CS0 2130

TOY Proccessor

Daniel G. Graham PhD

1. A full overview of Toy ISA
2. Build a machine (Toy Processor) that can execute our Toy ISA
3. Discuss the fetch, decode, execute, memory, and writeback stages
4. Discuss the steps need to synthesize or toy processor

FULL ISA

icode	b	meaning
0		`rA = rB`
1		`rA += rB`
2		`rA &= rB`
3		`rA` = read from memory at address `rB`
4		write `rA` to memory at address `rB`
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`
6	0	`rA` = read from memory at `pc + 1`
	1	`rA` += read from memory at `pc + 1`
	2	`rA` &= read from memory at `pc + 1`
	3	`rA` = read from memory at the address stored at `pc + 1`
		For icode 6, increase `pc` by 2 at end of instruction
7		Compare `rA` as 8-bit 2's-complement to `0`
		if `rA <= 0` set `pc = rB`
		else increment `pc` as normal

Let look at each of these instructions

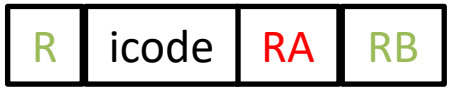
EXAM QUESTION

9. [8 points] Complete the table below listing all the register values as hex digits after the following code executes. Assume that all registers start with value `0x00`.

```
6c 20 60 FF 2c 04 54 19
```

Register	Value
0	
1	
2	
3	

icode	b	meaning
0		`rA = rB`
1		`rA += rB`
2		`rA &= rB`
3		`rA` = read from memory at address `rB`
4		write `rA` to memory at address `rB`
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`
6	0	`rA` = read from memory at `pc + 1`
	1	`rA` += read from memory at `pc + 1`
	2	`rA` &= read from memory at `pc + 1`
	3	`rA` = read from memory at the address stored at `pc + 1`
		For icode 6, increase `pc` by 2 at end of instruction
7		Compare `rA` as 8-bit 2's-complement to `0`
		if `rA <= 0` set `pc = rB`
		else increment `pc` as normal




```
6c 20 60 FF 2c 04 54 19
```

Register	Value
0	
1	
2	
3	

EXAM QUESTION

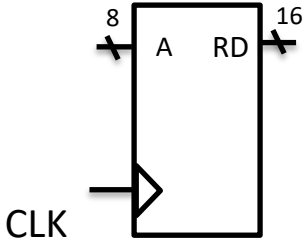
9. [8 points] Complete the table below listing all the register values as hex digits after the following code executes. Assume that all registers start with value `0x00`.

```
6c 20 60 FF 2c 04 54 19
```

Register	Value
0	09
1	07
2	00
3	2D

LET'S BUILD A TOY PROCESSOR THAT CAN EXECUTE OUR TOY ISA

INSTRUCTION MEMORY AND INSTRUCTION REGISTER

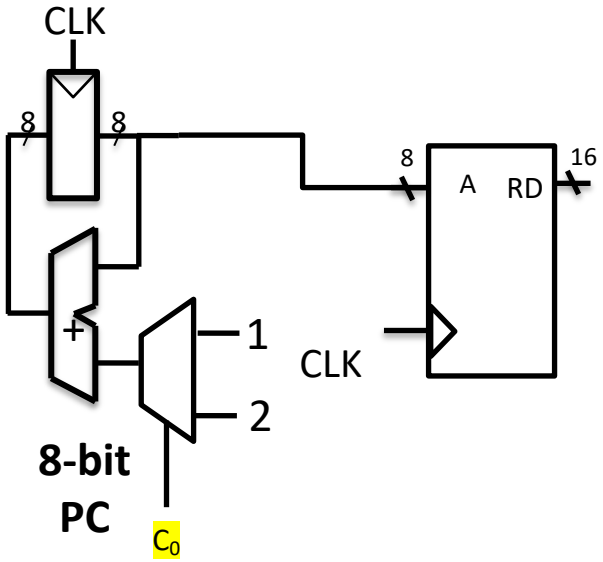


Instruction register (IR)

Our diagram is going to have several comments so I will not draw the IR

Note: input and output widths on the Instruction memory. The memory is byte-addressable but reads 2 bytes at a time

1 BYTE AND 2 BYTE INSTRUCTIONS

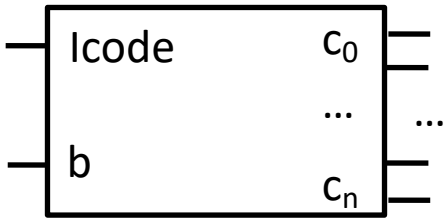


We'll add a mux that will select passing one to adder or two.

The mux will be controlled with a control line C_0. But what component provides the control signal? Answer the Controller

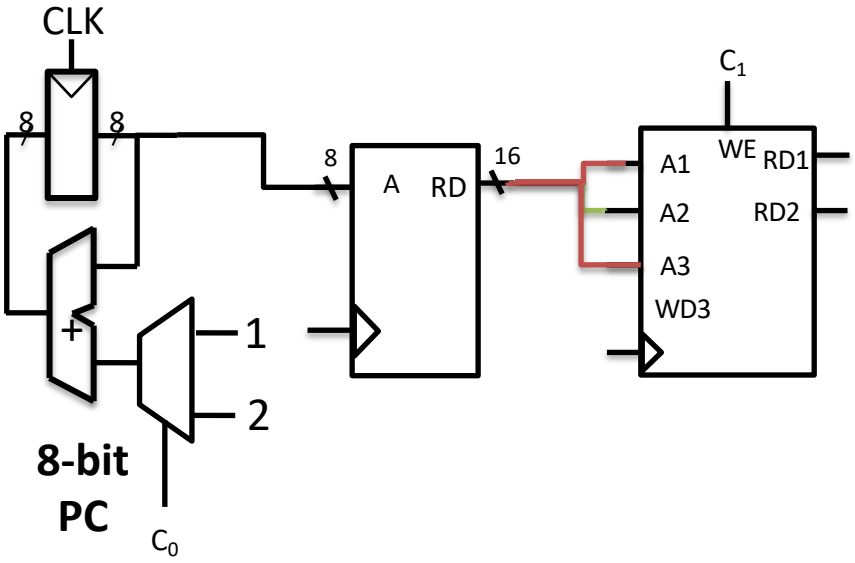
HARDWIRED CONTROL UNIT

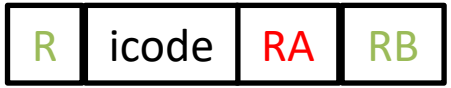
icode	b	C_0	....	C_n
6	x	1		
....				



icode	b	meaning
0		`rA = rB`
1		`rA += rB`
2		`rA &= rB`
3		`rA` = read from memory at address `rB`
4		write `rA` to memory at address `rB`
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`
6	0	`rA` = read from memory at `pc + 1`
	1	`rA` += read from memory at `pc + 1`
	2	`rA` &= read from memory at `pc + 1`
	3	`rA` = read from memory at the address stored at `pc + 1`
		For icode 6, increase `pc` by 2 at end of instruction
7		Compare `rA` as 8-bit 2's-complement to `0`
		if `rA <= 0` set `pc = rB`
		else increment `pc` as normal

icode b	meaning
0	`rA = rB`
1	`rA += rB`
2	`rA &= rB`



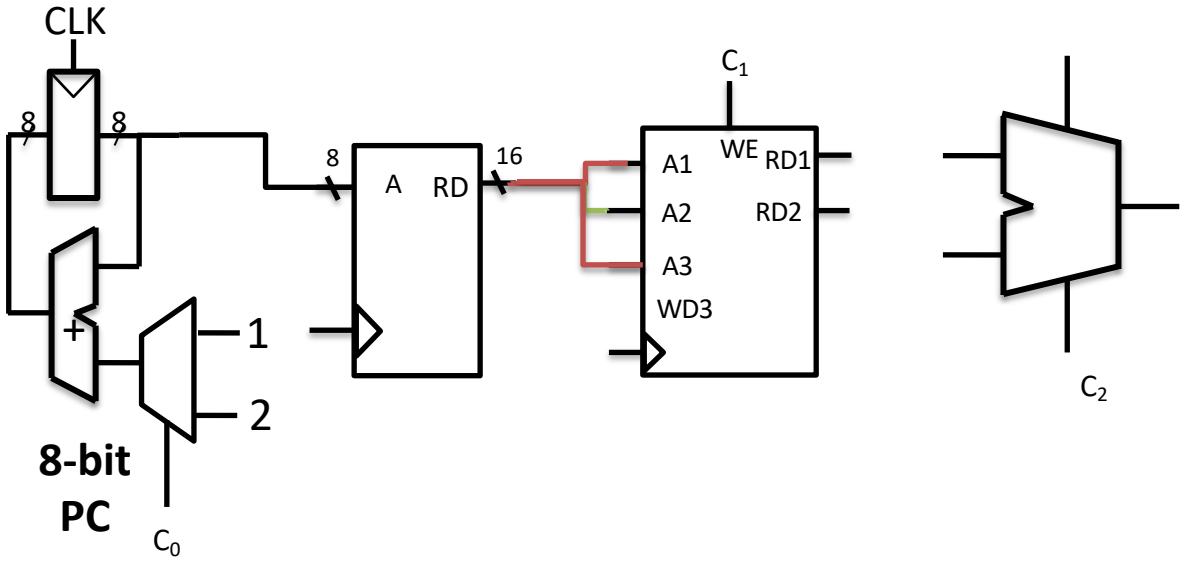


Only the relevant part instruction is going to register file input.
Icode section and RB will also go to the controller.

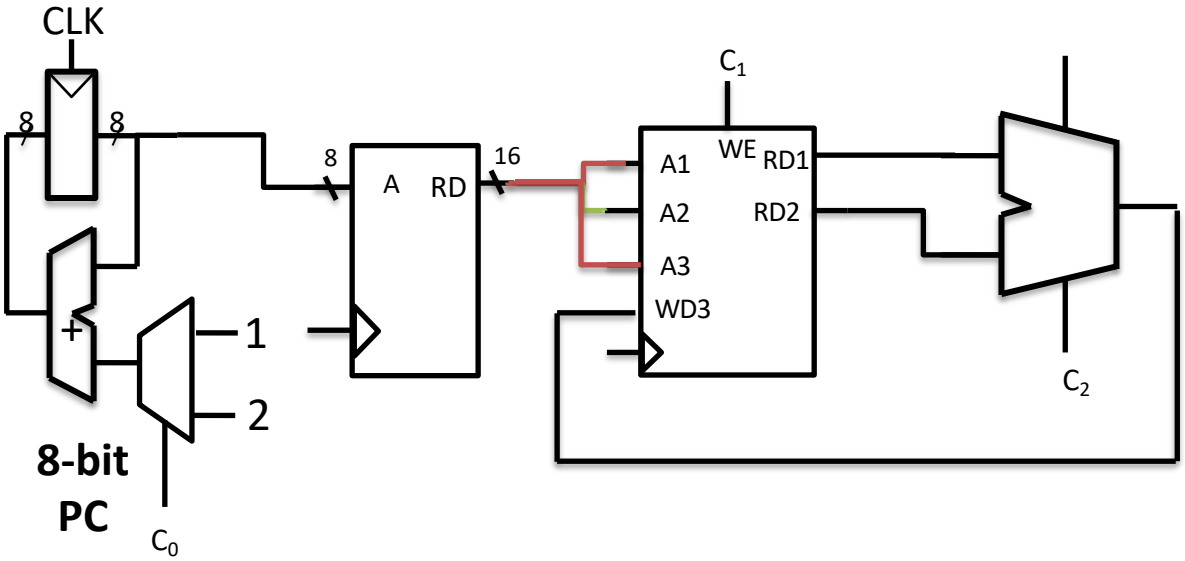
These are not depicted for simplicity

icode b	meaning
0	`rA = rB`
1	`rA += rB`
2	`rA &= rB`

How would we wire up the ALU?



