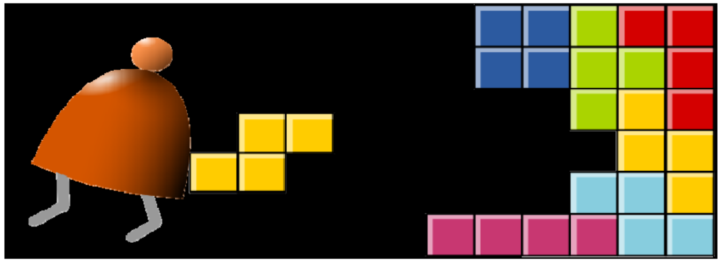
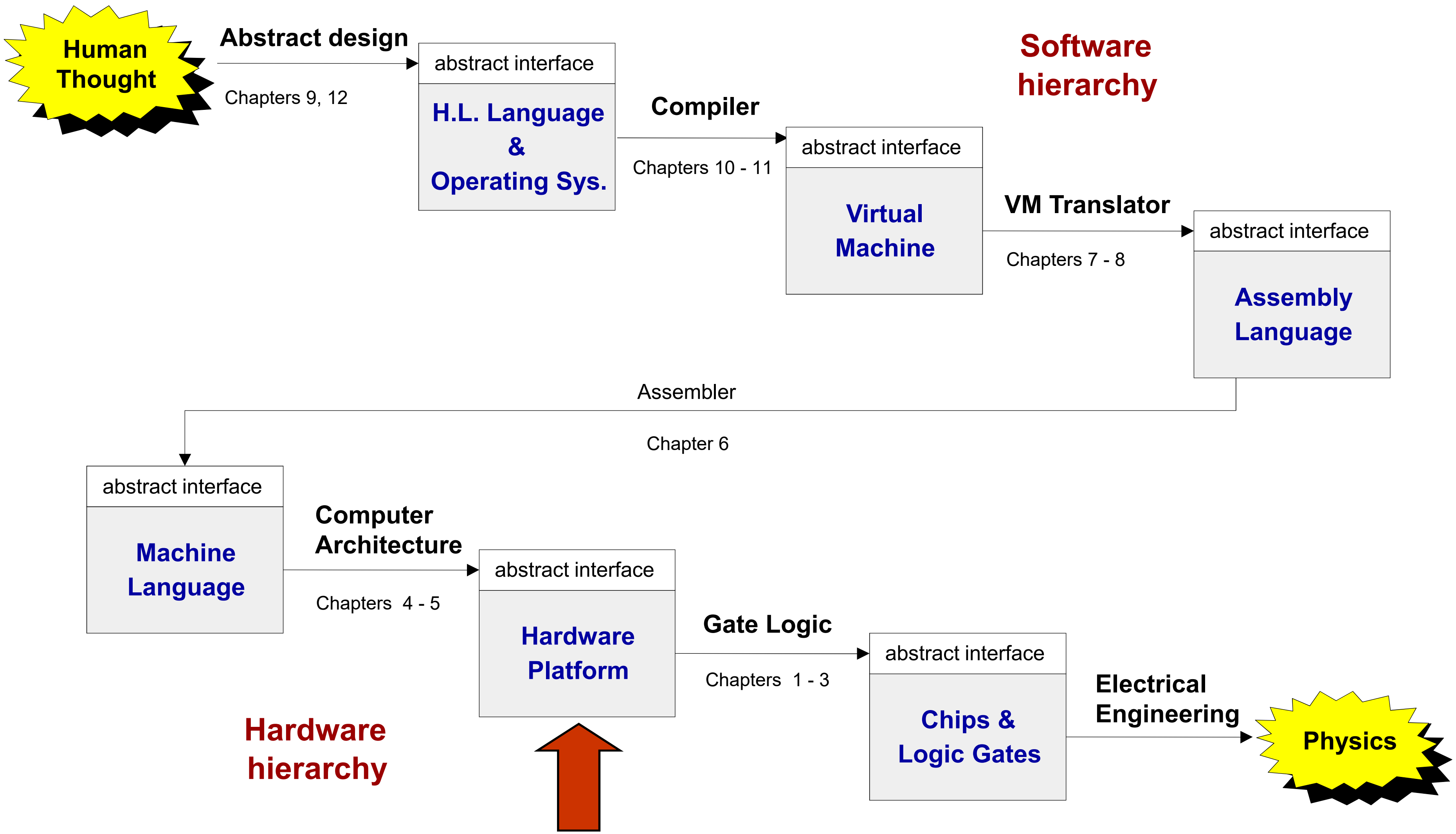
Computer Architecture



Building a Modern Computer From First Principles

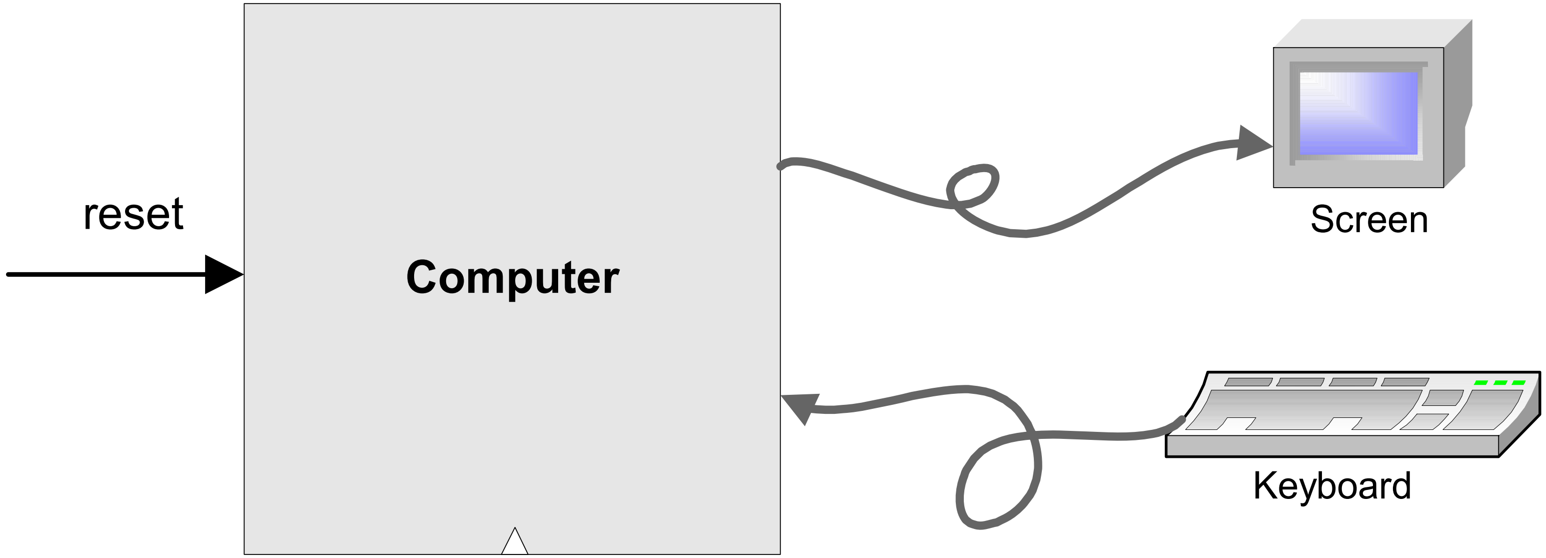
www.nand2tetris.org

Where we are at:



The Hack computer

A 16-bit machine consisting of the following elements:



The program is stored in a ROM.

The A-instruction

symbolic

@value

- *value* is a non-negative decimal number <= 2^{15}-1 or
- A symbol referring to such a constant

binary

0*value*

- *value* is a 15-bit binary number

Example

@21

0000 0000 0001 0101

The C-instruction

symbolic

dest = *comp* ; *jump*

binary

111A $C_1C_2C_3C_4$ C_5C_6 D_1D_2 $D_3J_1J_2J_3$

opcode

not used

comp

dest

jump

The **C**-instruction

111A $C_1C_2C_3C_4$ C_5C_6 D_1D_2 $D_3J_1J_2J_3$

comp dest jump

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

The **C**-instruction

111A $C_1C_2C_3C_4$ C_5C_6 D_1D_2 $D_3J_1J_2J_3$

comp dest jump

A d1	D d2	M d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

The **C**-instruction

111A $C_1C_2C_3C_4$ C_5C_6 D_1D_2 $D_3J_1J_2J_3$

comp — dest — jump

j1 ($out < 0$)	j2 ($out = 0$)	j3 ($out > 0$)	**Mnemonic**	**Effect**
0	0	0	null	No jump
0	0	1	JGT	If $out > 0$ jump
0	1	0	JEQ	If $out = 0$ jump
0	1	1	JGE	If $out \geq 0$ jump
1	0	0	JLT	If $out < 0$ jump
1	0	1	JNE	If $out \neq 0$ jump
1	1	0	JLE	If $out \leq 0$ jump
1	1	1	JMP	Jump

Hack assembly/machine language

Source code (example)

```
// Computes 1+...+RAM[0]
// And stored the sum in RAM[1]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0
(LOOP)
    @i    // if i>RAM[0] goto WRITE
    D=M
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

assemble

Hack assembler or CPU emulator

Target code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010110
1110101010000111
```

assembly code v.s. machine code

The Hack computer

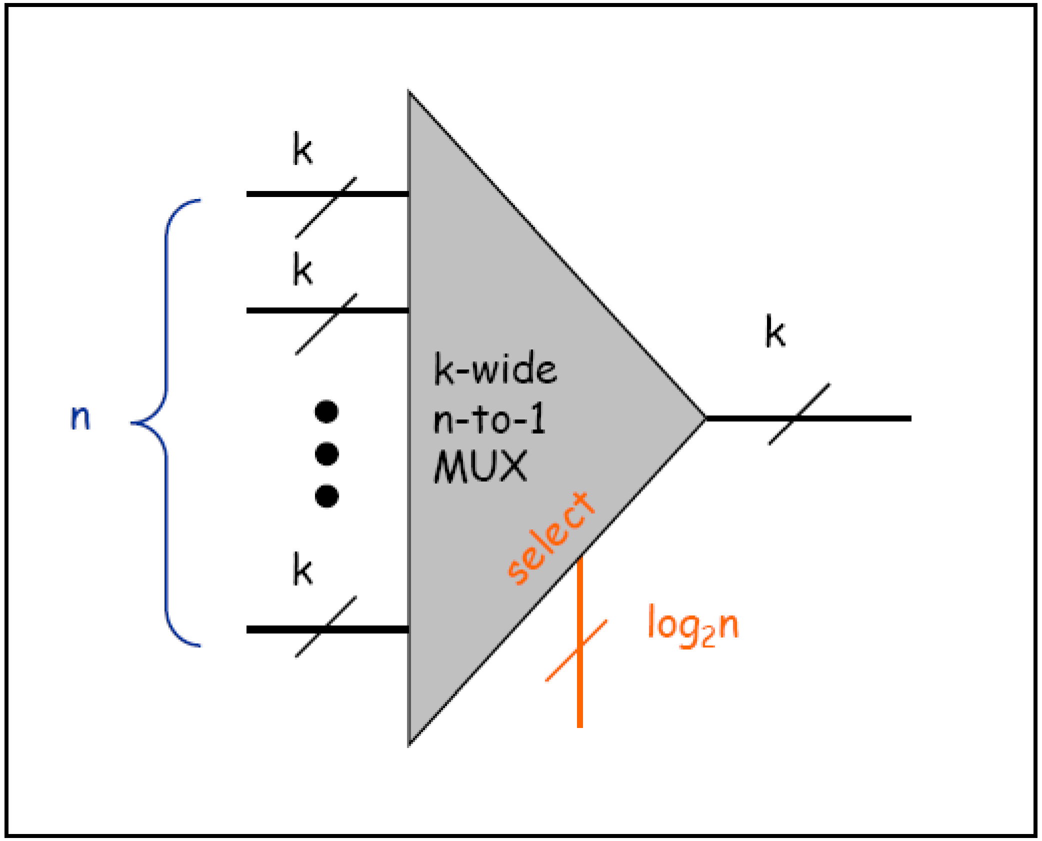
- A 16-bit Von Neumann platform
- The *instruction memory* and the *data memory* are physically separate
- Screen: 512 rows by 256 columns, black and white
- Keyboard: standard
- Designed to execute programs written in the Hack machine language
- Can be easily built from the chip-set that we built so far in the course

<u>Main parts of the Hack computer:</u>

- Instruction memory (ROM)
- Memory (RAM):
  - Data memory
  - Screen (memory map)
  - Keyboard (memory map)
- CPU
- Computer (the logic that holds everything together).

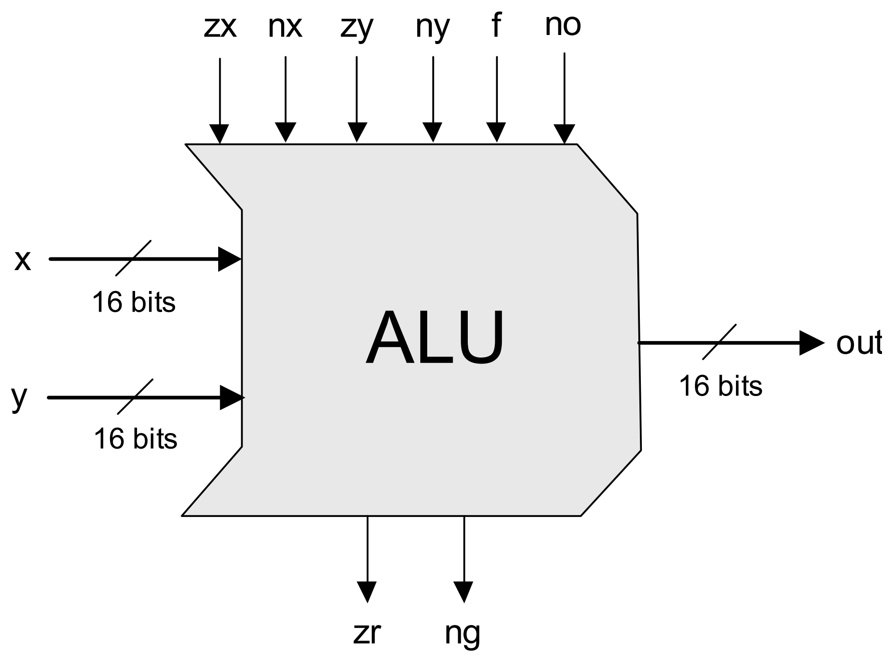
Goal: select from one of n k-bit buses

- Implemented by layering k n-to-1 multiplexer



Interface

Hack ALU



out(x, y, control bits) =

x+y, x-y, y–x,

0, 1, -1,

x, y, -x, -y,

x!, y!,

x+1, y+1, x-1, y-1,

x&y, x|y

Hack ALU

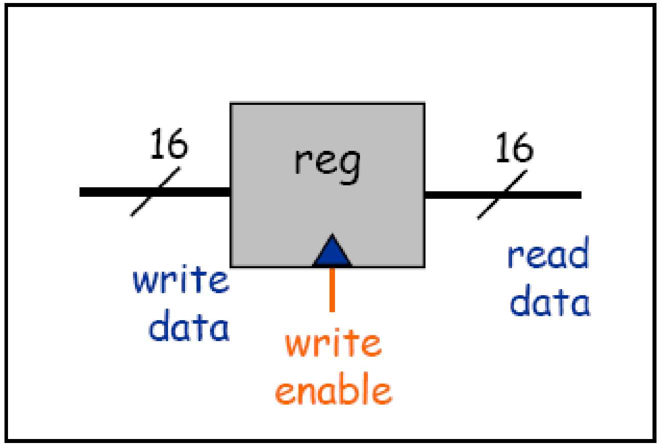
These bits instruct how to preset the x input		These bits instruct how to preset the y input		This bit selects between + / And	This bit inst. how to postset out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	f(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Registers

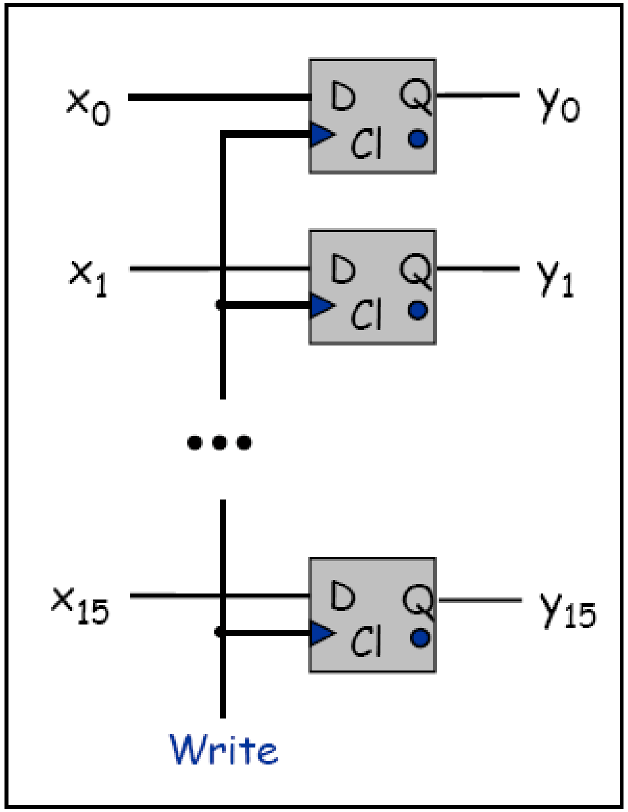
k-bit register.

- Stores k bits.
- Register contents always available on output.
- If write enable is asserted, k input bits get copied into register.

Ex: Program Counter, 16 TOY registers, 256 TOY memory locations.

