Midterm 1 Review

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

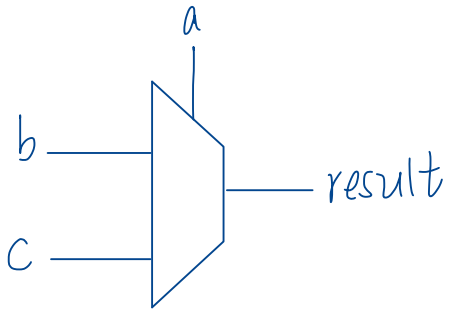
UNIVERSITY of VIRGINIA | ENGINEERING

Multiplexer (mux)

ternary operator

x=a ? b : c

A multiplexer (mux) is commonly drawn as a trapezoid in circuit diagrams.




```
if(a){
    b
}else{
    c
}
```

a	b	c	result
0			c (depends on c)
1			b (depends on b)

a, b, c could be anything, they could be strings, numbers, functions.....

a.b.c are all one bit.

a	b	c	result
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

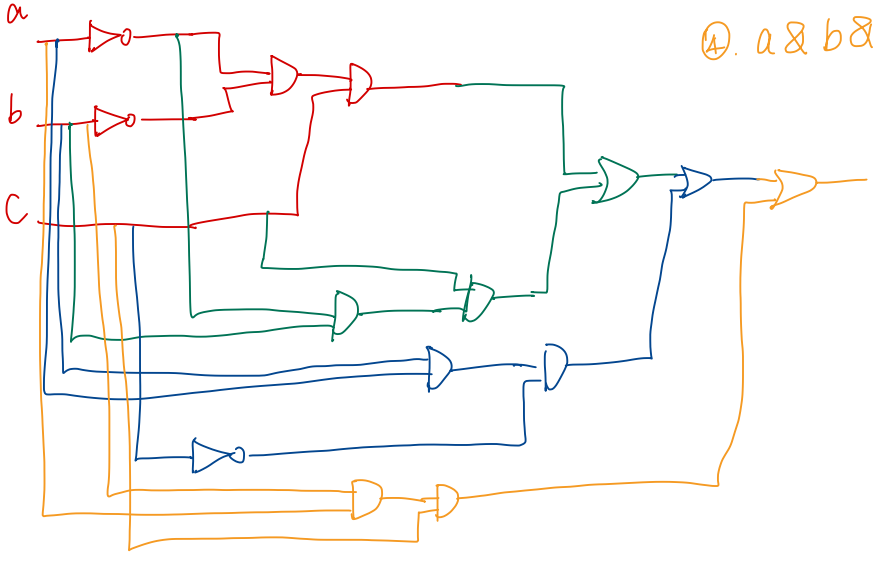
when the result is true ?

(!a&!b&c) | (!a&b&c) |

(a&b&!c) | (a&b&c)

① !a&!b&c. ②. !a&b&c. ③. a&b&!c.

④. a&b&c



Binary

Any downsides to binary?

Turn 2130_{10} into base-2:

hint: find largest power of 2 and subtract

2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
2048	1024	512	256	128	64	32	16	8	4	2	1
1	0	0	0	0	1	0	1	0	0	1	0

2130 - 2048 = 82 82 - 64 = 18 18 - 16 = 2

Exercise

Turn 1101011110010_2 into base-10:

1 1 0 1 0 1 1 1 1 0 0 1 0

$2^{12}+2^{11}+2^9+2^7+2^6+2^5+2^4+2^1$

$=4096+2048+512+128+64+32+16+2$

$=6898$

Exercise

Turn 342_{10} into binary:

	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
512	256	128	64	32	16	8	4	2	1
	1	0	1	0	1	0	1	1	0

342 - 256 = 86

86 - 64 = 22

22 - 16 = 6

6 - 4 = 2

2 - 2 = 0

Exercise

Turn 1101011110010_2 into **hexadecimal**:

1 1010 1111 0010

0001 1010 1111 0010

1 A F 2

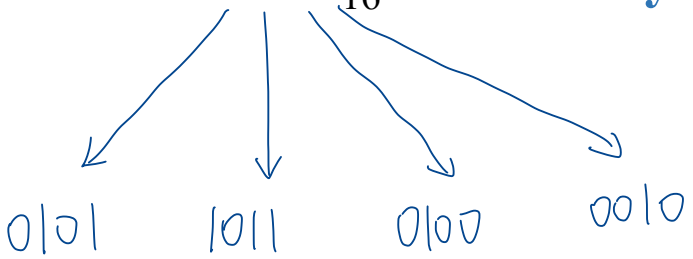
$1AF2_{16}$

//Turn 1101011110010_2 into **8-bit hexadecimal**:

How to extend it ?

