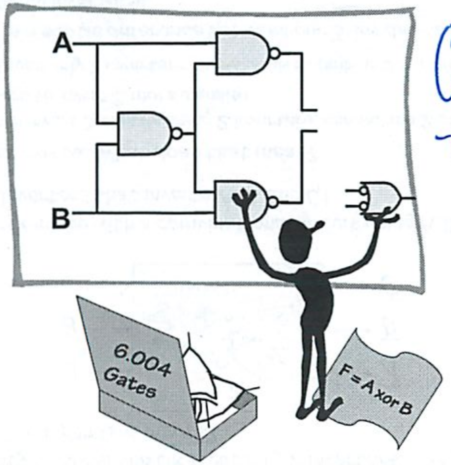
Synthesis of Combinational Logic

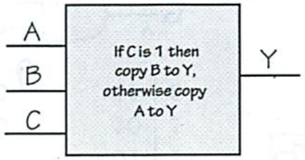


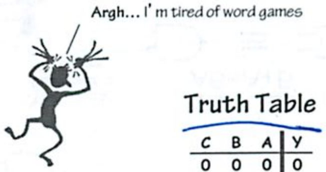
Quiz Fri
-up to last weeks materials

Lab 1 is due Thursday 9/22
Quiz 1 is Friday 9/23 (in section)

Functional Specifications

There are many ways of specifying the function of a combinational device, for example:





Words

Concise alternatives:

truth tables are a concise description of the combinational system's function.

Boolean expressions form an algebra in whose operations are **AND** (multiplication), **OR** (addition), and **inversion** (overbar).

Any combinational (Boolean) function can be specified as a truth table or an equivalent sum-of-products Boolean expression!

Truth Table

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

$Y=\overline{C}\,\overline{B}A+\overline{C}BA+CB\overline{A}+CBA$

Can simplify it

Here's a Design Approach

a fool proof way to build boolean expression

Truth Table

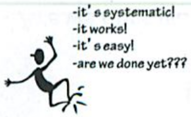
C	B	A	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

- 1) Write out our functional spec as a truth table
- 2) Write down a Boolean expression with terms covering each '1' in the output:

$$Y=\overline{C}\,\overline{B}A+\overline{C}BA+CB\overline{A}+CBA$$

just look at rows

- 3) Wire up the gates, call it a day, and declare success!
- This approach will always give us Boolean expressions in a particular form: SUM-OF-PRODUCTS

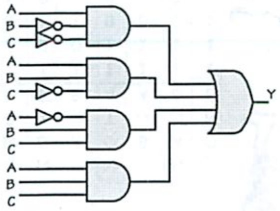


Straightforward Synthesis

↓ foolproof way to implement

We can implement SUM-OF-PRODUCTS with just three levels of logic.

INVERTERS/AND/OR



Propagation delay --
No more than 3 gate delays
(assuming gates with an arbitrary number of inputs)

▷o for items w/ ‾ over

Worst case for any boolean expression

9/20

Basic Gate Repertoire

Are we sure we have all the gates we need?

Just how many two-input gates are there?

How were these 4 chosen?

AND

AB	Y
00	0
01	0
10	0
11	1

OR

AB	Y
00	0
01	1
10	1
11	1

NAND

AB	Y
00	1
01	1
10	1
11	0

NOR

AB	Y
00	1
01	0
10	0
11	0

cheaper

Hmmmm… all of these have 2-inputs (no surprise)

… each with 4 combinations, giving 2^2 output cases

How many ways are there of assigning 4 outputs? $2^{2^2} = 2^4 = 16$

4 from each column

16 two value functions

There are only so many gates

There are only 16 possible 2-input gates

… some we know already, others are just silly

I N P U T AB	Z E R O	A N D	A > B	A	B > A	B	X O R	O R	N O R	X N O R	N O T 'B'	A <= B	N O T 'A'	B <= A	N A N D	O N E
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

How many of these gates can be implemented using a single CMOS gate?

Circled

CMOS gates are inverting; we can always respond positively to positive transitions by cascaded gates. But suppose our logic yielded cheap positive functions, while inverters were expensive…

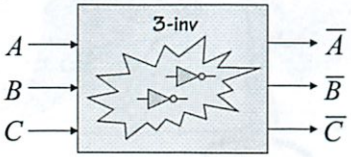
Anything where in 1->0 can not go 1->0

AND 0->1 0->1

So output transitions only opposet input

Logic Geek Party Games

You have plenty of ANDs and ORs, but only 2 inverters. Can you invert more than 2 independent inputs?



You can do it

CHALLENGE: Come up with a combinational circuit using ANDs, ORs, and at most 2 inverters that inverts A, B, and C!

Such a circuit exists. What does that mean?

- If we can invert 3 signals using 2 inverters, can we use 2 of the pseudo-inverters to invert 3 more signals?
- Do we need only 2 inverters to make ANY combinational circuit? No

Hint: there's a subtle difference between our 3-inv device and three combinational inverters!

Is our 3-inv device LENIENT? No

On output feeds back in as an input

Fortunately, we can get by with a few basic gates…

AND, OR, and NOT are sufficient… (cf Boolean Expressions):

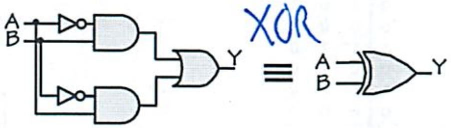
$\overline{A}B$

B>A

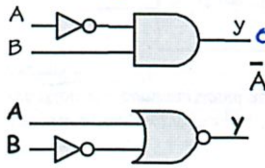
AB	Y
00	0
01	1
10	0
11	0

XOR

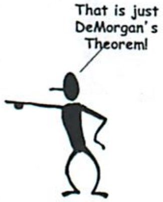
AB	Y
00	0
01	1
10	1
11	0



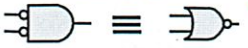
XOR



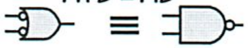
$\overline{A}B=\overline{A+\overline{B}}$



$\overline{A}\,\overline{B}=\overline{A+B}$



$\overline{A}+\overline{B}=\overline{AB}$



note matters where put bar

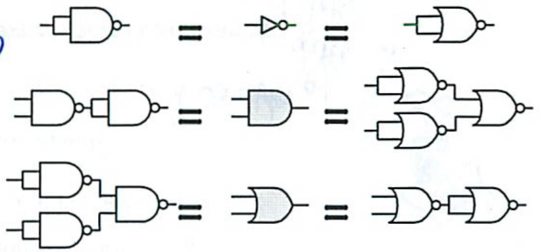
How many different gates do we really need?

better to do w/ NAND gate

One will do!

NANDs and NORs are universal: ← implement anything using just these two

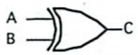
nothing but NAND gates



Ah!, but what if we want more than 2-inputs?

Stupid Gate Tricks

Suppose we have some 2-input XOR gates:



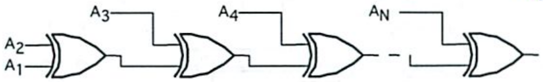
told

$t_{pd} = 1$

$t_{cd} = 0$

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

And we want an N-input XOR: Can cascade



output = 1
iff number of 1s
input is ODD
("ODD PARITY")

t_{pd} = O(N) -- WORST CASE.

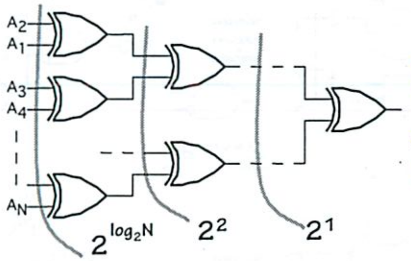
Can we compute N-input XOR faster?

I think that I shall never see

a circuit lovely as...

tree

faster



increase depth =1

2x inputs so inputs ↑ exponentially

N-input TREE has O(log N) levels...

Signal propagation takes O(log N) gate delays.

Question: Can EVERY N-Input Boolean function be implemented as a tree of 2-input gates? not all

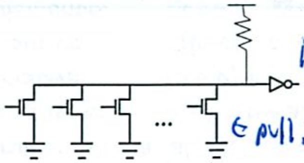
Other strategies

Are Trees Always Best?

No

Alternate Plan: Large Fan-in gates

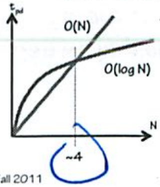
- N pulldowns with complementary pullups
- Output HIGH if any input is HIGH = "OR"



logical 1 default

← pull down

- Propagation delay: O(N) since each additional MOSFET adds C



Don't be mislead by the "big O" stuff... the constants in this case can be much smaller... so for small N this plan might be the best.

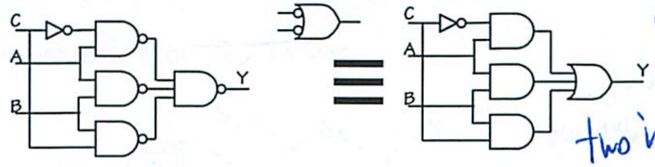
Practical SOP Implementation

how can put together & optimize
-will help at end of year in design project

NAND-NAND

$\overline{AB}=\overline{A}+\overline{B}$

"Pushing Bubbles"

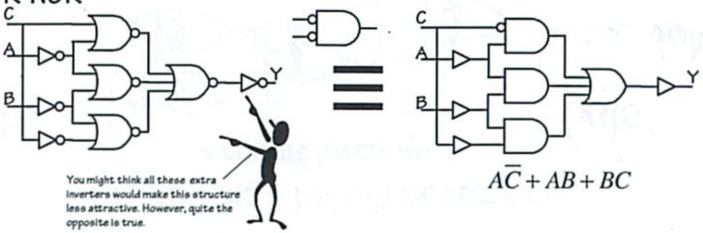


two inverters cancel

$A\overline{C}+AB+BC$

NOR-NOR

$\overline{A}\,\overline{B}=\overline{A+B}$



$A\overline{C}+AB+BC$

You might think all these extra inverters would make this structure less attractive. However, quite the opposite is true.

less work

Logic Simplification

know rules

Can we implement the same function with fewer gates?
Before trying we'll add a few more tricks in our bag.

BOOLEAN ALGEBRA:

OR rules:	$a+1=1,\ a+0=a,\ a+a=a$
AND rules:	$a1=a,\ a0=0,\ aa=a$
Commutative:	$a+b=b+a,\ ab=ba$
Associative:	$(a+b)+c=a+(b+c),\ (ab)c=a(bc)$
Distributive:	$a(b+c)=ab+ac,\ a+bc=(a+b)(a+c)$
Complements:	$a+\overline{a}=1,\ \ a\overline{a}=0$
Absorption:	$a+ab=a,\ \ a+\overline{a}b=a+b$ $a(a+b)=a,\ \ a(\overline{a}+b)=ab$
Reduction:	$\boxed{ab+\overline{a}b=b,}\ (a+b)(\overline{a}+b)=b$
DeMorgan's Law:	$\overline{a}+\overline{b}=\overline{ab},\ \ \overline{a}\,\overline{b}=\overline{a+b}$

Boolean Minimization:

An Algebraic Approach

Lets (again!) simplify

$$Y=\overline{C}\overline{B}A+CB\overline{A}+CBA+\overline{C}BA$$

Can't he come up with a *new* example???

Using the identity

$$\alpha A+\alpha\overline{A}=\alpha$$

if variables only differ in 1 letter, can combine

For any expression α and variable A:

$$Y=\overline{C}\overline{B}A+CB\overline{A}+CBA+\overline{C}BA$$

$$Y=\overline{C}\overline{B}A+CB+\overline{C}BA$$

$$Y=\overline{C}A+CB$$

Hey, I could write A program to do That!

lots of algorithms use

Can simplify

Simplify! or maybe not

A Case for Non-Minimal SOP

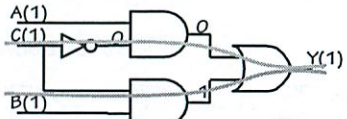
not just pure basic math! Correct implementation

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

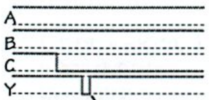
$\overline{C}A$

CB

BA



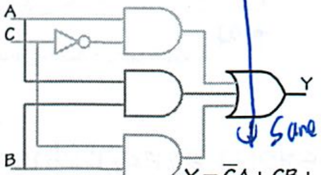
$Y=\overline{C}A+CB$ $t_{CD}=1\text{ nS}$ $t_{PD}=2\text{nS}$



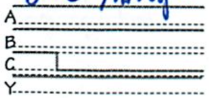
That's what we call a "glitch" or "hazard"

NOTE: The steady state behavior of these circuits is identical. They differ in their transient behavior.

propagation delay
Changing # of gates gone through



Same



$Y=\overline{C}A+CB+AB$

Now it's LENIENT!

- hazzard Free implictation

Can have a bit more leneant by adding more hardware

What if hardware fixed?

Truth Tables with "Don't Cares"

One way to reveal the opportunities for a more compact implementation is to rewrite the truth table using "don't cares" (--) to indicate when the value of a particular input is irrelevant in determining the value of the output.

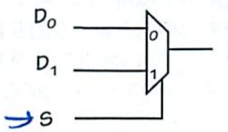
C	B	A	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

↓ repeat

C	B	A	Y	
0	--	0	0	
0	--	1	1	→ $\bar{C}A$
1	0	--	0	
1	1	--	1	→ CB
--	0	0	0	
--	1	1	1	→ BA

Irrelevant
- can be 0 or 1
or otherwise

We've been designing a "mux"



select line

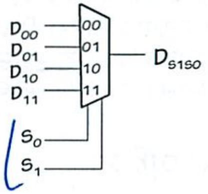
2-input Multiplexer

Truth Table

S	D_1	D_0	D_S
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

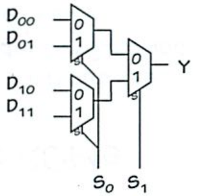
Complicated inside
- See study on 9/25

MUXes can be generalized to 2^k data inputs and k select inputs ...



bigger

... and implemented as a tree of smaller MUXes:



Or make tree of multiple

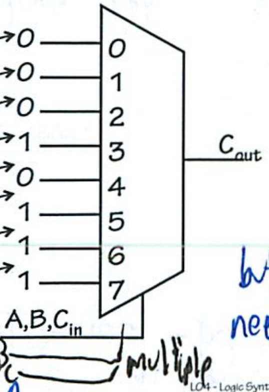
Systematic Implementations of Combinational Logic

Consider implementation of some arbitrary Boolean function, F(A,B,C) ... using a MULTIPLEXER as the only circuit element:

Wire in as constants

Full-Adder Carry Out Logic

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

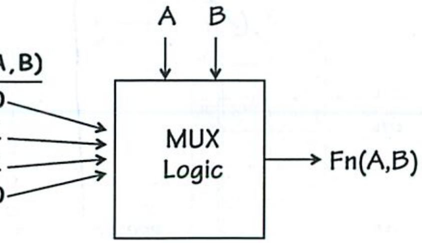


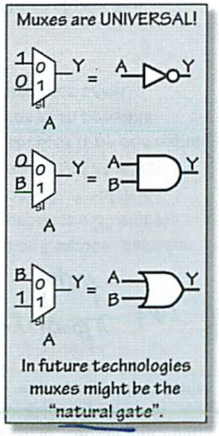
but for n inputs need 2^n

implemented table lookup

General Table Lookup Synthesis

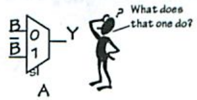
AB	Fn(A,B)
00	0
01	1
10	1
11	0





In future technologies muxes might be the "natural gate".

but are made up of NAND/NOR



Generalizing:
In theory, we can build any 1-output combinational logic block with multiplexers.

For an N-input function we need a 2^N input mux.

One for each row

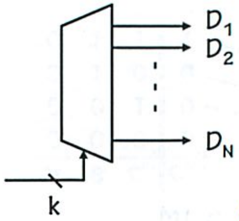
Is this practical for BIG truth tables?
How about 10-input function? 20-input?

∴ so that is why we don't use it more?

not inverse of multiplexer

A New Combinational Device

– but looks like it



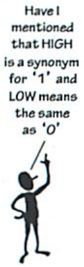
DECODER:

k SELECT inputs,

N = 2^k DATA OUTPUTs.

Selected D_i HIGH; all others LOW.

One of them driven high


Have I mentioned that HIGH is a synonym for '1' and LOW means the same as '0'

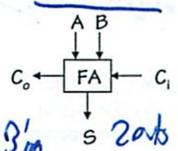
NOW, we are well on our way to building a general purpose table-lookup device.

We can build a 2-dimensional ARRAY of decoders and selectors as follows ...

Secret sause behind ROM

Read-only memories (ROMs)

Full Adder



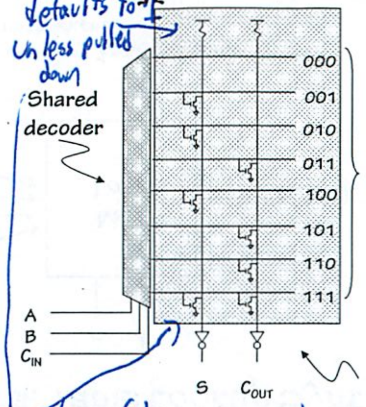
3 in 2 outs

A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Each column is large fan-in "OR" as described on slide #12. Note location of pulldowns correspond to a "1" output in the truth table!

defaults to 1 unless pulled down by nfet

Shared decoder



Systematic way to input

For K inputs, decoder produces 2^K signals, only 1 of which is asserted at a time -- think of it as one signal for each possible product term.

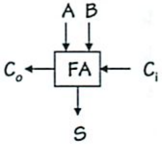
One column for each output

OR From nfets

lines can get long – but then capacitance

Read-only memories (ROMs)

Full Adder

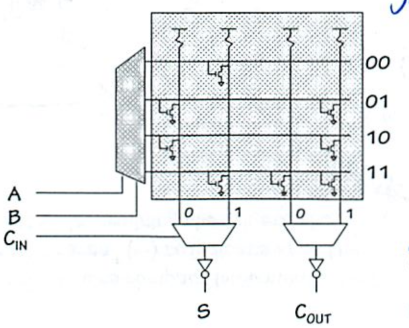


A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

LONG LINES slow down propagation times…

The best way to improve this is to build *square arrays*, using some inputs to drive output selectors (MUXes):

Shorter lines better



Same as previous slide

2D Addressing: Standard for ROMs, RAMs, logic arrays…

Logic According to ROMs

ROMs *ignore* the structure of combinational functions ...

- Size, layout, and design are independent of function
- Any Truth table can be "programmed" by minor reconfiguration:

  - Metal layer (masked ROMs)
  - Fuses (Field-programmable PROMs)
  - Charge on floating gates (EPROMs)
  ... etc.

ROMs tend to generate "glitchy" outputs. WHY?

not linear

Model: LOOK UP value of function in truth table...

Inputs: "ADDRESS" of a T.T. entry

ROM SIZE = # TT entries...

... for an N-input boolean function, size ≅ 2^N x #outputs

size close to proportional to # outputs

¿So how does that store? ¿So how does that help us store stuff?

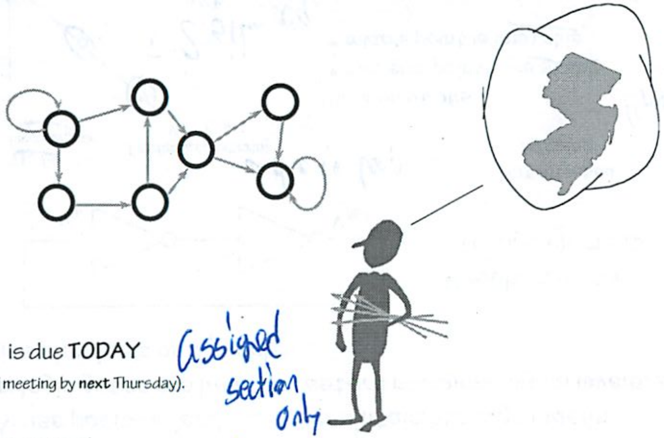
Summary

- Sum of products
  - Any function that can be specified by a truth table or, equivalently, in terms of AND/OR/NOT (Boolean expression)
  - "3-level" implementation of any logic function
    - Limitations on number of inputs (fan-in) increases depth
  - SOP implementation methods
    - NAND-NAND, NOR-NOR
- Muxes used to build table-lookup implementations
  - Easy to change implemented function -- just change constants
- ROMs
  - Decoder logic generates all possible product terms
  - Selector logic determines which p' terms are or' ed together

Sequential Logic:

adding a little state

Sequential Circuits



Lab #1 is due TODAY Assigned section only

(checkoff meeting by **next** Thursday).

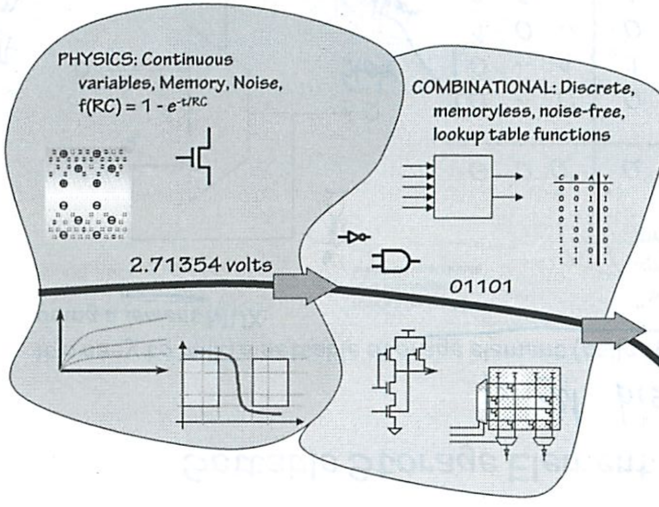
QUIZ #1 Friday! tomorrow!

(covers thru L3/R3)

Review Session 7:30pm 32-144

Before: no state. Output only dependent on input

6.004: Progress so far...



PHYSICS: Continuous variables, Memory, Noise, $f(RC) = 1 - e^{-t/RC}$

2.71354 volts

COMBINATIONAL: Discrete, memoryless, noise-free, lookup table functions

01101

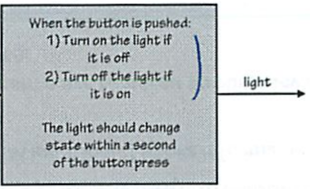
What other building blocks do we need in order to compute?

Something We Can't Build (Yet)

State machine

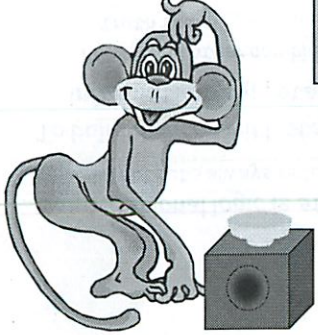
- depends on history of inputs

What if you were given the following design specification:



When the button is pushed:
1) Turn on the light if it is off
2) Turn off the light if it is on

The light should change state within a second of the button press



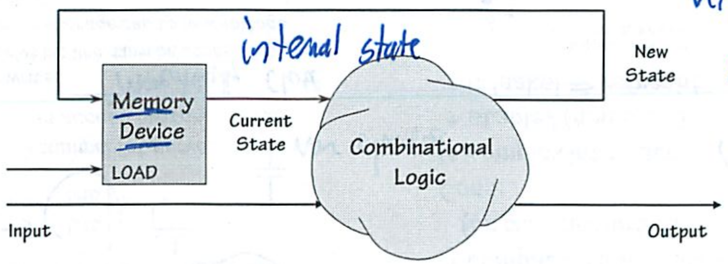
What makes this circuit so different from those we've discussed before?

1. "State" – i.e. the circuit has memory
2. The output was changed by a input "event" (pushing a button) rather than an input "value"

Digital State

One model of what we'd like to build

Very workable model



Plan: Build a Sequential Circuit with stored digital STATE –

- Memory stores CURRENT state, produced at output
- Combinational Logic computes
  - NEXT state (from input, current state)
  - OUTPUT bit (from input, current state)
- State changes on LOAD control input

Needed: Storage

Combinational logic is *stateless*:
valid outputs always reflect current inputs.

To build devices with state, we need components which *store* information (e.g., state) for subsequent access.

ROMs (and other combinational logic) store information "wired in" to their truth table

Read/Write memory elements are required to build devices capable of changing their contents.

Use physical penomnon to store bits

How can we store – and subsequently access -- a bit?

- Mechanics: holes in cards/tapes
- Optics: Film, CDs, DVDs, …
- Magnetic materials
- Delay lines; moonbounce
- Stored charge

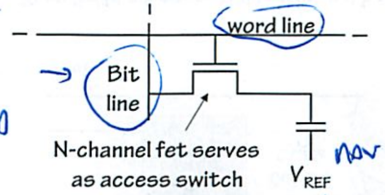
c tanks of mecury 1940s ←

Storage: Using Capacitors

DRAM - good for large storage

We've chosen to encode information using voltages and we know from 6.002 that we can "store" a voltage as charge on a capacitor:

Store



apply 0 or V_{DD} →

now charged

To write: — Momentarilly close
Drive bit line, turn on access fet,
force storage cap to new voltage

To read:
precharge bit line, turn on access fet,
detect (small) change in bit line voltage

read
put bit line ~ half
close fet
see if bit line goes up + down
need careful amplifier

Pros:
- compact – low cost/bit (on BIG memories)

Cons:
- complex interface
- stable? (noise, …)
- it leaks! ⇒ refresh

read and rewrite

Suppose we refresh CONTINUOUSLY?

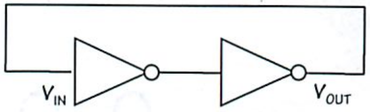
simpler tech

Storage: Using Feedback

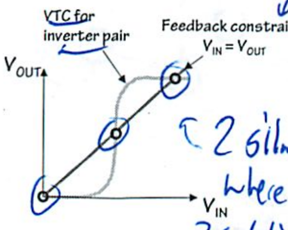
bi-stable - 2

IDEA: use positive feedback to maintain storage indefinitely. Our logic gates are built to restore marginal signal levels, so noise shouldn't be a problem!

stable states 0 or 1



Result: a bistable storage element



due to loop

2 silm eq where intersect - 3 solutions

Three solutions:
- two end-points are stable
- middle point is unstable

Not affected by noise

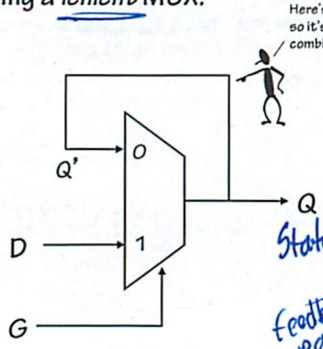
We'll get back to this!
1 entire lecture

"metastable state"

Settable Storage Element

to get bistable behavior

It's easy to build a settable storage element (called a latch) using a *lenient* MUX:



Here's a feedback path, so it's no longer a combinational circuit.

State

feedback path

for diff reason (input) → G

relable terminals

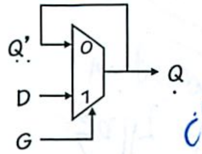
"state" signal appears as both input and output

G	D	Q'	Q	
0	--	0	0	Q stable } stores Q
0	--	1	1	
1	0	--	0	Q follows D
1	1	--	1	

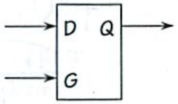
not ↑ combo circuit

TD - Latch

New Device: D Latch

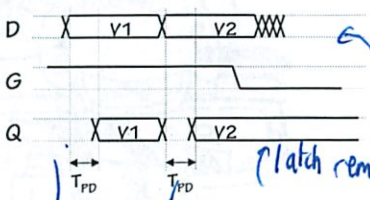


circuit diagram



G=1: Q follows D

G=0: Q holds



← no matter what D

latch remains

G=1 so follows

BUT… A change in D or G contaminates Q, hence Q' … how can this possibly work?

G=1: Q Follows D, *independently of Q'*

G=0: Q Holds stable Q', *independently of D*

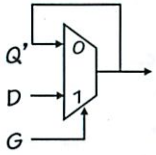
How can we guarentee?

A Plea for Lenience…

✓ much of the time don't need – but critical here

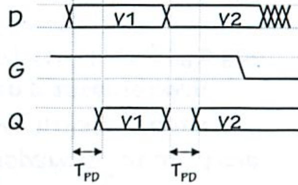
└ means non-strict

any combo of inpt on TT are held stable, then item remains stable



G	D	Q'	Q
1	0	X	0
1	1	X	1
X	0	0	0
X	1	1	1
0	X	0	0
0	X	1	1

↗ X is don't care

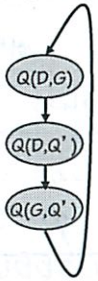


Assume LENIENT Mux, propagation delay of T_{PD}

Then output valid when

- G=1, D stable for T_{PD}, *independently of Q'; or*
- Q' =D stable for T_{PD}, *independently of G; or*
- G=0, Q' stable for T_{PD}, *independently of D*

need all of them to get it to work



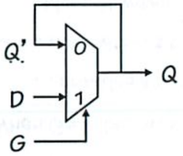
Does lenience *guarantee* a working latch?

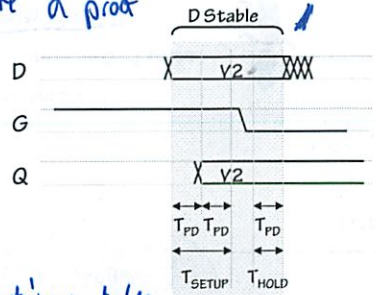
What if D and G change at about the same time…

Timing Diagram

… with a little discipline

– like a proof





To reliably latch V2:

- Apply V2 to D, holding G=1
- After T_{PD}, V2 appears at Q=Q' ↑ nothing matches
- After another T_{PD}, Q' & D both valid for T_{PD}; *will hold Q=V2 independently of G* stable valid
- Set G=0, while Q' & D hold Q=D
- After another T_{PD}, G=0 and Q' are sufficient to hold Q=V2 *independently of D*

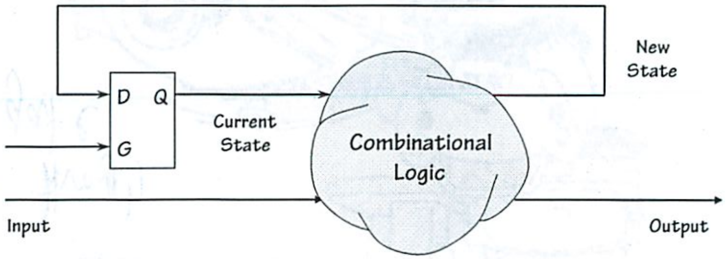
now ind. →

need 2nd tpd

↑ does not depend on D

> *Dynamic Discipline* for our latch:
>
> $T_{SETUP} = 2T_{PD}$: interval *prior to* G transition for which D must be stable & valid
>
> $T_{HOLD} = T_{PD}$: interval *following* G transition for which D must be stable & valid

Lets try it out!



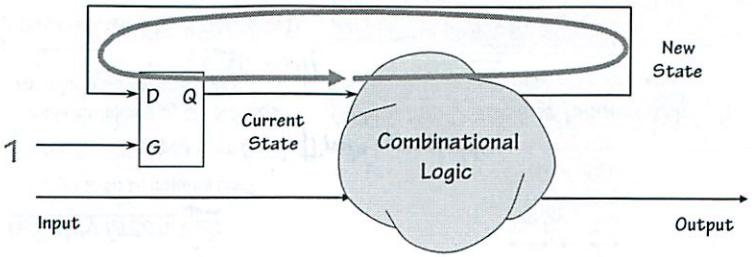
Plan: Build a Sequential Circuit with one bit of STATE –

- Single latch holds CURRENT state
- Combinational Logic computes
  - NEXT state (from input, current state)
  - OUTPUT bit (from input, current state)
- State changes when G = 1 (briefly!)

What happens when G=1?

Transparent = open

Combinational Cycles



When G=1, latch is *Transparent*…

… provides a combinational path from D to Q.

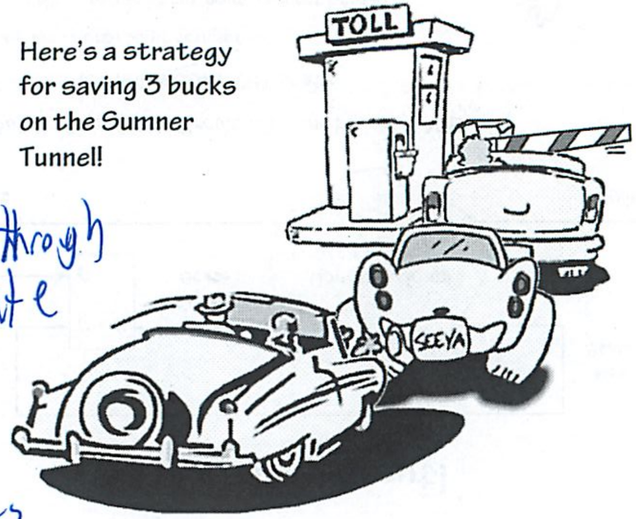
Can't work without tricky timing constrants on G=1 pulse:

- Must fit within contamination delay of logic
- Must accommodate latch setup, hold times

Want to signal an INSTANT, not an INTERVAL…

Flakey Control Systems

Here's a strategy for saving 3 bucks on the Sumner Tunnel!

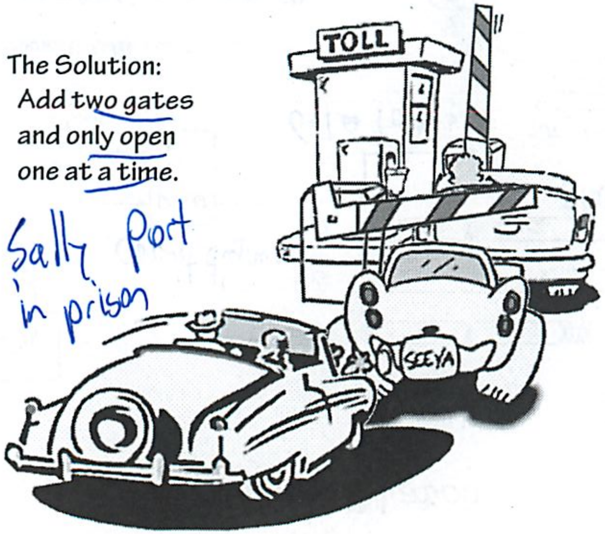


gun it through toll gate

When opens, for previous

Escapement Strategy

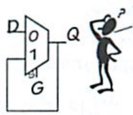
The Solution:
Add two gates
and only open
one at a time.

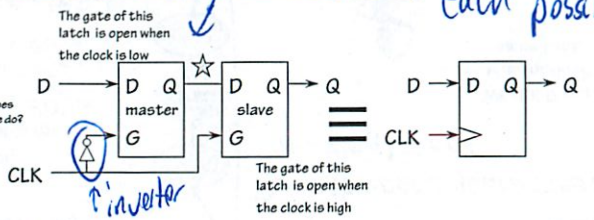


Sally Port in prison

Edge-triggered Flip Flop

Arrange opposite valued gates
Only 1 open for each possible path




The gate of this latch is open when the clock is low
D
D Q
master
G
☆
D Q
slave
G
Q
CLK
inverter
The gate of this latch is open when the clock is high
≡
D
D Q
Q
CLK

Observations:

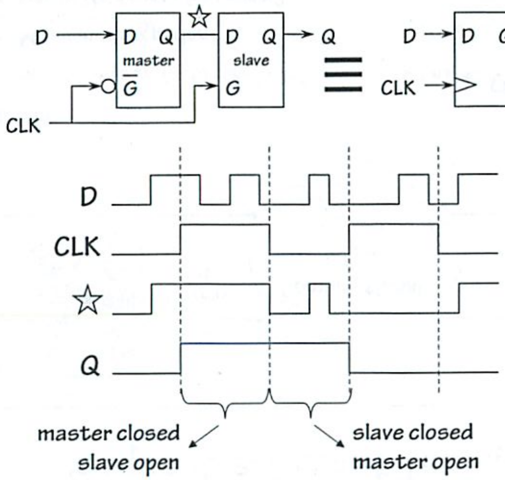
- only one latch "transparent" at any time:
  - master closed when slave is open
  - slave closed when master is open

  ⇒ no combinational path through flip flop

  (the feedback path in one of the master or slave latches is always active)
- Q only changes shortly after 0 →1 transition of CLK, so flip flop appears to be "triggered" by rising edge of CLK

Flip Flop Waveforms

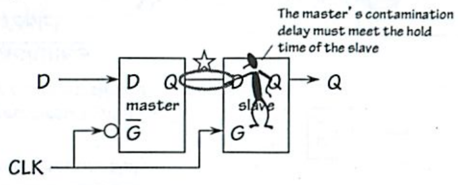
details of timing diygram



prop delays not really shown

dynamic displin - obey set up + hold time requirements

Um, about that hold time…



The master's contamination delay must meet the hold time of the slave

Consider HOLD TIME requirement for slave:

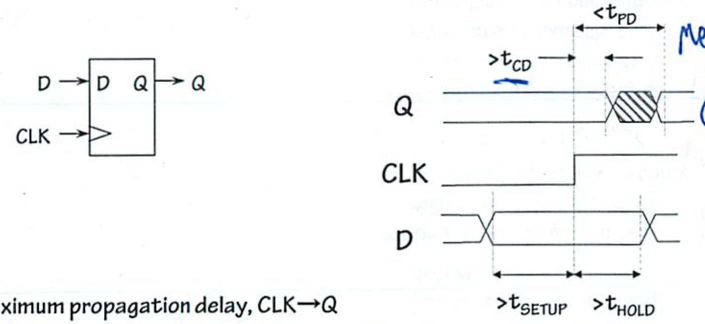
- Negative (1 →0) clock transition ⇒ slave freezes data:
  - SHOULD be no output glitch, since master held constant data; BUT
  - master output contaminated by change in G input!
- HOLD TIME of slave not met, UNLESS we assume sufficient contamination delay in the path to its D input!

Accumulated t_{CD} thru inverter, $G \rightarrow Q$ path of master must cover slave t_{HOLD} for this design to work!

need long enough CD not 0 ← not many of those

external view

Flip Flop Timing - I



measure from active clock edges

t_{PD}: maximum propagation delay, CLK→Q

t_{CD}: minimum contamination delay, CLK→Q usually assume 0 here

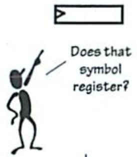
t_{SETUP}: setup time
guarantee that D has propagated through feedback path before master closes

extends after clock edge

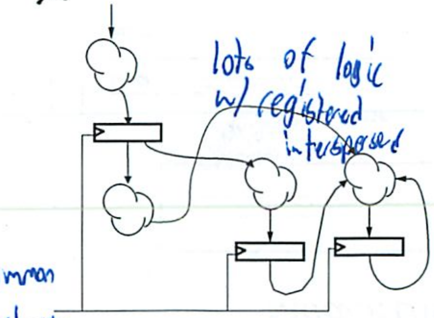
t_{HOLD}: hold time
guarantee master is closed and data is stable before allowing D to change

Single-clock Synchronous Circuits

Clean, small reliable



We'll use Flip Flops and *Registers* – groups of FFs sharing a clock input – in a highly constrained way to build digital systems:



lots of logic w/ registered interspersed

Common clock wire

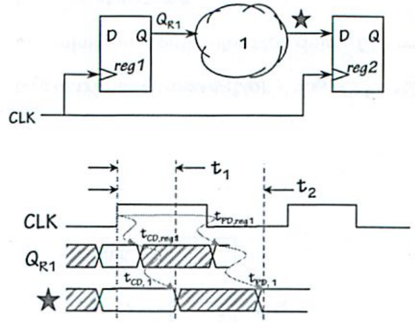
Single-clock Synchronous Discipline

- No combinational cycles must go through register
- Single periodic clock signal shared among all clocked devices
- Only care about value of register data inputs just before rising edge of clock
- Period greater than every combinational delay + setup time
- Change saved state after noise-inducing logic transitions have stopped!

make clock long enough so > every Σ tcd through each possible path

inputs from 1 registe
outputs to other registe

Flip Flop Timing - II



$t_1 = t_{CD,reg1} + t_{CD,1} > t_{HOLD,reg2}$

$t_2 = t_{PD,reg1} + t_{PD,1} < t_{CLK} - t_{SETUP,reg2}$

-hold time req
-tcd requirement

Questions for register-based designs:

- how much time for useful work (i.e. for combinational logic delay)?
- does it help to guarantee a minimum t_{CD}? How about designing registers so that $$t_{CD,reg} > t_{HOLD,reg}?$$
- what happens if CLK signal doesn't arrive at the two registers at exactly the same time (a phenomenon known as "clock skew")?

tcd must be long enough to cover register tinp

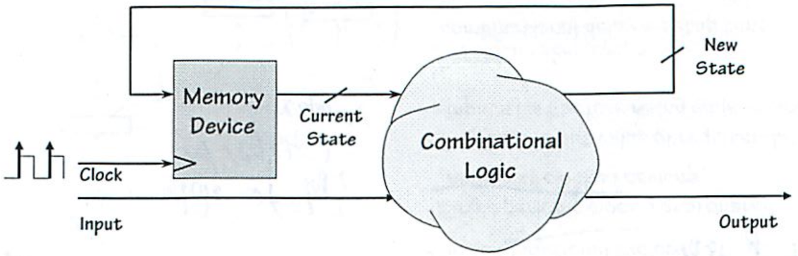
← need to get clocks on time

and setup time

then guarentee dynamic displine

move to

Model: Discrete Time w/ clock periods



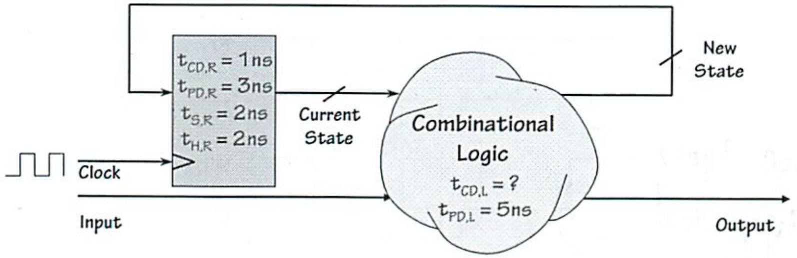
Active Clock Edges punctuate time ---

- Discrete Clock periods
- Discrete State Variables
- Discrete specifications (simple rules – eg tables – relating outputs to inputs, state variables)

- ABSTRACTION: Finite State Machines (next lecture!)

Can simplify a lot

Sequential Circuit Timing



Questions:

- Constraints on T_{CD} for the logic? > 1 ns ← for hold time
- Minimum clock period? > 10 ns ($T_{PD,R}+T_{PD,L}+T_{S,R}$) how fast can actually run
- Setup, Hold times for Inputs? $T_S = T_{PD,L} + T_{S,R}$
  $T_H = T_{H,R} - T_{CD,L}$

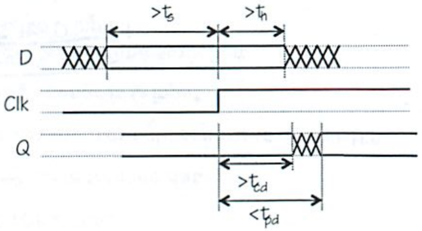
This is a simple *Finite State Machine* … more next lecture!!

Summary

"Sequential" Circuits (with memory):

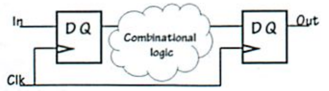
Basic memory elements:

- Feedback, detailed analysis => basic level-sensitive devices (eg, latch)
- 2 Latches => Flop
- Dynamic Discipline: constraints on input timing



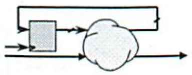
Synchronous 1-clock logic:

- Simple rules for sequential circuits
- Yields clocked circuit with T_S, T_H constraints on input timing



Finite State Machines

Next Lecture Topic!



Hint, lst problem "just say no"

Two's complement

Read 9/25

From Wikipedia, the free encyclopedia

The **two's complement** of a binary number is defined as the value obtained by subtracting the number from a large power of two (specifically, from 2^N for an N-bit two's complement). The two's complement of the number then behaves like the negative of the original number in most arithmetic, and it can coexist with positive numbers in a natural way.

Two's Complement is referred to as **Binary Number Representation** (or BNR) in protocols used in Aviation (ARINC_429).

A **two's-complement system**, or **two's-complement arithmetic**, is a system in which negative numbers are represented by the two's complement of the absolute value;[1] this system is the most common method of representing signed integers on computers.[2] In such a system, a number is negated (converted from positive to negative or vice versa) by computing its two's complement. An N-bit two's-complement numeral system can represent every integer in the range -2^{N-1} to 2^{N-1}-1.

The two's-complement system has the advantage of not requiring that the addition and subtraction circuitry examine the signs of the operands to determine whether to add or subtract. This property makes the system both simpler to implement and capable of easily handling higher precision arithmetic. Also, zero has only a single representation, obviating the subtleties associated with negative zero, which exists in ones'-complement systems.

The method of complements can also be applied in base-10 arithmetic, using ten's complements by analogy with two's complements.

easy to do in CMOS + various logic gates

Contents

Most-significant bit

0 1 1 1 1 1 1 1	=	127
0 1 1 1 1 1 1 0	=	126
0 0 0 0 0 0 1 0	=	2
0 0 0 0 0 0 0 1	=	1
0 0 0 0 0 0 0 0	=	0
1 1 1 1 1 1 1 1	=	−1
1 1 1 1 1 1 1 0	=	−2
1 0 0 0 0 0 0 1	=	−127
1 0 0 0 0 0 0 0	=	−128

8-bit two's-complement integers

Explanation

Two's-complement numbers

Two's complement numbers is a way to encode negative numbers into ordinary binary, such that addition still works. Adding −1 + 1 should equal 0, but ordinary addition gives the result of 2 or −2 unless the operation takes special notice of the sign bit and performs a subtraction instead. Two's complement results in the correct sum without this extra step.

A two's-complement number system encodes positive and negative numbers in a binary number representation. The bits have a binary radix point and the bits are weighted according to the position of the bit within the array. A convenient notation is the big-endian ordering. In this notation, the bit to the left of the binary point has a bit index of 0 and a weight of 2^0. The bit indices increase, by one, to the left of the binary point, and decrease, by one, to the right of the binary point. The weight of each bit is 2^i, except for the left-most bit, whose weight is -2^i. With this bit

$-2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

numbering, a two's complement number with *m* integer bits and *n* fractional bits is represented by the array of bits

$$v = a_{m-1}, a_{m-2}, \ldots, a_1, a_0.a_{-1}, a_{-2}, \ldots a_{-n}.$$

? are we doing fractional bits ?

The value of this number is given by the following formula.

$$-a_{m-1} \times 2^{m-1} + \sum_{i=-n}^{m-2} a_i \times 2^i$$

The left-most bit, also called the most-significant bit (MSB), determines the sign of the number, but, unlike the sign-and-magnitude representation, also has a weight, -2^{m-1}, as shown in the formula above. Because of this weight, it is misleading to call this bit the "sign bit".

The two's complement encoding shown above can represent the following range of numbers

What is this?

Why?

Zero representation is

$$0 : 0, 0, \ldots, 0$$

The maximum positive number is

$$2^{m-1} - 2^{-n} : 0, 1, 1, 1, \ldots, 1, 1$$

The minimum, non-zero, positive number (smallest absolute value) is

$$2^{-n} : 0, 0, 0, \ldots, 0, 0, 1$$

The minimum negative number is

$$-2^{m-1} : 1, 0, 0, 0, \ldots, 0, 0$$

The maximum negative number (smallest absolute value) is

$$-2^{-n} : 1, 1, 1, \ldots, 1, 1$$

Making the Two's Complement of a number

Positive numbers are represented in two's complement as binary numbers whose most significant bit is zero. Negative numbers are represented with the most-significant bit being one, making use of the left-most bit's negative weight. All radix complement number systems use a fixed-width encoding. Every number encoded in such a system has a fixed width so the most-significant digit can be examined.