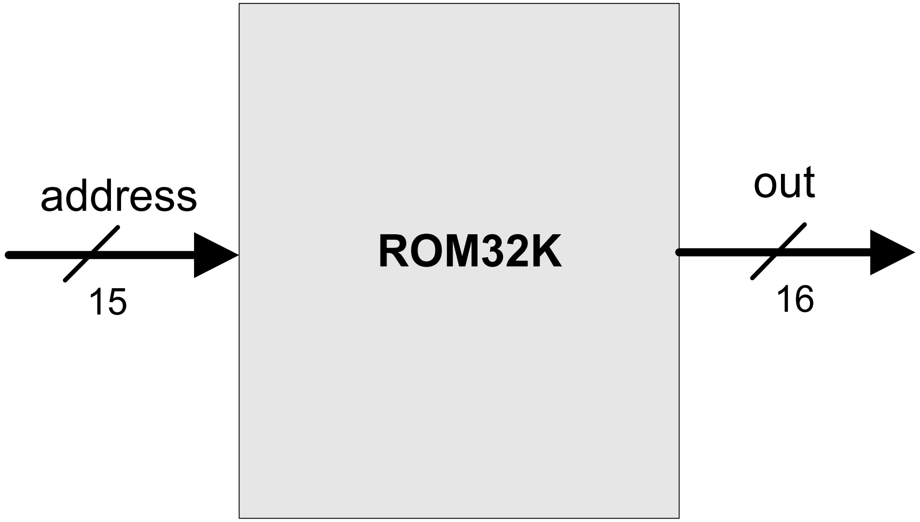


16-bit Register Interface



16-bit Register Implementation

ROM (Instruction memory)



Function:

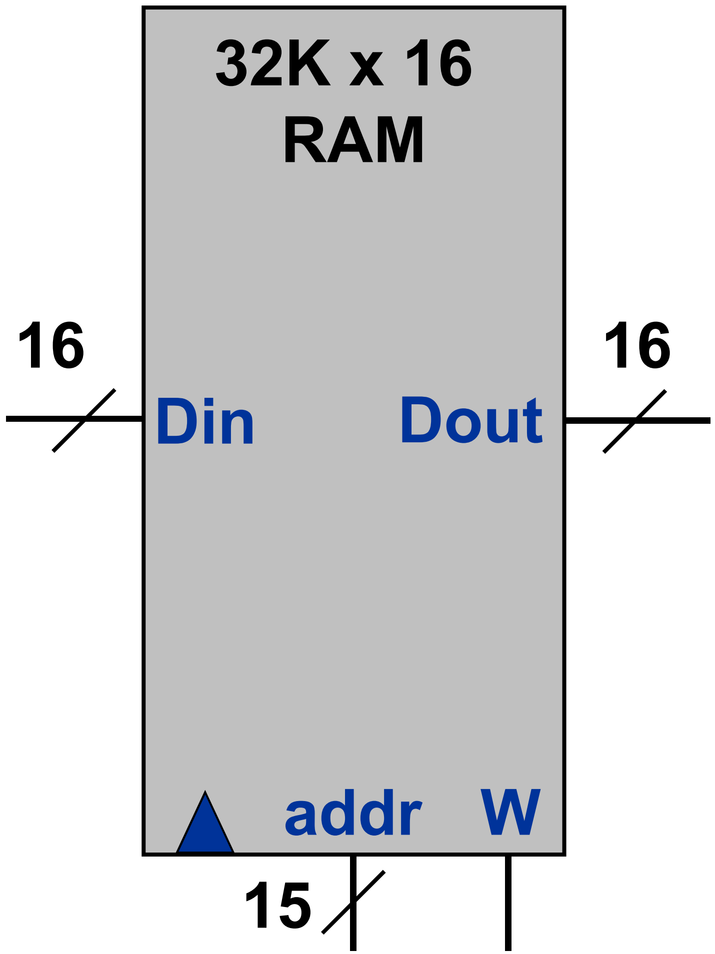
- The `ROM` is pre-loaded with a program written in the Hack machine language
- The `ROM` chip always emits a 16-bit number:

```
out = ROM32K[address]
```

- This number is interpreted as the *current instruction.*

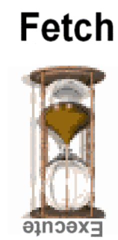
RAM (data memory)

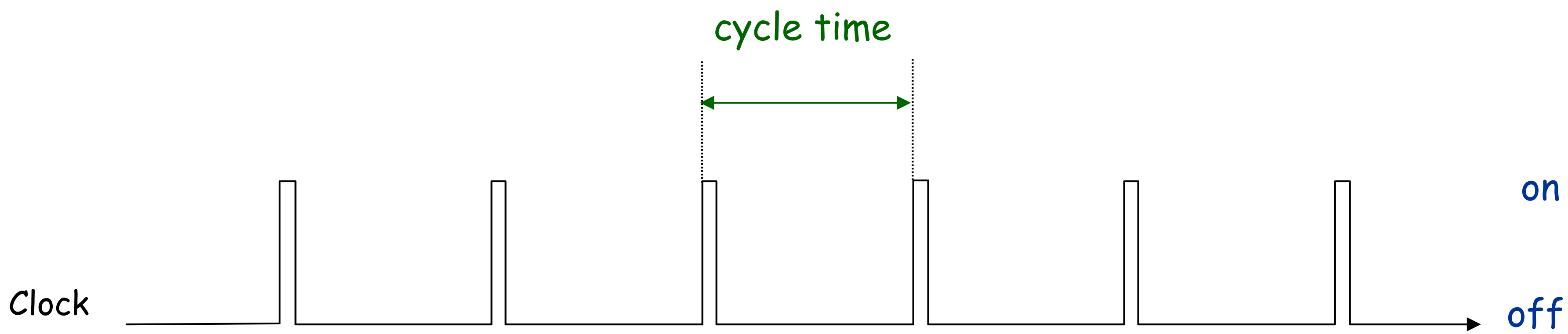
- We will discuss the details for Hack's data memory later.



Clock

- Clock.
  - Fundamental abstraction: regular on-off pulse.
    - on: fetch phase
    - off: execute phase
  - External analog device.
  - Synchronizes operations of different circuit elements.
  - Requirement: clock cycle longer than max switching time.





Design a processor

- How to build a processor (Hack, this time)
  - Develop instruction set architecture (ISA)
    - 16-bit words, two types of machine instructions
  - Determine major components
    - ALU, registers, program counter, memory
  - Determine datapath requirements
    - Flow of bits
  - Analyze how to implement each instruction
    - Determine settings of control signals

Hack programming reference card

Hack commands:

A-command: @value // A<-value; M=RAM[A]

C-command: dest = comp ; jump // dest = and ;jump
// are optional

Where:

comp =
0 , 1 , -1 , D , A , !D , !A , -D , -A , D+1 , A+1 , D-1, A-1 , D+A , D-A , A-D , D&A , D|A,
M , !M , -M , M+1, M-1 , D+M, D-M, M-D, D&M, D|M

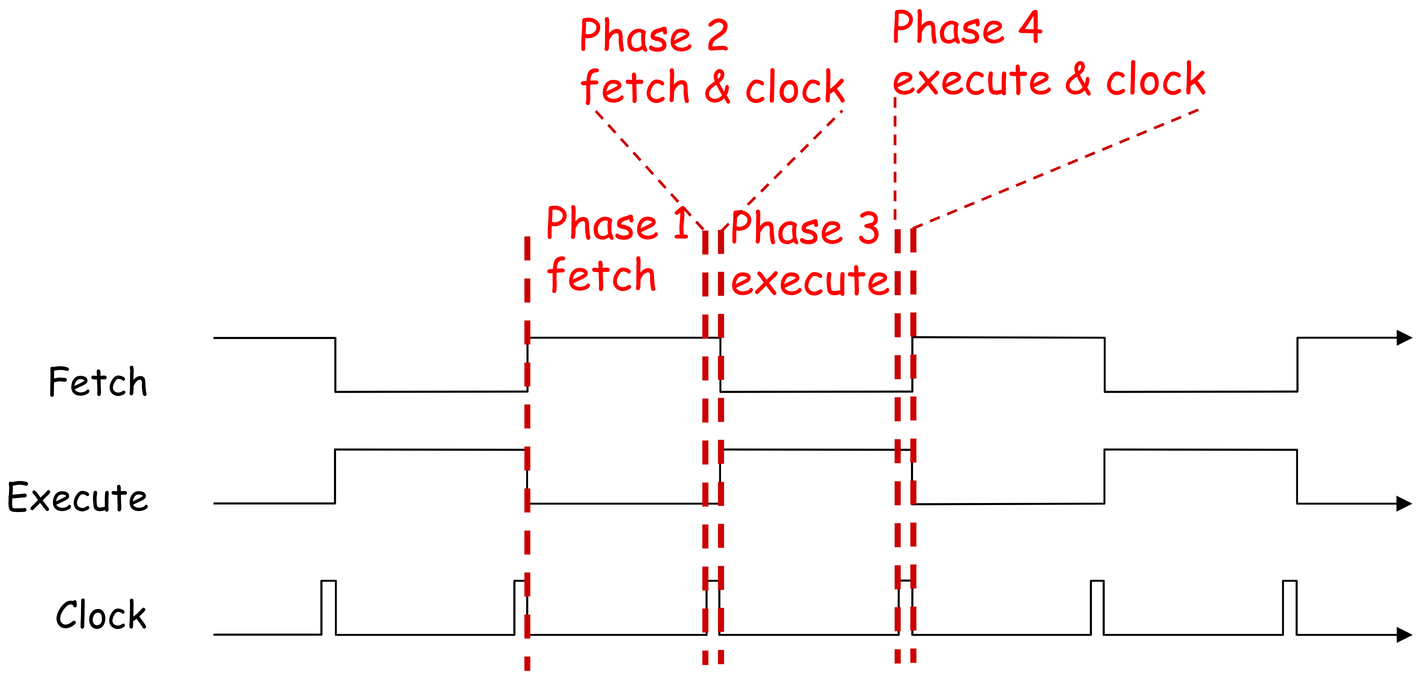
dest = M, D, A, MD, AM, AD, AMD, or null

jump = JGT , JEQ , JGE , JLT , JNE , JLE , JMP, or null

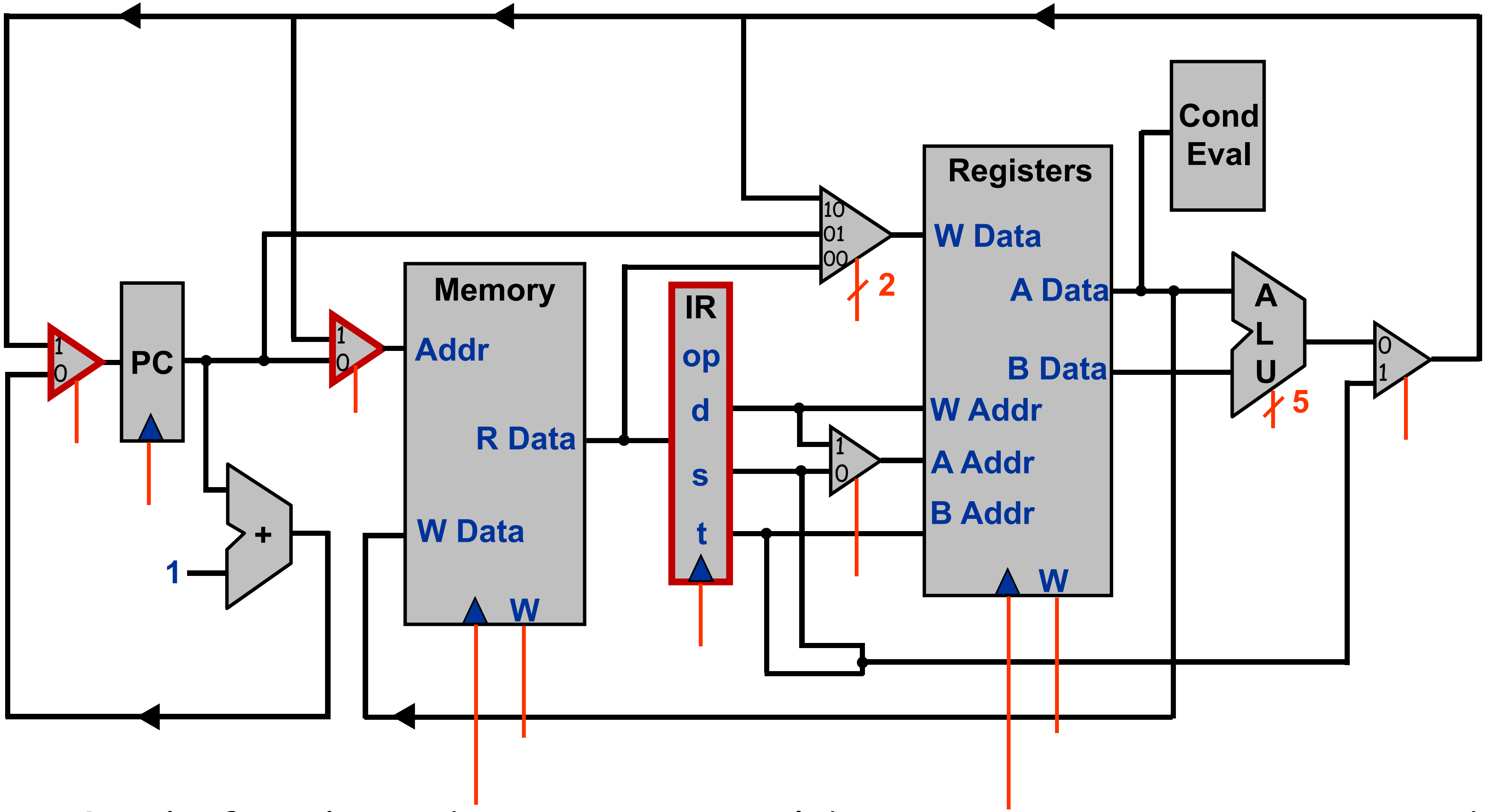
In the command dest = comp; jump, the jump materialzes (PC<-A) if (comp jump 0) is true. For example, in D=D+1,JLT, we jump if D+1 < 0.

Fetch and execute

- In Toy, we have two phases: fetch and execution .
- We use two cycles since fetch and execute phases each access memory and alter program counter.
  - fetch [set memory address from pc]
  - fetch and clock [write instruction to IR]
  - execute [set ALU inputs from registers]
  - execute and clock [write result of ALU to registers]



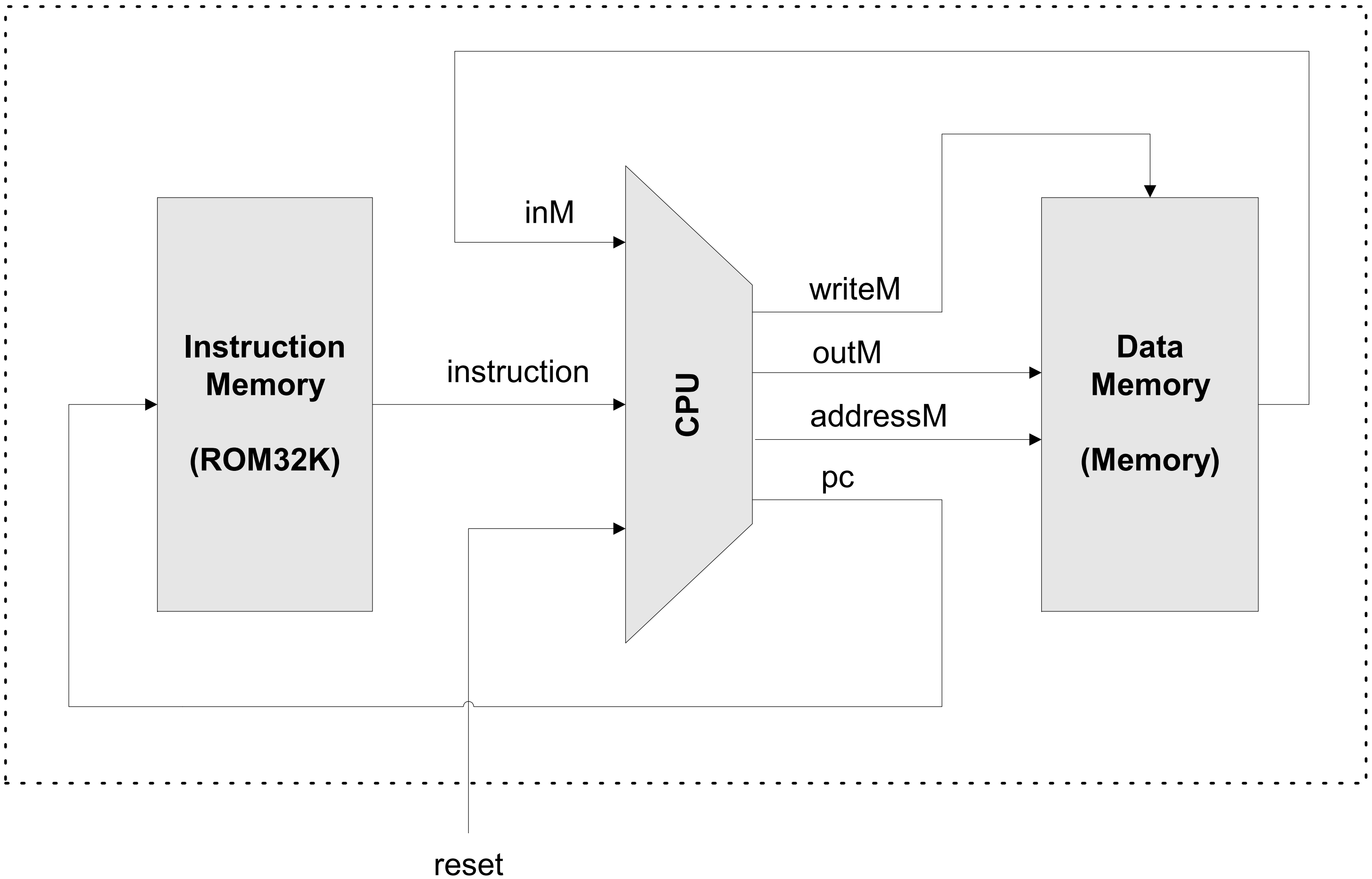
Toy architecture



- Both fetch and execute would access memory. To avoid conflict, we add a MUX. Similar for PC.
- In addition, we need a register IR to store the instruction.

Fetch and execute

- In Hack, we avoid it by using two separate memory chips, one for data and the other for instruction.

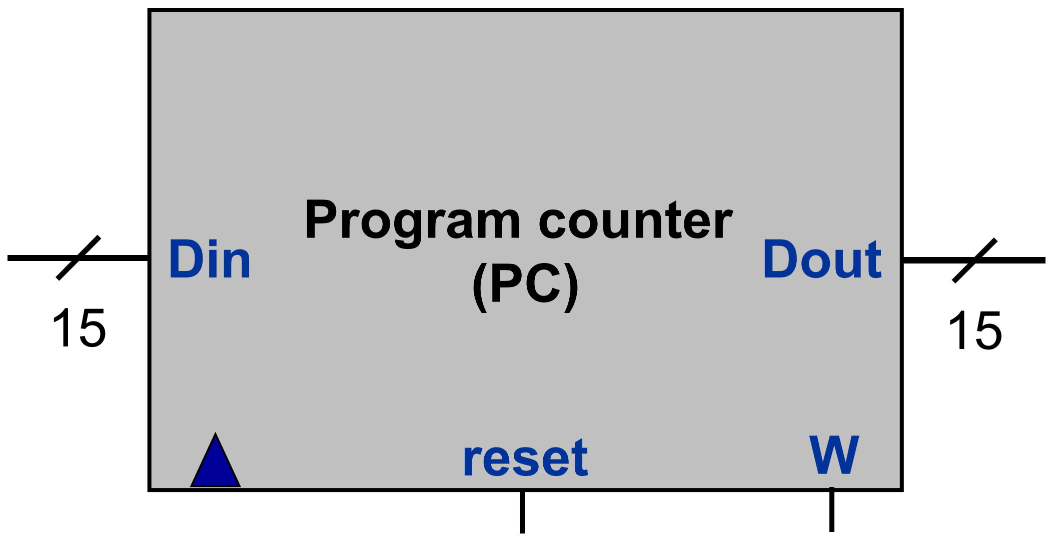


Design a processor

- How to build a processor (Hack, this time)
  - Develop instruction set architecture (ISA)
    - 16-bit words, two types of machine instructions
  - Determine major components
    - ALU, registers, program counter, memory
  - Determine datapath requirements
    - Flow of bits
  - Analyze how to implement each instruction
    - Determine settings of control signals

Program counter

- Program counter emits the address of the next instruction.
  - To start/restart the program execution: PC=0
  - No jump: PC++
  - Unconditional jump: PC=A
  - Conditional jump: if (cond.) PC=A else PC++



Note that the design is slightly different from your project #3.

