



Lecture Notes:

- CSE is like an onion, has many layers

ApplicationOperating System

High-level language program

assembly language program

machine code program

instruction set processor

high-level state machine

finite-state machine

sequential logic

combinational logic

Software

Hardware

Boolean
Algebra

Switches

TheoryImplementation

What is a bit? 0 or 1

What is a byte? A group of 8 bits

What is a nibble? A group of 4 bits

* Computers represent everything w/ bits
 i.e. binary

- Operated w/ logic operations

- * - AND, OR, NOT

- XOR, NAND, NOR

Decimal (base 10) and Binary (base 2) :

$$10_{10} = 1 \times 10^1 + 0 \times 10^0 = 10_{10}$$

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

$$10_{16} = 1 \times 16^1 + 0 \times 16^0 = 16_{10}$$

$$101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{10}$$

$$10001 = 1 \times 2^4 + 1 \times 2^0 = 17_{10}$$

note. can skip the 0's

Downsides of binary?

- hard to understand/read
- not as compact
- hard to recognize patterns

→ Solution - represent groups of 4 binary digits as hexadecimal

Lecture Notes:

1.16.25

Symbolic Computing Machines:

- machines that process symbolic information
- Logical View
 - symbol systems: 1, 0; true, false, high voltage, low voltage
 - rules for processing, notation?
 - How is this interpreted - operand
 - what operations can be performed
- Physical View
 - 1 + 0 → transistors (voltage)
 - connect transistors → gates (AND, OR, NOT)

Symbols and Computation:

Consider a function f : takes a bit x as input and returns a bit z as output

$$z \leftarrow f(x)$$

input

x	z
0	0, 1
1	0, 1

exs.

NOT

x	z
0	1
1	0

x	z
0	0
1	0

* one bit not that interesting *

now consider an f that takes 2 bits (x, y) as input ...

AND

$$z \leftarrow f(x, y)$$

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

exs.

* table of all functions in 8 rows *

AND, OR, XOR, NOR, NAND, XNOR

note. NOR is the opposite of OR

NAND is the opposite of AND

"not AND" "not OR"

Basic Boolean Operations: AND, OR, NOT

can create all other logic gates w/ these

AND : 1 if both inputs 1

OR : 1 if either input 1

NOT : 1 if 0, 0 if 1

1/1/true

1/0/false

Ex. The lights are on and somebody is home
 $p \text{ AND } q; p \& q; p \wedge q; (pq)$

Ex. The lights are on and nobody's home
 $p \wedge \neg q; \dots (pq')$

Unsigned Numbers:

all bit positions have positive weight:

$$0100_2 \rightarrow 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 4_{10}$$

computers are finite, must assume N bits of storage:

$N=4$ for now

add unsigned numbers:

$$\begin{array}{r} 0100 \\ + 0001 \\ \hline 0101 \end{array}$$

$$\begin{array}{r} 0011 \\ 0001 \\ \hline 0100 \end{array}$$

overflow: what happens when we need more than our N bits of storage

$$\begin{array}{r} 0100 \\ + 0100 \\ \hline 1000 \end{array} \quad \checkmark$$

$$\begin{array}{r} 1000 \\ + 1000 \\ \hline 10000 \end{array}$$

↑ not enough space, so ignored

Properties of Negative Numbers:

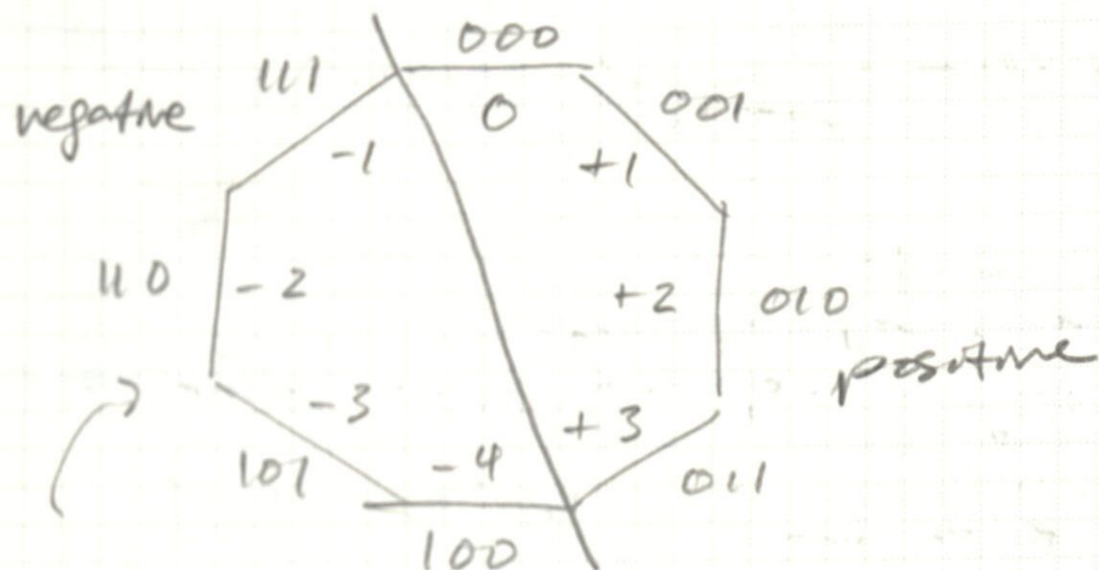
$-13 \rightarrow$ "negative 13"
↑ sign symbol

$$\begin{array}{ccccccc} - & + & - & - & - \\ + & + & + & - & + \\ - & + & + & = & \\ + & + & - & = & \end{array}$$

$>$ depends

Representing Negatives efficiently:

assume $N=3$ bits of storage



Two's complement notation:

Most significant bit (MSB) indicates sign:

- 1 - negative
- 0 - positive

Algorithm:

- The additive inverse is the logical inverse + 1
- To convert, flip the bits, add 1

Ex. $100_2 = (-1 \cdot 2^2) + (0 \cdot 2^1) + (0 \cdot 2^0) = -4_{10}$

Ex. $101_2 = (-1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) = -3_{10}$

Note, leftmost bit is the only bit with a negative weight

Ex. $001_2 \rightarrow 110_2 \rightarrow 111_2$
 $+1_{10} \quad -1_{10}$

Ex. 1001 1010 - 8 bits at storage

could do $(-1 \cdot 2^7) + \dots + 0 \cdot 2^0 = \dots$

much faster:

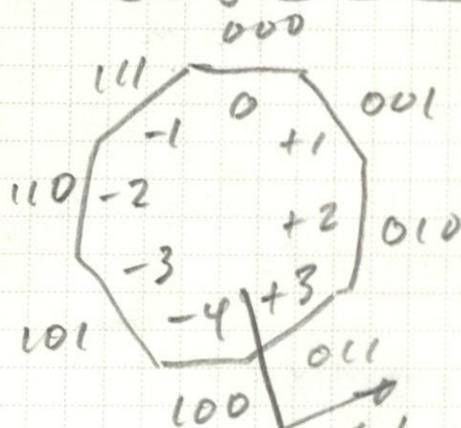
$$\begin{array}{r} 1001\ 1010 \rightarrow 0110\ 0101 \\ + \\ \hline 0110\ 0110 \rightarrow 102 \end{array}$$

which uses original number is -102

- figure out magnitude of two's complement
- already known sign from MSB

note, interpretation of signed vs. unsigned is based on context

Adding signed numbers and overflow:



Assuming 3-bit number
in 2's complement notation;
what is $2_{10} + 3_{10}$?

$$\begin{array}{r} 010 \\ + 011 \\ \hline 101 \end{array} \quad \begin{array}{l} 2 + 3 = -3 \\ 101 \rightarrow -3 \text{ overflow} \end{array}$$

* breaks a property of our negative number

! = carry-in $\rightarrow 000$
carry-out $\rightarrow 101$

$$\begin{array}{r} 101 \\ + 110 \\ \hline 1011 \end{array} \quad -3 + -2 = 3$$

Solution: add a theoretical 4th bit, aka
"carry-out" to fix sign by MSB

overflow: carry-in MSB ! = carry-out MSB

Overflow: Summary

Unsigned Overflow: when adding two unsigned numbers, if the leftmost bit generates a carry bit, overflow has occurred

Signed Overflow: the carry-in to the MSB differs from its carry-out, we need the $(N+1)^{th}$ bit to represent the number correctly

negative + negative = positive

positive + positive = negative

0 0 0 ← cin "carry-in"

ex.

$$\begin{array}{r} 100 \\ + 100 \\ \hline \end{array}$$

→ 1 000, sum

ie. cin
for MSB

!= cout
for MSB

cout
"carry-out"

note, cin and cout is in reference to specific column of addition

note, cin on LSB for addition will always be 0

IEEE Floating Point:

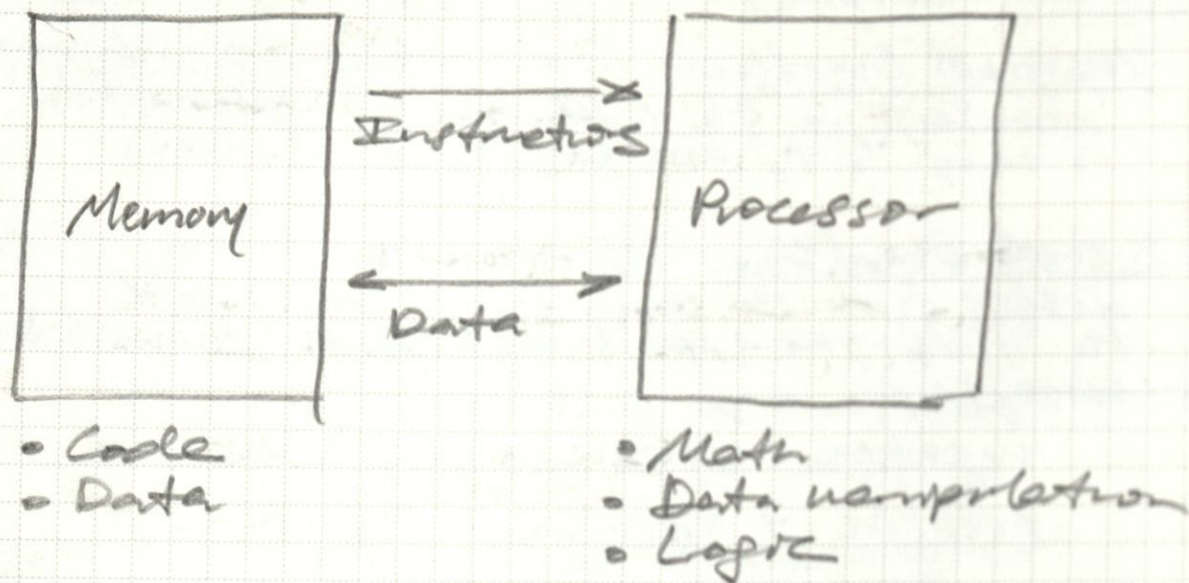
$\sim 10^{\pm 38}$ single precision "float"

$\sim 10^{\pm 308}$ double precision "double"

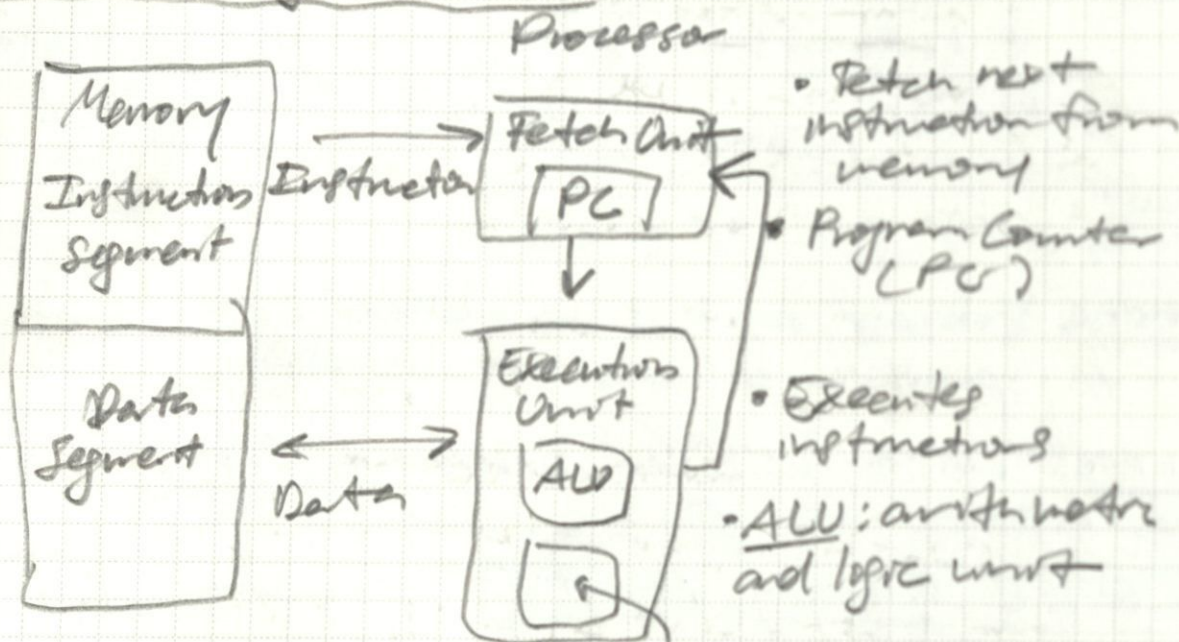
has finite storage $\rightarrow$ cannot represent infinite storage

& can only represent numbers as sums of powers of 2

How do computers work?



Processor Organization:



alpha Core:

- 16-bit processor with full set of instructions
- Can build a computer with it

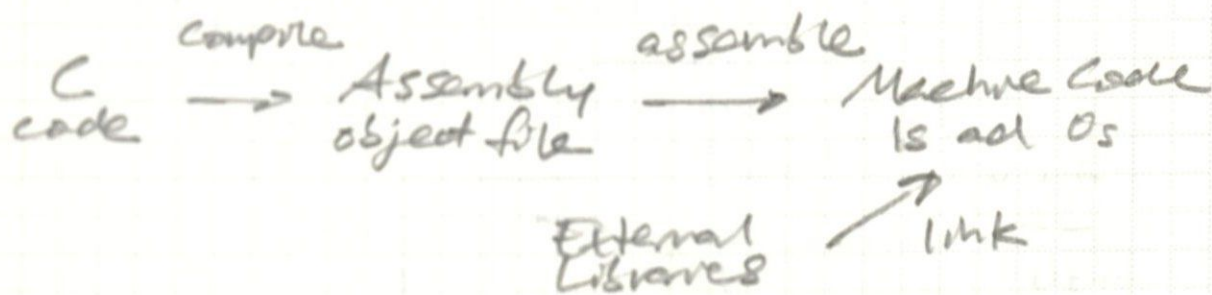
Register File: stores "scratch-work" for ALU computations

Smaller and faster than memory

HLL $\rightarrow$ Machine Code:

HLL: high-level language, like C

Machine Code: is code that computers understand



Instructions:

3 classes of instructions

1. Move Data to code from memory
2. Do arithmetic and logic on memory
3. Make decisions based on that data

Arithmetic / Logic, Data, Branching

alpha cores 16 bit ISA for MD use

ISA: instruction set architecture

$\rightarrow$ describes capabilities

Arithmetic / Logic Instructions:

add r5, r1, r2

destination
(write)

source
(reads)

source
(reads)



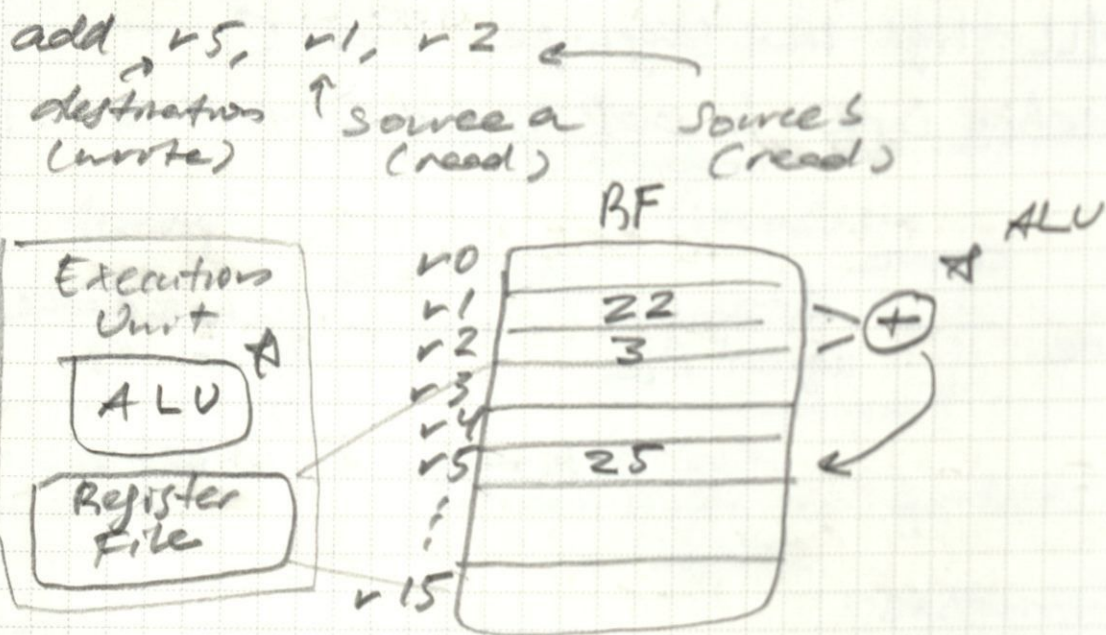
7+12

r0	12
r1	7
r5	19
r15	

note referring to registers: where data is stored

Lecture Notes:

1, 23, 25



alpha Core Arithmetic/Logic Instructions:

add
subtract
and
or
not

shift left
shift right

load immediate

add r_w, r_a, r_b
sub r_w, r_a, r_b
and r_w, r_a, r_b
or r_w, r_a, r_b
not r_w, r_b
shl r_w, r_a, r_b
shr r_w, r_a, r_b
loli r_w, imm

poorly named,
arithmetic
operation

destination
register

$R[r_w] \leftarrow R[r_a] + R[r_b]$
 $R[r_w] \leftarrow R[r_a] - R[r_b]$
 $R[r_w] \leftarrow R[r_a] \& R[r_b]$
 $R[r_w] \leftarrow R[r_a] | R[r_b]$
 $R[r_w] \leftarrow \sim R[r_b]$
 $R[r_w] \leftarrow R[r_a] \ll R[r_b]$
 $R[r_w] \leftarrow R[r_a] \gg R[r_b]$
 $R[r_w] \leftarrow \text{imm}$

load immediate: gets a constant value into a register

Ex. $x = 5$; // ldi r1, 5
(x = 4) (r2)

$x = y + z$ // add r1, r2, r3

$x = x + y - z$ // add r1, r1, r2
// sub r1, r1, r3

note, albaCore is a RISC ISA

note, only have 16 registers. so
range is 0-15

Ex. $z = x + 2$ // ldi r2, 2
// add r3, r1, r2

Arithmetic and Logic Instructions Encoding:

* there exists an albaCore reference
sheet in class google drive
→ given an exam!

instruction

encoding

<u>immediate</u>	<u>op1</u>	<u>op2</u>	<u>op3</u>	<u>opcode</u>	<u>rw</u>	<u>ra</u>	<u>r3</u>
15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0				
4 bits				4 bits	4 bits	4 bits	4 bits

$x = 5$ // ldi r1, 5

machine code: 0111 0001 0000 0101

opcode rw N/A 5
7 r1

~ 0x7105

note - refer to registers in binary

$r0 - 0000$ $r1 - 0001$... $r14 - 1110$ $r15 - 1111$

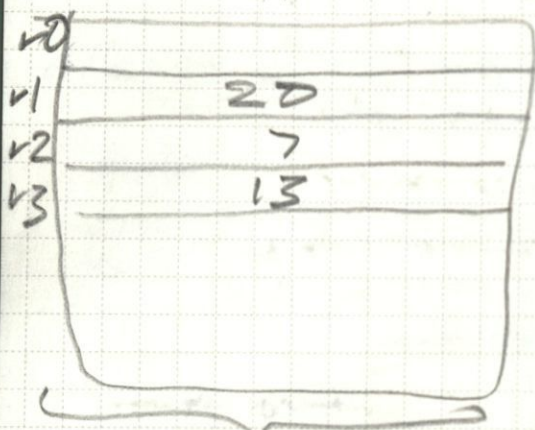
Ex. add $r1, r2, r3$

0000	0001	0010	0011
opcode	rw	ra	rs

~ 0x0123

Ex. $y = 7;$ // $\text{ldi } r2, 7$ // $r2 = 7$
 $z = 13;$ // $\text{ldi } r3, 13$ // $r3 = 13$

RF



16 bits wide

note, register

selection is arbitrary

→ usually chosen by the compiler

$x = y + z;$ // $\text{add } r1, r2, r3$ // $r1 = r2 + r3$

Checking Work:

1. write code in `code.s` file
2. execute command;
as is a smth `code.s` `code.mem`
hex encoding
3. open `code.mem` to look at hex
4. simulate program with
asasimk -i `code.mem`

Code. s files :

- text
- instruction
- instruction

quit

note. must open .oem files in Waz

- everything in this file is a lex
- even instruction addresses

albasan > // opens a prompt

albasan > c // run program

albasan > rd // shows contents of registers in hex

rh // ... in binary

albasan > s // steps through program

OpCodes and Immediates :

ldi : takes an imm8 - 8 bit constant

ldst : takes an imm4 - 4 bit offset

br

br : takes an imm8 - 8 bit offset

br

4 page 33 in albasan textbook has full instruction set

Loads and Stores: logical view of memory
 ↳ i.e. logic design

Addresses Data &

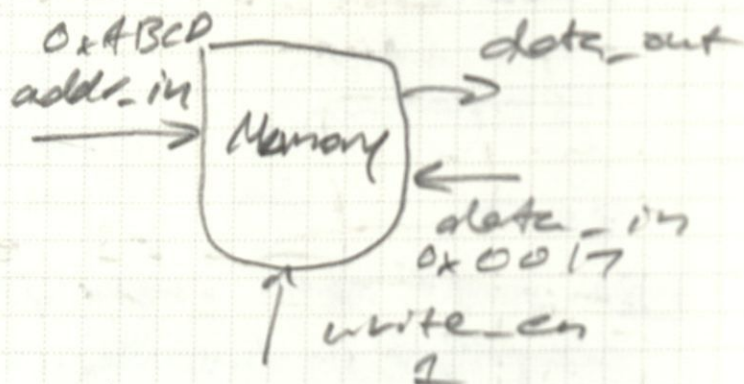
0x0000

⋮

0xABCD 0x0017

⋮

0xFFFF



address:

int *z = malloc(...);

data:

*z = 0x17;

notation: $M[0xABCD] \leftarrow 0x0017$

↗ $v9 \leftarrow M[0xABCD]$
 ↘ $(0x0017)$

store load

Lecture Notes:

1.28.25

Load from a Low (8-bit) Address:

load $ld\ r4, r5, imm4$

$B[r4] \leftarrow M[r5 + imm4]$

Ex.

Load data at mem 0x0040 into r3:-

// $r10 \leftarrow 0x40$, really $0x0040$

$ldi\ r10, 0x40$

↳ implied

$ld\ r3, r10, 0$ // $r3 \leftarrow M[r10 + 0]$

Load data at mem 0x0041 into r4:

$ld\ r4, r10, 1$ // $r4 \leftarrow M[r10 + 1]$

$0x0040 + 1$

note could do this like how we did in prev question, but this is most efficient way

Addr	Data	
...	...	
00A0	~	→ v3
00A1	~	→ v4
...	...	

logical view

Store to a Low (8-bit) Address:

* no registers are written/updated on this instruction *

⇒ just places a value in memory

store st r_a, r_b, imm_u $M[r_b + \text{imm}_u] \leftarrow R[r_a]$

Write addr to

- Place value 20 into v3
- Store R[v3] to addr 0xAB → 00AB
- Then store to addr 0xAC → 00AC

Addr Data

00AB	20
00AC	20

ldi v3, 20

ldi v7, 0xAB

st v3, v7, 0 $\ll M[AB] \leftarrow 20$

st v3, v7, 1 $\ll M[AC] \leftarrow 20$

Challenge: ldi immediate is only 8-bits

Goal: place 0x1989 in r4

ldi r4, 0x1989 cannot be encoded in 1-instruction

0111 | rr | imm8 | 0000 1001 1000 1001

& fundamental limitation

Solution: need 5 instructions to accomplish this

ldi r1, 0x19 0000 0000 0001 1001 r1

ldi r2, 8 0001 1001 0000 0000 r2

shl r2, r1, r2

ldi r3, 0x89 0000 0000 1000 1001 r3

or r4, r2, r3 0001 1001 1000 1001 r4

Storing to a High (16-bit) Address?

