MACHINE LEVEL REPRESENTATION I

CAS CS 210 Computer Systems

Some slides based on CMU Slides

CS:APP2e Ch 3.1-3.4

WHAT IS A CPU?
WHAT IS IT DOING?

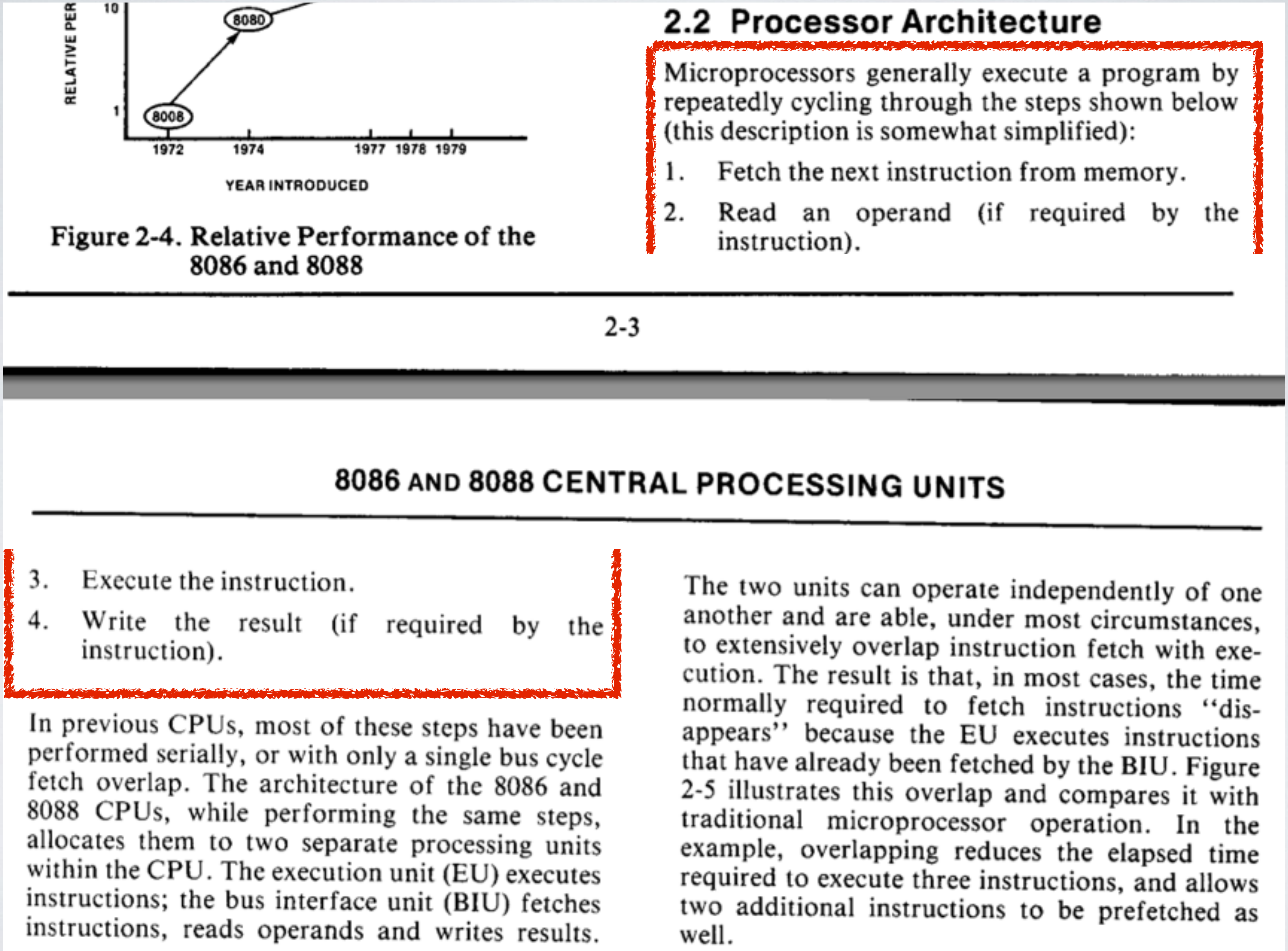

RELATIVE PER
10
1
8080
8008
1972
1974
1977
1978
1979
YEAR INTRODUCED

Figure 2-4. Relative Performance of the 8086 and 8088

2.2 Processor Architecture

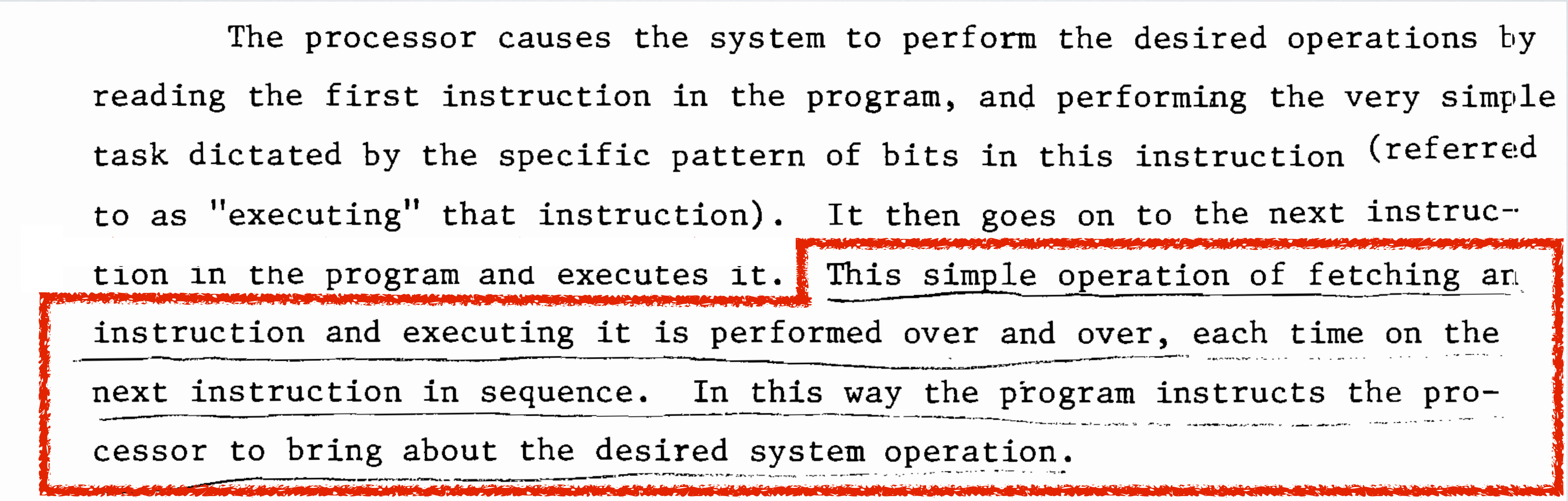
Microprocessors generally execute a program by repeatedly cycling through the steps shown below (this description is somewhat simplified):

