The Elements of Computing Systems

NAND2TETRIS

Notes

20 July 2020 — Ch. 1

Boolean Expressions - AND → $(x+y)$

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

$f(x,y) = z$

OR → $x \cdot y = xy$

x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

NOT → $\bar{x} = \bar{y}$ →

x	$\bar{x}$
0	1
1	0

Ex: $(x+y)\cdot\bar{z}$ → $f(x,y,z)$

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

But how to deduce??

→ FIG 1

Canonical Representation

↳ Use truth table to deduce representation.

↳ Focus on what makes truth table = 1.

↳ Fig 1 → check Row 3 (true)

↳ Row 3: $x=0,\ y=1,\ z=0$.

$\therefore \bar{x}y\bar{z}$

$R5 = x\bar{y}\bar{z}$

$R7 = xy\bar{z}$

$$\sum_{\text{all true rows}\ (3,5,7)} = f(xyz) = \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z}$$

Common factor??

$(\bar{x}y + x\bar{y} + xy)\cdot\bar{z}$

$x\bar{y} + (x(\bar{y}+y)$

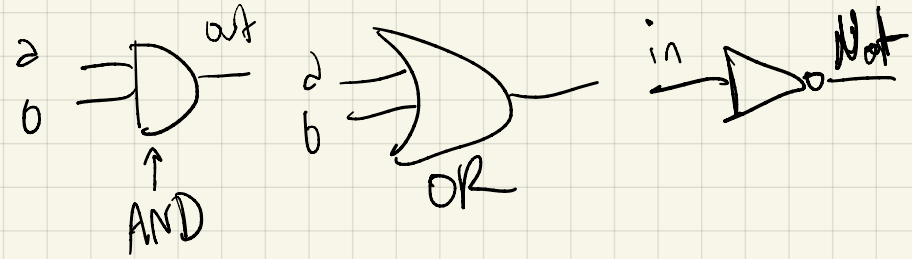
No.

x	y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

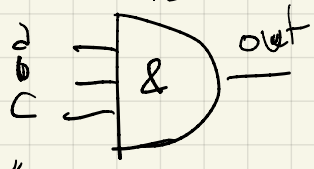
$\bar{y}+y = 1+0 = 0$

Thus, all boolean expressions boil down to AND, OR & NOT.

→ More Advanced versions of the operators include: NOR ($\overline{OR}$), XOR

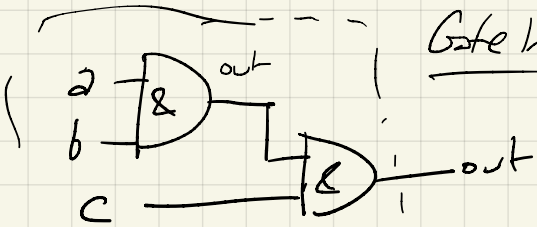


Gate Interface



"real world"

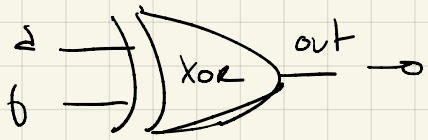
it's actually



Gate Implementation

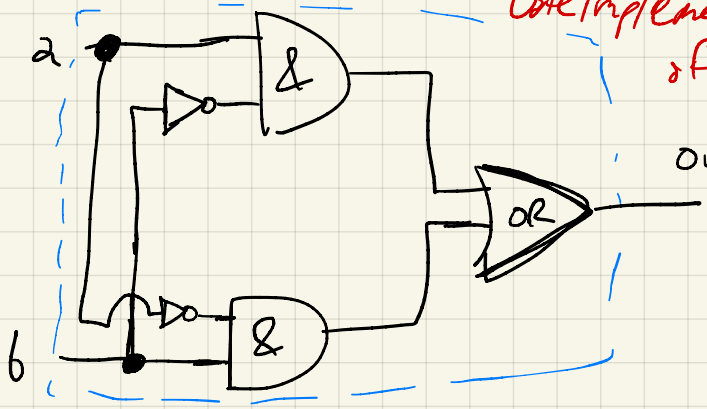
How conceptually it works.

Gate Interface of XOR is unique...



a	b	f
0	0	0
0	1	1
1	0	1
1	1	0

Gate Implementation of XOR



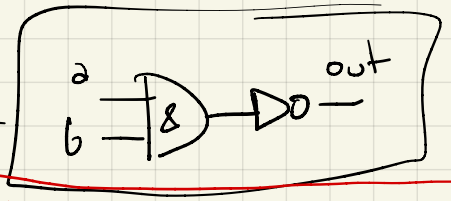
HDL = Hardware Description Language

VHDL → Virtual

★ Page 16 has HDL Code Example for XOR Gate! ★

1.2.1 – NAND Gate

a	b	NAND(a,b)
0	0	1
0	1	1
1	0	1
1	1	0



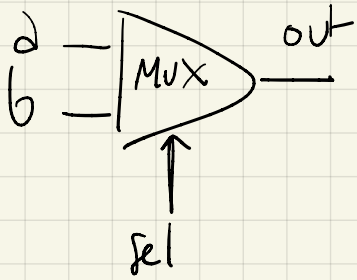
"Primitive" Chip (No need to implement)

"The starting point of our computer architecture is the NAND Gate
↳ From which ALL other gates & chips are built."

→ NOT Gate known as "Converter"

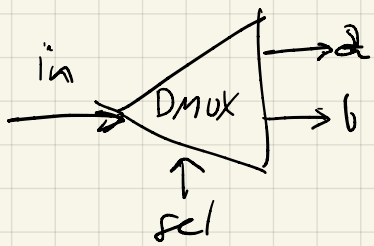
→ Multiplexor ("MUX") → 3 input gate to select an input as output.

↳ i.e. → If selector bit = 0, out = a, else out = b



Demultiplexor (DEMUX) →

sel	a	b
0	in	0
1	0	in



1) Multibit Gates are n-bit versions of primitive gates that still perform the same function.

* Not16 → 16 input converter
* And16 → 2*16 inputs, 16 pairs of AND Gates. Requires 16 inputs.

etc., etc.

* Mux16 → Same as binary Mux except with 16 pairs of inputs but still a single selector.

