



Design of Hack Computer

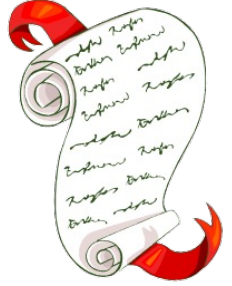
```
0:  b8 01 00 00 00
5:  bf 01 00 00 00
a:  48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```

Instructor: Muhammad Arif Butt, Ph.D.





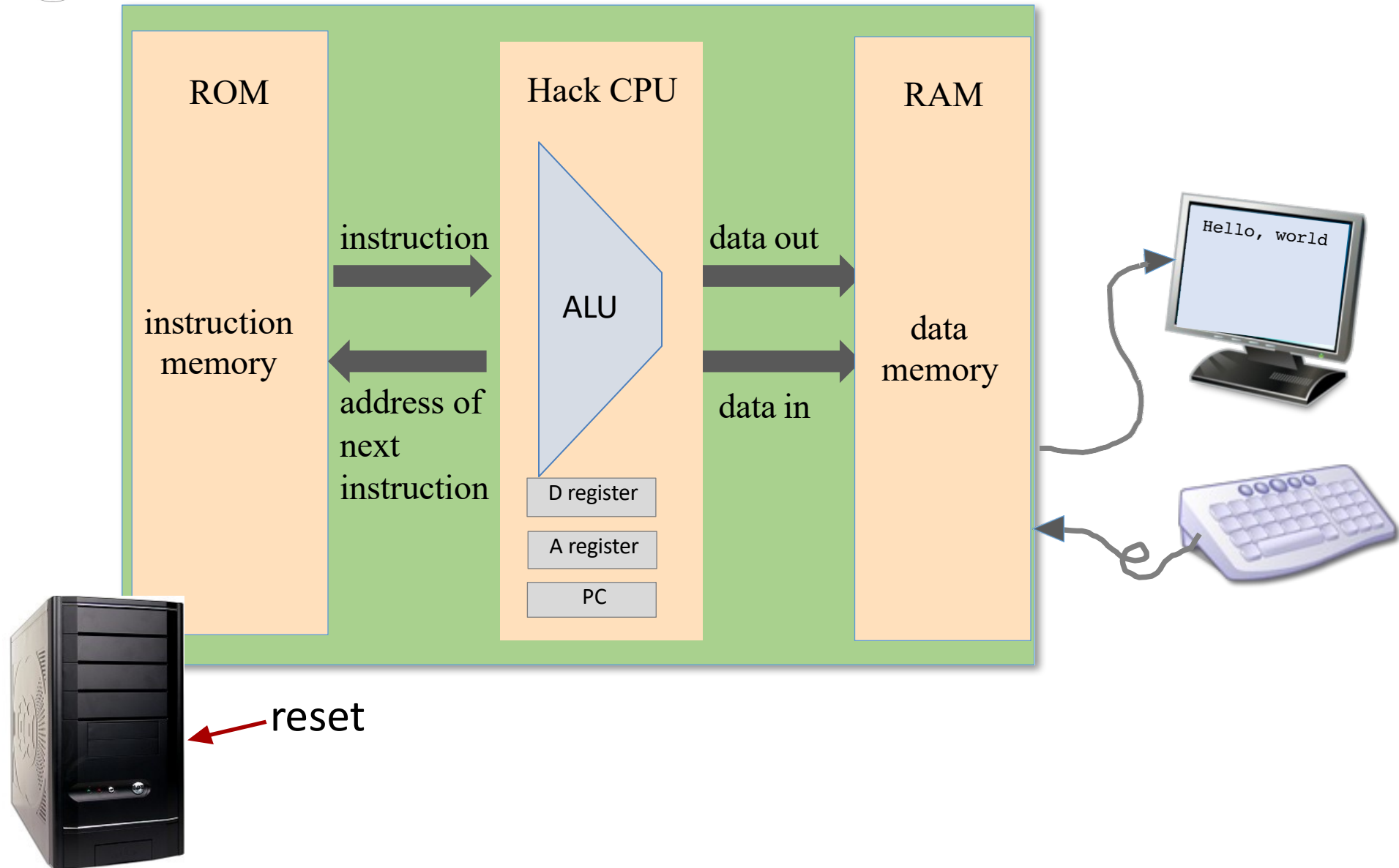
Today's Agenda



- Recap of Hack Computer Architecture
- Implementation of Hack CPU Chip (**CPU.hdl**)
- Implementation of Hack Memory Chip (**Memory.hdl**)
 - RAM16 chip (**RAM16K.hdl**)
 - Screen chip (**Screen.hdl**)
 - Keyboard chip (**Keyboard.hdl**)
