CS0 2130
Overview

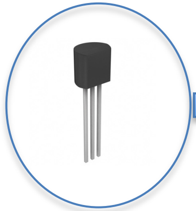
Daniel G. Graham PhD

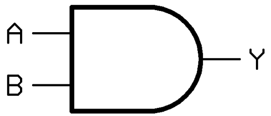
University of Virginia | ENGINEERING

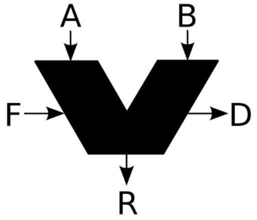
Contents

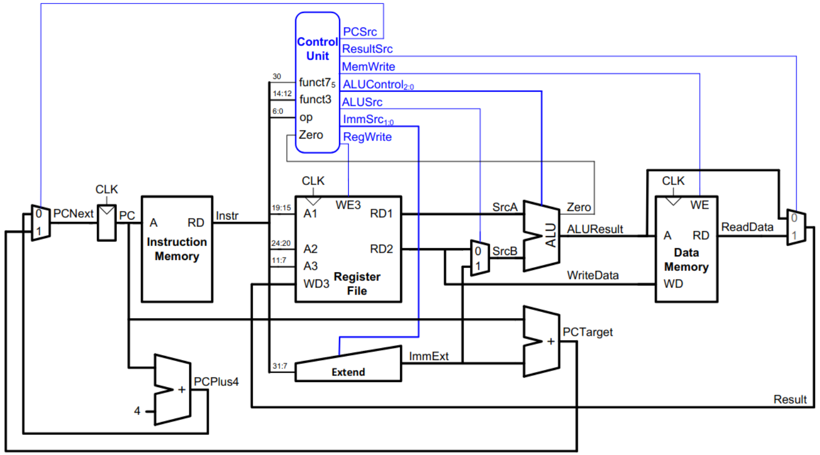
1. Discuss the ideas from our last 14 lectures.
2. Only 25 lectures left after this one. (Yeah, it goes by fast)
3. Topics, and tools for continuing your hardware journey
4. Now we move to software x86 assembly and C

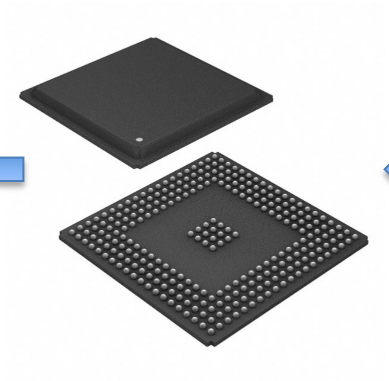
THE MAP (THE MACHINE)





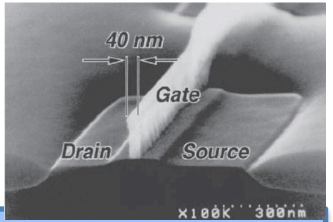


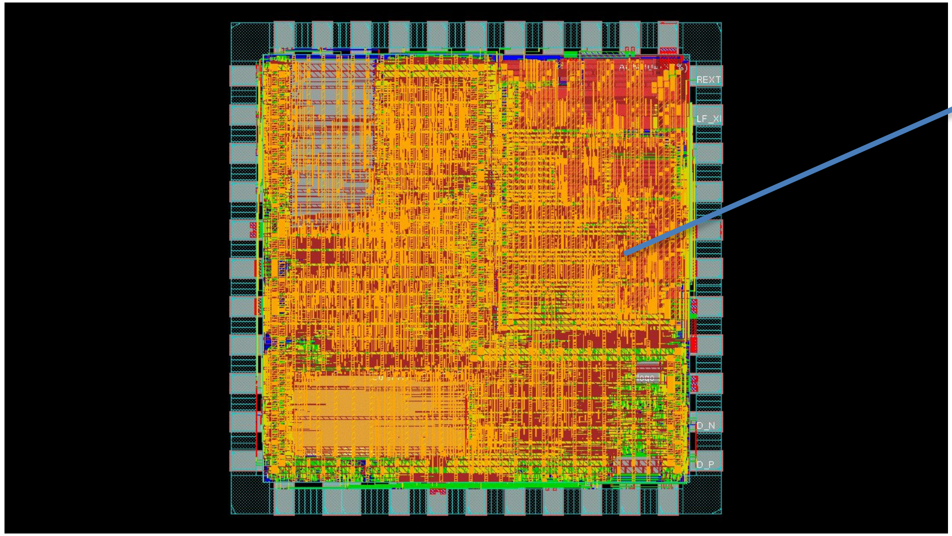


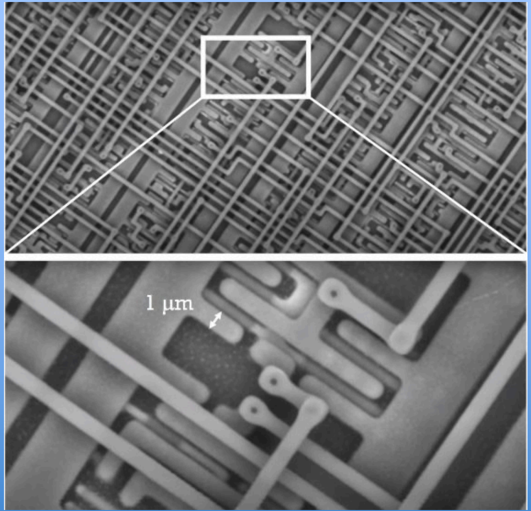


https://github.com/MKrekker/SINGLE-CYCLE-RISC-V

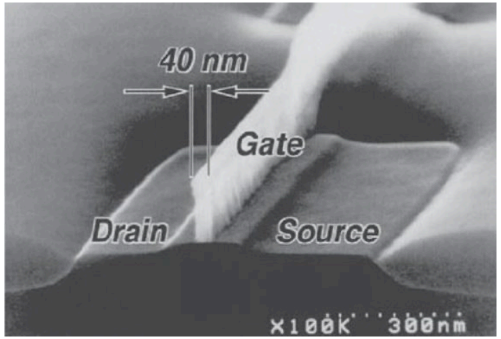
BOTTOM-UP APPROACH

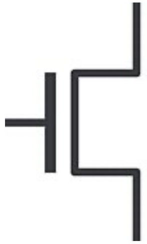


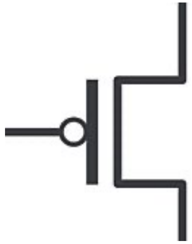




THIS WERE WE’LL START OUR JOURNEY



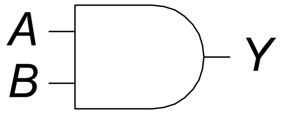




WHAT ARE LOGIC GATES

- Logic gates are circuits that perform logic functions
    - such as AND, OR, (NOT) , etc
- Logic gates have different symbols and their behavior is normally described using a truth table.
-

AND



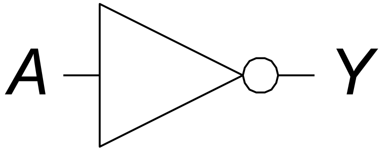
$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

UNIVERSITY of VIRGINIA | ENGINEERING

SINGLE INPUT VS TWO INPUT GATES

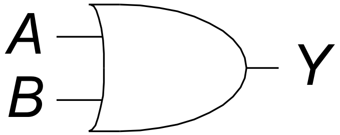
NOT



$$Y = \overline{A}$$

A	Y
0	
1	

OR

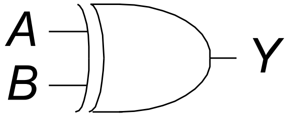


$$Y = A + B$$

A	B	Y
0	0	
0	1	
1	0	
1	1	

MORE LOGIC GATES

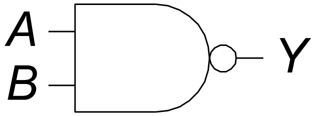
XOR



$Y = A \oplus B$

A	B	Y
0	0	
0	1	
1	0	
1	1	

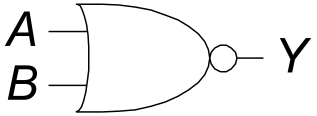
NAND



$Y = \overline{AB}$

A	B	Y
0	0	
0	1	
1	0	
1	1	

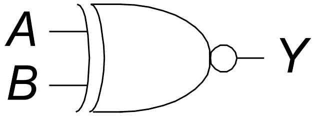
NOR



$Y = \overline{A + B}$

A	B	Y
0	0	
0	1	
1	0	
1	1	

XNOR



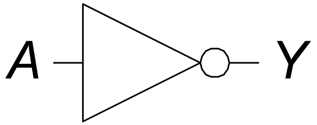
$Y = \overline{A \oplus B}$

A	B	Y
0	0	
0	1	
1	0	
1	1	

UNIVERSITY of VIRGINIA | ENGINEERING

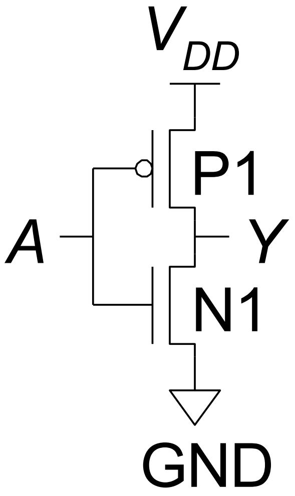
PULL UP PULL DOWN NETWORKS

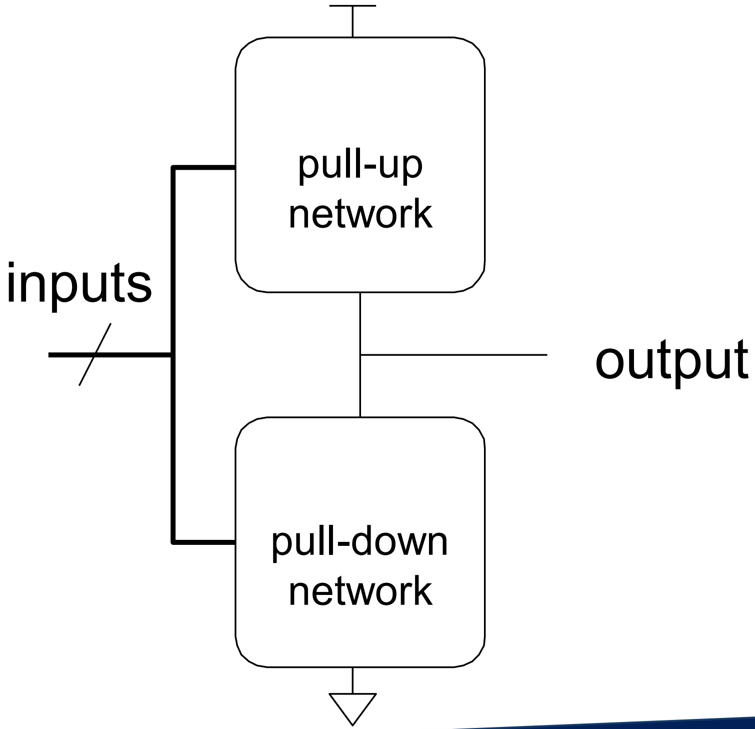
NOT



$Y = \overline{A}$

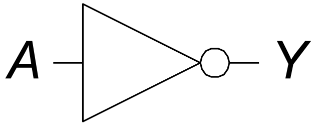
A	Y
0	1
1	0





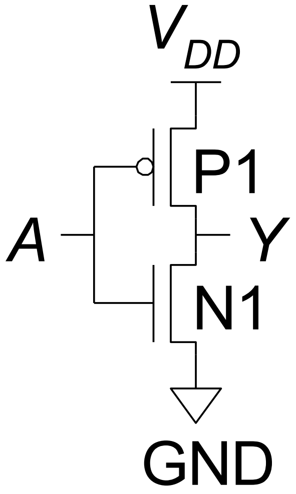
NOT GATE

NOT



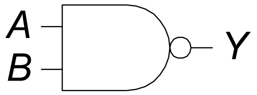
$Y = \overline{A}$

A	Y
0	1
1	0



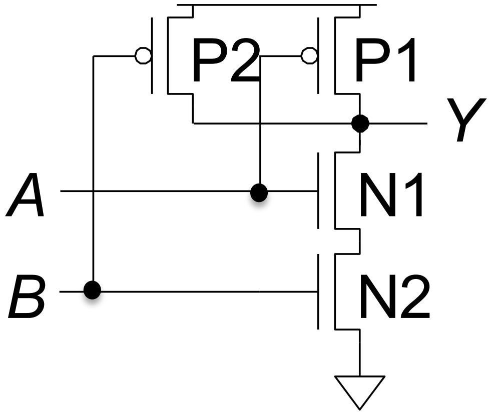
A	**P1**	**N1**	***Y***
0			
1			

NAND



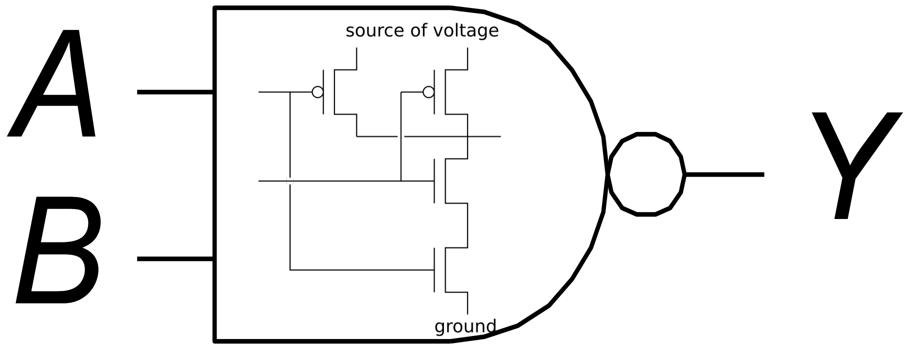
$Y = \overline{AB}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

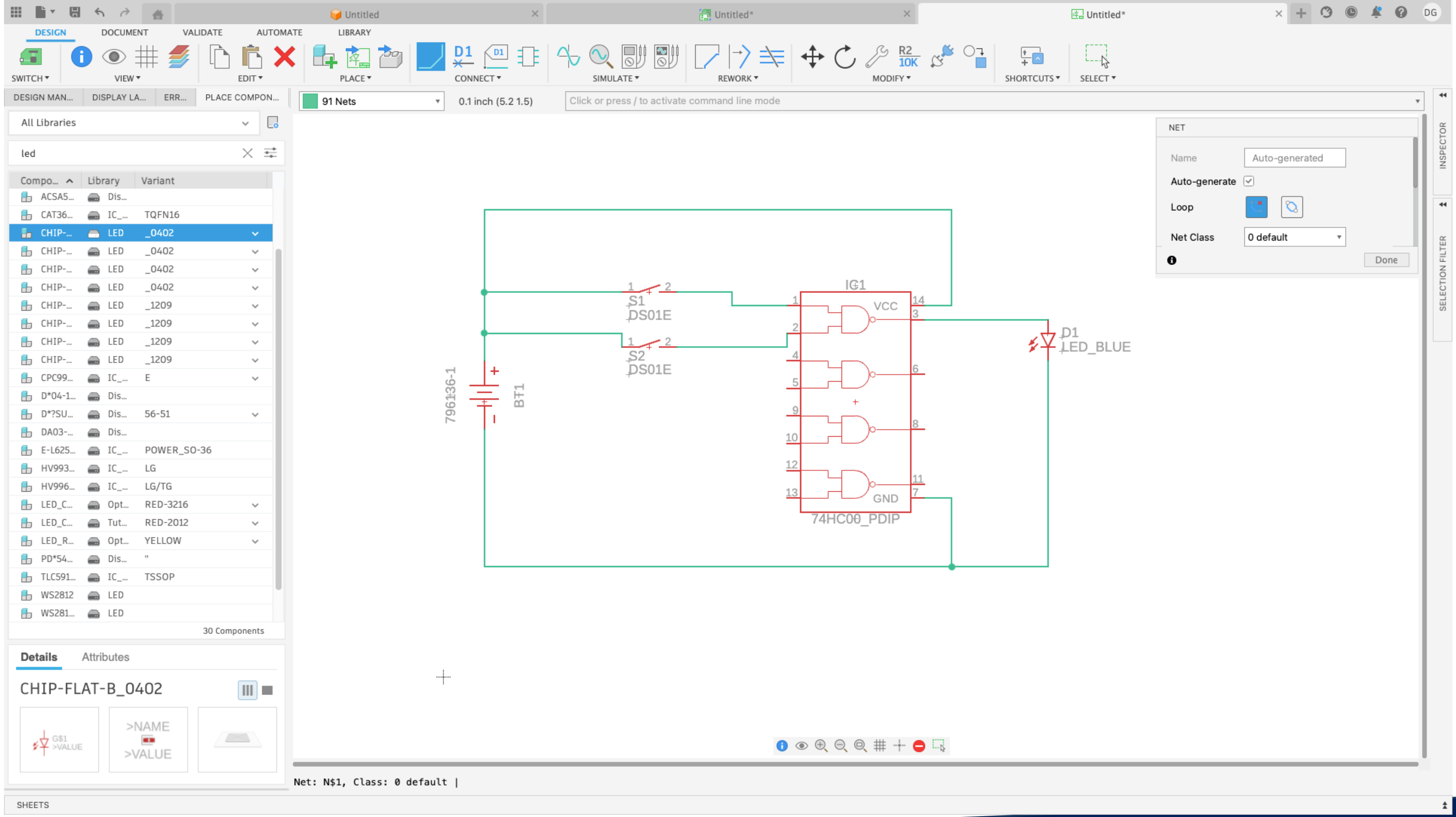


NAND gates are Turning complete you can build all other gates from them

NAND



$$Y = \overline{AB}$$



91 Nets | 0.1 inch (5.2 1.5) | Click or press / to activate command line mode

All Libraries

led

Compo...	Library	Variant
ACSA5...	Dis...	
CAT36...	IC_...	TQFN16
CHIP-...	LED	_0402
CHIP-...	LED	_0402
CHIP-...	LED	_0402
CHIP-...	LED	_0402
CHIP-...	LED	_1209
CHIP-...	LED	_1209
CHIP-...	LED	_1209
CHIP-...	LED	_1209
CPC99...	IC_...	E
D*04-1...	Dis...	
D*?SU...	Dis...	56-51
DA03-...	Dis...	
E-L625...	IC_...	POWER_SO-36
HV993...	IC_...	LG
HV996...	IC_...	LG/TG
LED_C...	Opt...	RED-3216
LED_C...	Tut...	RED-2012
LED_R...	Opt...	YELLOW
PD*54...	Dis...	"
TLC591...	IC_...	TSSOP
WS2812	LED	
WS281...	LED	

30 Components

Details | Attributes

CHIP-FLAT-B_0402

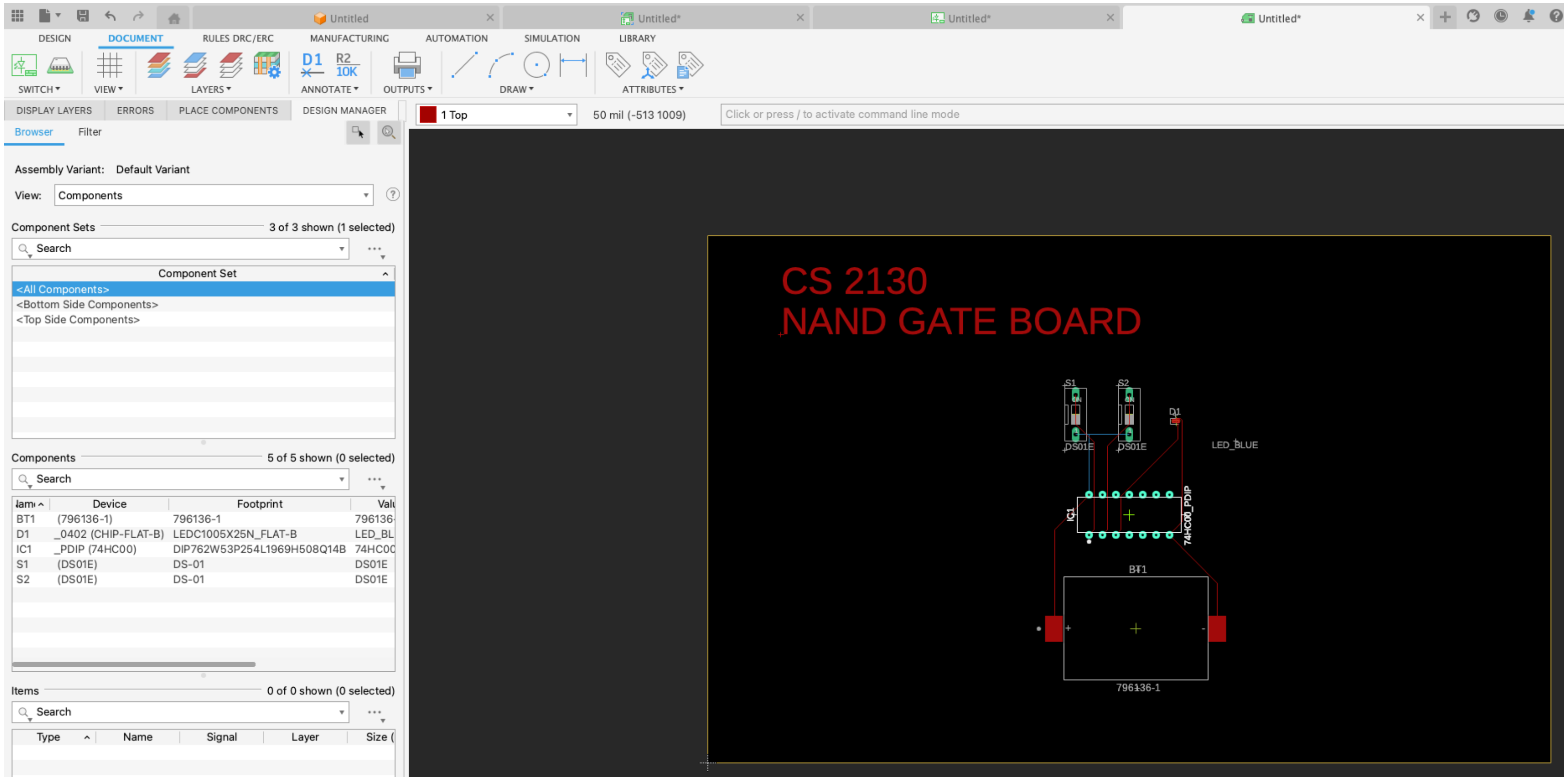
NET

Name	Auto-generated
Auto-generate	✓
Loop	
Net Class	0 default

Done

```
Net: N$1, Class: 0 default |
```

UNIVERSITY of VIRGINIA | ENGINEERING



Assembly Variant: Default Variant

View: Components

Component Sets 3 of 3 shown (1 selected)

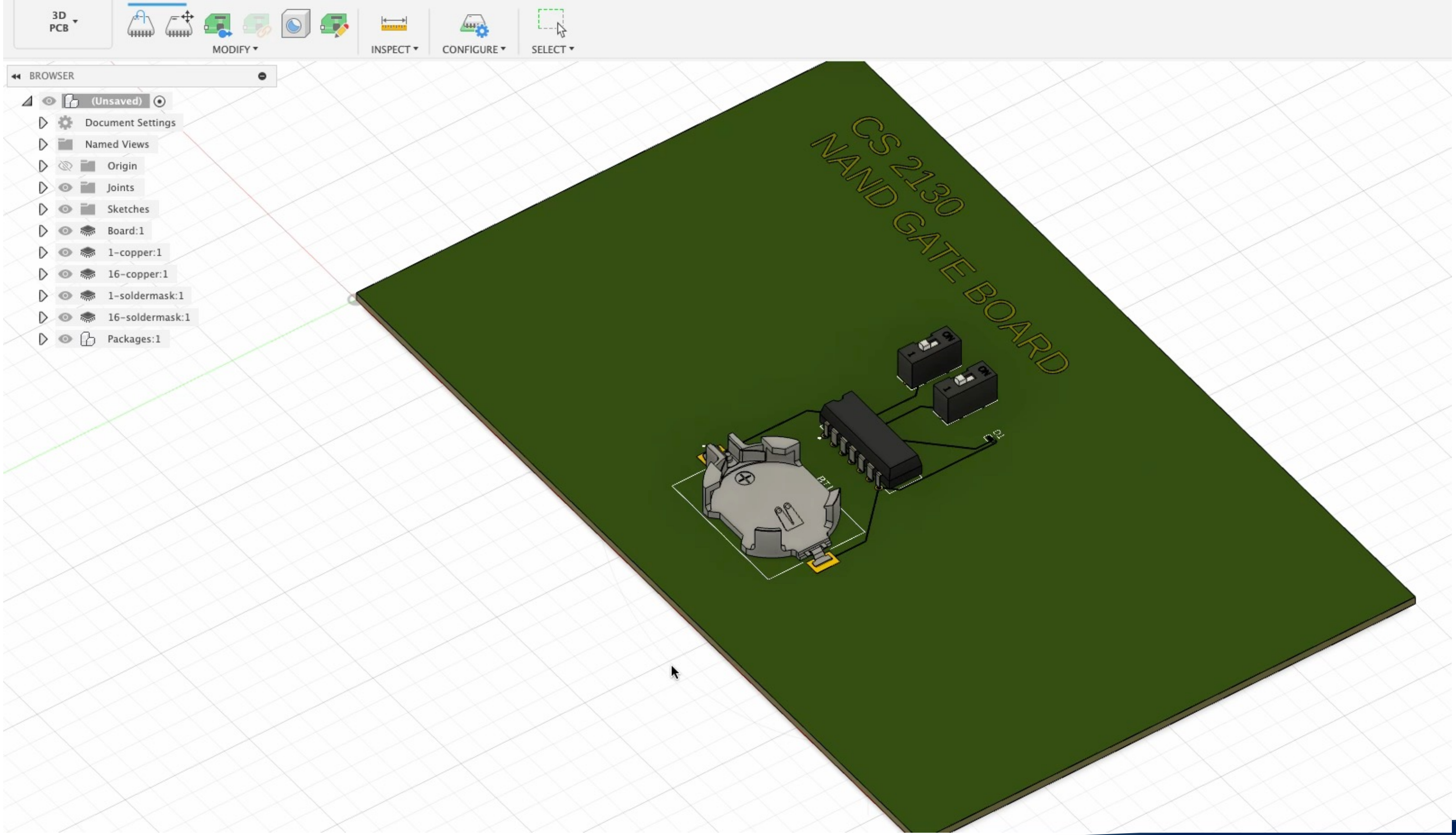
Component Set
<All Components>
<Bottom Side Components>
<Top Side Components>

Components 5 of 5 shown (0 selected)

Name	Device	Footprint	Value
BT1	(796136-1)	796136-1	796136-
D1	_0402 (CHIP-FLAT-B)	LEDC1005X25N_FLAT-B	LED_BL
IC1	_PDIP (74HC00)	DIP762W53P254L1969H508Q14B	74HC0C
S1	(DS01E)	DS-01	DS01E
S2	(DS01E)	DS-01	DS01E

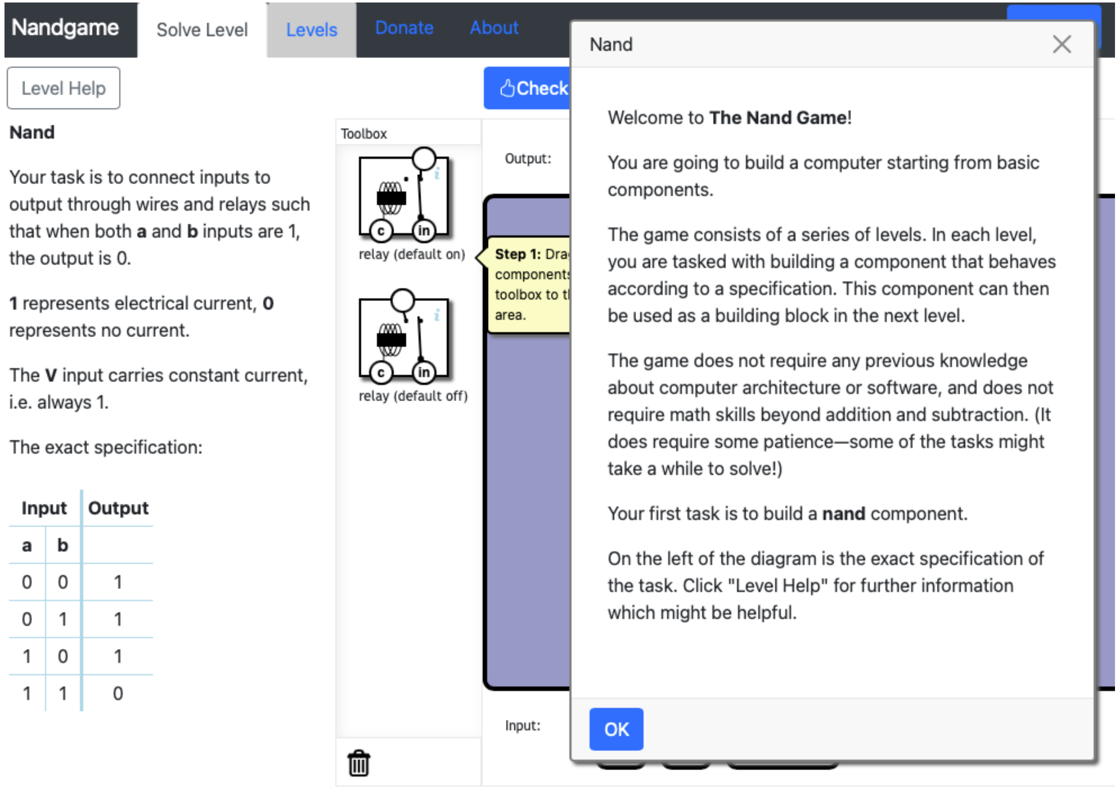
Items 0 of 0 shown (0 selected)

Type	Name	Signal	Layer	Size (



UNIVERSITY of VIRGINIA | ENGINEERING

THE NAND GAME



Nand

Your task is to connect inputs to output through wires and relays such that when both **a** and **b** inputs are 1, the output is 0.

