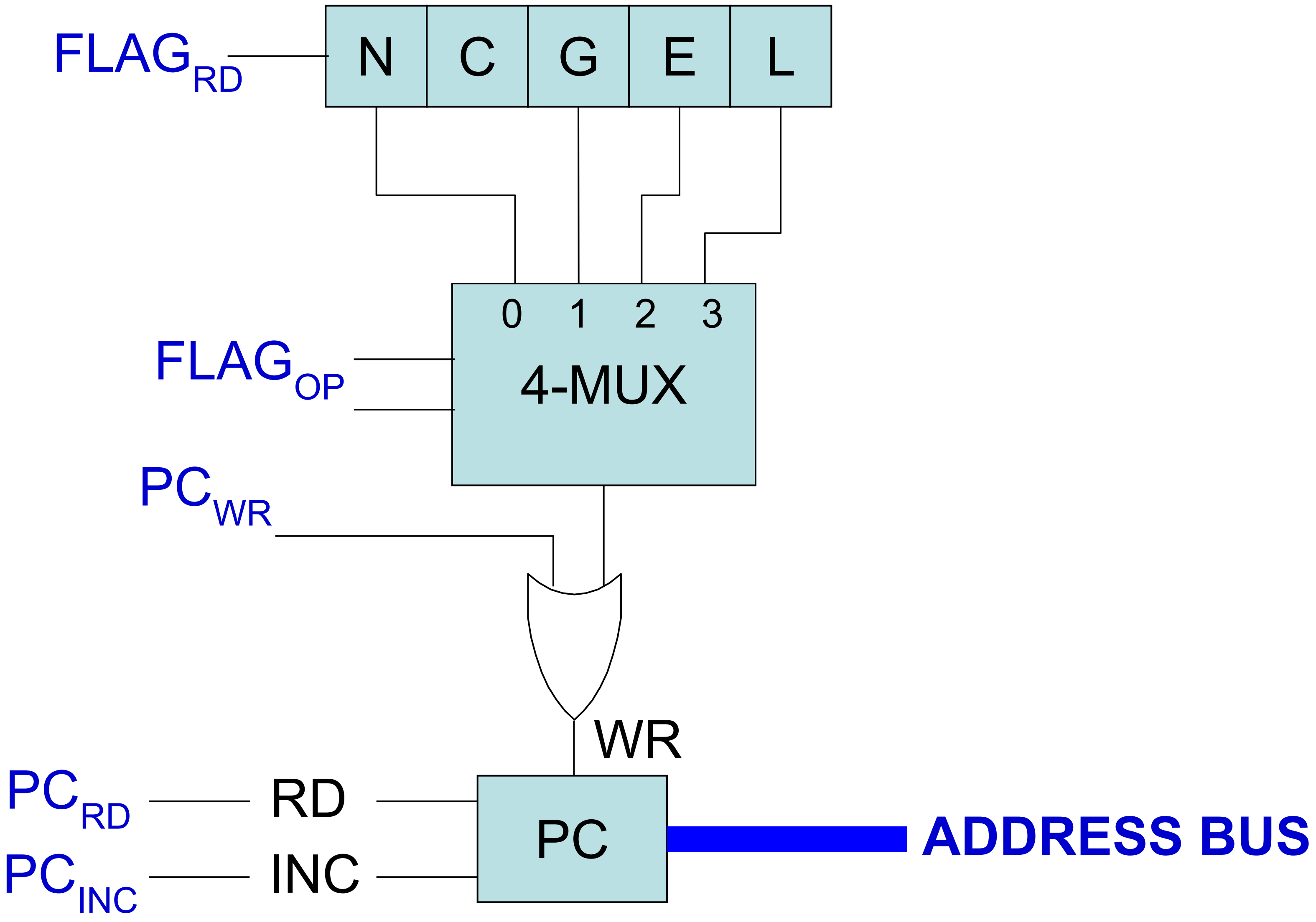


ADD (execution cycle 4): ALU_{10},ACC_{WR}

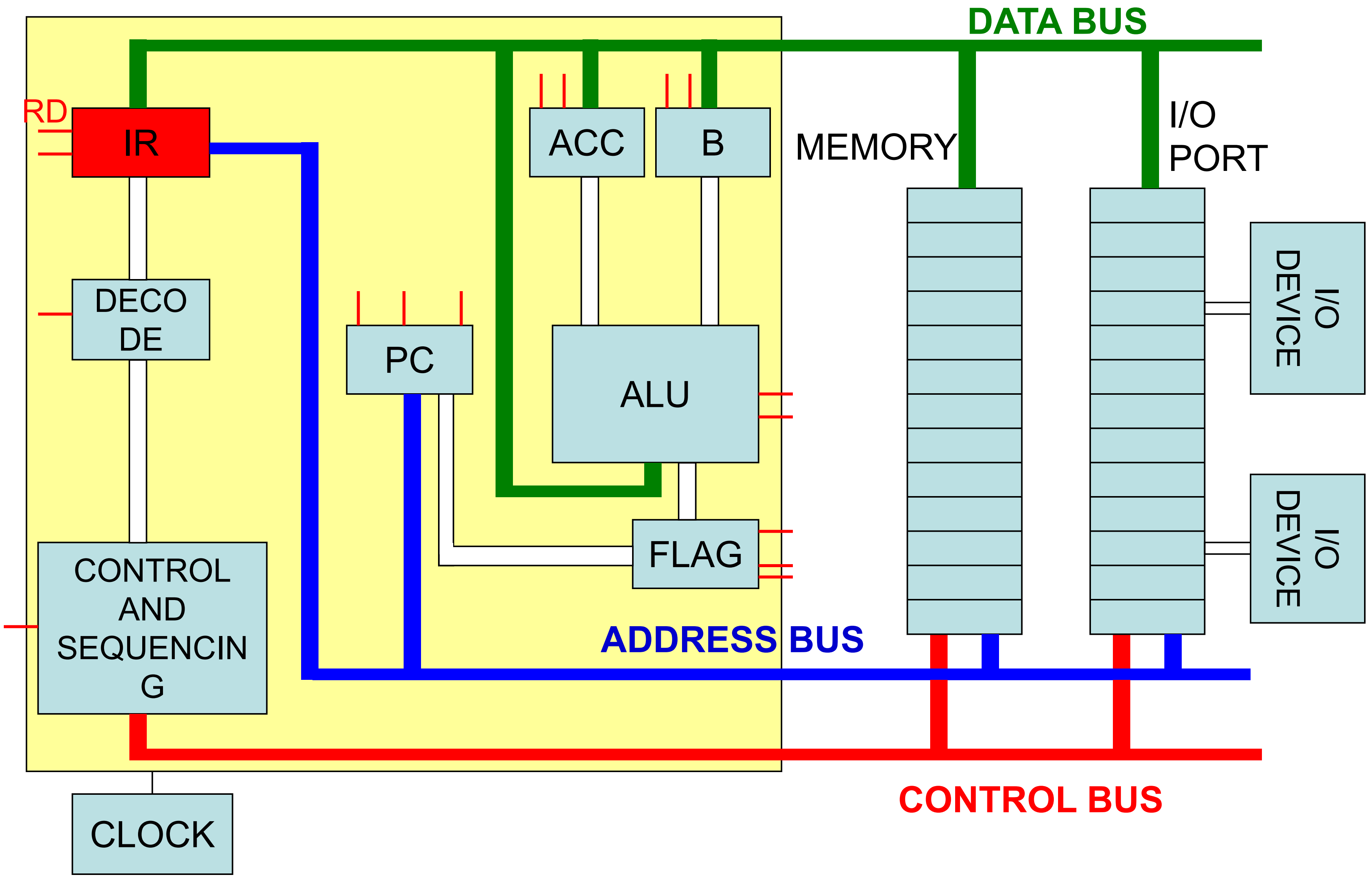




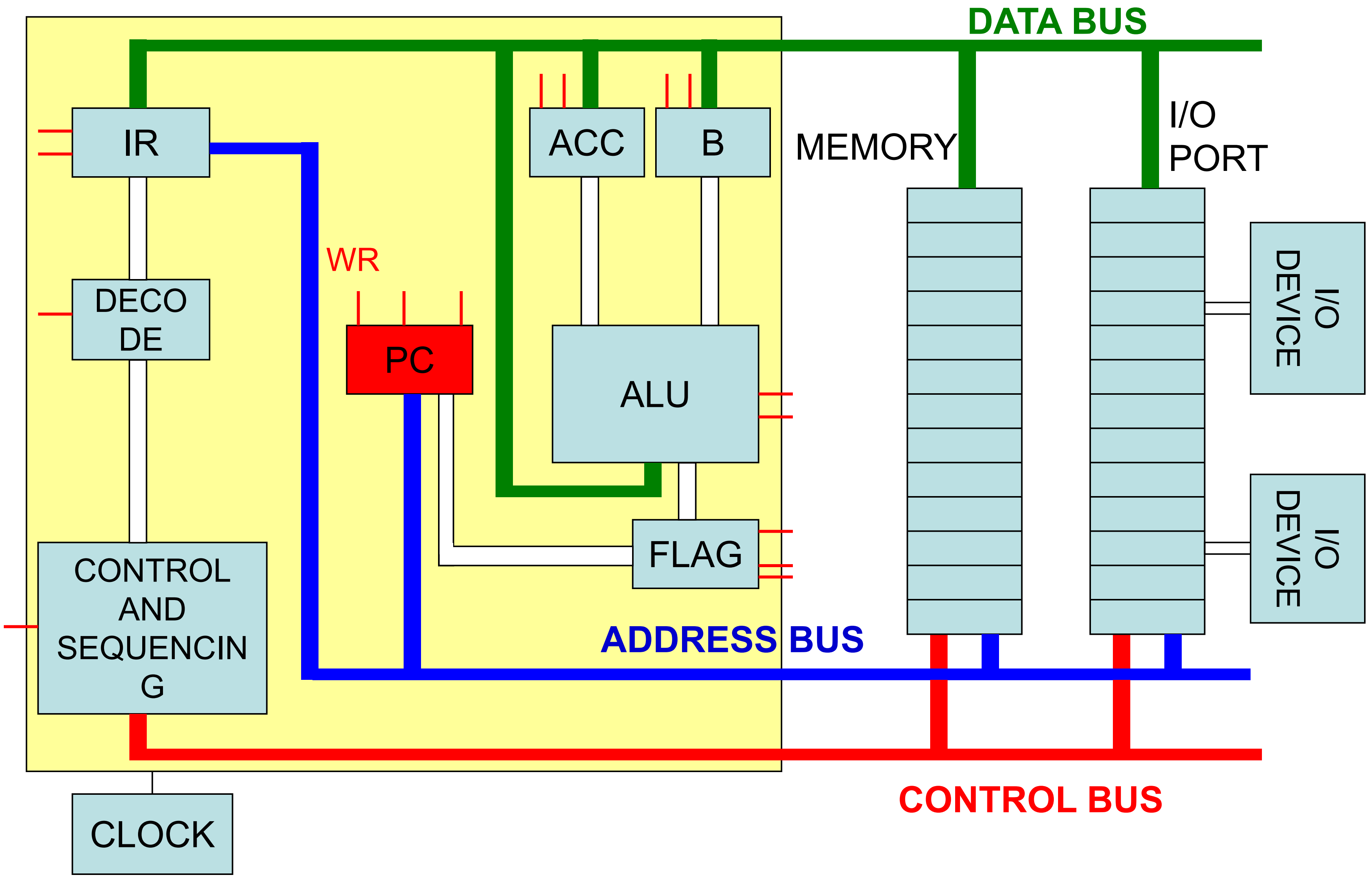
Flags


FLAG_RD
N
C
G
E
L
0
1
2
3
4-MUX
FLAG_OP
PC_WR
WR
PC_RD
RD
PC_INC
INC
PC
ADDRESS BUS

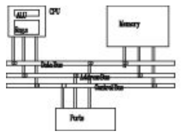
JMP (execution cycle 1): IR_{RD}

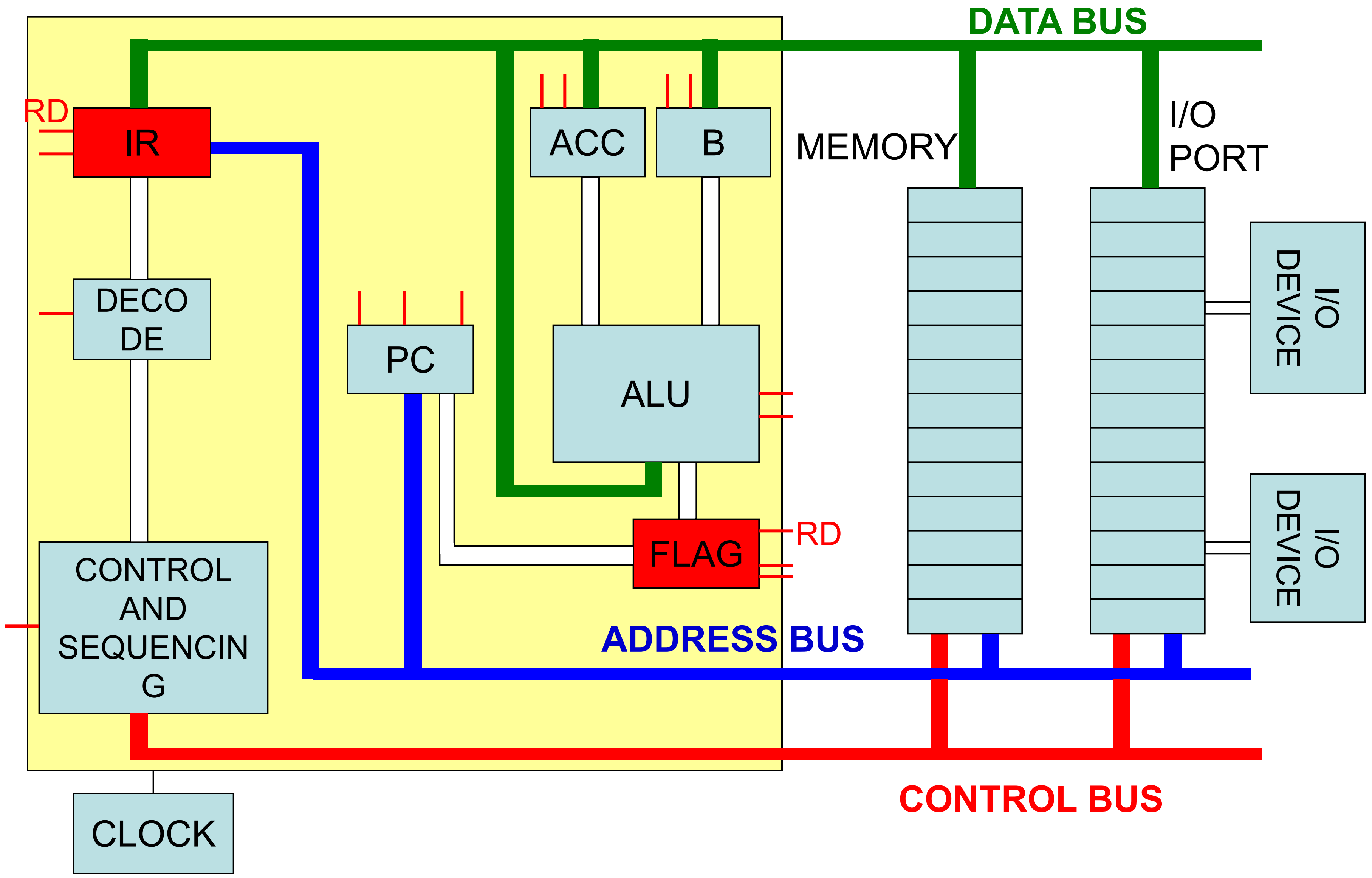


JMP (execution cycle 2): PC_{WR}

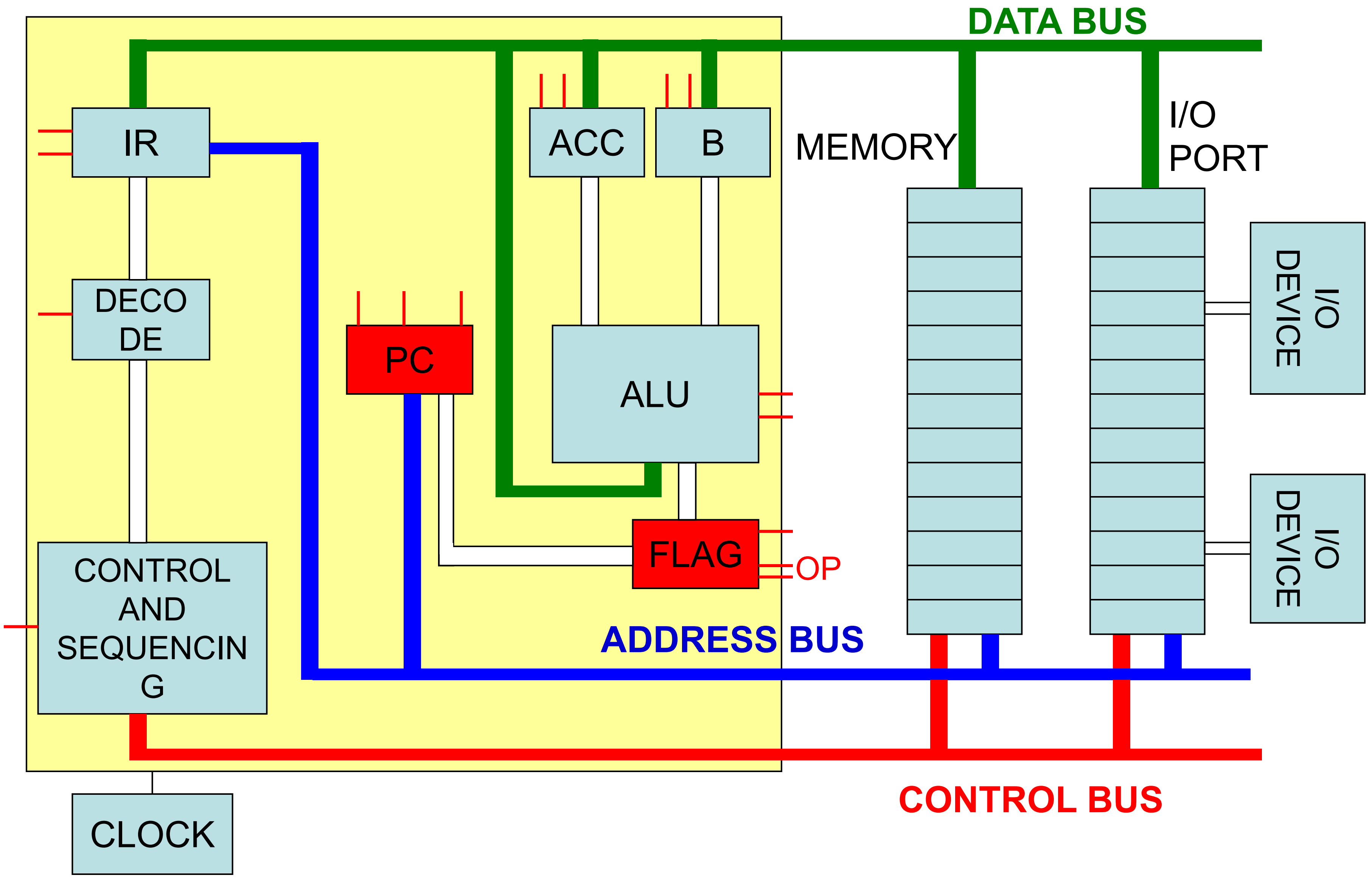


JG (execution cycle 1): IR_{RD},$FLAG_{RD}$

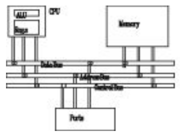




