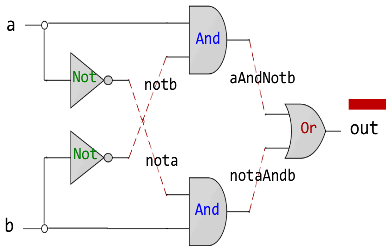
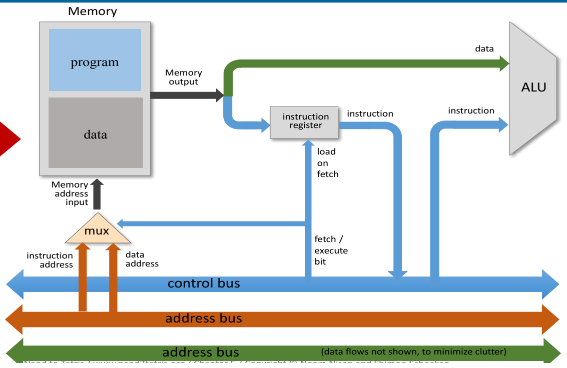
Digital Logic Design



```
CHIP Xor {
  IN a, b;
  OUT out;
  PARTS:
  Not(in=a, out=nota);
  Not(in=b, out=notb);
  And(a=nota, b=b, out=w1);
  And(a=a, b=notb, out=w2);
  Or(a=w1, b=w2, out=out);
}
```



```
@R1
D=M
@temp
M=D
```

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
```

Lecture # 26
Data Path of Hack CPU

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```

```
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
```

Slides of first half of the course are adapted from:
https://www.nand2tetris.org
Download s/w tools required for first half of the course from the following link:
https://drive.google.com/file/d/0B9c0BdDJz6XpZUh3X2dPR1o0MUE/view

Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

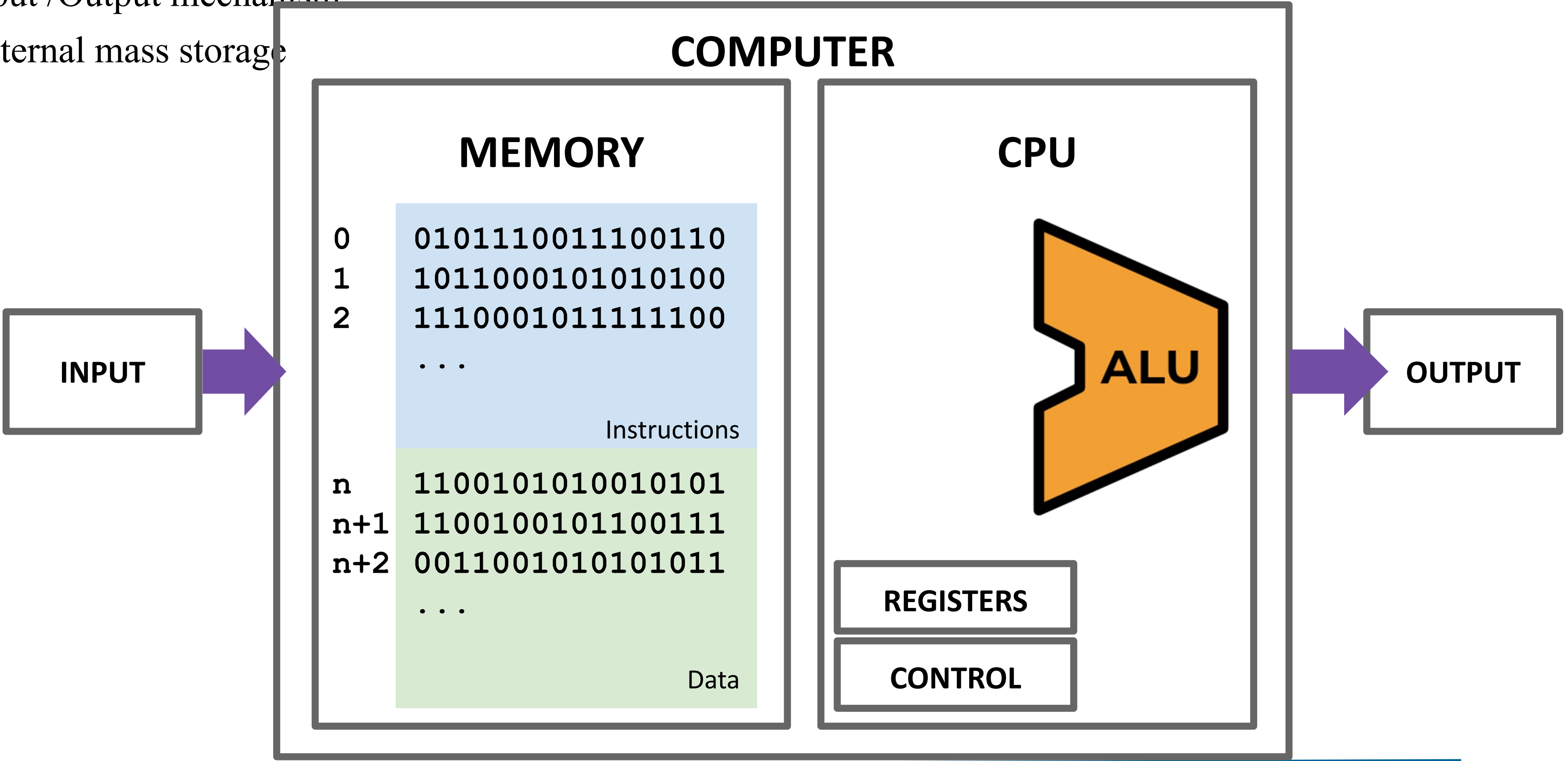
- Von Neumann Architecture
- Flow of Information inside Computers
- Buses
  - Data Bus
  - Address Bus
  - Control Bus
- Fetch Execute Cycle
- Fetch Execute Clash
- Harvard Architecture
- Hack CPU Interface
- Hack CPU Implementation
- Input/output and Operations of Hack ALU
- Control Logic of Hack CPU

Von Neumann Architecture

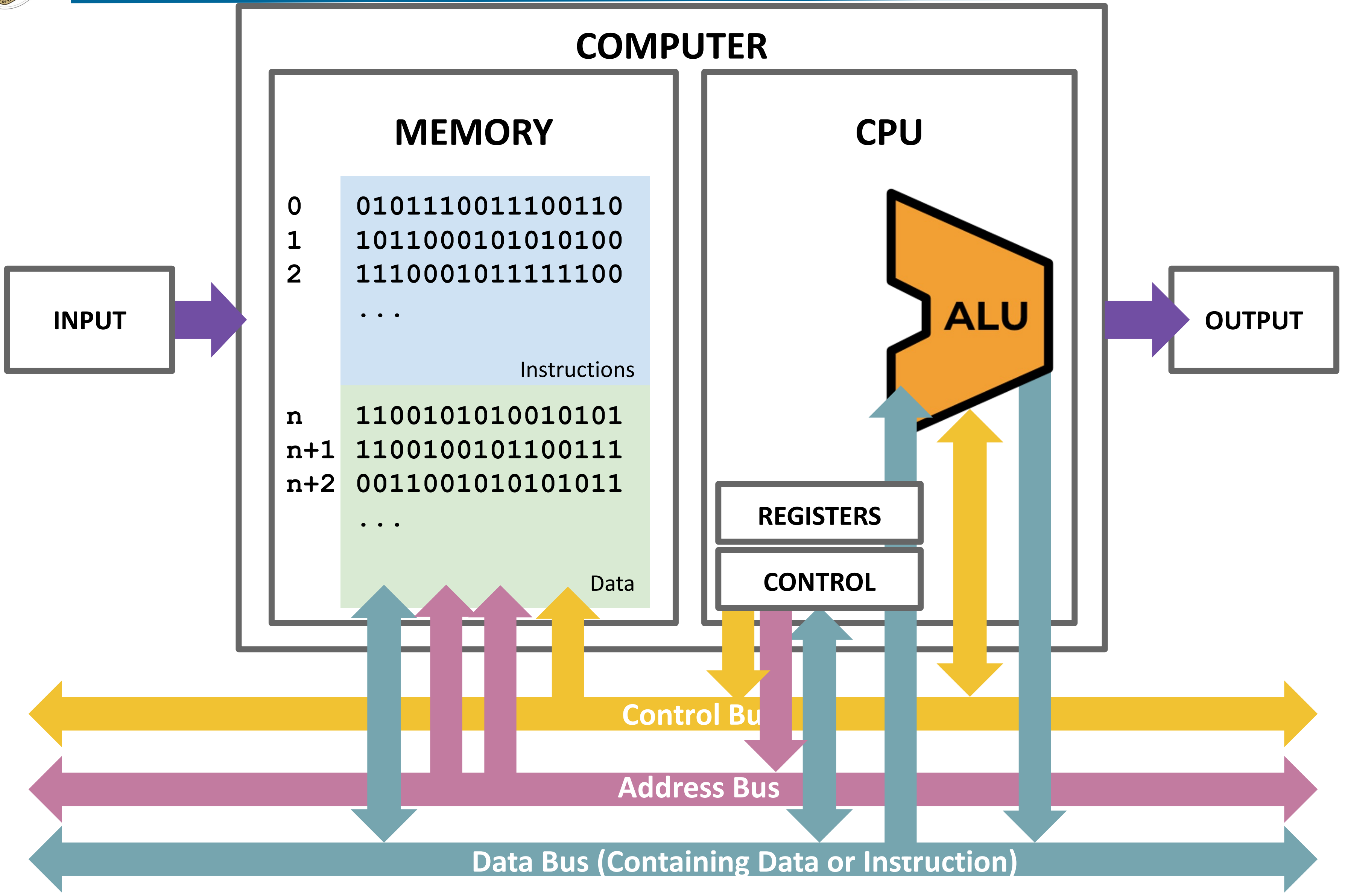
Von Neumann Architecture

The Von Neumann architecture is a computer architecture given by a mathematician and physicist John von Neumann describes the design architecture for an electronic digital computer with these components:

- A Processing Unit that contains an ALU and registers
- A Control Unit that contains an instruction register and program counter
- **A Memory unit that stores both data and instructions**
- Input /Output mechanism
- External mass storage



Information Flow / CPU Data Path



Instructor: Muhammad Arif Butt, Ph.D.