1 represents electrical current, **0** represents no current.

The **V** input carries constant current, i.e. always 1.

The exact specification:

Input		Output
a	b	
0	0	1
0	1	1
1	0	1
1	1	0

Nand

Welcome to **The Nand Game**!

You are going to build a computer starting from basic components.

The game consists of a series of levels. In each level, you are tasked with building a component that behaves according to a specification. This component can then be used as a building block in the next level.

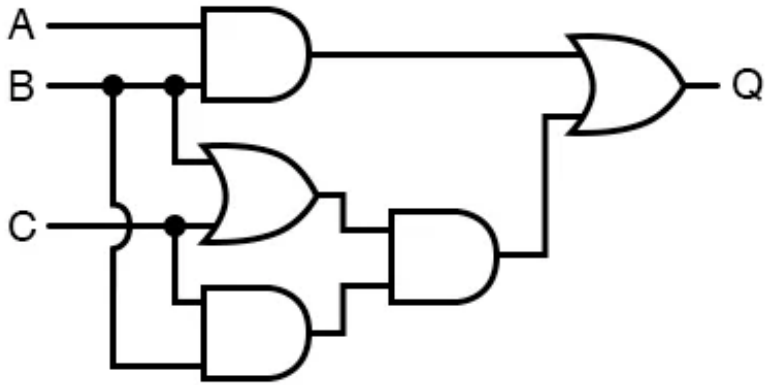
The game does not require any previous knowledge about computer architecture or software, and does not require math skills beyond addition and subtraction. (It does require some patience—some of the tasks might take a while to solve!)

Your first task is to build a **nand** component.

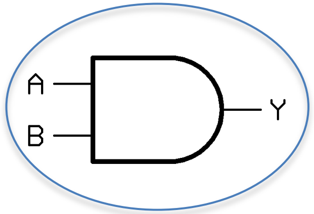
On the left of the diagram is the exact specification of the task. Click "Level Help" for further information which might be helpful.

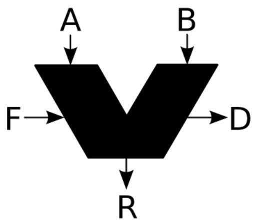
WHAT IS THE OUTPUT OF THIS CIRCUIT?

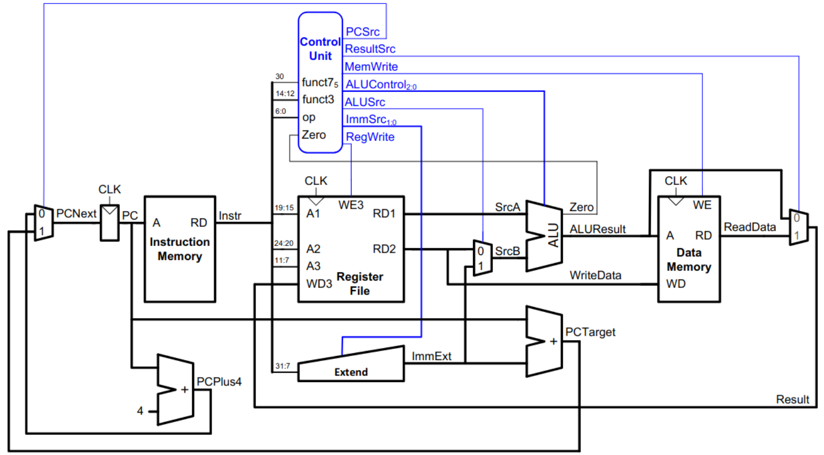
A	*B*	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

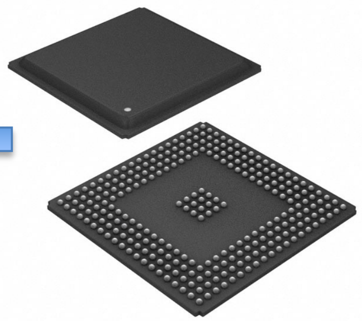


THE MAP (THE MACHINE)





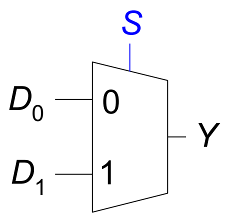




https://github.com/MKrekker/SINGLE-CYCLE-RISC-V

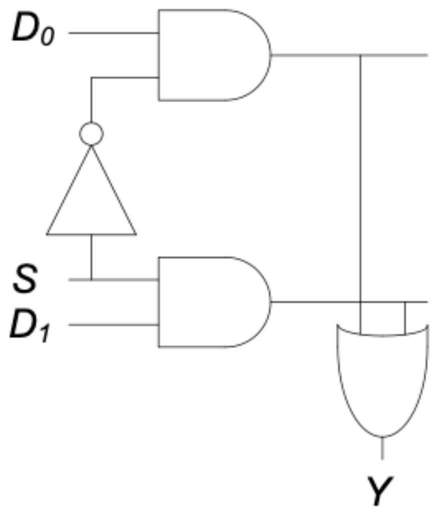
1 BIT MUX

$$Y = D_0\overline{S} + D_1S$$

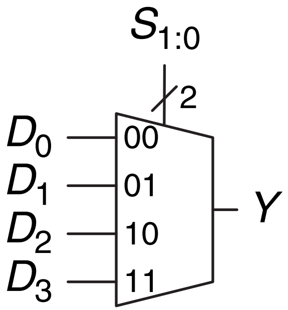


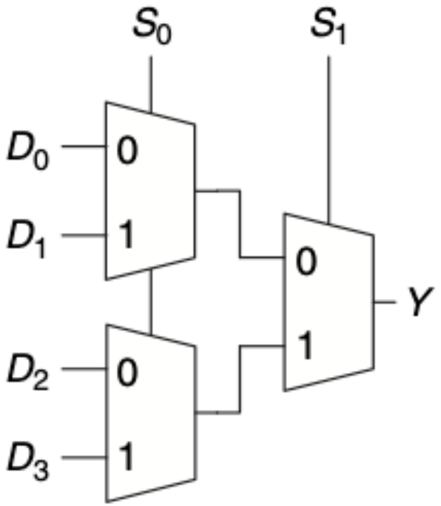
S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

S	Y
0	D_0
1	D_1

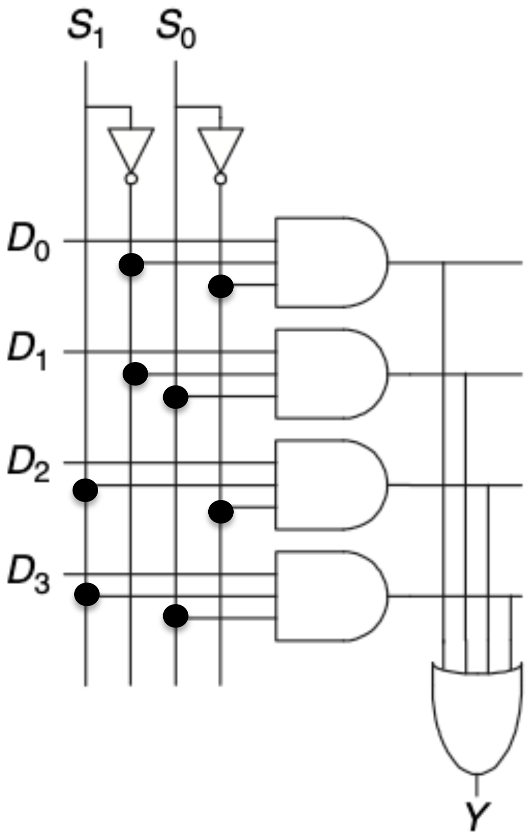


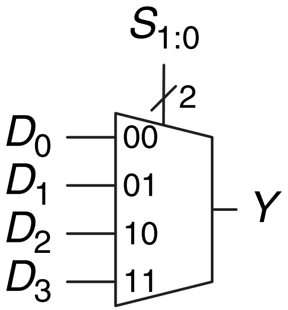
2 BIT MUX



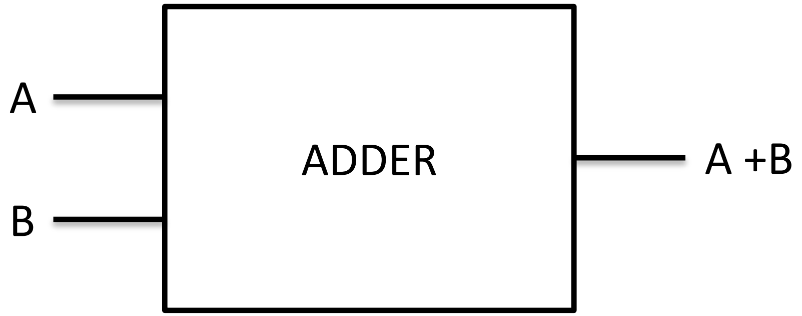


2 BIT MUX





THE IDEA



THE CHALLENGE

Our gates only support 0 and 1s.

How can we represent other decimal numbers?

How can we present negative numbers?

What about fractions ☺?

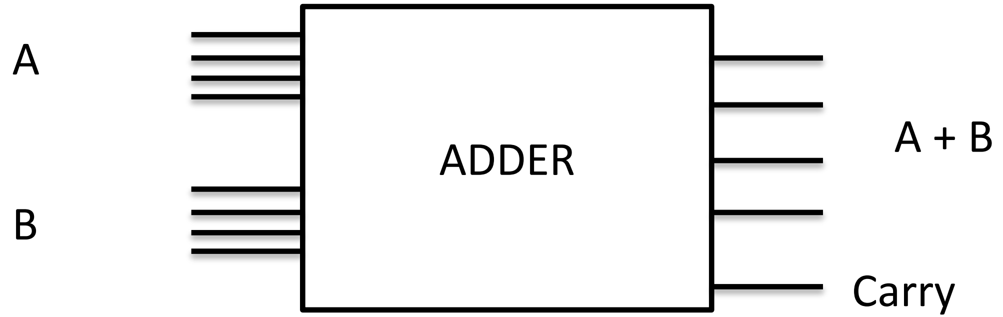
BINARY

8's column
4's column
2's column
1's column

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13_{10}$$

one eight	one four	no two	one one

4-BIT ADDER



Sign Bit

Signed	Bits	Unsigned
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
-0	1000	8
-1	1001	9
-2	1010	10
-3	1011	11
-4	1100	12
-5	1101	13
-6	1110	14
-7	1111	15

Bias

Signed	Bits	Unsigned
-7	0000	0
-6	0001	1
-5	0010	2
-4	0011	3
-3	0100	4
-2	0101	5
-1	0110	6
0	0111	7
1	1000	8
2	1001	9
3	1010	10
4	1011	11
5	1100	12
6	1101	13
7	1110	14
8	1111	15

$\text{Floor}((2^n -1)/2) = 7$

Two’s Complement

Signed	Bits	Unsigned
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
-8	1000	8
-7	1001	9
-6	1010	10
-5	1011	11
-4	1100	12
-3	1101	13
-2	1110	14
-1	1111	15

Hex Digit	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

HEXADECIMAL

Convert 00101110 to hexadecimal Answer: 2E

Group them

0010 = 2

1110 = E

Final 0x2E

- Some programming languages uses prefixes
  - Hex: 0x
    - 0x23AB = $23AB_{16}$
  - Binary: 0b
    - 0b1101 = 1101_2

UNIVERSITY of VIRGINIA | ENGINEERING

BITWISE OR |

$$\begin{array}{r} 1100_2 \\ |\ \ 0110_2 \\ \hline 1110_2 \end{array}$$

```
#python example
x = 12
y = 6
z = x | y
print(z)
#prints 14
```

BITWISE OR XOR ^

$$\begin{array}{r} 1100_2 \\ \wedge\ \ 0110_2 \\ \hline 1010_2 \end{array}$$

```
#python example
x = 12
y = 6
z = x ^ y
print(z)
#prints 10
```

FLIPPING BITS

File the second bit of x. 1 => 0 and 0 => 1

$$\begin{array}{r} 1100_2 \\ \wedge\quad 0010_2 \\ \hline 1110_2 \end{array}$$

What if the nth bit was 1 instead?

MASKING (EXTRACTING BITS)

The Idea of masking with can extra a certain section of bits by anding.

$$\begin{array}{r} 11011100_2 \\ \& \ 00001111_2 \\ \hline 0000\boxed{1100_2} \end{array}$$

Lower 4 bits extracted

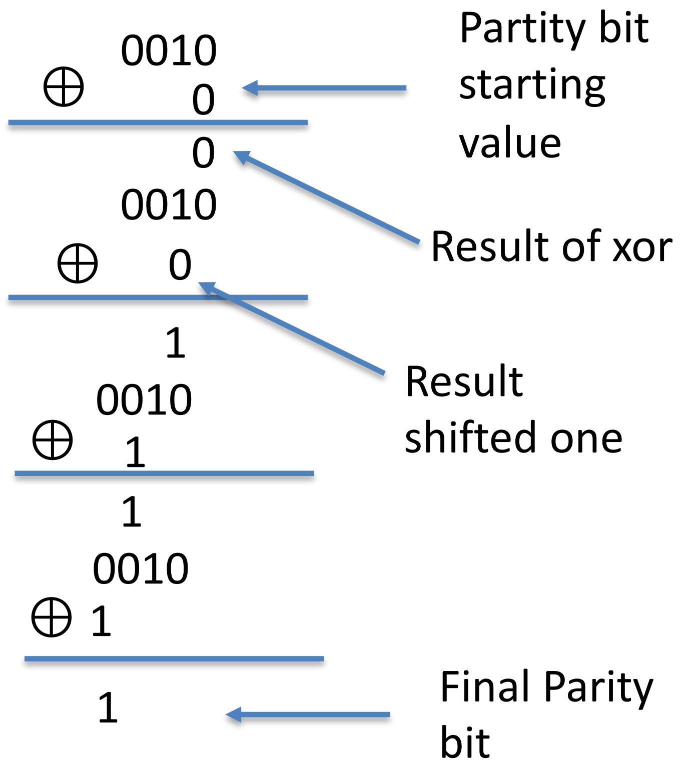
```python
#python example
x = 220
mask = 0x0F
x = x & mask
print(x)
#prints 12
```

PARITY

0010 parity bit is 1

0110 parity bit is 0

```
parity = 0
repeat 32 times:
     parity ^= (x&1)
      x >>= 1
```



Same as just xoring each bit 0 ⊕ 0 ⊕ 1 ⊕ 0 = 0

PARALLEL EVALUATION

Observe that xor is both transitive and associative; thus we can re-write

$x0 \oplus x1 \oplus x2 \oplus x3 \oplus x4 \oplus x5 \oplus x6 \oplus x7$

using transitivity as
$x0 \oplus x4 \oplus x1 \oplus x5 \oplus x2 \oplus x6 \oplus x3 \oplus x7$

and using associativity as
$(x0 \oplus x4) \oplus (x1 \oplus x5) \oplus (x2 \oplus x6) \oplus (x3 \oplus x7)$

and then compute the contents of all the parentheses at once via
`x ^ (x>>4)`.

PARALLEL EVALUATION

$x0 \oplus x1 \oplus x2 \oplus x3 \oplus x4 \oplus x5 \oplus x6 \oplus x7$
using transitivity as

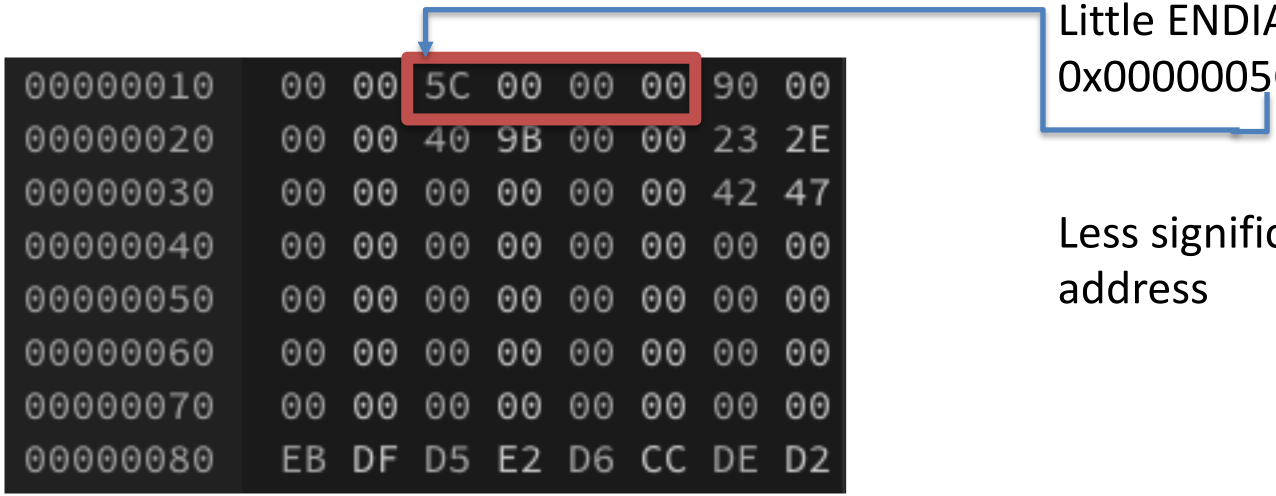
$x0 \oplus x4 \oplus x1 \oplus x5 \oplus x2 \oplus x6 \oplus x3 \oplus x7$

and using associativity as
$(x0 \oplus x4) \oplus (x1 \oplus x5) \oplus (x2 \oplus x6) \oplus (x3 \oplus x7)$

and then compute all at once via
x ^ (x>>4).

```
x ^= (x>>16)
x ^= (x>>8)
x ^= (x>>4)
x ^= (x>>2)
 x ^= (x>>1)
parity = (x & 1)
```

ENDIANNESS

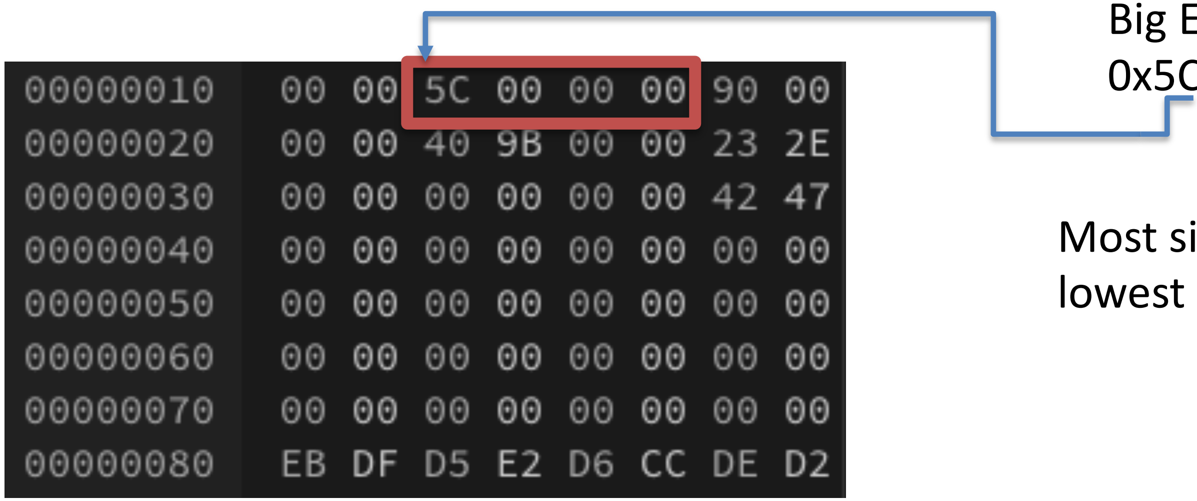


```
00000010   00 00 5C 00 00 00 90 00
00000020   00 00 40 9B 00 00 23 2E
00000030   00 00 00 00 00 00 42 47
00000040   00 00 00 00 00 00 00 00
00000050   00 00 00 00 00 00 00 00
00000060   00 00 00 00 00 00 00 00
00000070   00 00 00 00 00 00 00 00
00000080   EB DF D5 E2 D6 CC DE D2
```

Little ENDIAN
0x0000005C (92_{10})

Less significant at Lowest address

ENDIANNESS

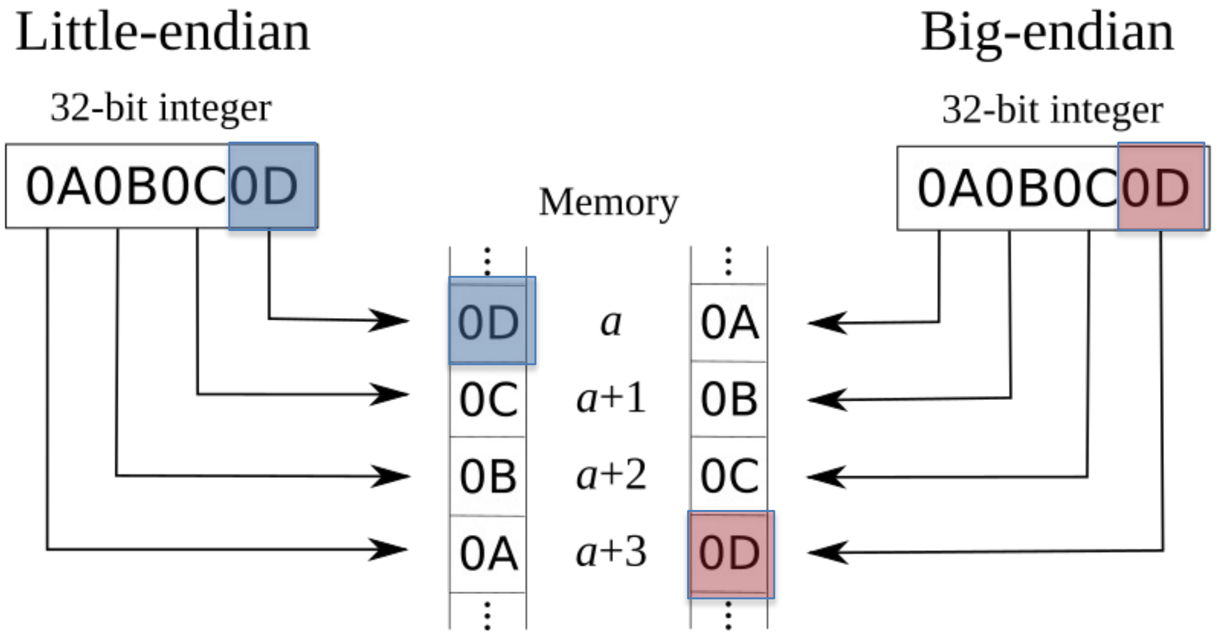


```
00000010    00 00 5C 00 00 00 90 00
00000020    00 00 40 9B 00 00 23 2E
00000030    00 00 00 00 00 00 42 47
00000040    00 00 00 00 00 00 00 00
00000050    00 00 00 00 00 00 00 00
00000060    00 00 00 00 00 00 00 00
00000070    00 00 00 00 00 00 00 00
00000080    EB DF D5 E2 D6 CC DE D2
```

Big ENDIAN
0x5C000000 (1543503872_{10})

Most significant Byte at lowest address

ENDIANNESS


Little-endian
32-bit integer
0A0B0C0D
Big-endian
32-bit integer
0A0B0C0D
Memory
0D
0C
0B
0A
a
a+1
a+2
a+3
0A
0B
0C
0D

FLOATING POINT

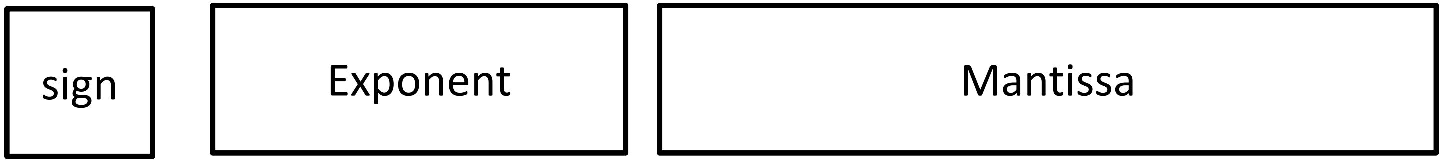
- How can we represent decimal values in binary?
- Why do errors like these occur?