Exercise

Turn $5b42_{16}$ into **binary** :



$5b42_{16} = 0101\ 1011\ 0100\ 0010_2$

Exercise

Turn $5b42_{16}$ into **decimal** :

5 b 4 2

16^3 16^2 16^1 16^0

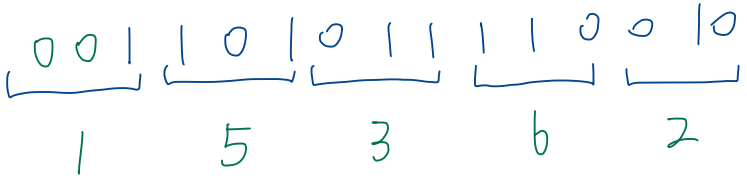
$5\times16^3 + 11\times16^2 + 4\times16^1 + 2\times16^0$

$= 20480 + 2816 + 64 + 2$

$= 23362$

Exercise

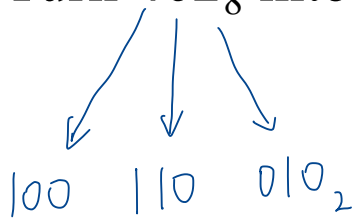
Turn 1101011110010_2 into **octal** :



So, $1101011110010_2 = 15362$

Exercise

Turn 462_8 into **binary** :



Binary Addition

01101011 + 01100101

```
  0 1 1 0 1 0 1 1   (107)
+ 0 1 1 0 0 1 0 1   (101)
-----------------
  1 1 0 1 0 0 0 0   (208)
```

11101011 + 11100101

```
  1 1 1 0 1 0 1 1   (235)
+ 1 1 1 0 0 1 0 1   (229)
-----------------
1 1 1 0 1 0 0 0 0   (208)
```

The correct answer should be 464.
But I only have 8 bits for my result (read as 208)

range for 8 bits: (0 ~ 255)

$2^8 - 1 = 256 - 1 = 255$

Binary Subtraction

01111011 - 01100101

```
  0 1 1 1 1 0 1 1   (123)
- 0 1 1 0 0 1 0 1   (101)
-----------------
  0 0 0 1 0 1 1 0   (22)
```

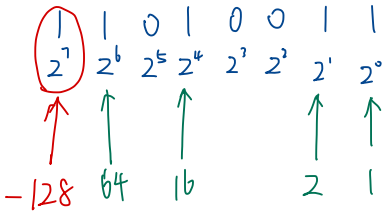
Values of Two's Complement Numbers

Consider the following 8-bit binary number in Two's Complement:

11010011

What is its value in decimal?

Method 1:



$-128+64+16+2+1=-45$

Values of Two's Complement Numbers

Consider the following 8-bit binary number in Two's Complement:

11010011

What is its value in decimal?

1 1 0 1 0 0 1 1
128 64 32 16 8 4 2 1

1. Flip all bits
2. Add 1

00101100
32 16 8 4 2

\+ 1

00101101

32+8+4+1 = 45 ⇒ −45

−128 +64+16+2+1 = −45

Example: Bitwise AND

```
  11001010
& 01111100
----------
  01001000
```

Example: Bitwise OR

```
  11001010
| 01111100
----------
  11111110
```

Example: Bitwise XOR

```
  11001010
^ 01111100
----------
  10110110
```

Operations (on Integers)

Logical not: $!x$

- $!0 = 1$ and $!x = 0, \forall x \neq 0$
- Useful in C, no booleans
- Some languages name this one differently

0 means false
Any non-zero value means true.

C does not have a built-in Boolean type in older standards.
Logical expressions return integers (0 or 1)

Example:
```
if(!ptr) {
    // ptr is NULL
}
```

Python: not
Java/C/C++: !
Bash: !

Operations (on Integers)

Left shift: $x << y$ - move bits to the left

- Effectively multiply by powers of 2

Right shift: $x >> y$ - move bits to the right

- Effectively divide by powers of 2
- Signed (extend sign bit) vs unsigned (extend 0)

Floating Point Example

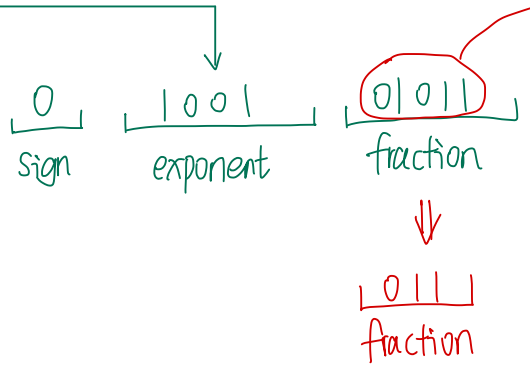
1 bit: sign
4 bits: exponent
3 bits: fraction

$$101.011_2$$

1.011×2^2

2's complement for 2: 0010

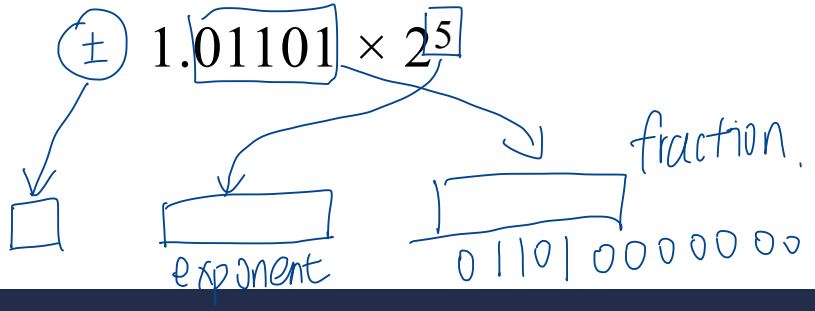
add the bias: 0010
+0111
1001



Floating Point in Binary

How do we store them?