Overview of General Fetch-Execute Cycle

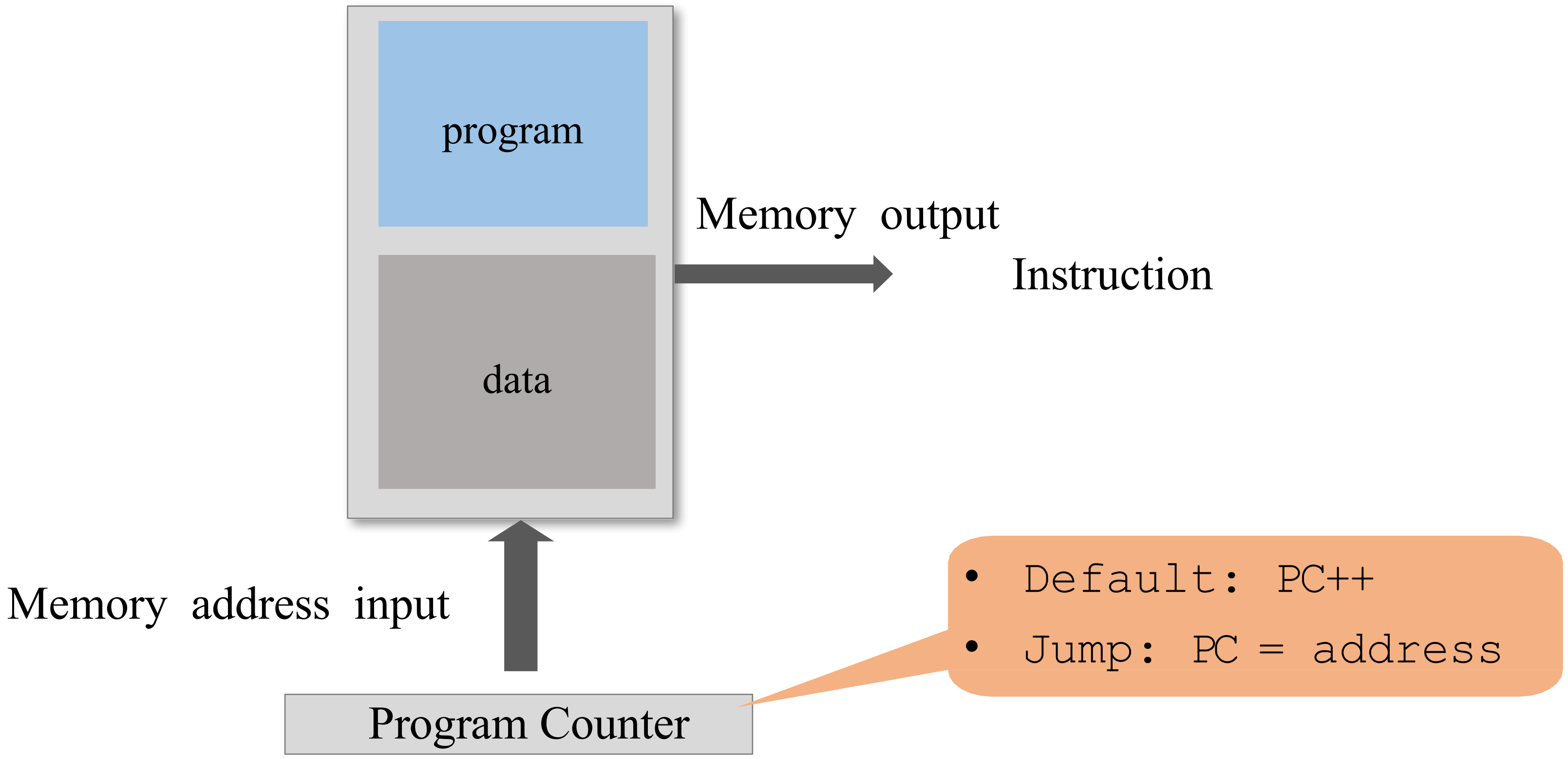
Basic CPU Loop

Repeat:

- **Fetch** an instruction from the program memory
- **Execute** the instruction

Fetching

- Put the location of the next instruction in the Memory address input
- Read the contents of the memory from that location to get the instruction code



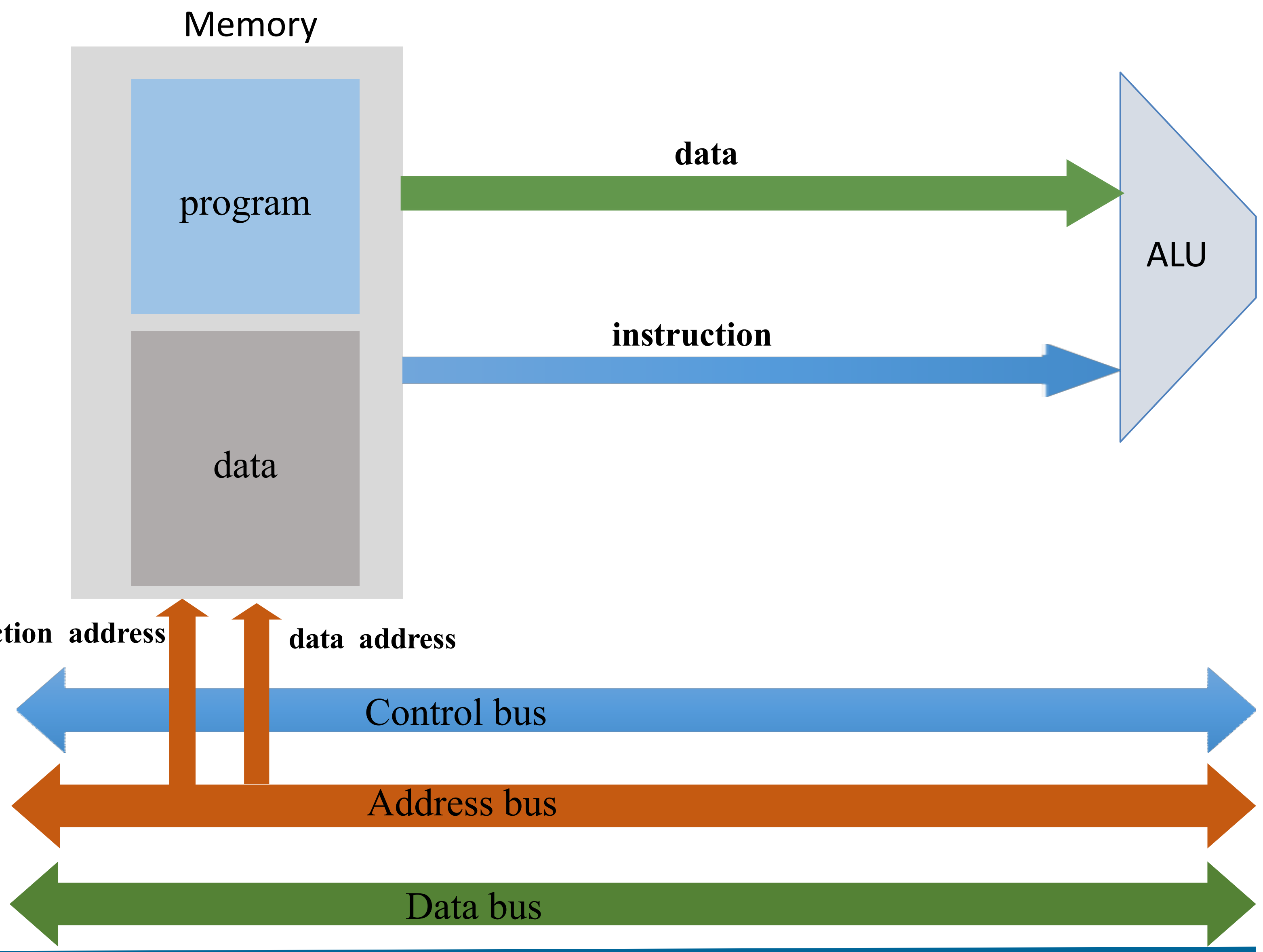
Executing

- The instruction code specifies “what to do”
  - Which arithmetic or logical instruction to execute
  - Which memory address to access (for read / write)
  - If / where to jump
  - …

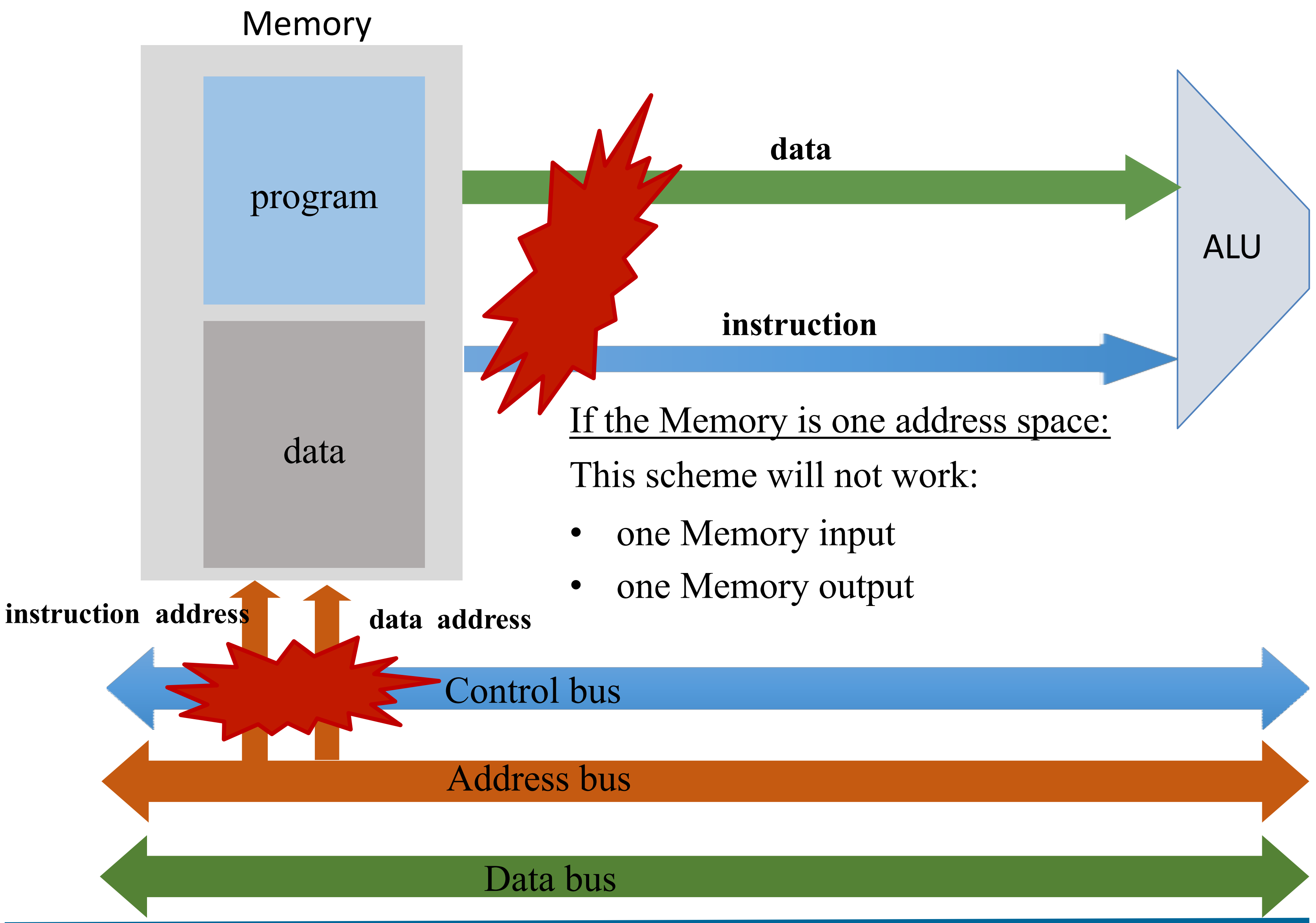
Different subset of the instruction bits controls different aspects of the operation

- Executing the instruction involves:
  - accessing registers and / or
  - accessing the data memory

Fetch Execute



Fetch-Execute Clash

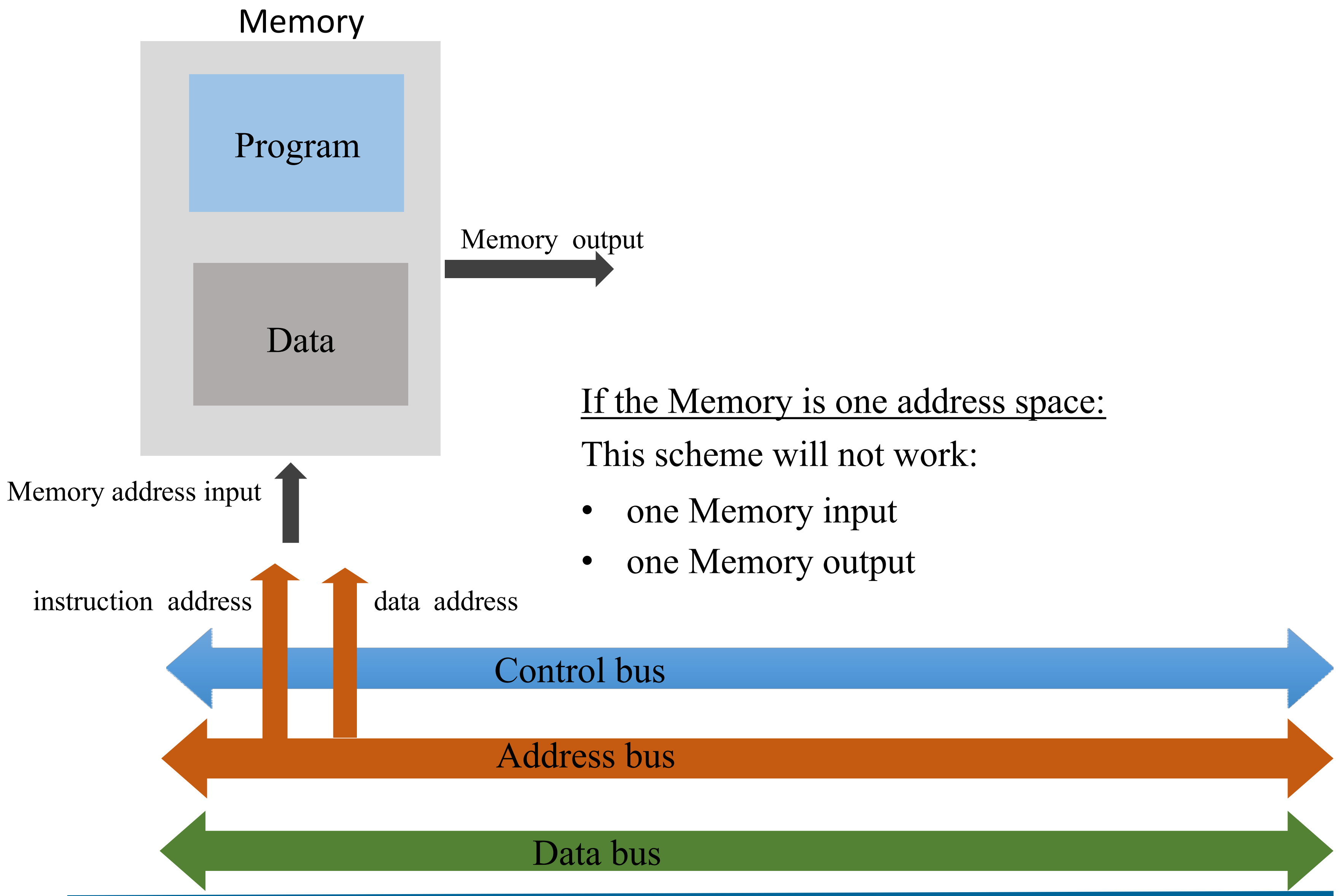


If the Memory is one address space:

This scheme will not work:

- one Memory input
- one Memory output

Fetch-Execute Clash (cont…)