Program counter

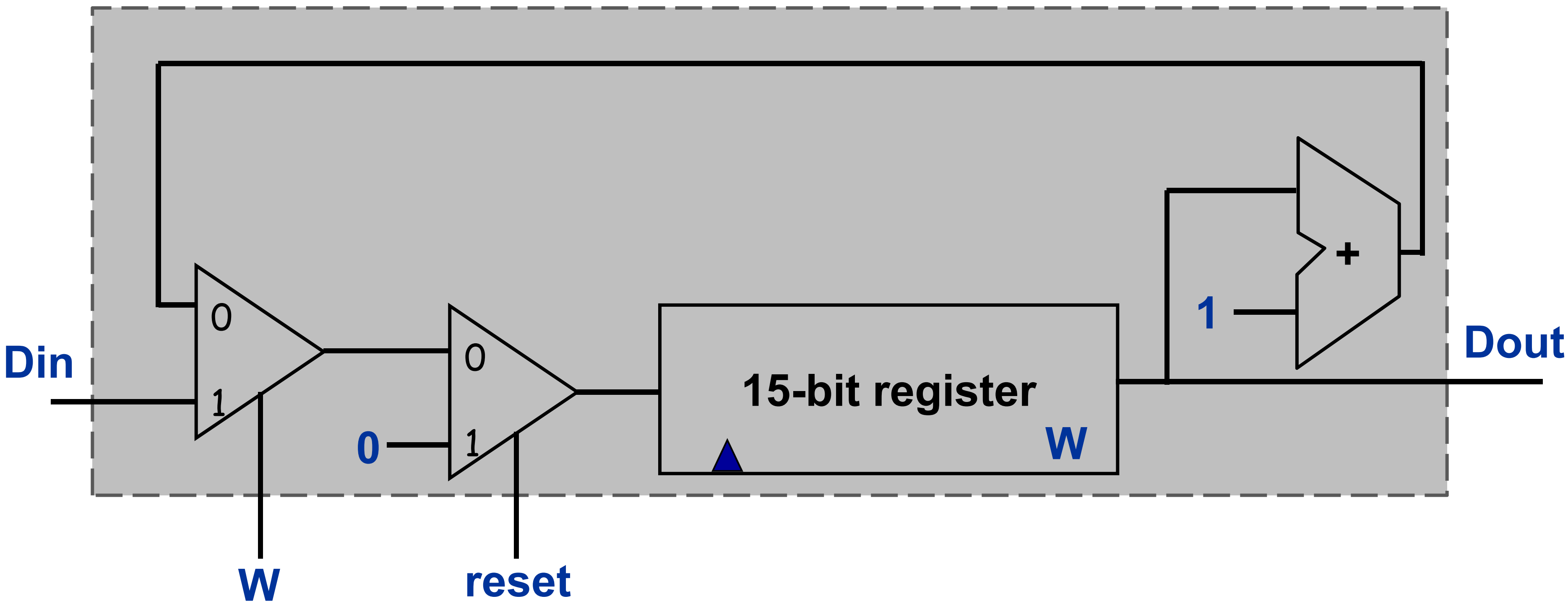
```
if (reset) PC=0
else if (W) PC=Din
     else PC++
```

Program counter

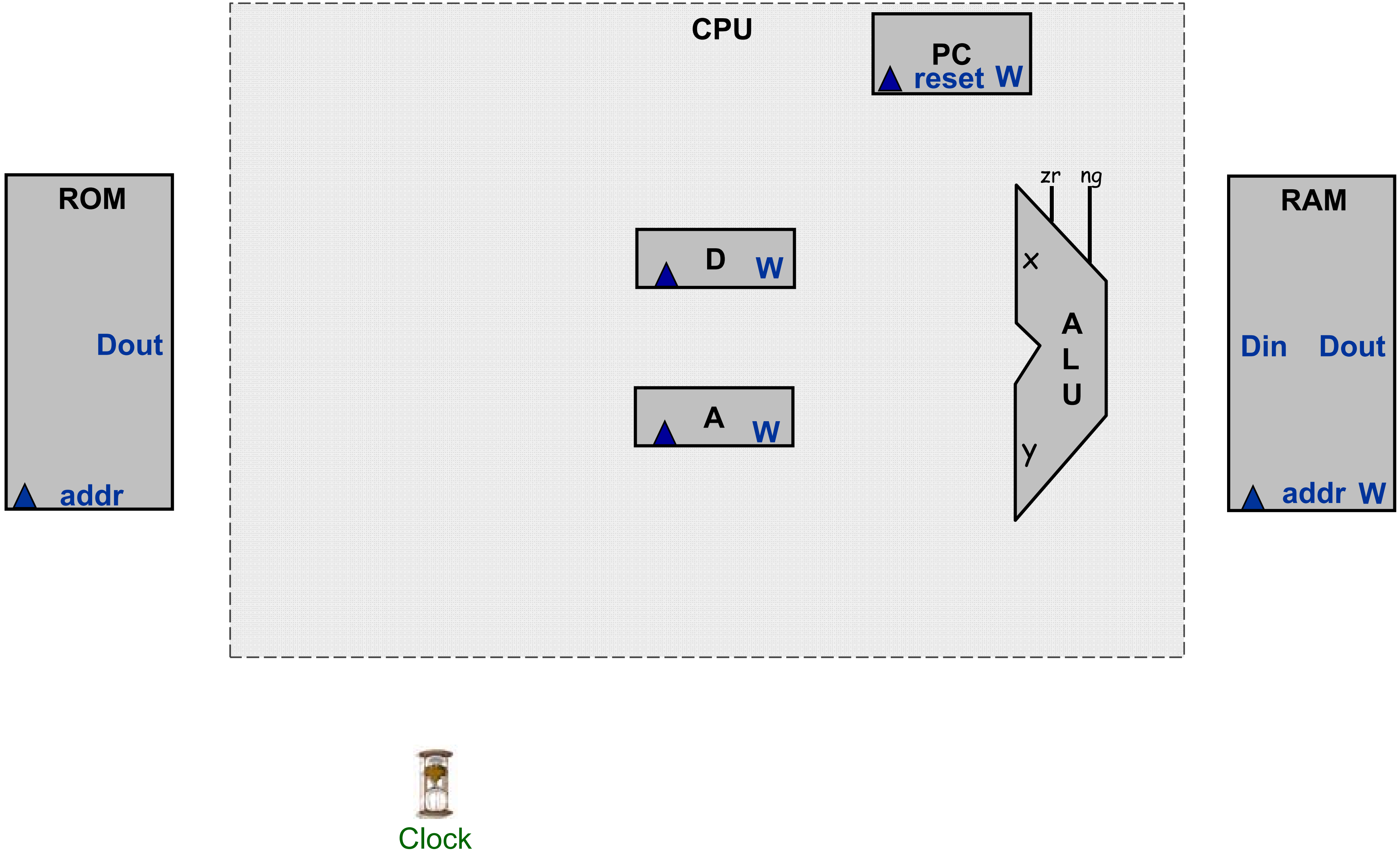
if (reset) PC=0

else if (W) PC=Din

else PC++



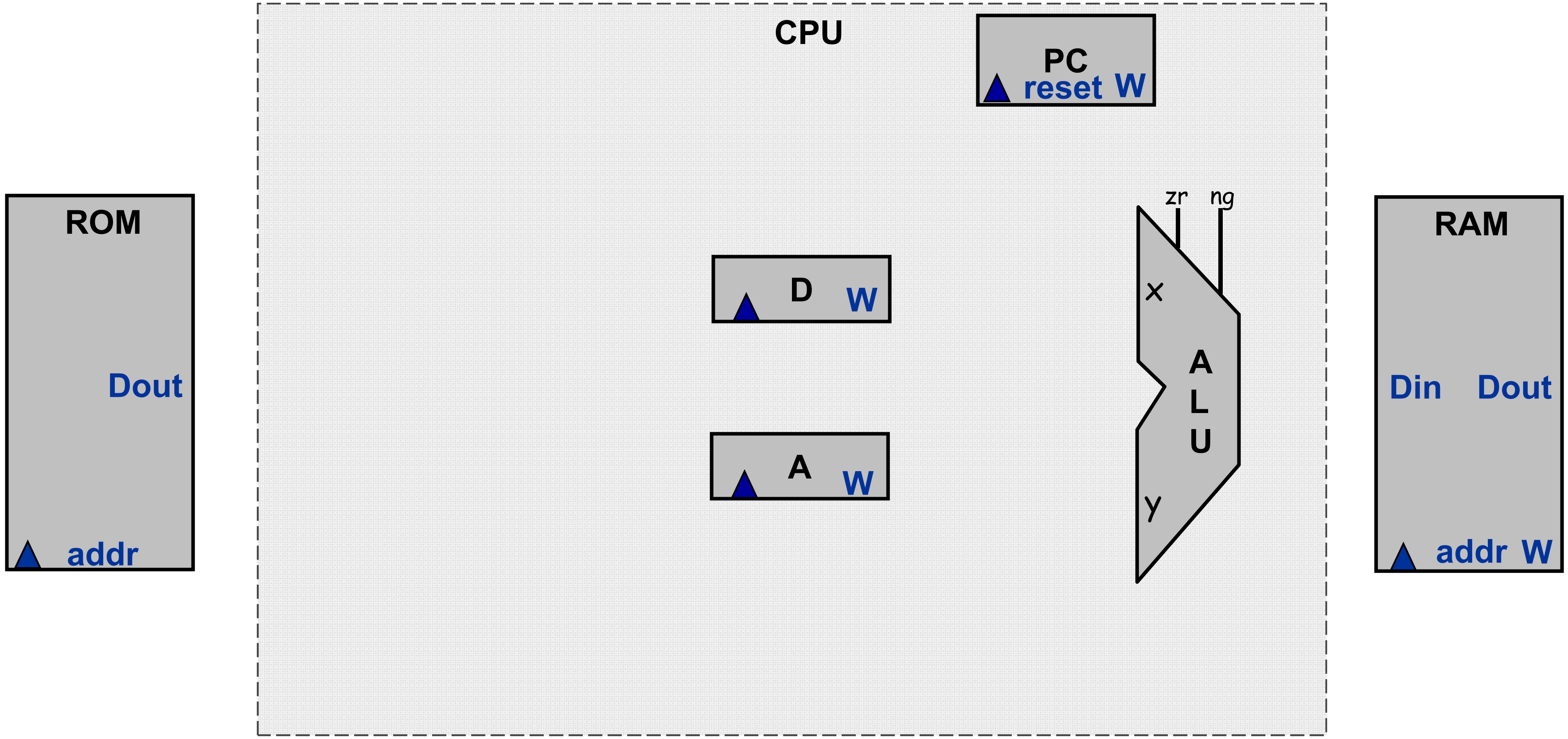
Hack architecture (component)



Design a processor

- How to build a processor (Hack, this time)
  - Develop instruction set architecture (ISA)
    - 16-bit words, two types of machine instructions
  - Determine major components
    - ALU, registers, program counter, memory
  - Determine datapath requirements
    - Flow of bits
  - Analyze how to implement each instruction
    - Determine settings of control signals

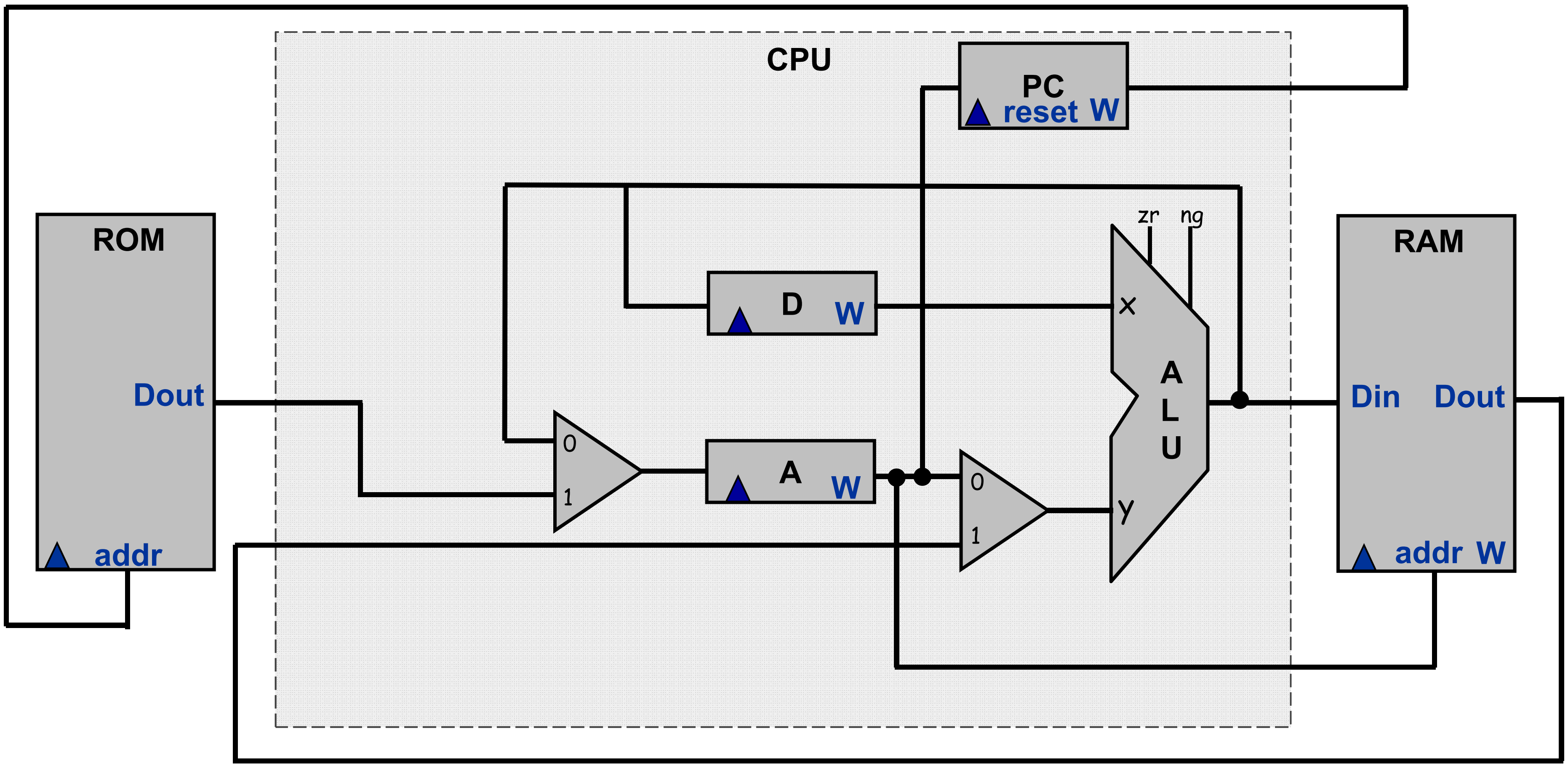
Hack architecture (data path)



@value // A<-value; M=RAM[A]

[ADM] = x op y; jump // x=D; y=A or M; if jump then PC<-A

Hack architecture (data path)



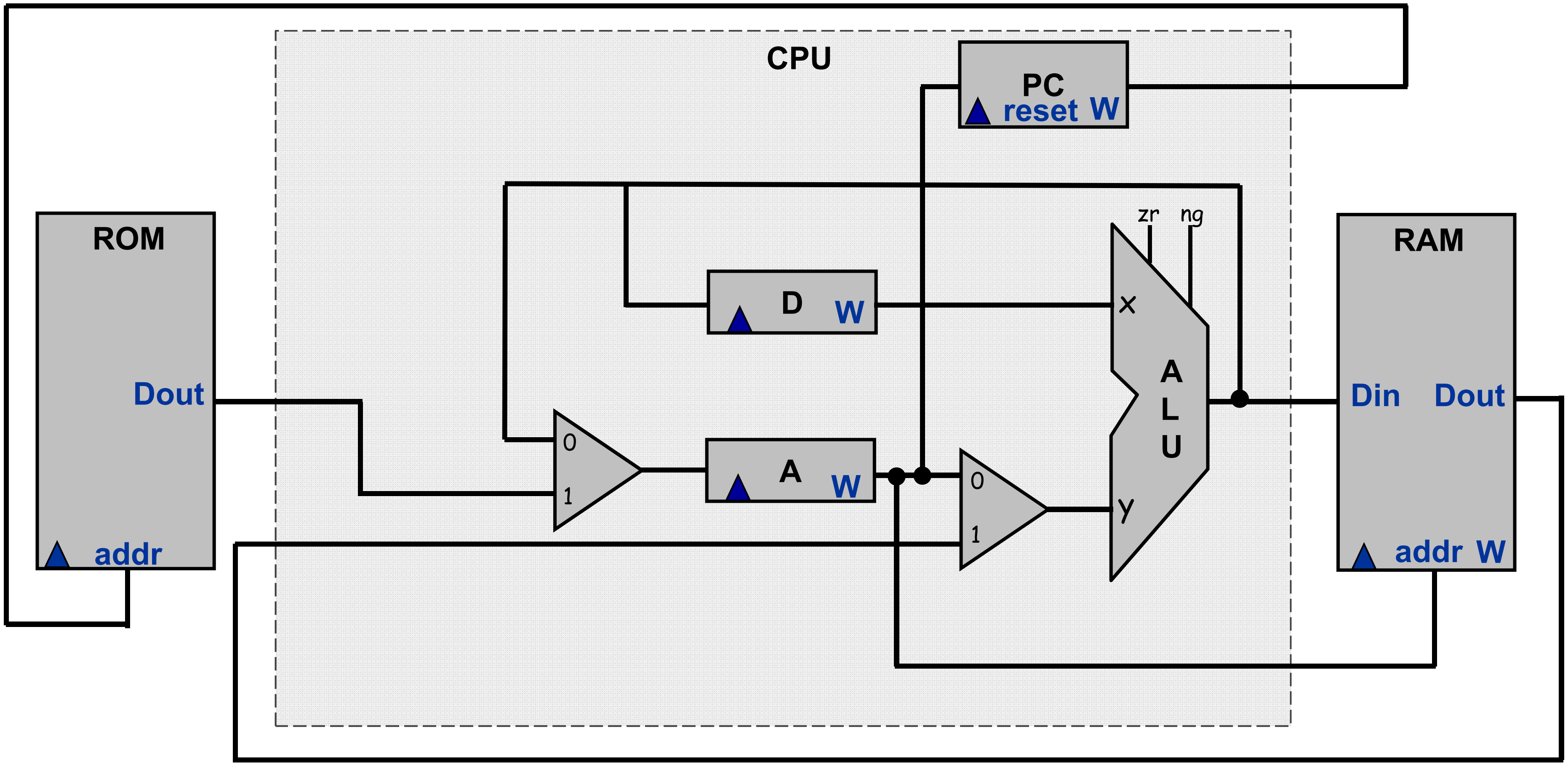
@value // A<-value; M=RAM[A]

[ADM] = x op y; jump // x=D; y=A or M; if jump then PC<-A

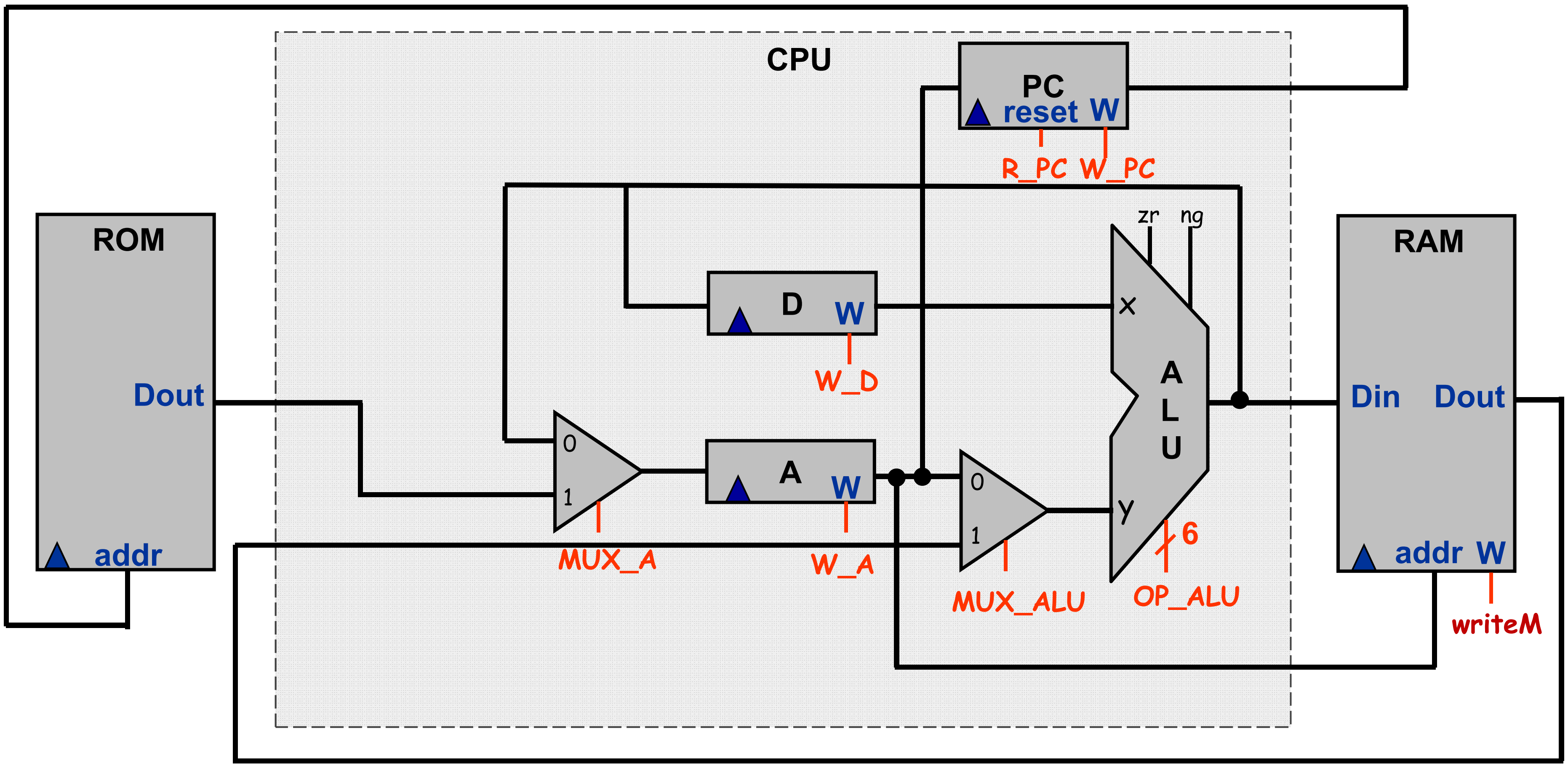
Design a processor

- How to build a processor (Hack, this time)
  - Develop instruction set architecture (ISA)
    - 16-bit words, two types of machine instructions
  - Determine major components
    - ALU, registers, program counter, memory
  - Determine datapath requirements
    - Flow of bits
  - Analyze how to implement each instruction
    - Determine settings of control signals

