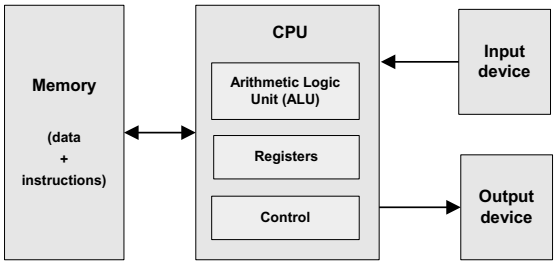
CSCE 312: Computer Organization

David Kebo Houngninou

Processor Architecture

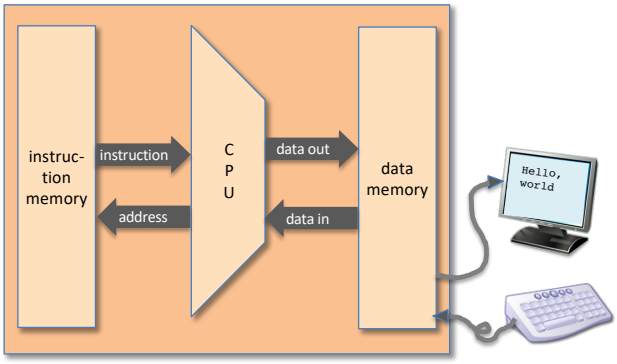
Computer operation at a high level



In a stored program computer, operations at a high level require two steps:

1. Fetch the instruction from the instruction memory
2. Execute the instruction

Hack Computer



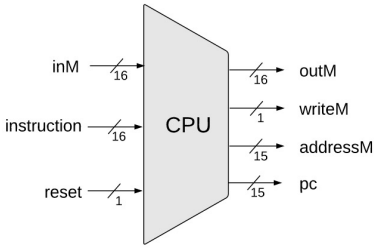
Abstraction:

A computer capable of running programs written in the Hack machine language

Implementation:

Built from the Hack chip-set.

Hack CPU



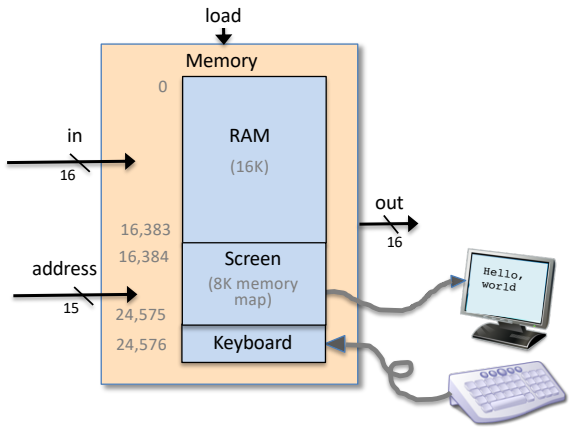
CPU abstraction:

Executes a Hack instruction and figures out which instruction to execute next

CPU Implementation:

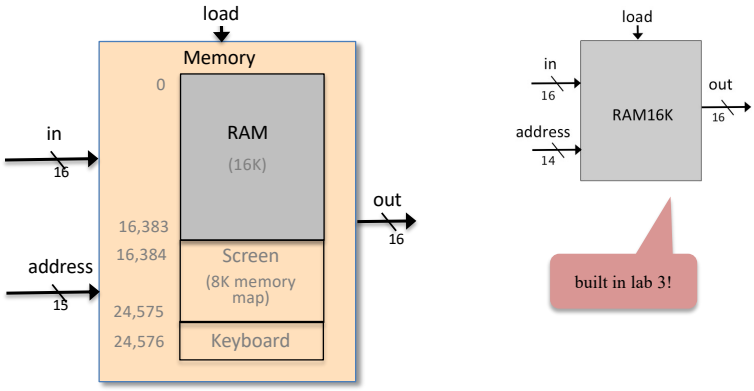
To be discussed

Data memory: abstraction



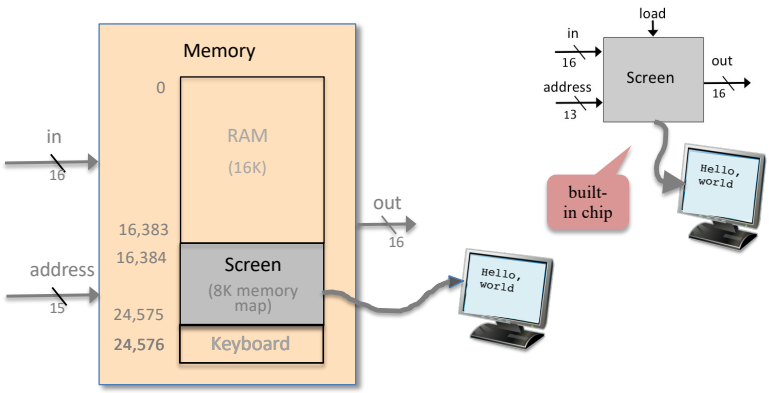
- Addressable size: 32K 16-bit words
- Address 0 to 16383: data memory
- Address 16384 to 24575: screen memory map
- Address 24576: keyboard memory map

RAM



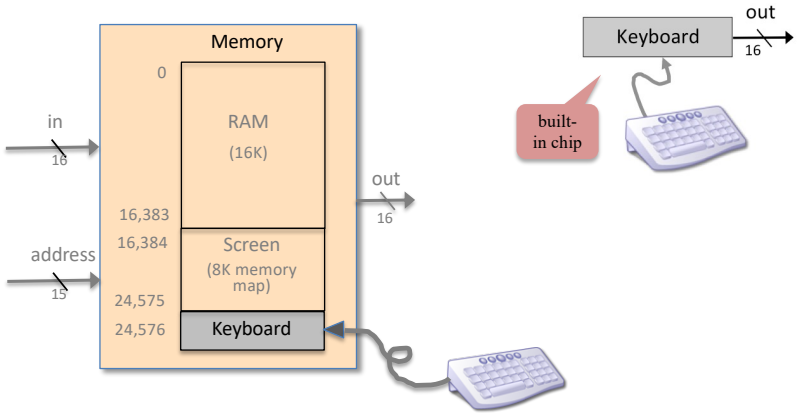
The Hack RAM is built by the RAM16K chip implemented in lab 3.

Screen



- The Hack screen is realized by a built-in chip named Screen
- Screen: a regular RAM + display output side-effect.

Keyboard



- Realized by a built-in chip named Keyboard
- Keyboard: A read-only 16-bit register + a keyboard input side-effect.