Algorithmically, to create a two's complement binary value:

1. express the binary value for the positive number
2. if the original value was negative,
   2a. complement the value
   2b. add one

3a. if the value is positive, add leading zeros to achieve the proper number of bits
3b. if the value is negative, add leading ones to achieve the proper number of bits
(3. replicate the MSB to achieve the proper number of bits)

In general, for a radix r's complement encoding, with *r* the base (radix) of the number system, an integer part of *m* digits and fractional part of *n* digits, then the r's complement of a number $0 \leq N < r^{m-1} - r^{-n}$ is determined by the formula:

$$N^{**} = (r^m - N) \bmod (r^m)$$

The (r−1)'s complement of a number is determined by the formula:

$$N^{*} = r^m - r^{-n} - N$$

We can also find the r's complement of a number *N* by adding r^{-n} to the (r-1)'s complement of the number *i.e.*,

$$N^{**} = N^{*} + r^{-n}$$

Alternative conversion process

A shortcut to manually convert a binary number into its two's complement is to start at the least significant bit (LSB), and copy all the zeros (working from LSB toward the most significant bit) until the first 1 is reached; then copy that 1, and flip all the remaining bits. This shortcut allows a person to convert a number to its two's complement without first forming its ones' complement. For example: the two's complement of "0011 1100" is "1100 0100", where the underlined digits were unchanged by the copying operation (while the rest of the digits were flipped).

In computer circuitry, this method is no faster than the "complement and add one" method; both methods require working sequentially from right to left, propagating logic changes. The method of complementing and adding one can be sped up by a standard carry look-ahead adder circuit; the alternative method can be sped up by a similar logic transformation.

Sign extension

When turning a two's-complement number with a certain number of bits into one with more bits (e.g., when copying from a 1 byte variable to a two byte variable), the most-significant bit must be repeated in all the extra bits and lower bits.

Some processors have instructions to do this in a single instruction. On other processors a conditional must be used followed with code to set the relevant bits or bytes.

Decimal	4-bit notation	8-bit notation
5	0101	0000 0101
−5	1011	1111 1011

sign-bit repetition in 4 and 8-bit integers

Similarly, when a two's-complement number is shifted to the right, the most-significant bit, which contains magnitude and the sign information, must be maintained. However when shifted to the left, a 0 is shifted in. These rules preserve the common semantics that left shifts multiply the number by two and right shifts divide the number by two.

Both shifting and doubling the precision are important for some multiplication algorithms. Note that unlike addition and subtraction, precision extension and right shifting are done differently for signed vs unsigned numbers.

The most negative number

With only one exception, when we start with any number in two's-complement representation, if we flip all the bits and add 1, we get the two's-complement representation of the negative of that number. Negative 12 becomes positive 12, positive 5 becomes negative 5, zero becomes zero, etc.

The two's complement of the minimum number in the range will not have the desired effect of negating the number. For example, the two's complement of −128 in an 8-bit system results in the same binary number. This is because a positive value of 128 cannot be represented with an 8-bit signed binary numeral. Note that this is detected as an overflow condition since there was a carry into but not out of the most-significant bit. This can lead to unexpected bugs in that a naive implementation of absolute value could return a negative number.

−128	1000 0000
invert bits	0111 1111
add one	1000 0000

The two's complement of -128 results in the same 8-bit binary number.

The most negative number in two's complement is sometimes called "the weird number," because it is the only exception.[3][4]

Although the number is an exception, it is a valid number in regular two's complement systems. All arithmetic operations work with it both as an operand and (unless there was an overflow) a result.

Why it works

Given a set of all possible n-bit values, we can assign the lower (by binary value) half to be the integers from 0 to (2^{n-1}−1) inclusive and the upper half to be -2^{n-1} to −1 inclusive. The upper half can be used to represent negative integers from -2^{n-1} to −1 because, under addition modulo 2^n they behave the same way as those negative integers. That is to say that because i + j mod 2^n = i + (j − 2^n) mod 2^n any value in the set $\{j + k2^n \mid k \text{ is an integer}\}$ can be used in place of j.

For example, with eight bits, the unsigned bytes are 0 to 255. Subtracting 256 from the top half (128 to 255) yields the signed bytes −128 to 127.

The relationship to two's complement is realised by noting that 256 = 255 + 1, and (255 − x) is the ones' complement of x.

Example

−95 modulo 256 is equivalent to 161 since

−95 + 256
= −95 + 255 + 1
= 255 − 95 + 1
= 160 + 1
= 161

```
 1111 1111                       255
- 0101 1111                     -  95
===========                     =====
 1010 0000  (ones' complement)    160
+        1                      +   1
```

```
===========                     =====
 1010 0001  (two's complement)    161
```

Decimal	Two's complement
127	0111 1111
64	0100 0000
1	0000 0001
0	0000 0000
-1	1111 1111
-64	1100 0000
-127	1000 0001
-128	1000 0000

Some special numbers to note

Fundamentally, the system represents negative integers by counting backward and wrapping around. The boundary between positive and negative numbers is arbitrary, but the de facto rule is that all negative numbers have a left-most bit (most significant bit) of one. Therefore, the most positive 4-bit number is 0111 (7) and the most negative is 1000 (−8). Because of the use of the left-most bit as the sign bit, the absolute value of the most negative number (|−8| = 8) is too large to represent. For example, an 8-bit number can only represent every integer from −128 to 127 (2^(8−1) = 128) inclusive. Negating a two's complement number is simple: Invert all the bits and add one to the result. For example, negating 1111, we get 0000 + 1 = 1. Therefore, 1111 must represent −1.

The system is useful in simplifying the implementation of arithmetic on computer hardware. Adding 0011 (3) to 1111 (−1) at first seems to give the incorrect answer of 10010. However, the hardware can simply ignore the left-most bit to give the correct answer of 0010 (2). Overflow checks still must exist to catch operations such as summing 0100 and 0100.

The system therefore allows addition of negative operands without a subtraction circuit and a circuit that detects the sign of a number. Moreover, that addition circuit can also perform subtraction by taking the two's complement of a number (see below), which only requires an additional cycle or its own adder circuit. Lastly, the two's complement system allows a subtraction circuit to return 1001, equivalent to −0001, for 0001 − 0010 rather than 1111. To perform the former, the circuit merely pretends an extra left-most bit of 1 exists. To perform the latter, there must be a sign check, a possible rearrangement of the number, and finally a subtraction.

Two's complement	Decimal
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	−1
1110	−2
1101	−3
1100	−4
1011	−5
1010	−6
1001	−7
1000	−8

Two's complement using a 4-bit integer

So basically one means add 2^i

So 7
$2^2 + 2^1 + 2^0$
4 + 2 + 1

Oh easy!
first is −

Oh I see its not! also effects like −7
$-2^3 + 2^0$
−8 + 1

Clever

Calculating two's complement

In two's complement notation, a positive number is represented by its ordinary binary representation, using enough bits that the high bit (the sign bit) is 0. The two's complement operation is the negation operation, so negative numbers are represented by the two's complement of the representation of the absolute value.

In finding the two's complement of a binary number, the bits are inverted, or "flipped", by using the bitwise NOT operation; the value of 1 is then added to the resulting value. Bit overflow is ignored, which is the normal case with the zero value.

For example, beginning with the signed 8-bit binary representation of the decimal value 5, using subscripts to indicate the base of a representation needed to interpret its value:

$00000101_2 = 5_{10}$

Now how do you add 1

I guess we will learn about adders

The most significant bit is 0, so the pattern represents a non-negative (positive) value.

To convert to −5 in two's-complement notation, the bits are inverted; 0 becomes 1, and 1 becomes 0:

11111010

At this point, the numeral is the ones' complement of the decimal value 5. To obtain the two's complement, 1 is added to the result, giving:

$$11111011_2 = -5_{10}$$

The result is a signed binary number representing the decimal value −5 in two's-complement form. The most significant bit is 1, so the value represented is negative.

The two's complement of a negative number is the corresponding positive value. For example, inverting the bits of −5 (above) gives:

00000100

And adding one gives the final value:

$$00000101_2 = 5_{10}$$

The value of a two's-complement binary number can be calculated by adding up the power-of-two weights of the "one" bits, but with a negative weight for the most significant (sign) bit; for example:

$$11111011_2 = -128 + 64 + 32 + 16 + 8 + 0 + 2 + 1 = (-2^7 + 2^6 + \ldots) = -5$$

Note that the two's complement of zero is zero: inverting gives all ones, and adding one changes the ones back to zeros (the overflow is ignored). Also the two's complement of the most negative number representable (e.g. a one as the most-significant bit and all other bits zero) is itself. Hence, there appears to be an 'extra' negative number.

Since it wraps around

A more formal definition of a two's-complement negative number (denoted by N^* in this example) is derived from the equation $N^* = 2^n - N$, where N is the corresponding positive number and n is the number of bits in the representation.

For example, to find the 4 bit representation of −5:

$N = 5_{10}$ therefore $N = 0101_2$
$n = 4$

yeah how do it in reverse?
take 7 to 0111

Hence:

$$N^* = 2^n - N = 2^4 - 5_{10} = 10000_2 - 0101_2 = 1011_2$$

The calculation can be done entirely in base 10, converting to base 2 at the end:

$$N^* = 2^n - N = 2^4 - 5 = 11_{10} = 1011_2$$

Arithmetic operations

Addition

Adding two's-complement numbers requires no special processing if the operands have opposite signs: the sign of the result is determined automatically. For example, adding 15 and -5:

```
11111 111   (carry)
 0000 1111  (15)
+ 1111 1011  (-5)
==================
 0000 1010  (10)
```

The cool part

So AND each bit?

- but the carry!

This process depends upon restricting to 8 bits of precision; a carry to the (nonexistent) 9th most significant bit is ignored, resulting in the arithmetically correct result of 10_{10}.

The last two bits of the carry row (reading right-to-left) contain vital information: whether the calculation resulted in an arithmetic overflow, a number too large for the binary system to represent (in this case greater than 8 bits). An overflow condition exists when these last two bits are different from one another. As mentioned above, the sign of the number is encoded in the MSB of the result.

In other terms, if the left two carry bits (the ones on the far left of the top row in these examples) are both 1s or both 0s, the result is valid; if the left two carry bits are "1 0" or "0 1", a sign overflow has occurred. **Conveniently, an XOR operation on these two bits can quickly determine if an overflow condition exists.** As an example, consider the 4-bit addition of 7 and 3:

```
 0111   (carry)
  0111  (7)
+ 0011  (3)
=============
  1010  (-6)  invalid!
```

In this case, the far left two (MSB) carry bits are "01", which means there was a two's-complement addition overflow. That is, $1010_2 = 10_{10}$ is outside the permitted range of −8 to 7.

In general, any two *n*-bit numbers may be added *without* overflow, by first sign-extending both of them to *n*+1 bits, and then adding as above. The *n*+1 bit result is large enough to represent any possible sum (*e.g.*, 5 bits can represent values in the range −16 to 15) so overflow will never occur. It is then possible, if desired, to 'truncate' the result back to *n* bits while preserving the value if and only if the discarded bit is a proper sign extension of the retained result bits. This provides another method of detecting overflow—which is equivalent to the method of comparing the carry bits—but which may be easier to implement in some situations, because it does not require access to the internals of the addition.

Subtraction

Computers usually use the method of complements to implement subtraction. Using complements for subtraction is closely related to using complements for representing negative numbers, since the combination allows all signs of operands and results; direct subtraction works with two's-complement numbers as well. Like addition, the advantage of using two's complement is the elimination of examining the signs of the operands to determine if addition or subtraction is needed. For example, subtracting −5 from 15 is really adding 5 to 15, but this is hidden by the two's-complement representation:

```
 11110 000   (borrow)
  0000 1111  (15)
- 1111 1011  (-5)
===========
  0001 0100  (20)
```

Overflow is detected the same way as for addition, by examining the two leftmost (most significant) bits of the borrows; overflow has occurred if they are different.

Another example is a subtraction operation where the result is negative: 15 − 35 = −20:

```
 11100 0000  (borrow)
  0000 1111  (15)
- 0010 0011  (35)
===========
  1110 1100  (-20)
```

As for addition, overflow in subtraction may be avoided (or detected after the operation) by first sign-extending both inputs by an extra bit.

Multiplication

The product of two *n*-bit numbers requires 2*n* bits to contain all possible values. If the precision of the two two's-complement operands is doubled before the multiplication, direct multiplication (discarding any excess bits beyond that precision) will provide the correct result. For example, take 6 × −5 = −30. First, the precision is extended from 4 bits to 8. Then the numbers are multiplied, discarding the bits beyond 8 (shown by 'x'):

```
  00000110  (6)
× 11111011  (-5)
==========
       110
      110
     000
    110
   110
  110
 x10
xx0
==========
xx11100010  (-30)
```

This is very inefficient; by doubling the precision ahead of time, all additions must be double-precision and at least twice as many partial products are needed than for the more efficient algorithms actually implemented in computers. Some multiplication algorithms are designed for two's complement, notably Booth's multiplication algorithm. Methods for multiplying sign-magnitude numbers don't work with two's-complement numbers without adaptation. There isn't usually a problem when the multiplicand (the one being repeatedly added to form the product) is negative; the issue is setting the initial bits of the product correctly when the multiplier is negative. Two methods for adapting algorithms to handle two's-complement numbers are common:

- First check to see if the multiplier is negative. If so, negate (*i.e.*, take the two's complement of) both operands before multiplying. The multiplier will then be positive so the algorithm will work. Because both operands are negated, the result will still have the correct sign.
- Subtract the partial product resulting from the MSB (pseudo sign bit) instead of adding it like the other partial products. This method requires the multiplicand's sign bit to be extended by one position, being preserved during the shift right actions.[5]

As an example of the second method, take the common add-and-shift algorithm for multiplication. Instead of shifting partial products to the left as is done with pencil and paper, the accumulated product is shifted right, into a second register that will eventually hold the least significant half of the product. Since the least significant bits are not changed once they are calculated, the additions can be single precision, accumulating in the register that will eventually hold the most significant half of the product. In the following example, again multiplying 6 by −5, the two registers and the extended sign bit are separated by "|":

```
0 0110  (6)  (multiplicand with extended sign bit)
× 1011 (-5)  (multiplier)
=|====|====
0|0110|0000  (first partial product (rightmost bit is 1))
0|0011|0000  (shift right, preserving extended sign bit)
0|1001|0000  (add second partial product (next bit is 1))
0|0100|1000  (shift right, preserving extended sign bit)
0|0100|1000  (add third partial product: 0 so no change)
0|0010|0100  (shift right, preserving extended sign bit)
1|1100|0100  (subtract last partial product since it's from sign bit)
1|1110|0010  (shift right, preserving extended sign bit)
 |1110|0010  (discard extended sign bit, giving the final answer, -30)
```

Two's complement and universal algebra

In the classic "HAKMEM" published by the MIT AI Lab in 1972, Bill Gosper noted that whether or not a machine's internal representation was two's-complement could be determined by summing the successive powers of two. In a flight of fancy, he noted that the result of doing this algebraically indicated that "algebra is run on a machine (the universe) which is twos-complement."[6]

Gosper's end conclusion is not necessarily meant to be taken seriously, and it is akin to a mathematical joke. The critical step is "$\ldots 110 = \ldots 111 - 1$", *i.e.*, "$2X = X - 1$". This presupposes a method by which an infinite string of 1s is considered a number, which requires an extension of the finite place-value concepts in elementary arithmetic. It is meaningful either as part of a two's-complement notation for all integers, as a typical 2-adic number, or even as one of the generalized sums defined for the divergent series of real numbers $1 + 2 + 4 + 8 + \cdots$.[7]

Potential ambiguities in usage

One should be cautious when using the term *two's complement*, as it can mean either a number format or a mathematical operator. For example 0111 represents 7 *in two's-complement notation,* but 1001 is the *two's complement of* 7, which is the *two's complement representation of* −7. In code notation or conversation the statement "convert x to two's complement" may be ambiguous, as it could describe either the change in representation of x to two's-complement notation from some other format, or else (if the writer really meant "convert x to *its* two's complement") the calculation of the negated value of x.

See also

- Division (digital), including restoring and non-restoring division in two's-complement representations
- Signed number representations
- p-adic numbers
- One's complement
- Offset binary

External links

- Tutorial: Two's Complement Numbers (http://www.vb-helper.com/tutorial_twos_complement.html)
- Two's complement array multiplier JavaScript simulator (http://www.ecs.umass.edu/ece/koren/arith/simulator/ArrMlt/)

References

1. ^ David J. Lilja and Sachin S. Sapatnekar, *Designing Digital Computer Systems with Verilog*, Cambridge University Press, 2005 online (http://books.google.com/books?vid=ISBN052182866X&id=5BvW0hYhxkQC&pg=PA37&lpg=PA37&ots=l-E0VjyPt8&dq=%22two%27s+complement+arithmetic%22&sig=sS5_swrfrzcQI2nHWest75sIjgg)
2. ^ E.g. "Signed integers are two's complement binary values that can be used to represent both positive and negative integer values.", Section 4.2.1 in Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, November 2006
3. ^ Reynald Affeldt and Nicolas Marti. "Formal Verification of Arithmetic Functions in SmartMIPS Assembly" (http://www.ipl.t.u-tokyo.ac.jp/jssst2006/papers/Affeldt.pdf) . http://www.ipl.t.u-tokyo.ac.jp/jssst2006/papers/Affeldt.pdf.
4. ^ "Digital Design and Computer Architecture" (http://books.google.com/books?id=5X7JV5-n0FIC&pg=PA19&dq=%22weird+number%22+binary) by David Harris, David Money Harris, Sarah L. Harris. 2007. Page 18.
5. ^ John F. Wakerly, *Digital Design Principles & Practices*, Prentice Hall, 3rd edition 2000, page 47
6. ^ Hakmem - Programming Hacks - Draft, Not Yet Proofed (http://www.inwap.com/pdp10/hbaker/hakmem/hacks.html#item154)
7. ^ For the summation of 1 + 2 + 4 + 8 + ··· without recourse to the 2-adic metric, see Hardy, G.H. (1949). *Divergent Series*. Clarendon Press. LCC QA295 .H29 1967 (http://catalog.loc.gov/cgi-bin/Pwebrecon.cgi?Search_Arg=QA295+.H29+1967&Search_Code=CALL_&CNT=5) . (pp. 7–10)

- Israel Koren, *Computer Arithmetic Algorithms*, A.K. Peters (2002), ISBN 1-56881-160-8
- Ivan Flores, *The Logic of Computer Arithmetic*, Prentice-Hall (1963)

Retrieved from "http://en.wikipedia.org/wiki/Two%27s_complement"
Categories: Computer arithmetic

- This page was last modified on 19 September 2011 at 22:01.
-

Hamming Distance

9/25

Review

2 out of 5 code

- did have Happing distance 2

But what did example on test have?

- Not posted yet

~~* Code can detect n bit errors~~

↑ I got that don't want anoter valid code word

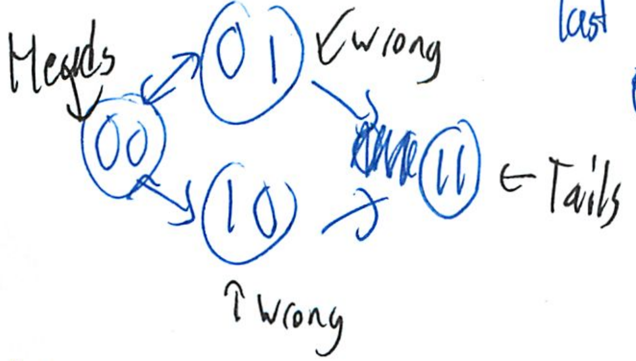
* If D is min Hamming Distance, can <u>detect</u> up to D-1 errors

↑ better statement than on review sheet

Did not read Hamming distance note when reviewing

Can add a parity bit to garentee HD at least 2

In example



last bit to make total # of 1s even

So now HD = 2 b/w 00, 11

Before just had 1 bit 0 or 1

②

To correct

* If D is min hamming distance, can correct up to $\left\lfloor \frac{D-1}{2} \right\rfloor$ bit errors

So if ↑ HD to 3 for transmitting 1 bit

Can correct single bit errors

heads

000

111

tails

just swap the one back

Review section

Detect D bit errors $HD > D$

Correct " " " $HD > 2D$

So I think I was thinking this - but prob made mistakes

- good to formalize

Timing Review

9/25

What did I get wrong here?

Mostly analysis

Think forgot exact specifics

— but I did review / write down on review sheet

So CD is we start an invalid in to when our output turns invalid

PD is ~~[illegible] valid input~~ our input has just become valid — how long for proper output?

?Remember

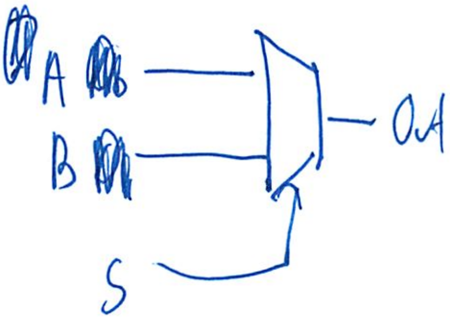
Multiplexers Study

9/25

up **Multiplexer/Mux** Selects 1 of several analog/digital signals and combines them to a single line

- 2^n inputs has n select lines that sends to out
- called data selector
- ↑ # of data sent over network with same time/bandwidth

2-to-1 multipler



$$Out = (A \cdot \bar{S}) + (B \cdot S)$$

S	A	B	Out
0	1	1	1
	1	0	1
	0	1	0
	0	0	0
1	1	1	1
	1	0	0
	0	1	1
	0	0	0

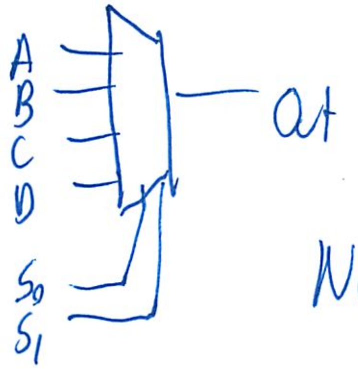
so when S=1 B outputs

when S=0 A outputs

Why it is called a selector

Then can make it bigger

4 to 1



Needs 2 selectors

∴ need $\lceil \log_2(n) \rceil$ selectors

$\text{Out} = (A \cdot \overline{S_0}\,\overline{S_1}) + (B \cdot S_0 \cdot \overline{S_1}) + (C \cdot \overline{S_0} \cdot S_1) + (D \cdot S_0 \cdot S_1)$

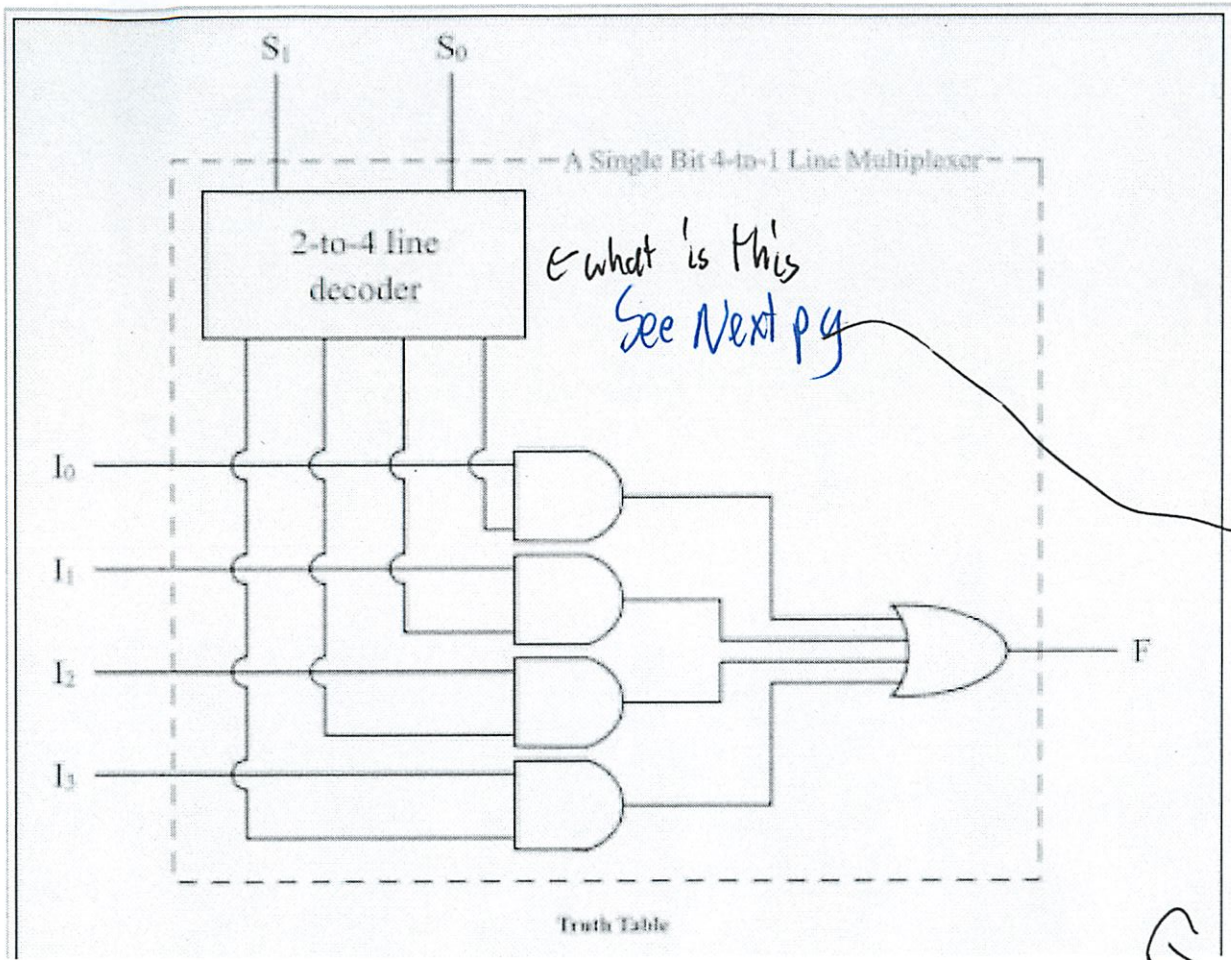
4 · 16 line TT!

~~Ba~~ Basically when $S_0 = 0$ $S_1 = 0$ outputs A

0	1	B
1	0	C
1	1	D

Is made of combo of or gates

↓

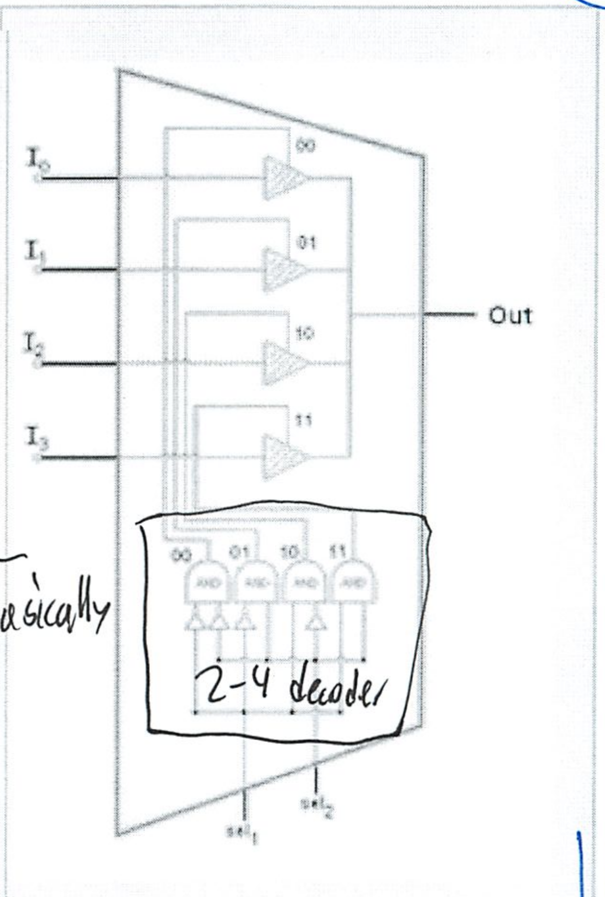


Truth Table

← what is this

See Next pg

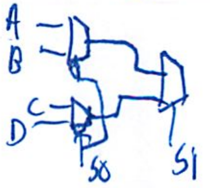
basically



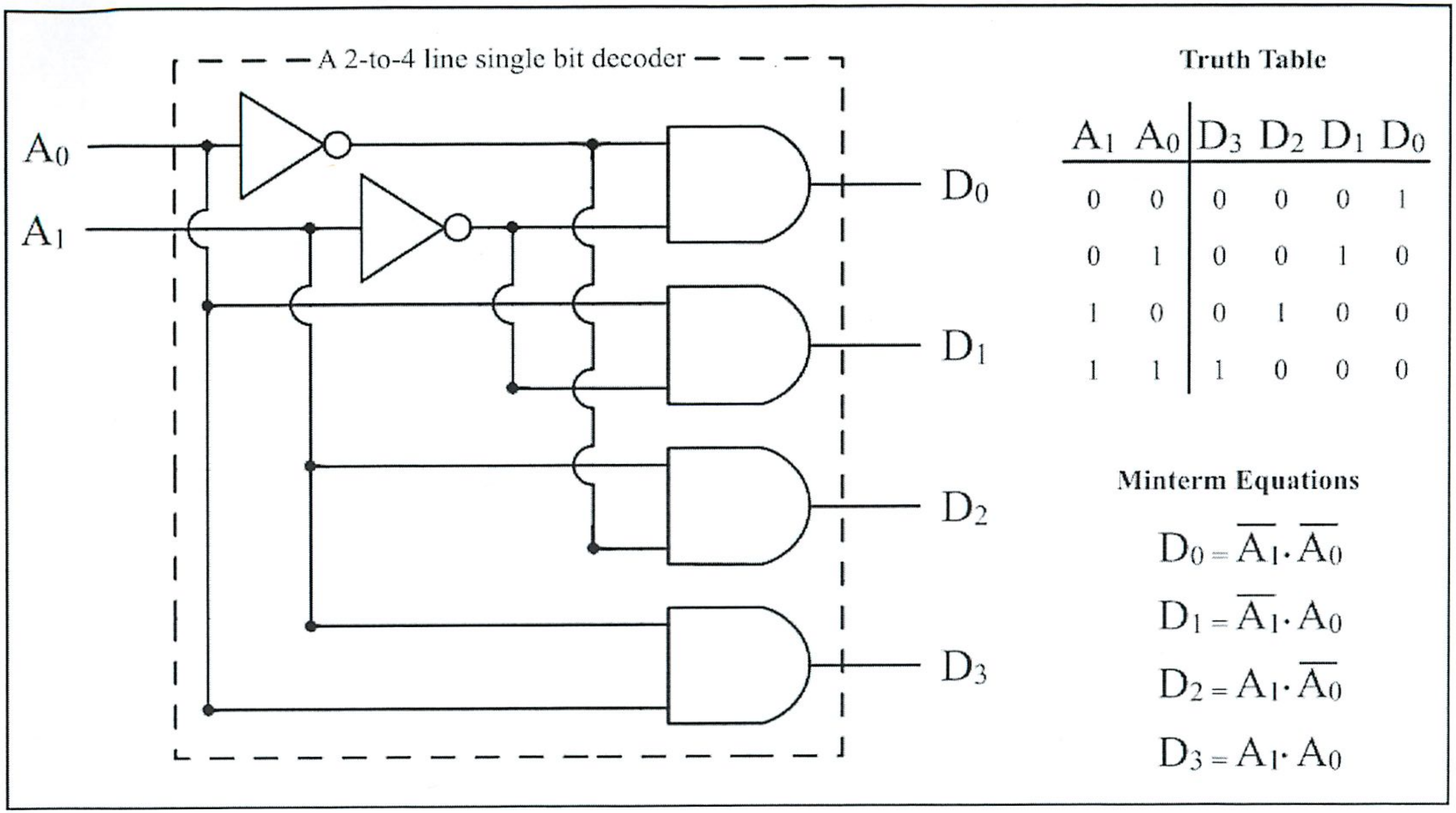
Wikipedia

two possible internal ways of implementing

Also lecture notes show are just combos of smaller muxes



3



Truth Table

A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Minterm Equations

$$D_0 = \overline{A_1} \cdot \overline{A_0}$$

$$D_1 = \overline{A_1} \cdot A_0$$

$$D_2 = A_1 \cdot \overline{A_0}$$

$$D_3 = A_1 \cdot A_0$$

~~HA~~ Adder Study

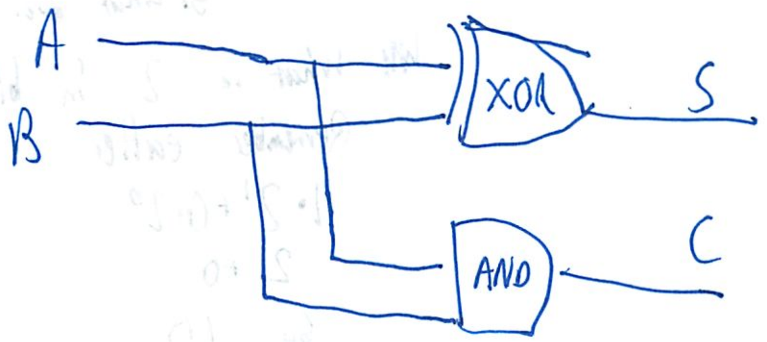
9/28

Adder - addition

- construct for your numeric representation
  - binary coded decimal
  - excess 3
  - binary # ← most common

Half adder - adds 2 1-bit # A, B

- 2 outs S, C ← can carry
- Sum is 2C + S



Can't use compositly

- no "carry in" bit

? is this carry flag? bit if carry or borrow has been generated - allows # > single ALU of least significant bit back ↪

②

for example 255 + 255 = 510

1_1111_1110

9 bits

So return 1111_1110 + carry

In 2's complement this is $-1 + -1 = -2$

correct!

So lets try $A=1$, $B=1$ → $S = 1 \text{ XOR } 1 = 0$

$C = 1 \text{ AND } 1 = 1$

So what does this mean?

What is 2 in binary?

Remember earlier

$1 \cdot 2^1 + 0 \cdot 2^0$

$2 + 0$

So $\underset{1}{1}\underset{0}{0}$

So makes sense

But then where is carry problem?

Something about composibility

~~[illegible]~~ 3

Full adder - Adds binnary # and keeps carries

A ~~[illegible]~~ "1 bit full adder" has 3 inputs

A
B) operand
C - carryed in from previous
This is difference here
Can string them together

So what if

$\underset{A}{1} + \underset{B}{1} + \underset{D}{1}$ using half

$\begin{matrix} A=1 \\ B=1 \end{matrix} \rightarrow \begin{matrix} S=0 \\ C=1 \end{matrix} \rightarrow \begin{matrix} 0 & 1 \\ C & 1 \end{matrix} = \begin{matrix} S=0 \\ C=1 \end{matrix}$

So
$2 \cdot C + S$
$\underset{2}{2 \cdot 1 + 0}$

but what about bit positions, etc
and where did S go
Look at how full is different.

Still $2C_{out} + S$

$S = A \oplus B \oplus C_{in}$

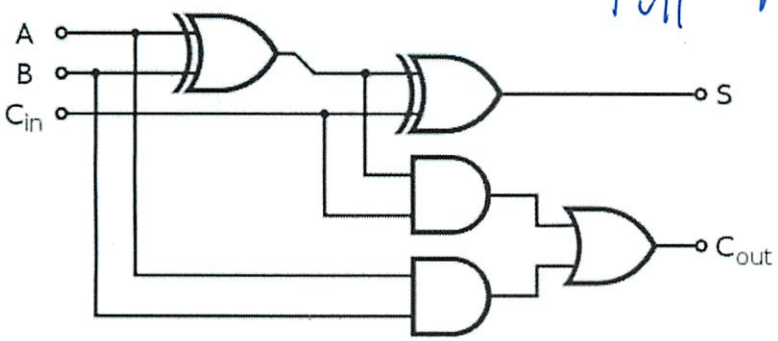
$C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$ $\oplus$ = XOR

Or

$C_{out} = (A \cdot B) \oplus (C_{in} \cdot (A \oplus B))$

4

Full Adder



AND/OR gates can be replaced w/ NAND for same

Ripple Adder

Use multiple full adders to add N-bit #

The C_{in} is the previous Car

kinda slow

- 32 bit so worst case

$$\underbrace{31 \cdot 2}_{\text{Carry Prop}} + \underbrace{3}_{\text{Sum}} = 65 \text{ gate delays}$$

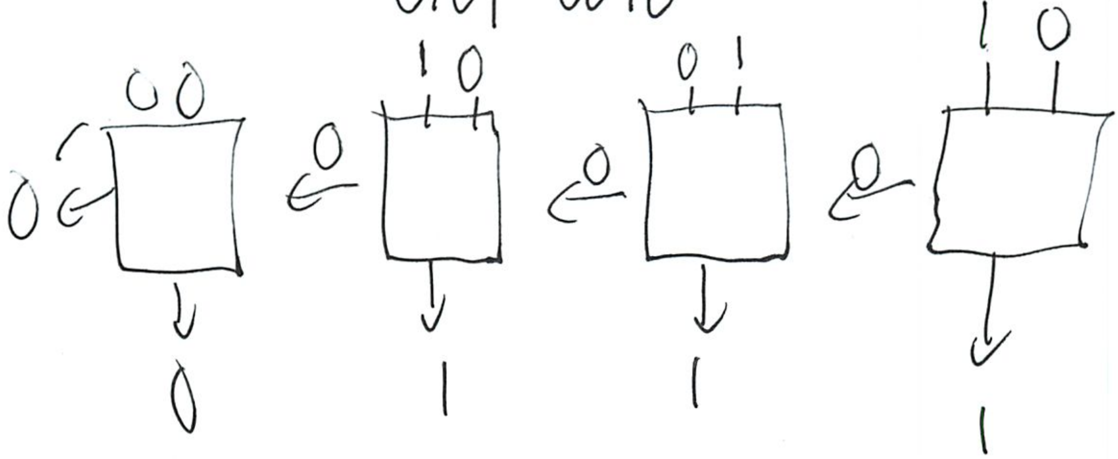
¿So where do you put the #s?

I guess each bit one at a time

5

So 5+2

0101 0010



is

$0\cdot2^3+1\cdot2^2+1\cdot2^1+1\cdot2^0$

$4+2+1$

7 ✓

Decoder

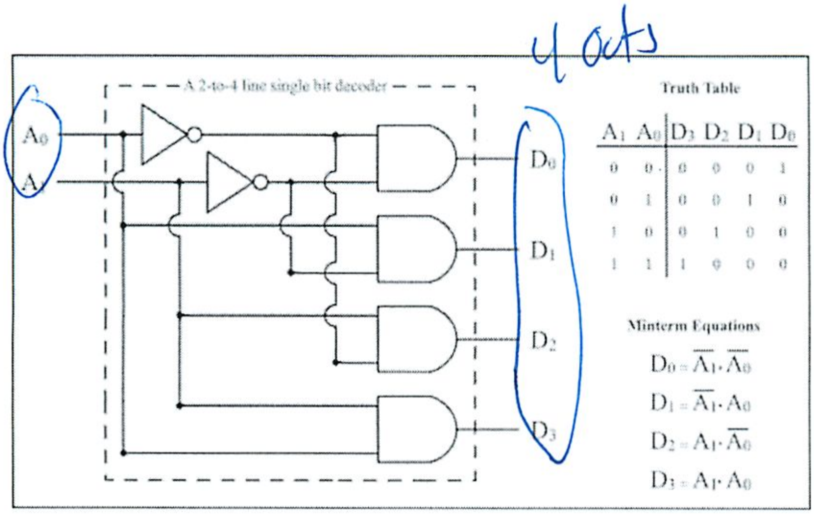
9/25

Decoder
- Reverse of encoder
- same thing in reverse
- Most basic: AND gate
  - only 1 when all ins are high
- n-to-n^2 more complex

  ^ Up to max n^2
  but some may be unused

4 outs

2 ins



Truth Table

A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Minterm Equations

$D_0 = \overline{A_1} \cdot \overline{A_0}$

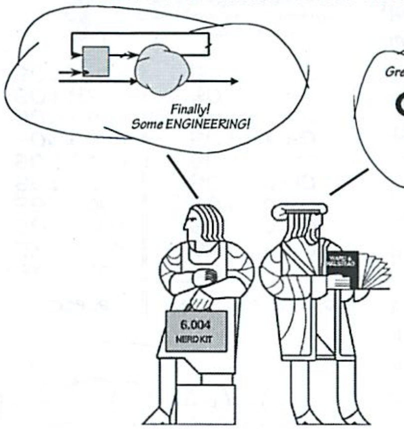
$D_1 = \overline{A_1} \cdot A_0$

$D_2 = A_1 \cdot \overline{A_0}$

$D_3 = A_1 \cdot A_0$

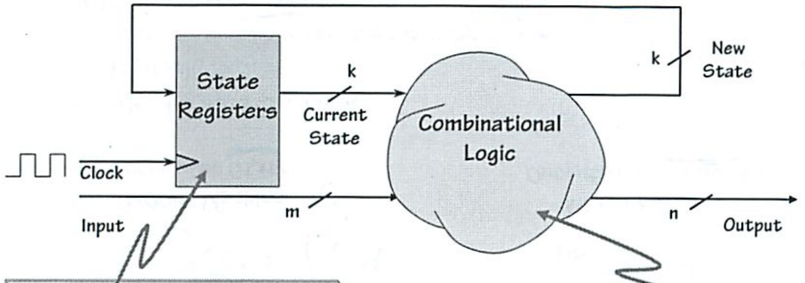
? Same decoder we looked at earler as part of encoder

(Synchronous) Finite State Machines



Lab 2 is due Thursday!

Our New Machine



- Engineered cycles
- Works only if dynamic discipline obeyed
- Remembers k bits for a total of 2^k unique combinations

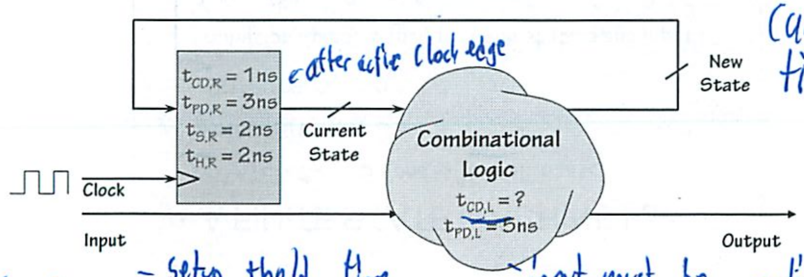
On every clock edge can change state

- Acyclic graph
- Obeys static discipline
- Can be exhaustively enumerated by a truth table of 2^{k+m} rows and k+n output columns

very careful cycles - tricky to implement properly

dynamic discipline

Must Respect Timing Assumptions!



after the clock edge

Can compute timing

- setup + hold time

input must be valid for some time

setup before edge
hold after "

Questions:

- Constraints on T_{CD} for the logic?
- Minimum clock period?
- Setup, Hold times for Inputs?

$t_{CD,R}(1\text{ ns}) + t_{CD,L}(?) > t_{H,R}(2\text{ ns})$

$t_{CD,L} > 1\text{ ns}$ answer - hold time for flip flop

$t_{CLK} > t_{PD,R}+t_{PD,L}+t_{S,R} > 10\text{nS}$ for setup

$t_S = t_{PD,L} + t_{S,R} = 7\text{ nS}$

$t_H = t_{H,R} - t_{CD,L} = 1\text{ nS}$

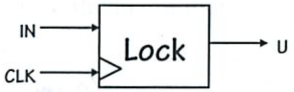
look at specs

We know how fast it goes... But what can it do? inputs

today

A simple sequential circuit...

Lets make a digital binary *Combination Lock:*



Specification:

- One input ("0" or "1") String of bits
- One output ("Unlock" signal)
- Assert UNLOCK is 1 if and only if:

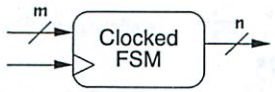
Last 4 inputs were the "combination": 0110

How much state to save?

How many registers do I need?

9/27

Abstraction du jour: Finite State Machines

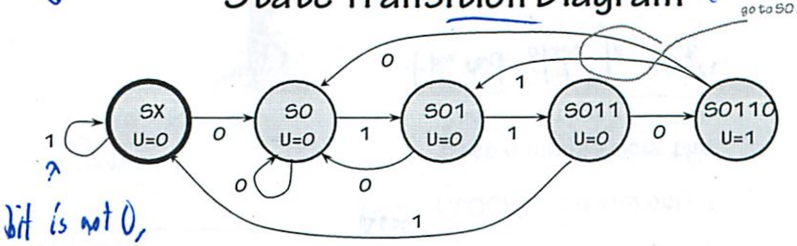


- A FINITE STATE MACHINE has
  - k STATES: $S_1 \ldots S_k$ (one is "initial" state)
  - m INPUTS: $I_1 \ldots I_m$
  - n OUTPUTS: $O_1 \ldots O_n$
  - Transition Rules: s'(s, I) for each state s and input I
  - Output Rules: Out(s) for each state s

Map out functionality

State Transition Diagram

(can ^have been the 1st ~~state~~ digits of correct string



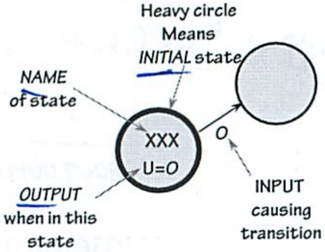
Why do these go to S0 and S01?

?

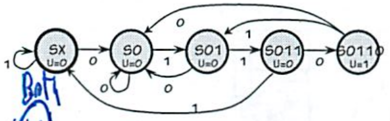
1st bit is not 0, reset

Designing our lock …

- Need an initial state; call it SX.
- Must have a separate state for each step of the proper entry sequence
- Must handle other (erroneous) entries



Yet Another Specification



Put in table form

IN	Current State		Next State Unlock	
0	SX	000	S0	0010
1	SX	000	SX	0000
0	S0	001	S0	0010
1	S0	001	S01	0110
0	S01	011	S0	0010
1	S01	011	S011	0100
0	S011	010	S0110	1000
1	S011	010	SX	0000
0	S0110	100	S0	0011
1	S0110	100	S01	0111

The assignment of codes to states can be arbitrary, however, if you choose them carefully you can greatly reduce your logic requirements.

5 distinct states = 3 bits

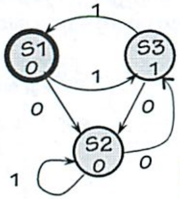
All state transition diagrams can be described by truth tables…

Binary encodings are assigned to each state (a bit of an art)

The truth table can then be simplified using the reduction techniques we learned for combinational logic

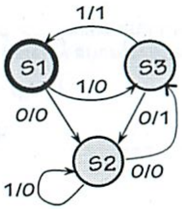
how will we represent state in binary?

Valid State Diagrams



MOORE Machine: Outputs on States

6.004

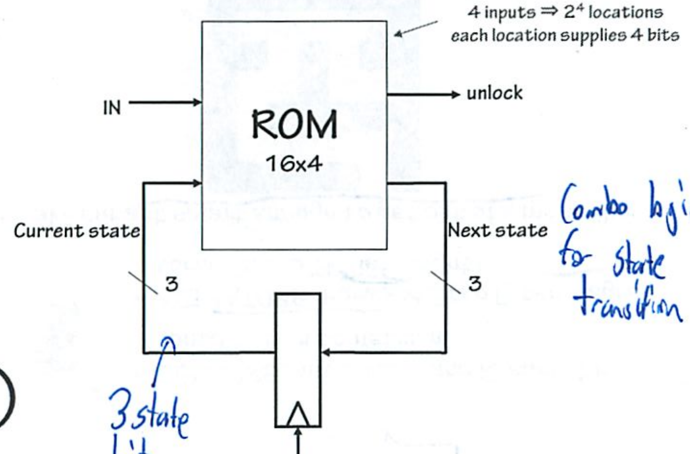


MEALY Machine: Outputs on Transitions

I was thinking this

- Arcs leaving a state must be:
- (1) mutually exclusive
  - *can't have two choices for a given input value*
- (2) collectively exhaustive
  - *every state must specify what happens for each possible input combination. "Nothing happens" means arc back to itself.*

Now put it in Hardware!