JG (execution cycle 2): $FLAG_{01}$



Microcode sequence



`LDA 510` PC_{RD}

MEM_{RD}

IR_{WR} PC_{INC}

IR_{RD}

$DECODER_{RD}$

μPC_{WR}

IR_{RD}

MEM_{RD}

ACC_{WR}

`JMP 10` PC_{RD}

MEM_{RD}

IR_{WR} PC_{INC}

IR_{RD}

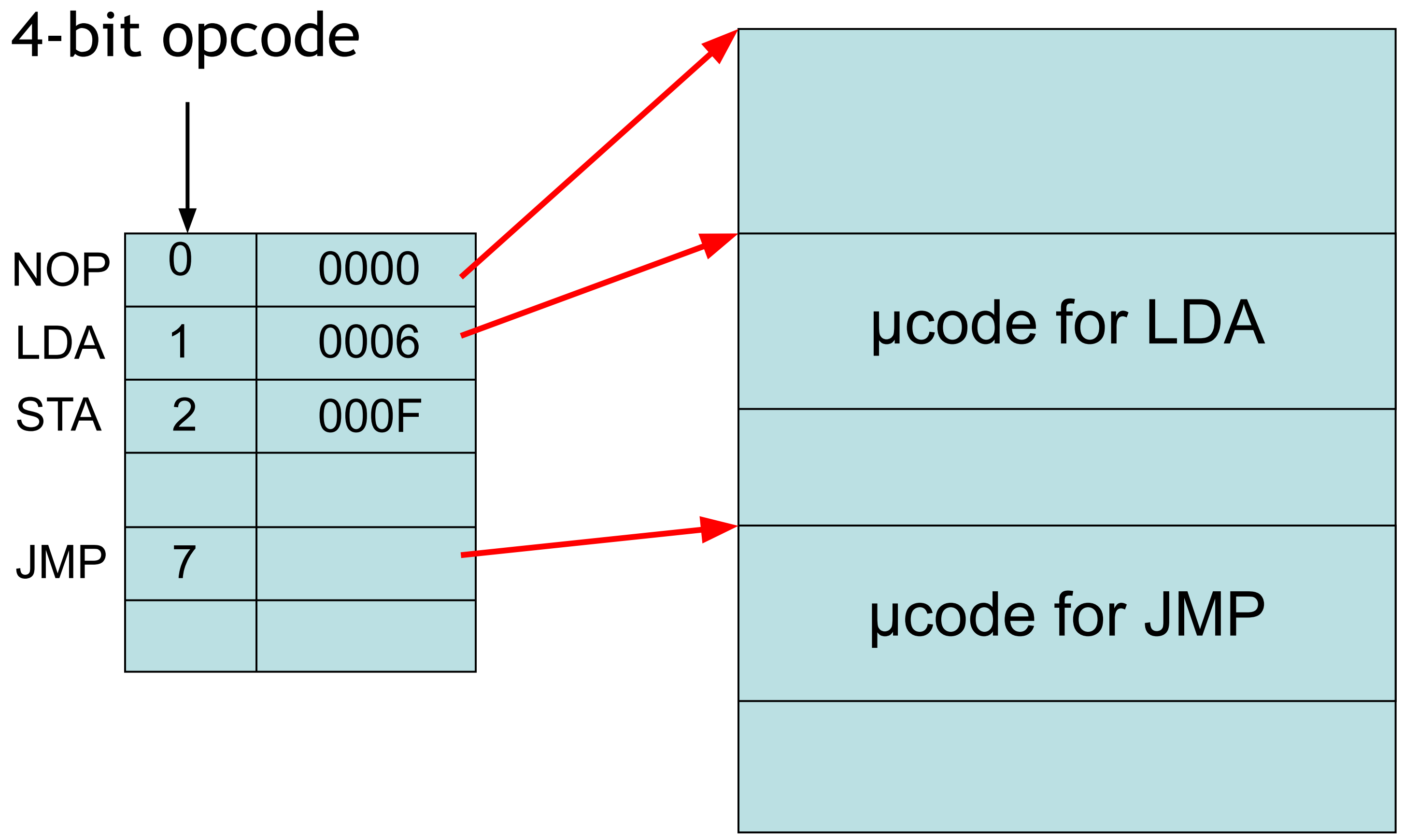
$DECODER_{RD}$

μPC_{WR}

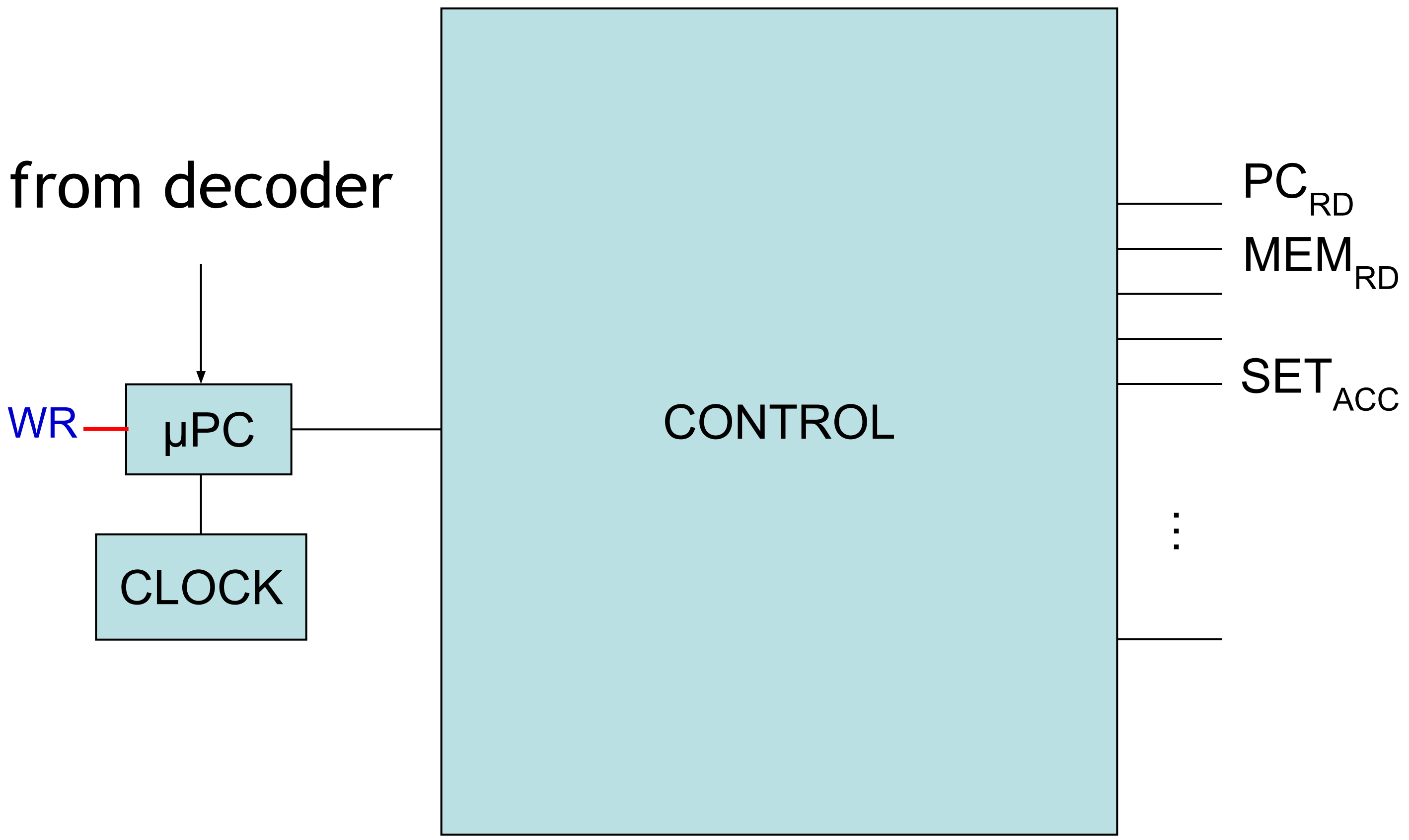
IR_{RD}

PC_{WR}

Decoder


4-bit opcode
NOP
0
0000
LDA
1
0006
STA
2
000F
JMP
7
µcode for LDA
µcode for JMP

Control and sequencing unit

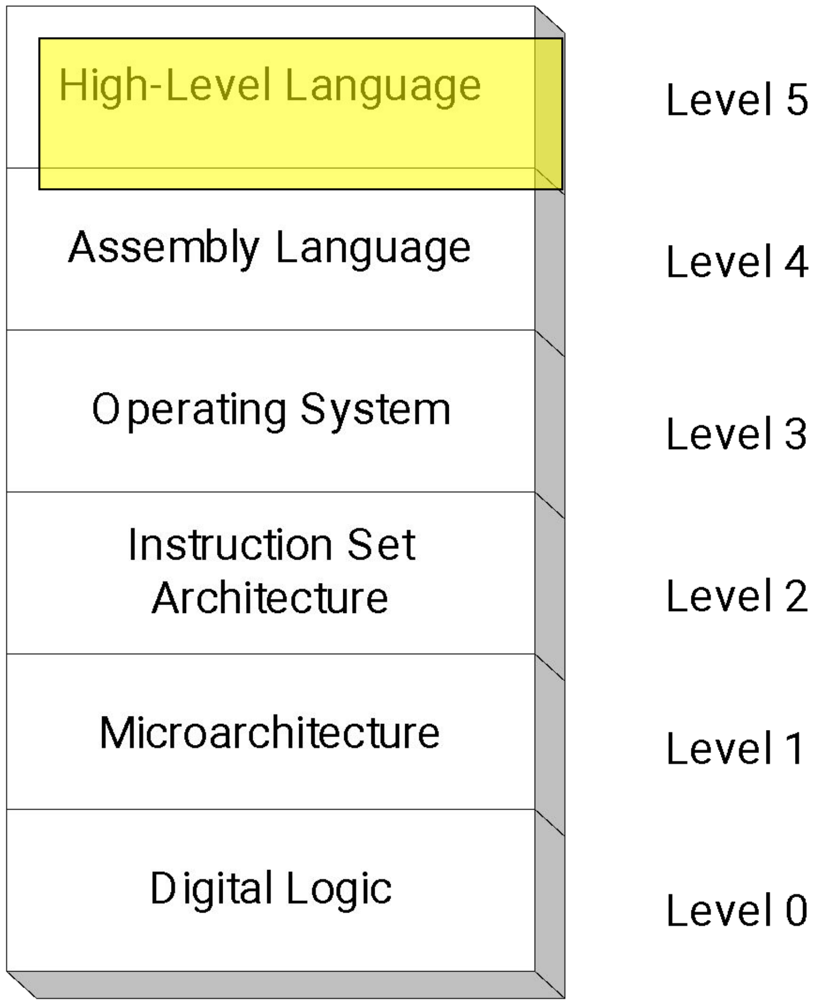


Control and sequencing unit