- Originally many different systems
- IEEE standardized system (IEEE 754 and IEEE 854)
- Agreed-upon order, format, and number of bits for each



Floating Point Example

$\underbrace{101}_{5}.\underbrace{011}_{}_2$

couple of ways to think about what comes after the binary point.

① in fractions

$1\ 0\ 1\ .\ 0\ 1\ 1$

$2^2\ 2^1\ 2^0\ \ 2^{-1}\ 2^{-2}\ 2^{-3}$

$4+1=5 \qquad 2^{-2}+2^{-3}=\frac{1}{4}+\frac{1}{8}=\frac{3}{8}$

so: $101.011 = 5\frac{3}{8}$

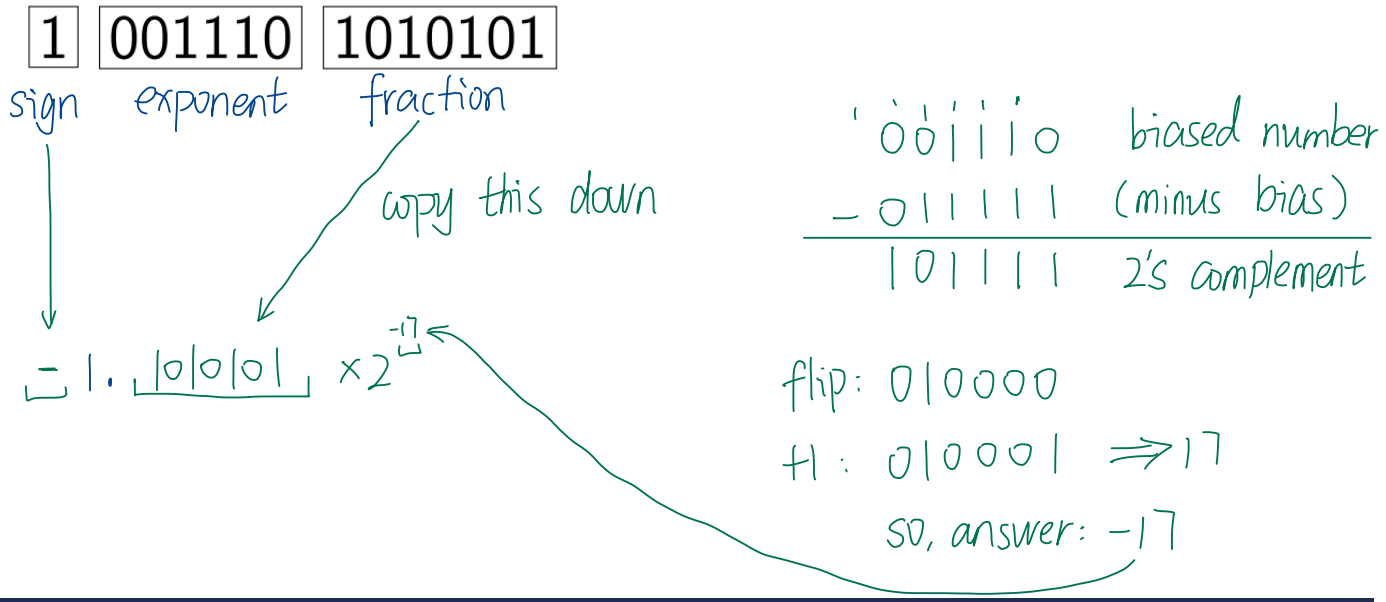
② positionally

3 positions : up to $2^3=8$ values (positions)

011 is 3, so 3/8 (three · eight)

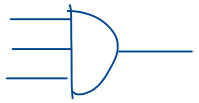
Floating Point Example

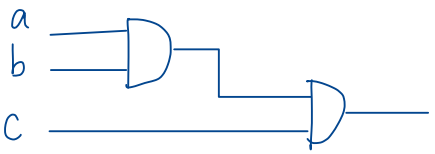
What does the following encode? 14 bits.



Warm up!

Can I make an *n*-input AND from 2-input AND gates?



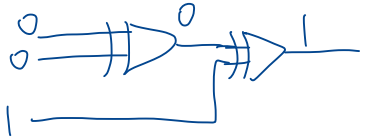


What about XOR gates?

parity → reading

0- even number of bits that are one 110 → 0.