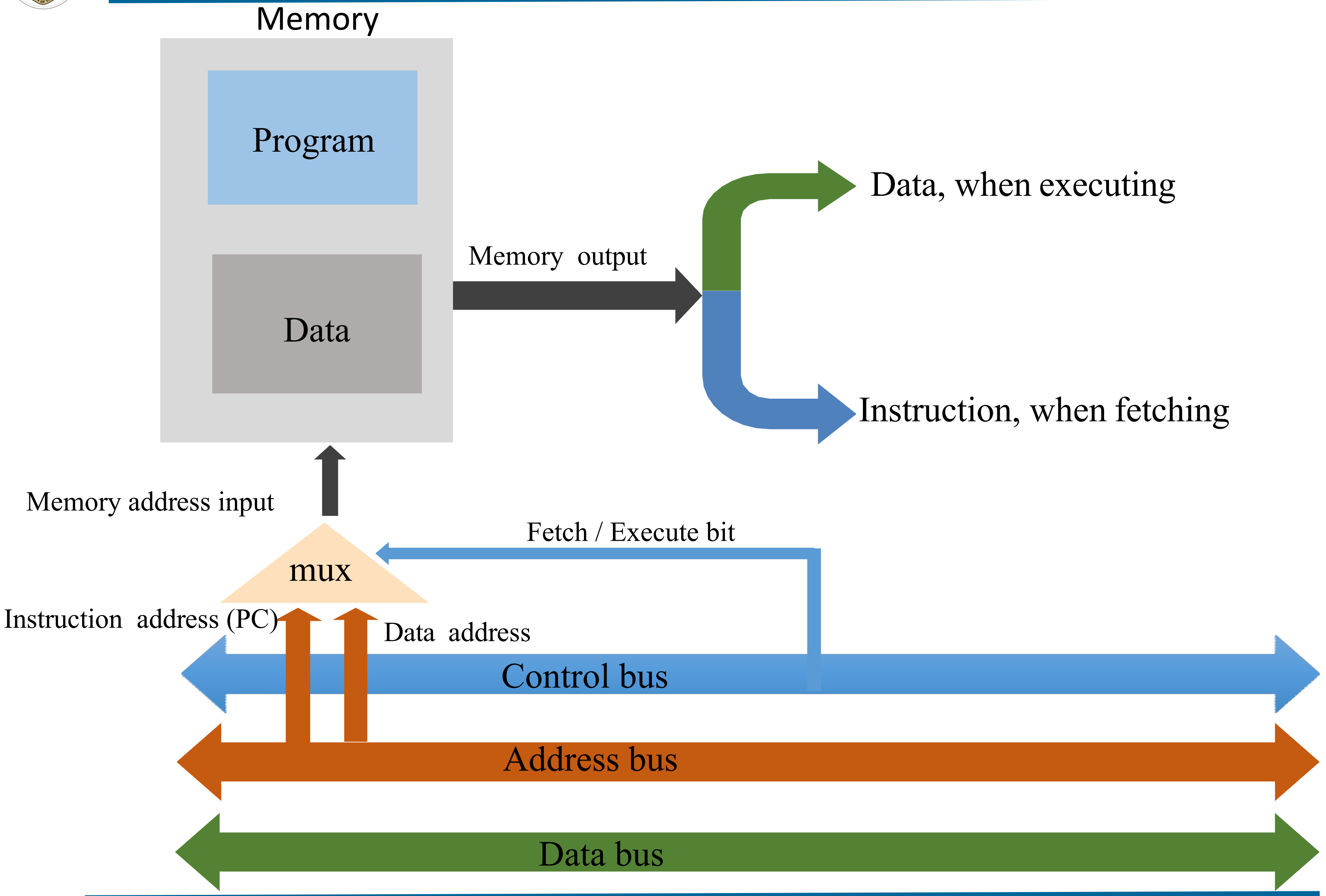


<u>If the Memory is one address space:</u>

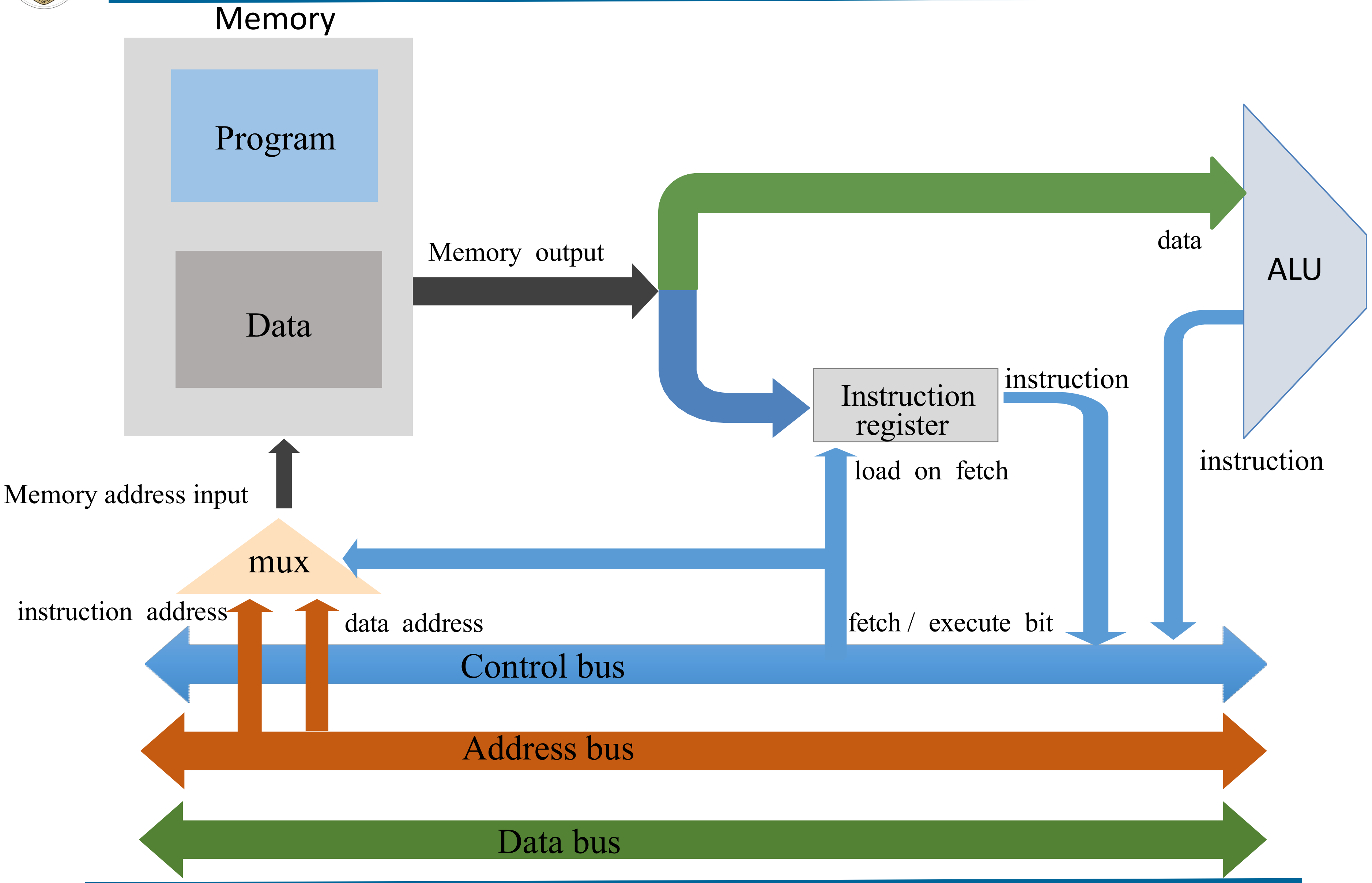
This scheme will not work:

- one Memory input
- one Memory output

Solution: multiplex, using an instruction register



Solution: multiplex, using an instruction register



Simpler Solution: Harvard Architecture

Variant of von Neumann Architecture (used by the Hack computer):

Two physically separate memory units:

- Instruction memory
- Data memory

$\left.\begin{array}{l} \text{Instruction memory} \\ \text{Data memory} \end{array}\right\}$ Each can be addressed and manipulated seperately, and simultaneously

Advantage:

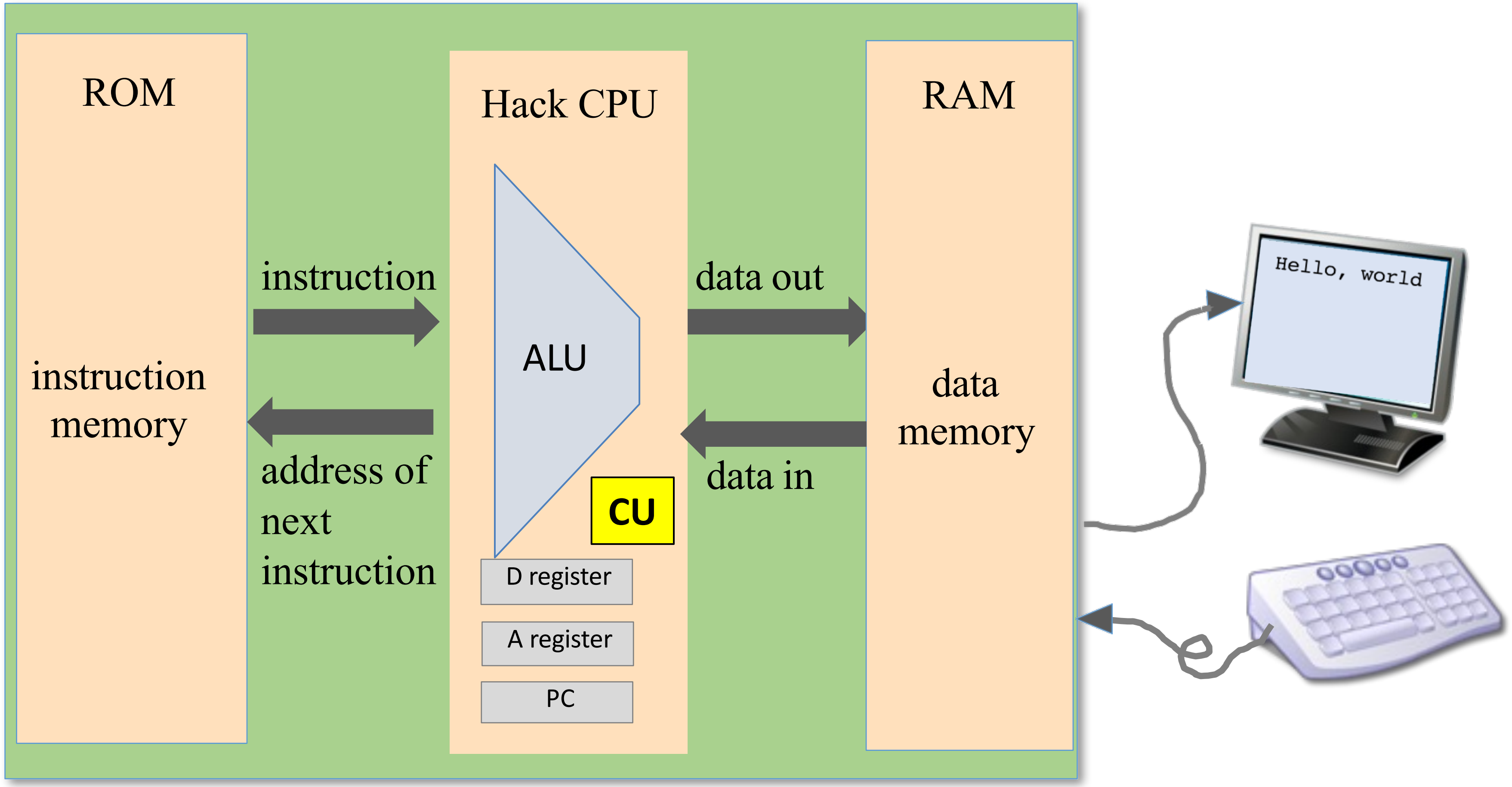
- Complication avoided

Disadvantage:

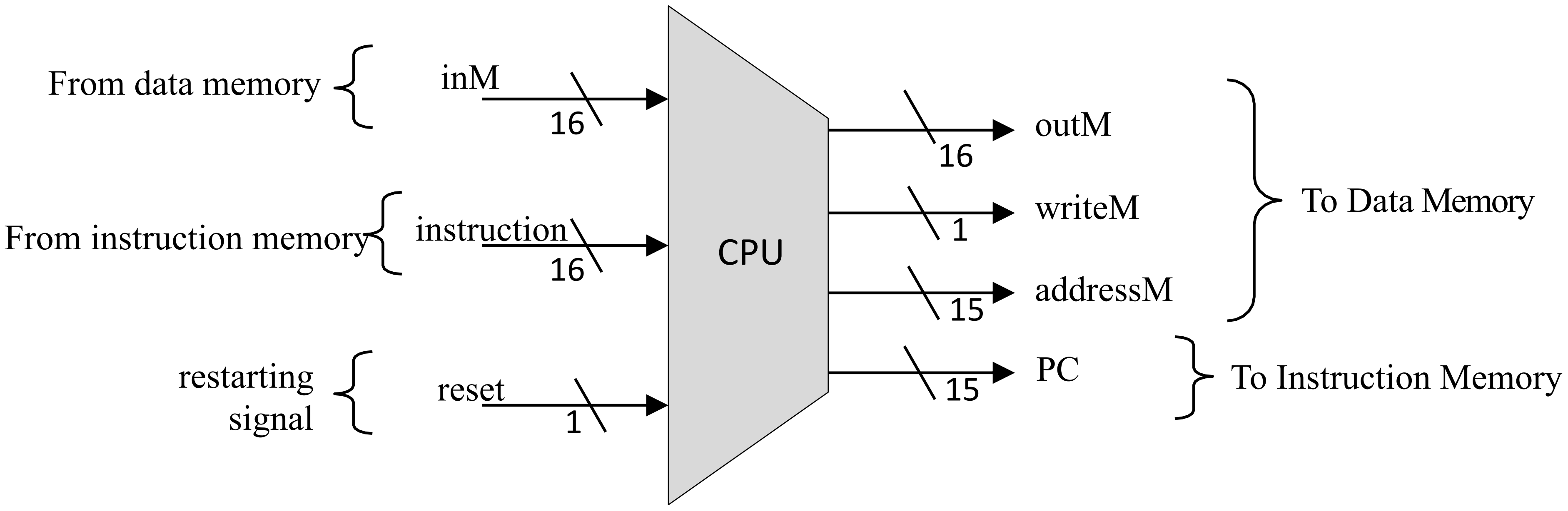
- Two memory chips instead of one
- The size of the two chips is fixed

Designing the Data Path of Hack CPU

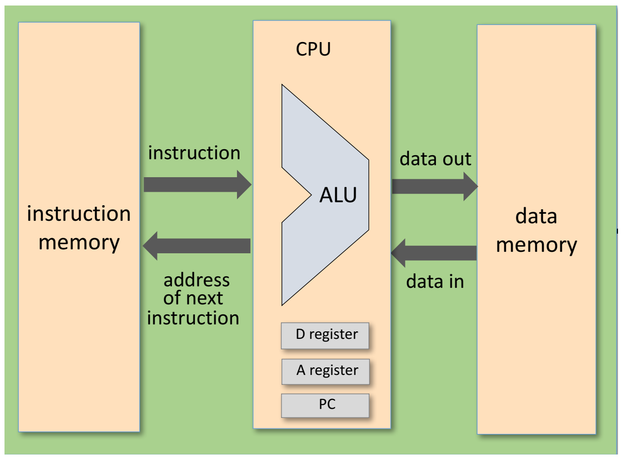
Hack Computer Architecture



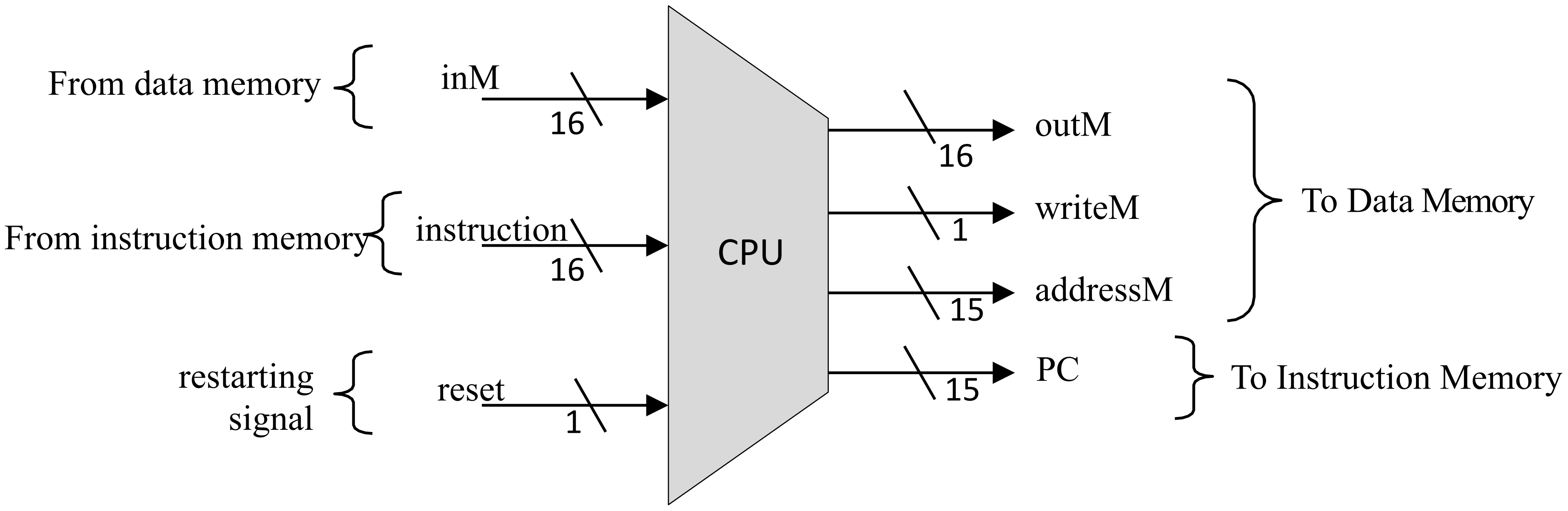
Hack CPU Interface