Hack architecture (data path)

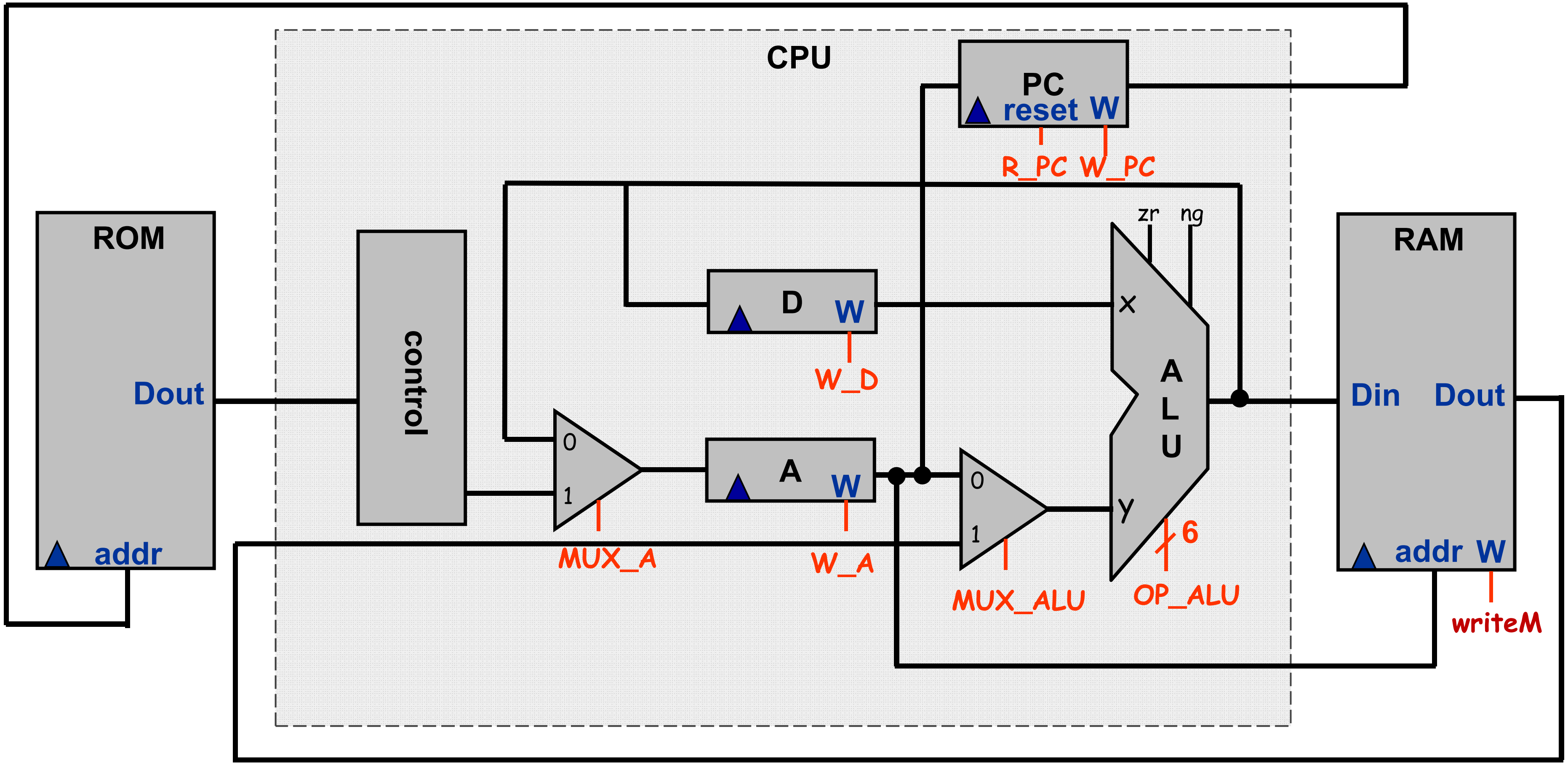


Hack architecture (control)

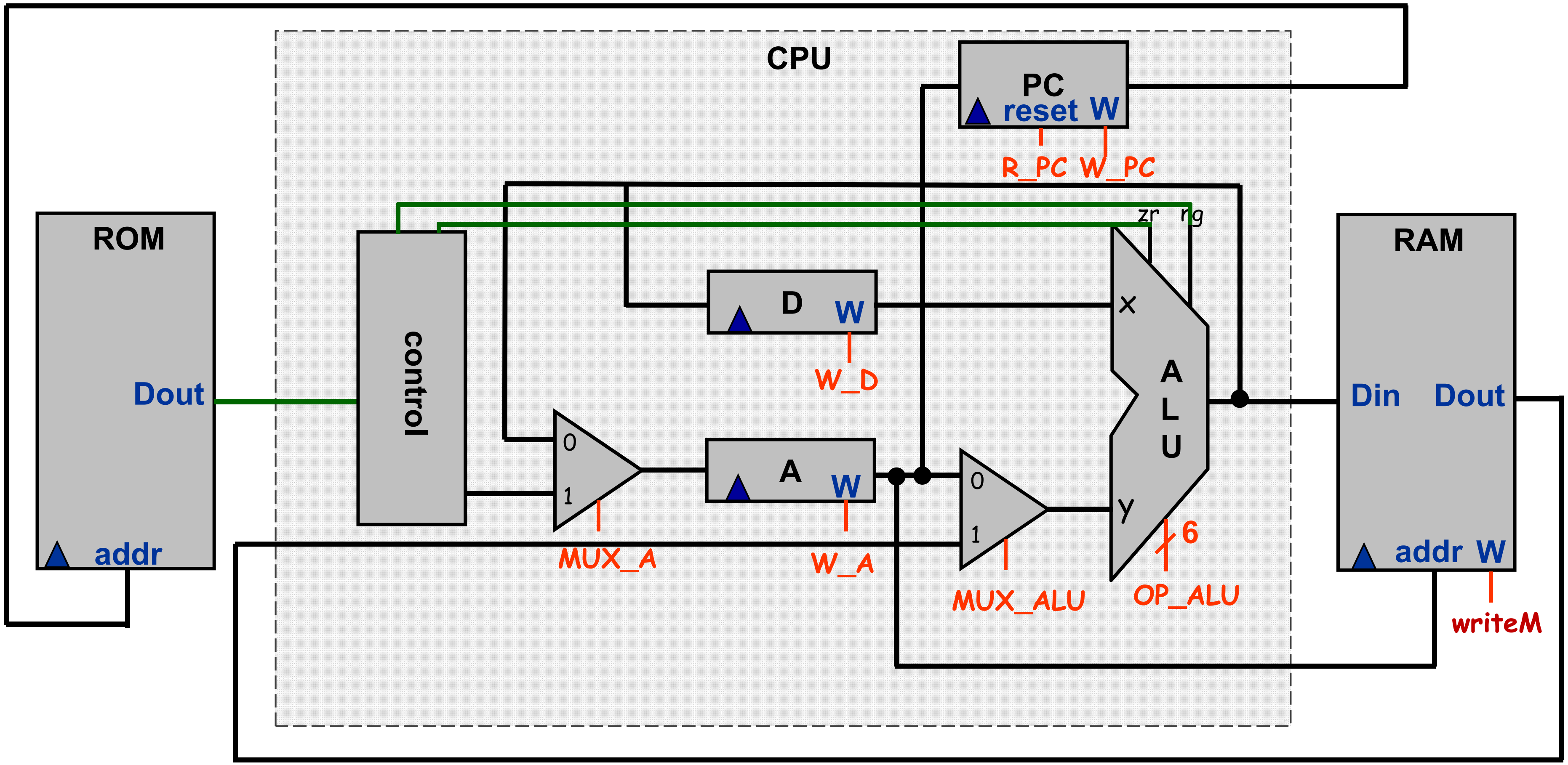


A total of 13 control signals

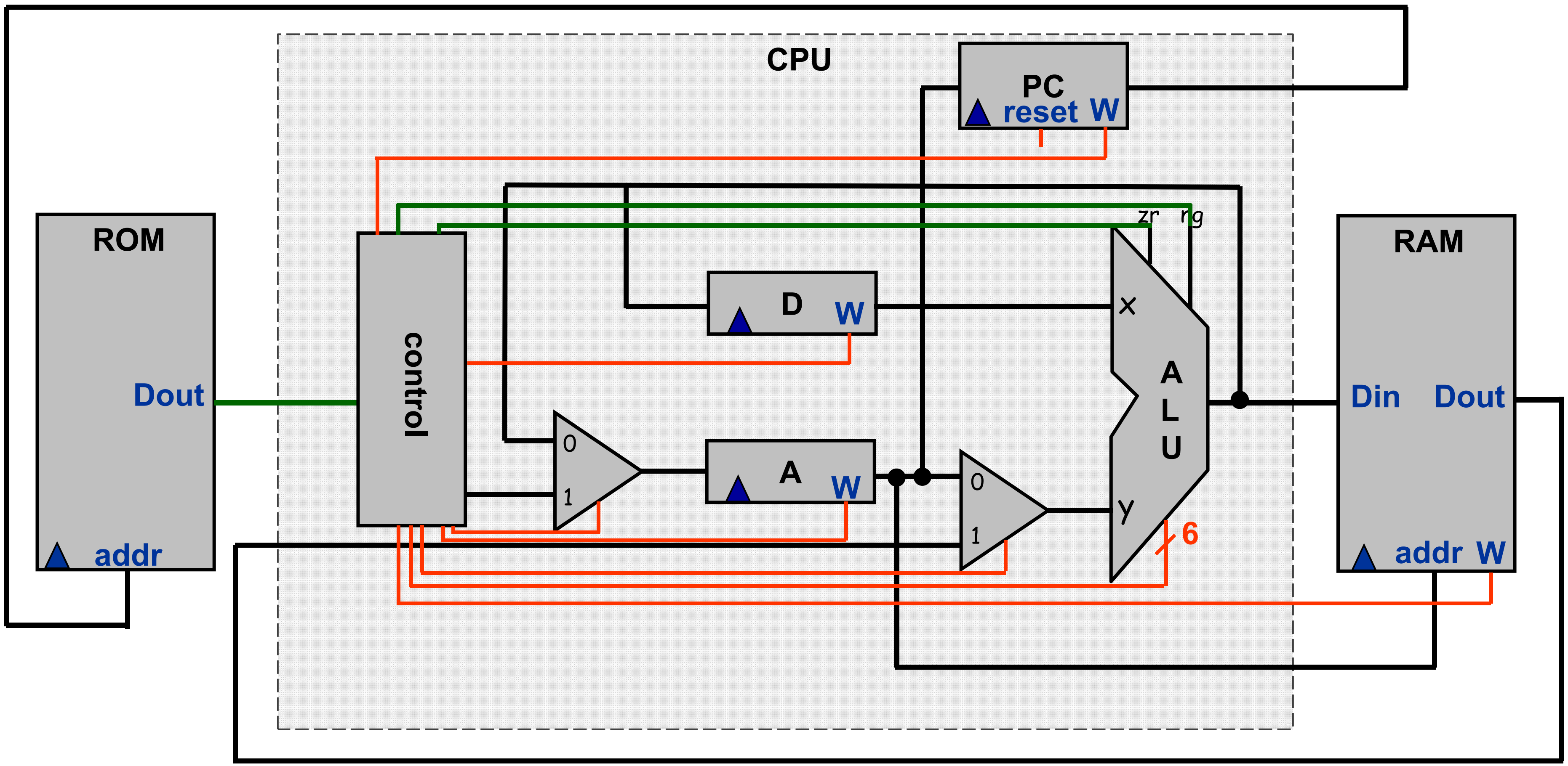
Hack architecture (control)



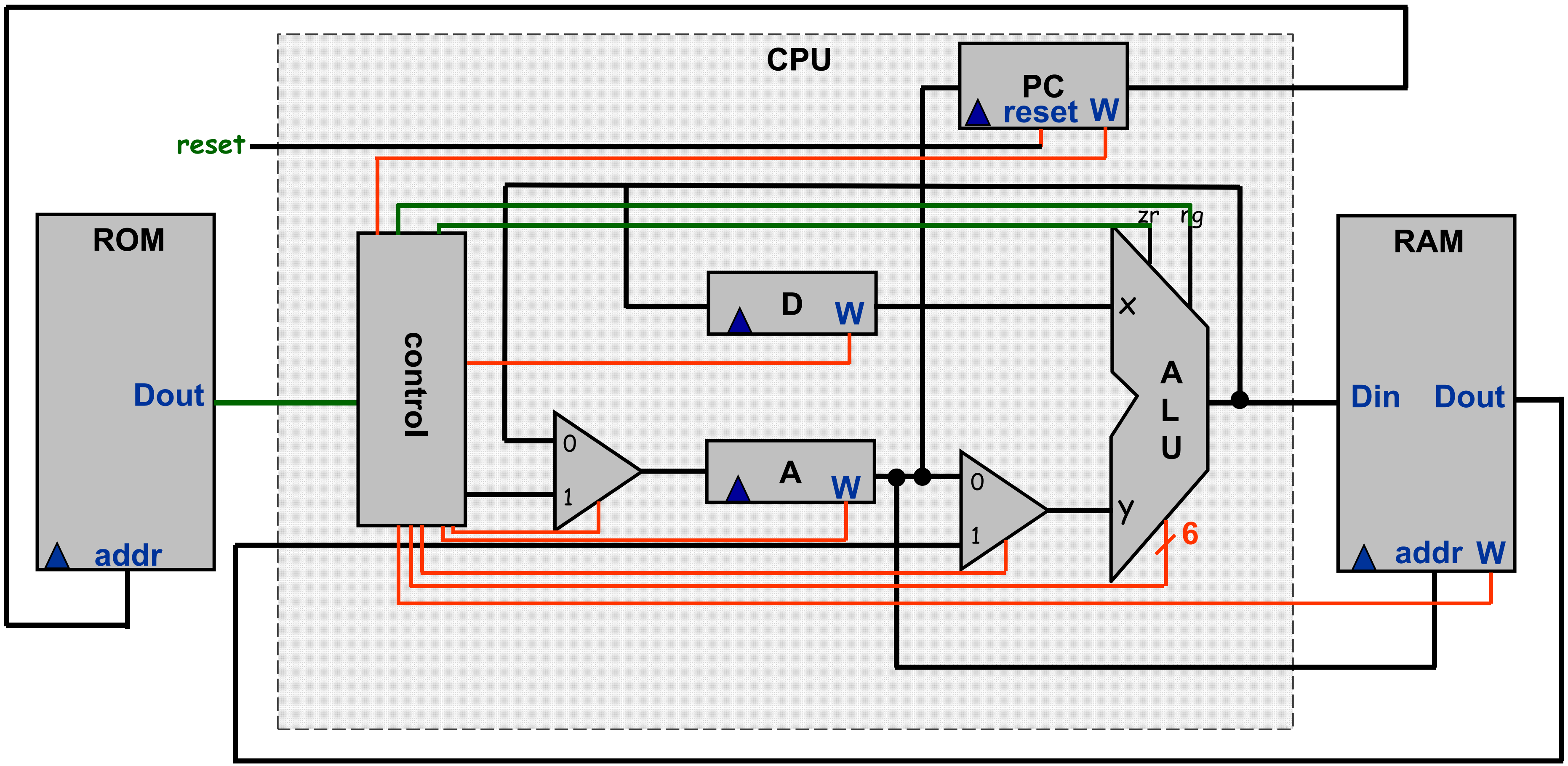
Hack architecture (control)



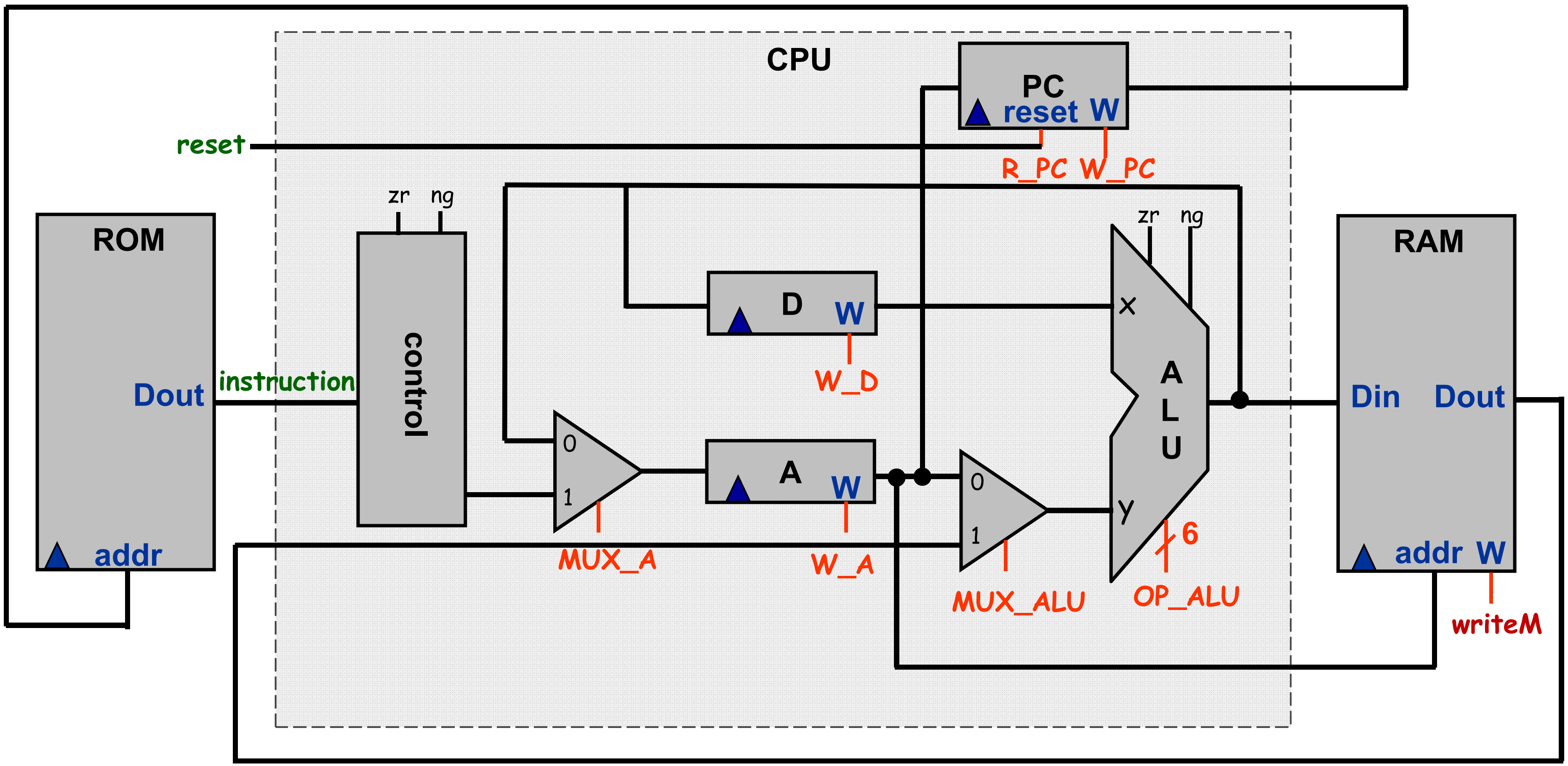
Hack architecture (control)



Hack architecture (control)



Hack architecture (control)



Hack architecture (control)