1- odd number of bits that are one. 001 → 1



useful in error checking:

send: 101 1^0^1 = 0

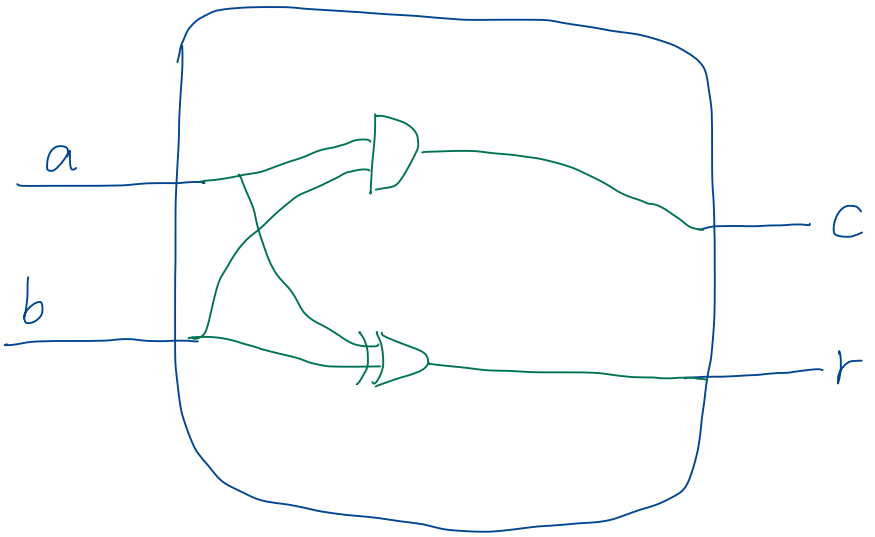
receive: 100 1^0^0 = 1

Adder

Add 2 1-bit numbers: *a*, *b*

```
   a
 + b
-----
 ⌴ ⌴
 c r
```

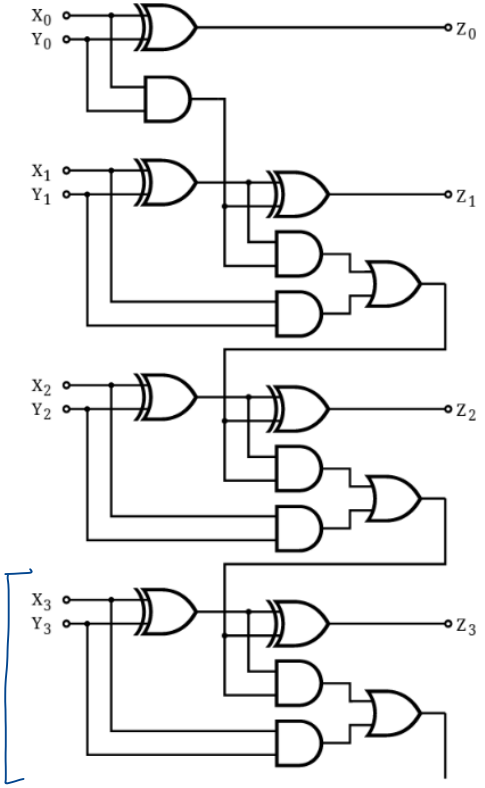
a	b	c	r
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Is that a one bit value?

How big could that value be?

Ripple-Carry Adder



$X_3\ X_2\ X_1\ X_0$
$+\ Y_3\ Y_2\ Y_1\ Y_0$
$Z_3\ Z_2\ Z_1\ Z_0$

repeat if more bits

verify:

unsigned:

```
  1 1 1 1  (15)
+ 1 1 1 1  (15)
-----------
  1 1 1 0  (14)
```

drop ← (1)