- Implementation of Hack ROM Chip (**ROM32K.hdl**)
- Implementation of Hack Computer Chip (**Computer.hdl**)



Recap: Hack Computer Architecture

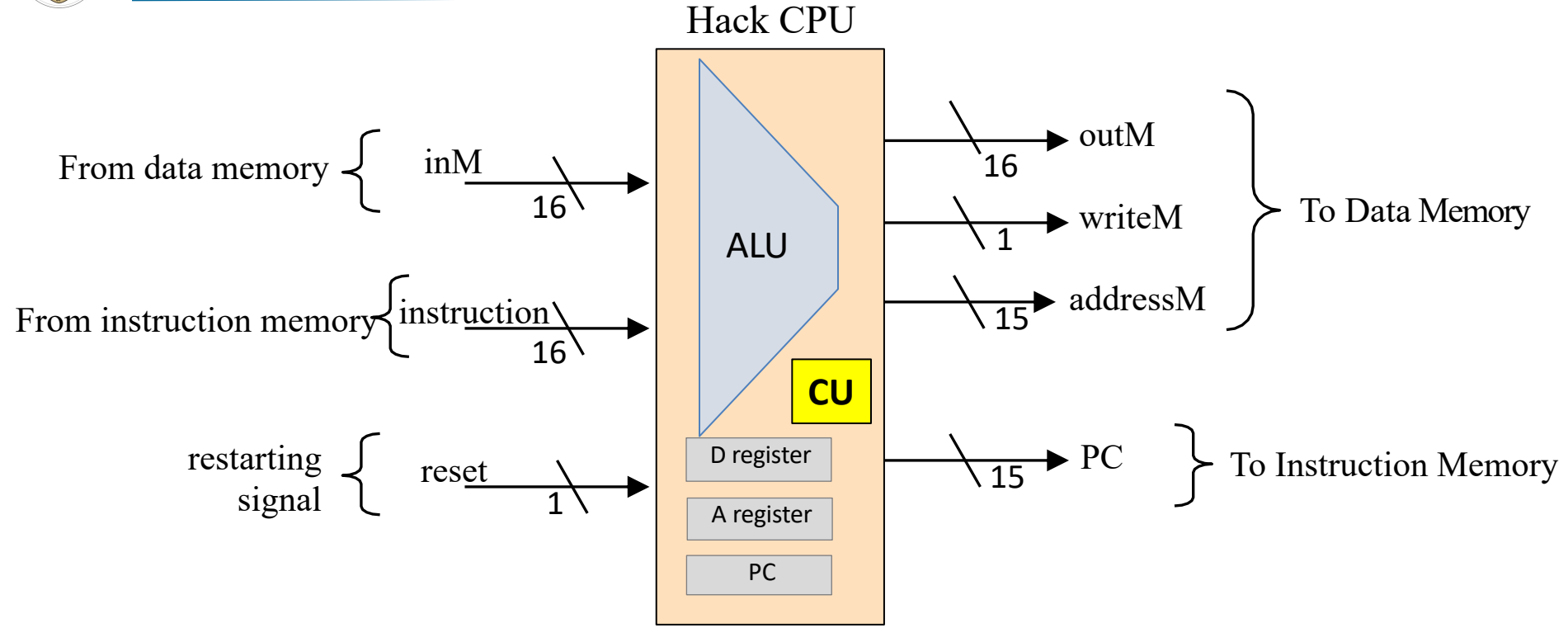




Implementation of Hack CPU Chip



Hack CPU Interface (Recap)



Inputs:

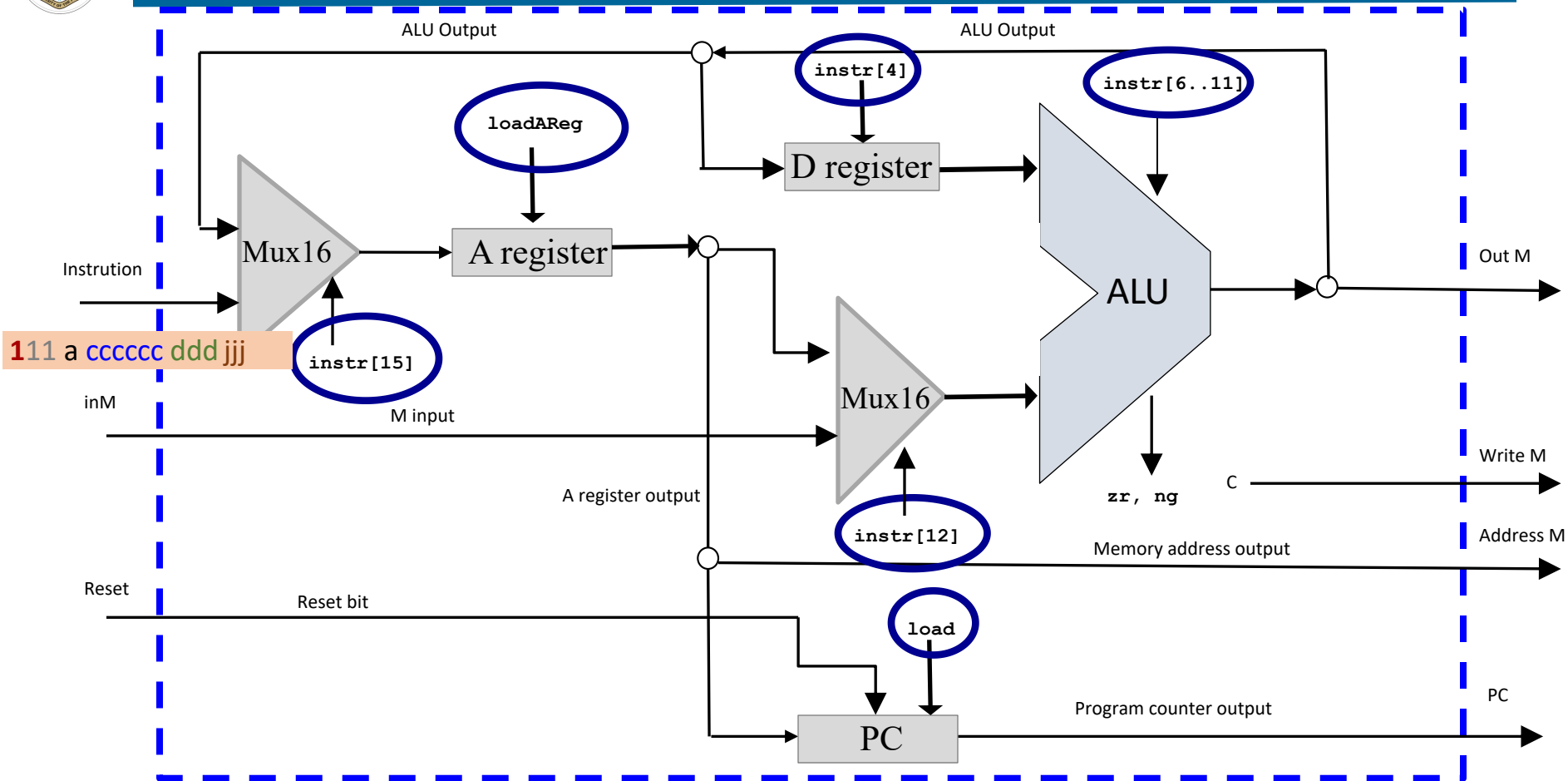
- Data Value
- Instruction
- Reset Bit

Outputs:

- Data Value
- Write to Memory? (yes/no)
- Memory Address
- Address of next instruction

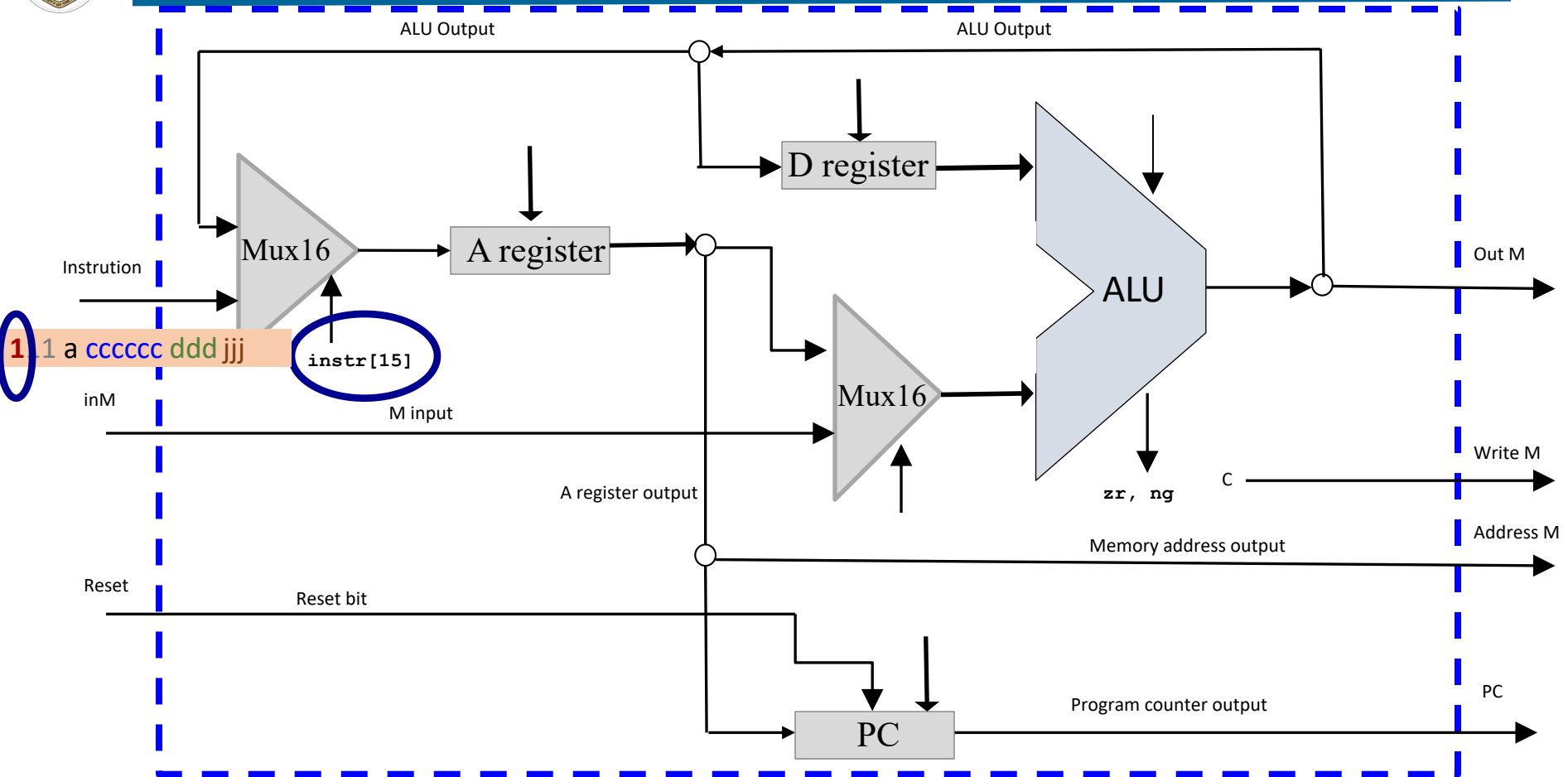


Chip Parts of Hack CPU and their Control