2) Multi-Way → generalizes gates to n-inputs to 1 output.

↳ Ex: n-way OR Gate → outputs 1 when at least 1 input of n bits = 1.

↳ An m-way, n-bit MUX selects one of m n-bit input buses & outputs it to a single n-bit output bus. The selection is specified by a set of "k" control bits, where $k = \log_2 m$ (see fig. 1.10, pg. 24)

→ The platform developed in this book requires two variations of this chip:

A 4-way 16-bit MUX & an 8-way, 16-bit MUX.

We must construct all gates from NAND.

NOT
AND
OR/XOR
MUX/DMUX
Multibit NOT/AND/OR
Multibit MUX
Multiway Gates.

Chapter 1 Project

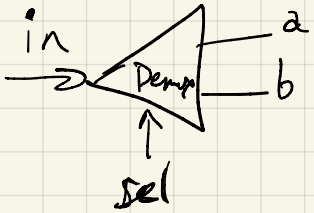
→ Construct ALL Gates from chapter 0/NAND.

1) Read Appendix A, Sections A1-A6 only.
2) Go through Hardware Simulator Tutorial, parts I, II, & III only.
3) Build & simulate all chips specified in projects/01 directory.
   - 1) ~~AND~~, ~~AND16~~
   - 2) ~~DMux~~, DMux4way, DMux8way,
   - 3) ~~Mux~~, ~~Mux4way16~~, ~~Mux8Way16~~
   - 4) ~~Not~~, ~~Not16~~
   - 5) ~~Or~~, ~~Or8way~~, ~~Or16~~
   - 6) ~~XOR~~

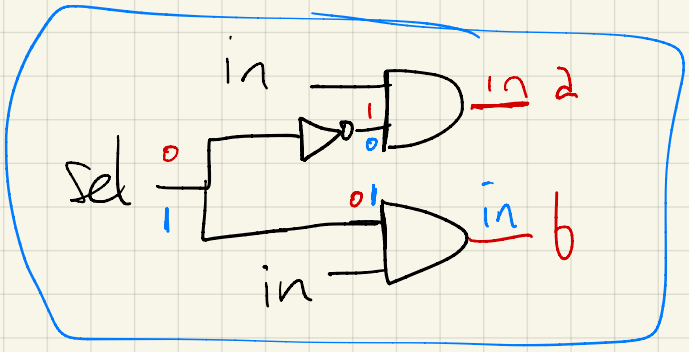
~~Not~~, ~~And~~, ~~Or~~, ~~Xor~~, ~~Mux~~, ~~DeMux~~, [Multi bit ~~Not~~, ~~And~~, ~~Or~~], Multibit Mux,
Multi-Way Gates [~~Not16~~, ~~And16~~, ~~Or16~~, Mux 8way16, DMux8way]

Demux:

sel	a	b
0	in	0
1	0	in



Or(a,b)		Nand(a,b)		And(a,b)		Xor(a,b)	
a	b	f	a	b	f	a	b	f	a	b	f
0	0	0	0	0	1	0	0	0	0	0	0
0	1	1	0	1	1	0	1	0	0	1	1
1	0	1	1	0	1	1	0	0	1	0	1
1	1	1	1	1	0	1	1	1	1	1	0



Multibit Basic Gates (Not, And, Or)

1) Multibit Not

Chapter 2 Notes – 8 Aug 2020

→ Goal of chapter 2 is to fully implement a functional ALU.

↳ Context → Adder Chips

★ Binary Addition →

```
  0 0 0 1   ← carry flow
    1 0 0 1
+   0 1 0 1
-----------
  0 1 1 1 0
```

1+1 = 0 (+1 carry)

since = 0 → ∴ No overflow

registers involved

carry: 1+1+1 = 1+(1)

```
  1 1 1 1
    1 0 1 1
+   0 1 1 1
-----------
1 0 0 1 0
```

overflow!

Signed Binary Numbers → Most significant Bit = negative connotation.

	Positive #'s	Negative #'s (Signed)	
0	0 0 0 0	1 1 1 1	−1
1	0 0 0 1	1 1 1 0	−2
2	0 0 1 0	1 1 0 1	−3
3	0 0 1 1	1 1 0 0	−4
4	0 1 0 0	1 0 1 1	−5
5	0 1 0 1	1 0 1 0	−6
6	0 1 1 0	1 0 0 1	−7
7	0 1 1 1	1 0 0 0	−8

2's Complement (Radix Complement)

$$\bar{x} = \begin{cases} 2^n - x & \text{if } x \neq \phi \\ \phi & \text{otherwise} \end{cases}$$

Example:

−2 for 5-bit system

↳ $\bar{-2} = 2^5 - 2_{10}$ ↓

$= 32_{10} - 2_{10} = 30_{10}$ ↓

$30_2 = 11110_2 = -2_{10}$

To check, $-2_{10} + 2_{10} = \phi_2$

$11110_2 + 00010_2$ ↓

```
  1 1 1 1 0
  1 1 1 1 0
+ 0 0 0 1 0
-----------
1 0 0 0 0 0   ← 5-bit system, = 0!
```

01

10

overflow is ignored b/c book said so...

Quick Method to achieving negative number in binary:

★ MSB = 1, invert all other bits, then add 1. ★

Subtraction is just adding negative # → $x - y = x + (-y)$

Adders: 1) Half Adder: Designed to add 2 bits.
2) Full Adder: " " " 3 bits.
3) Adder: " " " n bits.

Half-Adder Chip:

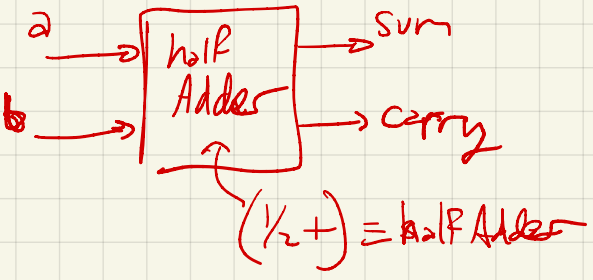
IN = a, b

OUT = sum, carry

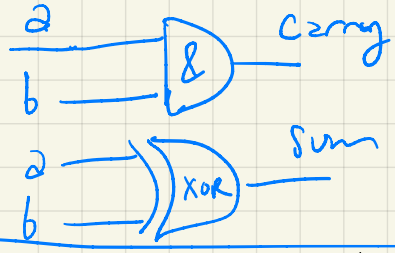
↳ fn: LSB of a+b = sum
MSB of a+b = carry

a	b	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

AND XOR



← halfAdder!



