CS0 2130

TOY Proccessor

Daniel G. Graham PhD

UNIVERSITY of VIRGINIA | ENGINEERING

Contents

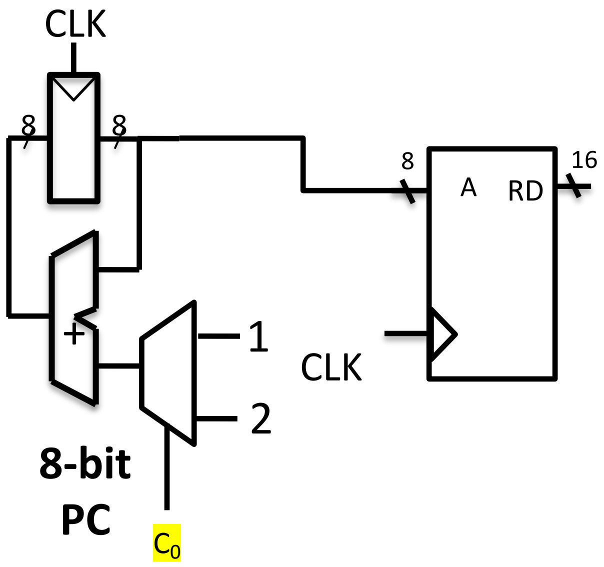
1. A full overview of Toy ISA
2. Build a machine (Toy Processor) that can execute our Toy ISA
3. Discuss the fetch, decode, execute, memory, and writeback stages
4. Discuss the steps need to synthesize or toy processor

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

1 BYTE AND 2 BYTE INSTRUCTIONS



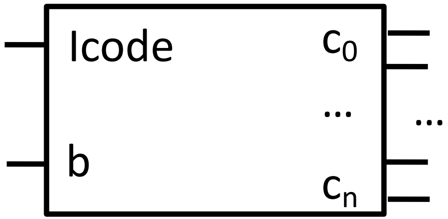
We'll add a mux that will select passing one to adder or two.

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

UNIVERSITY of VIRGINIA | ENGINEERING

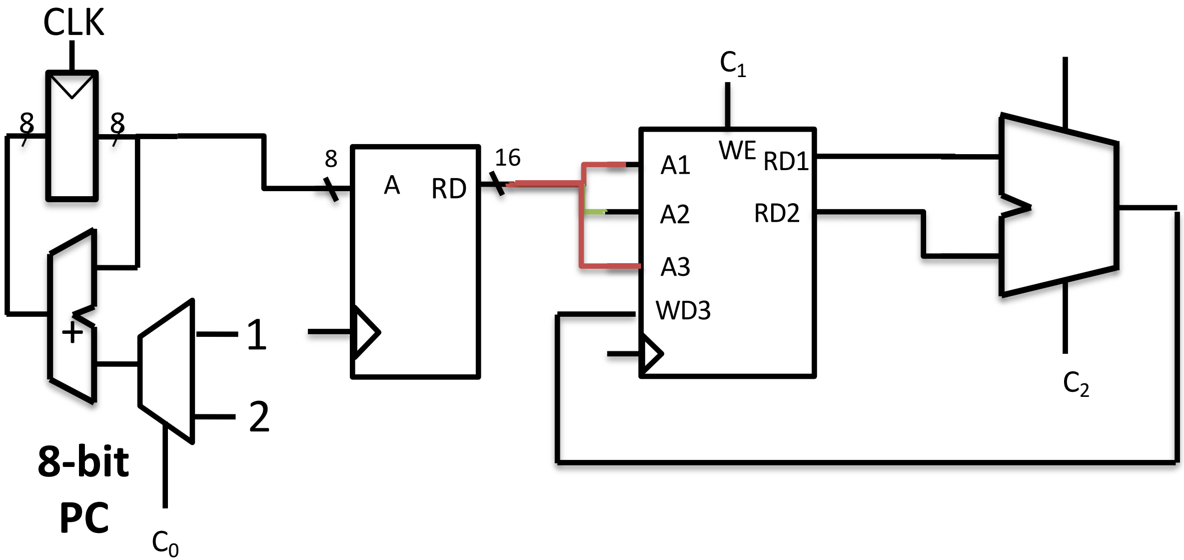
HARDWIRED CONTROL UNIT

icode	b	C_0	….	C_n
6	x	1		
….				



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

R	icode	RA	RB

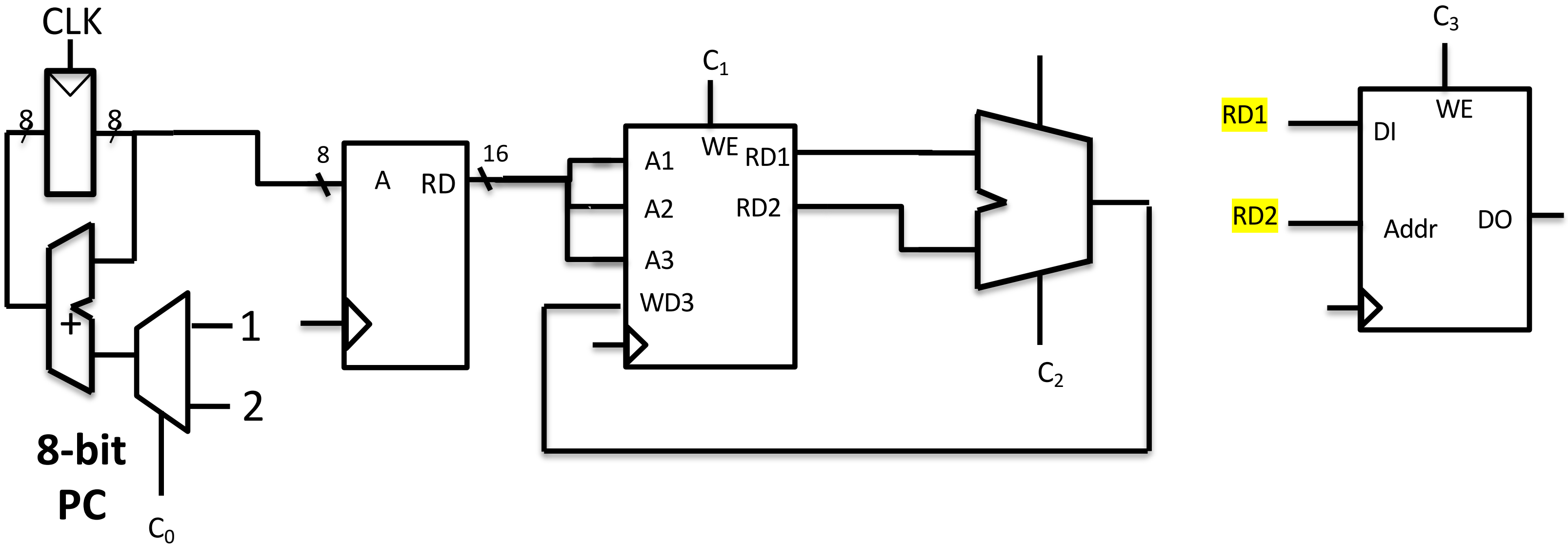


University of Virginia | ENGINEERING

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

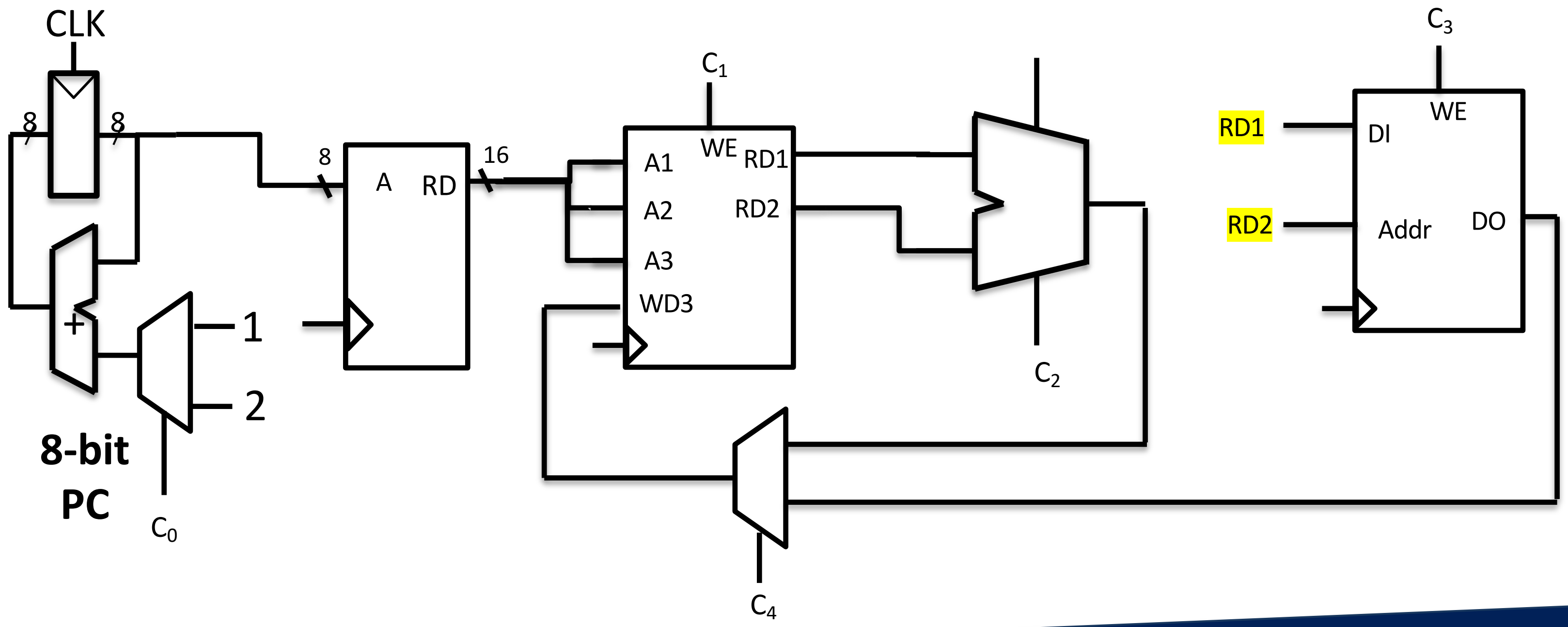
3	`rA` = read from memory at address `rB`
4	write `rA` to memory at address `rB`

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



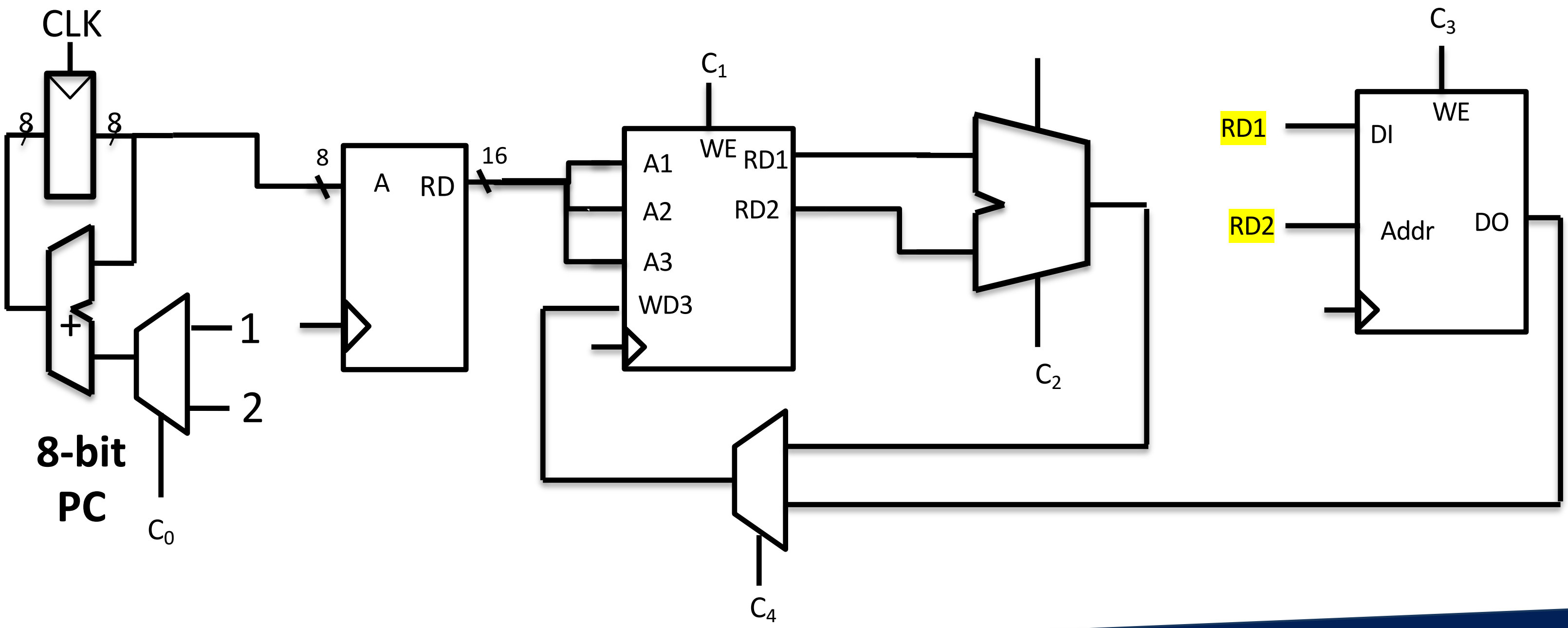
3	$\boxed{\texttt{rA} = \text{read from memory at address } \texttt{rB}}$
4	write **rA** to memory at address **rB**

