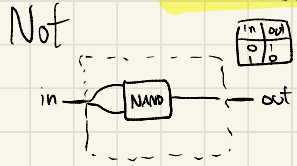
Project 1 Scratch

Elementary chips

Not

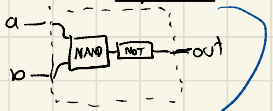


a	b	NAND
0	0	1
0	1	1
1	0	1
1	1	0

only dealing w/ these situation

And

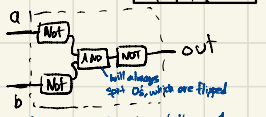
a	b	out
0	0	0
0	1	0
1	0	0
1	1	1



basically opposite of NAND

Or

a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1



will always spit OR, inputs are flipped

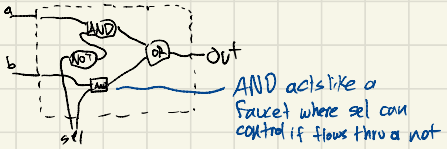
AND: always 0 unless both are 1
OR: always 1 unless both are 0

XOR

See 1.4

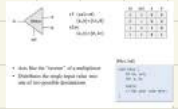
Mux gate → if gate!

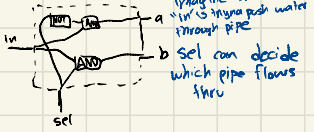
sel	out
0	a
1	b



AND acts like a faucet where sel can control if flows thru a not

DMux





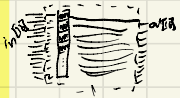
imagine in=1 means "in" is tryna push water through pipe

sel can decide which pipe flows thru

16-bit chips – Each bit processed independently

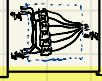
Not16

→ just a bunch of Not chips, 1 per bit



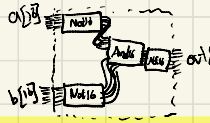
And16 Mux16

→ same as Not16, many elementary chips



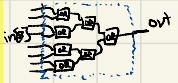
Or16

→ Can be made w/ Not16 and And16 chips OR Or chips



Multi-Way Chips – bit values processed together

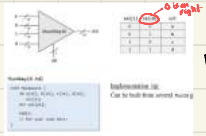
Or8Way

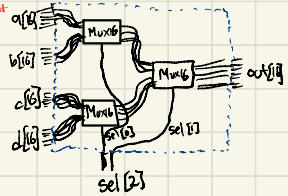


→ ORing values together
– if a single bit is 1 in input, out is 1

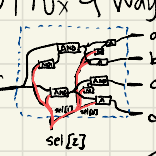
Mux 4 way 16

→ 4, 16-bit inputs
→ selector bus is 2-bit (need 2 bits to select from 4 possibilities)



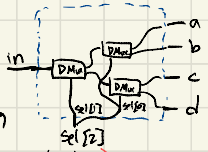


DMux 4 Way



sel[2]

complex way

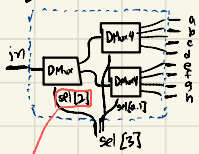


Sel[2] better way

2nd bit first

sel[1]	sel[0]	in
0	0	a
0	1	b
1	0	c
1	1	d

DMux 8Way



Sel[3]

sel[2]	sel[1]	sel[0]	out
0	0	0	a
0	0	1	b
0	1	0	c
0	1	1	d
1	0	0	e
1	0	1	f
1	1	0	g
1	1	1	h

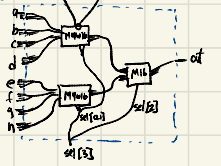
use last bit of selector first

split into 2, then 4s

Mux 8way16

→ Same as above but 8 choices

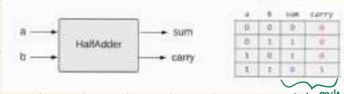
sel[2]	sel[1]	sel[0]	out
0	0	0	a
0	0	1	b
0	1	0	c
0	1	1	d
1	0	0	e
1	0	1	f
1	1	0	g
1	1	1	h



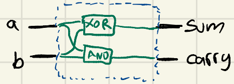
Project 2

Half Adder

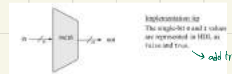
input two bits
outputs sum and carry bit



