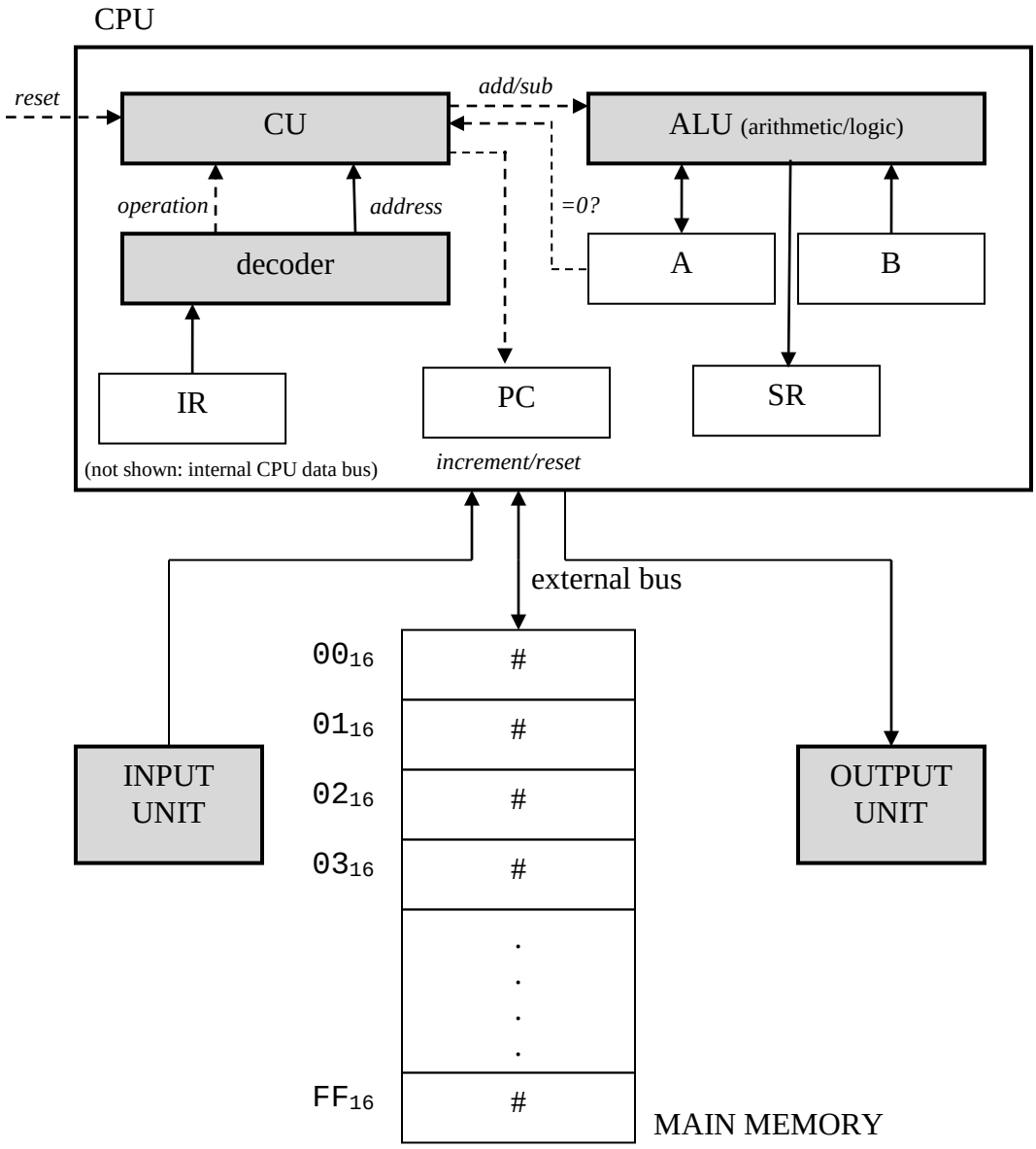
A Simple[1] Computer!

1. General Overview



Note: This computer works with 14-bit values.

[1] This computer is a "toy" example. We could actually build it, given a little knowledge of electronics, but its processing power is very limited. It does, however, illustrate the basic principles by which all digital computers work. In this description some details have been omitted, including various details of the internal CPU bus, the external control bus, and the implementation of the main memory subsystem.

2. CPU Organization

The CPU has 5 onboard registers. All contain 14-bit values, except the PC which contains an 8-bit value and the SR which contains a 3-bit value.