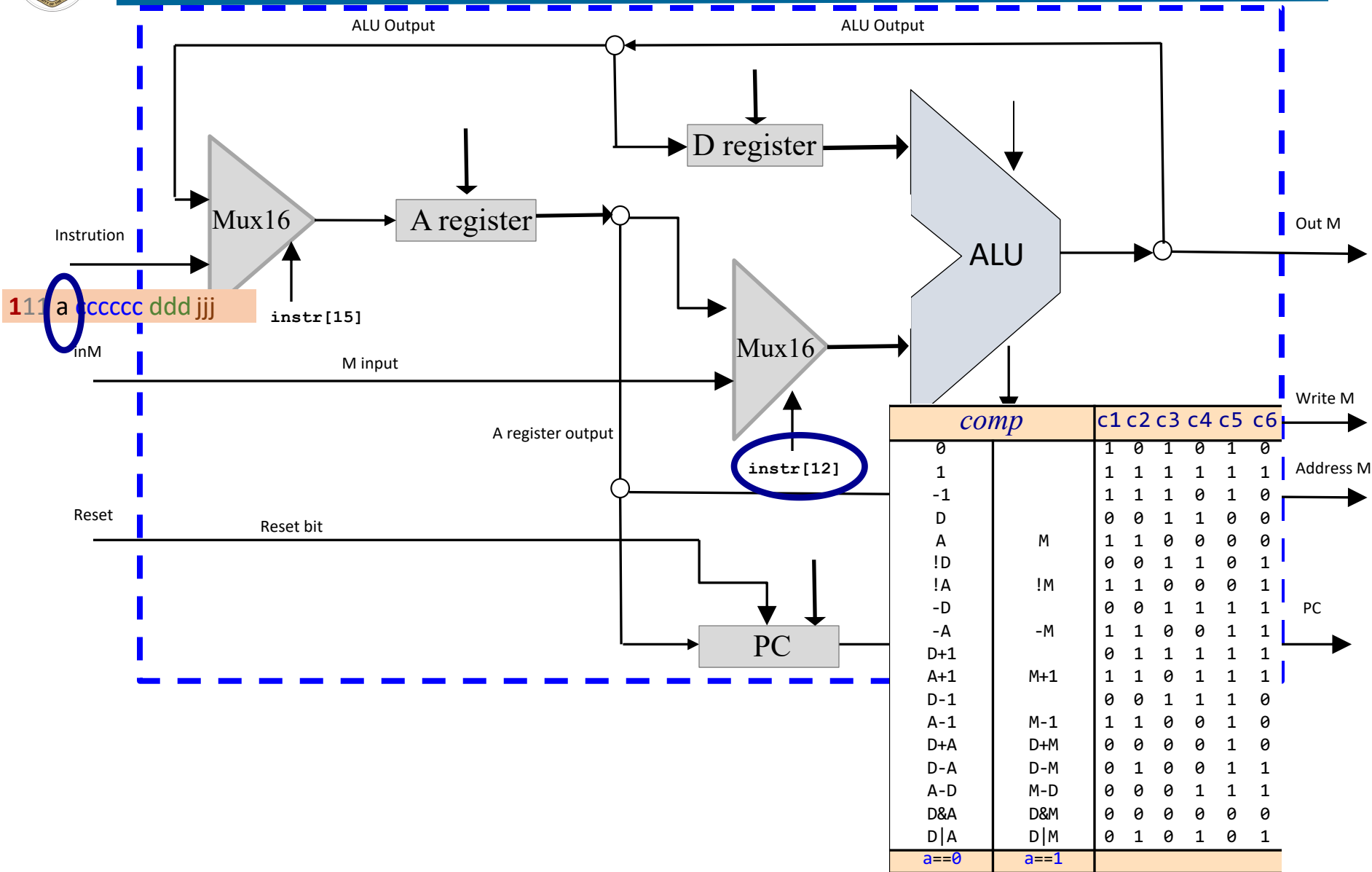


Select Input of First Mux16 Chip (Recap)





Select Input of Second Mux16 Chip (Recap)

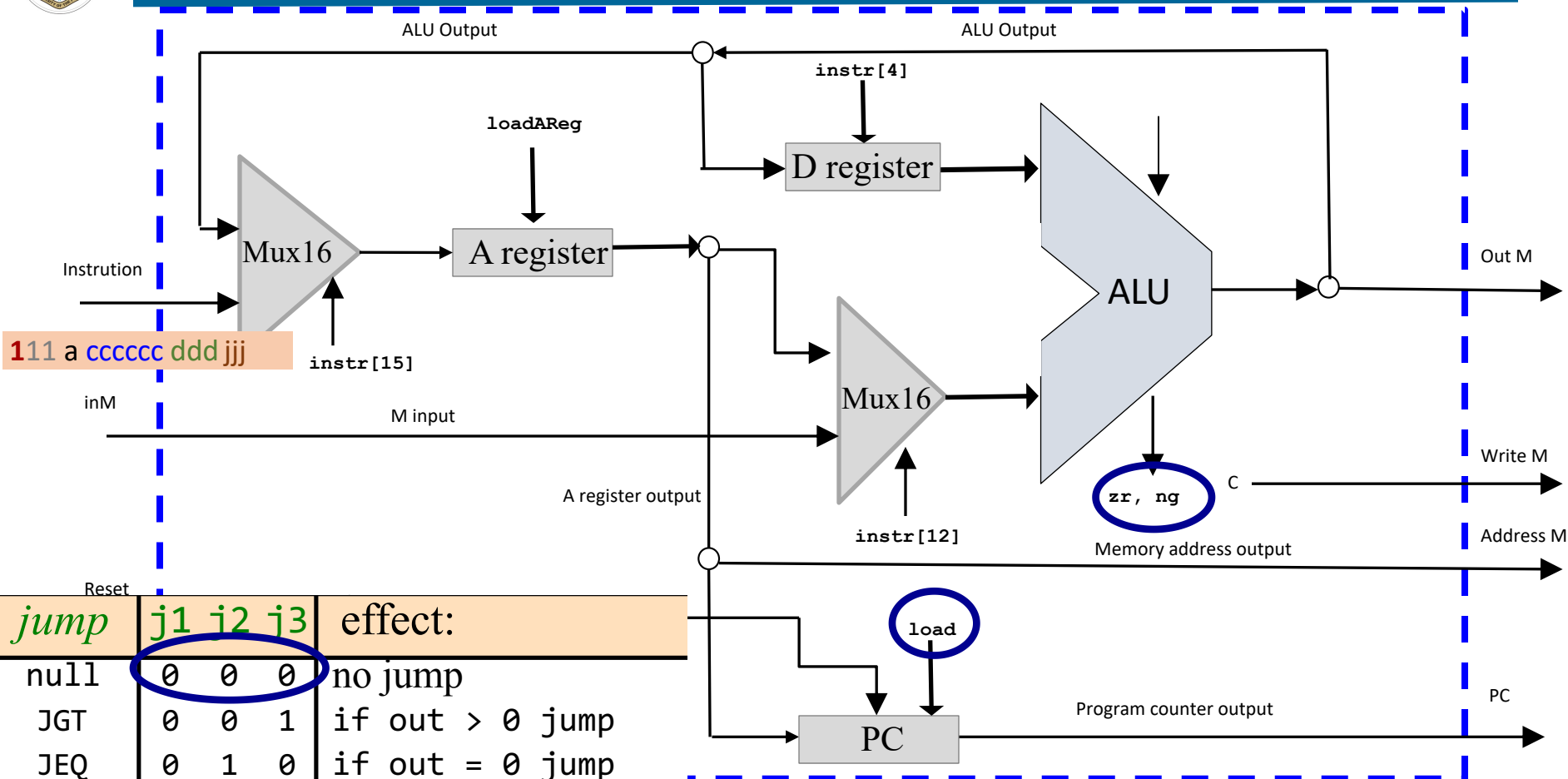








Load Input of PC-Register (Recap)

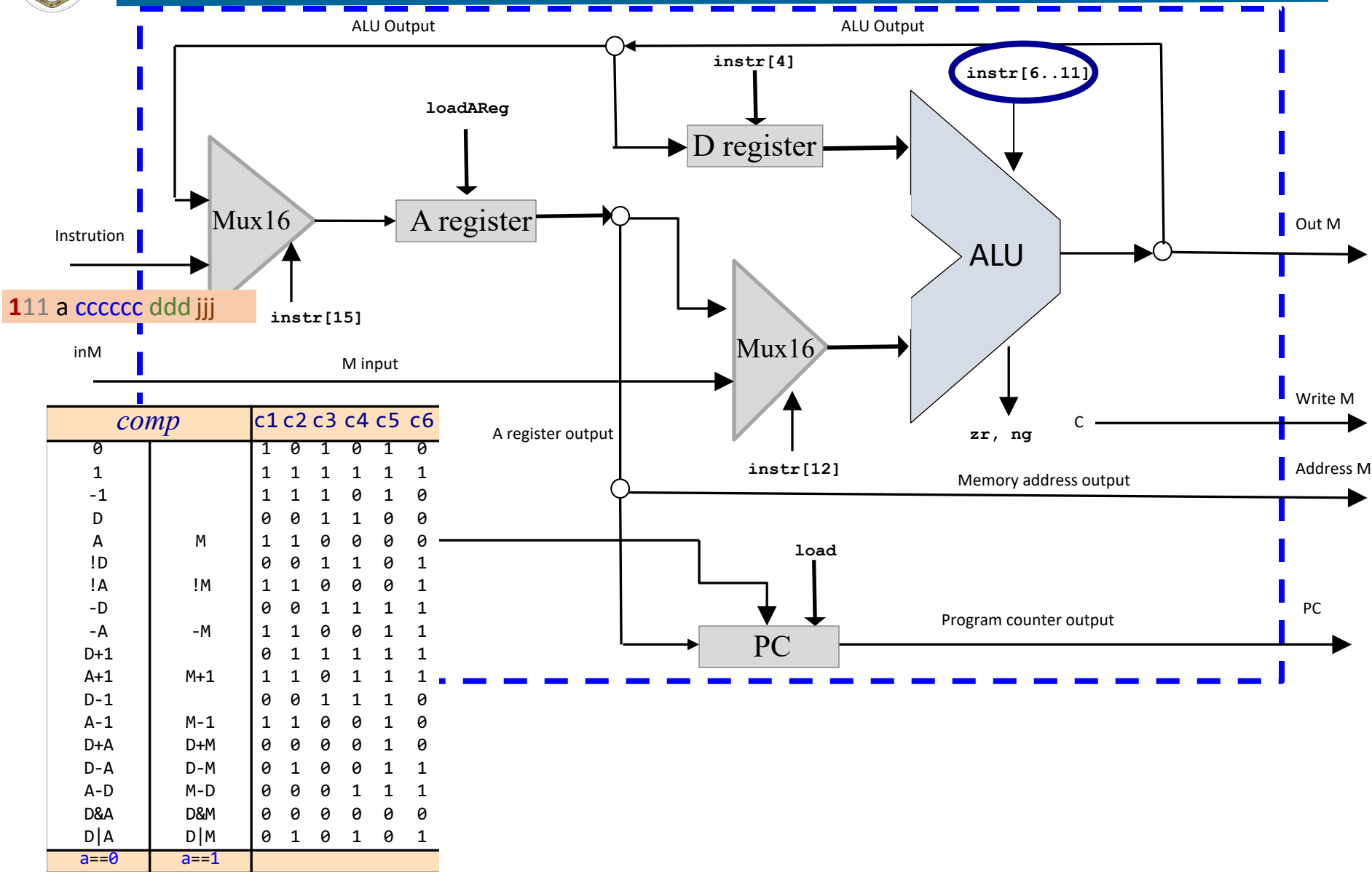


Reset	jump			effect:
	j1	j2	j3	
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

load = f(jump bits, zr/ng bits)
 if (load == 1)
 PC = A



Control Input of ALU Chip (Recap)





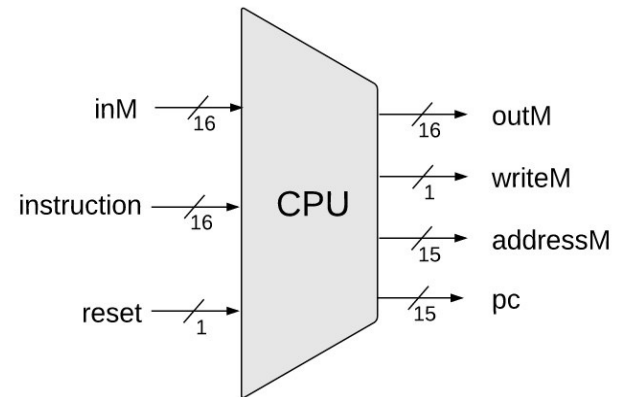
CPU Implementation

CPU.hdl