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`

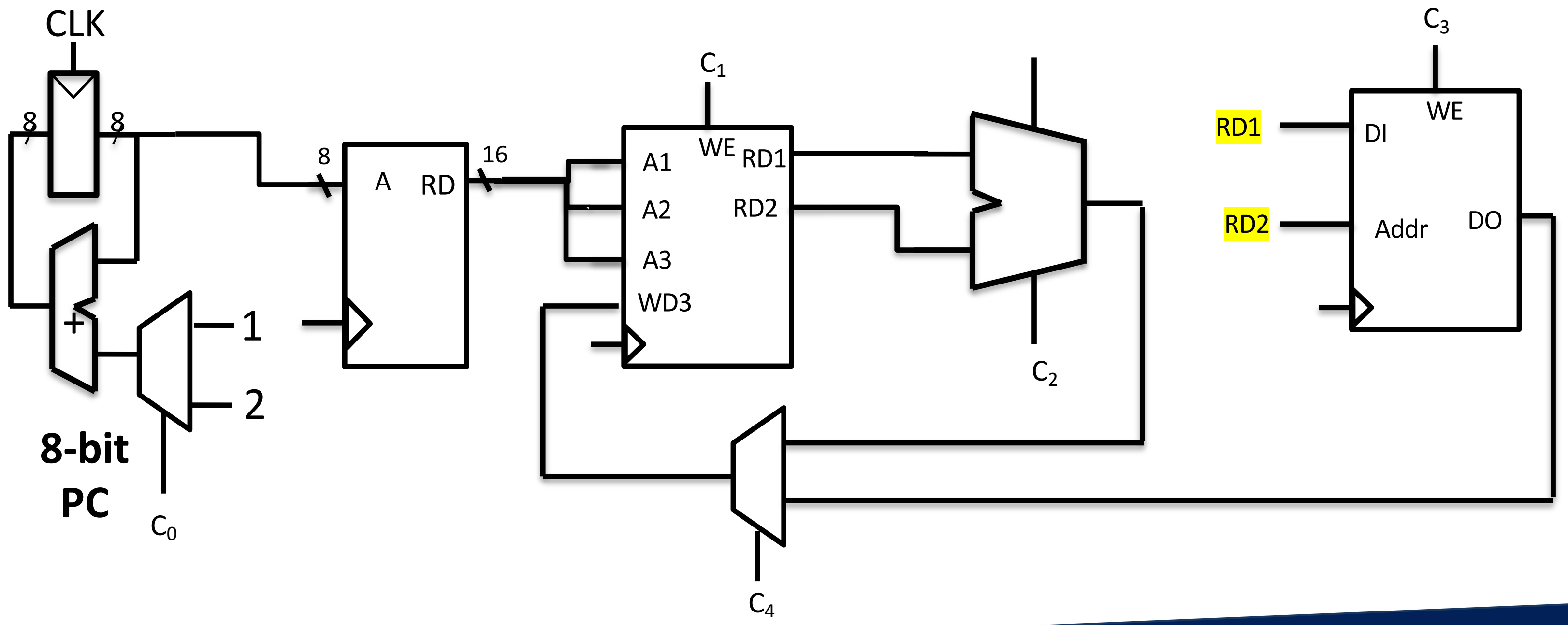
Let's execute some sample instructions



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

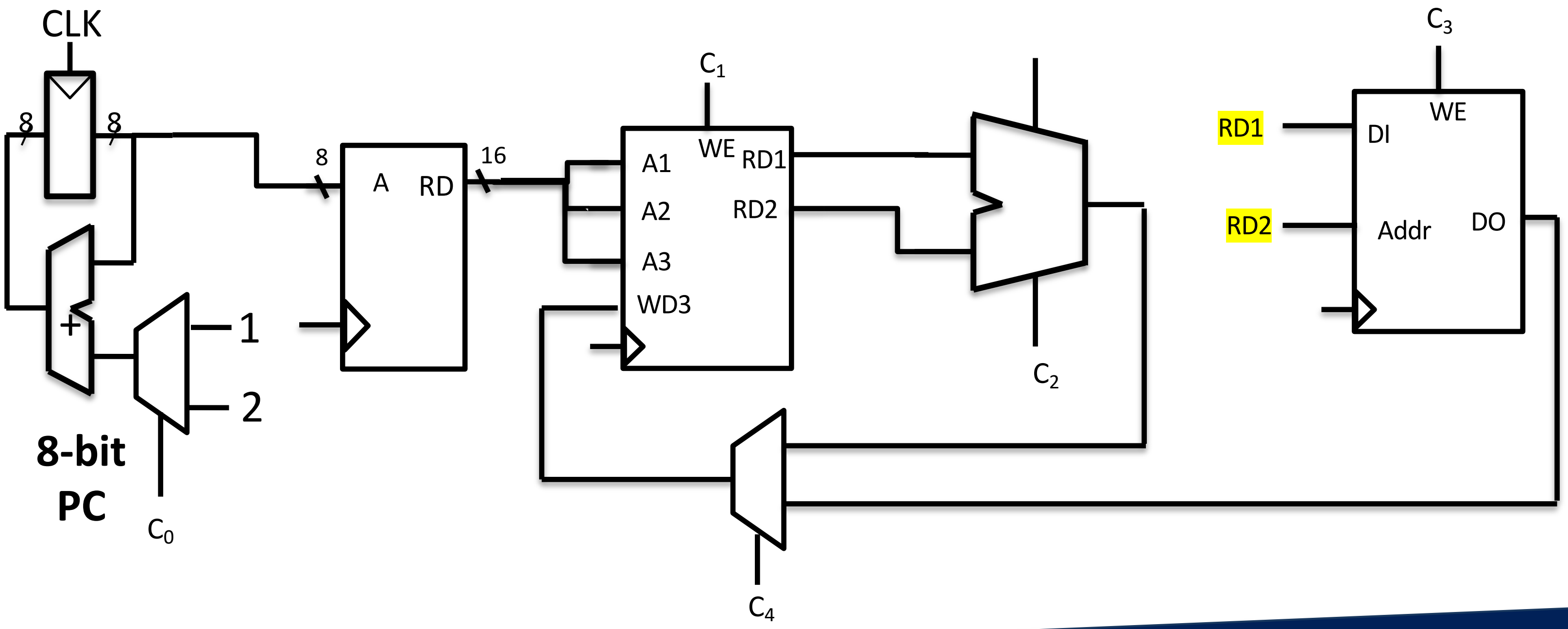
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`

Draw out the flow here



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

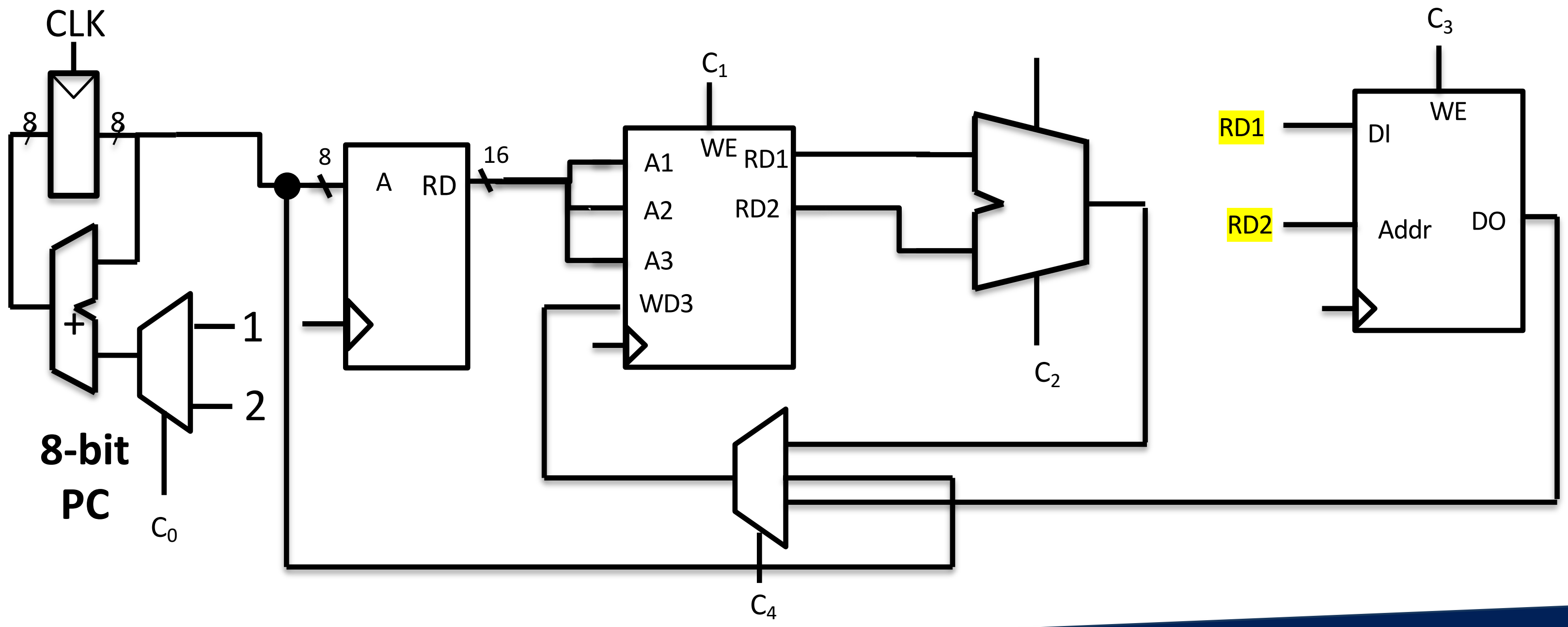
How can we update RA with the PC value?