icode b	meaning
0	`rA = rB`
1	`rA += rB`
2	`rA &= rB`

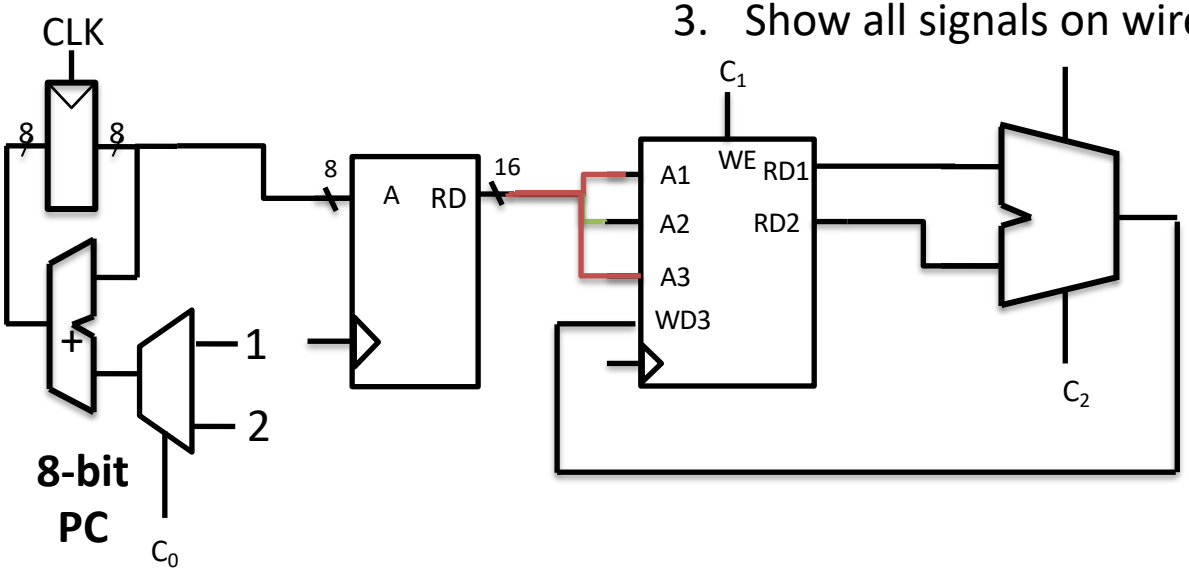


Let's run three instructions

R	icode	RA	RB
0	000	00	10
0	001	10	10
0	010	10	11

icode b	meaning
0	`rA = rB`
1	`rA += rB`
2	`rA &= rB`

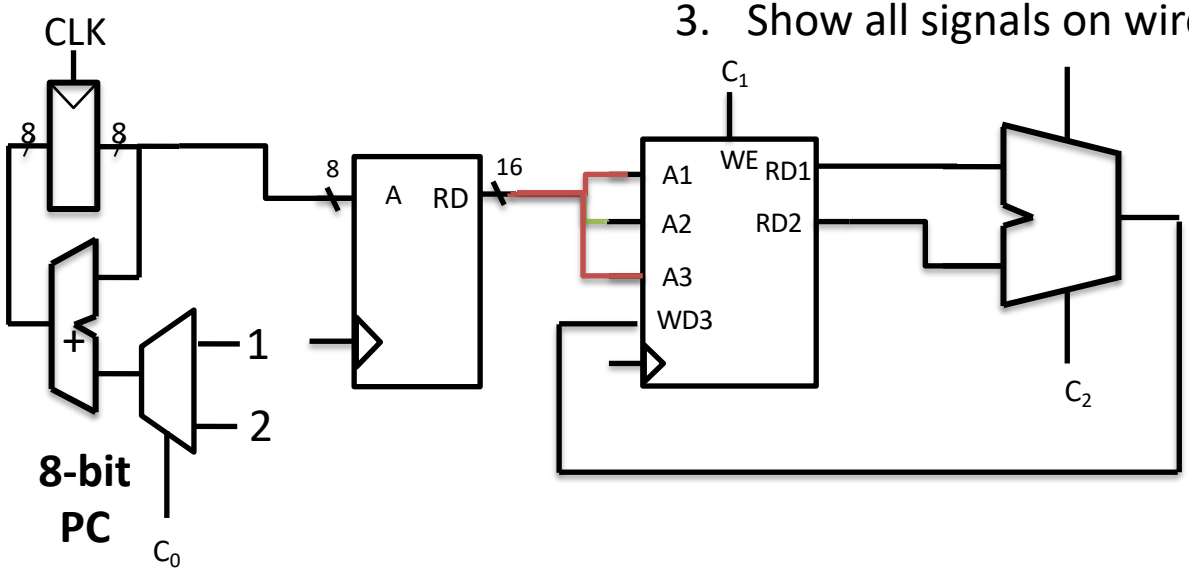
1. **Hex encode each instruction**
2. Assume R0 = 1, R1 = 4, R2 = 3, R3 = 0
3. Show all signals on wires



R	icode	RA	RB
0	000	00	10
0	001	10	10
0	010	10	11

icode b	meaning
0	`rA = rB`
1	`rA += rB`
2	`rA &= rB`

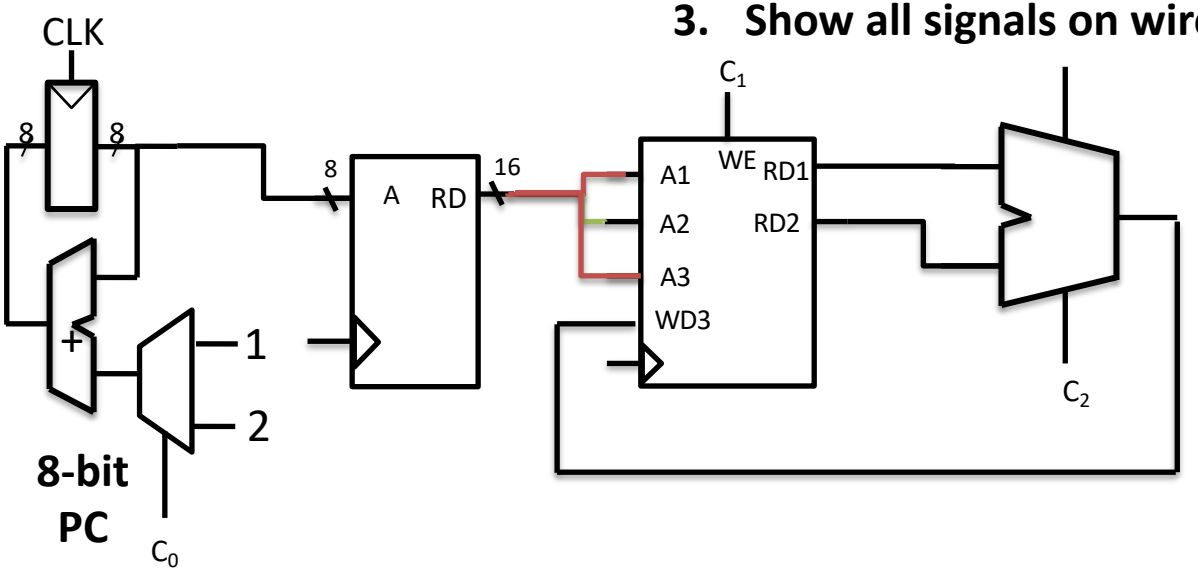
1. Hex encode each instruction
2. **Assume R0 = 1, R1 = 4, R2 = 3, R3 = 0**
3. Show all signals on wires



R	icode	RA	RB
0	000	00	10
0	001	10	10
0	010	10	11

icode b	meaning
0	`rA = rB`
1	`rA += rB`
2	`rA &= rB`

1. Hex encode each instruction
2. Assume R0 = 1, R1 = 4, R2 = 3, R3 = 0
3. **Show all signals on wires**



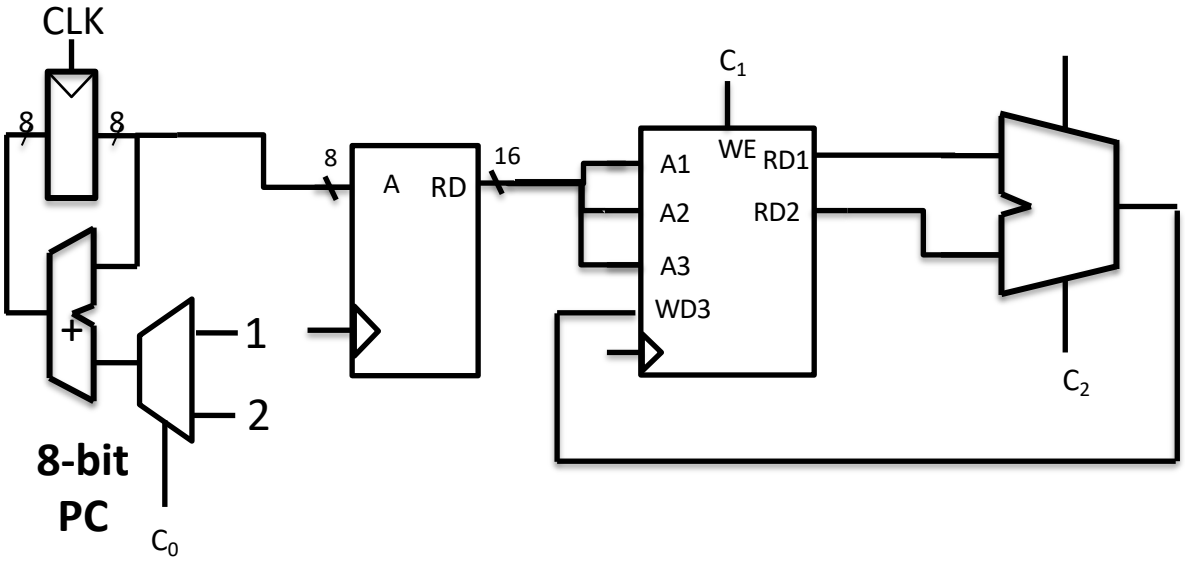
R	icode	RA	RB
0	000	00	10
0	001	10	10
0	010	10	11

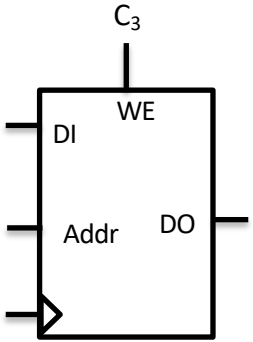
LET’S NEXT INSTRUCTIONS THAT USE MAIN MEMORY

icode	b	meaning
0		`rA = rB`
1		`rA += rB`
2		`rA &= rB`
3		`rA` = read from memory at address `rB`
4		write `rA` to memory at address `rB`
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`
6	0	`rA` = read from memory at `pc + 1`
	1	`rA` += read from memory at `pc + 1`
	2	`rA` &= read from memory at `pc + 1`
	3	`rA` = read from memory at the address stored at `pc + 1`
		For icode 6, increase `pc` by 2 at end of instruction
7		Compare `rA` as 8-bit 2's-complement to `0`
		if `rA <= 0` set `pc = rB`
		else increment `pc` as normal

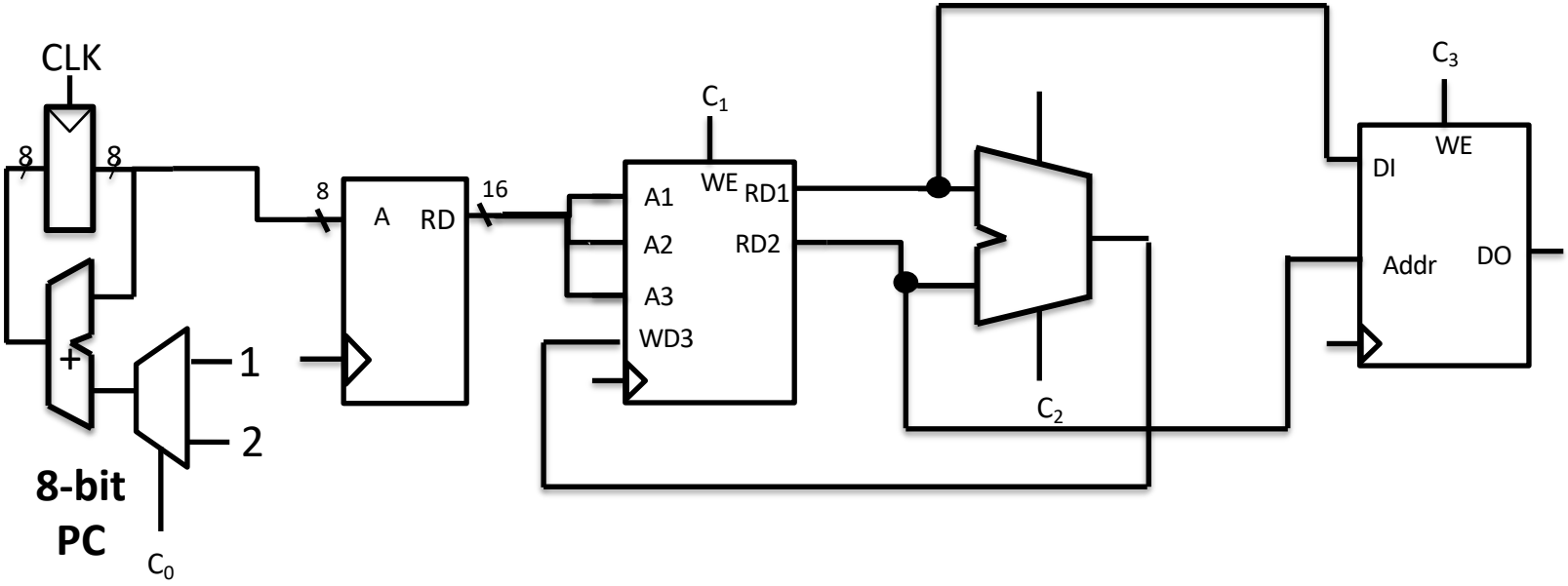
3	**rA** = read from memory at address **rB**
4	write **rA** to memory at address **rB**

Talk to use neighbor to see if you can wire this up.