		PC_{RD}	MEM_{RD}	MEM_{WR}	IR_{WR}	PC_{INC}	….
NOP fetch	0000	1	0	0	0	0	0….
	0001	0	1	0	0	0	
	0002	0	0	0	1	1	
decode	0003	IR_{RD}					
	0004	$DECODER_{RD}$					
	0005	μPC_{WR}					
LDA fetch decode							
exec	0006	IR_{RD}					
	0007	MEM_{RD}					
	0008	ACC_{WR}					
	000F						

Virtual machines

Abstractions for computers



X=min of X,Y,Z

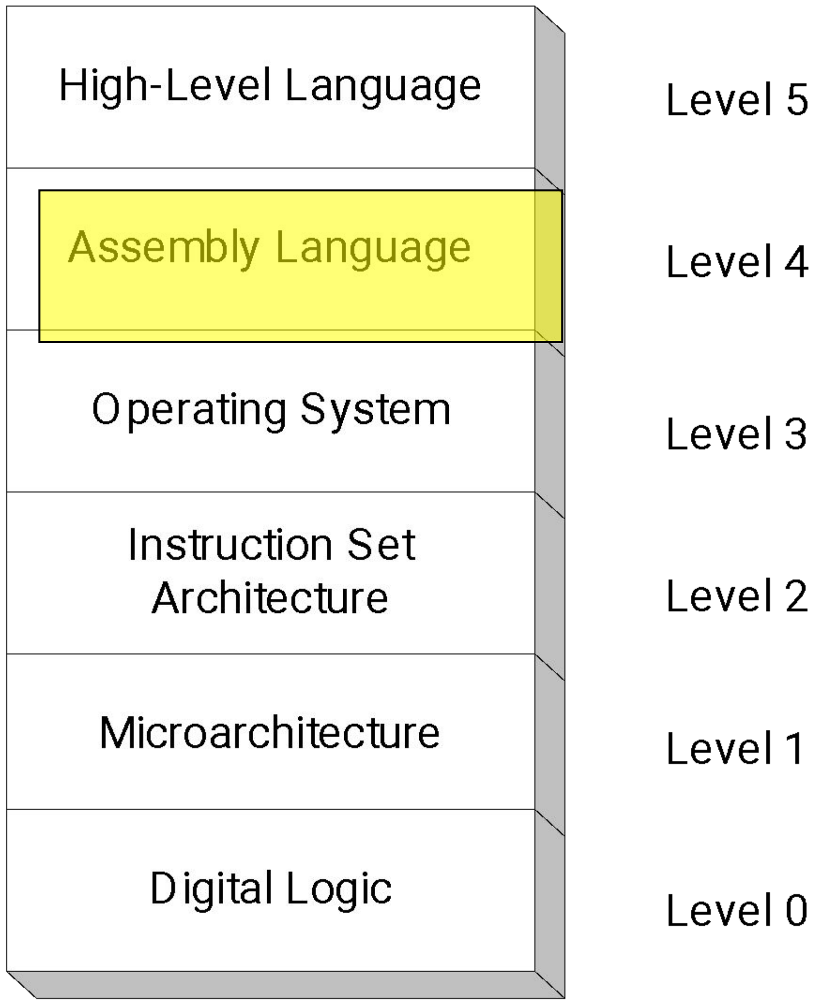
```
int X=7; Y=2; Z=9;
if (X>Y) then
   if (Y>Z) then
      X=Z;
   else
      X=Y;
   end
else
   if (X<Z) then
      X=Z;
   end             else?
end
```