Hint, each column identical to output of elementary gate
XOR AND



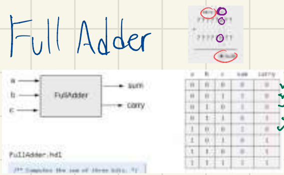
16 bit incrementor



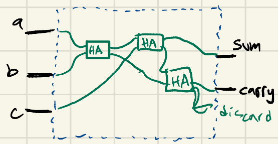
Single input, adds 1 to it, outputs value
→ add true w/ 16-bit adder?

cando
b[0]=true
b[1..15]=false } = 0000000000000001

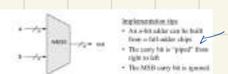
Full Adder



✓Can build from
✓2 half adders
+
✓ bit more functionality



16 bit adder

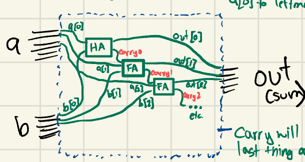


16 full adders

→ Now full adders
▿ Know each step, now just need to do repeatedly

▿For 16-bit adder, simply need
1 half adder (first step)
15 full adders
for each colored step

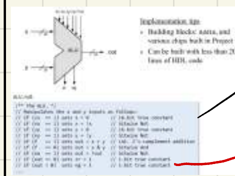
need to go from rightmost bit (LSB) Least sig. bit
a[0] to leftmost



Carry will be last thing out

more efficient implementation would precalculate carry — carry look ahead "carry prediction"
— Problem — need to calculate all carries in order to get last carry

ALU

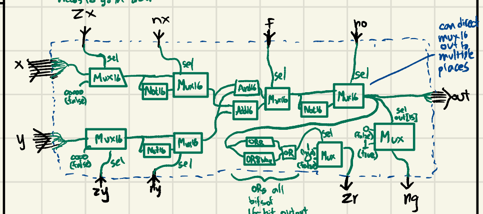


Mux gates are good for "if" statements

→ Mux16 is one of the few chips so far that takes 1-bit & 16bit at same time

to check negative
→ check if MSB is 1
→ if input[15]==1

needs to go in order



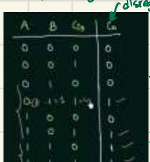
can direct mux16 out to multiple places

Possibilities w/ ALU

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	nx	zy	ny	f	no	out
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Predict Carry?

disregard sum for now



if A is diff from B (A xor B) then carry is 1 if carry$_{in}$ = 1

when A & B are both 1, carry is 1 regardless of carry$_{in}$

C_0 = (A AND B) OR ((A XOR B) AND C_{in})

carry generator — A/B "generate" carry — "G"
"carry propagate" — propagates C_{in} forward — "P"

C_0 = G OR P AND C_{in}

Also, $C_{in} = C_{0-1}$ (carry before)

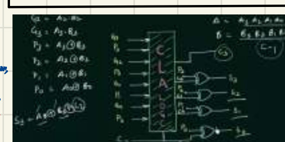
$C_i = G_i + P_i C_{i-1}$

i=0
$C_0 = G_0 + P_0 C_{-1}$ — basically 0

i=1
$C_1 = G_1 + P_1 C_0$
$C_1 = G_1 + P_1 (G_0 + P_0 C_{-1}) = G_1 + P_1 G_0 + P_1 P_0 C_{-1}$

i=2
$C_2 = G_2 + P_2 C_1$
$C_2 = G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_{-1})$
...etc.

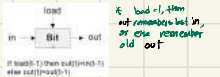
→ As can be seen, w/ equation, any carry can be computed w/ first A and B values
→ Don't need to sequentially compute carries to get any carry, also have all Ps/Gs
→ can compute adds in parallel



Project 3 - Memory and RAM

1-bit register

Goal: Remember bit 'forever' until asked to load new value



ex: load=1, then out remembers last in, or else remember old out

Can implement w/ Mux chip

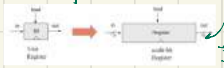
↻ → keeps looping value when Mux changes w/ load

DFF
- → only stores info for 1 time unit
- → always stores "it" bit

Bit chip
- → stores for many cycles
- → only stores "in" if load=1

16-bit Register

- Most basic memory element: Register → from multiple single bit registers
- → can easily make multi-bit



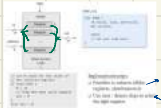
(w can be 16, 64, 32-bit (width))

→ block comp is 16-bit

→ "state" = value stored in register

- User perspective
- → to READ: probe out
- → to Write: set in=V, set load = 1
  - → from next cycle onwards emits V