Full-Adder Chip:

IN = a, b, c

OUT = sum, carry

↳ fn: sum = LSB of a+b+c
carry = MSB of a+b+c

a	b	c	Y	Σ
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

Half Adder

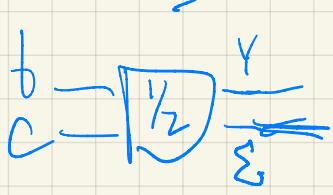
0	0	0	0 X = X
0	1	1	

a Σ	Y Σ
1 0	0 1
1 1	1 0
1 1	1 0
1 0	0 1

do-over.

a+b+c = (b+c) + a

↳

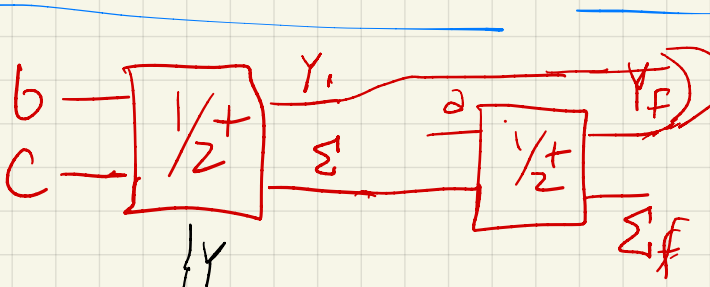


if a = 0 → Σ = sum

if a = 1 → Σ = Not sum

Full Adder via Half Adders: a+b+c (3bit)

$\Sigma \equiv$ sum, $Y \equiv$ carry



Y_f ~~MSB wrong → doesn't matter!!~~

a	b	Carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

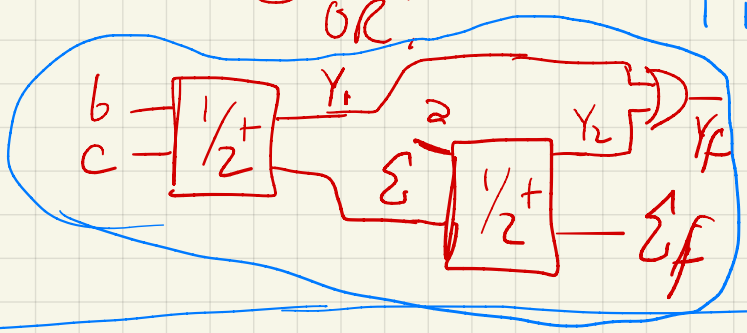
AND XOR

a	Σ	Y	Σ
1	0	0	1
1	1	1	0
1	1	1	0
1	0	0	1

MSB wrong??

Y
0
0
0
1

OR



16 Bit Adder:

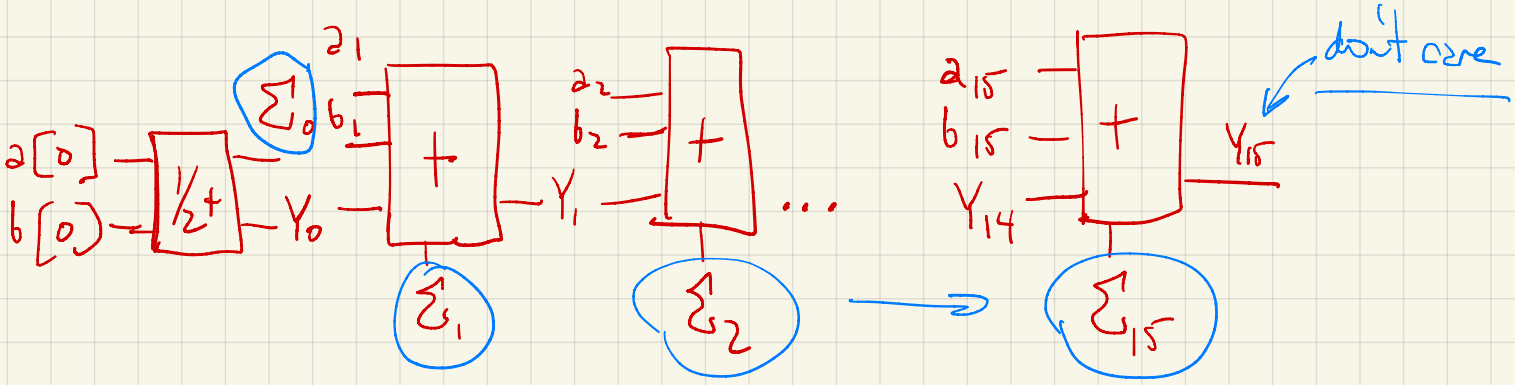
IN => a[16], b[16]

OUT => out[16]

fn: out = a+b

Integer 2's compliment Addition
overflow in either not detected nor handled.

from §2.3 (pg 38) → the LSBs are added, & the carry fed into the addition of the next pair of significant bits. (full adder; i.e 3bits).



Incrementer, 16-Bit:

IN => in[16]

OUT => out[16]

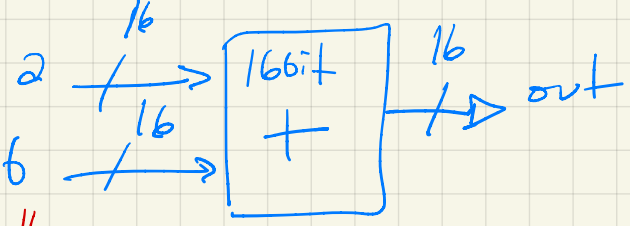
fn: out = in + 1

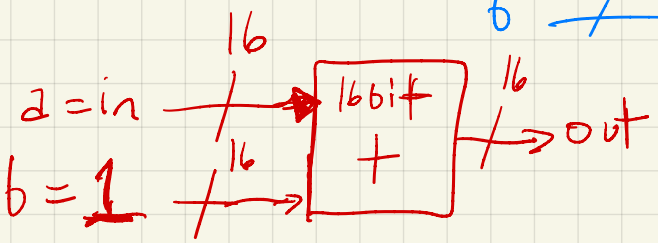
Integer 2's complement addition.