UNIVERSITY of VIRGINIA | ENGINEERING

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

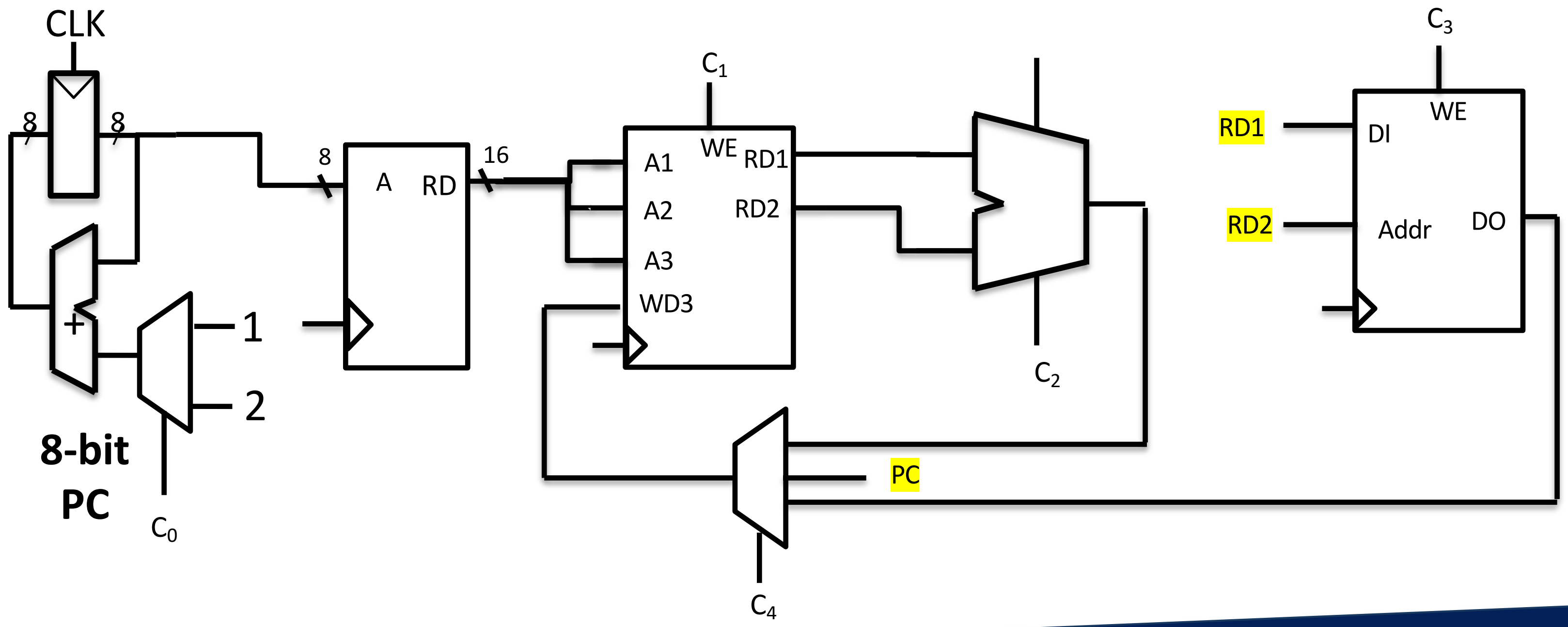
How can we update RA with the PC value?



UNIVERSITY of VIRGINIA | ENGINEERING

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

Changed it to just be the label

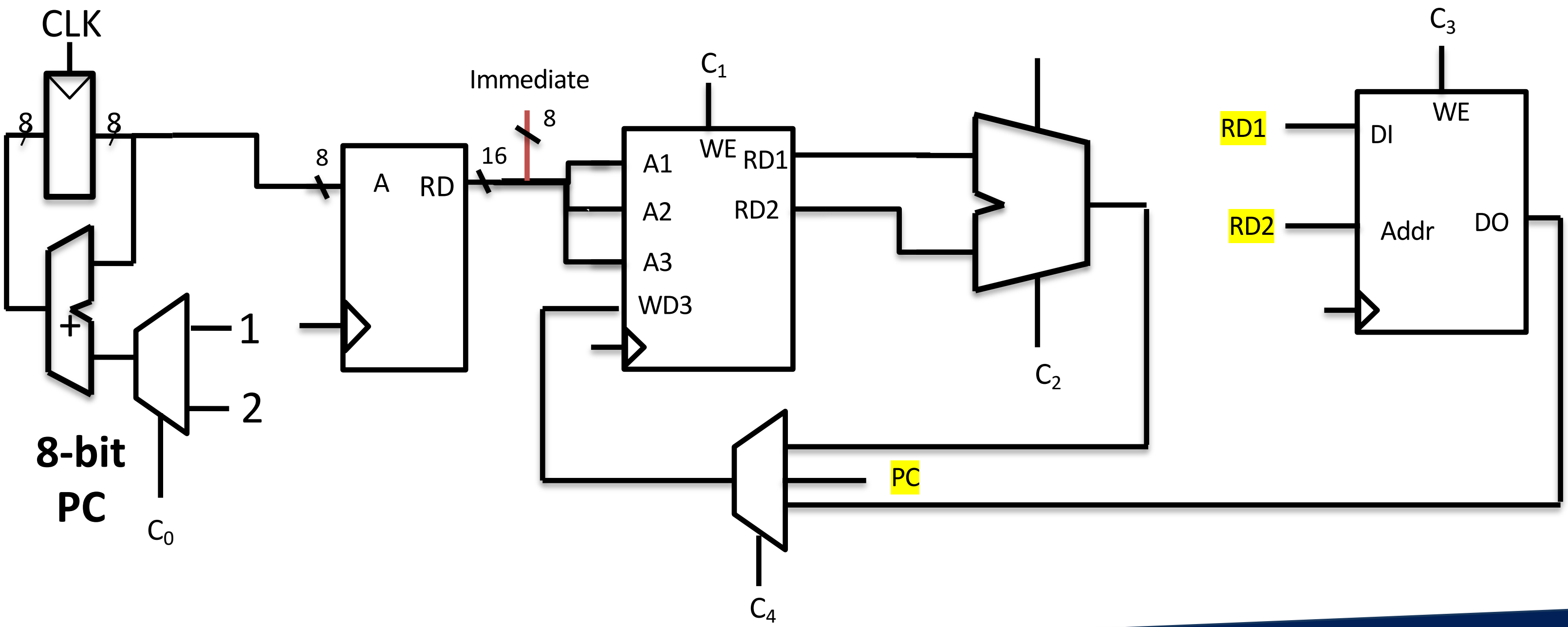


University of Virginia ENGINEERING

6	0	`rA` = read from memory at $\boxed{\texttt{pc + 1}}$
	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