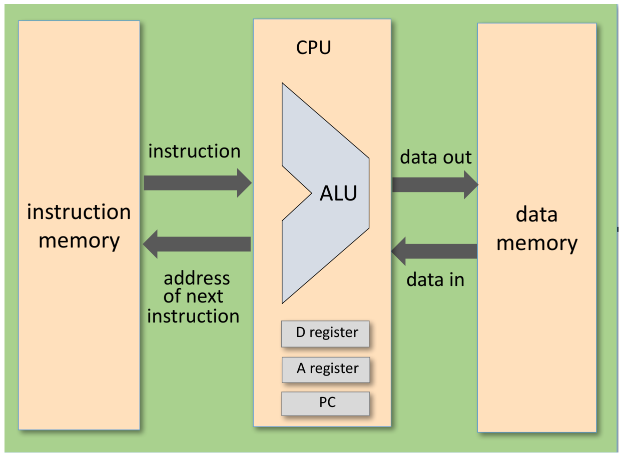
Inputs:
- Data Value
- Instruction
- Reset Bit



Hack CPU Interface



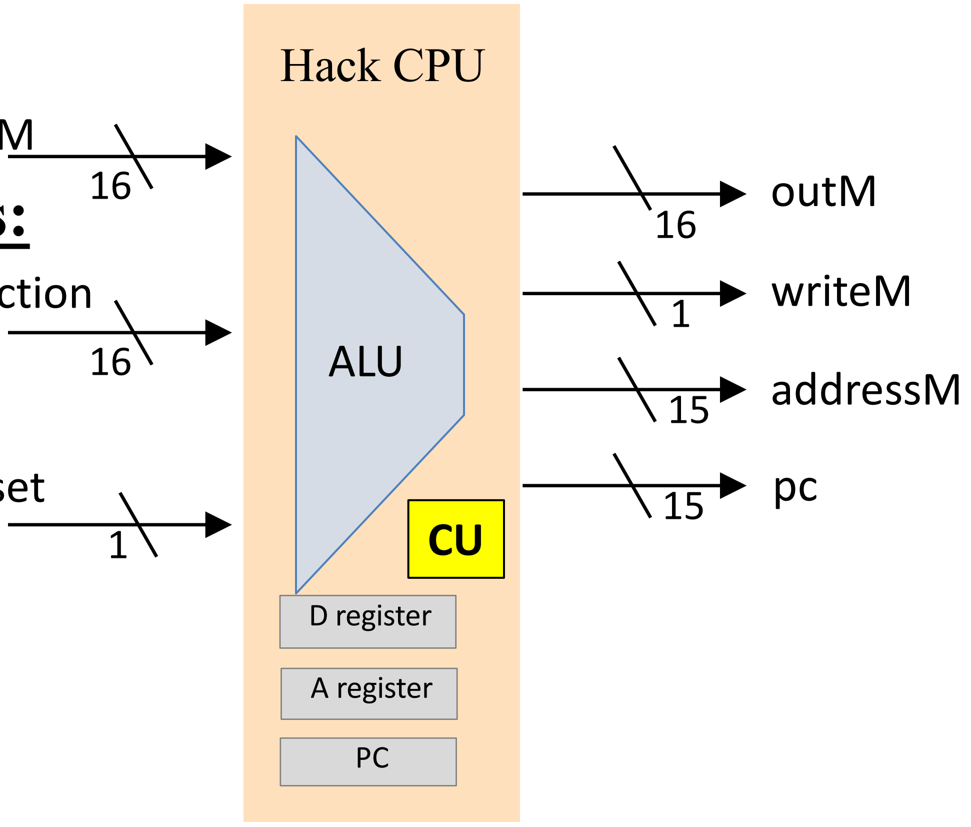
Outputs:
- Data Value
- Write to Memory? (yes/no)
- Memory Address
- Address of next instruction



Abstract View of Execution of A and C-Instruction

Hack Instructions:

`D=M+1`



Abstract View of Address of next Instruction

Reset Bit and PC:

If (reset == 1)

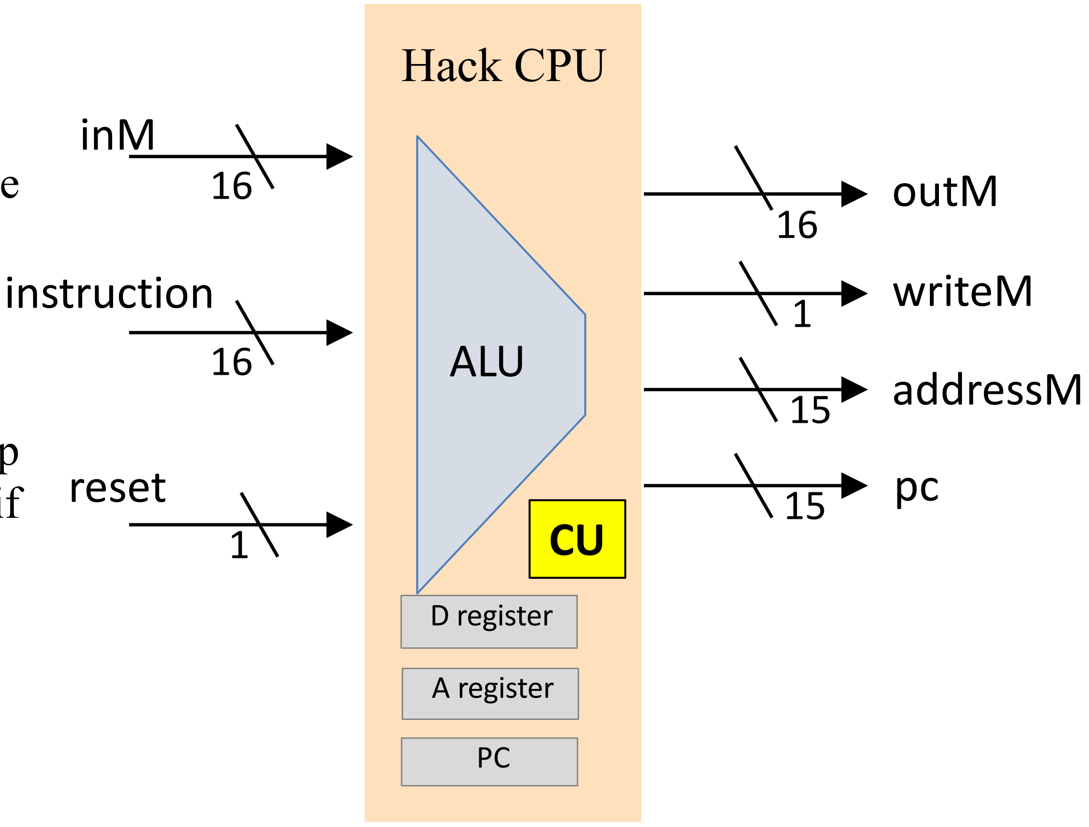
The 15 bit output **pc** emits 0, causing the program to restart

If (reset==0)

The CPU logic uses the instruction's jump bits and the ALU's output to decide if there should be a jump. For example:

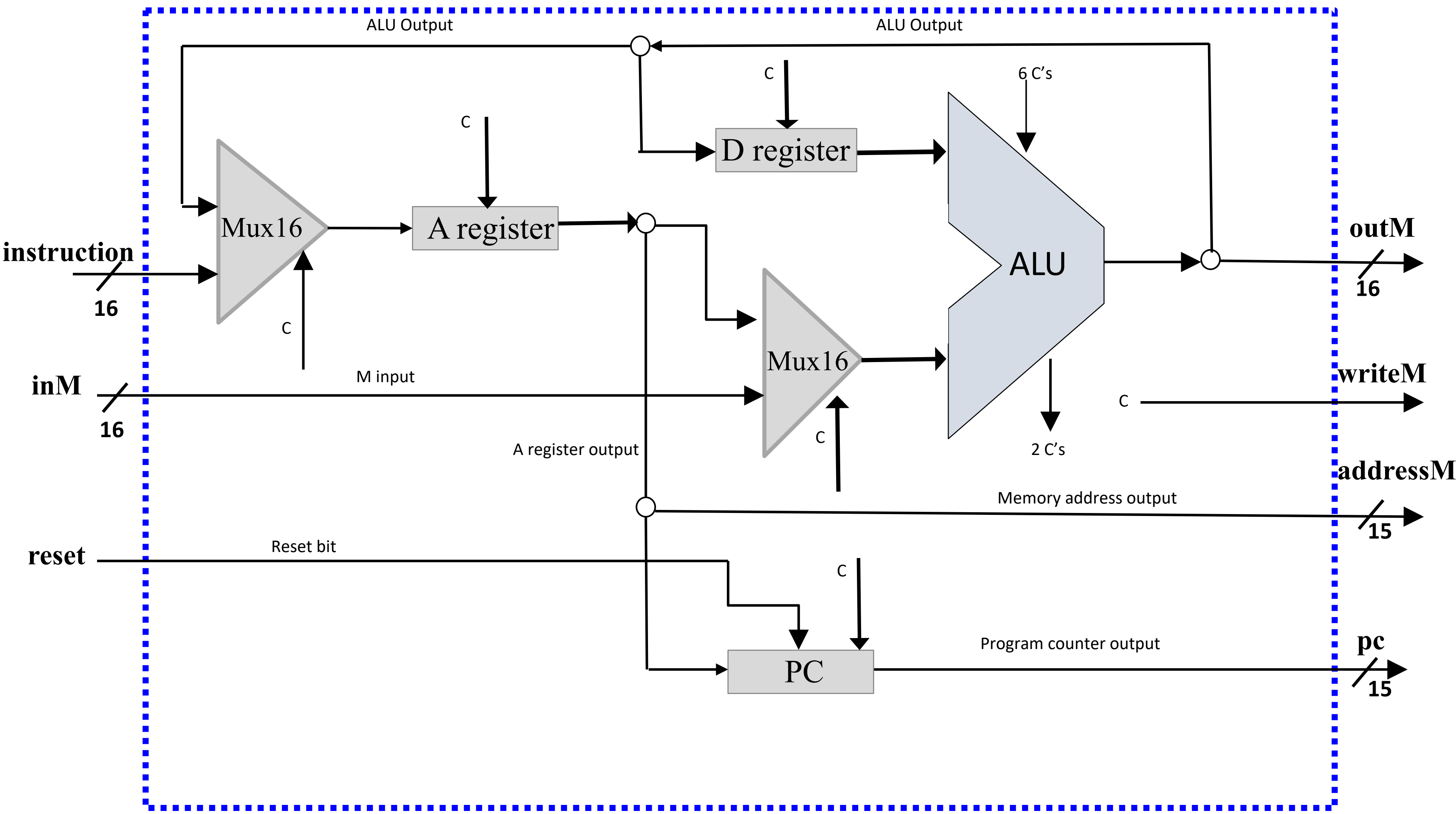
```
@54
D-1;JEQ
```

```
If (D-1==0)
    PC is set to the value of the A-register
else
    PC++
```



The updated PC value is emitted by 15 bit output named **PC**

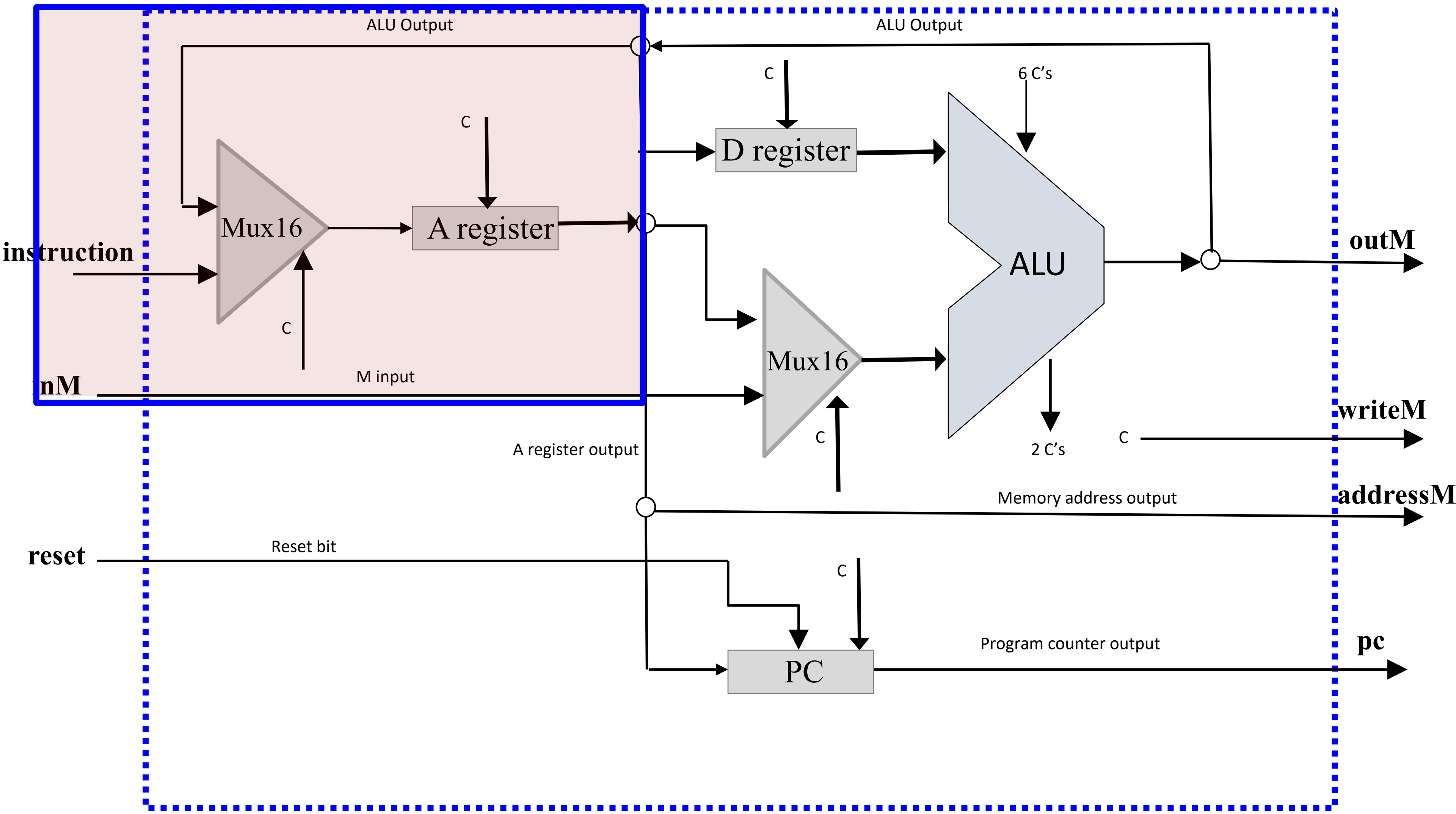
Hack CPU Implementation



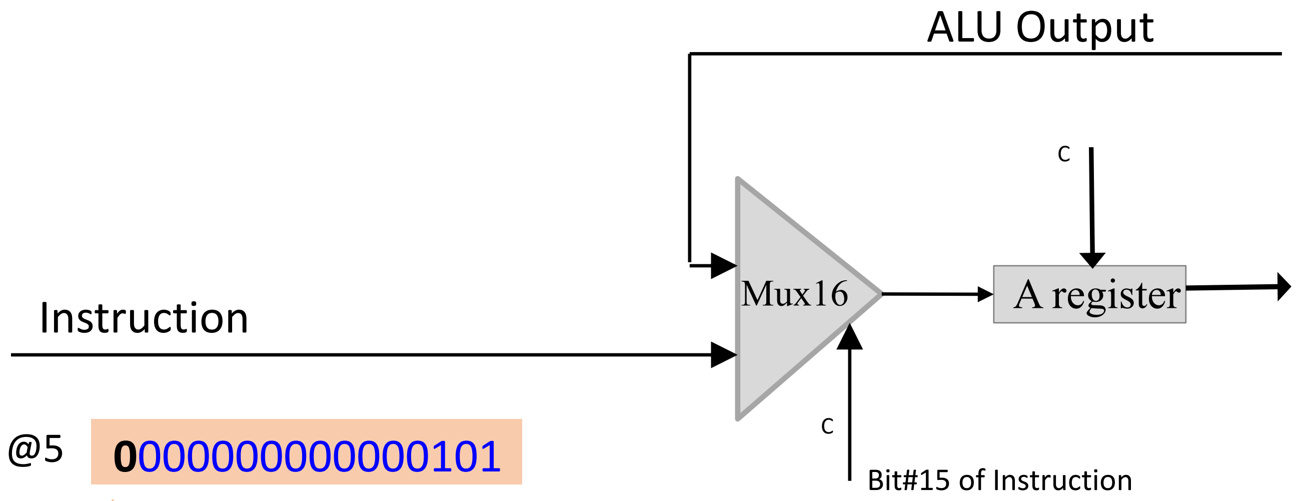
(each "C" symbol represents a control bit)

How A/C-Instructions Execute?

