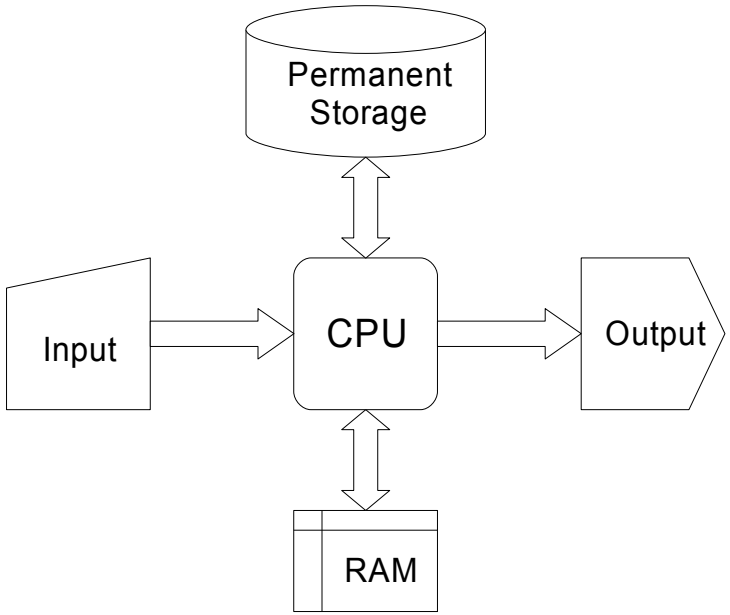
Class #1

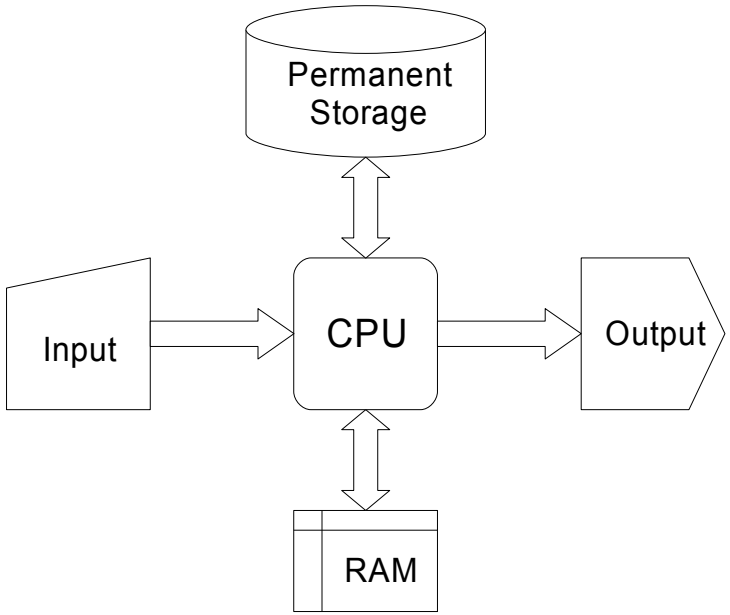
Programming and Numerical Methods for Scientists and Engineers

“What every computer has to have”



- Central Processing Unit.
- At least one input device.
- At least one output device.
- Memory:
  - almost always some Random Access Memory
  - usually some permanent storage.

“All Computing is Numeric”



- The memory stores numbers in locations with numeric addresses.
- The I/O devices read and write numbers.
- The CPU processes numbers.
- Text? Each character has a numeric code.
- Images? Each pixel has a numeric value.