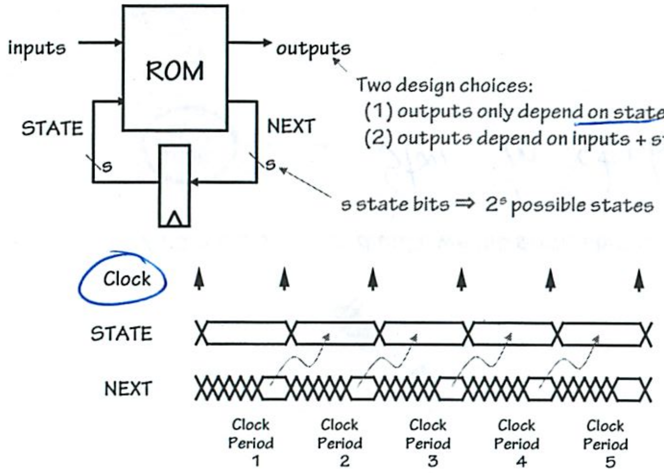

4 inputs ⇒ 2^4 locations
each location supplies 4 bits
IN
ROM
16x4
unlock
Current state
Next state
3
3
Trigger update periodically ("clock")

Combo logic for state transition

3 state bit register

We assume inputs are synchronized with clock...

Discrete State, Time



Two design choices:

(1) outputs only depend on state (Moore)

(2) outputs depend on inputs + state (Mealy)

s state bits ⇒ 2^s possible states

Asynchronous Inputs - I

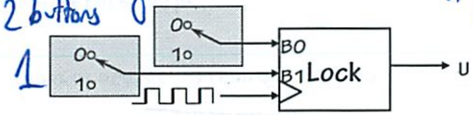
New bit at every clock edge - not good for people

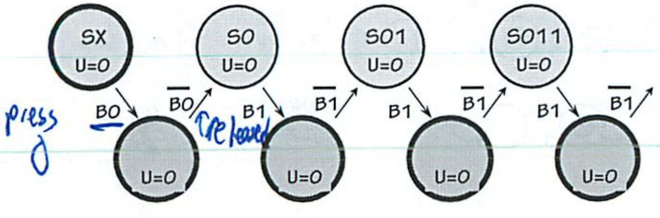
Our example assumed a single clock transition per input. What if the "button pusher" is unaware of, or not synchronized with, the clock?

What if each button input is an asynchronous 0/1 level? How do we prevent a single button press, e.g., from making several transitions?

2 buttons 0

1





press 0

released

But what About the Dynamic Discipline?

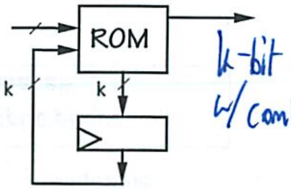
glossing over

Use intervening states to synchronize button presses!

FSM Party Games

1. What can you say about the number of states?

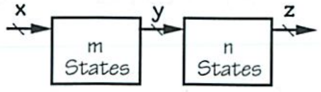
$< 2^k$



k-bit register w/ combo logic

2. Same question:

$< m \cdot n$



3. Here's an FSM. Can you discover its rules?

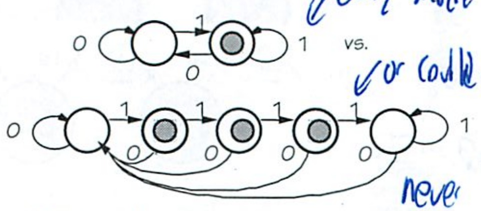
not in general case

might

What’s My Transition Diagram?

0=OFF, 1=ON?

"1111" = Surprise!



easy model

or could it be

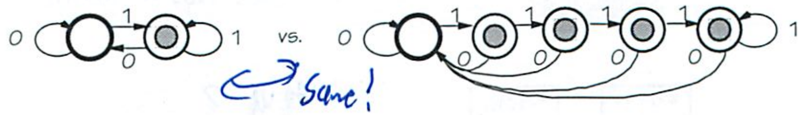
never know!

- If you know NOTHING about the FSM, you’re never sure!
- If you have a BOUND on the number of states, you can discover its behavior:

K-state FSM: Every (reachable) state can be reached in < k steps.

BUT ... states may be equivalent! modulo

FSM Equivalence



Same!

ARE THEY DIFFERENT?

NOT in any practical sense! They are EXTERNALLY INDISTINGUISHABLE, hence interchangeable.

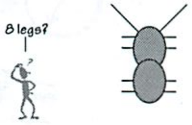
> FSMs *EQUIVALENT* iff every input sequence yields identical output sequences.

ENGINEERING GOAL:

- HAVE an FSM which *works...*
- WANT simplest (ergo cheapest) equivalent FSM.

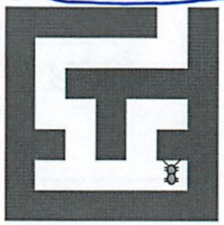
Just use cheaper - if equivilant

Lets build an Ant



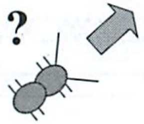
- SENSORS: antennae L and R, each 1 if in contact with something.
- ACTUATORS: Forward Step F, ten-degree turns TL and TR (left, right).

GOAL: Make our ant smart enough to get out of a maze like:

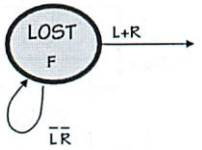


STRATEGY: "Right antenna to the wall"

Lost in space



Action: *Go forward until we hit something.*

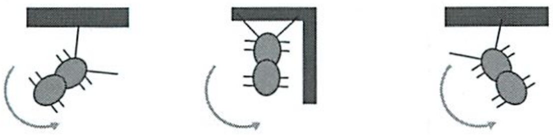


“lost” is the initial state

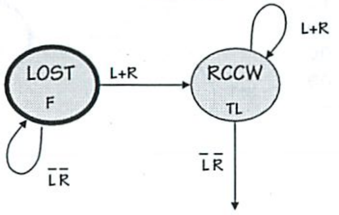
Stay in state

Bonk!

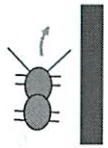
I remember this in 6.01



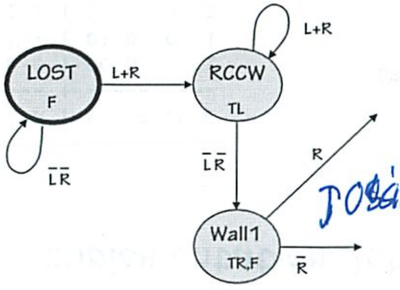
Action: Turn left (CCW) until we don't touch anymore



A little to the right…



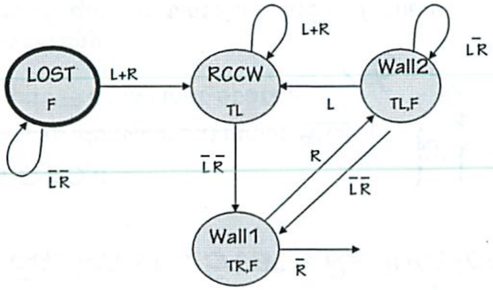
Action: Step and turn right a little, look for wall



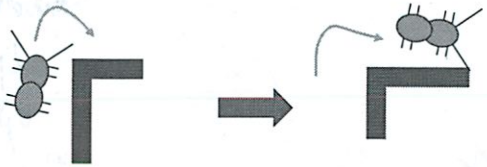
Toscilate along wall

Then a little to the left

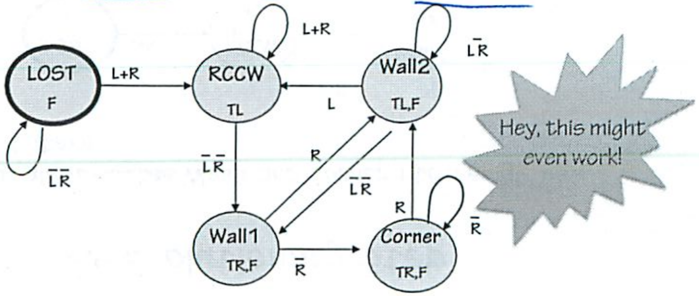
Action: Step and turn left a little, till not touching (again)



Dealing with corners



Action: Step and turn right until we hit perpendicular wall



Hey, this might even work!

Can we simplify it?

Equivalent State Reduction

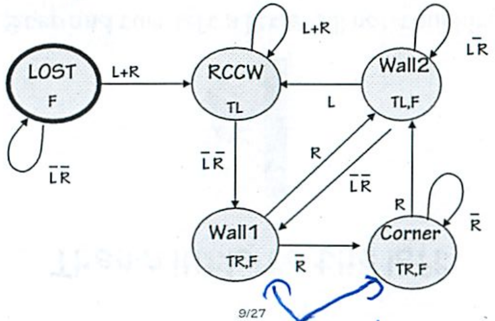
Observation: $S_i \cong S_j$ if

1. States have identical outputs; AND
2. Every input ⇒ equivalent states.

look for equivilant

Reduction Strategy:

Find pairs of equivalent states, MERGE them.

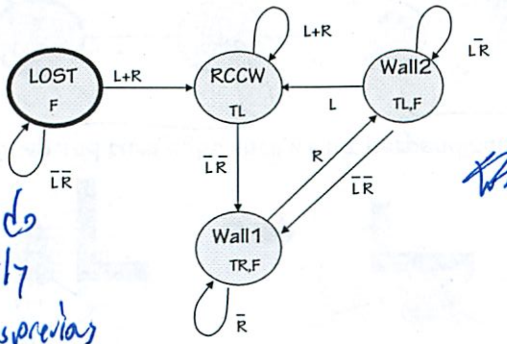


same outs

merge

An Evolutionary Step

Merge equivalent states Wall1 and Corner into a single new, combined state.



LOST
F
$\overline{L}\,\overline{R}$
L+R
RCCW
TL
L+R
$\overline{L}\,\overline{R}$
Wall2
TL,F
$L\overline{R}$
L
R
$\overline{L}\,\overline{R}$
Wall1
TR,F
$\overline{R}$

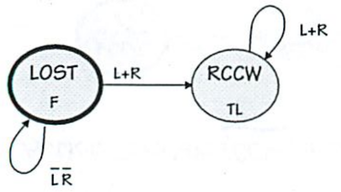
will do exactly same as previous

~~Wall2~~ do

Behaves exactly as previous (5-state) FSM, but requires half the ROM in its implementation!

Building the Transition Table

put together trans table



S	L	R	S'	TR	TL	F
00	0	0	00	0	0	1
00	1	-	01	0	0	1
00	0	1	01	0	0	1
01	1	-	01	0	1	0
01	0	1	01	0	1	0

Implementation Details

do state assignment

	S	L	R	S'	TR	TL	F
LOST	00	0	0	00	0	0	1
	00	1	-	01	0	0	1
	00	0	1	01	0	0	1
RCCW	01	1	-	01	0	1	0
	01	0	1	01	0	1	0
	01	0	0	10	0	1	0
WALL1	10	-	0	10	1	0	1
	10	-	1	11	1	0	1
WALL2	11	1	-	01	0	1	1
	11	0	0	10	0	1	1
	11	0	1	11	0	1	1

Complete Transition table

S1'

$$\begin{array}{cc|cccc} & & \multicolumn{4}{c}{S_1S_0} \\ & & 00 & 01 & 11 & 10 \\ \hline & 00 & 0 & 1 & 1 & 1 \\ \text{LR} & 01 & 0 & 0 & 1 & 1 \\ & 11 & 0 & 0 & 0 & 1 \\ & 10 & 0 & 0 & 0 & 1 \end{array}$$

$$S_1' = S_1\overline{S_0} + \overline{L}S_1 + \overline{LR}S_0$$

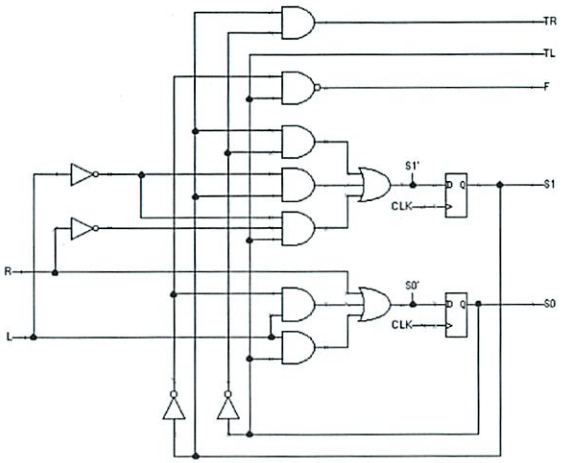
S0'

$$\begin{array}{cc|cccc} & & \multicolumn{4}{c}{S_1S_0} \\ & & 00 & 01 & 11 & 10 \\ \hline & 00 & 0 & 0 & 0 & 0 \\ \text{LR} & 01 & 1 & 1 & 1 & 1 \\ & 11 & 1 & 1 & 1 & 1 \\ & 10 & 1 & 1 & 1 & 0 \end{array}$$

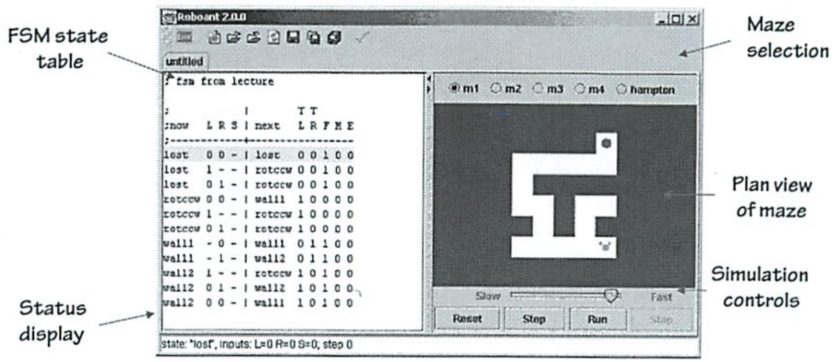
$$S_0' = R + L\overline{S_1} + LS_0$$

Ant Schematic

Can do it in logic



Roboant®



```
; fsm from lecture

;              |          T T
;now    L R S | next     L R F M E
;-------------+----------------
lost    0 0 - | lost     0 0 1 0 0
lost    1 - - | rotccw   0 0 1 0 0
lost    0 1 - | rotccw   0 0 1 0 0
rotccw  0 0 - | wall1    1 0 0 0 0
rotccw  1 - - | rotccw   1 0 0 0 0
rotccw  0 1 - | rotccw   1 0 0 0 0
wall1   - 0 - | wall1    0 1 1 0 0
wall1   - 1 - | wall2    0 1 1 0 0
wall2   1 - - | rotccw   1 0 1 0 0
wall2   0 1 - | wall2    1 0 1 0 0
wall2   0 0 - | wall1    1 0 1 0 0
```

```
state: "lost", inputs: L=0 R=0 S=0; step 0
```

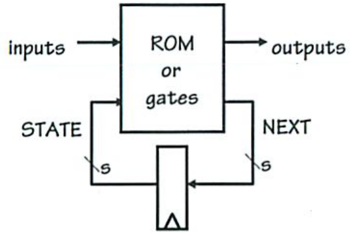
Featuring the new Mark-II ant: can add (M), erase (E), and sense (S) marks along its path.

But strategy won't work for

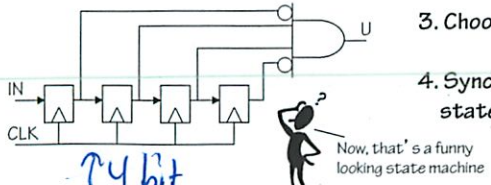
so breadcrums to detect cycles

Things to worry about

Housekeeping issues...



1. Initialization? Clear the memory? async clear -force all 0000
2. Unused state encodings? Unused states
   - waste ROM (use PLA or gates)
   - what does it mean?
   - can the FSM recover?
3. Choosing encoding for state? State assignments — good to be clever
4. Synchronizing input changes with state update?



- 4 bit combo locks
- remembers last 4 checks logic

Can also do w/ equiv state reduction

Twisting you Further...

- MORE THAN ANTS:
  Swarming, flocking, and schooling can result from collections of very simple FSMs
- PERHAPS MOST PHYSICS:
  Cellular automata, arrays of simple FSMs, can more accurately model fluilds than numerical solutions to PDEs
- WHAT IF:
  We replaced the ROM with a RAM and have outputs that modify the RAM?

... You'll see FSMs for the rest of your life!

Did we all descend from FSMs???
I prefer to think we ascended...
6.004 NERD KIT
WARD & HALSTEAD

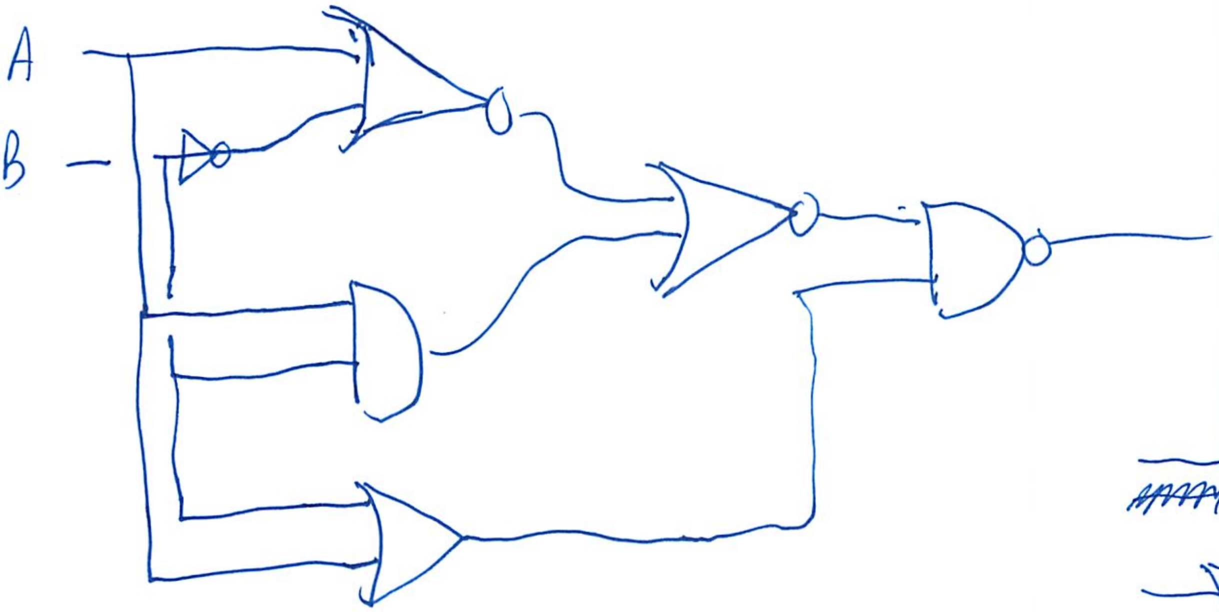
Huge lit on this

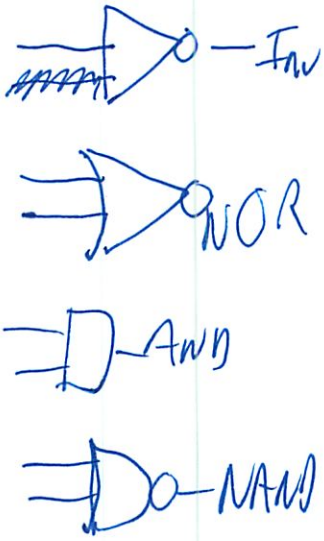
Recitation

9/28

- 1 recitation per lecture
  - so a bit behind

Today: Building logic w/ CMOS





So truth table
- go through each possible input
- try each row
  - fill in w/ pencil
  - slow + plodding

A	B	Z
0	0	1
0	1	1
1	0	etc
1	1	

- N inputs = 2^N outputs

2

$2^{(2^N)}$ possible TT rows
possible gates w/ N inputs
(given for 2 on slides)

Turniary logic · 2 inputs

~~Alsdo~~ Perforance

Valid output 0
↓ to valid input 1

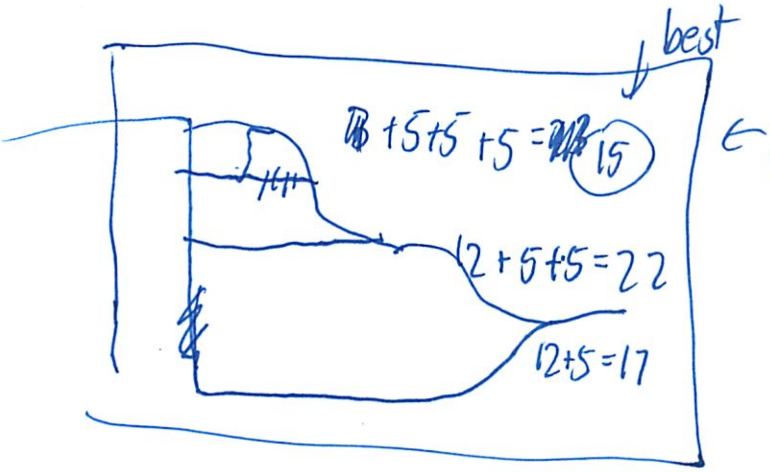
← Given for each date

		t_{CD}	t_{PD}	t_R	t_F
Shorthand	I	3	15	8	5
NAND →	ND2	5	30	11	7
AND →	AN2	12	50	13	9
NOR →	NR2	5	30	7	11
OR →	OR2	12	56	9	13
	Z Whole system				

← Now get for Z

t_{CD} = minimum of all paths
- not just of visually shortest path!

So calculate each
- add along each possible row



← don't need to do ^inverter one since will be longer than the one above it

t~pd~ - now looking for longest

- innumerate each one again, sum

- using t_{pd}

- $\sum t_{pd}$

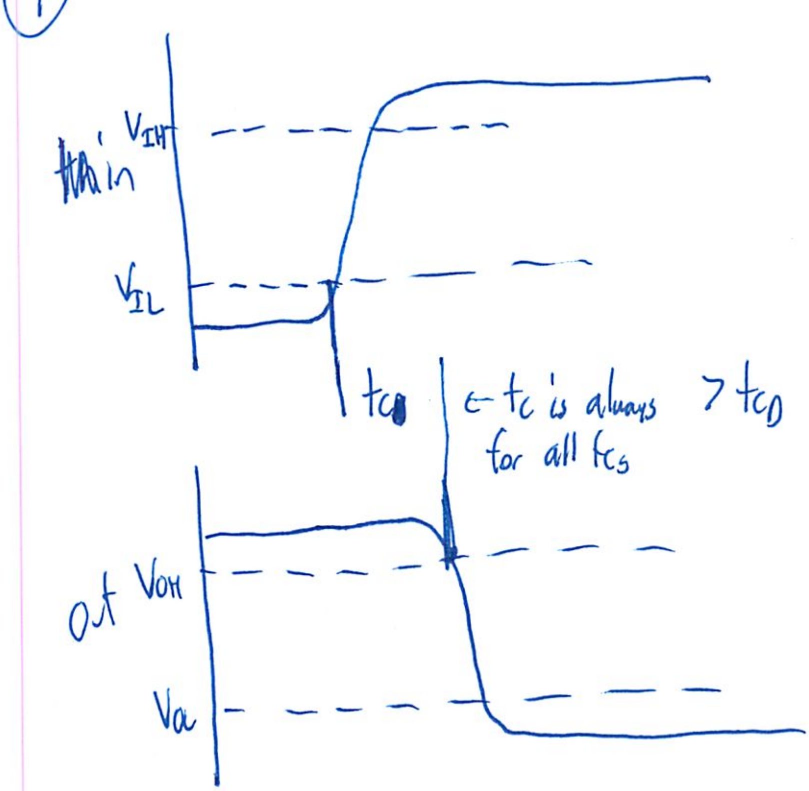
t~rise~ + t~fall~

- When Z goes 0 → 1

- Just case about ^last gate the output is hooked to

? since this just looks at output

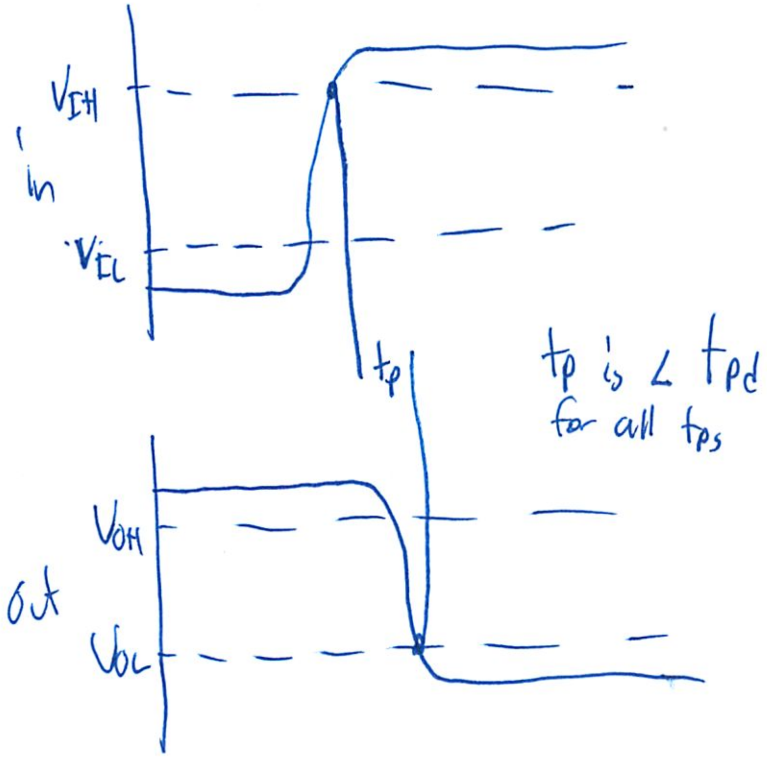
4



← t_c is always > t_{cD} for all t_cs

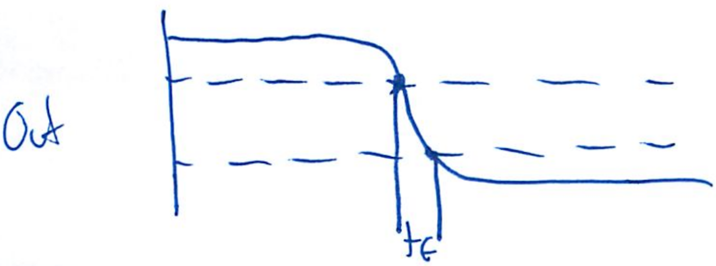
t_{cd} → invalid in to invalid out

- too short t_{cd} will be a problem later



t_p is < t_{pd} for all t_ps

t_{pd} valid in → valid out



t_f valid 1 → valid 0

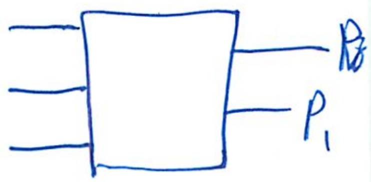
← to reverse ⎽/⎺ for t_r

5

Now the reverse.

Get truth table.
Build circuit.

Priority		
3	11	A
2	10	B
1	01	C



This is priority encoder
–like interupt circuit

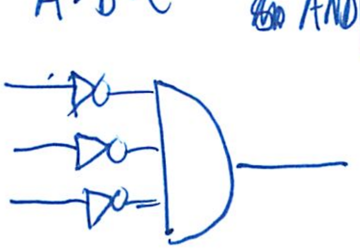
2^3 → rows

A B C	P0 P1
000	00
001	01
010	10
011	10
100	11
101	11
110	11
111	11

Now instead of drawing gates, write expression

Each row of TT represents product term
(did in class)

First row $\bar{A}\cdot\bar{B}\cdot\bar{C}$ so AND(AND($\bar{A}$,$\bar{B}$),C)



Then or together all the results that are 1 for P_0

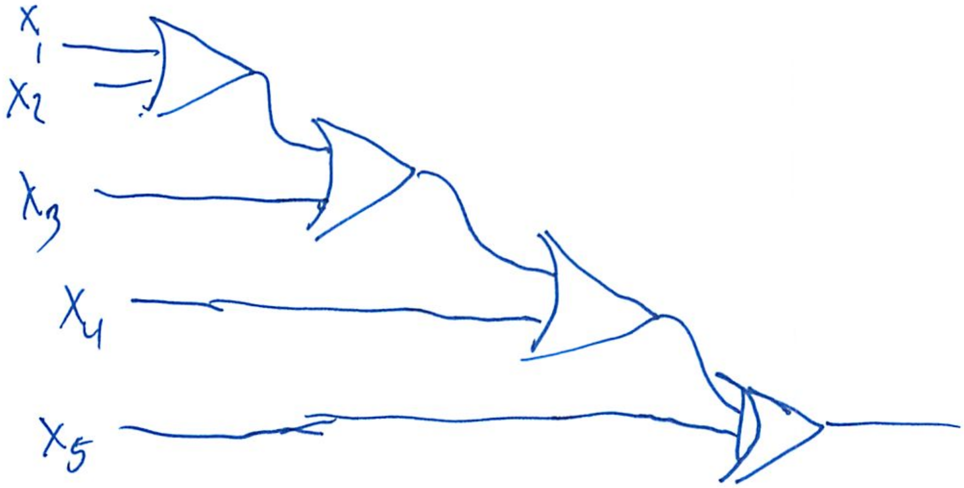
$\bar{A}B\bar{C}$
$\bar{A}BC$
$A\bar{B}C$
$A\bar{B}\bar{C}$
~~B~~$AB\bar{C}$
ABC

OR → P_0

(or write w/ +)

repeat for P_1

Could also cascade it



But slower if all inputs arrived at same time

Becomes important in lab 3

Now can you do it w/ less gates

First can build don't care TTs

since $A\bar{B}\bar{C} = A\bar{B}C = AB\bar{C} = ABC$

all same output

just say A - -

Can represent as factor

$$\bar{A}B(\underbrace{\bar{C}+C}_{1}) = \bar{A}B$$

Can do w/ table

- "k-maps"
- "karnaugh"

2 variables

$\overbrace{AB}$

PØ	00	01	11	10
0	0	1	1	1
C 1	0	1	1	1

← order very specific

← fill in from TT

gray code →

only 1 changes

loops back

Good for up to 4 nodes

⑧

Can have patches of sizes:

1 x 1

1 x 2   2 x 1

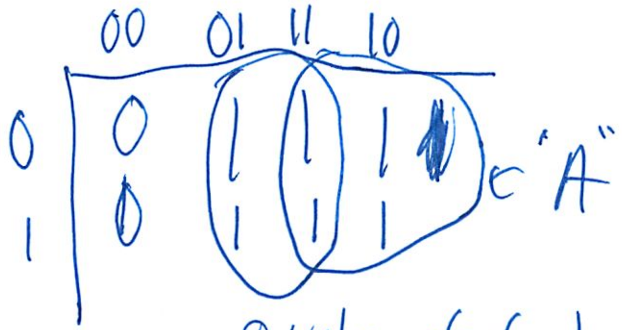
2 x 2

4 x 1   1 x 4

4 x 4

Think of largest patches can circle

- Overlapping is ~~fine~~ fine, ⇒ key to sucess



	00	01	11	10
0	0	1	1	1
1	0	1	1	1

? value of C does not matter
Vale of B = 1
A does not matter } "B" ← so what does that mean.

So for 1s   A + B ← how did we get this?

~~What this~~

On your own ; ROM

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

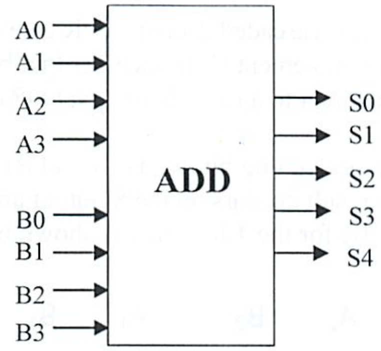
6.004 Computation Structures
Lab #2

Read 9/24

Introduction

Your mission this week is to design and test a CMOS circuit that performs addition on two unsigned 4-bit numbers, producing a 5-bit result:

have not leared yet



When you've completed and tested your design, you can ask JSim to send your circuit to the on-line assignment system using the process described at the end of Lab #1. The checkoff file for Lab #2 (lab2checkoff.jsim) checks that your circuit has the right functionality; the on-line system will give you 5 points for checking off your lab using this file. (You'll receive your points after completing the on-line questions and a checkoff meeting with a TA.)

Note: Our ability to provide automated checkoffs is predicated on trusting that you'll use the checkoff and library files as given. Since these files are included in your submission, we will be checking to see if these files have been used as intended. Submittals that include modified checkoff or library files will be regarded as a serious breach of our trust and will be dealt with accordingly.

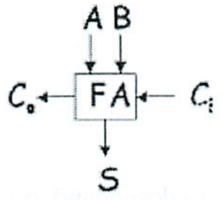
1: Ripple Adders

Let's start with a simple ripple-carry adder based on the full-adder module discussed in lecture. Later we'll discuss higher performance adder architectures you can use in the implementation of the Beta (the computer central processing unit we'll be designing in later labs).

The full adder module has 3 inputs (A, B and C_i) and 2 outputs (S and C_o). The logic equations and truth table for S and C_o are shown below.

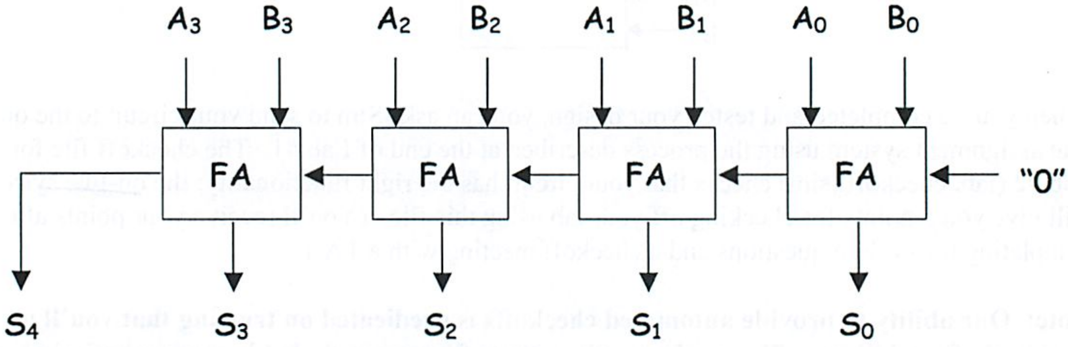
$$S = A \oplus B \oplus C_{in} \qquad C_o = A \cdot B + A \cdot C_{in} + B \cdot C_{in}$$

C_i	A	B	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Typically S is implemented using two cascaded 2-input XOR gates. One can use three 2-input NANDs and one 3-input NAND to implement C_o (remember that by Demorgan's Law two cascaded NANDs is logically equivalent to a cascade of AND/OR).

The module performs the addition of two one-bit inputs (A and B) incorporating the carry in from the previous stage (C_i). The result appears on the S output and a carry (C_o) is generated for the next stage. A possible schematic for the 4-bit adder is shown below:

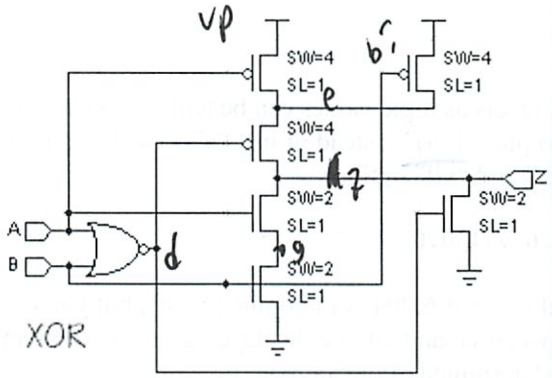


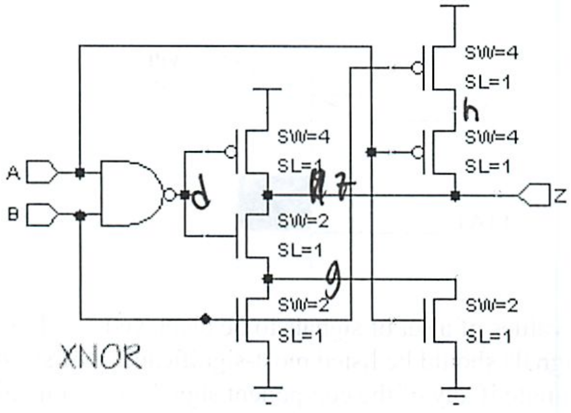
2: XOR/XNOR Gates

Since we're using individual gates to implement the logic, a good place to start is to build your own gate library (e.g., inverter, 2-input NAND, 2-input NOR, 2-input XOR), test them individually, and then use them to implement your design. It's much easier to debug your circuit module-by-module rather than as one big lump. XOR/XNOR can be challenging gates to design; here's one suggestion for how they might be implemented:

They should pre build

- can copy from last week





3: Generating Test Signals

You can use voltage sources with either a pulse or piece-wise linear waveforms to generate test signals for your circuit (see Lab #1 for details). Another source of test waveforms is the file "/mit/6.004/jsim/8clocks.jsim" which can be included in your netlist. It provides eight different square waves (50% duty cycle) with different periods:

clk1	period = 10ns
clk2	period = 20ns
clk3	period = 40ns
clk4	period = 80ns
clk5	period = 160ns
clk6	period = 320ns
clk7	period = 640ns
clk8	period = 1280ns

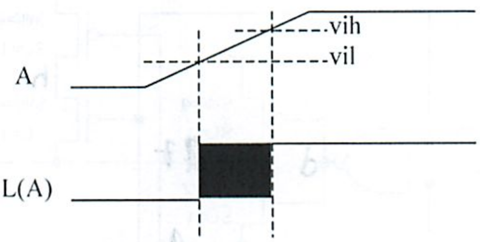
For example, to completely test all possible input combinations for a 2-input gate, you could connect clk1 and clk2 to the two inputs and simulate for 20ns.

4: Plotting Results

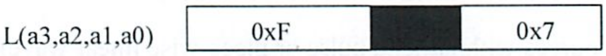
Interpreting analog signal levels as logic values can be tedious. JSim will do it for you automatically if you ask to plot "L(a)" instead of just "a". The logic-high and logic-thresholds are determined by the "vih" and "vil" options:

```
.options vih=2.6 vil=0.6
```

Initial values are specified in "/mit/6.004/jsim/nominal.jsim", but you can respecify them in your own netlist. Voltages between vil and vih are displayed as a filled-in rectangle to indicate that the logic value cannot be determined. For example:



You can also ask for the values of a set of signals to be displayed as a bus, e.g., "L(a3,a2,a1,a0)". The signals should be listed most-significant bit first. A bus waveform is displayed as a filled-rectangle if any of the component signals has an invalid logic level or as a hexadecimal value otherwise. In the following plot the four signals a3, a2, a1 and a0 are interpreted as a 4-bit integer where the high-order bit (a3) is making a 1→0 transition. The filled-in rectangle represents the period of time during which a3 transitions from V_{IH} to V_{IL}.



5: Design Guidelines

Here's a list of design tasks you might use to organize your approach to the lab:

1. Draw a gate-level schematic for the full-adder module. XOR gates can be used to implement the S output; two levels of NAND gates are handy for implementing C_o as a sum of products.

2. Create a MOSFET circuit for each of the logic gates you used in step 1.

3. Enter .subckt definitions in your netlist for each of the logic gates. Use Jsim to test each logic gate with all possible combinations of inputs. Debugging your gate designs one-by-one will be much easier than trying to debug them as part of the adder circuit. Here's a sample netlist for testing a 2-input NAND gate called nand2:

```
.include "/mit/6.004/jsim/nominal.jsim"
.include "/mit/6.004/jsim/8clocks.jsim"
```

```
.subckt nand2 a b z
… internals of nand2 circuit here
.ends
Xtest clk1 clk2 z nand2
.tran 20ns
.plot clk1
.plot clk2
.plot z
```

4. Enter a .subckt definition for the full-adder, building it out of the gates you designed and tested above. Use Jsim to test your design with all 8 possible combinations of the three inputs. At this point you probably want to switch to using "Fast Transient Analysis" do to the simulations as it is much faster than "Device-level Simulation".

5. Enter the netlist for the 4-bit adder and test the circuit using input waveforms supplied by lab2checkoff.jsim. Note that the checkoff circuitry expects your 4-bit adder to have exactly the terminals shown below – the inside circuitry is up to you, but the ".subckt ADDER4..." line in your netlist should match exactly the one shown below.

```
.include "/mit/6.004/jsim/nominal.jsim"
.include "/mit/6.004/jsim/lab2checkoff.jsim"

… subckt definitions of your logic gates

.subckt FA a b ci s co
… full-adder internals here
.ends

.subckt ADDER4 a3 a2 a1 a0 b3 b2 b1 b0 s4 s3 s2 s1 s0
* remember the node named "0" is the ground node
* nodes c0 through c3 are internal to the ADDER module
Xbit0 a0 b0 0 s0 c0 FA
Xbit1 a1 b1 c0 s1 c1 FA
Xbit2 a2 b2 c1 s2 c2 FA
Xbit3 a3 b3 c2 s3 s4 FA
.ends
```

lab2checkoff.jsim contains the necessary circuitry to generate the appropriate input waveforms to test your adder. It includes a .tran statement to run the simulation for the appropriate length of time and a few .plot statements showing the input and output waveforms for your circuit.

When debugging your circuits, you can plot additional waveforms by adding .plot statements to the end of your netlist. For example, to plot the carry-out signal from the first full adder, you could say

```
.plot Xtest.c0
```

where Xtest is the name lab2checkoff.jsim gave to the ADDER4 device it created and c0 is the name of the internal node that connects the carry-out of the low-order FA to the carry-in of the next FA.

6.004 Lab 2

9/28

Build adder

- 2 4-bit unsigned #

Oh researched this this weekend

So build gates

- did we not do this last time?

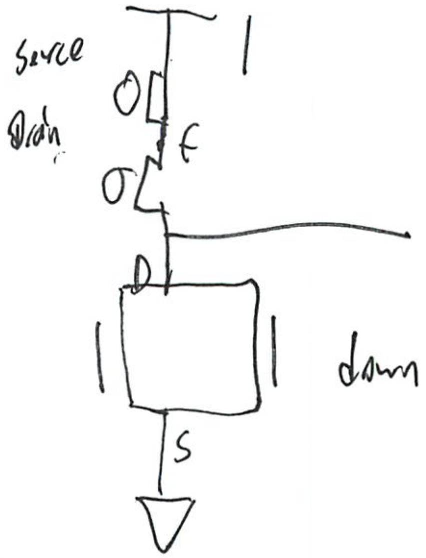
NAND, Inverter

NOR

- try to do self
- or look in lecture notes
- no do self

TT

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



N

~~drain gate bulk source~~ drain gate source bulk

P

Ⓠ

How to test?

Test later

Now build XOR

- they tell us how

Need to convert

- looks complicated

How do you come up w/ these??

Tested inverter ✓

(they gave instructions later

NAND2

	AND	NAND	
00	0	1	✓
01	0	1	✓
10	0	1	✓
11	1	0	✓

✓

NOR2

	OR	NOR	
00	0	1	✓
01	1	0	✓
10	1	0	✓
11	1	0	✓

✓ woot!

6
③

XOR

00	0	✓
01	1	✓
10	1	✓
11	0	✓

✓ woot

XNOR

No - doesnot work!

Oh e = 2

Needed to do Fast analysis?

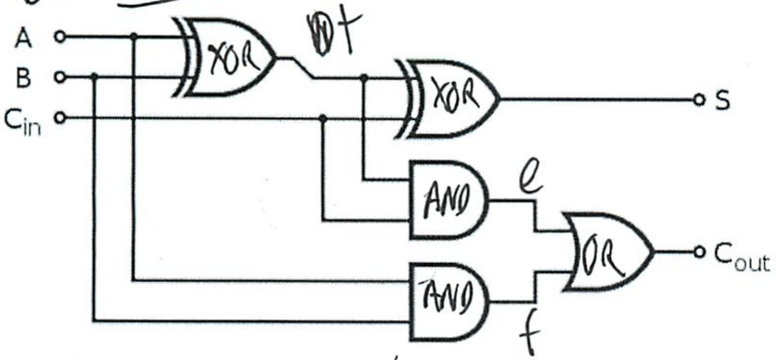
0 0	1	✓
0 1	0	✓
1 0	0	✓
1 1	1	✓

"Invert other XOR output?

✓ works

(I an kinda shortcutting lab :) know my way arround)

4 Do Fast Adder



↑

Can use 3 2-input NANDs and 1 3-in NAND

? or 2 cascaded NANDS?

No limit on # of chips

- So INV

- Need to think about

What is fast way to do

So

	AND	NAND
00	0	1
01	0	1
10	0	1
11	1	0

↑ So can do inverted input

	flip
11	1
10	0
01	0
00	0

is NOR w/ inverted inputs

C → inv → g

D → inv → h

NOR same out

A → inv → j

B → inv → k

~~5~~

Now ~~AND~~ OR

00	0
01	1
10	1
11	1

invert in

11	0
10	1
01	1
00	1

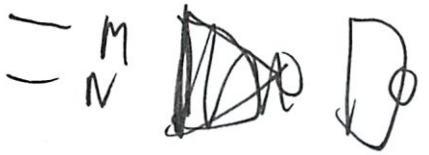
flip

1
1
1
0
NAND

invert all and NAND

e → inv → m
f → inv → n

This will be too many gates
Must be easier way



Ok test that

C	A	B	S	C	
0	0	0	0	0	✓
0	0	1	1	0	✓
0	1	0	1	0	✓
0	1	1	0	1	✓
1	0	0	1	0	✓
1	0	1	0	1	✓
1	1	0	0	1	✓
1	1	1	1	1	✓

✓ Works
- Cool 1st try!
4/24

Now I need to put them together

- - almost forgot

Now read how they are specing it

Oh can have multi letter nodes

They even tell you how ~~they are~~ to do it

Duplicate device name when try to run test

Oh included file twice (duh)

- it should know this

Plot is really unreadable!

But passed verification ✓

Now questions + meeting

- That took 1.5 hrs

∴ Do tomorrow

Lab 2 Checkoff

4/28

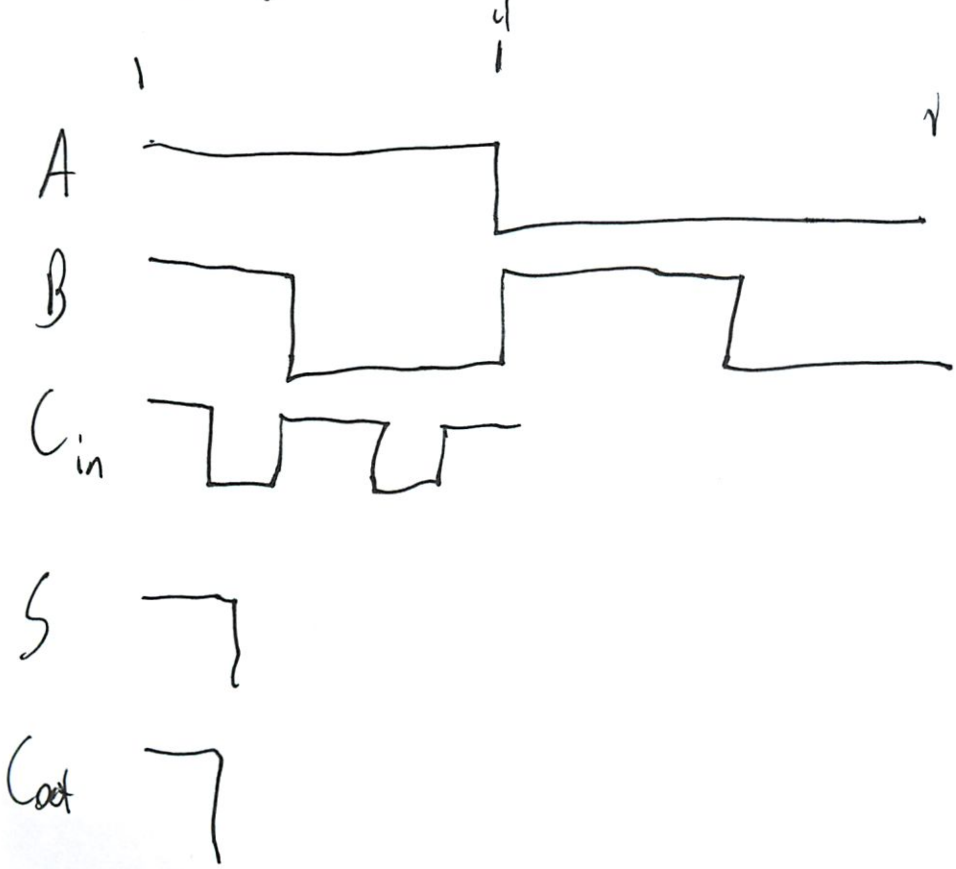
Cascading ANDs into ORs → can use NOR

Can build 3-input NAND

Going to need to optimize

Redo
- fix critical path

Since need to use it later
- lab 6



✓ checked off

6.004 On-line: Questions for Lab 2

When you're done remember to save your work by clicking on the Save'button at the bottom of the page. You can check if your answers are correct by clicking on the Check'button.