The Hack character set

key	code
(space)	32
!	33
"	34
#	35
$	36
%	37
&	38
'	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

key	code
0	48
1	49
...	...
9	57

:	58
;	59
<	60
=	61
>	62
?	63
@	64

key	code
A	65
B	66
C	...
...	...
Z	90

[	91
/	92
]	93
^	94
_	95
`	96

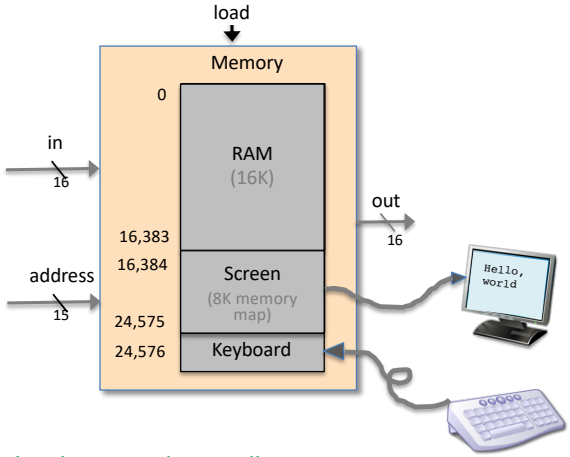
key	code
a	97
b	98
c	99
...	...
z	122

{	123
|	124
}	125
~	126

key	code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
...	...
f12	152

Credit: www.nand2tetris.org

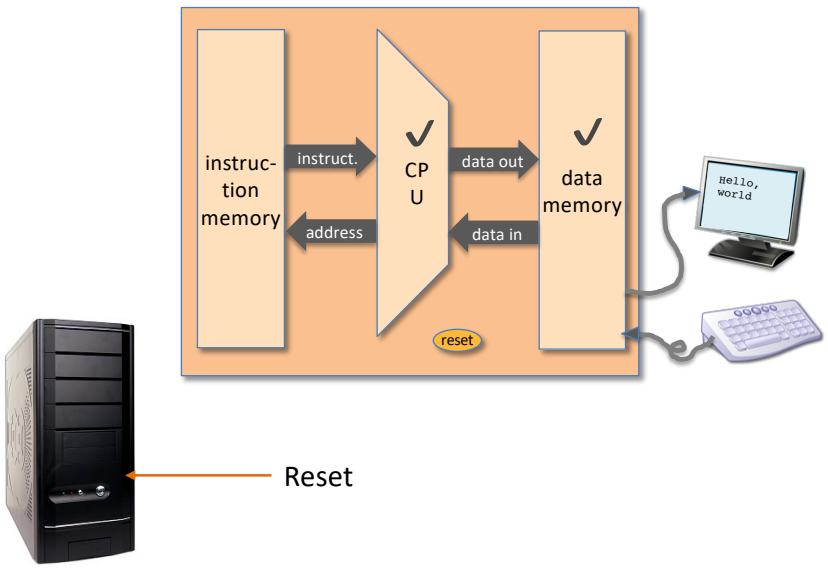
Memory implementation



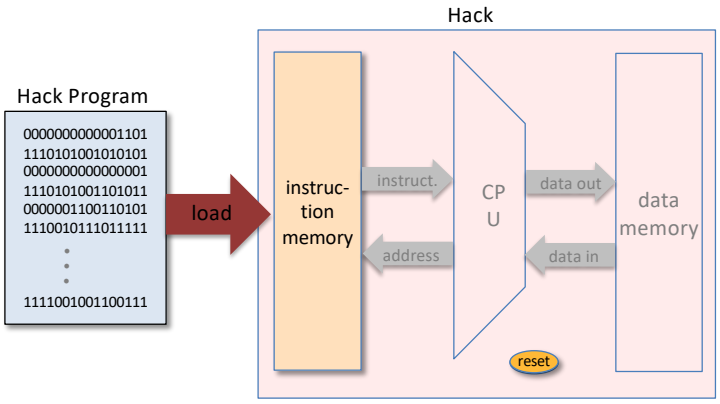
Implementation outline:

- Uses the three chip-parts RAM16K, Screen, and Keyboard (as just described)
- Routes the address input to the correct address input of the relevant chip-part.

Hack Computer



Instruction memory



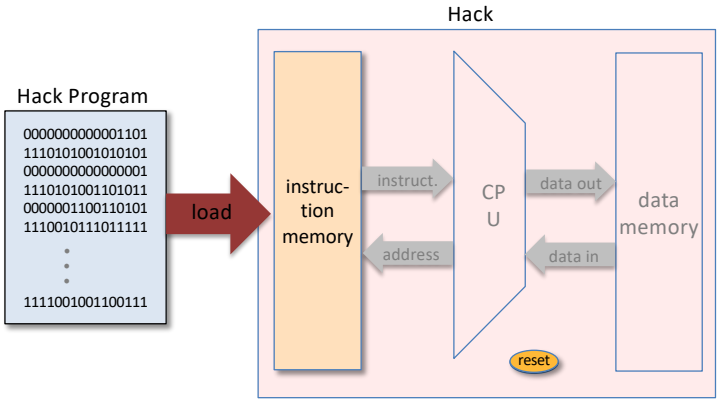
Load a program into the Instruction Memory?

How?

To run a program on the Hack computer:

1. Load the program into the Instruction Memory
2. Press "reset"
3. The program starts running.

Instruction memory



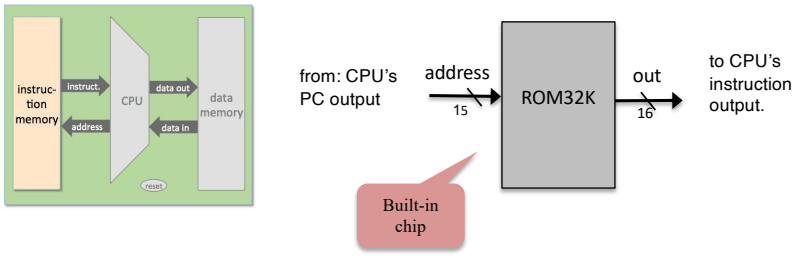
Loading a program into the Instruction Memory:

Hardware implementation:

plug-and-play ROM chips (each comes pre-loaded with a program's code)

Hardware simulation: programs are stored in text files; the simulator's software features a load-program service.