Binary Numbers

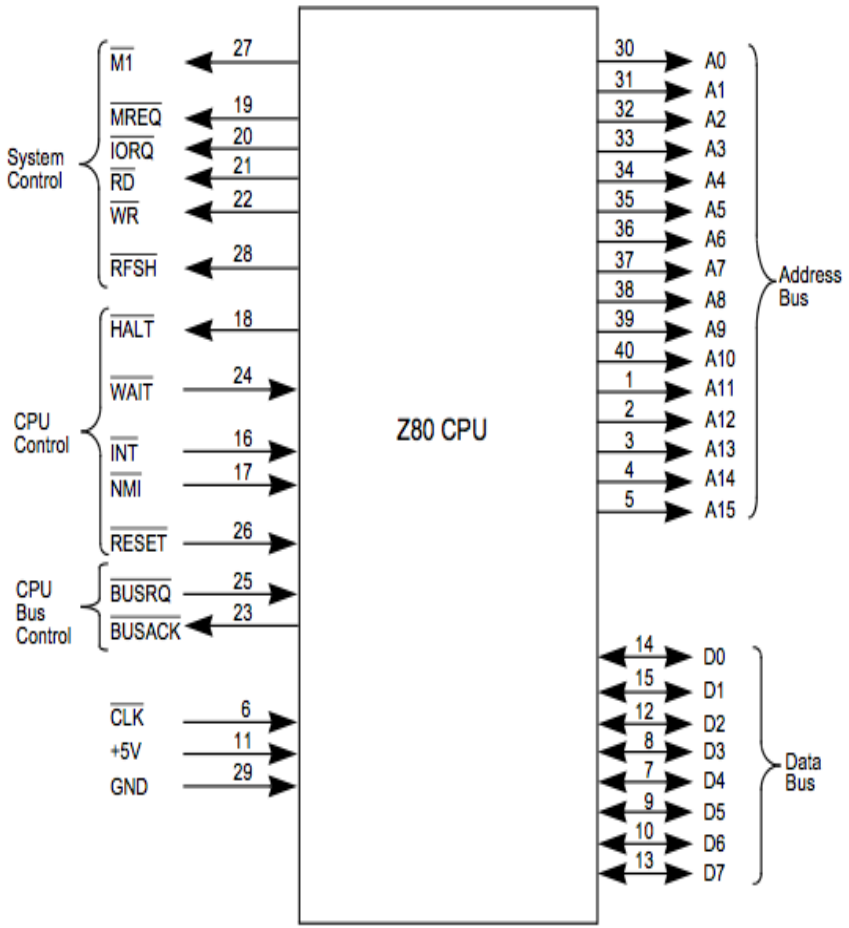
```
binary    octal  hex    decimal
00000000  000    00     0
00000001  001    01     1
00000010  002    02     2
00000011  003    03     3
00000100  004    04     4
00000101  005    05     5
00000110  006    06     6
00000111  007    07     7
00001000  010    08     8
...
00001111  017    0F     15
00010000  020    10     16
00010001  021    11     17
...
```

- Groups of 3 or 4 bits separate conveniently into the octal (base 8) or hexidecimal (base 16) numbers.
- In C, C++, and some other languages, hexidecimal constants begin with 0x to distinguish them from decimal. E.g., 0x42=66≠42=0x2A
- Z80 assembly language uses a trailing ``h'' to indicate a hexidecimal number. (E.g., 66=42h≠42.)

Memory, Input, and Output

- RAM is a collection binary numbers (“bytes” or “words”), each accessible by a unique numeric address. (E.g., 8bit values in 2^{16} addresses for a lowly Z80 microprocessor.)
- Simple input devices provide one byte or word at a time. Each device has its own address.
- Simple output devices accept one byte or word at a time. Each device has its own address.
- More sophisticated I/O devices are possible, such as block-transfer or direct memory access. (Typically used for mass storage devices like disk drives.)