```
/**
The Hack CPU (Central Processing unit), consisting of an ALU, two registers named A and D, and a
program counter named PC. The CPU is designed to fetch and execute instructions written in the
Hack machine language. In particular, functions as follows:
Executes the inputted instruction according to the Hack machine language specification. The D
and A in the language specification refer to CPU-resident registers, while M refers to the
external memory location addressed by A, i.e. to Memory[A]. The inM input holds the value of
this location. If the current instruction needs to write a value to M, the value is placed in
outM, the address of the target location is placed in the addressM output, and the writeM
control bit is asserted. (When writeM==0, any value may appear in outM). The outM and writeM
outputs are combinational: they are affected instantaneously by the execution of the current
instruction. The addressM and pc outputs are clocked: although they are affected by the
execution of the current instruction, they commit to their new values only in the next time
step. If reset==1 then the CPU jumps to address 0 (i.e. pc is set to 0 in next time step) rather
than to the address resulting from executing the current instruction.
*/
CHIP CPU {
    IN inM[16],          // M value input (M = contents of 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],        // M value output
    writeM,              // Write to M?
    addressM[15],        // Address in data memory (of M)
    pc[15];              // address of next instruction

    PARTS:
    // Chip implementation code on next slide
}
```





CPU Implementation

CPU.hdl (cont...)

PARTS:

```
Not(in=instruction[15],out=Ainst);
Not(in=Ainst,out=Cinst);
Mux16(a=instruction, b=ALUout, sel=Cinst, out=ARegBefore);
And(a=instruction[5], b=Cinst, out=d1); // instruction[5] = d1 if it's a C instruction
Or(a=d1, b=Ainst, out=storeAReg);
ARegister(in=ARegBefore, load=storeAReg, out=A, out[0..14]=addressM);
Mux16(a=A, b=inM, sel=instruction[12], out=MOrA); // instruction[12] = a
And(a=instruction[4], b=Cinst, out=d2); // instruction[4] = d2 if it's a C instruction
DRegister(in=ALUout, load=d2, out=D);

ALU(x=D, y=MOrA, zx=instruction[11], nx=instruction[10], zy=instruction[9],
    ny=instruction[8], f=instruction[7], no=instruction[6], out=ALUout, out=outM, zr=zr, ng=ng);

And(a=instruction[3], b=Cinst, out=writeM); // instruction[2] = d3 if it's a C instruction
Not(in=zr, out=notzr);
Not(in=ng, out=notng);
And(a=notzr, b=notng, out=pos);
And(a=instruction[2], b=ng, out=jneg);
And(a=instruction[1], b=zr, out=jzer);
And(a=instruction[0], b=pos, out=jpos);
Or(a=jneg, b=jzer, out=jzerneg);
Or(a=jzerneg, b=jpos, out=jumpIfCinst);
And(a=jumpIfCinst, b=Cinst, out=jump);