A and B	"operand registers"	these hold the CPU's current working values
IR	"instruction register"	this holds the most recently fetched program instruction
PC	"program counter"	this holds the address of the next program instruction
SR	"status register"	this holds the status results of the most recent operation in the ALU, can be checked for ALU errors

Although it is not shown in the previous diagram, data is transferred within the CPU by an internal CPU data bus. This allows data to be moved:

- from the external bus into A, and from A onto the external bus
- from the external bus into B
- from the external bus into IR
- from the "address" output of the decoder to various destinations, including into PC

The CU (Control Unit) is the "traffic cop" of the system:

- it commands the movement of values from one location to another within the CPU, and across the external bus
- it can command the ALU to perform an operation
- it can command the PC to increment itself, or to reset itself back to 0.

The CU's operation follows the "fetch-execute" cycle, described below. The CU's operation is guided by the most recently fetched program instruction, as stored in IR. This information is forwarded to the CU by the "decoder" unit. In some cases it is also guided by whether or not A contains 0.

The ALU (Arithmetic/Logic Unit) can perform various operations, listed below, on the values in A and B. The result is stored back in A.

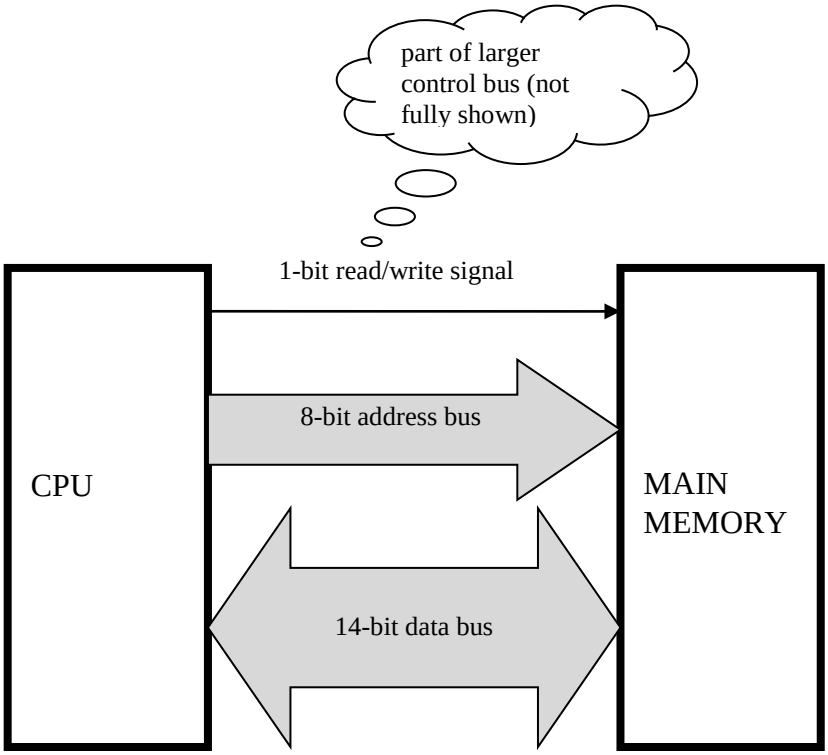
The SR (Status Register) stored information on the results given by the previous operation completed by the ALU. The SR has three bits that are given as NZV with the bits storing the following information:

N	The N bit has a value of 1 if the result of the operation is **n**egative; 0 otherwise.
Z	The Z bit has a value of 1 if the result of the operation is **z**ero, 0 otherwise.
V	The V bit has a value of 1 if the result of the operation caused an o**v**erflow, 0 otherwise.

Note: The N bit is the zero(0) bit, the Z bit is the one(1) bit, and the V bit is the two(2) bit of the SR (Status Register).

3. Main Memory Organization

Main memory consists of 256 locations. Each contains a 14-bit value. The CPU can individually read or write the contents of any location, using that location's address.



When the CPU needs to read a value, it places an 8-bit address on the address bus[2], and then signals a "read". Main memory responds by placing the corresponding 14-bit value on the data bus for the CPU.

When the CPU needs to write a value, it places both an 8-bit address on the address bus *and* a 14-bit value on the data bus, and then signals a "write". Main memory responds by storing that value at the given address.

4. I/O Unit Organization

The input and output units represent connections to the outside world, via input/output devices. The CPU can read a value from the outside world via the input unit, and can write a value to the outside world via the output unit.

The connections between the CPU and each I/O unit are similar to those needed for memory transfers.

[2] A bus is a data interconnection mechanism between functional units. It is often implemented as a set of parallel writes, or traces on a circuit board, with one line per bit. A bus writer signals a 0 or 1 on each line using discrete voltage levels (e.g. +5V for 1 and +0V for 0). Serial busses are also common.