RAM8

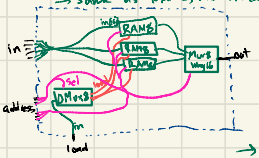


- → sequence of n addressable registers, addresses 0 to n-1
- → basically array of registers
- → at any given time, only focus on one register, based on address

- → in[16] sent to all registers
- → DMux decides via address where load is sent → place where it is sent will get value of in if load=1
- → All Registers output/emit to Mux, only one chosen based on address

RAM64

- → 8 RAM8s
  - → same as RAM8, with RAM8s instead of registers

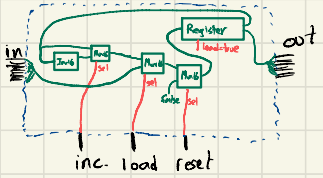


→ All following are progression of previous one

- → 8x RAM512
- ↓ 8x RAM4k
- ↓ 4x RAM16K

Program Counter

ctrl bits: 4 most important things
- → set 0
- → set to value
- → change counter value
- → stop counting

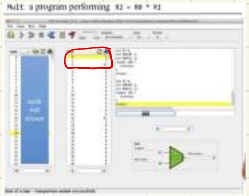


- → if reset = 1, PC emits 0
- → else if load == 1, out[t+1] = in[t] (setting counter value)
- → else if (inc == 1), counter increments
- → else out[t+1] = out[t] (counter won't change)
- → for if, else, use Mux chips → b=1, a=0
- → do last else if first

Project 4

Mult

▽ program that multiplies



→ Machine language does not have mult

→ need to use loop, and addition/subtraction

R2 = R0 * R1

Pseudocode:

```
int i = 0
int mult2 = R1
int mult1 = R0
int product, R2 = 0
for (int i=1 ; i ≤ R1; i++){
    product += R0
}
```

```
@i
M=0
@R0
D=M
@mult1
M=D
@R1
D=M
@mult2
M=D
@product
M=0
@R2
M=0
(LOOP)
@mult1
D=M
@i
D=D-M
@STOP
D;JLE
@mult2
D=M
@product
M=D+M
@i
M=M+1
@LOOP
0;JMP
(STOP)
@product
D=M
@R2
M=D
(END)
@END
0;JMP
```

Checks if counter == R1

if R1 - counter ≤ 0

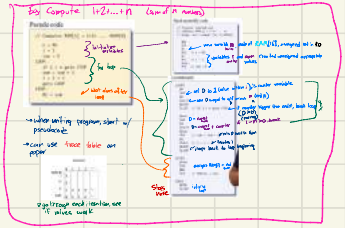
i

7 - 2 ≤ 0

product = product + R0

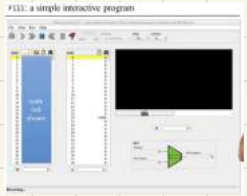
iterating counter

unconditional JUMP to beg. of loop



Fill

▽ if any key is pressed, screen turns black, when lifted, screen goes back to normal



→ infinite loop that listens to keyboard

1) Must probe keyboard

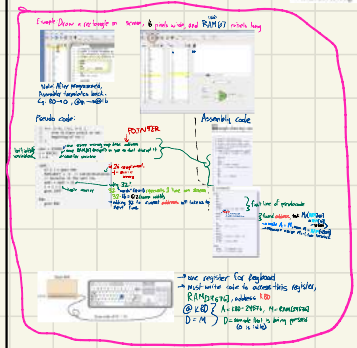
2) Blacken or whiten screen (entire memory map)

Implementation strategy

- Listen to the keyboard
- To blacken / clear the screen, write code that fills the entire screen memory map with either "white" or "black" pixels
- Addressing the memory requires working with pointers

Testing

- Select "no animation"
- Manual testing



→ infinite loop that probes keyboard

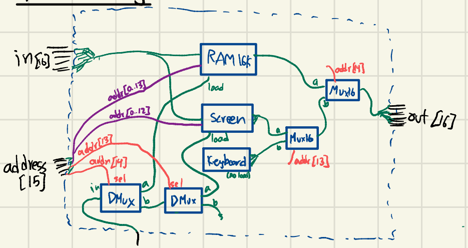
```
addr = SCREEN
code = 0
lastScreenWord = 24575
(PROBE)
→ Get scancode of keypress
→ if code changes (called or released)
    ▽ FILLSCREEN
→ else
    ▽ LIGHTENSCREEN
repeat PROBE
(FILLSCREEN)
(LOOP)
if addr ≥ lastScreenWord, goto STOP
RAM[addr] = FLIP BITS
addr = addr+1
goto LOOP
(STOP)
reset addr
→ go to PROBE
```