- Inputs: instruction, zr, ng
  - instruction

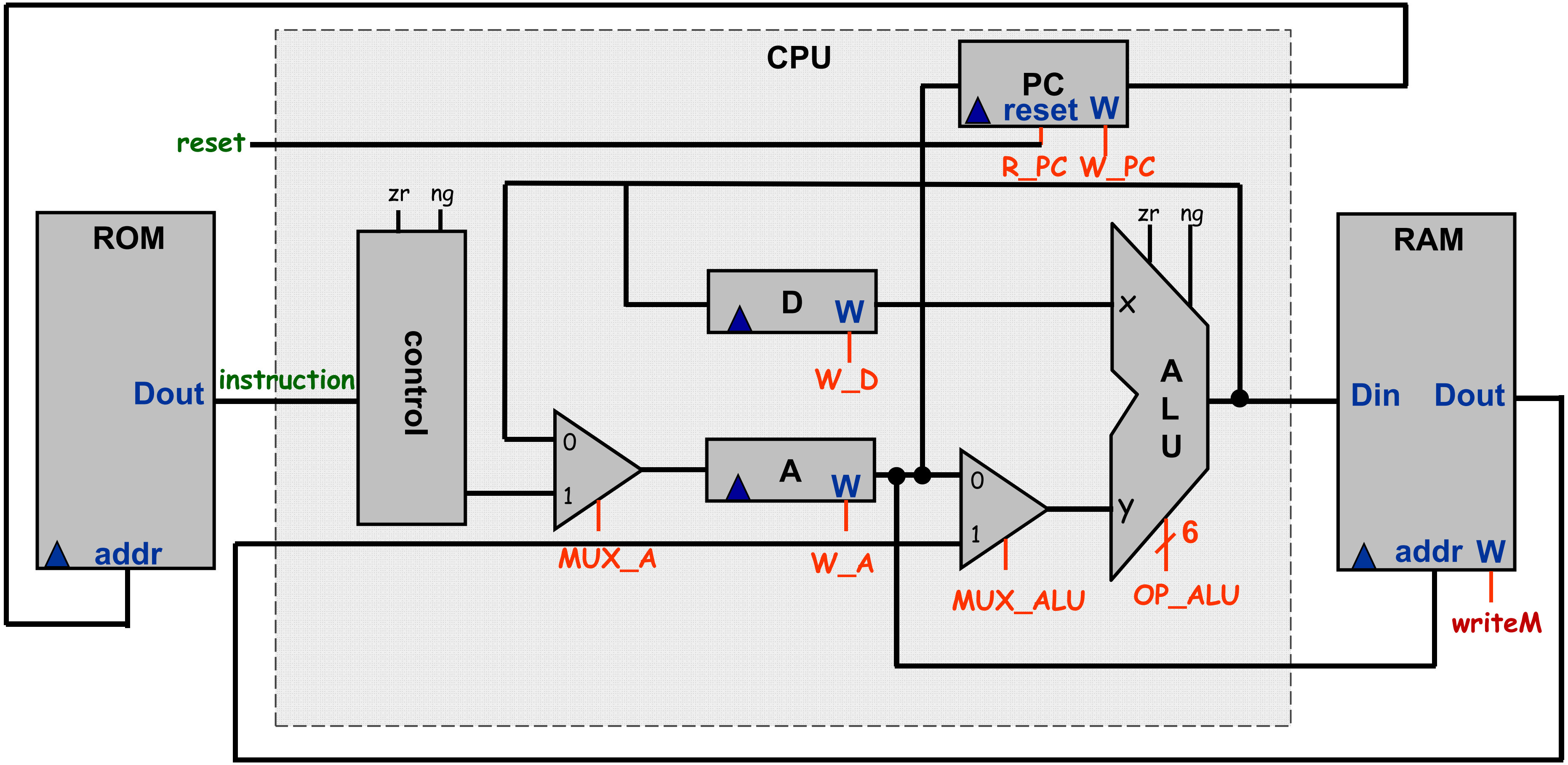
I_{15} $I_{14..0}$

0	value

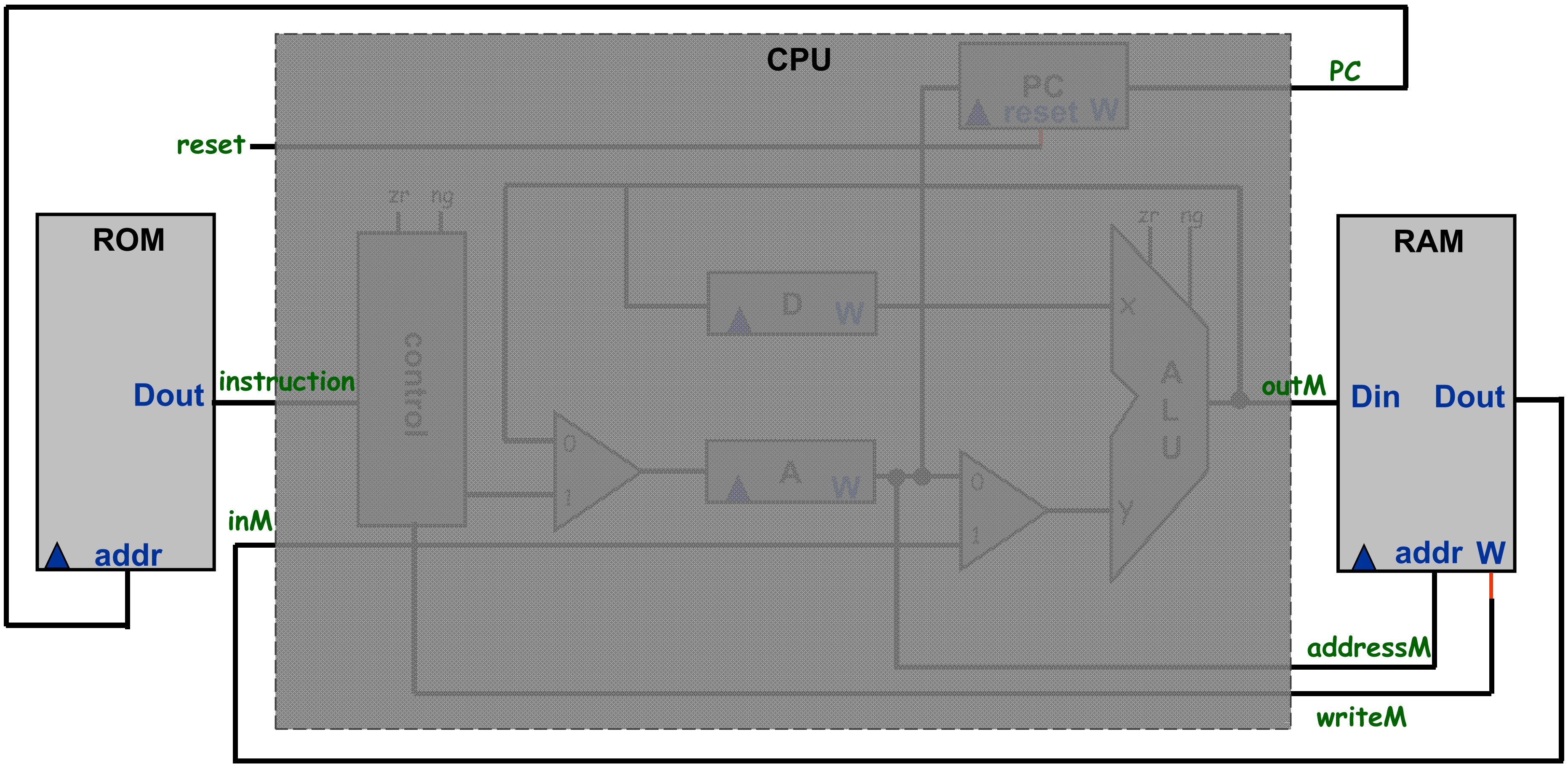
$111A\ C_1C_2C_3C_4\ C_5C_6\ D_1D_2\ D_3J_1J_2J_3$

- Outputs:
  - OP_ALU
  - MUX_A
  - MUX_ALU
  - W_A
  - W_D
  - writeM
  - W_PC

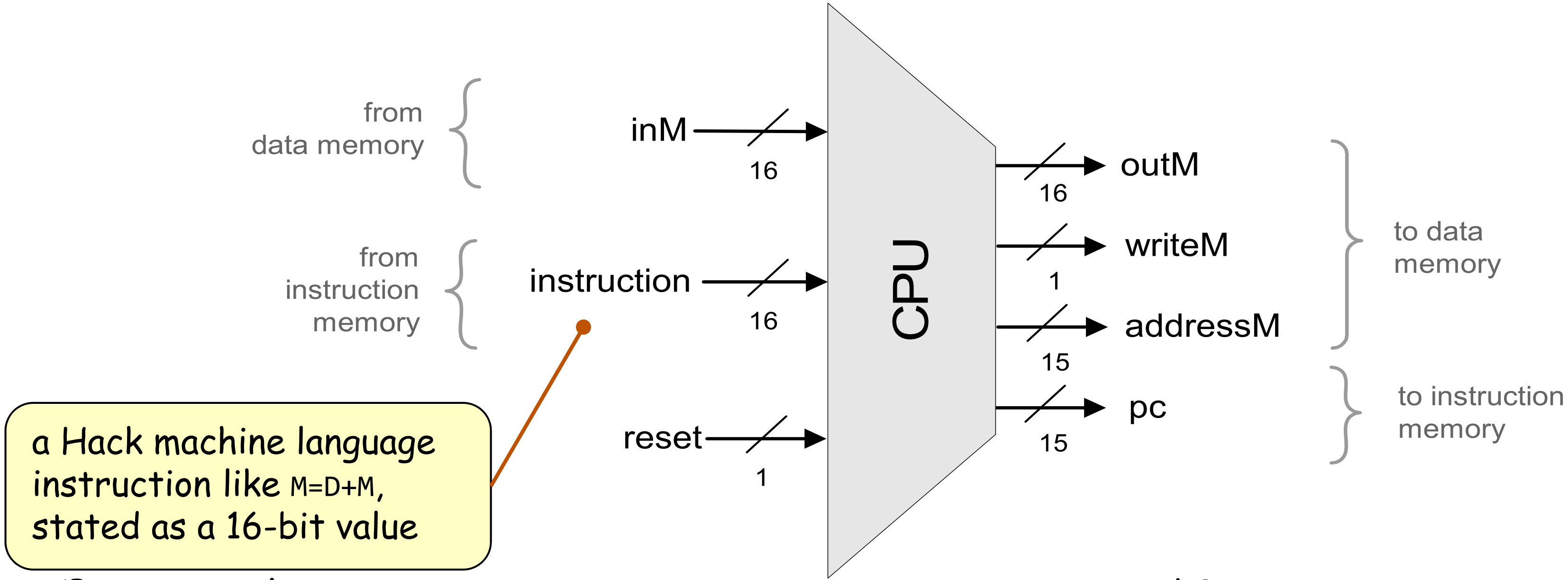
Hack architecture (trace @10 / D=M+1;JGE)



Hack architecture (CPU interface)



Hack CPU



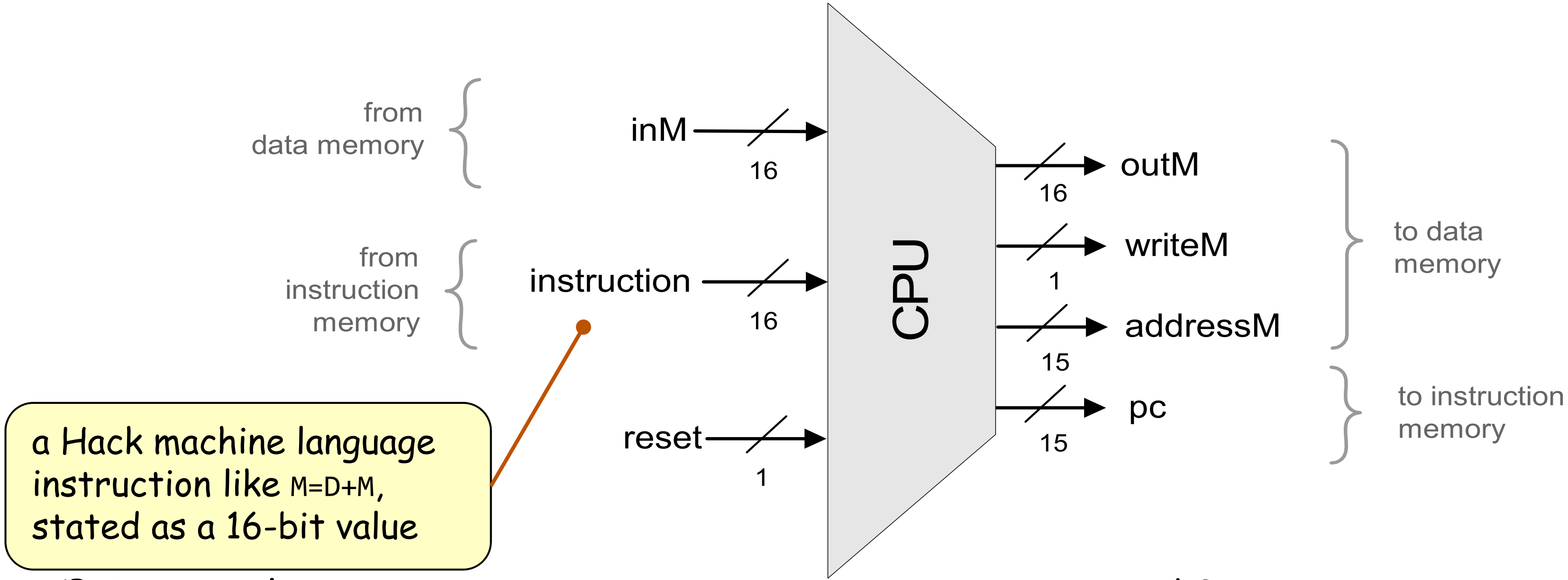
CPU internal components (invisible in this chip diagram): `ALU` and 3 registers: `A`, `D`, `PC`

CPU execute logic:

The CPU executes the `instruction` according to the Hack language specification:

- The `D` and `A` values, if they appear in the instruction, are read from (or written to) the respective CPU-resident registers
- If the instruction is @x, then x is stored in the A-register; and the emitted `addressM` is updated.
- The `M` value, if there is one in the instruction's RHS, is read from `inM`
- If the instruction's LHS includes `M`, then the ALU output is placed in `outM`, the value of the CPU-resident `A` register is placed in `addressM`, and `writeM` is asserted.

Hack CPU



CPU internal components (invisible in this chip diagram): ALU and 3 registers: A, D, PC

CPU fetch logic:

Recall that:

1. the `instruction` may include a jump directive (expressed as non-zero jump bits)
2. the ALU emits two control bits, indicating if the ALU output is zero or less than zero

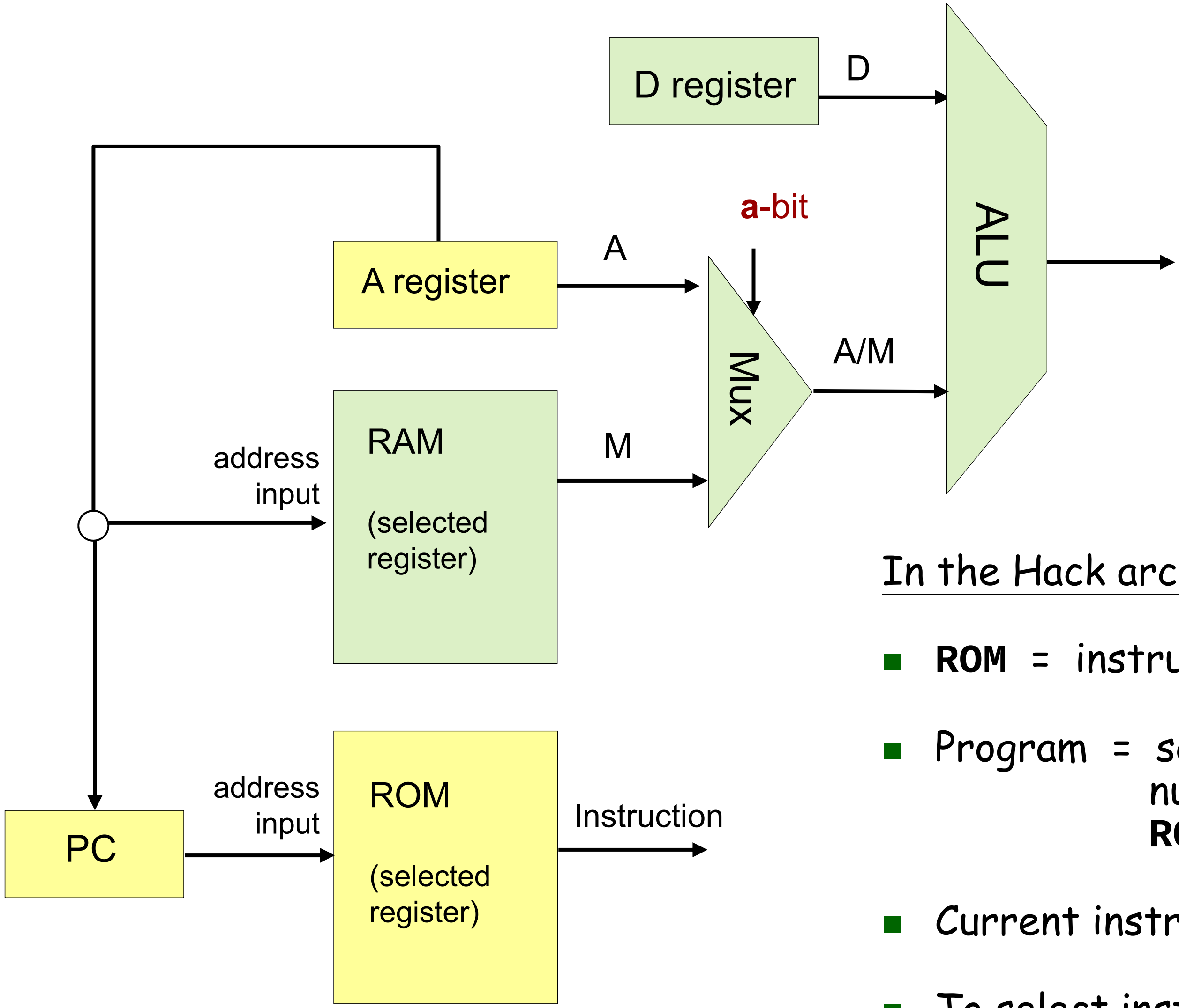
If `reset==0`: the CPU uses this information (the jump bits and the ALU control bits) as follows:

If there should be a jump, the PC is set to the value of A; else, PC is set to PC+1

The updated PC value is emitted by `pc`.