Addr Data

	0000
	...
	00FF
{	0100
	...
{	FFFF

must use 5 instruction seq to access this

Store R[r2] to mem addr 0x1ff

Goal: st r2, r1, 0

How do we get 0x1ff into r1

ldi r1, 0x01

ldi r2, 8

shl r2, r1, r2

ldi r3, 0xFF

or r1, r2, r3

↳ can overwrite r1

Data Segment ad high(), low() directives:

- Need to get address into a register before we can load or store
- ldi is the only way to get a constant into a register — can only write 8 at a time

Addr Mem

ABCD	3
ABCE	4
ABCF	0

← X

note: data vars chosen right after last program instruction

Using offsets for array indexing:

// $X[2] = X[0] + X[1]$

// assume X starts at 0xABCD

• data

X: 3, 4, 0

• text

← AB

ldi r1, high(X) // r1 = 0xAB00

ldi r0, 8

shl r1, r1, r0

↓ CD

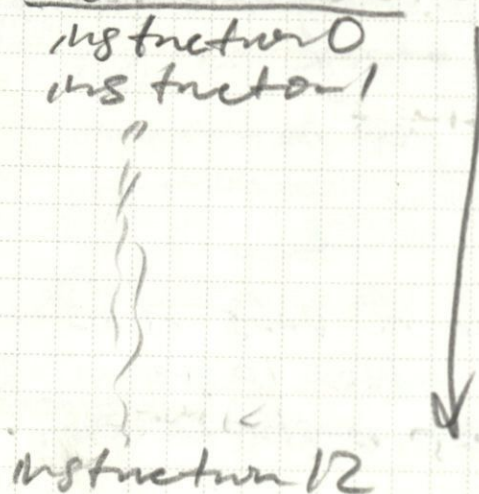
ldi r0, low(X) // r0 = 0x00CD

or r0, r1, r0 // r0 = 0xABCD

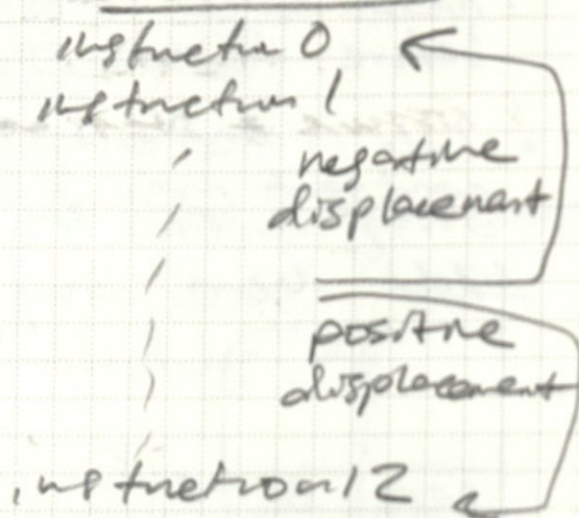
quit

Branch Instructions :

Sequential Flow :



Branch Flow :



negative displacement : loops

positive displacement : selection

unconditional :

br imm8

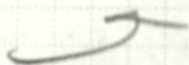


displacement

conditional :

sz vs, imm8

br vs, imm8



Range for 8 bits (signed) :

-128 to 127



1000 0000 to 0111 1111

Lecture Notes:

1.30.25

Albalore Instruction Encoding: Load/Store

// Goal: $y = z + 1$

// Assume z starts at $0x34CE$

.data

$z: 15$

$y: 0$

.text

ldi $r1, \text{high}(z)$ // $r0 \leftarrow z$

ldi $r0, 8$

shl $r1, r1, r0$

ldi $r0, \text{low}(z)$

or $r0, r1, r0$

$\uparrow$

$\downarrow$ ldi $r10, 1$

// assume $r10 \leftarrow 1$

ld $r5, r0, 0$ // load z into $r5$

add $r6, r5, r10$ // add 1 to z , place in $r6$

st $r6, r0, 1$ // store $r6$ to y

quit $\uparrow \uparrow$ ($M[r0+1] \leftarrow r6$)
($z+1 = y$)

Encoding:

$0x8500 \rightarrow r0$

$\rightarrow \text{imm4} = 0$

$0x065A \rightarrow r5$

$0x9160 \rightarrow r6$

$\rightarrow r9$
 $\rightarrow \text{imm4}$

note. instruction order and bit encoding order do not match up
 $\rightarrow$ must match them according to table

Note. no memory protection exists in assembly

- can "de-reference" a null pointer
- can access ANY mem location
 - storage for instructions and data
 - out of scope functions

Ex. $0x9521 \text{ is } st\ r2, r1, 5$

$r2$ and $r1$ are both 0

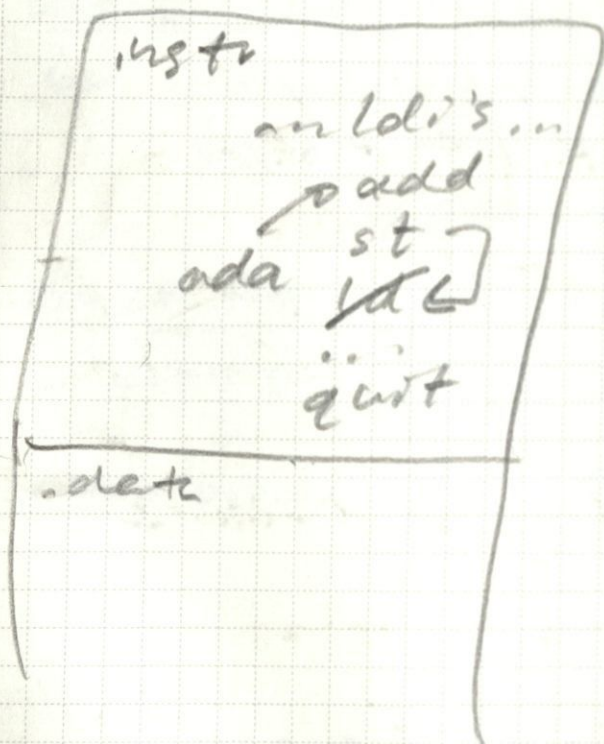
effect: $M[r1+5] \leftarrow r2$

i.e. $M[0+5] \leftarrow 0$

& address 5 is the load instruction from the problem

$ld\ r10, r8, 0 \rightarrow 0x0000$

Memory



alkacone Branch Instructions:

- range for signed displacement: -128 to 127
- b2 is an unconditional branch
- b2, b2n are conditional

Conditional Branch Terms:

Taken. $PC \leftarrow PC + \text{SignedEx}(\text{imm8})$
condition to branch was met,
change control flow

Not Taken $PC \leftarrow PC + 1$

condition to branch was not met,
do not change control flow, continue
sequential execution

Using Conditional Branches:

Testing for Equality:

// use r0 for x, r10 for y, r8 for z
if $(x == y)$
z = 1;

1. compare two registers using subtraction
Sub r7, r0, r10

2. use b2 instruction to check if result
of subtraction was 0

b2 r7, IF_BODY ← label
br OVER

IF_BODY: ldi r8, 1
OVER:

displacement = 2

Test for less-than:

// use r0 for x, r10 for y, r8 for z
if (x < y)
z = 1;

1. compare two registers using subtraction
note, order matters now!

sub r7, r0, r10

2. use bn to check if result is negative

bn r7, IF_BODY

bn OVER

ldi r8, 1

... some code ...

C goto statements:

translating C to use goto statements
is a good first step

if (i < 0) goto skip-body;

...

skip-body: label

...

}

Ex. convert C to assembly

asm:

// r1 ← 2024

// r3 ← 1927

ldi r2, 1

sub r6, r1, r3

bn r6, skip-body

add r4, r4, r2

skip-body:

ldi r5, 1

goto:

year = 2024;

if (year < 1927) goto skip-body

goIrish++;

skip-body:

playVictoryMarch = 1;

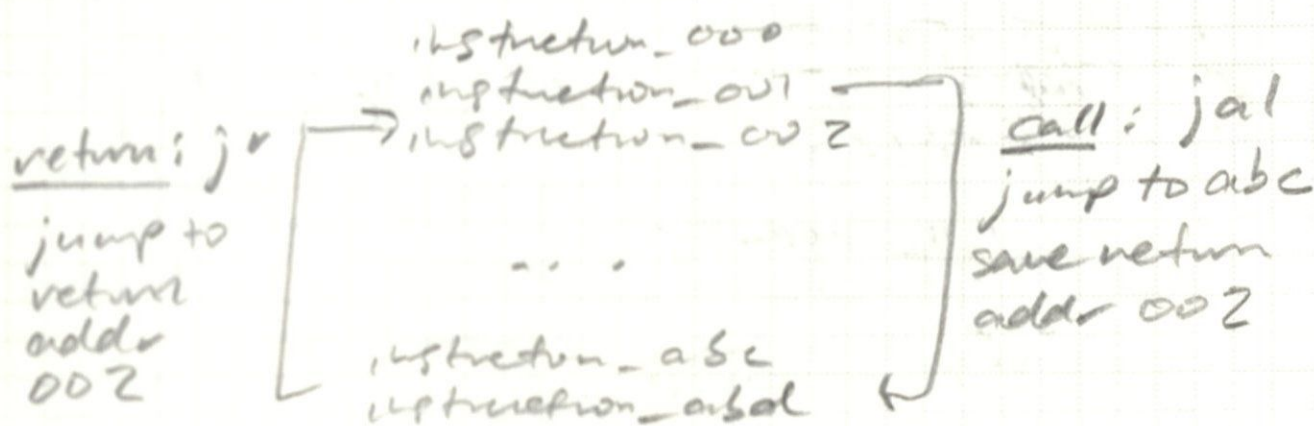
Works with loops too:

```
{  
  int i = 3;  
  int x = 0;  
  while (i >= 0) {  
    ...  
    i = i - 1;  
    ...  
  }
```

goto:

```
{  
  int i = 3;  
  int x = 0  
  loop-entr:  
    if (i < 0) goto loop-exit;  
    ...  
    i = i - 1;  
    goto loop-entr  
loop-exit:  
  ...  
}
```

Procedure Calls:



jump and link: jal imm12

link: Set r15 to return address

jump: branch unconditionally to function itself

jump register: jr ra

branch unconditionally to location in ra

jr r15 → return to function after function call

Lecture Notes:

Loop Using Branches - convert:

<pre>j = 4; k = 0; while (j >= 2) { k++; j--; } i = k</pre>	<pre>j = 4; // r1 k = 0; // r2 loop-cond: if (i < 3) goto loop-exit; k++; j--; goto loop-cond; loop-exit: i = k // r5</pre>
--	--

translate to goto

now can translate to alphaCore;

<pre>ldi r1, 4 loop-cond: ldi r2, 0 sub r11, r1, r4 brn r11, loop-exit add r2, r2, r3 sub r1, r1, r3 br loop-cond loop-exit: add r5, r2, r10</pre>	all registers chosen arbitrarily if $j < 2$ goto // k++ // j-- // br -4
--	--

assume this holds 10
this holds 0 for line 2

alphaCore Procedure Calls: Jump Instructions

```
.text
// main
jal fun // addr 0
sub     // returns here
r15
```

fun: // does something
jr r15 // goto addr 1

albacore Call and Return:

```
Caddr) .text
0000: ldi r1, 1 // r1 = 1
0001: ldi r2, 2 // r2 = 2
0002: jal 0xabc (fun) // call jump to 0xabc
0003: ldi r3, 7 // r3 = 7
0004: add r4, r3, r0 // r4 = r3 + r0
:
```

[...]: quit

```
fun:
0abc: add r0, r1, r2 // r0 = r1 + r2
0abd: jr r15 // return: jump to addr
// in r15 (003)
```

PC History: hex

```
0000
0001
0002
0abc
0abd
0003
0004
:
```

quit

note jal 0xabc

link: $r15 \leftarrow PC + 1$

$r15 \leftarrow 0002 + 1$

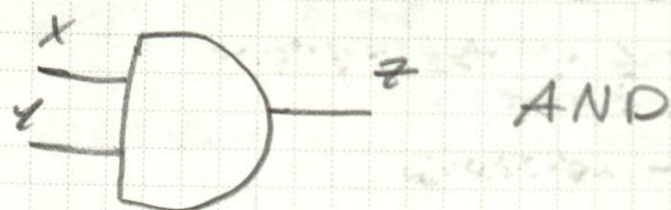
jump: $PC \leftarrow target (imm)$
 $PC \leftarrow 0xabc$

i.e.
always saves return location
in r15

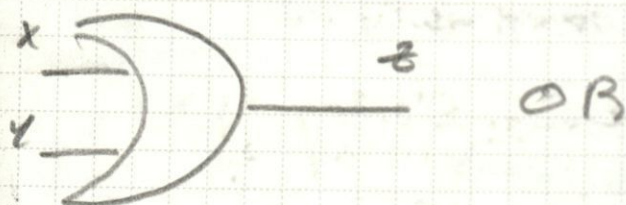
Digital Logic: albacore Hardware Requirements

- Storage
 - main memory - ld/st, every instruction
 - register file - add r0, r10, r3
 - program counter
- Arithmetic/Logic unit - add, sub, mul, div, shifts
- Controller
 - decode instruction and sequence operations
- Design Methodology
 - hierarchical structure - top-level design
 - tools for design and simulation

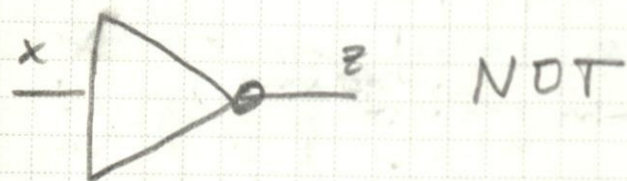
Boolean Operations and Logic Gates:



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

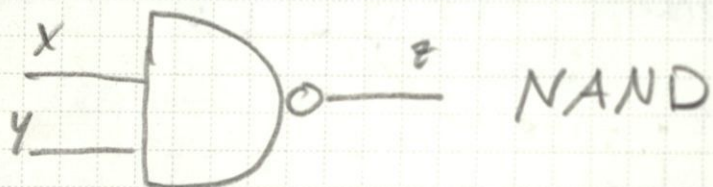
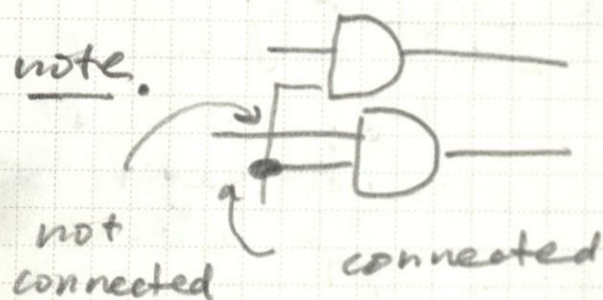


x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

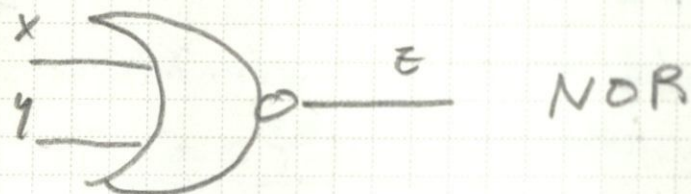


x	z
0	1
1	0

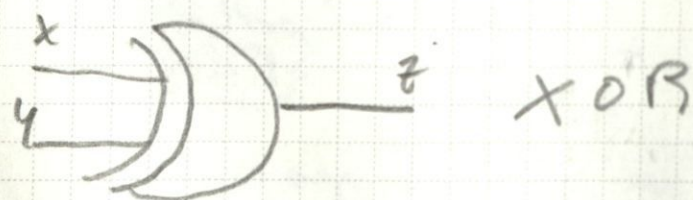
note.



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



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

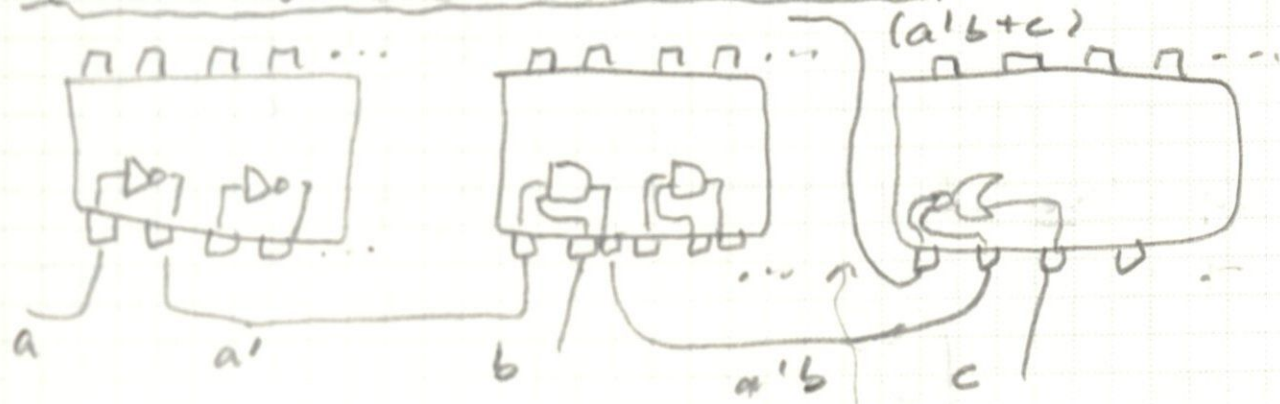


x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

Transistors: Building Blocks for Processors and Gates

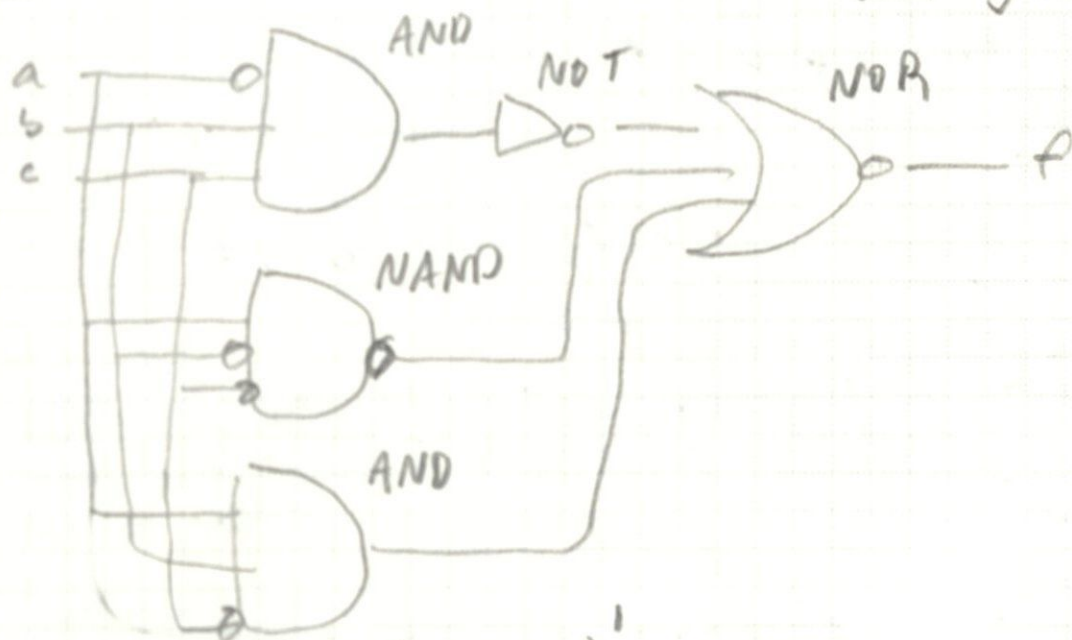
- Gates are abstractions of transistors that create these functions
- CPU has billions of transistors
- digital integrated circuits cover this

$(a'b + c)'$ from 7400 series chips



Ex. convert to equation

send to another
not gate



$$(a'bc)' + (ab'c')' + abc' = f$$

Lecture Notes:

2.6.25

Top-Level Design:

truth table $\rightarrow$ Boolean equation $\rightarrow$ circuit

Ex. A call should go to voicemail if there is an incoming call and either the phone is offline or the phone is online and there are 4 rings and no answer.

$$V = C(P + P')A' = CA'$$

top-level design:

C	P	A	V
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

note. this one is impossible

Boolean Expressions: for vars a, b, c

Product terms: $abc, a'c$

- AND and NOT only!

Sum-of-products: $abc + a'c$

- OR product terms

Minterms: $abc, a'b'c, abc'$

- AND and NOT all literals exactly once

Sum of Minterms: $abc + abc'$

- OR minterms

note: all minterms are product terms

Sum of Minterms Widely Used:

- 1) regular, recognizable
- 2) easily optimizable w/ k-maps

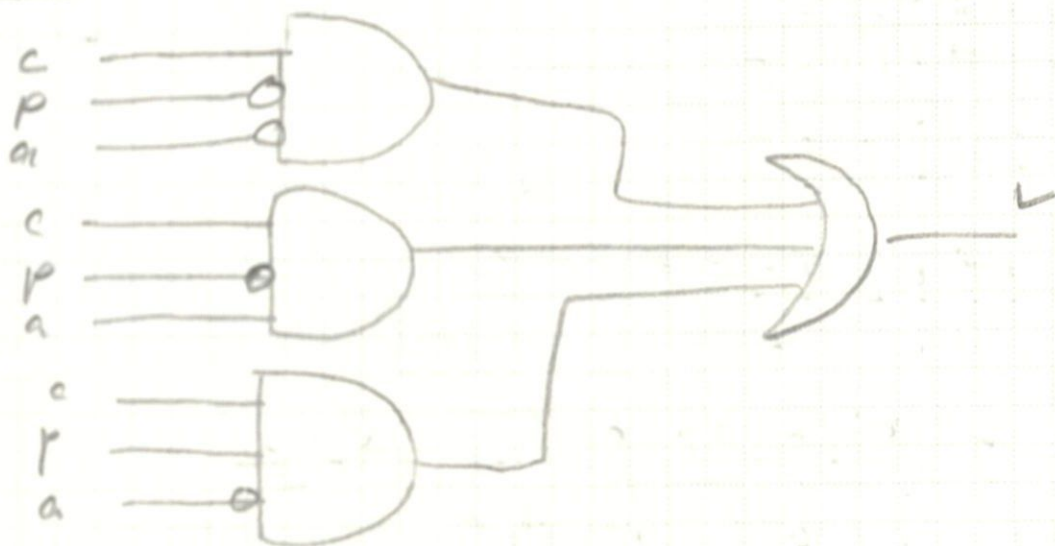
Ex. truth table from last example

Sum of Minterms:

- look for 1's in output
- input = 1 $\rightarrow$ un-complemented
- input = 0 $\rightarrow$ complemented

$$cp'a' + cp'a + cpa' = V$$

Circuit:



note, in actual circuits will need to wire one instance of c, p, a to all AND gates, but can do this in drawing to simplify

Reduced Circuit:



Ex. A half adder takes 2 single bit numbers a, b and produces 2 outputs, called sum and carry

$$\begin{array}{r} a \\ + b \\ \hline cs \end{array}$$

truth table:

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Equations:

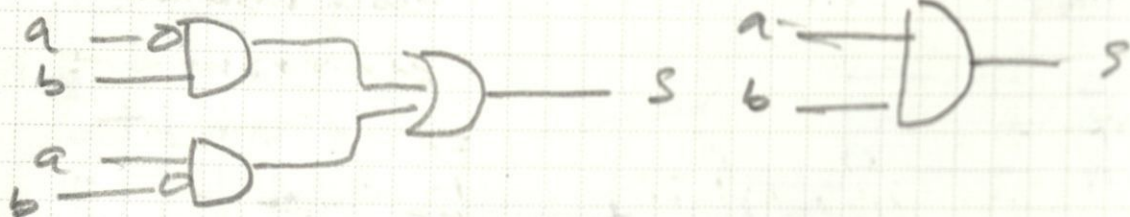
$$s = a'b + ab'$$

$$c = ab$$

sum of minterms

Logic:

need 2 circuits



Simplification: Prove DeMorgan's Law

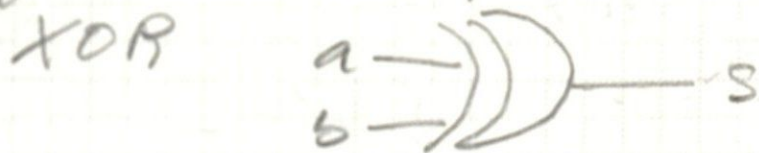
$(a' + b') = (ab)'$ this is a NAND operation

a	b	a'	b'	a' + b'	ab	(ab)'
0	0	1	1	1	0	1
0	1	1	0	1	0	1
1	0	0	1	1	0	1
1	1	0	0	0	1	0

equivalent

Simplifying the Half Adder:

equivalent circuit for S with a single gate:



note. no boolean algebra way to recognize this

Ex. simplify w/ properties of Boolean alg

show

$$ac + [(ac)' + (b'd')] + (ac')(b+d') + ac' = a$$

$$ac + ac' + \dots$$

$$a(c+c') + \dots$$

$$a + \dots$$

$$a + [(ac)' + (b+d)']' + \dots$$

$$a + ac(b+d) + (ac')(b+d')$$

$$a [1 + c(b+d) + c'(b+d')]]$$

$$a(1)$$

$$a$$

commutative

distributive

complement

DeMorgans

DeMorgans

distributive

complement

identity

Karnaugh Maps (K-Maps):

1. Choose size, determine literals
2. Fill in map with 1s for minterms 0s otherwise
3. Group cells containing 1s
 - minimum groups
 - maximum size - must be powers of 2
 - no diagonals or L shapes
 - groups can wrap around
4. Write simplified minterms
 - eliminate literal that changes

2-var k-maps:

Ex. $z = xy' + x'y + x'y'$

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

can go straight from truth table

x \ y	0	1
0	1	1
1	1	0

can also go straight from sum of minterms

simplified eqn = $x' + y' = z$

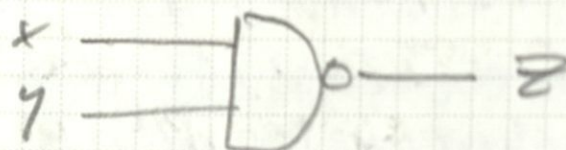
under the hood: k-map is finding $i(j + j')$ opportunities

Logic:



can simplify further w/ NAND:

- look at truth table for this



3, 4 var k-maps: 4 will be max by hand

$q = x'y'z' + x'y'z + x'yz'$

x \ yz	00	01	11	10
0	1	1	0	1
1	0	0	0	0

$f = abcd + a'b'cd' + a'b'cd'$

ab \ cd	00	01	11	10
00	0	0	0	1
01	0	0	1	1
11	0	0	1	0
10	0	0	0	0

Common Mistakes:

- Ordering is 00, 01, 11, 10
→ not in order

- Forgetting to wrap around

Ex. $f = x'y'z' + xy'z' + xyz'$

$x \backslash yz$	00	01	11	10
0	0	0	0	1
1	1	0	0	1

$f = yz' + xz'$

Ex. A call should go to voicemail if there is an incoming call and either the phone is offline or the phone is online and there are 4 rings and no answer

$V = C(p + p')a' = ca'$

K-Map process:

$V = cpa' + cp'a' + cpa'$

$c \backslash pa$	00	01	11	10
0	0	0	0	0
1	1	1	0	1

c	p	a	V
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	X
1	1	0	1
1	1	1	0

$V = cp' + ca'$ X

invalid

• for invalid input minimize eqn with either a 1 or 0 since it does not matter

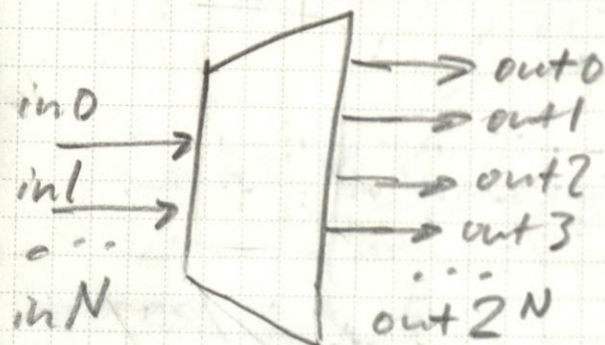
$c \backslash pa$	00	01	11	10
0	0	0	0	0
1	1	0	0	1

$V = ca'$ ✓

Decoders: take N inputs, set 1 of 2^N outputs to a 1, rest to 0

Ex. 2x4 decoder, "2 to 4"

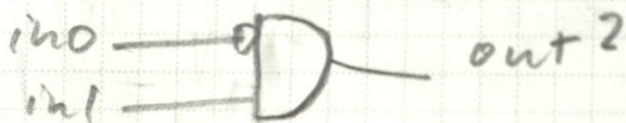
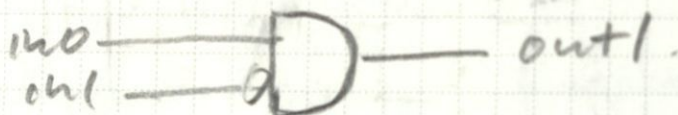
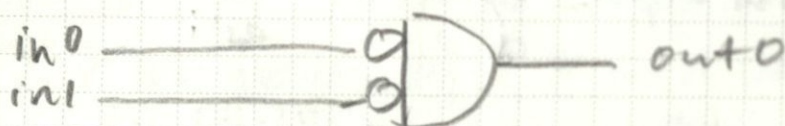
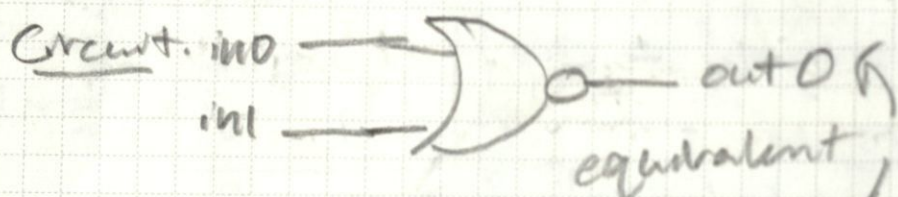
Schematic Symbol:



Truth Table:

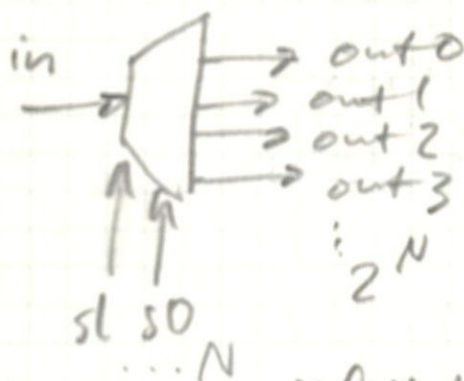
in1	in0	out3	out2	out1	out0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

note corresponds to binary ↗



Demultiplexor (Demux): take 1 input and send it to 2^N selected outputs, rest are 0

Schematic Symbol: Truth Table: (compact)



s1	s0	out3	out2	out1	out0
0	0	0	0	0	in
0	1	0	0	in	0
1	0	0	in	0	0
1	1	in	0	0	0

& full truth table in slides &

Lecture Notes:

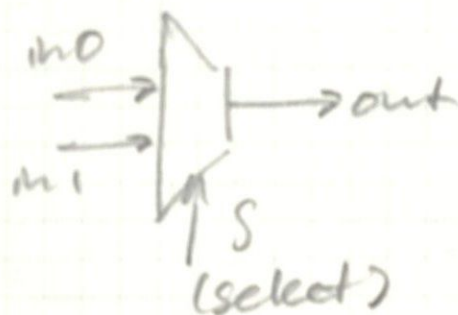
2.13.15

Multiplexor (Mux): select from among N inputs to send an output

Ex. 2x1 mux, called "2 to 1"

Schematic Symbol:

Compact Truth Table:



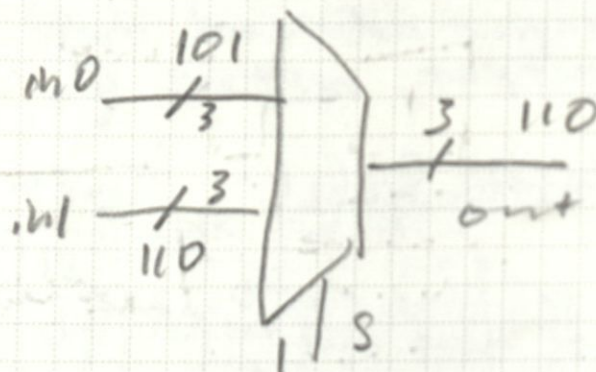
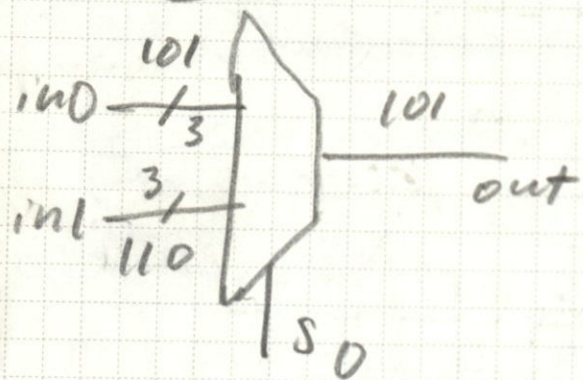
s	out
0	in0
1	in1

Full Truth Table

in1	in0	s	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

& example of
4:1 mux circuit
in slides &

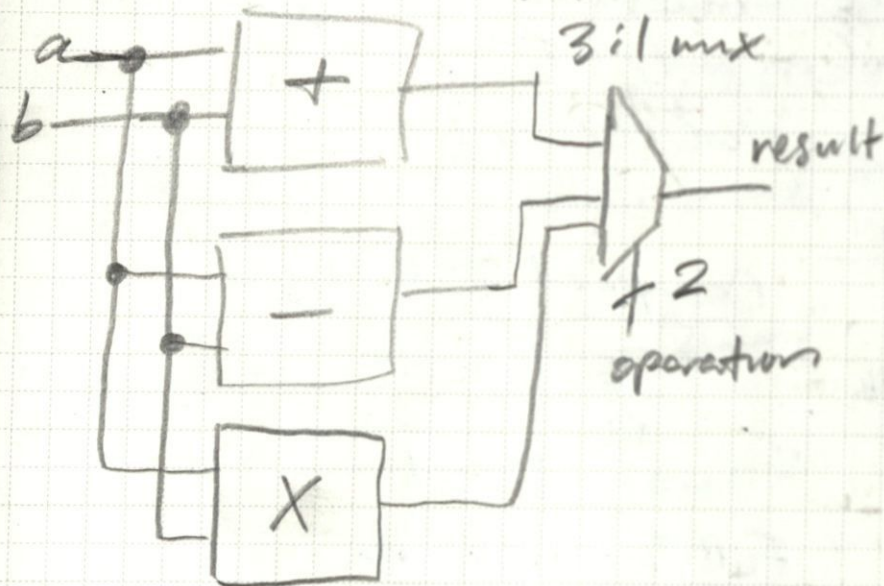
Multi-bit Mux Ex:



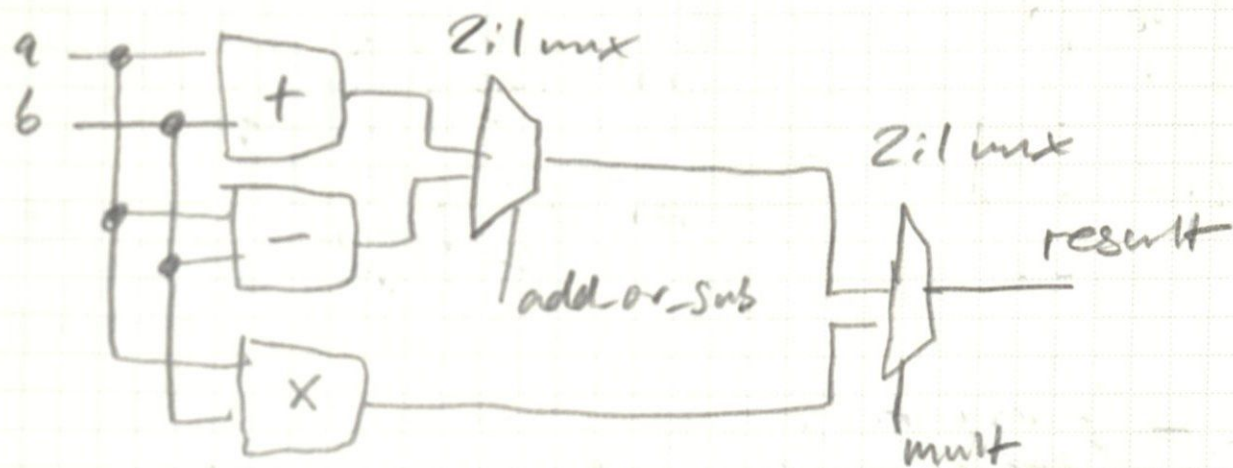
generally, number of bits for select is $\lceil \log_2 (\# \text{ of inputs}) \rceil$

Mux: Ex Use

Sketch a circuit to select between addition and subtraction. What if we also added multiplication?



Is a cascaded mux a good idea?



note. $n \times$ is called a bus (bundle of wires)

using one mux always better than cascading
but sometimes you have to cascade

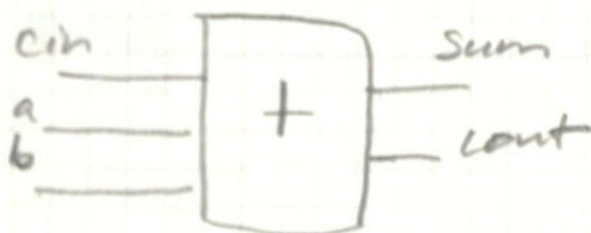
Binary Addition Recap:

ex. add 2 3-bit numbers

$$\begin{array}{r} 010 \\ + 011 \\ \hline 101 \end{array}$$

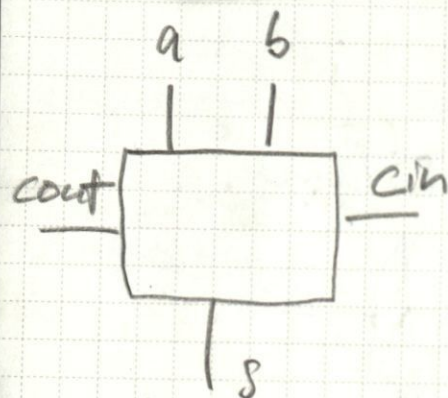
cin

cont { single "column" of addition



Called a Full Adder
(half adder cannot take
in a cin)

Full Adder:



a	b	cin	cout	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

$$S = a \oplus b \oplus cin$$

$$cout = (ab) + (bcin) + (acin)$$

note. $\oplus = \text{xor}$

options:

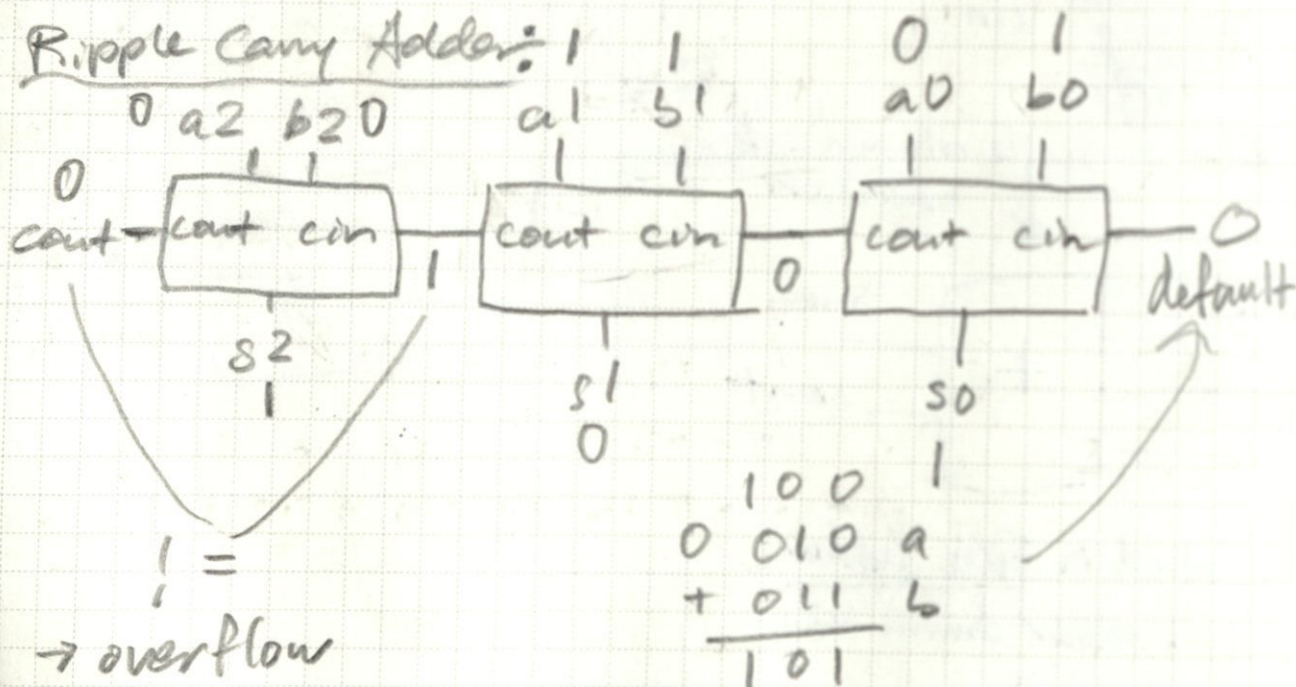


★ 1) output = 1 when we have odd # inputs

2) output = 1 when exactly 1 input = 1

1 is rarely used but logisim evolution defaults to 2

Ripple Carry Adder:



The Illusion of Subtraction:

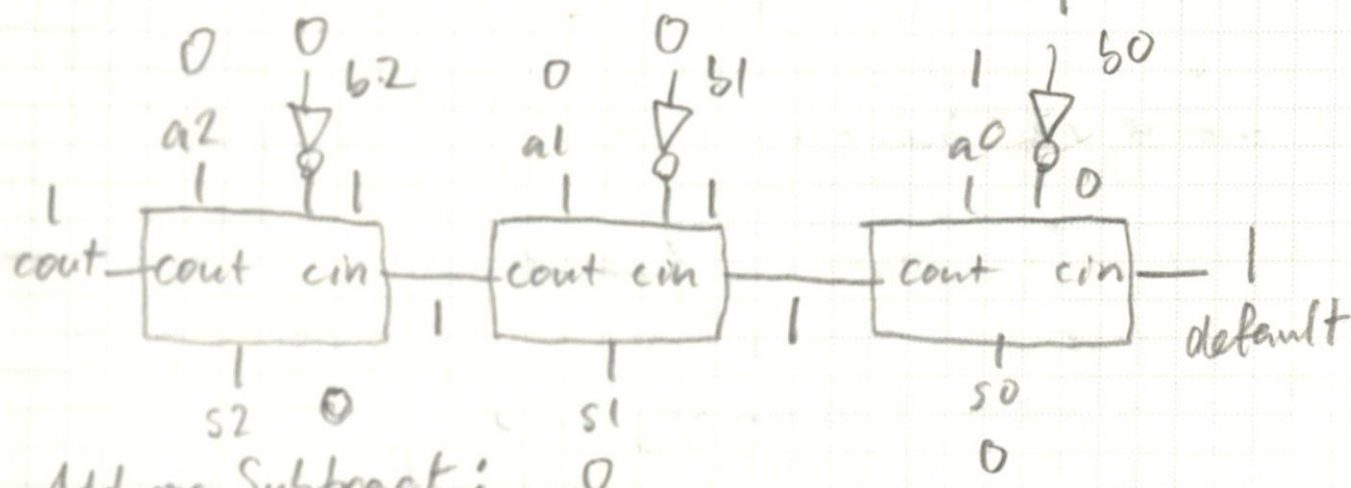
$$\begin{array}{r} 001 \\ - 001 \\ \hline \end{array} + (-)001 \rightarrow \begin{array}{r} 001 \text{ a} \\ + 111 \text{ b} \\ \hline \end{array} + -1$$

must negate b:

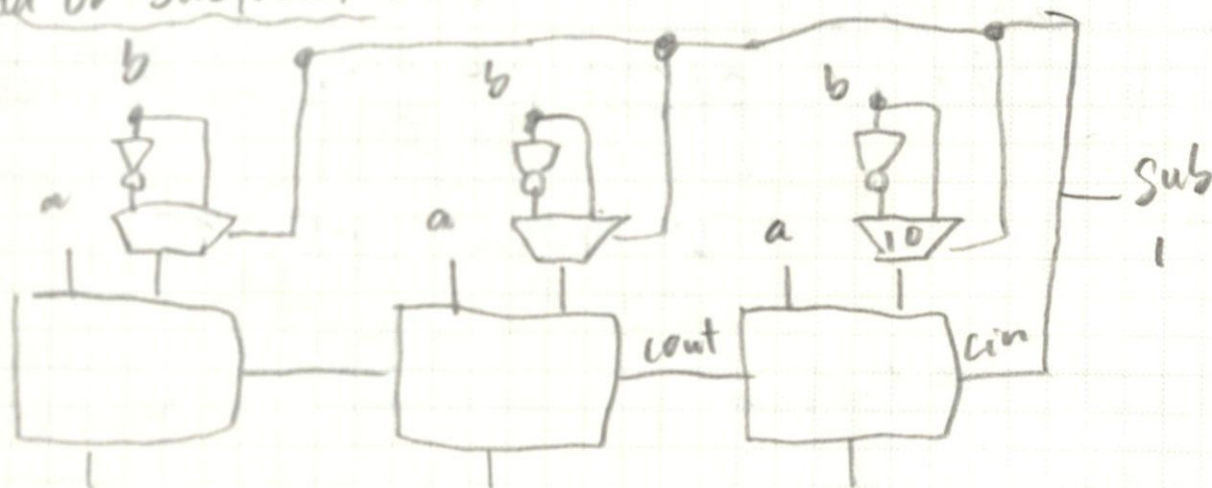
- flip the bits
- add 1

two's complement
 $001 \rightarrow 110$
 $110 \rightarrow 111$

decimal



Add or Subtract:



Sidenote: this is slow,

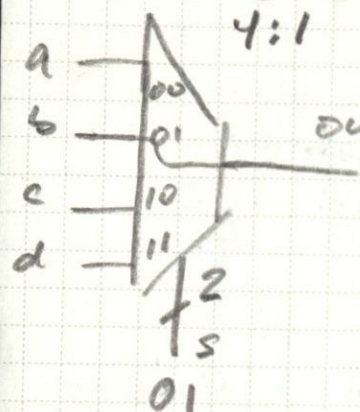
- the carry lookahead adder is faster but more complicated
- same with the Kogge-Stone Adder, created by an MD-prof!

Lecture Notes: Exam 1 Review

2.18.25

Multiplexers: always has 1 output

4:1 always has $\log_2 N$ selects



Mux: many to one

Demux: few to many

Decoder: few to many

Branches and Jumps:

br - unconditional branch

bz, bz - conditional branch

jal - call functions

jr - return from functions

Ex. if ($x == y$) {
 $z = 8$;
}

Sub $r13, r11, r10$
 $x: r10$ $bz r13, IF_BODY$
 $y: r11$ $br OVER$
 $z: r12$

IF-BODY: $ldi r12, 8$
OVER:

Displacement:

Hex addresses

0000

0001

0002

0003 IF-BODY: $ldi r12, 8$

0004 OVER: qwr

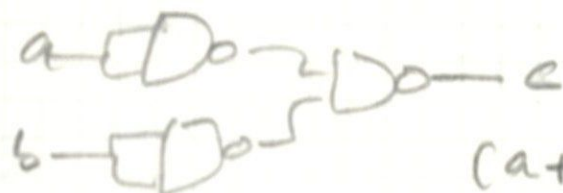
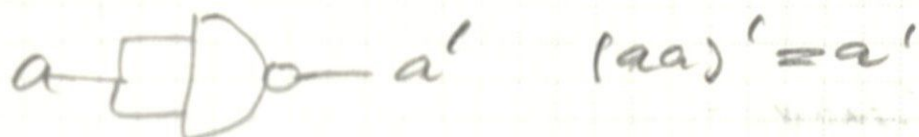
$bz r13, IF_BODY$ $ll bz r13, 2$
 $br OVER$ $ll br 2$

LBL: add
 { sus
 { or
 { br LBL

// br - 3

signed number in
 two's complement notation

NAND:



$$\begin{aligned}(a+b) &= ((aa)'(bb)')' \\ &= (a'b')' \\ &= a+b\end{aligned}$$

Universal Gate:

A single gate that can implement any
 Boolean logic function

NAND, NOR

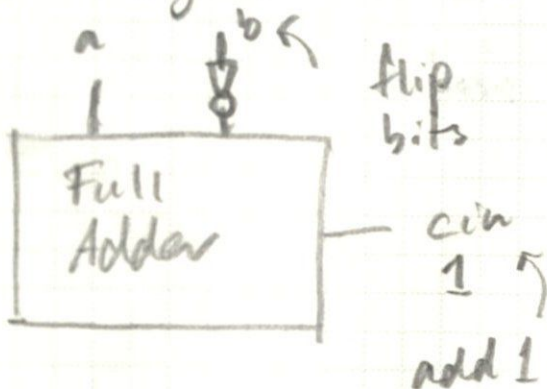
Adder / Subtractor:

Subtraction is addition with a negative number

$$\begin{array}{r} 3 \\ -2 \end{array} \rightarrow \begin{array}{r} 3 \\ + -2 \end{array}$$

$$\begin{array}{r} 011 \\ -010 \end{array} \rightarrow \begin{array}{r} 011 \\ +110 \end{array}$$

We use a multiplexer
 to choose between these two
 options



Lecture Notes:

2.25.25

Recall, our goal is to build full adders
Datapath (in slides)

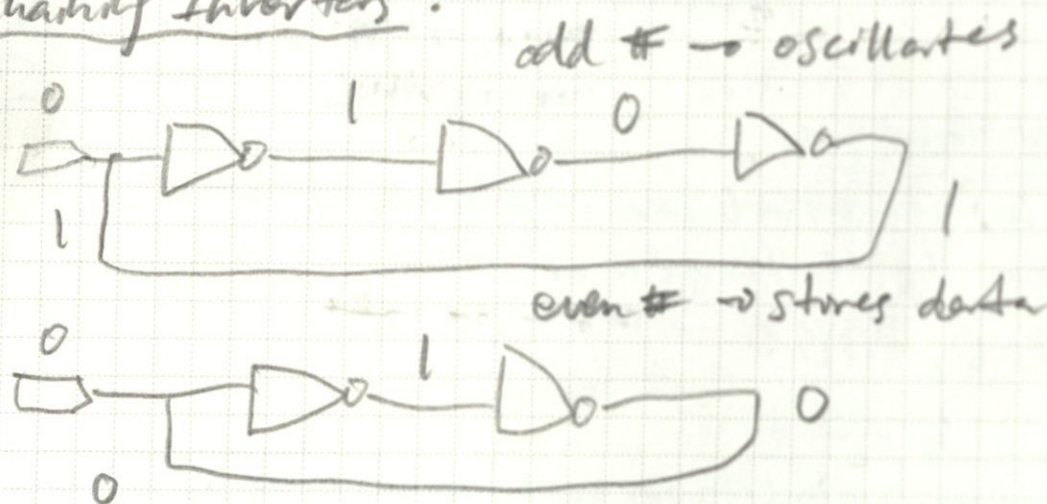
Missing, way to save data in register file
and instructions/data in memory

Have covered: Combinational logic

- AND, OR, NOT, NAND, NOR
- Decoder, Multiplexer, Demultiplexer
- Adder/Subtractor $\rightarrow$ ALU

Next, manipulate Memory
 $\rightarrow$ sequential logic/circuits!

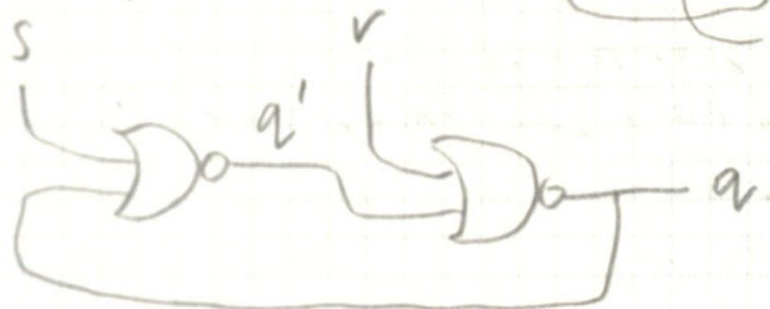
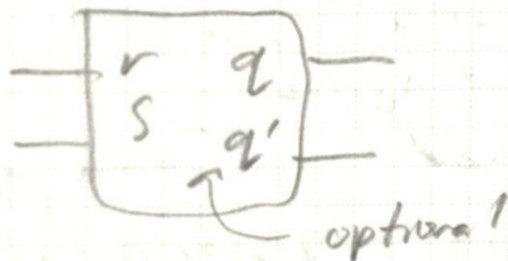
Chaining Inverters:



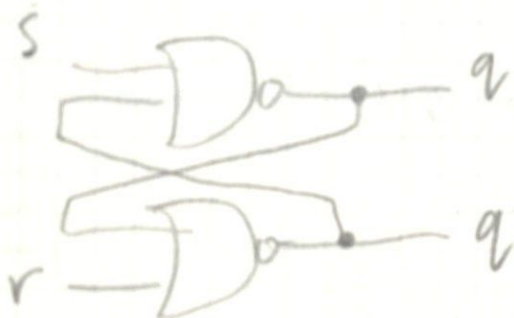
idea of a latch! $\nearrow$

Fast Stop: SR Latch "set-reset latch"

s	r	q - depends on previous inputs
0	0	hold
0	1	0
1	0	1
1	1	problematic!



note: same circuit



can work out
same truth table
manually

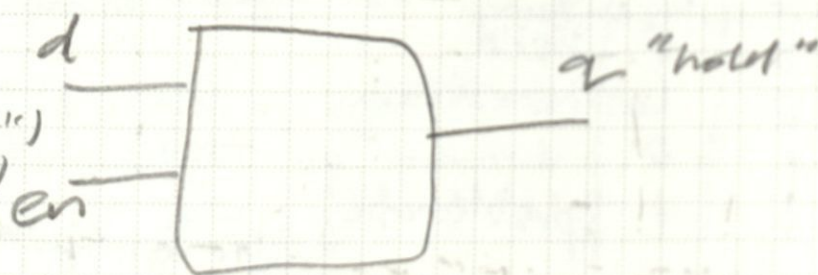
$s=1, r=1$ problem case causes
oscillation!