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
>>> (0.1 + 0.1 + 0.1) == 0.3
False
>>> x = 0.1 + 0.1 + 0.1
>>> x
0.30000000000000004

>>> 0.3 + 0.3 + 0.3
>>> 0.8999999999999999
```

Floating point rounding error $\longrightarrow$

IEEE 754



$$\text{number} = \text{sign}(\,1 + \text{Mantissa}) \times 2^{\text{exponent} - \text{bias}}$$

On 32 bit machines bias in normal 127 (Yes this is bias representation we talked about earlier)

BINARY STRING

$0.1101 = 1.101 \times 2^{-1}$

Keep going until you get to your first 1.

$0.01101 = 1.101 \times 2^{-2}$

$0.001101 = 1.101 \times 2^{-3}$

CONVERSION EXAMPLE

Let’s convert 0.8125 to floating-point representation

0.8125 x 2=1.6250 **1**
0.6250 x 2=1.2500 **1**
0.2500 x 2=0.5000 **0**
0.5000 x 2=1.0000 **1**

2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	1	0	1

1 x 1/2 + 1x 1/4 + 0 x 1/8 + 1 x 1/16 = 13/16

0.1101
= 1.101 x 2^{-1}

CONVERSION EXAMPLE PART 2

$0.1101 = 1.\boxed{101} \times \boxed{2^{-1}}$

Sign: 0

Mantissa: 101

Exponent: -1 + 127 = 126(d) = 1111110(b)

0 01111110 1010000 00000000 00000000

CONVERSION PART 3

0.1 x 2 = 0.2	**0**
0.2 x 2 = 0.4	**0**
0.4 x 2 = 0.8	**0**
0.8 x 2 = 1.6	**1**
0.6 x 2 = 1. 2	**1**
0.1 x 2 = 0.2	**0**

…… repeats …

Just like the 1/3 0.1 keeps repeating

0 01111011 1001100 11001100 1100110

123

$\frac{1}{2} + 1/2^4 + 1/2^5 + 1/2^8 + 1/2^9 + 1/2^{12} + 1/2^{13} + 1/2^{16} + 1/2^{17} + 1/2^{20} + 1/2^{21}$

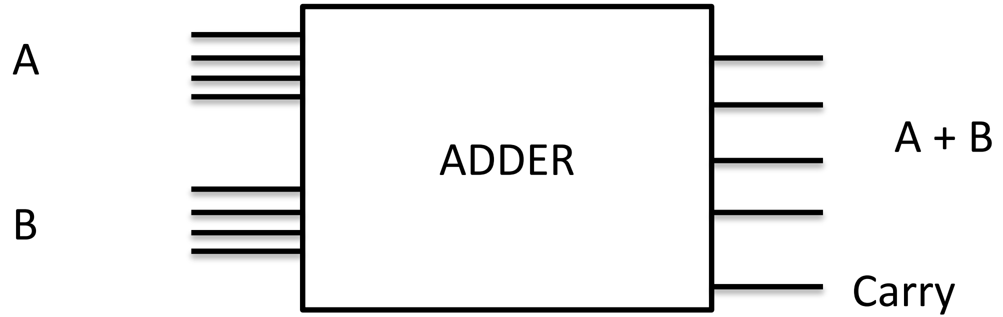
$$(-1)^s \times (1 + m) \times 2^{\text{exponent} - \text{bias}}$$

$$0.0999999940395355224609375 = (-1)^0 \times \left(1 + \frac{1\,258\,291}{2\,097\,152}\right) \times 2^{123 - 127}$$

No quite 0.1

GREAT NOW WE HAVE ALL WE NEED TO THINK ABOUT DESIGNING OUR ADDER

4-BIT ADDER



Great now let’s build it with gates.

ADDING

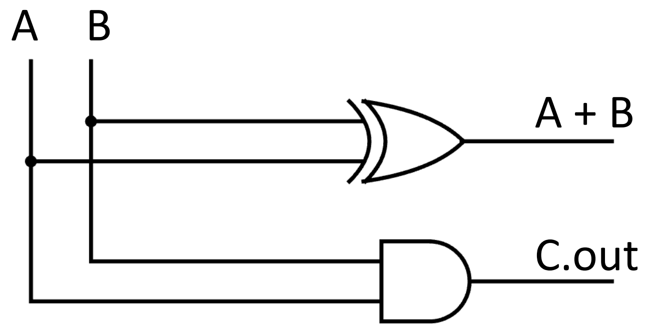
```
1 1 1 1   ← Carries
  0 1 1 1
+ 1 0 1 1
  0 0 1 0
```

Let start by building a half adder something that just adds two bits.

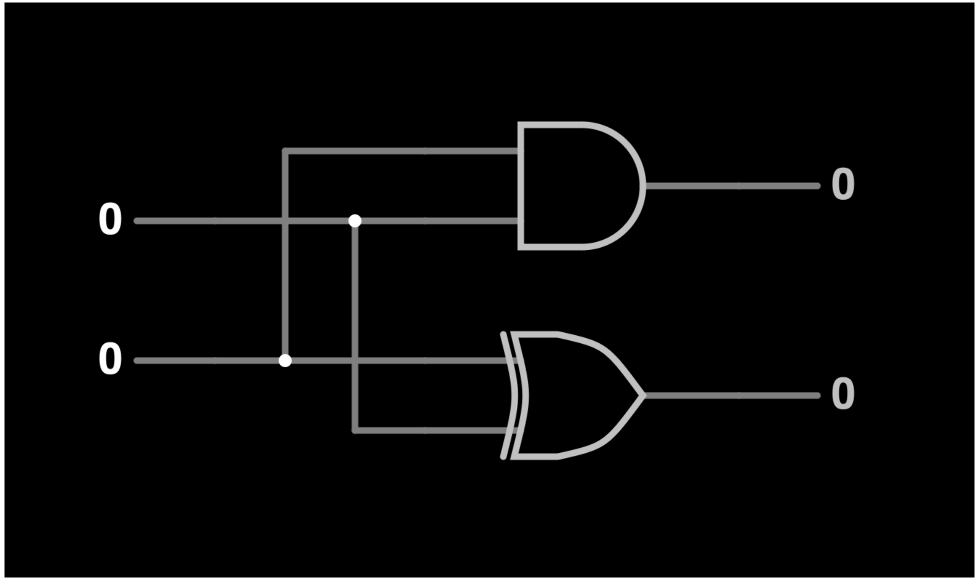
Let's build a truth table.

A	B	A + B	C.out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

We can implement
A + B with an XOR gate
And the C.out (Carry out)
With an AND gate



HALF ADDER DEMO



https://tinyurl.com/ygpea8v4

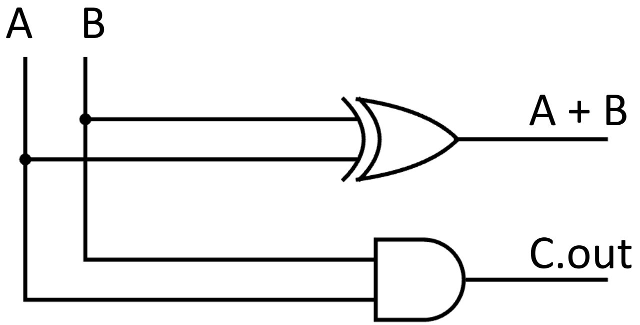
http://www.falstad.com/circuit/circuitjs.html?ctz=CQAgjCAMB0l3BWc0FwCwCY0HYEA4cEMElURTJyBTAWjDACgwE0QMs21KBmANj06VKGKOSZl2rMGlZ8B01sNEIGAGXAZ5vSnkphtbUQDMAhgBsAzlXJQ1GgZJC62HEZVOXrSSAwDu9lykDRx9-fWEOcIDQ8AMwTUDov1iI1kcQ5PitPQBOESiYsDyU8GLiXlswsoQK9JrK0vzgyIMfAFkQOXAZEDR9brS2FAYOrqxKPtquQwxhoA

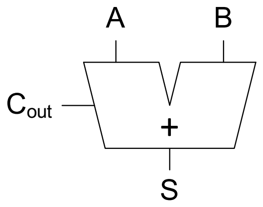
ADDING

Half Adder

```
1 1 1 1   ← Carries
  0 1 1 1
+ 1 0 1 1
  0 0 1 0
```

We can implement
A + B with an XOR gate
And the C.out (Carry out)
With an AND gate

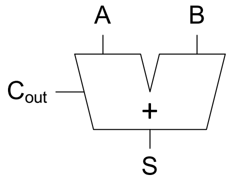




A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$
$$C_{out} = AB$$

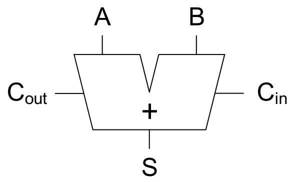
Half Adder



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$
$C_{out} = AB$

Full Adder



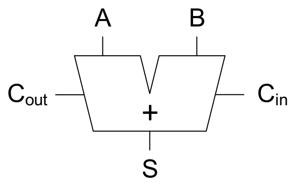
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

```
1 1 1 1   ← Carries
  0 1 1 1
+ 1 0 1 1
  0 0 1 0
```

Note on special case 3 input xor. Draw the three gates. Really Three xors stacked.

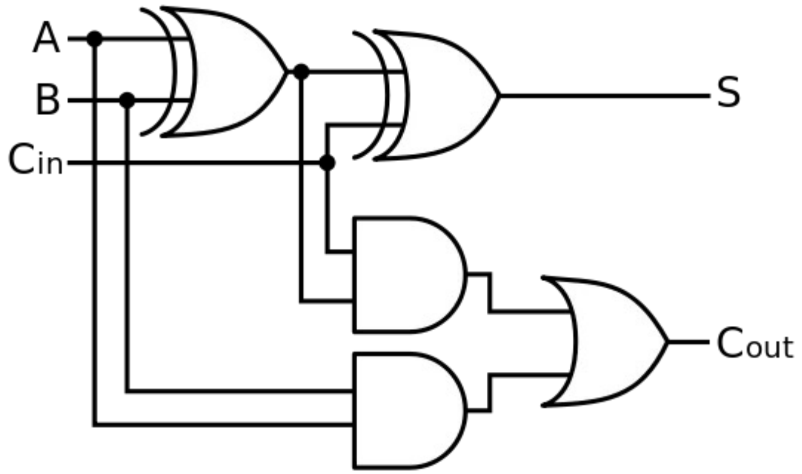
Full Adder



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

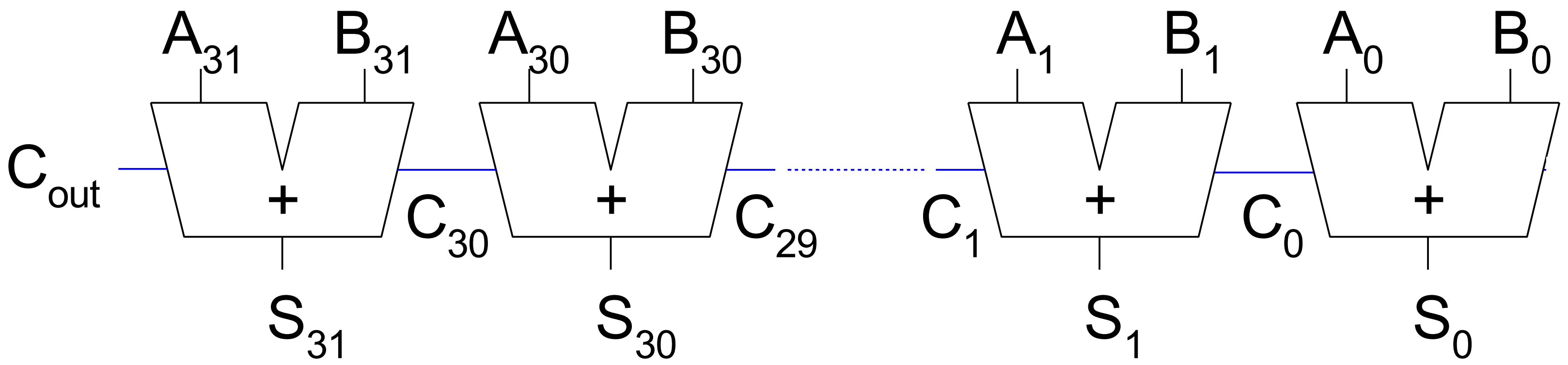
```
 1 1 1 1   ← Carries
   0 1 1 1
 + 1 0 1 1
   1 1 1 0
```

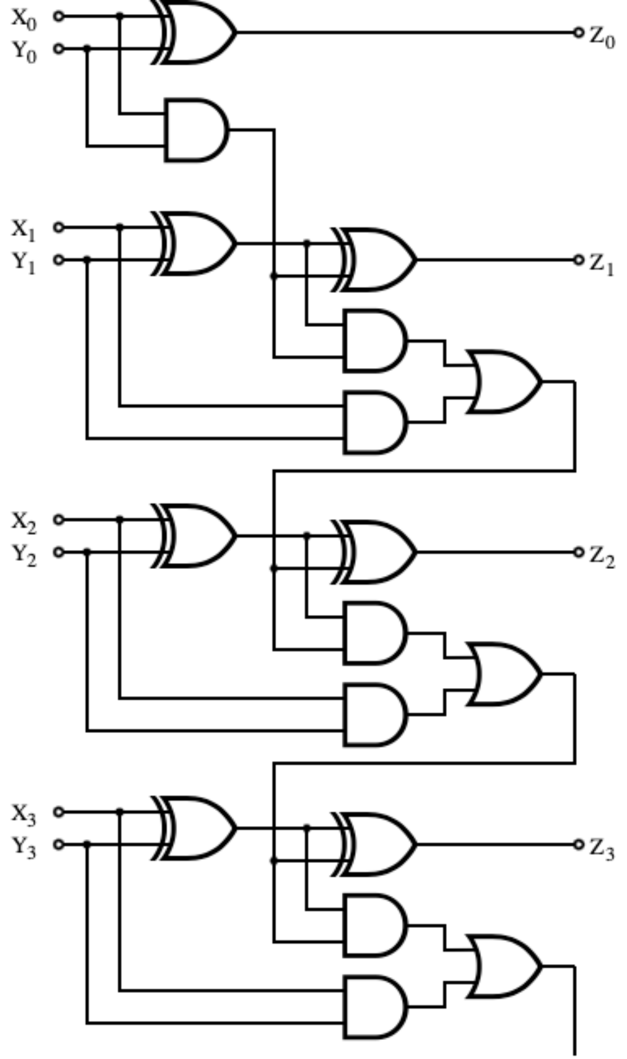


C.out has been rewritten to reduce the number of gates needed.

RIPPLE CARRY ADDER

Next let's build a full adder



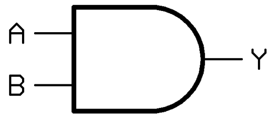


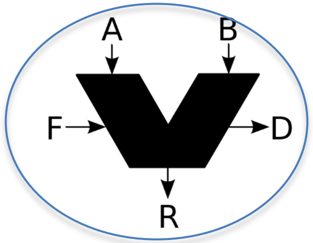
RIPPLE CARRY ADDER

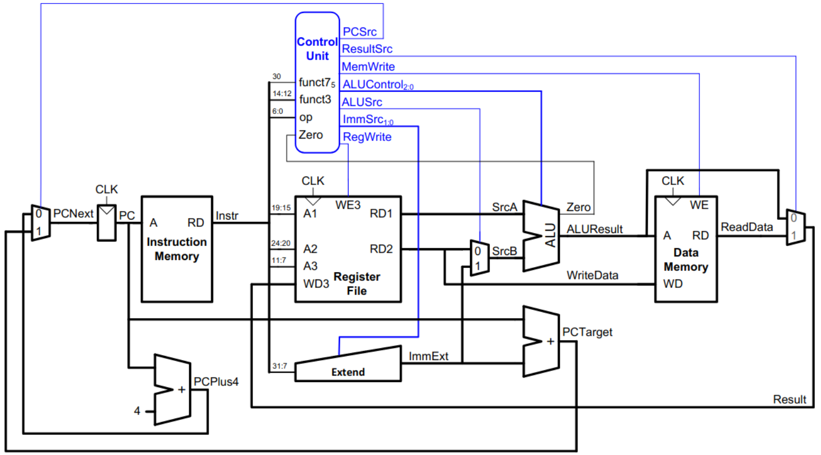
```
1 1 1 1    ← Carries
  0 1 1 1
+ 1 0 1 1
  -------
   0 0 1 0
```

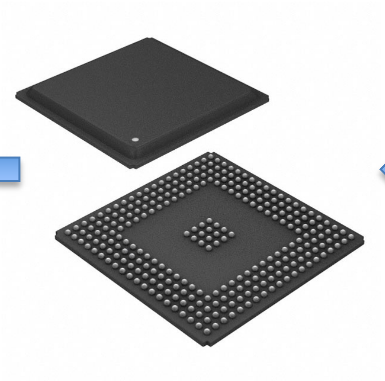
University of Virginia | ENGINEERING

THE MAP (THE MACHINE)





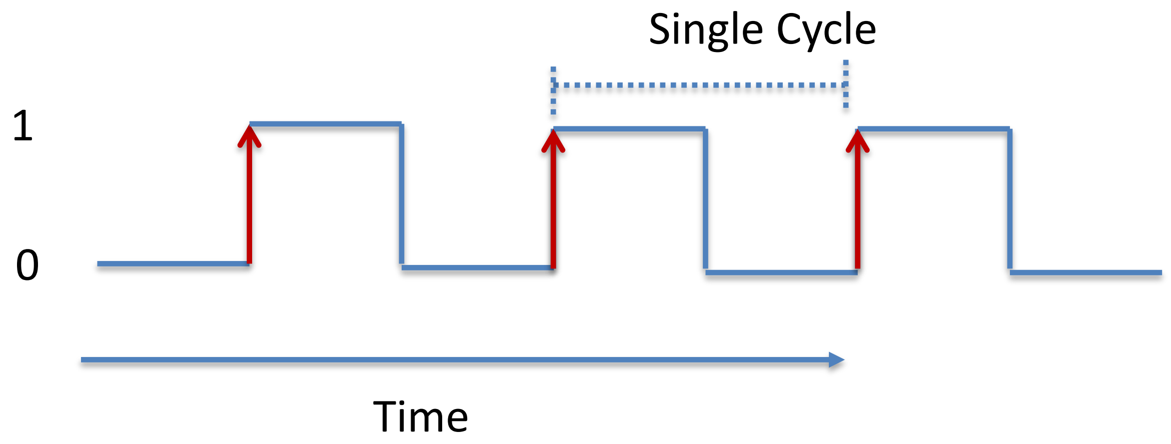




https://github.com/MKrekker/SINGLE-CYCLE-RISC-V

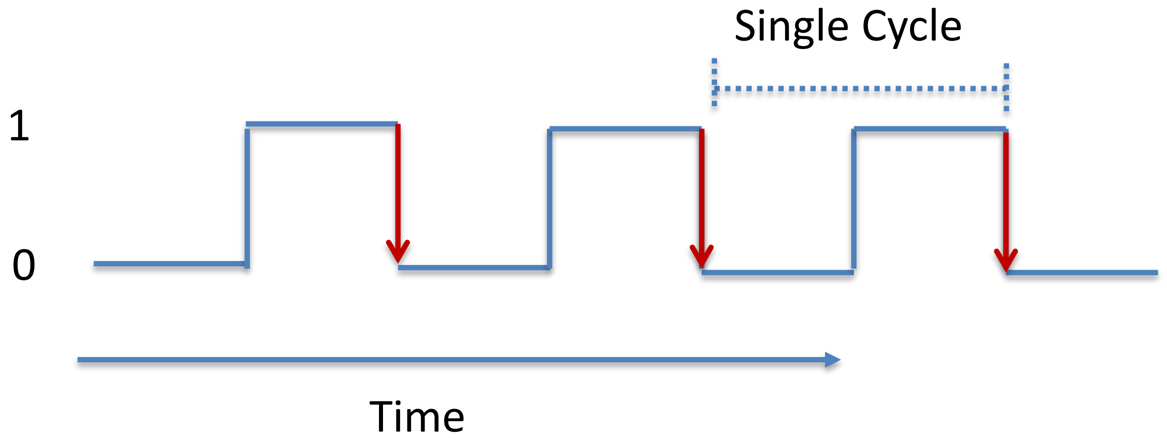
CLOCKS EDGES

Rising Edge



CLOCKS EDGES

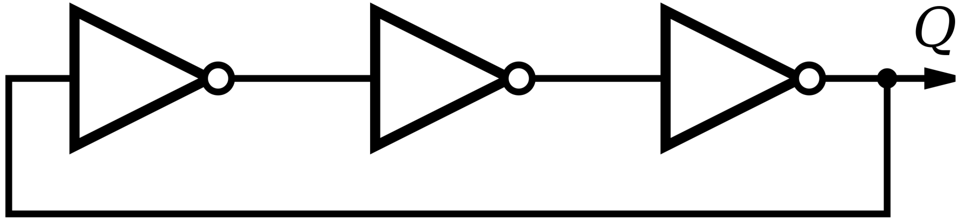
Falling Edge.



We will build a single cycle machine it will complete all the computation in a single cycle

USING RING OSCILLATORS TO GENERATE CLOCKS

A clock is something that produces a periodic signal



Let’s walk through an example assume that Q starts of as 0

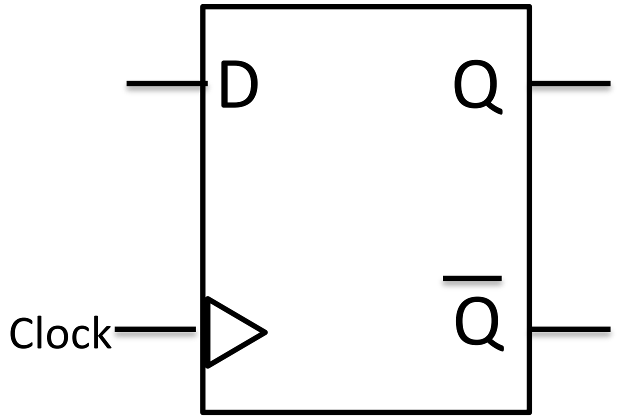
Frequency = 1/(2*t*n)

Where t is time delay of an inverter and n is number of inverters

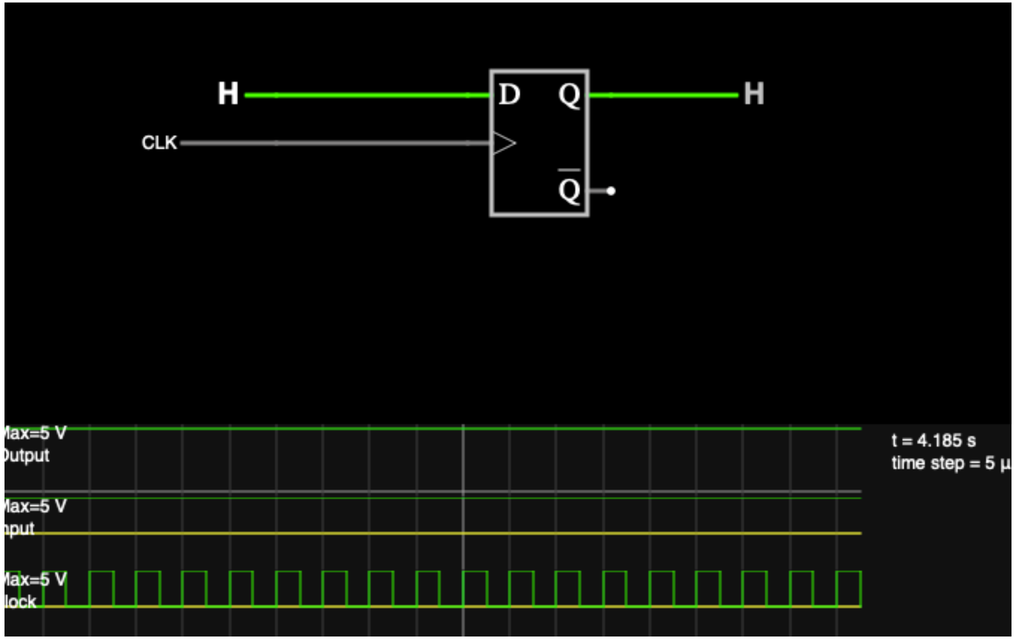
STORING SINGLE

Goal

1. Understand the behavior of a positive edge-triggered D flip-flop.
    - How do we store a bit
    - What happens when the clock changes
    - What does it mean to be a positive edge triggered flip flop
    - What is Q and $\overline{Q}$



BUILT SIMULATOR VERSION

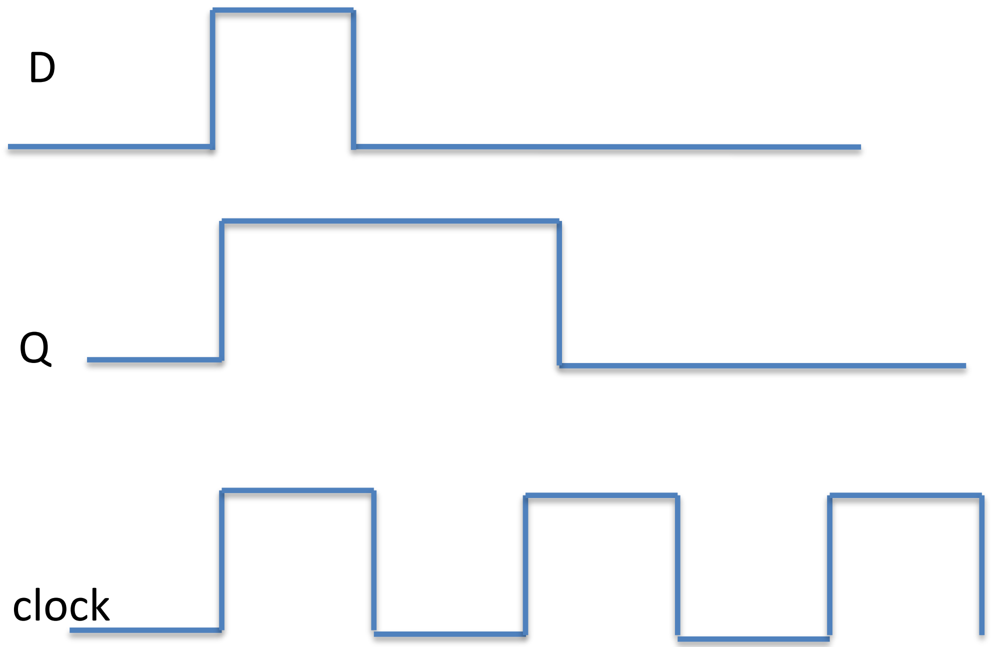


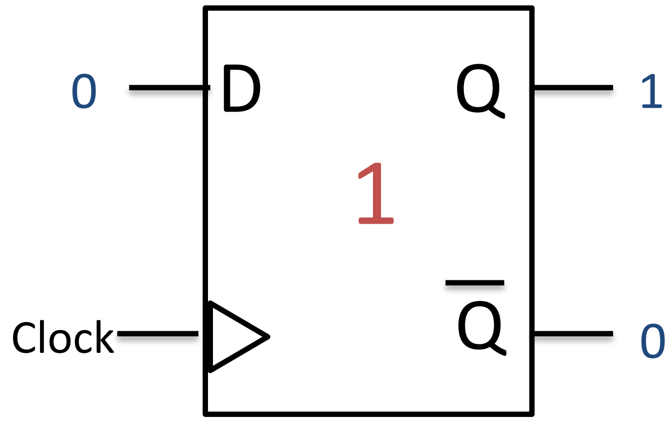
Use this link to experiment with the flipflop during lecture. Try different things and see how it works

https://tinyurl.com/2dhk5kvg

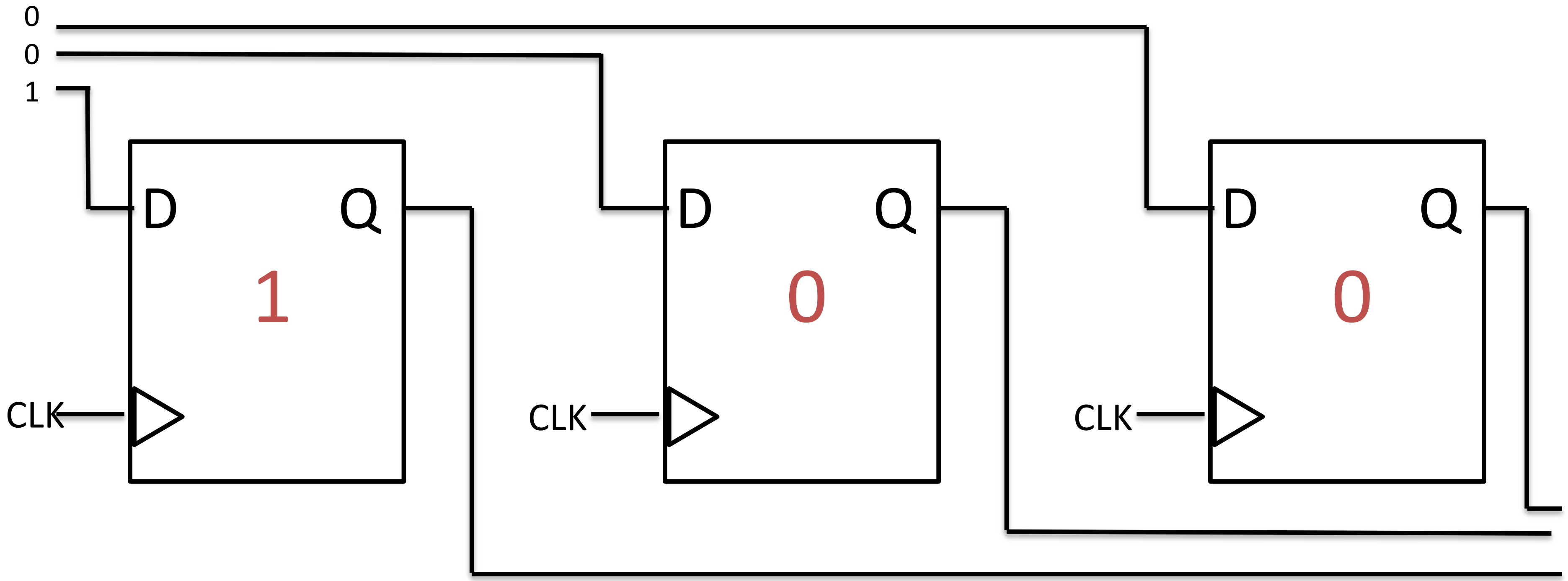
http://www.falstad.com/circuit/circuitjs.html?ctz=CQAgjCAMB0l3BWcMBMcUHYMGZIA4UA2ATmIxAUgoqoQFMBaMMAKDASUPxABZsUQGPD178oFFgHcQXPKIE88VPgMhSZ3HoRGLl2qCwAyvJfN6KzVCADMAhgBsAznWpqASib064vfRAFgPijQSMFIVDAILACygpA6YkrKYlRhLAD21PrKkKSu0BBWIADyAK4ALgAOFRngMil5eeGw8GSECIQo4SABINggAJYAdtXltQLZvLnE+fC5GO2d3QIC-QDG9ulrANYsQA

THE FLIP FLOP HOLD HOLDS THE VALUE FOR A CLOCK CYCLE



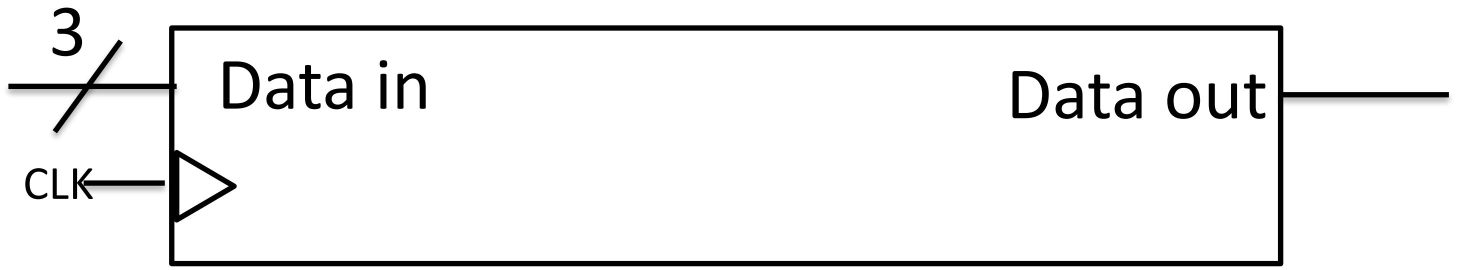


BUILDING A REGISTER FROM FLIP FLOPS



Removed Q (bar) for reability

REGISTER SYMBOLS



3-BIT COUNTER

Let’s put it all together and build a 3-bit counter

Circuit that counts from

000,

001,

010,

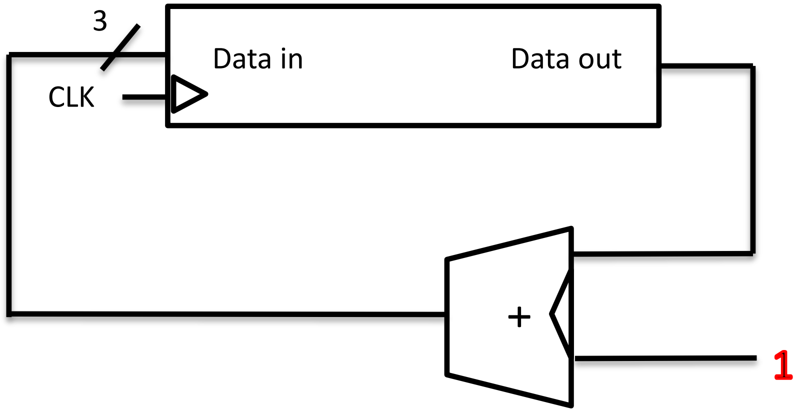
011,

100,

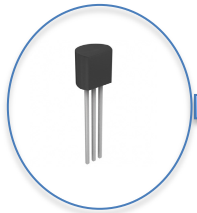
101,

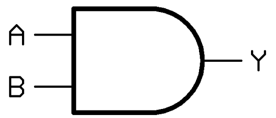
110,

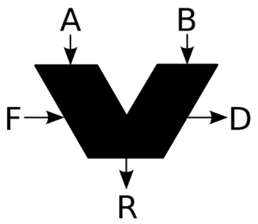
111

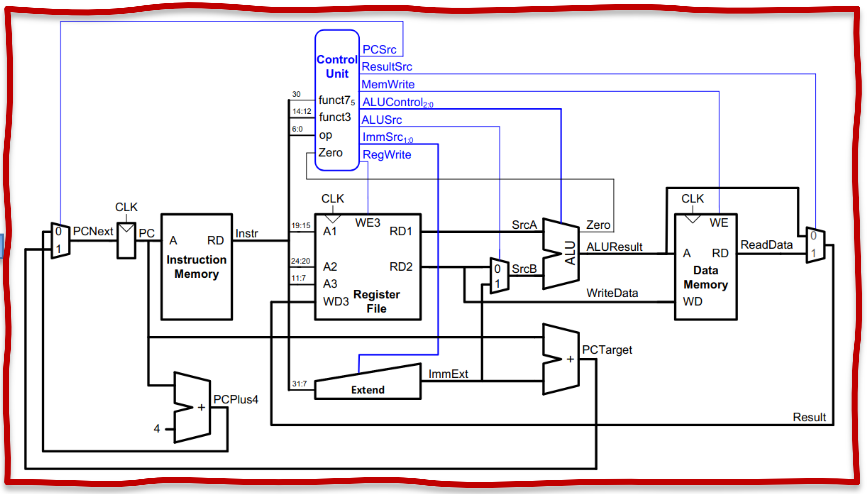


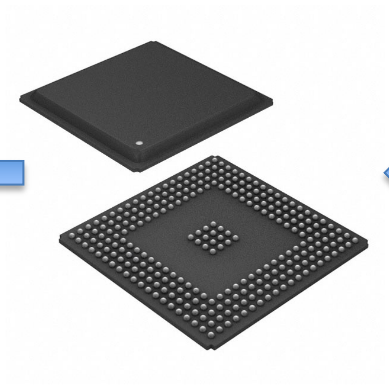
THE MAP (THE MACHINE)





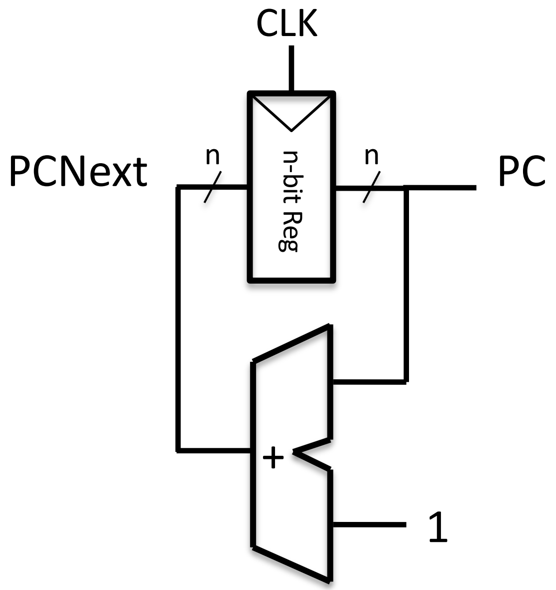






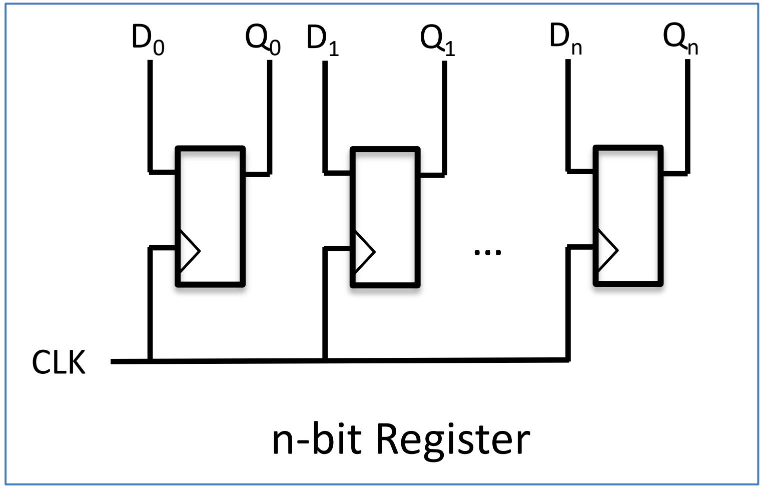
https://github.com/MKrekker/SINGLE-CYCLE-RISC-V

PROGRAM COUNTER

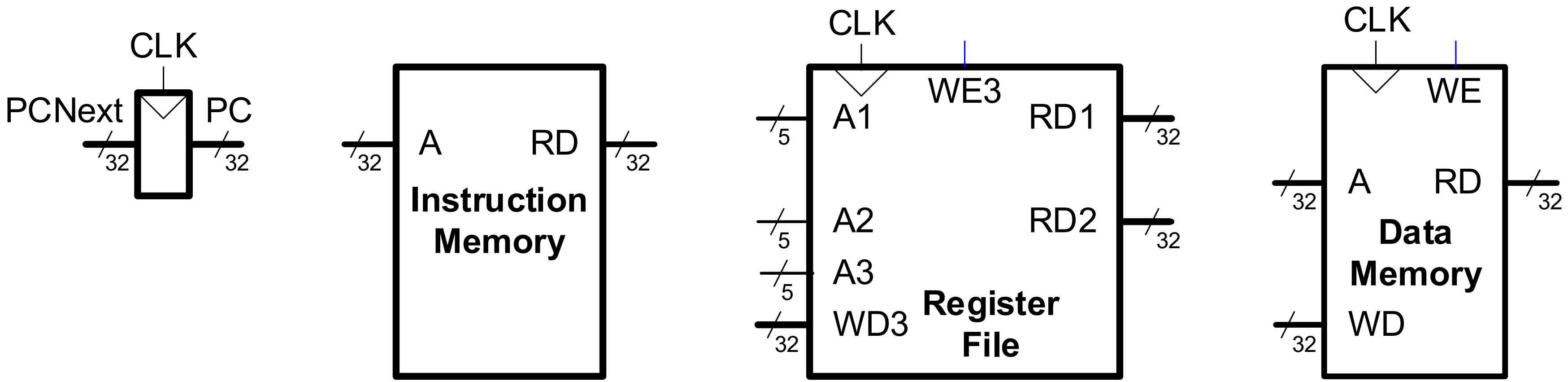


n-bit PC

- To track where we are in a program

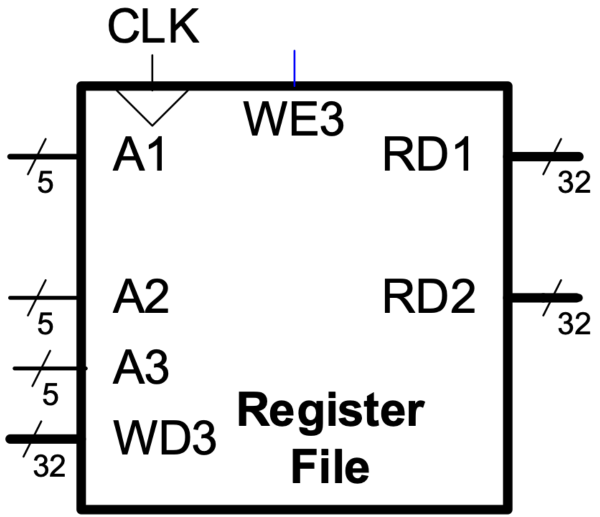


MEMORY COMPONENTS OF A PROCESSOR

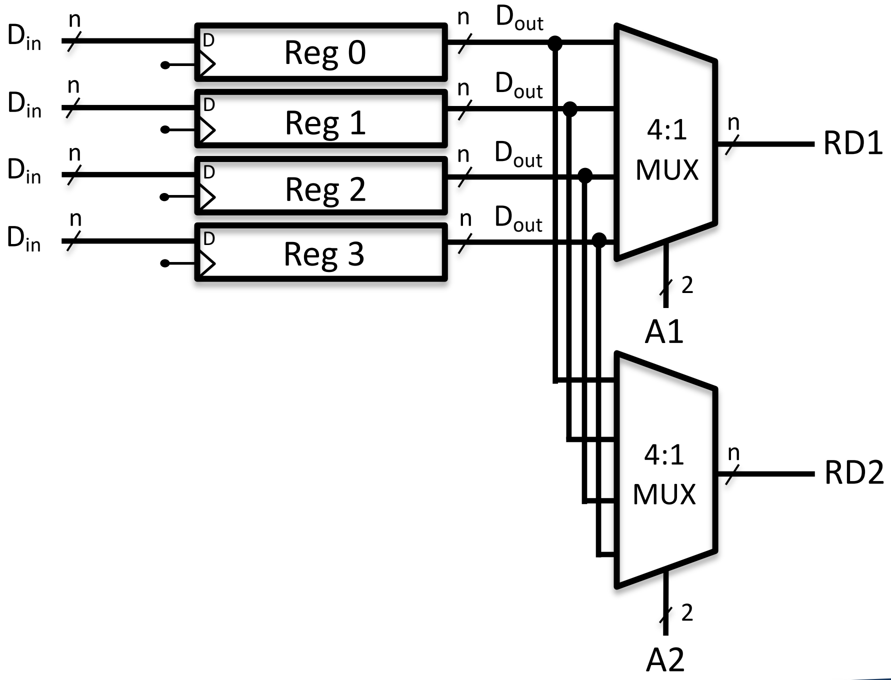


REGISTER FILE

- Temporary storage location
- Stores immediately needed variables
- External interface
    - Addresses: A1, A2, A3
    - Data: RD1, RD2, WD3
    - Enable: WE3
    - Clock: CLK

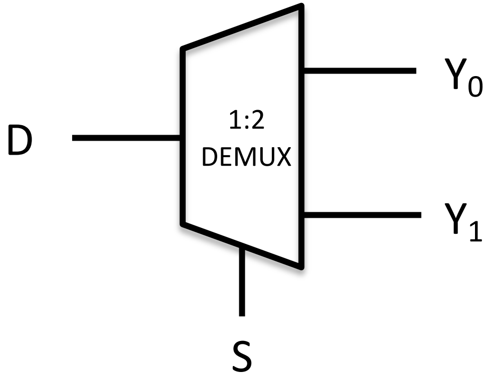


READ FROM A REGISTER FILE



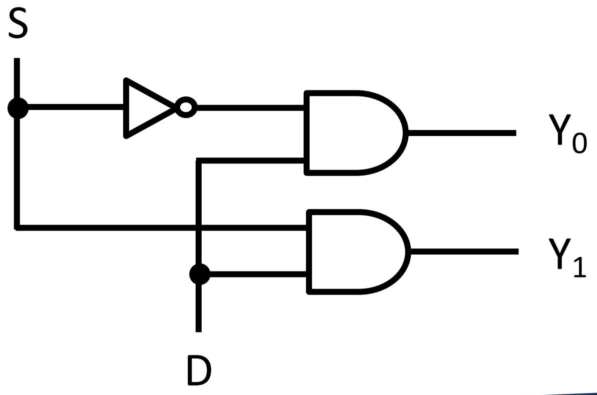
DEMULTIPLEXER (DEMUX)

Example: **1:2 Demux**

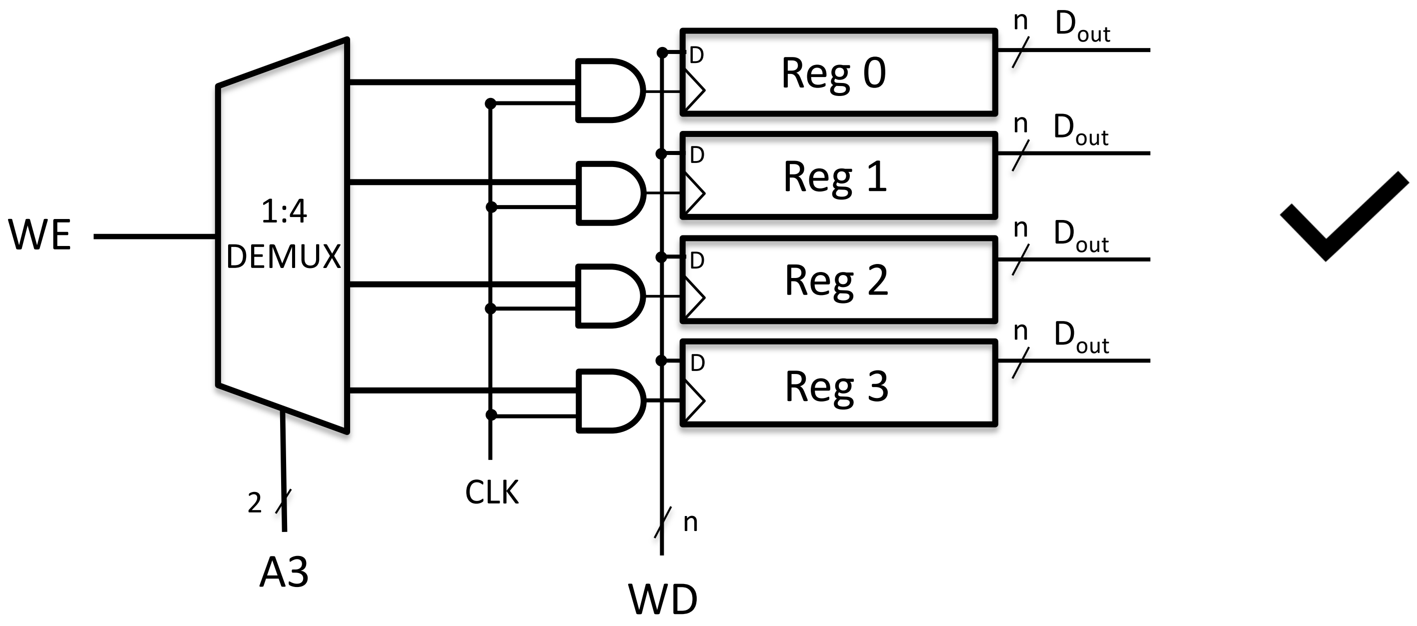


S	**Y0**	**Y1**
0	**D**	0
1	0	**D**

- Connects one input to one of the **N** outputs
- **Select** input is **$\log_2 N$ bits** – control input



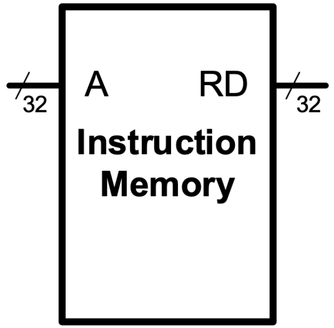
WRITE TO A REGISTER FILE



INSTRUCTION MEMORY

- Stores the program

➢ Read data (RD) for a given address (A)

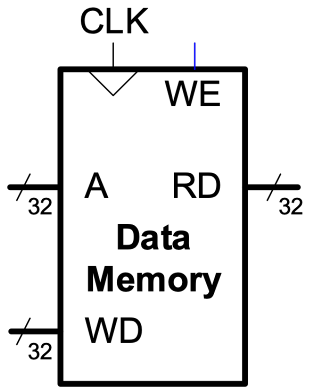


For this class, we will assume we cannot write to Instruction Memory.

DATA MEMORY

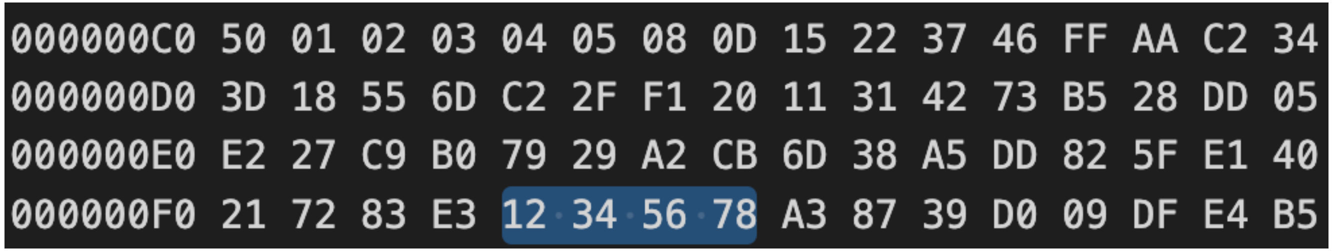
- Contains data needed by the program

- ➢ Read data (RD) from a given address (A)
- ➢ Write data (WD) to a given address (A)



EXERCISE

Answer:



```
000000C0 50 01 02 03 04 05 08 0D 15 22 37 46 FF AA C2 34
000000D0 3D 18 55 6D C2 2F F1 20 11 31 42 73 B5 28 DD 05
000000E0 E2 27 C9 B0 79 29 A2 CB 6D 38 A5 DD 82 5F E1 40
000000F0 21 72 83 E3 12 34 56 78 A3 87 39 D0 09 DF E4 B5
```

MEMORY HIERARCHY

Levels in Speed and Capacity of the Memory

Higher levels larger and slower memory

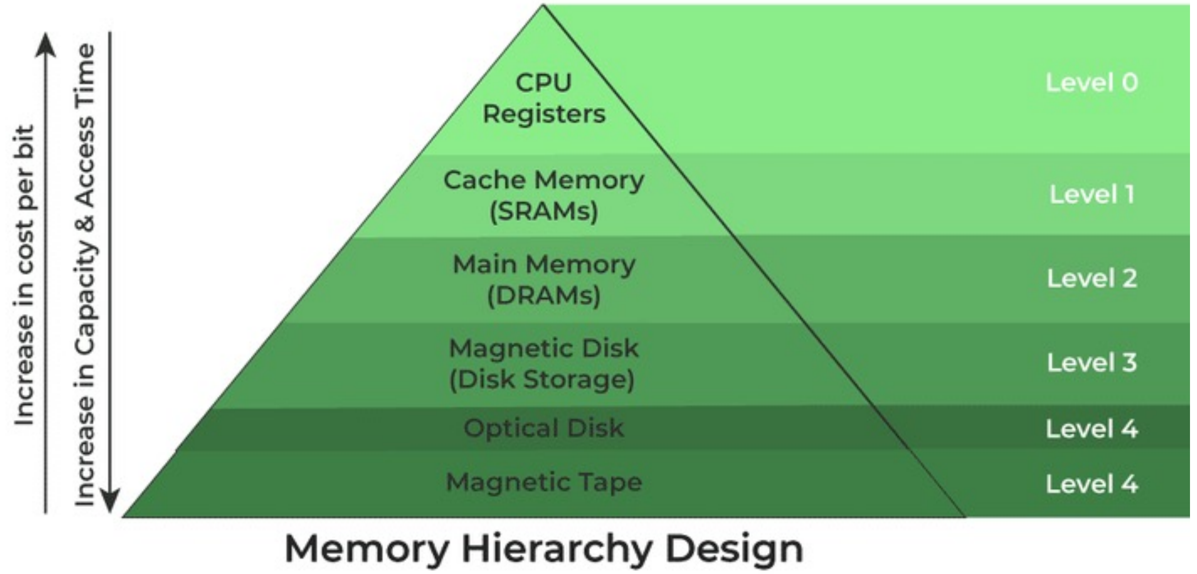
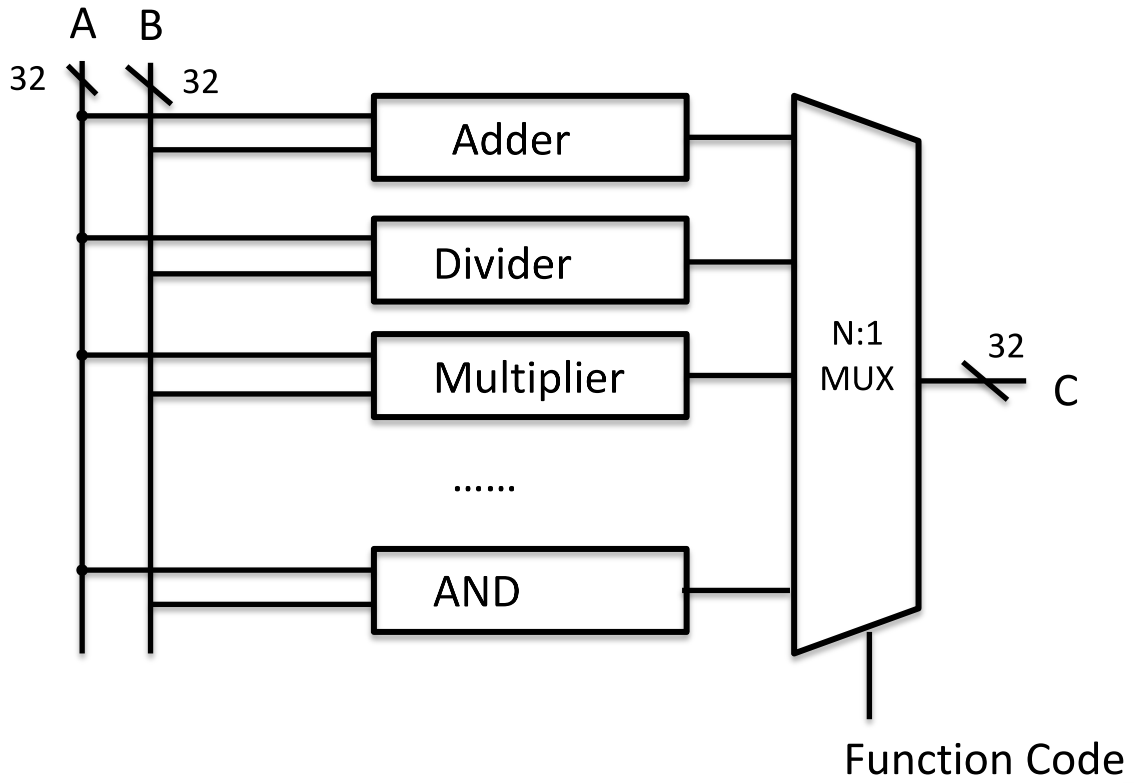
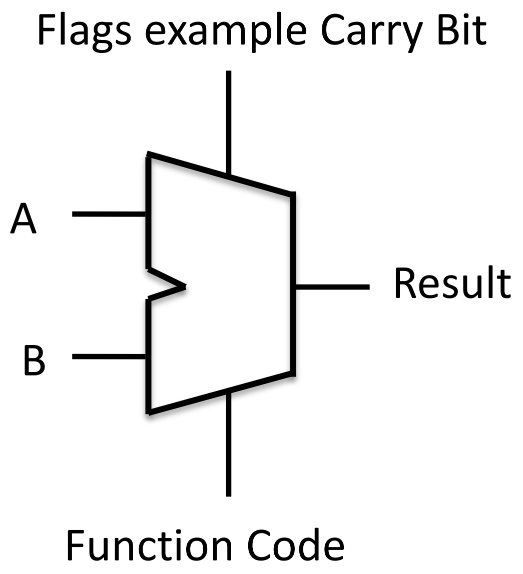


Figure from: https://www.geeksforgeeks.org/memory-hierarchy-design-and-its-characteristics/

ARITHMETIC LOGIC UNIT



ALU SYMBOL AND INPUTS



TOY ISA AND PROCESS

VERSION 0.1

WE’LL MAKE IT BETTER DON’T WORRY

TINY PROGRAM TO ASSEMBLY