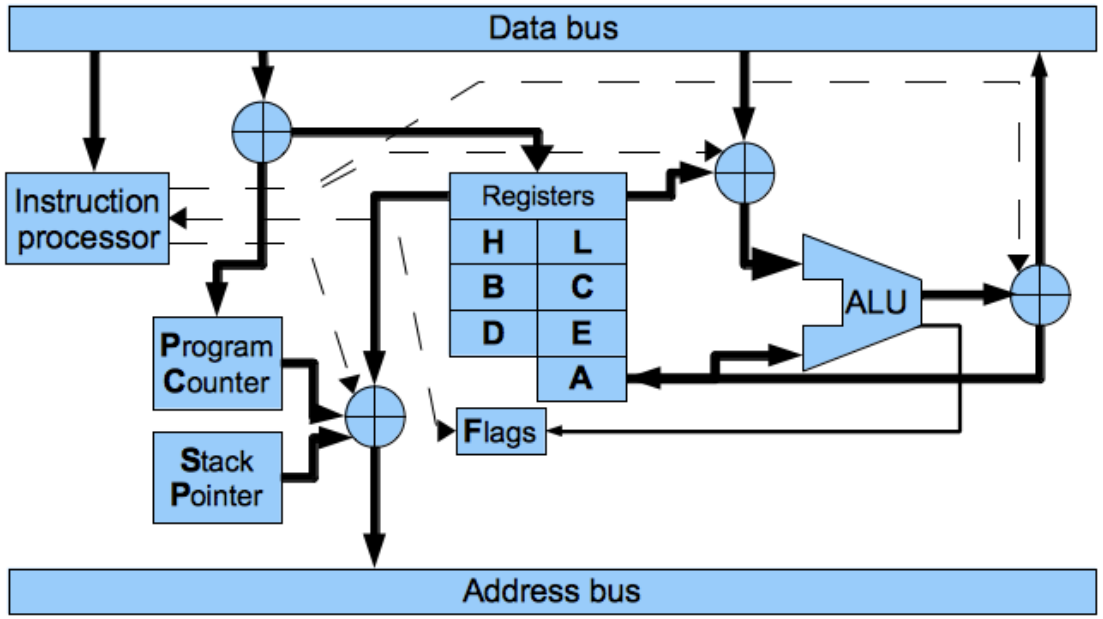
What a CPU is



From Z80 User's Manual, (c) Zilog 2004.

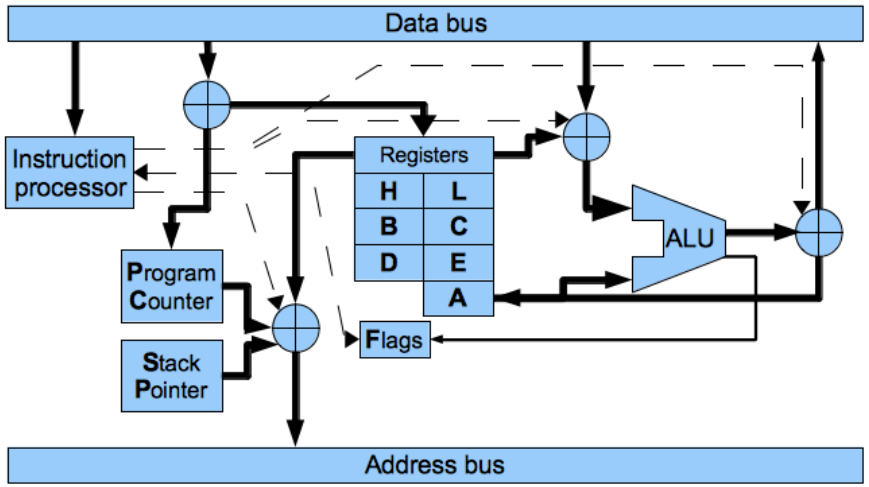
- Usually a big integrated circuit.
- Memory read/write pins initiate memory access, address pins select what memory location to access, data pins have the actual data read in/out.
- Some CPUs have an I/O pin to direct data from/to input/output devices. (Alternatively, they may use "memory mapped I/O".)

What's inside the CPU?



- Counters, “registers”, an “arithmetic logic unit”, an instruction decoder, and a lot of switches.

What does the CPU do?