PC(in=A, load=jump, inc=true, reset=reset, out[0..14]=pc);
}
```



Hack CPU Chip

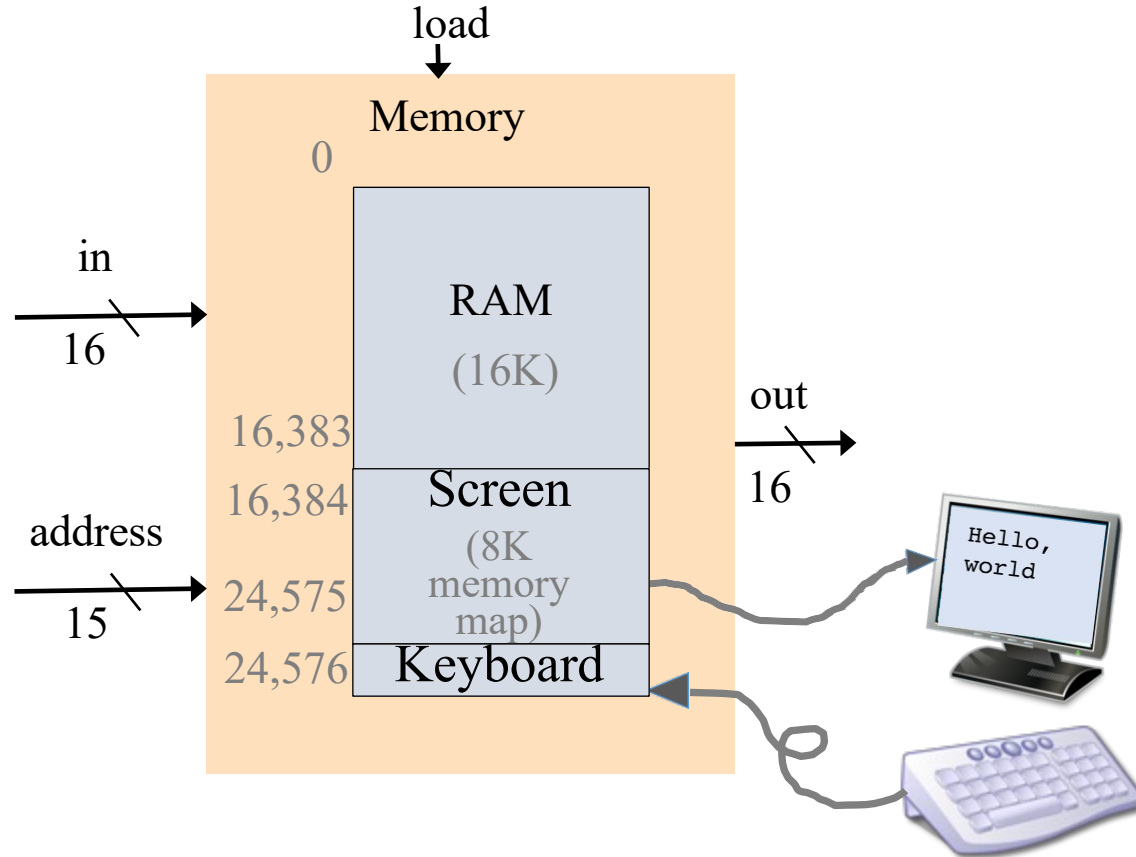




Implementation of Hack Data Memory Chip (RAM)



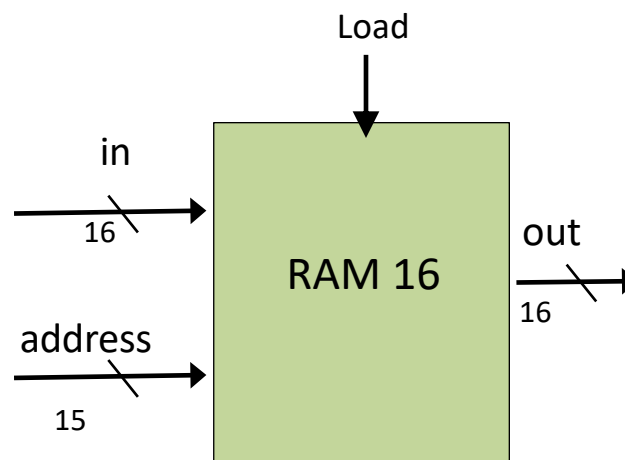
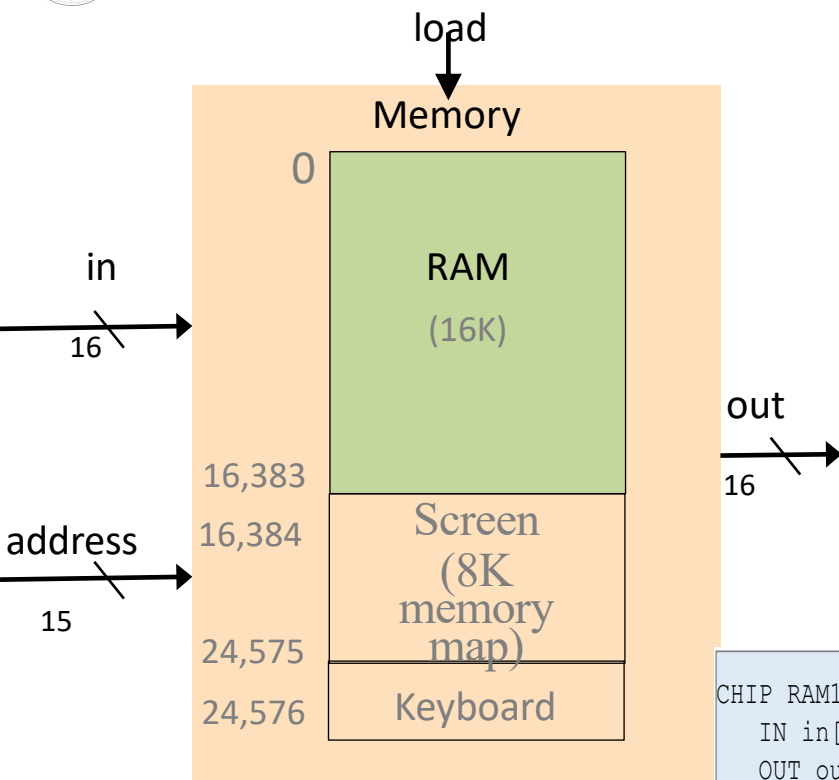
Memory Chip



- **RAM/Data Memory:** 16 bit, 16K RAM Chip (address 0 to 16383)
- **Screen Memory Map:** 16 bit, 8K memory Chip with a raster display side effect (16384 to 24575)
- **Keyboard Memory Map:** 16 bit register with a keyboard side effect (address 24576)



RAM16 Chip Implementation

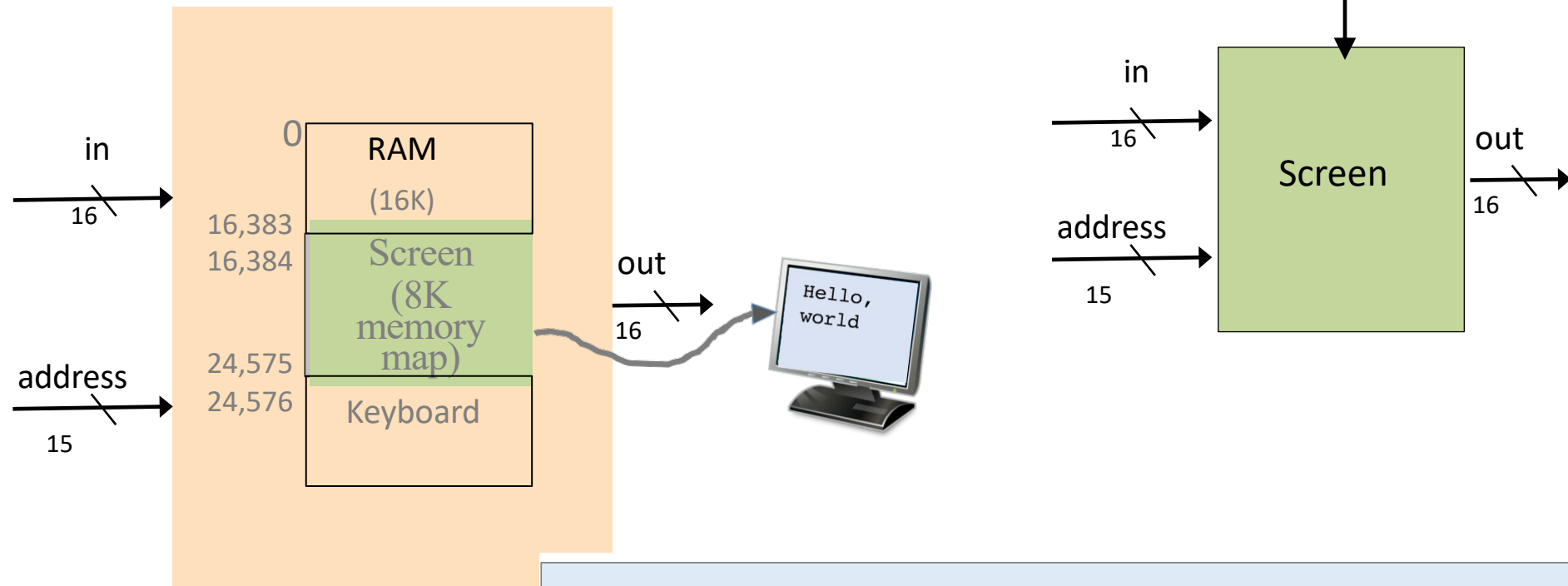


```
CHIP RAM16K {  
  IN in[16], load, address[14];  
  OUT out[16];  
  PARTS:  
    DMux4Way(in=load, sel=address[12..13], a=load0, b=load1, c=load2, d=load3);  
  