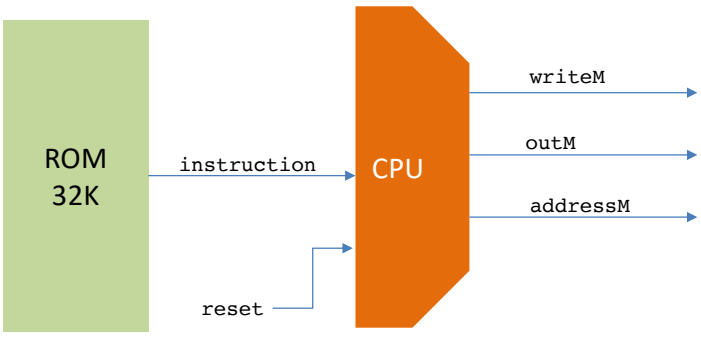
Instruction memory



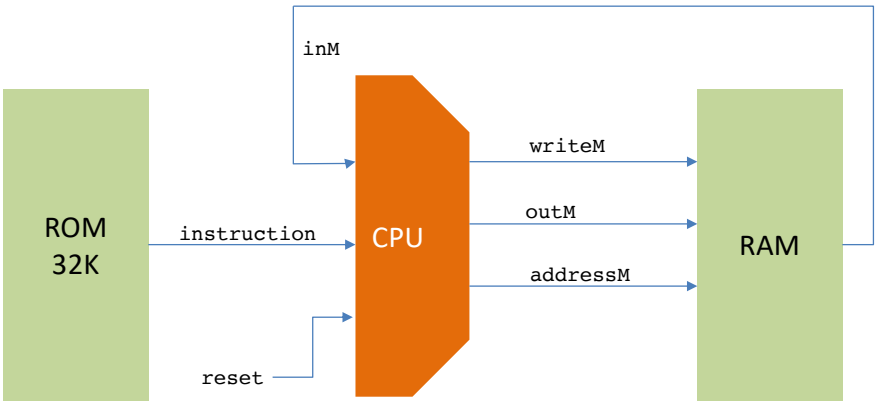
The Hack Instruction Memory is realized by a built-in chip named ROM32K

ROM32K: a read-only, 16-bit, 32K RAM chip + program loading side-effect.

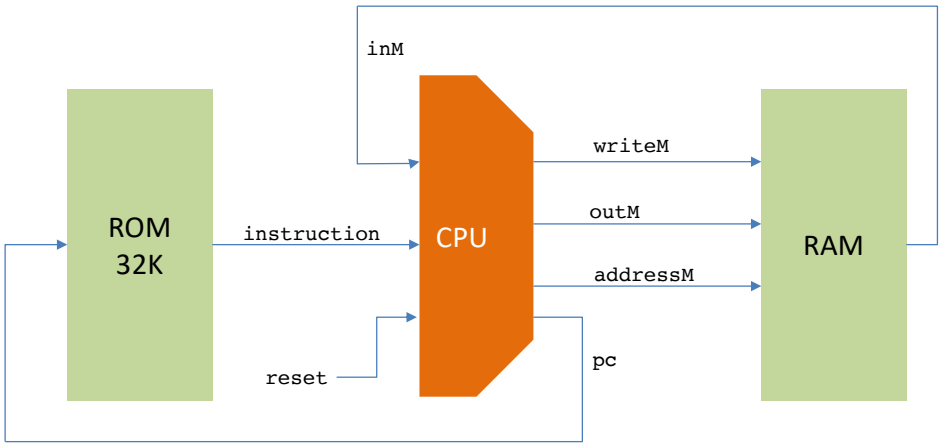
Hack Computer implementation



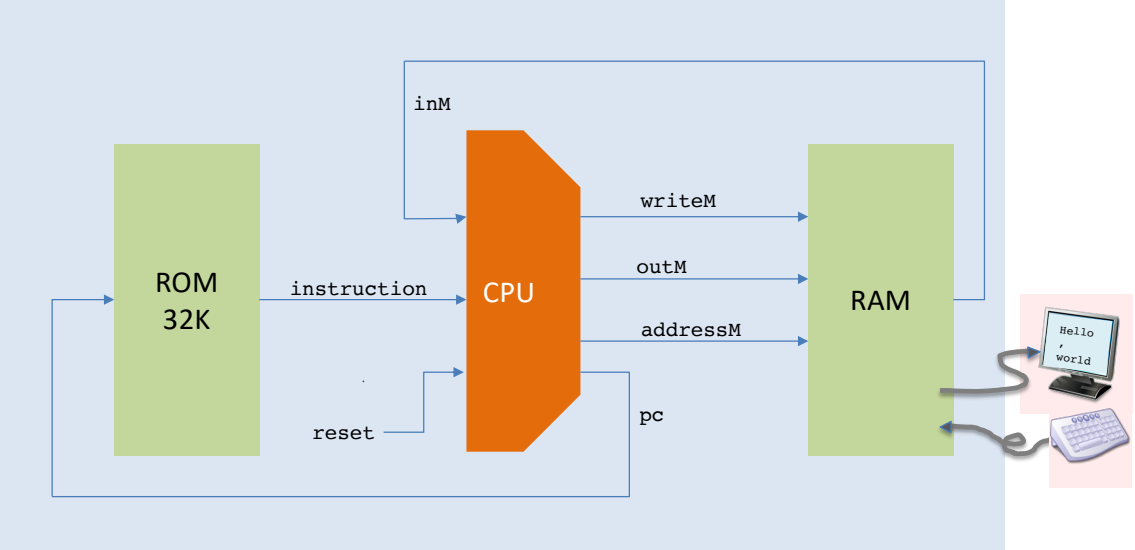
Hack Computer implementation



Hack Computer implementation

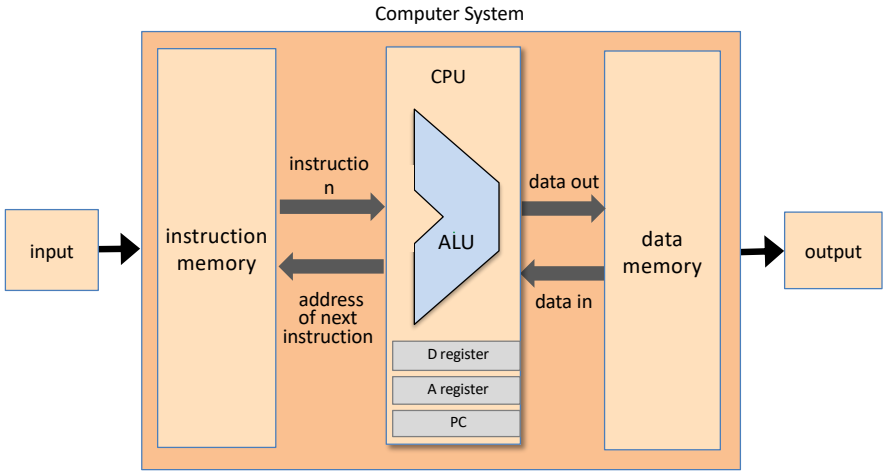


Hack Computer implementation



The HACK CPU

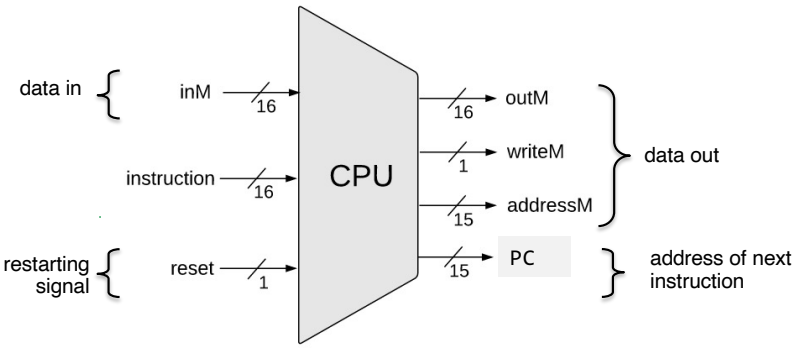
Hack computer



The Hack CPU is a 16-bit processor, designed to:

1. Execute the current instruction: dataOut = instruction(dataIn)
2. Figure out which instruction to execute next.

Hack CPU Interface



Hack language: C instructions

Symbolic syntax: *dest* = *comp* ; *jump*

Binary syntax: 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a=0	a=1						

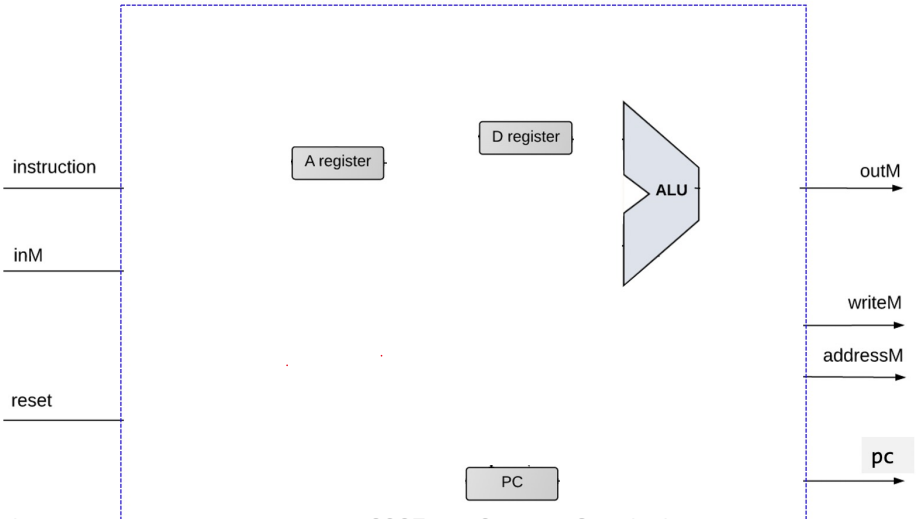
dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a=0	a=1						

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

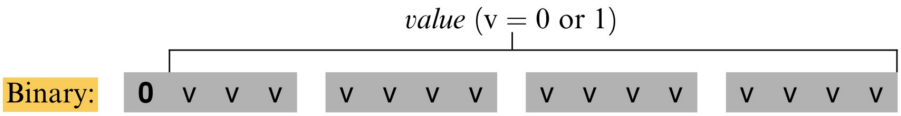
jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump



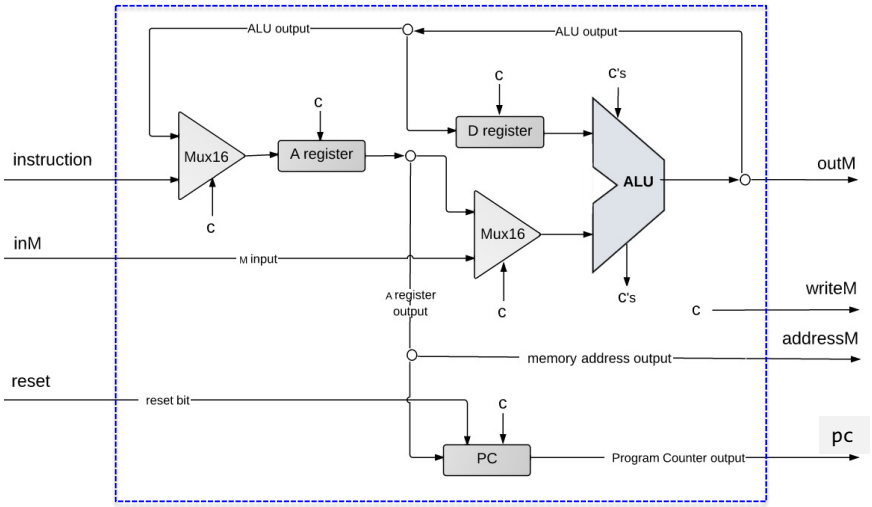
Hack language: A instructions

The *A*-instruction is used to set the A register to a 15-bit value:

A-instruction: *@value* // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.

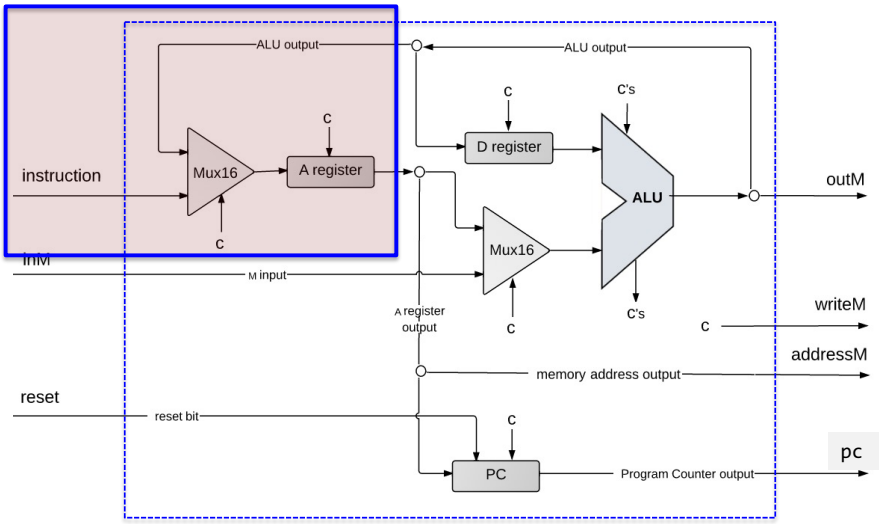


Hack CPU Implementation

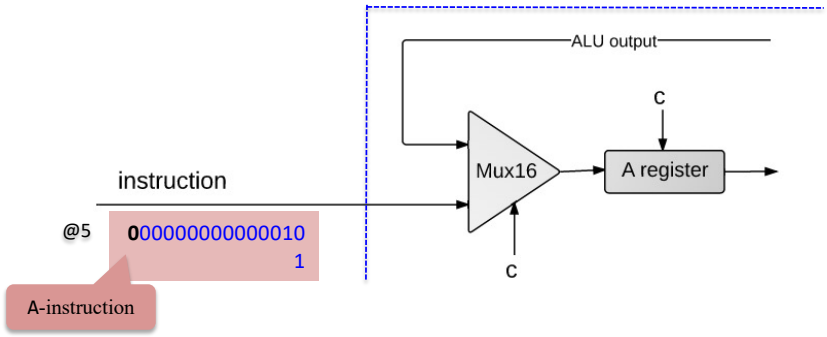


(each "c" symbol represents a control bit)