When entering numeric values in the answer fields, you can use integers (1000), floating-point numbers (1000.0), scientific notation (1e3), or JSim numeric scale factors (1K).

Problem 1. The following questions are multiple-choice. Using the "check" button, you can of course simply keep guessing until you get the right answer. But you'll be in a much better position to take the quizzes if you take the time to actually figure out the answers.

A. If we set the inputs of a particular CMOS gate to voltages that correspond to valid logic levels, we would expect the *static* power dissipation of the gate to be

non-zero, but very small (picowatts) ✓

B. Measuring a particular CMOS device G, we find 1.5V noise margins. If the *width* of all mosfets inside of G were doubled, we would expect the noise margins of the new gate to

stay about the same ✓

C. To *decrease* the output rise time of a CMOS gate one could

increase the width of all pfets ✓

D. Suppose one wanted to *decrease* the propagation time of a CMOS circuit. Which of the following actions would lead to the greatest possible speed up?

increase the power supply voltage and lower the operating temperature ✓

Problem 2. Almost all of the power dissipated by CMOS circuits goes into charging and discharging nodal capacitances. This power can be computed as **C(V^2)F** where **C** is the capacitance being switched, **V** is the change in voltage, and **F** is the frequency at which the switching happens. In CMOS circuits, nodes are switched between ground (0 volts) and the power supply voltage (VDD volts), so **V** is either +VDD or -VDD and so V^2 is VDD^2.

A. Suppose we have a device implemented in a technology where VDD = 5V. If we have the option of reimplementing the device in a technology where VDD = 3.3V, what sort of speedup (i.e., change in **F**) could be specified for the reimplementation assuming we want to keep the power budget unchanged?

Speedup (eg, 2.0 would be twice as fast): 2.29 ✓

Problem 3. As we saw in Lecture 4, there are 16 possible 2-input combinational logic gates. The cost of implementing these gates varies dramatically, requiring somewhere between 0 and 10 mosfets depending on the gate. For example, it takes 2 mosfets to implement "F = NOT A", but 4 mosfets (organized as two inverters) to implement "F = A".

For each of the 2-input gates whose Karnaugh maps are given below, indicate the minimum number of mosfets required to implement the gate. You should only consider static fully-complementary circuits like those shown in lecture; these implementations meet the following criteria:

- no static power dissipation
- Vol = 0V, Voh = power supply voltage
- Nfets appear only in pulldown paths, Pfets appear only in pullup paths

- the pullup and pulldown are complementary, i.e., when one path is "on", the other is "off"
- the pullup and pulldown circuits can be decomposed into series and parallel connections of mosfets
- all gate implementations restore incoming logic levels (so a wire connecting an input terminal to an output terminal would not be a legal gate implementation)

A.

NOR		A	
		0	1
B	0	1	0
	1	0	0

So A=1, B varies so $\bar{A}$ since A supposed to be 1

$\bar{B}$ So $\bar{A}+\bar{B}+\bar{A}\bar{B}$? but the 1?

Number of mosfets needed to implement "NOR" : 4

B.

AND		A	
		0	1
B	0	0	0
	1	0	1

NAND = 4 + inverter = 5 ⊗

No need to invert ins Not out 6 ✓

Number of mosfets needed to implement "AND" : ~~5~~ 6

C.

XNOR		A	
		0	1
B	0	1	0
	1	0	1

See p-set

Number of mosfets needed to implement "XNOR" : 6 + NAND 4 ⊗

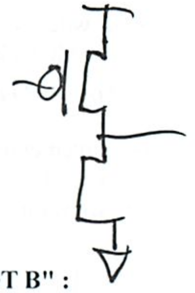
D.

NOT B		A	
		0	1
B	0	1	1
	1	0	0

So don't care A

But how to check if B true

2



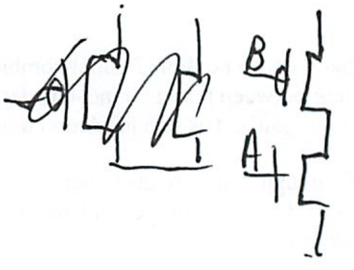
Number of mosfets needed to implement "NOT B" :

E.

A > B		A	
		0	1
B	0	0	1
	1	0	0

How would I build?

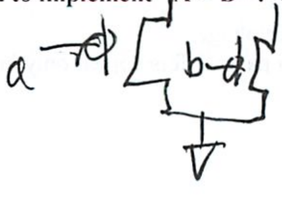
← just target this?



No burn up - not balanced

what I thought

Number of mosfets needed to implement "A > B" :



Check Save

6.004 Lab 2 Qu

9/29

a) CMOS valid inputs

Static power → 0 or near 0 ✓

b) doubling width

noise margins

- no clue   Stay the same ✓
- its speed

c) to ↓ output rise time

double w

but for which

0 → 1 so pullup ✓

p-fet

↑ → ↓   remember "np"

d) ↓ prop time

- oh w/ power supply v + opp temp

↑ want ~~↓~~   ↑ no clue

↑ power supply V   ↓ op temp

②

2. Almost all power goes to charging + discharging nodal capacitance

$C(V^2)F$ ↑ Frequency switching shall

↑ Capitence

↑ Change in Voltage

VDD-0

So have in VDD= 5V

If redo in VDD = 3.3V what speedup (change in F) cald be specified assuming want to keep power budget unchanged

↑ I don't get this

Oh $P = C(V^2)F$

Capaticance same ?

So basically

$5^2 \, 1 = 3.3^2 F$

↑ assign vallue

So $\frac{5^2}{3.3^2} = 2.29$ ✓

③

16 possible 2-input combo logic gates

Cost varries dramatically

0 -> 10

Karnaugh maps

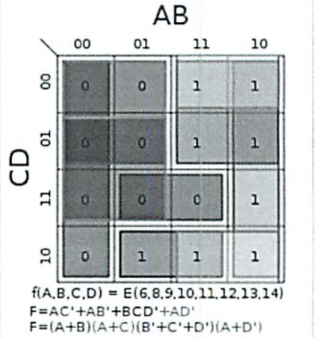
- indicate min # of mosfets

(need more practice with)

- Static, fulll complementry
- V_{ol} = 0V V_{oh} = V_{DD}
- Nfets pulldown
- P up
- Complementry - when one off other is on
- ~~Seller~~ decompose into series + parallel connection mosfets
- restore voltage

Karnaugh map

From Wikipedia, the free encyclopedia


AB
00
01
11
10
CD
00
01
11
10
0
0
1
1
0
0
1
1
0
0
0
1
0
1
1
1
f(A,B,C,D) = E(6,8,9,10,11,12,13,14)
F=AC'+AB'+BCD'+AD'
F=(A+B)(A+C)(B'+C'+D')(A+D')

An example Karnaugh map

The **Karnaugh map** (**K-map** for short), Maurice Karnaugh's 1953 refinement of Edward Veitch's 1952 **Veitch diagram**, is a method to simplify Boolean algebra expressions. The Karnaugh map reduces the need for extensive calculations by taking advantage of humans' pattern-recognition capability, permitting the rapid identification and elimination of potential race conditions. ← ?

In a Karnaugh map the boolean variables are transferred (generally from a truth table) and ordered according to the principles of Gray code in which only one variable changes in between adjacent squares. Once the table is generated and the output possibilities are transcribed, the data is arranged into the largest possible groups containing 2^n cells (n=0,1,2,3...)[1] and the minterm is generated through the axiom laws of boolean algebra.

Contents

Race condition
-output is unexpectily/critically dependent on sequence and timing of events
-logic circuits
-multithreaded programs

Example

Karnaugh maps are used to facilitate the simplification of Boolean algebra functions. The following is an unsimplified Boolean Algebra function with Boolean variables *A*, *B*, *C*, *D*, and their inverses. They can be represented in two different notations:

- $f(A,B,C,D) = \sum(6,8,9,10,11,12,13,14)$ Note: The values inside $\sum$ are the minterms to map (i.e. rows which have output 1 in the truth table). ↑ oh shortened notation I have not yet seen
- $f(A,B,C,D) = (\overline{A}BC\overline{D})+(A\overline{B}\,\overline{C}\,\overline{D})+(A\overline{B}\,\overline{C}D)+(A\overline{B}C\overline{D})+(A\overline{B}CD)+(AB\overline{C}\,\overline{D})+(AB\overline{C}D)+(ABC\overline{D})$

the brain dead way to do

Truth table

Using the defined minterms, the truth table can be created:

#	*A*	*B*	*C*	*D*	*f*(*A*,*B*,*C*,*D*)
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0

) can build

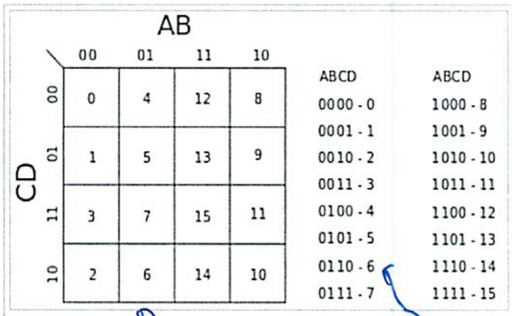
5	0	1	0	1	0
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	0

Karnaugh map

for 4×4

The input variables can be combined in 16 different ways, so the Karnaugh map has 16 positions, and therefore is arranged in a 4 × 4 grid.

The binary digits in the map represent the function's output for any given combination of inputs. So 0 is written in the upper leftmost corner of the map because $f = 0$ when $A = 0$, $B = 0$, $C = 0$, $D = 0$. Similarly we mark the bottom right corner as 1 because $A = 1$, $B = 0$, $C = 1$, $D = 0$ gives $f = 1$. Note that the values are ordered in a Gray code, so that precisely one variable changes between any pair of adjacent cells.



CD \ AB	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

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

K-map construction.

could also say 0110

After the Karnaugh map has been constructed the next task is to find the minimal terms to use in the final expression. These terms are found by encircling groups of 1s in the map. The groups must be rectangular and must have an area that is a power of two (i.e. 1, 2, 4, 8...). The rectangles should be as large as possible without containing any 0s. The optimal groupings in this map are marked by the green, red and blue lines. Note that groups may overlap. In this example, the red and green groups overlap. The red group is a 2 × 2 square, the green group is a 4 × 1 rectangle, and the overlap area is indicated in brown.

The grid is toroidally connected, which means that the rectangular groups can wrap around edges, so $A\overline{D}$ is a valid term, although not part of the minimal set—this covers Minterms 8, 10, 12, and 14.

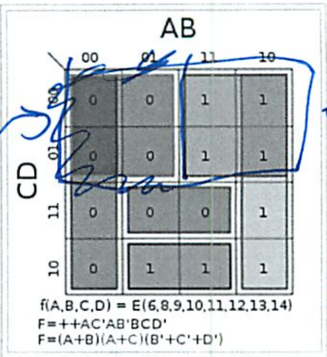
Perhaps the hardest-to-visualize wrap-around term is $\overline{B}\,\overline{D}$ which covers the four corners—this covers minterms 0, 2, 8, 10.

but where are doing this -individual bits

Solution

Once the Karnaugh Map has been constructed and the groups derived, the solution can be found by eliminating extra variables within groups using the axioms of boolean algebra. It can be implied that rather than eliminating the variables that change within a grouping, the minimal function can be derived by noting which variables stay the same.

by which values??



K-map showing minterms and boxes covering the desired minterms. The brown region is an overlapping of the red (square) and green regions.

← circle all the same

For the Red grouping:

- The variable A maintains the same state (1) in the whole encircling, therefore it should be included in the term for the red encircling.
- Variable B does not maintain the same state (it shifts from 1 to 0), and should therefore be excluded.
- C does not change: it is always 0. Because C is 0, it has to be negated before it is included (thus, $\overline{C}$).
- D changes, so it is excluded as well.

d always same

Thus the first term in the Boolean sum-of-products expression is $\boxed{A\overline{C}.}$

For the Green grouping we see that A and B maintain the same state, but C and D change. B is 0 and has to be negated before it can be included. Thus the second term is $A\overline{B}$.

In the same way, the Blue grouping gives the term $BC\overline{D}$.

what is always true inside box

The solutions of each grouping are combined into: $A\overline{C} + A\overline{B} + BC\overline{D}$.

add as ORs

Inverse

The inverse of a function is solved in the same way by grouping the 0s instead.

The three terms to cover the inverse are all shown with grey boxes with different colored borders:

can also look at the 0s

- brown—$\overline{A}\,\overline{B}$
- gold—$\overline{A}\,\overline{C}$
- blue—BCD

This yields the inverse:

$$\overline{F} = \overline{A}\,\overline{B} + \overline{A}\,\overline{C} + BCD$$

Through the use of De Morgan's laws, the product of sums can be determined:

$$\overline{\overline{F}} = \overline{\overline{A}\,\overline{B} + \overline{A}\,\overline{C} + BCD}$$
$$F = (A + B)(A + C)\left(\overline{B} + \overline{C} + \overline{D}\right)$$

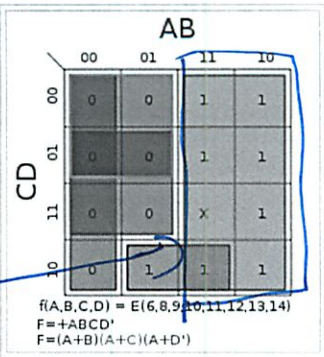
) prove that same

Don't cares

↓If TT includes

Karnaugh maps also allow easy minimizations of functions whose truth tables include "don't care" conditions (that is, sets of inputs for which the designer doesn't care what the output is) because "don't care" conditions can be included in a circled group in an effort to make it larger. They are usually indicated on the map with a dash or X.

i.So can include as 1 or 0



The minterm 15 is dropped and replaced as a don't care, this removes the green term completely but restricts the blue inverse term

The example to the right is the same above example but with minterm 15 dropped and replaced as a don't care. This allows the red term to expand all the way down and, thus, removes the green term completely.

This yields the new minimum equation:

$$F = \underbrace{A}_{\text{A is always 1}} + BC\overline{D}$$

- other values don't matter

Note that the first term is just A not $A\overline{C}$. In this case, the don't care has dropped a term (the green); simplified another (the red); and removed the race hazard (the yellow as shown in a following section).

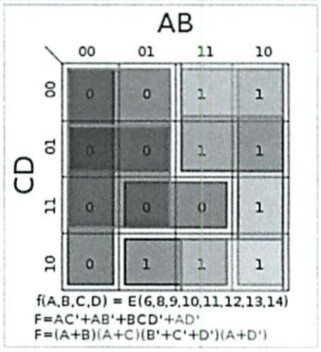
Also, since the inverse case no longer has to cover minterm 15, minterm 7 can be covered with $\left(\overline{A}D\right)$ rather than (BCD) with similar gains.

Race hazards

Elimination

Karnaugh maps are useful for detecting and eliminating race hazards. Race hazards are very easy to spot using a Karnaugh map, because a race condition may exist when moving between any pair of adjacent, but disjointed, regions circled on the map.

- In the example to the right, a potential race condition exists when C is 1 and D is 0, A is 1, and B changes from 1 to 0 (moving from the blue state to the green state). For this case, the output is defined to remain unchanged at 1, but because this transition is not covered by a specific term in the equation, a potential for a *glitch* (a momentary transition of the output to 0) exists.
- There is a second potential glitch in the same example that is more difficult to spot: when D is 0 and A and B are both 1, with C changing from 1 to 0 (moving from the blue state to the red state). In this case the glitch wraps around from the top of the map to the bottom.



Above k-map with the $A\overline{D}$ term added to avoid race hazards

Whether these glitches will actually occur depends on the physical nature of the implementation, and whether we need to worry about it depends on the application.

In this case, an additional term of $A\overline{D}$ would eliminate the potential race hazard, bridging between the green and blue output states or blue and red output states: this is shown as the yellow region.

The term is redundant in terms of the static logic of the system, but such redundant, or consensus terms, are often needed to assure race-free dynamic performance.

Similarly, an additional term of $\overline{A}D$ must be added to the inverse to eliminate another potential race hazard. Applying De Morgan's laws creates another product of sums expression for F, but with a new factor of $\left(A + \overline{D}\right)$.

2-variable map examples

The following are all the possible 2-variable, 2 × 2 Karnaugh maps. Listed with each is the minterms as a function of $\sum()$ and the race hazard free (*see previous section*) minimum equation.

I still don't get

Hard to see B+W

See recitation

k-Map videos

4

9/20
Home

Very confused still

A) $\overline{A} + \overline{B} + \overline{AB}$

↑ since is 1 but is 0

So think do seperate for 1 and 0

1 = $\overline{A}\,\overline{B}$ ← is it not since these are always (

so when A = 0 and B = 0

0 = $\overline{A} + \overline{B}$

Now how to convert to do # mosfets

I know ans here is 4

but did not use k-map

(Pratice paper)

Or this is not a k-map av!

- it never said
- just read from past answers

Practice w/ kmaps

9/29

Problem 3. In the Karnaugh maps below the use of "X" in a cell indicates a "don't care" situation where the value of the function for those inputs can be chosen to minimize the size of the overall expression.

~ $A\bar{C} + AD + \bar{A}D + \bar{A}B\bar{C}\bar{D}$

? individual

? So just cover everything once??

They do $D + A \cdot \bar{C} + B\bar{C}$

? did know was legal ← smaller, less gates

but how many?

? Need to draw them

but my A answer was far less than rational

And that ANDs not OR

Is this Connected?

CD \ AB	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	0	0	0	0

A

	0	0	0	1
C	1	1	0	0

B

(a)

A

	1	0	1	1	
	1	0	0	0	D
C	0	0	X	0	
C	1	1	X	1	

B

(b)

A

	1	0	1	1
C	1	0	0	0

B

(c)

A

	1	0	1	1	
	1	0	0	0	D
C	1	0	0	0	
C	1	1	0	1	

B

(d)

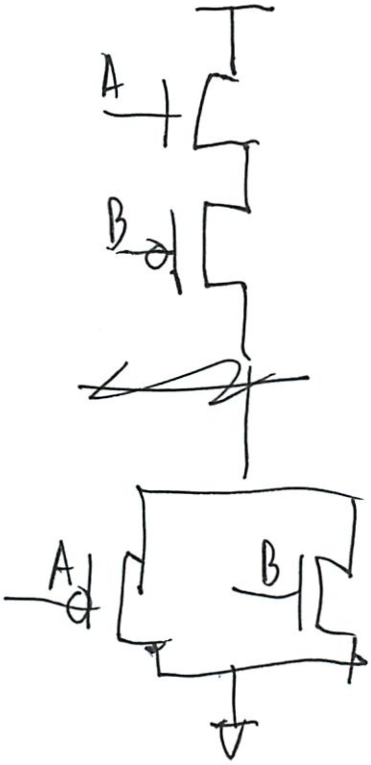
A. Circle the prime implicants in the Karnaugh maps and write a minimal sum-of-products expression for each of the maps.

B 5

E How to build A > B

Think about it sam more

Its really $A\bar{B}$
↑ when A is 1 and B is 0



← just did this

← but bottom wrong?

← so this is open if
A is low or
B is high

		A	
		0	1
B	0	0	1
	1	0	0

which is what we want

So good

⊗ Not Y

Its 6

✓ Done QV

Rebuild 6.004 adder

9/28
Home

Redoing Cout

- with 3 2 inputs NANDs
1 3 input NAND

? build first

4 instead of 2

Now Cout

A	B	C_{in}	Cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

②

Now how to build?

Can do stupid way

$$F = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

Or k map - 1st time setting up

C \ A B	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$F = AB + BC + AC$$

↑ so this is 3 AND, 1 OR

De morgan

$$\bar{F} = (\bar{A} + \bar{B}) \cdot (\bar{B} + \bar{C}) \cdot (\bar{A} + \bar{C})$$

↑ but now have ORS

Or shorter demorgan

$$\bar{A} \cdot \bar{B} = \overline{A + B}$$

③ Or perhaps negitive logic

$$\overline{F} = AC + AB + \overline{A}BC$$

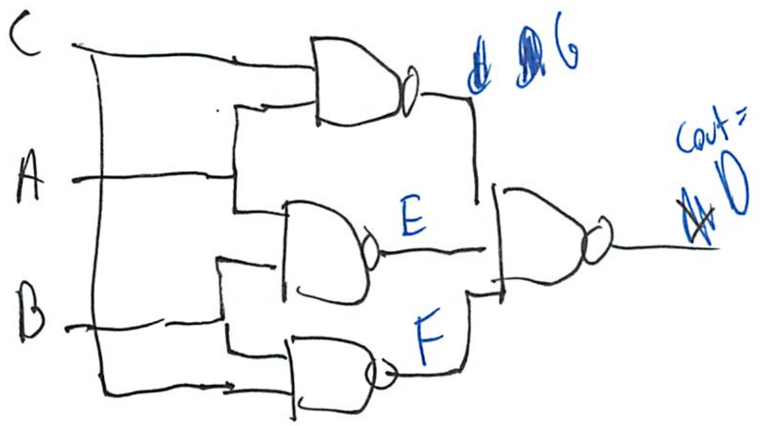
¿ but ~~that~~ does that help?

How to I get it w/ NANDs

¿ Guess?

NAND $= \overline{AB} = \overline{A} + \overline{B}$

So Actually there is a gate exaple in lecture notes



But how would we have known that

If look at $\overline{A} + \overline{B}$ That is NAND

So have 3 Then $\overline{F} = O_1 \cdot O_2 \cdot O_3$

Demorgan

So guess could have seen $F = \overline{O_1} + \overline{O_2} + \overline{O_3}$ ← a 3 NAND

(4) But is $(\bar{A}+\bar{B}) = O_1$ or $\bar{O}_1$

Well is NAND so I guess $\bar{O}_1$

But did not know that

Now I do

NAND "inverts" than inverts back

↑ is actually prob pretty bad

$$\bar{A}+\bar{B} = \overline{AB}$$

↑ Yeah inverse of AND

So try w/ TT

AND		Invert out	invert ins	Run OR on
00	0	1	11	1
01	0	1	10	1
10	0	1	01	1
11	1	0	00	0

↑ This is what I got wrong befor

✓ Correct

So could take my inital sol w/ inverters and simplify

2nd inverter converts back

⑤ New actually implement

ⓧ Wrong
- no longer pasts tests

Test parts

Fixed something

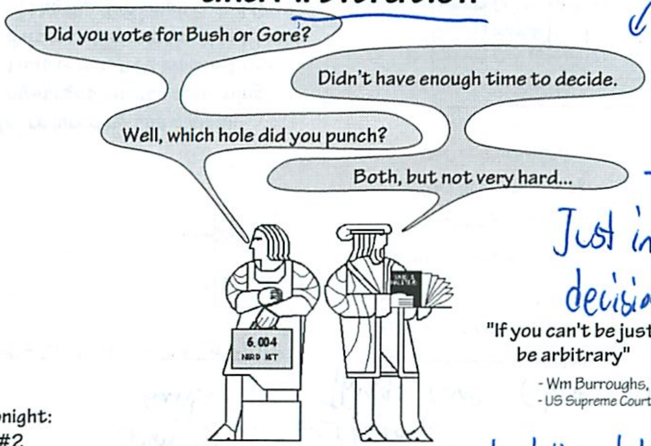
even worse

confusing variable names

(✓) Fixed - confused variable letters

So apparently faster today

Synchronization, Metastability and Arbitration


Did you vote for Bush or Gore?
Didn't have enough time to decide.
Well, which hole did you punch?
Both, but not very hard...
6.004 NERD KIT

"If you can't be just, be arbitrary"

- Wm Burroughs, *Naked Lunch*
- US Supreme Court 12/00

Due tonight:
❑Lab #2
❑Lab #1 checkoff meeting

failure to make decisions in timely manner

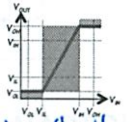
Just in time decisions

metastable state

The Importance of being Discrete

We avoid possible errors by disciplines that avoid asking the tough questions – using a *forbidden zone* in both voltage and time dimensions:

Digital Values:
Problem: Distinguishing voltages representing "1" from "0"
Solution: *Forbidden Zone:* avoid using similar voltages for "1" and "0"

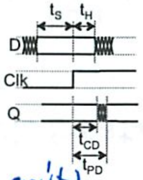


Static displine

Digital Time:
Problem: "Which transition happened first?" questions
Solution: Dynamic Discipline: avoid asking such questions in close races

albitraily close in time

wait for that time

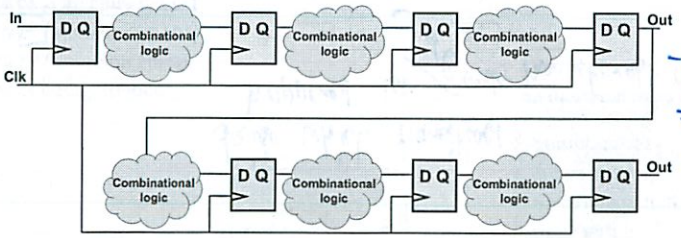


avoid arrit by making time forbidden zone
– with window thold time

If we follow these simple rules…

can build abitrally complex systems

Can we guarantee that our system will always work?



– meet static + dynamic disipline

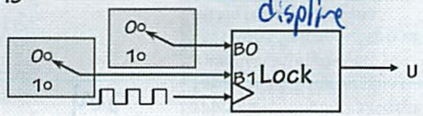
With careful design we can make sure that the dynamic discipline is obeyed everywhere*…

* well, *almost* everywhere… – not the inputs

The world doesn't run on our clock!

What if each button input is an asynchronous 0/1 level?

violate dynamic displine

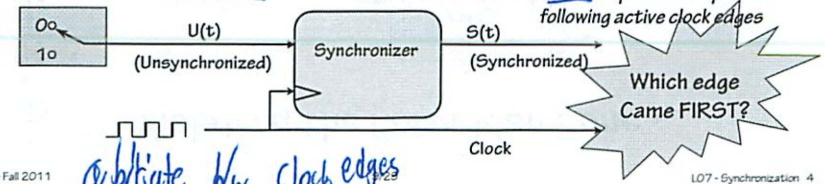


Cause transitions at active clock edges

To build a system with asynchronous inputs, we have to break the rules: *we cannot guarantee that setup and hold time requirements are met at the inputs!*

So, lets use a "synchronizer" at each input:

copy of signal – but follows clock



Valid except for brief periods following active clock edges

arbitriate b/w clock edges

9/29

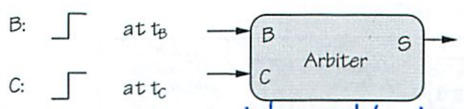
The Asynchronous Arbiter:

a classic problem

UNSOLVABLE

e clasic formal problen

define reasonably - so can build



For NO finite values of t_E and t_D is this spec realizable, even with reliable components!

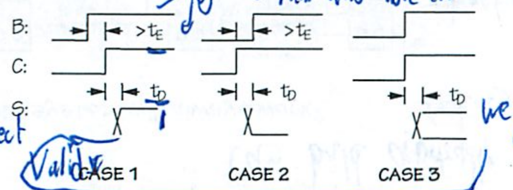
decide which transition happened 1st

Arbiter specifications:

- finite t_D (decision time)
- finite t_E (allowable error)
- value of S at time t_C+t_D:
  - 1 if $t_B < t_C - t_E$
  - 0 if $t_B > t_C + t_E$
  - 0, 1 otherwise

not perfect

greater than allowable error



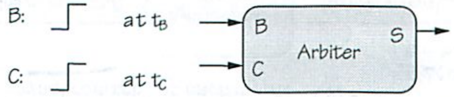
1 if B 1st

0 if C 1st

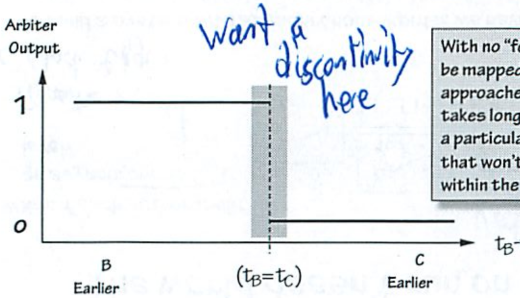
T's actually unsolvably even in theory

Violating the Forbidden Zone

Trying to map contiuous variable to discrete variable



Issue: Mapping the *continuous* variable $(t_B - t_C)$ onto the *discrete* variable S in bounded time.



With no "forbidden zone," all inputs have to be mapped to a valid output. As the input approaches discontinuities in the mapping, it takes longer to determine the answer. Given a particular time bound, you can find an input that won't be mapped to a valid output within the allotted time.

if had a knob that outs to 0,1 - where to transition?

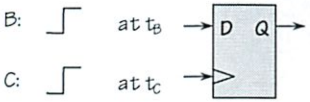
Unsolvable?

Community said:

that can't be true...

Things come close to solving

Lets just use a D Flip Flop:



DECISION TIME is T_{PD} of flop.

ALLOWABLE ERROR is max(t_{SETUP}, t_{HOLD})

Our logic:

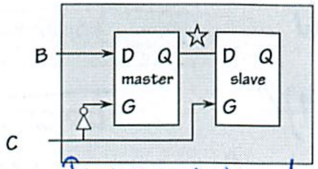
T_{PD} after T_C, we'll have

$Q=0$ iff $t_B + t_{SETUP} < t_C$

$Q=1$ iff $t_C + t_{HOLD} < t_B$

$Q=0$ or 1 otherwise.

We're lured by the digital abstraction into assuming that Q must be either 1 or 0. But lets look at the input latch in the flip flop when B and C change at about the same time...

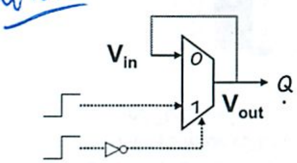


but look inside box

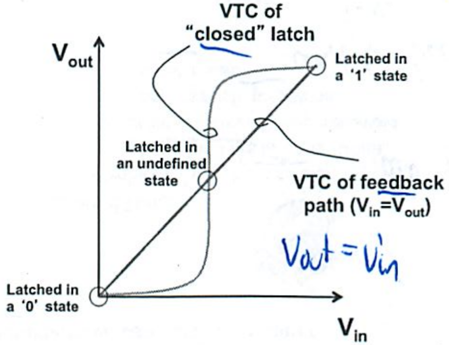
if timings close - violate dd w/ ~~latch~~ master latch

The Mysterious Metastable State

Latch



2 simltaneous equations



Recall that the latch output is the solution to two simultaneous constraints:

1. The VTC of path thru MUX; and
2. $V_{in} = V_{out}$

In addition to our expected stable solutions, we find an unstable equilibrium in the forbidden zone called the "Metastable State"

2 stable solutions

1 unstable

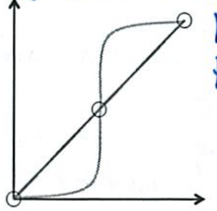
- but can't bound time it will take

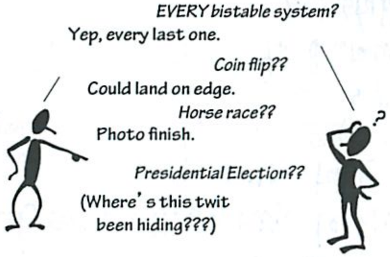
[illegible]

Metastable State: Properties

the Closer we are → the longer it takes

1. It corresponds to an *invalid* logic level – the switching threshold of the device.
2. Its an *unstable* equilibrium; a small perturbation will cause it to accelerate toward a stable 0 or 1.
3. It will settle to a valid 0 or 1... eventually.
4. BUT – depending on how close it is to the $V_{in}=V_{out}$ "fixed point" of the device – it may take arbitrarily long to settle out.
5. EVERY bistable system exhibits at least one metastable state!





real world flip flop

Observed Behavior:
typical metastable symptoms

Can observe ~~[illegible]~~

Following a clock edge on an asynchronous input:

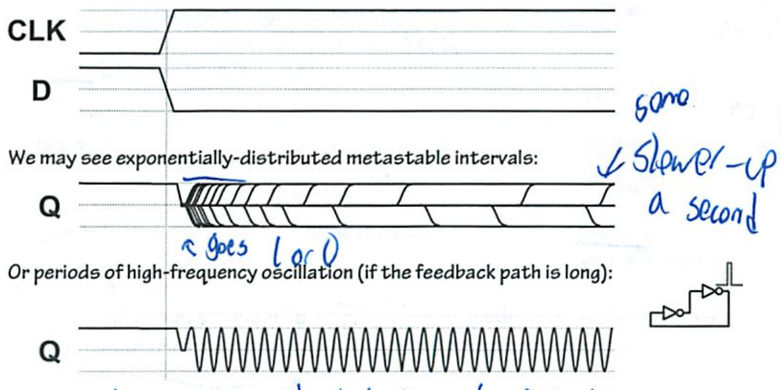
since violating d.d.

We may see exponentially-distributed metastable intervals:

↑ goes 1 or 0

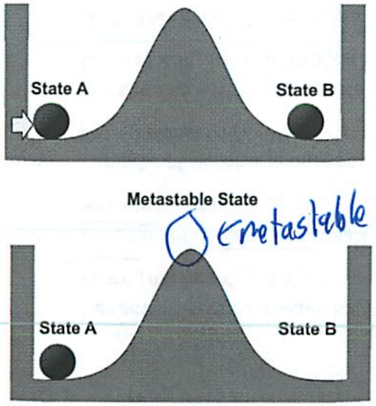
some ↓ slower – up to a second

Or periods of high-frequency oscillation (if the feedback path is long):



much more complicated dynamic behavior

Mechanical Metastability



← metastable state

If we launch a ball up a hill we expect one of 3 possible outcomes:

a) Goes over
b) Rolls back
c) Stalls at the apex

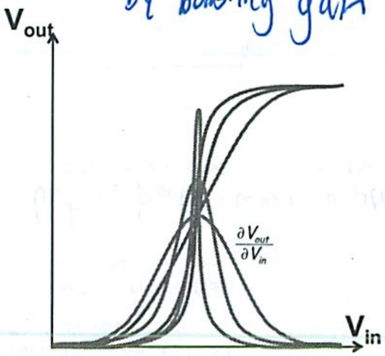
That last outcome is not stable.
- a gust of wind
- Brownian motion
- it doesn't take much

– arbitrary length time
– can't bound

↑ animation

How do balls relate to digital logic?

make hill steeper by boosting gain



Our hill is analogous to the *derivative* of the VTC (Voltage Transfer Curve)... at the metastable point, the derivative (slope) is ZERO.

Notice that the higher the gain thru the transition region, the steeper the peak of the hill... making it harder to get into a metastable state...

We can decrease the probability of getting into the metastable state, but – assuming continuous models of physics – we can't eliminate the slope=0 point!

The Metastable State:

Why is it an inevitable risk of synchronization?

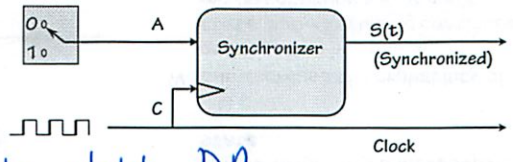
- Our active devices always have a fixed-point voltage, V_M, such that $V_{IN}=V_M$ implies $V_{OUT} = V_M$
- Violation of dynamic discipline puts our feedback loop at some voltage V_O near V_M
- The rate at which V progresses toward a stable "0" or "1" value is proportional to $(V - V_M)$
- The time to settle to a stable value depends on $(V_O - V_M)$; its theoretically infinite for $V_O = V_M$
- Since there' s no lower bound on $(V_O - V_M)$, there's no upper bound on the settling time.
- Noise, uncertainty complicate analysis (but don't help).

Closer we are to metastable is proportional to distance from it

Sketch of analysis... I.

Assume asynchronous 0->1 at T_A, clock period CP:

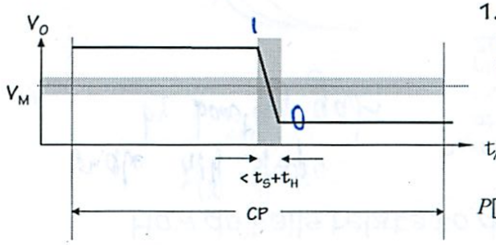
Whats the FF output voltage, V_O, immediately after T_A?



Get in trouble when violate D.D.

Potential trouble comes when V_O is near the metastable point, V_M...

Setup+hold time = another parameter



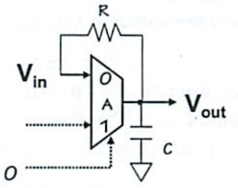
1. Whats the probability that the voltage, V_O, immediately after TA is within ε of V_M?

metastable

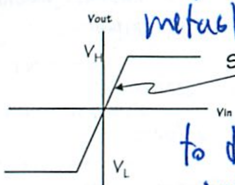
$$P[|V_O - V_M| \le \varepsilon] \le \frac{(t_S + t_H)}{CP} * \frac{2\varepsilon}{(V_H - V_L)}$$

Sketch of analysis... II.

rate at which voltage leaves metastable voltage is proportional to distance from metastable V.



We can model our combinational cycle as an amplifier with gain A and saturation at V_H, V_L



2. For V_{out} near V_M, $V_{out}(t)$ is an exponential whose time constant reflects RC/A:

$$V_{out}(t) - V_M \cong \varepsilon e^{t(A-1)/RC} \cong \varepsilon e^{t/\tau}$$

So this is exponential

3. Given interval T, we can compute how small $\varepsilon = |V_O - V_M|$ must be for output to still be invalid after T seconds: still invalid

$$\varepsilon(T) \cong (V_H - V_M) e^{-T/\tau}$$

4. Probability of metastability after T is computed by probability of a V_O yielding ε (T) ...

$$P_M(T) \cong P[|V_O - V_M| < \varepsilon(T)] \cong K e^{-T/\tau}$$

relationship time we wait and prob flip flop is still invalid

Failure Probabilities vs Delay

Making conservative assumptions about the distribution of V_O and system time constants, and assuming a 100 MHz clock frequency, we get results like the following:

Delay	P(Metastable)	Average time between failures
31 ns	3×10^{-16}	1 year
33.2 ns	3×10^{-17}	10 years
100 ns	10^{-45}	10^{30} years!

So low probability

[For comparision:
Age of oldest hominid fossil: 5×10^{6} years
Age of earth: 5×10^{9} years]

So can have practical answers

Lesson: Allowing a bit of settling time is an easy way to avoid metastable states in practice!

but lots of ~~f~~ flip flops

The Metastable State:

a brief history

Antiquity: Early recognition — Buriden's Ass, and other fables…

people tried to brush it aside

Denial: Early 70s — Widespread disbelief. Early analyses documenting inevitability of problem rejected by skeptical journal editors.

Folk Cures: 70s-80s — Popular pastime: Concoct a "Cure" for the problem of "synchronization failure". Commercial synchronizer products.

would fail every once a while

hard to prove did not work

Reconciliation: 80s-90s — Acceptance of the reality: synchronization takes time. Interesting special case solutions.

learned to cope with it

Ancient Metastability

Metastability is the occurrence of a persistent invalid output… an unstable equilibria.

The idea of Metastability is not new:

The Paradox of Buridan's Ass

Buridan, Jean (1300-58), French Scholastic philosopher, who held a theory of determinism, contending that the will must choose the greater good. Born in Bethune, he was educated at the University of Paris, where he studied with the English Scholastic philosopher William of Ockham (whom you might recall from his razor business). After his studies were completed, he was appointed professor of philosophy, and later rector, at the same university. Buridan is traditionally, but probably incorrectly, associated with a philosophical dilemma of moral choice called "Buridan's ass."

In the problem an ass starves to death between two alluring bundles of hay because it does not have the will to decide which one to eat.

Can put donkey in meta stable state

- in middle of 2 piles of food

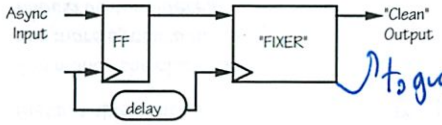
-can't make up mind

Folk Cures

the "perpetual motion machine" of digital logic

entertaining reading

Bad Idea # 1: Detect metastable state & Fix

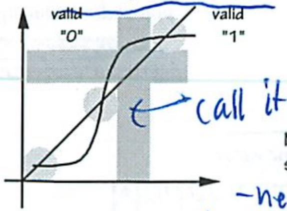


to guarantee

Bug: detecting metastability is itself subject to metastable states, i.e., the "fixer" will fail to resolve the problem in bounded time.

Can't build a fixer that has metastable problem too

Bad Idea #2: Define the problem away by making metastable point a valid output



call it a valid 0

Bug: the memory element will flip some valid "0" inputs to "1" after a while.

Many other bad ideas – involving noise injection, strange analog circuitry, … have been proposed.

-need to explain this

-will arbitrarily jump from 0 to 1!

There's no easy solution

… so, embrace the confusion.

"Metastable States":

try to avoid

- Inescapable consequence of bistable systems
- Eventually a metastable state will resolve itself to valid binary level.
- However, the recovery time is UNBOUNDED … but influenced by parameters (gain, noise, etc)
- Probability of a metastable state falls off EXPONENTIALLY with time -- modest delay after state change can make it very unlikely.

Our STRATEGY; since we can't eliminate metastability, we will do the best we can to keep it from contaminating our designs

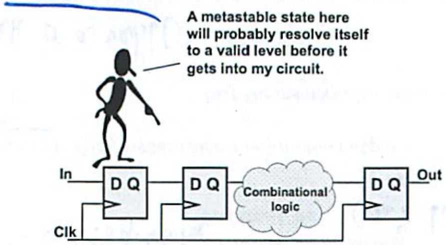
Can build enough delay time

Modern Reconciliation:

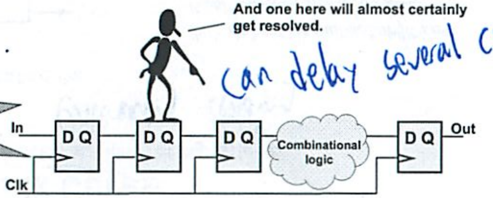
delay buys reliability

Synchronizers, extra flip flops between the asynchronous input and your logic, are the best insurance against metastable states.

The higher the clock rate, the more synchronizers should be considered.



A metastable state here will probably resolve itself to a valid level before it gets into my circuit.



And one here will almost certainly get resolved.

can delay several cycles

SETTLING TIME Cures Metastability!

can be on 6.004 quizzes

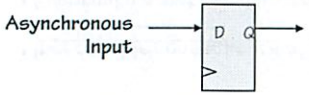
Things we CAN'T build

1. Bounded-time Asynchronous Arbiter:

B
C
Arbiter
S

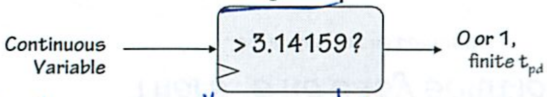
S=0 iff B edge first, 1 iff C edge first, 1 or 0 if nearly coincident
S valid after t_{pd} following (either) edge

2. Bounded-time Synchronizer:



Output = D at active clock edge, either 1 or 0 iff D invalid near clock edge
Q valid after t_{pd} following active clock edge

3. Bounded-time Analog Comparator:



Can't map continuous variable to discrete (in bounded time)

Some things we CAN build

1. Unbounded-time Asynchronous Arbiter:

B
C
Arbiter
S
Done

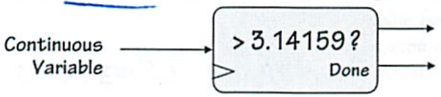
S valid when Done=1; unbounded time.

S=0 iff B edge first, 1 iff C edge first, 1 or 0 if nearly coincident

↑ when made up mind

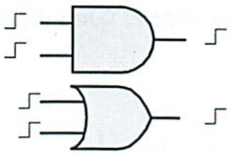
can take arbitrary amt of time

2. Unbounded-time Analog Comparator:



After arbitrary interval, decides whether input at time of last active clock edge was above/below threshold.

3. Bounded-time combinational logic:



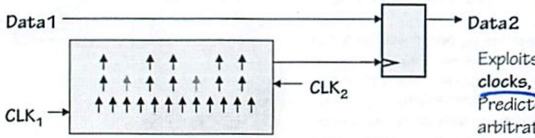
Produce an output transition within a fixed propagation delay of first (or second) transition on the input.

Not making which care lot

Not on quiz

Interesting Special Case Hacks

Predictive periodic synchronization:

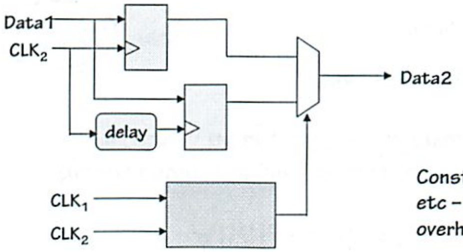


if each input has predictable clocks

he worked on

Exploits fact that, given 2 periodic clocks, "close calls" are predictable. Predicts, and solves in advance, arbitration problems (thus eliminating cost of delay)

Mesochronous communication:

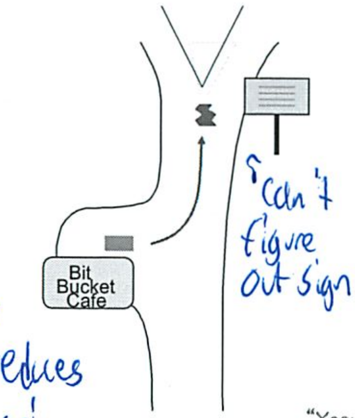


For systems with unsynchronized clocks of same nominal frequency. Data goes to two flops clocked a half period apart; one output is bound to be "clean". An observer circuit monitors the slowly-varying phase relationship between the clocks, and selects the clean output via a lenient MUX.

Constraints on clock timing – periodicity, etc – can often be used to "hide" time overhead associated with synchronization.

Use other timing info to fix

Every-day Metastability - I



can't figure out sign

beer reduces ones gain

Ben Bitdiddle tries the famous "6.004 defense":

Ben leaves the Bit Bucket Café and approaches fork in the road. He hits the barrier in the middle of the fork, later explaining "I can't be expected to decide which fork to take in bounded time!".

Is the accident Ben's fault?

"Yes; he should have stopped until his decision was made."

Judge R. B. Trator, MIT '86

wait for delay

Every-day Metastability - II

GIVEN:

- Normal traffic light:
- GREEN, YELLOW, RED sequence
- 55 MPH Speed Limit
- Sufficiently long YELLOW, GREEN periods
- Analog POSITION input
- digital RED, YELLOW, GREEN inputs
- digital GO output

Can one reliably obey....

- LAW #1: DON'T CROSS LINE while light is RED.

  GO = GREEN

Can you run a red light

- LAW #2: DON'T BE IN INTERSECTION while light is RED.

PLAUSIBLE STRATEGIES:

A. Move at 55. At calculated distance D from light, sample color (using an unbounded-time synchronizer). GO ONLY WHEN stable GREEN.

B. Stop 1 foot before intersection. On positive GREEN transition, gun it.

Summary

The most difficult decisions are those that matter the least.

As a system designer...

Avoid the problem altogether, where possible

- Use single clock, obey dynamic discipline
- Avoid state. Combinational logic has no metastable states!

Delay after sampling asynchronous inputs: a fundamental cost of synchronization

Sometimes, I just sit in my maze -- motionless-- for a very long time.