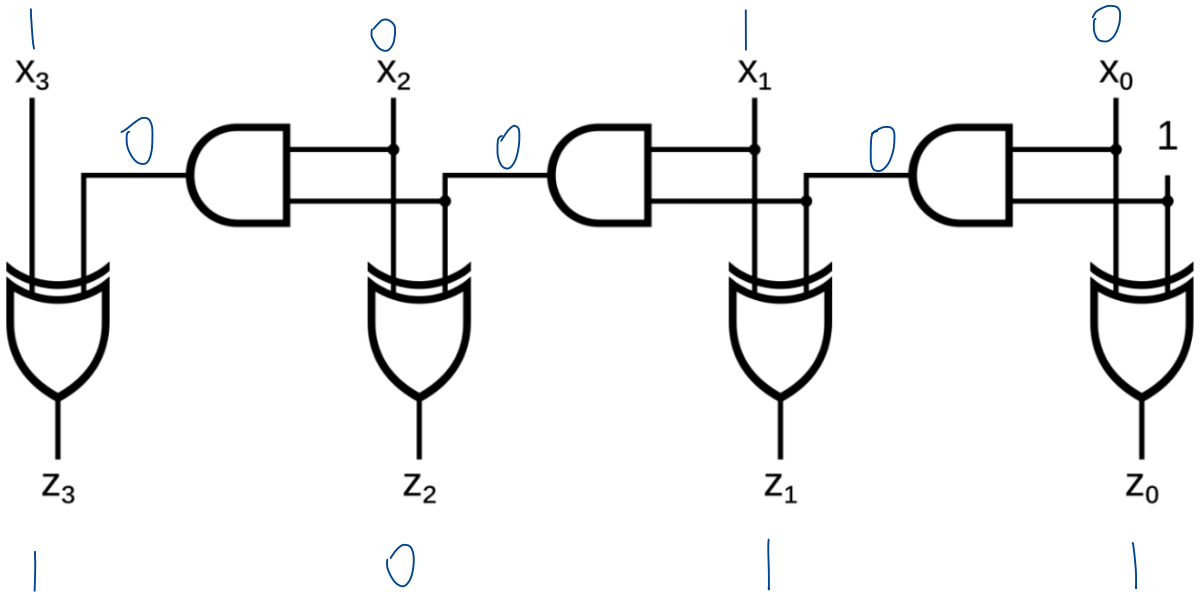
overflow!

signed?

```
  1 1 1 1  (-1)
+ 1 1 1 1  (-1)
-----------
  1 1 1 0  (-2)
```

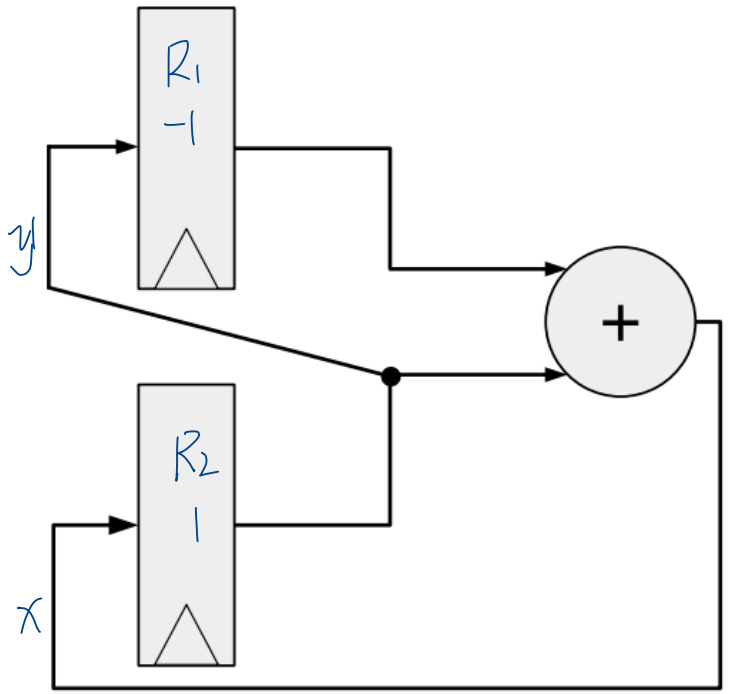
works!

What does this circuit do?



It adds 1 !