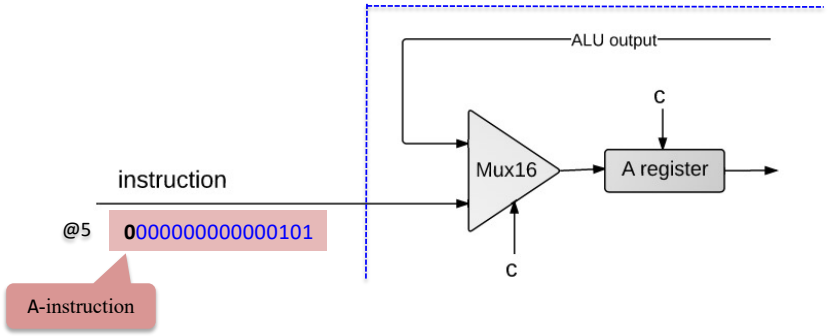
CPU operation: instruction handling



CPU operation: instruction handling



CPU operation: handling A-instructions



CPU handling of an A-instruction:

Decodes the instruction into:

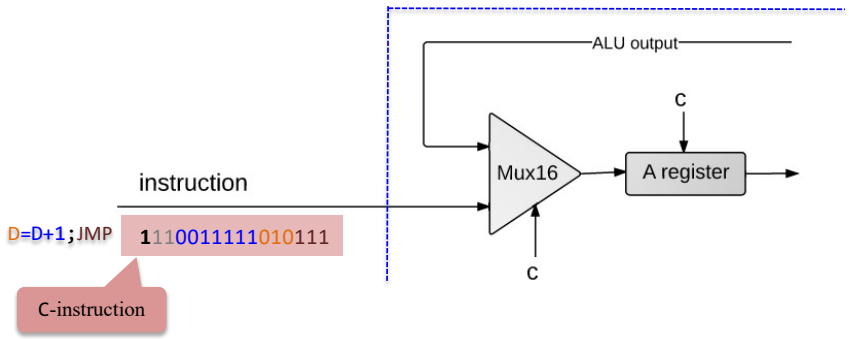
op-code

15-bit value

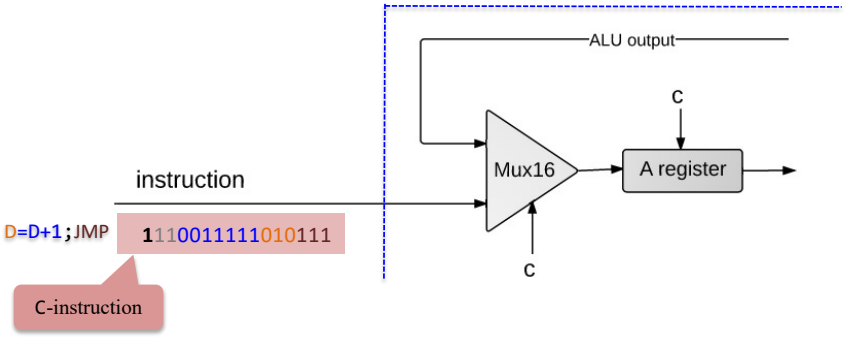
Stores the value in the A-register

Outputs the value (not shown in this diagram).

CPU operation: instruction handling



CPU operation: handling C-instructions



CPU handling of a C-instruction:

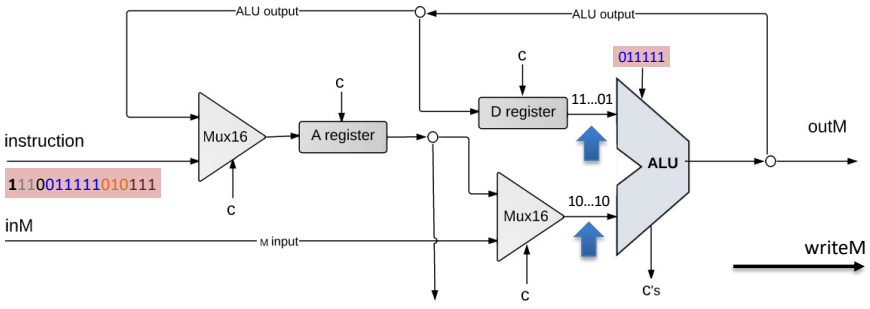
Decodes the instruction bits into:

- Op-code
- ALU control bits
- Destination load bits
- Jump bits

Routes these bits to their chip-part destinations

The chip-parts (most notably, the ALU) execute the instruction.

CPU operation: handling C-instructions



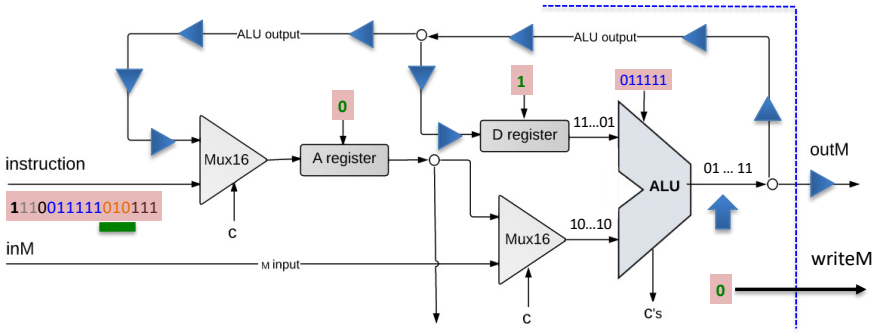
ALU data inputs:

- Input 1: from the D-register
- Input 2: from either:
  - A-register, or
  - data memory

ALU control inputs:

- control bits (from the instruction)

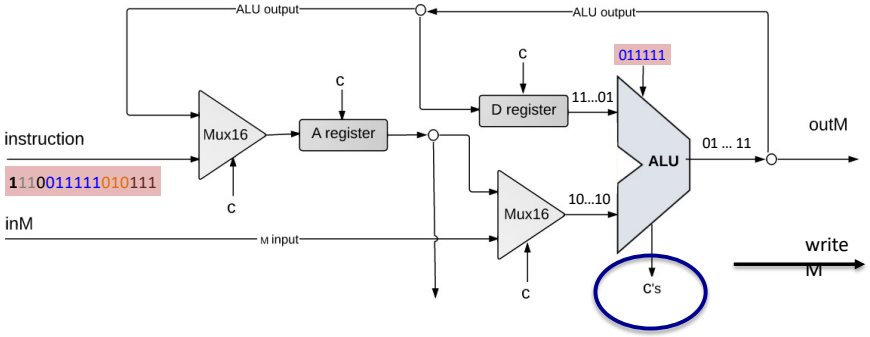
CPU operation: handling C-instructions



ALU data output

- Result of ALU calculation
- Fed simultaneously to: D-register, A-register, data memory
- Which destination actually commits to the ALU output is determined by the instruction's destination bits.