If `reset==1`: the PC is set to 0. `pc` emits `0`. (restarting the computer)

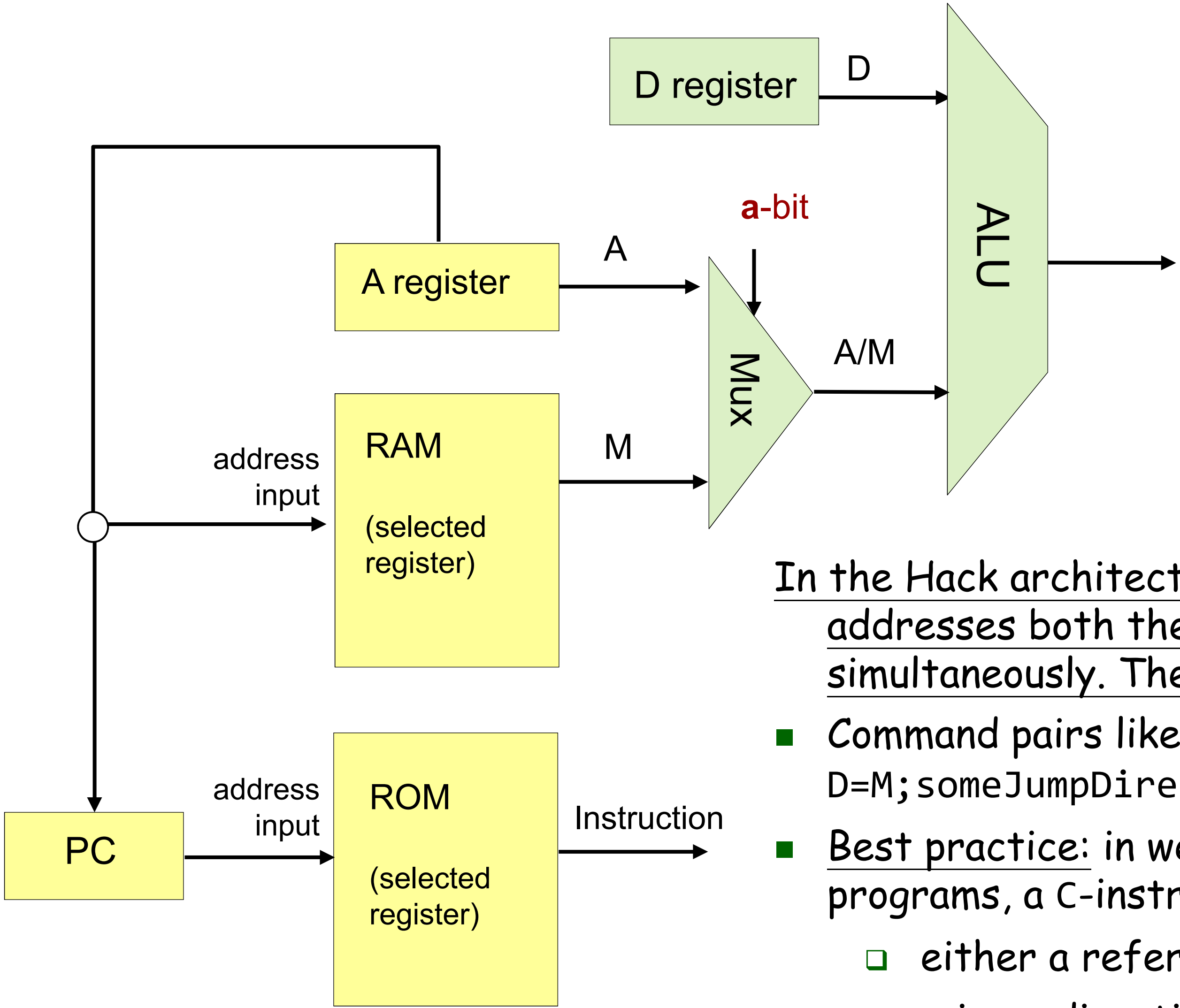
Control (focus on the yellow chips only)



In the Hack architecture:

- **ROM** = instruction memory
- Program = sequence of 16-bit numbers, starting at **ROM[0]**
- Current instruction = **ROM[PC]**
- To select instruction *n* from the **ROM**, we set **A** to *n*, using the instruction **@n**

Side note (focus on the yellow chip parts only)



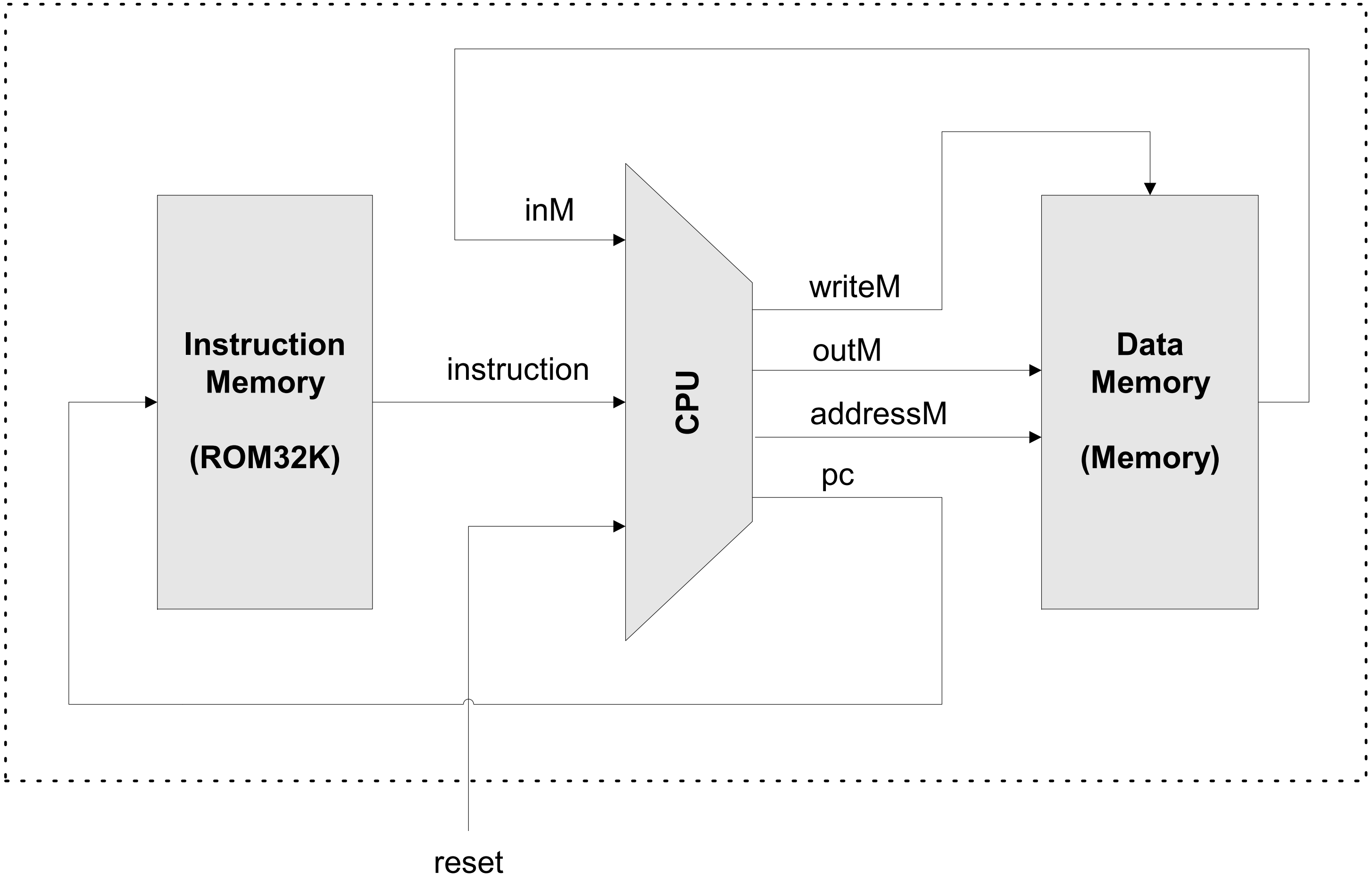
In the Hack architecture, the A register addresses both the RAM and the ROM, simultaneously. Therefore:

- Command pairs like `@addr` followed by `D=M;someJumpDirective` make no sense
- Best practice: in well-written Hack programs, a C-instruction should contain
  - either a reference to M, or
  - a jump directive,

  but not both.

The Hack computer (put together)

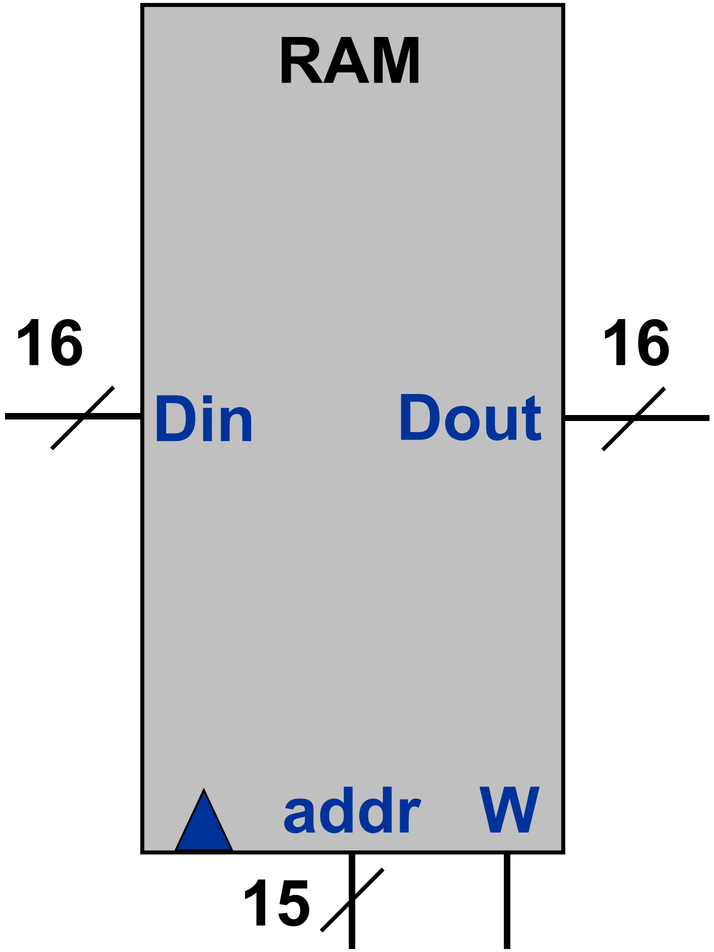
A 16-bit machine consisting of the following elements:



Both memory chips are 16-bit wide and have 15-bit address space.

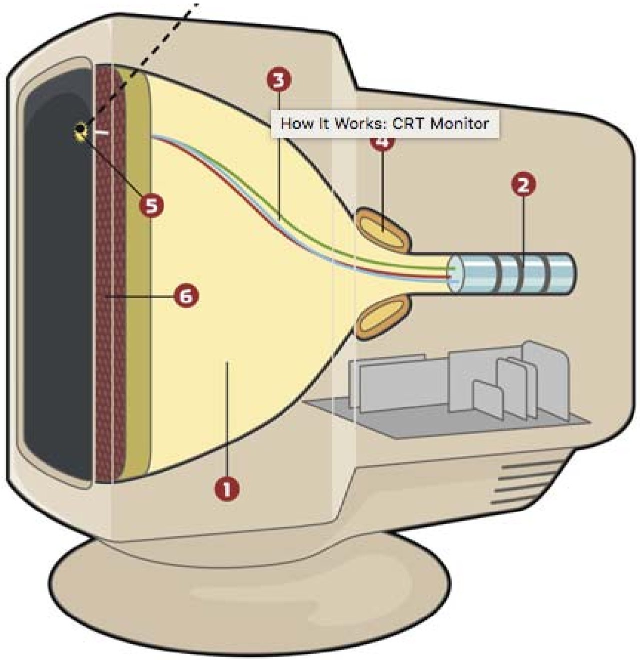
RAM (data memory)

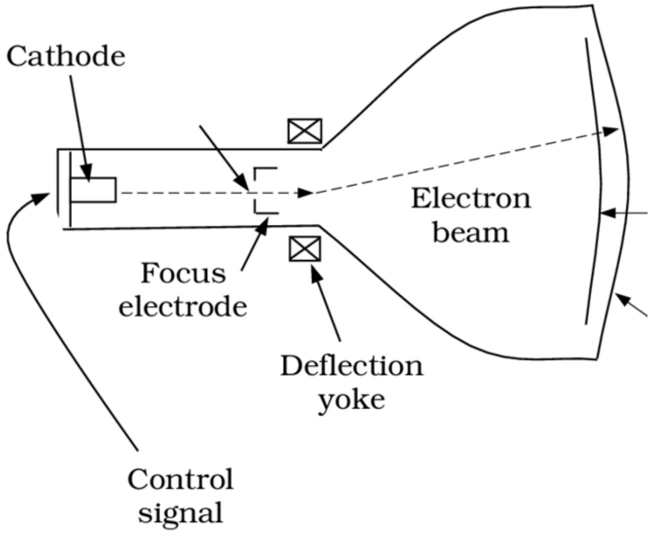
- The RAM used in Hack is different from a normal RAM. It also plays the role for I/O.
- Programmers usually use high-level library for I/O, such as printf, drawline.
- But, at low-level, we usually need to manipulate bits directly for I/O.



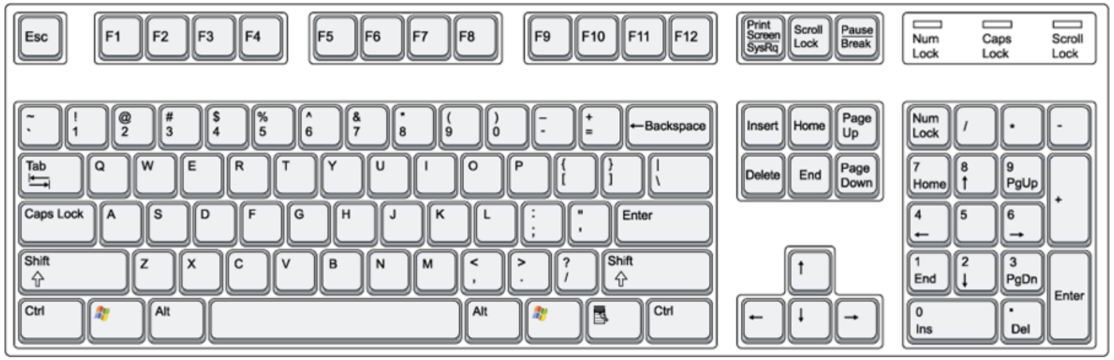
Displays

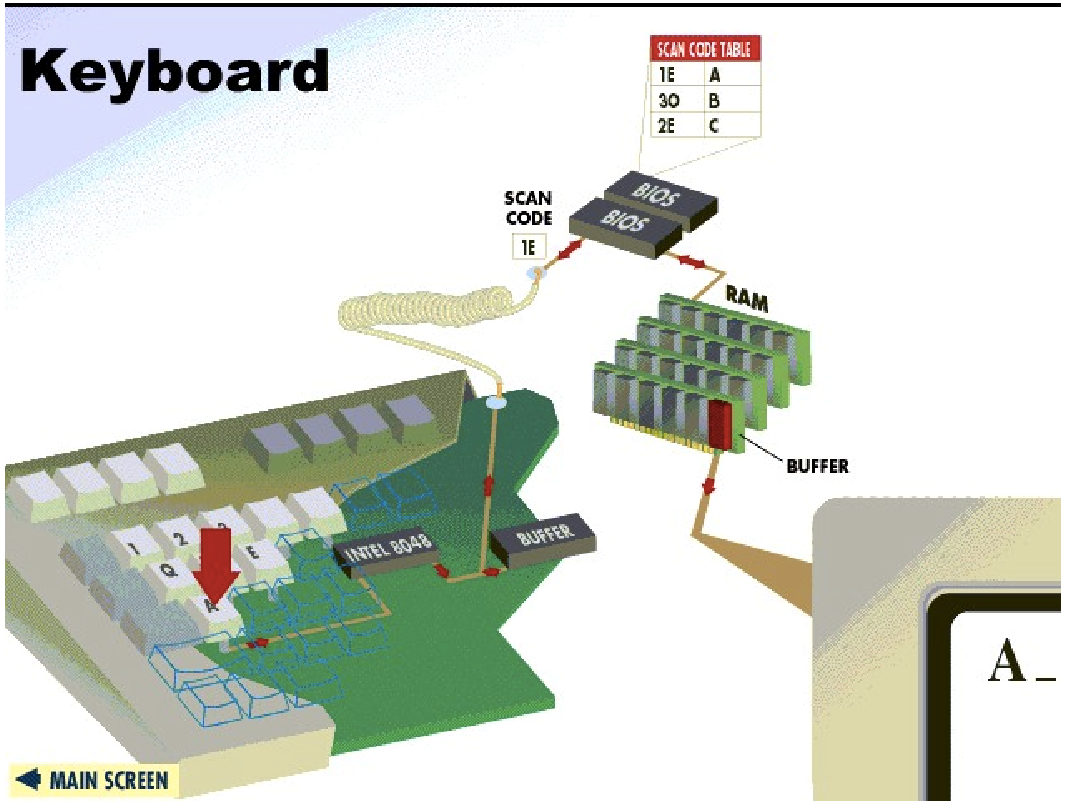
- CRT displays
  - resolution
  - refresh rate



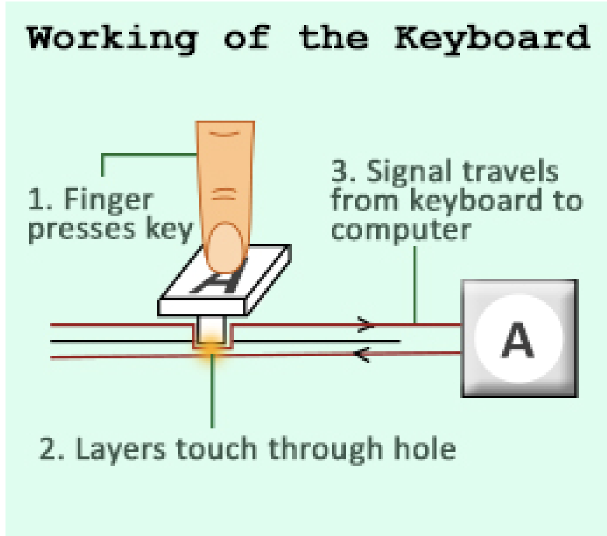


keyboard

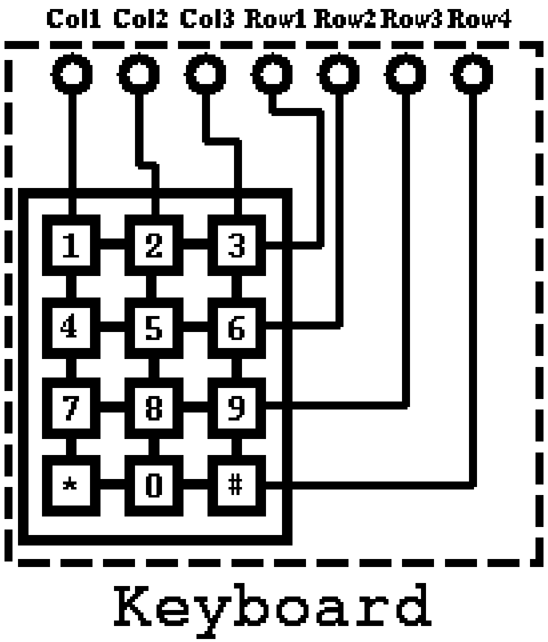




Keyboard
SCAN CODE TABLE
1E A
30 B
2E C
SCAN CODE
1E
BIOS
BIOS
RAM
BUFFER
INTEL 8048
BUFFER
A_
MAIN SCREEN

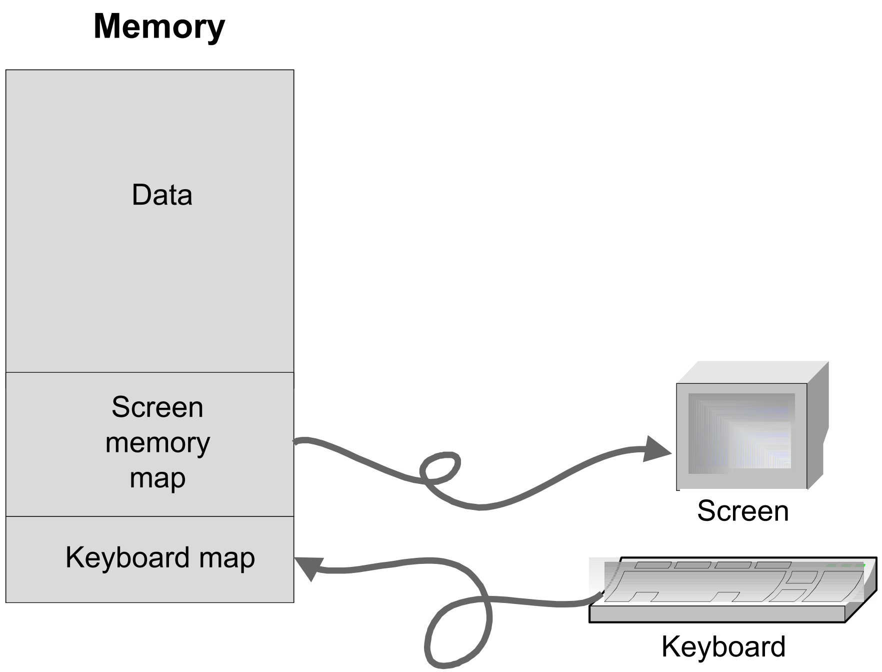


Working of the Keyboard
1. Finger presses key
3. Signal travels from keyboard to computer
A
2. Layers touch through hole



