A Program for Elsie, a 4-bit Computer

Or, Your Code *Really* Looks Like *This*

What is a computer program?

What is a computer program?

It’s a set of computer instructions that accomplish a task

What is Elsie?

Elsie is a tiny computer with a 4-bit bus

Elsie can:

- ★ Move a number from memory into Register A (where math can happen)

Elsie can:

- ★ Move a number from memory into the A register (where math can happen)
- ★ Add a number to Register A

Elsie can:

- ★ Move a number from memory into the A register (where math can happen)
- ★ Add a number to the A register
- ★ Move the result back into memory

Elsie, like all digital computers, only understands ones and zeros

Elsie is really, really simple but

All our computers work very much like Elsie

So writing a program for Elsie shows us what programs really look like and how they run.

What does a program for Elsie look like?

```
# this is a line comment
# (first thing’s first!)
```

these are the only instructions
Elsie knows

```
load        # move data value into regA
add         # add data value to regA
store       # store contents of regA
goto        # this one we skip for now
```

```
# this program adds 1 + 2 and
# stores the result in
# memory location 10

load 1
add 2
sto 10
```

How does Elsie know what to do?

When Elsie powers up, the hardware starts reading 4 binary bits (a nybble) at a time.

Where do these nybbles come from?

Elsie has an Instruction Pointer, a register that knows where the next instruction is.

Let’s have a look at Elsie

Elsie, a 4-bit computer

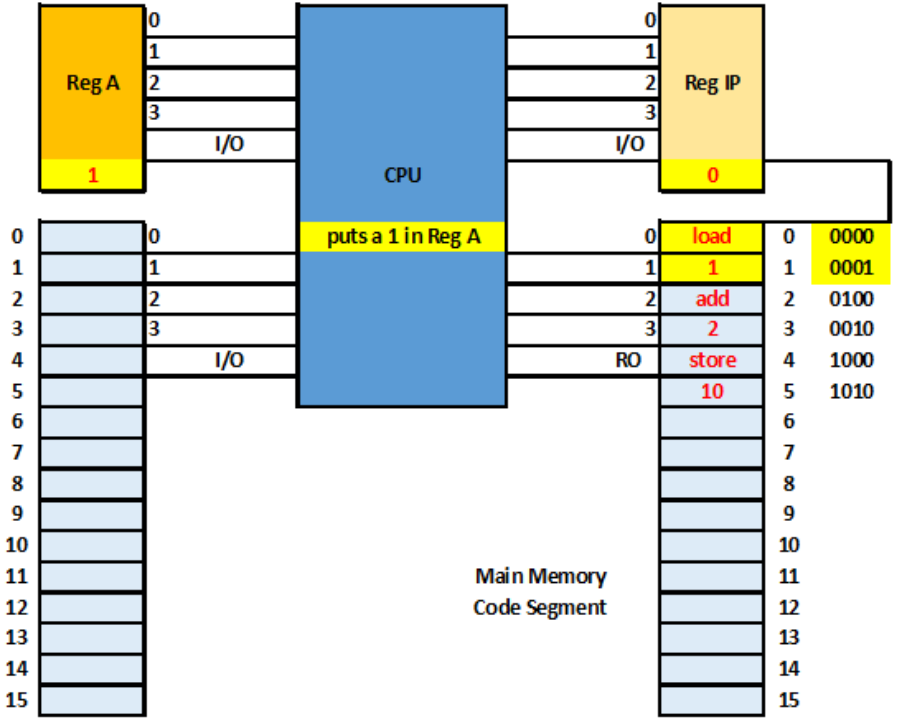
Addressable units are 4-bit nybbles
Reg A is for math (Rev 0 only adds)
Reg IP is instruction pointer, 0 on boot
Code segment is read only
Address and data lines are shared

Instructions

load	00
add	01
sto	10
goto	11

Mode

immediate	00
index	01



Back to the 4-bit nybbles Elsie’s been reading all this time

Elsie’s hardware endlessly feeds the CPU 4-bit nybbles in pairs

The first nybble is the instruction.

The second nybble is data for that instruction to use.

Elsie’s hardware

1)fetches a 4-bit instruction nybble,

Elsie’s hardware

1) fetches a 4-bit instruction nybble,
2) fetches a 4-bit data nybble,

Elsie’s hardware

1) fetches a 4-bit instruction nybble,
2) fetches a 4-bit data nybble,
3) executes the instruction using the data, and

Elsie’s hardware

1) fetches a 4-bit opcode nybble,
2) fetches a 4-bit data nybble,
3) executes the instruction using the data, and
4) repeats as long as power is on.