    RAM4K(in=in, load=load0, address=address[0..11], out=out0);  
    RAM4K(in=in, load=load1, address=address[0..11], out=out1);  
    RAM4K(in=in, load=load2, address=address[0..11], out=out2);  
    RAM4K(in=in, load=load3, address=address[0..11], out=out3);  
  
    Mux4Way16(a=out0, b=out1, c=out2, d=out3, sel=address[12..13], out=out);  
}
```



Screen Built-in Chip

Memory

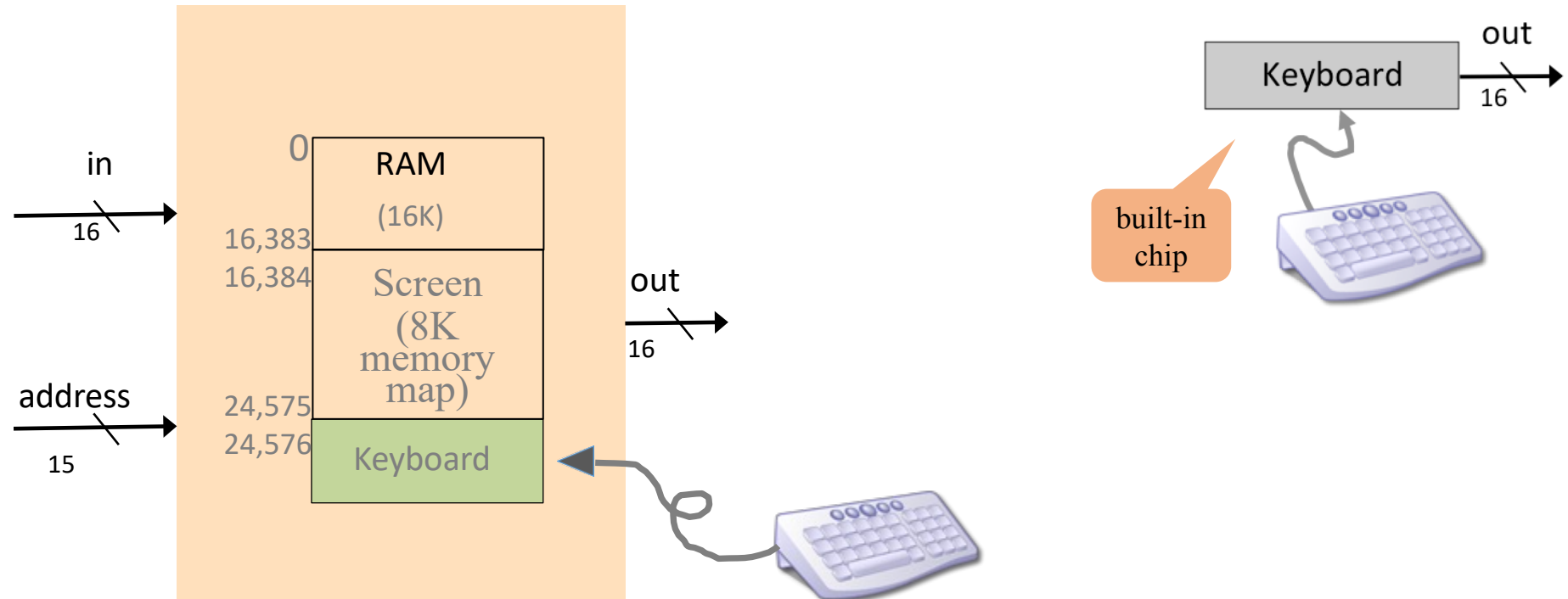


```
CHIP Screen {  
    IN in[16],    // what to write  
    load,        // write-enable bit  
    address[13]; // where to read/write  
    OUT out[16]; // Screen value at the given address  
    BUILTIN Screen;  
    CLOCKED in, load;  
}
```



Keyboard Built-in Chip

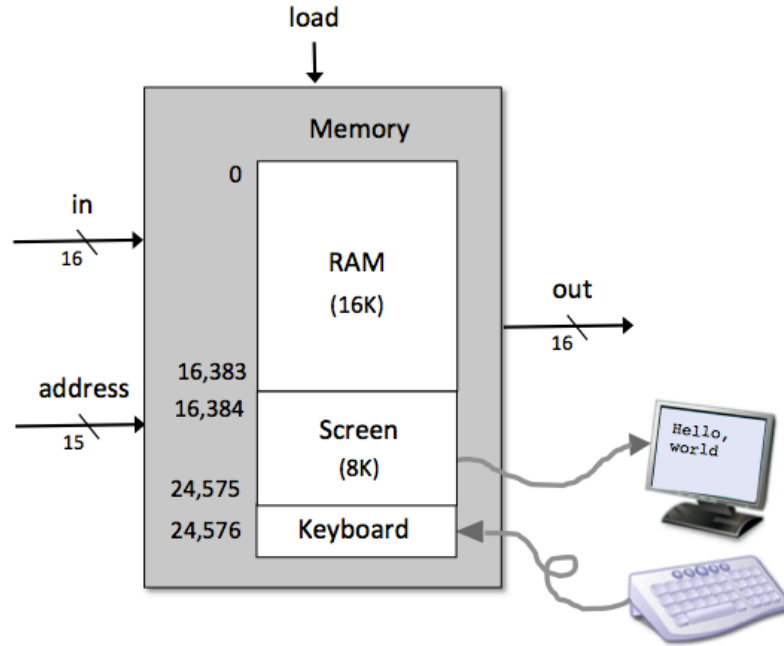
Memory



```
CHIP Keyboard {  
  
    OUT out[16]; // The ASCII code of the pressed key,  
                // or 0 if no key is currently pressed,  
                // or one the special codes  
    BUILTIN Keyboard;  
}
```



Memory Chip Implementation



Memory.hdl

```
CHIP Memory {
  IN in[16], load, address[15];
  OUT out[16];
  PARTS:
    DMux(in=load, sel=address[14], a=loadRam, b=loadScreen);
    RAM16K(in=in, address=address[0..13], load=loadRam, out=outRam);
    Screen(in=in, address=address[0..12], load=loadScreen, out=outScreen);
    Keyboard(out=outKeyboard);
    Mux4Way16(a=outRam, b=outRam, c=outScreen, d=outKeyboard, sel=address[13..14],
out=out);
}
```



Hack Data Memory Chip (RAM)





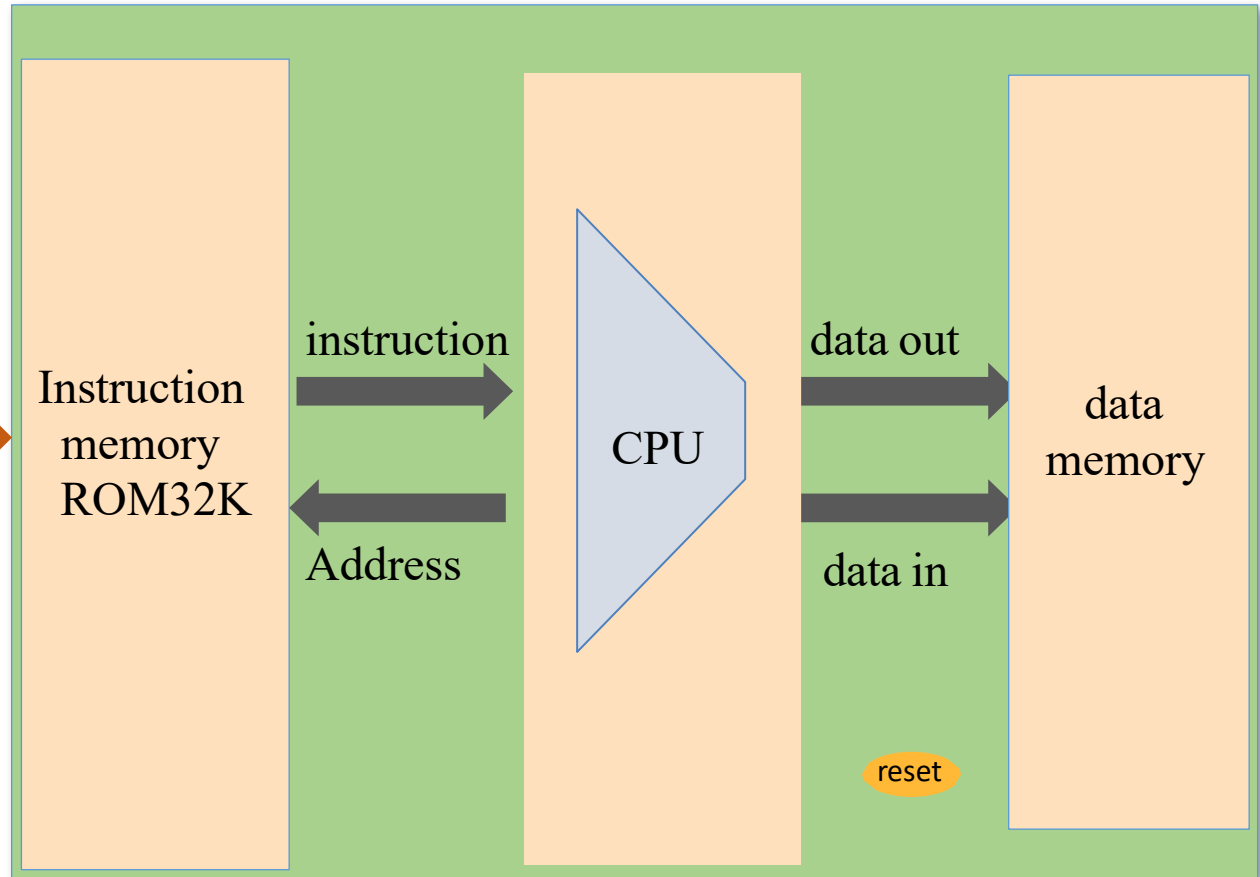
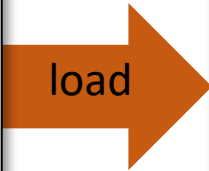
Implementation of Hack Instruction Memory Chip (ROM)



Instruction Memory

Hack Program