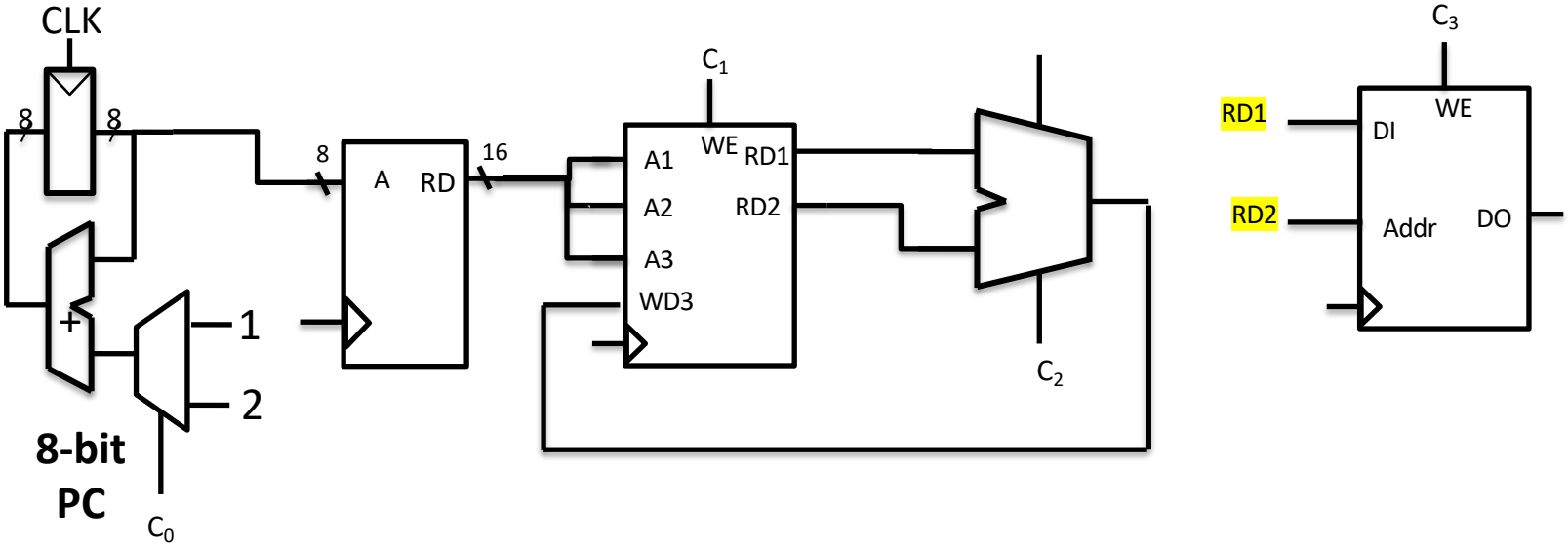


3	**rA** = read from memory at address **rB**
4	write **rA** to memory at address **rB**



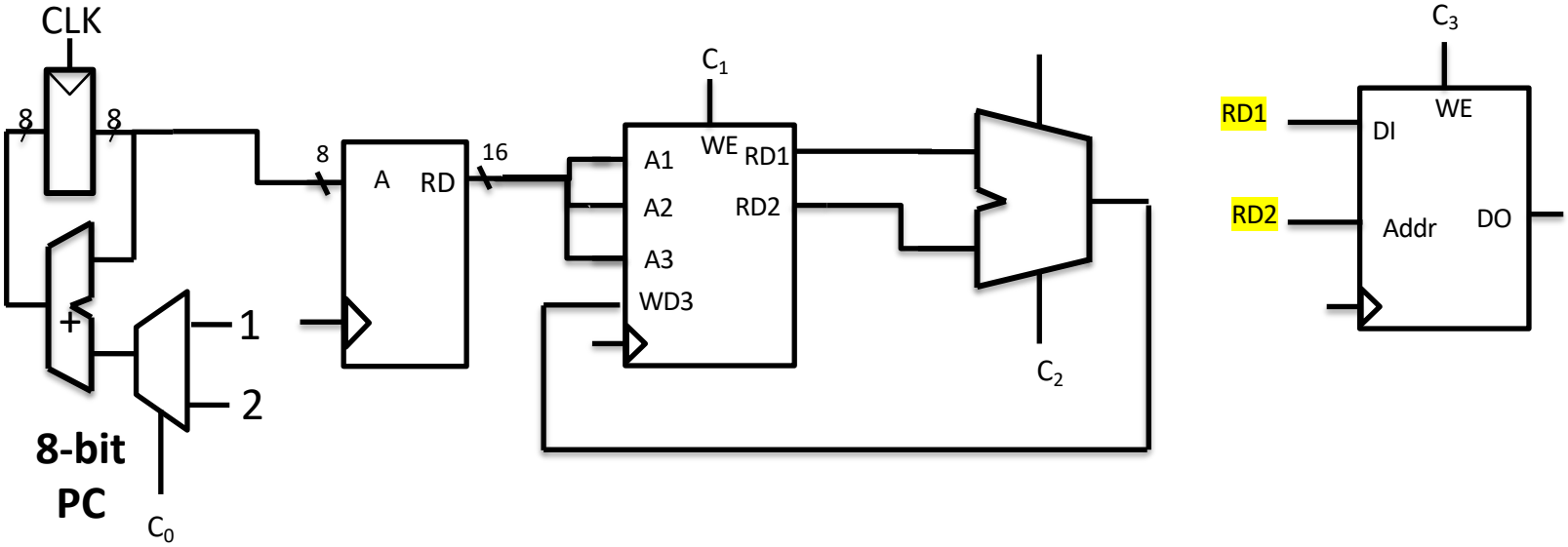
3	**rA** = read from memory at address **rB**
4	write **rA** to memory at address **rB**

Writing labels for a cleaner look



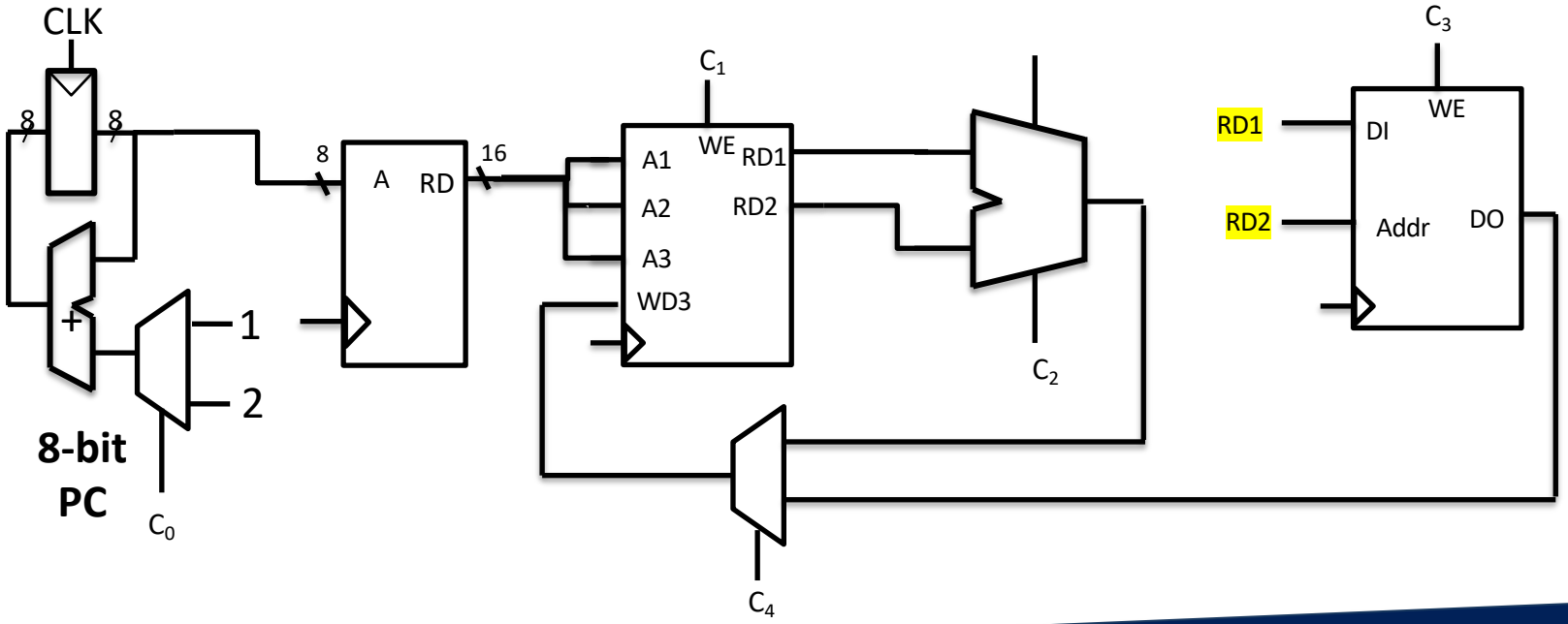
3	**rA** = read from memory at address **rB**
4	write **rA** to memory at address **rB**

Looks like we have a conflict. Thoughts on how we could fix this?



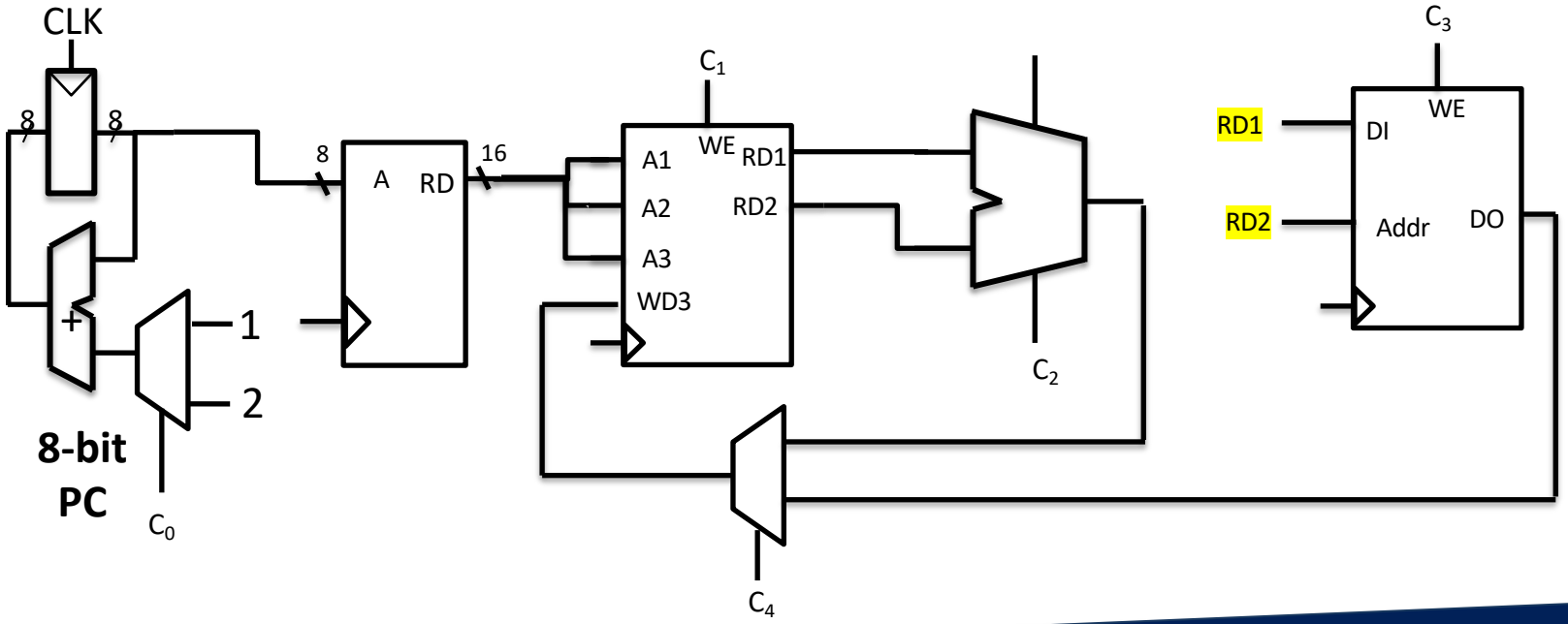
3	**rA** = read from memory at address **rB**
4	write **rA** to memory at address **rB**