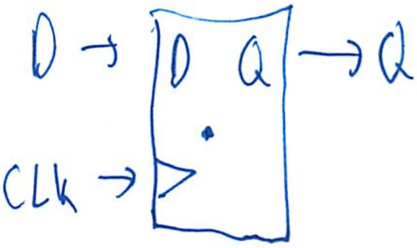
6.004 Recitation

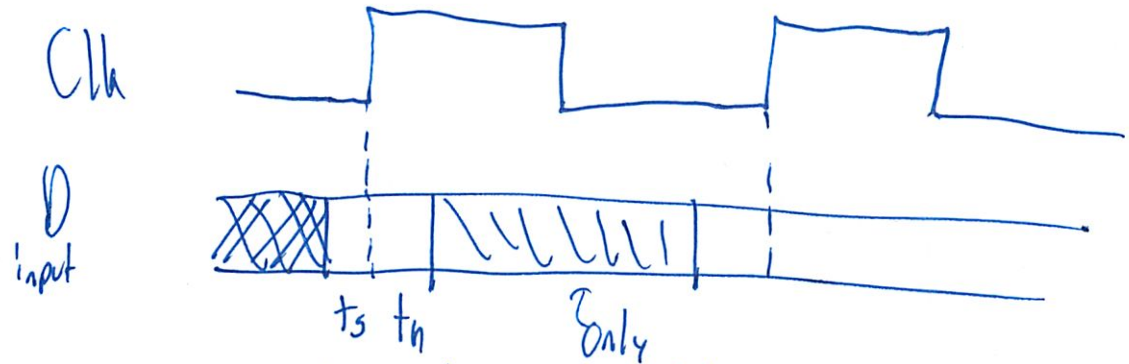
9/30

Register

1-bit memory

"Flip flop"





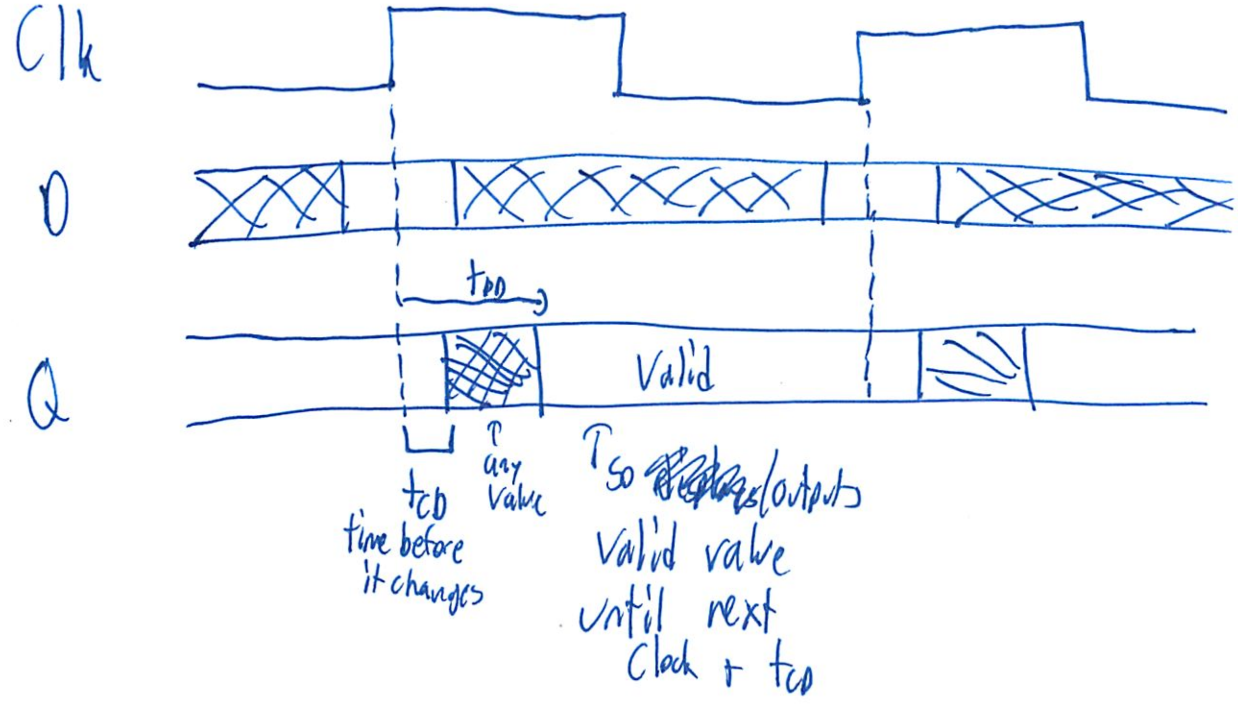
must be held constant
ts before clock
th after edge

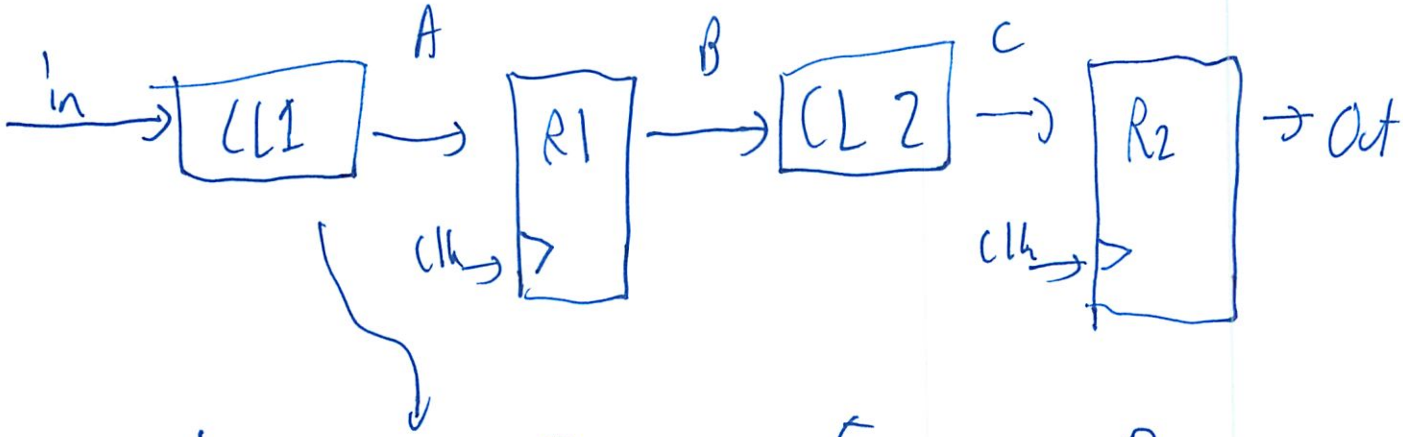
Only care about rising clock edge for D

If can't meet constraints, violate dynamic disipline
can't make guarentees about what it will do
Can go into meta stable state (last lecture)

–

2

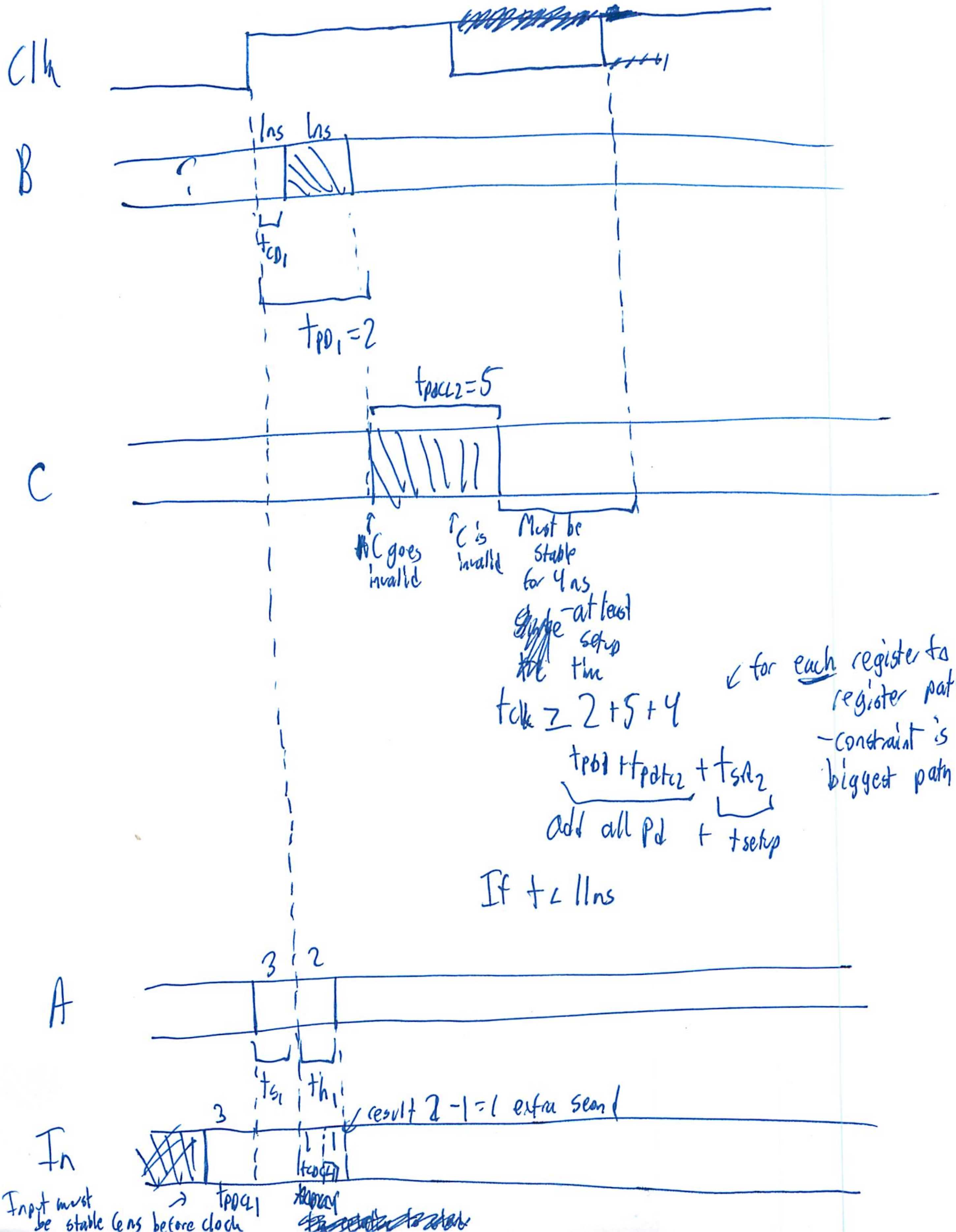




t_{PD}	3	2	5	8
t_{CD}	1	1	1	2
t_s	—	3	—	4
t_h	—	2	—	2

③

Draw some very careful timing diagrams



$t_{clk} \geq 2+5+4$

$\underbrace{t_{PD1}+t_{PDCL2}}_{\text{add all pd}} + \underbrace{t_{SR_2}}_{\text{tsetup}}$

↙ for each register to register path
-constraint is biggest path

If $t < 11ns$

Must also check

$$\sum t_{CD} \geq t_h$$

So check

$$t_{clk} \geq \left(\sum t_{PD}\right) + t_S$$

$$\sum t_{CD} \geq t_h$$

For 100,000 registers a lot of work

But buy a program

JSim has one

Can always lenghten clock period

Its the 2nd constraint thats hard

- circuit goes meta stable
- need to throw it away

t_h ∫

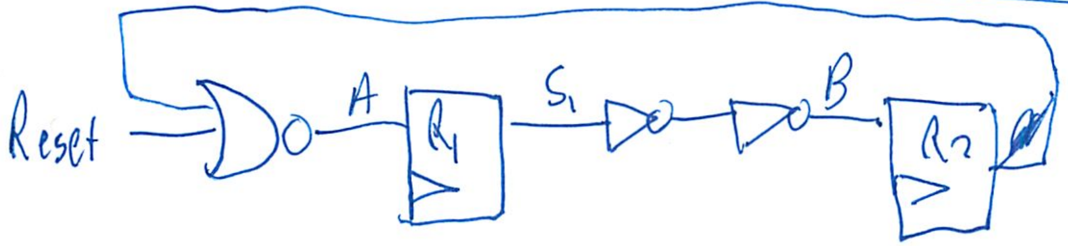
Go from 1 register to another

Use $\sum t_{CD}$ from all CL before that

and the register before that

5

Do a timing diagram to check every little piece
- rather than try to do in your head



t_{pd}	2	2	2	2	2
t_{cd}	1.5	0	1	1	0
t_s	—	3	—	—	3
t_h	—	1	—	—	1

t_{clk} - two register paths to check

$R_1 \to R_2$

$R_2 \to R_1$

$R_1 \to R_2$ $t_{clk} \geq t_{PD,R1} + t_{PD,INV} + t_{PD,INV} + t_{S,R_2}$

$= 2 + 2 + 2 + 3$

$= 9$ ⟸ So largest overall

$R_2 \to R_1$ $t_{ck} \geq t_{PD\,R_2} + t_{PD,NOR} + t_{S\,R_1}$

$= 2 + 2 + 3$

$= 7$

Why 2 invertes?

Since we need $\sum t_{cD} \geq t_h$

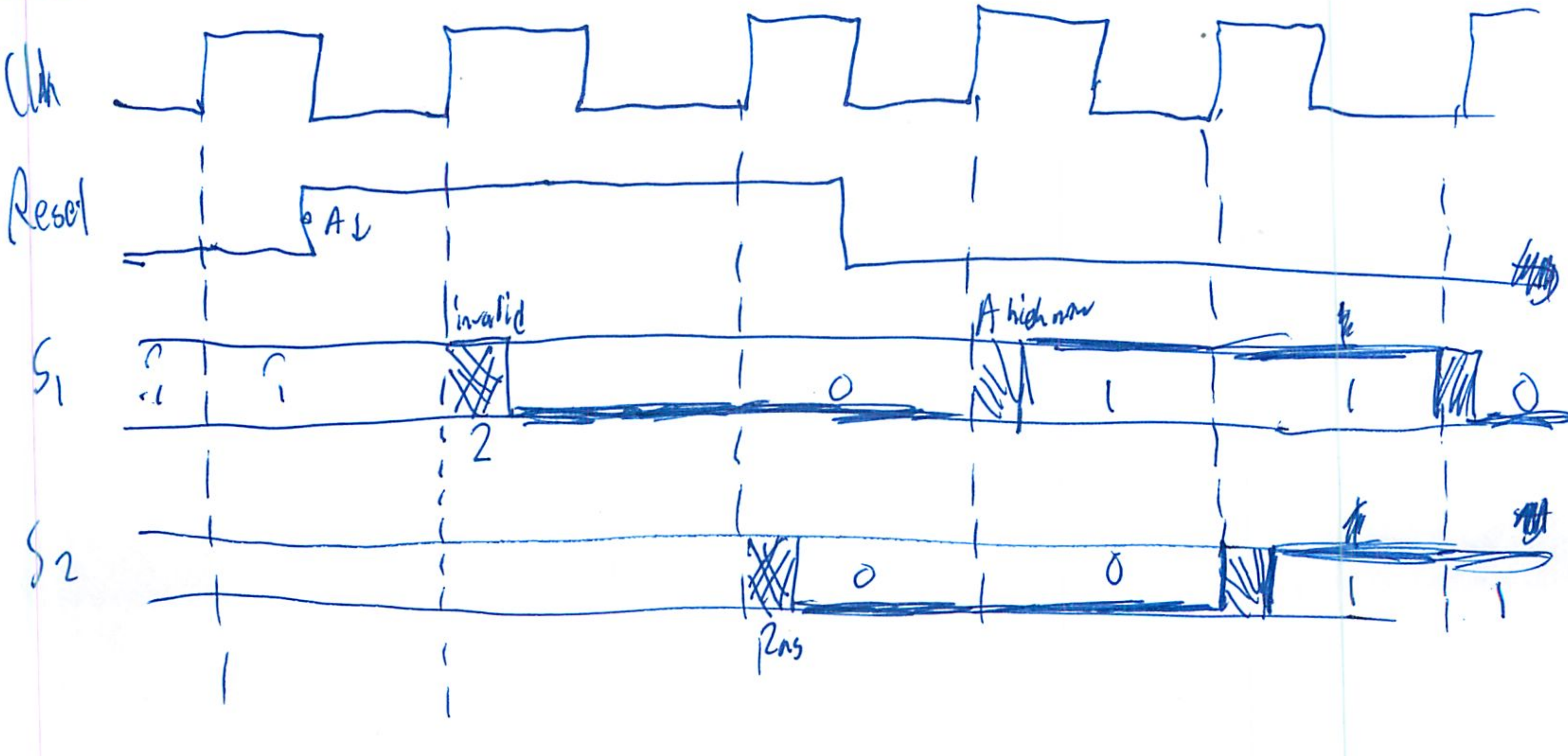
Then $0 \geq 1$ ⊗ Nope

As ICs got faster, had to build some speed bumps

Or could make t_{cD} R_1 ≥ t_h R_2 but ~~[illegible]~~

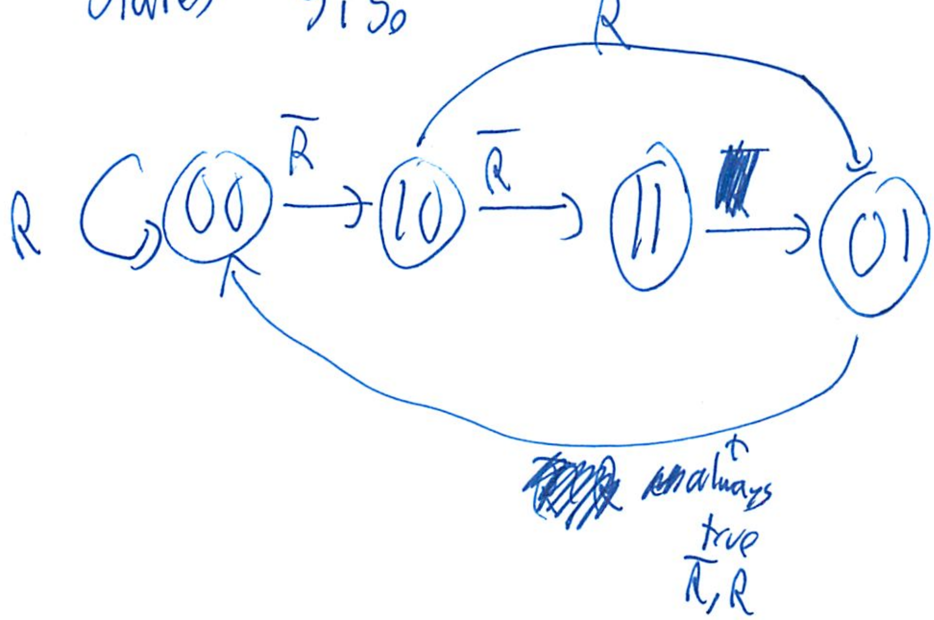
Ya ~~could~~ don't really have control over

Overclocking ~~[illegible]~~ happens because #s given are worse case # - with good cooling, etc are 40% higher than actual



7

States $S_1 S_0$



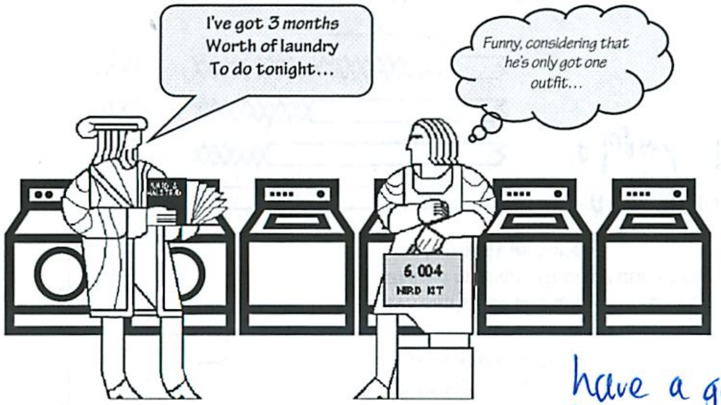
Next time Think about them & turn into scamatics

Old Curmudgeon

10/4

Pipelining

what Seymour Cray taught the laundry industry


I've got 3 months Worth of laundry To do tonight…
Funny, considering that he's only got one outfit…
6.004 NERD KIT

have a good set of building blocks ~~+~~ engineering principles

Lab #3 due thursday!

2nd hardest

today: architectural elements

Forget circuits… lets solve a "Real Problem"

2 components

INPUT:
dirty laundry

OUTPUT:
6 more weeks

Device: Washer

Function: Fill, Agitate, Spin

Washer_{PD} = 30 mins

Device: Dryer

Function: Heat, Spin

Dryer_{PD} = 60 mins

look at like a combo device

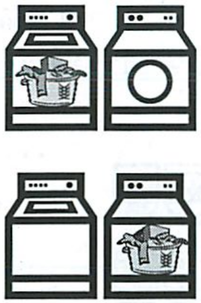
One load at a time

Everyone knows that the real reason that MIT students put off doing laundry so long is not because they procrastinate, are lazy, or even have better things to do.

The fact is, doing one load at a time is not smart.

Step 1:

Step 2:



Total = Washer_{PD} + Dryer_{PD}

= 90 mins

Doing N loads of laundry

Here's how they do laundry at Harvard, the "combinational" way.

(Of course, this is just an urban legend. No one at Harvard actually *does* laundry. The butlers all arrive on Wednesday morning, pick up the dirty laundry and return it all pressed and starched in time for afternoon tea)

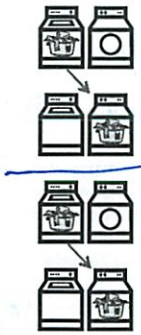
Step 1:

Step 2:

Step 3:

Step 4:

…

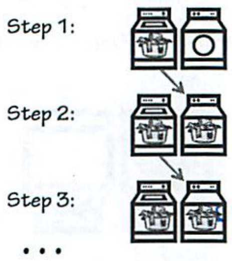


Total = N*(Washer_{PD} + Dryer_{PD})

= N*90 mins

Doing N Loads… the MIT way

MIT students "pipeline" the laundry process.



put 1 in washer while other in dryer

That's why we wait!

Actually, it's more like N*60 + 30 if we account for the startup transient correctly. When doing pipeline analysis, we're mostly interested in the "steady state" where we assume we have an infinite supply of inputs.

Total = N * Max(Washer$_{PD}$, Dryer$_{PD}$)

= N*60 mins

↑ come back in an hr

60 min clock cycle ←

Performance Measures

Latency: ← term invented at 6.004

The delay from when an input is established until the output associated with that input becomes valid.

performing 1 computation

(Harvard Laundry = 90 mins)

(MIT Laundry = 120 mins) ← Assuming that the wash is started as soon as possible and waits (wet) in the washer until dryer is available.

60 min clock cycle, takes 2 cycle

Throughput:

The *rate* of which inputs or outputs are processed.

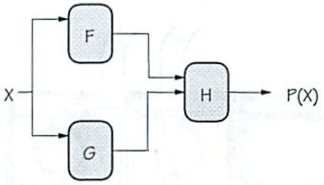
but we care about min total time

(Harvard Laundry = 1/90 outputs/min)

(MIT Laundry = 1/60 outputs/min)

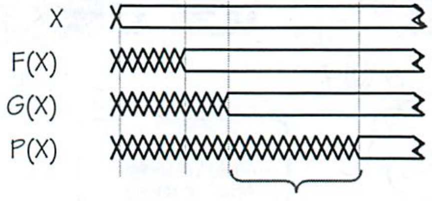
lots of inputs to compute

Okay, back to circuits…



For combinational logic:
latency = t_{PD},
throughput = $1/t_{PD}$.

We can't get the answer faster, but are we making effective use of our hardware at all times?



– min total elapsed time

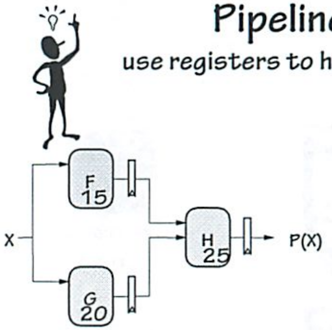
F & G are "idle", just holding their outputs stable while H performs its computation

Could we make more efficient use of hw

registers cheaper to hold F, G

Pipelined Circuits

use registers to hold H's input stable!



Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage *pipeline*: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal zero-delay registers:

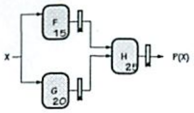
20+25

	latency	throughput
unpipelined	45	1/45 ← inverse of latency
2-stage pipeline	50	1/25
	worse	better

Can clock max of 25 ns (since H)

↓ Common way to do different stages of multiple Computations

Pipeline diagrams



Clock cycle →

Pipeline stages ↓

	i	i+1	i+2	i+3	
Input	X_i	X_{i+1}	X_{i+2}	X_{i+3}	...
F Reg G Reg		$F(X_i)$ $G(X_i)$	$F(X_{i+1})$ $G(X_{i+1})$	$F(X_{i+2})$ $G(X_{i+2})$	...
H Reg			$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.

parallism
better use of HW

↓ Stylized

Pipeline Conventions

DEFINITION:

a *K-Stage Pipeline* ("K-pipeline") is an acyclic circuit having exactly K registers on *every* path from an input to an output.

delay of k clock cycles

a COMBINATIONAL CIRCUIT is thus an 0-stage pipeline.

CONVENTION:

Every pipeline stage, hence every K-Stage pipeline, has a register on its *OUTPUT* (not on its input).

all combo logic up to register

ALWAYS:

The CLOCK common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register t_{PD} PLUS (output) register t_{SETUP}.

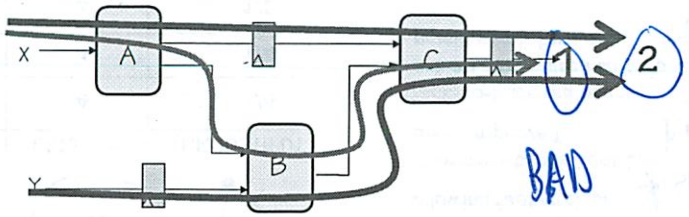
> The LATENCY of a K-pipeline is K times the period of the clock common to all registers.
>
> The THROUGHPUT of a K-pipeline is the frequency of the clock.

k-stage pipeline

Ill-formed pipelines

throwing registers in willy nilly — bad

Consider a BAD job of pipelining:



BAD

For what value of K is the following circuit a K-Pipeline? ANS: none

Problem:

Successive inputs get mixed: e.g., $B(A(X_{i+1}), Y_i)$. This happened because some paths from inputs to outputs have 2 registers, and some have only 1!

This CAN'T HAPPEN on a well-formed K pipeline!

A pipelining methodology

to make a well formed pipeline

Step 1:
Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

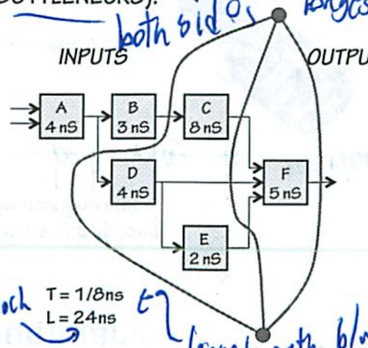
Step 2:
Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate pipeline stages.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.

STRATEGY:
Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).

reduce PD along longest path

both sides



T = 1/8ns
L = 24ns

8 clock
3 stages

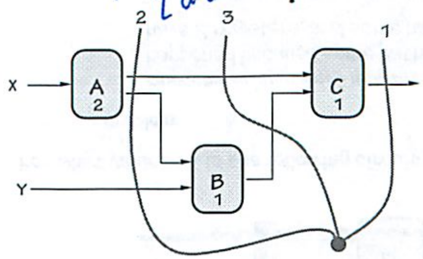
longest path b/w registers

? don't you have to watch internal timing lining up

Pipelining allows us to ↑ speed

to divide 2 and 1+1

Pipeline Example



OBSERVATIONS:

- 1-pipeline improves neither L or T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don't improve T.
- Back-to-back registers are often required to keep pipeline well-formed.

new pipeline * should always break longest combo paths *

	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:	4	1/4
2-pipe:	4	1/2
3-pipe:	6	1/2

← makes things worse!

* Lines geometrically proved to do this well *

Pipelining Summary

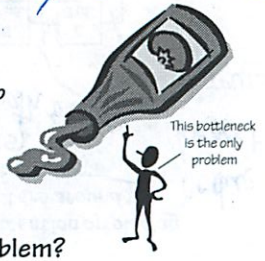
Advantages:

- Allows us to increase thruput, by breaking up long combinational paths and (hence) increasing clock frequency

but may be bottleneck device

Disadvantages:

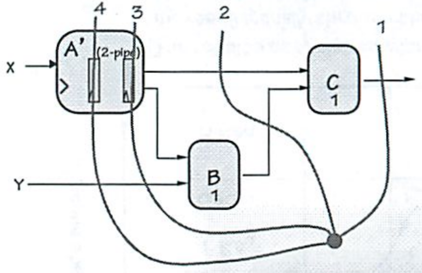
- May increase latency...
- Only as good as the weakest link: slowest step constrains system thruput.



Isn't there a way around this "weak link" problem?

Can replace component w/ pipeline components

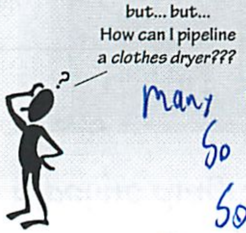
Pipelined Components



4-stage pipeline, thruput=1

Pipelined systems can be hierarchical:

- Replacing a slow combinational component with a k-pipe version may increase clock frequency
- Must account for new pipeline stages in our plan

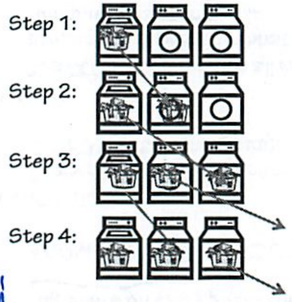


many circuits not pipelinable So need to try Something else

Hard to pipeline

How do 6.004 Aces do Laundry?

They work around the bottleneck. First, they find a place with twice as many dryers as washers.



Throughput = 1/30 loads/min

doing a load every 30 min

Latency = 90 mins/load

Back to our bottleneck...

Slow module, can't pipeline

Recall our earlier example...

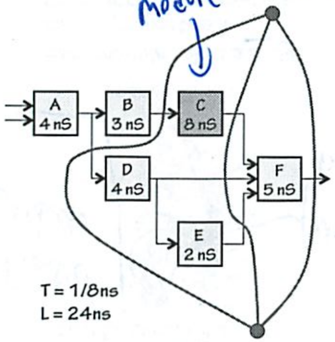
- C – the slowest component – limits clock period to 8 ns.
- HENCE throughput limited to 1/8ns.

We could improve throughput by

- Finding a pipelined version of C;

OR ...

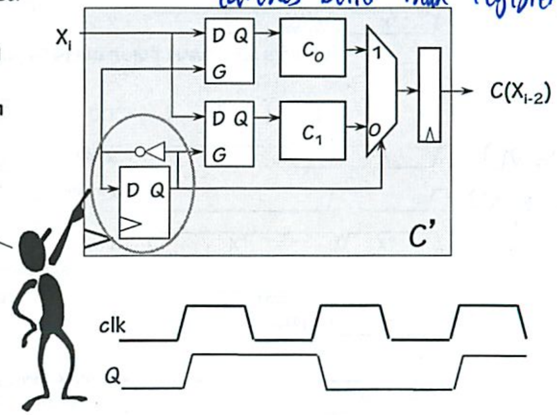
- *interleaving* multiple copies of C!



Circuit Interleaving

latches better than registers

We can simulate a pipelined version of a slow component by replicating the critical element and alternate inputs between the various copies.



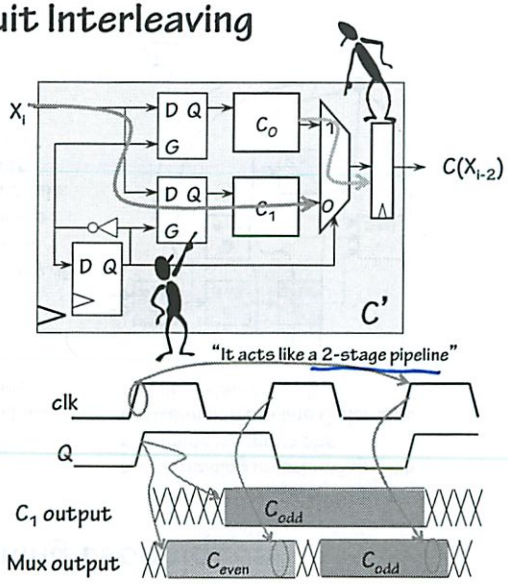
This is a simple 2-state FSM that alternates between 0 and 1 on each clock

Circuit Interleaving

We can simulate a pipelined version of a slow component by replicating the critical element and alternate inputs between the various copies.

When Q is 1 the lower path is combinational (the latch is open), yet the output of the upper path will be enabled onto the input of the output register ready for the NEXT clock edge.

Meanwhile, the other latch maintains the input from the last clock.



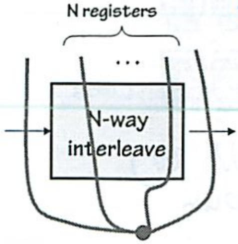
Circuit Interleaving

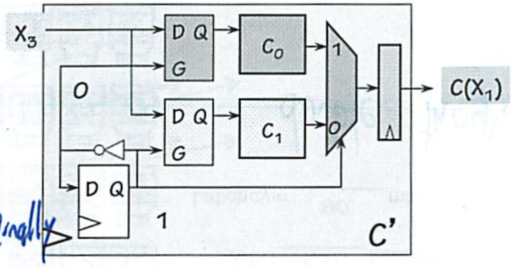
2-Clock Martinizing

"In by t_i, out by t_{i+2}"

N-way interleaving is equivalent to N pipeline Stages...

externally





Latency = 2 clocks

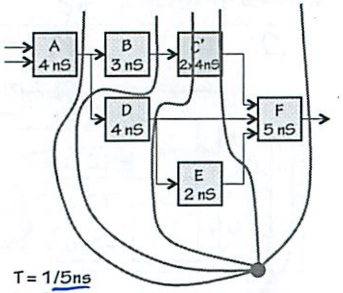
- Clock period 0: X_0 presented at input, propagates thru upper latch, C_0.
- Clock period 1: X_1 presented at input, propagates thru lower latch, C_1. $C_0(X_0)$ propagates to register inputs.
- Clock period 2: X_2 presented at input, propagates thru upper latch, C. $C_0(X_0)$ loaded into register, appears at output.

Combining techniques

We can combine interleaving and pipelining. Here, C' interleaves two C elements with a propagation delay of 8 nS.

The resulting C' circuit has a throughput of 1/4 nS, and latency of 8 nS. This can be considered as an extra pipelining stage that passes through the middle of the C' module. One of our separation lines must pass through this pipeline stage.

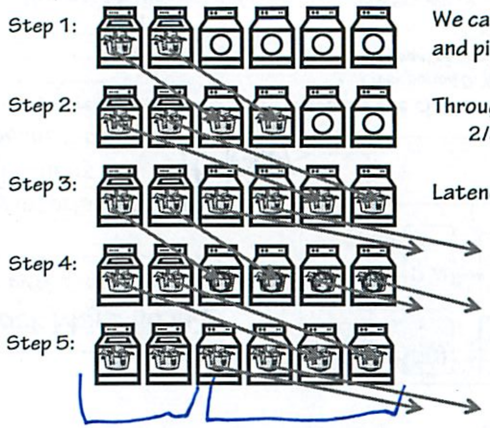
By combining interleaving with pipelining we move the bottleneck from the C element to the F element.



T = 1/5ns
L = 25ns

And a little parallelism…

Can replicate brute force



washer dryer

We can combine interleaving and pipelining with parallelism.

Throughput = 2/30 = 1/15 load/min

Latency = 90 min

doble through force

Control Structure Approaches

Synchronous

ALL computation "events" occur at active edges of a periodic clock: time is divided into fixed-size discrete intervals.

w/ glass door

niches →

no glass door

Asynchronous

Events -- eg the loading of a register -- can happen at at arbitrary times.

RIGID

Laid Back

Globally Timed

Timing dictated by centralized FSM according to a fixed schedule.

Mistro Conductor

passing baton

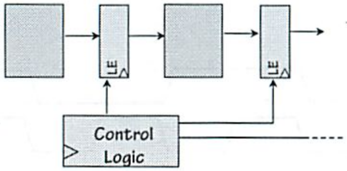
Locally Timed

Each module takes a START signal, generates a FINISHED signal. Timing is dynamic, data dependent.

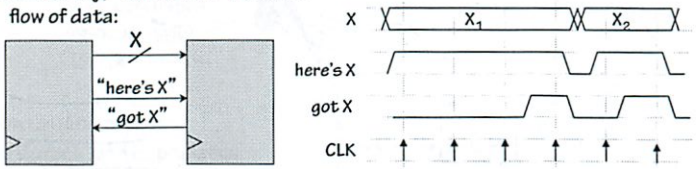
seperate, but related

Control Structure Alternatives

Synchronous, globally-timed: Control signals (e.g., load enables) From FSM controller

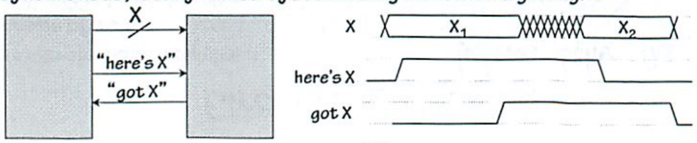


Synchronous, locally-timed: Local circuitry, "handshake" controls flow of data:



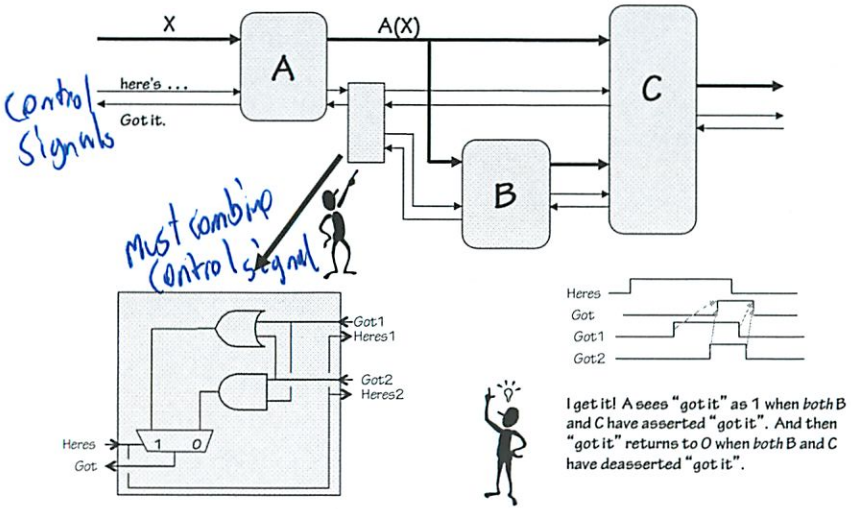
could use clock

Asynchronous, locally-timed system using *transition signaling:*



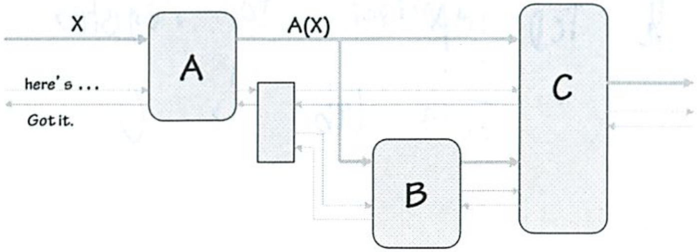
or not

Self-timed Example



I get it! A sees "got it" as 1 when *both* B and C have asserted "got it". And then "got it" returns to 0 when *both* B and C have deasserted "got it".

Self-timed Example



Elegant, timing-independent design:

- Each component specifies its own time constraints
- Local adaptation to special cases (eg, multiplication by 0)
- Module performance improvements automatically exploited
- Can be made asynchronous (no clock at all!) or synchronous

w/t following

Control Structure Taxonomy

Easy to design but fixed-sized interval can be wasteful (no data-dependencies in timing)

Large systems lead to very complicated timing generators… just say no!

	Synchronous	Asynchronous
Globally Timed	Centralized clocked FSM generates all control signals. 6.004 + most of industry	Central control unit tailors current time slice to current tasks. very bad idea
Locally Timed	Start and Finish signals generated by each major subsystem, synchronously with global clock.	Each subsystem takes asynchronous Start, generates asynchronous Finish (perhaps using local clock).

The best way to build large systems that have independently-timed components.

The "next big idea" for the last several decades: a lot of design work to do in general, but extra work is worth it in special cases

Summary

- Latency (L) = time it takes for given input to arrive at output
- Throughput (T) = rate at each new outputs appear
- For combinational circuits: L = t_{PD} of circuit, T = 1/L
- For K-pipelines (K > 0):
  - always have register on output(s)
  - K registers on *every* path from input to output
  - Inputs available shortly after clock i, outputs available shortly after clock (i+K)
  - $T = 1/(t_{PD,REG} + t_{PD}$ of slowest pipeline stage $+ t_{SETUP})$
    - more throughput ⇒ split slowest pipeline stage(s)
    - use replication/interleaving if no further splits possible
  - L = K / T
    - pipelined latency ≥ combinational latency

t_{cD} important for registes

- in lab is 0

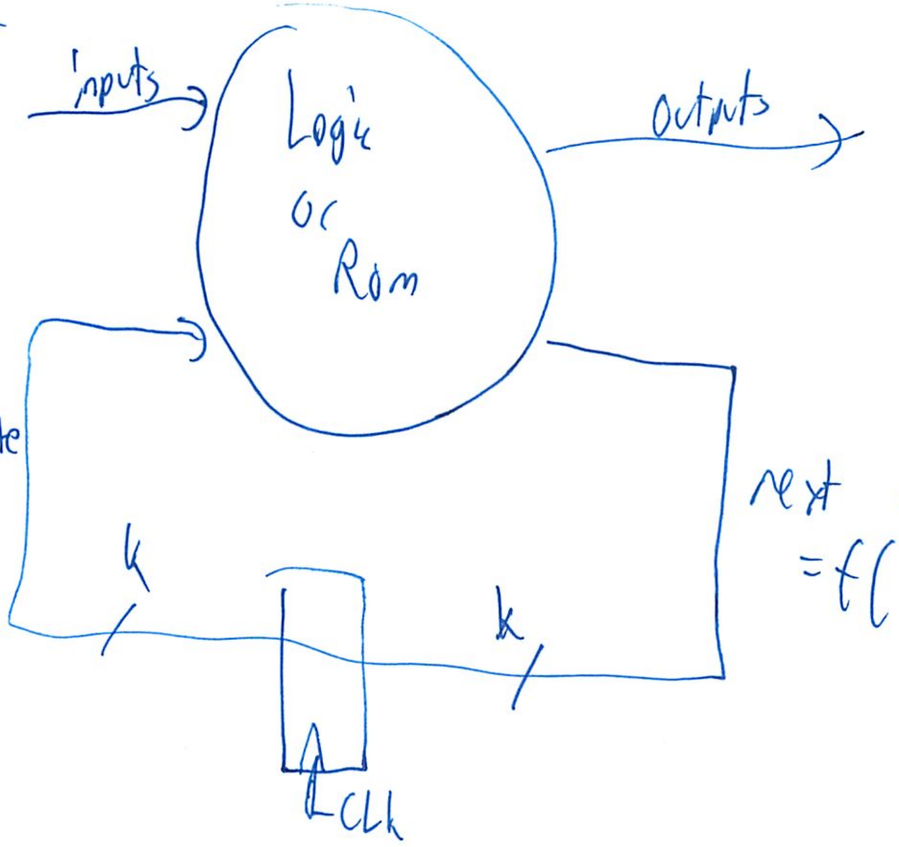
6.004 Recitation

10/5

(3 min late)

✓ generic SM

FSMs



next state
= f(input, state)

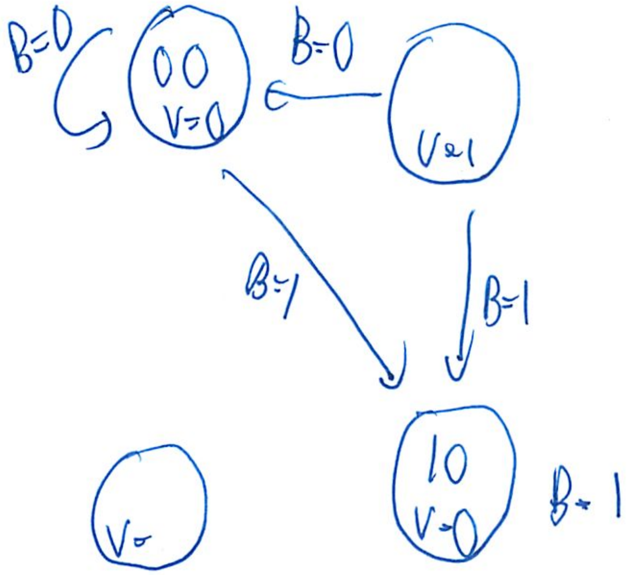
k bits
2^k possible states

Output → { f(state) ← Moore
f(state, input) ← Meeley

don't care about forms

②

State transition diagram



table

State		input	next state	Unlock
S_1	S_0	B	S_1' S_0'	V
0	0	0	00	_
0	0	1	10	_
0	1	0	11	0
0	1	1	_	0
1	0	0	01	_
1	0	1	_ _	_
1	1	0	_ _	1
1	1	1	_ _	1

Fill in table + state diagram

- N digit combo lock
- V = 1 if last N digits = Combo
  = 0 otherwise
- Combo is unique

(3)

Go through and solve new stuff

for state 00 v is always 0

(etc) - missed some

know combo = 100

Oh state is **not** past digits entered

Can check TT - if filled in

SM diagram - from every state an arrow leaves it for each possible input - and only 1 arrow

---

Give a lot of digits

1 0 1 1 0 0 1 1 1 0

MSB LSB

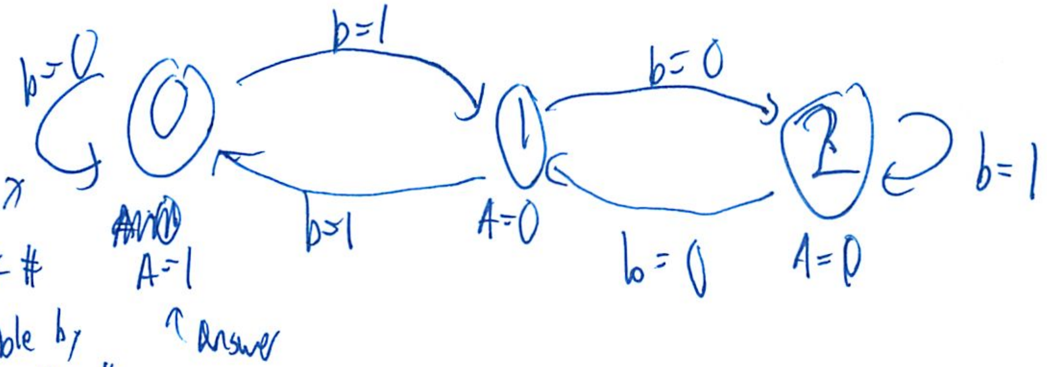
~~AA~~ ~~an~~ Build SM so at any point in time when A is divisable by 3

(4)

So M mod 3 = 0

$$\begin{cases}0\\1\\2\end{cases}$$

$$M' = 2\underbrace{M}_{\text{previous } \#} + \underbrace{b}_{\text{next digit}}$$



So if # divisible by 3 then 2x # is still ÷3

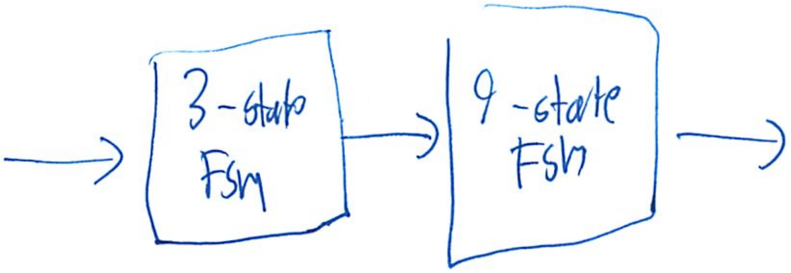
Is there an equivilant 2-state machine?

- would ~~gra~~ give same output

Not here

(5)

How many states?



= 27 total states?

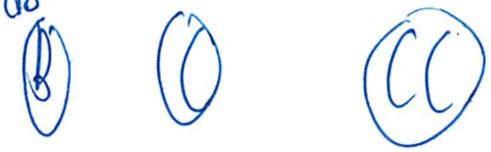
Actually might never get to some

≤ 27 states

Balance ()

((( )) ( ))

Could you do?



etc

How many bits of state does it have?