→ see slides

Working to Memory: SR Latch w/ D Logic

SR Latch:

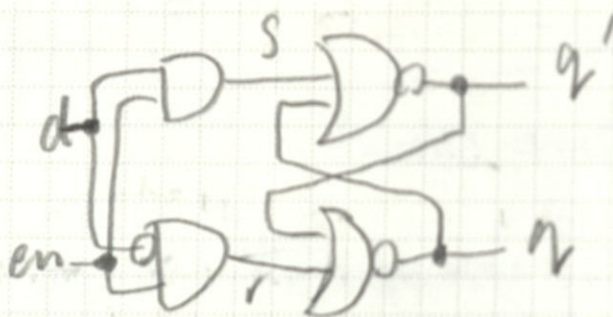
s	r	q
0	0	hold
0	1	0 ("reset")
1	0	1 ("set")
1	1	bad



Goal, hold whenever en (enable) is 0
store d in SR latch ($q = d$) when $en = 1$

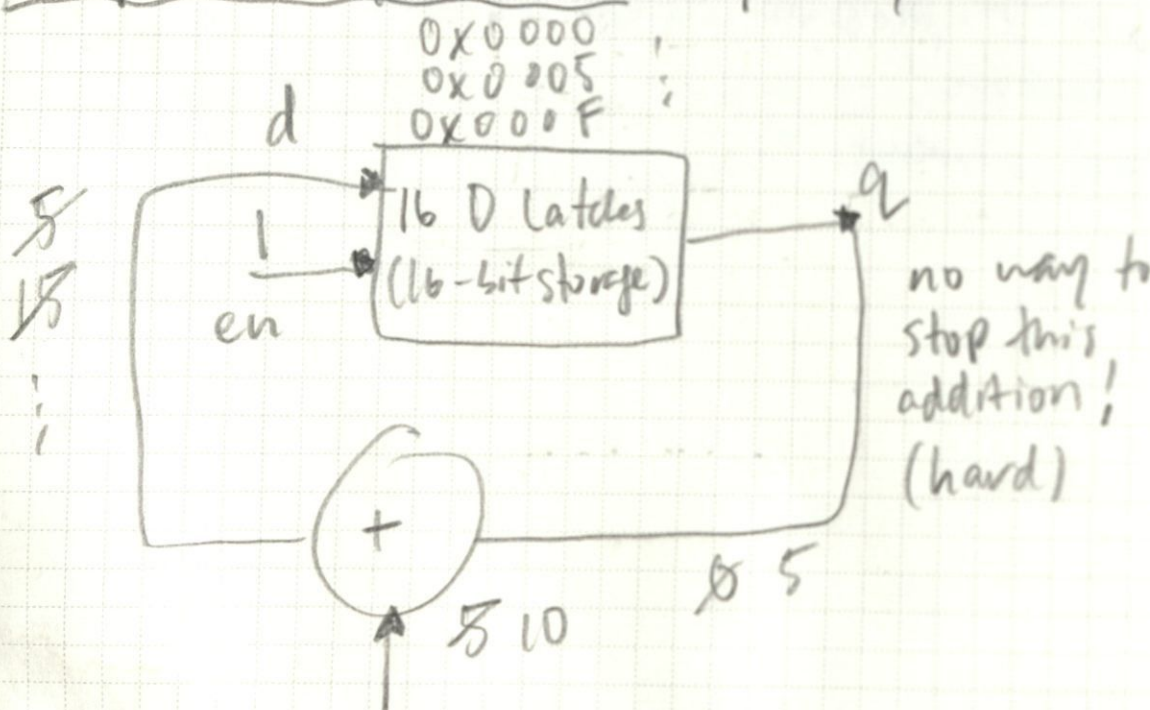
D Latch: ^{internal}

en	d	q	s	r
0	0	hold	0	0
0	1	hold	0	0
1	0	0	0	1
1	1	1	1	0

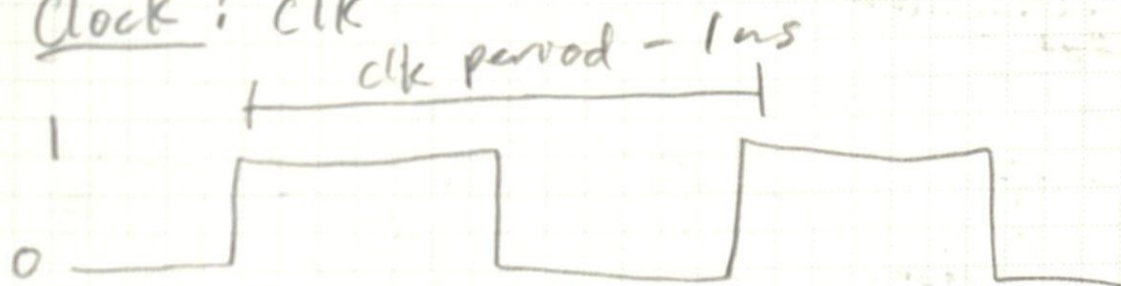


solves the $s=1, r=1$ problem

The problem w/ Latches: Transparency



Clock : clk



clock rate :

$$1 \text{ GHz} = 1 \text{E}9 \text{ Hz} = 1 \text{E}9 \text{ cycles/second}$$

Clock period : $1 / \text{clock rate} = 1 \text{ ns}$

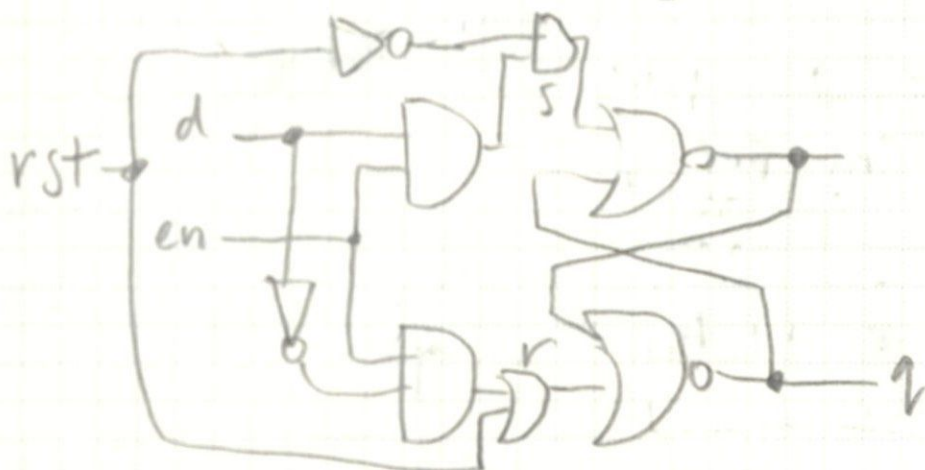
To solve transparency issue must flip en
between a cycle !

Lecture Notes :

2.27.25

Ex. D-Latch w/ Reset

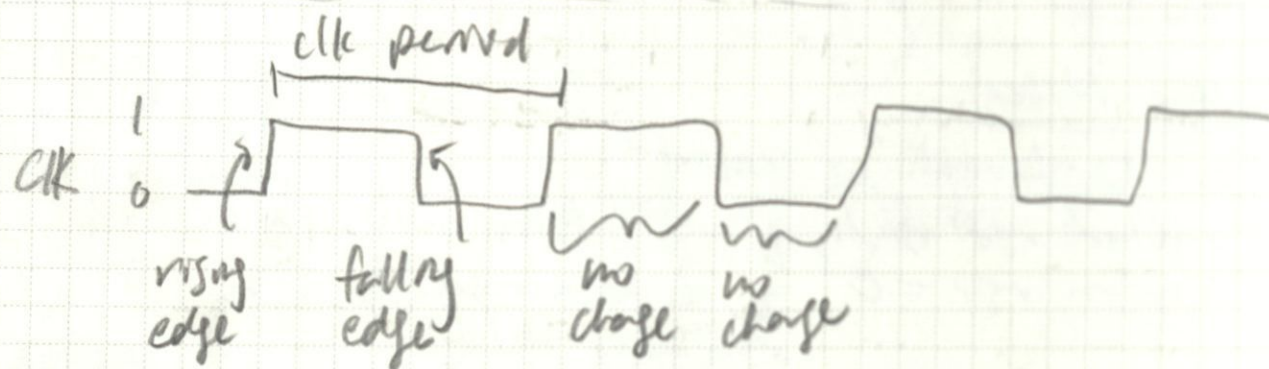
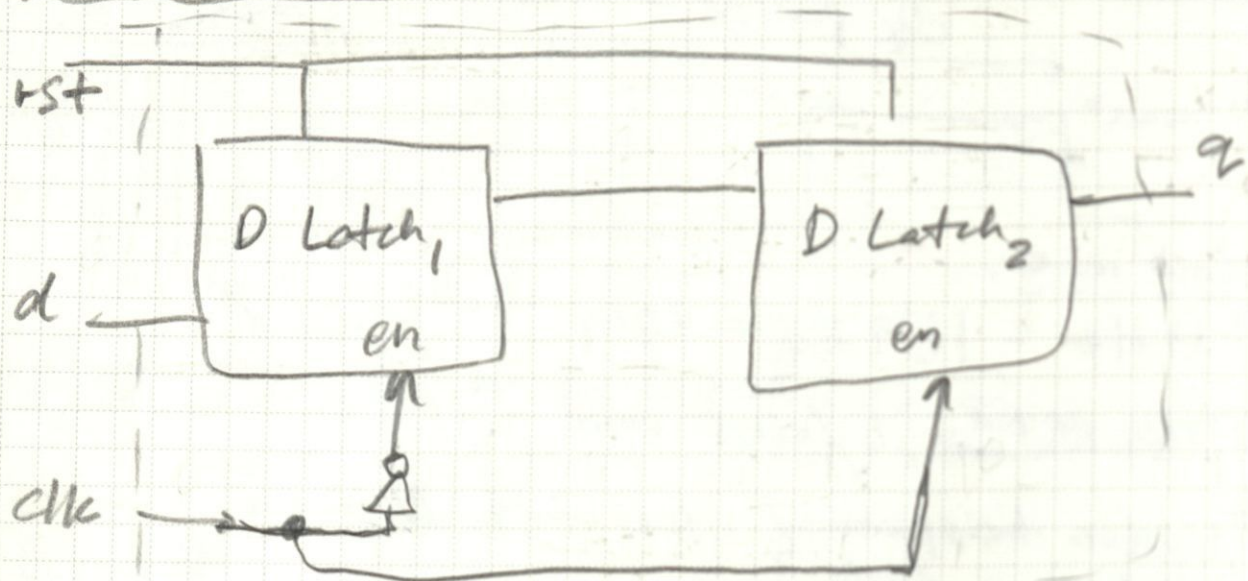
- when rst is 1, $q = 0$ (force $s=0, r=1$)
- when rst = 0, q determined by en, d is usual



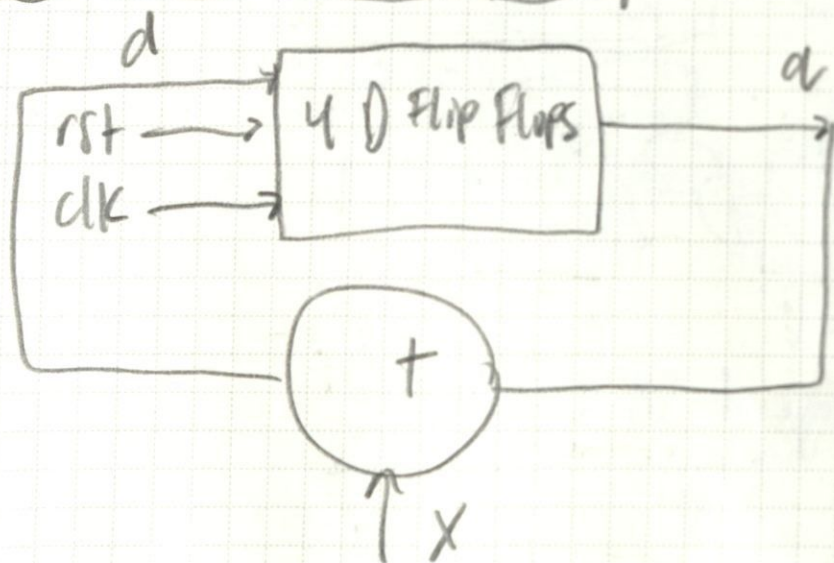
force $s=0, r=1$

ie. take advantage of SR Latch

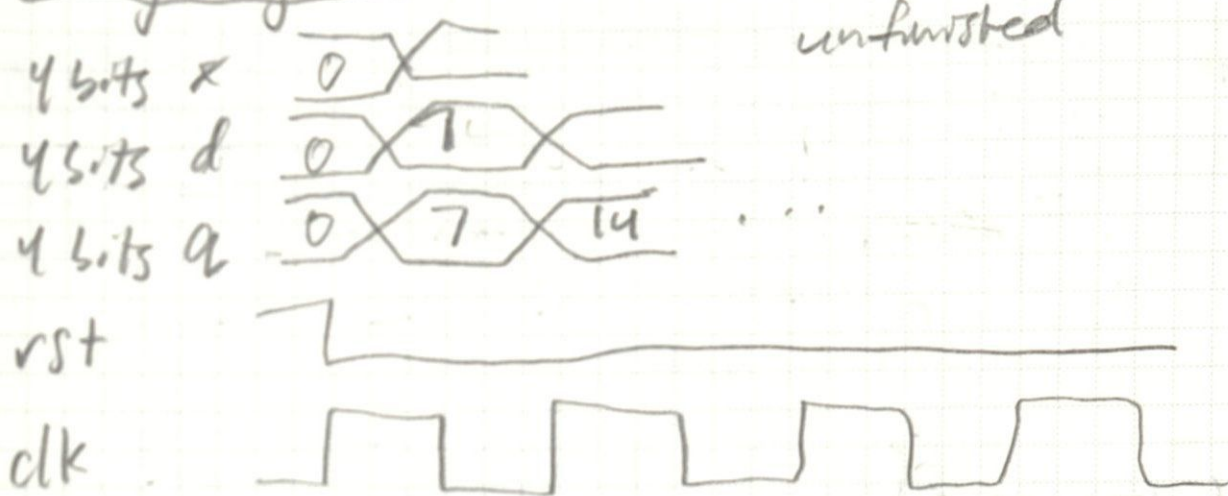
Fix: D Flip-Flop:



only updates on rising (positive) edge
How they solve transparency:



Timing Diagram:



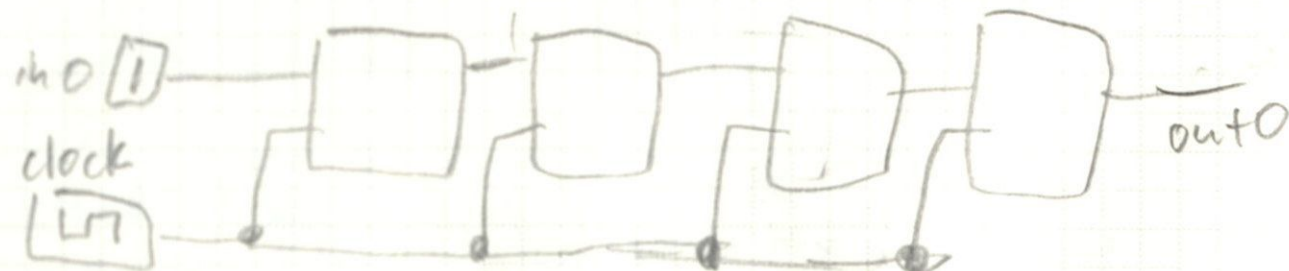
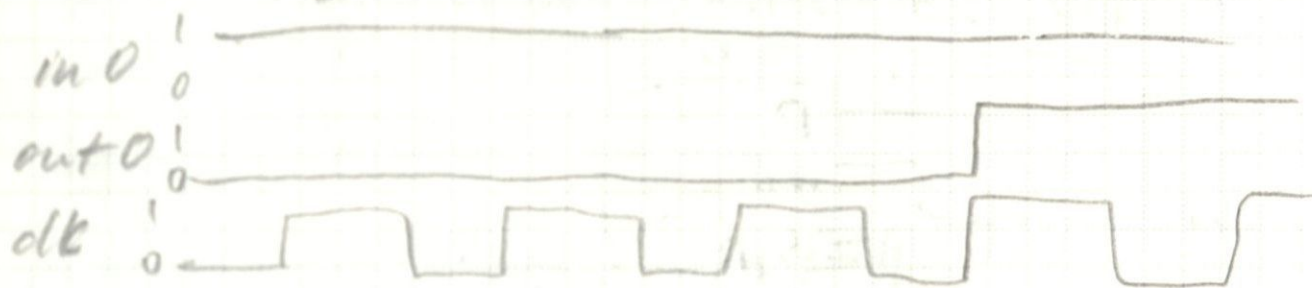
only updates on rising edge

Ex of creating timing diagram in
logisim evolution in slides

Lecture Notes:

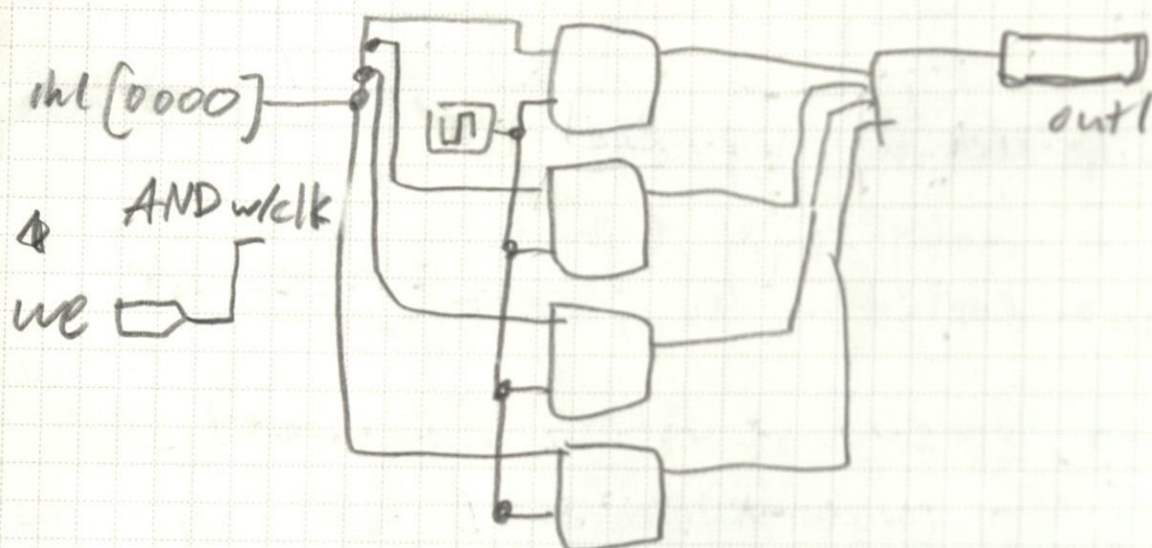
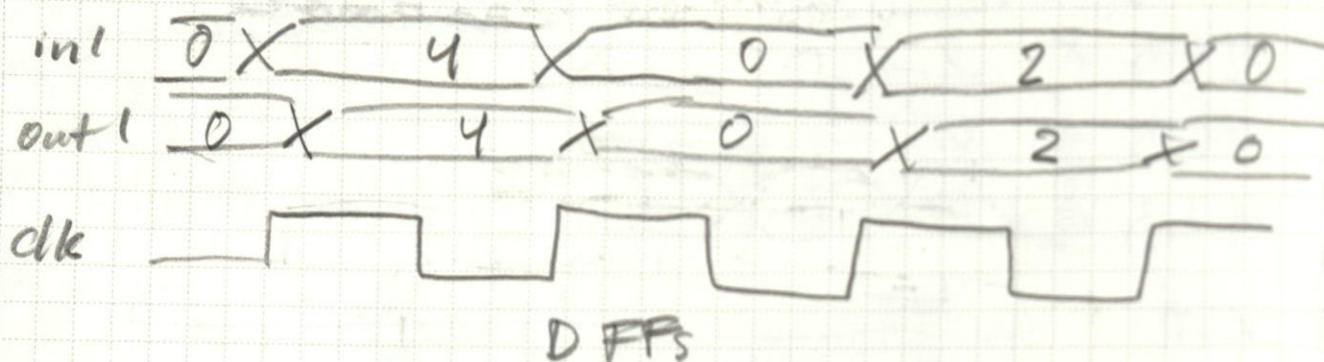
3.4.25

Shift Registers: These are D FFs, assume
initially holds 0

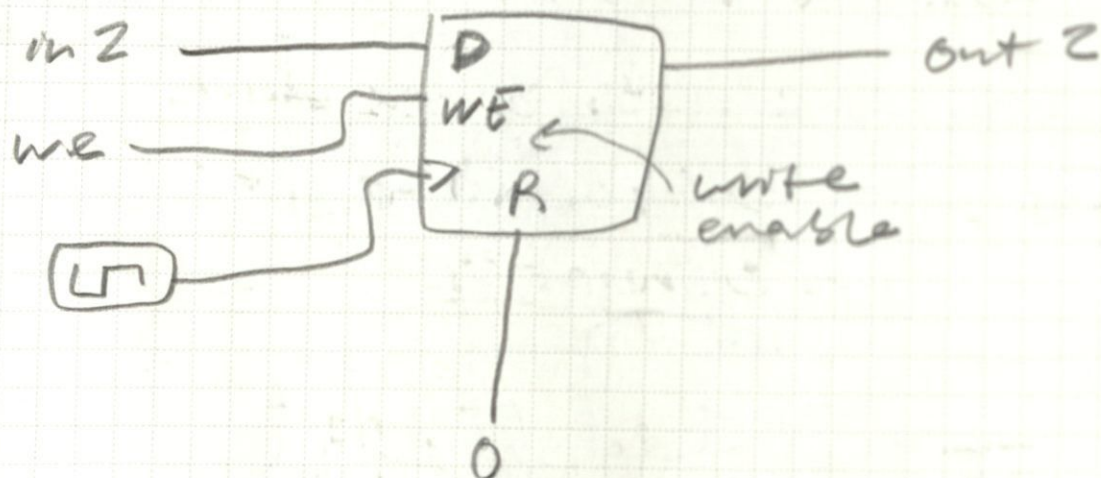


shifts data by 4 clock cycles

N-bit load registers:



Top-level view:

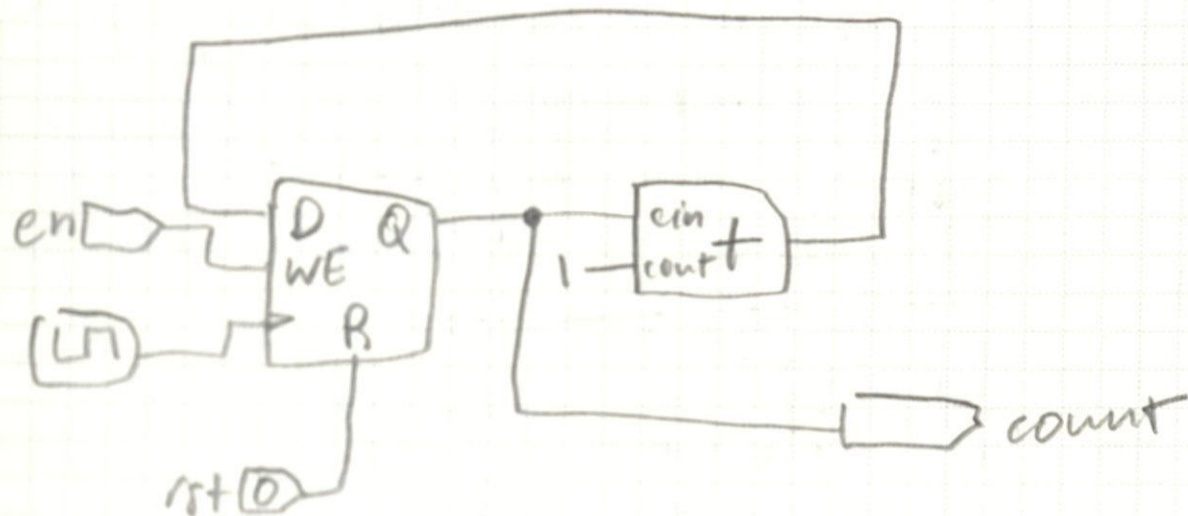


so generally try to not mess w/ the clock
another solution w/ multiplexers

Counters :

increment or decrement when enable
(++) (--)

4-bit incrementing counter:



taking advantage of rst to initialize
 $\text{rst} = 1 \rightarrow \text{counter} = 0$

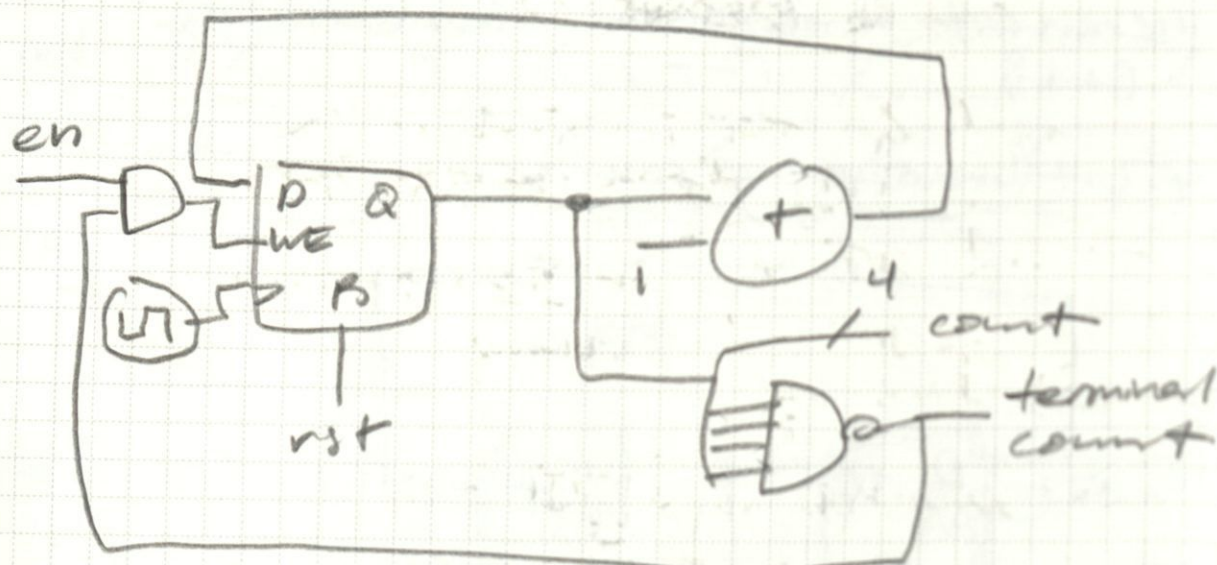
max value is $1111 = 15$, and
then resets to $0000 = 0$

Saturating Counter: stop at max value

A synchronous reset: when $\text{rst} = 1$,
set counter to 0, regardless of all
other values

when $\text{rst} = 0$ and $\text{en} = 1$, as
clock ticks, count to $(2^n - 1)$
and no further

Saturating Counter:



Counter in Logisim:

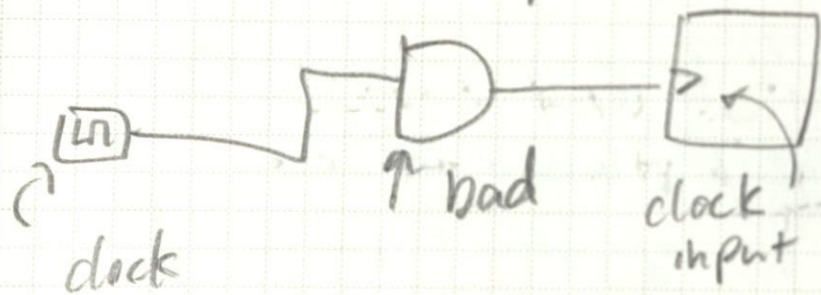
- uses count from adder to saturate
- abstracts this entire process

Lecture Notes:

3.625

Clock Best Practices: "default" approach:

do not add anything b/w the clock and the clock input — leads to skew/delay



called clock gating: can be useful if done with care

Timer. outputs 2 periodically, each time a counter reaches 0

How would you design this with a 3-bit saturating-down counter and a 3-bit load register

→ reset counter automatically when time up

→ 3210000 ...