Overflow neither detected nor handled.

§2.3 (pg. 38) → "n-bit incrementer can be trivially implemented from an n-bit Adder."

Add16 logic:





ALU:

IN => x[16], y[16]

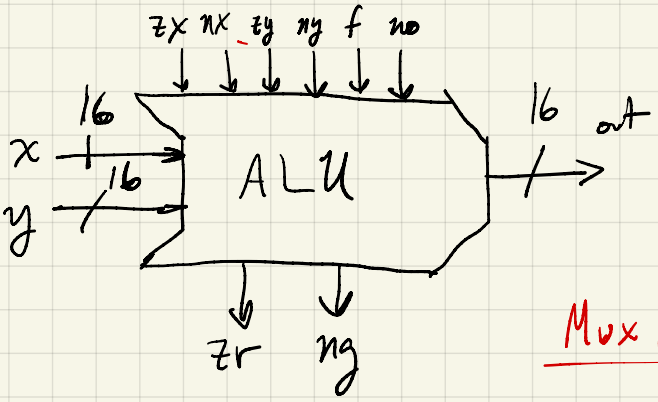
6 control bit inputs:

- zx → zero x-input
- nx → negate x-input
- zy → zero y
- ny → negate y
- f0 → 0 = And, 1 = Add
- no → negate output.

OUT => out[16]

- zr: 1 → out = 0; 0 → out ≠ 0 ← 16-bit eq. comparison
- ng: 0 → out ≥ 0; 1 → out < 0 ← 16-bit negative comparison

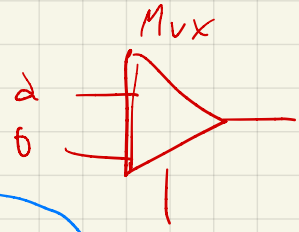
§2.3 (pg. 39): "Our first step will likely be to create a logic circuit that manipulates a 16-bit input according to zx + nx (+ y). Then integrate the "f" bit functionality.



8 inputs, 3 outputs

→ two 16bit / one 16bit.
6 bits / 2 bits.

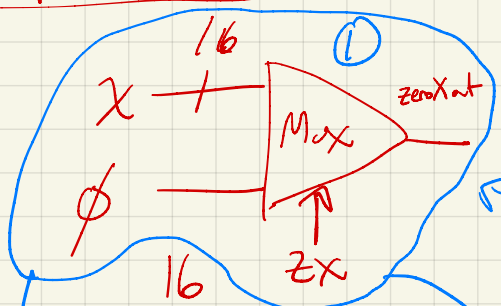
sel	out
0	a
1	b



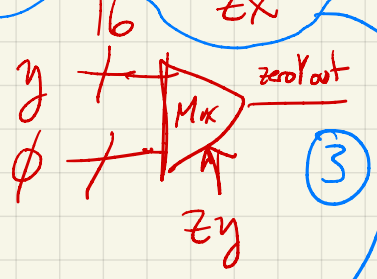
Mux per x or y??

1) construct zero x ✓
2) construct neg. x ✓
3) zero y ✓
4) neg y ✓
5) x+y ✓
6) x·y ✓
7) not output

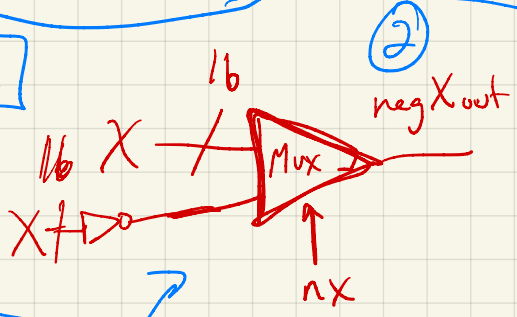
8) zr check
9) ng check



zx	out
0	x[16]
1	0



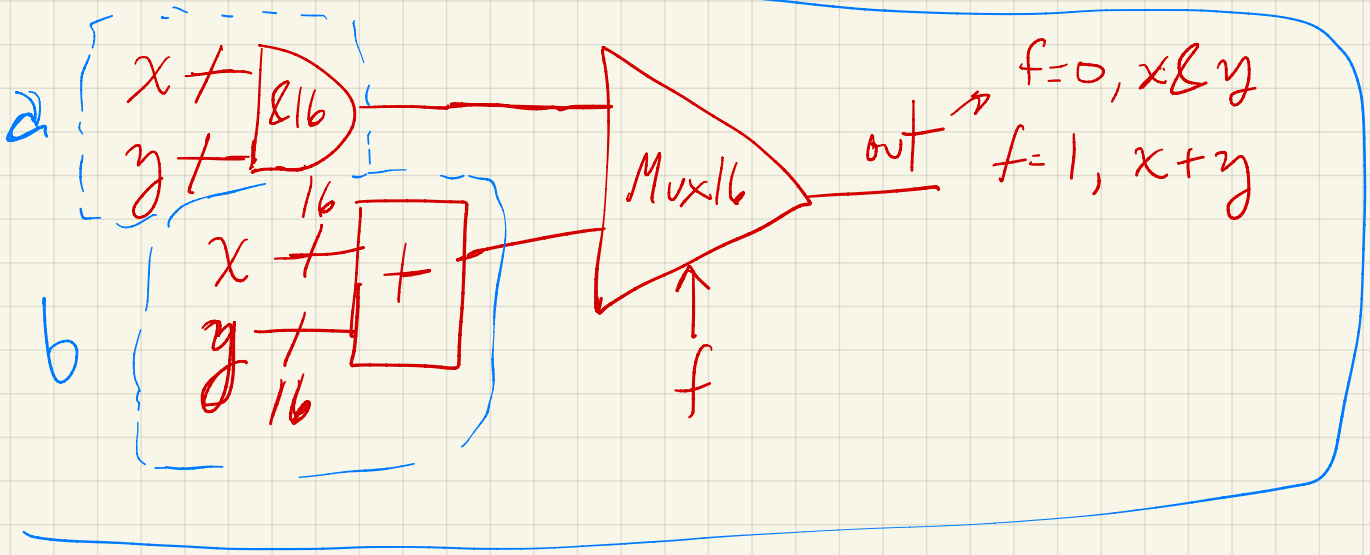
will need to keep track of each OUT during intermediate