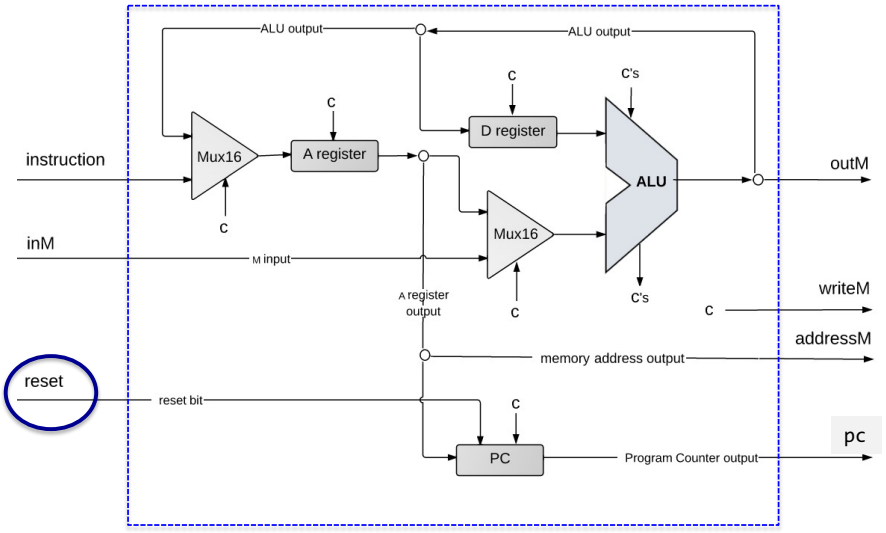
CPU operation: handling C-instructions



ALU control outputs:

- is the output negative?
- is the output zero?

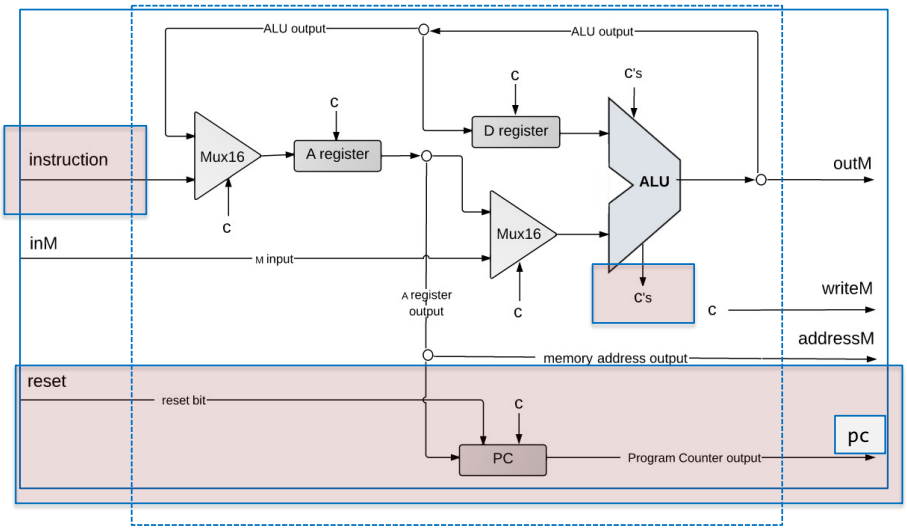
CPU operation: control



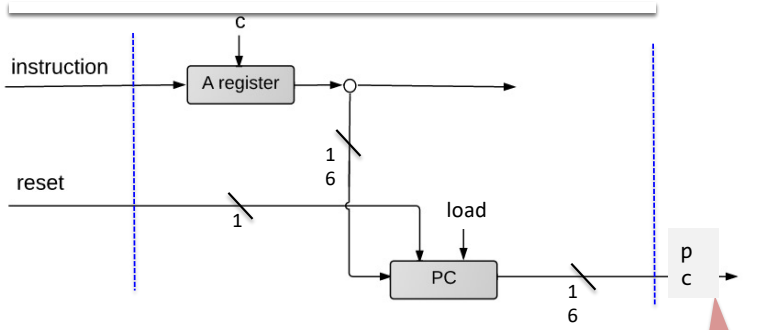
CPU operation: control

- The computer is loaded with some program;
- Pushing r**eset** causes the program to start running.

CPU operation: control



CPU operation: control



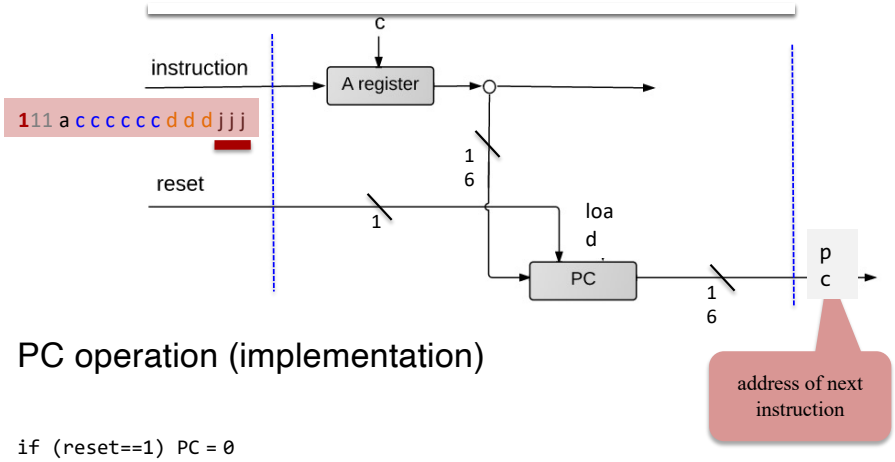
PC operation (abstraction)

Emits the address of the next instruction:

address of next instruction

- restart: `PC = 0`
- no jump: `PC++`
- goto: `PC=A`
- conditional goto: if (*condition*) `PC = A else PC++`