Let's execute some sample instructions



1	`rA += rB`
2	`rA &= rB`
3	`rA` = read from memory at address `rB`
4	write `rA` to memory at address `rB`

Let's execute some sample instructions



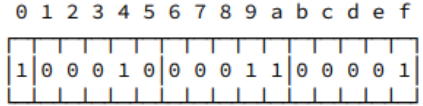
icode	b	meaning
0		`rA = rB`
1		`rA += rB`
2		`rA &= rB`
3		`rA` = read from memory at address `rB`
4		write `rA` to memory at address `rB`
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`
6	0	`rA` = read from memory at `pc + 1`
	1	`rA` += read from memory at `pc + 1`
	2	`rA` &= read from memory at `pc + 1`
	3	`rA` = read from memory at the address stored at `pc + 1`
		For icode 6, increase `pc` by 2 at end of instruction
7		Compare `rA` as 8-bit 2's-complement to `0`
		if `rA <= 0` set `pc = rB`
		else increment `pc` as normal

NEXT TIME

(d) Throughout the semester we looked at various components associated with a single cycle processor. Here we designed a simple machine that we call the Interceptor 1000. This machine has only has two instructions, listed below by their opcodes:

0. `nop` which does nothing
1. `intcep $x, $m, $b` which computes $mx + b$

An example instruction for this machine might be `intcep 2 3 1` meaning x is 2, m is 3 and b is 1. This instruction would be encoded as follows:



In the diagram below, `O(1)` represents the opcode that is 1 bit long, while `X(5)` represents the `x` value that is 5 bits long.

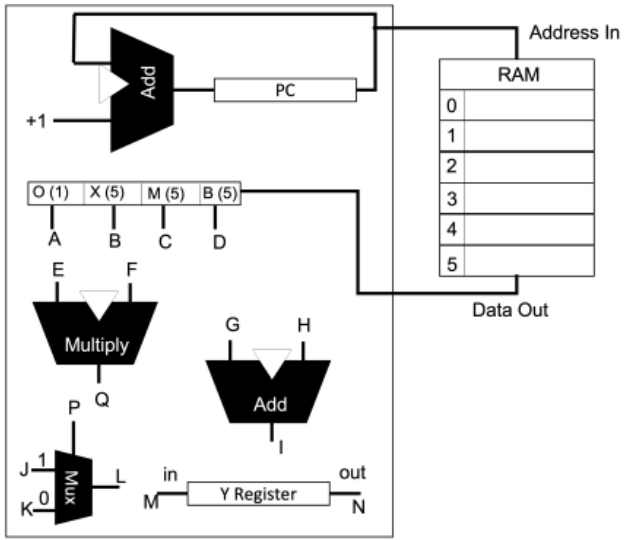
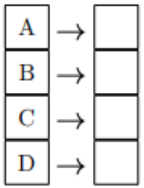
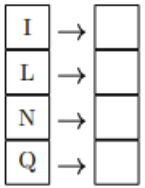


Figure 1: Block Diagram for Interceptor 10000

EXAM QUESTION

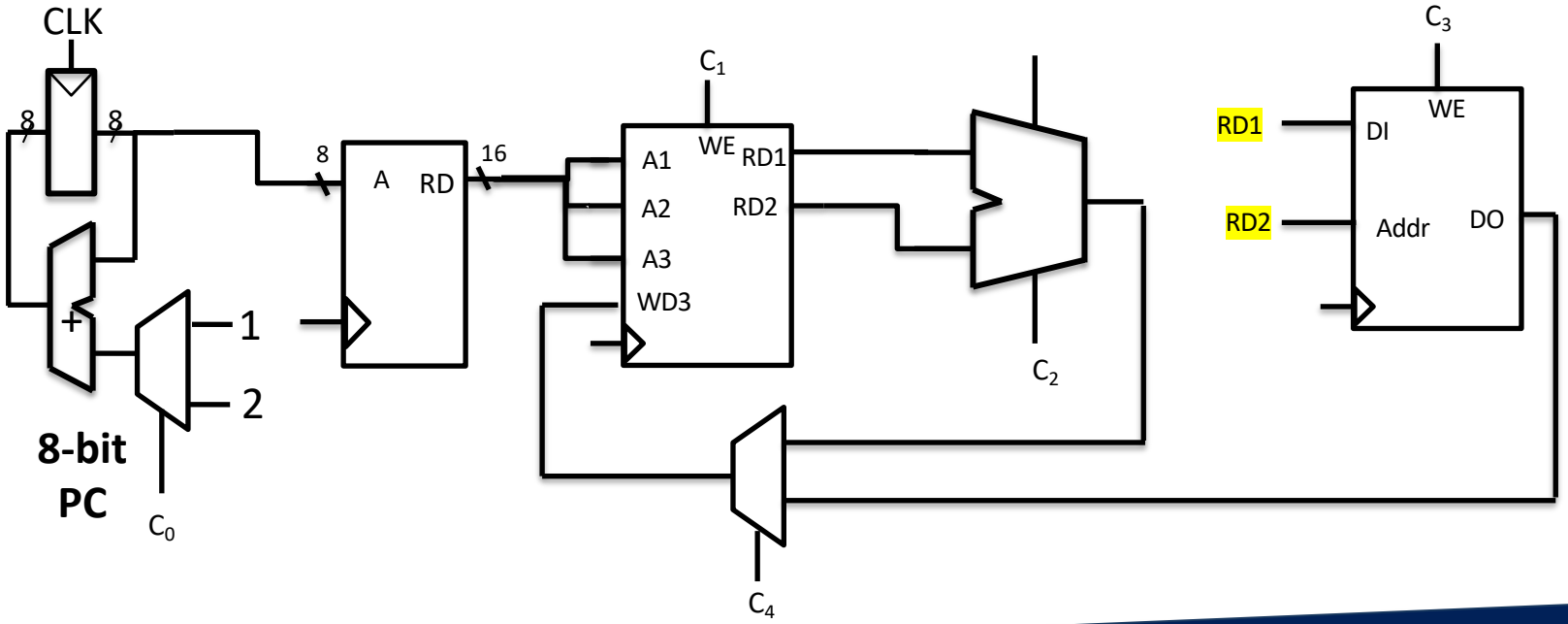
Show how to complete the diagram by writing the name of the wire each given wire should be connected to. For example, if A is connected to E, write E in the box next to A.





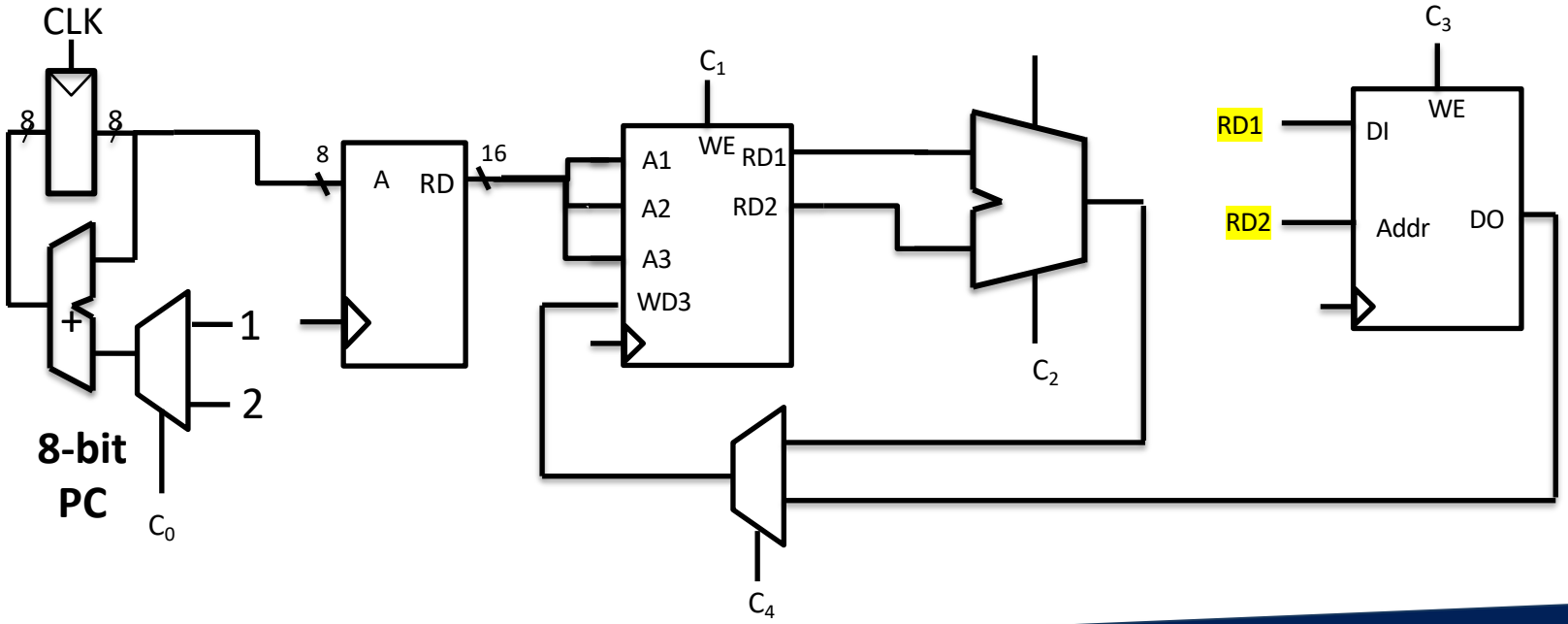
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`

Draw out the flow here



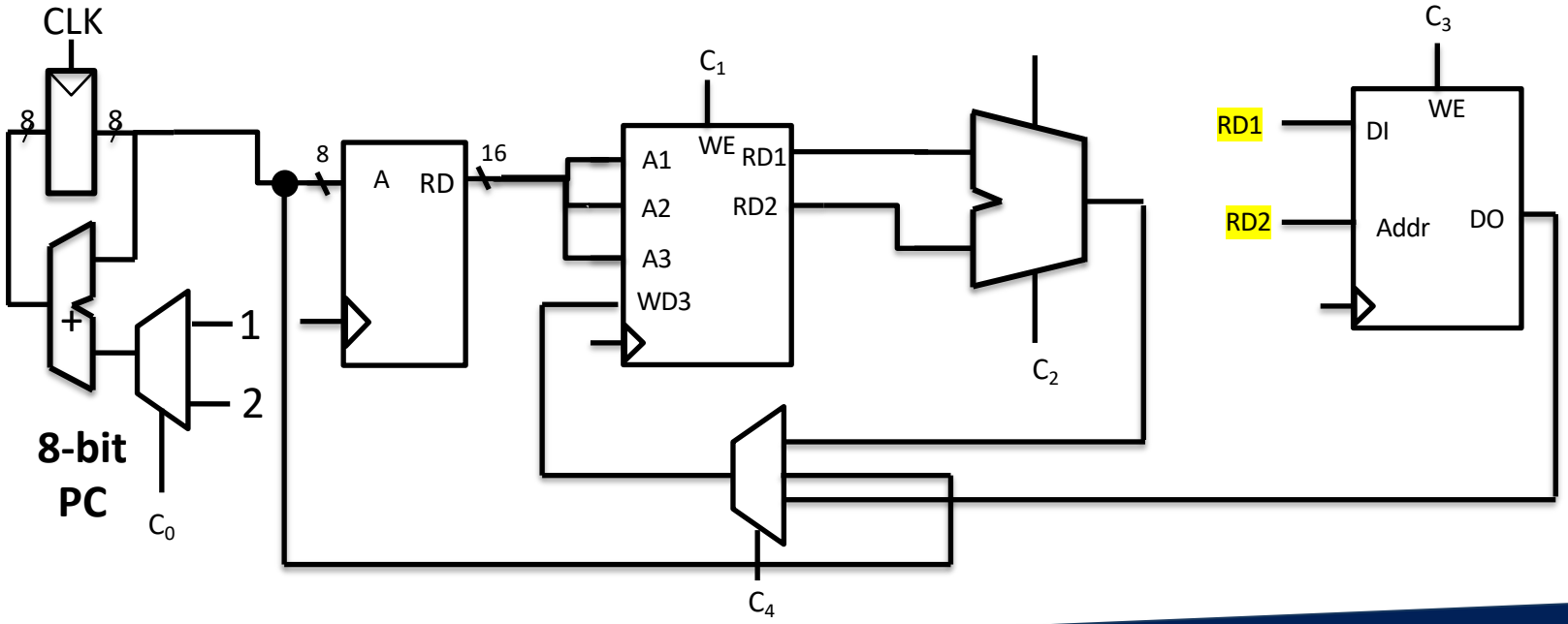
5	0	rA = ~rA
	1	rA = -rA
	2	rA = !rA
	3	rA = pc

How can we update RA with the PC value?