```
m = 4
x = 2
b = -1
y = m*x*b
```

Looks like we need two types on instructions

1. An instruction to load values
2. An instruction to computation (multiply)

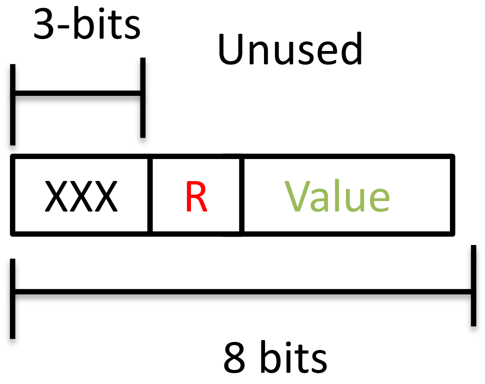
LET'S DECIDE HOW WE ARE GOING TO LAYOUT OUR BITS

1. An instruction to load values into **Registers**

m = 3
x = 2
b = -1

→

R0 =3
R1 = 2
R2 = -1



We just make these zeros
XXX = 000

NOW LET’S TRANSLATE OUT PROGRAM TO ONES AND ZERO

1. An instruction to load values into **Registers**

		XXX	R	Value	
m = 4	R0 = 3	000	00	011	0x03
x = 2	R1 = 2	000	01	010	0x0A
b = -1	R2 = -1	000	10	111	0x17

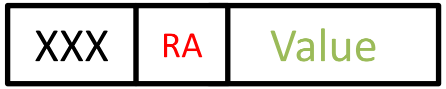
FULL ISA

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

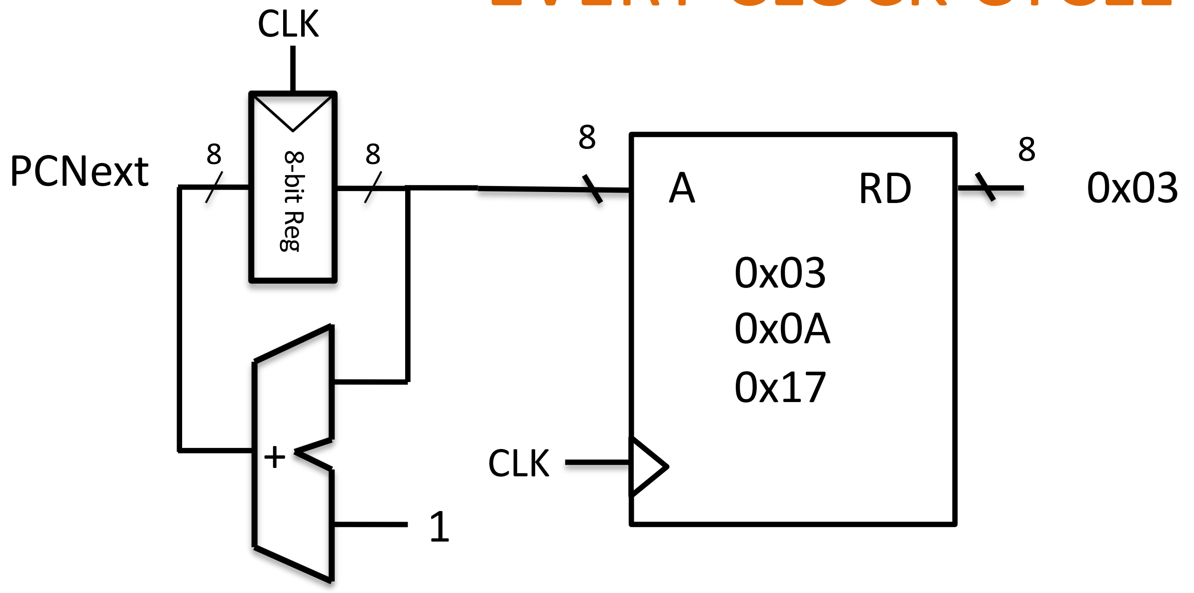
UNIVERSITY of VIRGINIA | ENGINEERING

GREAT WE HAVE OUR FIRST INSTRUCTION

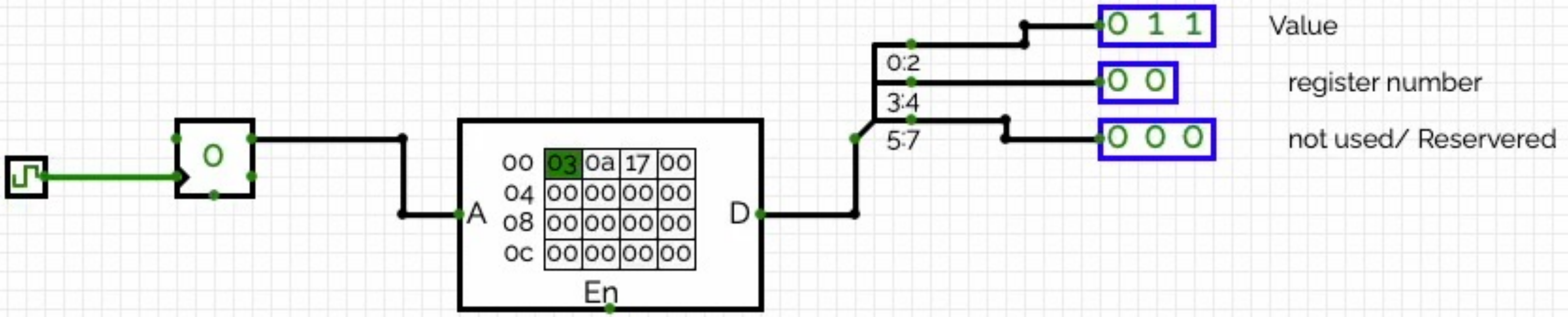


RA = Value

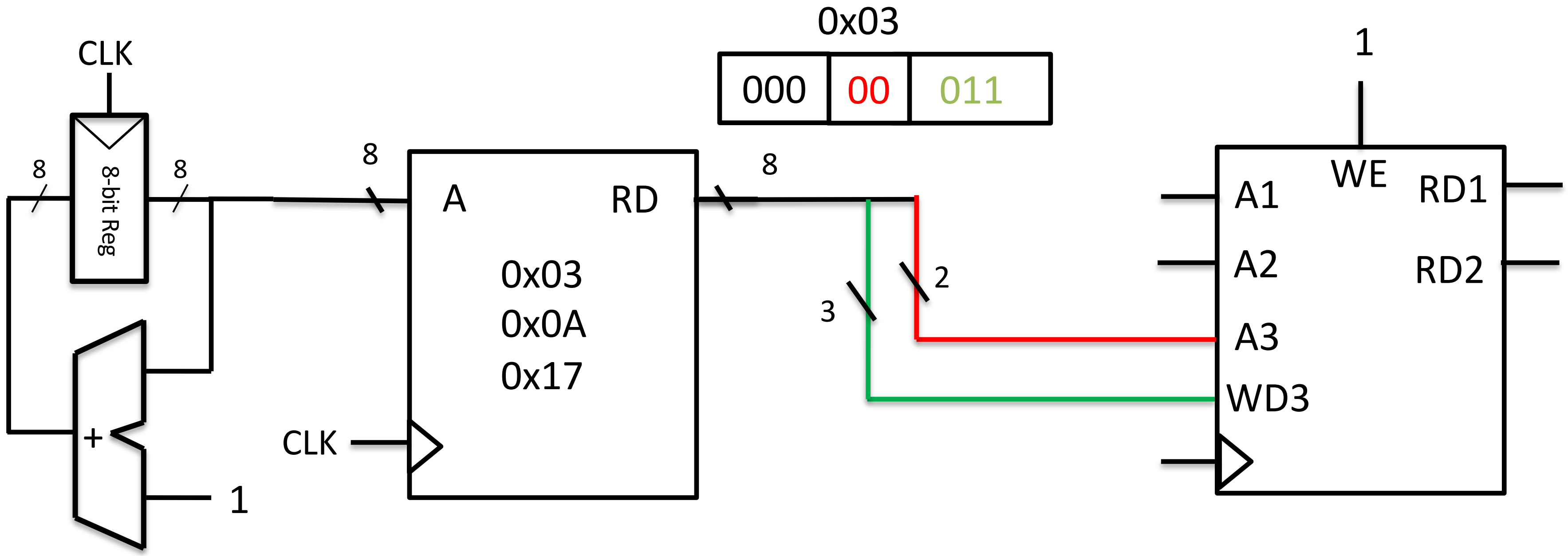
AUTOMATICALLY FETCH A NEW INSTRUCTION EVERY CLOCK CYCLE



n-bit PC



0
00
04
08
0c
03 0a 17 00
00 00 00 00
00 00 00 00
00 00 00 00
A
D
En
0:2
3:4
5:7
0 1 1
0 0
0 0 0
Value
register number
not used/ Reservered



8-bit PC

Our program would have loaded values into the register file

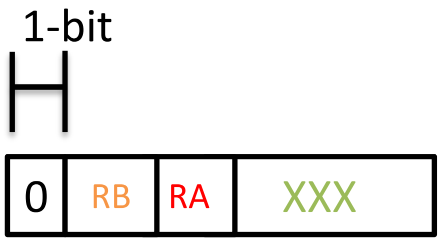
R0 =3
R1 = 2
R2 = -1

OPCODE

Multiply **Registers**

```
y = m*x*b
```

R0 *= R1
R0 *= R2



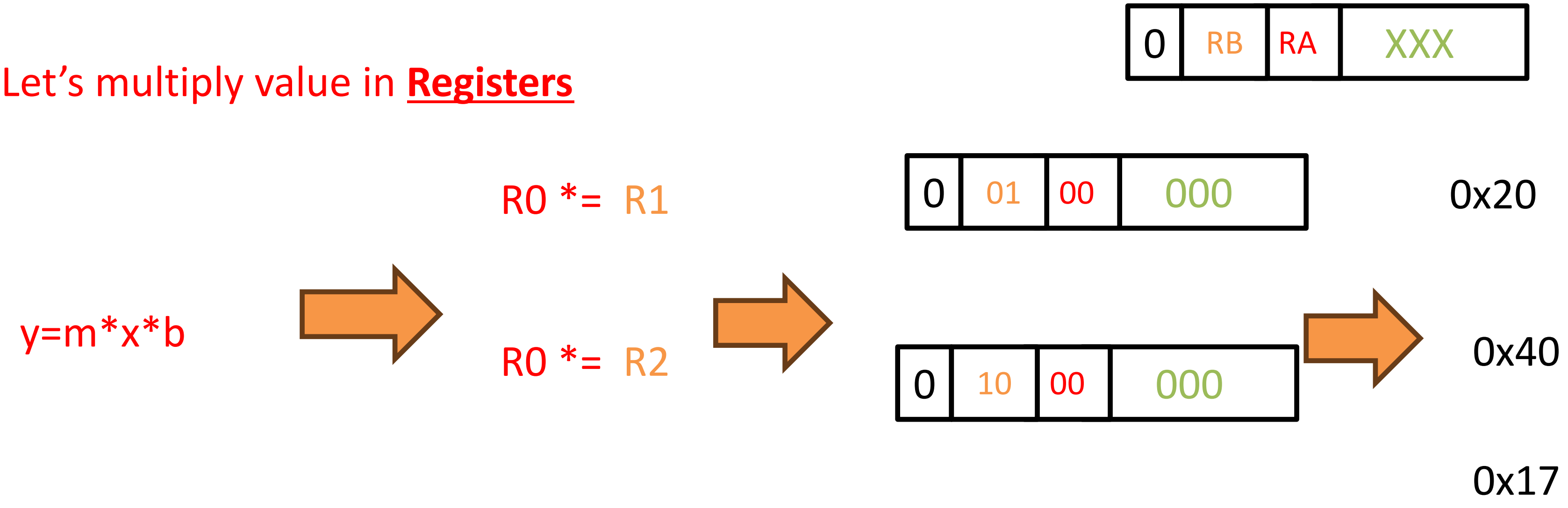
0 --> Multiply
1 --> Save Value to register

Finally, we need an opcode to distinguish our load instruction from our multiple

ENCODING

Let's multiply value in **Registers**

0	RB	RA	XXX



0	01	00	000

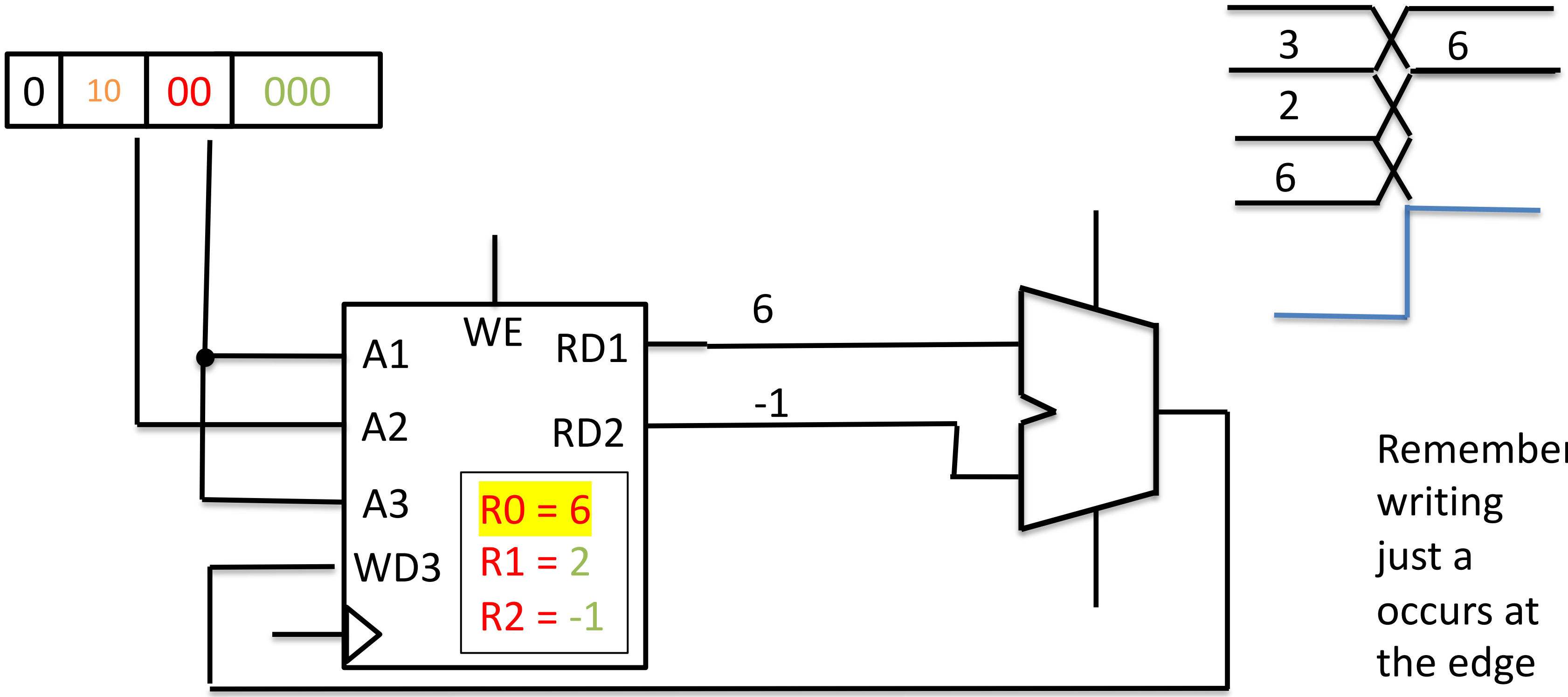
0x20

0	10	00	000

0x40

0x17

BUILDING MACHINE TO COMPUTE THIS



Remember writing just a occurs at the edge

NOTE WE ALSO NEED TO UPDATE THE ENCODING OF OUR LOADS

1. An instruction to load values into **Registers**

		1	RB	RA	Value	
m = 4	R0 = 3	1	00	00	011	0x83
x = 2	R1 = 2	1	00	01	010	0x8A
b = -1	R2 = -1	1	00	10	111	0x97

1. An instruction to load values into **Registers**

1	RB	RA	Value

m = 4 → R0 = 3

x = 2 → R1 = 2

b = -1 → R2 = -1

1	00	00	011	0x83
1	00	01	010	0x8A
1	00	10	111	0x97

Let's multiply value in **Registers**

0	RB	RA	XXX

y=m*x*b → R0 *= R1, R0 *= R2

0	01	00	000	0x20
0	10	00	000	0x40

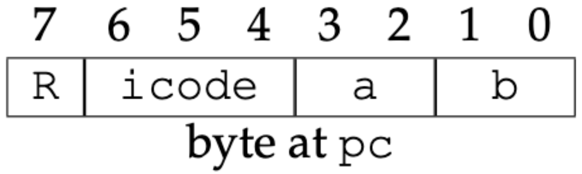
0x17

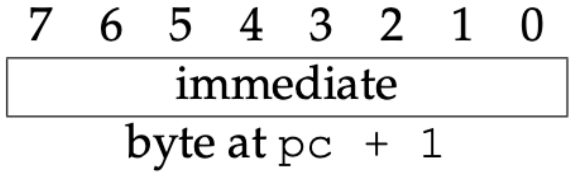
INSTEAD GOING INSTRUCTION BY INSTRUCTION LET’S DESIGN THE ISA AND THE MACHINE

TOY INSTRUCTION SET ARCHITECTURE (ISA)

The ISA defines:

1. Instructions and their layout
2. Data types
3. Registers we'll have





How instructions are laid out in our ISA

ENCODING OUR FIRST INSTRUCTION

Try to encode the following instruction R0 = R1

7	6	5	4	3	2	1	0
R	icode			a		b	

byte at pc

7	6	5	4	3	2	1	0
immediate							

byte at pc + 1

icode 0 RA = RB

7	6	5	4	3	2	1	0
0	0 0 0			0 0		0 1	

Not used
This instruction is not using a value

icode	b	Behavior
0		rA=rB
1		rA+=rB
2		rA&=rB
6	0	rA=read from memory at pc + 1 Also written as rA = M[pc+1]

```
R0 = 8
R1 = -1
R0 += R1
```

7 6 5 4 3 2 1 0

R	icode	a	b
0	110	00	00

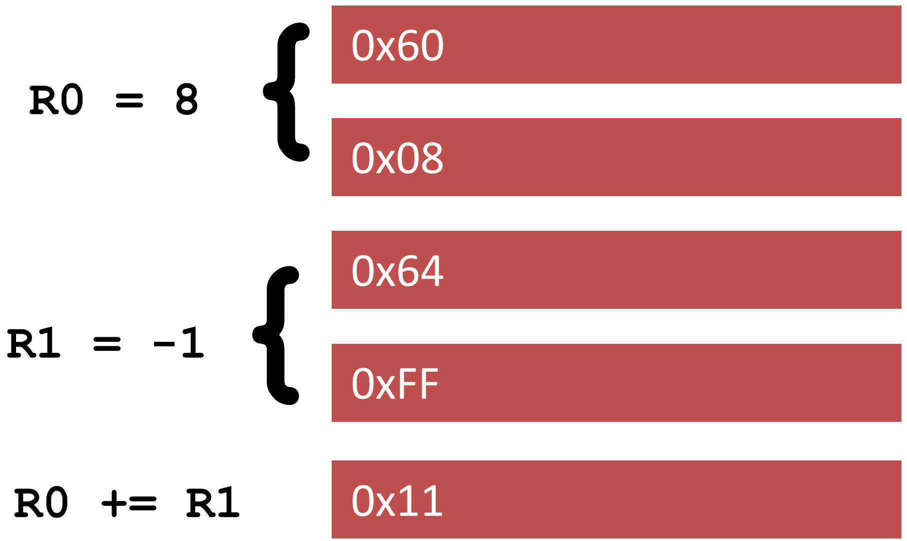
0x60

7 6 5 4 3 2 1 0

immediate
0 0 0 0 1 0 0 0

0x08

Notice that we have to increment the Program Counter by **two** for these instructions. Because they are two bytes long while the other instructions are only 1 byte

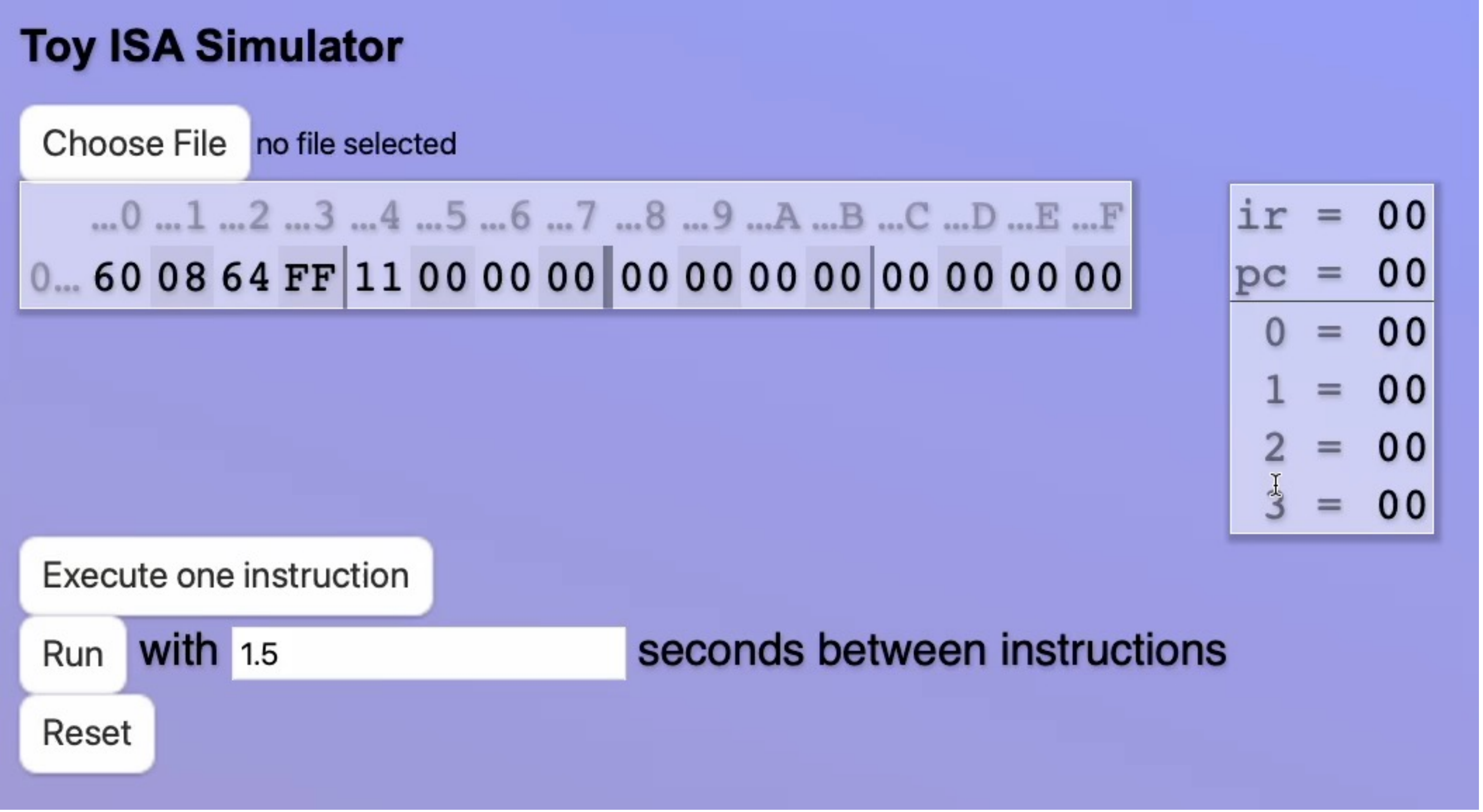


THE FLOW

```
x = 8
y = -1
z = x + y
```

```
R0 = 8
R1 = -1
R0 += R1
```

```
0x60 0x08 0x64 0xFF 0x11
```



Toy ISA Simulator

Choose File no file selected

	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	60	08	64	FF	11	00	00	00	00	00	00	00	00	00	00	00

```
ir = 00
pc = 00
 0 = 00
 1 = 00
 2 = 00
 3 = 00
```

Execute one instruction

Run with 1.5 seconds between instructions

Reset

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

We'll give the full description of ISA at the begin of every exam. In fact this a picture of what we will give you.

READ FROM MEMORY ADDRESS STORED IN RB

Registers

R0	X
R1	X
R2	X
R3	X
PC	00

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	64	23	31													
10																
20				FF												
30																

What are the values of R0 and R1. Once program completes?

REGISTER SPILLING

Because we have a limited number of registers, we can't store all variables in registers, so we must store some in memory and read them into a register when we need them.

Here is the strategy

1. Read the register value to a predetermined location in memory.
2. Use the register
3. Write the register value back to memory, so that it can be used to store something else

Architecture	8 bit	32 bit	64 bit
ARM	X	15	31
Intel x86	X	8	16
Toy ISA	4	X	X

LET'S CALCULATE WHERE TO JUMP TO

Memory Address		Size of Instruction
0x00	`R0 = M[0x20]`	2 Bytes
0x02	`R1 = 0x07`	2 Bytes
0x04	`If R0 <= 0 set PC= R1`	1 Byte
0x05	`R0 += 1`	2 Bytes
0x07	`R0 &= 2`	2 Bytes

So what address do we want R1 to be?

WRITE A LOOP

First, rewrite as a do-while loop. (This due to limitation in Toy ISA) reasons will be clear later.