On Elsie, each instruction has a 2-bit ID called an opcode:

load = 00
add = 01
sto = 10
goto = 11

Each 4-bit instruction nybble also contains a 2-bit mode that we skip for now

opcode	**mode (always 00 for now)**

Back to our program..

```
# this program adds 1 + 2 and
# stores the result in
# memory location 10

load 1
add 2
sto 10
```

```
# this program is
# much easier to read
# with line comments!

load 1          # put a 1 in register A
add 2           # add 2 to register A
sto 10          # store the result
```

But Elsie only knows ones and zeros so we must translate

We know the 2-bit opcode for each instruction, and

We know the data for each instruction.

To Elsie, our program looks like this

load 1	0000 0001
add 2	0100 0010
sto 10	1000 1010

(This translation to ones and zeros is what compilers do.)

Elsie’s CPU is wired to execute each instruction.

Elsie, a 4-bit computer

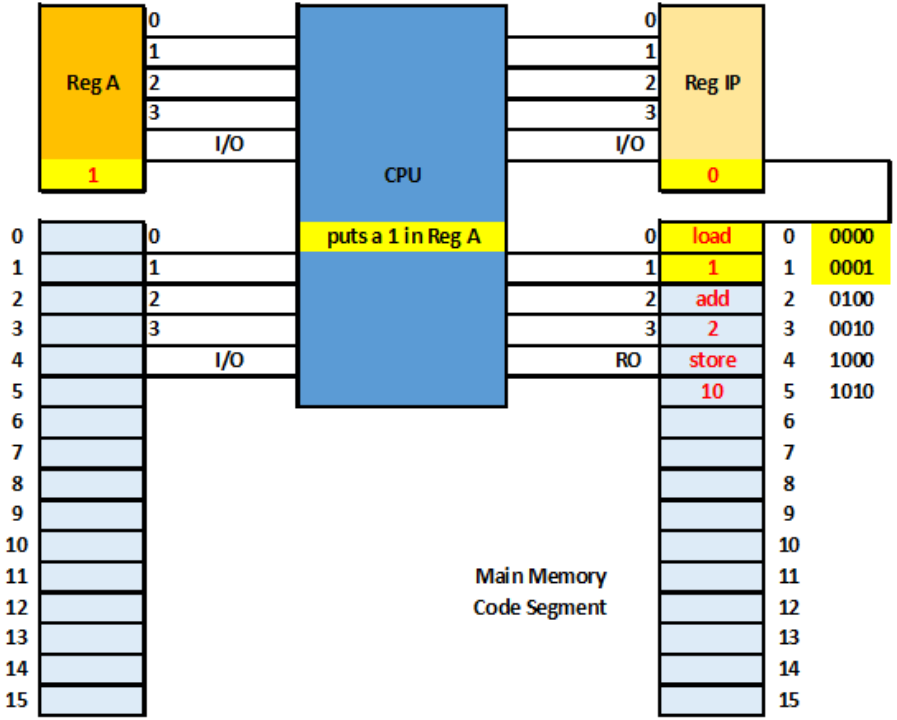
Addressable units are 4-bit nybbles
Reg A is for math (Rev 0 only adds)
Reg IP is instruction pointer, 0 on boot
Code segment is read only
Address and data lines are shared

Instructions

load	00
add	01
sto	10
goto	11

Mode

immediate	00
index	01



Elsie, a 4-bit computer

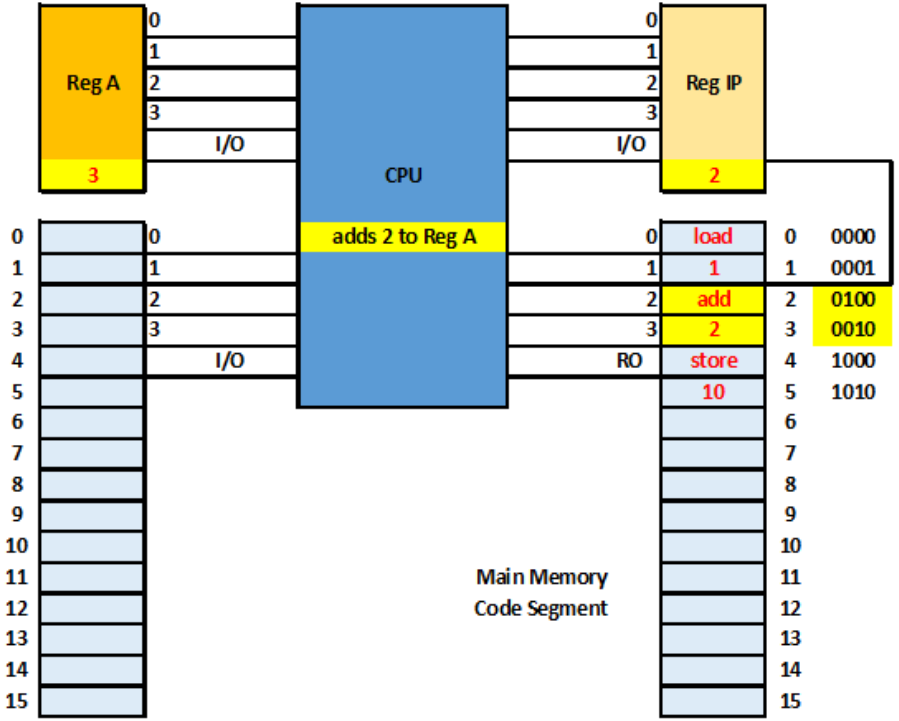
Addressable units are 4-bit nybbles
Reg A is for math (Rev 0 only adds)
Reg IP is instruction pointer, 0 on boot
Code segment is read only
Address and data lines are shared