Col1 Col2 Col3 Row1 Row2 Row3 Row4
1 2 3
4 5 6
7 8 9
* 0 #
Keyboard

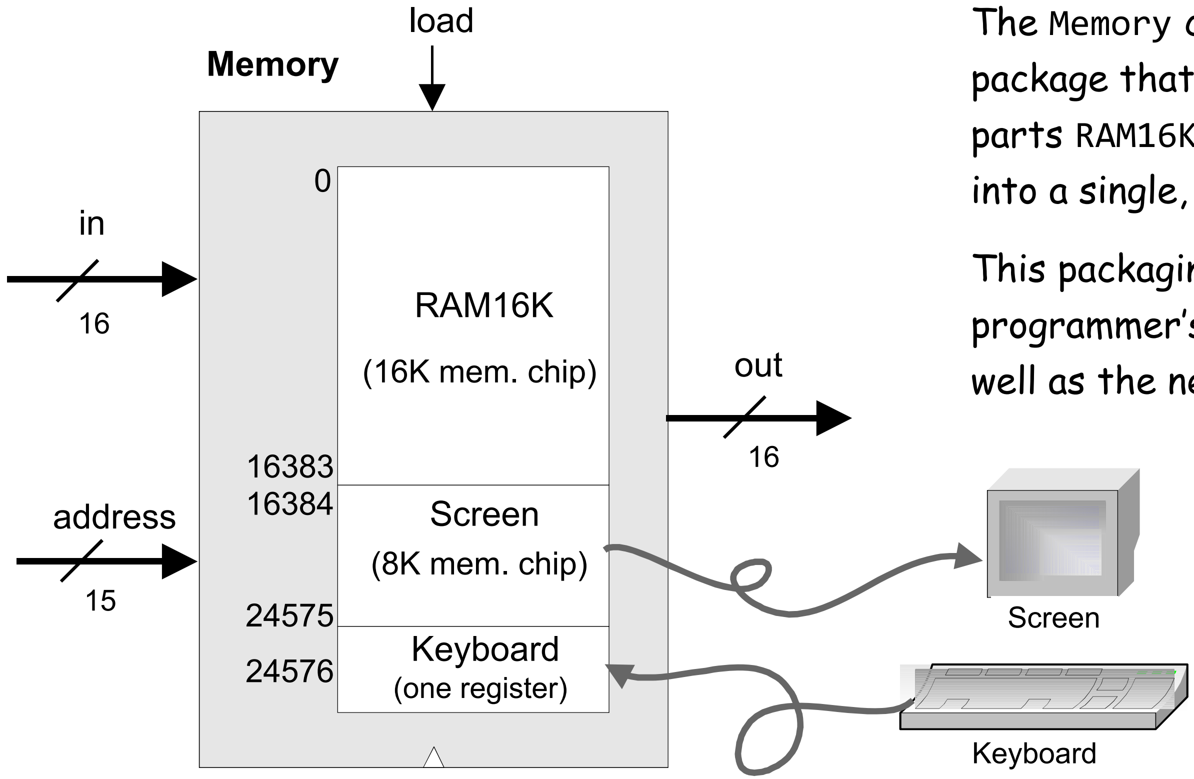
Memory: conceptual / programmer's view



Using the memory:

- To record or recall values (e.g. variables, objects, arrays), use the first 16K words of the memory
- To write to the screen (or read the screen), use the next 8K words of the memory
- To read which key is currently pressed, use the next word of the memory.

Memory: physical implementation



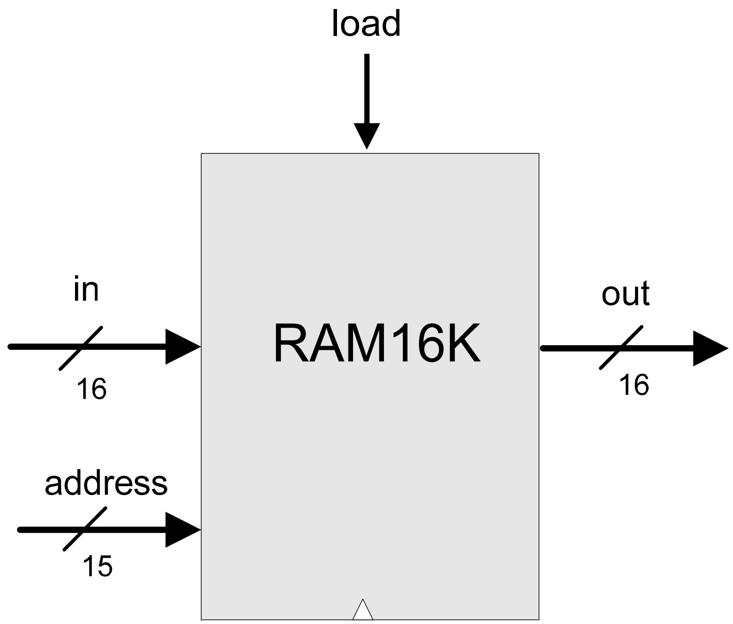
The `Memory` chip is essentially a package that integrates the three chip-parts `RAM16K`, `Screen`, and `Keyboard` into a single, contiguous address space.

This packaging effects the programmer's view of the memory, as well as the necessary I/O side-effects.

Access logic:

- Access to any address from 0 to 16,383 results in accessing the `RAM16K` chip-part
- Access to any address from 16,384 to 24,575 results in accessing the `Screen` chip-part
- Access to address 24,576 results in accessing the `keyboard` chip-part
- Access to any other address is invalid.

Data memory



Low-level (hardware) read/write logic:

To read RAM[k]: set address to k,
probe out

To write RAM[k]=x: set address to k,
set in to x,
set load to 1,
run the clock

High-level (OS) read/write logic:

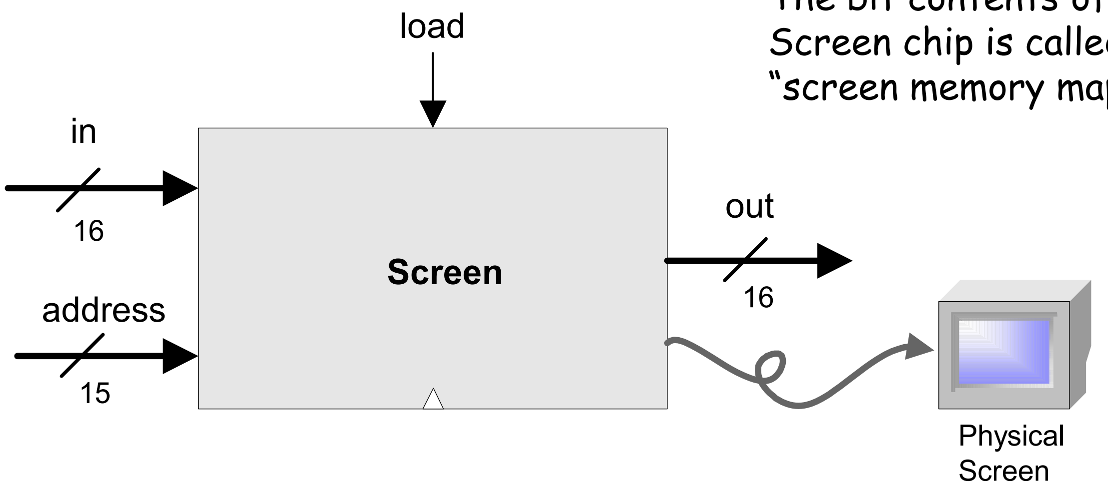
To read RAM[k]: use the OS command `out = peek(k)`

To write RAM[k]=x: use the OS command `poke(k,x)`

peek and poke are OS commands whose implementation should effect the same behavior as the low-level commands

More about peek and poke this later in the course, when we'll write the OS.

Screen



The bit contents of the Screen chip is called the "screen memory map"

In the Hack platform, the screen is implemented as an 8K 16-bit RAM chip with a side effect of refreshing.

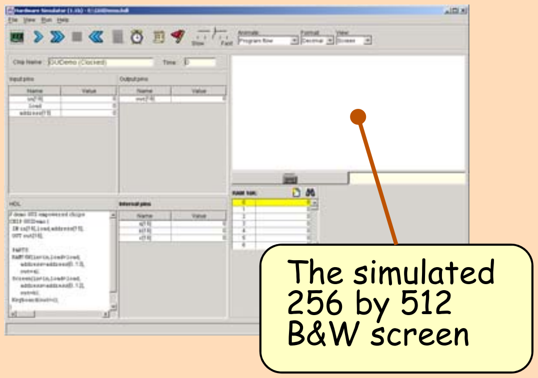
The Screen chip has a basic RAM chip functionality:

- read logic: `out = Screen[address]`
- write logic: `if load then Screen[address] = in`

Side effect:

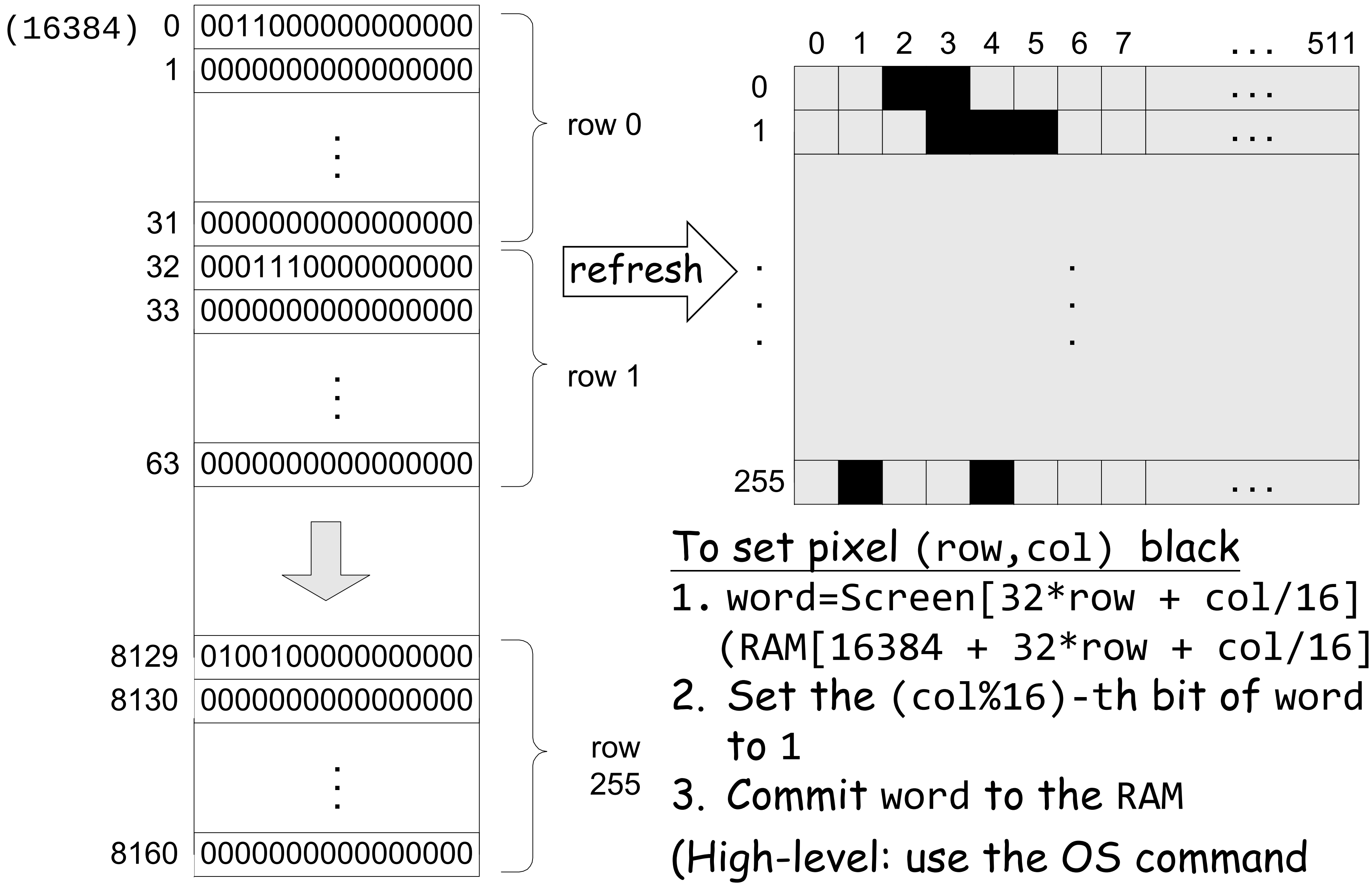
Continuously refreshes a 256 by 512 black-and-white screen device

Simulated screen:



When loaded into the hardware simulator, the built-in `Screen.hdl` chip opens up a screen window; the simulator then refreshes this window from the screen memory map several times each second.

Screen memory map



To set pixel (row,col) black

1. word=Screen[32*row + col/16]
   (RAM[16384 + 32*row + col/16])
2. Set the (col%16)-th bit of word to 1
3. Commit word to the RAM

(High-level: use the OS command drawPixel(row,col))

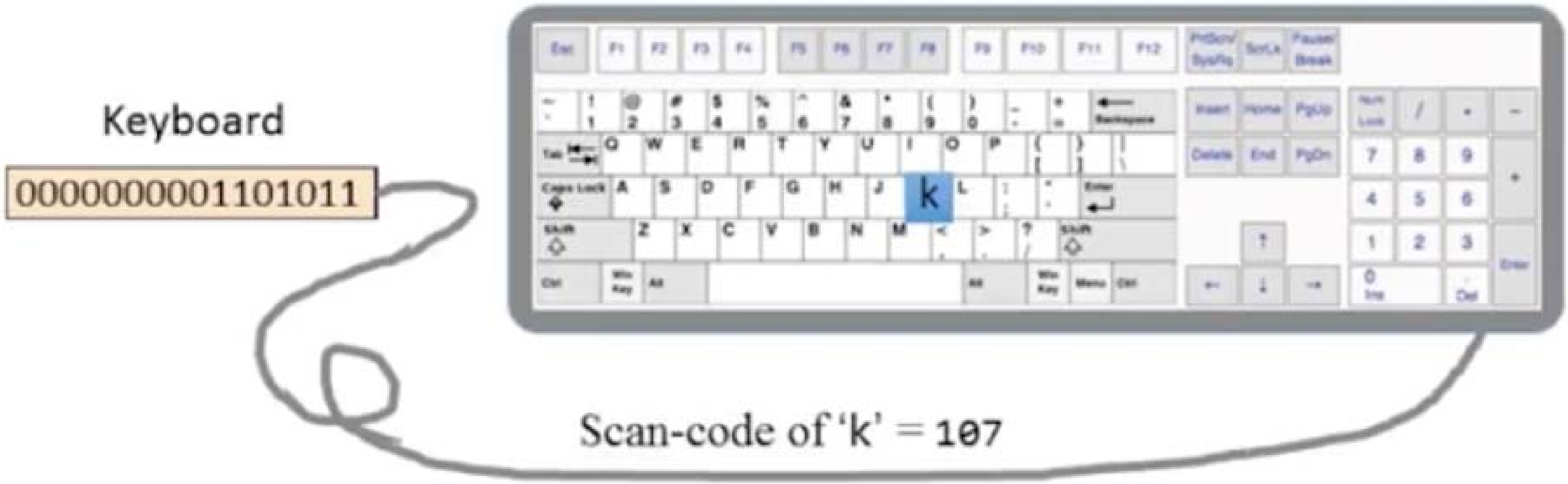
keyboard

- A 16-bit register is used to keep the key stroke.

When a key is pressed on the keyboard, the key's scan code appears in the keyboard memory map .

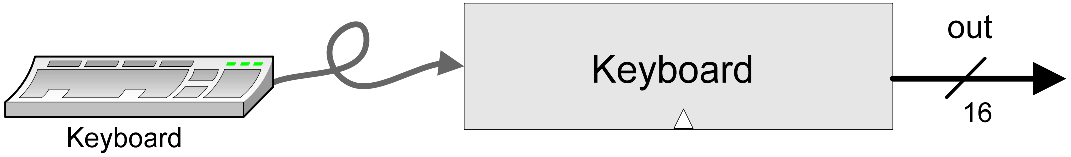
keyboard

- A 16-bit register is used to keep the key stroke.



When a key is pressed on the keyboard, the key's scan code appears in the keyboard memory map .

Keyboard



Keyboard chip: a single 16-bit register

Input: scan-code (16-bit value) of the currently pressed key, or 0 if no key is pressed

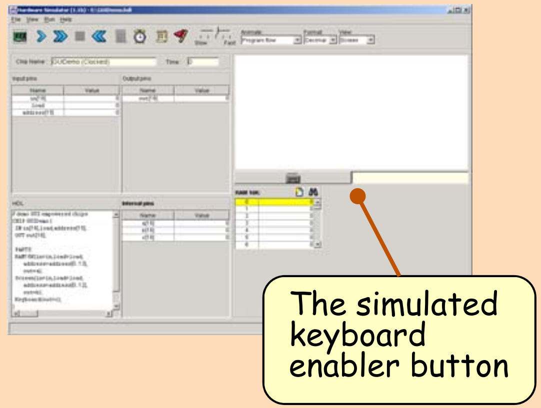
Output: same

Special keys:

Key pressed	Keyboard output
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134

Key pressed	Keyboard output
end	135
page up	136
page down	137
insert	138
delete	139
esc	140
f1-f12	141-152

Simulated keyboard:



The keyboard is implemented as a built-in **`Keyboard.hdl`** chip. When this java chip is loaded into the simulator, it connects to the regular keyboard and pipes the scan-code of the currently pressed key to the keyboard memory map.

How to read the keyboard:

- Low-level (hardware): probe the contents of the `Keyboard` chip
- High-level: use the OS command `keyPressed()` (effects the same operation, discussed later in the course, when we'll write the OS).