```
x = 2
for (i = 0; i < 5; i++){
    x+=1
}
```

```
x = 2
i = 0
do{
    x+=1
    i++
}while(i<5)
```

WRITE A LOOP

```
x = 2
i = 0
do{
    x+=1
    i++
}while(i<5)
```

```
R0 = 2
R1 = 0
R2 = PC
R0 += 1
R1 += 1
R3 = R1
R3+= -5
if R3 <=0 then PC = R2
```

But wait is that correct? Translating the condition can be tricky

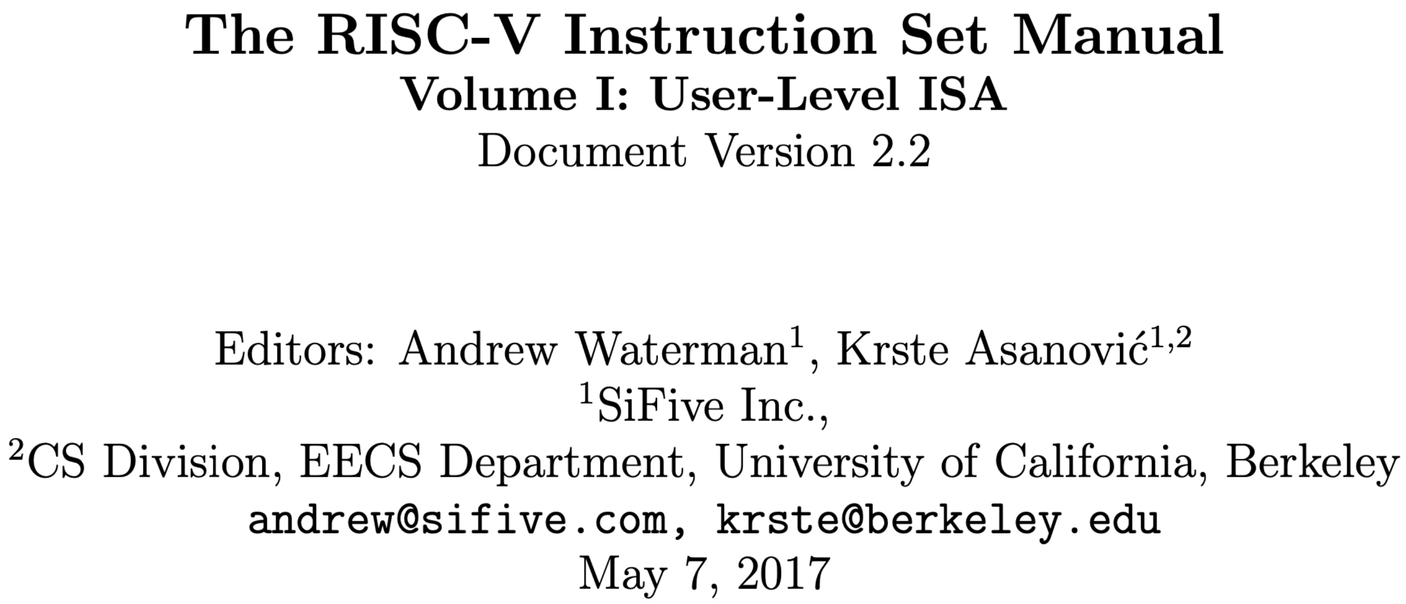
WRITE A LOOP