∞ it can have

Up to max disc space

Tomorrows lecture: Turing machine

Can't compute w/ FSM

6

Will do () balancing on turing machine

Are there other functions that even Turing machine can do

10/5

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Lab #3

In this laboratory exercise, we'll build the *arithmetic and logic unit* (ALU) for the Beta processor. The ALU has two 32-bit inputs (which we'll call "A" and "B") and produces one 32-bit output. We'll start by designing each piece of the ALU as a separate circuit, each producing its own 32-bit output. Then we'll combine these outputs into a single ALU result.

How can we have 32 bits on an input? Guess upcoming lectures

When designing circuitry there are three separate factors that can be optimized:

(1) design for maximum performance (minimum latency)
(2) design for minimum cost (minimum area)
(3) design for the best cost/performance ratio (minimize area*latency)

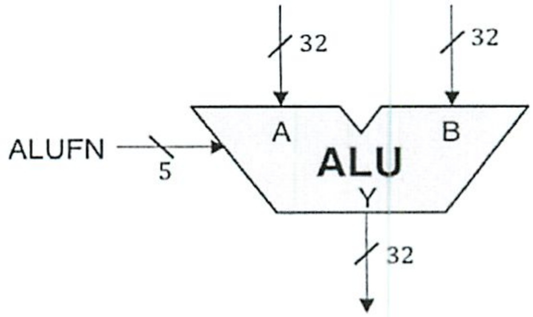
Happily it's often possible to do all three at once but in some portions of the circuit some sort of *design tradeoff* will need to be made. When designing your circuitry you should choose which of these three factors is most important to you and optimize your design accordingly.

A functional ALU design will earn six points. Four additional points can be earned if you implement the optional multiplier unit – see the section labeled "Optional Design Problem: Implementing Multiply" for details.

ALU Specification

The 32-bit ALU we will build will be a component in the Beta processor we will address in subsequent laboratories. The logic symbol for our ALU is shown to the right. It is a combinational circuit taking two 32-bit data words A and B as inputs, and producing a 32-bit output Y by performing a specified arithmetic or logical function on the A and B inputs. The particular function to be performed is specified by a 5-bit control input, ALUFN, whose value encodes the function according to the following table:

is this → the computer we are building?



ALUFN[4:0]	Operation	Output value Y[31:0]
1abcd	Bitwise Boolean	Y[i] = F_{abcd}(A[i], B[i])
00000	32-bit ADD	Y = A+B
00001	32-bit SUBTRACT	Y = A-B
00010	32-bit MULTIPLY (optional)	Y = A*B
00101	CMPEQ	Y = (A == B)
00111	CMPLT	Y = (A < B)
01101	CMPLE	Y = (A <= B)
01000	Shift left (SHL)	Y = A << B
01001	Shift right (SHR)	Y = A >> B
01011	Shift right, sign extended (SRA)	Y = A >> B (sign extended)

+ abcd are TT → see next pg

How do you do more than 1 thing at once?

Note that by specifying an appropriate value for the 5-bit ALUFN input, the ALU can perform a variety of arithmetic operations, comparisons, shifts, and bitwise Boolean combinations required by our Beta processor.

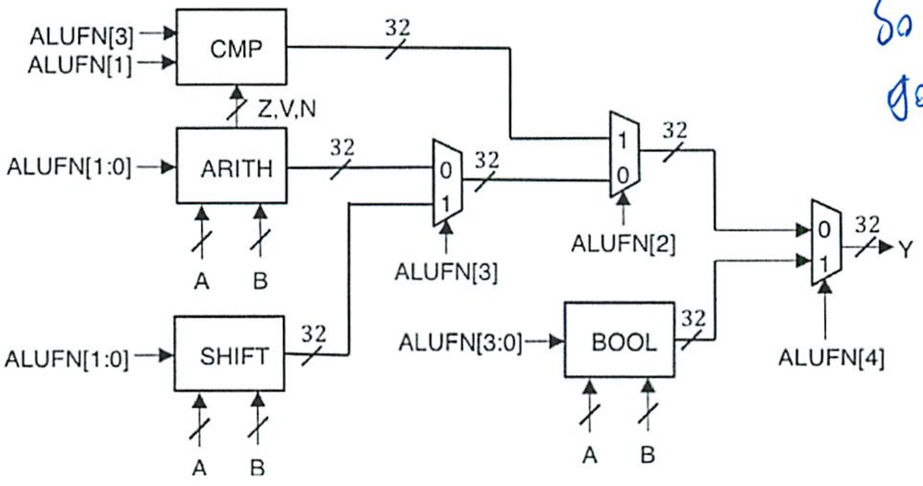
The bitwise Boolean operations are specified by ALUFN[4]=1; in this case, the remaining ALUFN bits abcd are taken as entries in the truth table describing how each bit of Y is determined by the corresponding bits of A and B, as shown to the right.

B_i	A_i	Y_i
0	0	*d*
0	1	*c*
1	0	*b*
1	1	*a*

The three compare operations each produce a Boolean output. In these cases, Y[31:1] are all zero, and the low-order bit Y[0] is a 0 or 1 reflecting the outcome of the comparison between the 32-bit A and B operands.

We can approach the ALU design by breaking it down into subsystems devoted to arithmetic, comparison, Boolean, and shift operations as shown below:

So each fn goes to ~~separate~~ specific part?



By following this strategy, you can use supplied test jigs to debug each of the four modules separately before assembling them to make your ALU.

Standard Cell Library

The building blocks for our design will be a family of logic gates that are part of a *standard cell library*. The available combinational gates are listed in the table below along with information about their timing, loading and size. You can access the library by starting your netlist with the following include statements:

```
.include "/mit/6.004/jsim/nominal.jsim"
.include "/mit/6.004/jsim/stdcell.jsim"
```

← I thought we built last lab

Everyone should use the provided cells in creating their design. The timings have been taken from a 0.18 micron CMOS process measured at room temperature.

Cell Library

Netlist	*Function*	t_{CD} *(ns)*	t_{PD} *(ns)*	t_R *(ns/pf)*	t_F *(ns/pf)*	*load (pf)*	*size* (μ^2)
`Xid z constant0`	$Z = 0$	—	—	—	—	—	0
`Xid z constant1`	$Z = 1$	—	—	—	—	—	0
`Xid a z inverter`	$Z = \overline{A}$	.005	.02	2.3	1.2	.007	10
`Xid a z inverter_2`		.009	.02	1.1	.6	.013	13
`Xid a z inverter_4`		.009	.02	.56	.3	.027	20
`Xid a z inverter_8`		.02	.11	.28	.15	.009	56
`Xid a z buffer`	$Z = A$	.02	.08	2.2	1.2	.003	13
`Xid a z buffer_2`		.02	.07	1.1	.6	.005	17
`Xid a z buffer_4`		.02	.07	.56	.3	.01	30
`Xid a z buffer-8`		.02	.07	.28	.15	.02	43
`Xid e a z tristate`	$Z = A$ when e=1 else Z not driven	.03	.15	2.3	1.3	.004	23
`Xid e a z tristate_2`		.03	.13	1.1	.6	.006	30
`Xid e a z tristate_4`		.02	.12	.6	.3	.011	40
`Xid e a z tristate_8`		.02	.11	.3	.17	.02	56
`Xid a b z and2`	$Z = A \cdot B$	.03	.12	4.5	2.3	.002	13
`Xid a b c z and3`	$Z = A \cdot B \cdot C$	.03	.15	4.5	2.6	.002	17
`Xid a b c d z and4`	$Z = A \cdot B \cdot C \cdot D$	.03	.16	4.5	2.5	.002	20
`Xid a b z nand2`	$Z = \overline{A \cdot B}$	.01	.03	4.5	2.8	.004	10
`Xid a b c z nand3`	$Z = \overline{A \cdot B \cdot C}$	.01	.05	4.2	3.0	.005	13
`Xid a b c d z nand4`	$Z = \overline{A \cdot B \cdot C \cdot D}$	.01	.07	4.4	3.5	.005	17
`Xid a b z or2`	$Z = A + B$	.03	.15	4.5	2.5	.002	13
`Xid a b c z or3`	$Z = A + B + C$	.04	.21	4.5	2.5	.003	17
`Xid a b c d z or4`	$Z = A + B + C + D$	.06	.29	4.5	2.6	.003	20
`Xid a b z nor2`	$Z = \overline{A + B}$	.01	.05	6.7	2.4	.004	10
`Xid a b c z nor3`	$Z = \overline{A + B + C}$	.02	.08	8.5	2.4	.005	13
`Xid a b c d z nor4`	$Z = \overline{A + B + C + D}$	.02	.12	9.5	2.4	.005	20
`Xid a b z xor2`	$Z = A \oplus B$	.03	.14	4.5	2.5	.006	27
`Xid a b z xnor2`	$Z = \overline{A \oplus B}$	.03	.14	4.5	2.5	.006	27
`Xid a1 a2 b z aoi21`	$Z = \overline{(A1 \cdot A2) + B}$	.02	.07	6.8	2.7	.005	13
`Xid a1 a2 b z oai21`	$Z = \overline{(A1 + A2) \cdot B}$	.02	.07	6.7	2.7	.005	17
`Xid s d0 d1 z mux2`	Z = D0 when S = 0 Z = D1 when S = 1	.02	.12	4.5	2.5	.005	27
`Xid s0 s1 d0 d1 d2 d3 z mux4` *(Note order of s0 and s1!)*	Z=D0 when S_0=0, S_1=0 Z=D1 when S_0=1, S_1=0 Z=D2 when S_0=0, S_1=1 Z=D3 when S_0=1, S_1=1	.04	.19	4.5	2.5	.006	66
`Xid d clk q dreg` t_{setup} = .15, t_{hold} = 0	D→Q on CLK↑	.03	.19	4.3	2.5	.002	56

Gate-level Simulation

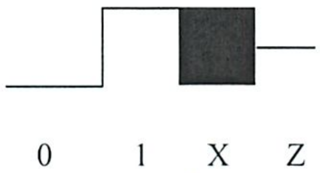
Since we're designing at the gate level we can use a faster simulator that only knows about gates and logic values (instead of transistors and voltages). You can run JSim's gate-level simulator by clicking ▷ in the toolbar. Note that your design can't contain any mosfets, resistors, capacitors, etc.; this simulator only supports the gate primitives in the standard cell library.

So just logical 1s or 0s

Inputs are still specified in terms of voltages (to maintain netlist compatability with the other simulators) but the gate-level simulator converts voltages into one of three possible logic values using the VIL and VIH thresholds specified in nominal.jsim:

0	logic low (voltages less than or equal to VIL threshold)
1	logic high (voltages greater than or equal to VIH threshold)
X	unknown or undefined (voltages between the thresholds, or unknown voltages)

A fourth value "Z" is used to represent the value of nodes that aren't being driven by any gate output (e.g., the outputs of tristate drivers that aren't enabled). The following diagram shows how these values appear on the waveform display:



any

Connecting electrical nodes together using .connect

JSim has a control statement that lets you connect two or more nodes together so that they behave as a single electrical node:

```
.connect node1 node2 node3...
```

renaming / alies

The .connect statement is useful for connecting two terminals of a subcircuit or for connecting nodes directly to ground. For example, the following statement ties nodes cmp1, cmp2, ..., cmp31 directly to the ground node (node "0"):

```
.connect 0 cmp[31:1]
```

multiselect

Note that the `.connect` control statement in JSim works differently than many people expect. For example,

```
.connect A[5:0] B[5:0]
```

will connect **all** twelve nodes (A5, A4, ..., A0, B5, B4, ..., B0) together -- usually not what was intended. To connect two busses together, one could have entered

```
.connect A5 B5
.connect A4 B4
...
```

which is tedious to type. Or one can define a two-terminal device that uses `.connect` internally, and then use the usual iteration rules (see next section) to make many instances of the device with one "X" statement:

```
.subckt knex a b
.connect a b
.ends
X1 A[5:0] B[5:0] knex
```

Using iterators to create multiple gates with a single "X" statement

what is an "X" statement?

JSim makes it easy to specify multiple gates with a single "X" statement. You can create multiple instances of a device by supplying some multiple of the number of nodes it expects, e.g., if a device has 3 terminals, supplying 9 nodes will create 3 instances of the device. To understand how nodes are matched up with terminals specified in the .subckt definition, imagine a device with P terminals. The sequence of nodes supplied as part of the "X" statement that instantiates the device are divided into P equal-size contiguous subsequences. The first node of each subsequence is used to wire up the first device, the second node of each subsequence is used for the second device, and so on until all the nodes have been used. For example:

```
Xtest a[2:0] b[2:0] z[2:0] xor2 On
```

is equivalent to xtest a2 a1 a0 b2 b1 b0 z2 z1 z0 xnor3

```
Xtest#0 a2 b2 z2 xor2
Xtest#1 a1 b1 z1 xor2
Xtest#2 a0 b0 z0 xor2
```

Opposite of previous?

since xor2 has 3 terminals. There is also a handy way of duplicating a signal: specifying "foo#3" is equivalent to specifying "foo foo foo". For example, xor'ing a 4-bit bus with a control signal could be written as

```
Xbusctl in[3:0] ctl#4 out[3:0] xor2
```

which is equivalent to

```
Xbusctl#0 in3 ctl out3 xor2
Xbusctl#1 in2 ctl out2 xor2
Xbusctl#2 in1 ctl out1 xor2
Xbusctl#3 in0 ctl out0 xor2
```

Using iterators and the "constant0" device from the standard cell library, here's a better way of connecting cmp[31:1] to ground:

```
Xgnd cmp[31:1] constant0
```

Since the "constant0" has one terminal and we supply 31 nodes, 31 copies of the device will be made.

ALU Design

Designing a complex system like an ALU is best done in stages, allowing individual subsystems to be designed and debugged one at a time. The steps below follow that approach to implementing the ALU block diagram shown on page 2. We begin by implementing an ALU framework with dummy subcircuits for each of the four major subsystems (BOOL, ARITH, CMP, and SHIFT); we then implement and debug real working versions of each subsystem. To help you follow this path, we provide jsim "test jigs" which test each class of ALU operations separately.

NOTE: the ALUFN signals used to control the operation of the ALU circuitry use an encoding chosen to make the design of the ALU circuitry simple. This encoding is *not* the same as the one used to encode the 6-bit opcode field of Beta instructions. In Lab 6, you'll build some logic (actually a ROM) that will translate the opcode field of an instruction into the appropriate ALUFN control bits.

building a compter from grand up

(A) Make a new file (called, say, "lab3.jsim") and paste in the following code:

```
.include "/mit/6.004/jsim/nominal.jsim"
.include "/mit/6.004/jsim/stdcell.jsim"
.include "/mit/6.004/jsim/lab3_test_bool.jsim"

.subckt BOOL alufn[3:0] A[31:0] B[31:0] OUT[31:0]
xdummy OUT[31:0] constant0
.ends
```

so each of the parts

```
.subckt ARITH alufn[1:0] A[31:0] B[31:0] OUT[31:0] Z V N
xdummy OUT[31:0] Z V N constant0
.ends

.subckt SHIFT alufn[1:0] A[31:0] B[31:0] OUT[31:0]
xdummy OUT[31:0] constant0
.ends

.subckt CMP alufn3 alufn1 Z V N OUT[31:0]
xdummy OUT[31:0] constant0
.ends

.subckt alu alufn[4:0] a[31:0] b[31:0] out[31:0] z v n

*** Generate outputs from each of BOOL, SHIFT, ARITH, CMP subcircuits:
xbool alufn[3:0] a[31:0] b[31:0] boolout[31:0] BOOL
xshift alufn[1:0] a[31:0] b[31:0] shiftout[31:0] SHIFT
xarith alufn[1:0] a[31:0] b[31:0] arithout[31:0] z v n ARITH
xcmp alufn[3] alufn[1] z v n cmpout[31:0] CMP
```

testing

```
*** Combine them, using three multiplexors:
xmux1 alufn[4]#32 nonbool[31:0] boolout[31:0] out[31:0] mux2
xmux2 alufn[2]#32 arithshift[31:0] cmpout[31:0] nonbool[31:0] mux2
xmux3 alufn[3]#32 arithout[31:0] shiftout[31:0] arithshift[31:0] mux2
```

chart p 2

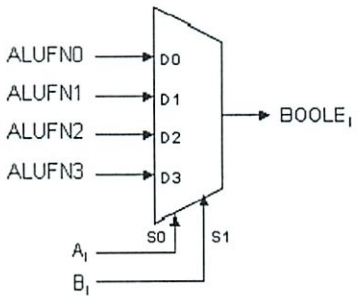
```
.ends
```

handels selection

Note the relationship between this file and the block diagram on page 2. The file defines an **alu** subcircuit, with its appropriate inputs and outputs. The body of this subcircuit creates instances of modules called **BOOL, ARITH, CMP**, and **SHIFT** (corresponding to blocks within the diagram), and muxes them together according to our **ALUFN** coding scheme. However, each of the four component modules is simply a dummy definition that connects each output wire to logical 0 (using the **constant0** device).

(B) Now, design the circuitry to implement the Boolean operations for your ALU. To do this, replace the "xdummy ..." line in the definition of the **BOOL** subcircuit with jsim code that combines its 32-bit A and B arguments according to the supplied 4-bit alufn code, and sets OUT[31:0] to the result.

Our implementation of the 32-bit boolean unit uses a 32 copies of a 4-to-1 multiplexer where ALUFN0, ALUFN1, ALUFN2, and ALUFN3 encode the operation to be performed, and A_i and B_i are hooked to the select inputs. This implementation can produce any of the 16 2-input Boolean functions.



The following table shows the encodings for some of the ALUFN[3:0] control signals used by the test jig (and in our typical Beta implementations):

Operation	*ALUFN[3:0]*
AND	1000
OR	1110
XOR	0110
"A"	1010

The test jig actually checks all 16 boolean operations on a selection of arguments, and will report any errors that it finds. It is specified in your lab3.jsim file by the line that reads

```
.include "/mit/6.004/jsim/lab3_test_bool.jsim"
```

Then do a gate-level simulation; a waveform window showing the ALU inputs and outputs should appear. Next click the checkoff button (the green checkmark) in the toolbar. JSim will check your circuit's results against a list of expected values and report any discrepancies it finds. Using this test jig file, nothing will be sent to the on-line server – it's provided to help test your design as you go. Once your ALU passes this test suite, you have some moderate assurance that it is working properly.

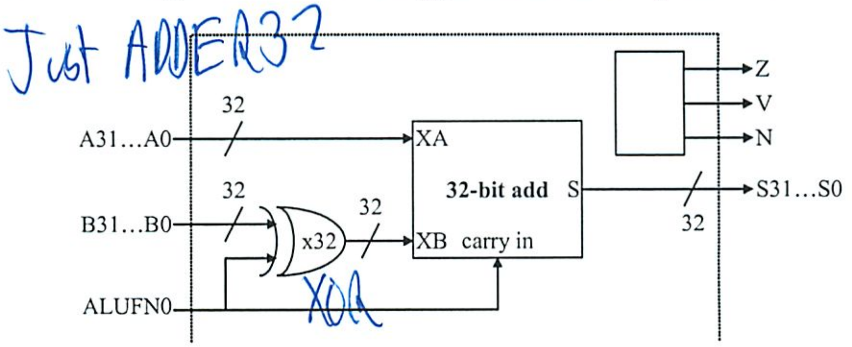
(C) Design an adder/subtractor (**ARITH**) unit that operates on 32-bit two's complement inputs and generates a 32-bit output. It will be useful to generate three other output signals to be used by the comparison logic in part (B): "Z" which is true when the S outputs are all zero, "V" which is true when the addition operation overflows (i.e., the result is too large to be represented in 32 bits), and "N" which is true when the S is negative (i.e., $S_{31} = 1$). Overflow can never occur when the two operands to the addition have different signs; if the two operands have the same sign, then overflow can be detected if the sign of the result differs from the sign of the operands:

$$V = XA_{31} \cdot XB_{31} \cdot \overline{S_{31}} + \overline{XA_{31}} \cdot \overline{XB_{31}} \cdot S_{31}$$

Note that this equation uses XB_{31}, which is the high-order bit of the B operand to the adder itself (i.e., *after* the XOR gate – see the schematic below).

ALUFN0 will be set to 0 for an ADD (S = A + B) and 1 for a SUBTRACT (S = A – B); A[31:0] and B[31:0] are the 32-bit two's complement input operands; S[31:0] is the 32-bit result; z/v/n are the three condition code bits described above. We'll be using the "little-endian" bit numbering convention where bit 31 is the most-significant bit and bit 0 is the least-significant bit.

The following schematic is one suggestion for how to go about the design:



The ALUFN0 input signal selects whether the operation is an ADD or SUBTRACT. To do a SUBTRACT, the circuit first computes the two's complement negation of the "B" operand by inverting "B" and then adding one (which can be done by forcing the carry-in of the 32-bit add to be 1). Start by implementing the 32-bit add using a ripple-carry architecture (you'll get to improve on this later on the lab). You'll have to construct the 32-input NOR gate required to compute Z using a tree of smaller fan-in gates (the parts library only has gates with up to 4 inputs).

We've created a test jig to test your adder. Once you've filled in definition of the ARITH subcircuit, change the line including the Boolean test jig to:

```
.include "/mit/6.004/jsim/lab3_test_adder.jsim"
```

and debug your newly implemented adder circuitry.

The Beta instruction set includes three compare instructions (CMPEQ, CMPLT, CMPLE) that compare the "A" and "B" operands. We can use the adder unit designed above to compute "A–B" and then look at the result (actually just the Z, V and N condition codes) to determine if A=B, A<B or A≤B. The compare instructions generate a 32-bit Boolean result, using "0" to

represent false and "1" to represent true.

(D) Design a 32-bit compare (**CMP**) unit that generates one of two constants ("0" or "1") depending on the ALUFN control signals (used to select the comparison to be performed) and the Z, V, and N outputs of the adder/subtractor unit. Clearly the high order 31 bits of the output are always zero. The least significant bit of the output is determined by the comparison being performed and the results of the subtraction carried out by the adder/subtractor:

	Comparison	*Equation for LSB*	*ALUFN3*	*ALUFN1*
CMPEQ	A = B	LSB = Z	0	0
CMPLT	A < B	LSB = N $\oplus$ V	0	1
CMPLE	A <= B	LSB = Z + (N $\oplus$ V)	1	0

ALUFN bits 3 and 1 are used to control the compare unit since we also need to control the adder/subtractor unit (i.e., ALUFN0 = 1 to force a subtract).

Performance note: the Z, V and N inputs to this circuit can only be calculated by the adder/subtractor unit after the 32-bit add is complete. This means they arrive quite late and then require further processing in this module, which in turn makes cmp0 show up very late in the game. You can speed things up considerably by thinking about the relative timing of Z, V and N and then designing your logic to minimize delay paths involving late-arriving signals.

We've created a test jig to test your compare circuitry. Once you've filled in definition of the **CMP** subcircuit, change the line including the test jig to:

```
.include "/mit/6.004/jsim/lab3_test_cmp.jsim"
```

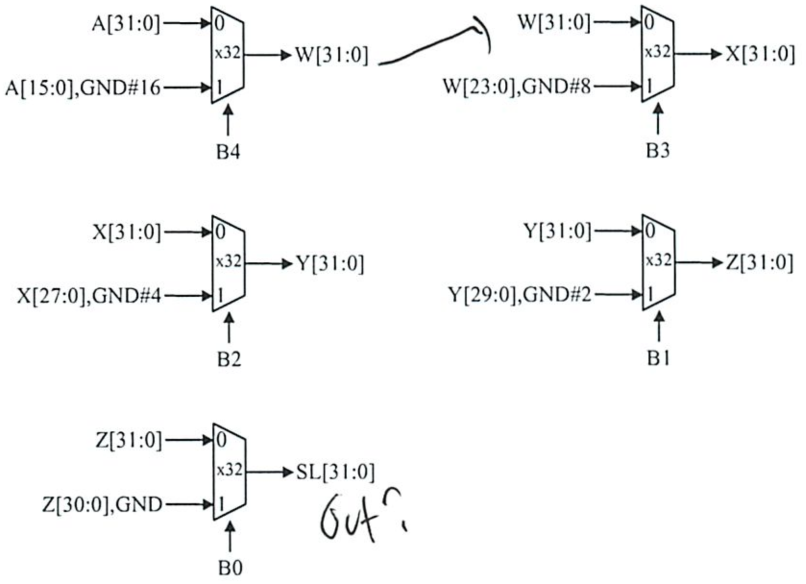
and debug your newly implemented functions.

(E) Design a 32-bit shifter (**SHIFT** block) that implements SRA, SHR and SHL instructions. The "A" operand supplies the data to be shifted and the low-order 5 bits of the "B" operand are used as the shift count (i.e., from 0 to 31 bits of shift). The desired operation will be encoded on ALUFN[1:0] as follows:

Operation	*ALUFN[1:0]*
SHL (shift left)	00
SHR (shift right)	01
SRA (shift right with sign extension)	11

With this encoding, ALUFN0 is 0 for a left shift and 1 for a right shift and ALUFN1 controls the sign extension logic on right shift. For SHL and SHR, 0's are shifted into the vacated bit positions. For SRA ("shift right arithmetic"), the vacated bit positions are all filled with A31, the sign bit of the original data so that the result will be the same as dividing the original data by the appropriate power of 2.

The simplest implementation is to build two shifters—one for shifting left and one for shifting right—and then use a 2-way 32-bit multiplexer to select the appropriate answer as the unit's output. It's easy to build a shifter after noticing that a multi-bit shift can be accomplished by cascading shifts by various powers of 2. For example, a 13-bit shift can be implemented by a shift of 8, followed by a shift of 4, followed by a shift of 1. So the shifter is just a cascade of multiplexers each controlled by one bit of the shift count. The schematic below shows a possible implementation of the left shift logic; the right shift logic is similar with the slight added complication of having to shift in either "0" or "A31." Another approach that adds latency but saves gates is to use the left shift logic for both left and right shifts, but for right shifts, reverse the bits of the "A" operand on the way in and reverse the bits of the output on the way out.



We've created a test jig to test your compare circuitry. Once you've filled in definition of the **SHIFT** subcircuit, change the line including the test jig to:

```
.include "/mit/6.004/jsim/lab3_test_shift.jsim"
```

and debug your newly implemented shift functions.

(F) When you've completed your design, you can use lab3checkoff_6.jsim to test your circuit. Change the test jig include line to

```
.include "/mit/6.004/jsim/lab3checkoff_6.jsim"
```

and run the gate level simulation. This runs each of the test suites that you've used to debug the component subcircuits, so unless there's some unforeseen interaction among your blocks you're likely to pass the test. Clicking the green checkmark on success will record your success: you've earned 6 points and are ready for your checkoff interview!

Optional Design Problem: Implementing Multiply

The goal of this design project is build a combinational multiplier that accepts 32-bit operands and produces a 32-bit result. Multiplying two 32-bit numbers produces a 64-bit product; the result we're looking for is the low-order 32-bits of the 64-bit product.

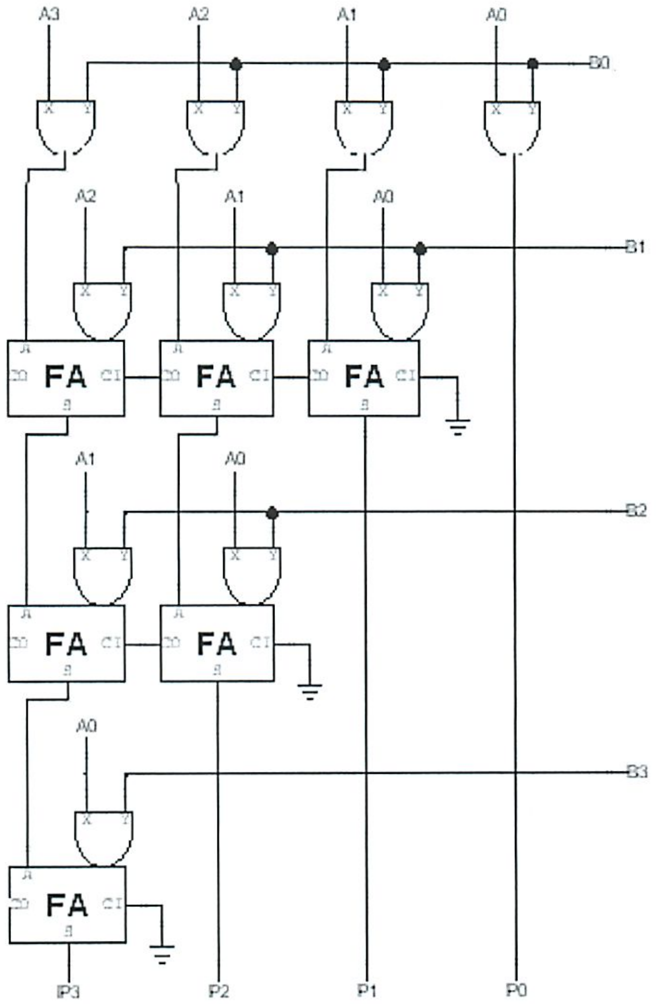
Your multiplier circuitry should be integrated into the **ARITH** design you completed in the first part of this lab. Your augmented **ARITH** unit must be modified to produce a product if ALUFN[1] is asserted, otherwise outputting from the adder/subtractor as it did before.

Here's a detailed bit-level description of how a 4-bit by 4-bit unsigned multiplication works. This diagram assumes we only want the low-order 4 bits of the 8-bit product.

```
        A3     A2     A1     A0     (multiplicand)
  *     B3     B2     B1     B0     (multiplier)
   ---------------------------
     A3*B0  A2*B0  A1*B0  A0*B0     (partial product)
  +  A2*B1  A1*B1  A0*B1      0
  +  A1*B2  A0*B2      0      0
  +  A0*B3      0      0      0
    --------------------------
        P3     P2     P1     P0
```

This diagram can be extended in a straightforward way to 32-bit by 32-bit multiplication. Note that since we only want the low-order 32-bits of the result, you don't need to include the circuitry that generates the rest of the 64-bit product.

As you can see from the diagram above, forming the partial products is easy! Multiplication of two bits can be implemented using an AND gate. The hard part is adding up all the partial products (there will be 32 partial products in your circuit). One can use full adders (FAs) hooked up in a ripple-carry configuration to add each partial product to the accumulated sum of the previous partial products (see the diagram below). The circuit closely follows the diagram above but omits an FA module if two of its inputs are "0".



The circuit above works with both unsigned operands and signed two's complement operands. This may seem strange – don't we have to worry about the most significant bit (MSB) of the operands? With unsigned operands the MSB has a weight of 2^{MSB} (assuming the bits are numbered 0 to MSB) but with signed operands the MSB has a weight of -2^{MSB}. Doesn't our circuitry need to take that into account?

It does, but when we're only saving the lower half of the product, the differences don't appear. The multiplicand (A in the figure above) can be either unsigned or two's complement, the FA circuits will perform correctly in either case. When the multiplier (B in the figure above) is signed, we should *subtract* the final partial product instead of adding it. But subtraction is the same as adding the negative, and the negative of a two's complement number can be computed by taking its complement and adding 1. When we work this through we see that the low-order bit of the partial product is the same whether positive or negated. And the low-order bit is all that we need when saving only the lower half of the product! If we were building a multiplier that computed the full product, we'd see many differences between a multiplier that handles unsigned operands and one that handles two's complement operands, but these differences only affect how the high half of the product is computed.

We've created a test jig to help debug your multiplier. Instead of including the checkoff file, use

```
.include "/mit/6.004/jsim/lab3_test_mult.jsim"
```

This test jig includes test cases for

all combinations of (0, 1, -1)*(0,1,-1),
2^i*1 for i = 0, 1, …, 31
-1*2^i for i = 0, 1, …, 31
(3 << i) * 3 for i = 0, 1, …, 31

When you've completed your design, you can use lab3checkoff_10.jsim to test your improved ALU implementation and record your successful completion of the optional multiplier. This checkoff file contains all the tests from lab3checkoff_6.jsim plus the multiplier test suite.

Design Note: Combinational multipliers implemented as described above are pretty slow! There are many design tricks we can use to speed things up – see the appendix on "Computer Arithmetic" in any of the editions of ***Computer Architecture** A Quantitative Approach* by John Hennessy and David Patterson (Morgan Kauffmann publishers).

Lab3 ALU

10/5

Read WP on ALUs

- 2s Complement
- Subtraction A + ¬B
- input registers
- Control unit

How multiple w/ 32

TA 32 seperate wires

Support a # of fns by breaking it down

Have st cell lib

Use gut sim

Iterators for multiple gates

- since each bit seperate

Provided test jigs

②

Boolean

Use 32 of 4-1 multiplexer

Right so for each bit using TT

Multiplexor $Z = (A \cdot \bar{S}) + (B \cdot S)$

? But this is 4

B	A	Y
0	0	d
0	1	c
1	0	b
1	1	a

← So just output D? which is what? Some sort of 4 bit boolean as in the mux TT

[Its so interesting Muxes built w/ math - but now do math on top]

So A, B are input Its the a,b,c,d that are selectors

1000 is AND

∴ just means about TT out

So

$A = 1$
$B = 1$

then TT output for 1000 is 1

A = 1	B = 0	1000	is 0
0	1	"	0
0	0	"	0

Interesting... * Remember it selects row of TT to output

Don't really see why works - some special math property - well reverse of how MUXs normally work - or how we were taught to think about them.

Just build it

* Watch order!

- sometimes → sometimes ←

(4)

~~B~~ My "A" is actually B

(✓) Bool works

Arith

3 outputs

Z when all outs are 0

V true when overflow

N when neg

thought this was 2s complement so had first bit as that

Overflows if sign of result differs from sign of operands

$$V = XA_{31} \cdot XB_{31} \cdot \overline{S}_{31} + \overline{XA_{31}} \cdot \overline{XB_{31}} \cdot S_{31}$$

↑ high order bit

ALUFN 0 for add

1 subtract

5

little endian

- bit 31 msb
  0 Lsb

So chart of what to bild

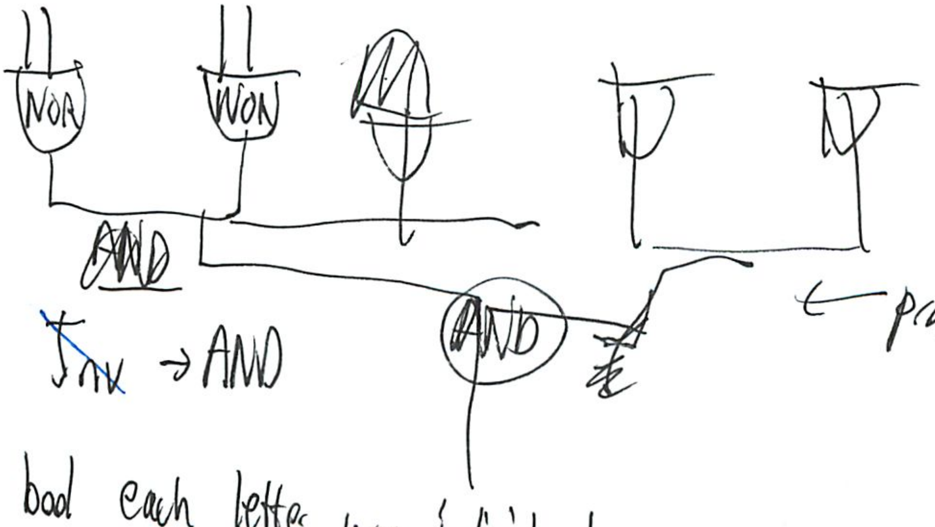
Subtract invert and add 1

Need to bild 32 NOR

So NOR

	NOR
00	1
01	0
10	0
11	0

So for 4 ins



So for bad each letter was individual

Oh can do NOR 4

6

NAND = ~~INV a~~ AND → INV

So need 2 ANDS.

Can do 2 NANDS ??

?? will go 1 way invert it back

↓

TA 2 ANDS works - can you make it faster?

$\overline{\overline{ab}\,\overline{cd}} = \bar{a}\cdot b$ or cd

$\overline{\overline{a+b}\,\overline{cd}} = a+b+c+d$

$\overline{\overline{\overline{ab}+\overline{cd}}\;\overline{ab}+\overline{cd}} =$

$\overline{\overline{a+b}\,\overline{c+d}} + \overline{e+f}\,gh$

$\overline{a+b}\,\overline{c+d}$ + ~~d~~

=

Let me try w/ TT

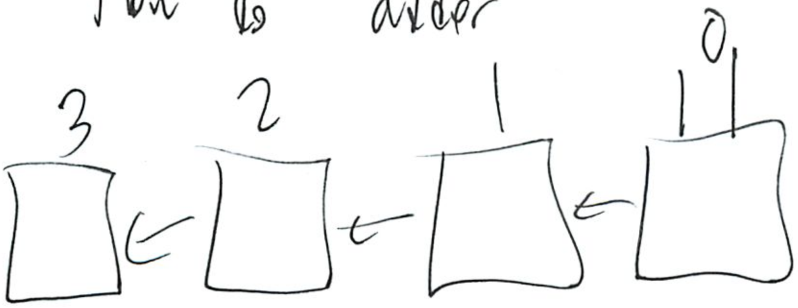
	AND	NAND	2 AND	2NAND	
00	0	1	0	11	0
01	0	1	0	11	0
10	0	1	0	11	0
11	1	0	1	00	1

= AND + INV

TA says not right
- didn't say why

Go w/ 2 ANDs for Now

How do adder



How get z v n i

In adder - seems they want

8 So for S inv all and

~~NAND~~ AND all togher

NOR is same ↑

Thats why built 32 bit NOR!

wire is connect!

Now have somme error

I was offsetting stuff wrong

My NOR was not considering the 3rd part

: NOR those 2 results?

Or we are Nor-ing each bit?
^ XOR-ing

Since is 32 out

~~Order of t O~~ ~~Ord~~ Order of Ops

() first then +

like normal

I had a bunch of small errors

XORs not NOR

↑ when have 32 seperate of them

↑ here elsewhere

Got a bunch more working...

3 - 5 = -2

✓ Got it to work! - Adder

---

3 Compare Compare operands. Do A - B and look at results

Higher order 31 bits are always 0 ←

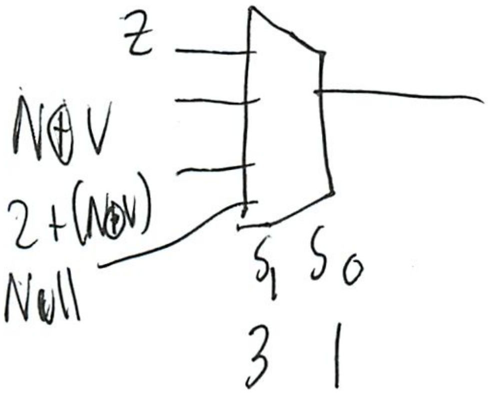
LSB (#0) is the output

LLUFN 3, 1 used to pick

Shows up late

So fix the ones at

Then mux based on values ?



Got working fast

Wrong

Swap S_0 S_1

Oh ~~a~~ ⊕ is XOR duh...

So lab actually totally changed - need to redownload!

Re-check boolean ✓

adder ✗

F2 − F0 = E2

= 2 get −2 ?!

ii

I thought this was right....

Check compare fast

5, 5 CmpEQ

Should be 1 is 0

Z is 0

? So roll over from last time?

How is adder diff on later bits

$$\underbrace{\begin{matrix}00000\\11111\end{matrix}}_{\text{different}}\quad\begin{matrix}\rightarrow\\\rightarrow\end{matrix}\quad\underbrace{\begin{matrix}00010\\00010\end{matrix}}_{\text{Same}}$$

TAs went home...

Z false ✓

N is 1 - should it be? - No
- but factor of else

V is 0 - should not be

Oh alufns might have been different here

✓ Fixed adder

Now compare - Seems to be working

No LT is Wot working - fixed earlier problem

n, V both 0

hmm

Selector not working? - its pretty far in...

Oh flip 1 and 3

✓ Compare done

Shift

A = data

B = how much shift

	ALUFN 1	0
SHL	0	0
SHR	0	1
SRA	1	1

So ALUFN() is 0 1
left right

1 sign extension

Wow divide thing is cool

- So do both and mux result

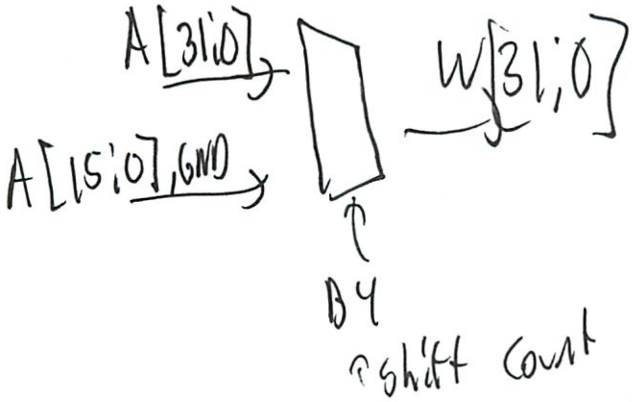
Cascade shifts of 2

So 13 bit is

$\xrightarrow{8} \xrightarrow{4} \xrightarrow{1}$

So shifter is cascade of multiplexer

But how connect togeter ?



oh so this is most

? So how combine ?

14

Oh A[15,0] GND #16

is use first 16 bits then fill rest w/ GND - ~~#~~ I see

Ok just connect them together

lots of individual muxes

(Interface for jsim should be better - how I visualize it)

So how compare

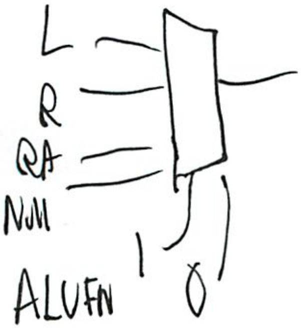
That is SHL

- need to select

Then do shift right

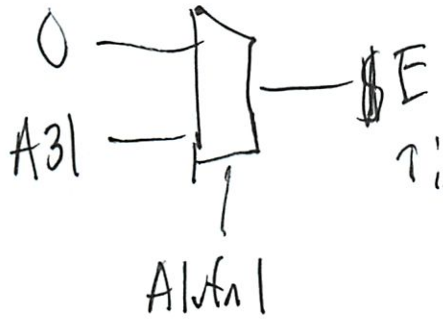
Do select first

- mux



(15)

~~A~~ For SRA / SAR make a wire w/
0 or A31



↑ instead of gnd before

↑ take a out of other one

~~(Most of datalog~~

⊗ Majorly wrong

~~A~~ Well not really 000...01 expected
actual 100...00

Why does it give z, n, v samples?

Mux may be wrong
- Yeah bilt it wrong

↳ Flip the A and GND

16

Also need to switch A[31:16]

31:8

⋮

✓ shifter

Now try checkoff

✓ 6 points

6.004 On-line: Questions for Lab 3

When you're done remember to save your work by clicking on the "Save" button at the bottom of the page. You can check if your answers are correct by clicking on the "Check" button.

When entering numeric values in the answer fields, you can use integers (1000), floating-point numbers (1000.0), scientific notation (1e3), or JSim numeric scale factors (1K).

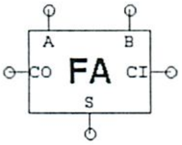
Problem 1. lab3checkoff_10.jsim tests your ALU circuitry by applying 169 different sets of input values. These questions explore how those values were chosen.

No designer I know thinks testing is fun -- designing the circuit seems so much more interesting than making sure it works. But a buggy design isn't much fun either! Remember that a good engineer not only *knows* how to build good designs but also *actually* builds good designs, and that means testing the design to make sure it does what you say it does.

An obvious way to test a combinational circuit is to try all possible combinations of inputs, checking for the correct output values after applying each input combination. This type of *exhaustive test* proves correct operation by enumerating the truth table of the combinational device. This is a workable strategy for circuits ~~with a~~ few inputs but quickly becomes impractical for circuits with many inputs. By taking advantage of information about how the circuit is constructed we can greatly reduce the number of input combinations needed to test the circuit.

Boolean reduction? - CNF? 6.005

The ripple-carry adder architecture suggested in Lab 3 uses 32 copies of the **full adder** module to create a 32-bit adder. Each full adder has 3 inputs (A, B, CI) and two outputs (S, CO):



2^3 ✓

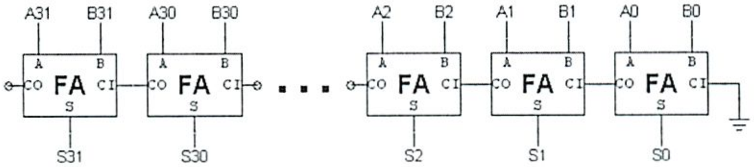
A. A single *test vector* for the full adder consists of 3 input values (one each for A, B and CI) and 2 output values (S and CO). To run a test the input values from the current test vector are applied to the device under test and then the actual output values are compared against the expected values listed by the test vector. This process is repeated until all the test vectors have been used. Assuming we know nothing about the internal circuitry of the full adder, how many test vectors would we need to exhaustively test its functionality?

Number of test vectors to exhaustively test full adder?: □

B. Consider a 32-bit adder with 64 inputs (two 32-bit input operands, assume CIN is tied to ground as shown in the diagram below) and 32 outputs (the 32-bit result). Assume we don't know anything about the internal ~~circuitry and~~ so can't rule out the possibility that it might get the wrong answer for any particular combination of inputs. ~~In other words, just because~~ the adder got the correct answer for 2 + 3 doesn't allow us to draw any conclusions about what answer it would get for 2 + 7. If we could apply one test vector every 100ns, how long would it take to exhaustively test the adder?

Time to exhaustively test 32-bit adder? (in years): □

C. Shown below is a schematic for a 32-bit ripple-carry adder.



$$\frac{2^{64} \cdot 100ns}{ns\ in\ a\ year}$$

58494 ✓

Except for the carry-in from the bit to the right, each bit of the adder operates independently. We can use this observation to test the adder bit-by-bit and with a bit of thought we can actually run many of these tests in parallel. In this case the fact that the adder got the correct answer for 2 + 3 actually tells us a lot about the answer it will get for 2 + 7. Since the computation done by adder bits 0 and 1 is same in both cases, if the answer for 2 + 3 is correct, the low-order two bits of the answer for 2 + 7 will also be correct.

So our plan for testing the ripple-carry adder is to test each full adder independently. When testing bit N we can set A[N] and B[N] directly from the test vector. It takes a bit more work to set CI[N] to a particular value, but we can do it with the correct choices for A[N-1] and B[N-1].

If we want to set CI[N] to 0, what values should A[N-1] and B[N-1] be set to? If we want to set CI[N] to 1? Assume that we can't assume anything about the value of CI[N-1].

when is carry = 0?

00
~~01~~
~~10~~) I think these should also work
11

Values of A[N-1] and B[N-1] to make C[N]=0?: --select answer-- ✓

Values of A[N-1] and B[N-1] to make C[N]=1?: --select answer-- ✓

D. Here's a proposed set of 10 test vectors which we'd like to use as an exhaustive test of the 32-bit ripple carry adder.

Test Vector #	A[31:0]	B[31:0]
1	0x00000000	0x00000000
2	0x55555555 (even bits)	0x00000000
3	0x00000000	0x55555555 (even bits)
4	0x55555555 (even bits)	0x55555555 (even bits)
5	0xAAAAAAAA (odd bits)	0x00000000
6	0x00000000	0xAAAAAAAA (odd bits)
7	0xAAAAAAAA (odd bits)	0xAAAAAAAA (odd bitts)
8	0xFFFFFFFF	0xFFFFFFFF
9	0x00000001	0xFFFFFFFF
10	0xFFFFFFFF	0x00000001

To see if the tests are exhaustive, fill in the following table, indicating which test vectors tested which combinations of input values. There are separate tables below for even adder bits (bits 2, 4, 6, ...) and odd adder bits (1, 3, 5, ...). Ignore adder bit 0 when filling in the "Even adder bits" entries since it is a special case with its CIN tied to ground.

not at

Then yet

don't yet!