5. CPU Instruction Set

The CPU provides the following 24 instruction types:

op-code	**operation (mnemonic)**	**mode**	**operand**	**Description**
0	READ			reads a value from the input unit into register A
1	WRITE			writes the value from register A to the output unit
2	LOAD	*A(0)*	*address*	reads a value from the specified memory address into register A
2	LOAD	*B(1)*	*address*	reads a value from the specified memory address into register B
3	LOADMEM		*address*	loads the value from memory location memory[address + B] to register A
4	STORE	*A(0)*	*address*	writes the value from register A to the specified memory address
4	STORE	*B(1)*	*address*	writes the value from register B to the specified memory address
5	STOREMEM		*address*	stores the value from register A to memory location memory[address + B]
6	ADD	*REG(0)*		adds the values in A and B, storing the result back in A
6	ADD	*MEM(1)*	*address*	adds the values in A and memory[address + B], storing the result back in A
7	ADDI	*A(0)*	*#*	adds the value in A with the operand, storing the result back in A
7	ADDI	*B(1)*	*#*	adds the value in B with the operand, storing the result back in B
8	SUB	*REG(0)*		subtracts the value in B from A, storing the result back in A
8	SUB	*MEM(1)*	*address*	subtracts the value in memory[address + B] from A, storing the result back in A
9	SUBI	*A(0)*	*#*	subtracts the operand from A, storing the result back in A
9	SUBI	*B(1)*	*#*	subtracts the operand from B, storing the result back in B
A	MUL	*REG(0)*		multiplies the values in A and

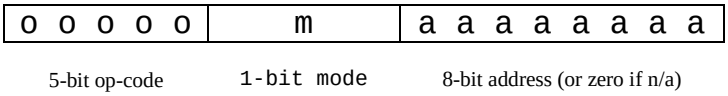
				B, storing the result back in A
A	MUL	*MEM(1)*	*address*	multiplies the values in A and memory[address + B], storing the result back in A
B	MULI	*A(0)*	*#*	multiplies the value in A by the operand, storing the result back in A
B	MULI	*B(1)*	*#*	multiplies the value in B by the operand, storing the result back in B
C	DIV	*REG(0)*		divides the value in A by B, storing the result back in A
C	DIV	*MEM(1)*	*address*	divides the value in A by memory[address + B], storing the result back in A
D	DIVI	*A(0)*	*#*	divides the value in A by the operand, storing the result back in A
D	DIVI	*B(1)*	*#*	divides the value in B by the operand, storing the result back in B
E	MOD	*REG(0)*		remainder from dividing the value in A by B, storing the result back in A
E	MOD	*MEM(1)*	*address*	remainder from dividing the value in A by memory[address + B], storing the result back in A
F	MODI	*A(0)*	*#*	remainder from dividing the value in B by the operand, storing the result back in B
F	MODI	*B(1)*	*#*	remainder from dividing the value in A by the operand, storing the result back in A
10	AND			logical bitwise and the values in A and B, storing the result back in A
11	OR			logical bitwise or the values in A and B, storing the result back in A
12	NOT	*A(0)*		logical bitwise the value in A, storing the result back in A
12	NOT	*B(1)*		logical bitwise the value in B, storing the result back in B
13	TESTSTATUS		*0-2*	Tests the bit in the Status Register (SR) specified by the operand
14	CLEAR	*A(0)*		set A to 0
14	CLEAR	*B(1)*		set B to 0

15	JUMPIFNOTZERO	*address*	loads PC with the specified address, if A does not contain 0
16	JUMPIFZERO	*address*	loads PC with the specified address, if A contains 0
17	JUMP	*address*	loads PC with the specified address
18	HALT		halts the CPU, no further instructions will be executed (until reset)

6. Machine Language

A machine language program consists of a sequence of 14-bit instructions, stored in consecutive memory locations (recall: each memory location is 14-bits wide).

Each instruction has the following format:



E.g. ADD REG would be encoded as:

```
00 1100 0000 0000      (in binary)
 0    C    0    0      (in hex)
```

E.g. The instruction to load B with the contents of memory at address 10 (decimal) would be encoded as:

```
00 0101 0000 1010      (in binary)
 0    5    0    A      (in hex)
```