Instruction Handling



Handling A-Instruction



A-instruction

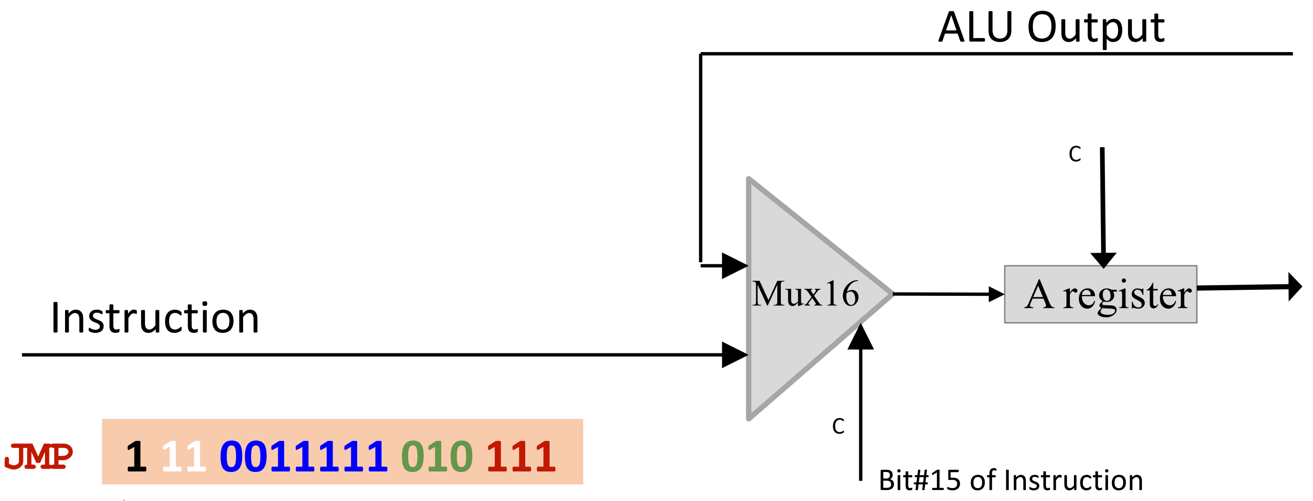
CPU handling of an A-instruction:

- Decodes the instruction into
  - op-code
  - 15-bit value
- Stores the 15 bit value in the A-register
- Outputs the value to ALU via Mux (not shown in this diagram)

Note:

- In case of A-instruction, the A-register get its input from the instruction part
- In case of C-instruction, the A-register get its input from the ALU output

Handling C-Instruction



`D= D+1 ;JMP` 1 11 0011111 010 111

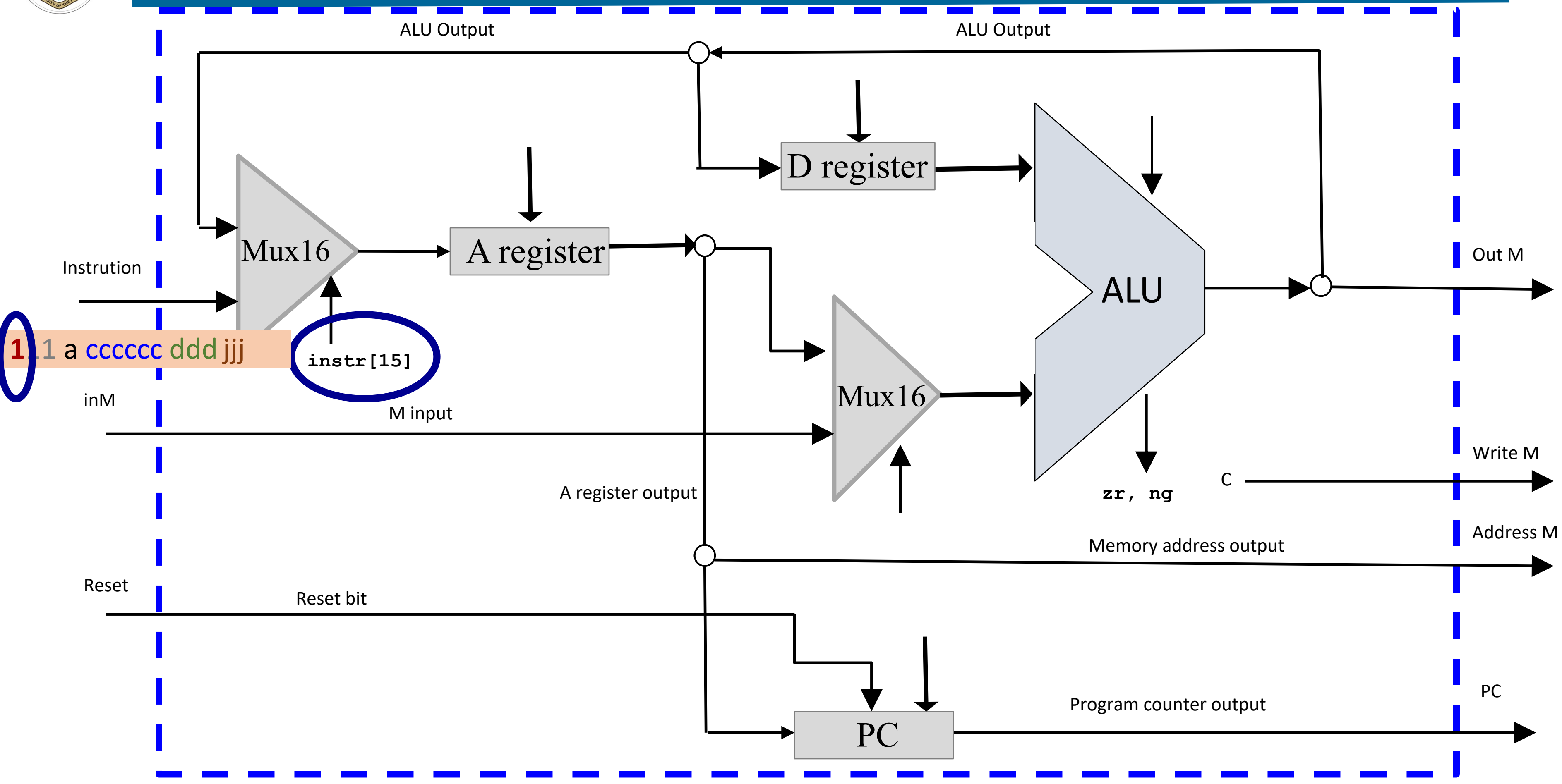
C-instruction

<u>CPU handling of C-instruction:</u> `dest= comp ;jump`

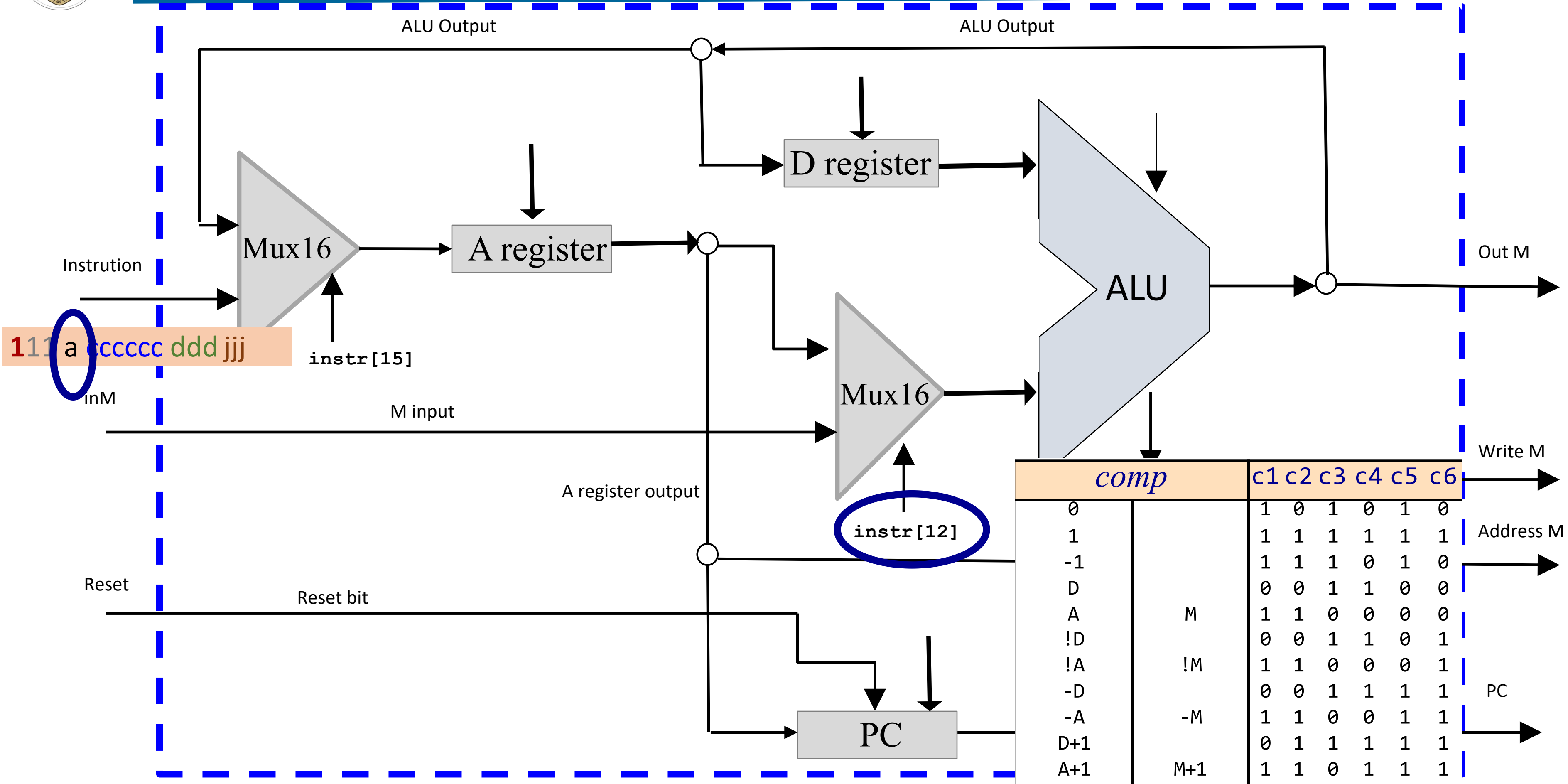
- Decodes the instruction bits into:
  - Op-code
  - ALU control bits (D+1)
  - Destination load bits (D-Register)
  - Jump bits (Un-condional jump)
- Routes these bits to their chip-part destinations
- The chip-parts (most notably, the ALU) execute the instruction

Control Input of two Mux16 Chips

Select Input of First Mux16 Chip



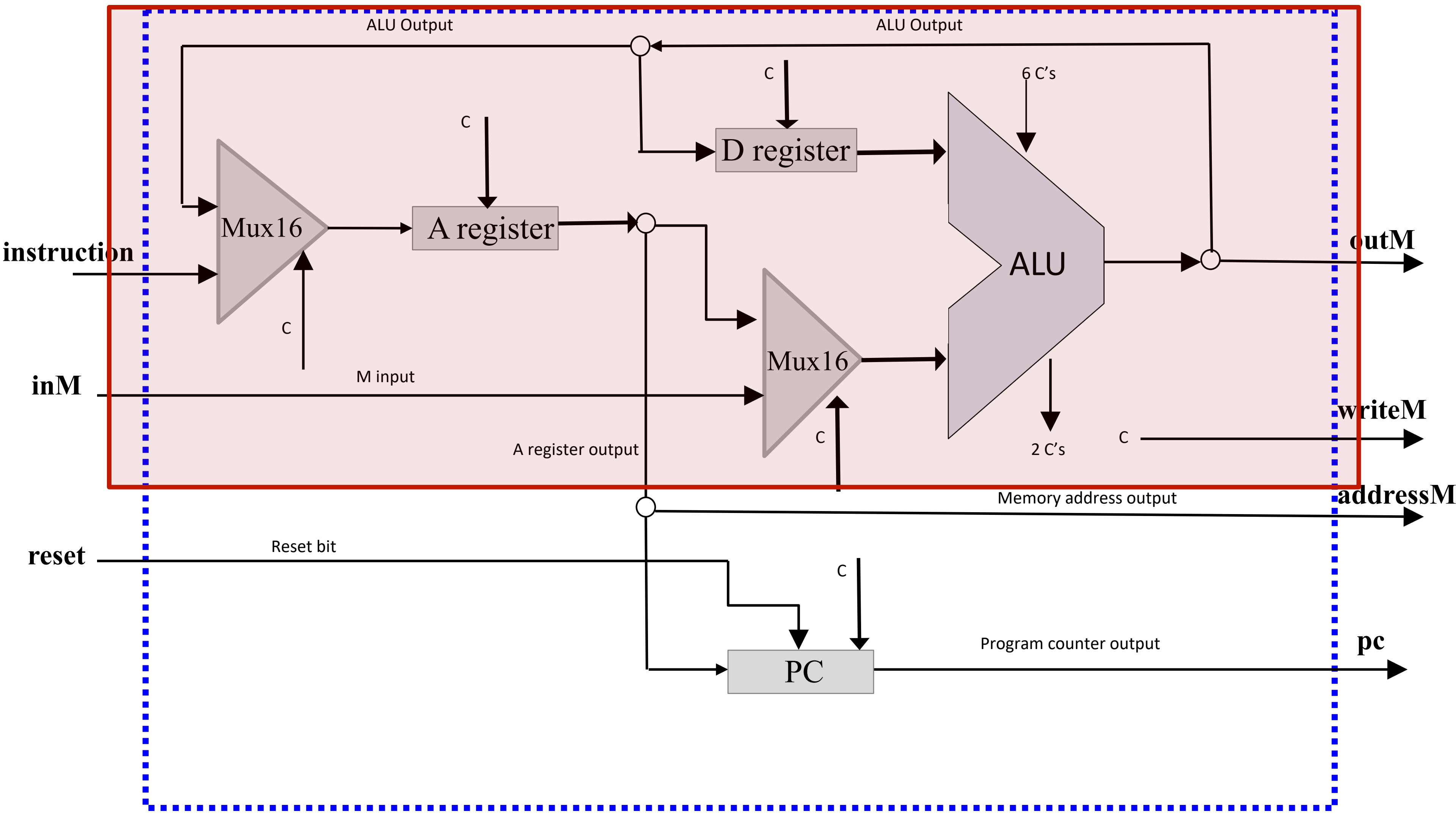