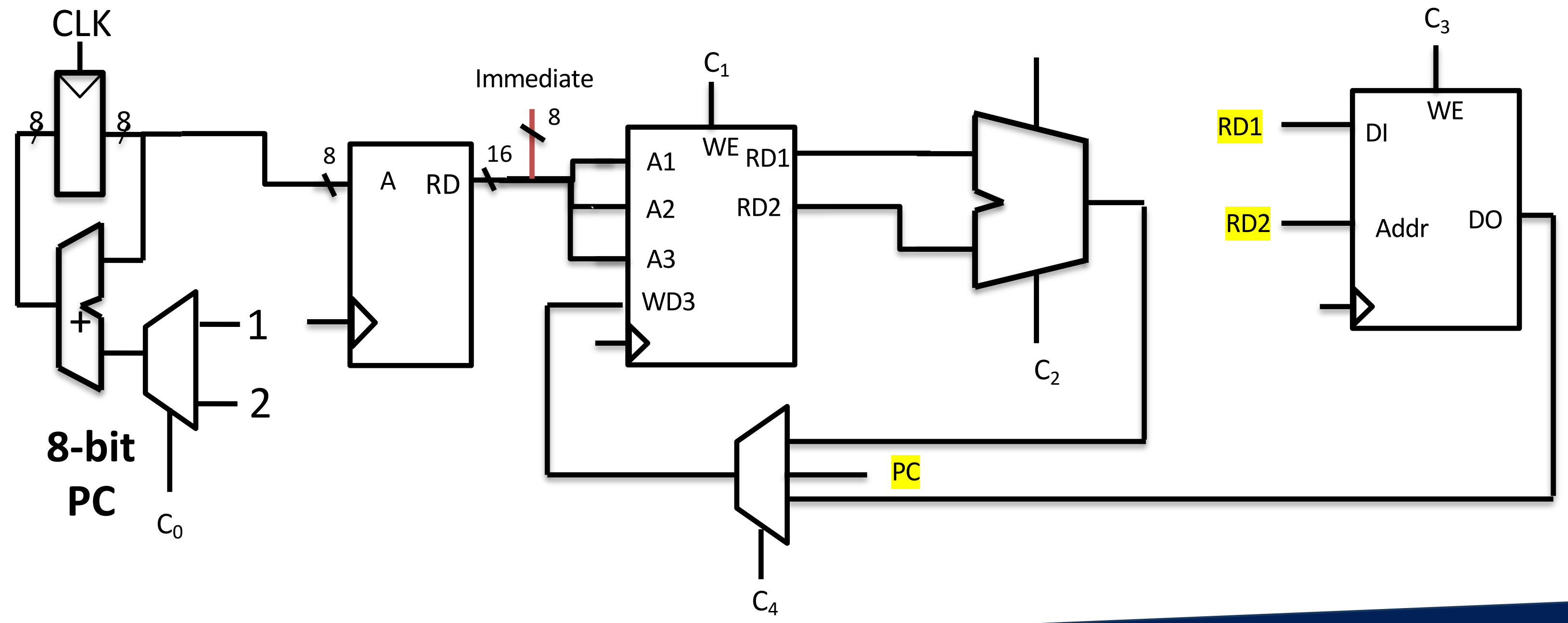
$\boxed{\text{immediate}}$



UNIVERSITY of VIRGINIA | ENGINEERING

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`

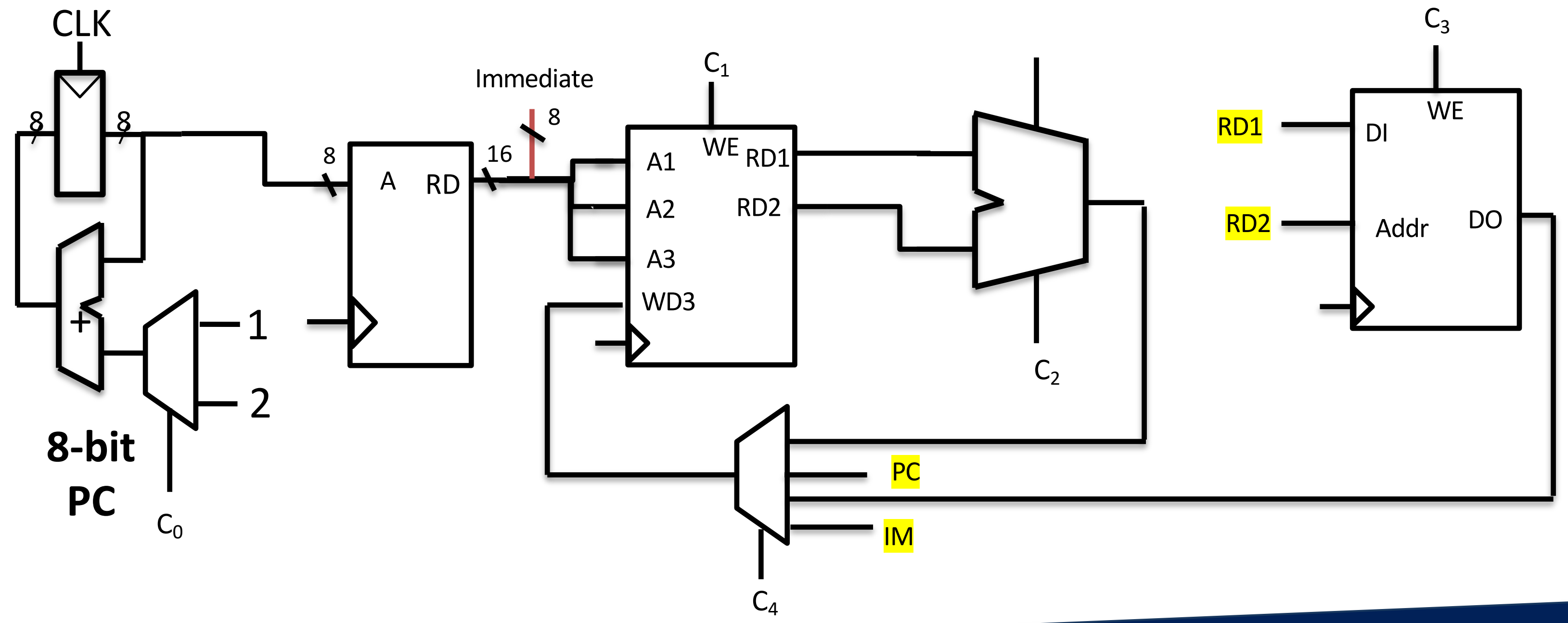
How could we implement this instruction



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

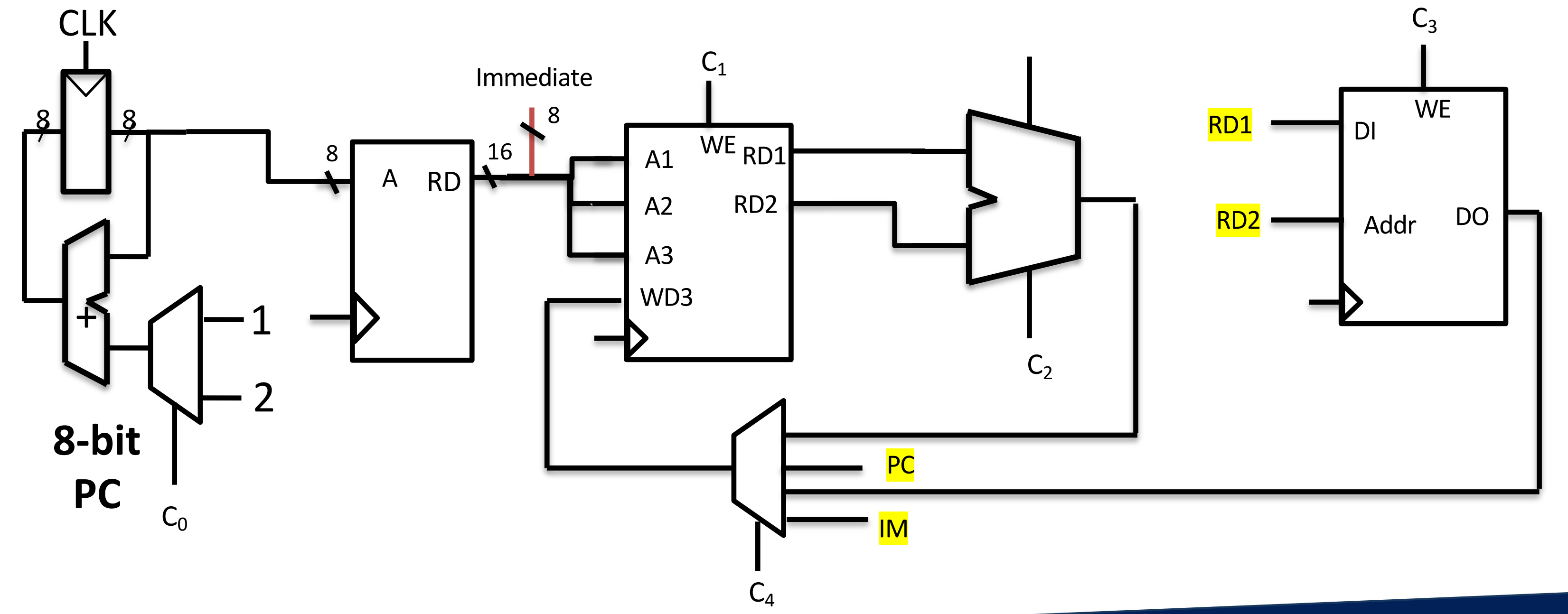
R	icode	RA	RB



University of Virginia ENGINEERING

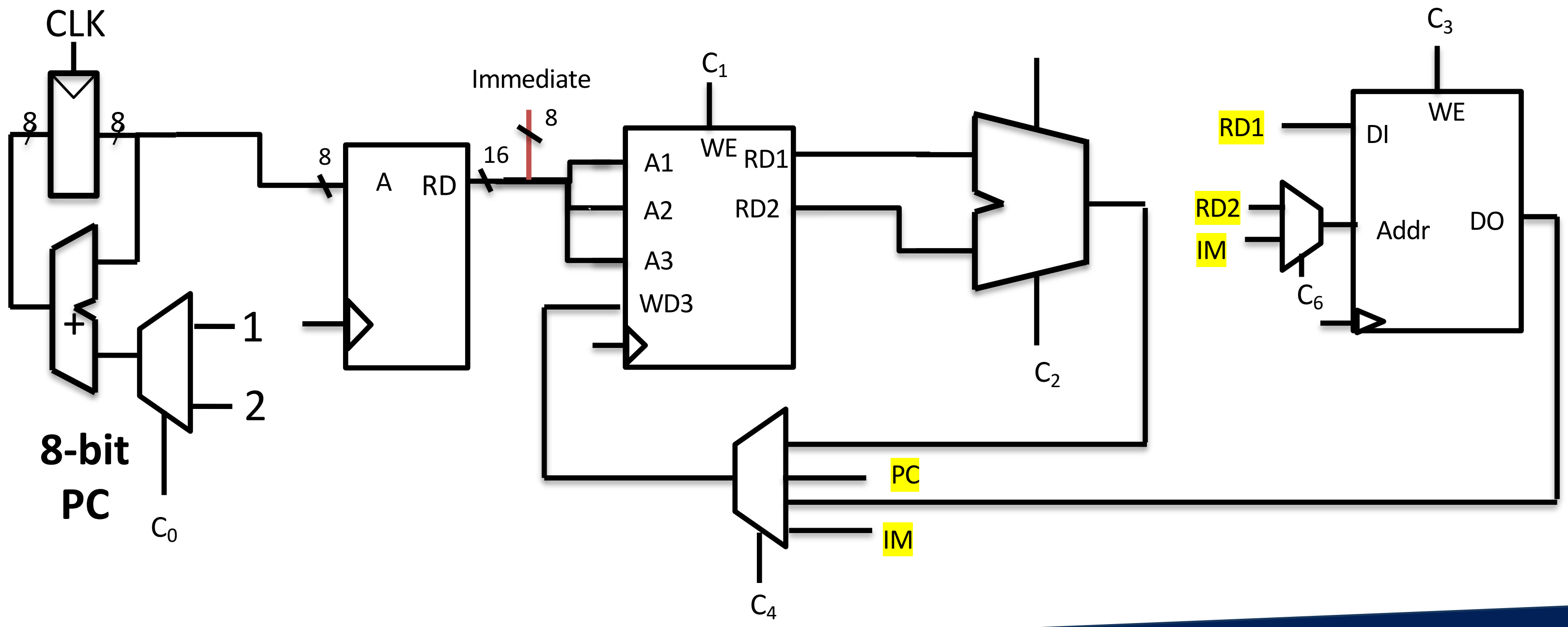
6	0	**rA** = read from memory at **pc + 1**
	1	**rA** += read from memory at **pc + 1**
	2	**rA** &= read from memory at **pc + 1**
	3	**rA** = read from memory at the address stored at **pc + 1**

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**

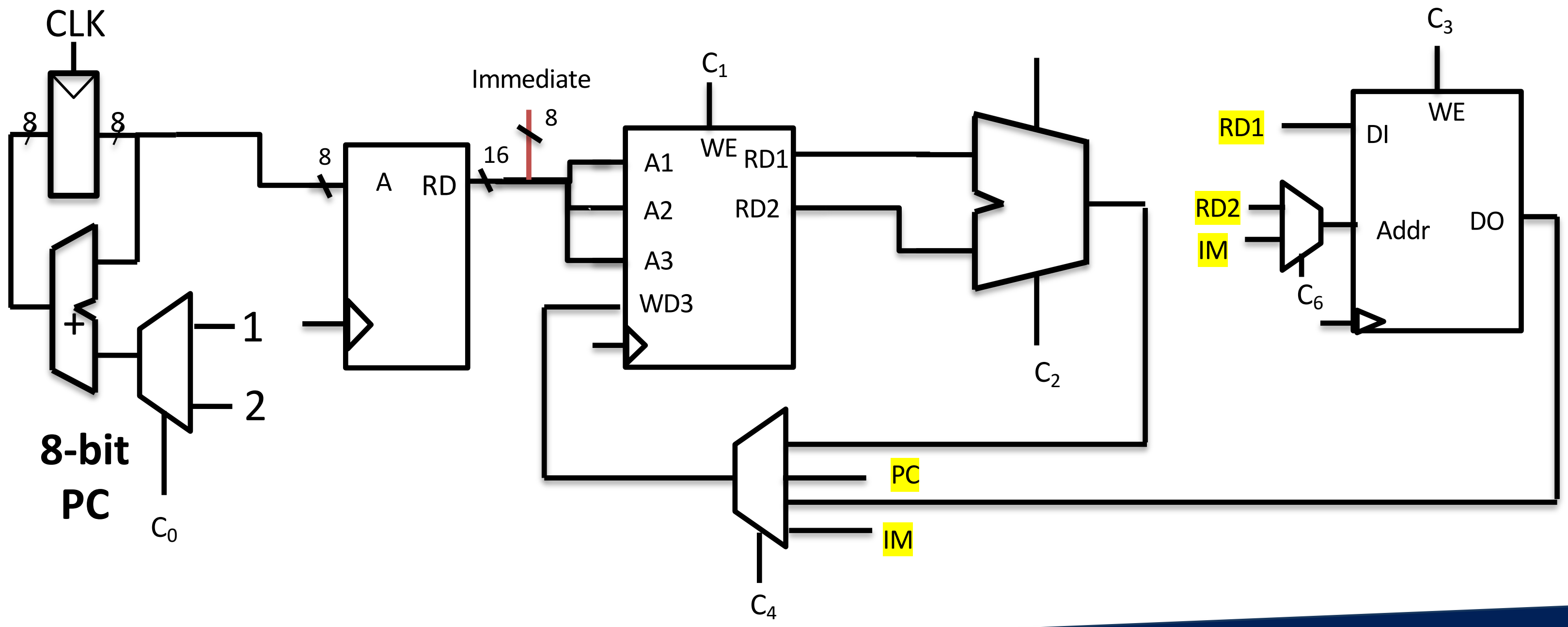
Again we just need a mux



UNIVERSITY of VIRGINIA ENGINEERING

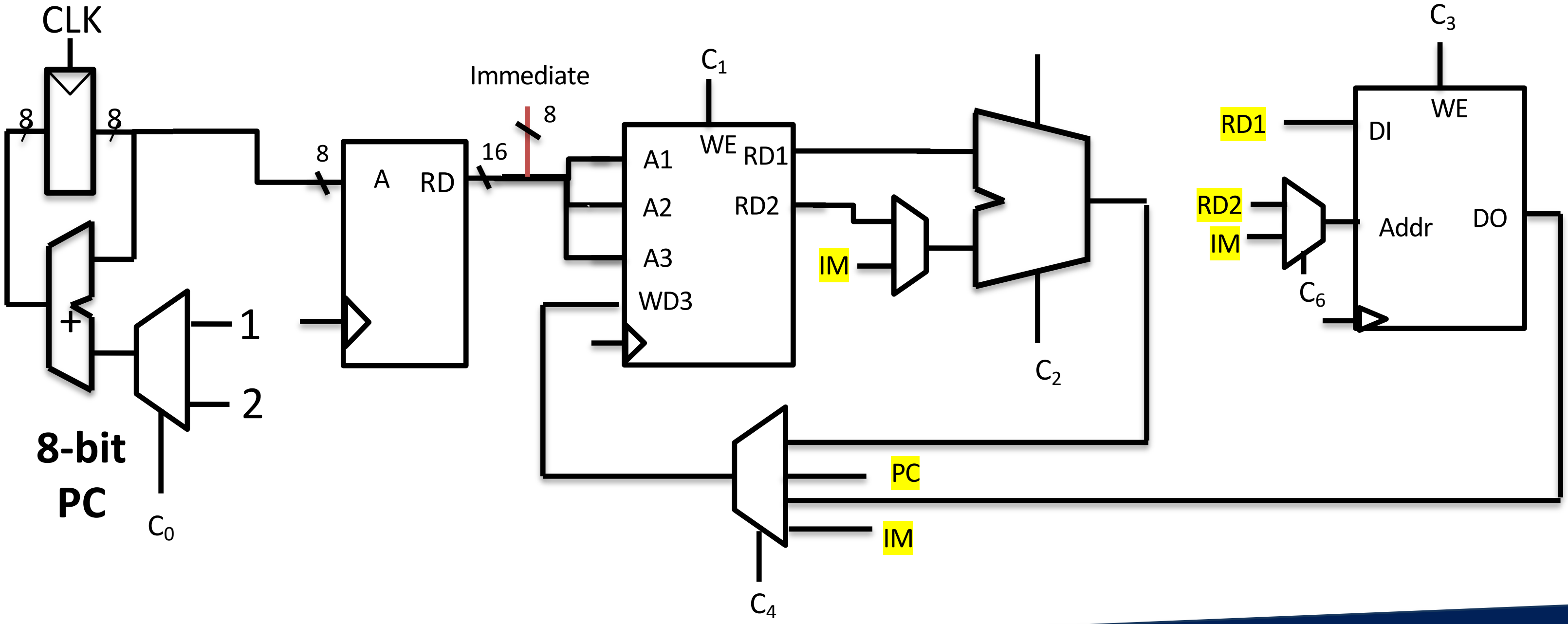
6	0	**rA** = read from memory at **pc + 1**
	1	**rA** += read from memory at **pc + 1**
	2	**rA** &= read from memory at **pc + 1**
	3	**rA** = read from memory at the address stored at **pc + 1**