timer schematic in logisim

alphaCore: Register File and Memory

- done multiplexors ✓
- done ALU ✓
- done registers ✓

NEXT. Register files and Memory

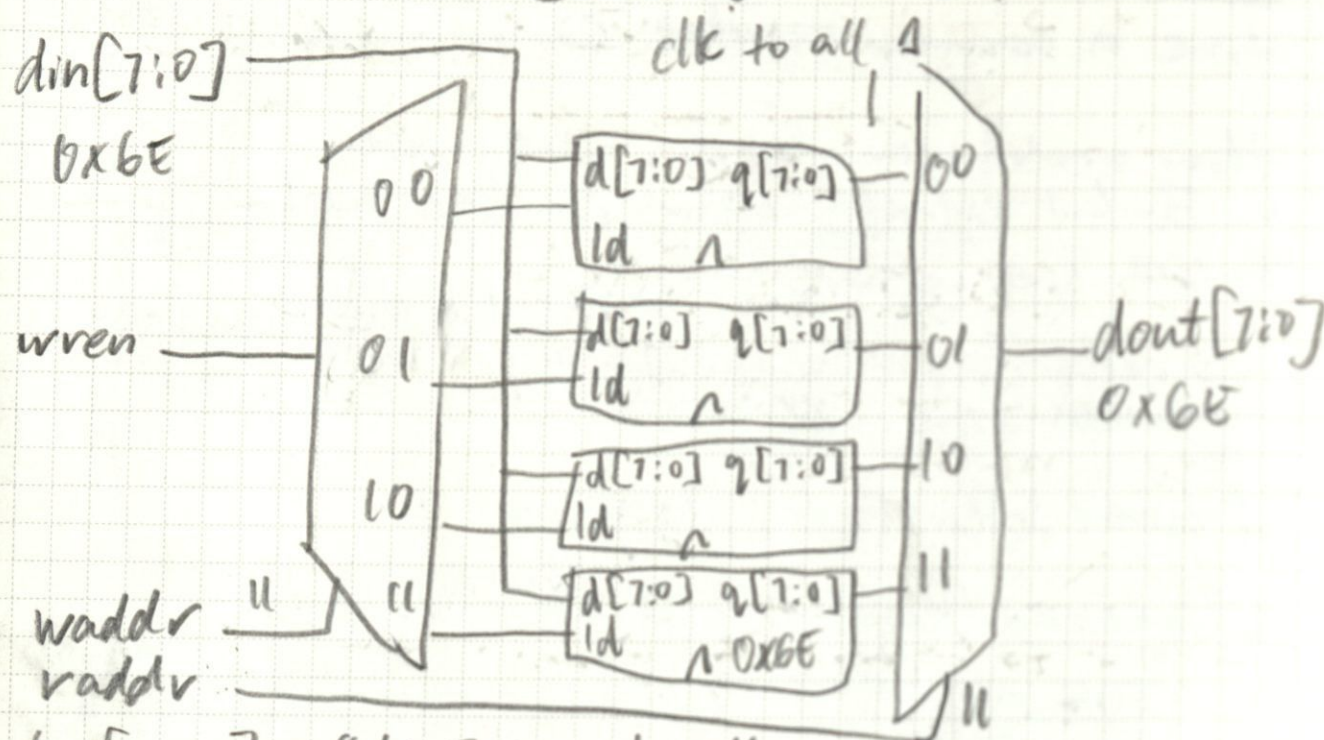
note. that's all there is to alphaCore

→ look at top-down view in slides

Register Files:

a bunch of registers used together with the ability to store to and read from any register

Single Write, Single Read Register File:

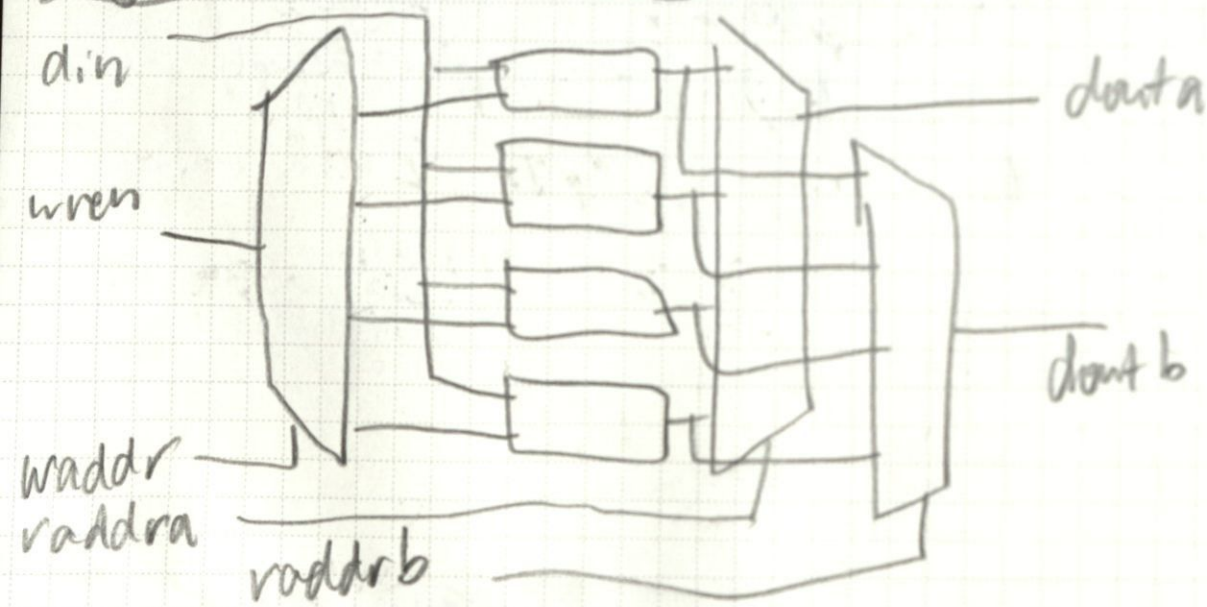


$din[7:0]$ - "din 7 down to 0"

This RF stores 8 bits of data in each register
Synchronous Write: only write on rising clock edge

Asynchronous Read: always read out, regardless of clock

Single Write, Double Read:



Types of Memory:

Asynchronous: output changes when input changes

Synchronous: output can only change on clock edge

Random Access Memory: RAM

• same time to read to/from any addresses
(i.e. from any register in register file)

• ex. register file

Read Only Memory: ROM

• want to be written once, read frequently

• ex. NES ROM cartridge

Larger Memories:

should we use D-FFs to create absolute memory? — NO

Each 16-bit storage location requires:

16 bits * [1 D-FF per bit]

= 16 bits * [2 D latches + 1 inverter]

= 16 bits * [1 inverter + 2 AND + 2 NOR] + 1 inverter

= $16 * (2 * (2 + 12 + 8) + 2) = 736$ transistors
for 1 16-bit register

& need more efficient memory &
SRAM and DRAM

SRAM and DRAM are volatile!

- means memory will be lost if power is lost

use non-volatile storage:

- hard disk (magnetic)
- Solid state drive (transistor based)

Lecture Notes:

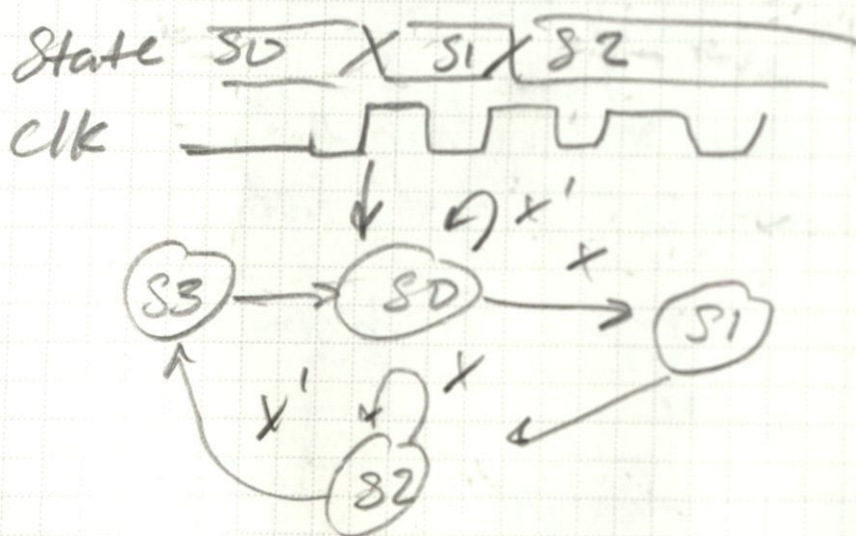
Finite State Machine:

Sequential circuit's behavior depends on its previous state

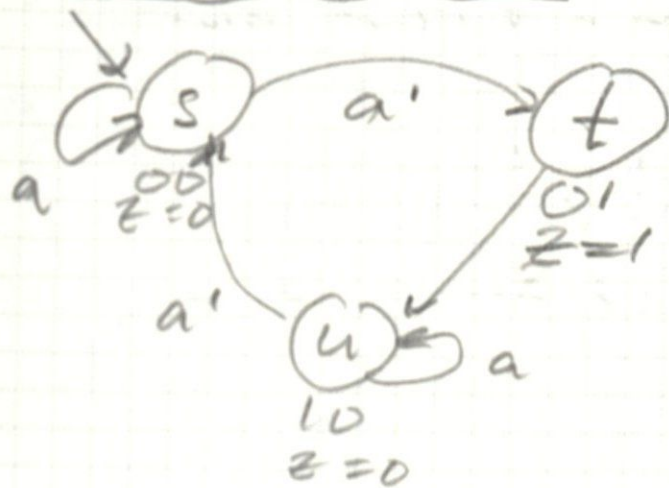
- Boolean eqns can't model these
- use FSM

FSMs define:

- A set of states
- Transitions in/out of those states based on inputs
- Actions (output) to take upon reaching each state



Components of FSMs:



- set of states
- transitions in/out of states
- actions to take

states? S, t, u
00 01 10

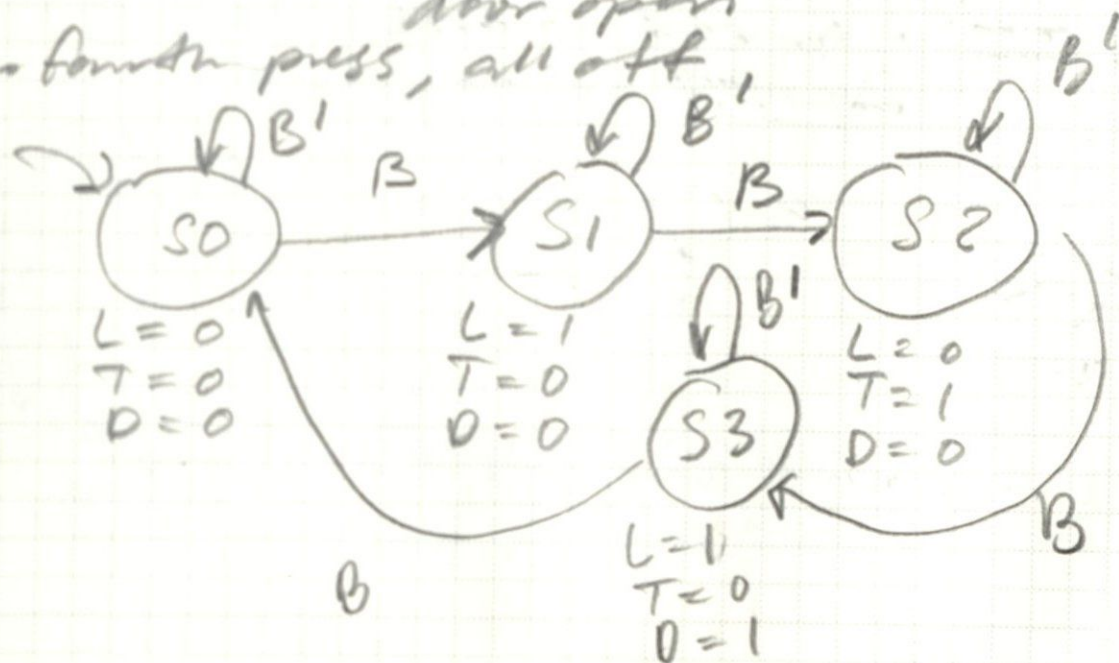
initial state: S
(arrow in)

inputs? a

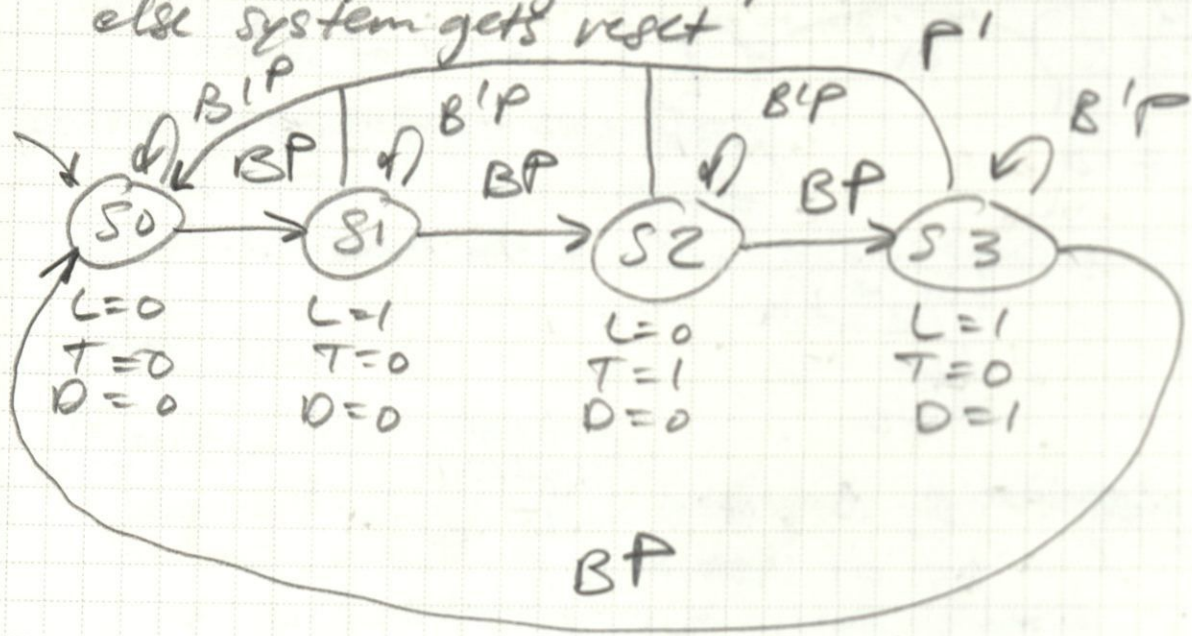
actions? (output) z

Ex. Design a FSM for a doorbell where:

- first press, lights turn on
- second press, lights off and tune on
- third press, tune off, lights on, door open
- fourth press, all off

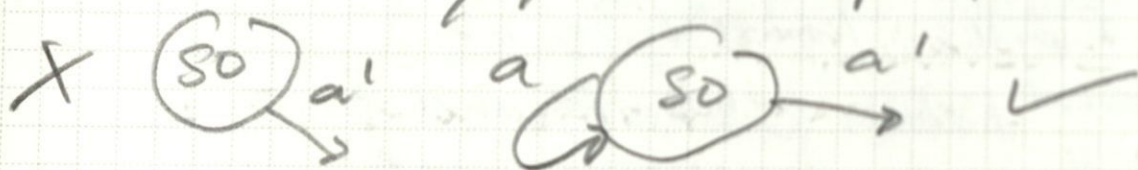


Ex. same as prev ex, but only works if currently standing on a pressure plate, else system gets reset

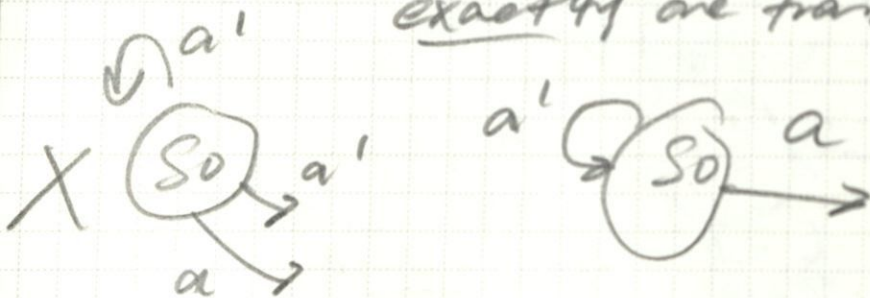


Existence and Uniqueness:

Existence: there is a transition for every possible input



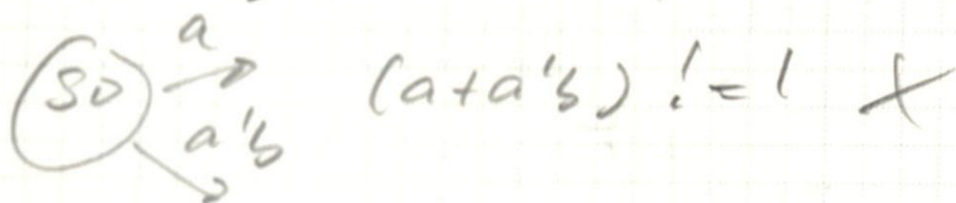
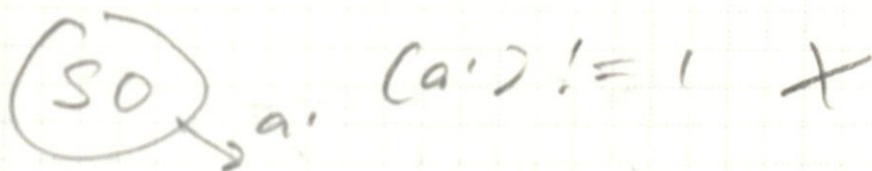
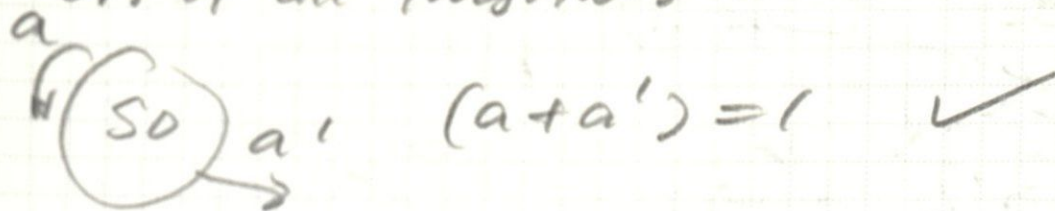
Uniqueness: for each condition, there is exactly one transition



GOAL - eliminate ambiguities in circuit

Test for Existence:

OR of all transitions = 1

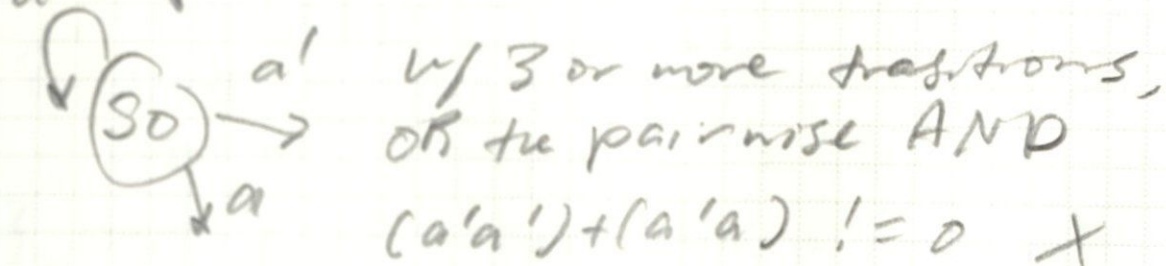
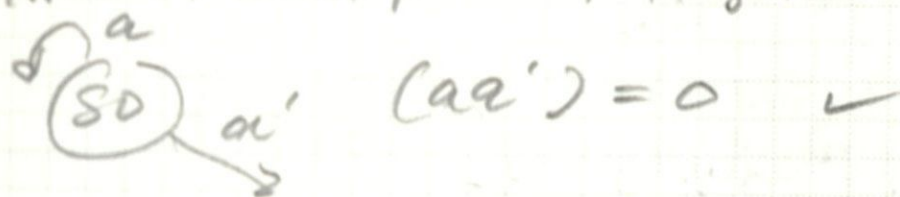


note. leave state regardless of input OK

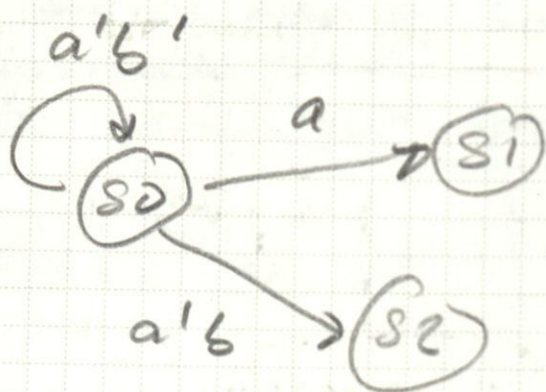


Test for Uniqueness:

AND of each pair of transitions = 0



Ex.



Existence:

OR of all transitions = 1

$$a'b' + a + a'b$$

$$a'(b' + b) + a$$

$$a' + a = 1$$

Uniqueness:

AND of all pairs of transitions = 0

$$(a'b')a +$$

$$(a'b')(a'b) +$$

$$a(a'b)$$

$$= b' \cdot 0 + a' \cdot 0 + 0 \cdot b = 0$$

3, 20, 24

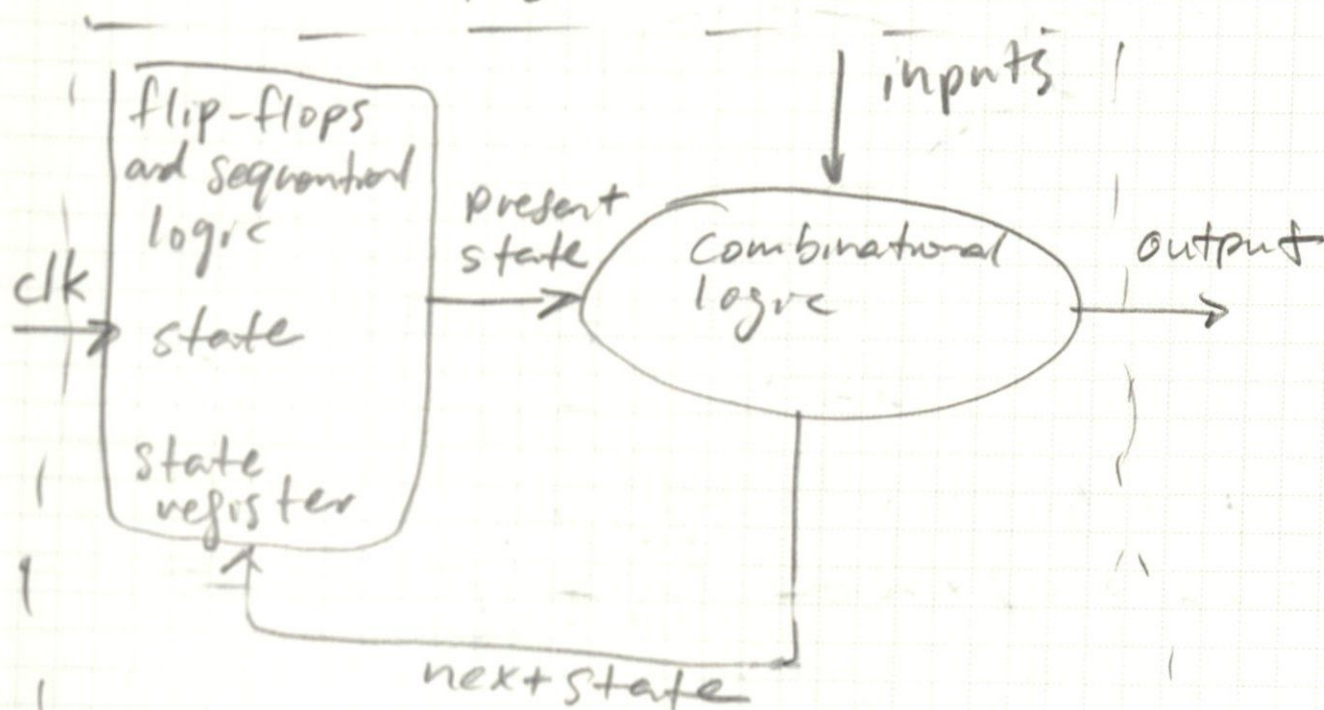
Lecture Notes:

can also use truth tables to prove existence and uniqueness

note. have to check all states of a FSM for completeness

FSMs in this class will boil down to...

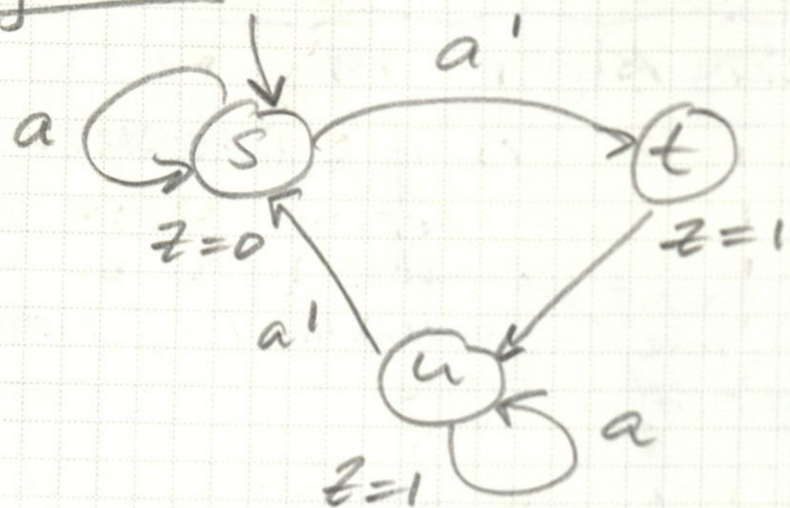
FSM



Design Process :

1. create a state transition table (truth table)
 - give a unique binary encoding to each state
 - create transition table with columns:
state bits, inputs, next state bits, outputs
 - define state transitions and actions for each state depending on inputs
 - fill table
2. derive minimized equations (using k-maps or autogenerated in LE)
3. connect combinational logic to state register

Design Ex.