(next RAM loc in screen)

```
@SCREEN
D=A
@addr
M=D
@
```

Project 5

Memory



address 01100 00000 10110 — $0-2^{15} \Rightarrow 0-32768$

First 14-bits → $0-2^{14} \Rightarrow 0-16384$ (RAM)

15th bit → $2^{14}-2^{15} \Rightarrow 16384-32768$

if 15th bit = 0, then accessing RAM
if 15th bit = 1, then maybe screen/KBD

address 11100 00000 10110

15th bit = 1 → $2^{14}-2^{15} \Rightarrow (16384-32768)$

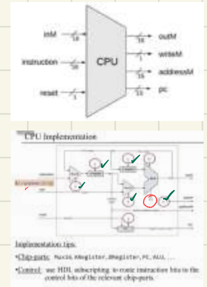
14th bit → $2^{14}+0\cdot2^{13} = 16384$, $1\cdot2^{13} = 24576$ (Screen)

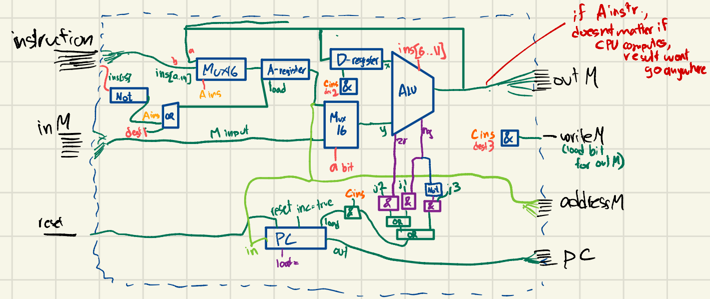
if 15th bit = 1, and 14th bit = 0, then accessing SCREEN

if 14th bit = 1, then accessing KBD

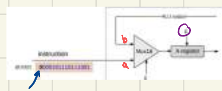
24576

CPU





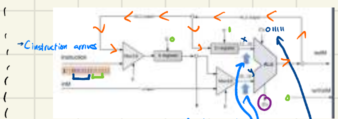
→ Need to take control bits, maybe do processing, send to various parts



if op code 0 = A instruction, A reg takes instruction
if op code 1 = C instruction, A reg takes ALU output (if dest[2]==1)

A ins OR dest[2] = C

A reg



ALU inputs
→ From D-register
→ From A-reg/M-reg

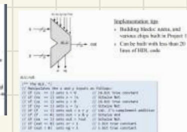
ALU control inputs
→ control bits from instructions

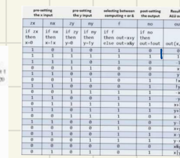
ALU outputs
→ Result of ALU calc.
→ fed to D-reg/A-reg
→ fed outside to M-reg

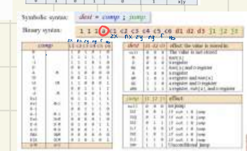
Fed into all 3, but which one recieves is determined by destination bits
0 1 0
A D M

ALU control outputs
→ ng (negative output?)
→ zr (zero output?)

Dreg
→ only takes if C ins and if dest bit = 1

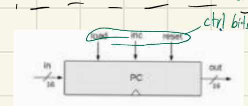






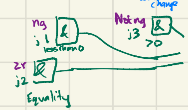
M reg
→ replaces A if a bit = 1

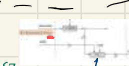
ctrl bits

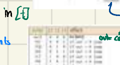


if load = 1, then out = whatever in A reg. (JUMP)

→ if reset = 1, PC emits 0
→ else if load == 1, out[t+1] = in[t] (setting counter value)
→ else if (inc == 1), counter increments
→ else out[t+1] = out[t] counter won't change

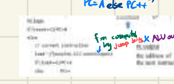






→ Jump bits
eg: 000 - No Jump
111 - unconditional jump
xxx - conditional jump

Program counter does this
→ Emit address of next instruction
• to start/restart → PC = 0
• No jump → PC++
• go to (jump) → PC = A (programmer will have loaded Jump instruction address into A)
• conditional goto → if condition is true, PC = A else PC++



Computer