1. Fetch the next instruction from memory.
2. Read an operand (if required by the instruction).

3. Execute the instruction.
4. Write the result (if required by the instruction).

In previous CPUs, most of these steps have been performed serially, or with only a single bus cycle fetch overlap. The architecture of the 8086 and 8088 CPUs, while performing the same steps, allocates them to two separate processing units within the CPU. The execution unit (EU) executes instructions; the bus interface unit (BIU) fetches instructions, reads operands and writes results.

The two units can operate independently of one another and are able, under most circumstances, to extensively overlap instruction fetch with execution. The result is that, in most cases, the time normally required to fetch instructions "disappears" because the EU executes instructions that have already been fetched by the BIU. Figure 2-5 illustrates this overlap and compares it with traditional microprocessor operation. In the example, overlapping reduces the elapsed time required to execute three instructions, and allows two additional instructions to be prefetched as well.



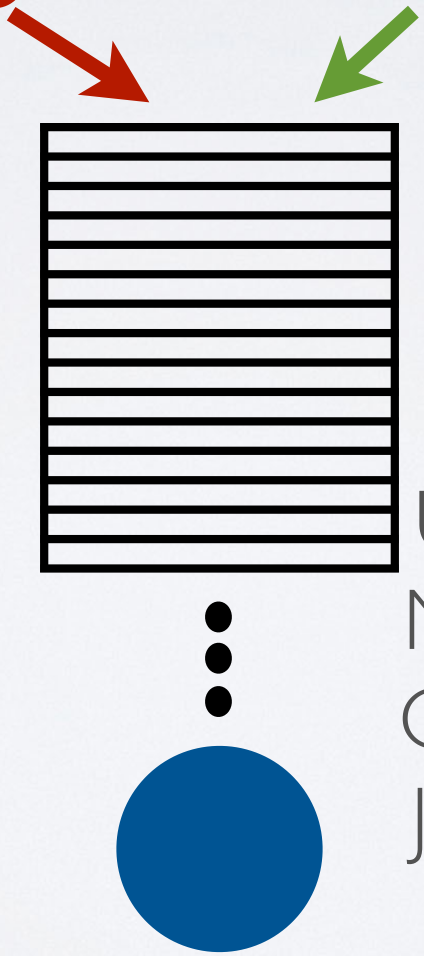
The processor causes the system to perform the desired operations by reading the first instruction in the program, and performing the very simple task dictated by the specific pattern of bits in this instruction (referred to as "executing" that instruction). It then goes on to the next instruction in the program and executes it. This simple operation of fetching an instruction and executing it is performed over and over, each time on the next instruction in sequence. In this way the program instructs the processor to bring about the desired system operation.

THE STORED PROGRAM
&
THE LOOP!

STORED PROGRAM CONCEPT

PROGRAMS (INSTRUCTIONS)