compiler

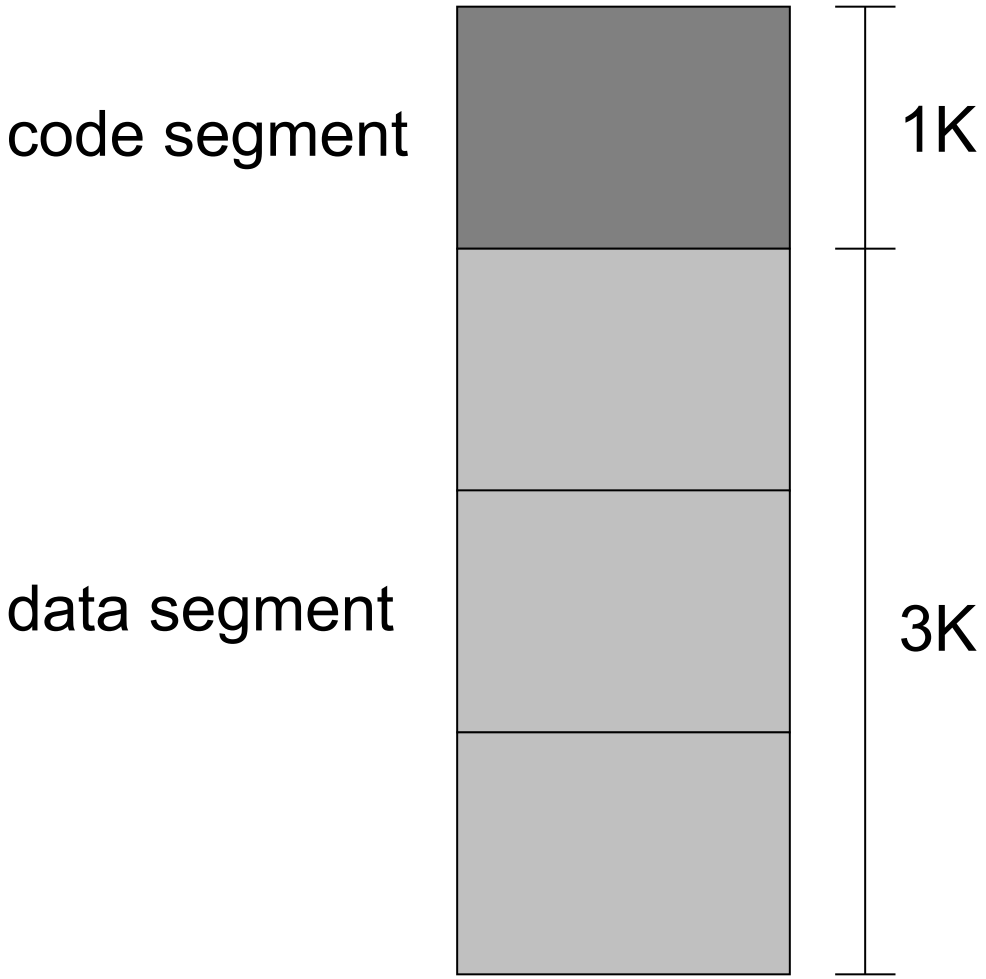
```
.DATA
X   007
Y   002
Z   009
.CODE
    LDA   X
    CMPY
    JG L1
    CMPZ
    JL L0
    JMP   END
L0  LDAZ
    STAX
L1  LDAY
    CMPZ
    JG L2
    STAX
    JMPEND
L2  LDAZ
    STAX
ENDHLT
```

Virtual machines

Abstractions for computers



Memory layout



X=min of X,Y,Z

```
.DATA
X  007
Y  002
Z  009
.CODE
   LDA   X
   CMP   Y
   JL    L1
   LDA   Y
L1 CMP   Z
   JL L2
   LDA   Z
L2 STA   X
   HLT
```

```
.DATA
X  007
Y  002
Z  009
.CODE
   LDA   Y
   CMP   Z
   JL    L1
   LDA   Z
L1 CMP   X
   JG    END
   STA   X
END  HLT
```

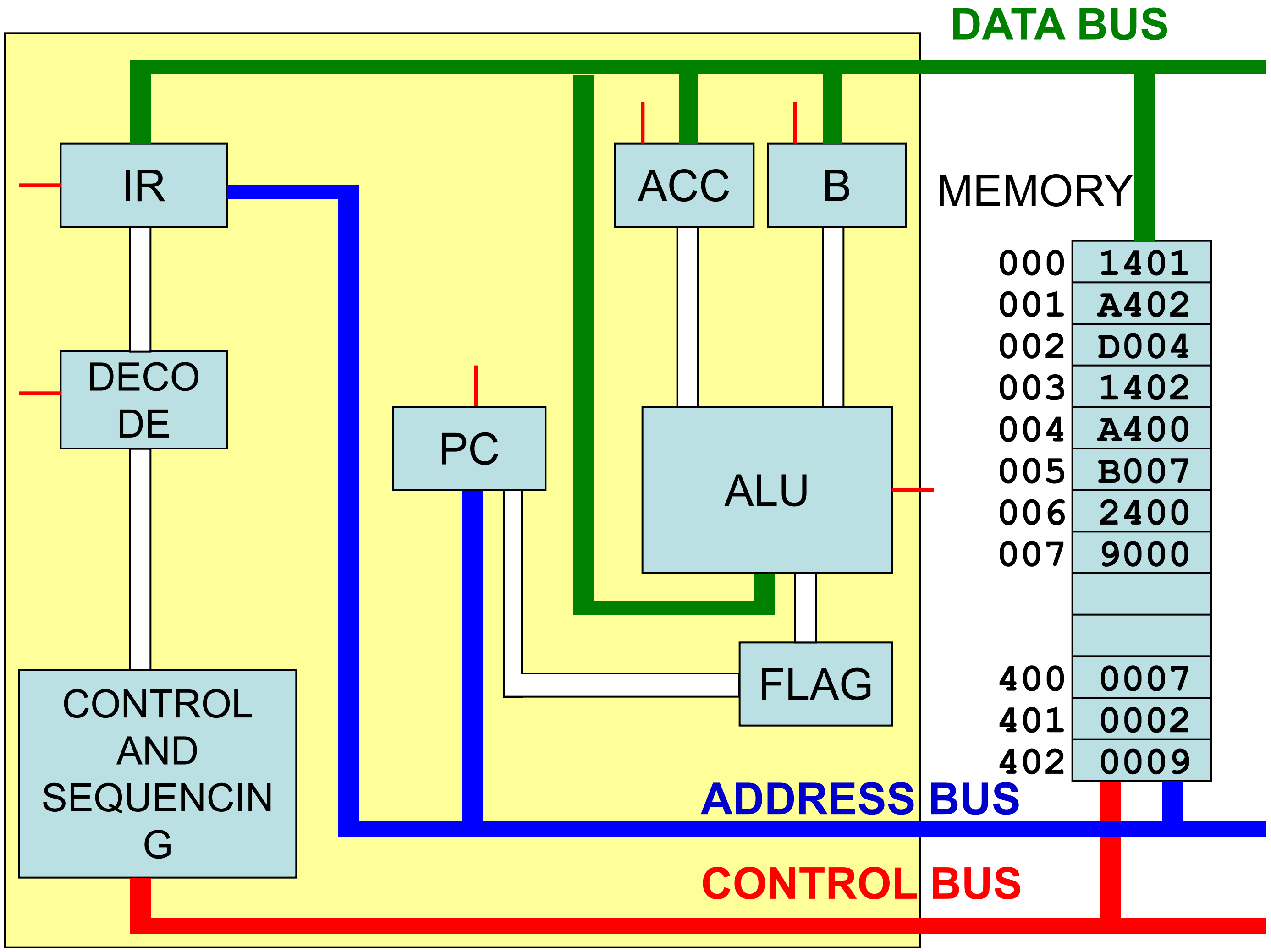
X=min of X,Y,Z

```
 .DATA
 X  007
 Y  002
 Z  009
 .CODE
0      LDA    Y
1      CMP    Z
2      JL     L1
3      LDA    Z
4 L1   CMP    X
5      JG     END
6      STA    X
7 END  HLT
```

```
1401
A402
D004
1402
A400
B007
2400
9000
```

X	400
Y	401
Z	402

L1	4
END	7



```
LDA     401
CMP     402
JL  004
LDA     402
CMP     400
JG  007
STA     400
HLT
```

Advanced architecture

Multi-stage pipeline

- Pipelining makes it possible for processor to execute instructions in parallel
- Instruction execution divided into discrete stages

Example of a non-pipelined processor. For example, 80386. Many wasted cycles.

Stages

Cycles

	S1	S2	S3	S4	S5	S6
1	I-1					
2		I-1				
3			I-1			
4				I-1		
5					I-1	
6						I-1
7	I-2					
8		I-2				
9			I-2			
10				I-2		
11					I-2	
12						I-2

Pipelined execution

- More efficient use of cycles, greater throughput of instructions: (80486 started to use pipelining)

Stages

Cycles	S1	S2	S3	S4	S5	S6
1	I-1					
2	I-2	I-1				
3		I-2	I-1			
4			I-2	I-1		
5				I-2	I-1	
6					I-2	I-1
7						I-2

For *k* stages and *n* instructions, the number of required cycles is:

$$k + (n - 1)$$

compared to k*n

Wasted cycles (pipelined)

- When one of the stages requires two or more clock cycles, clock cycles are again wasted.

Stages

				exe		
Cycles	S1	S2	S3	S4	S5	S6
1	I-1					
2	I-2	I-1				
3	I-3	I-2	I-1			
4		I-3	I-2	I-1		
5			I-3	I-1		
6				I-2	I-1	
7				I-2	▨	I-1
8				I-3	I-2	▨
9				I-3	▨	I-2
10					I-3	▨
11						I-3

For *k* stages and *n* instructions, the number of required cycles is:

$k + (2n - 1)$

Superscalar

A superscalar processor has multiple execution pipelines. In the following, note that Stage S4 has left and right pipelines (u and v).