Even adder bits				Odd adder bits			
A	B	CIN	Tested?	A	B	CIN	Tested?
0	0	0	Yes, by A = 0x00000000 and B=0x00000000	0	0	0	Yes, by A = 0x00000000 and B=0x00000000
0	0	1	Yes, by A=0xAAAAAAAA and B=0xAAAAAAAA	0	0	1	Yes, by A=0x55555555 and B=0x55555555
0	1	0	--select answer-- 000 555 ✓	0	1	0	--select answer-- ✓ 000 AAA
0	1	1	--select answer-- 001 FFF	0	1	1	--select answer-- 001 FFF
1	0	0	--select answer-- 555 000	1	0	0	--select answer-- AAA 000
1	0	1	--select answer-- FFF 001	1	0	1	--select answer-- FFF 001
1	1	0	--select answer-- 555 555	1	1	0	--select answer-- AAA AAA
1	1	1	--select answer-- FFF FFF	1	1	1	--select answer-- FFF FFF

0

E. Three of the compare unit's inputs (Z, V and N) come from the adder/subtractor running in subtract mode. To test the compare unit, we'll need to pick operands for the adder/subtractor that generate all possible combinations of Z, V and N. It's easy to see that any combination with Z = 1 and N = 1 is not possible (the output of the adder cannot be negative and zero at the same time!). It also turns out that combinations with Z = 1 and V = 1 cannot be produced by a subtract operation.

For each of the combinations of Z, V and N shown below, choose the subtraction operation that will produce the specified combination of condition codes.

Subtraction that produces Z=0, V=0, N=0?: --select answer-- 06F - DEAD

Subtraction that produces Z=1, V=0, N=0?: --select answer-- 123 - 123

Subtraction that produces Z=0, V=1, N=0?: --select answer-- 800 - 001

Z: S are 0

V: overflows (complex formula)

N: negative

Subtraction that produces Z=0, V=0, N=1?: --select answer-- DEAD - 0005

Subtraction that produces Z=0, V=1, N=1?: --select answer-- 7FF - FFF

Check Save

source: on_line_questions.py, lab3questions.xdoc

✓ Questions

Cost/Performance Tradeoffs:

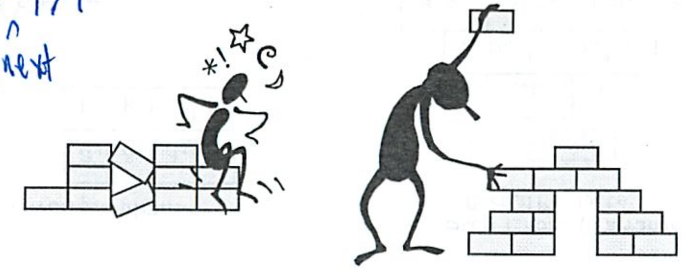
a case study

10/6

No lecture Tue, ~~[illegible]~~

Digital Systems Architecture 1.01

Quiz 2 next Fri

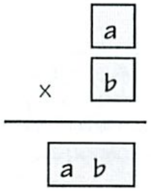


Lab #3 due tonight!

Tactally 6Am tomorow

Try to formally characterize problem
- ins
- outs

Binary Multiplication



n bits

n bits

2n bits to store product

since $(2^n-1)^2 < 2^{2n}$

EASY PROBLEM: design combinational circuit to multiply tiny (1-, 2-, 3-bit) operands...

HARD PROBLEM: design circuit to multiply BIG (32-bit, 64-bit) numbers

want to have big multipliers

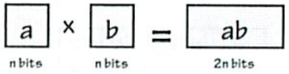
We can make big multipliers out of little ones!

Engineering Principle:
Exploit STRUCTURE in problem.

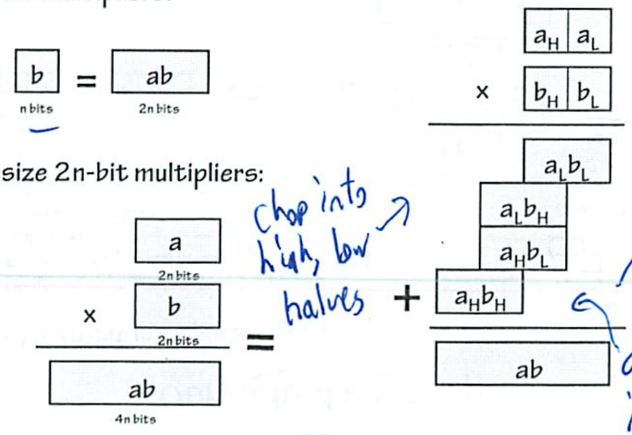
Inductive proof

Making a 2n-bit multiplier using n-bit multipliers

Given n-bit multipliers:



Synthesize 2n-bit multipliers:



chop into high, low halves

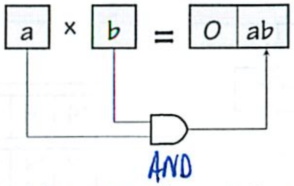
set of partial products

offset like in elementary school

Our Basis:

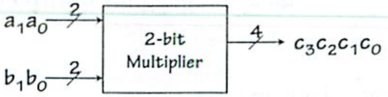
n=1: minimalist starting point

Multiplying two 1-bit numbers is pretty simple:



← prereq start here

Of course, we could start with optimized combinational multipliers for larger operands; e.g.



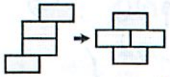
the logic gets more complex, but some optimizations are possible...

Can optimize - save some pico seconds

Our induction step:

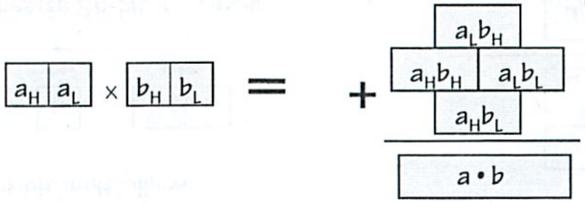
2n-bit by 2n-bit multiplication:

1. Divide multiplicands into n-bit pieces
2. Form 2n-bit partial products, using n-bit by n-bit multipliers.
3. Align appropriately
4. Add.



REGROUP partial products - 2 additions rather than 3!

math trick

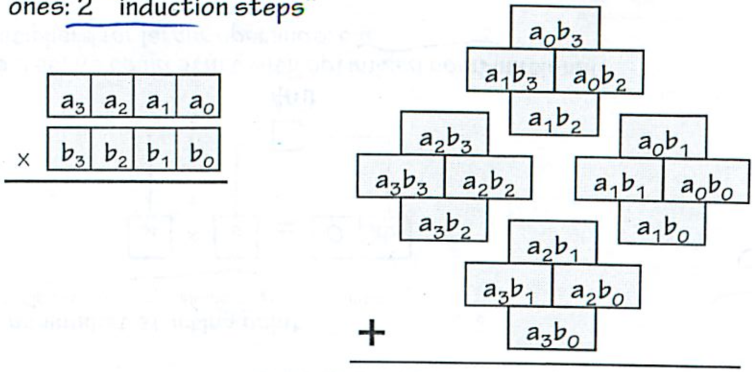


Induction: we can use the same structuring principle to build a 4n-bit multiplier from our newly-constructed 2n-bit ones...

Brick Wall view

of partial products

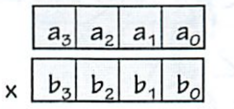
Making 4n-bit multipliers from n-bit ones: 2 "induction steps"



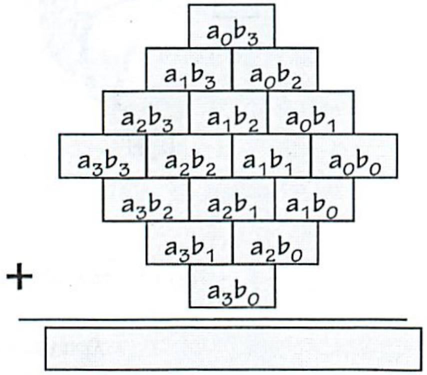
can repeat → larger

Multiplier Cookbook: Chapter 1

Given problem:

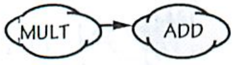


Step 1: Form (& arrange) Partial Products:



a_0b_3
a_1b_3 a_0b_2
a_2b_3 a_1b_2 a_0b_1
a_3b_3 a_2b_2 a_1b_1 a_0b_0
a_3b_2 a_2b_1 a_1b_0
a_3b_1 a_2b_0
a_3b_0
+

Subassemblies:
- Partial Products
- Adders



Step 2: Sum

throughput / latency as n grows

Performance/Cost Analysis

"Order Of" notation:

"g(n) is of order f(n)" $g(n) = \Theta(f(n))$

upper + lower bound

$g(n) = \Theta(f(n))$ if there exist $C_2 \geq C_1 > 0$, such that for all but finitely many integral $n \geq 0$

$$\underbrace{c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)}_{g(n) = O(f(n))}$$

just 2nd inequality - upper bound only

Example:

$n^2+2n+3 = \Theta(n^2)$

since

$n^2 \leq (n^2+2n+3) \leq 2n^2$

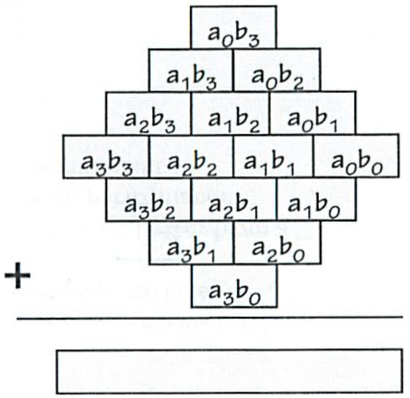
"almost always"

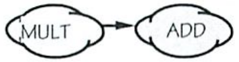
Θ(...) implies both inequalities; O(...) implies only the second.

Partial Products:	n^2	=	$\Theta(n^2)$
Things to Add:	2n-2	=	$\Theta(n)$
Adder Width:	2n	=	$\Theta(n)$
Hardware Cost:	?	=	$\Theta(n^2)$

Latency: $O(n^2)$??

Observations:





$\Theta(n^2)$ partial products.
$\Theta(n^2)$ full adders.
Hmmm.

Need a bundle of wires to connect

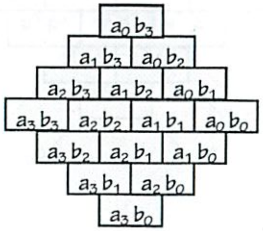
- Can we repackage these 2 units?

Repackaging Function

Engineering Principle #2:

Put the Solution where the Problem is.

$\Theta(n^2)$ partial products.
$\Theta(n^2)$ full adders.

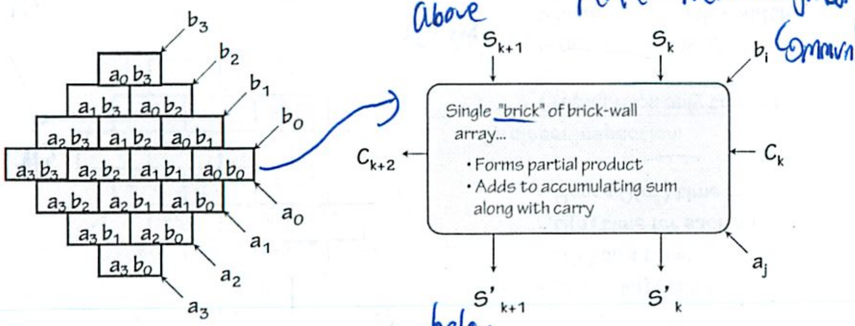


How about n^2 blocks, each doing a little multiplication and a little addition?

Goal:

Array of Identical Multiplier Cells

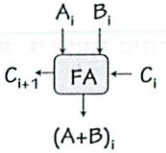
Try to get module perfer near neighbor Communications



above

Single "brick" of brick-wall array...
- Forms partial product
- Adds to accumulating sum along with carry

below

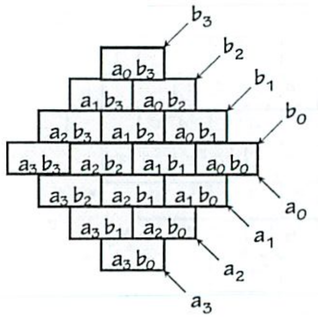


Necessary Component: Full Adder

Takes 2 addend bits plus carry bit. Produces sum and carry output bits.

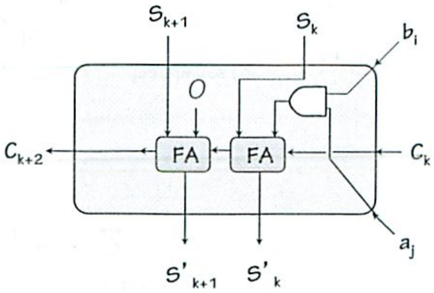
CASCADE to form an n-bit adder.

Design of 1-bit multiplier "Brick":



Array Layout:
- operand bits bused diagonally
- Carry bits propagate right-to-left
- Sum bits propagate down

does multiplication and adds



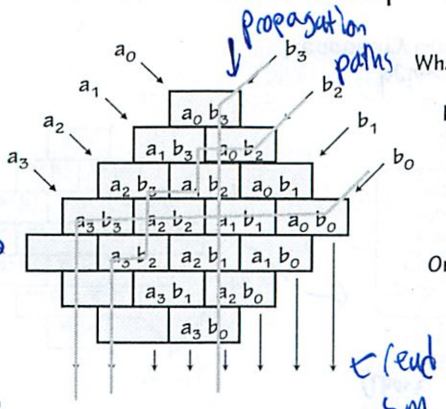
Brick design:
- AND gate forms 1x1 product
- 2-bit sum propagates from top to bottom
- Carry propagates to left

Wastes some gates... but consider (say) optimized 4x4-bit brick!

could optimize more
- but do theory here

Latency revisited

Here' s our combinational multiplier:



Carry bits left over - just put more yellow bricks in

What' s its propagation delay?

Naive (but valid) bound:

- $O(n)$ additions
- $O(n)$ time for each addition
- Hence $O(n^2)$ time required

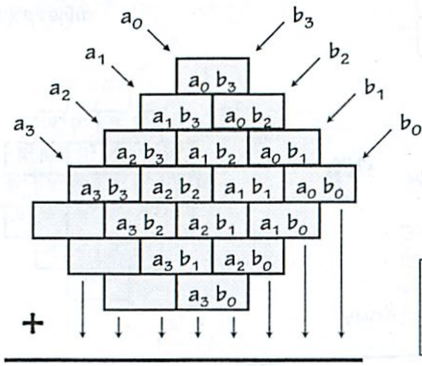
On closer inspection:

- Propagation only toward left, bottom
- Hence longest path bounded by length + width of array:

$O(n+n) = O(n)!$

Multiplier Cookbook: Chapter 2

Combinational Multiplier:



Hardware for n by n bits:	$\Theta(n^2)$
Latency:	$\Theta(n)$
Throughput:	$\Theta(1/n)$

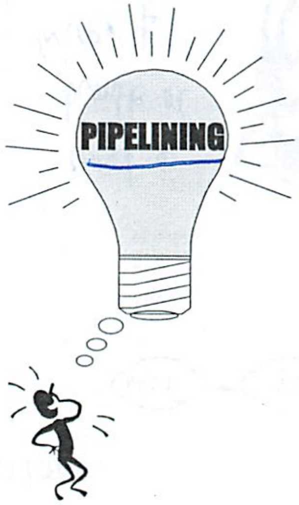
Note: lots of tricks are available to make a faster combinational multiplier...

Combinational Multiplier:

best bang for the buck?

Suppose we have LOTS of multiplications.

Can we do better from a cost/performance standpoint?



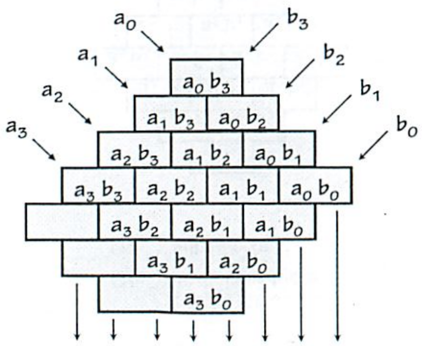
The Pipelining Bandwagon...

where do I get on?

WE HAVE:

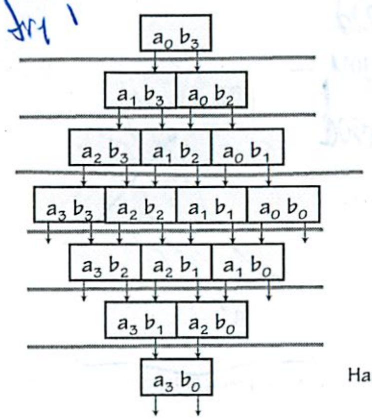
- Pipeline rules - "well formed pipelines"
- Plenty of registers
- Demand for higher throughput.

What do we do? Where do we define stages?



Stupid Pipeline Tricks

natural, 1st try!



$a_0 b_3$
$a_1 b_3$ $a_0 b_2$
$a_2 b_3$ $a_1 b_2$ $a_0 b_1$
$a_3 b_3$ $a_2 b_2$ $a_1 b_1$ $a_0 b_0$
$a_3 b_2$ $a_2 b_1$ $a_1 b_0$
$a_3 b_1$ $a_2 b_0$
$a_3 b_0$

gotta break that long carry chain!

No permance improvements - gets worse actually

Stages: $\Theta(n)$

Clock Period: $\Theta(n)$

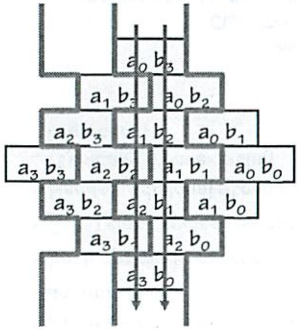
Hardware cost for n by n bits: $\Theta(n^2)$

Latency: $\Theta(n^2)$

Throughput: $\Theta(1/n)$

have not broke the long propagation paths

Even Stupider Pipeline Tricks



$a_0 b_3$
$a_1 b_3$ $a_0 b_2$
$a_2 b_3$ $a_1 b_2$ $a_0 b_1$
$a_3 b_3$ $a_2 b_2$ $a_1 b_1$ $a_0 b_0$
$a_3 b_2$ $a_2 b_1$ $a_1 b_0$
$a_3 b_1$ $a_2 b_0$
$a_3 b_0$

WORSE idea:

- Doesn't break long combinational paths
- NOT a well-formed pipeline…
  - … different register counts on alternative paths
  - … data crosses stage boundaries in *both directions!*

Back to basics:

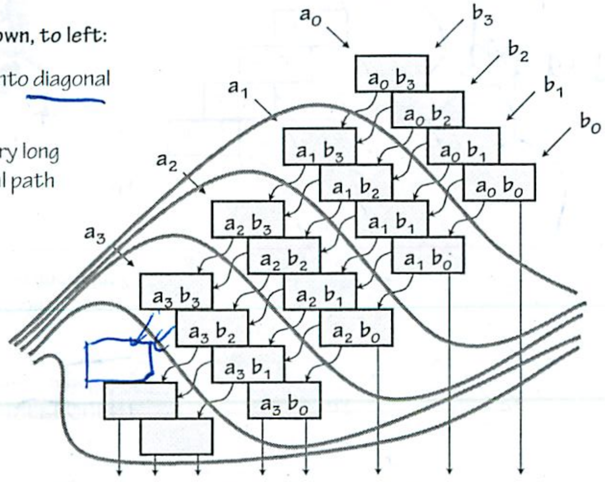
what's the point of pipelining, anyhow?

Breaking O(n) combinational paths

LONG PATHS go down, to left:

- Break array into diagonal slices
- Segment every long combinational path

No O(n) path here

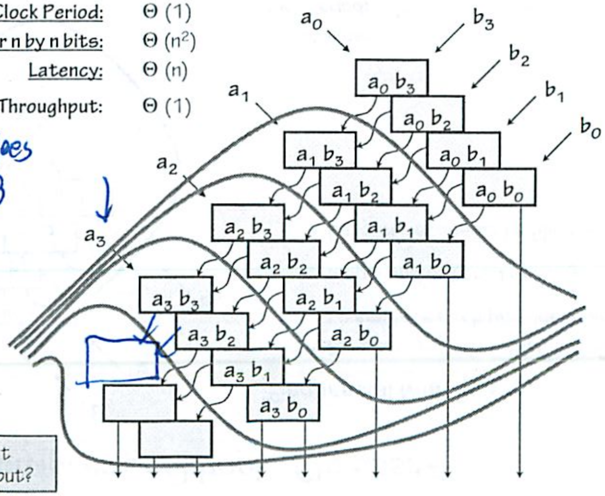


GOAL: Θ (n) stages; Θ (1) clock period!

Multiplier Cookbook: Chapter 3

Stages:	Θ (n)
Clock Period:	Θ (1)
Hardware cost for n by n bits:	Θ (n^2)
Latency:	Θ (n)
Throughput:	Θ (1)

note goes through a3 inputs!



- Well-formed pipeline (careful!)
- Constant (high!) throughput, *independently of operand size.*

… but suppose we don't need the throughput?

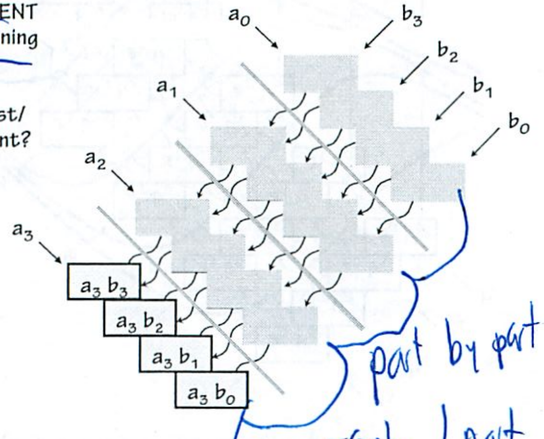
make it clocked sequential circuit

if only 1 multiplication to perform?

Moving down the cost curve...

Suppose we have INFREQUENT multiplications... pipelining doesn't help us.

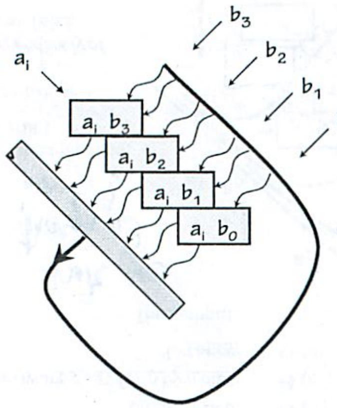
Can we do better from a cost/ performance standpoint?



part by part

-Only 1 part active at a time

Multiplier Cookbook: Chapter 4



Sequential Multiplier:

- Re-uses a single n-bit "slice" to emulate each pipeline stage
- a operand entered serially
- Lots of details to be filled in...

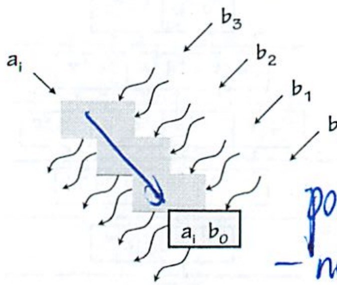
Stages:	1
Clock Period:	$\Theta(1)$ (constant!)
Hardware cost for n by n bits:	$\Theta(n)$ linear cost
Latency:	$\Theta(n)$
Throughput:	$\Theta(1/n)$

next trick

(Ridiculous?)

Extremes Dept...

Cost minimization: how far can we go?



possible

- not very pratical at all

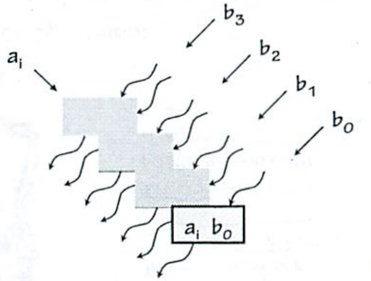
Suppose we want to minimize hardware (at any cost)...

- Consider *bit-serial!*
  - Form and add 1-bit partial product per clock
  - Reuse single "brick" for each bit b_j of slice;
  - Re-use slice for each bit of a operand

Multiplier Cookbook: Chapter 5

Bit Serial multiplier:

- Re-uses a single brick to emulate an n-bit slice
- both operands entered serially
- $O(n^2)$ clock cycles required
- Needs additional storage (typically from existing registers)



Stages:	$\Theta(1/n)$
Clock Period:	$\Theta(1)$ (constant)
Hardware cost for n by n bits:	$\Theta(1) + ?$ need a lot of registers $O(n)$
Latency:	$\Theta(n^2)$
Throughput:	$\Theta(1/n^2)$

Summary:

Regular/

Scheme:	$	Latency	Thruput
Combinational	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1/n)$
N-pipe	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$
Slice-serial	$\Theta(n)$	$\Theta(n)$	$\Theta(1/n)$
Bit-serial	$\Theta(1)^*$	$\Theta(n^2)$	$\Theta(1/n^2)$

+registers

Lots more multiplier technology: fast adders, Booth Encoding, column compression, ...

10/6

Lab 3
Checkoff

Just reviewed my code + explained

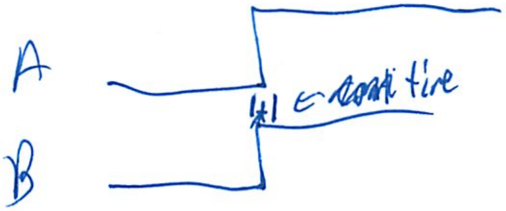
Did not ask me about my And vs NAND

6.004 Recitation

10/7

Arbitration + Meta Stability

① Can we design a circuit that determines which of A or B was 1st



"the Jeporty problem"

if unbounded time → Yes

if bounded time → No

t is continuous variable

try to map to discrete time { yes
no

we require an answer – no "unsure" interval

Same if ask if $V > 2.0$

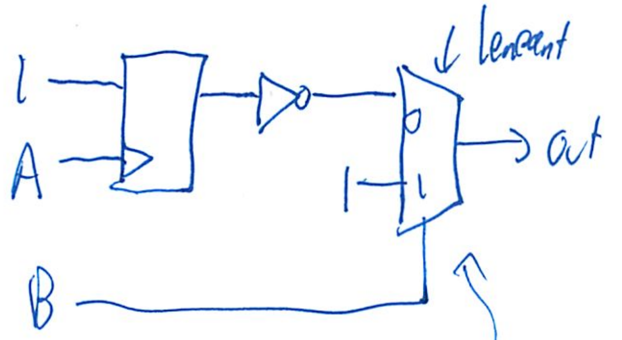
– can get arbitrarily close to 2

2

What you do is take difference and amplify exponentially
if difference is very small still hard

2 av
Is there an issue of metastability
Does it work?

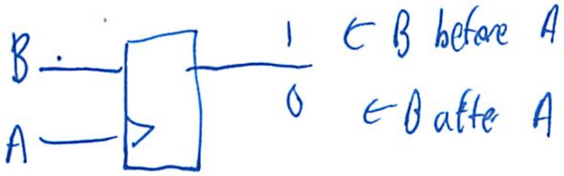
Metastability is bad for flip flop



Can't go metastable
Since D never
changes
but wrong since will
always give ~~[illegible]~~ 1st to B

All circuits leniant

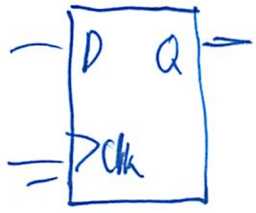
Combo logic does not have metastability issues



Goes metastable when violate dynamic displine
└ input signal can't change during t_{setup} before clock edge
or t_{hold} after clock edge

Metastability = no valid digital output

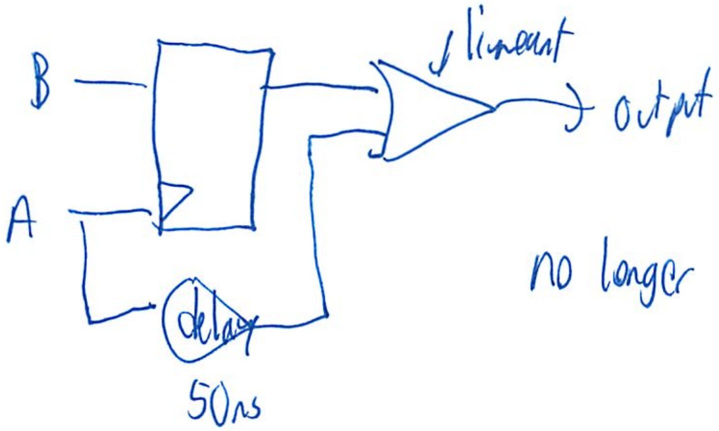
Flipflop review



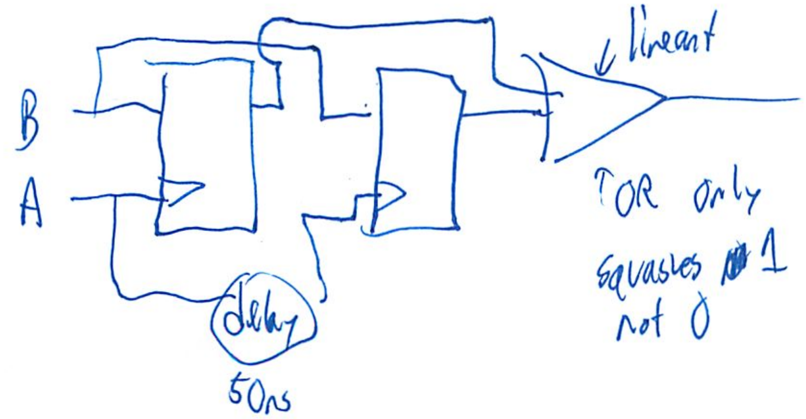
On rising clock edge sample D and then continue outputting that to Q till next clock rise

All flip flops start at 0 or X
└ invalid

Another problem

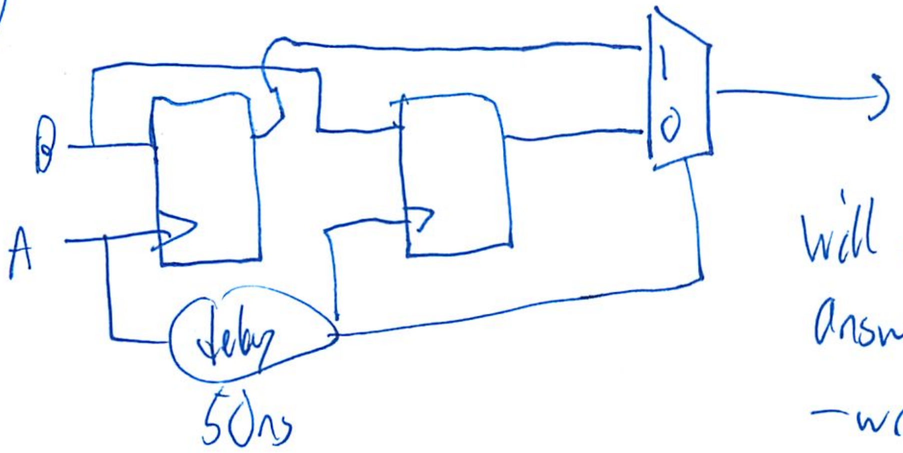


no longer metastable after 50ns



TOR only squashes 1 not 0

Either 1 or other metastable
But wrong answer w/ OR



Will always output same answer
-wrong

So we either have metastability or wrong answer in bounded time

---

Can we build 1 if exactly A + B is pressed ?
0 otherwise

? Is that not just XOR?

So how do you sort if problem easy or hard?
This question does not involve past history of inputs.

Can we produce where goes 1 after 2nd input has happened?

Just AND gate!

We could write Truth table - so easy problem

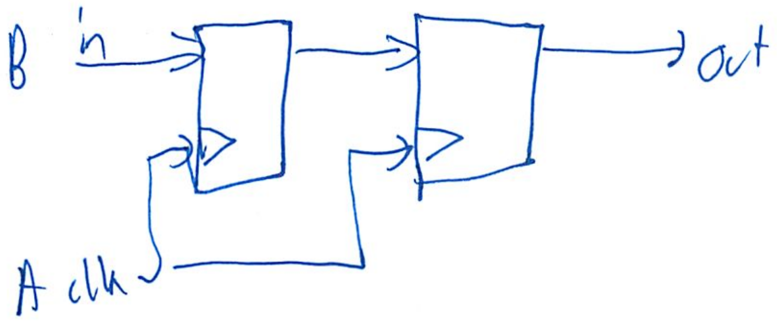
Can you flip a coin and report back in <60seconds?

- Maybe not!

Can you ask clock to ~~[illegible]~~ report at 55 ns

- don't know if it went ding then

The "sol" we propose <u>pulse syncronizer</u>



P(output is metastable) =

Can choose A clock to get ~~[illegible]~~ specified probability

~30ns many years ago

~1ns now

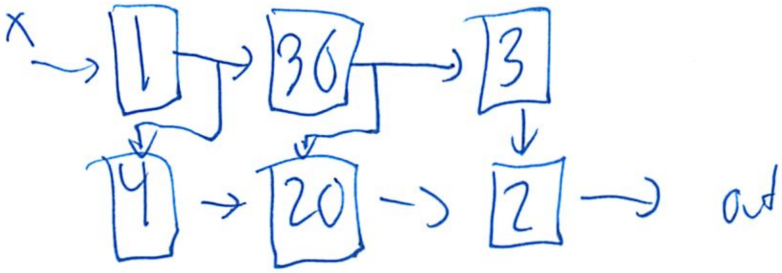
So sync the button push to line up w/ clock

So far lower prob of metastabilty button

6

Throughput + Latency

- start today, do more next time



Latency / t_{PD} of this

- ~~[illegible]~~ sum ~~[illegible]~~ all possible paths
- take max
- 53 ~~[illegible]~~ units
- so takes t_{PD} for output to reflect new in
- ~~that~~ is the latency of the circuit

Throughput max rate can get new values at output

$$\frac{1}{L} = \frac{1}{t_{PD}}$$

7

Pipelining attempts to ↑ Throughput

- comes at expense of Latency

  ← will never decrease

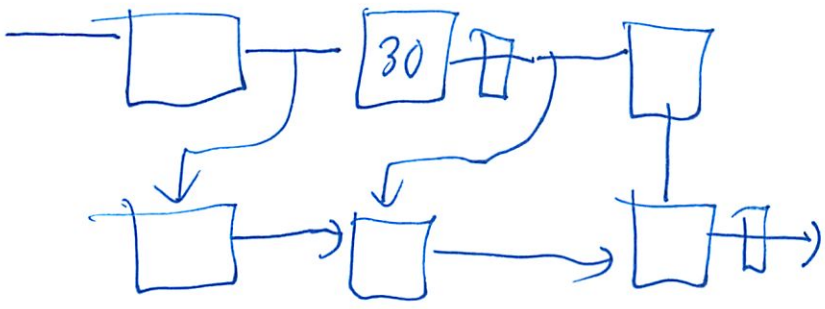
  but often makes it bigger

Best pipelining ↑ Throughput at same latency

Add registers to circuit

- always one on the output

- how can we add some anywhere else?



Count # of registers on each possible path

- 2 in above diagram

$T = \frac{1}{t_{clk}} = \frac{1}{31}$ ← get one ans every 31 units

$l = 2 \cdot t_{clk} = 62$ ← but particular ans takes 62 units

6.004 Recitation

10/12

Quiz Friday

- from history - some stuff on Q1 is on this Q2
  - or other way around

Combo logic

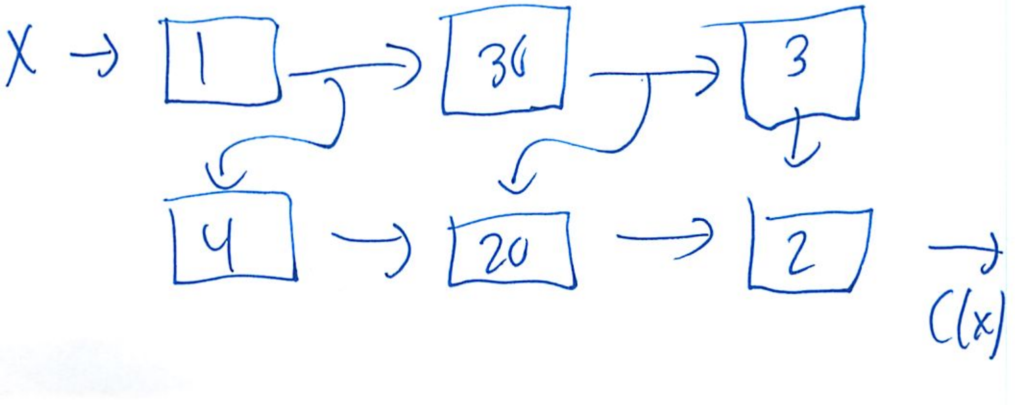
Latency t_{PD}

Throughput $T = \frac{1}{L} = \frac{1}{t_{PD}}$

Pipelining

tries to ↑ throughput

byproduct: latency stays same (best rare case) or goes ↓



Combo

$L = 53$

$T = \frac{1}{53}$

Want to ↑ throughput

- Pipeline!

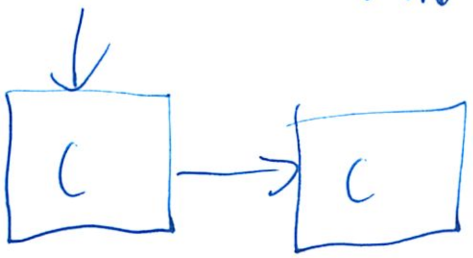
#1. Put register on output

- Clock once every 53 time units
- have not acomplished any thing yet

Combo	1-pipe
$l = 53$	$l = (\# \text{ pipeline stages}) \cdot t_{clk}$
$T = \frac{1}{53}$	$T = \frac{1}{t_{clk}}$
	$t_{clk} = 53$

Why is this interesting?

Since its a building block



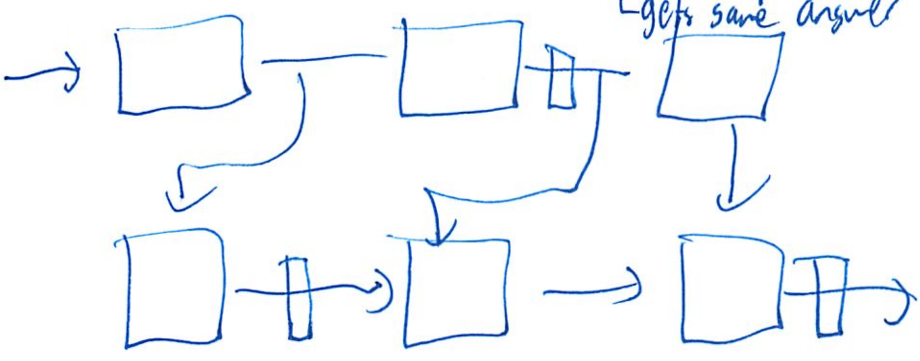
↑ can do this w/o having to change clk

③

Its the same: registers on output or registers on input

Is this a well formed pipeline?

└gets same answer



Can't registers on every possible path

Should all =

Here it should (and does) all = 2

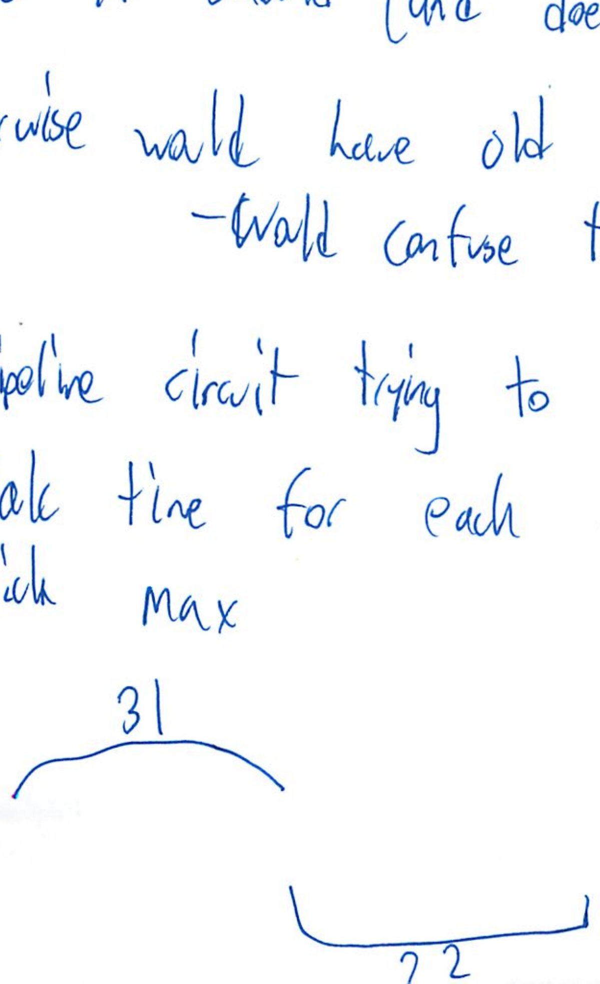
Otherwise would have old data

-would confuse time steps

In pipeline circuit trying to find t_{clk}

-Calc time for each section

-pick max



2 pipeline

$l = 2 \cdot 31 = 62$

$T = \frac{1}{31}$

$t_{clk} = 31$

Timing is based on longest clock edge

To make latency same try for all piplines stages

t_{PD} = ~~same~~ each other

Techniques ~~for~~ optimal pipelining

L-pipeline you always do at start or end

We know we can't do better than $t_{CLK} = 30$

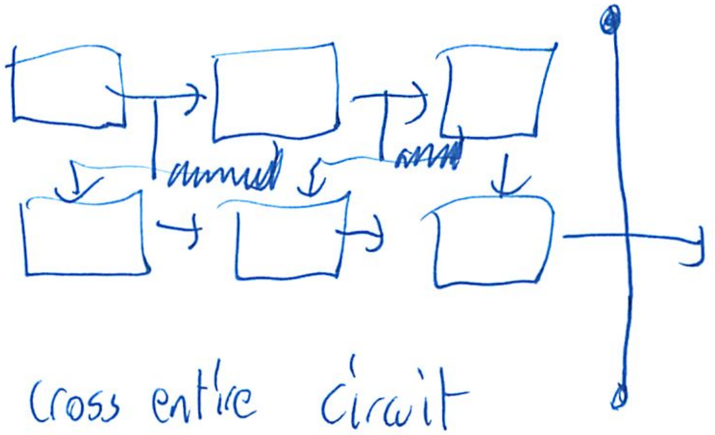
by only adding registers

And want to add min # of registers

Draw 2 dots - ~~on on~~ each

On either side of ins/outs

-draw contours

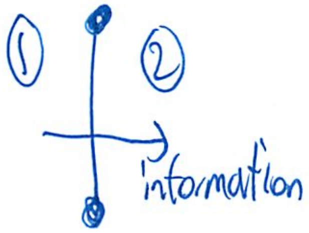


-need to cross entire circuit

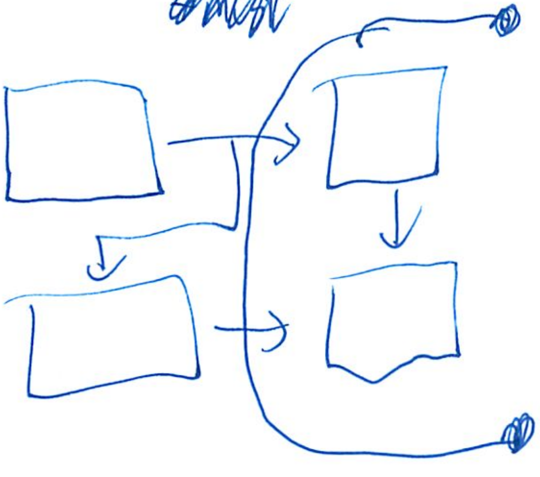
(5)

Have 2 sides

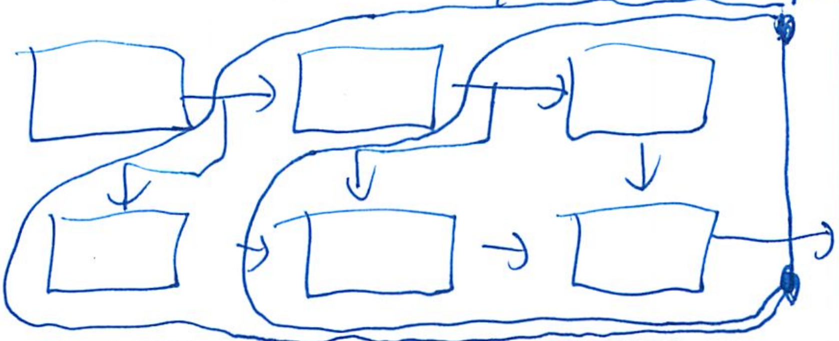
- all wires must cross from one side to other



Goal: have a register before and after longest element



would work
but not shortest!
need to draw it before/after splits



$l = 3 \cdot 30 = 90$

$T = \frac{1}{30}$

$t_{clk} = 30$

Rember we are also trying to min # registers
Cald also bild it so use more than min register

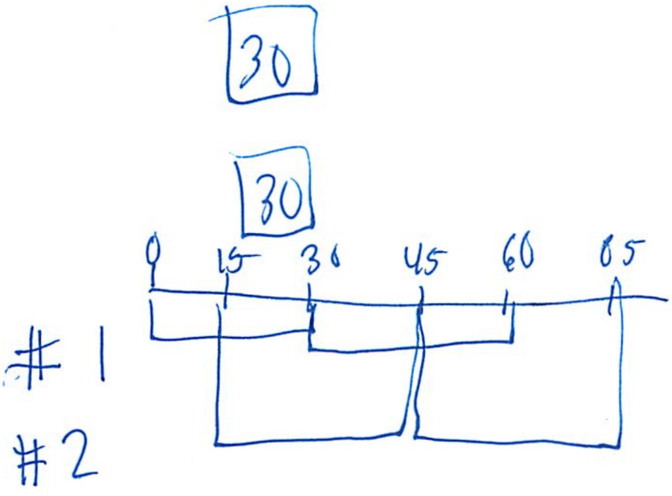
Countors: guarenteed to be well-formed if Follow the rules

On our chat: 3 different computations at once from 3 generations on inputs

---

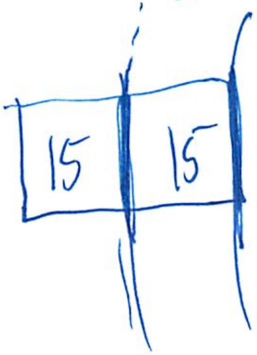
I really want $t_{clk} = 15$

But I'm rich so I can buy 2 × 30s
-paralism!



So multiplexers + latches to do this

So from pipeline register it looks like



So need to draw contours through middle as well

Qu 3 implementations of a multiplier

	I	L		
Cooker	1/T	5T	pipe	$1000
Sizzler	1/4T	4T	combo	
Gaunter	1/32T	32T	seq.	

Could use any combo of these
Want min $
Multiplexers, etc free

(8)

A) What is the max price for a Sizzler to still sell some?