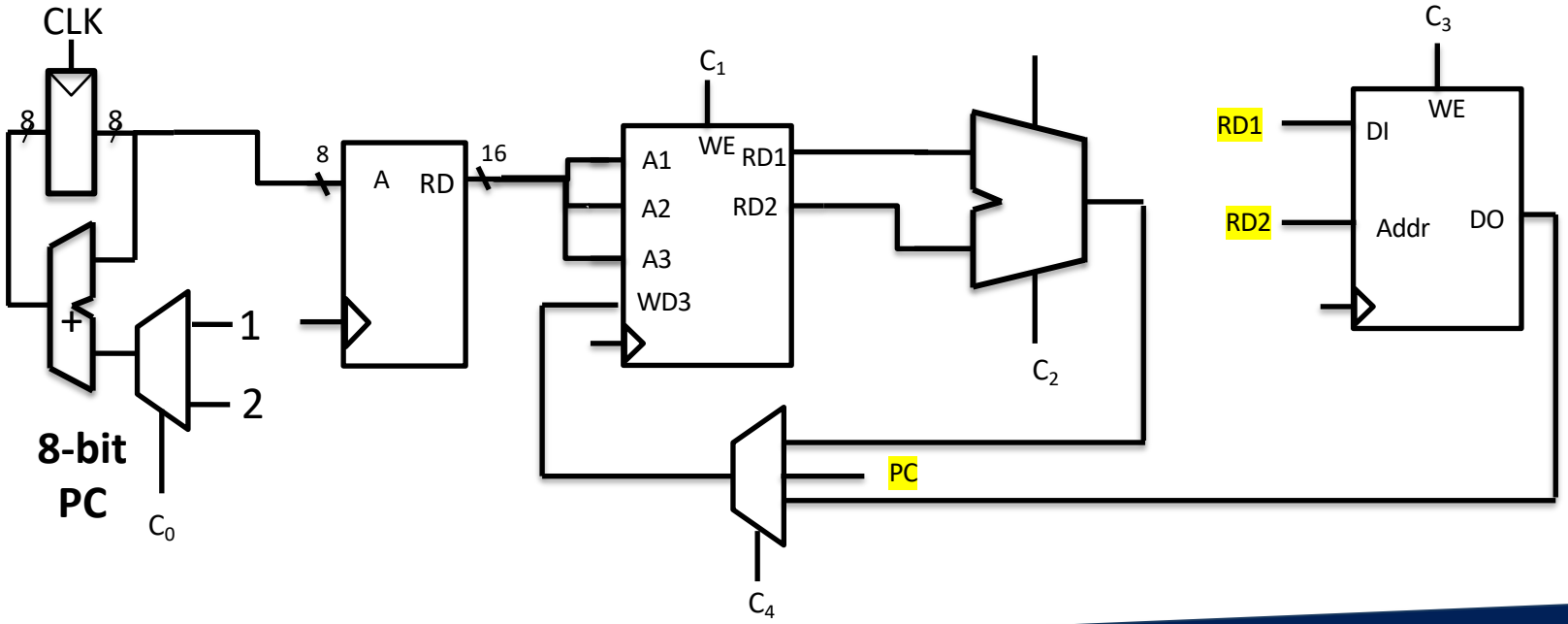
5	0	rA = ~rA
	1	rA = -rA
	2	rA = !rA
	3	rA = pc

How can we update RA with the PC value?



5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`

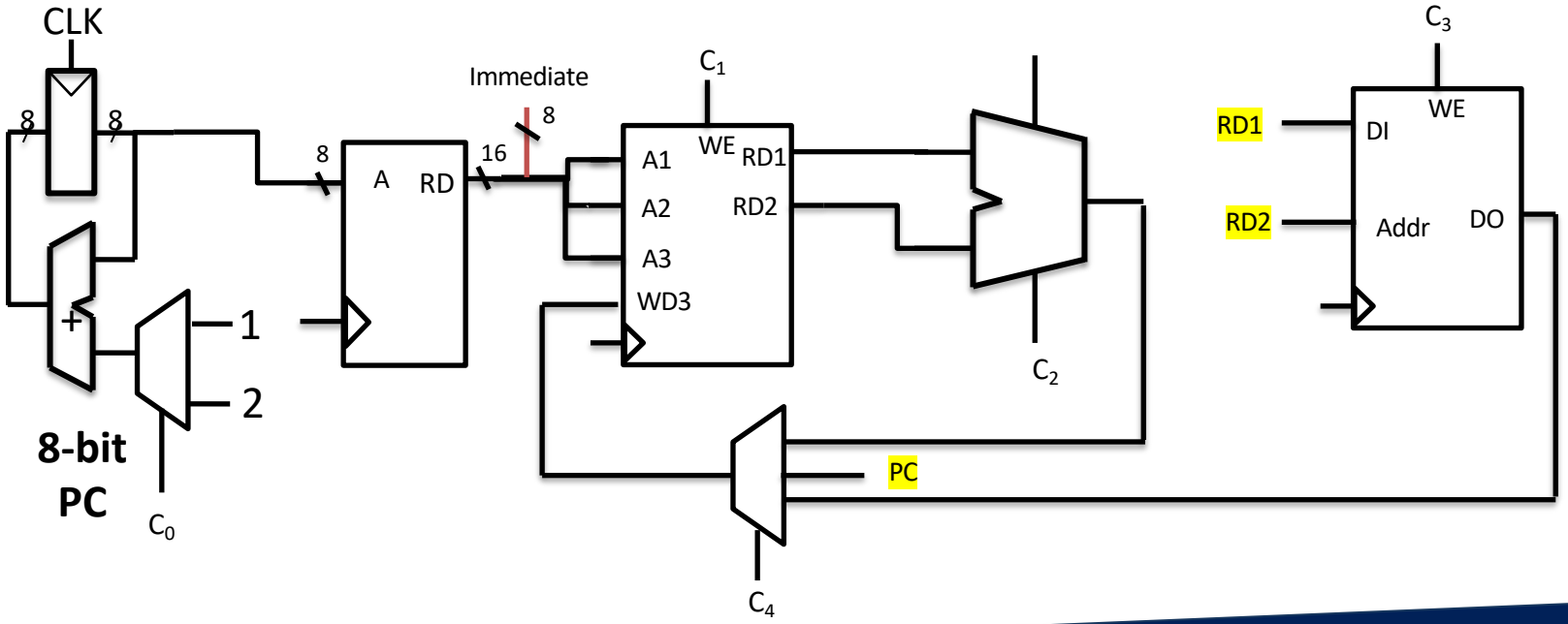
Changed it to just be the label



6	0	**rA** = read from memory at **pc + 1**
	1	**rA** += read from memory at **pc + 1**
	2	**rA** &= read from memory at **pc + 1**
	3	**rA** = read from memory at the address stored at **pc + 1**

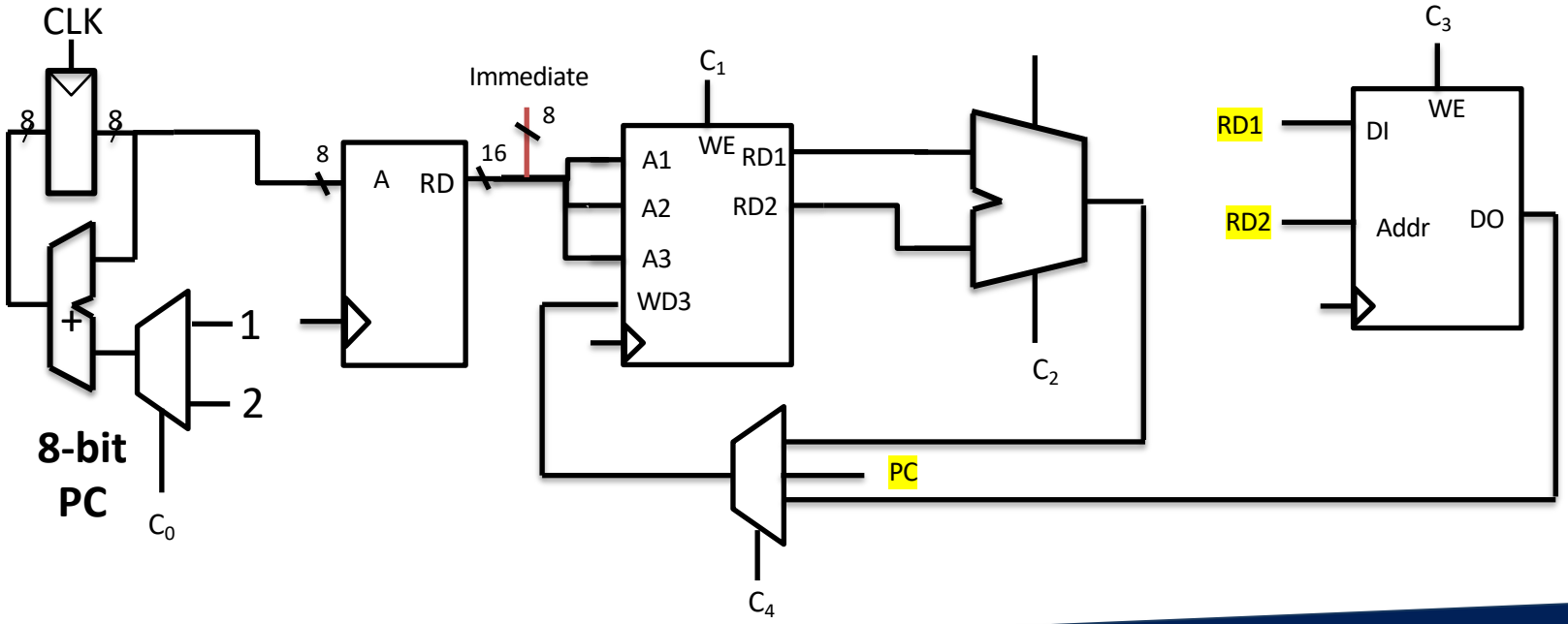
The immediate

immediate



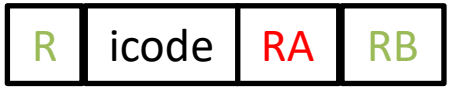
6	0	**rA** = read from memory at **pc + 1**
	1	**rA** += read from memory at **pc + 1**
	2	**rA** &= read from memory at **pc + 1**
	3	**rA** = read from memory at the address stored at **pc + 1**