Stages

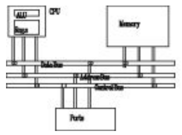
				S4			
Cycles	S1	S2	S3	u	v	S5	S6
1	I-1						
2	I-2	I-1					
3	I-3	I-2	I-1				
4	I-4	I-3	I-2	I-1			
5		I-4	I-3	I-1	I-2		
6			I-4	I-3	I-2	I-1	
7				I-3	I-4	I-2	I-1
8					I-4	I-3	I-2
9						I-4	I-3
10							I-4

For *k* states and *n* instructions, the number of required cycles is:

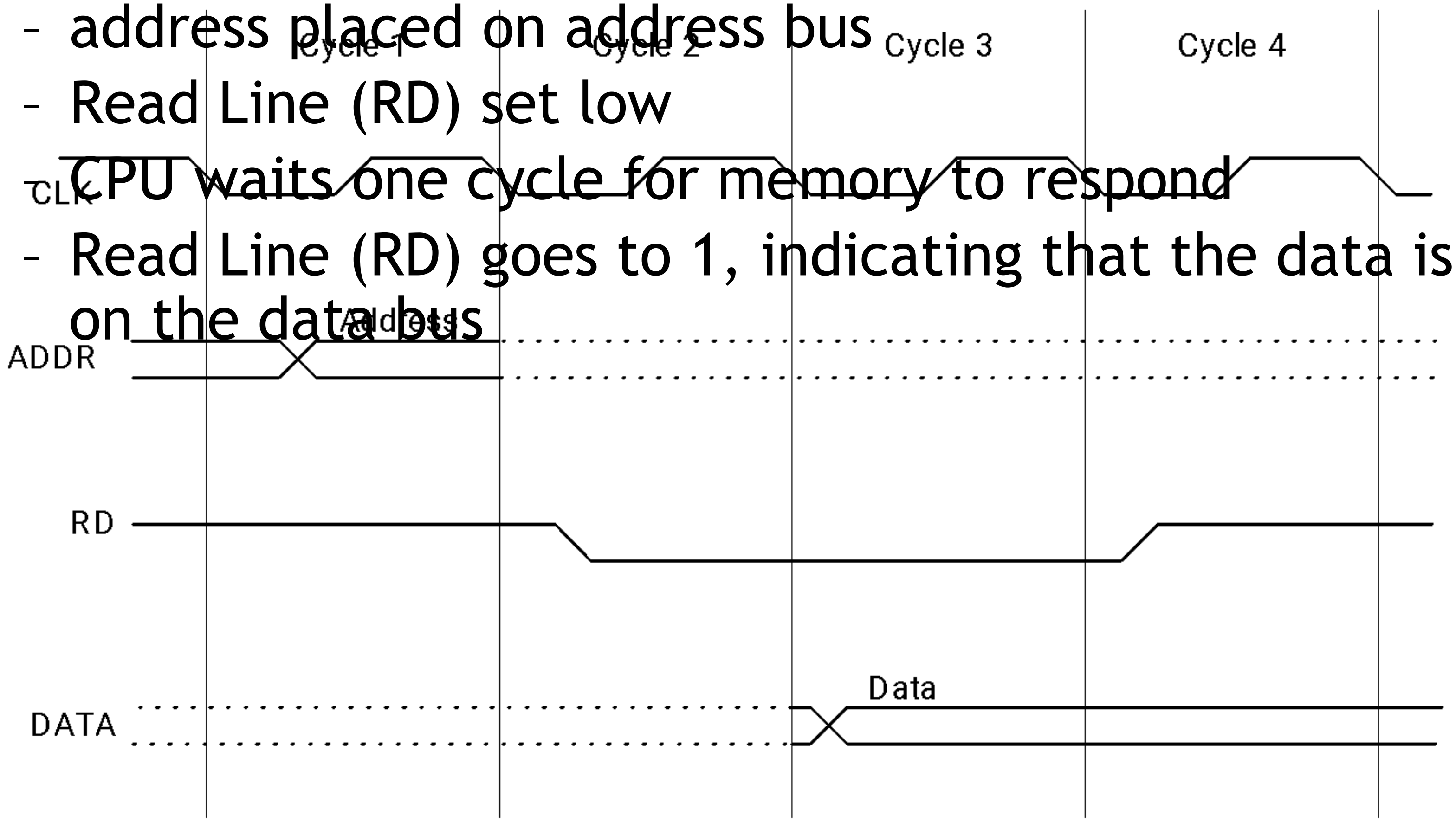
$k + n$

Pentium: 2 pipelines
Pentium Pro: 3

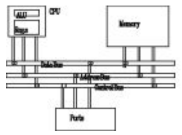
Reading from memory



- Multiple machine cycles are required when reading from memory, because it responds much more slowly than the CPU. The four steps are:
  - address placed on address bus
  - Read Line (RD) set low
  - CPU waits one cycle for memory to respond
  - Read Line (RD) goes to 1, indicating that the data is on the data bus



Cache memory



- High-speed expensive static RAM both inside and outside the CPU.
  - Level-1 cache: inside the CPU
  - Level-2 cache: outside the CPU

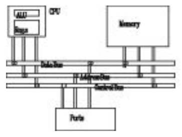
Because conventional memory is so much slower than the CPU, computers use high-speed cache memory to hold the most recently used instructions and data. The first time a program reads a block of data, it leaves a copy in the cache. If the program needs to read the same data a second time, it looks for the data in cache.

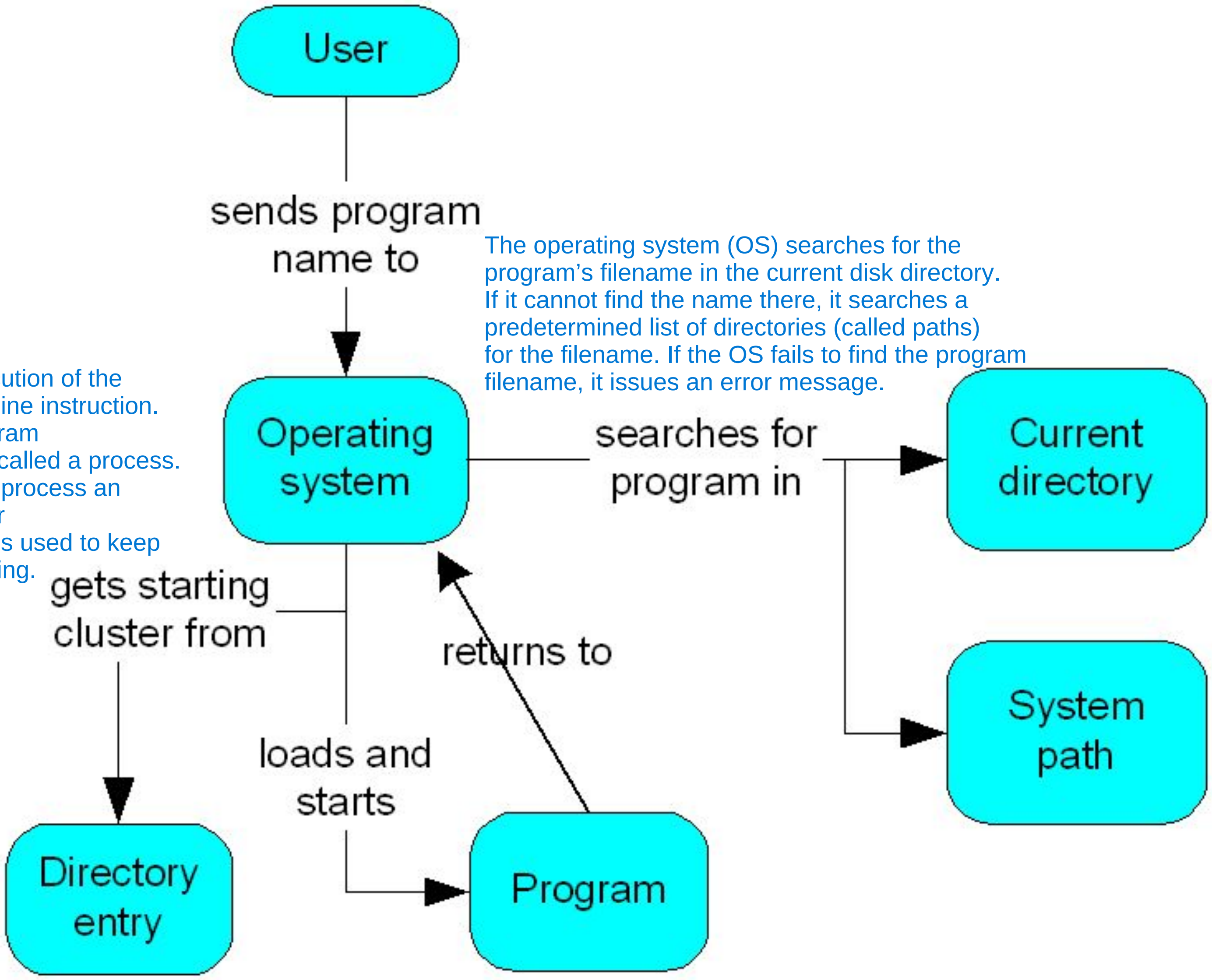
- Cache hit: when data to be read is already in cache memory

A cache hit indicates the data is in cache; a cache miss indicates the data is not in cache and must be read from conventional memory. In general, cache memory has a noticeable effect on improving access to data, particularly when the cache is large.

- Cache miss: when data to be read is not in cache memory. When? compulsory, capacity and conflict.
- Cache design: cache size, n-way, block size, replacement policy

How a program runs

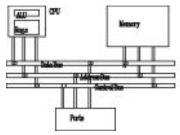




The operating system (OS) searches for the program's filename in the current disk directory. If it cannot find the name there, it searches a predetermined list of directories (called paths) for the filename. If the OS fails to find the program filename, it issues an error message.

The OS begins execution of the program's first machine instruction. As soon as the program begins running, it is called a process. The OS assigns the process an identification number (process ID), which is used to keep track of it while running.

Multitasking



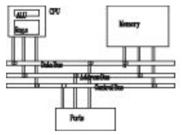
- OS can run multiple programs at the same time.
- Multiple threads of execution within the same program.
- Scheduler utility assigns a given amount of CPU time to each running program.
- Rapid switching of tasks
  - gives illusion that all programs are running at once
  - the processor must support task switching
  - scheduling policy, round-robin, priority

IA-32 Architecture

IA-32 architecture

- From 386 to the latest 32-bit processor, P4
- From programmer's point of view, IA-32 has not changed substantially except the introduction of a set of high-performance instructions

Modes of operation



- Protected mode
  - native mode (Windows, Linux), full features, separate memory

Protected mode is the native state of the processor, in which all instructions and features are available. Programs are given separate memory areas named segments, and the processor prevents programs from referencing memory outside their assigned segments.

- Virtual-8086 mode
  - hybrid of Protected
  - each program has its own 8086 computer

if an MS-DOS program crashes or attempts to write data into the system memory area, it will not affect other programs running at the same time.

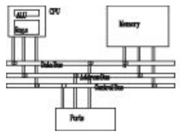
- Real-address mode
  - native MS-DOS

Real-address mode implements the programming environment of the Intel 8086 processor with a few extra features, such as the ability to switch into other modes.

- System management mode
  - power management, system security, diagnostics

System Management mode (SMM) provides an operating system with a mechanism for implementing functions such as power management and system security. These functions are usually implemented by computer manufacturers who customize the processor for a particular system setup