Select Input of Second Mux16 Chip



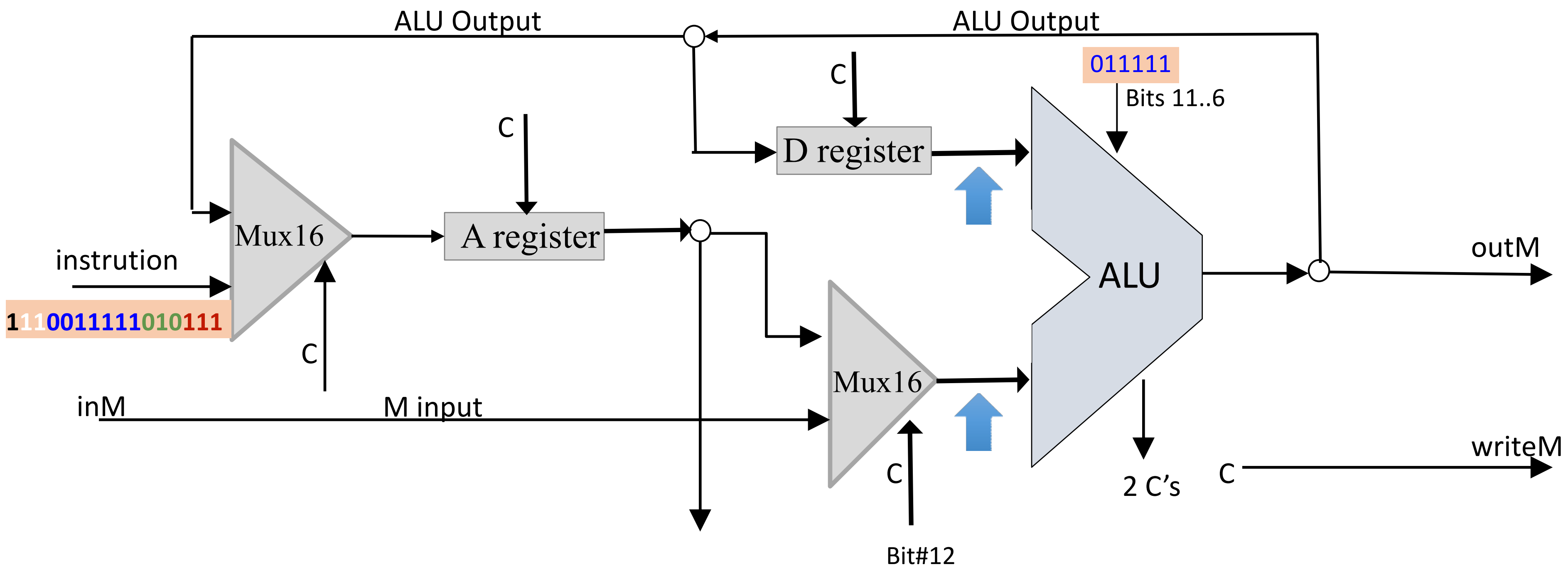
comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	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						

ALU Operations

ALU Operation



ALU Operation: Inputs



ALU data inputs:

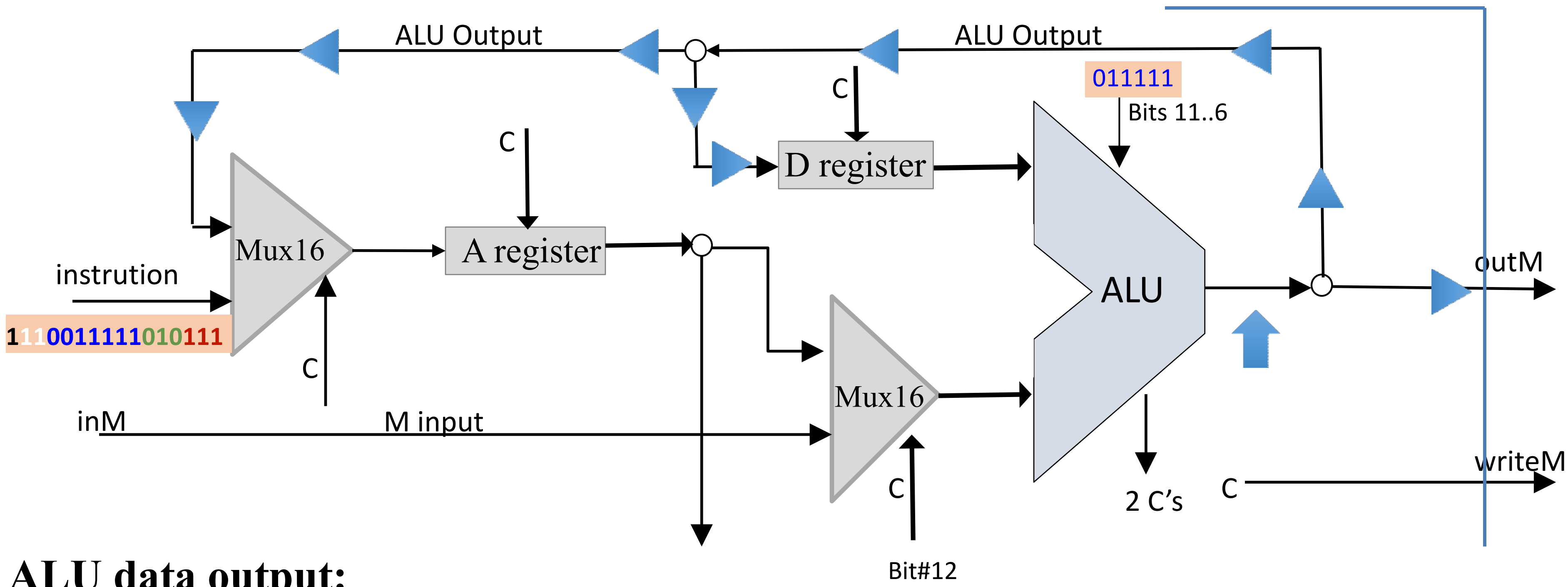
- Input 1: From D-register
- Input 2: From A-register or M-register (decided by bit#12 of Instruction: **a**)

1 11 a cccccc ddd jjj

ALU control inputs:

- 6 x Control bits (from bits 6-11 of the instruction: **cccccc**)

ALU Operation: Outputs

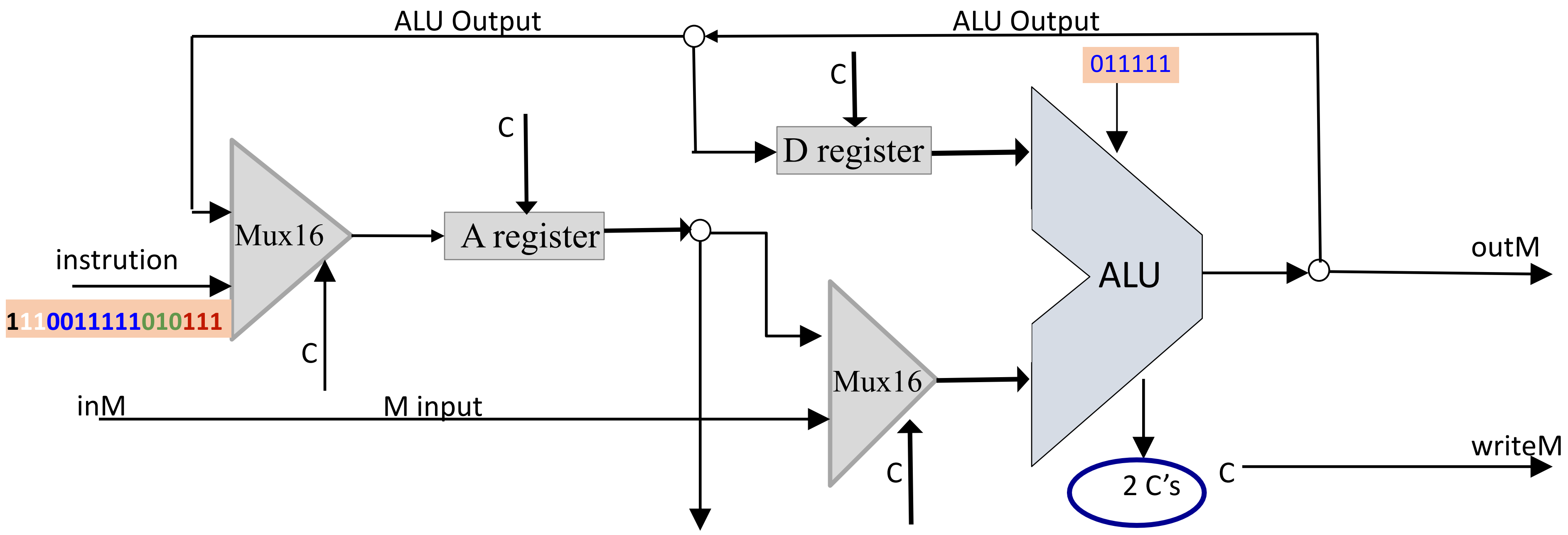


ALU data output:

- Result of ALU calculation, fed simultaneously to three locations:
  - D-register, A-register, M-register (data memory)
- Which out of these three destinations actually commits to the ALU output is determined by the instruction's destination bits

Note: 000 in the destination bits means don't save the ALU output, and 111 means save it simultaneously in D, A and M registers

ALU Operation: Control Outputs