State Transition Table:

columns: present state bits (p1, p0), inputs, next state bits (n1, n0), outputs

p1	p0	a	n1	n0	z
0	0	0	0	1	0

when in state 00 and input is 0

next state is 01 and output is 0 now in 00

inputs to combinational logic

outputs from combinational logic

p1	p0	a	n1	n0	z
0	0	0	0	1	0
0	1	0	0	0	0
0	0	1	1	0	1
0	1	1	1	0	1
1	0	0	0	0	1
1	1	0	1	0	1
1	1	1	X	X	X
1	0	1	X	X	X

en
outputs
earliest
to-do first

does not
matter

we will
use to our
advantage to
optimize our circuit

unused

p1	p0, a	00	01	11	10
0	0	0	0	1	1
1	0	0	1	1	1

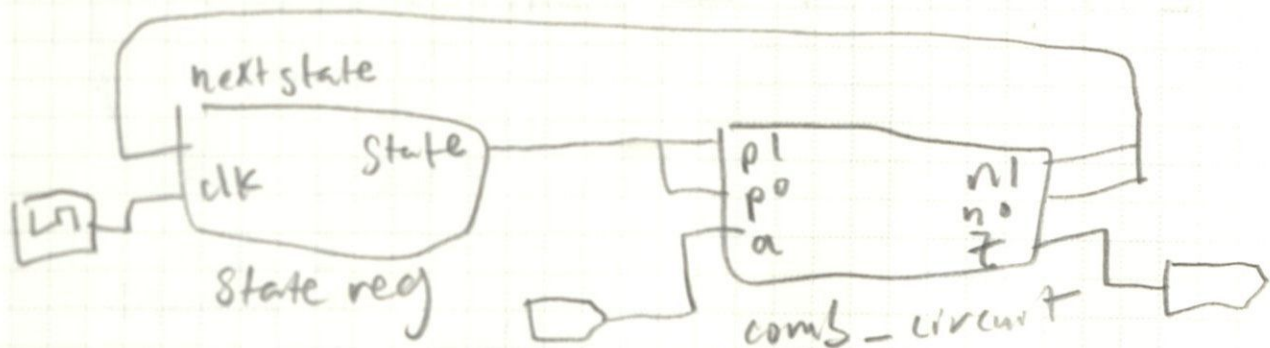
do for all outputs:
get eqns:

$$n1 = p0 + p1 \cdot a$$

$$n0 = p1 \cdot p0 \cdot a$$

$$z = p0 + p1$$

make circuit and put into schematic:

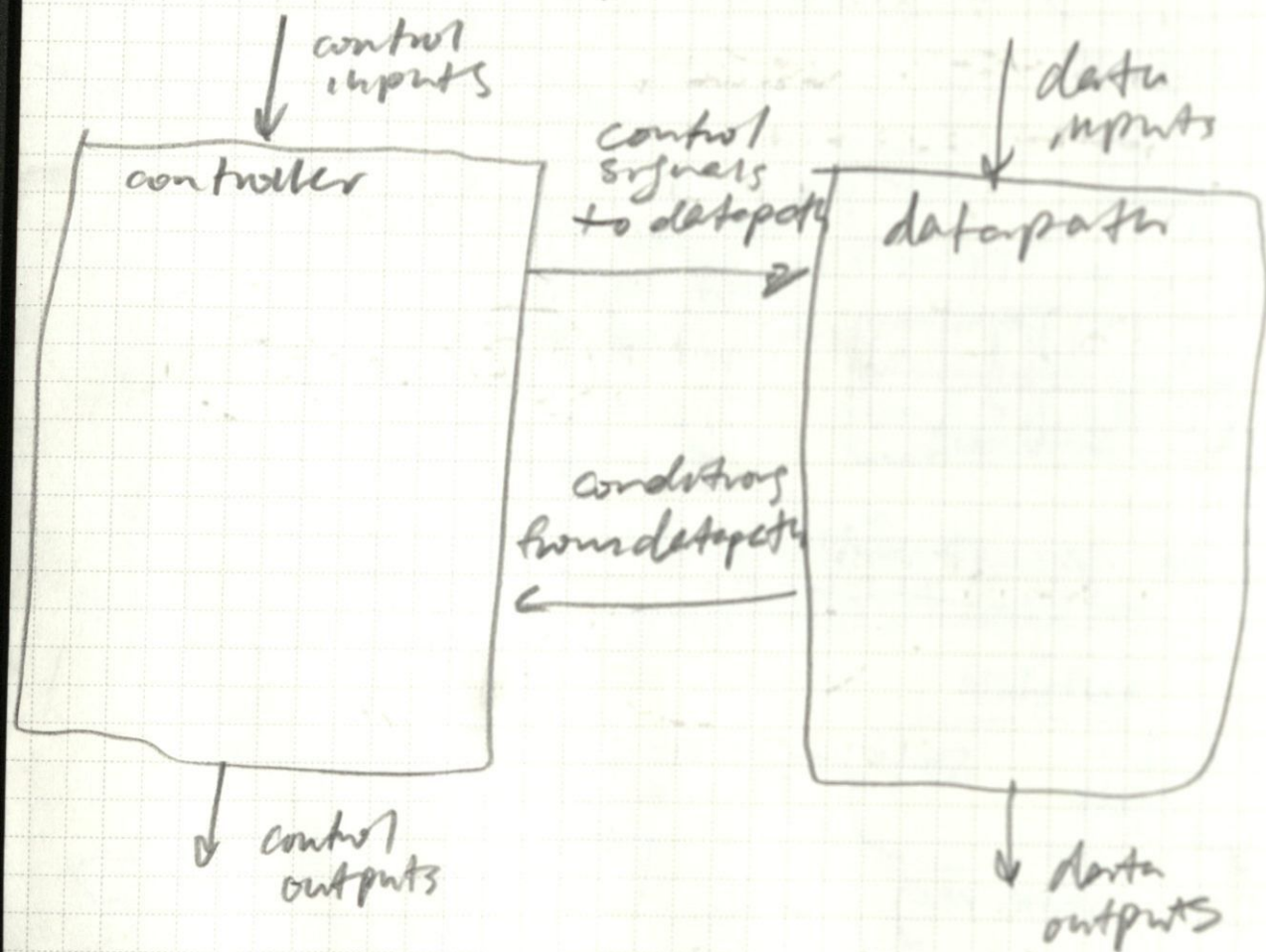


The other direction: circuit $\rightarrow$ FSM

- focus on combinational logic
- det next state and outputs for all combinations of current state and inputs

What's Left For Alibacore?

we need a controller to select unit signals at the right times



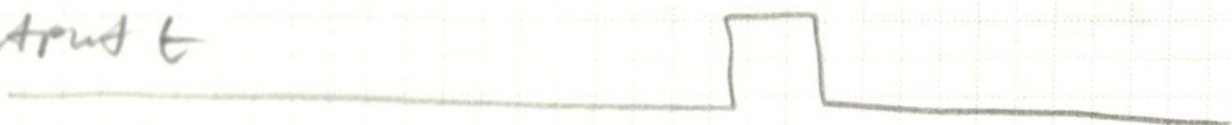
Timer Finite State Machine :

System has button b and output t
Push the button N cycles later, t lights
up for one cycle

input b



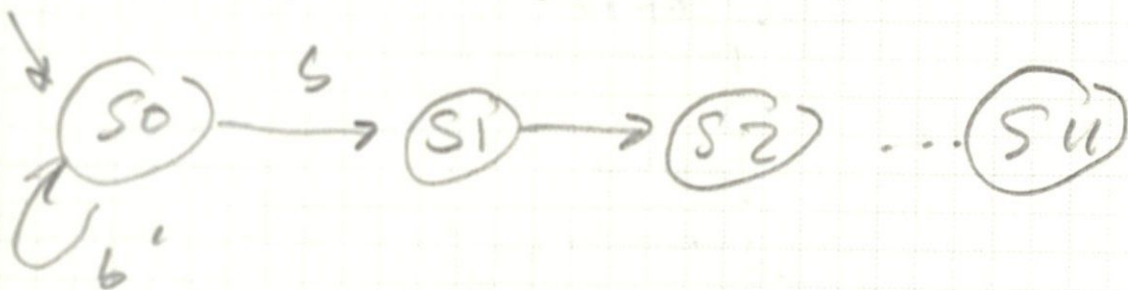
output t



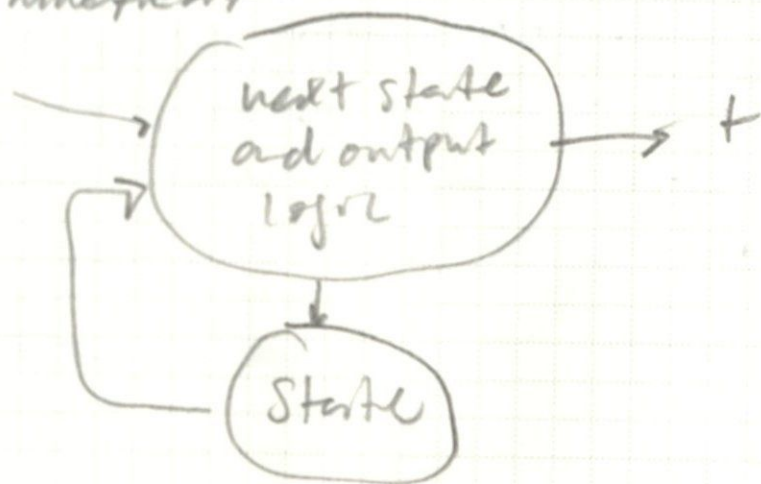
10 cycles

Parts :

- set of states
- start state
- $input(s) : \text{boolean}$
- $output(s) : \text{boolean}$
- transitions, conditions on inputs



- very inefficient



FSM vs. HLSM:

FSM:

- set of states
- start state
- input(s): boolean
- output(s): boolean
- actions(s)
- transitions: conditioned on input(s)

HLSM:

- set of states
- start state
- input(s): integers, chars, anything
- variable(s)
- actions/output(s), operations on variables
- transition: conditioned on operations on variables

Timer as a HLSM:

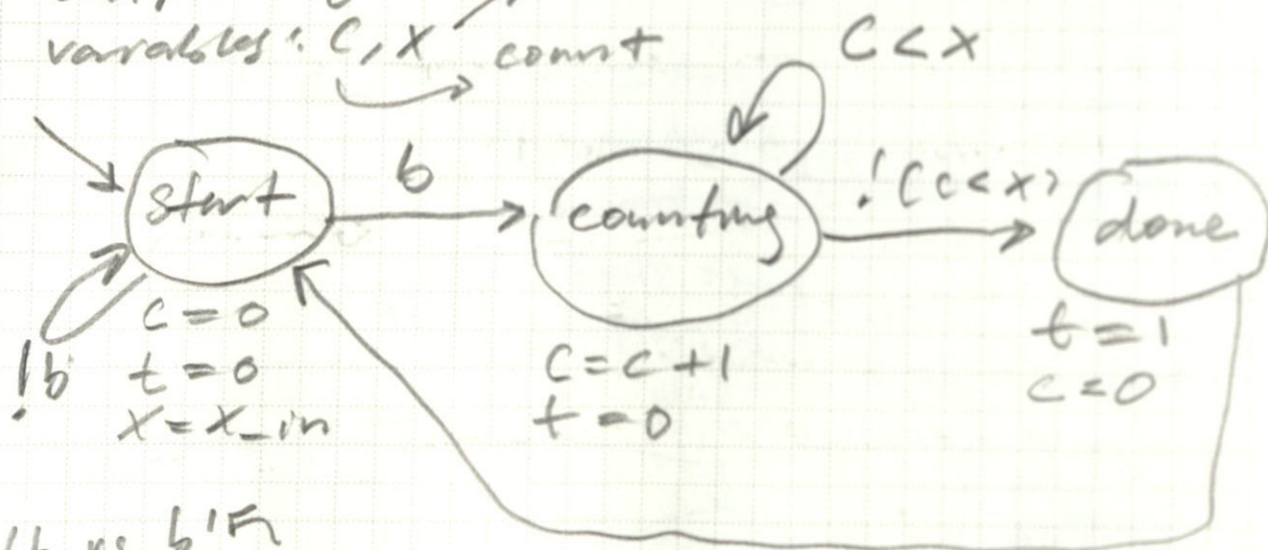
Goal. description of timer $\rightarrow$ circuit

Step 1. define inputs, outputs, variables and draw HLSM

input: b, x_{in}

output: t

variables: c, x $\rightarrow$ max count



$!b$ vs $b \text{ if}$
HLSM FSM

HSLM Table:

Step 2: define table describing HSLM

state	actions	transitions
START	$c = 0$ $t = 0$ $x = x_{in}$	if (6) goto counting else goto start
COUNTING	$c = c + 1$	if ($c < x$) goto counting else goto done
DONE	$t = 1$	goto start

* can write this as if (!b) goto ...
* if ($c >= x$) not best practice

RTL Design: HSLM $\rightarrow$ circuit process

Step 3. Design a datapath that supports the data operations

1. any operation that a FSM cannot handle

for our timer, vars, comparators, etc

Step 4. Design a controller to generate control signals for the datapath

1. is a FSM that takes HSLM inputs and inputs produced by datapath, and generates all datapath control signals

for our timer i.e., a signal to increment when $c < x$, etc...

Datapath Stages and Conditions:

support all data operations in HCSM

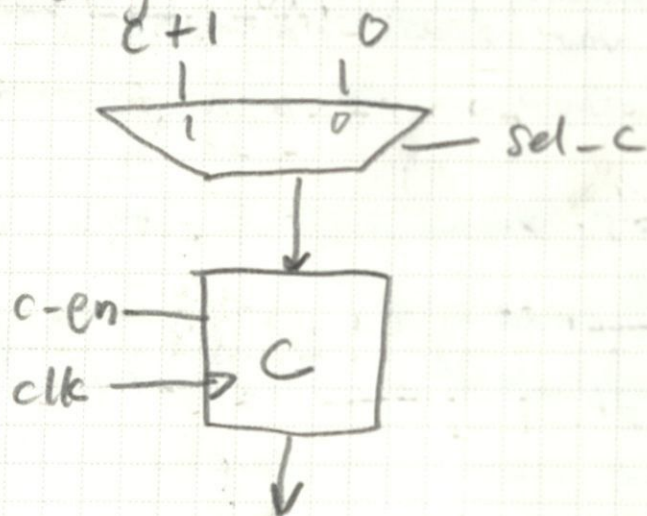
- assignment
- transition conditions

typically only multi-bit data operations

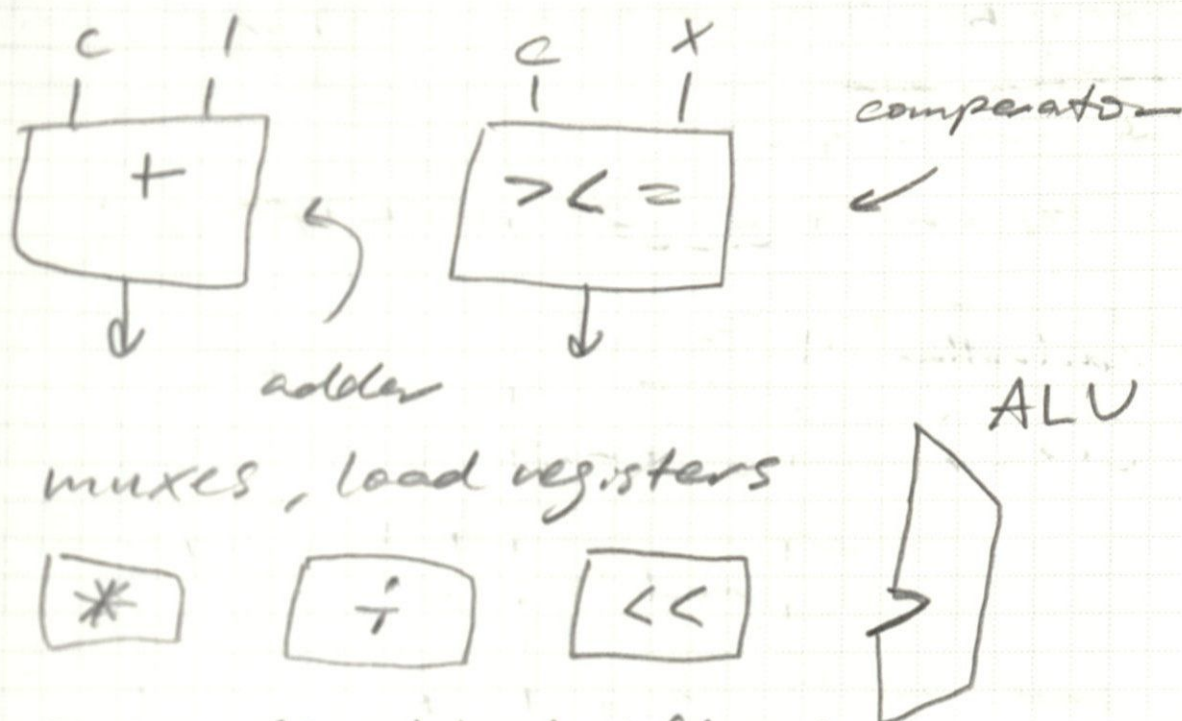
Creating a Datapath

1. Add load registers for each data output, variable
2. Instantiate components to perform each data operation
3. Add multiplexers if data outputs can receive multiple values

load register: has load-enable and rising edge loads



Datapath Components :



i.e. can use a lot of different components for datapaths

Datapath Stage Table :

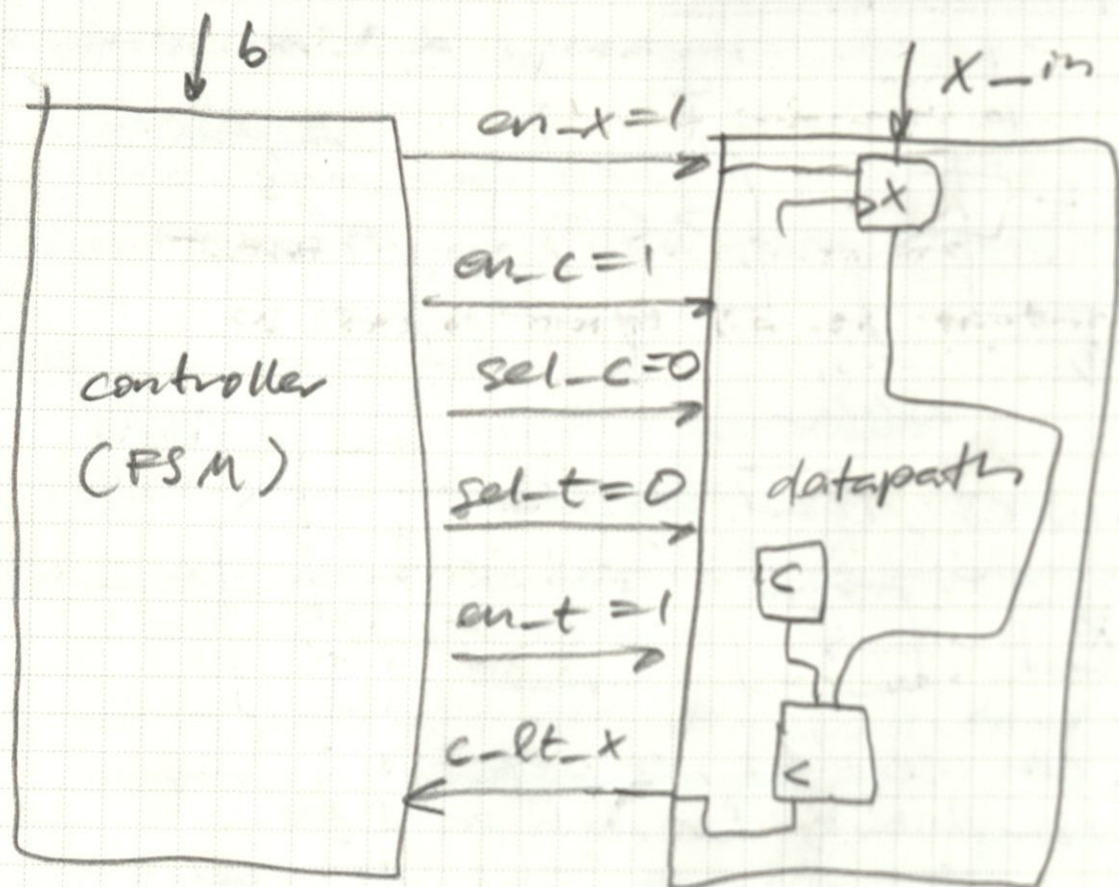
1. each output or variable needs a load register
2. Define the possible sources for load reg
3. Define control signals to control load register and all muxes

Load Register	Sources	Control Signals
C	0: 0 1: c+1	en-c, set-c
X	X-in	en-X
t	0: 0 1: 1	en-t, set-t

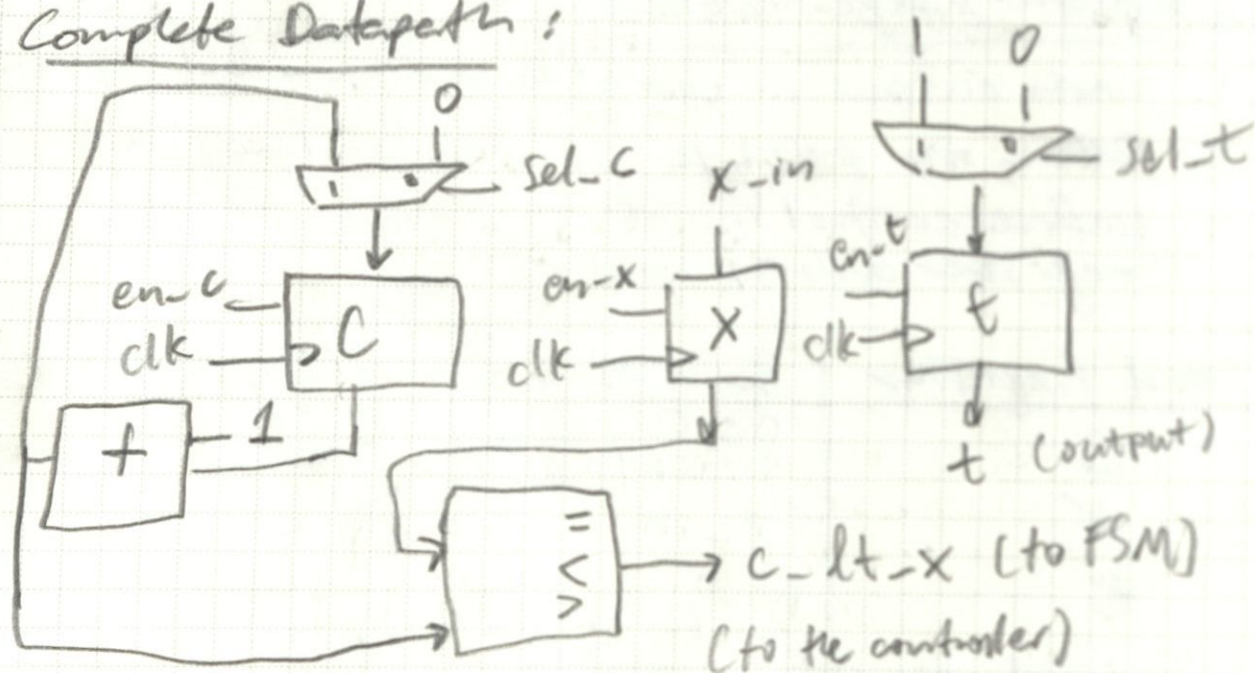
Give each transition condition a flag:

Condition	Flag
c < x	c - t - x
b	b

System Block Design: called a processor



Complete Datapath:



make sure to
avoid register delay and
get val of C after incremented

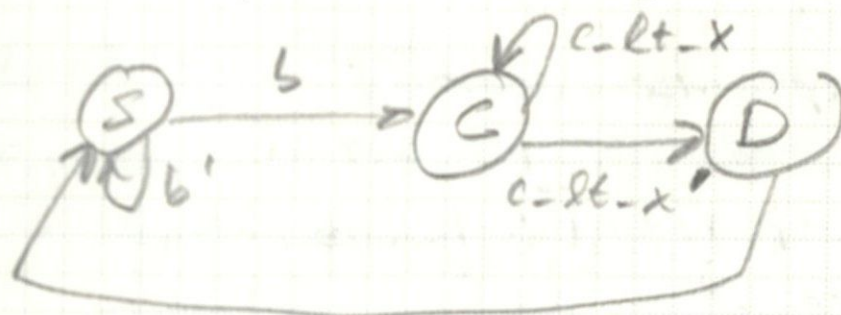
be careful of conditions
based on regs updated
in that state

Timer Controller: controller is a FSM with the same structure as HLSM

- inputs are original HLSM inputs plus any new signals from datapath
- outputs are all signal inputs to the datapath

Can represent with controller FSM table

state	actions	transitions
START	en-c=1 en-t=1 en-x=1	if (b) goto COUNTING; else goto START;
COUNTING	en-c=1 sel-c=1	if (c-lt-x) goto COUNTING; else goto DONE;
DONE	sel-t=1 en-t=1	goto START;



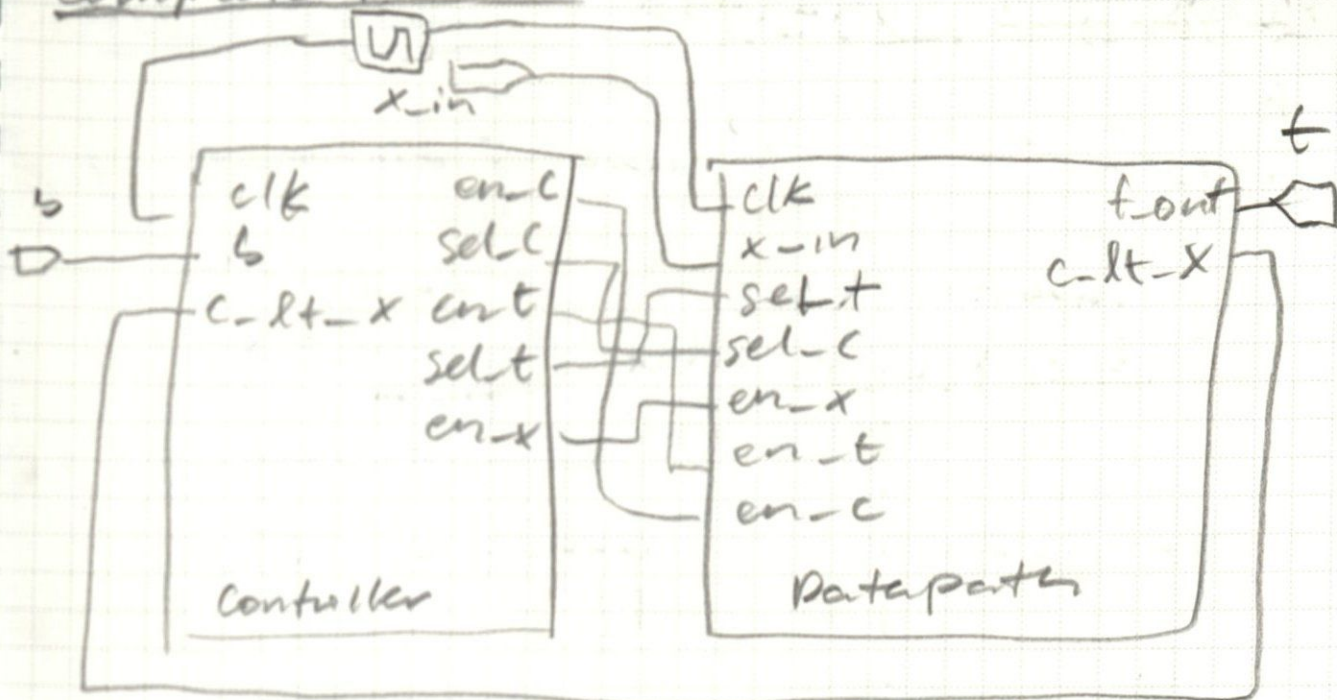
* en-x=1

en-c=1 sel-c=0

en-t=1 sel-t=0

can just say default is 0!