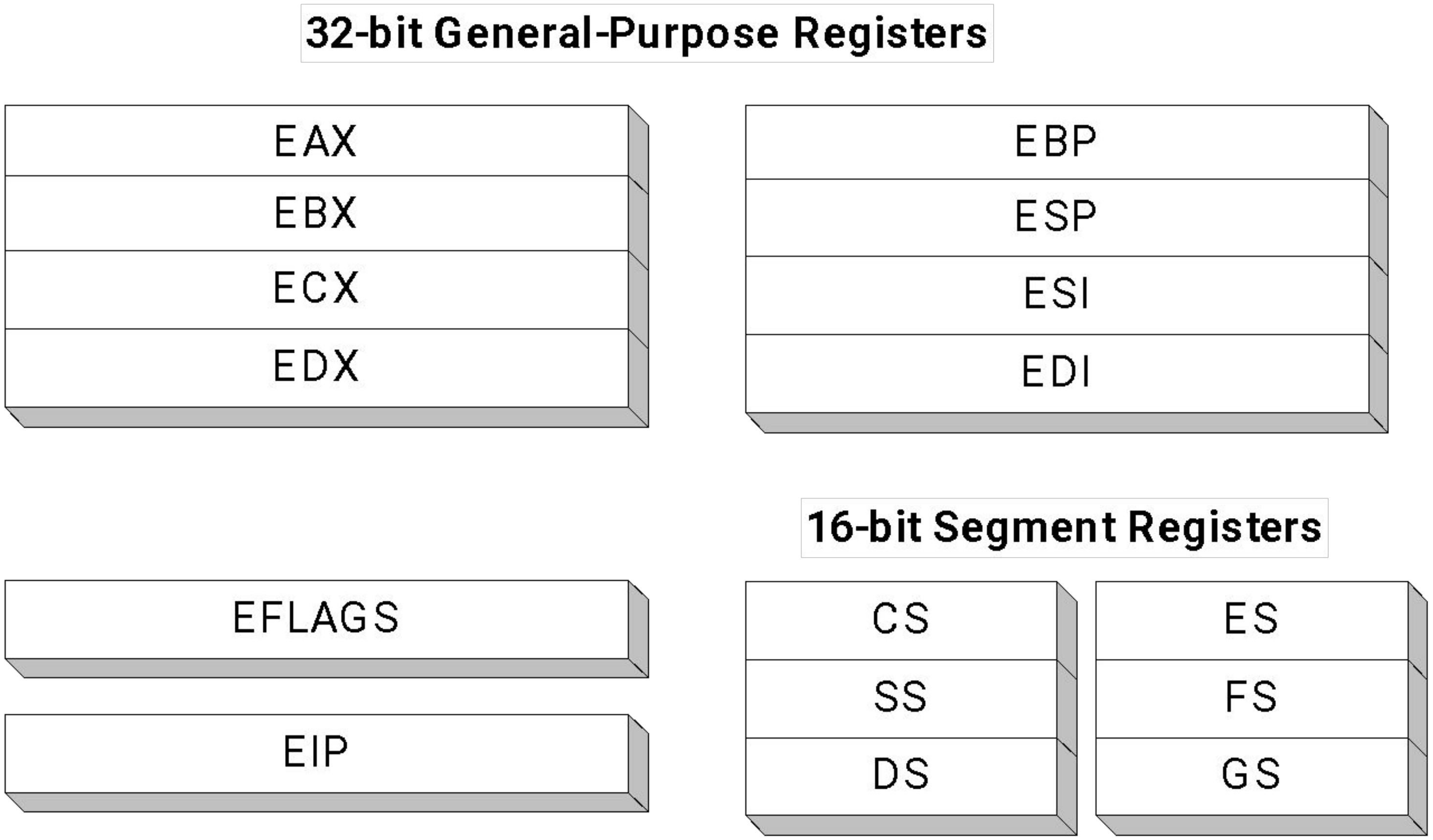
Addressable memory



- Protected mode
  - 4 GB
  - 32-bit address
- Real-address and Virtual-8086 modes
  - 1 MB space
  - 20-bit address

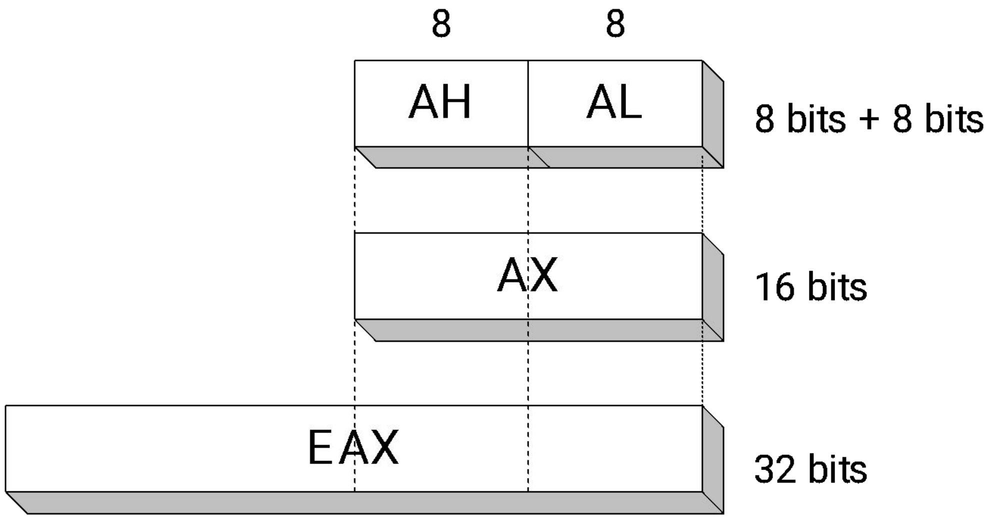
General-purpose registers

Named storage locations inside the CPU, optimized for speed.



Accessing parts of registers

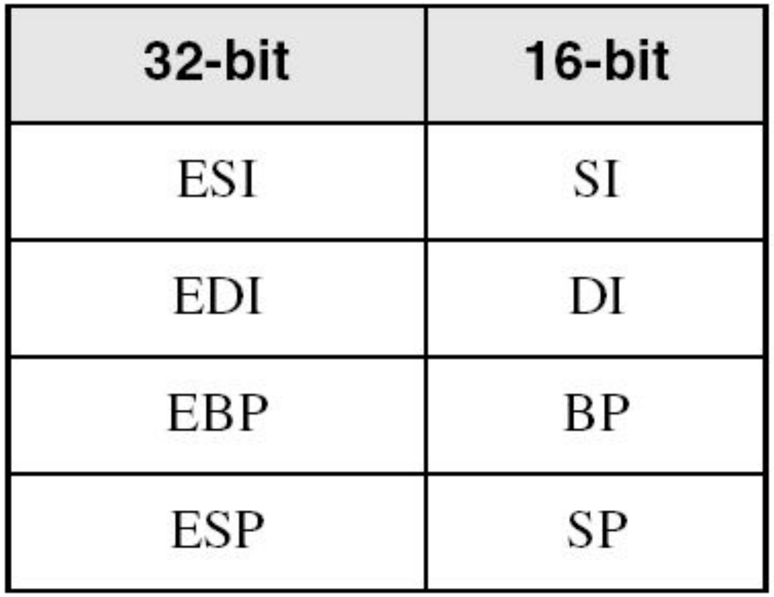
- Use 8-bit name, 16-bit name, or 32-bit name
- Applies to EAX, EBX, ECX, and EDX



32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Index and base registers

- Some registers have only a 16-bit name for their lower half. The 16-bit registers are usually used only in real-address mode.

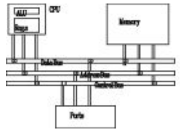


32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Some specialized register uses (1 of 2)

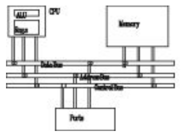
- General-Purpose
  - EAX – accumulator (automatically used by division and multiplication)
  - ECX – loop counter
  - ESP – stack pointer (should never be used for arithmetic or data transfer)
  - ESI, EDI – index registers (used for high-speed memory transfer instructions)
  - EBP – extended frame pointer (stack)

Some specialized register uses **(2 of 2)**



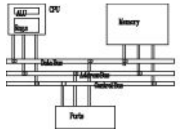
- Segment
  - CS – code segment
  - DS – data segment
  - SS – stack segment
  - ES, FS, GS - additional segments
- EIP – instruction pointer
- EFLAGS
  - status and control flags
  - each flag is a single binary bit (*set* or *clear*)

Status flags

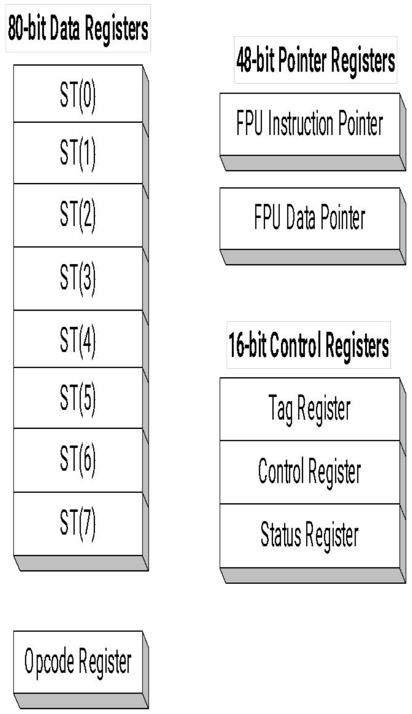


- Carry
  - unsigned arithmetic out of range
- Overflow
  - signed arithmetic out of range
- Sign
  - result is negative
- Zero
  - result is zero
- Auxiliary Carry
  - carry from bit 3 to bit 4
- Parity
  - sum of 1 bits is an even number

Floating-point, MMX, XMM registers

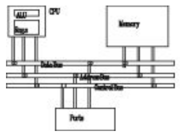


- Eight 80-bit floating-point data registers
  - ST(0), ST(1), . . . , ST(7)
  - arranged in a stack
  - used for all floating-point arithmetic
- Eight 64-bit MMX registers
- Eight 128-bit XMM registers for single-instruction multiple-data (SIMD) operations



IA-32 Memory Management

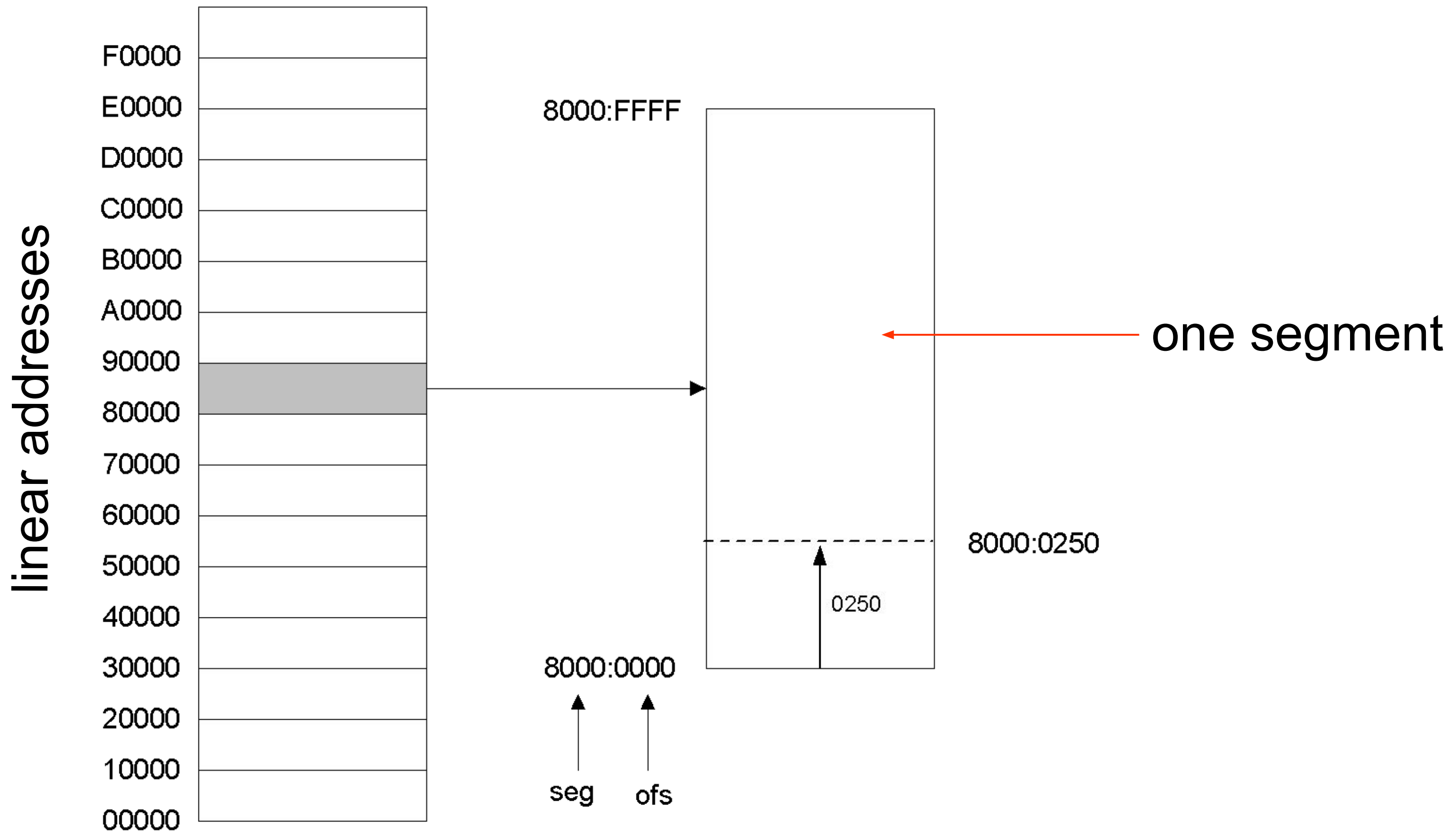
Real-address mode

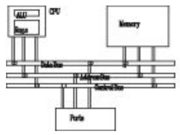


- 1 MB RAM maximum addressable (20-bit address)
- Application programs can access any area of memory
- Single tasking
- Supported by MS-DOS operating system