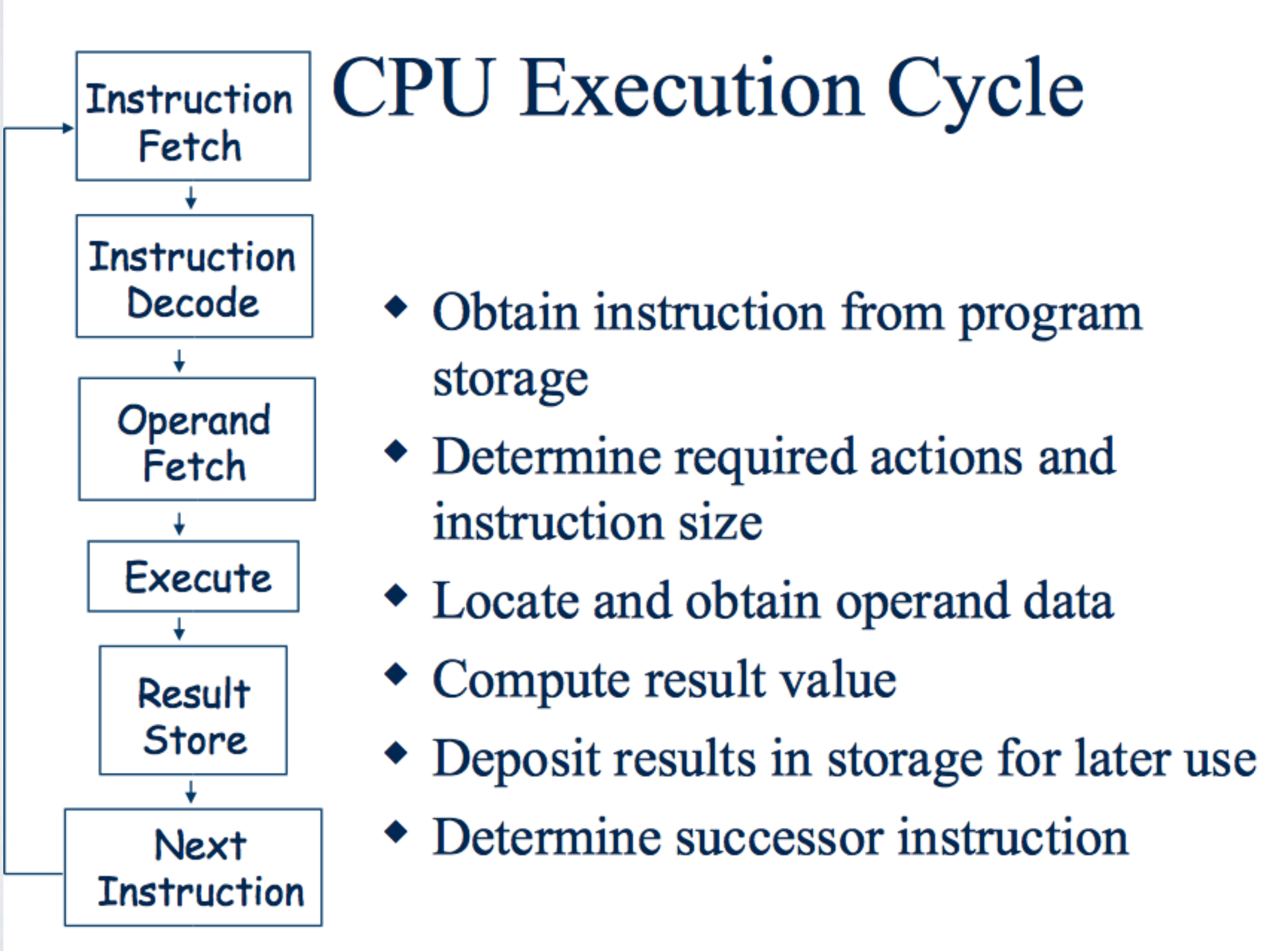
DATA



BOTH STORED IN MEMORY

UNIFIED PROCESSING MODEL EMBODIED BY CPU -- INSTRUCTIONS JUST ANOTHER FORM OF DATA

THE LOOP!
THE HEART OF MACHINE



CPU Execution Cycle

Instruction Fetch

Instruction Decode

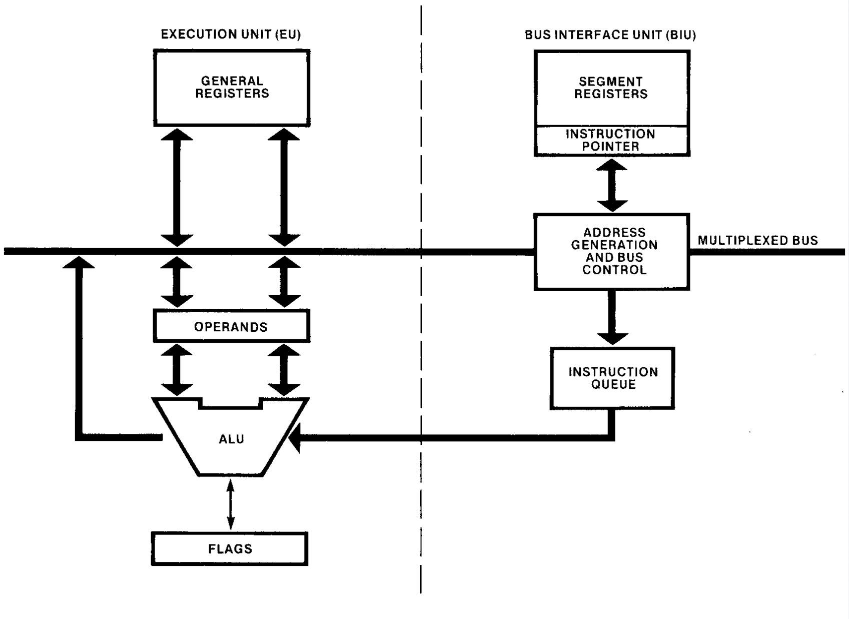
Operand Fetch

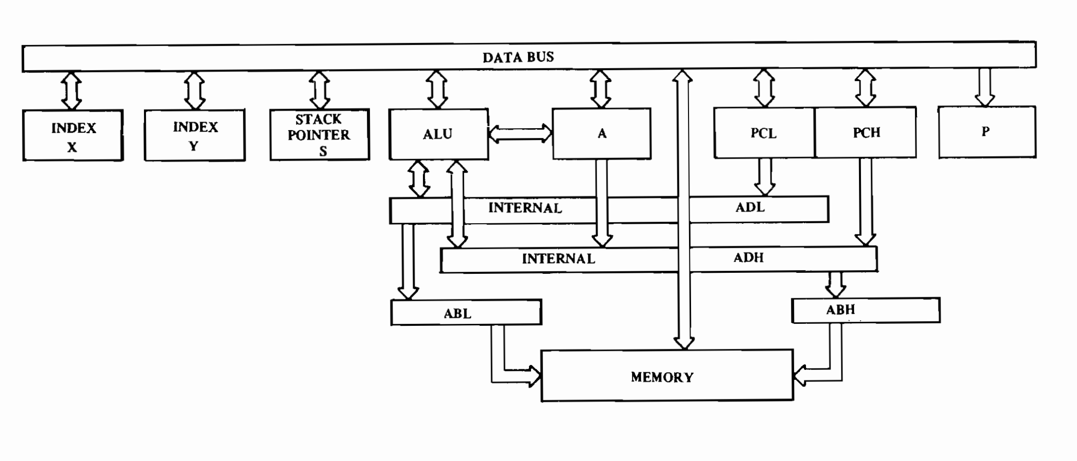
Execute

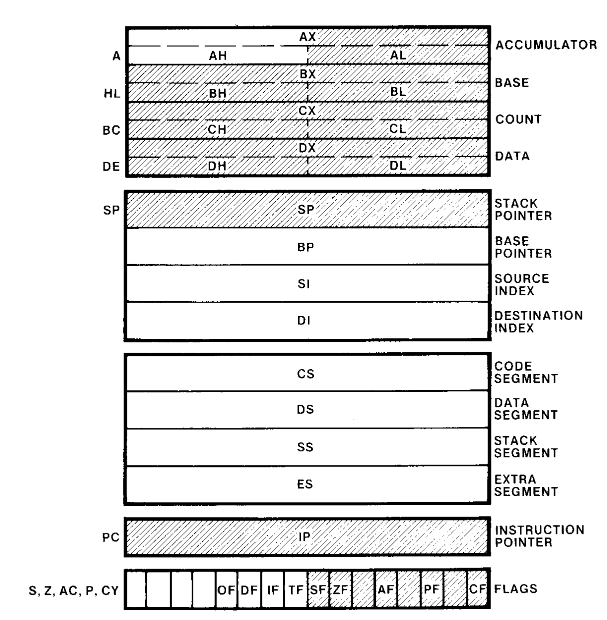
Result Store

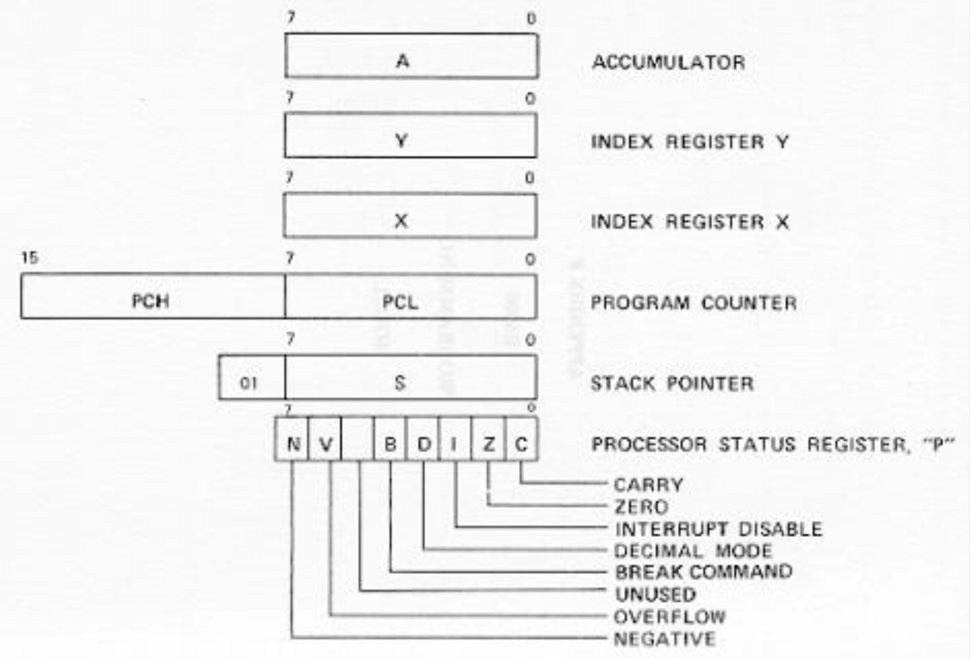
Next Instruction

- Obtain instruction from program storage
- Determine required actions and instruction size
- Locate and obtain operand data
- Compute result value
- Deposit results in storage for later use
- Determine successor instruction


EXECUTION UNIT (EU)
BUS INTERFACE UNIT (BIU)
GENERAL REGISTERS