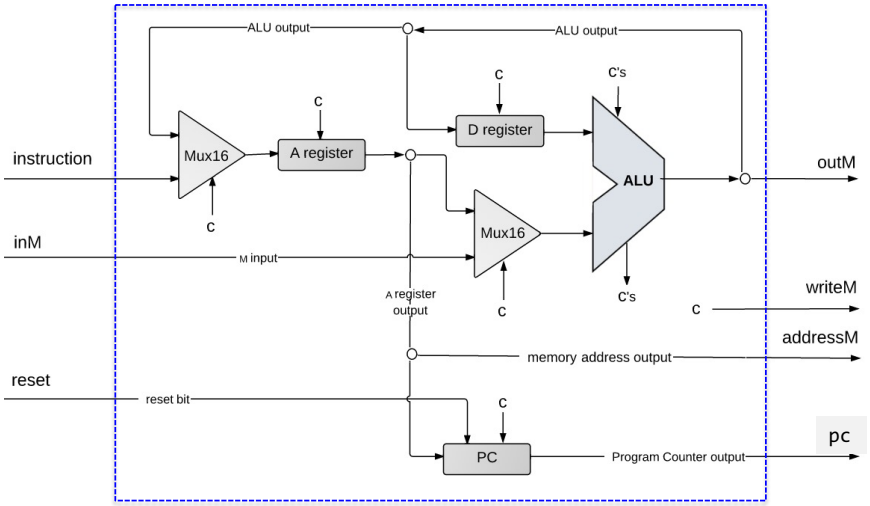
CPU operation: control



PC operation (implementation)

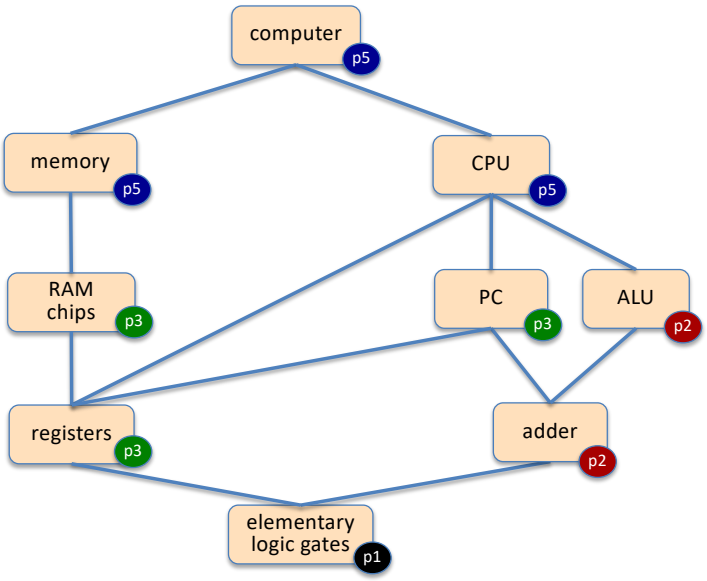
```
if (reset==1) PC = 0
else
    // in the course of handling the current instruction:
    load = f (jump bits, ALU control outputs)
    if (load == 1)  PC = A  // jump
    else  PC++  // next instruction
```


Hack CPU Implementation

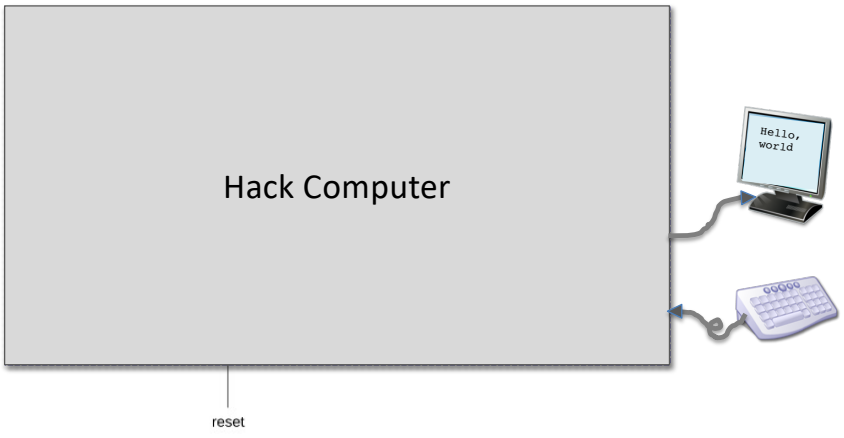


That's It!

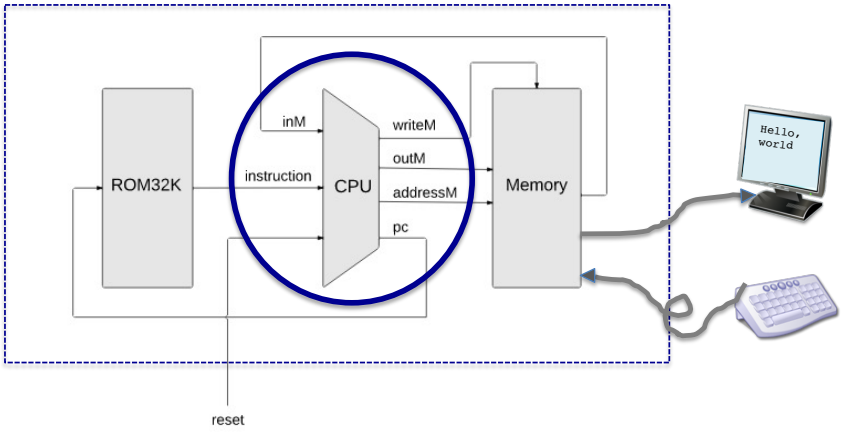
Nand2Tetris labs



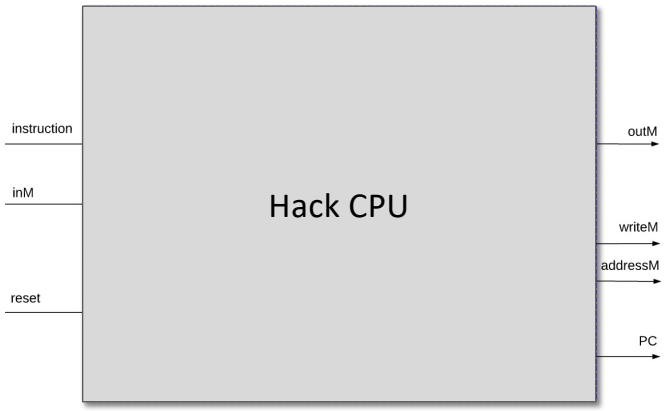
Abstraction



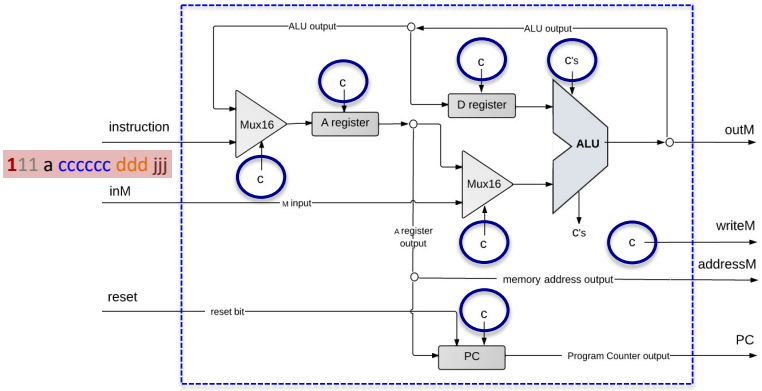
Implementation



CPU Abstraction



CPU Implementation



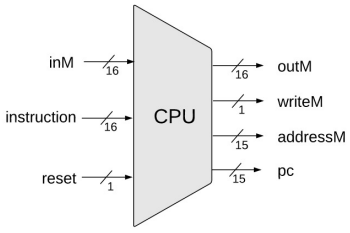
Implementation tips:

- Chip-parts: Mux16, ARegister, DRegister, PC, ALU, …
- Control: use HDL subscripting to parse and route the instruction bits to the control bits of the relevant chip-parts.