Another Counter

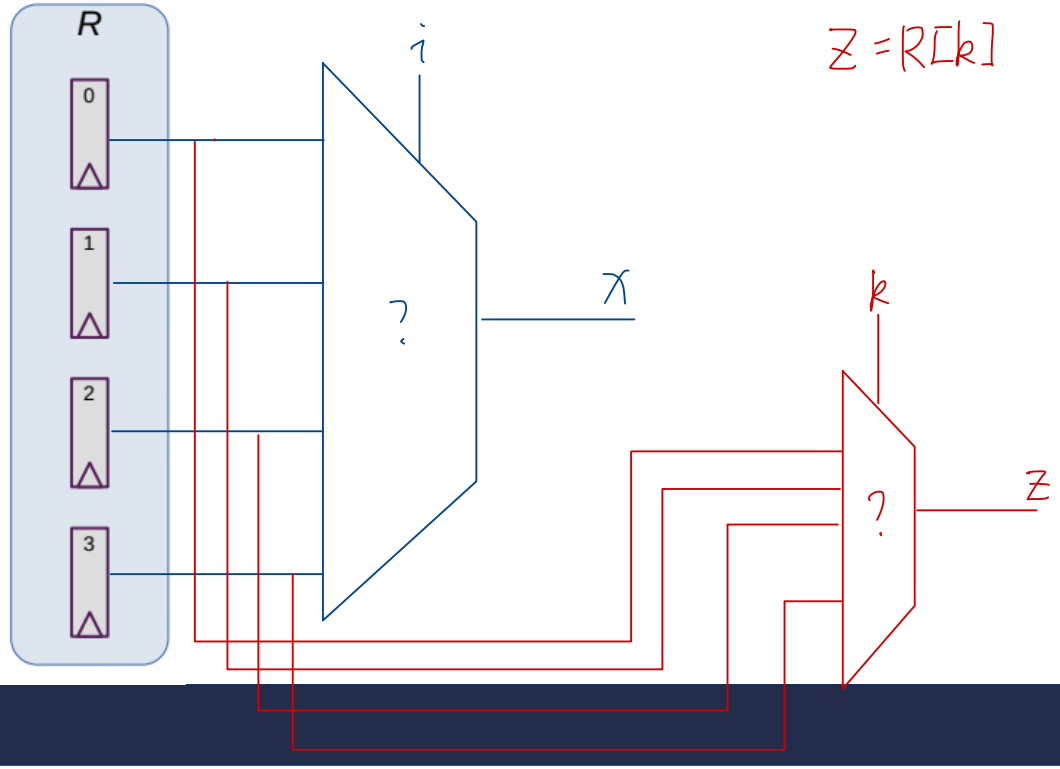


clock	R_1	R_2	x	y
0	-1	1	0	1
1	1	0	1	0
2	0	1	1	1
3	1	1	2	1
4	1	2	3	2
5			5	3

x : Fibonacci Sequence.

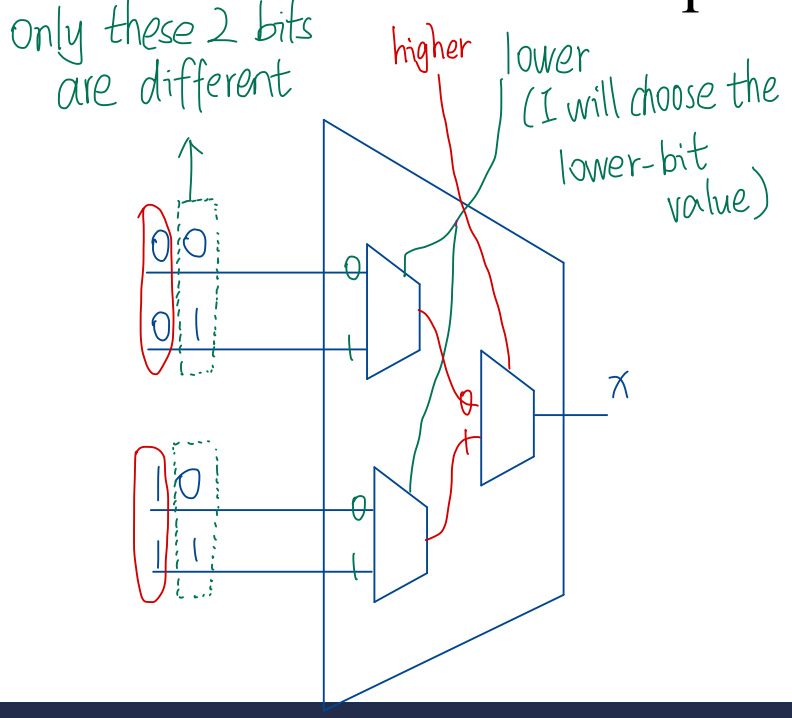
Reading

x = R[i] - connect output of registers to *x* based on index *i*



Aside: 4-input Mux

How do we build a 4-input mux? How many wires should *i* be?



i should be 2 bits.

How about 8-input mux?

$2^3 = 8$ so. i should be 3 bits.