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**

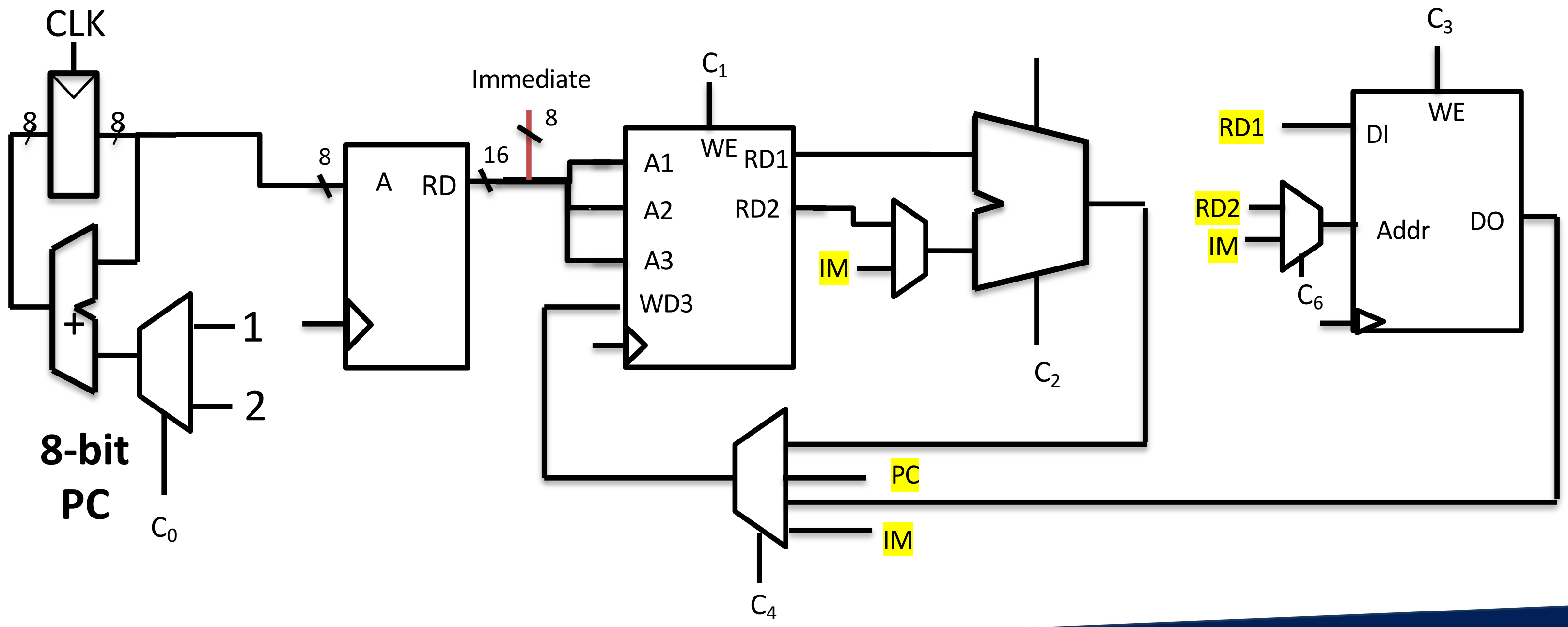
Just need a mux



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

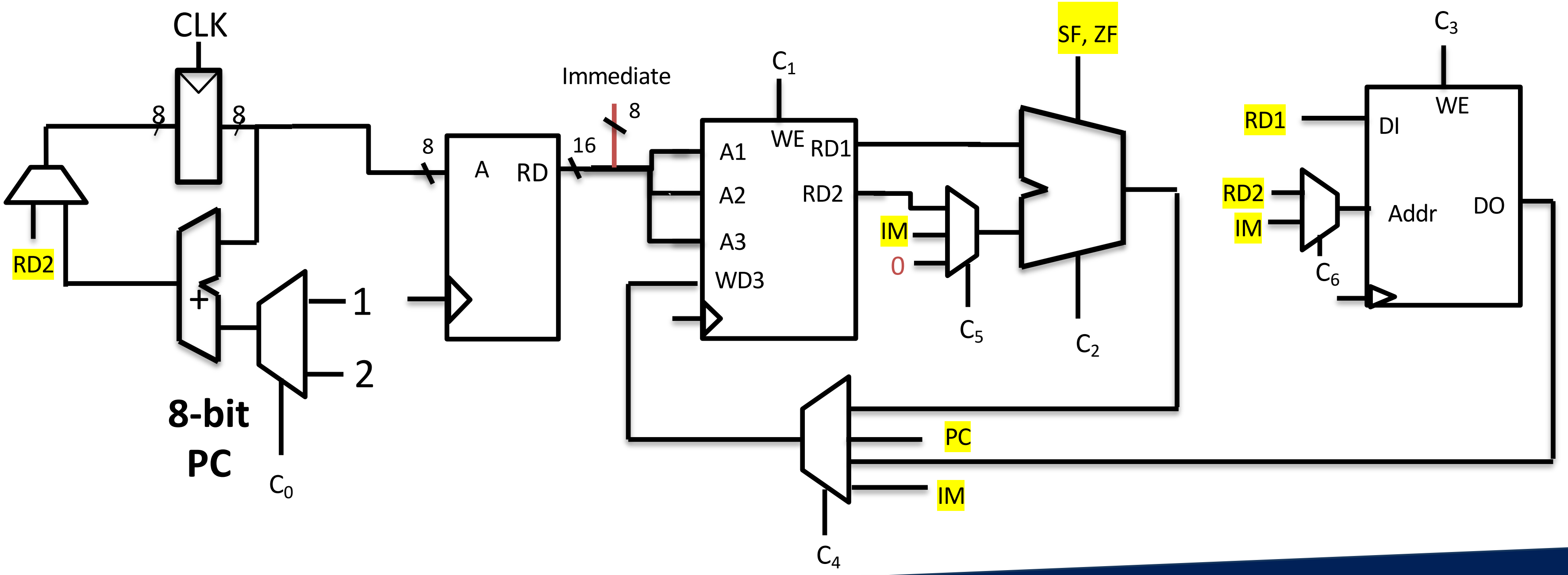
How do we implement this one?

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

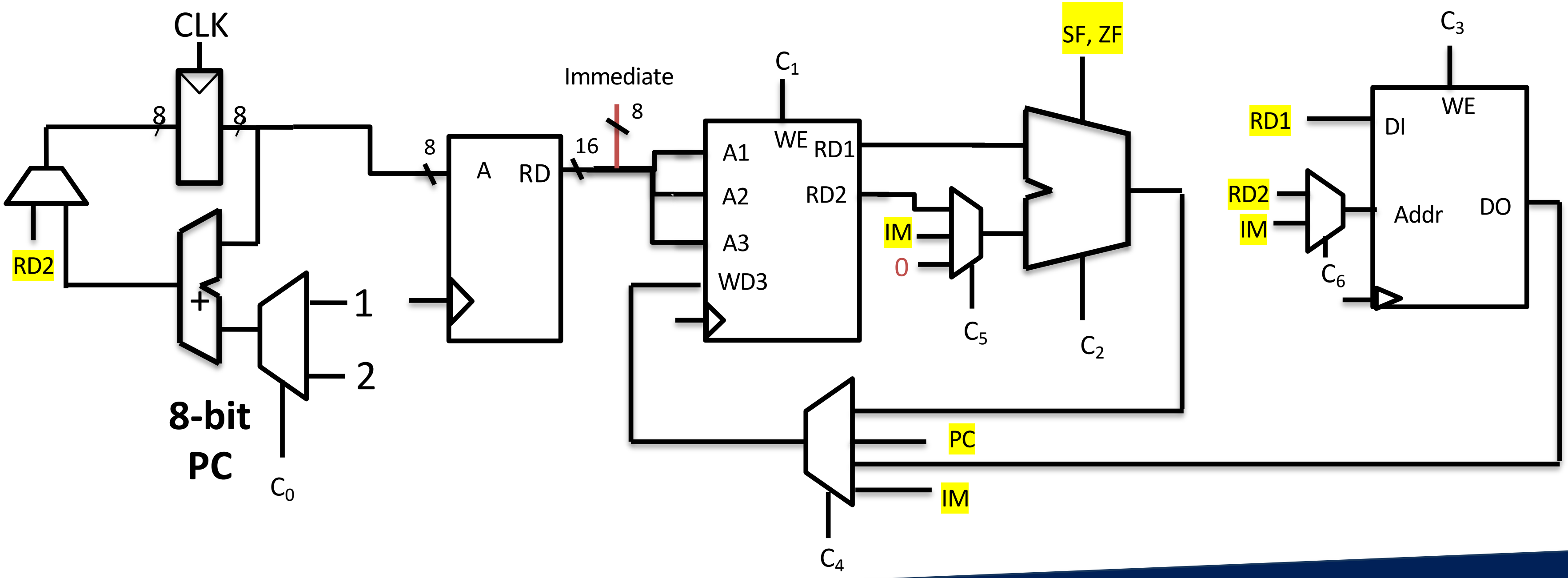
Notice the sign flag output by ALU



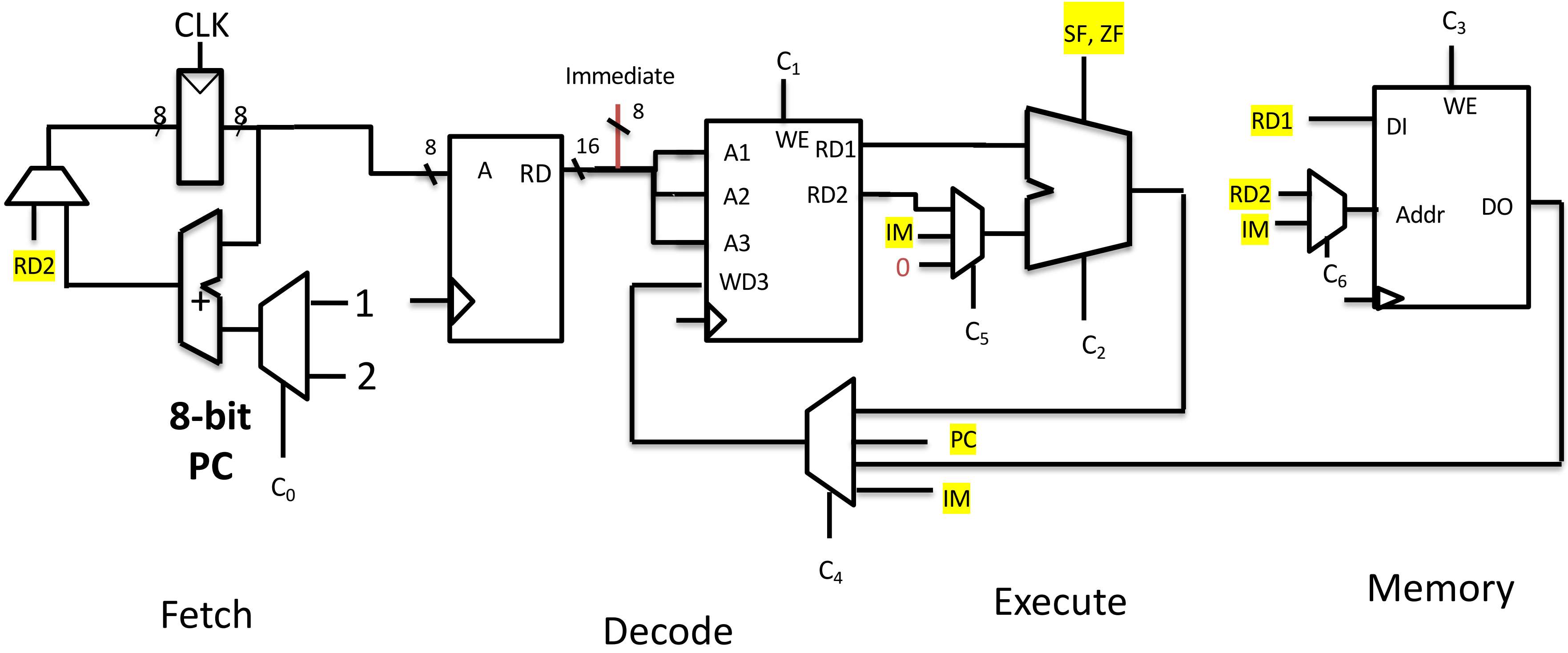
7	Compare **`rA`** as 8-bit 2's-complement to **`0`** if **`rA <= 0`** set **`pc = rB`** else increment **`pc`** as normal