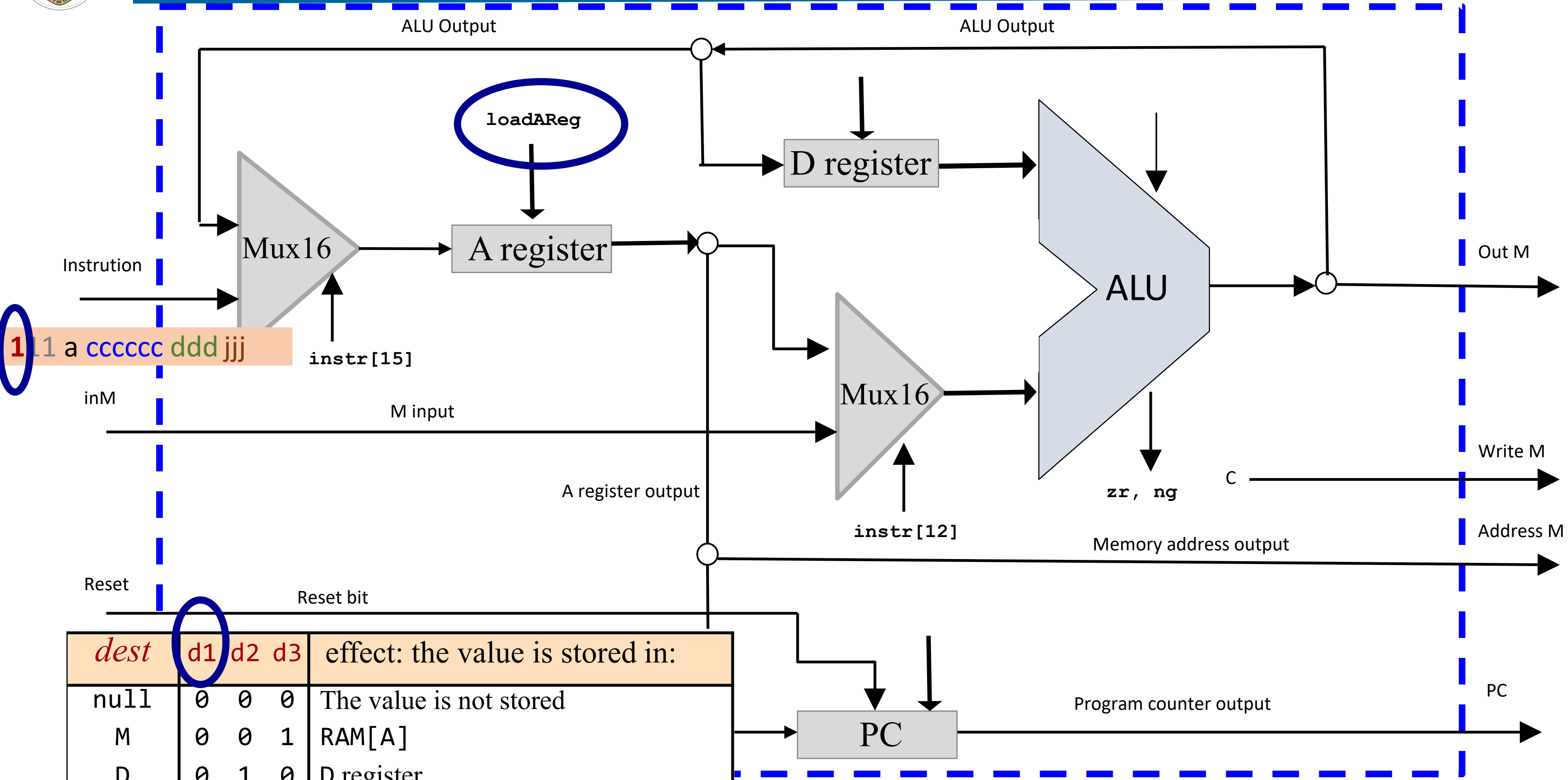


<u>ALU control outputs:</u>

- is the output negative? (ng)
- is the output zero? (zr)

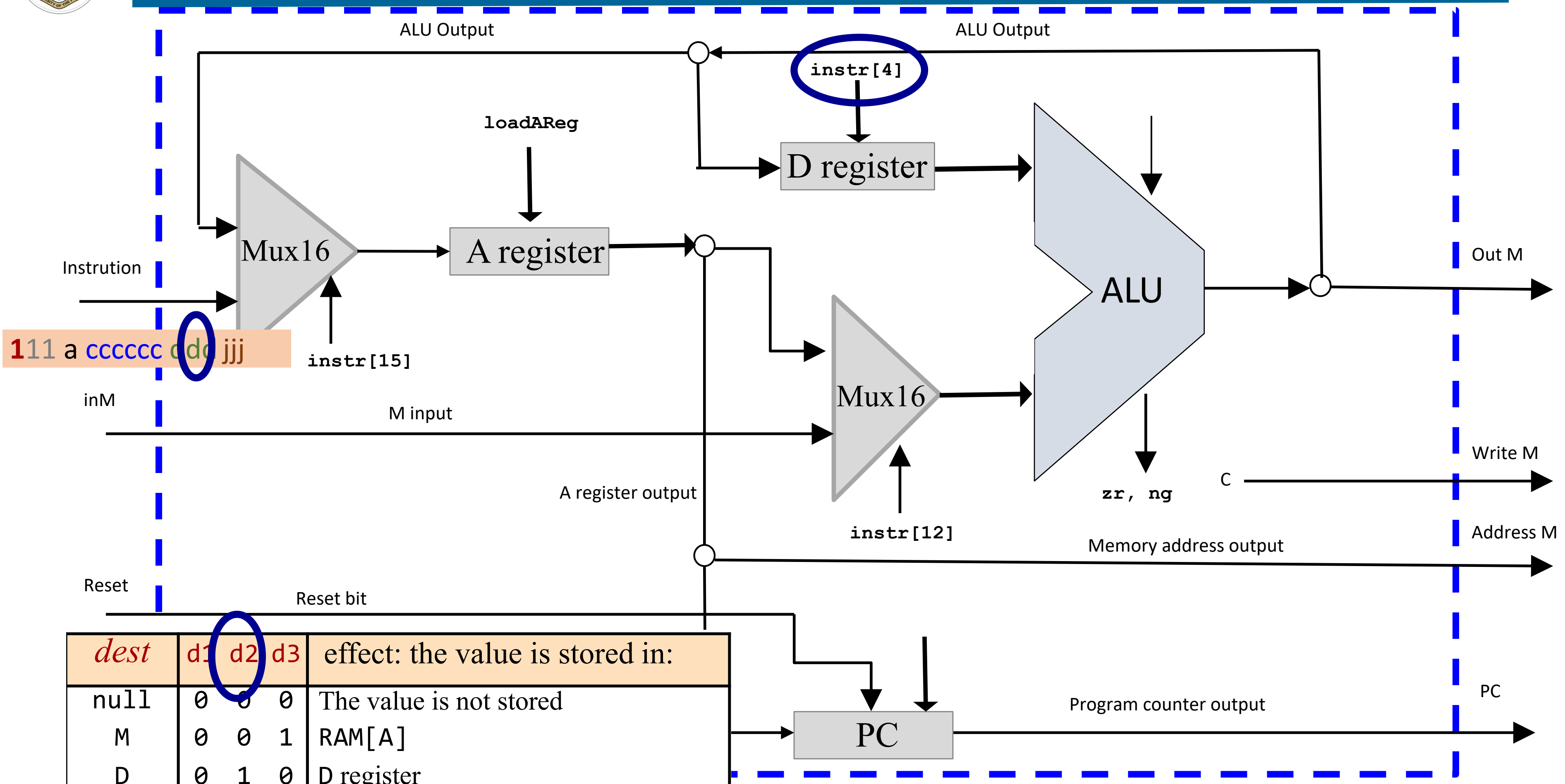
Control Inputs of A-Register and D-Register Chips

Load Input of A-Register Chip



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

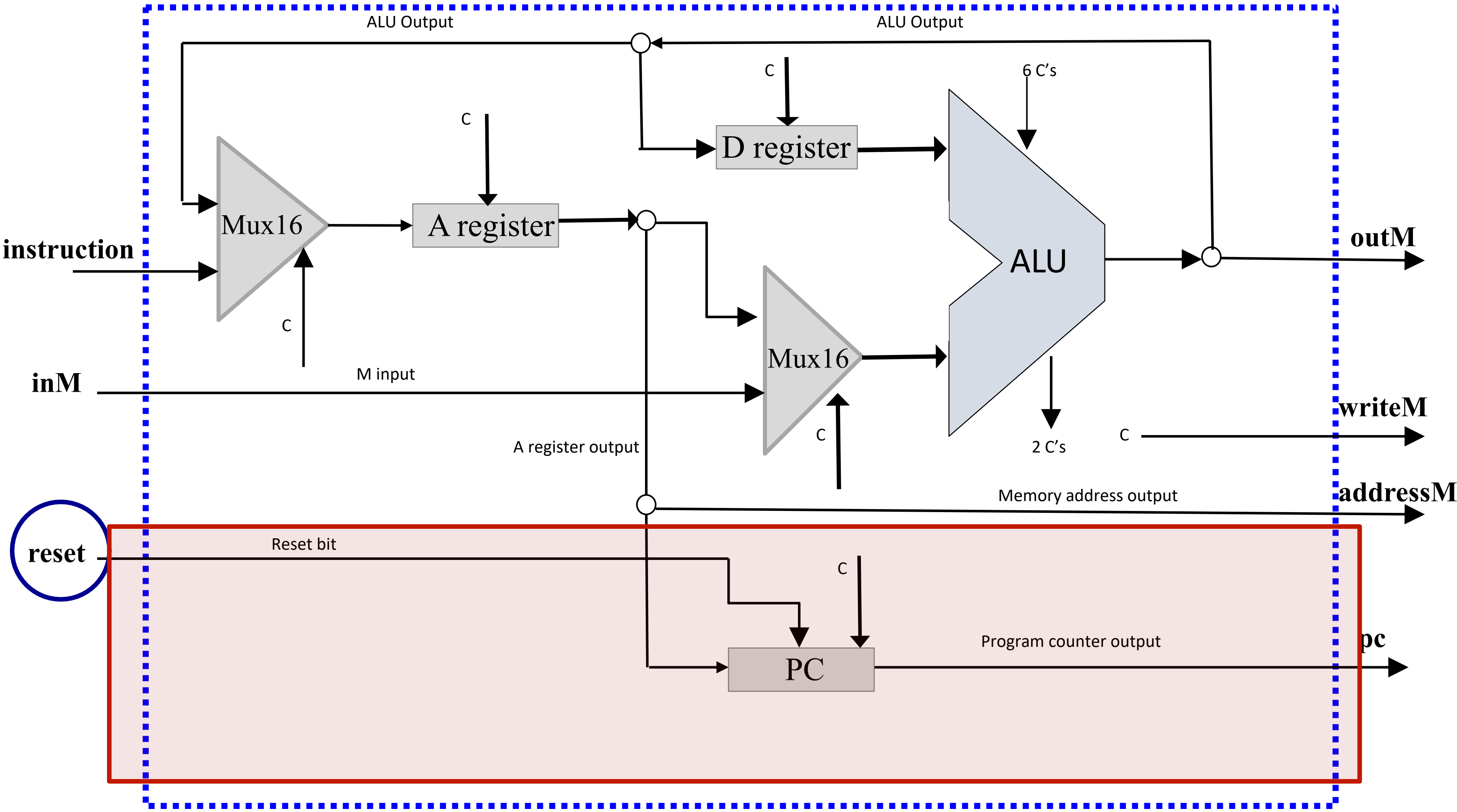
Load Input of D-Register Chip



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

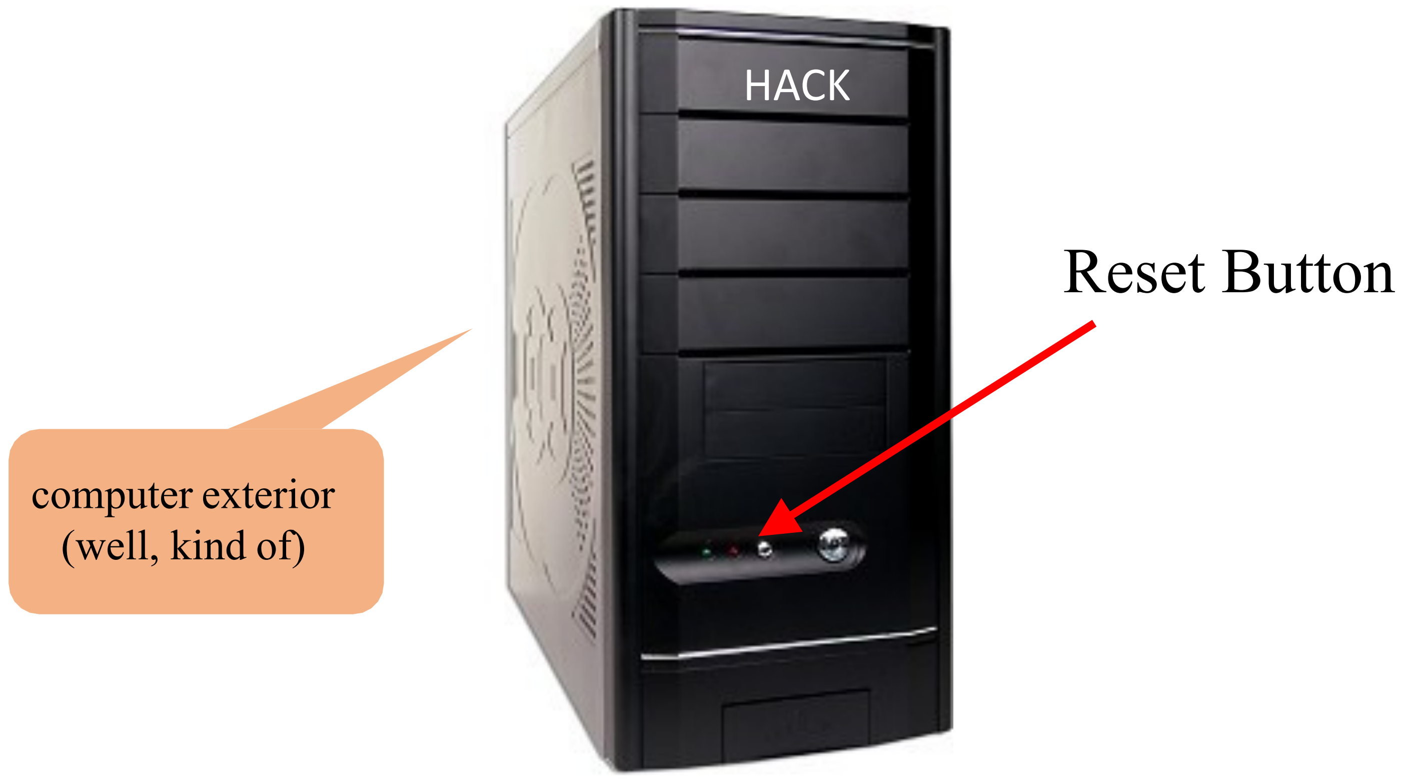
Control Logic of PC Register

Control Logic of CPU



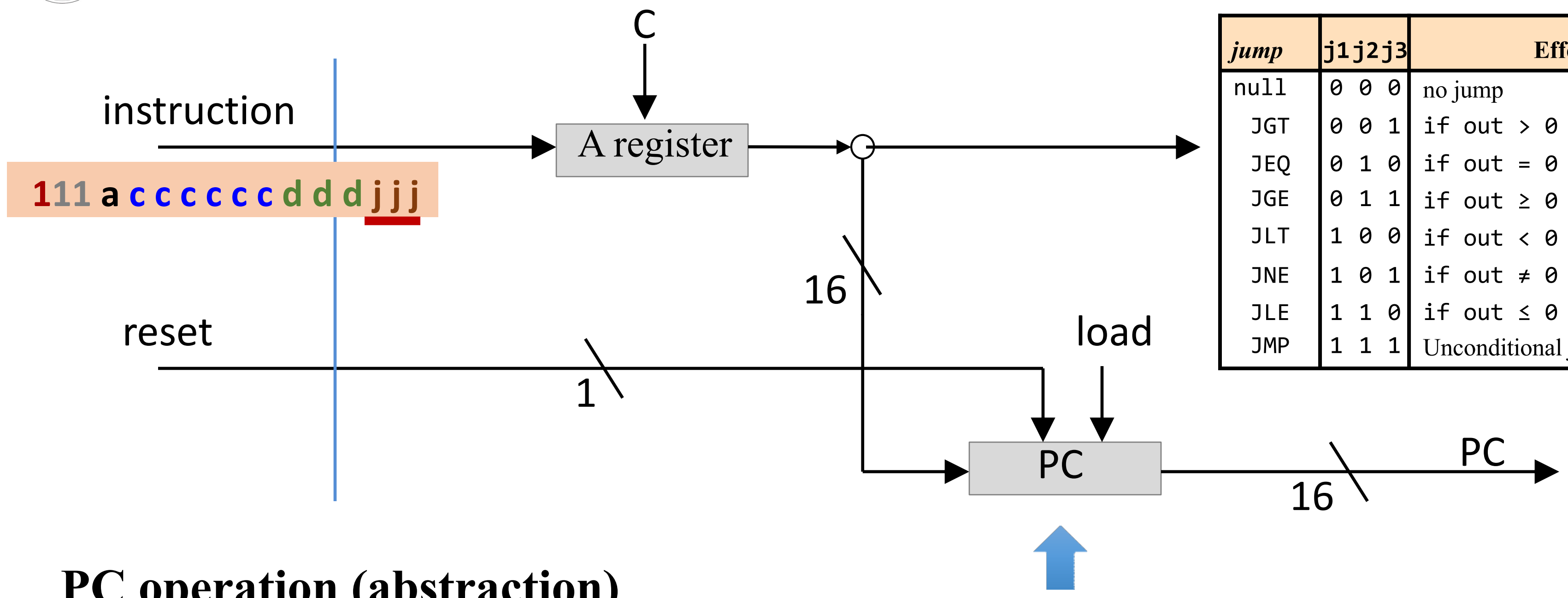
(each "C" symbol represents a control bit)

Possible Outside View of Hack Computer



- The computer is loaded with some program
- Pushing reset button causes the program to start running from beginning

Control Abstraction

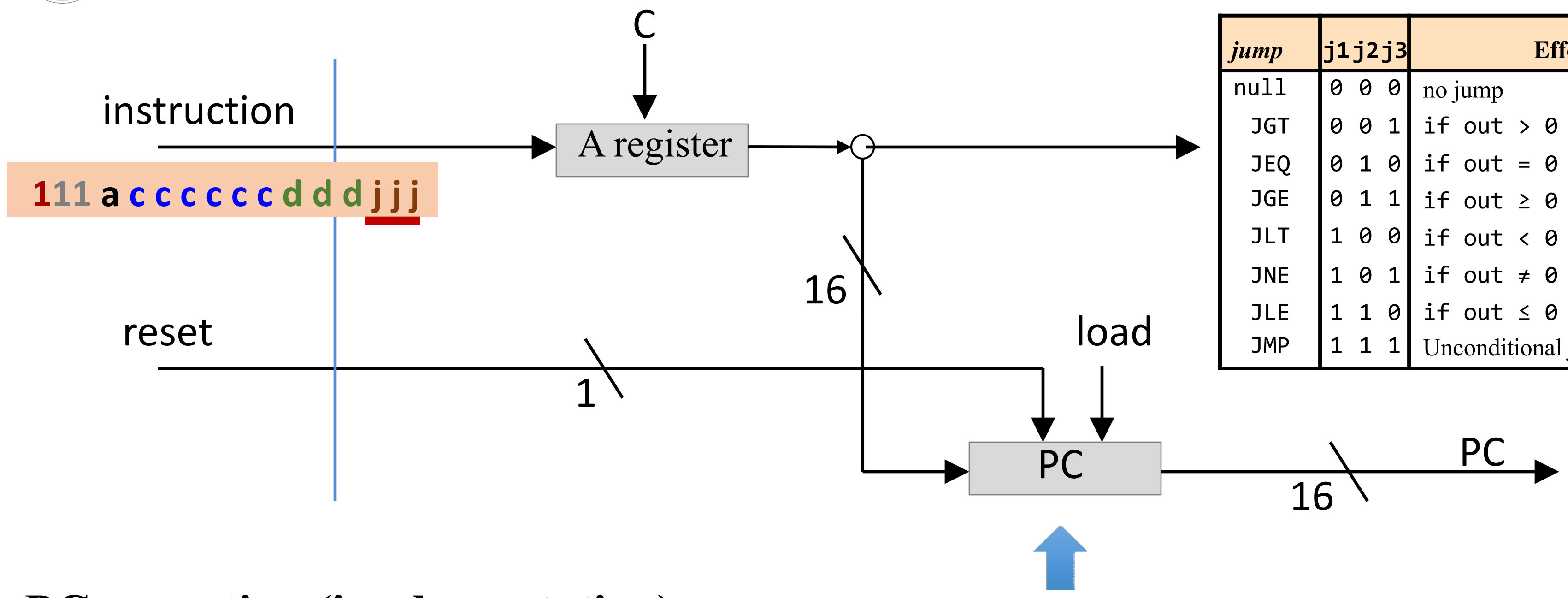


jump	j1j2j3	Effect
null	0 0 0	no jump
JGT	0 0 1	if out > 0 jump
JEQ	0 1 0	if out = 0 jump
JGE	0 1 1	if out ≥ 0 jump
JLT	1 0 0	if out < 0 jump
JNE	1 0 1	if out ≠ 0 jump
JLE	1 1 0	if out ≤ 0 jump
JMP	1 1 1	Unconditional jump