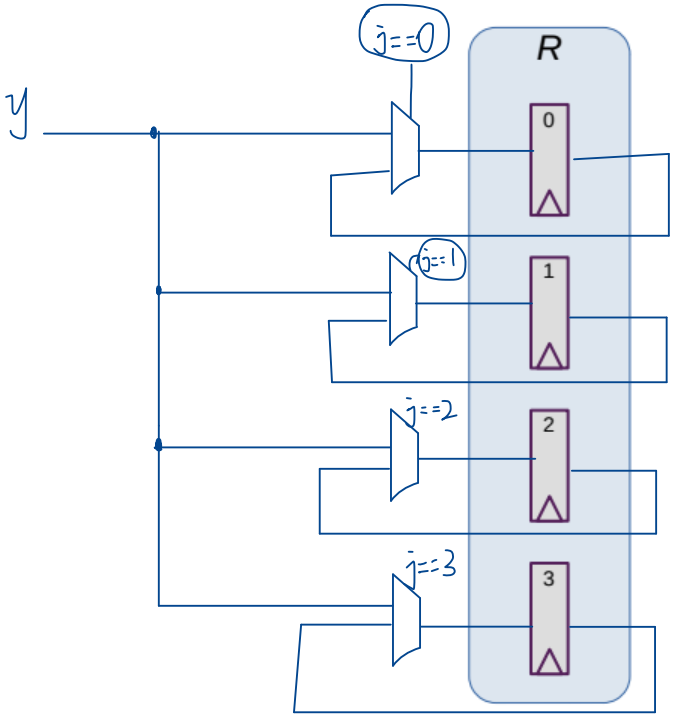
Writing

R[j] = y - connect *y* to input of registers based on index *j*

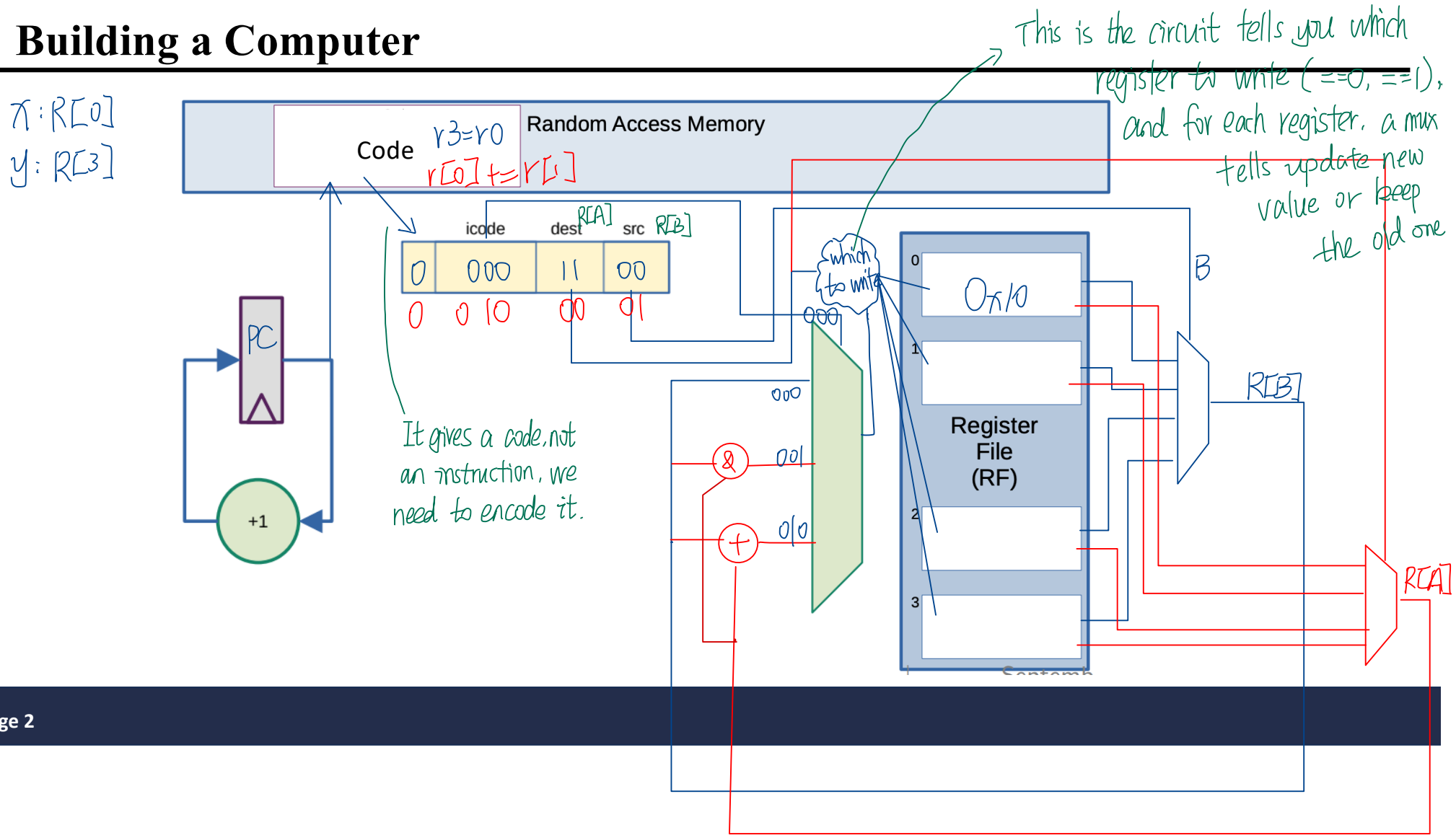
2 things to be considered.

① How do I choose which register I want to hook it up to?

② For all other registers that I'm not writing y to, they need to keep the current value. I don't want to loose it.



Building a Computer



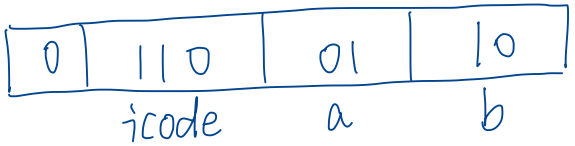
Encoding Instructions

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

in decimal

Example 1: r1 += 19

19 in hexdecimal: 0x13



hex: 0b13

Encoding Instructions

idea: ①. I have a value in memory at address hex 82

②. I want to add whatever in R3 to that value.