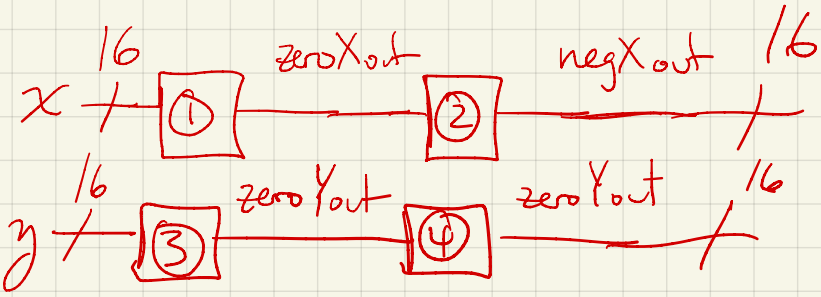
5 & 6

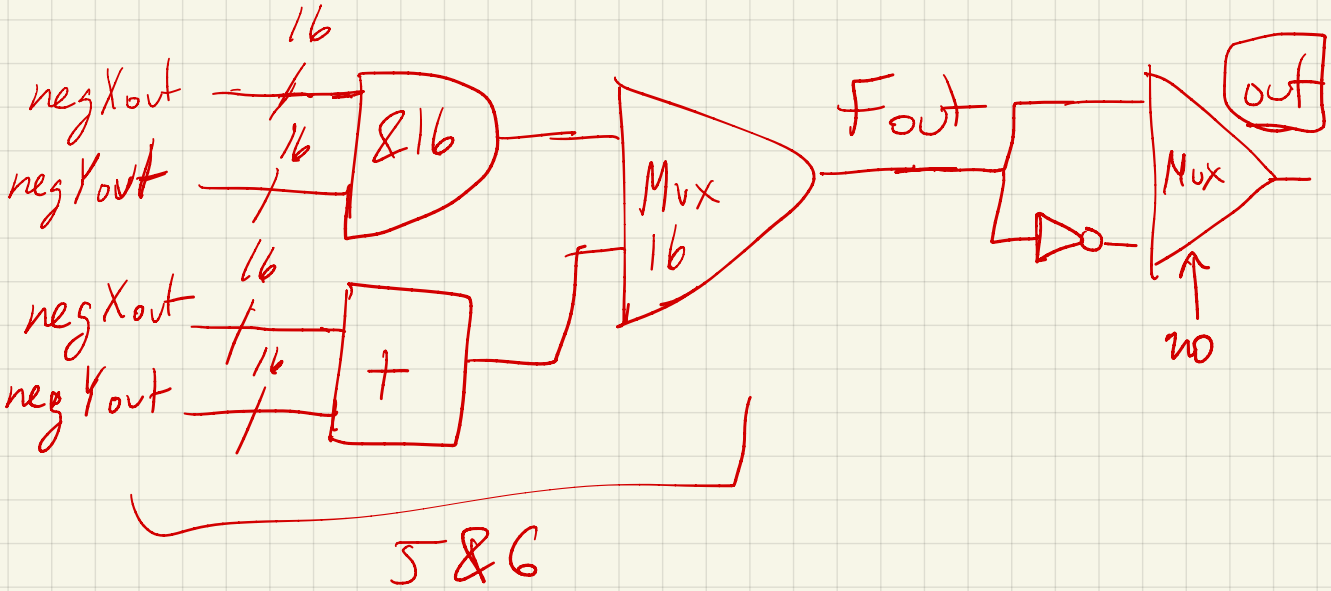
if f = 0 → x & y

f = 1 → x + y



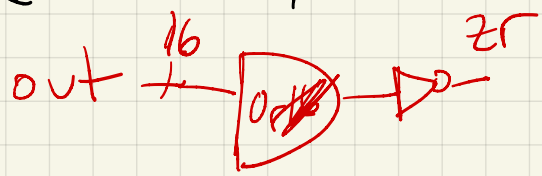
Chip 1 = zero x
2 = neg x
3 = zero y
4 = neg y
5&6 = f
7) not out

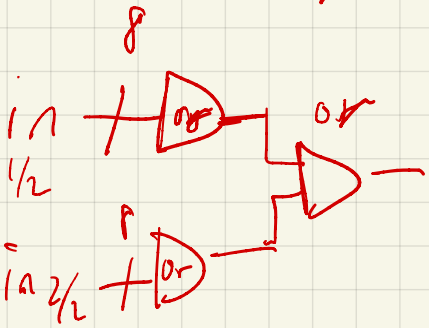




zr:

$$zr \begin{cases} 1 & out = 0 \\ 0 & out \neq 0 \end{cases}$$





ng: (MSB = 1 or not)

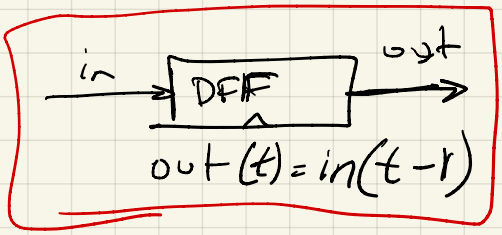
$$ng \begin{cases} 0 & out \geq 0 \\ 1 & out < 0 \end{cases}$$