Complete Processor:



two inputs + clk : b , x_in

one output : t

Datapath : actually doing the work

Controller : telling the Datapath what to do

Handling the Count:

Problem. $t=1$ after 4 cycles when $x=3$

- starts counting at 0
- register for t adds delay

Fix by:

1. Start counting at 1
2. immediately decrement x
3. change c_lt_x to $c_lt_x - 1$

HLSM Ex #2: Maxfinder

follows familiar algorithm

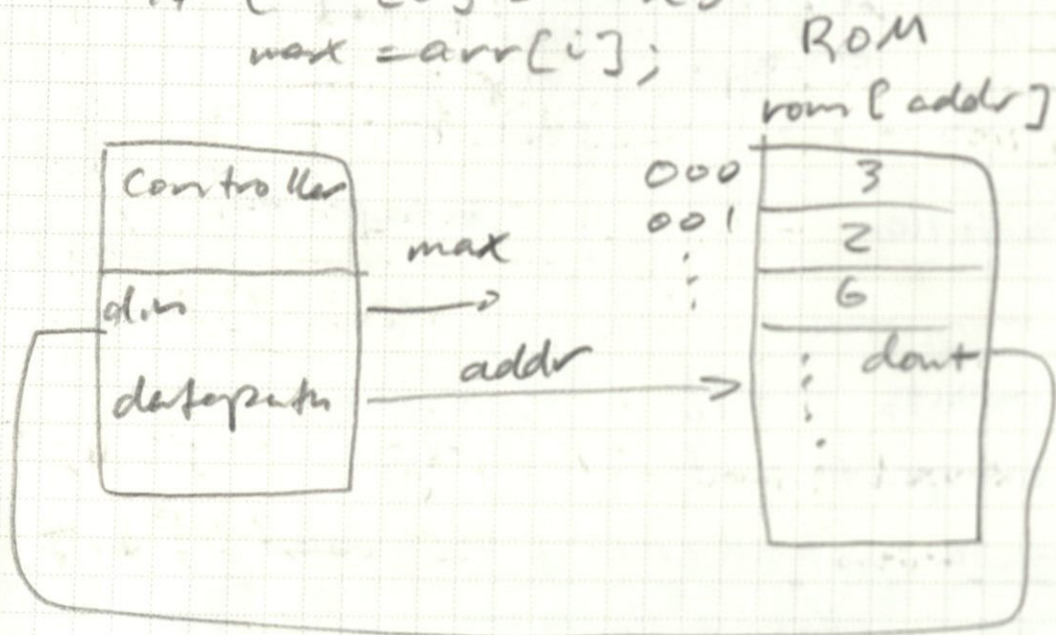
to assume arr exists with S elements

```
int max = 0;
```

```
for (int i = 0; i < S; i++)
```

```
    if (arr[i] > max)
```

```
        max = arr[i];
```

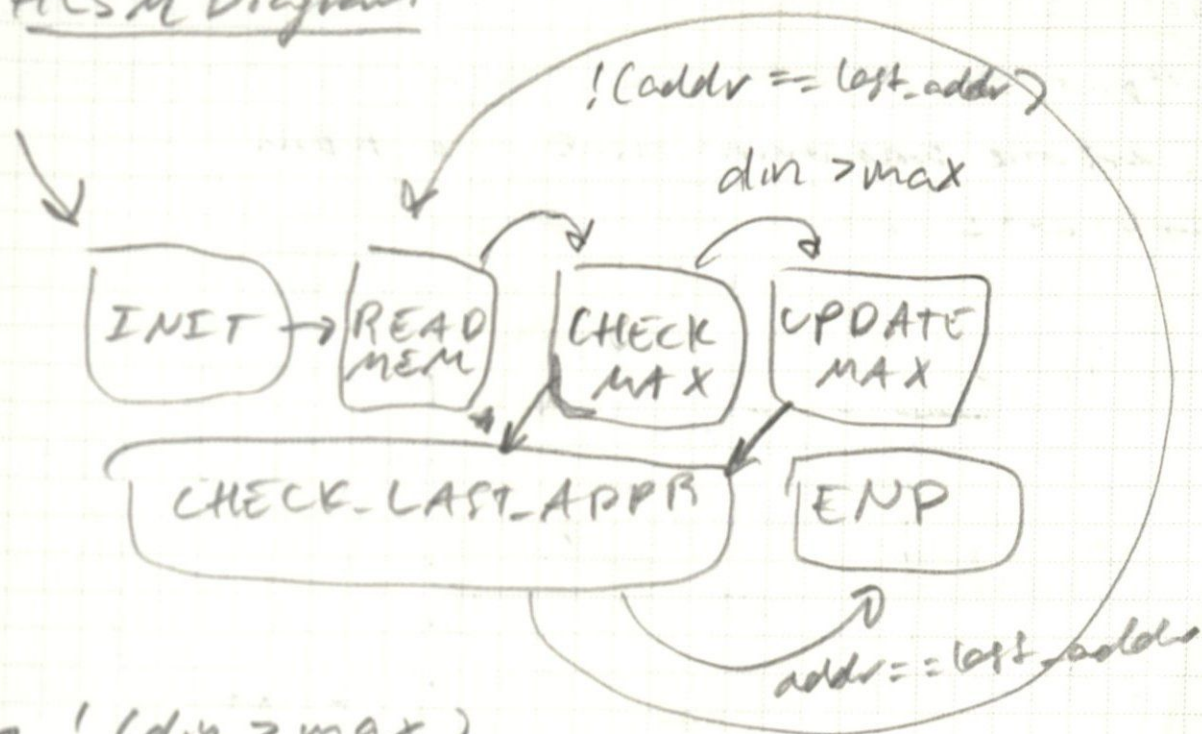


Find max value in a ROM!

HLSM Table:

State	Actions	Transitions
INIT	addr ← 0 max ← 0	goto READ-MEM; if (reset) goto INIT;
READ-MEM	data ← rom[addr]	goto CHECK-MAX;
CHECK-MAX		if (data > max) goto UPDATE-MAX; else goto CHECK-LAST-ADDR;
UPDATE-MAX	max ← data	goto CHECK-LAST-ADDR;
CHECK-LAST-ADDR	addr ← addr + 1	if (addr == LAST-ADDR) goto END; else goto READ-MEM;
END		goto END

HLSM Diagram



• ! (din > max)

Filled out table and clear diagram on slides!

What size state register will FSM need: 3 bits

Recall: ROM vs RAM

ROM: read-only memory RAM: random access memory

- only has a reader
- simpler

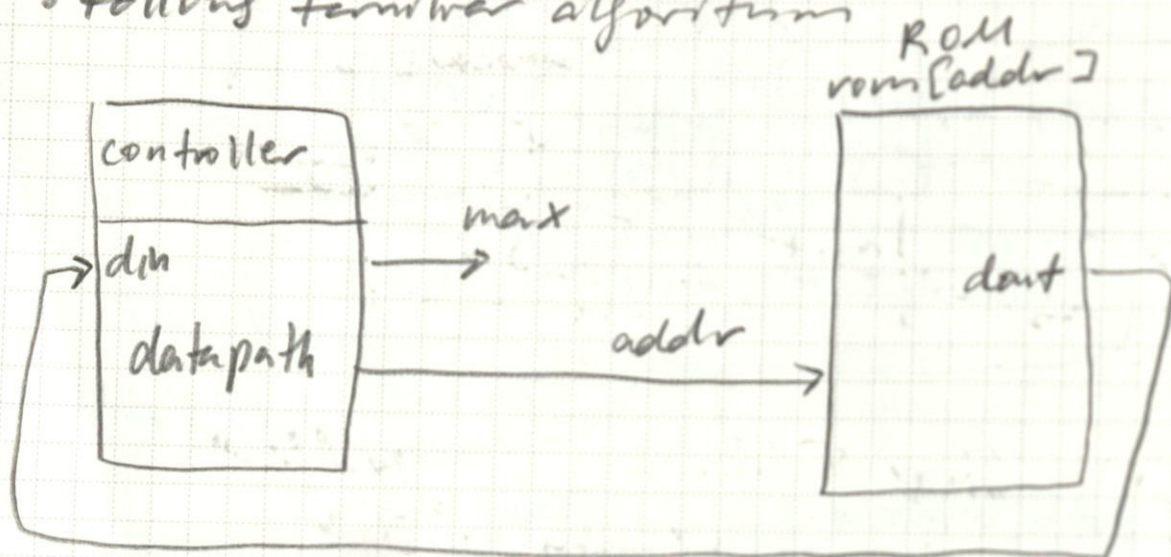
- also has
- writer
- din
- wen

Lecture Notes:

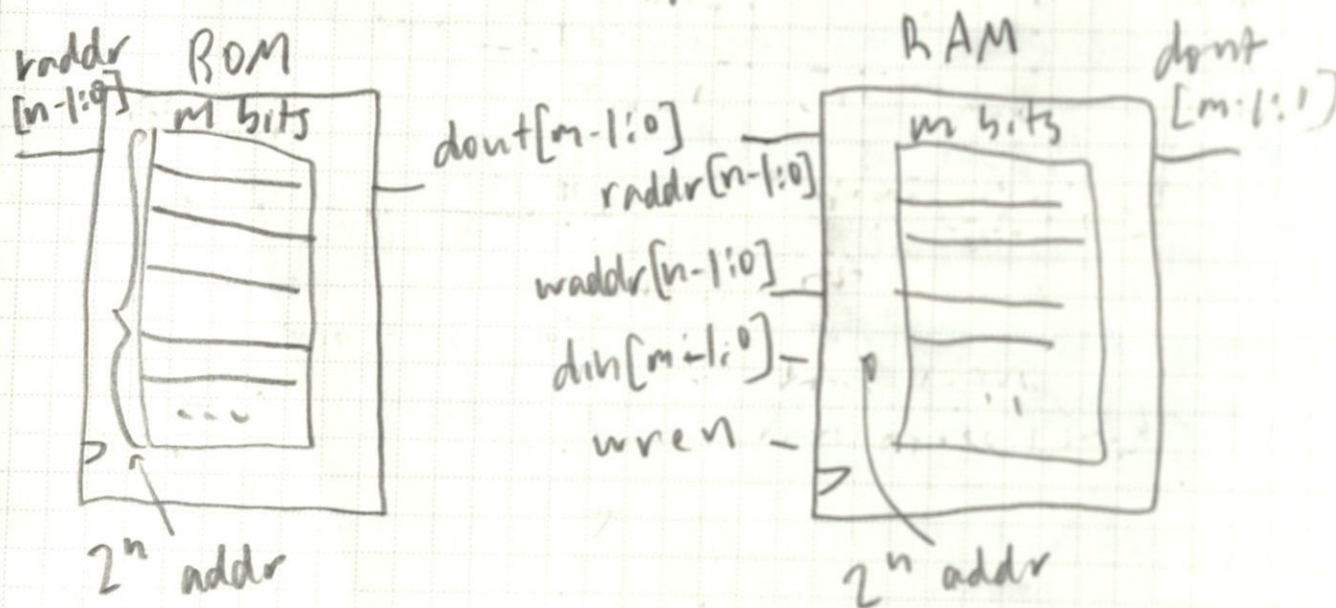
4.3.25

Recap: Max Finder

- Find the maximum value in a ROM
- Following functional algorithm



Recall: Asynchronous vs Synchronous Reads



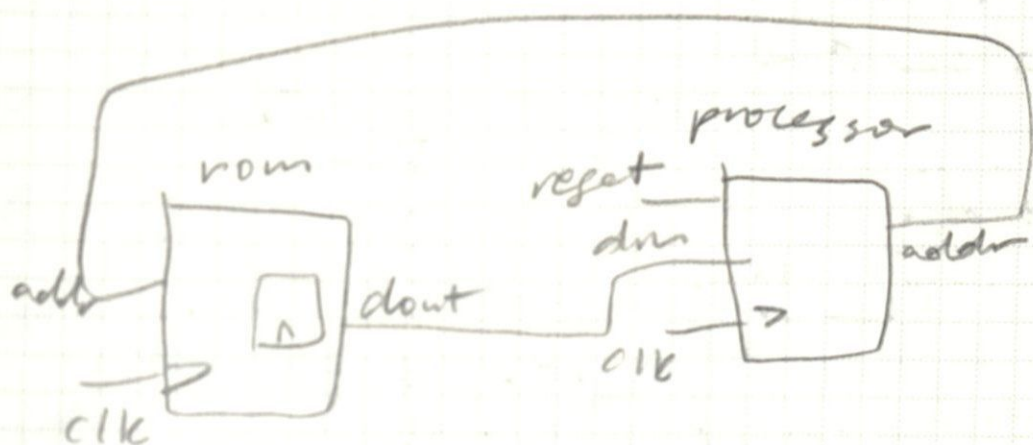
Asynchronous: read immediately on addr change

Synchronous: read on next rising clock edge

✳ writes always synchronous in this class!
✳ look in notes for difference in diagrams

Implementing Max Finder :

processor to find max value in synchronous ROM



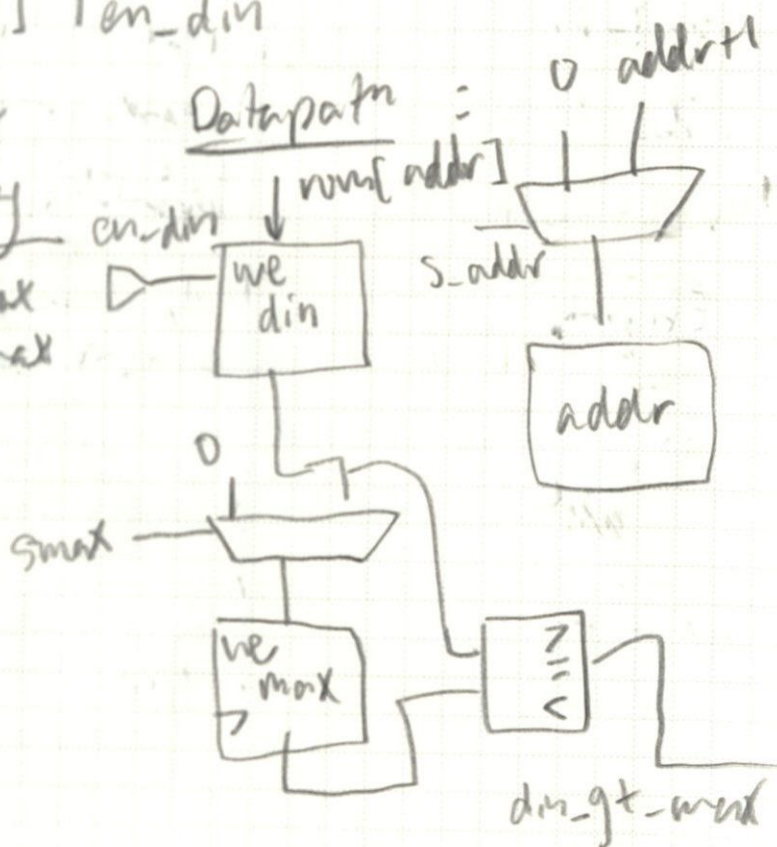
Datapath Stage Table

Destination	Source	Control Signals	enables selects
addr	0, addr+1	en_addr, S_addr	
max	0, din	en_max, S_max	
din	rom[addr]	en_din	

Condition/Flag Table

Condition	Flag
din > max	din_gt_max
addr == LAST ADDR	addr_eq_max

full datapath on logisim in lecture video



And then...

- translate HLSM Table $\rightarrow$ FSM Table
- create TF for FSM
- generate circuits
- ... connect datapath / controller ...

Back to albaCore:

Goal: use RTL design process to
implement albaCore processor as a HLSM
& albaCore HLSM as general processor
in slides

Five Stages of Instruction Processing:

Instruction	Memory Interaction
Fetch	read instruction from memory
Decode	read registers ra, rb use opcode to determine next step
Execute	Arithmetic, logic, comparison operations
Memory	Access memory for load, store
Write Back	write results to register file

Examples:

	add	st	ld
F	✓	✓	✓
D	✓	✓	✓
E	✓	✓	✓
M	x	✓	✓
WB	✓	x	✓

Lecture Notes:

4.15.25

architecture Registers:

Architecturally-visible: programmer/
compiler have access to these directly
"general purpose" registers

$R[0:15]$: register file (RF)

Internal working registers: not accessible
by programmer

a: value from "a" side of RF

b: value from "b" side of RF

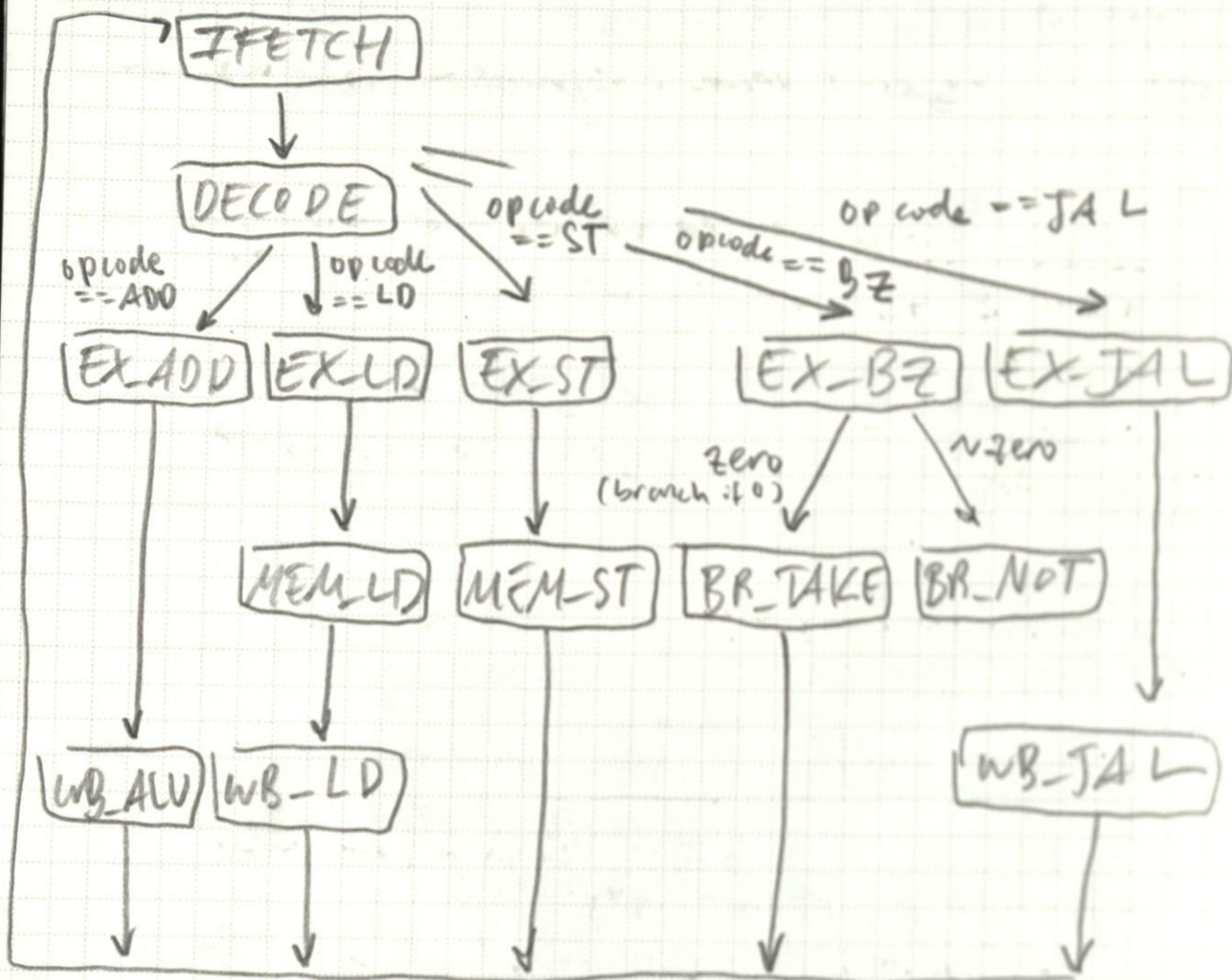
f: result of arithmetic/logic operation

inst: current 16-bit instruction

mdr: memory data register - data read from
memory

pc: program counter

alphaCore HLSM:



(alphaCore general purpose processor HLSM for subset of instructions)

* more detailed HLSM for specific instructions in slides

alphaCore Datapath - 1 instruction

Stage

Instruction Fetch: read instruction from memory

$$\text{inst} \leftarrow 0x0458 \quad \# \text{CC} - 1$$

Instruction Decode: use opcode to determine next step

$$\text{opcode } 0 \text{ to controller} \quad \text{CC} - 2 \quad \#$$

Register Fetch: read registers ra, rb

$$a \leftarrow 0x19, b \leftarrow 0x89 \quad \text{CC} - 2 \quad \#$$

Execute: ALU operations

$$f \leftarrow 0x19 + 0x89 \quad \text{CC} - 3$$

Memory: Access memory for load/store

Write Back: write results to register file
update PC

$$r10 \leftarrow 0xA2 \quad \text{CC} - 4$$

$$\text{PC} \leftarrow \text{PC} + 1 \leftarrow 95$$

$$\text{Ex. } \text{PC} = 94 \quad \text{--- } 0x19 \quad \text{--- } 0x89$$

94: add r10, r5, r8

CC - clock cycle

instruction decode and register fetch happen on same CC

Ex of data flow in datapath circuit explained in slides/recording

alphaCore Function Review:

ex. want to make a function "someFun"

.S file:

.text

```
100: jal someFun ← v15 holds address  
101: add v10, r1, v2      right after jal  
      qurt                (101)
```

someFun:

// do something

jr v15 ← jump back to main,
address 101

Functions calling functions:

```
int sumFun(n) {  
    if (n == 0) return 0;  
    else return (1 + sumFun(n-1));  
}
```

alphaCore:

```
jal sumFun  
addi  
dout
```

sumFun:

if recursive functions cause problems
we need to fix!

Functions Calling Functions Problem:

main:

jal sumFun

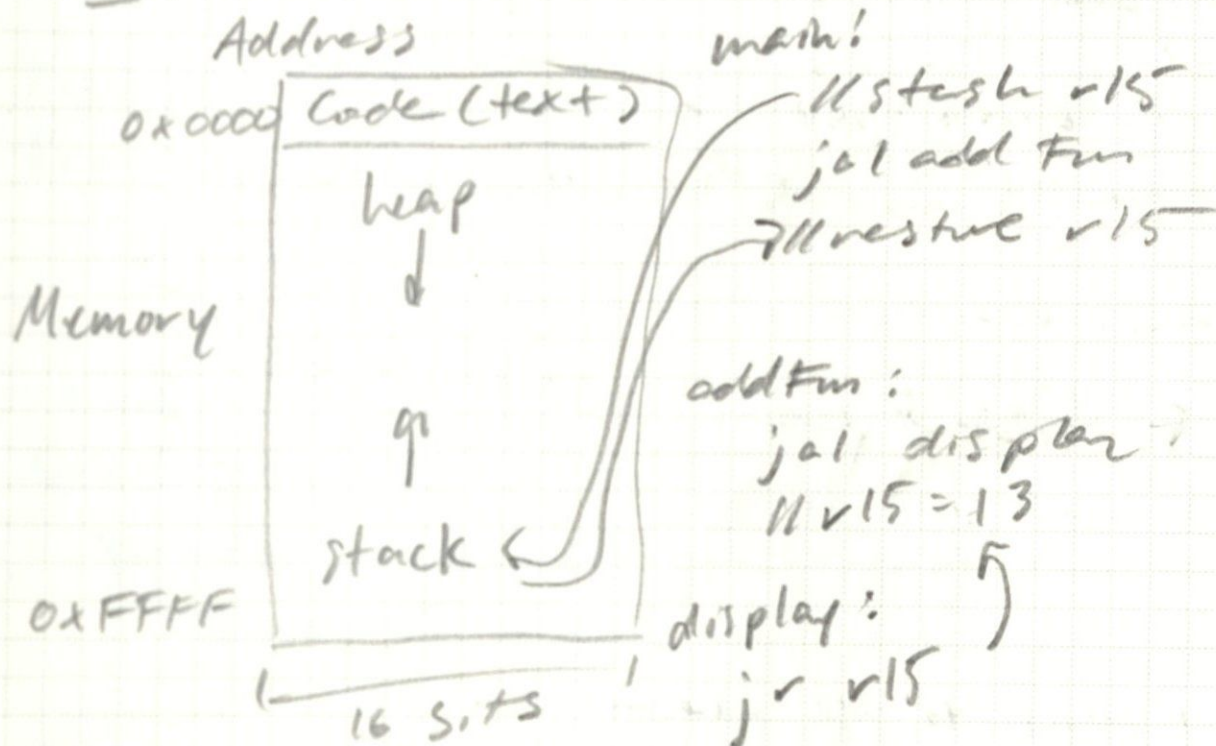
sumFun:

jal sumFun

- r15 is overwritten, don't know where to jump back to
- gets stuck and stays in sumFun function (infinite loop)

Solution: use the stack to store multiple return addresses

note. clobber = overwrite



albacore Solution to Nested Functions

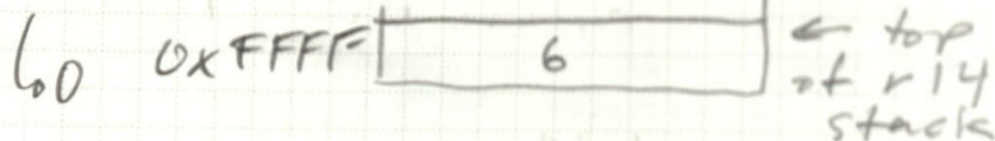
solution template - will be on final exam!