Segmented memory

Segmented memory addressing: absolute (linear) address is a combination of a 16-bit segment value added to a 16-bit offset





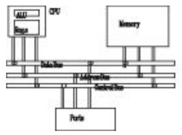
Calculating linear addresses

- Given a segment address, multiply it by 16 (add a hexadecimal zero), and add it to the offset
- Example: convert 08F1:0100 to a linear address

```
Adjusted Segment value: 0 8 F 1 0
Add the offset:           0 1 0 0
Linear address:         0 9 0 1 0
```

- A typical program has three segments: code, data and stack. Segment registers CS, DS and SS are used to store them separately.

Example



What linear address corresponds to the segment/offset address 028F:0030?

In real-address mode, the linear (or absolute) address is 20 bits, ranging from 0 to FFFFF hexadecimal. Programs cannot use linear addresses directly, so addresses are expressed using two 16-bit integers.
A segment-offset address includes the following:
• A 16-bit segment value, placed in one of the segment registers (CS, DS, ES, SS)
• A 16-bit offset value

028F0 + 0030 = 02920

Always use hexadecimal notation for addresses.

The CPU automatically converts a segment-offset address to a 20-bit linear address. Suppose a variable's hexadecimal segment-offset address is 08F1:0100. The CPU multiplies the segment value by 16 (10 hexadecimal) and adds the product to the variable's offset:

08F1h * 10h = 08F10h
Adjusted Segment value: 0 8 F 1 0
Add the offset: 0 1 0 0
Linear address: 0 9 0 1 0

$6 \times 10 = 6\underline{0}$

$08F1h \times 16h = 08F1\underline{0}$

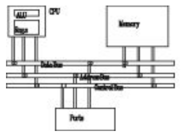
Example

What segment addresses correspond to the linear address 28F30h?

Many different segment-offset addresses can produce the linear address 28F30h. For example:

28F0:0030, 28F3:0000, 28B0:0430, . . .

Protected mode (1 of 2)



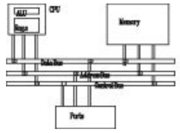
- 4 GB addressable RAM (32-bit address)
  - (00000000 to FFFFFFFFh)
- Each program assigned a memory partition which is protected from other programs
- Designed for multitasking
- Supported by Linux & MS-Windows

Protected mode is the more powerful “native” processor mode. When running in protected mode, a program’s linear address space is 4 GBytes, using addresses 0 to FFFFFFFF hexadecimal. In the context of the Microsoft Assembler, the flat segmentation model is appropriate for protected mode programming. The flat model is easy to use because it requires only a single 32-bit integer to hold the address of an instruction or variable. The CPU performs address calculation and translation in the background, all of which are transparent to application programmers. Segment registers (CS, DS, SS, ES, FS, GS) point to segment descriptor tables, which the operating system uses to keep track of locations of individual program segments. A typical protected-mode program has three segments: code, data, and stack, using the CS, DS, and SS segment registers:

• CS references the descriptor table for the code segment
• DS references the descriptor table for the data segment
• SS references the descriptor table for the stack segment

Simplified: Protected mode is a native processor mode. When protected mode is enabled program's linear address space is 4GB. For protected mode flat segmentation model is appropriate. Because it uses 32-bit integer to hold the address of an instruction. Segment registers(CS, DS, SS, ES, FS, GS) point to segment descriptor tables. Protected mode has three segments.

Protected mode (2 of 2)



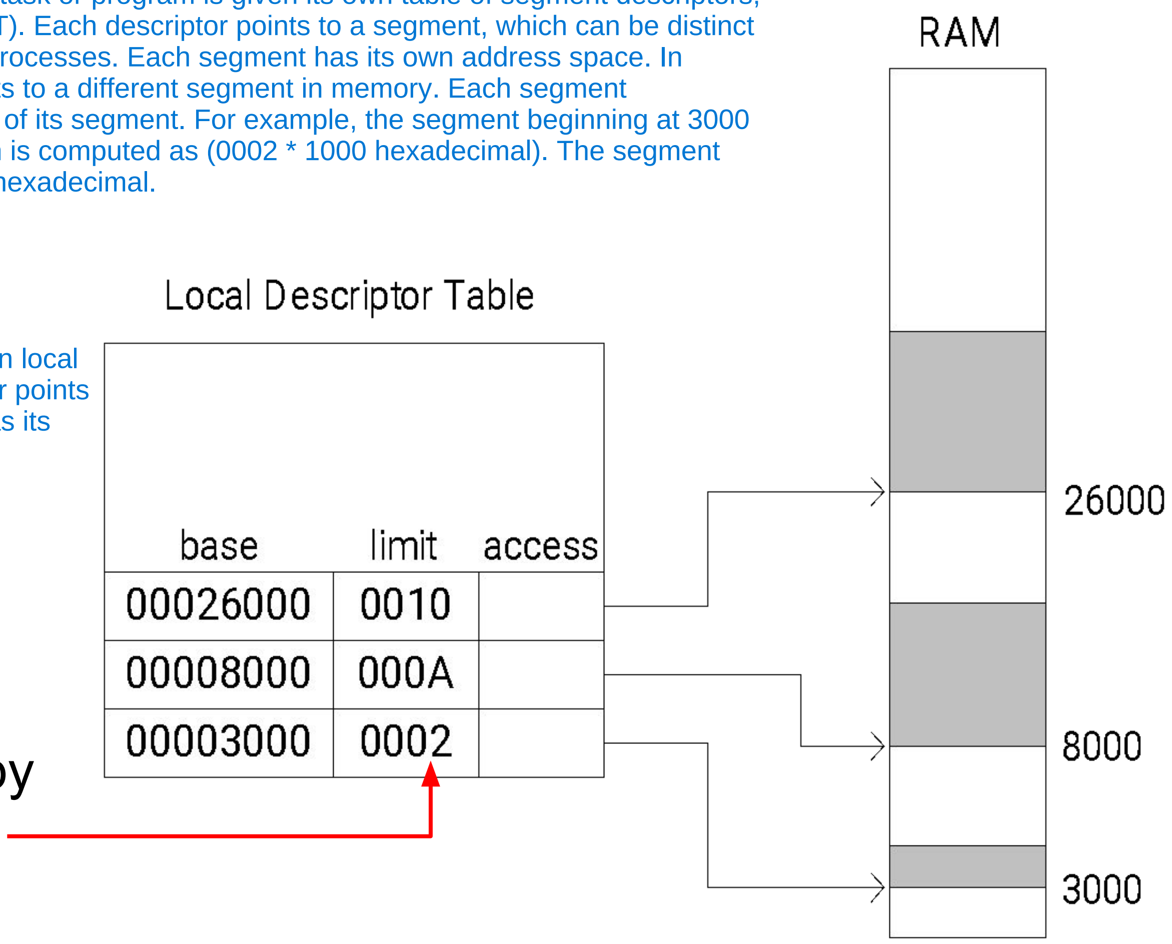
- Segment descriptor tables
- Program structure
  - code, data, and stack areas
  - CS, DS, SS segment descriptors
  - global descriptor table (GDT)
- MASM Programs use the Microsoft flat memory model

Multi-segment model

- Each program has a local descriptor table (LDT)
  - holds descriptor for each segment used by the program

In the multi-segment model, each task or program is given its own table of segment descriptors, called a local descriptor table (LDT). Each descriptor points to a segment, which can be distinct from all segments used by other processes. Each segment has its own address space. In Figure each entry in the LDT points to a different segment in memory. Each segment descriptor specifies the exact size of its segment. For example, the segment beginning at 3000 has size 2000 hexadecimal, which is computed as (0002 * 1000 hexadecimal). The segment beginning at 8000 has size A000 hexadecimal.

Simplified: Each task has its own local descriptor table. Each descriptor points to a segment. Each segment has its own address space.



base	limit	access
00026000	0010	
00008000	000A	
00003000	0002	

multiplied by

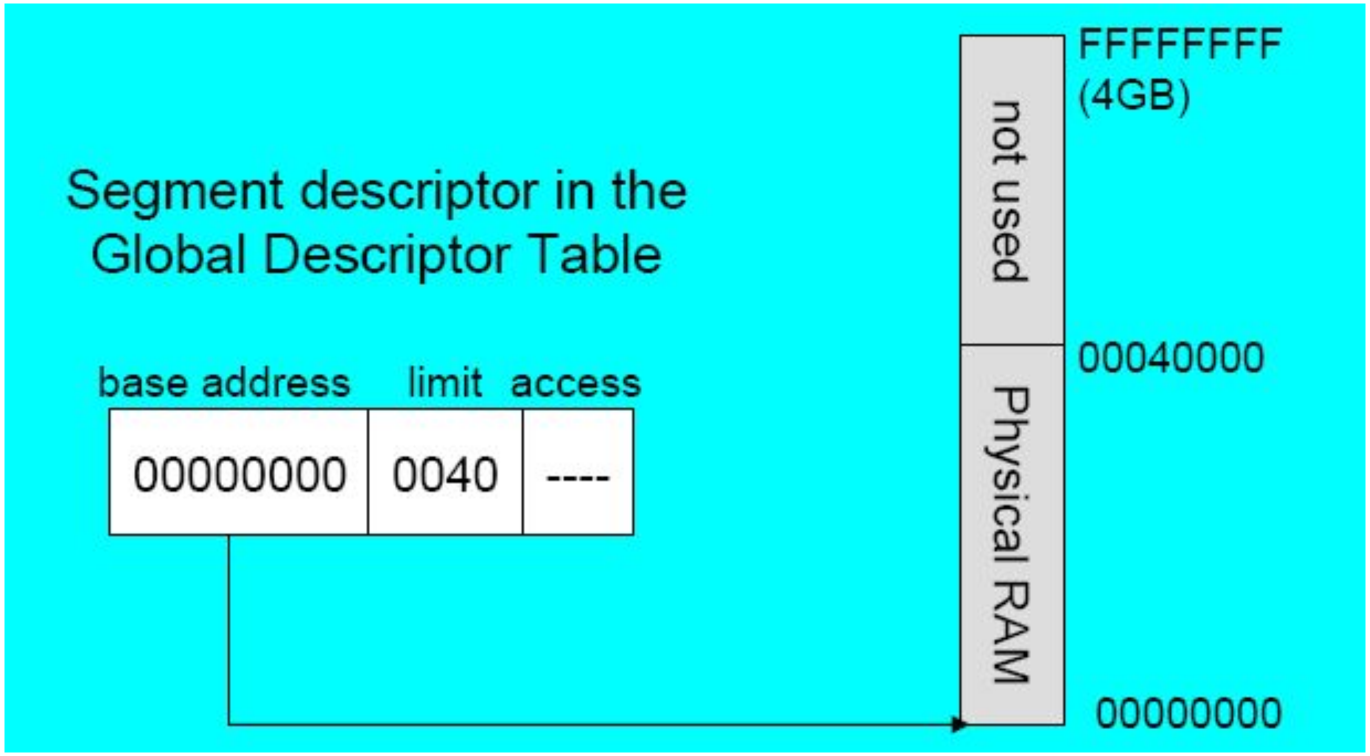
1000h

Flat segmentation model

- All segments are mpped to the entire 32-bit physical address space, at least two, one for data and one for code

In the flat segmentation model, all segments are mapped to the entire 32-bit physical address space of the computer. At least two segments are required, one for code and one for data.
Each segment is defined by a segment descriptor, a 64-bit integer stored in a table known as the global descriptor table (GDT)

- global descriptor table (GDT)



Paging

- Virtual memory uses disk as part of the memory, thus allowing sum of all programs can be larger than physical memory
- Divides each segment into 4096-byte blocks called pages

- Page fault (supported directly by the CPU) – issued by CPU when a page must be loaded from disk
- Virtual memory manager (VMM) – OS utility that manages the loading and unloading of pages

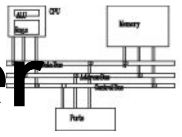
x86 processors support paging, a feature that permits segments to be divided into 4,096-byte blocks of memory called pages. Paging permits the total memory used by all programs running at the same time to be much larger than the computer's physical memory.