PC operation (abstraction)

- Emits the address of the next instruction
- reset: PC=0
- no jump: **000** PC++
- goto: **111** PC=A
- conditional goto: if (condition) PC= A else PC++

Control Implementation



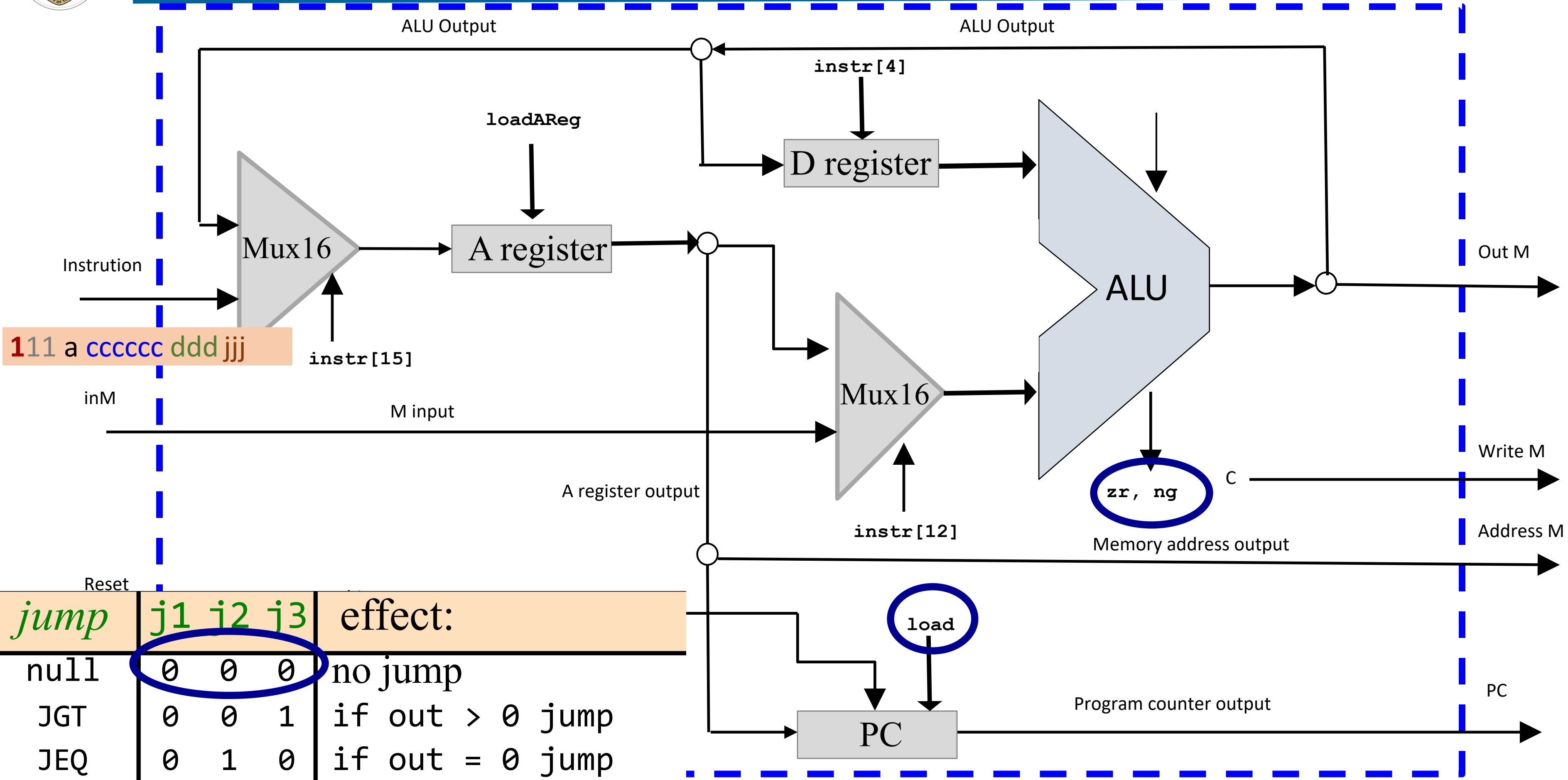
jump	j1j2j3	Effect
null	0 0 0	no jump
JGT	0 0 1	if out > 0 jump
JEQ	0 1 0	if out = 0 jump
JGE	0 1 1	if out ≥ 0 jump
JLT	1 0 0	if out < 0 jump
JNE	1 0 1	if out ≠ 0 jump
JLE	1 1 0	if out ≤ 0 jump
JMP	1 1 1	Unconditional jump

address of next instruction

PC operation (implementation)

```
if (reset == 1)
    PC = 0
else
    load = f(jump bits, ALU control outputs (zr,ng))
    if (load == 1)    PC = A      // jump
    else         PC++            // next instruction
```

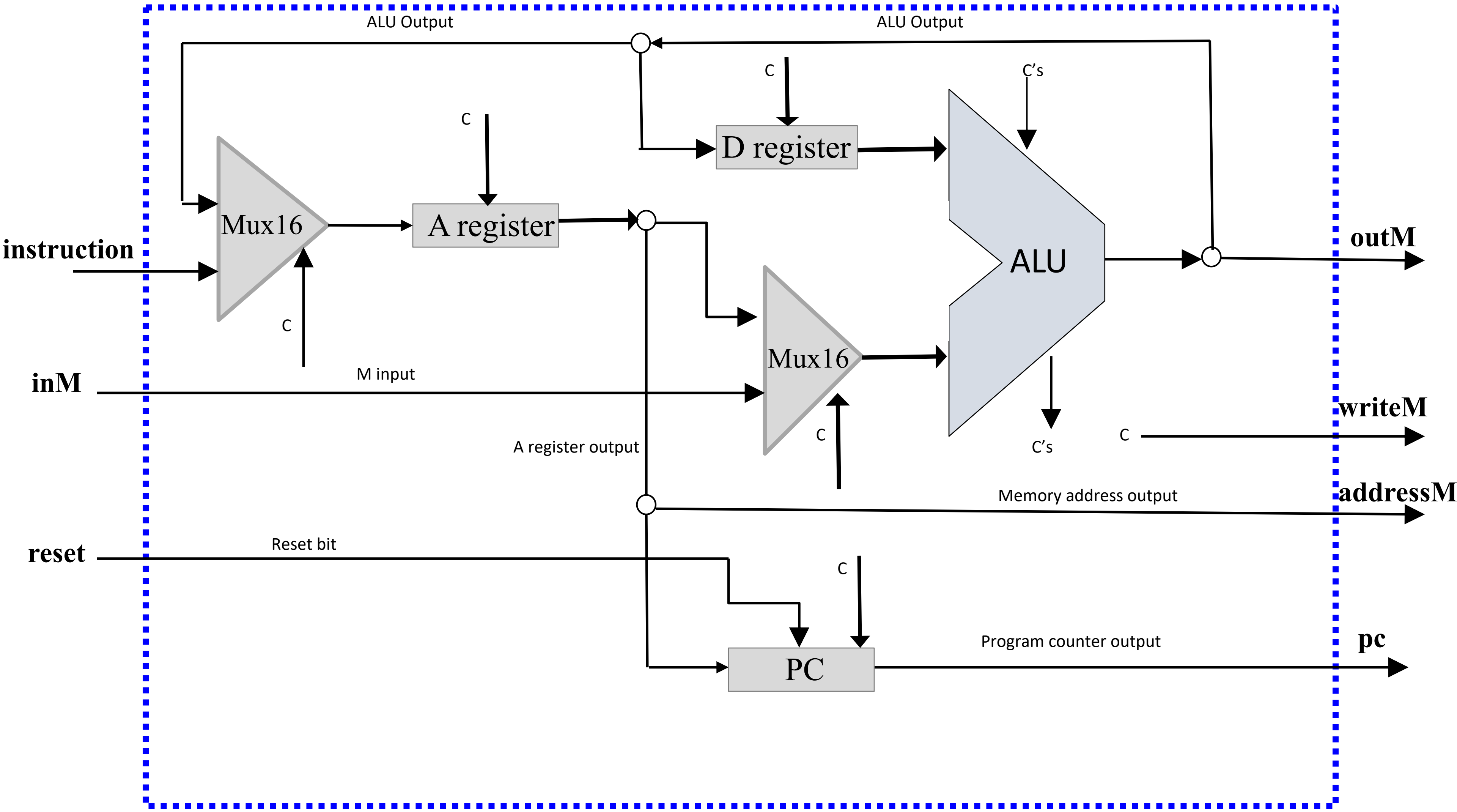
Load Input of PC-Register



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

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

Hack CPU Implementation



That's It! **All that remains is to actually build it ☺**

Things To Do

O.k., and now you'll do exactly what I'm telling you !
Access denied