Some scan codes

Key	Code
0	48
1	49
...	...
9	57

Key	Code
A	65
B	66
...	...
Z	90

When no key is pressed, the resulting code is 0

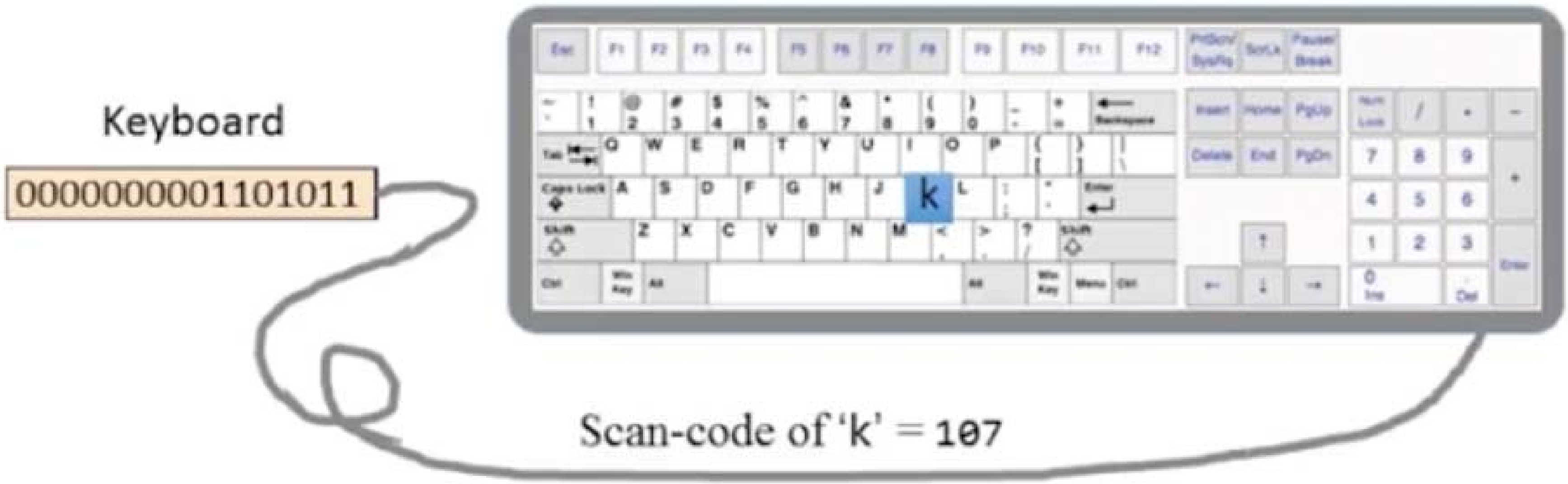
Key	Code
(space)	32
!	33
"	34
#	35
$	36
%	37
&	38
'	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

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

Key	Code
[	91
/	92
]	93
^	94
_	95

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

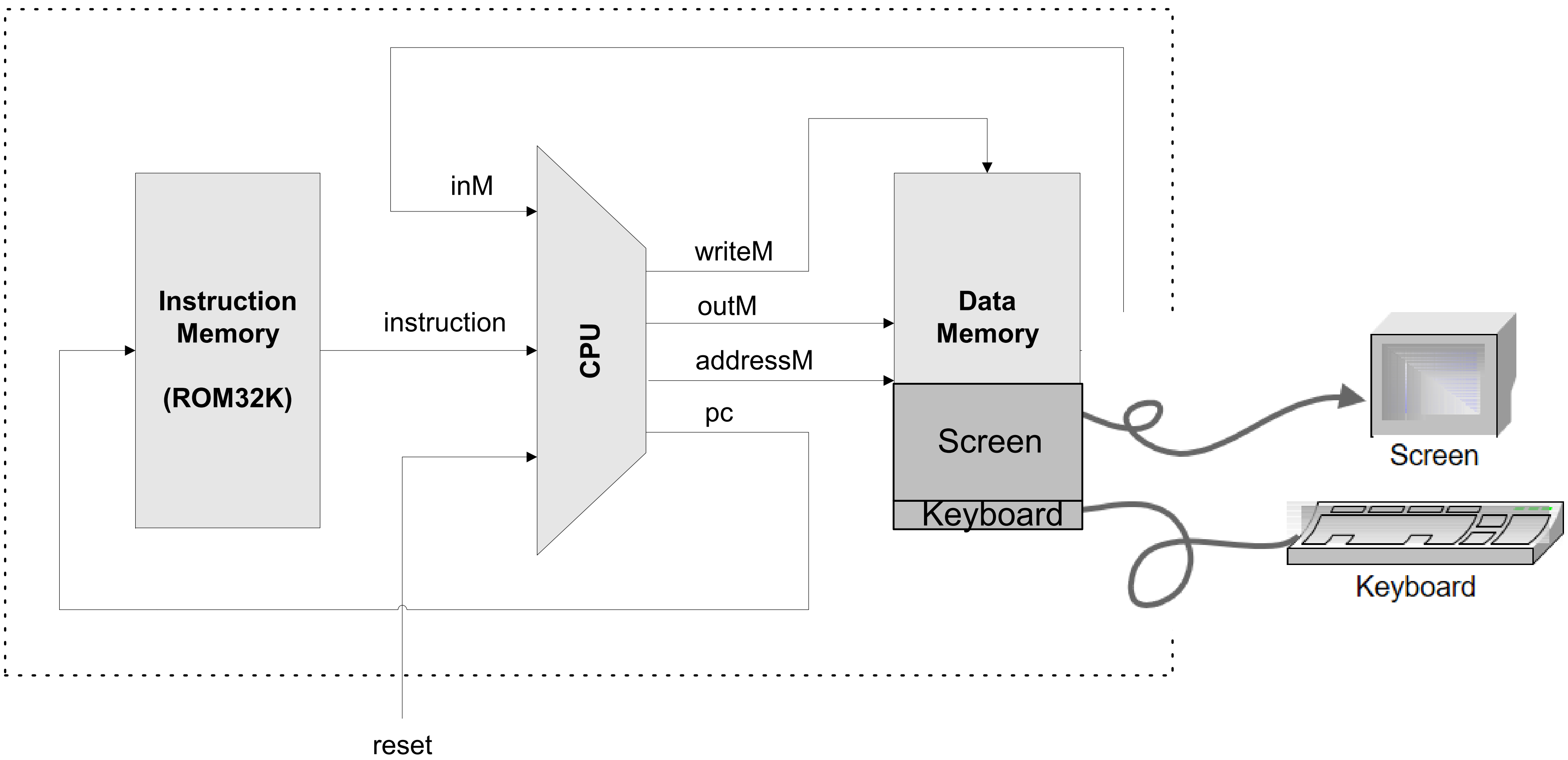
Keyboard memory map



- <u>To check which key is currently pressed:</u>
  - Probe the content of the `Keyboard` chip
  - In the Hack computer, probe the content of `RAM[24576]`
  - If the register contains `0`, no key is pressed.

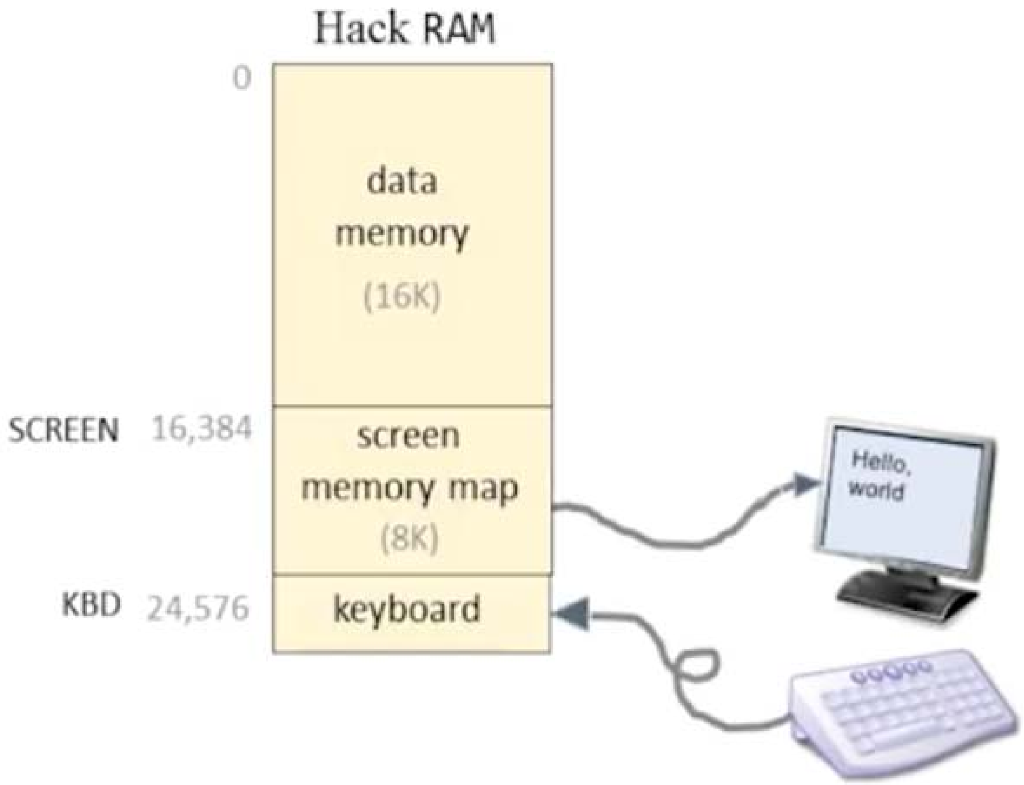
The Hack computer (put together)

A 16-bit machine consisting of the following elements:



Both memory chips are 16-bit wide and have 15-bit address space.

Assembly programming with I/O

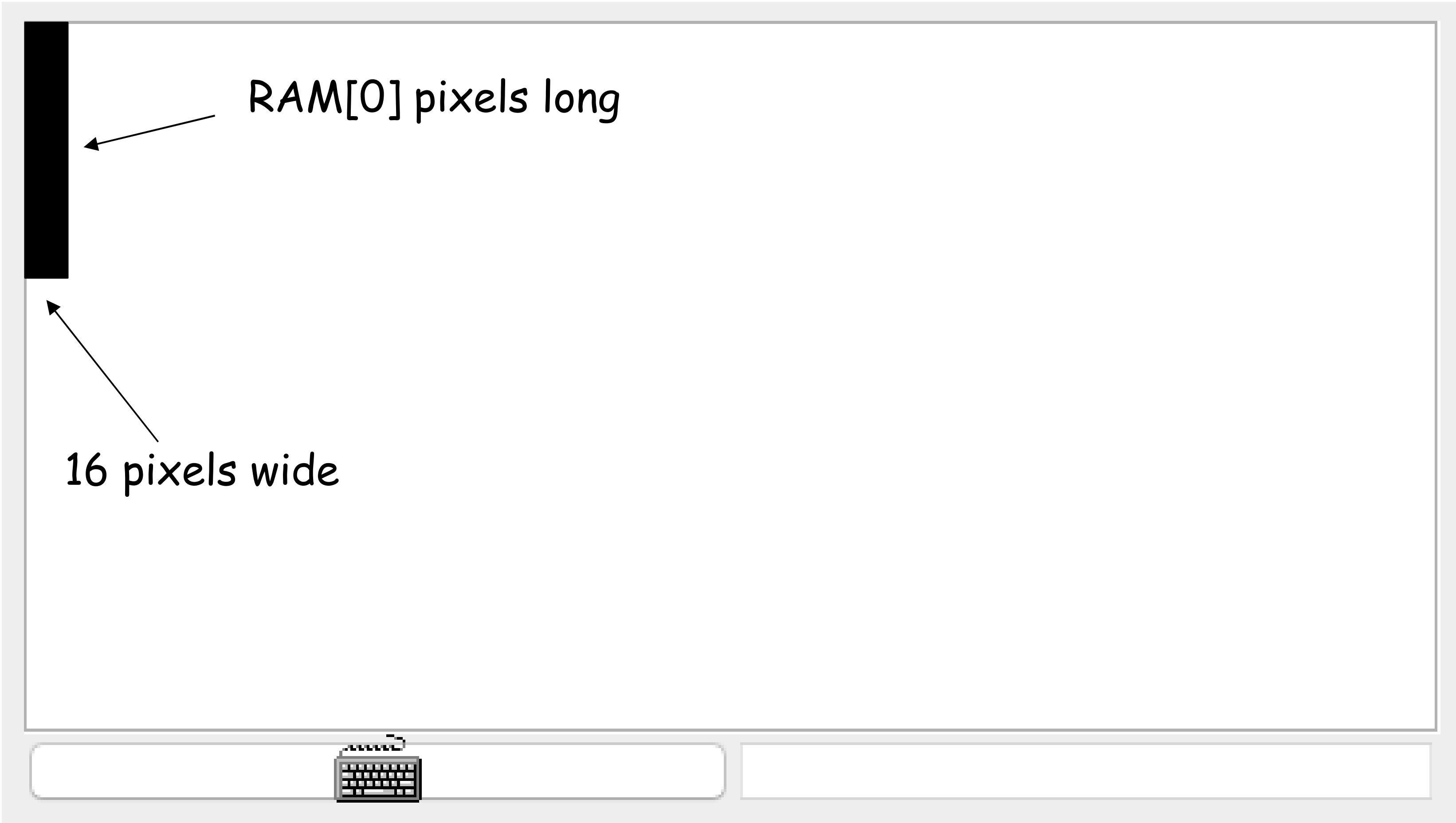


Hack language convention:

- SCREEN: base address of the screen memory map, 16,384.
- KBD: address of the keyboard memory map, 24,576.

Example: draw a rectangle

- Draw a filled rectangle at the upper left corner of the screen, 16 pixels wide and RAM[0] pixels long. (demo)



Example: draw a rectangle (pseudo code)

```
// for (i=0; i<n; i++)
//    draw 16 black pixels at the beginning of row i

addr = SCREEN
n = RAM[0]
i = 0


LOOP:
   if (i>n) goto END
   RAM[addr] = -1  // 1111 1111 1111 1111
   addr = addr+32  // advances to the next row
   i++;
   goto LOOP
END:
   goto END
```

Example: draw a rectangle (assembly)

```
    @SCREEN
    D=A
    @addr
    M=D         // addr = SCREEN

    @0
    D=M
    @n
    M=D         // n = RAM[0]

    @i
    M=0         // i=0
```

```
addr = SCREEN
n = RAM[0]
i = 0

LOOP:
   if (i>n) goto END
   RAM[addr] = -1
   addr = addr+32
   i++;
   goto LOOP
END:
   goto END
```

Example: draw a rectangle (assembly)

```
(LOOP)
    @i
    D=M
    @n
    D=D-M
    @END
    D; JGT

    @addr
    A=M
    M=-1
```

```
addr = SCREEN
n = RAM[0]
i = 0

LOOP:
  if (i>n) goto END
  RAM[addr] = -1
  addr = addr+32
  i++;
  goto LOOP
END:
  goto END
```

Example: draw a rectangle (assembly)

```
(LOOP)
    @i
    D=M
    @n
    D=D-M
    @END
    D; JGT

    @addr
    A=M
    M=-1
```

```
addr = SCREEN
n = RAM[0]
i = 0

LOOP:
  if (i>n) goto END
  RAM[addr] = -1
  addr = addr+32
  i++;
  goto LOOP
END:
  goto END
```

Example: draw a rectangle (assembly)

```
    @32
    D=A
    @addr
    M=D+M    // addr = addr+32

    @i
    M=M+1    // i++

    @LOOP
    0; JMP   // goto LOOP

(END)
    @END
    0; JMP
```

```
addr = SCREEN
n = RAM[0]
i = 0

LOOP:
  if (i>n) goto END
  RAM[addr] = -1
  addr = addr+32
  i++;
  goto LOOP
END:
  goto END
```

Example: draw a rectangle (assembly)

```
    @32
    D=A
    @addr
    M=D+M    // addr = addr+32

    @i
    M=M+1    // i++

    @LOOP
    0; JMP   // goto LOOP

(END)
    @END
    0; JMP
```

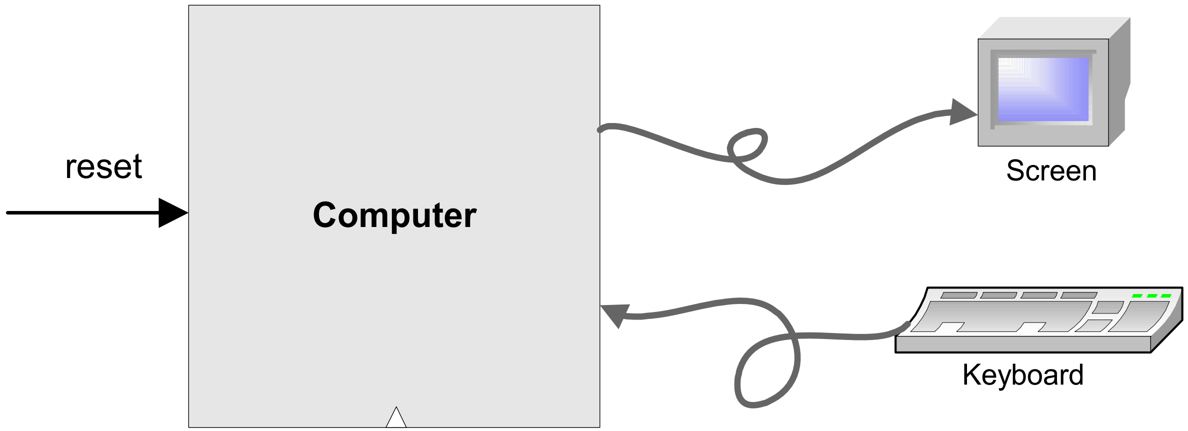
```
addr = SCREEN
n = RAM[0]
i = 0

LOOP:
  if (i>n) goto END
  RAM[addr] = -1
  addr = addr+32
  i++;
  goto LOOP
END:
  goto END
```

Example: draw a rectangle (assembly)

```
    @32
    D=A
    @addr
    M=D+M   // addr = addr+32

    @i
    M=M+1   // i++

    @LOOP
    0; JMP  // goto LOOP

(END)
    @END
    0; JMP
```

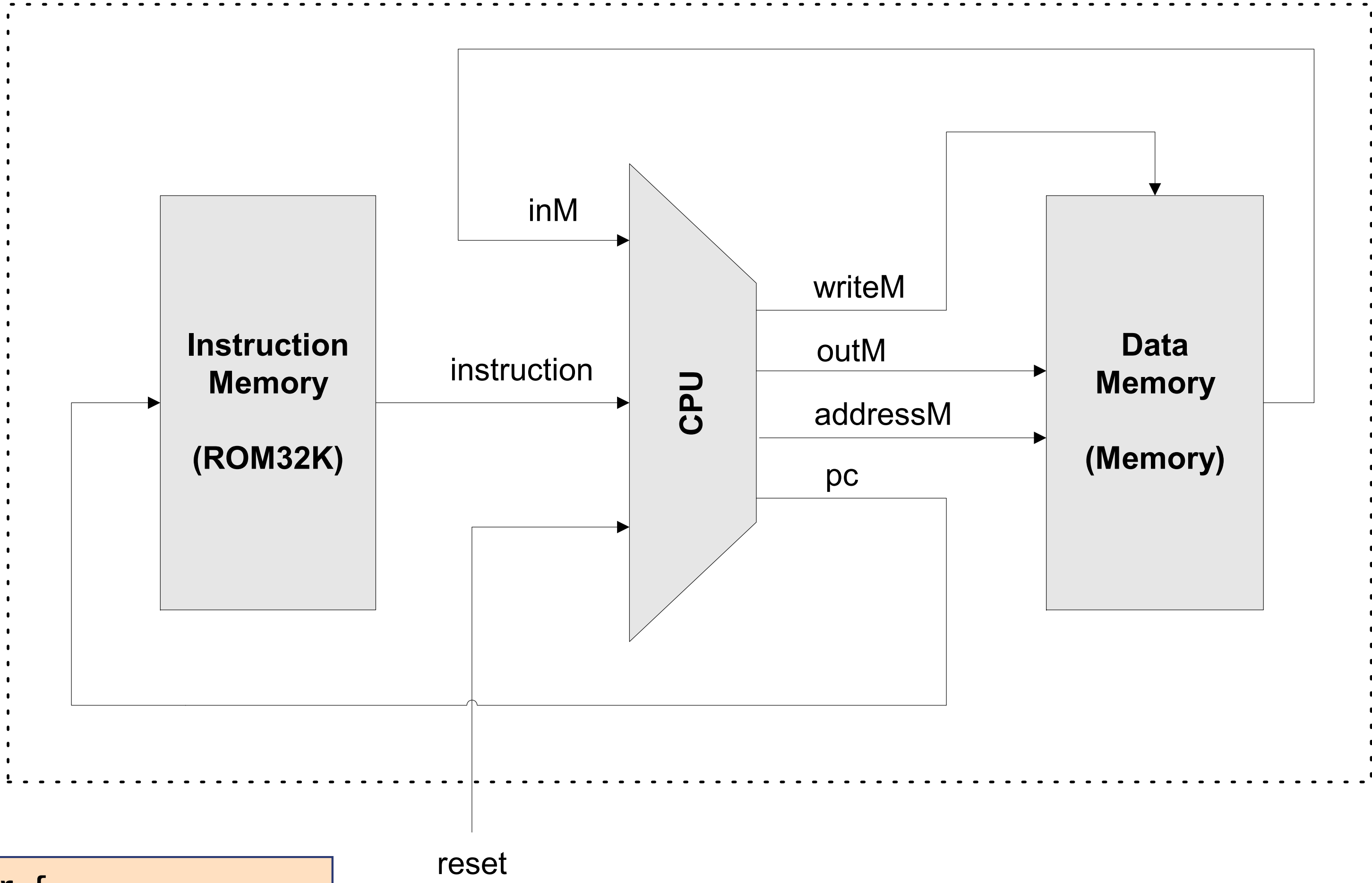
```
addr = SCREEN
n = RAM[0]
i = 0

LOOP:
  if (i>n) goto END
  RAM[addr] = -1
  addr = addr+32
  i++;
  goto LOOP
END:
  goto END
```

Project #5: Computer-on-a-chip interface




```
Chip Name: Computer  // Topmost chip in the Hack platform
Input:     reset
Function:  When reset is 0, the program stored in the
           computer's ROM executes. When reset is 1, the
           execution of the program restarts. Thus, to start a
           program's execution, reset must be pushed "up" (1)
           and "down" (0).

           From this point onward the user is at the mercy of
           the software. In particular, depending on the
           program's code, the screen may show some output and
           the user may be able to interact with the computer
           via the keyboard.
```

Computer-on-a-chip implementation




```
CHIP Computer {
    IN reset;
    PARTS:
    // implementation missing
}
```

Implementation:

- You need to implement Memory and CPU first.
- Simple, the chip-parts do all the hard work.

Perspective: from here to a “real” computer

- Caching
- More I/O units
- Special-purpose processors (I/O, graphics, communications, …)
- Multi-core / parallelism
- Efficiency
- Energy consumption considerations
- And more ...