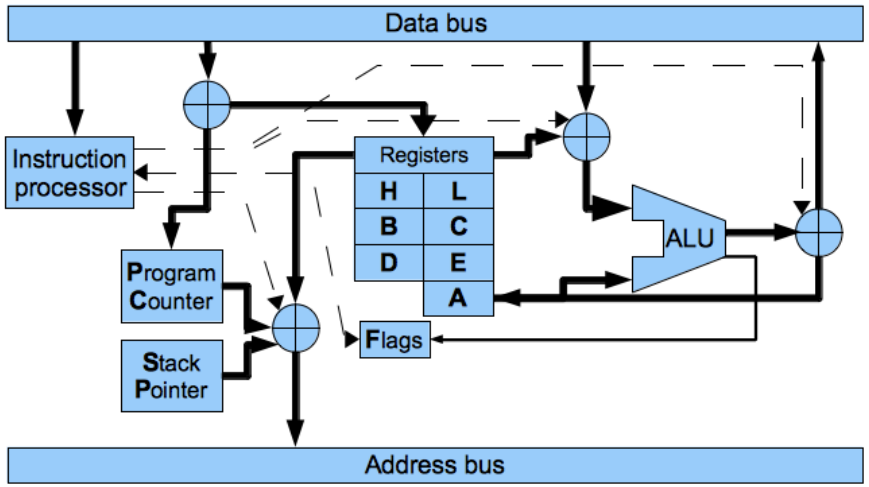
The CPU repeatedly follows an "instruction cycle". Simplified version:

1. Put the value of the Program Counter on the address bus, read the number there.

2. Operate on the registers, flags, and memory in some way depending on that number.

The number fetched in step 1 is called the instruction or operation code.

What kinds of operations can the CPU perform? (Just some examples.)



- Read a value from a memory location into a register.
- Add, subtract, or compare two registers, shift the bits in a register left or right, other kinds of mathematical operations.
- Store a value from a register into a memory location.
- Set the program counter to a different value (“jump”), possibly conditionally on a flag (“conditional jump”), possibly saving the old value first (“call”).

Examples of opcodes in Z80 instruction set

#bytes	binary (or hex)	operation	mnemonic	description
1	01(r) ^{3}(r') ^{3}	*r←r'*	LD r, r'	stores contents of register r' into register r
2	00(r) ^{3}110, (n) ^{8}	*r←n*	LD r, n	register r is set to the byte n
1	01(r) ^{3}110	*r←(HL)*	LD r, (HL)	byte at address (HL) is loaded into r
1	01110(r) ^{3}	(HL)←r	LD (HL), r	the byte in register r is stored to address (HL)
3	00(d d) ^{2}0001, (n n) ^{16}	*d d←n n*	LD dd, nn	register pair d d is set to 16-bit word n n
3	2A, (n n) ^{16}	HL ← (n n)	LD HL, (nn)	register pair HL is set from (nn + 1) and (nn)
3	22, (n n) ^{16}	(n n)←HL	LD (nn), HL	HL is stored to address (nn + 1) and (nn)
2	D3,(n) ^{8}	O(n)←A	OUT (n), A	outputs byte in A to I/O address (n)
1	00(d d) ^{2}0011	*d d←d d + 1*	INC dd	increments register pair dd
2	FE, (n) ^{8}	*A - n*	CP n	compares A to byte n, sets flags
3	C3, (n n) ^{16}	PC ← n n	JP nn	jumps to location nn
3	C2, (n n) ^{16}	PC ← n n iff not Z	JP NZ, nn	jumps only if Z flag not set
2	00(d d) ^{2}1001	HL←HL + d d	ADD HL, dd	adds register dd to HL and stores in HL
1	76	halt	HALT	halts the CPU
1	0	"no operation"	NOP	does nothing but advance PC

See also the Zilog Z80 manual [Z80manual] in References.