Instructions

load	00
add	01
sto	10
goto	11

Mode

immediate	00
index	01



Elsie, a 4-bit computer

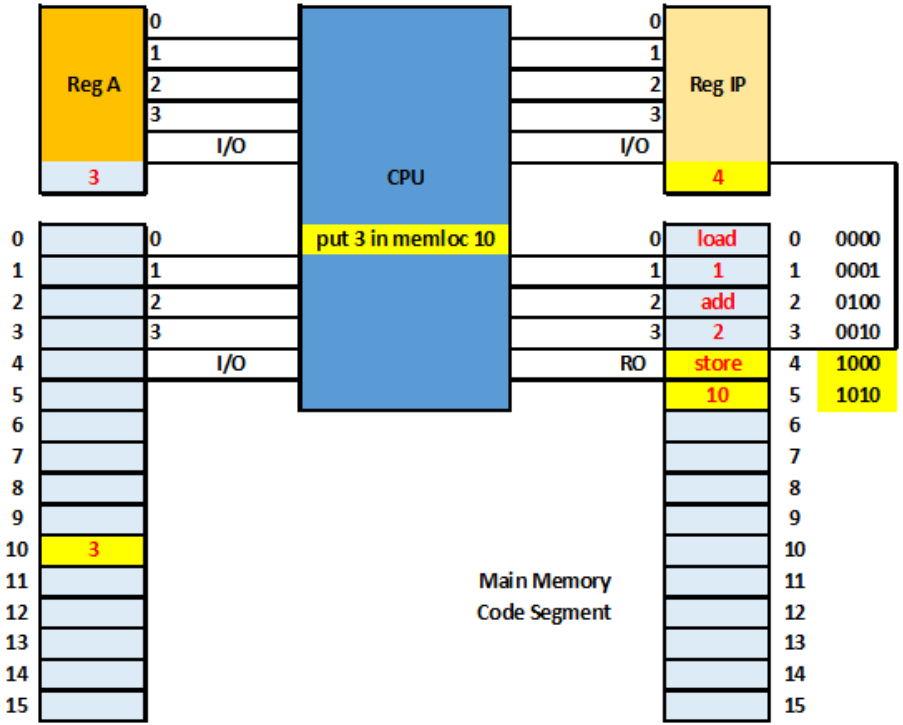
Addressable units are 4-bit nybbles
Reg A is for math (Rev 0 only adds)
Reg IP is instruction pointer, 0 on boot
Code segment is read only
Address and data lines are shared

Instructions

load	00
add	01
sto	10
goto	11

Mode

immediate	00
index	01



What do real computer instructions look like?

A similar program in C++:

like_Elsie.cpp | like_Elsie_1.cod

like_Elsie | (Global Scope) | main()

```cpp
// like_Elsie.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

int main()
{
    int a = 1 + 2;
}
```

90 % | No issues found | Ln: 6 | Ch: 1 | SPC | CRLF

Now the real computer instructions:

like_Elsie.cpp | like_Elsie_1.cod

```
00000 55        push    ebp
00001 8b ec         mov     ebp, esp
00003 81 ec e4 00 00
  00        sub     esp, 228        ; 000000e4H
00009 53        push    ebx
0000a 56        push    esi
0000b 57        push    edi
0000c 8d bd 1c ff ff
  ff        lea     edi, DWORD PTR [ebp-228]
00012 b9 39 00 00 00    mov     ecx, 57             ; 00000039H
00017 b8 cc cc cc cc    mov     eax, -858993460         ; ccccccccH
0001c f3 ab         rep stosd
0001e b9 00 00 00 00    mov     ecx, OFFSET __17995459_like_Elsie@cpp
00023 e8 00 00 00 00    call    @__CheckForDebuggerJustMyCode@4

; 6    :     int a, b, c;
; 7    :
; 8    :     b = 1;

00028 c7 45 ec 01 00
  00 00         mov     DWORD PTR _b$[ebp], 1

; 9    :     c = 2;

0002f c7 45 e0 02 00
  00 00         mov     DWORD PTR _c$[ebp], 2

; 10   :     a = 1 + 2;

00036 c7 45 f8 03 00
  00 00         mov     DWORD PTR _a$[ebp], 3
```

90 % No issues found Ln: 1 Ch: 1 MIXED CRLF

Not *too* different from Elsie's code, eh?

load 1	0000 0001
add 2	0100 0010
sto 10	1000 1010