SEGMENT REGISTERS
INSTRUCTION POINTER
ADDRESS GENERATION AND BUS CONTROL
MULTIPLEXED BUS
OPERANDS
INSTRUCTION QUEUE
ALU
FLAGS


DATA BUS
INDEX X
INDEX Y
STACK POINTER S
ALU
A
PCL
PCH
P
INTERNAL
ADL
INTERNAL
ADH
ABL
ABH
MEMORY


AX
A
AH
AL
ACCUMULATOR
BX
HL
BH
BL
BASE
CX
BC
CH
CL
COUNT
DX
DE
DH
DL
DATA
SP
SP
STACK POINTER
BP
BASE POINTER
SI
SOURCE INDEX
DI
DESTINATION INDEX
CS
CODE SEGMENT
DS
DATA SEGMENT
SS
STACK SEGMENT
ES
EXTRA SEGMENT
PC
IP
INSTRUCTION POINTER
S, Z, AC, P, CY
OF DF IF TF SF ZF AF PF CF
FLAGS


7 0
A
ACCUMULATOR
7 0
Y
INDEX REGISTER Y
7 0
X
INDEX REGISTER X
15 7 0
PCH
PCL
PROGRAM COUNTER
7 0
01
S
STACK POINTER
7 0
N V B D I Z C
PROCESSOR STATUS REGISTER, "P"
CARRY
ZERO
INTERRUPT DISABLE
DECIMAL MODE
BREAK COMMAND
UNUSED
OVERFLOW
NEGATIVE