```
00000000000001101
1110101001010101
0000000000000001
1110101001101011
0000001100110101
1110010111011111
*
*
*
1111001001100111
```



To run a program on the Hack computer:

- Load the program into the Instruction Memory
- Press “reset”
- The program starts running



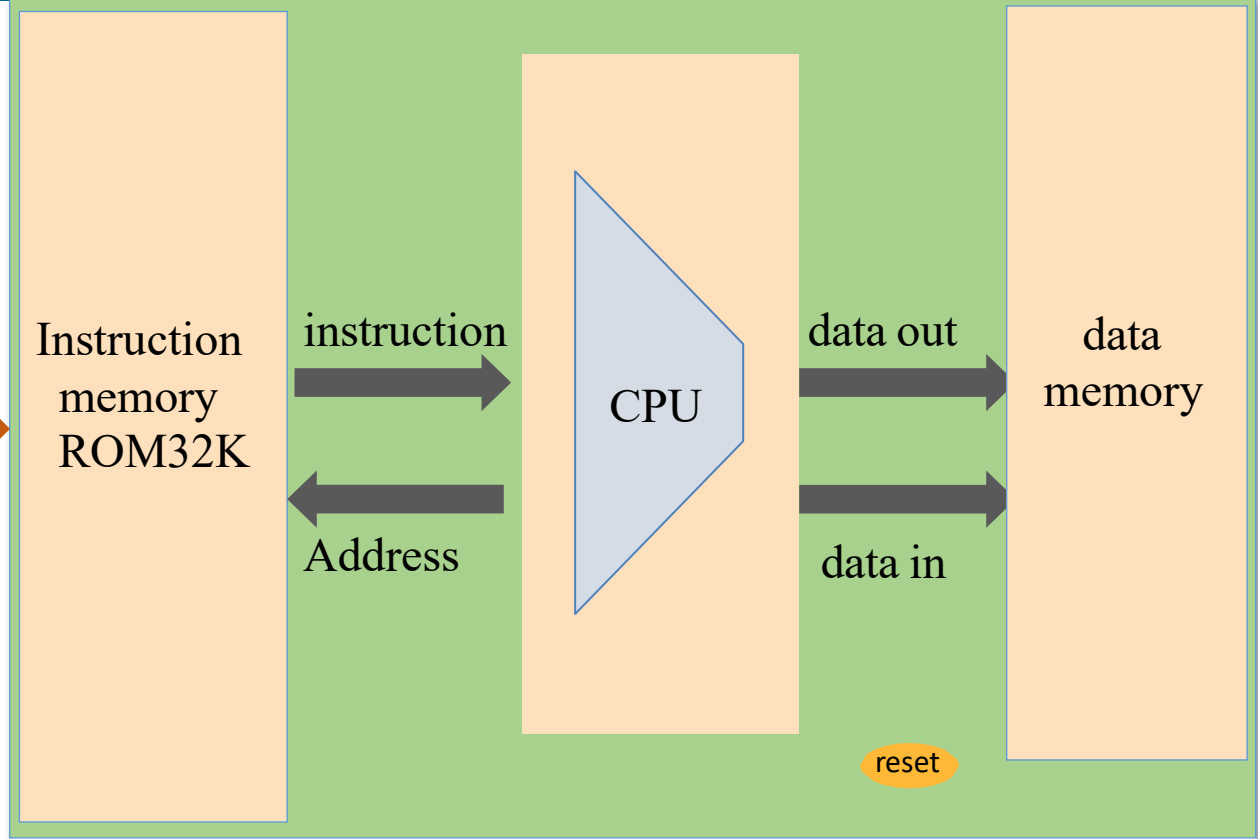
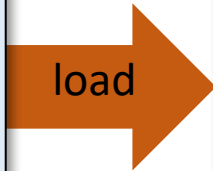
How do you load a program into the Instruction Memory?



Loading a Program in Instruction Memory

Hack Program

```
00000000000001101
1110101001010101
0000000000000001
1110101001101011
0000001100110101
1110010111011111
*
*
*
1111001001100111
```



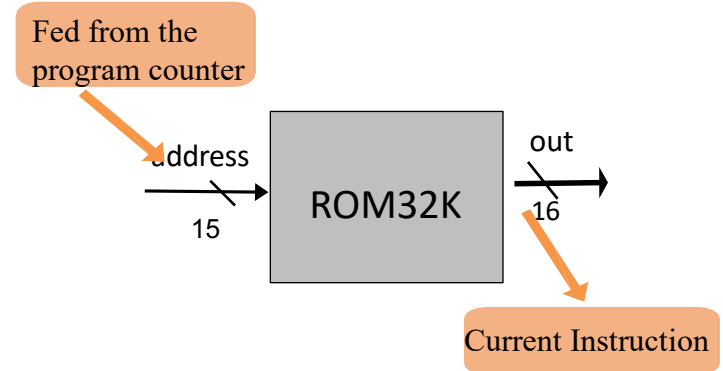
- **Hardware Implementation:** We use a specialized hardware normally called the programmers, which burn/write the hack machine code into a EEPROM chip. We then plug that EEPROM into our computer. Press the reset button and our program starts running. To run another program repeat process (game cartridges for game consoles)
- **Hardware Simulation:** Programs are stored in text files; Program loading is emulated by the built-in ROM chip. The simulator's software features a load-program service



ROM32K Built-in Chip

ROM32K.hdl