```
x = 2
i = 0
do{
    x+=1
    i++
}while(i<5)
```

-3 , -2, -1, 0, 1 (five times)

`R0 = 2`	0x60 02
`R1 = 0`	0x64 0x00
`R2 = PC`	0x5B
`R0 += 1`	0x61 0x01
`R1 += 1`	0x65 0x01
`R3 = R1`	0x0D
`R3+= -4`	0x6D 0xFC
`if R3 <=0 then PC = R2`	0x7E

SOME PERPECTIVE (RISC-V)



The RISC-V Instruction Set Manual
Volume I: User-Level ISA
Document Version 2.2

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
`andrew@sifive.com, krste@berkeley.edu`
May 7, 2017

Available at: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

RISC VS CISC

RISC-V ADD

https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html#addi

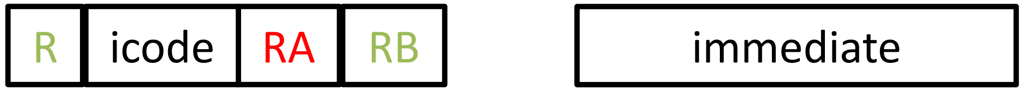
X86 Add

https://www.felixcloutier.com/x86/add

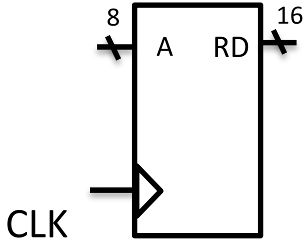
Detailed Data Sheet: https://www.elsevier.com/__data/assets/pdf_file/0011/297533/RISC-V-Reference-Data.pdf

NOW THAT WE HAVE OUR ISA LET'S DESIGN THE MACHINE

INSTRUCTION MEMORY AND INSTRUCTION REGISTER



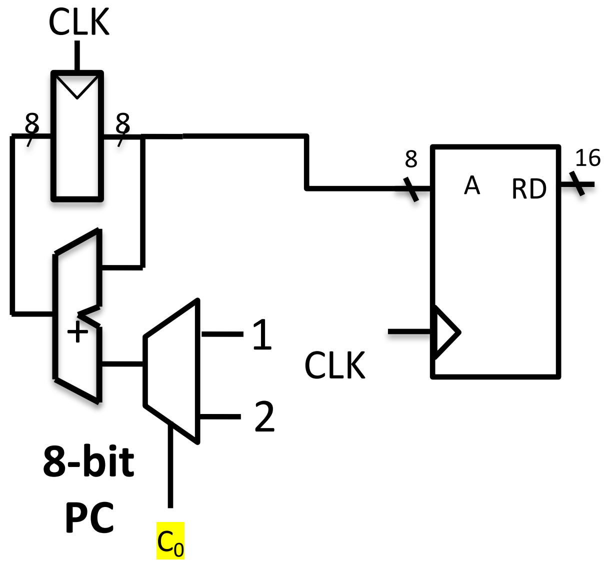
Instruction register (IR)



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

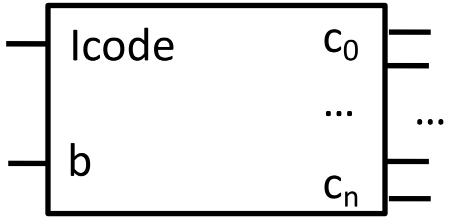


We'll add a mux that will select passing one to adder or two.

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

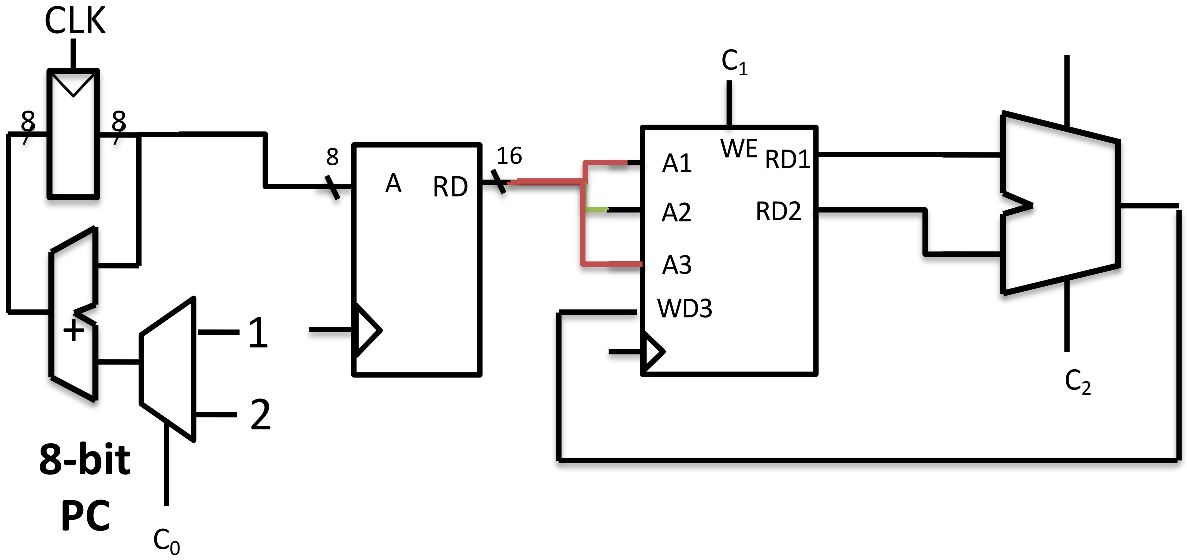
HARDWIRED CONTROL UNIT

icode	b	C_0	….	C_n
6	x	1		
….				



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

R	icode	RA	RB

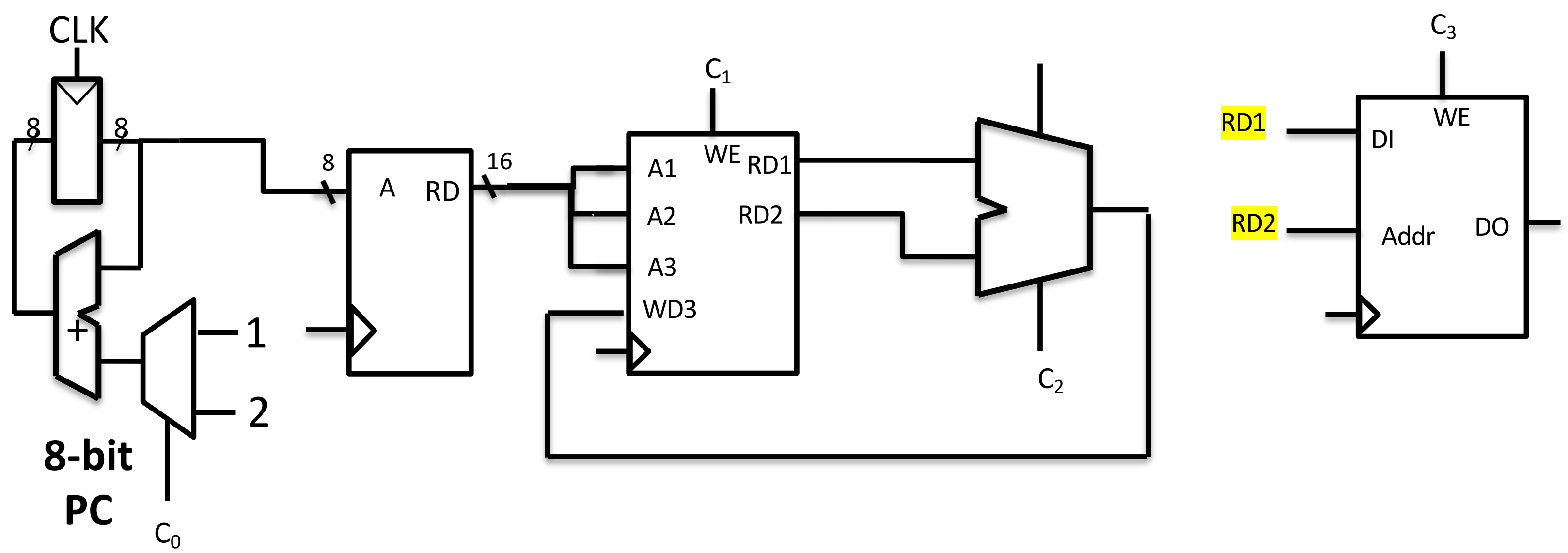


icode	b	meaning
0		`rA = rB`
1		`rA += rB`
2		`rA &= rB`
3		**rA** = read from memory at address **rB**
4		write **rA** to memory at address **rB**
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`
6	0	**rA** = read from memory at **pc + 1**
	1	**rA** += read from memory at **pc + 1**
	2	**rA** &= 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

UNIVERSITY of VIRGINIA ENGINEERING

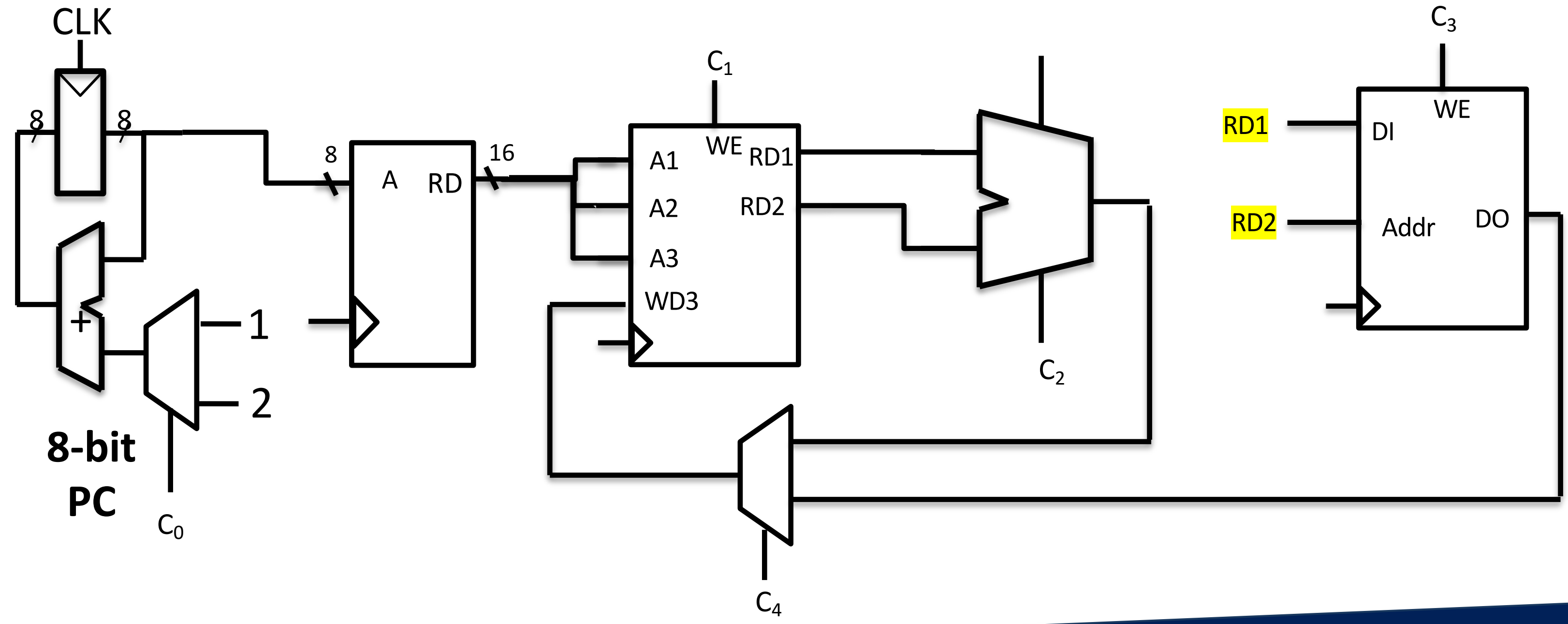
3	`rA` = read from memory at address `rB`
4	write `rA` to memory at address `rB`

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



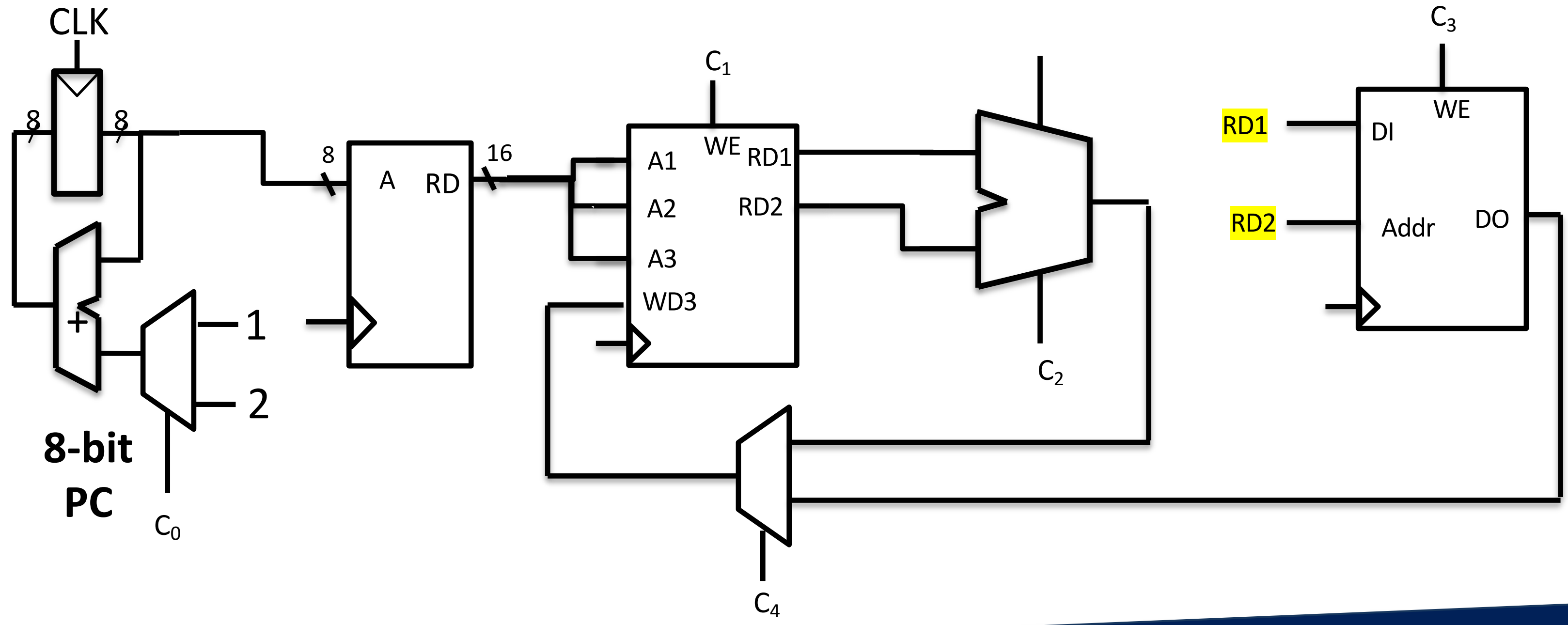
3	**rA** = read from memory at address **rB**
4	write **rA** to memory at address **rB**

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

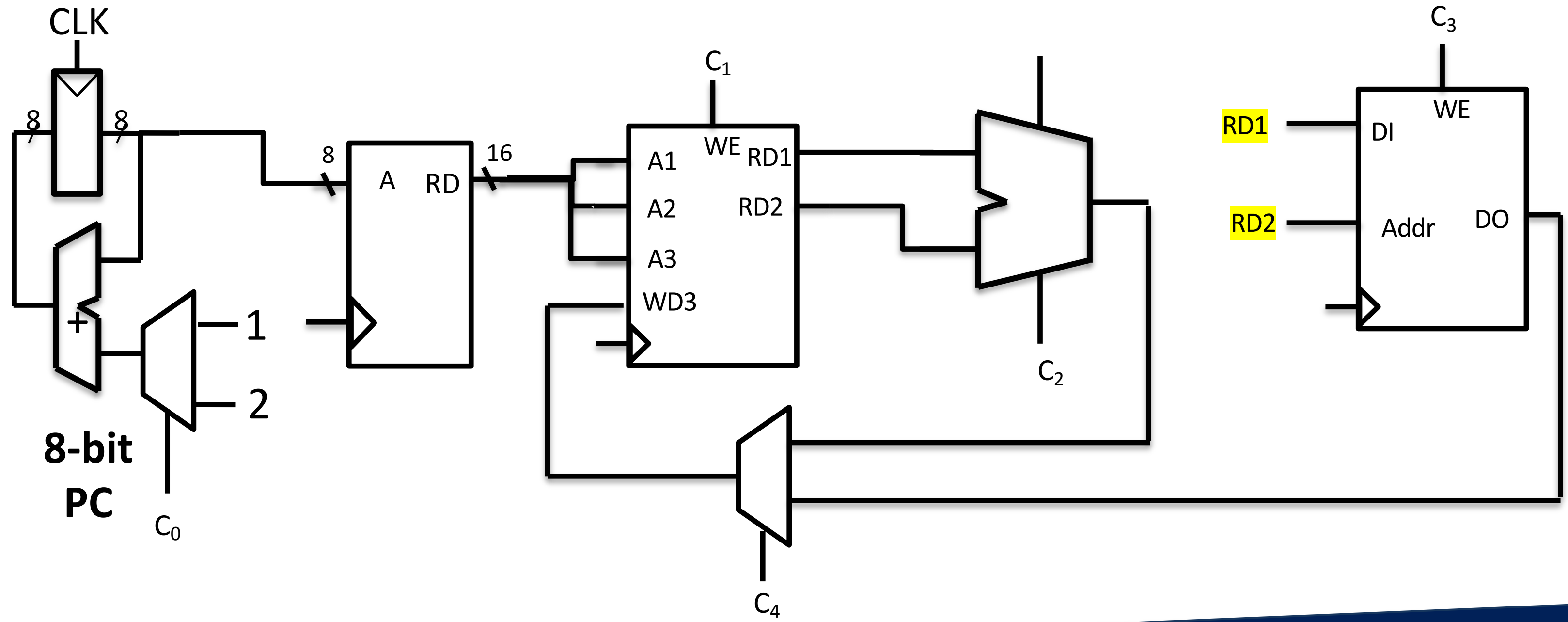


icode	b	meaning
0		`rA = rB`
1		`rA += rB`
2		`rA &= rB`
3		**rA** = read from memory at address **rB**
4		write **rA** to memory at address **rB**
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`
6	0	**rA** = read from memory at **pc + 1**
	1	**rA** += read from memory at **pc + 1**
	2	**rA** &= 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

UNIVERSITY of VIRGINIA ENGINEERING

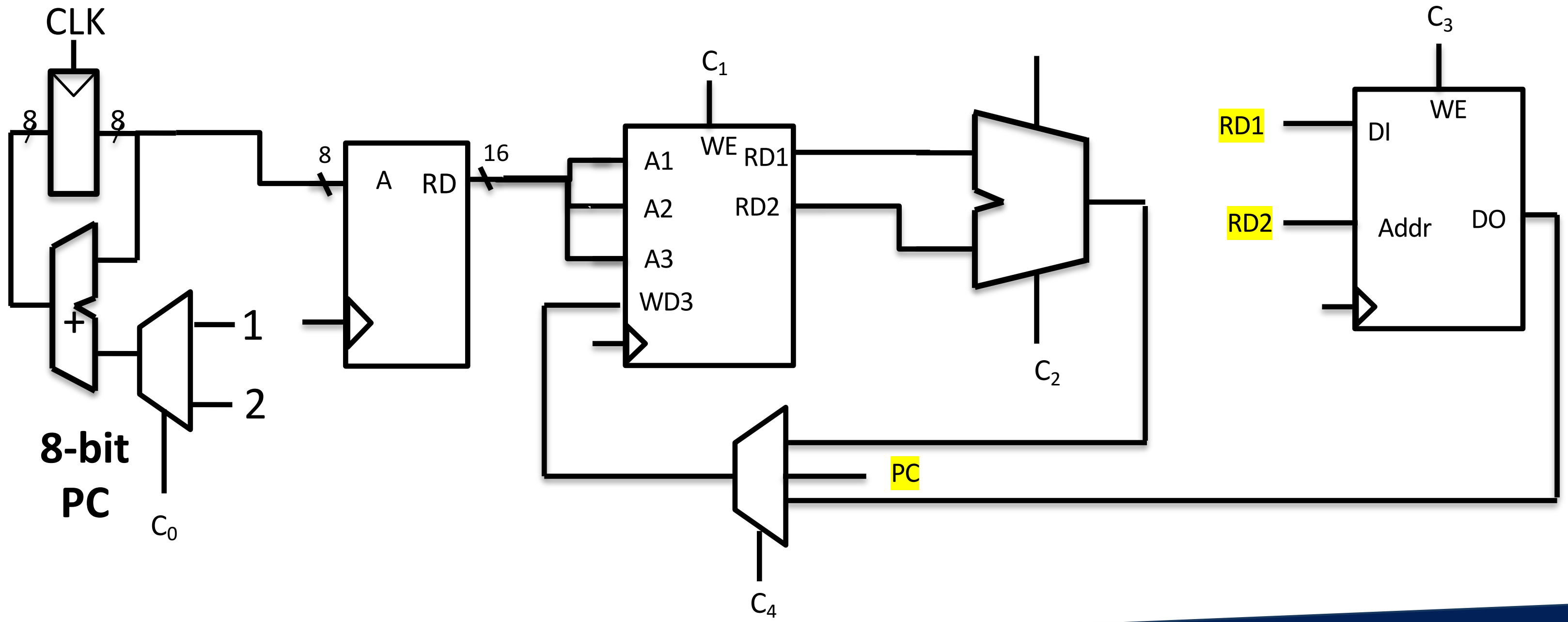
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`

Draw out the flow here



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

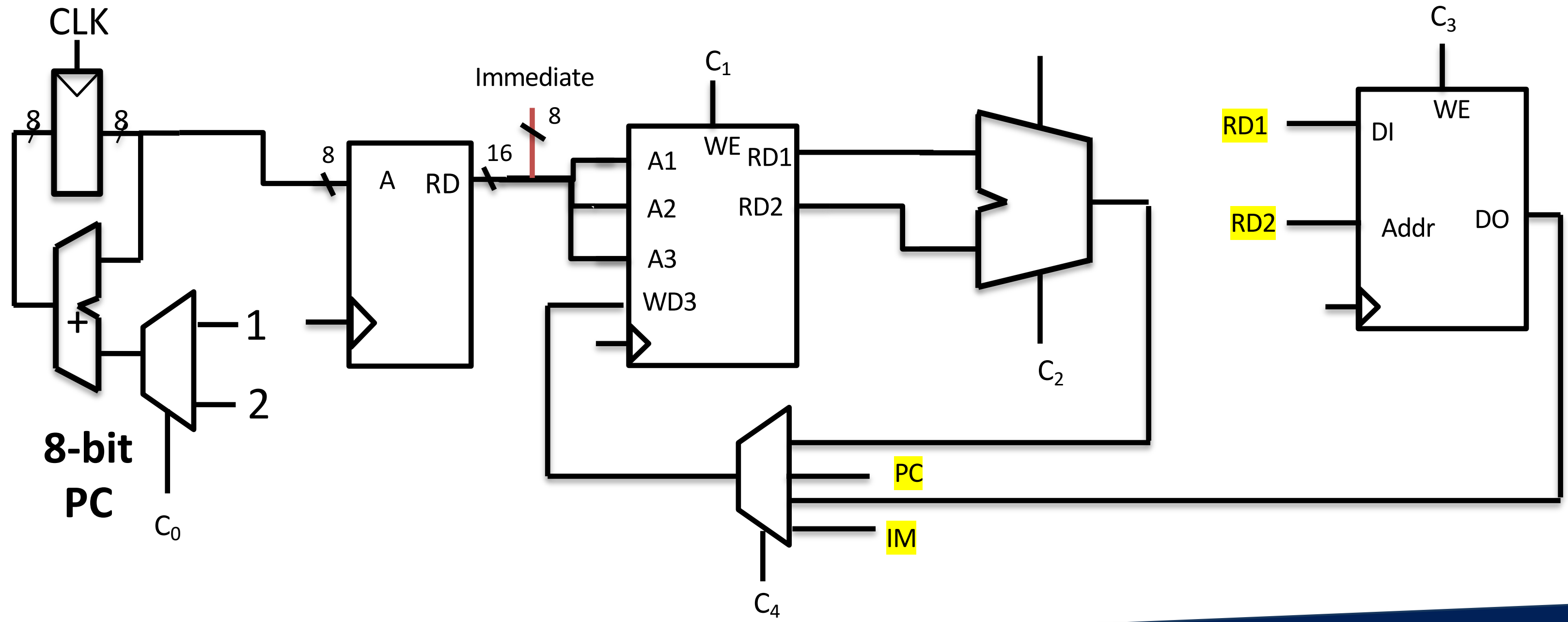
Changed it to just be the label



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`

Walk through the flow of an example instruction

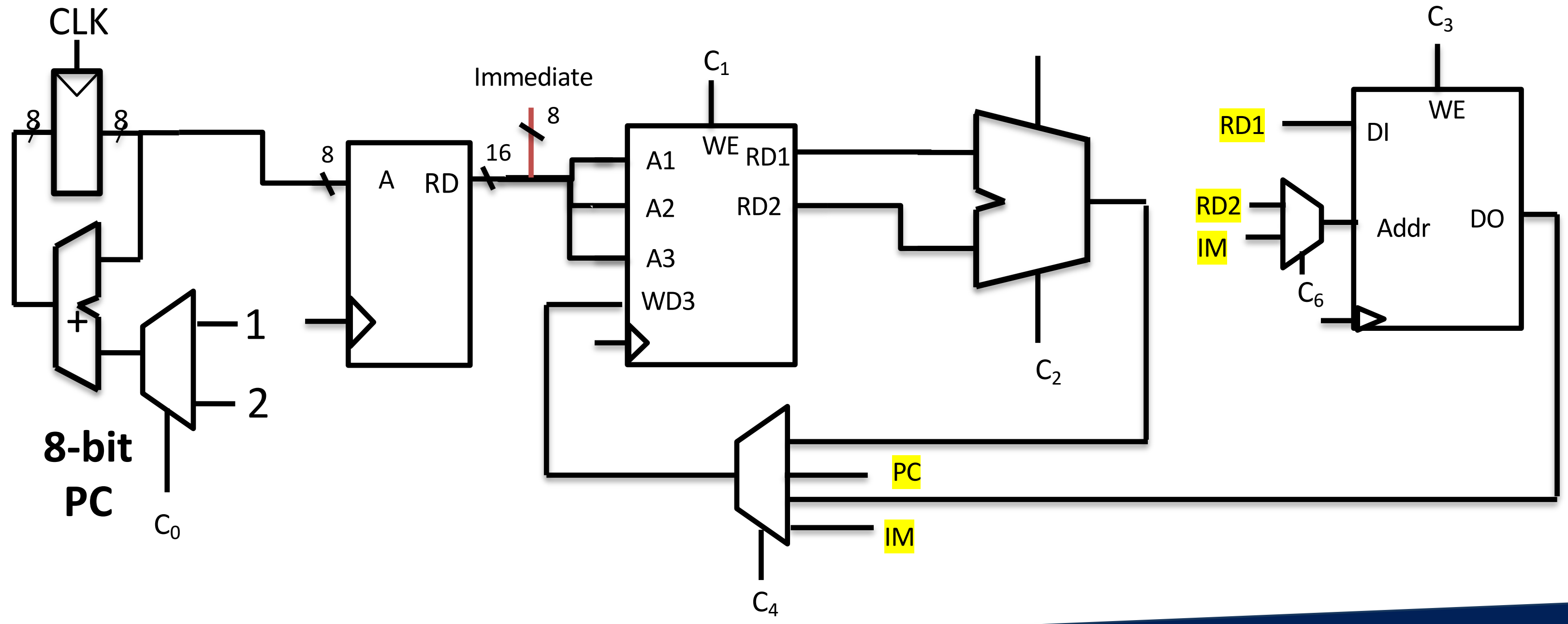
R	icode	RA	RB



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`

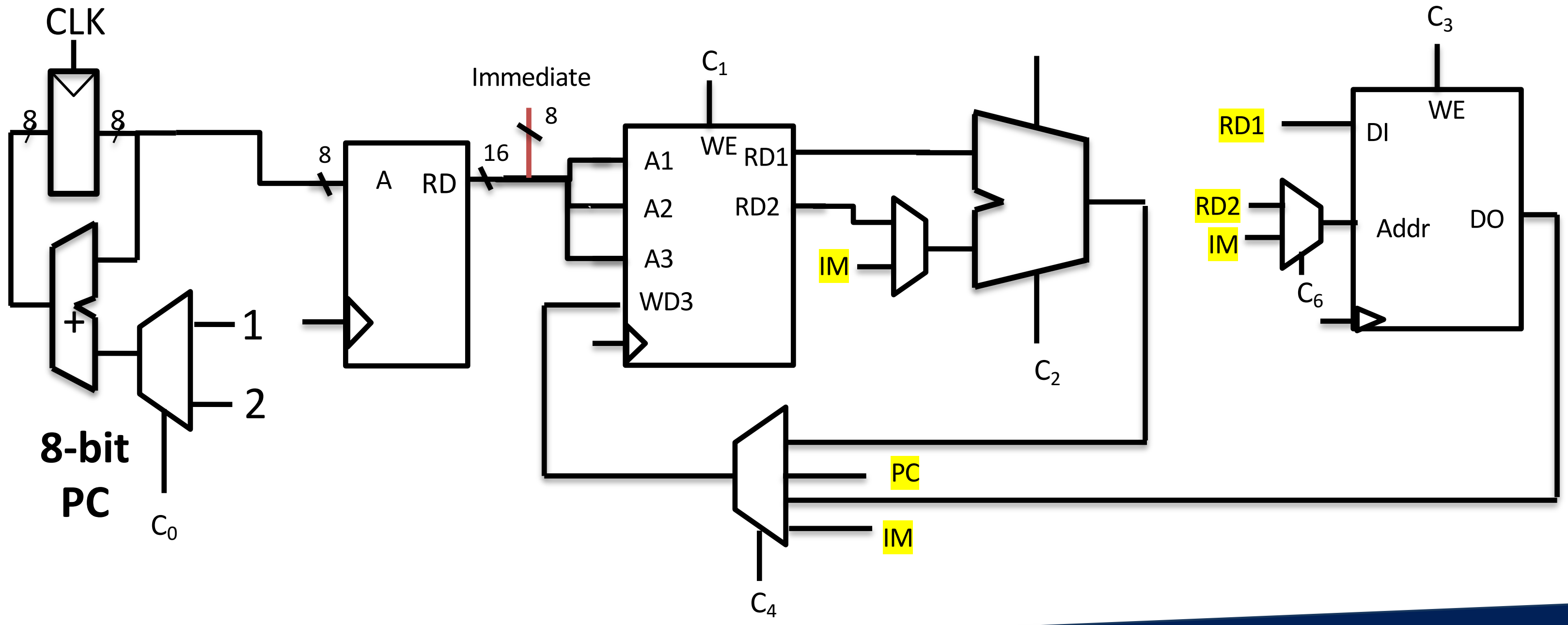
Again we just need a mux



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**

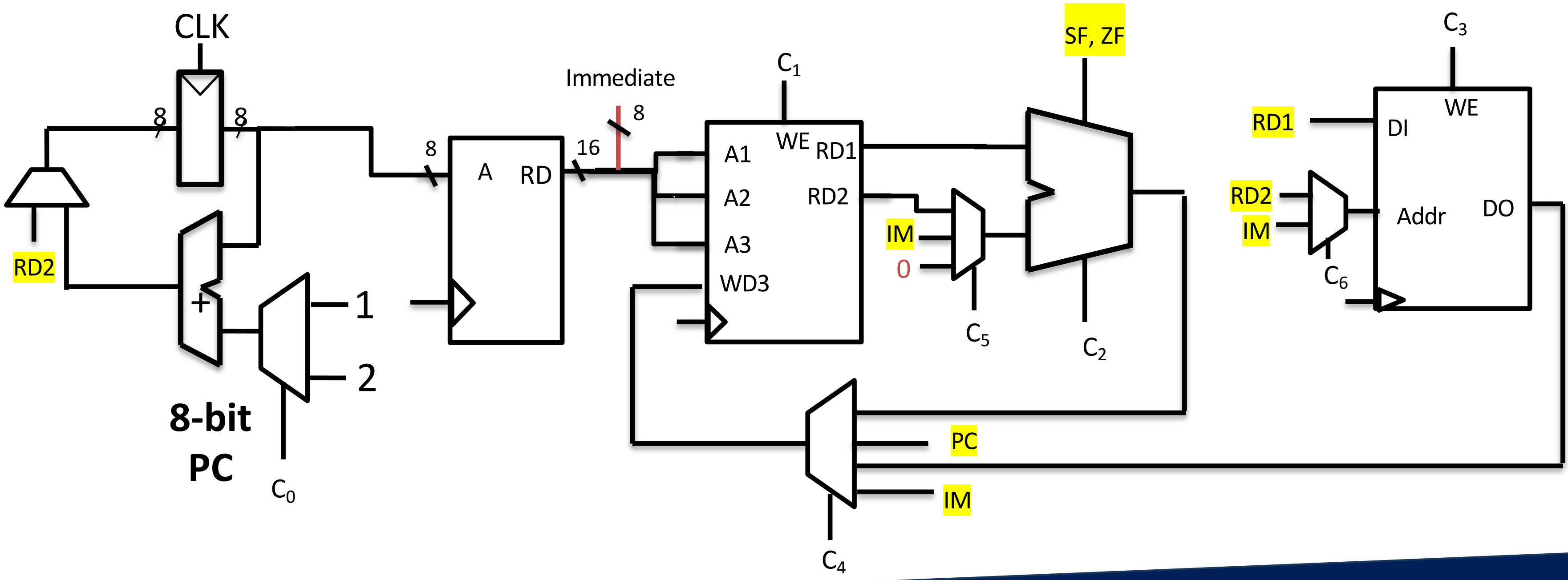
Just need a mux



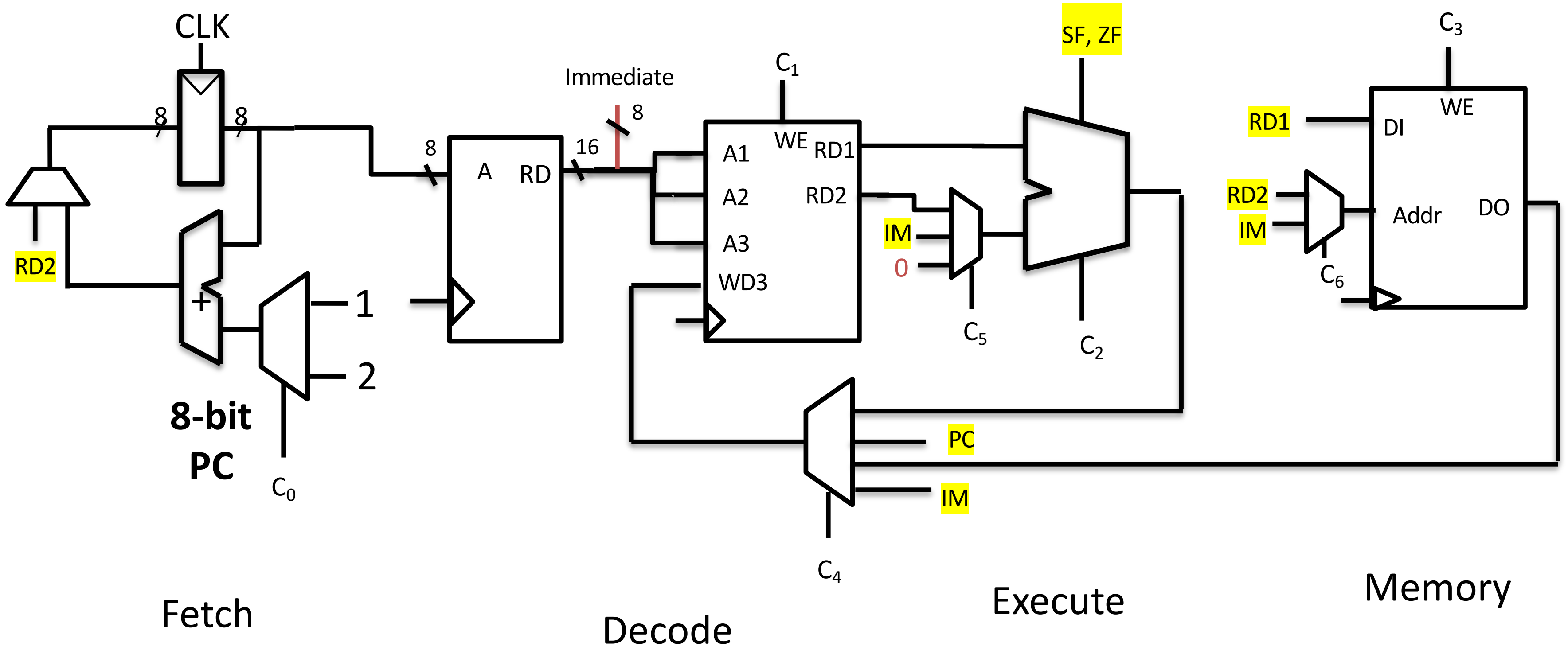
7	Compare **rA** as 8-bit 2's-complement to **0**
	if **rA <= 0** set **pc = rB**
	else increment **pc** as normal

Let's do a sample instruction

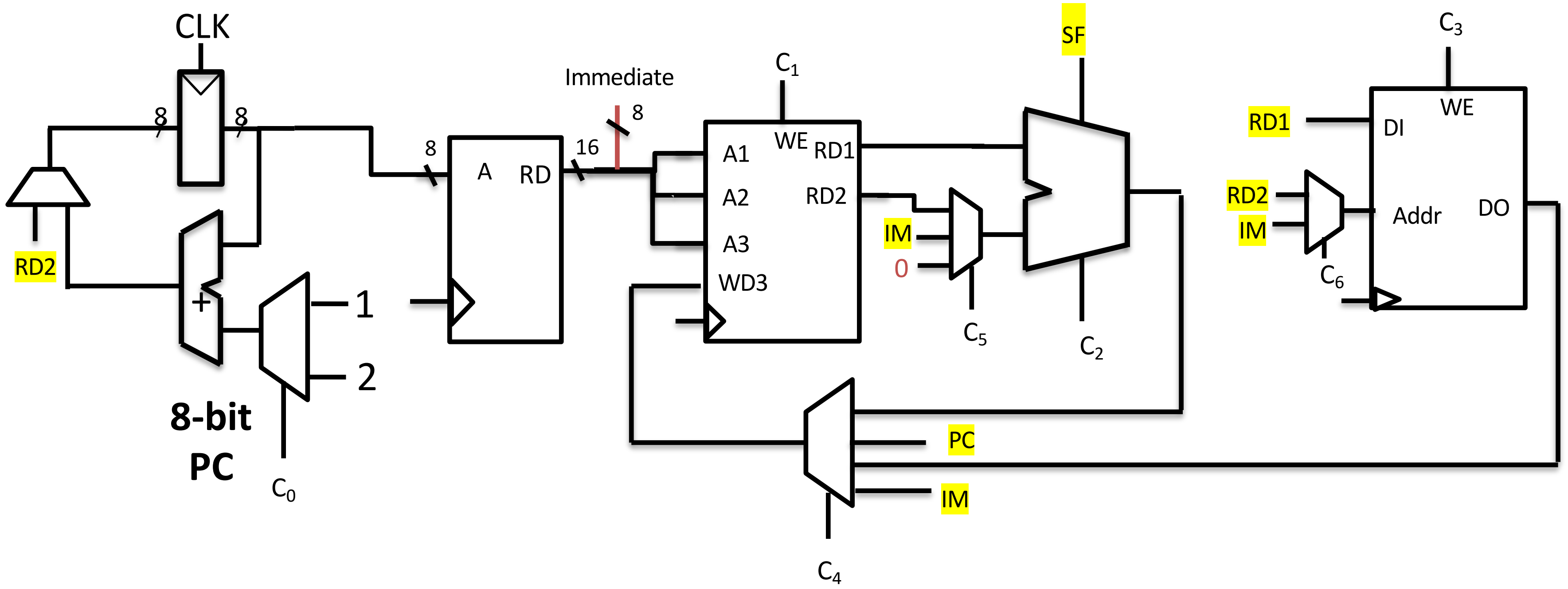
R	icode	RA	RB



OUR SINGLE CYCLE TOY PROCESSOR



OUR SINGLE CYCLE TOY PROCESSOR

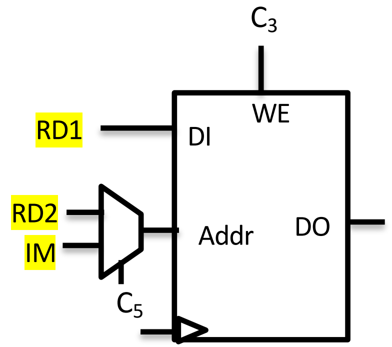


Write back stages

WHAT ABOUT DETAILS OF THE MAIN MEMORY

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

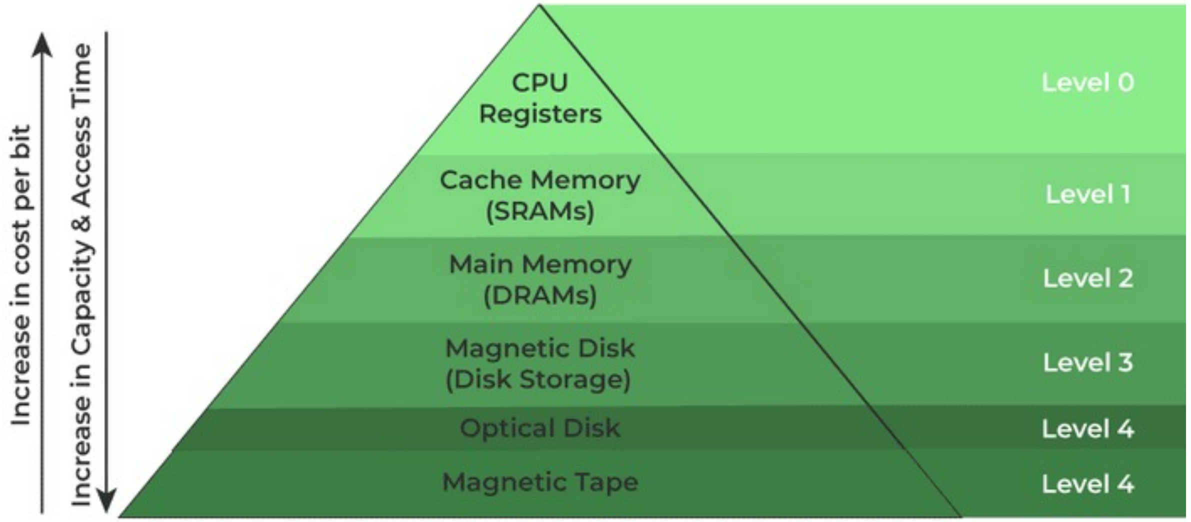
3. Don't worry we'll answer this in CSO 2.
    1. It is actually a complex hierarchy including a controller, caches, and Hardware support for virtual memory like TLBS (translation lookaside buffers)
    2. It doesn't always return a value in a single cycle so the controller might have to insert nops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



MEMORY HIERARCHY

Levels in Speed and Capacity of the Memory

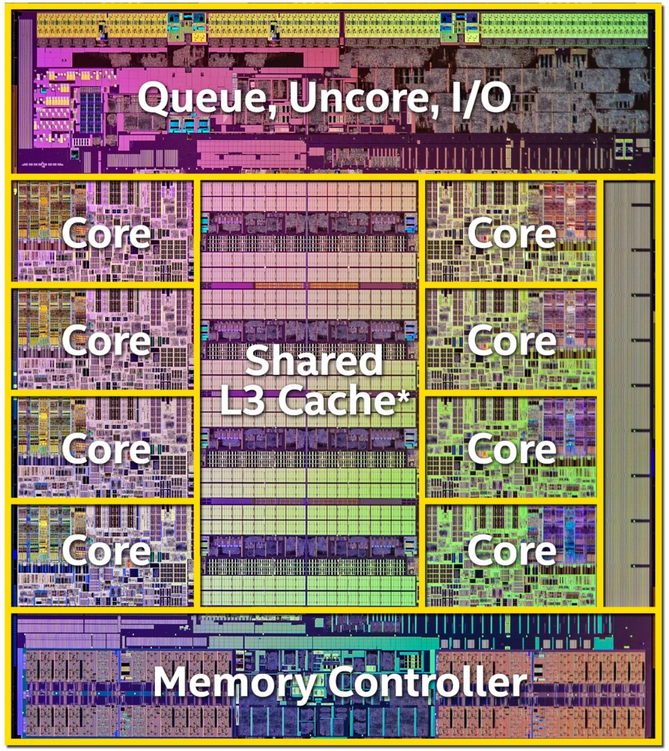
Higher levels larger and slower memory



Memory Hierarchy Design

Figure from: https://www.geeksforgeeks.org/memory-hierarchy-design-and-its-characteristics/

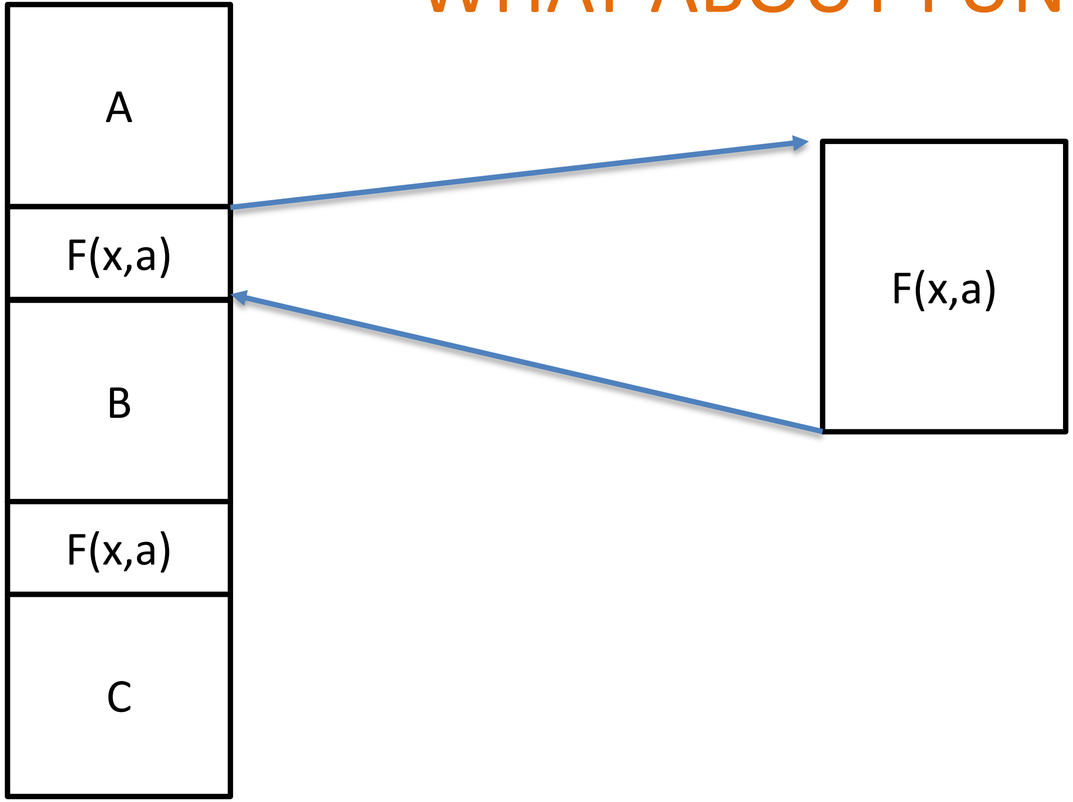
New 8-Core Intel® Core™ i7 Processor Extreme Edition



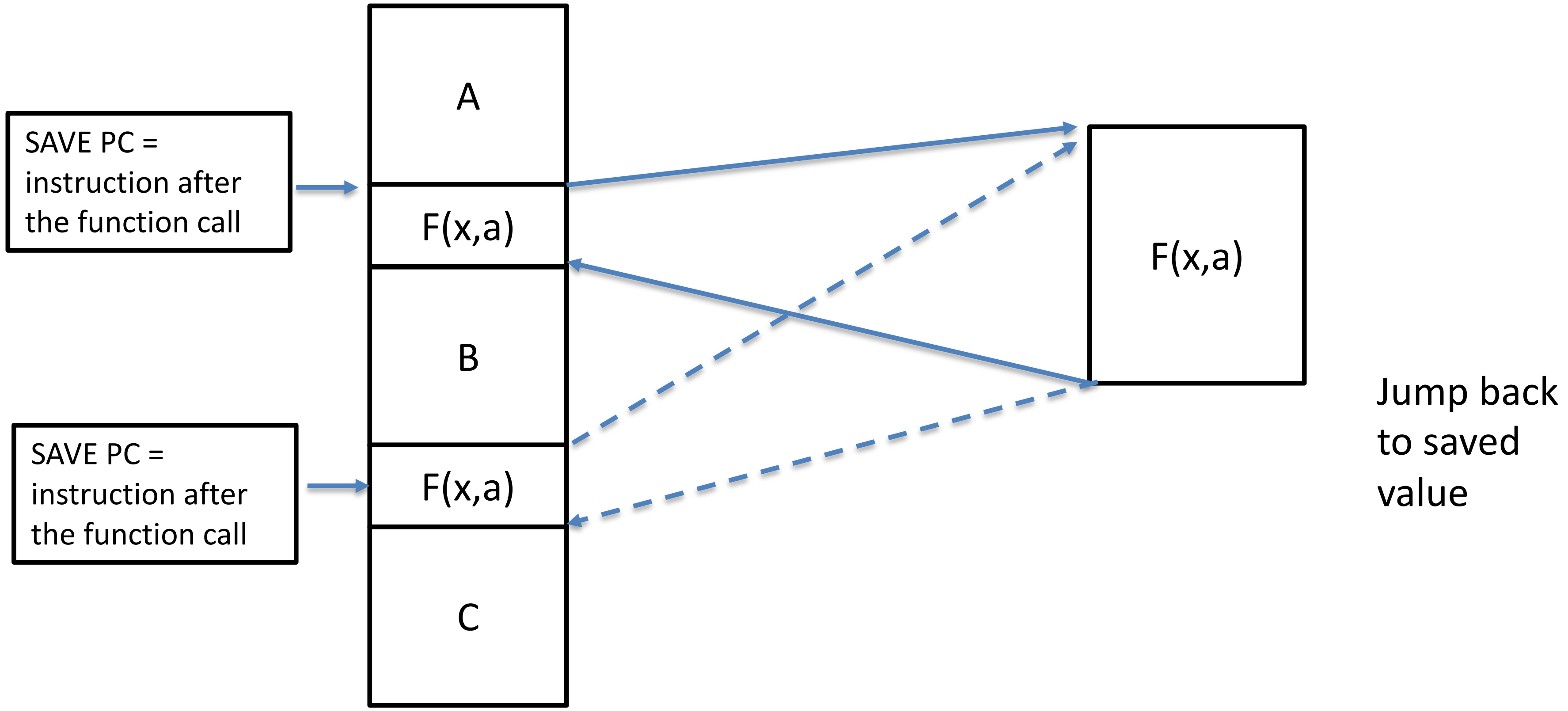
Intel® Core™ i7-5960X Processor Extreme Edition
Transistor count: 2.6 Billion
Die size: 17.6mm x 20.2mm

* 20MB of cache is shared across all 8 cores

WHAT ABOUT FUNCTIONS



Jump back to the main code


SAVE PC = instruction after the function call
A
F(x,a)
B
SAVE PC = instruction after the function call
F(x,a)
C
F(x,a)
Jump back to saved value

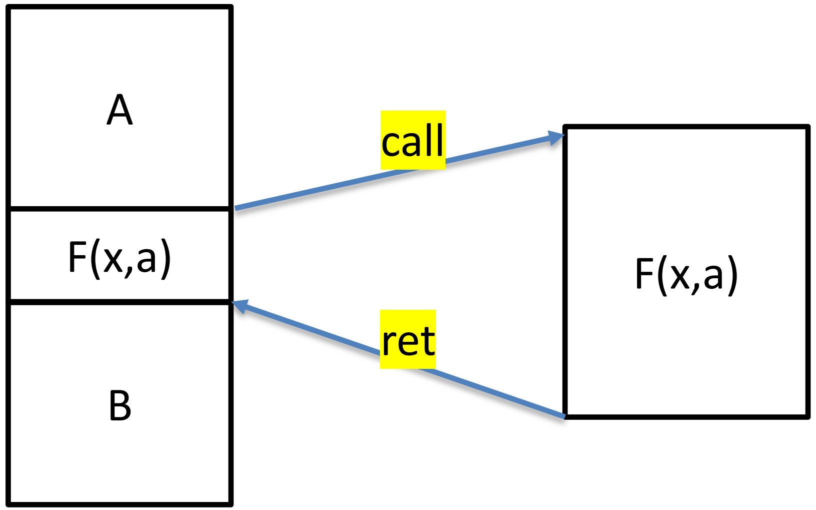
DEFINING A NEW INSTRUCTION

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

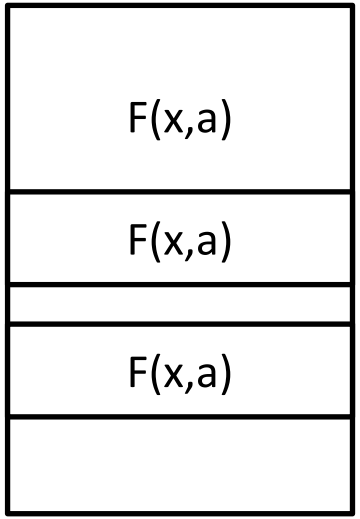
Save pc+2 , set pc = M[pc+1]

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

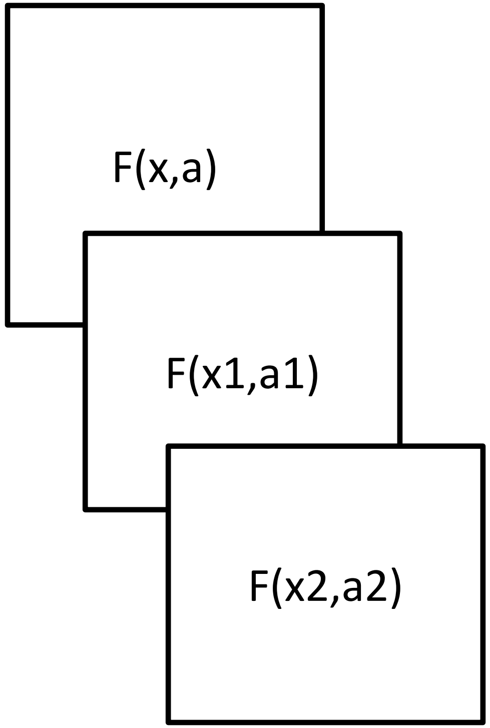


WHAT ABOUT FUNCTIONS



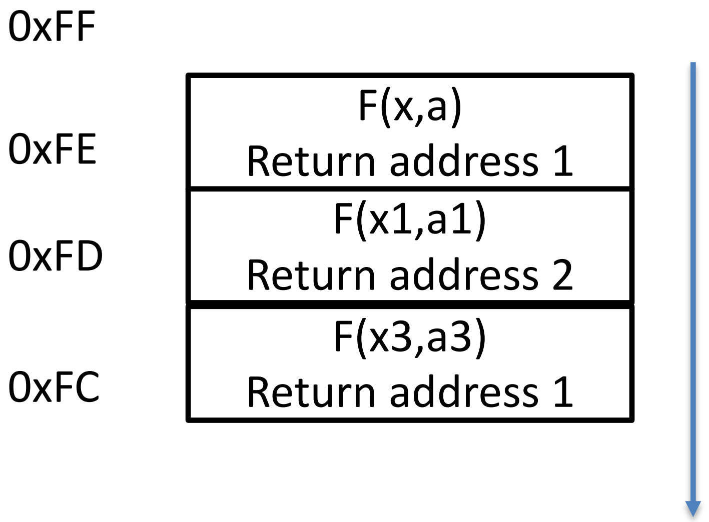
What about recursive functions? Functions that call themselves

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



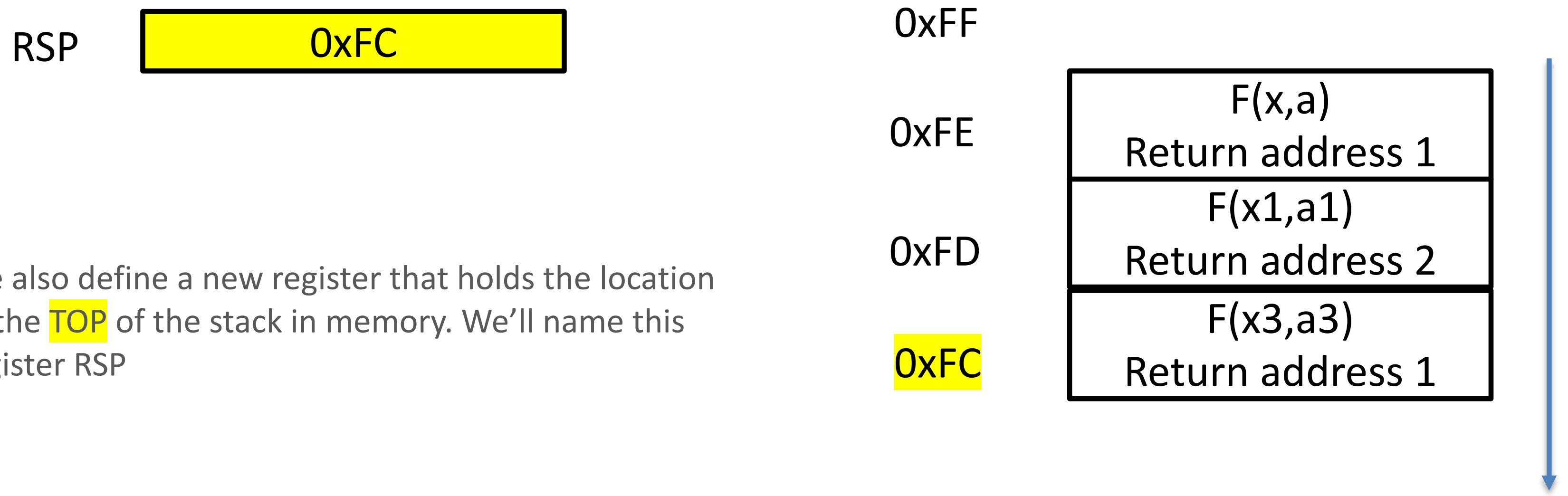
THE STACK

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



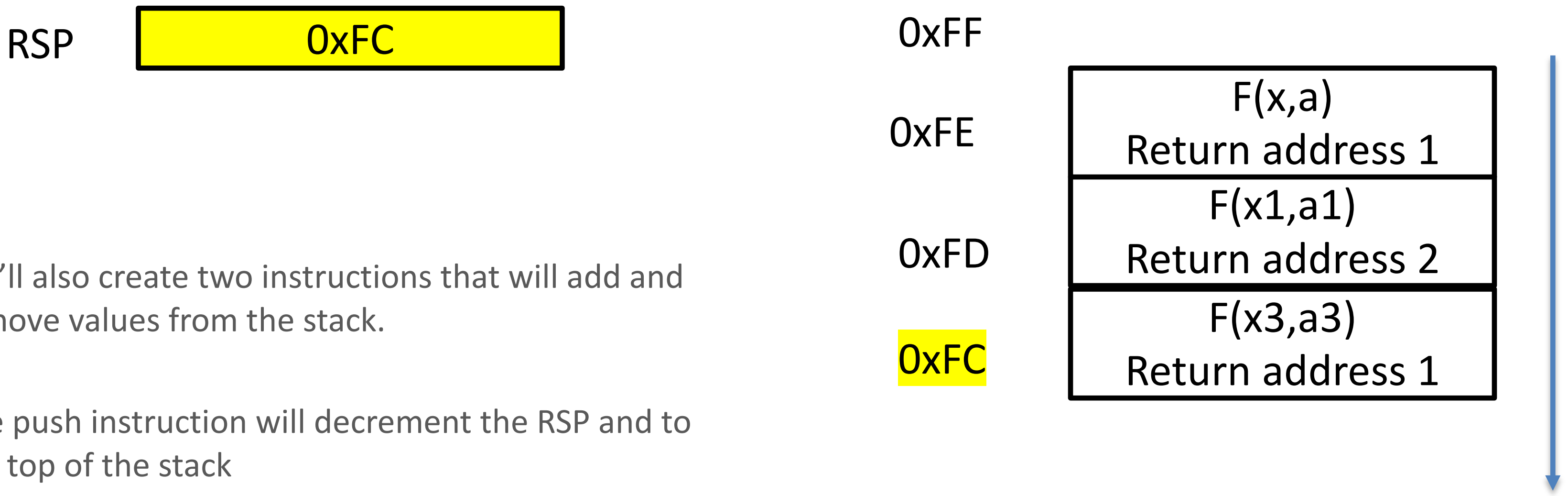
By convention keep adding new things to the stack by growing it to lower addresses

THE STACK



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

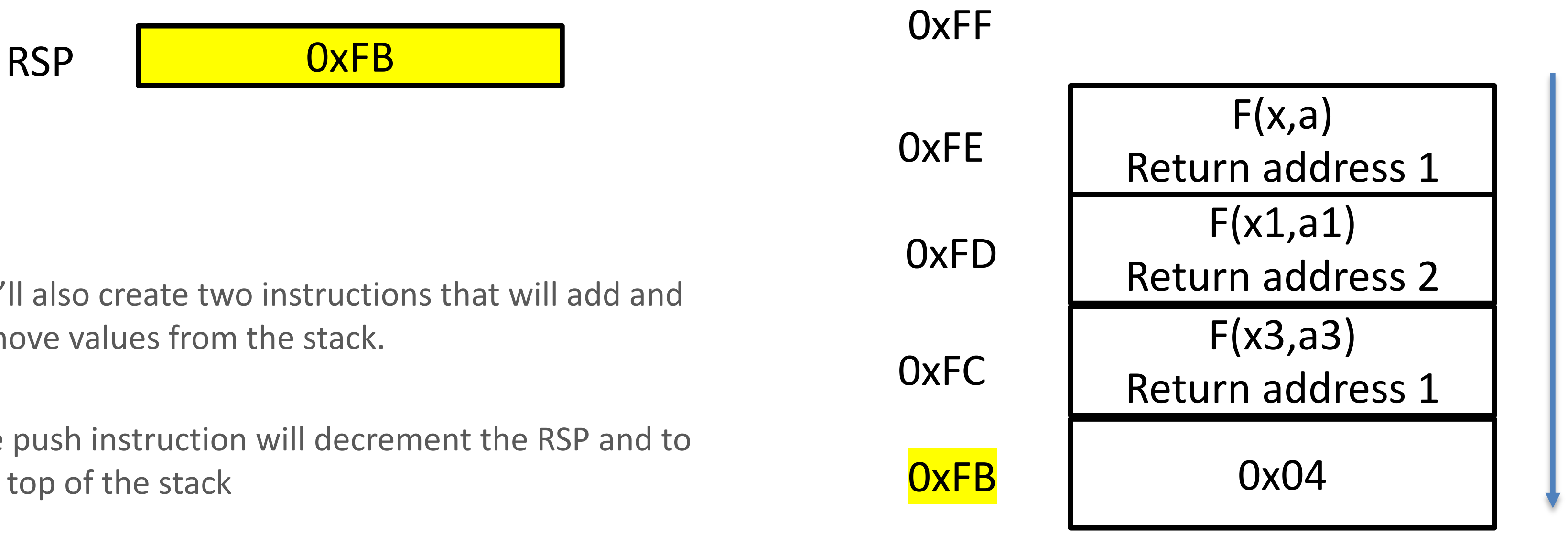


We'll also create two instructions that will add and remove values from the stack.

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

Example push(0x04)

PUSH AND POP INSTRUCTIONS

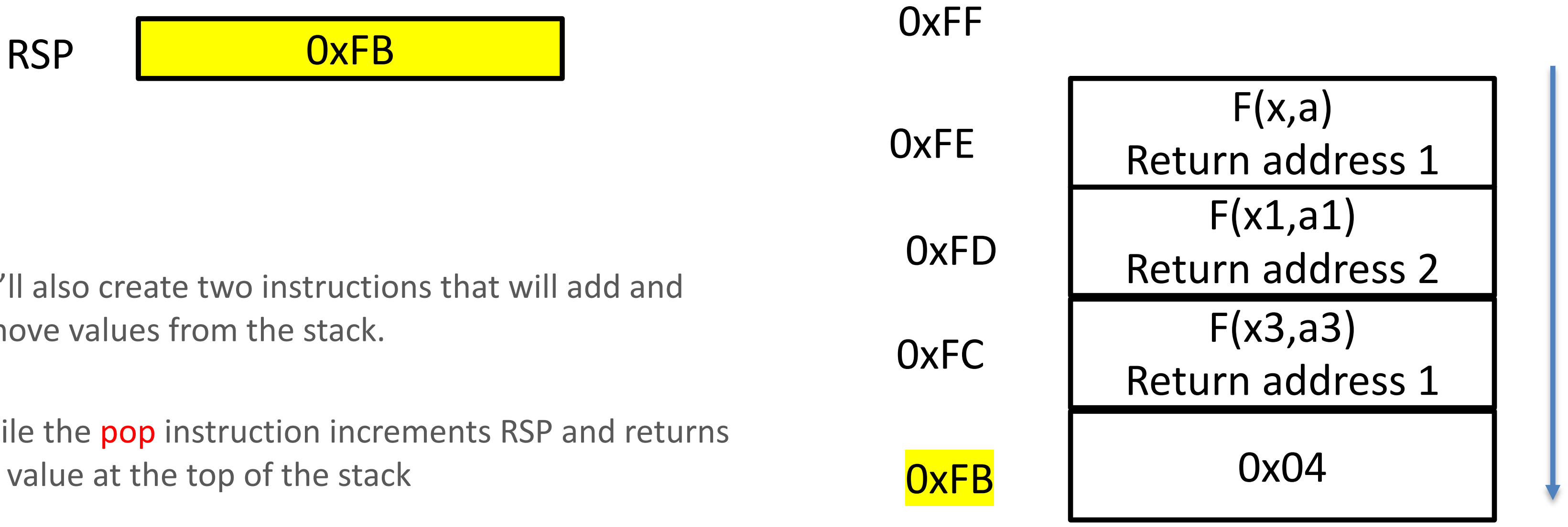


We'll also create two instructions that will add and remove values from the stack.

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

Example push(0x04)

PUSH AND POP INSTRUCTIONS

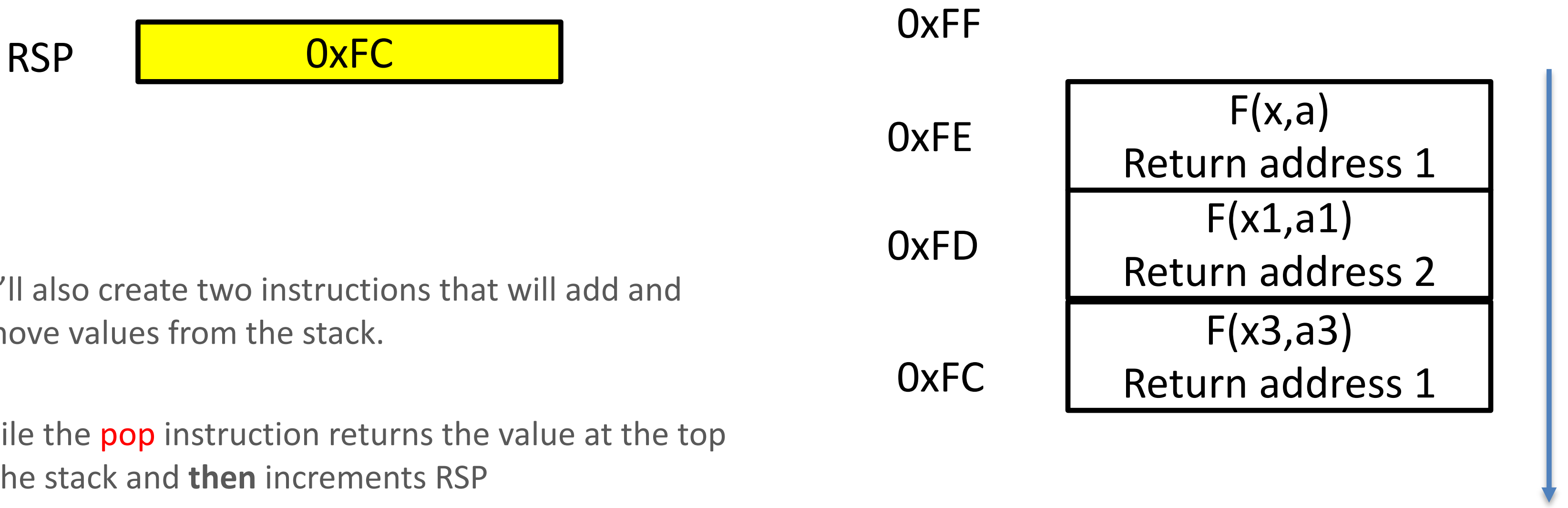


We'll also create two instructions that will add and remove values from the stack.

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

Example x = pop()

PUSH AND POP INSTRUCTIONS



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

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

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, shiftAmount)
```