No, if you have a target latency of 4T
-can't use any combo cooker and grunters
So max price for ∞

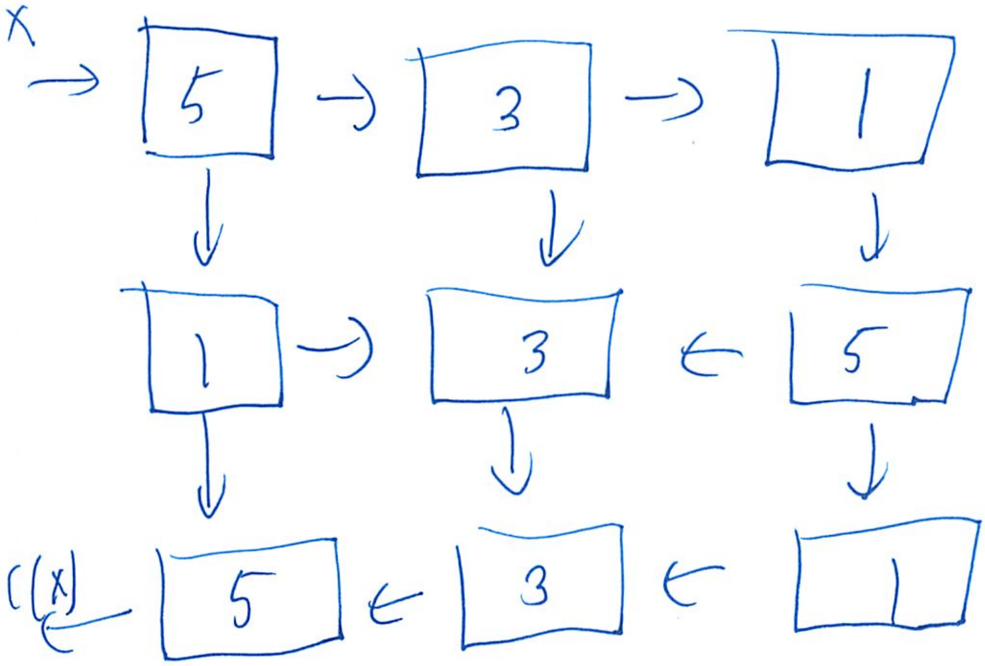
B.) Max price for ~~grunters~~ Sizzles to still sell cookers?

min = $251
? otherwise people buy a cooker

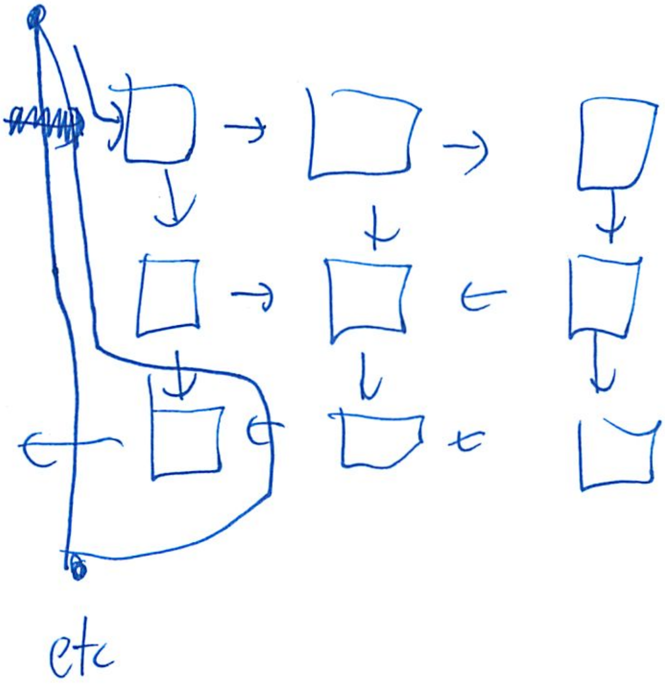
C) Max price for grunter?
No combo of grunters to be at least of price as cooker if you care about latency

Forgetting sizlers people will pay $999
if people perfectly happy w/ long latency +
Crummy throughput

Q2 Max throughput Min registers



Best we can do is 5



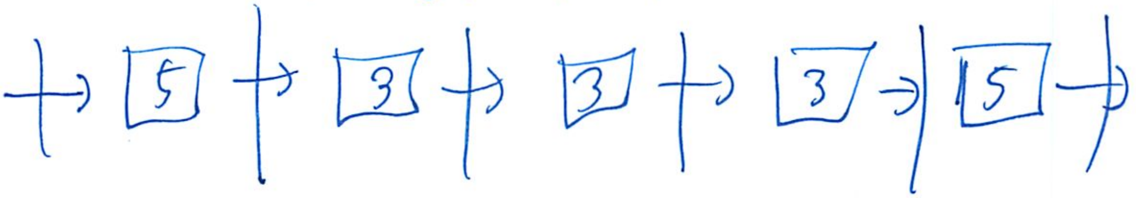
etc

10

To make easiest, redraw circuit

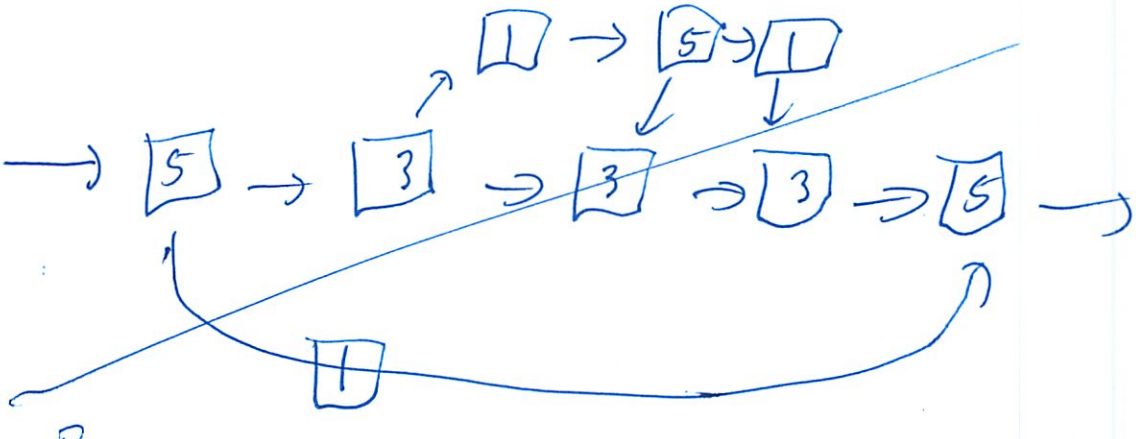
Draw critical path

- ~~shortest~~ longest way through
- so info goes l -> r



Want registers at every point along here

Need to draw the other circuit



↑ is kinda screwed up

Use original diagram to number

6.004 Concept Review

10/13

Combo logic

- putting together gates to make logic
- braindead way: sum of products
- NAND + NOR build everything
- trees good sometime

$\overline{AB} = \overline{A} + \overline{B}$ ← De morgan

$\overline{A}\,\overline{B} = \overline{A+B}$

- Don't cares
- Mux

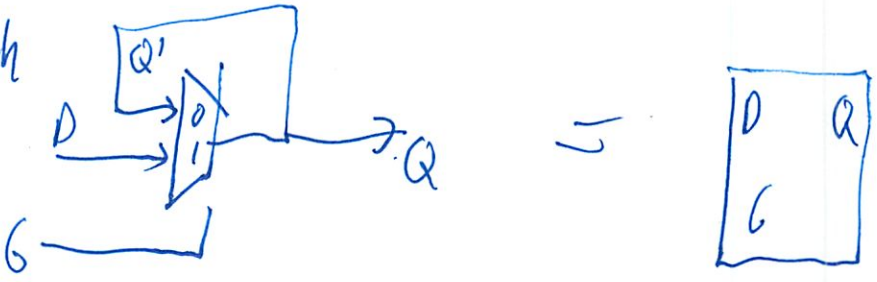
  — like a selector

- ROMs
  - general purpose look up tables
  - each output is on a line
  - MUX selects which one
  - long lines bad - so combine

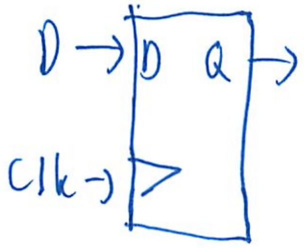
ROMs are not lenient

Sequential Logic

- adding state
- can store w/ capacitor
- Or just loop value around
- So make latch



- Lenient if have proper displine
- prevent transparency by using two = flip-flop



- Registers = groups of flip flops
- Discrete time w/ clock

3

Finite SMs

So build w/ flip flops like last week

But study w/ diagram

- every possible input from every state needed
  - arrow

Can bild in ROM

s state bits $= 2^s$ possible states

Need to build syncronizer to sync button presses

Can simplify equivilant FSMs

- like the ant problem from 6.01

---

t_{CD} = min of all paths - add up t_{CD}s from each el?

- not just visually shortest

$t_{PD} = \sum$ each t_{PD} ~~[illegible]~~ for each element for each path

take the max - so want longest path

$$\frac{t_{rise} + t_{fall}}{2}$$ goes 0→1

only care about __last__ gate output is hooked to

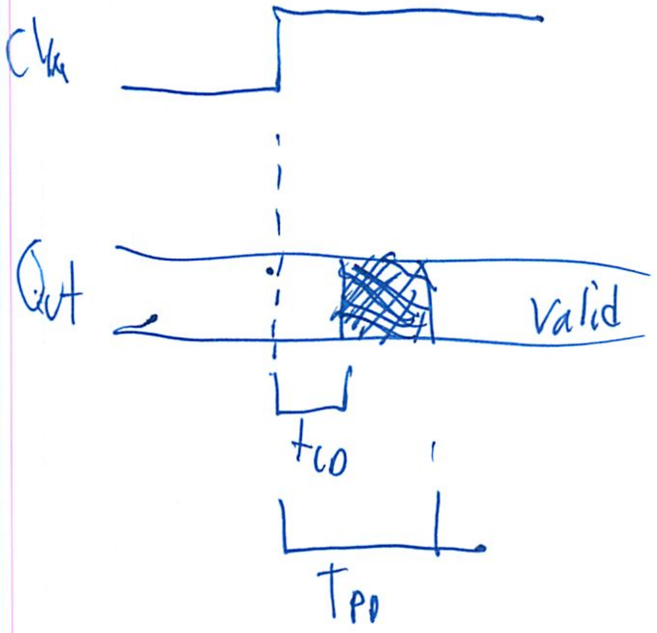
K maps let you reduce # gates
list stuff next to
Circle related stuff - write all stuff fixed
for 0 or 1

__Sync__

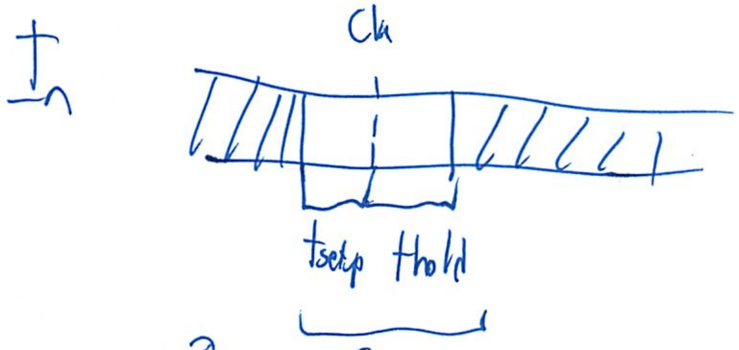
Need to know edge came first
- since no setup hold time for a button

Unsolvable - in bounded time
Since will go metastable
~~Can either be~~ if try to return an ans early →
it would be wrong
Lots of folk cures

Oh register = flip flop



Draw careful sequencing diagrams



when needs to be valid ?

So must be valid before clk edge

6

Always check

$$\sum t_{cd} \geq t_h$$ ← easy, lenghten Clk

$$t_{clk} \geq \left(\sum t_{pd}\right) + t_s$$ ← hard, don't go metastable

~~$\sum t_{cd} \geq t_{clk}$~~

~~I wish was exam~~

Check for each register path
- from one register to another

Set clk to lenght of longest one

Pipelining

-did a lot of this recently

Latency = input → output

Throughput = Rate at which outputs are processed

Each path Must go through same # registers draw lines to guarentee valid result

Put lines around slowest module

Can have 2 circuit elements with fancy circuit

- this is like a pipeline through it
- must include it in the lines

Some other exotic control structures

(I should review the building a SM question)

-did not do much

Cost / Performance trade offs

- exploit structure in problems
- like the 2n-bit multiplier
- the brick wall view

  (Never did this in recitation - do we need to know)
- Multipler brick
- Need to pipeline best place
- Could even have 1 section repeat = cheap to build
  - but slow!

8

6/13
Home

Jepordy problen

Metastability bad for flip flop
- violates dynamic disipline

So use pulse Syncronizer
- a very small prob metastability

Latency = t_{PD} for unpipelined $= k\, t_{clk}$ for pipelined

Throughput = $\frac{1}{t_{PD}}$ for unpipelined

$= \frac{1}{t_{clk}}$ otherwise

Can be interested in throughput, latency, both, or neither

Critical path = **longest** path through

1	/ 6
2	/ 10
3	/ 14
	/ 30

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

~~A~~ $\sum t_{CD} \geq t_H$

$t_{\text{clk}} \geq (\sum t_{pd}) + t_S$

6.004 Computation Structures
Spring 2011

Quiz #2: March 11, 2011

Name		*Athena login name*	*Score*
TA: Deborah, 34-303 □ WF 10 □ WF 11	TA: Arkajit, 34-302 □ WF 12 □ WF 1	TA: Caitlin, 34-301 □ WF 1 □ WF 2	TA: Eben, 34-302 □ WF 2 □ WF 3

Problem 1 (6 points): **Quickies and Trickies**

(A) NovaFlop, Inc advertises a reliable flipflop with an unusual guarantee: it may enter a metastable state if the dynamic discipline is not followed, but that when metastable state settles to a valid logic level, it will always be a 1 rather than a 0.

but not correct

Is their claim plausible? Circle one: [✓ YES] ... [✓ NO] ... Can't Tell

(B) MetaSure, Inc advertises a 2-input device that claims to produce a positive transition on its output within 1 ns after positive transitions have occurred on both of its inputs. + something

Is their claim plausible? Circle one: [✓ YES] ... NO ... Can't Tell

✓

(C) A latch is constructed from a 2-input lenient MUX having a propagation delay of 200ps and a contamination delay of 20ps, using the design shown in lecture. Give the minimum appropriate setup time specification for this latch.

oh comes around

Setup time (ps): 400ps

(D) Give the best achievable asymptotic throughput for a pipelined multiplier capable of multiplying two N-bit operands. Enter a number, a formula, or "CAN'T TELL".

Asymptotic throughput: Θ(1)

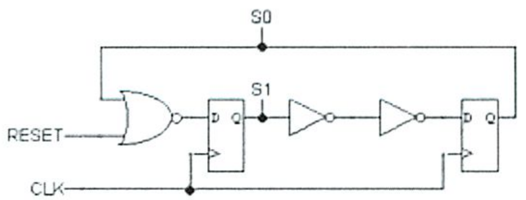
think about them

(E) A complex combinational circuit is constructed entirely from 2-input NAND gates having a propagation delay of 1 ns. If this circuit is pipelined for maximal throughput by adding registers whose setup time and propagation delay are each 1 ns, what is ~~the~~ throughput of the resulting pipeline? Enter a number, a formula, or "CAN'T TELL".

does not matter,

Throughput (ns⁻¹): $\frac{1}{3\text{ns}}$

$\frac{1}{t_{\text{clk}}}$ well $\frac{1}{\min t_{\text{clk}}}$



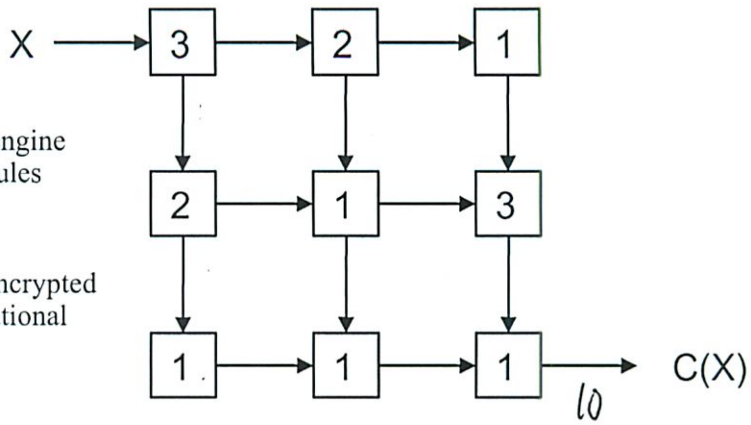
inverter: t_{CD}=1ns, t_{PD}=2ns
nor2: t_{CD}=1.5ns, t_{PD}=2ns
D register: t_{CD}=0ns, t_{PD}=2ns, t_H=1ns, t_S=3ns

(F) You are given the sequential circuit and component specs shown to the left. What is the shortest clock period that can be used?

Minimum clock period (ns): 9

t_{clk} ~~A~~ $\geq \sum t_{pd} + t_S$
for each section

Problem 2 (10 points): **Pipelining**



10

The RIAA has come up with a new media encryption engine called the Piper, consisting of nine combinational modules connected as shown to the right.

The device takes a music sample X and computes an encrypted version C(X). In the diagram to the right each combinational component is marked with its propagation delay in microseconds; contamination delays are zero for each component. Unfortunately, it is too slow.

(A) (2 points) What are the latency and throughput of the Piper device?

tpp longest - critical path

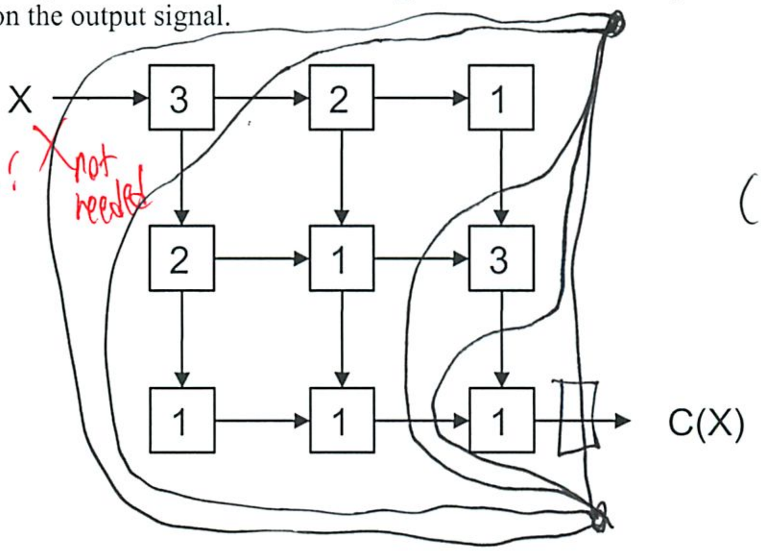
Latency (microseconds): 10 ✓

Throughput (1/microseconds): 1/10 ✓

(B) (4 points) Show the RIAA how to pipeline the Piper by adding registers to maximize throughput, but achieve the smallest latency that meets the maximum throughput constraint. Using the diagram below indicate the locations for ideal (zero-delay) registers to create a pipelined implementation that meets these goals. Remember that your answer should have a register on the output signal.

ii so never put on input? not needed

Extra copies of this diagram can be found on the back of the previous page.



Can't split 3 any more

(C) (2 points) What is the latency and throughput of your pipelined implementation?

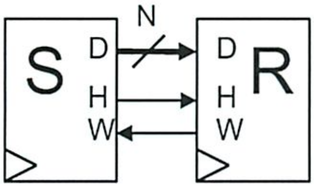
Latency (microseconds): 9 ~~18~~.3 ✓; **Throughput (1/microseconds):** 1/3 ✓

(D) (2 points) Suppose you found pipelined replacements for the components marked 3 and 2 that had 3 and 2 stages, respectively, and could be clocked at a 1 microsecond period. Using these replacements and pipelining for maximum throughput, what is the best achievable performance?

Latency (microseconds): ~~13.1~~ 10; **Throughput (1/microseconds):** 1/1

I guess counted wrong - want longest path - oh could remove other lines auto end line

Problem 3 (14 points): **Self-timed protocols**



Self Timed Systems, Inc makes clocked modules that pass data to each other using a simple, stylized protocol. Data is passed between connected modules - a "sender" and a "receiver" - using an interface consisting of three parts as diagrammed to the left. The diagram shows a typical connection, over which data is occasionally passed from sending module **S** to receiving module **R**.

The handshake protocol allows data to be transferred between sending and receiving modules on appropriate clock edges, selected by the availability of data at the sender and the capacity of the receiver to receive the data. The wires connecting sender and receiver include an N-bit **D** (data) word containing the data to be communicated, as well as control lines **H** (asserted by the sender to signal "**Have data**") and **W** (asserted by the receiver to signal "**Want data**"). The clock input to each module is connected to a single, global, periodic clock. Data is transferred from **S** to **R** on those active clock edges for which the **H** and **W** signals are both **1** (asserted).

oh a push pull thing

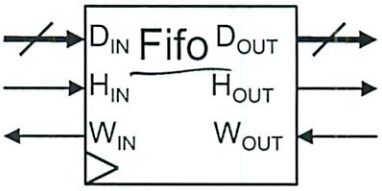
When the sending module has data for the receiver, it drives the data on the **D** lines and asserts **H** (sets it to 1) to indicate data availability. When the receiving module is ready to accept new input data, it asserts **W** and prepares to load data from **D** at the next active clock edge. At each active clock edge, an N-bit word of data is transferred from **S** to **R** if and only if both **H** and **W** signals are asserted. If **H** is asserted but not **W**, the sender keeps driving **D** and **H** so that **R** may accept the data in some subsequent cycle; if **W** is asserted but not **H**, the receiver ignores data it loaded and continues driving **W** until the sender is ready to respond positively to its request for data. Each of these events happens on an active clock edge (i.e., a positive transition), and each module samples incoming signals only on active clock edges. The protocol is self-timed, meaning that each module may take arbitrarily many clock cycles for each of its responses.

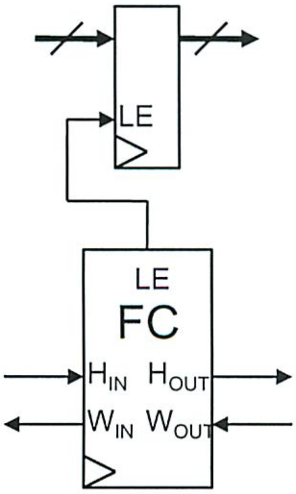
↓ clk

(A) (1 point) Suppose the period of the clock is **P** seconds. What is the maximum throughput, in N-bit words/seconds, of such a connection?

Max connection throughput for clock period P: $\frac{1}{P}$ **words/sec**

The flagship STS product is a FIFO ("First-In-First-Out") module useful for buffering streams of N-bit data, diagrammed to the right. The FIFO module incorporates an N-bit register capable of holding a single N-bit binary data word (for example, an N-bit binary number). Input and output to this register are performed using separate IN and OUT connections to the FIFO, each using the above connection protocol. The idea is that one or more FIFO modules may be spliced into a path between a source of data and its consumer, providing a buffer between these modules.

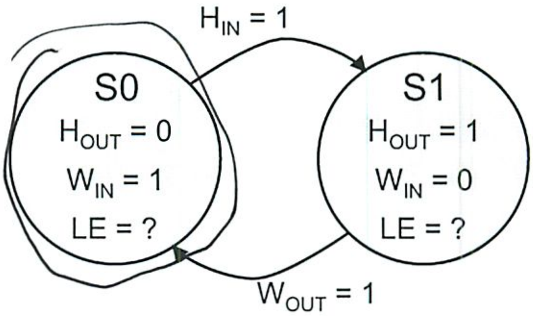




STS engineers choose to implement the FIFO module as a simple "Fifo Controller" (FC) FSM connected to an off-the-shelf N-bit register. The FC component implements the "Have" and "Want" control signals for both the input and output ports, and produces a Load Enable (LE) signal which, when 1, causes the register to load new data at the next active clock edge. The FC is to be implemented as a Moore machine, meaning that each of its output is a function only of its current state.

Very wordy questions...

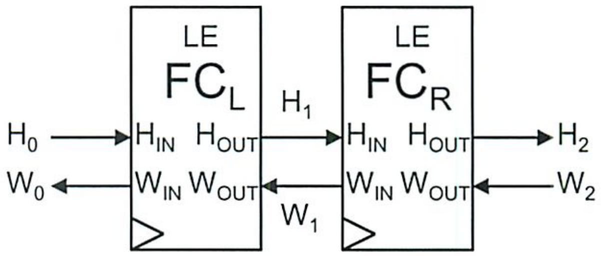
Problem 3 (continued)



The FC was designed by a summer intern from MIT, hired because he got an impressive score of 26 on Quiz 2 of 6.004. His design, shown to the right, is a simple 2-state Moore machine whose states are marked with FC output values and whose transitions are marked with appropriate input conditions. The initial state is **S0**. Same-state transitions are not shown; unless the FSM is in state **S0** and **H_{IN}=1** or is in **S1** and **W_{OUT}=1**, it remains in its current state. Unfortunately, the diagram is incomplete; the values for the Load Enable signal to the register have been omitted.

(B) (1 point) What value of **LE** should be produced for each state of FC?

LE value for state S0: _______; for state S1: ________



Cass Cade, VP of Performance for STS, decides to explore the throughput properties of two FIFO modules cascaded to make a 2-word FIFO buffer. Rather than experiment with complete FIFO modules, however, Cass finds it convenient to use just the FC portion since no actual data need be transferred in her experiments.

Cass models the case of two casaded FIFOs by the diagram to the right, showing an FSM consisting of two FC modules (Left and Right). She labels the two inputs, two outputs, and two internal connections (between the FC modules) as shown. Cass denotes the state of this FSM by S_LS_R, where S_L is the state (0 or 1 for S0 or S1 respectively) of the left FC, and S_R is the state of the right FC.

(C) (1 point) Viewing the two interconnected FC's as a single FSM, how many states does it exhibit?

Number of states in two cascaded FC modules: ________

Next, Cass models an always-available data source and an always-empty data sink **by tying H0 and W2 to 1** in the above picture. She runs the FSM for several clock cycles, intending to fill in the diagram below.

Cycle	**0**	**1**	**2**	**3**
State: S_LS_R	"00"			
H_0	1	1	1	1
W_0	1			
H_1	0			
W_1				
H_2	0			
W_2	1	1	1	1

(D) (5 points) Fill the missing entries in the table to the left, showing the behavior of the dual-FC FSM with inputs tied to 1 for the first few cycles of its operation.

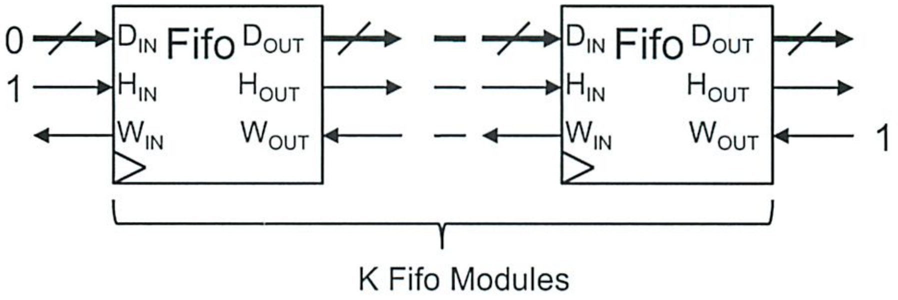
(Complete table to left)

From her two-FC experiment, Cass predicts the maximum throughput of a two-word FIFO buffer constructed from two cascaded FIFO modules.

(E) (1 point) For clock period **P**, what maximum throughput would you expect from two cascaded FIFO modules?

Maximum throughput for clock period P: ______________words/sec

Cass generalizes the above experiment to involve a chain of **K** FIFO modules, connected as shown below, and measures the throughput of the chain.



(F) (2 points) For clock period **P**, what throughput would you expect the **K** cascaded FIFO modules?

Throughput for clock period P: ______________words/sec

It occurs to Cass that she might improve throughput of the FIFO by using combinational logic to cause a "Want" signal on $\mathbf{W_{OUT}}$ to force $\mathbf{W_{IN}}$ to be asserted during the same cycle. Cass reasons that, even if the buffer is full during the current cycle, the $\mathbf{W_{OUT}}$ signal from its output side ensures that the register can be used to hold new data during the next clock cycle. She asks her engineers about this proposal, and gets a variety of counter-arguments:

C1: "That would cause a combinational cycle in a FIFO cascade!"
C2: "Your FC would no longer be a Moore machine."
C3: "That would introduce unsolvable arbitration problems."
C4: "The proposal wouldn't increase throughput at all"
C5: "The proposal would require a clock period proportional to **K**, for **K** cascaded FIFOs"

(G) (3 points) Which of the above complaints are valid? Circle each valid complaint, or NONE:

Valid complaints (circle all that apply): C1 ... C2 ... C3 ... C4 ... C5 ... NONE

END OF QUIZ!

1	/ 5
2	/ 19
3	/ 6
	/ 30

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Fall 2010

Quiz #2: October 15, 2010

Name		*Athena login name*	*Score*
TA: Caitlin, 26-322 □ WF 10 □ WF 11	TA: Quentin, 34-303 □ WF 11 □ WF 12	TA: Sabrina, 34-304 □ WF 12 □ WF 1	TA: Steve, 34-303 □ WF 2

Problem 1 (5 points): **Quickies and Trickies (1 point each)**

(A) A combinational circuit C, built entirely from 2-input NAND gates having a propagation delay of 2ns, has a propagation delay of 20ns. You pipeline C for maximum thruput using the minimum number of registers necessary; the registers have 1ns setup time and 1ns propagation delay. What would you expect for the latency of the resulting pipeline? Answer "None" if you can't tell from the information given.

Latency of pipelined version, or "None": ____________________

(B) If we account for fanin limitations but ignore wire delays, what is the asymptotic latency of the fastest combinational N-input AND circuit we can build?

Asymptotic latency of N-input AND, or "None": Θ(____________________)

(C) Is $\Theta(\log_2 N)$ the same as $\Theta(\log_{10} N)$?

Circle best choice: YES ... NO ... only for some N

(D) Putting latches on the shared A and B data inputs to the arithmetic, boolean, shifter, and comparator units of an ALU may improve

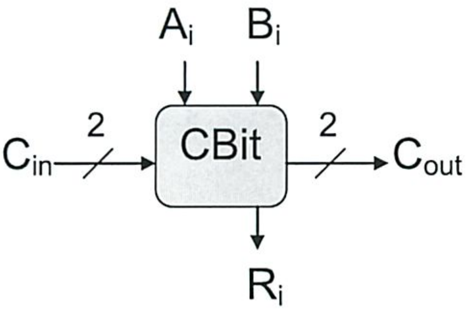
1) Latency
2) Reliability
3) Throughput
4) Power dissipation
5) None of the above

Select best choice: _____

(E) True or False: It is impossible, in theory, to build a 100% reliable bounded-time, bounded-error analog voltage comparator.

Circle best choice: TRUE ... FALSE

Problem 2 (19 points): **Comparative Anatomy**

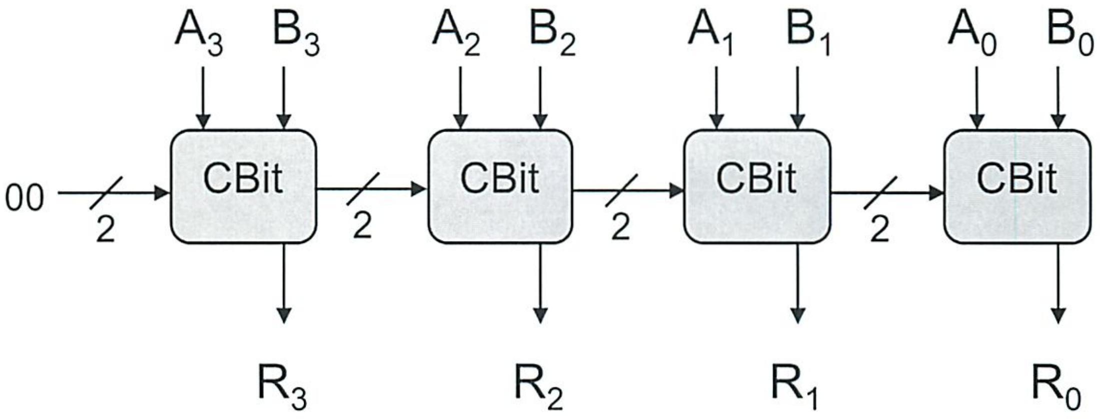


MaxOut is a Cambridge startup whose products are binary comparators which determine the largest of several unsigned binary integers. A building block common to all MaxOut products is the combinational CBit module depicted to the left.

Each CBit module takes corresponding bits of two unsigned binary numbers, A and B, along with two C_{in} bits from higher-order CBit modules. Its output bit, R, is the appropriate bit of the larger among A and B, as determined from these inputs; it passes two C_{out} bits to lower-order CBit modules.

The propagation delay of each CBit module is 4ns. The two C_{out} bits indicate, respectively, if A>B or B>A in the bits considered thus far.

The first MaxOut product is MAXC, a combinational device which determines the maximum of its two 4-bit unsigned binary inputs. It is constructed using 4 CBit modules:



In the above diagram, unused inputs are tied to 0. The output $R_{3:0}$ is the larger of $A_{3:0}$ or $B_{3:0}$.

(A) (2 points) What propagation delay specification is appropriate for the combinational MAXC module? What is its throughput?

MAXC propagation delay spec: ____________________ **ns**

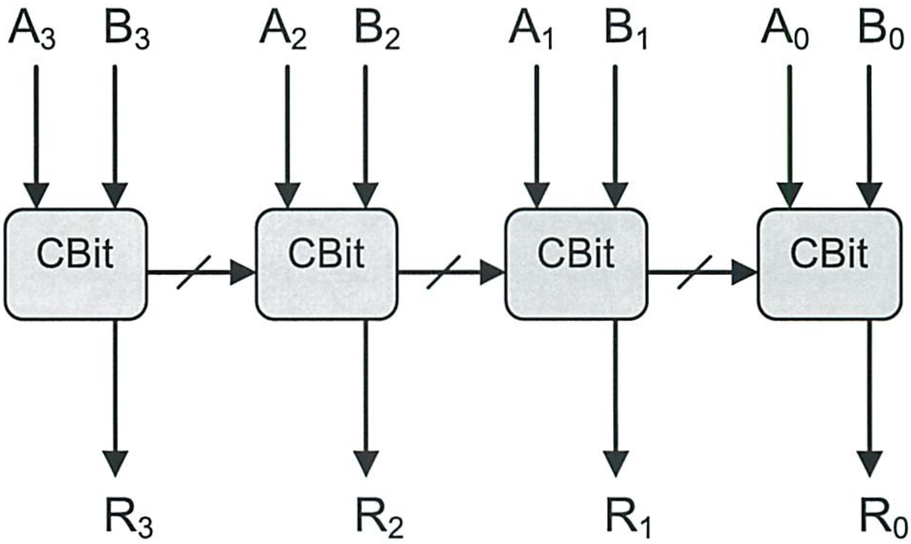
MAXC throughput: 1/____________________ **ns**

(B) (1 point) If $A_{3:0}$ and $B_{3:0}$ are identical numbers, what two bits would you expect to see coming out of the (unused) C_{out} outputs from the low-order CBit module?

Low-order C_{out} bits for A=B: ______ **and** _______

MaxOut's second product, MAXP, is identical to MAXC except that it includes the minimum number of registers necessary inserted to pipeline the circuit for maximum throughput.

(C) (2 points) On the diagram below, show contours indicating where ideal (zero delay, zero setup/hold time) registers are inserted to pipeline MAXC for maximum throughput. Be sure to include at least one register on each output.



(scratch copies are on back of previous page)

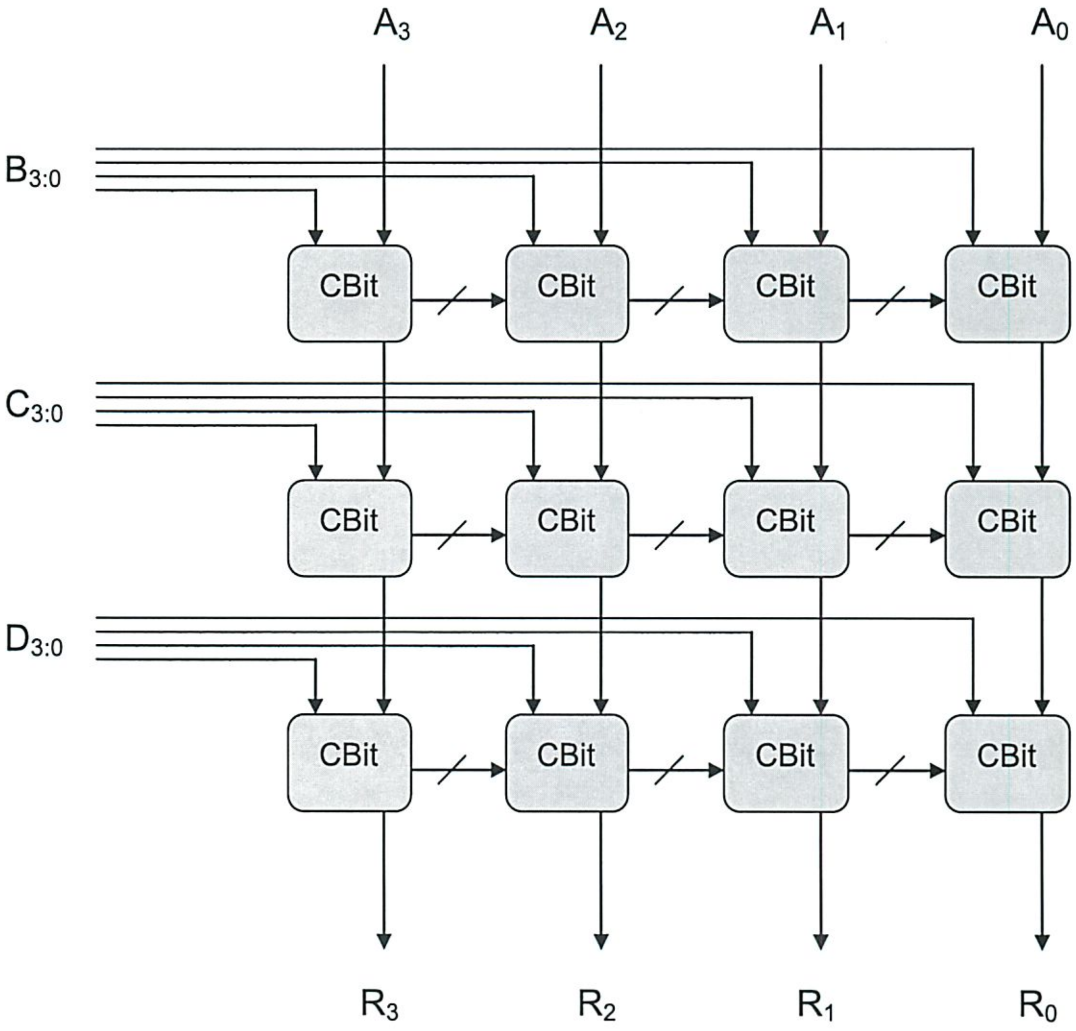
(mark diagram above)

(D) (2 points) What are the latency and throughput of your pipelined MAXC?

Pipelined MAXC latency: ____________________ns

Pipelined MAXC throughput: 1/____________________ns

Expanding their product line, MaxOut's next product – the MAX4X4 -- is a combinational multiplier capable of determining the maximum of four 4-bit binary inputs:



(E) (2 points) What are the best latency and throughput that can be achieved using the combinational MAX4X4?

Latency: ______________________ **ns**

Throughput: 1/______________________ **ns**

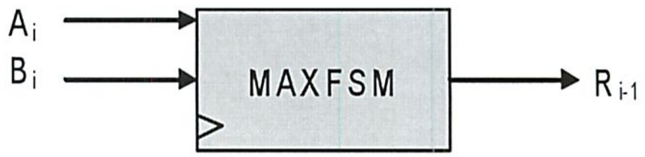
(F) (6 points) Mark, on the above diagram, contours indicating placement of ideal registers for pipelining the MAX4X4 for maximum throughput. Give the best latency and throughput that can be achieved by pipelining the MAX4X4. (Scratch copies on back of previous page).

Latency: ______________________ **ns**

Throughput: 1/______________________ **ns**

(mark diagram above)

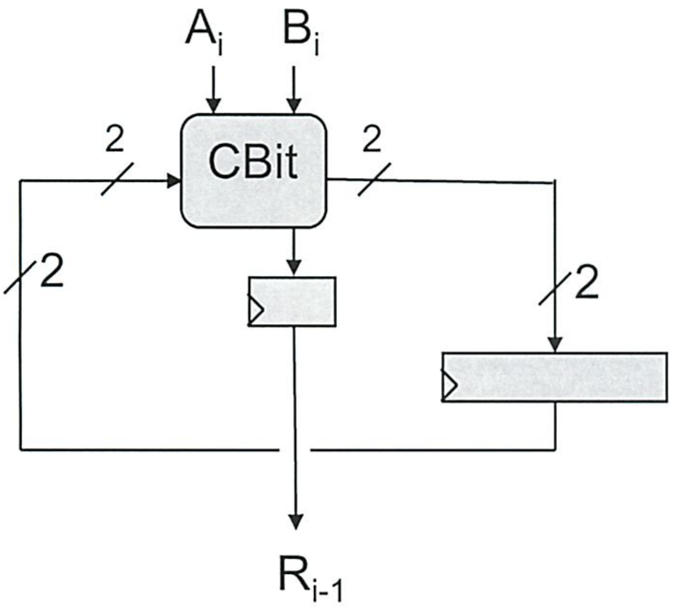
To round out their product line, MaxOut's latest product uses an alternative approach. The MAXFSM is to be a clocked finite state machine that takes two N-bit binary numbers, $A_{N:0}$ and $B_{N:0}$ in bit-serial form, most significant bit first, and outputs the larger of these numbers as $R_{N-1:0}$ also in bit-serial form. The MAXFSM is a Moore machine; recall that this means its output is strictly a function of its current state (like those FSMs shown in lecture).



Before each active clock edge, the ith bit of the A and B inputs are applied; during the *next* clock cycle, the ith bit of the larger of the two input numbers appears at the R output. Note that the serial output bits are delayed with respect to the input bits by one clock cycle, in order to allow each ith output bit to be influenced by the ith input bits.

(G) (1 point) What is the minimum number of states necessary to implement a Moore machine obeying the above specifications?

Number of states required: ____________________



Ignoring your answer, MaxOut decides to build the MAXFSM using a CBit module and several flipflops, as shown in the diagram to the left. The registers have the following specifications:

t_{pd}	4ns
t_{cd}	1ns
t_s	5ns
t_h	1ns

Recall that the CBit has a 4ns propagation delay; assume that its contamination delay is zero.

(H) (1 point) What is the shortest clock period for which this circuit will operate reliably?

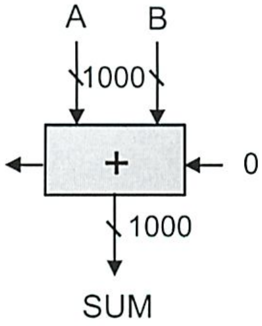
Clock period ≥ _____ns

(I) (2 points) What setup and hold time requirements should be specified for this FSM?

FSM Setup time: _____ns

FSM Hold time: _____ns

Problem 3 (6 points): **Big Adders**



Carrie Guess, a star 6.004 student, has taken a coveted summer internship at the behemoth Froogle, Inc designing hardware for their web servers. Her assignment is to develop an adder for two 1000-bit binary integers, the critical step in a new web service to be offered.

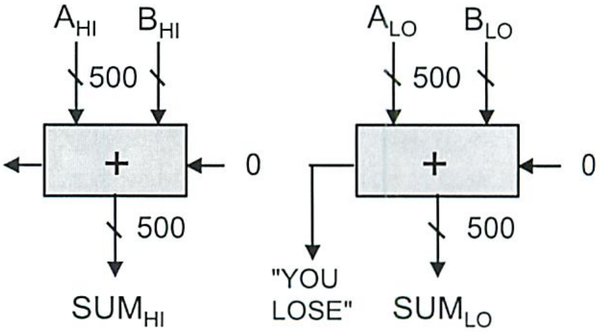
Remembering the ripple-carry adders she built in labs 2 and 3, her first approach is to build a 1000-bit ripple-carry adder using Froogle's standard Full Adder component, having a 1ns propagation delay. Asshe did in lab 2, she feeds 0 into the low-order carry input, and ignores the high-order carry output.

(A)(1 point) What is the propagation delay of the 1000-bit ripple-carry adder?

Propagation delay:_____________ns

Carrie's boss tells her that her adder is too slow, by about a factor of two. Searching for a scheme to speed up the adder, Carrie decides to break it into two 500-bit ripple carry adders, and simply "guess" that the carry input into the high-order sum will be a zero, as follows:

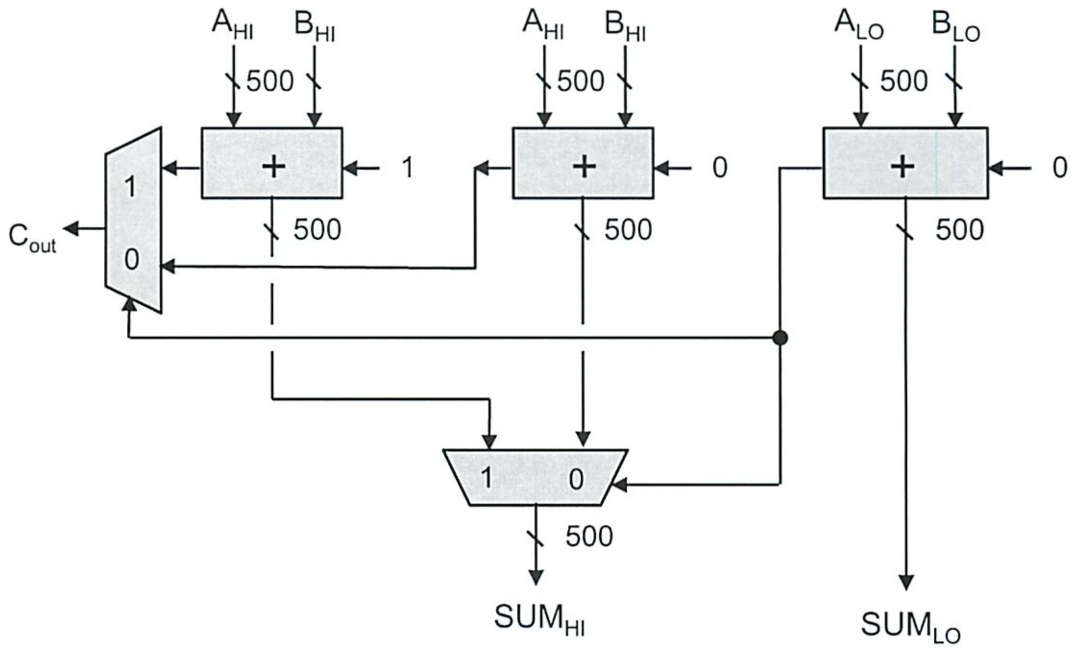
In this design, each of the A and B inputs, as well as the SUM outputs, is broken down into high and low halves, each 500 bits wide. Carrie's new adder now produces a 1000-bit sum whose high-order bits will be correct only about half the time; as an added feature, it produces an error signal to indicate when the sum is invalid. She stays up all night rewriting the user interface to Froogle's new service, replacing the "Compute" button with one that reads "I feel lucky", and introducing a response "You're not THAT lucky" when the adder fails.



(B) (1 point) What is the propagation delay of the new adder?

Propagation delay:_____________ns

Plagued by complaints from irate users, Carrie's boss reports that he likes the speedup of her new adder, but that reporting failure is not an acceptable option. After some thinking, Carrie comes up with the following revision to her adder:



This design uses three 500-bit ripple carry adders, computing the high-order sum under both possible assumptions about its carry input. A final 2-way MUX selects the proper sum and carry outputs, based on the actual carry from the low-order sum. The propagation delay of each MUX is 1ns. Carrie calls this design a "Carrie-select" adder.

(C) (2 points) Does the "Carrie-select" adder always compute the proper sum? What is its propagation delay?

Always works? Mark one: YES: ____; or NO: ___

Propagation delay:_____________ns

Carrie starts thinking about replacing each of the 3 500-bit ripple-carry adders in her "Carrie-select" design themselves with Carrie-select adders (each comprising three 250-bit ripple-carry adders).

(D) (2 points) What is the propagation delay of this 2-level Carrie-select adder? How many Full Adders are required in its construction?

Propagation delay:_____________ns

Total number of Full Adders required: ________________

(E) (1 point) Suppose the Carrie-select approach were used at *every* level of an N-bit adder -- i.e., each k-bit adder used as a component were replaced by a k-bit Carrie-select adder wherever that improves performance, and the components of that Carrie-select adder were recursively so upgraded. What is the asymptotic latency of the resulting N-bit adder?

Asymptotic latency of N-bit recursively-constructed carry-select adder: Θ(___________________)

END OF QUIZ 2
(phew!)

6.004
Quiz 2
Debreif

10/14

Considering minimal study think I did well

But didn't get latch one

Since thought about it wrong at first
Then did not have enough time to fix

And messed up how much setup + hold time need
- I should review
- Also did not fully think since ran out of time