```
CHIP ROM32K {  
    IN address[15];  
    OUT out[16];  
    BUILTIN ROM32K;  
}
```



- The Hack instruction memory is implemented as a built-in ROM32K chip, which is actually a read-only, 16-bit, 32K RAM chip with program loading side-effect. **The chip always outputs the value stored at the memory location specified by address (next instruction to be executed)**
- The built-in chip implementation (written in Java) has a GUI side-effect, showing an array-like component that displays the ROM's contents
- The ROM32K chip is supposed to be pre-loaded with a machine language program. However, once the built-in chip implementation is loaded inside the hardware simulator, it allows the user to load Hack machine program from a text file (We will soon see a demo)



Hack Instruction Memory Chip (ROM)

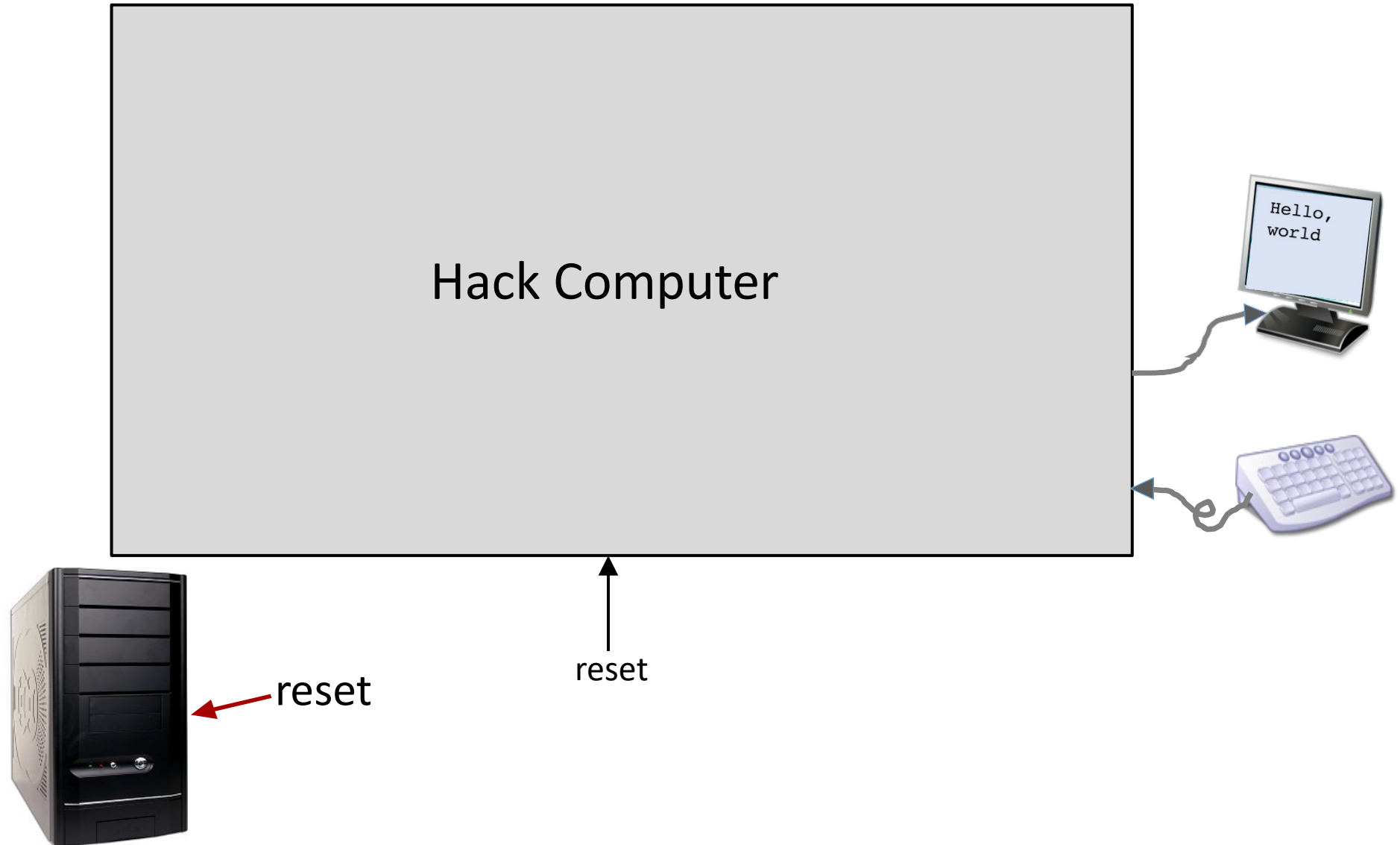




Implementation of Hack Computer Chip

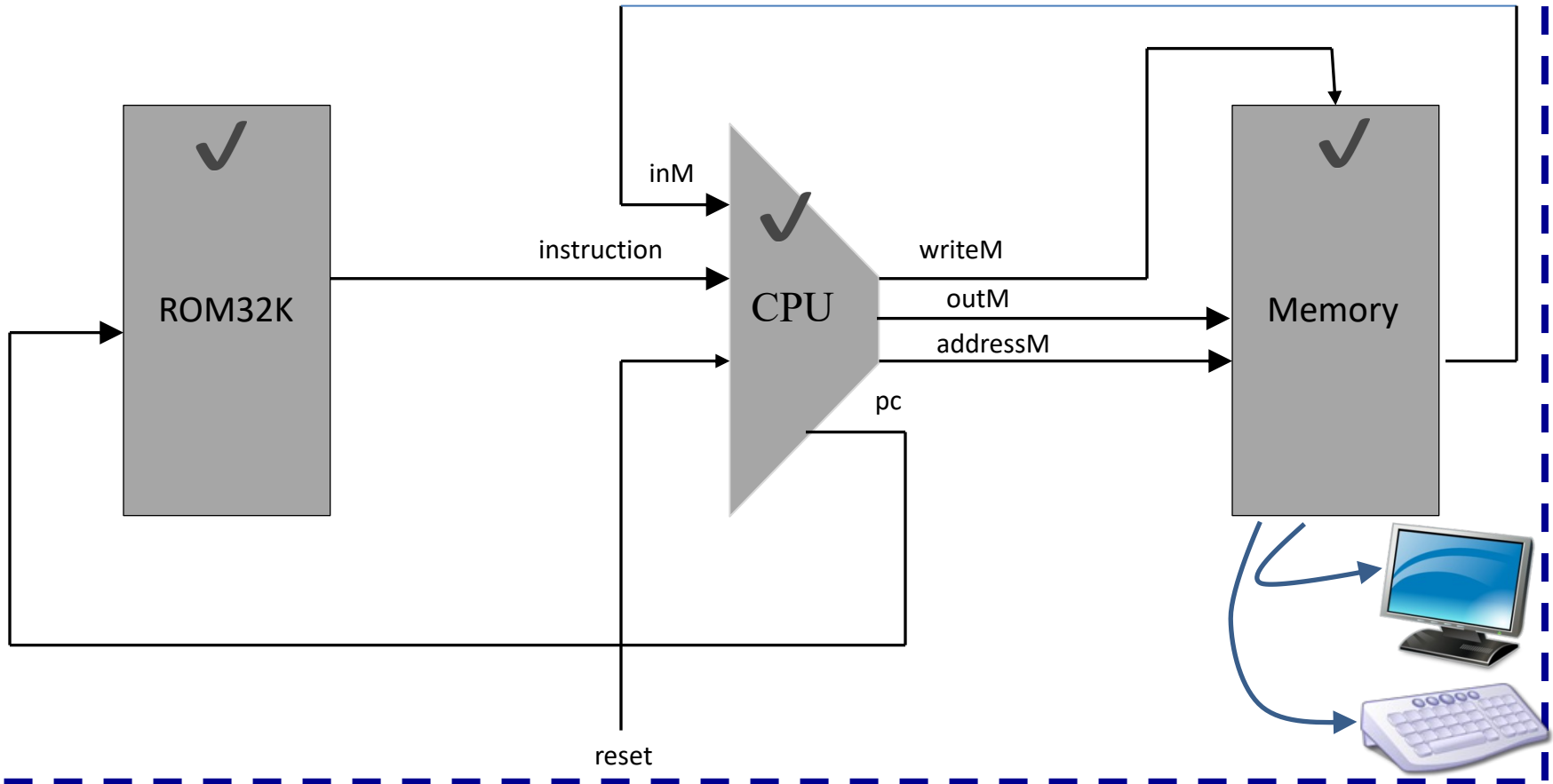


Hack Computer Abstraction





Hack Computer Implementation



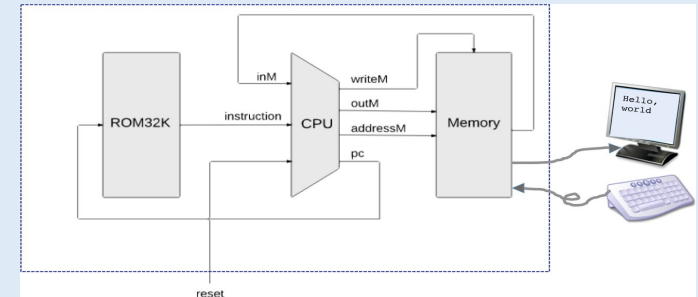


Hack Computer Chip

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.  
 * 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.  
 */
```

```
CHIP Computer {  
    IN reset;  
    PARTS:  
        ROM32K(address=pc, out=instruction);  
        CPU(inM=memoryOut, instruction=instruction, reset=reset, outM=outM,  
writeM=writeM, addressM=addressM, pc=pc);  
        Memory(in=outM, load=writeM, address=addressM, out=memoryOut);  
}
```



Simplicity at its peak ☺



Executing Assembly Programs in Hack Computer Chip





Things To Do