Example Z80 program:
output three bytes to I/O address 11(hex) and halt

address	opcode bytes	mnemonic
0000	3E 41	LD A, 41H
0002	D3 11	OUT (11H),A
0004	3E 0D	LD A, 0DH
0006	D3 11	OUT (11H),A
0008	3E 0A	LD A, 0AH
000A	D3 11	OUT (11H),A
000C	76	HALT

Exercise 1

- Enter and run the program on the previous slide in the Altair/Z80 emulator [SIMH].
  - Enter it byte-by-byte, as if you were setting the address and data switches on a very old-fashioned computer.
  - Instructor will demonstrate.

Exercise 2

- Enter a new program that uses the “LD A, (HL)”, “JP Z, (address)”, “CP (n)” and “INC HL” opcodes (and the ones in the previous example) to read and output from consecutive memory locations until a 0 byte is found.
  - For this one, you can use the opcode input option of the Altair/Z80 emulator. (Instructor will demonstrate.)
- Use it to print “Hello World!” to the “printer” on I/O port 11H, or any message of your choice.

Exercise 3 (start in class)

- Write a subroutine that prints out the value of A in hexidecimal.
  - A subroutine is just a section of code that ends with the RET instruction. CALL (address) will save the PC on the stack and jump to (address), and RET will get the new value of the PC off of the stack.
  - Copy the value of A to C first, since you'll need to use A for the character code to output to port 11H.
  - You can use AND A,0FH to clear the top 4 bits. You can use SRL C to shift the bits of C right by one (divide by 2). (Refer to the [Z80manual].) You can use the fact that 30H is the ASCII code for '0', and 41H is the ASCII code for 'A'.
- Test for all values of A.

How to multiply two 16-bit numbers with an 8-bit CPU that has no multiply opcode

- General description of algorithm: Use a binary version of the way we multiply two multi-digit numbers by hand, with the simplification that 1*x = x and 0*x = 0.
- Use registers as follows:
  - on entrance: multiplier in DE, multiplicand in HL.
  - on exit: result in HL.
    - H high order partial result, L low order partial result
    - D high order multiplicand, E low order multiplicand
    - B counter for number of shifts,
    - C high order bits of multiplier, A low order bits of multiplier

How to multiply two 16-bit numbers with an 8-bit CPU that has no multiply opcode

```
 0:  06 10      LD B,10h        ; load 16 into B
 2:  4A         LD C,D          ; move D into C
 3:  7B         LD A,E          ; move E into A
 4:  EB         EX DE,HL        ; swap DE and HL
 5:  21 00 00   LD HL,0000h     ; initialize HL with 0
 8:  CB 39      SRL C           ; shift C right, low bit to carry
 A:  1F         RRA             ; rotate A right through carry
 B:  30 01      JR NC,000Eh     ; jump if carry not set
 D:  19         ADD HL,DE       ; HL <- HL + DE
 E:  EB         EX DE,HL        ; temporarily swap DE<->HL
 F:  29         ADD HL,HL       ; use for quick multiply-by-two
10:  EB         EX DE,HL        ; swap back
11:  05         DEC B           ; count down
12:  C2 08 00   JP NZ,0008h     ; will loop 16 times
15:  C9         RET             ; return to caller
```

Note: you may want to relocate this code to start at some other address than 0.

Assignment #1

A) Write up your own explanation of how the multiplication subroutine implemented on the previous page actually works. Use flow charts and equations as necessary.

B) Write a test program to check the multiplication subroutine, and run it. The program should use the CALL instruction and work as follows:

- load HL and DE with the numbers to be multiplied
- print the values to be multiplied
- CALL (address of the multiplication subroutine)
- print resulting value of HL
- repeat with at least 10 different values.
- Submit a copy of the program and its output.

Goals

- Learn what computational algorithms fundamentally are.
- Learn how algorithms are implemented at the lowest level of the computer.
- Form a strong basis for understanding what the compiler of a higher level language actually does.

Resources

[SIMH] SIMH simulator from the Computer History Simulation Project: http://simh.trailing-edge.com/

[Z80manual] Zilog Z80 instruction manual: http://www.zilog.com/docs/z80/um0080.pdf