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

THOUGHT EXPERIMENTS

Could you implement the left_shift function using our toy ISA?

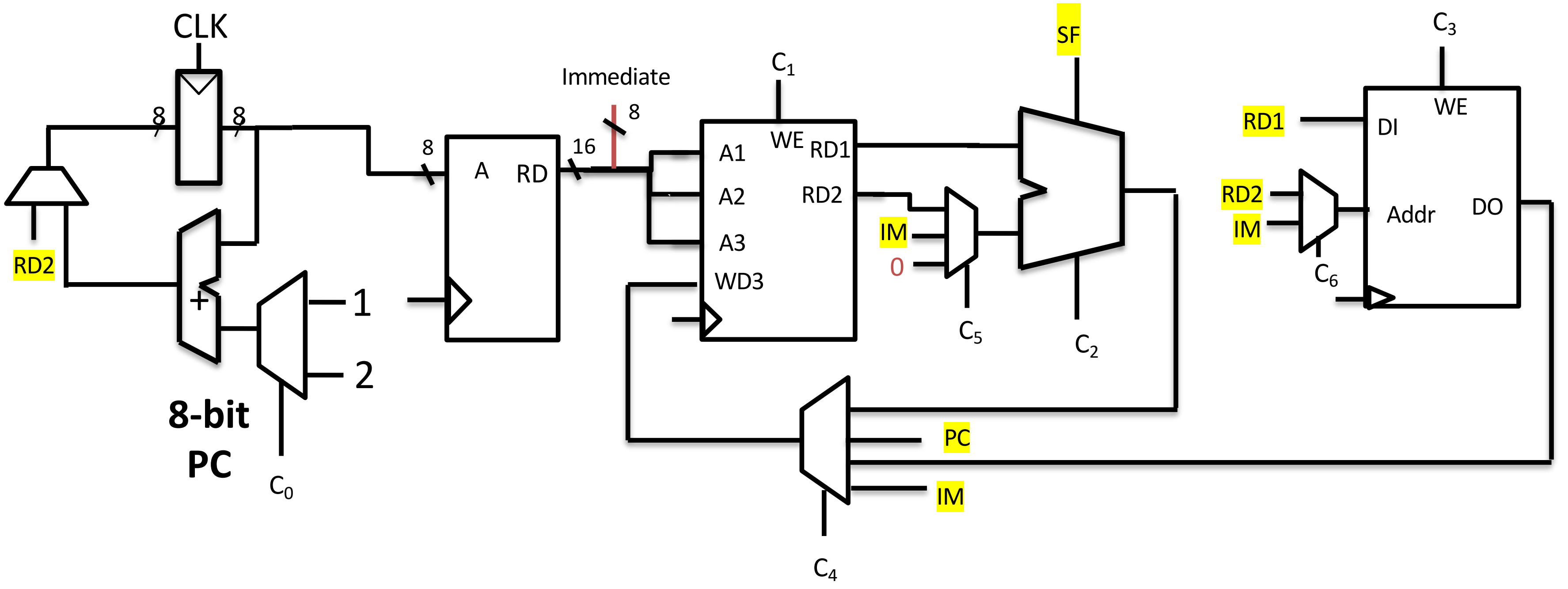
output = left_shift(input, shiftArmount)

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

ISA EXTENDED BY SETTING R BIT TO 1

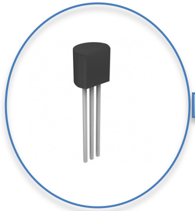
icode	b	operation
0		
	0	Decrement rsp and push the contents of rA to the stack
	1	Pop the top value from the stack into rA and increment rsp
	2	Push pc+2 onto the stack, set pc = M[pc+1]
	3	pc = pop the top value from the stack If b is not 2, update the pc as normal.

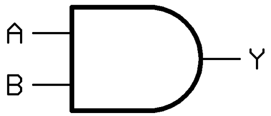
COULD YOU ADD PUSH, POP, CALL AND RET?

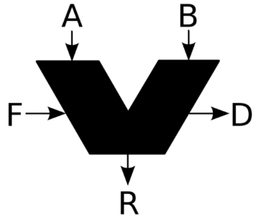


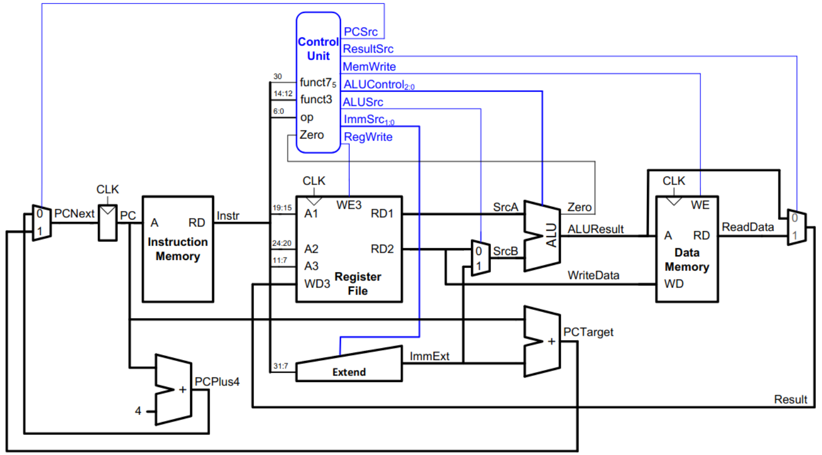
Write back stages

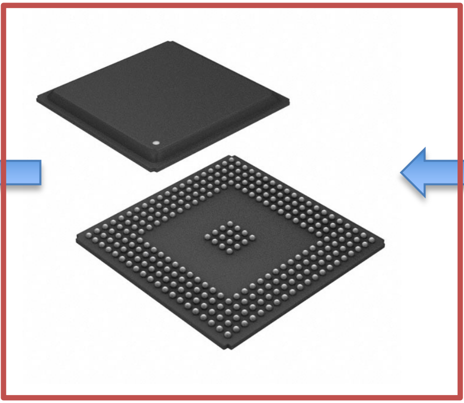
THE MAP (THE MACHINE)











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

WHAT ABOUT FABRICATING THESE

You can express our design in a programming language called VHDL.

Simulate your processor in model sim

And then send off the TSMC, UMC, or Samsung to get fabricated.

Don’t worry you’ll not have to write VHDL in this course. But ECE does offer courses. Maybe I will rework or simulation lab to give us a taste of this language.

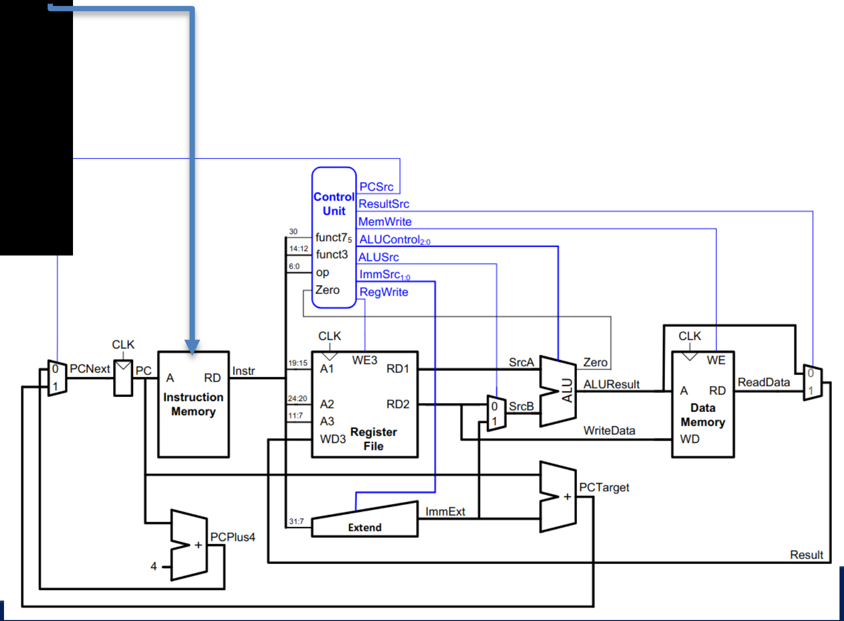
```
signal and_gate : std_logic;
and_gate <= input_1 and input_2;
```

```
entity example_and is
  port (
    input_1     : in  std_logic;
    input_2     : in  std_logic;
    and_result  : out std_logic
  );
end example_and;
```

```
architecture rtl of example_and is
  signal and_gate : std_logic;
begin
  and_gate <= input_1 and input_2;
  and_result <= and_gate;
end rtl;
```

THE MAP (THE CODE)

```
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
```



THE MAP (THE CODE)

```
#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

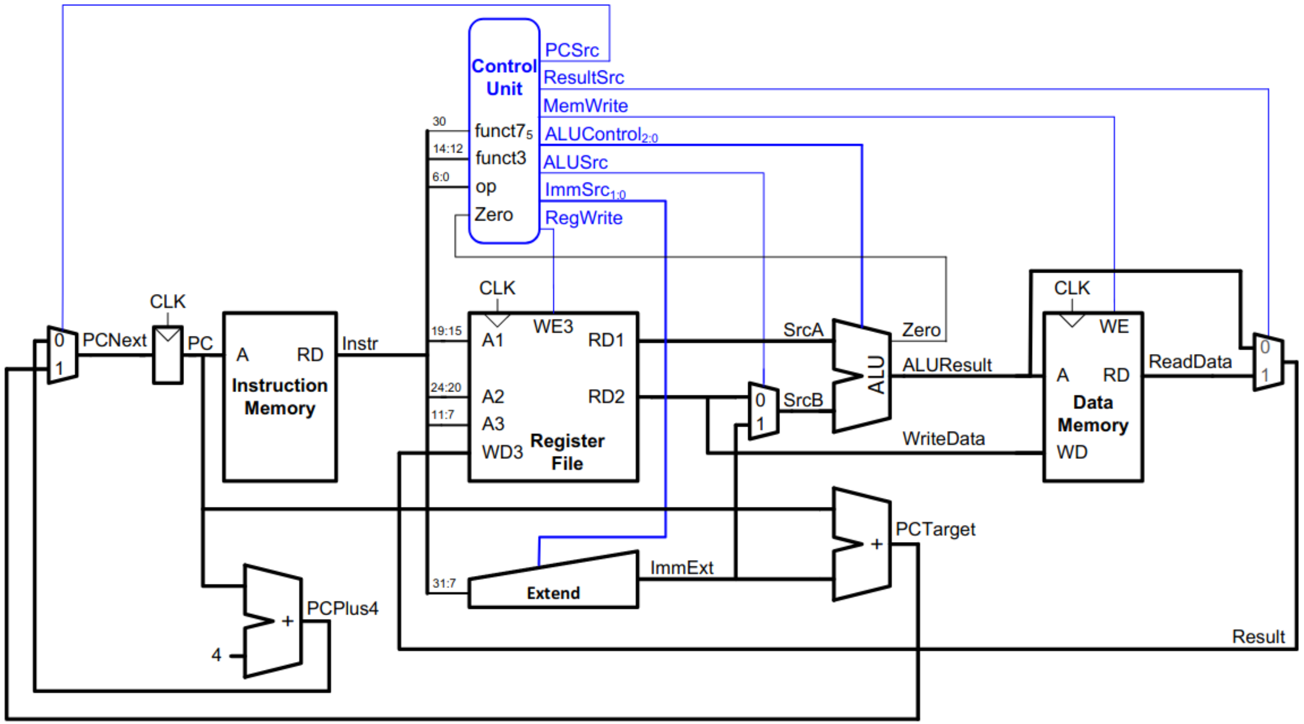
UNIVERSITY of VIRGINIA | ENGINEERING

THE STACK

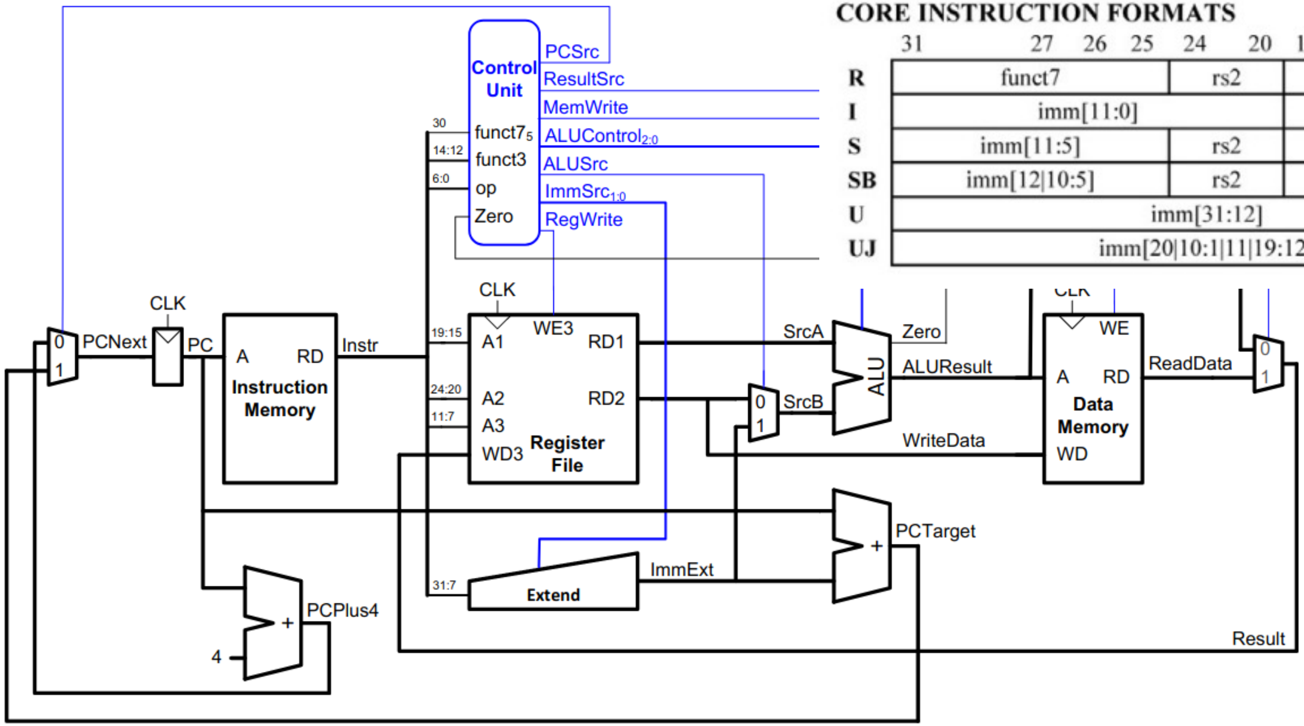
The Stack is a region of memory

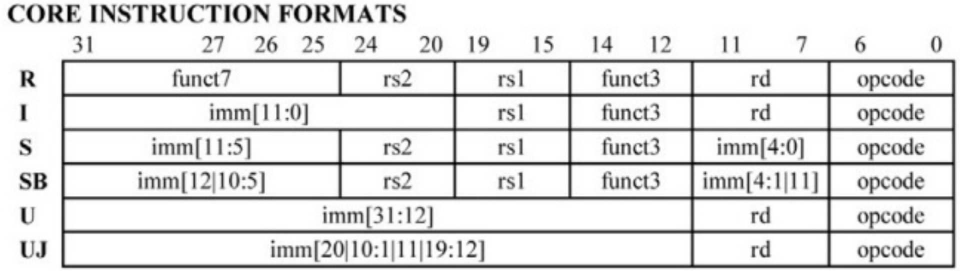
OUR JOURNEY SO FAR

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 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

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