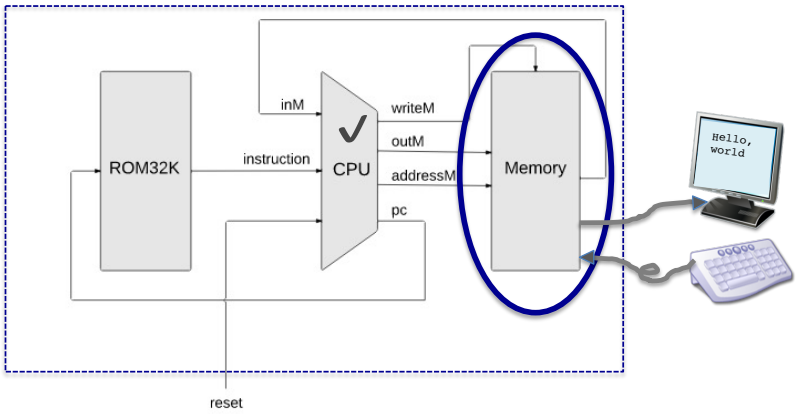
CPU Implementation

CPU.hdl

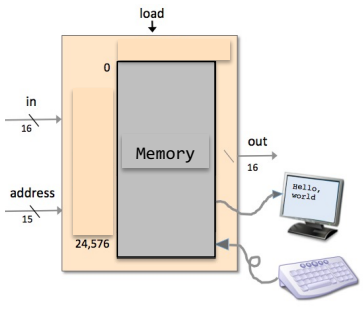
```
/**
 * The Central Processing unit (CPU).
 * Consists of an ALU and a set of registers,
 * designed to fetch and execute instructions
 * written in the Hack machine language.
*/
CHIP CPU {

   IN
      inM[16],         // value of M = RAM[A]
      instruction[16], // Instruction for execution
      reset;           // Signals whether to re-start the current program
                       // (reset == 1) or continue executing the current
                       // program (reset == 0).

    OUT
       outM[16]        // value to write into M = RAM[A]
       writeM,         // Write into M?
       addressM[15],   // RAM address (of M)
       pc[15];         // ROM address (of next instruction)

    PARTS:
    // Put you code here:
}
```



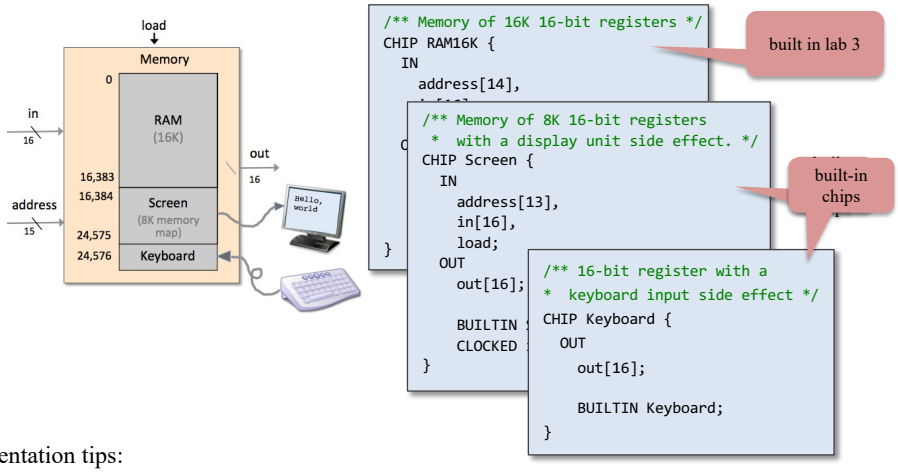
Hack Computer implementation



Memory implementation



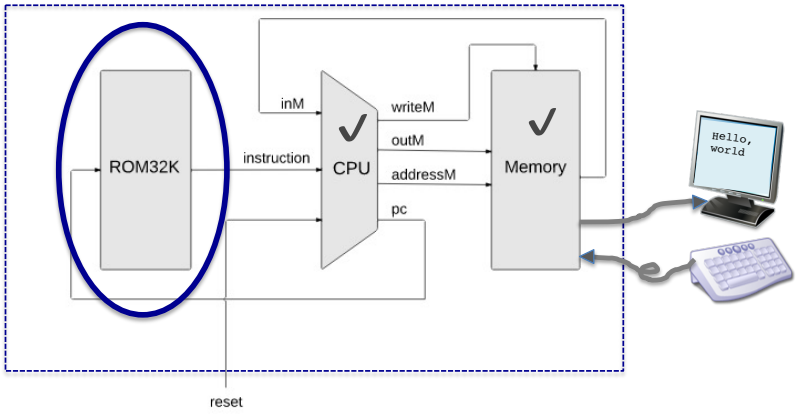
Memory implementation



Implementation tips:

- Use the three chip-parts:
  `RAM16K`, `Screen`, and `Keyboard`
- Route the `address` input to the correct `address` input of the relevant chip-part.

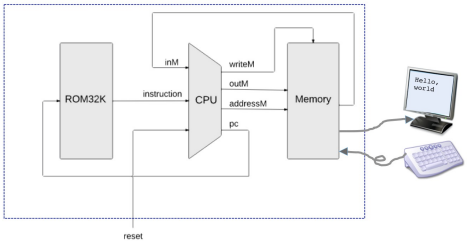
Hack Computer implementation



Implementation tip:

Use the built-in ROM32K chip.

Hack Computer implementation



Computer.hdl

```
/**
 * The HACK computer, including CPU, ROM and RAM.
 * When reset is 0, the program stored in the computer's ROM executes.
 * When reset is 1, the execution of the program restarts.
*/

CHIP Computer {

    IN reset;

    PARTS:
    // Put your code here.
}
```


Best practice advice

1. Try to implement the chips in the given order
2. Use as few chip-parts as possible
3. Use chips that you've implemented in previous labs
4. The best practice is to use their built-in versions.