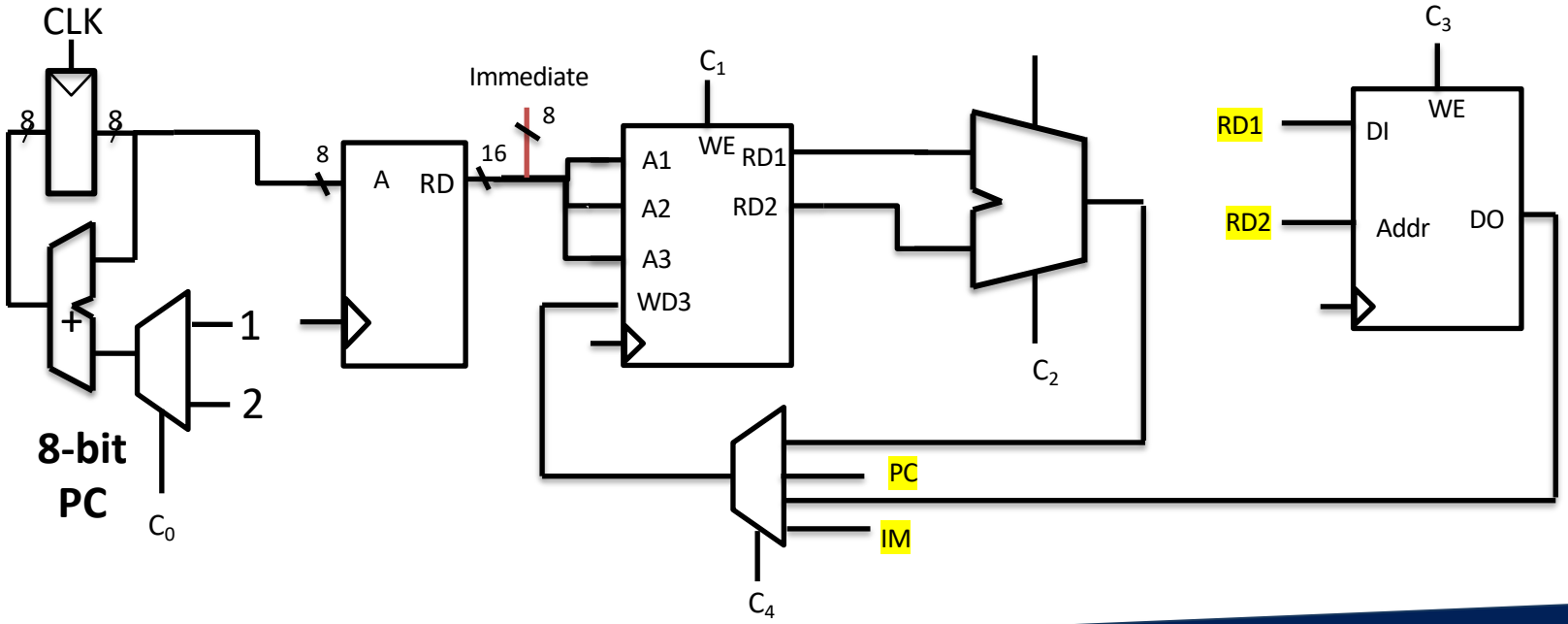
How could we implement this instruction



6	0	**rA** = read from memory at **pc + 1**
	1	**rA** += read from memory at **pc + 1**
	2	**rA** &= read from memory at **pc + 1**
	3	**rA** = read from memory at the address stored at **pc + 1**

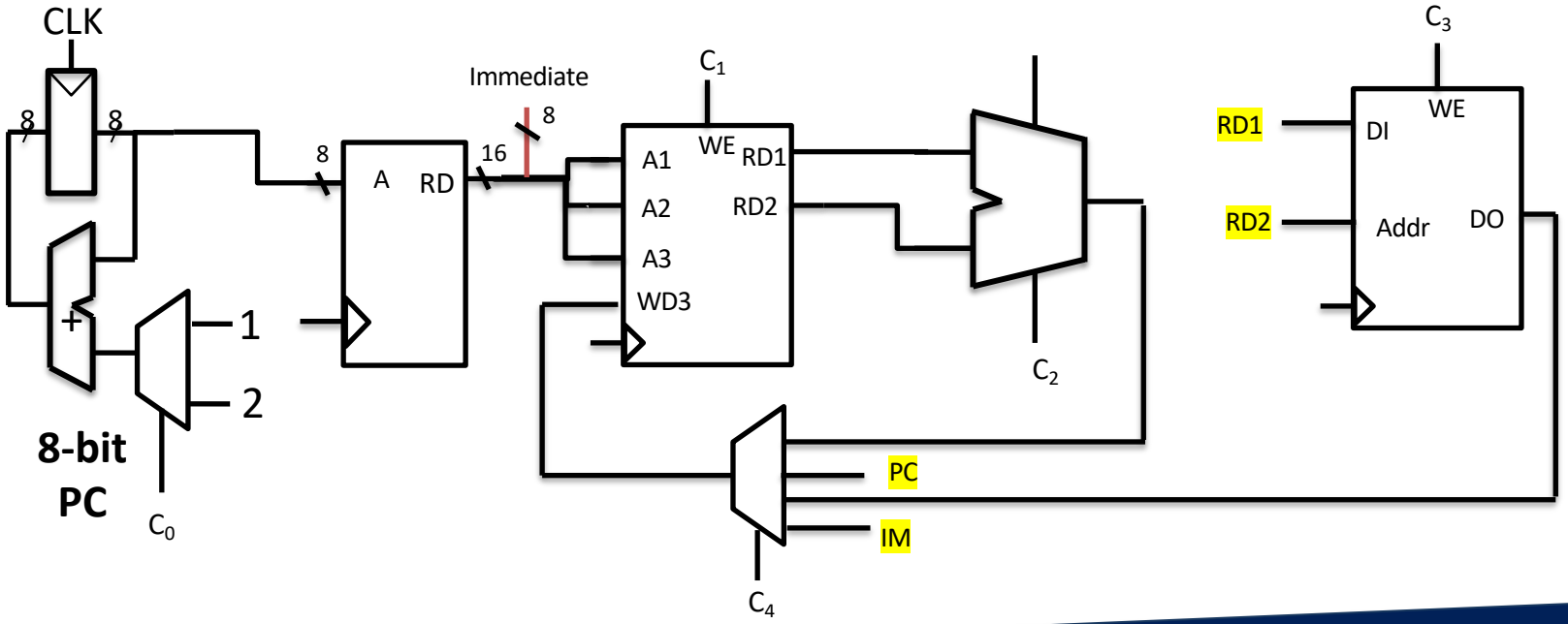
Walk through the flow of an example instruction





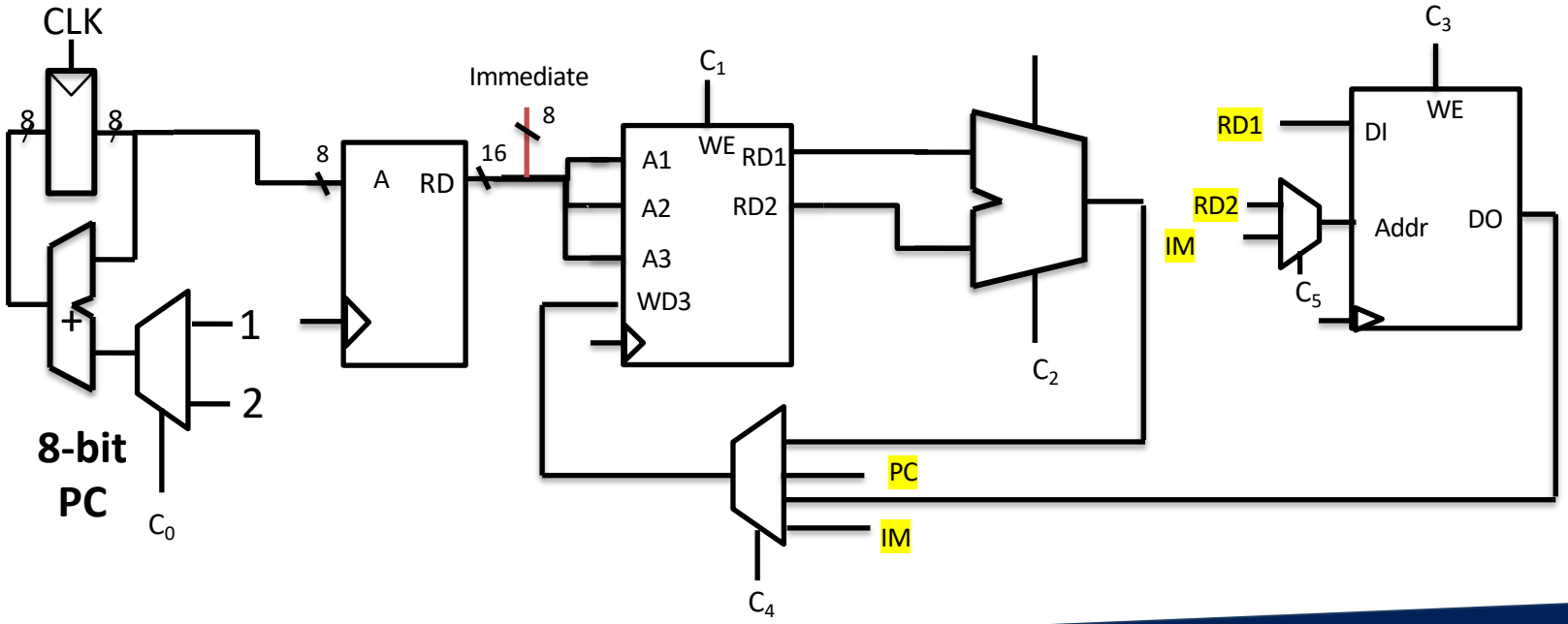
6	0	`rA` = read from memory at `pc + 1`
	1	`rA` += read from memory at `pc + 1`
	2	`rA` &= read from memory at `pc + 1`
	3	`rA` = read from memory at the address stored at `pc + 1`

What about this instruction?



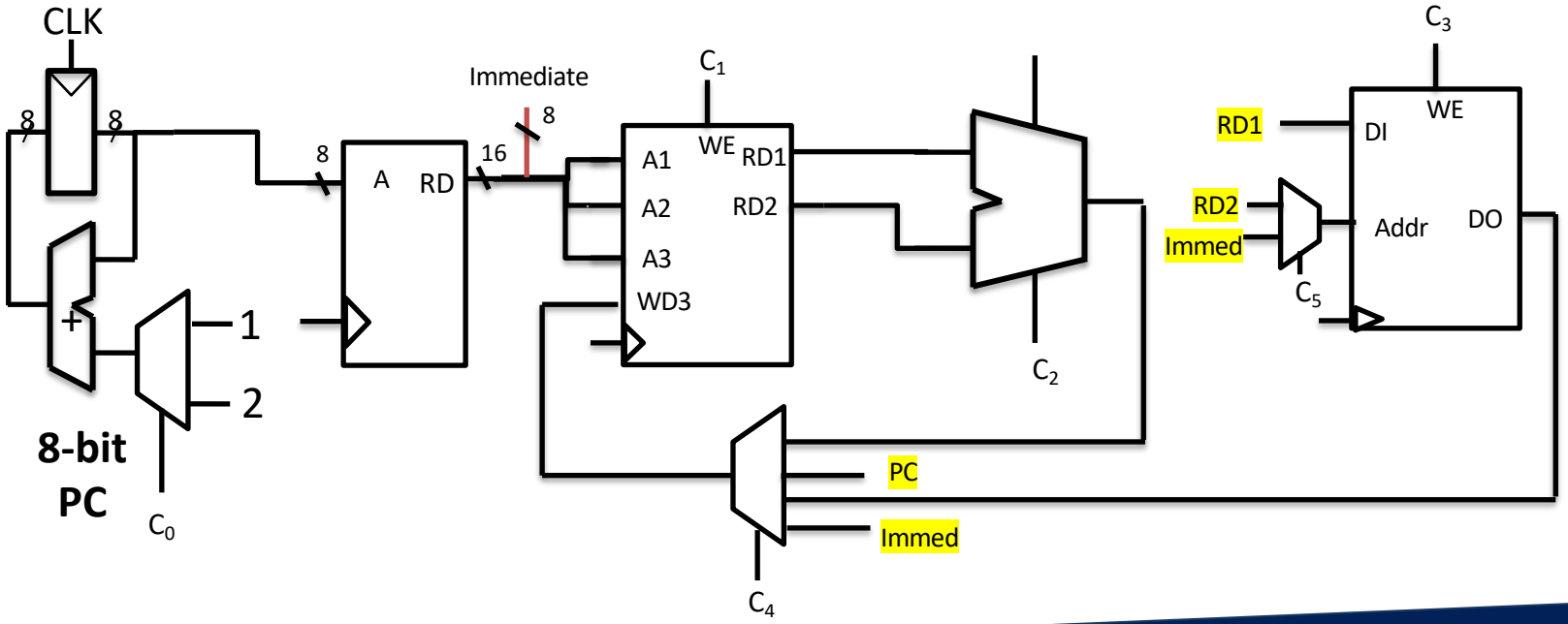
6	0	`rA` = read from memory at `pc + 1`
	1	`rA` += read from memory at `pc + 1`
	2	`rA` &= read from memory at `pc + 1`
	3	`rA` = read from memory at the address stored at `pc + 1`