Let's do a sample instruction

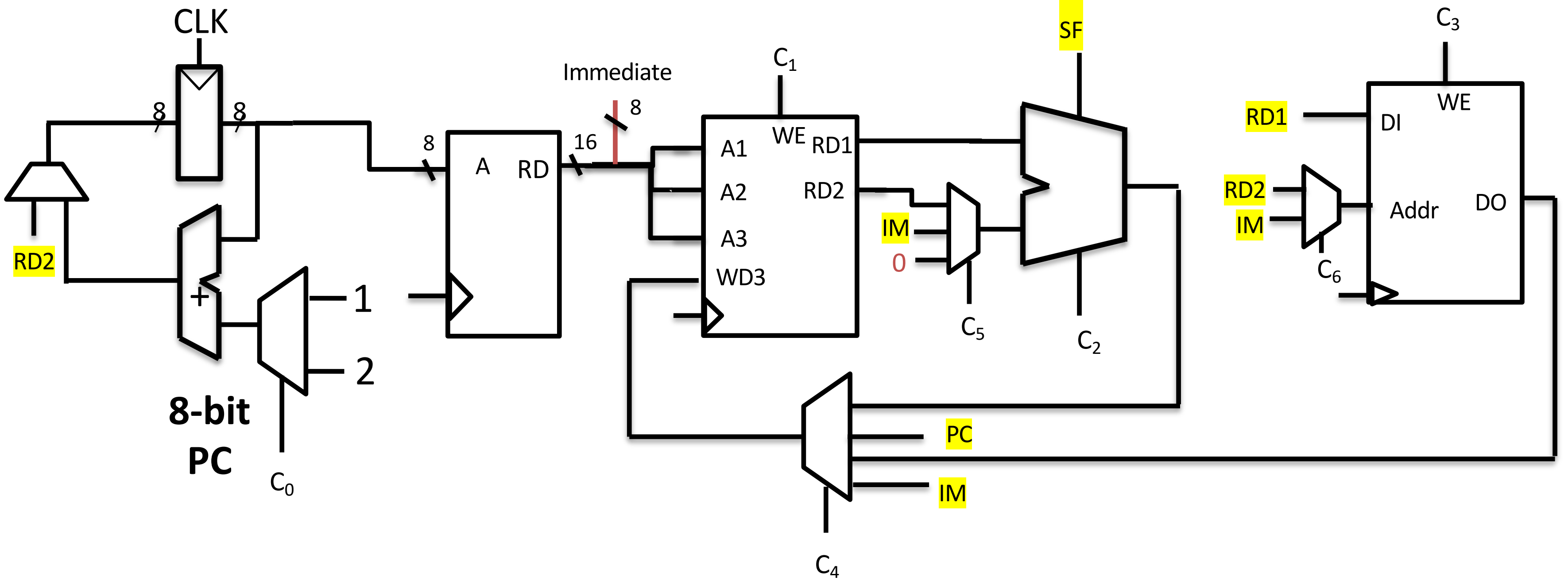
R	icode	RA	RB



OUR SINGLE CYCLE TOY PROCESSOR



OUR SINGLE CYCLE TOY PROCESSOR

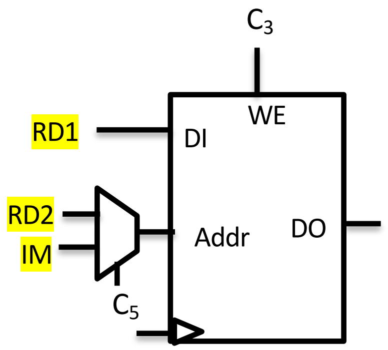


Write back stages

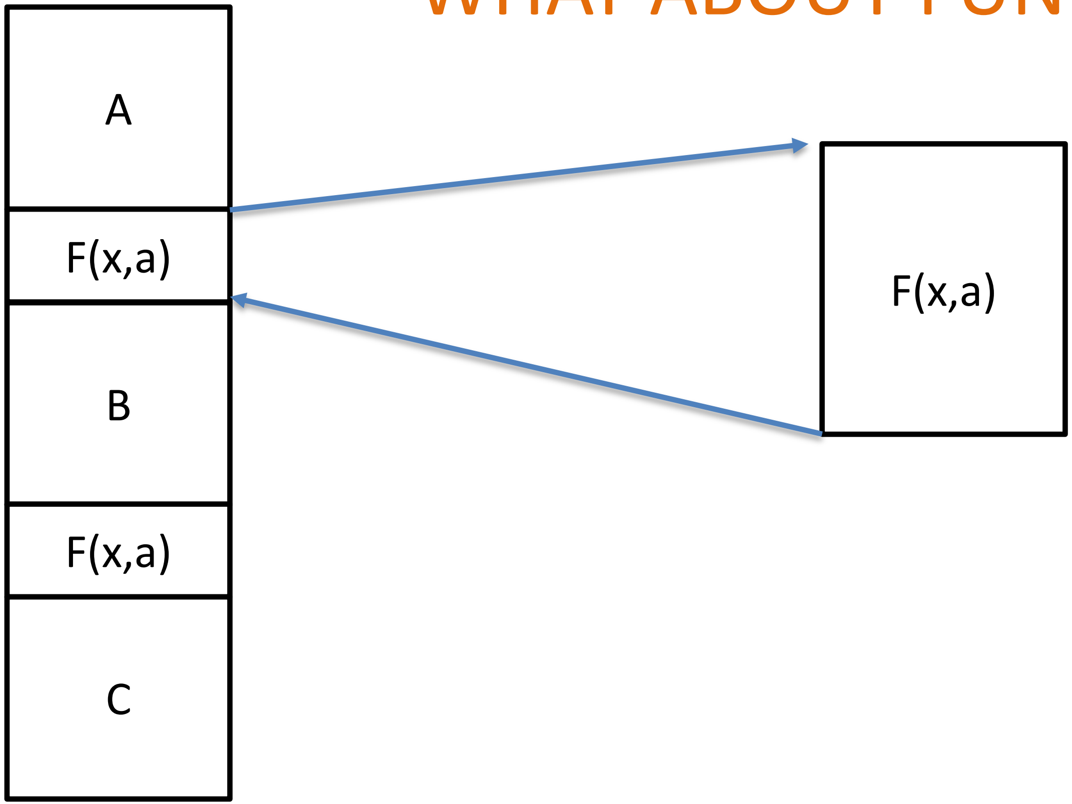
WHAT ABOUT DETAILS OF THE MAIN MEMORY

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.

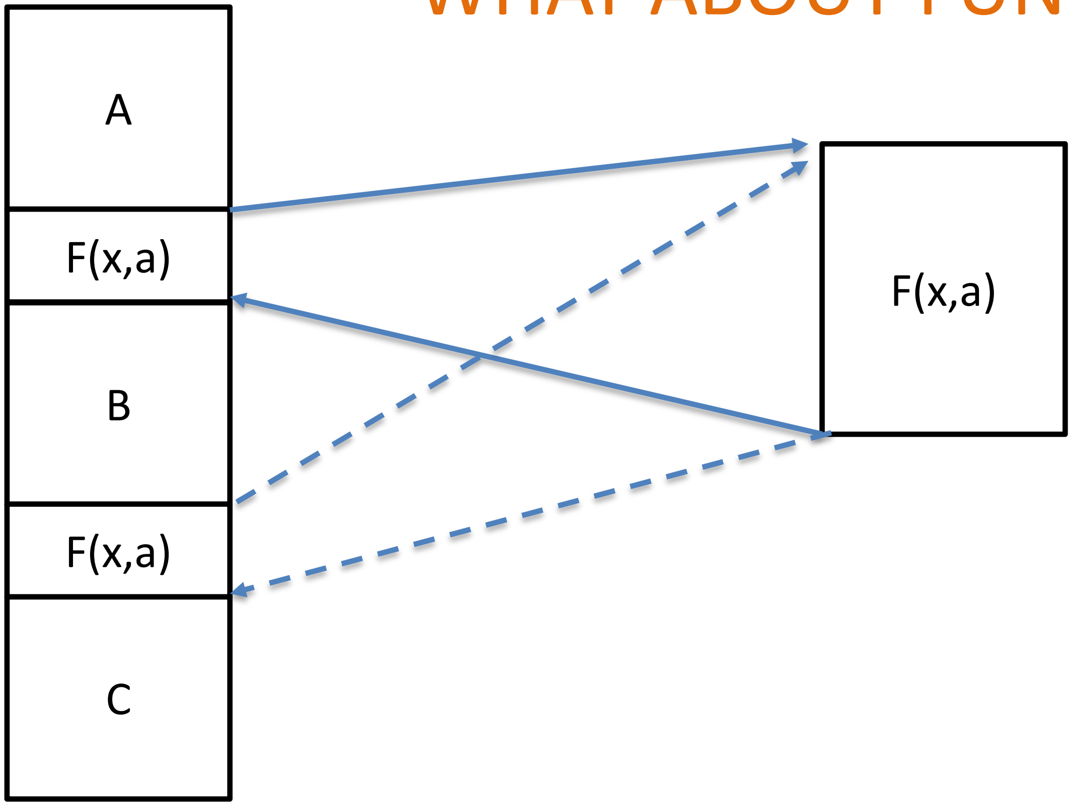


WHAT ABOUT FUNCTIONS

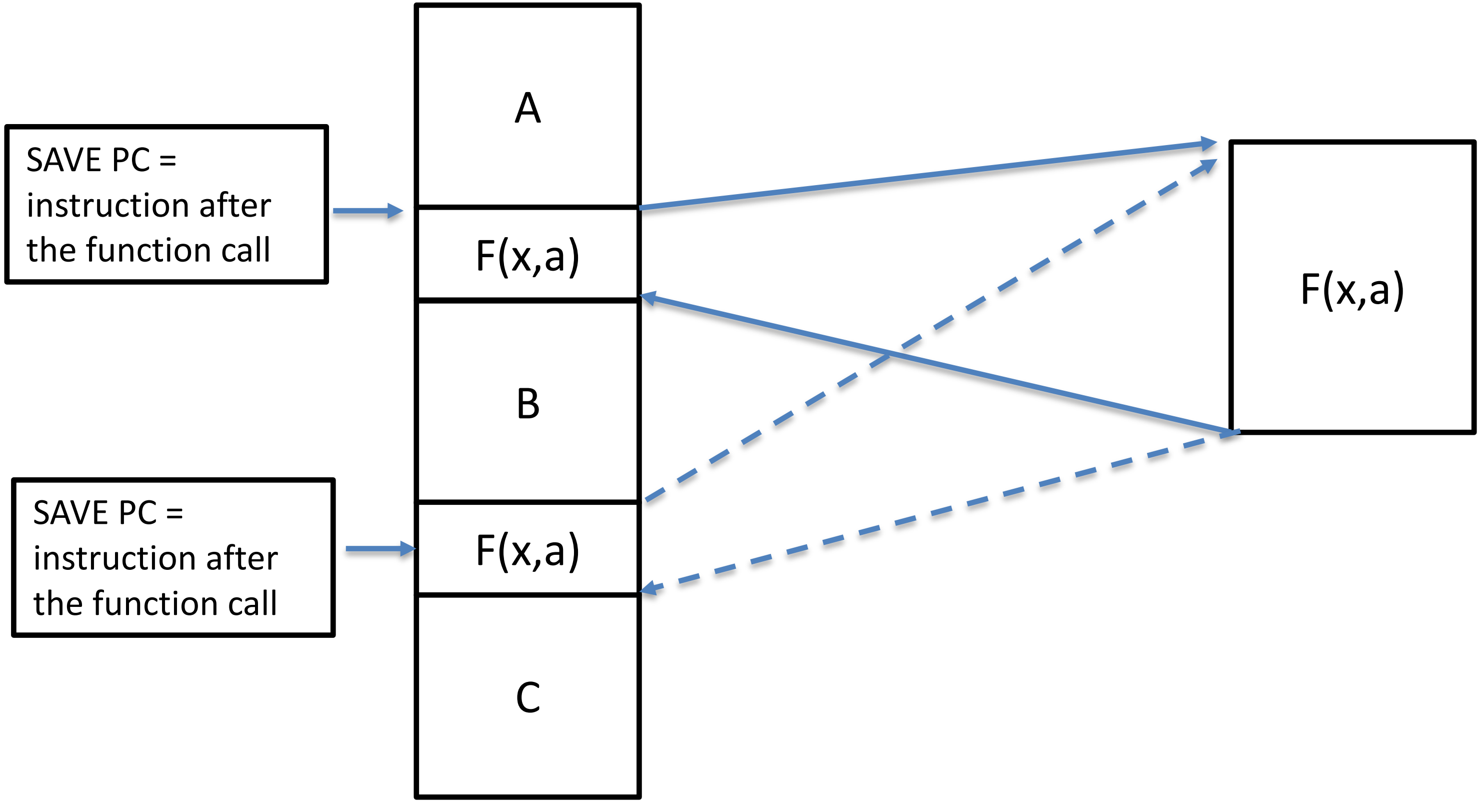


Jump back to the main code

WHAT ABOUT FUNCTIONS



But the next time we call the function. It needs to return to a different location