Example 2: M[0x82] += r3

Read memory at address 0x82, add r3, write back to memory at same address

One point: No instructions allow us to pass an immediate value as the address.
So let's save ourselves some time: just put 82 in a register.
Then we use icode 4 to read it out, icode 5 to write it back.

r2 = 0x82 — 0 110 10 00 — 82

r1 = M[r2] — 0 100 01 10 — (icode 4)

r1 += r3 — 0 010 01 11 — (icode 2)

M[r2] = r1 — 0 101 01 10 — (icode 5)

our machine reads 0 and 1.

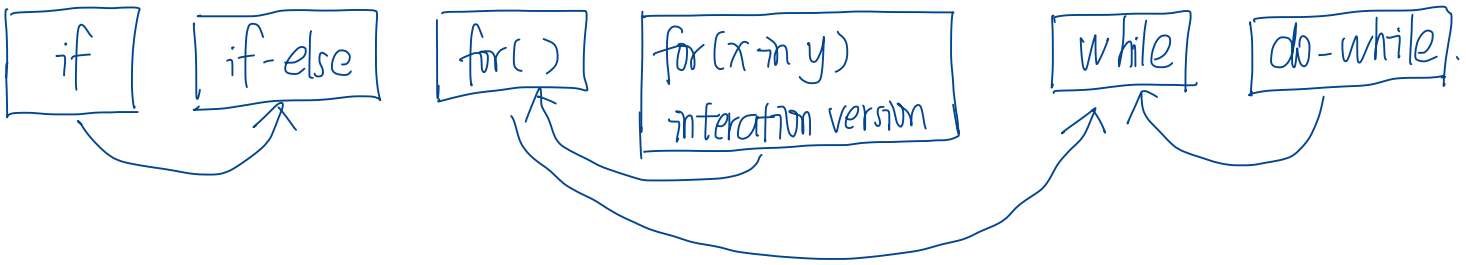
But, for us → easier to read → pairs of hex

68 82 46 27 56

One interesting finding: first hex is always our icode!

Our code to this machine code

How do we turn our control constructs into jump statements?



how to convert for to while:

```
for (int i=0; i<25; i++){
    =
    =
}
```

⇒

```
int i=0
while(i<25) {
    =
    =
    i++; }
```

In C/Java, we have if/else. But from the CPU perspective, there is no such thing as if/else — only sequential execution and jumps. Think of the CPU as walking forward on a road, Default behavior: keep moving forward. Only when we do NOT want to continue forward, we need to jump somewhere else. So the compiler naturally ask "When should I jump away?" instead of "When should I continue?"

UNIVERSITY of VIRGINIA

if/else to jump

```
if ( D ) {
    A
} else {
    B
}
C.
```

if condition D is true, I will do A, skip B, continue to do C

```
if( !D ), jump to B
    A
jump to C (unconditiond jump)
    B
    C
```

2 situations:
① if D is true: don't jump. continue A, then C.
② if D is false: jump to B, then C.

jumps happens at 2 places where?

For human: if D is true → A, otherwise → B. But for machine: 1. The CPU will continue downward by default. 2. If D is false, where should we go?

it looks like we're "thinking backwards", but actually we are just following the CPU's natural rule: continue unless force to jump.

Why do compilers generate code this way?

① "Fewer jumps" ⇒ faster execution

② "Sequential flow" ⇒ better branch prediction

③ "This pattern appears in all architectures".

while to jump

while (C) {

A

}

B

How do I know where to jump?
① hard code if you know the address
② icode 3-3

if (!C), jump to B

A

jump to A? No! Infinate loop!

jump to if, do the condition check!

B

each loop iteration has 2 jump checks.

if (!C), jump to B

A

if (C), jump to A.

B

each loop iteration only has 1 jump check.

It's not the same loop

It's a do-while loop!

Encoding Instructions

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

Example 3: if r0 < 9 jump to 0x42

I don't have an instruction say r0<9.
I need "r0<=0" for icode 7, what should I do?

r0<9 $\Leftrightarrow$ r0<=8 $\Leftrightarrow$ (r0-8)<=0
$\Leftrightarrow$ r0+=-8 (0xF8)
r0<=0

r1 = 0x42 — 0 110 01 00 — 42
6 4 42

r0 += F8 — 0 110 00 10 — F8
6 2 F8

if r0<=0, PC=r1 — 0 111 00 01
(R0) (R1)
7 1

b442b2F871

register : ir : current instruction.
PC : address of the next instruction

Encoding Instructions

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

Example 4: 0x17 * 3

0x17 * 3 ⇒ 0x17+0x17+0x17 ⇒

for(i=0;i<3;i++) x+=0x17;

```
x=0;
i=0;
while(i<3){
  x+=0x17;
  i+=1;
}
```

```
x=0;
i=-2
//HERE
x+=0x17
i+=1;
if(i<3) jump HERE
if(i<=2)
if(i-2<=0)
if(i<=0) jump HERE
```

icode 3-3: rA=PC (Save where to back)

0011 / 3 | 10 / r2 | 11 / 3

what values we need?

x → r0

i → r1

HERE → r2

y=0x17 → immediate