Again we just need a mux



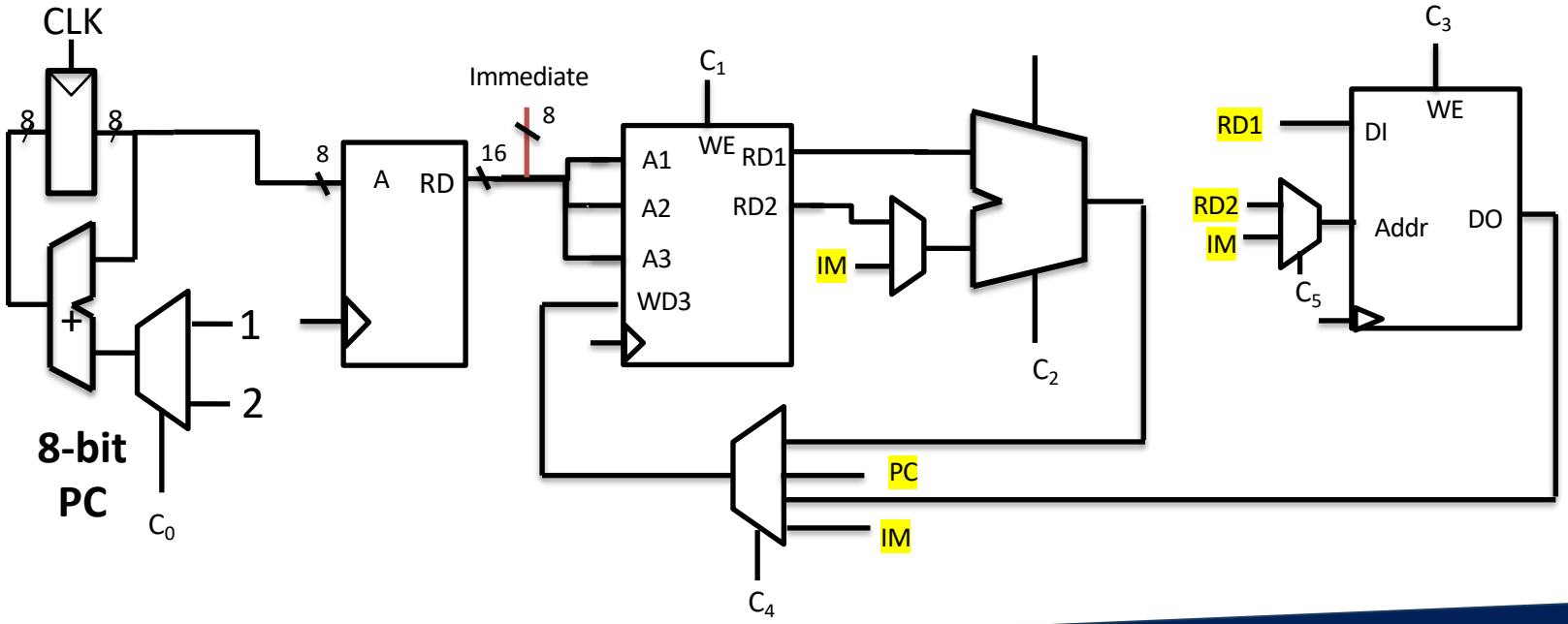
6	0	**rA** = read from memory at **pc + 1**
	1	**rA** += read from memory at **pc + 1**
	2	**rA** &= read from memory at **pc + 1**
	3	**rA** = read from memory at the address stored at **pc + 1**

What about these instructions



6	0	**rA** = read from memory at **pc + 1**
	1	**rA** += read from memory at **pc + 1**
	2	**rA** &= read from memory at **pc + 1**
	3	**rA** = read from memory at the address stored at **pc + 1**

Just need a mux

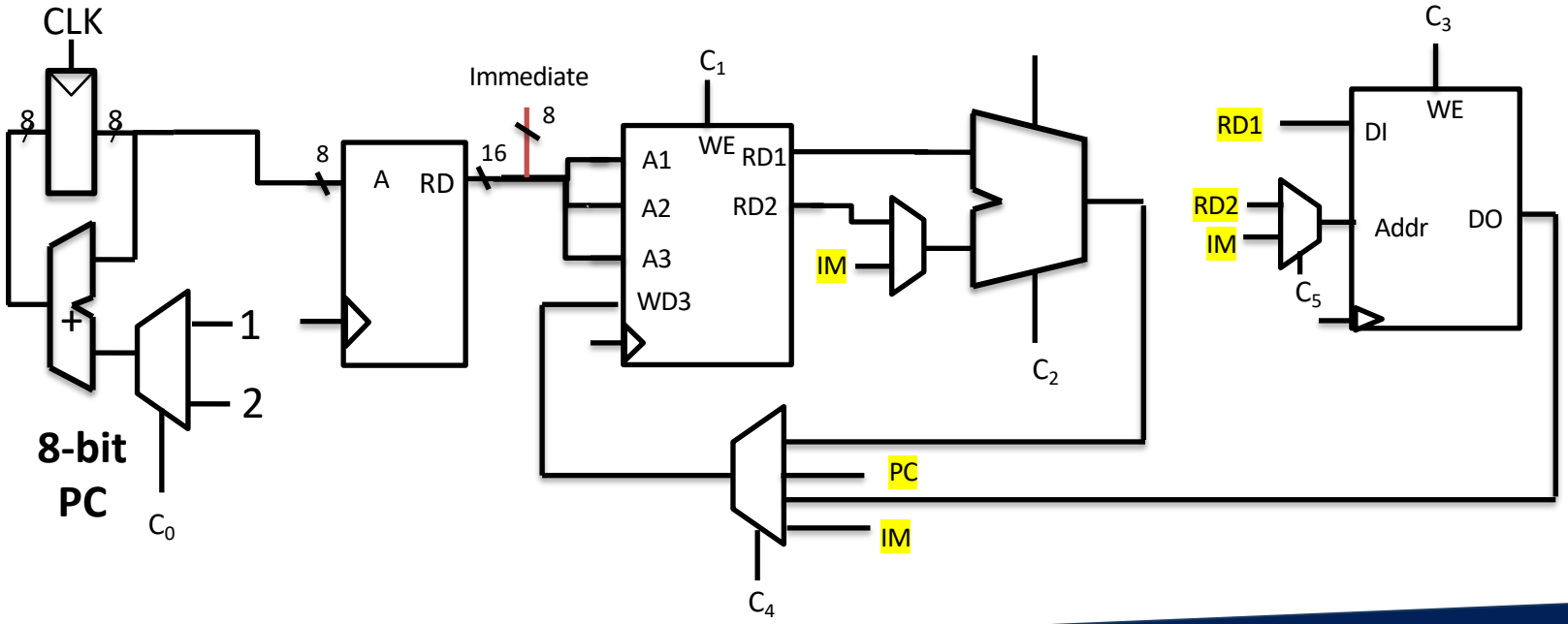


NOW FOR OUR FINAL INSTRUCTIONS

OUR CONDITIONAL JUMP

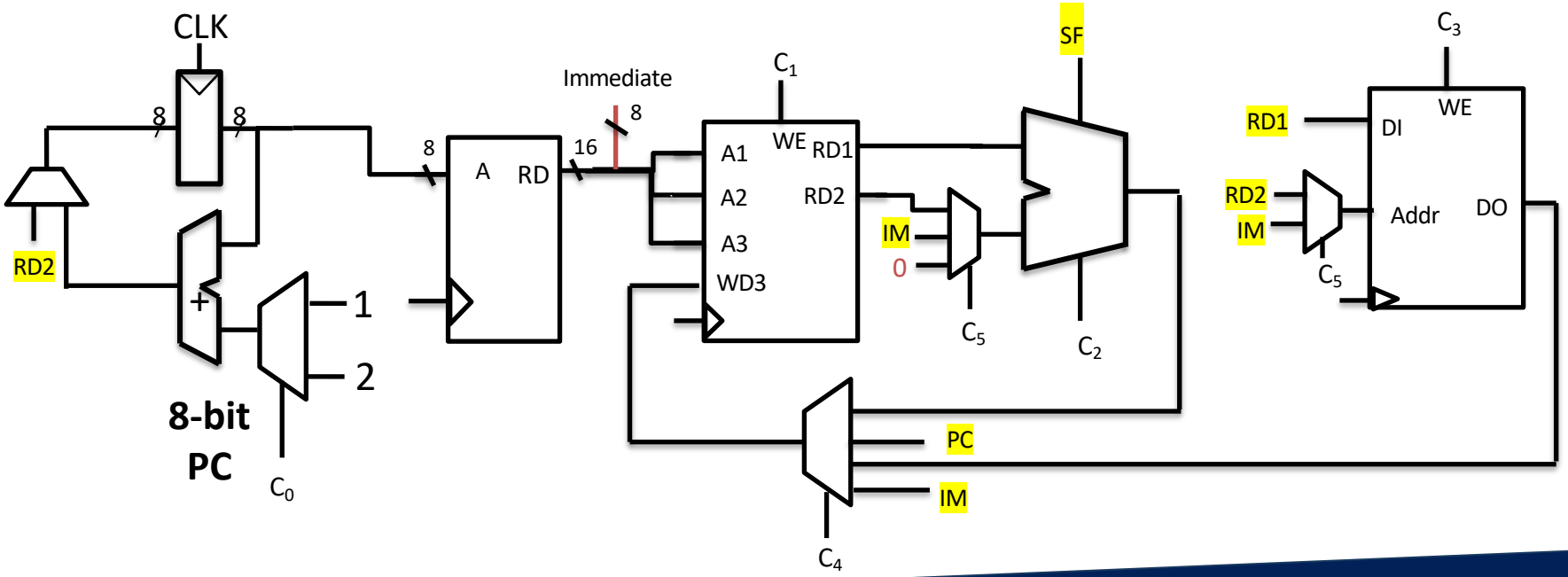
7	Compare **rA** as 8-bit 2's-complement to **0** if **rA <= 0** set **pc = rB** else increment **pc** as normal