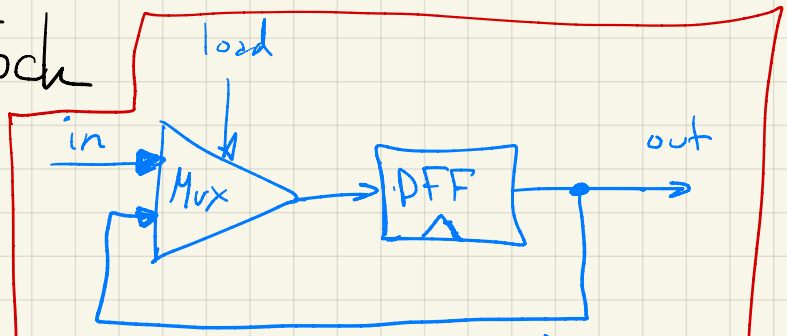
out[15] → ng

Ch. 3 Notes — 15 Aug 2020

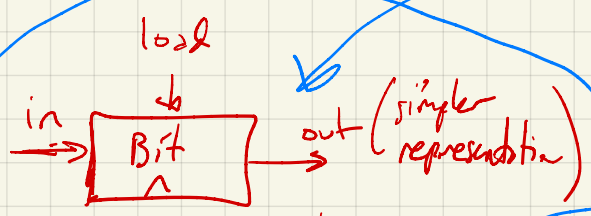
Flip-Flops — basis of our clock. (Verse DataFlipFlop, aka DFF)

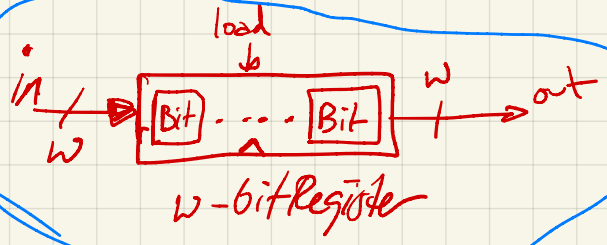
ClockCycle → Tick-Tock

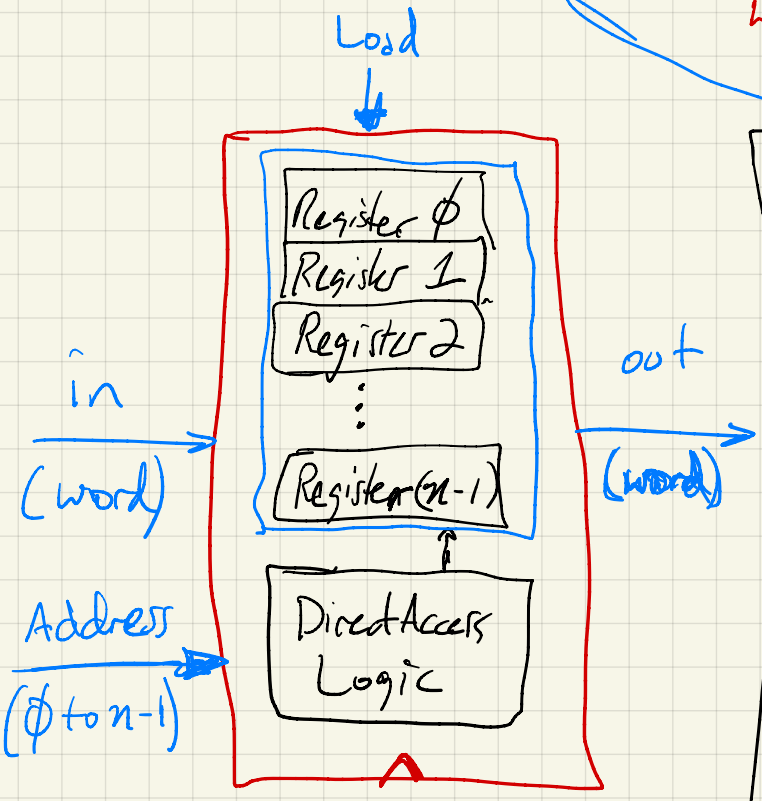




if load(t-1) then out(t) = in(t-1)
else out(t) = out(t-1)







RAM Chip Concept Model

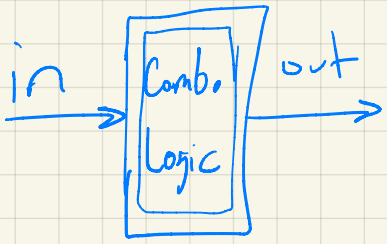
Counters — out(t) = out(t-1) + C, where C = 1 usually..

CPUs include program counters whose output is interpreted as the address of the instruction that should be executed next in the current program.

Sequential vs Combinational Chips

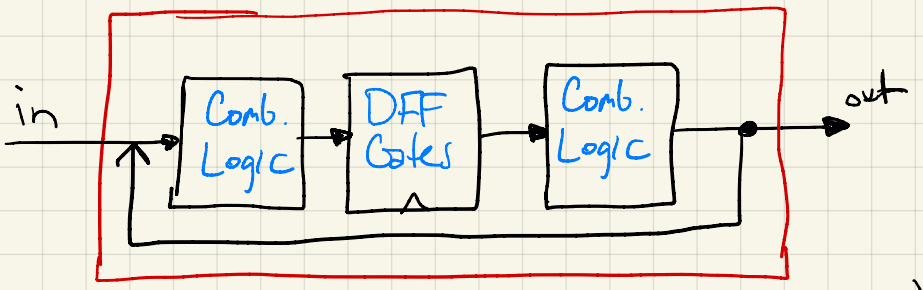
↳ Sequential chips always consist of a sandwiched layer of DFFs btw optional combinational logic layers.