Jump back to saved value

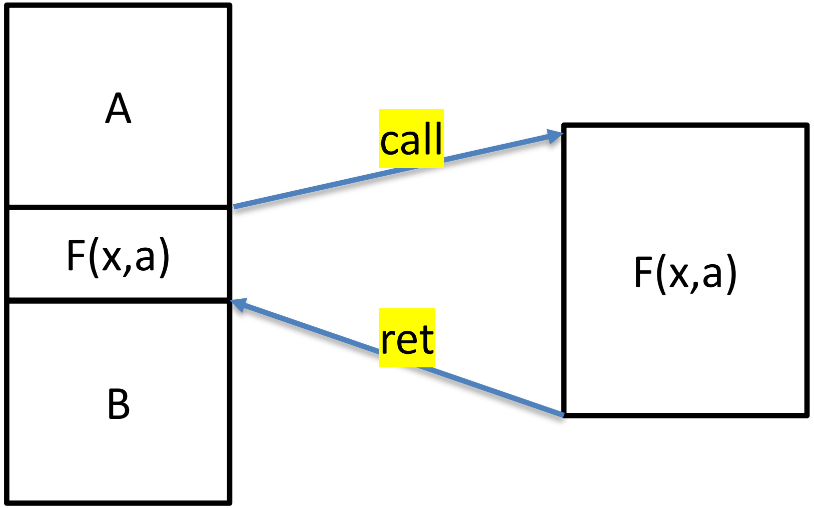
DEFINING A NEW INSTRUCTION

Let's create a new instruction that will both save the location to return and jump to the beginning of the function. We'll name this our **call** instruction

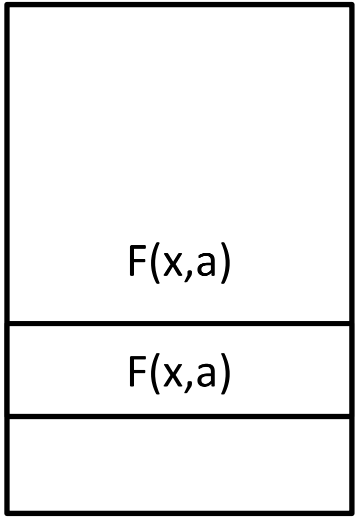
Save pc+2 , set pc = M[pc+1]

Let's also create an instruction that sets the PC back to the saved. We'll name this our return instruction or ret for short

pc = Saved Value

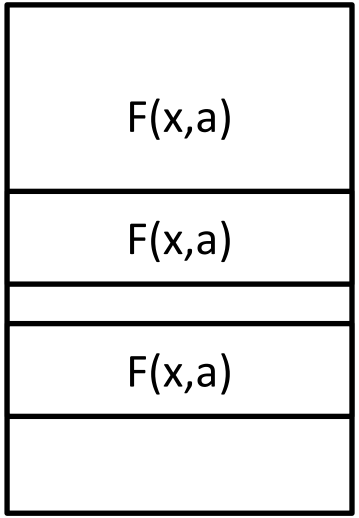


WHAT ABOUT FUNCTIONS



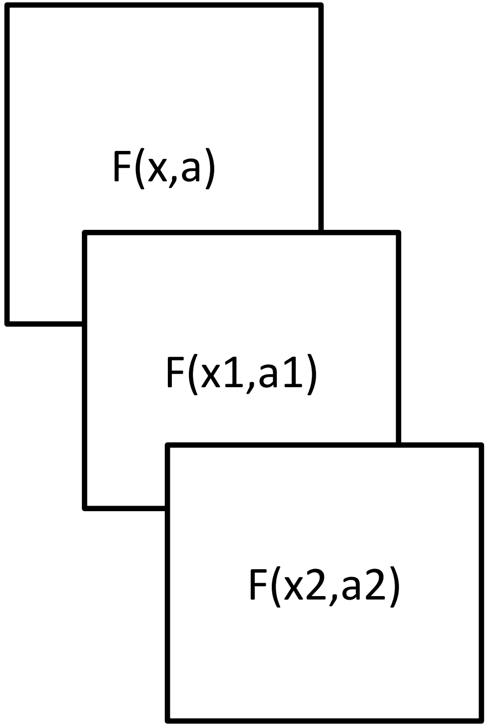
What about recursive functions? Functions that call themselves

WHAT ABOUT FUNCTIONS



What about recursive functions? Functions that call themselves

Now we need to keep track of both the location return to (multiple function calls and the register state of function before the call)



THE STACK

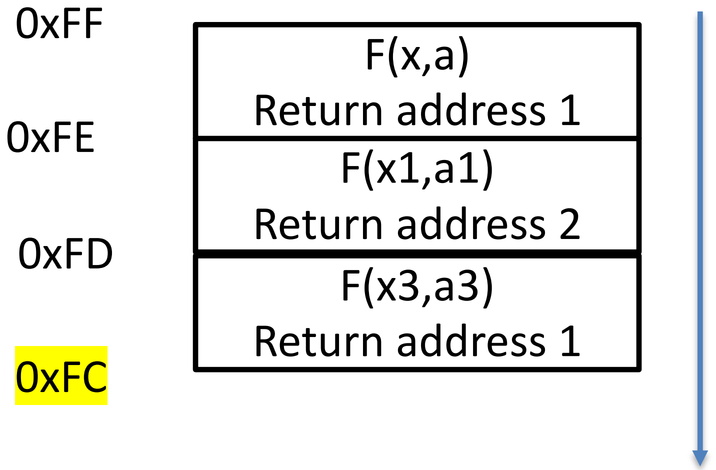
We are going to a region of memory that will hold the stack of function states and their associated return addresses.

0xFF	F(x,a) Return address 1
0xFE	F(x1,a1) Return address 2
0xFD	F(x3,a3) Return address 1
0xFC	

By convention keep adding new things to the stack by growing it to lower addresses

THE STACK

RSP 0xFC



We also define a new register that holds the location of the TOP of the stack in memory. We'll name this register RSP

PUSH AND POP INSTRUCTIONS

RSP 0xFC

Address	Stack
0xFF	F(x,a) Return address 1
0xFE	F(x1,a1) Return address 2
0xFD	F(x3,a3) Return address 1
0xFC	

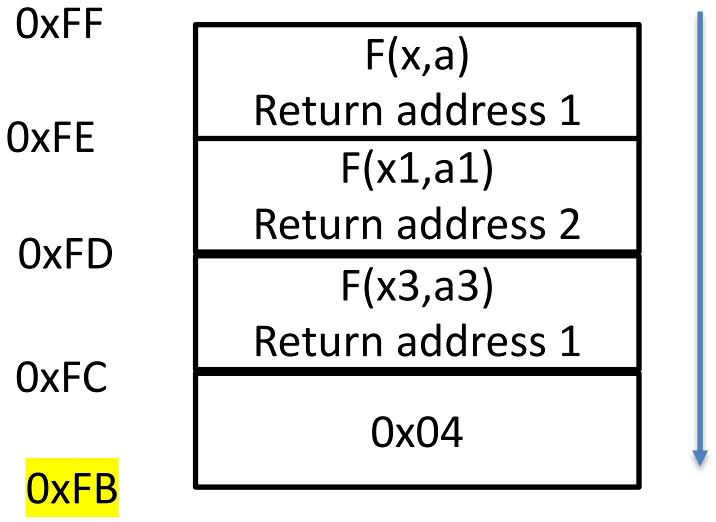
We'll also create two instructions that will add and remove values from the stack.

The push instruction will decrement the RSP and to the top of the stack

Example push(0x04)

PUSH AND POP INSTRUCTIONS

RSP 0xFB



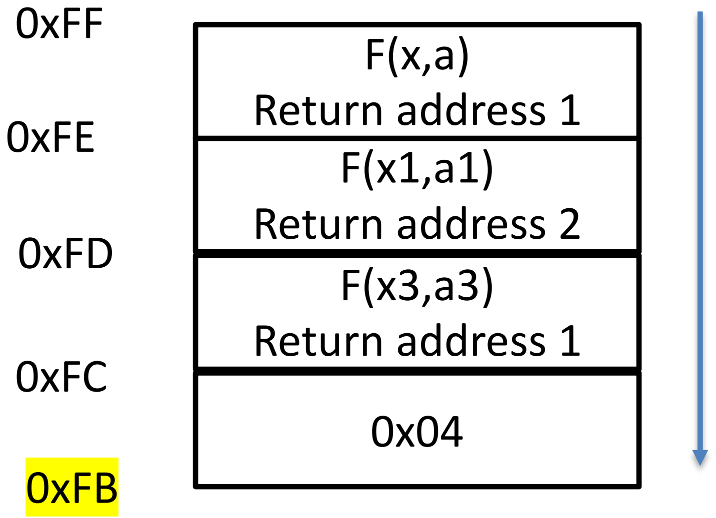
We'll also create two instructions that will add and remove values from the stack.

The push instruction will decrement the RSP and to the top of the stack

Example push(0x04)

PUSH AND POP INSTRUCTIONS

RSP 0xFB



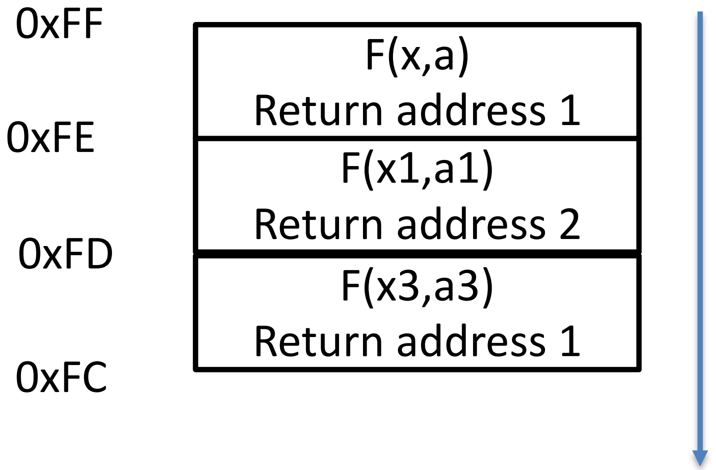
We'll also create two instructions that will add and remove values from the stack.

While the pop instruction increments RSP and returns the value at the top of the stack

Example x = pop()

PUSH AND POP INSTRUCTIONS

RSP | 0xFC



We'll also create two instructions that will add and remove values from the stack.

While the pop instruction increments RSP and returns the value at the top of the stack

Example x = pop() returns 0x04

WHAT ABOUT THE FUNCTION PARAMETERS

We need to define a calling convention. The rules that we'll follow when we call a function.