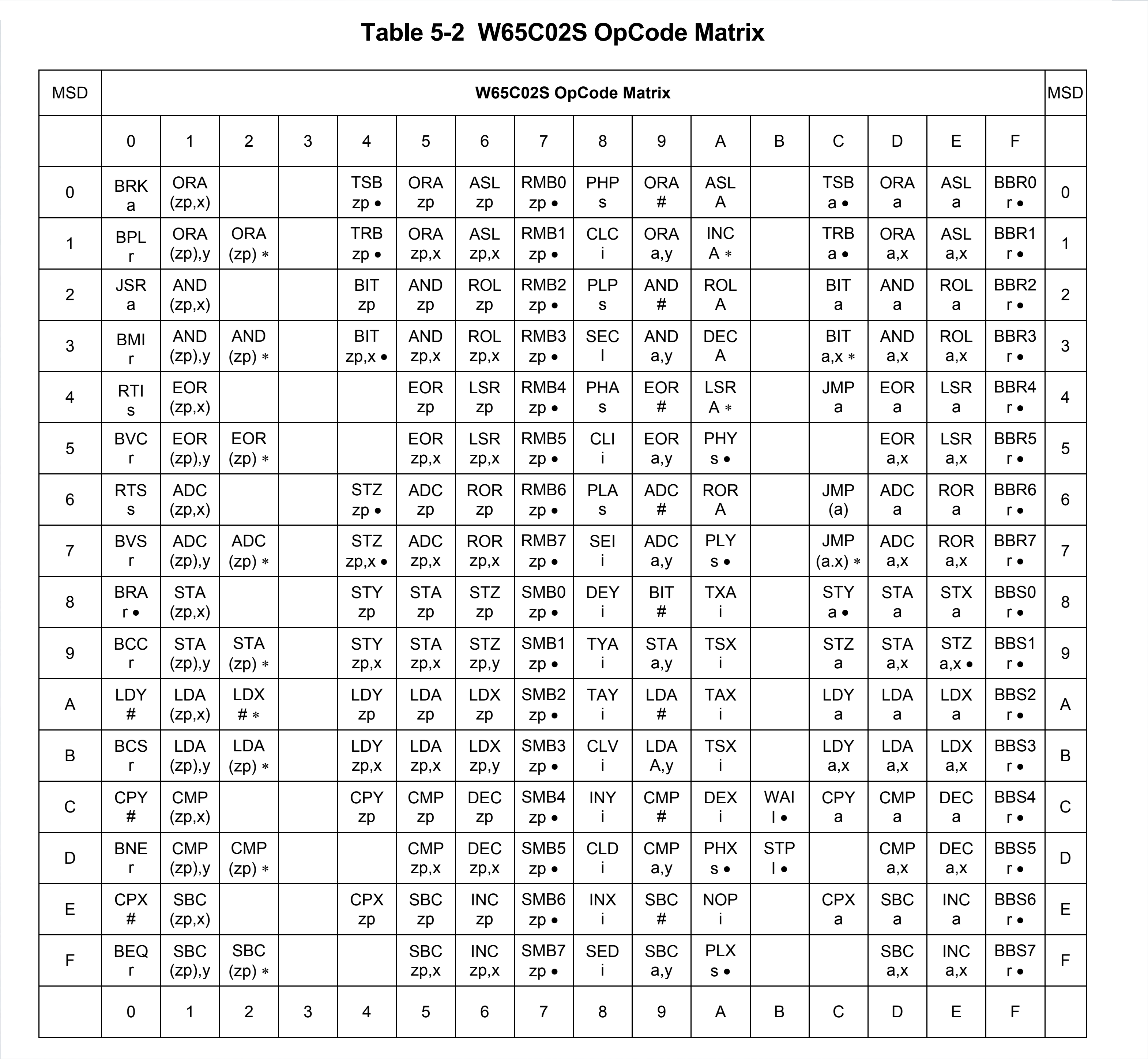
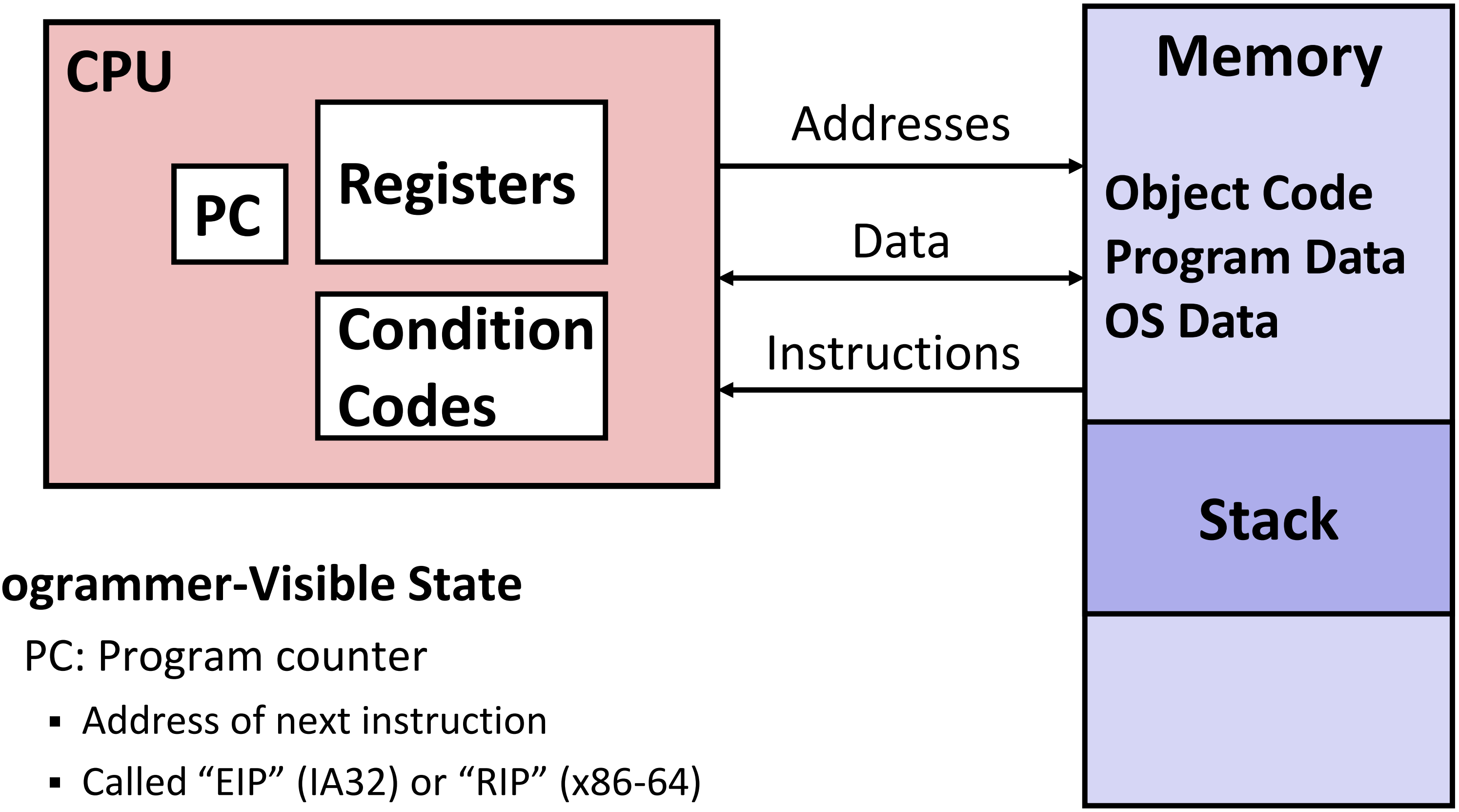