* $\text{ldi } r0, 1 \rightarrow 0$ can be a different register
 $\text{sub } r14, r14, r0 \rightarrow 0xFFFF$
 $\text{st } r15, r14, 0 \parallel M[0xFFFF] \leftarrow r15$

jal nested Function $\rightarrow r15 \leftarrow 3$

AA $\text{ld } r15, r14, 0 \parallel r15 \leftarrow M[0xFFFF]$

$\text{ldi } r0, 1$
 $\text{add } r14, r14, r0$



* "push"

* "pop"

albacore Calling Conventions

register conventions, on albacore sheet

* must use these on exam *

Also, function X (caller) calls function Y (callee):

- X must save $r0-r9$ to stack if X wants to ensure they are unchanged after Y
- Y must save $r10-r15$ to stack if Y wants to use them

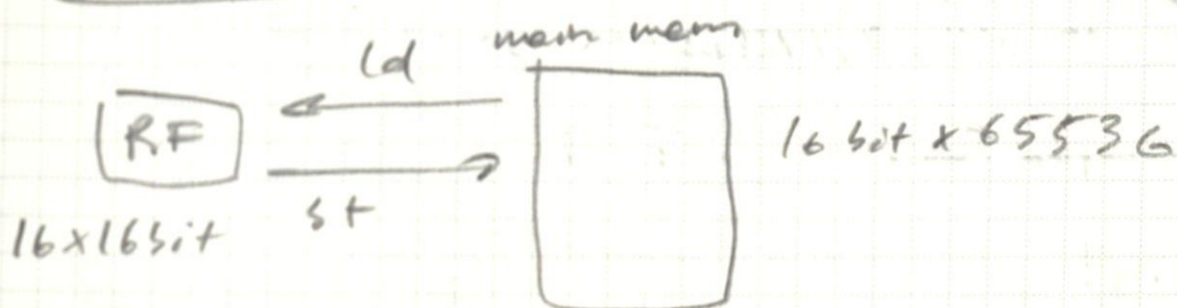
* caller vs callee saved ex in slides

Memory: Definitions:

storing data, temporary, fast, RAM

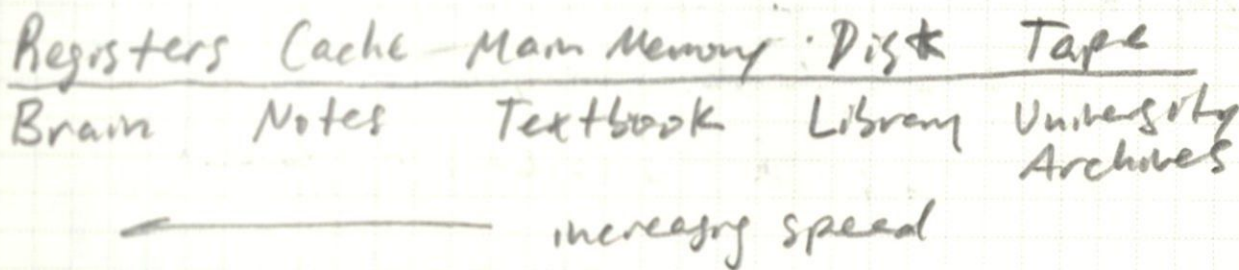
(16 GiB = $16 \cdot 2^{30}$ B)

in all alone:

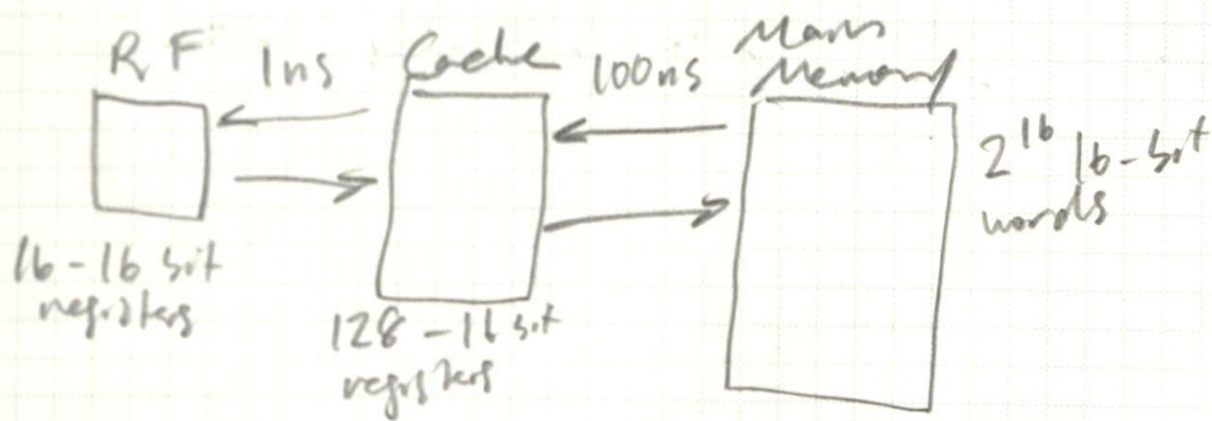


Cache: first attempt at optimization
"Bridge to Computer Architecture"

Memory Hierarchy:



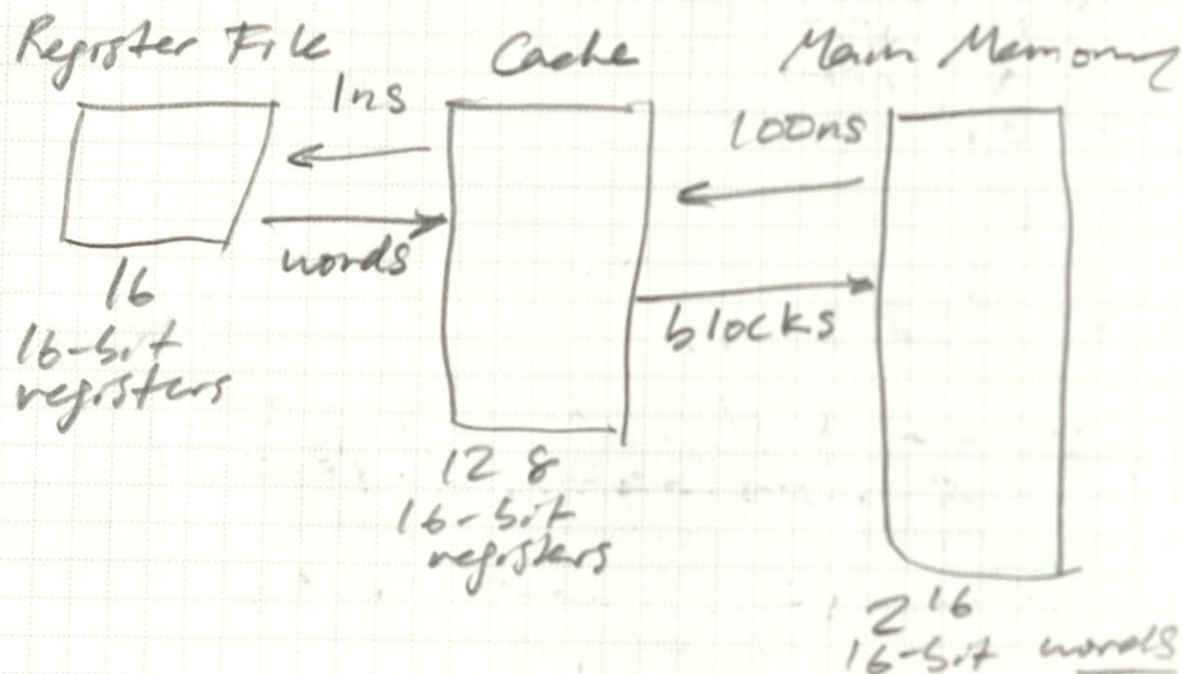
New Memory Hierarchy:



Memory Hierarchy: design the hierarchy based on two principles of locality

Temporal locality: recently used information likely to be reused

Spatial locality: nearby information is also likely to be used



Word: minimum addressable unit (1d/5t)
abalone: word is 16-bit

Block: minimum amount of data moved within the memory hierarchy

Cache Basics.

Ex.

<u>Main Mem</u>			100ns	<u>Cache</u>		Ins	<u>Register</u>
Addr	Value			Index	Data		
0	0	i					
1	10	$A[0]$	$\rightarrow$	0	01		☐
2	4	$A[1]$	$\rightarrow$	1	10		01
3	9	$A[2]$	$\rightarrow$	2	11		
4	22	$A[3]$	$\rightarrow$	3	9		
5	17	$A[4]$					
6	8	$A[5]$					
7	86	$A[6]$					

```
for (i=0; i<100; i++)
    print (A[i]);
```

Assume - 1 register, can only access i ,
A causes memory traffic

Each index stores 1 word of data
 $\rightarrow$ 1 word block size

ld from address	index	
0	0	M
1	1	M
0	0	H
2	2	M
0	0	H
3	3	M
0	0	H
4	0	M
0	0	M

} conflict

Ex. Block size : 4 words

Mem Addr	Mem Data
0	6
1	10
2	4
3	9
4	22
5	17
6	8
7	86
⋮	

i
 $A[0]$
 $A[i]$
 $\vdots$

Addr
0000
0001
0010
0011
0100
⋮

index chosen
by smallest
bits

	11	10	01	00
Cache Index	word 3	word 2	word 1	word 0
0	$A[2]$	$A[17]$	$A[0]$	i
1	$A[6]$	$A[5]$	$A[4]$	$A[3]$
2				
3				

Direct-mapped Caches:

both exs are direct-mapped caches
 each address maps to exactly 1 cache
 location

$$\lg = \log_2$$

\$ - cache

Index: #bits = $\lg(\# \text{ blocks in } \$)$

in Ex 1 . 1 word a block

Address	0000	maps to index	0
	0001		1
	0010		2
	0011		3
	0100		0
	10000		0
	<u> </u>		
	index		

in Ex 2 . 4 words a block

Address	0000	maps to index	0
	0001		0
	0010		0
	0011		0
	0100		1
	<u> </u>		
	index	block	
		offset	

10000

0

Block offset: if a block comprises of multiple words

$$\# \text{ bits} = \lg(\# \text{ words per block})$$

Tag: leftover bits

$$\# \text{ bits} = \text{total \# bits in addr} - \# \text{ index bits} - \# \text{ block offset bits}$$

Default Assumptions

- 16-bit, word addressable system (all values)
- word is 16 bits = 2 bytes
(2 bytes per word)

Putting it All Together :

Address (hex)		Data	3FEC			
			Index	Data	Tag	Valid
0000	0	4				
0001	1	8	0 00	42	0000	1
0002	2	15				
0003	3	16	1 01	8	0001	1
0004	4	23	2 10			0
...			3 11			0
000C		94				
000D		42				
000E		1138				
000F		111				
...						
FFFO		42				
FFF1		526				
FFF2		38				
FFF3		27				
...						
FFFF		17				

3 id's from
0x 0006
0001
FFFO

Addresses : 0 0 0				index
0000	0000	0000	0000	0000 M
0001	0000	0000	0000	0001 M
	3	F	F	C
FFFO	111	111	111	0000 M

\$ lookup

- 1) check valid bit
- 2) check tag

invalid
invalid
tag mismatch

Tag, Index, Offset Calculations: (4 words per block)

Addr (hex)	Data
0000	
0001	
0002	
0003	
0004	
...	
FFFC	
FFFD	
FFFE	
FFFF	

bit	15	4	3	2	1	0
	tag			index		offset

→ 2⁽²⁾ words/block
 ↳ offset bits
 $(\lg(\frac{\# \text{ words}}{\text{block}}))$

Index	Word 3	Word 2	Word 1	Word 0
00				
01				
10				
11				

→ 4 blocks

2⁽²⁾ blocks/cache

↳ index bits

$(\lg(\# \text{ blocks}))$

bits for tag: leftovers

Check for hit or miss:

1. Valid bit must be 1
2. index and tag for addr match
 index and tag in cache

Putting it All Together:

Addr(hex) | Data

0000	4	A[0]
0001	8	A[1]
0002	15	A[2]
0003	16	A[3]
0004	23	A[4]
...		
000C	94	
000D	42	
000E	1138	
000F	111	
...		
FFF0	42	
FFF1	326	
FFF2	38	
FFF3	27	
...		
FFFF	17	

1 block = 4 words
 would need new block here

Block offset = 1

↓ B0 = 0

Cache Index	Word 3	Word 2	Word 1	Word 0	Tag	Valid
0	16	15	8	4	0x000	1
1						0
2						0
3						0

ld from addr:

hex					I	B0	
0000	→	0000	0000	0000	00	00	M valid 0
0001	→	0000	0000	0000	00	01	H
FFF0	→	1111	1111	1111	00	00	M tag mismatch

- would cross out tag 000 → FFF
- bring in new data
- valid data still 1 because still has valid data

Calculation Summary :

- $\lg = \log_{\text{base-2}}$
- Given N -bit machine, All out fields for tag, index, offset
- For direct-mapped-cache, determine #bits for each field
 - offset: $\lg(\# \text{ words per block})$
 - index: $\lg(\# \text{ blocks in cache})$
 - tag: $N - \# \text{ bits index} - \# \text{ bits offset}$ (remaining bits)

Bit range:

N-1				0
-----	--	--	--	---

Field:

Tag	Index	Block offset
-----	-------	--------------

Sidenote : not on exam

- these are word-addressable machines (also Core)
- byte-addressable machines also exist
 - need another offset

Ex. 2 KiB direct-mapped cache
8 words per block; 32-bit word-addressable system

$$2^1 \cdot 2^{10} \text{ B} = 2^{11} \text{ Bytes}$$

$$\frac{8 \text{ words}}{\text{block}} = 2^3 \frac{\text{words}}{\text{block}}$$

$$\# \text{ bits offset} = \lg(2^3) = 3 \text{ bits}$$

$$\frac{2^{11} \text{ Bytes}}{\text{cache}} \cdot \frac{1 \text{ word}}{2^2 \text{ bytes}} \cdot \frac{1 \text{ block}}{2^3 \text{ words}} = 2^6 \frac{\text{blocks}}{\text{cache}}$$

$$\# \text{ bits index} = \lg(2^6) = 6 \text{ bits}$$

$$\# \text{ bits tag} = 32 - 6 - 3 = 23 \text{ bits}$$

note: $K_i = 2^{10}$

$$M_i = 2^{20}$$

$$G_i = 2^{30}$$

$$16 \text{ M.B}$$

$$2^4 \cdot 2^{20}$$

$$= 2^6 \frac{\text{blocks}}{\text{cache}}$$

Simulation Summary:

- After calculating the bits for tag, index, offset

Bit range:	15	5	4	2	1	0
Field:	Tag		Index		Block offset	

- For a list of accesses eg. sequence of ld/st instructions

1. Determine where it maps in the cache i.e. which block
2. Determine if the access is a hit or miss

Hit: the data is in the cache

- Must be valid, and tag in cache must match tag of access

Miss: the data is not in the cache

- Due to invalid or tag mismatch
- Bring data into cache, mark as valid, evict existing block if needed

Note. Final exam will only be about ld's.

What about writes? two options

write-through: write memory immediately on a st

write-back: write memory only when necessary

Lecture Notes:

4.29.25

Exam 1 Review: 8 questions

- Number Representation
binary, hex, signed vs unsigned, two's complement, overflow
- albacore
instruction encodings, C $\leftrightarrow$ albacore
- Boolean Algebra
minterms, sum of minterms, product terms, etc, minimization using boolean algebra, minimization using K-maps
- Logic Gates
AND, OR, NOT, NAND, NOR, XOR
- Combinational Logic Components
Demuxes, muxes, decoders, adders, subtractors (no carry lookahead)

Exam 2 Review: 6 questions

- Sequential Logic Components
SR latch, D latch, D Flip-Flop
- Registers, Memory, Counters, Timers
local Registers, shift registers, ROM/RAM, etc.
- Finite State Machines
problem statement $\rightarrow$ FSM design, existence/uniqueness, design (FSM $\rightarrow$ circuit)

• High-level State Machines

Problem statement $\rightarrow$ HLSM,

RTL design (HLSM $\rightarrow$ circuit),
albalone instruction processing for
ALU instructions (add, sub)

Also specific exs from class! *

Post-Exam 2 Review: ~17 questions

• Back to albalone

Purpose of and use of stack for nested
function calls, register usage conventions,
instruction processing steps/stages in
general and in albalone

• Direct-mapped Caches

Purpose, calculations, simulations

Review.

post-Exam 2 albalone

5 stages of instruction processing

F - instruction Fetch

D - instruction Decode / Reg. Fetch

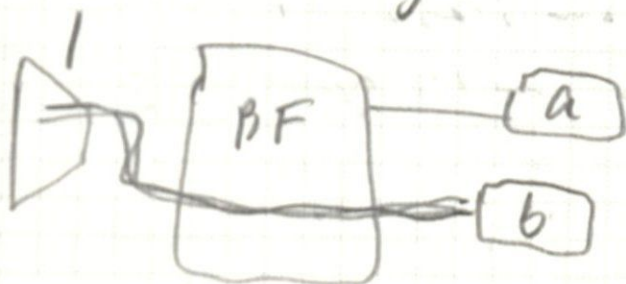
E - Execute

M - Memory

W - Write Back

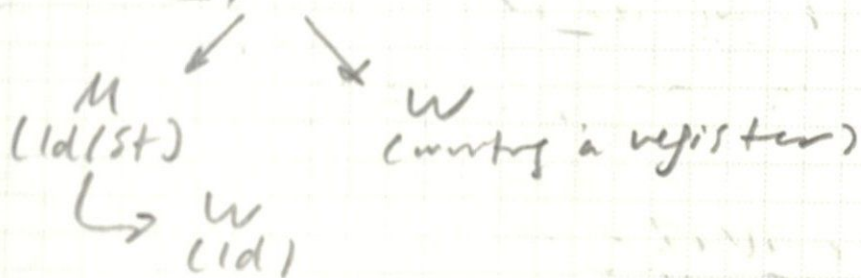
Decode.

- 1) instruction register's contents are fed to the controller, proceed of the datapath
- 2) $ra, rs \rightarrow a, b$
2 register file registers' contents to internal registers



For specific instruction, what stages do they go through?

minimally: FDE



Dealing with Nested Function Calls:

main:

```
jal fun // sets r15 to pc+1  
add r0, r1, r2  
qwert
```

fun:

```
jal fun2 // set r15 to pc+1  
sub r4, r5, r6  
jr r15 // go back to main @ the add
```

stuck in loop here since it's pointing
to sub from fun2

Solution:

r14 is stack register and starts
at top of memory

fun:

```
ldi r5, 1 // r14 = FFFF  
sub r14, r14, r5 // r14 = FFFE  
st r15, r14, 0 // M[FFFE] ← r15  
// (address of add in main)
```

```
jal fun2 // r15 ← address of sub  
sub r4, r5, r6  
ld r15, r14, 0 // r15 ← M[FFFE]
```

ldi r5, 1

add r14, r14, 1 // r14 ← FFFF

jr r15

(set r14 back to original)

(to main)

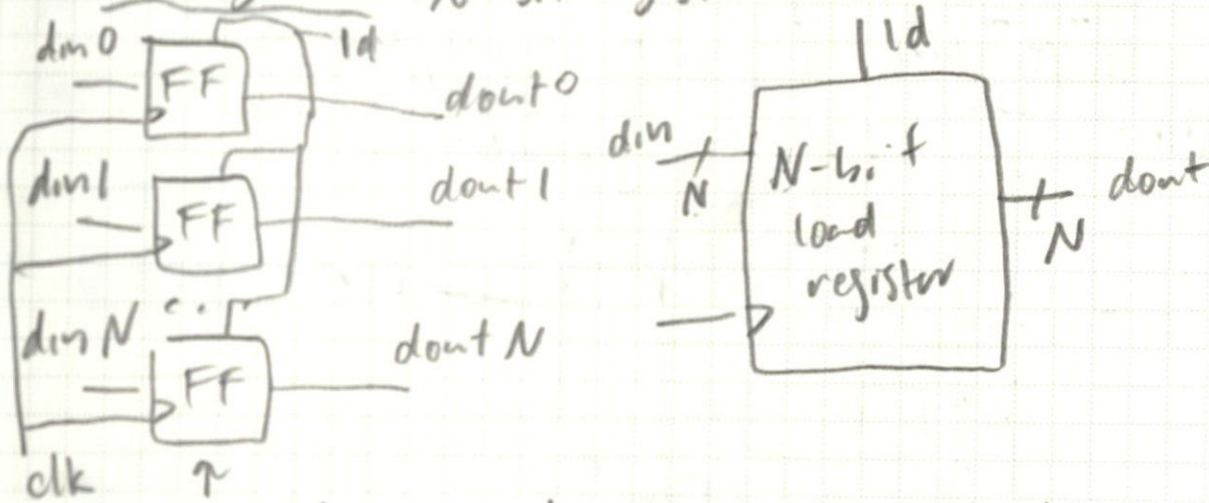
Usage / Calling Conventions:

- each register has a purpose
 - eg. $r14$ is the stack pointer
 - $r15$ is the return address
 - $r0$ is used to return values from fncs
- * all in abelwe sheet
- responsibility for saving/restoring register values across function calls
 - eg. $r10$: callee-saved register
 - function caller (you or compiler as function is being written)
 - must save $r10$ before overwriting
 - and must restore it before returning

load / shift registers:

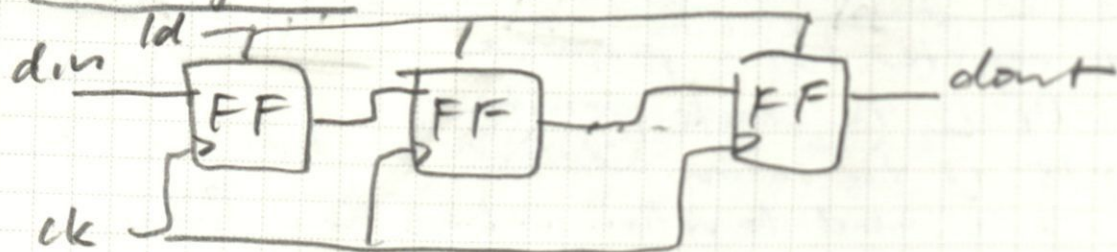
both created w/ D-FlipFlops, but connect them in series vs. in parallel

load register: N -bit register $\Rightarrow N$ Flip Flops



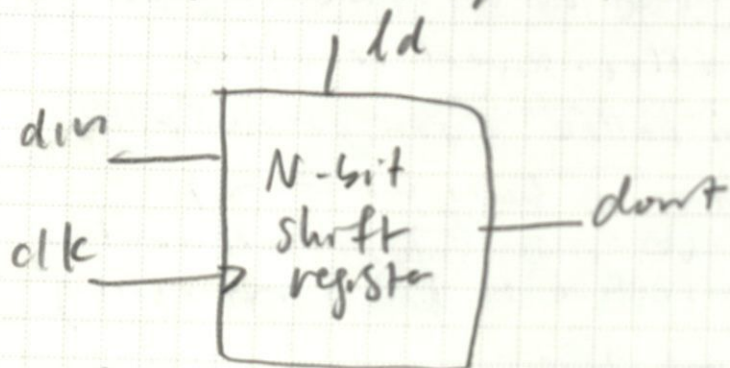
in parallel; can load entire register in single clock cycle

shift register:



N-bit register $\Rightarrow$ N registers

in series: takes N clock cycles to load register

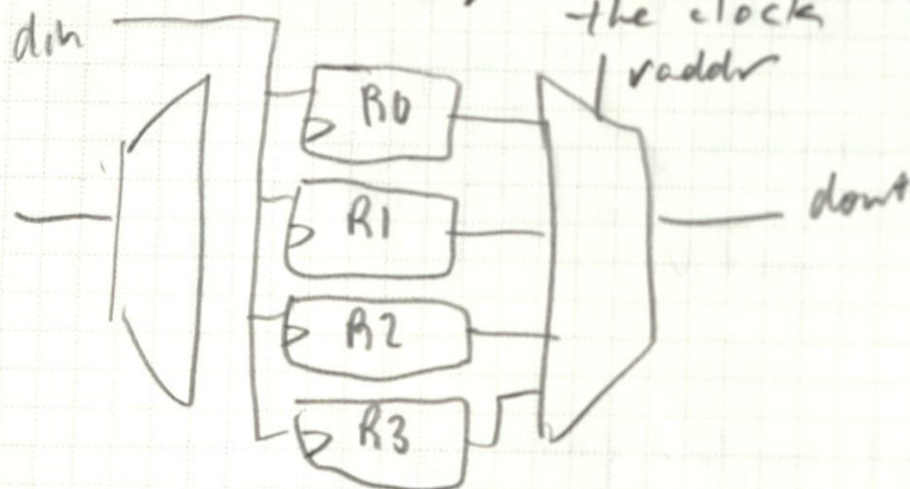


not ideal for a register file: use load regs
used in linear feedback shift registers

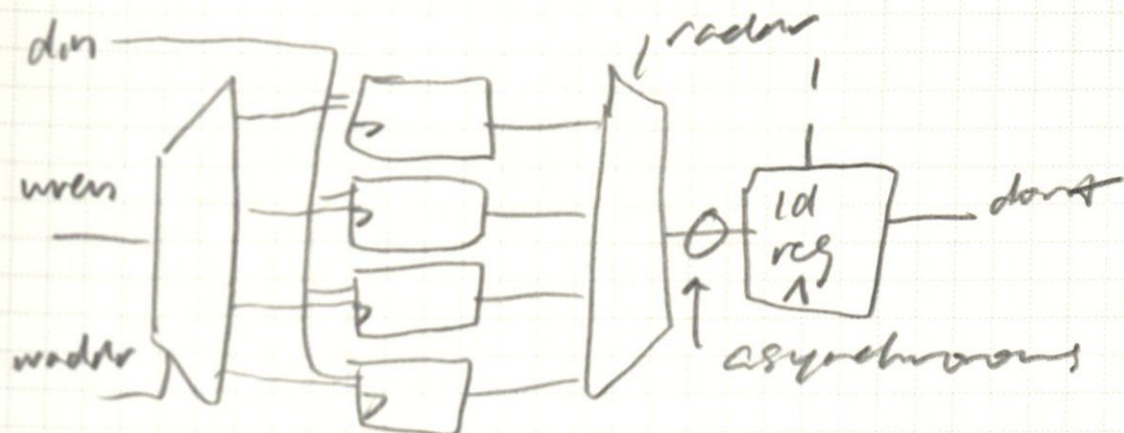
Synchronous / Asynchronous:

$\hookrightarrow$ with the clock

$\hookrightarrow$ without respect to the clock



Synchronous write
asynchronous read



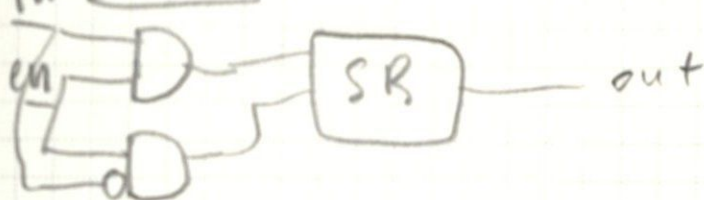
synchronous write, synchronous read
(exam 2 question)

Register from Basic Gates:

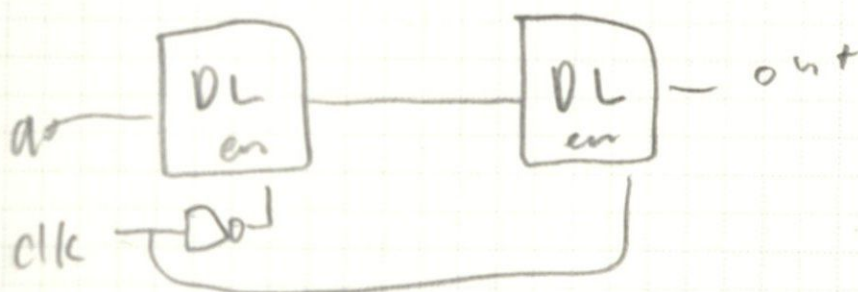
SR-Latch. 2 NOR gates



D Latch. SR + 2 AND + inverter



D Flip-Flop. 2 D-latches + inverter



FF's for sequential circuit: $\log(\# \text{ states})$
to represent 2 states we need 3 bits (3 FF's)