1. For our simple processor functions are limited to 2 parameters.
2. The first parameter will be stored in R2
3. The second parameter will be stored in R3
4. The return value of the function will be stored in R0
5. If the function uses any other registers save them before modifying them and restore them before returning.

```
input = 0xFF
shiftAmount = 0x02
output = left_shift(input, shiftArmount)
```

```
R2 = 0xFF
R3 = 0x02
call left_shift
R0 //Contains result
```

THOUGHT EXPERIMENTS

Could you implement the left_shift function using our toy ISA?

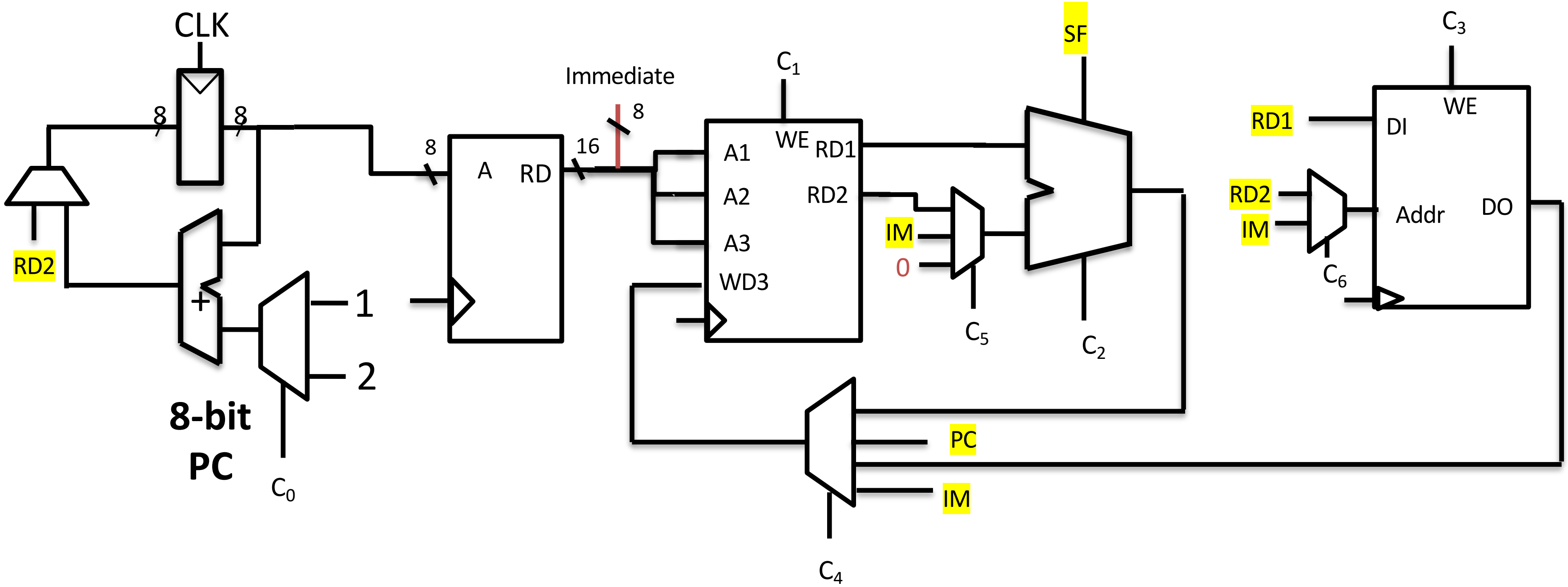
output = left_shift(input, shiftArmount)

Hint: Left shifts by 1 is equivalent to multiplying the number by 2.

ISA EXTENDED BY SETTING R BIT TO 1

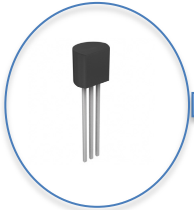
icode	b	operation
0		
	0	Decrement rsp and push the contents of rA to the stack
	1	Pop the top value from the stack into rA and increment rsp
	2	Push pc+2 onto the stack, set pc = M[pc+1]
	3	pc = pop the top value from the stack If b is not 2, update the pc as normal.

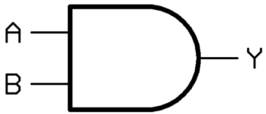
COULD YOU ADD PUSH, POP, CALL AND RET?

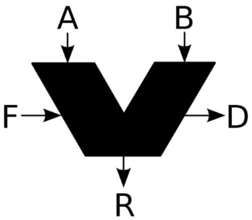


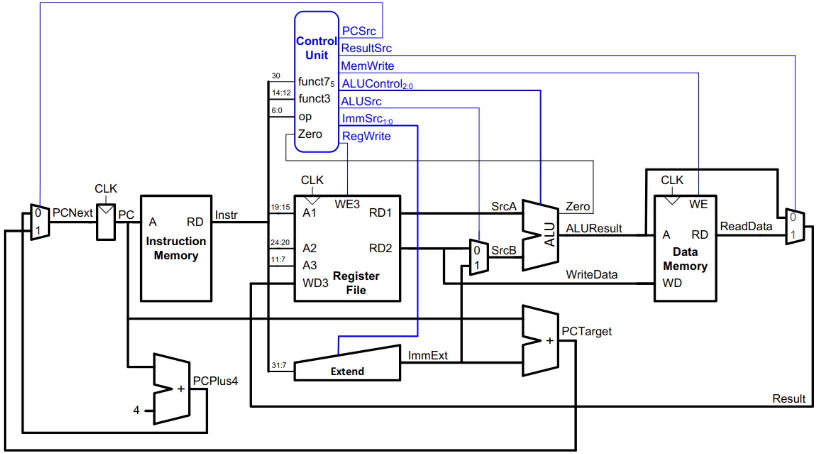
Write back stages

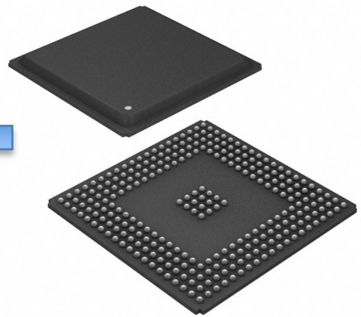
THE MAP (THE MACHINE)











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

THE STACK

The Stack is a region of memory

OUR JOURNEY SO FAR

THE MAP (THE CODE)

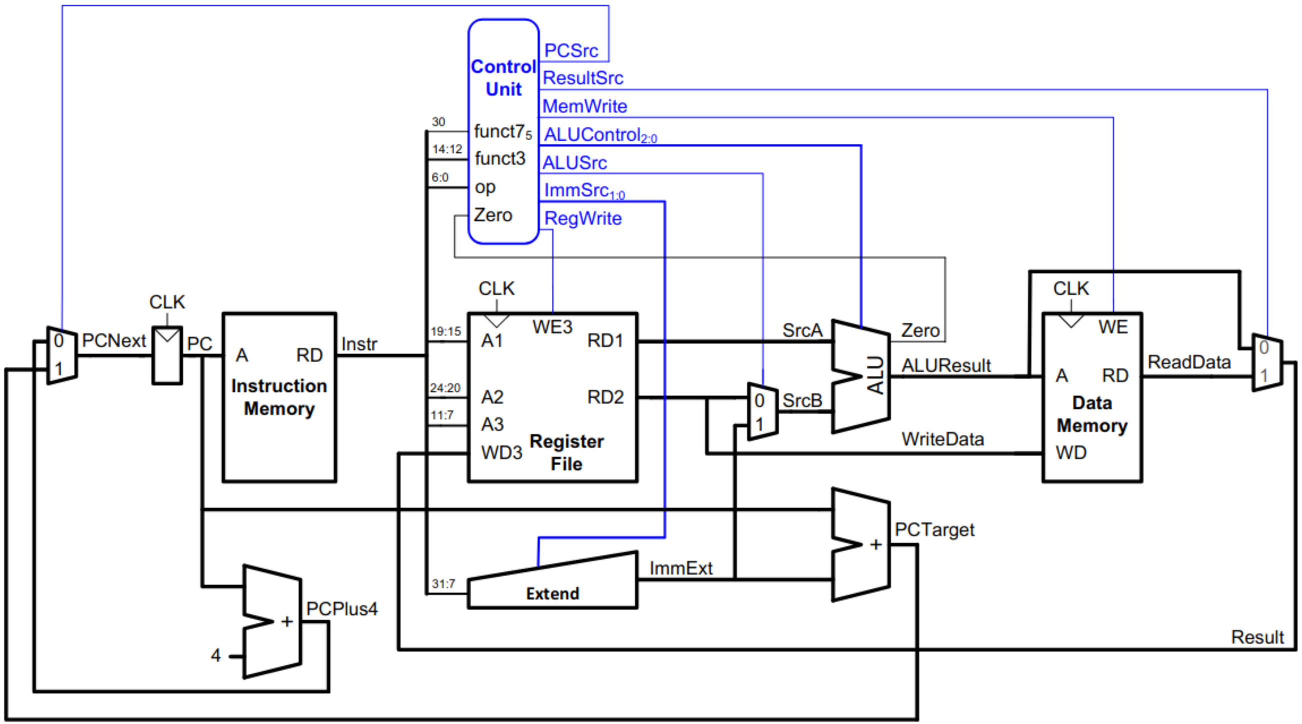
```c
#include <stdio.h>
int main() {
    printf("Hello, World!");
    return 0;
}
```

```
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
```

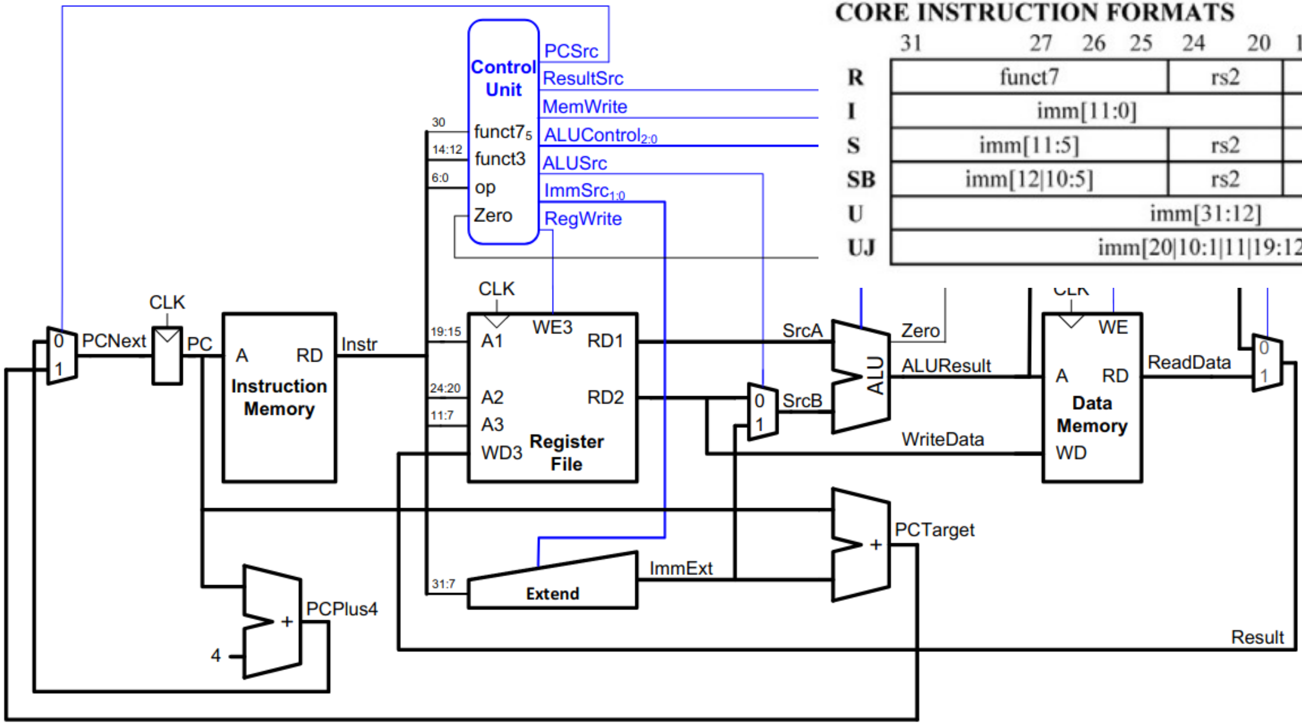
We will not cover this conversion in detail. CS 4620 - Compilers is a class dedicated to building and understanding the program designed to do this conversion.

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

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