Paging is an important solution to a vexing problem faced by software and hardware designers. A program must be loaded into main memory before it can run, but memory is expensive. Users want to be able to load numerous programs into memory and switch among them at will. Disk storage, on the other hand, is cheap and plentiful. Paging provides the illusion that memory is almost unlimited in size. Disk access is much slower than main memory access, so the more a program relies on paging, the slower it runs.

When a task is running, parts of it can be stored on disk if they are not currently in use. Parts of the task are paged (swapped) to disk. Other actively executing pages remain in memory. When the processor begins to execute code that has been paged out of memory it issues a page fault, causing the page or pages containing the required code or data to be loaded back into memory. To see how this works, find a computer with somewhat limited memory and run many large applications at the same time. You should notice a delay when switching from one program to another because the operating system must transfer paged portions of each program into memory from disk. A computer runs faster when more memory is installed because large application files and programs can be kept entirely in memory, reducing the amount of paging.

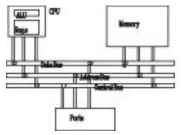
Components of an IA-32 microcomputer

Components of an IA-32 Microcomputer



- Motherboard
- Video output
- Memory
- Input-output ports

Motherboard



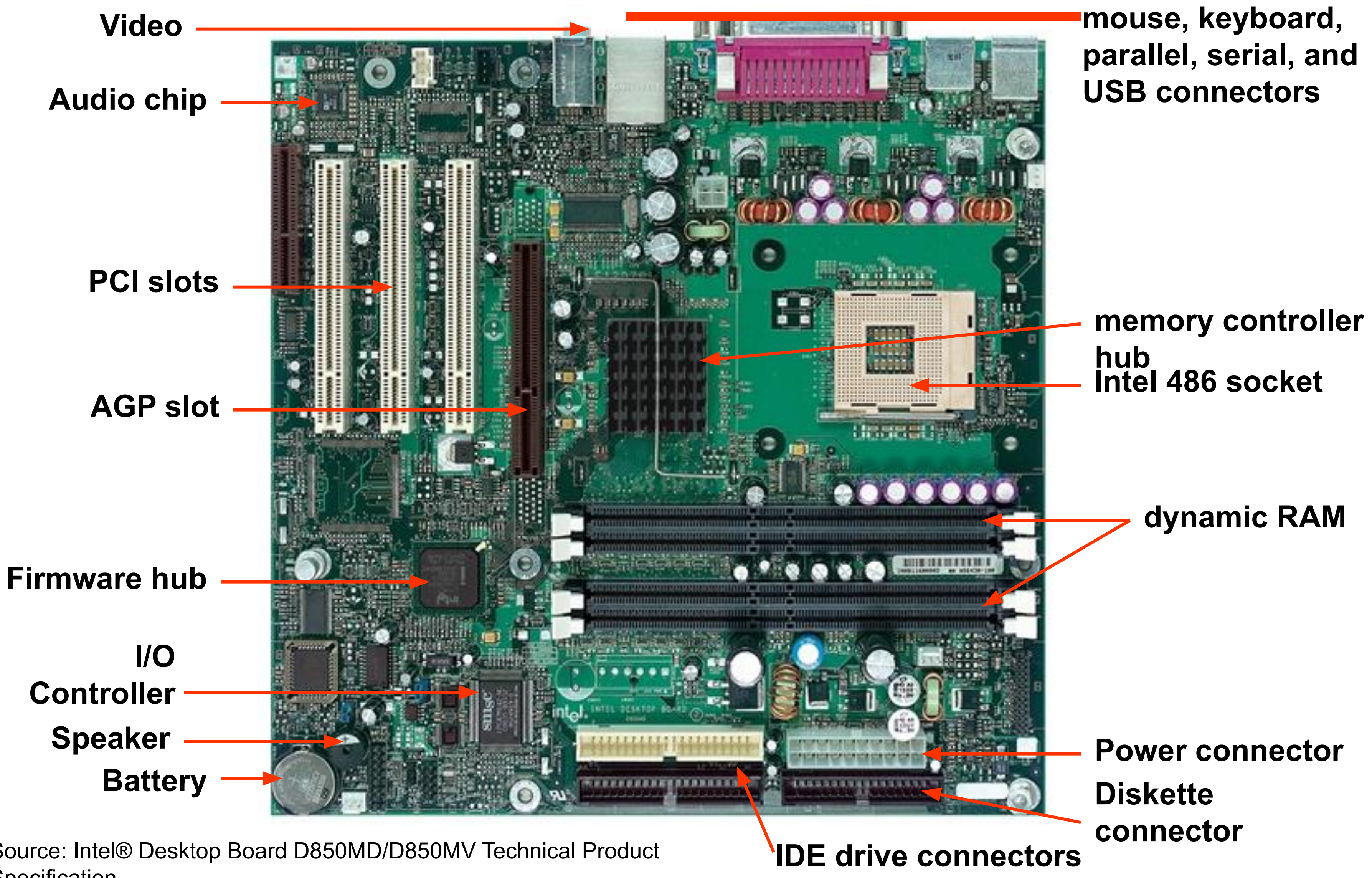
- CPU socket
- External cache memory slots
- Main memory slots
- BIOS chips

BIOS (basic input-output system) computer chips, holding system software

- Sound synthesizer chip (optional)
- Video controller chip (optional)
- IDE, parallel, serial, USB, video, keyboard, joystick, network, and mouse connectors
- PCI bus connectors (expansion cards)

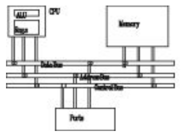
• Connectors for mass-storage devices such as hard drives and CD-ROMs
• USB connectors for external devices
• Keyboard and mouse ports
• PCI bus connectors for sound cards, graphics cards, data acquisition boards, and other input-output devices

Intel D850MD motherboard



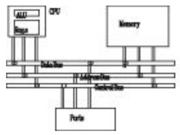
Source: Intel® Desktop Board D850MD/D850MV Technical Product Specification

Video Output



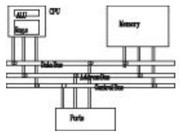
- Video controller
  - on motherboard, or on expansion card
  - AGP (accelerated graphics port)
- Video memory (VRAM)
- Video CRT Display
  - uses raster scanning
  - horizontal retrace
  - vertical retrace
- Direct digital LCD monitors
  - no raster scanning required

Memory



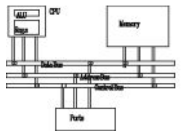
- ROM
  - read-only memory
- EPROM
  - erasable programmable read-only memory
- Dynamic RAM (DRAM)
  - inexpensive; must be refreshed constantly
- Static RAM (SRAM)
  - expensive; used for cache memory; no refresh required
- Video RAM (VRAM)
  - dual ported; optimized for constant video refresh
- CMOS RAM
  - refreshed by a battery
  - system setup information

Input-output ports



- USB (universal serial bus)
  - intelligent high-speed connection to devices
  - up to 12 megabits/second
  - USB hub connects multiple devices
  - *enumeration*: computer queries devices
  - supports *hot* connections
- Parallel
  - short cable, high speed
  - common for printers
  - bidirectional, parallel data transfer
  - Intel 8255 controller chip

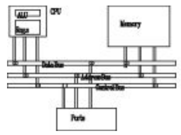
Input-output ports (cont)



- Serial
  - RS-232 serial port
  - one bit at a time
  - used for long cables and modems
  - 16550 UART (universal asynchronous receiver transmitter)
  - programmable in assembly language

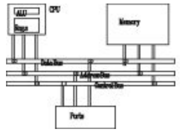
Intel microprocessor history

Early Intel microprocessors



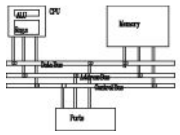
- Intel 8080
  - 64K addressable RAM
  - 8-bit registers
  - CP/M operating system
  - 5,6,8,10 MHz
  - 29K transistros
- Intel 8086/8088 (1978)
  - IBM-PC used 8088
  - 1 MB addressable RAM
  - 16-bit registers
  - 16-bit data bus (8-bit for 8088)
  - separate floating-point unit (8087)
  - used in low-cost microcontrollers now

The IBM-AT



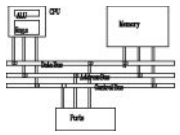
- Intel 80286 (1982)
  - 16 MB addressable RAM
  - Protected memory
  - several times faster than 8086
  - introduced IDE bus architecture
  - 80287 floating point unit
  - Up to 20MHz
  - 134K transistors

Intel IA-32 Family



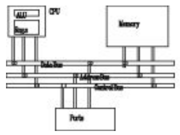
- Intel386 (1985)
  - 4 GB addressable RAM
  - 32-bit registers
  - paging (virtual memory)
  - Up to 33MHz
- Intel486 (1989)
  - instruction pipelining
  - Integrated FPU
  - 8K cache
- Pentium (1993)
  - Superscalar (two parallel pipelines)

Intel P6 Family



- Pentium Pro (1995)
  - advanced optimization techniques in microcode
  - More pipeline stages
  - On-board L2 cache
- Pentium II (1997)
  - MMX (multimedia) instruction set
  - Up to 450MHz
- Pentium III (1999)
  - SIMD (streaming extensions) instructions (SSE)
  - Up to 1+GHz
- Pentium 4 (2000)
  - NetBurst micro-architecture, tuned for multimedia
  - 3.8+GHz
- Pentium D (Dual core)

CISC and RISC



- CISC – complex instruction set
  - large instruction set
  - high-level operations (simpler for compiler?)
  - requires microcode interpreter (could take a long time)
  - examples: Intel 80x86 family
- RISC – reduced instruction set
  - small instruction set
  - simple, atomic instructions
  - directly executed by hardware very quickly
  - easier to incorporate advanced architecture design
  - examples:
    - ARM (Advanced RISC Machines)
    - DEC Alpha (now Compaq)