How do we implement this one?
Talk to your neighbor



7	Compare **rA** as 8-bit 2's-complement to **0** if **rA <= 0** set **pc = rB** else increment **pc** as normal

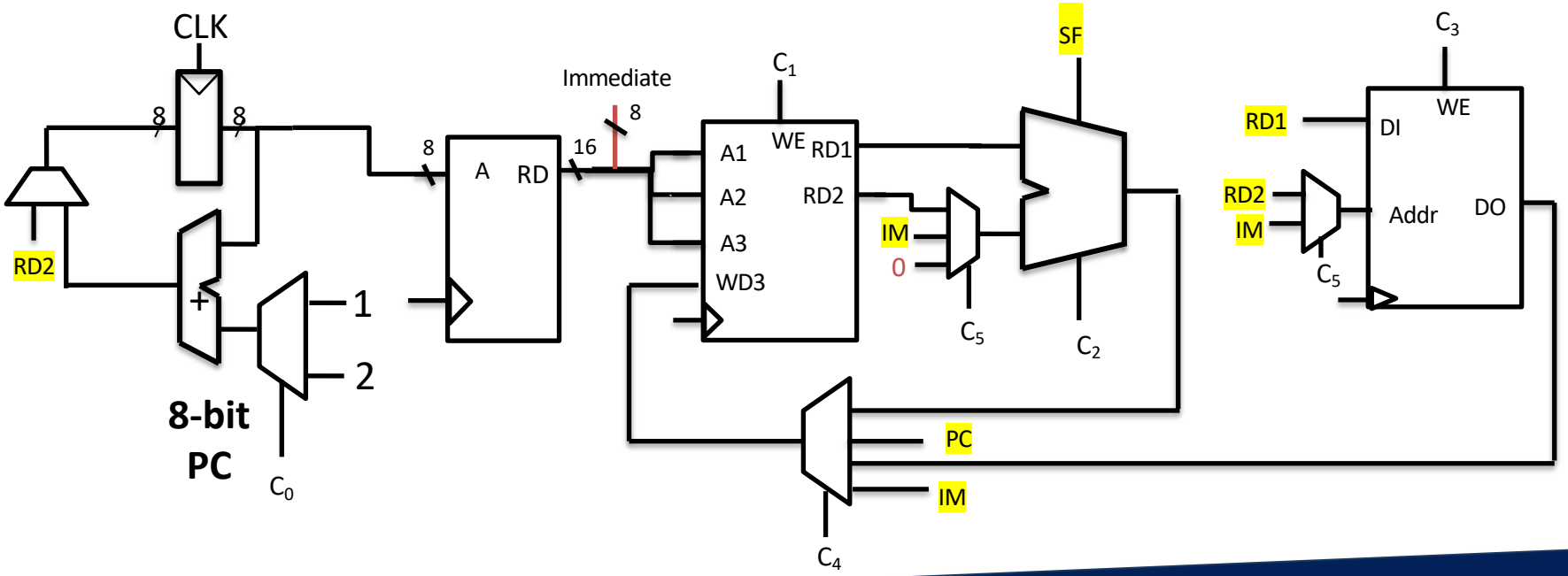
Notice the sign flag output by ALU



7	Compare **rA** as 8-bit 2's-complement to **0** if **rA <= 0** set **pc = rB** else increment **pc** as normal

Let's do a sample instruction

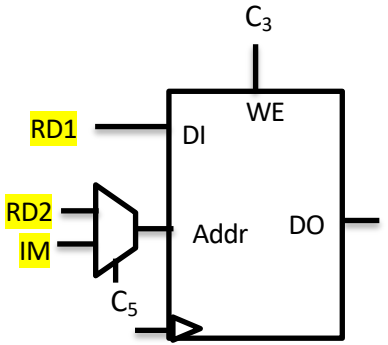
R	icode	RA	RB



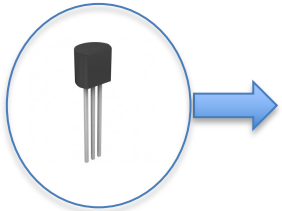
WHAT ABOUT DETAILS OF THE MAIN MEMORY

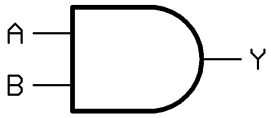
1. How is it implemented?
2. How does it work underhood?

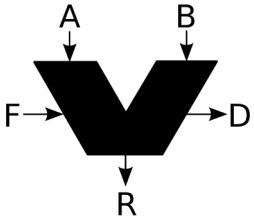
3. Don't worry we'll answer this in CSO 2.
    1. It is actually a complex hierarchy including a controller, caches, and Hardware support for virtual memory like TLBS (translation lookaside buffers)
    2. It doesn't always return a value in a single cycle so the controller might have to insert nops in the pipeline etc.

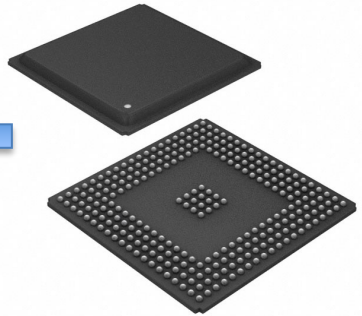


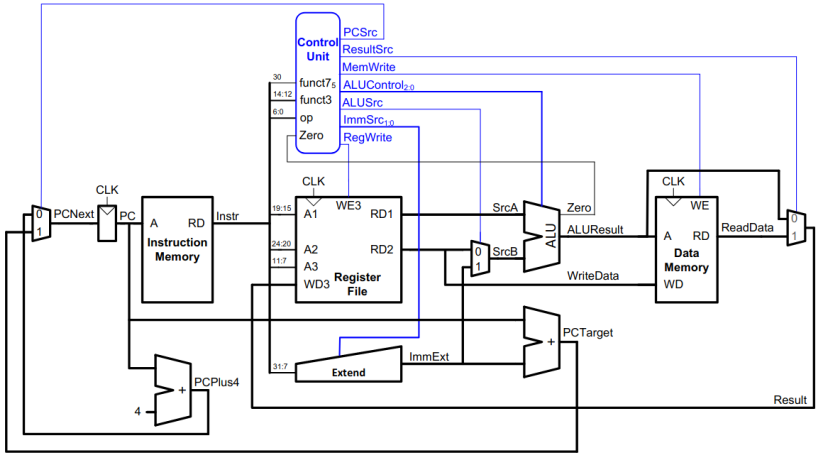
THE MAP (THE MACHINE)





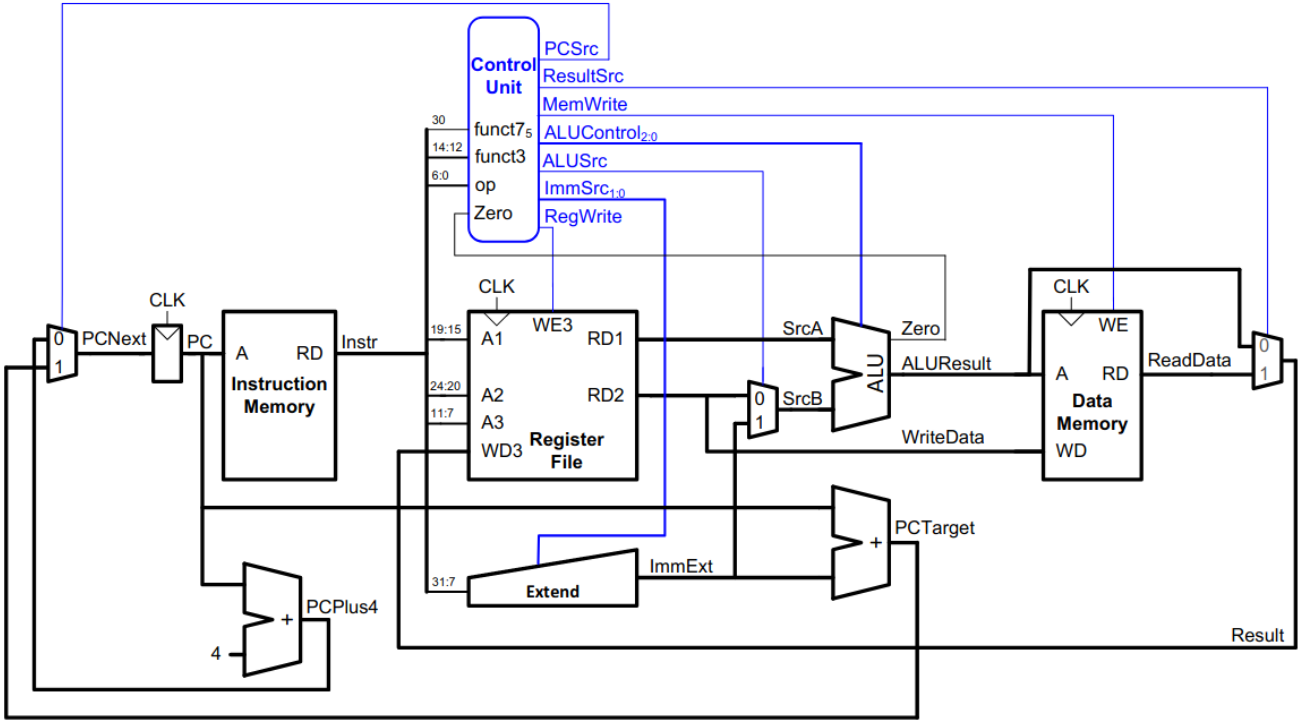




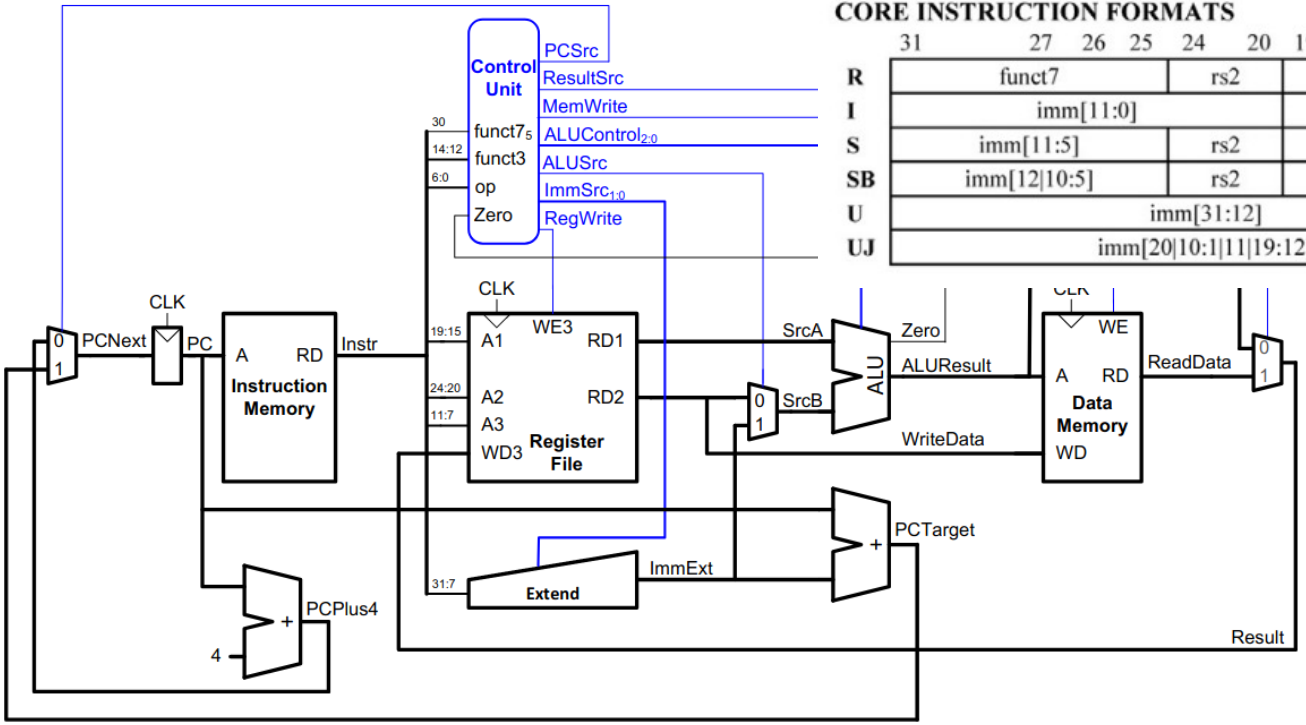


https://github.com/MKrekker/SINGLE-CYCLE-RISC-V

RISC-V MACHINE



RISC-V MACHINE



CORE INSTRUCTION FORMATS

	31–27	26–25	24–20	19–15	14–12	11–7	6–0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

THE ISA ALSO INCLUDES FLOATING LAYOUT SUPPORTED AND REGISTER AND THEIR DESCRIPTION

https://www.elsevier.com/__data/assets/pdf_file/0011/297533/RISC-V-Reference-Data.pdf

Let's look at the section that describes floating point And instruction encodings. Focus many on the second page

THE MAP (THE CODE)

```c
#include <stdio.h>
int main() {
    printf("Hello, World!");
    return 0;
}
```

```
0000000000001149 <main>:
    1149: f3 0f 1e fa          endbr64
    114d: 55                   push   %rbp
    114e: 48 89 e5             mov    %rsp,%rbp
    1151: 48 8d 05 ac 0e 00 00
lea    0xeac(%rip),%rax        # 2004
<_IO_stdin_used+0x4>
    1158: 48 89 c7             mov    %rax,%rdi
    115b: e8 f0 fe ff ff       call   1050 <puts@plt>
    1160: b8 00 00 00 00       mov    $0x0,%eax
    1165: 5d                   pop    %rbp
    1166: c3                   ret
```

We will not cover this conversion in detail. CS 4620 - Compilers is a class dedicated to building and understanding the program designed to do this conversion.

We'll focus on understanding the output of the program and how this output gets executed on a machine

NEXT WE'LL TALK ABOUT WRITING PROGRAMS THAT SIMULATE OUR ARCHITECTURE