Combinational Chip (pg. 46)

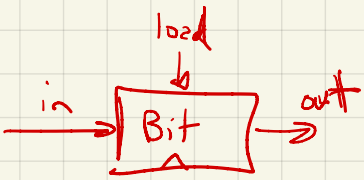


out = some function of (in)

Sequential Chip (pg. 46)



out(t) = some function of (in(t-1), out(t-1))



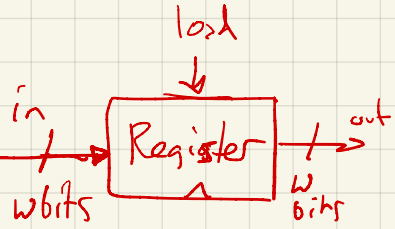
Bit

IN: in, load

OUT: out

fn: If load(t-1) then: out(t) = in(t-1)
└ else: out(t) = out(t-1)

} "=" is a 16-bit operation



Register (16-bit)

IN: in[16], load

OUT: out[16]

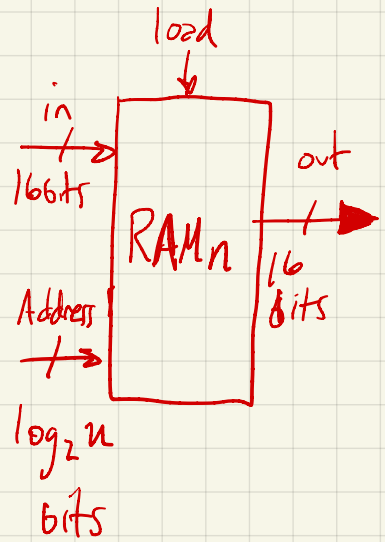
fn: If load(t-1) then: out(t) = in(t-1)
└ else: out(t) = out(t-1)

} "=" is a 16-bit operation

Register is just w-bits wide!

Memory: An array of n w-bit registers equipped w/ direct Access Circuitry.

The # of registers (n) & the width of each register (w) are called the memory's **size** & **width**, respectively.



RAMn:

IN: in[16], address[k], load

OUT: out[16]

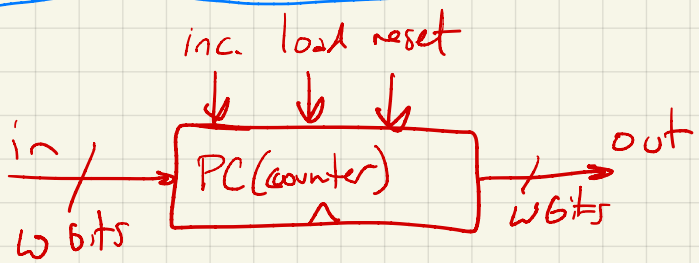
fn: out(t) = RAM[address(t)](t)

IF: load(t-1), then
└ RAM[address(t-1)](t) = in(t-1)

"=" is a 16-bit operation

The specific RAM chips needed for the Hack platform are:

ChipName	n	k
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14



PC (16-bit Counter) (w=16):

IN: in[16], inc, load, reset

OUT: out[16]

fn: If: reset(t-1) true, then out(t) = 0
↳ elseif load(t-1) true, then out(t) = in(t-1)
↳ elseif inc(t-1) true, then out(t) = out(t-1) + 1
↳ else out(t) = out(t-1)

To Build!

1-Bit Register (Bit)

16-Bit Register (Register)

RAM8, RAM64, RAM512

RAM4K, RAM16K

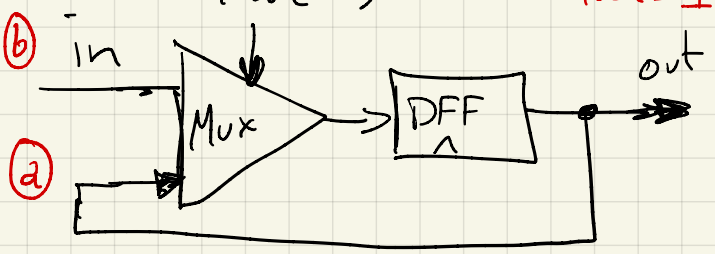
16-Bit Counter.

1-bit Register (pg. 43)

load (sel) ← b/c

load = 0 → feedback loop

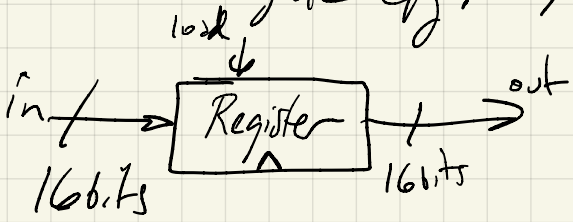
load = 1 → use new input!

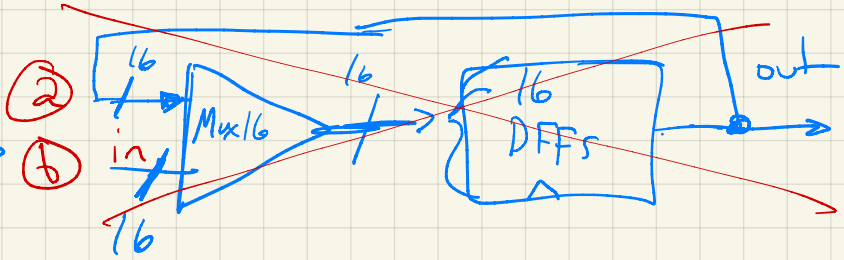


To Build!

- ✓ 1-Bit Register (Bit)
- ✓ 16-Bit Register (Register)
- ✓ RAM8, RAM64, RAM512
- ✓ RAM 4K, RAM16K
- → 16-Bit Counter.

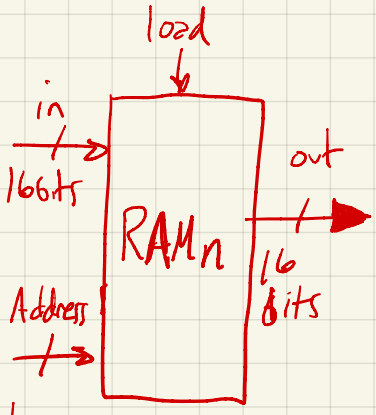
16-bit Register (pg. 49)





Use 16 Bits from above...

RAM8: → RAM8 is 8 16-bit registers...



$\log_2 8 = 3$

smells like a Mux...