Table 5-2 W65C02S OpCode Matrix

MSD	**W65C02S OpCode Matrix**																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK a	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x •	AND zp,x	ROL zp,x	RMB3 zp •	SEC I	AND a,y	DEC A		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A *		JMP a	EOR a	LSR a	BBR4 r •	4
5	BVC r	EOR (zp),y	EOR (zp) *			EOR zp,x	LSR zp,x	RMB5 zp •	CLI i	EOR a,y	PHY s •			EOR a,x	LSR a,x	BBR5 r •	5
6	RTS s	ADC (zp,x)			STZ zp •	ADC zp	ROR zp	RMB6 zp •	PLA s	ADC #	ROR A		JMP (a)	ADC a	ROR a	BBR6 r •	6
7	BVS r	ADC (zp),y	ADC (zp) *		STZ zp,x •	ADC zp,x	ROR zp,x	RMB7 zp •	SEI i	ADC a,y	PLY s •		JMP (a.x) *	ADC a,x	ROR a,x	BBR7 r •	7
8	BRA r •	STA (zp,x)			STY zp	STA zp	STZ zp	SMB0 zp •	DEY i	BIT #	TXA i		STY a •	STA a	STX a	BBS0 r •	8
9	BCC r	STA (zp),y	STA (zp) *		STY zp,x	STA zp,x	STZ zp,y	SMB1 zp •	TYA i	STA a,y	TSX i		STZ a	STA a,x	STZ a,x •	BBS1 r •	9
A	LDY #	LDA (zp,x)	LDX # *		LDY zp	LDA zp	LDX zp	SMB2 zp •	TAY i	LDA #	TAX i		LDY a	LDA a	LDX a	BBS2 r •	A
B	BCS r	LDA (zp),y	LDA (zp) *		LDY zp,x	LDA zp,x	LDX zp,y	SMB3 zp •	CLV i	LDA A,y	TSX i		LDY a,x	LDA a,x	LDX a,x	BBS3 r •	B
C	CPY #	CMP (zp,x)			CPY zp	CMP zp	DEC zp	SMB4 zp •	INY i	CMP #	DEX i	WAI I •	CPY a	CMP a	DEC a	BBS4 r •	C
D	BNE r	CMP (zp),y	CMP (zp) *			CMP zp,x	DEC zp,x	SMB5 zp •	CLD i	CMP a,y	PHX s •	STP I •		CMP a,x	DEC a,x	BBS5 r •	D
E	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp	SMB6 zp •	INX i	SBC #	NOP i		CPX a	SBC a	INC a	BBS6 r •	E
F	BEQ r	SBC (zp),y	SBC (zp) *			SBC zp,x	INC zp,x	SMB7 zp •	SED i	SBC a,y	PLX s •			SBC a,x	INC a,x	BBS7 r •	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Assembly Programmer’s View



- **Programmer-Visible State**
  - PC: Program counter
    - Address of next instruction
    - Called “EIP” (IA32) or “RIP” (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

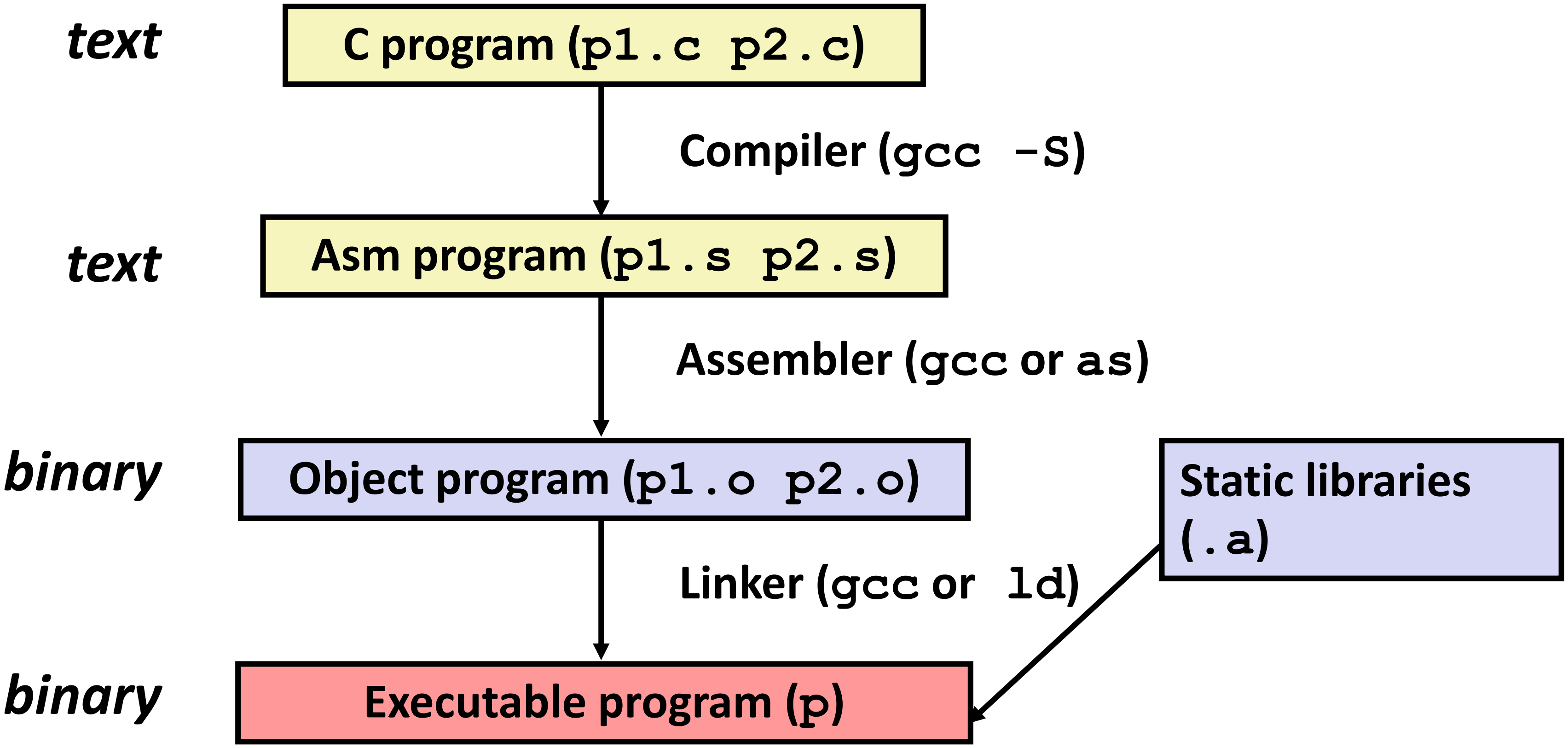
- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures

AS PROGRAMMERS WHAT HAVE WE BEEN DOING ALL THIS TIME?

HAVE WE REALLY BEEN PROGRAMMING THE COMPUTER?

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O p1.c p2.c -o p`
  - Use optimizations (`-O`)
  - Put resulting binary in file `p`



text

C program (`p1.c p2.c`)

Compiler (`gcc -S`)

text

Asm program (`p1.s p2.s`)

Assembler (`gcc` or `as`)

binary

Object program (`p1.o p2.o`)

Static libraries (`.a`)

Linker (`gcc` or `ld`)

binary

Executable program (`p`)

Compiling Into Assembly

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated IA32 Assembly

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Some compilers use single instruction "`leave`"

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`

Assembly Characteristics: Data Types

- **"Integer" data of 1, 2, or 4 bytes**
  - Data values
  - Addresses (untyped pointers)

- **Floating point data of 4, 8, or 10 bytes**

- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory

Assembly Characteristics: Operations

- **Perform arithmetic function on register or memory data**

- **Transfer data between memory and register**
  - Load data from memory into register
  - Store register data into memory

- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches

Object Code

Code for `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- **Total of 13 bytes**
- **Each instruction 1, 2, or 3 bytes**
- **Starts at address `0x401040`**

- **Assembler**
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- **Linker**
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for `malloc`, `printf`
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

Machine Instruction Example

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression:

```
x += y
```

More precisely:

```
int eax;
int *ebp;
eax += ebp[2]
```

```
0x401046:     03 45 08
```

C Code

- Add two signed integers

Assembly

- Add 2 4-byte integers
  - “Long” words in GCC parlance
  - Same instruction whether signed or unsigned
- Operands:

x:	Register	**%eax**
y:	Memory	**M[%ebp+8]**
t:	Register	**%eax**

– Return function value in **%eax**

Object Code

- 3-byte instruction
- Stored at address **0x401046**

Disassembling Object Code

Disassembled

```
00401040 <_sum>:
   0:       55                push   %ebp
   1:       89 e5             mov    %esp,%ebp
   3:       8b 45 0c          mov    0xc(%ebp),%eax
   6:       03 45 08          add    0x8(%ebp),%eax
   9:       89 ec             mov    %ebp,%esp
   b:       5d                pop    %ebp
   c:       c3                ret
   d:       8d 76 00          lea    0x0(%esi),%esi
```

- **Disassembler**

  `objdump -d p`

  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Disassembled

```
0x401040 <sum>:      push   %ebp
0x401041 <sum+1>:    mov    %esp,%ebp
0x401043 <sum+3>:    mov    0xc(%ebp),%eax
0x401046 <sum+6>:    add    0x8(%ebp),%eax
0x401049 <sum+9>:    mov    %ebp,%esp
0x40104b <sum+11>:   pop    %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea    0x0(%esi),%esi
```