7. The Fetch-Execute Cycle

The CU is a little machine which, on reset, begins executing the following *micro*-instruction sequence:

```
reset PC                                        (PC can reset itself to 0)
do
    IR ← memory[PC]
    increment PC                                (PC can increment itself)
    case operation of
        READ:           A ← input
        WRITE:          output ← A
        LOAD:           A/B ← memory[address]
        LOADMEM:        A ← memory[address + B]
        STORE:          memory[address] ← A/B
        STOREMEM:       memory[address + B] ← A
        ADD:            A ← A + (B or memory[address + B])
        ADDI:           A/B ← A/B + #
        SUB:            A ← A – (B or #)
        SUB:            A ← A – (B or #)
        MUL:            A ← A * (B or memory[address + B]))
        MULI:           A/B ← A/B * #
        DIV:            A ← A ÷ (B or memory[address + B]))
        DIVI:           A/B ← A/B ÷ #
        MOD:            A ← A % (B or memory[address + B]))
        MODI:           A/B ← A/B % #
        AND:            A ← A & B
        OR:             A ← A | B
        NOT:            A/B ← ~A/B
        TESTSTATUS:     A ← SR[#]
        CLEARA:         A ← 0
        CLEARB:         B ← 0
        JUMPIFNOTZERO:  PC ← address, if A != 0
        JUMPIFZERO:     PC ← address, if A = 0
        JUMP:           PC ← address
while operation ≠ HALT
```

Above, “address” and “#” refer to the value stored in the lowest 8 bits of the IR. As noted above in the descriptions for the operations “#” may have limitations dependant on the operation being performed.

The operations ADD, ADDI, SUB, SUBI, MUL, MULI, DIV, DIVI, MOD, MODI, AND, OR, and NOT are all performed by the arithmetic logic unit (ALU).

Note: on reset, main memory is assumed to already contain a machine language program, starting at address 0.

Exercises:

1. What is the largest unsigned integer this computer can process?

2. Why is the PC register 8-bits wide? Hint: think about the range of main memory addresses.

   Similarly, why is the 14-bit instruction format divided into a 5-bit region, 1-bit region, and an 8-bit region, i.e. why those region widths specifically?

3. Consider the connections between the CPU and the I/O units.

   a) The notes above state that the connections between the CPU and an I/O unit are similar to the connections between the CPU and main memory. Specifically, what connections are required for each I/O unit?
   b) Is it necessary to have separate busses for each unit the CPU is connected to? Or, is there a way to eliminate redundancy? Explain your answer.

4. Translate each of the following assembly instructions to a corresponding machine language instruction. Show each final answer in 14-bit binary, and also summarized in 4-digit hex.

   a) WRITE
   b) LOAD B 47_{10} ← load register B with the contents of memory location 47
   c) STORE A 47_{10} ← store the contents of register A at memory location 47
   d) SUB REG
   e) HALT

5. In English, summarize the effects of executing each of the following machine instructions (summarized in hex):

   a) `0520`
   b) `0FC9`
   c) `0000`
   d) `3400` ← trick question?

6. Assume the simple computer has just been reset. Also assume the first few memory locations contain the following sequence of 14-bit numbers (shown in hex). By tracing the fetch-execute cycle as precisely as possible, hand-execute this machine language program! Maintain a picture of main memory *and* the CPU registers. Show the final output.

Address	Contents
0:	0406
1:	0505
2:	1000
3:	0200
4:	3000
5:	0002
6:	000B

Hint: you'll have to hand-decode each instruction, once you've fetched it into IR, to figure out what it is. There are 5 instructions (the other two values are data).

7. Consider the following algorithm, expressed in pseudo-code:

```
set num = 10

while num ≠ 0
        write num
        decrement num
```

To express this in a form the computer can execute, we first perform a translation to an intermediate representation called "assembly language"[3]. Each line corresponds to one machine instruction. The instructions are assumed to be stored in memory at addresses 0 through 6, with data at addresses 7 and 8.

```
        LOAD A          ten
        LOAD B          one
loop    JUMPIFZERO      done
        WRITE
        SUB REG
        JUMP            loop
done    HALT
ten     10₁₀
one     1
```

Complete the translation to machine language (written in hex) by filling in the blanks below:

0407
0508
2C06
0001

8. Express the program from question 6 in (a) assembly language, and then (b) high-level pseudo-code.

[3] Assembly language is really just machine language written in a slightly more human-readable form. Instead of writing an op-code numerically, we use a self-descriptive "mnemonic". Instead of writing an address numerically, we give it a self-documenting name.

9. The following algorithm computes the sum of all whole numbers between 0 and the given non-negative whole number:

```
set sum = 0
read num

while num ≠ 0
        set sum = sum + num
        set num = num – 1

write sum
```

   Translate this algorithm to this simple computer's machine language, via assembly language.

10. Trace the execution of the simple computer for the machine language program from question 7. Assume an input value of 3.

11. Using assembly language notation, how would you code the following on this simple computer?

```
read x
read y

if x = 0
        increment y

write y
```

12. How about if the "if" statement above is changed to the following?

```
if x = 0
        increment y
else
        decrement y
```

13. We're learning how a simple computer works. But wait! The CU follows an algorithm, just like the main computer does. Can we build a computer out of a computer? Wouldn't the CU need its own internal CU? Where would it end?

    Explain why this isn't a problem.