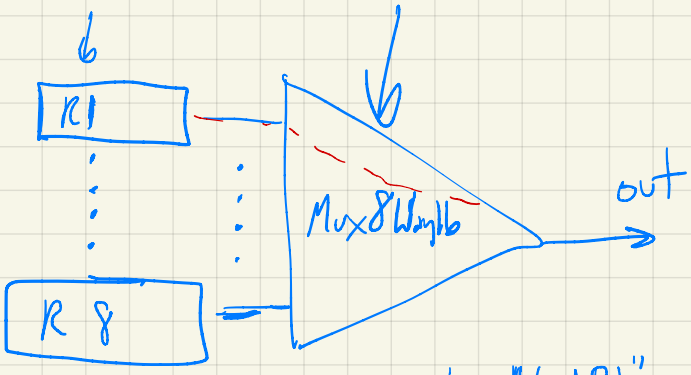
Mux8Way16

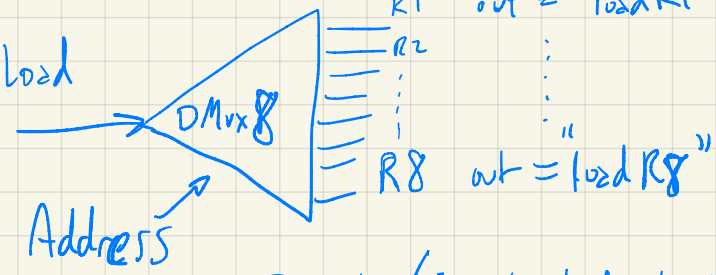
load = false??

Address

Address[0] → sel[0]

Address[2] → sel[2]





Address chooses Register to be written...

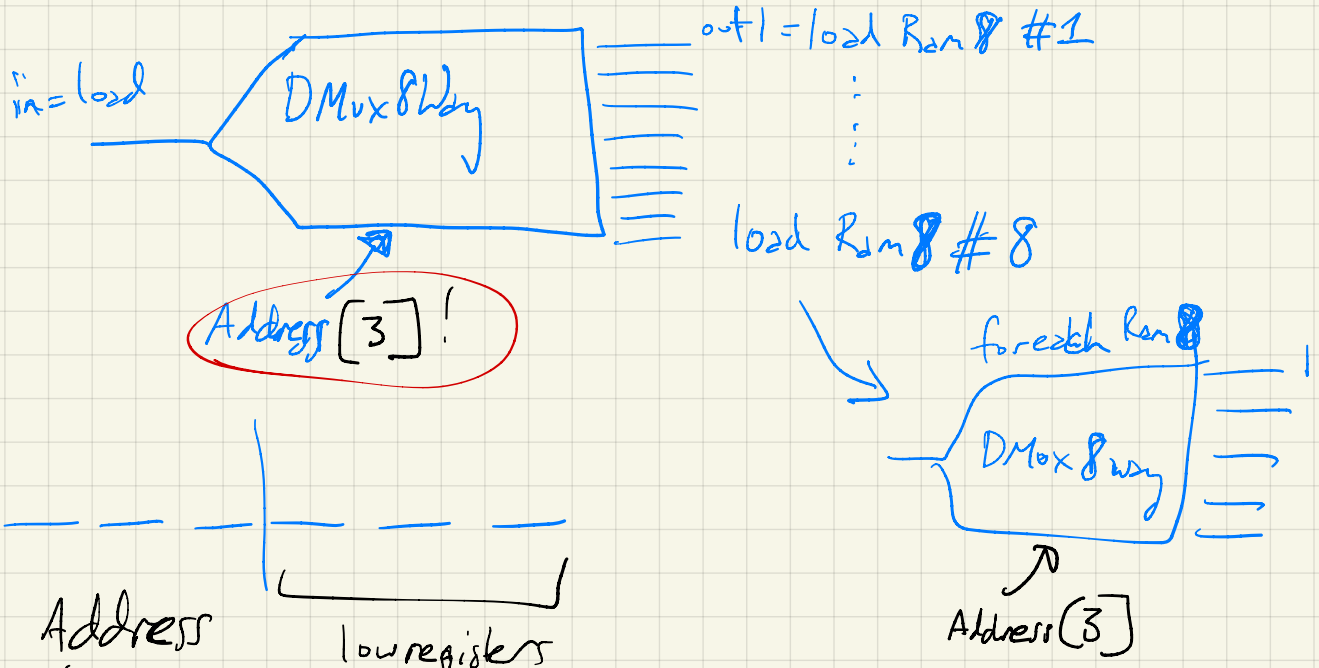
Load tells register to pick new input.

Register(in = in, load = loadRX, out = outToMuxX)

Mux8Way16(_____)

RAM64: Array of 8 RAM8 Chips? But how to select??

Address = 6 bits! ⇒ Address[6]

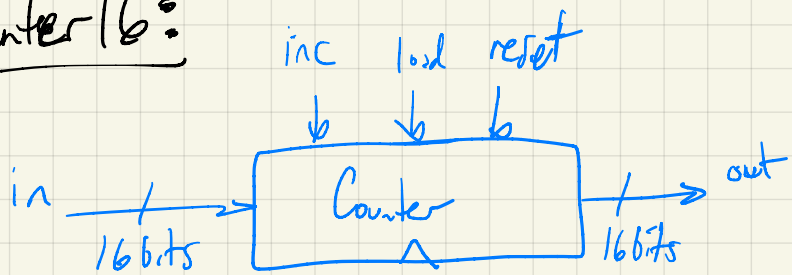


↳ Address is already used for RAM8 → Make sure no overlap??

↳ DMux can only take a 3-bit Address, therefore choose Largest Bits to choose RAM8!

↳ Similar Architecture to RAM8... → Same philosophy for 512, 4K, 16K.

Counter16:



Register will be end result.

if reset = 1,
↳ set out = 0.

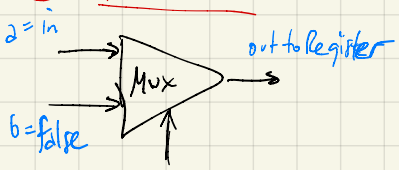
if load = 1,
↳ set out = in

if inc = 1
↳ out = out + 1

Mux →

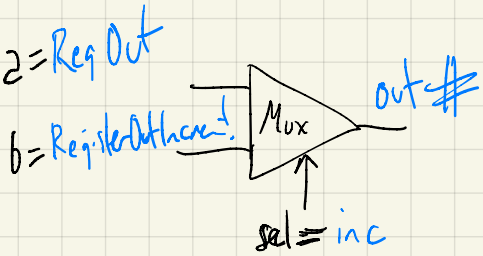
sel	out
0	a
1	b

③ Reset Control



Mux16 or no??

① Increment Control



② Load Control