- **Within gdb Debugger**

  `gdb p`

  `disassemble sum`

  - Disassemble procedure

  `x/13b sum`

  - Examine the 13 bytes starting at `sum`

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55               push   %ebp
30001001:  8b ec            mov    %esp,%ebp
30001003:  6a ff            push   $0xffffffff
30001005:  68 90 10 00 30   push   $0x30001090
3000100a:  68 91 dc 4c 30   push   $0x304cdc91
```

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

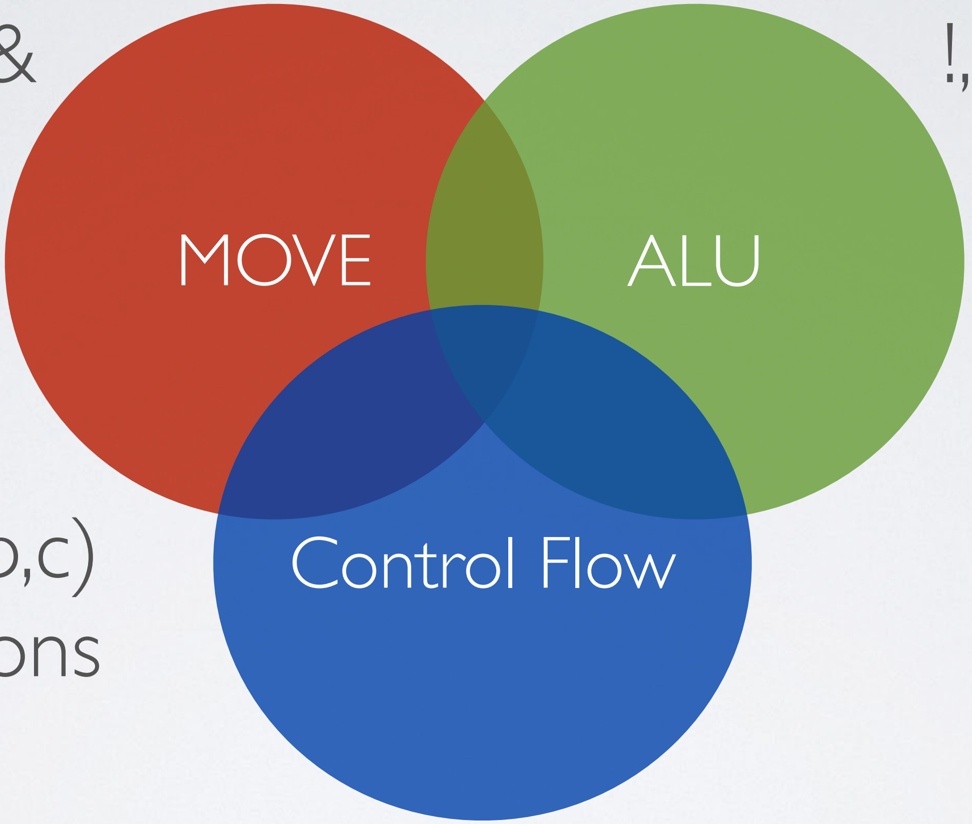
=

*,&

~,|,&,^,<<,>>

+,-,%,*,/

!,||,&&,==



MOVE

ALU

Control Flow

global vs local
variables

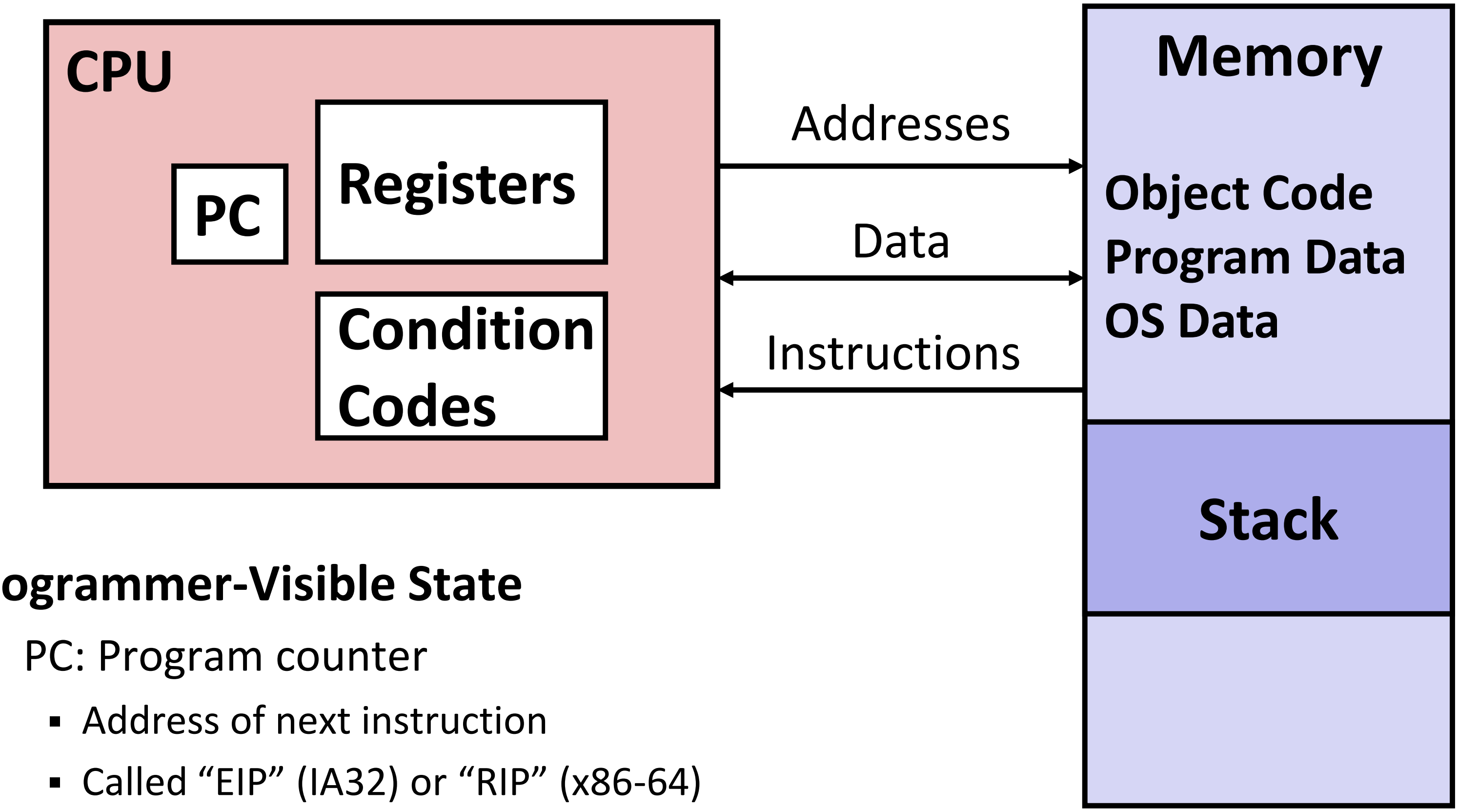
foo(a,b,c)

Functions

if;then;else

for, while, goto

Assembly Programmer’s View



- **Programmer-Visible State**
  - PC: Program counter
    - Address of next instruction
    - Called “EIP” (IA32) or “RIP” (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures