Reviewing CALL & Intro To Digital Circuits

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Example C Program: `Hello.c`


```
#include <stdio.h>
int main()
{
  printf("Hello, %s\n", "world");
  return 0;
}
```

Compiled `Hello.c`: `Hello.s`


```
.text                          # Directive: enter text section
  .align 2                     # Directive: align code to 2^2 bytes
  .globl main                  # Directive: declare global symbol main
main:                          # label for start of main
  addi sp,sp,-16               # allocate stack frame
  sw   ra,12(sp)               # save return address
  lui  a0,%hi(string1)         # compute address of
  addi a0,a0,%lo(string1)      #   string1
  lui  a1,%hi(string2)         # compute address of
  addi a1,a1,%lo(string2)      #   string2
  call printf                  # call function printf
  lw   ra,12(sp)               # restore return address
  addi sp,sp,16                # deallocate stack frame
  li   a0,0                    # load return value 0
  ret                          # return
  .section .rodata             # Directive: enter read-only data section
  .balign 4                    # Directive: align data section to 4 bytes
string1:                       # label for first string
  .string "Hello, %s!\n"       # Directive: null-terminated string
string2:                       # label for second string
  .string "world"              # Directive: null-terminated string
```

Assembled `Hello.s`: Linkable `Hello.o`


```
00000000 <main>:
0:  ff010113 addi sp,sp,-16
4:  00112623 sw ra,12(sp)
8:  00000537 lui a0,0x0      # addr placeholder
c:  00050513 addi a0,a0,0    # addr placeholder
10: 000005b7 lui a1,0x0      # addr placeholder
14: 00058593 addi a1,a1,0    # addr placeholder
18: 00000097 auipc ra,0x0    # addr placeholder
1c: 000080e7 jalr ra         # addr placeholder
20: 00c12083 lw ra,12(sp)
24: 01010113 addi sp,sp,16
28: 00000513 addi a0,a0,0
2c: 00008067 jalr ra
```

Linked `Hello.o`: `a.out`


```
000101b0 <main>:
  101b0: ff010113 addi sp,sp,-16
  101b4: 00112623 sw   ra,12(sp)
  101b8: 00021537 lui  a0,0x21
  101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
  101c0: 000215b7 lui  a1,0x21
  101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
  101c8: 288000ef jal  ra,10450    # <printf>
  101cc: 00c12083 lw   ra,12(sp)
  101d0: 01010113 addi sp,sp,16
  101d4: 00000513 addi a0,0,0
  101d8: 00008067 jalr  ra
```

LUI/ADDI Address Calculation in RISC-V

Target address of <string1> is **`0x00020 A10`**

Instruction sequence `LUI 0x00020`, `ADDI 0xA10` does not quite work because immediates in RISC-V are sign extended (and `0xA10` has a 1 in the high order bit)!

`0x00020 000 + 0xFFFFF A10 = 0x0001F A10` (Off by 0x00001 000)

So we get the right address if we calculate it as follows:

`(0x00020 000 + 0x00001 000) + 0xFFFFF A10 = 0x00020 A10`

What is `0xFFFFF A10`?

Twos complement of `0xFFFFF A10` = `0x00000 5EF + 1` = `0x00000 5F0` = 1520_{ten}

So `0xFFFFF A10` = -1520_{ten}

Instruction sequence `LUI 0x00021`, `ADDI -1520` calculates **`0x00020 A10`**

Static vs. Dynamically Linked Libraries

- What we've described is the traditional way: statically-linked approach
  - Library is now part of the executable, so if the library updates, we don't get the fix (have to recompile if we have source)
  - Includes the entire library even if not all of it will be used
  - Executable is self-contained
- Alternative is dynamically linked libraries (DLL), common on Windows & UNIX platforms

Dynamically Linked Libraries

en.wikipedia.org/wiki/Dynamic_linking

- Space/time issues
  - + Storing a program requires less disk space
  - + Sending a program requires less time
  - + Executing two programs requires less memory (if they share a library)
  - – At runtime, there's time overhead to do link
- Upgrades
  - + Replacing one file (libXYZ.so) upgrades every program that uses library "XYZ"
  - – Having the executable isn't enough anymore
    - Thus the rise of "Containers"

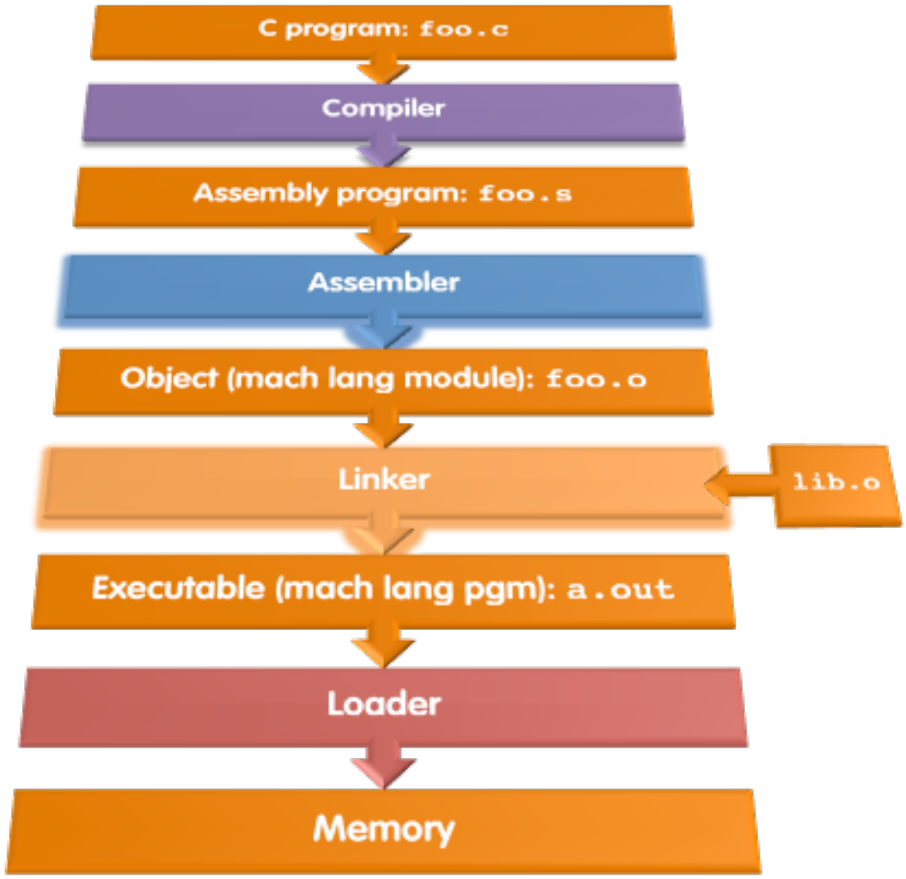
Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system. However, it provides many benefits that often outweigh these

Dynamically Linked Libraries

- Prevailing approach to dynamic linking uses machine code as the "lowest common denominator"
  - Linker does not use information about how the program or library was compiled (i.e., what compiler or language)
  - Can be described as "linking at the machine code level"
  - This isn't the only way to do it ...
- Also these days will ***randomize layout*** (Address Space Layout Randomization)
  - Acts as a defense to make exploiting C memory errors substantially harder, as modern exploitation requires jumping to pieces of existing code ("Return oriented programming") to counter another defense (marking heap & stack unexecutable, so attacker can't write code into just anywhere in memory).

In Conclusion…

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A `.s` file becomes a `.o` file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several `.o` files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.



Administrivia...

- Project 2 parrtayyy tonight at 8pm!
  - Wozniak lounge
- Remember the campground/frat-house rule...
  - The ***callee saved registers*** (e.g. sp, s0-s7) ***must be returned unchanged***
    - So either don't touch or save on the stack and restore prior to returning/tail call
  - The ***caller saved registers*** (e.g. t0-t9, a0-a7...) ***may be trashed at will by other functions***
    - So when you call another function, you know that everything in those can be overwritten
- Testing hint for the autograder...
  - We will call your functions and check that they respect the campground
  - When your functions call our functions, we have versions that put garbage in the caller-saved registers
- So you can build your own wrappers that do the same thing!

Hardware Design

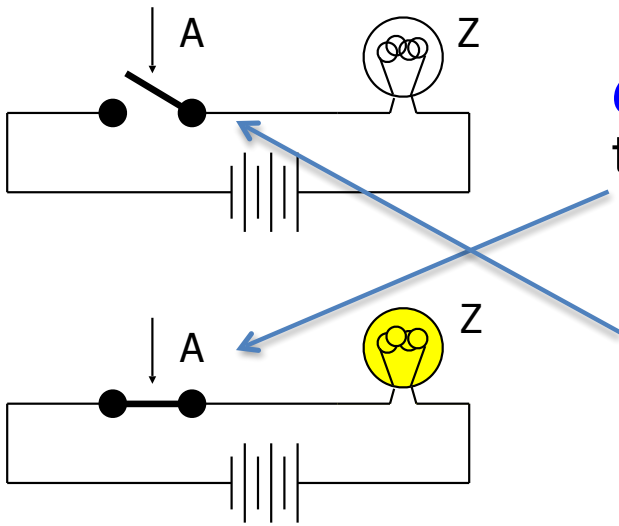
- Next weeks: how a modern processor is built, starting with basic elements as building blocks
- Why study hardware design?
  - Understand capabilities and limitations of HW in general and processors in particular
  - What processors can do fast and what they can't do fast (avoid slow things if you want your code to run fast!)
  - Background for more in-depth HW courses (EECS 151, CS 152)
  - Hard to know what you'll need for next 30 years
  - There is only so much you can do with standard processors: you may need to design own custom HW for extra performance
    - Even some commercial processors today have customizable hardware!

Synchronous Digital Systems: Almost Every Processor etc...

- ***Synchronous***:
  - All operations coordinated by a central clock
    - "Heartbeat" of the system!
    - ***Anynchronous*** systems much much much much harder to design & debug
- ***Digital***:
  - Represent all values by discrete values
  - Two binary digits: **1** and **0**, or **True** and **False**
    - Electrical signals are treated as 1's and 0's
      - 1 and 0 are complements of each other
  - High/low voltage for true/false, 1/0
  - These days, even a lot of ***analog*** circuitry is best done as:
    - Put it through an analog to digital converter, do it all digitally in a synchronous circuit, and output to a digital-to-analog converter

Switches: Basic Element of Physical Implementations

- Implementing a simple circuit (arrow shows action if wire changes to "1" or is *asserted*):



On-switch (if A is "1" or asserted) turns-on light bulb (Z)

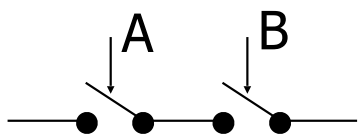
Off-switch (if A is "0" or unasserted) turns-off light bulb (Z)

$$Z \equiv A$$

Switches (cont'd)

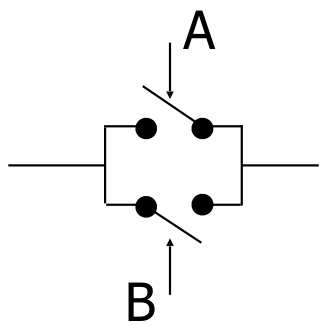
- Compose switches into more complex ones (Boolean functions):

AND



Z ≡ A <u>and</u> B

OR

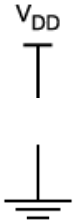


Z ≡ A <u>or</u> B

Historical Note

- Early computer designers built ad hoc circuits from switches
  - Began to notice common patterns in their work: ANDs, ORs, …
- Master's thesis (by Claude Shannon, 1940) made link between work and 19th Century Mathematician George Boole
  - Called it "Boolean" in his honor
- Could apply math to give theory to hardware design, minimization, …

Transistors

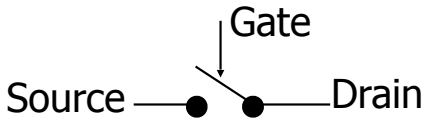


- High voltage (V_{dd}) represents 1, or true
  - The Raspberry Pi: V_{dd} ~ 1.2 Volt
- Low voltage (0 Volt or Ground) represents 0, or false
- Pick a midpoint voltage to decide if a 0 or a 1
  - Voltage greater than midpoint = 1
  - Voltage less than midpoint = 0
  - This removes noise as signals propagate –
    a big advantage of digital systems over analog systems
- If one switch can control another switch, we can build a computer!
- Our switches: CMOS transistors

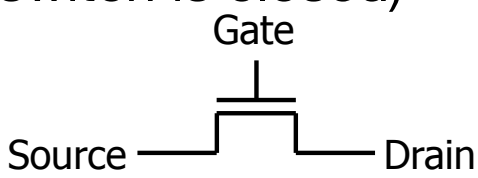
CMOS Transistor Networks

- Modern digital systems designed in CMOS
  - MOS: Metal-Oxide on Semiconductor
    - Describes how the transistors are constructed
    - These are "Field Effect Transistors" (MOSFETs)
  - C for complementary: use ***pairs*** of normally-on and normally-off switches
- CMOS transistors act as voltage-controlled switches
  - Similar, though easier to work with, than electro-mechanical relay switches from earlier era
  - Use energy primarily when switching
    - But not completely: they do "leak" a bit

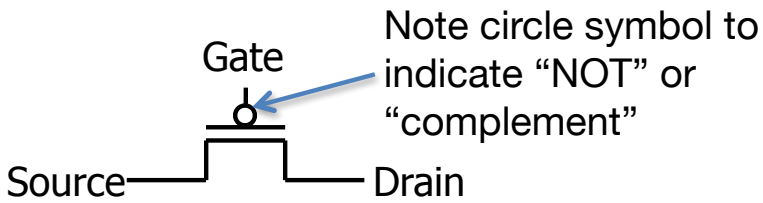
CMOS Transistors



- Three terminals: source, gate, and drain
  - Switch action:
    if voltage on gate terminal is (some amount) higher/lower than source terminal then conducting path established between drain and source terminals (switch is closed)

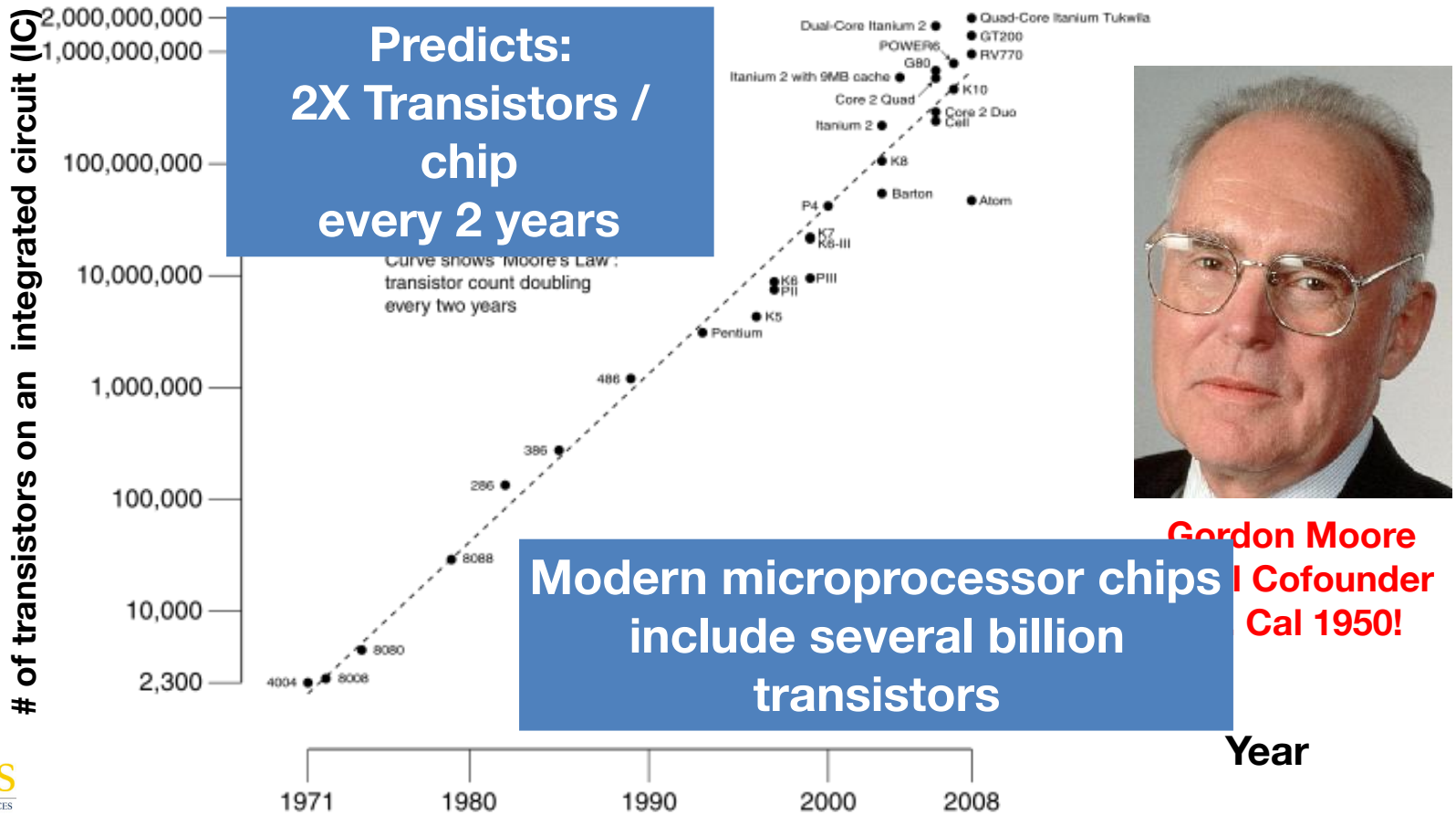


n-channel transitor
off when voltage at Gate is low
on when:
voltage(Gate) > voltage (Threshold)



p-channel transistor
on when voltage at Gate is low
off when:
voltage(Gate) > voltage (Threshold)

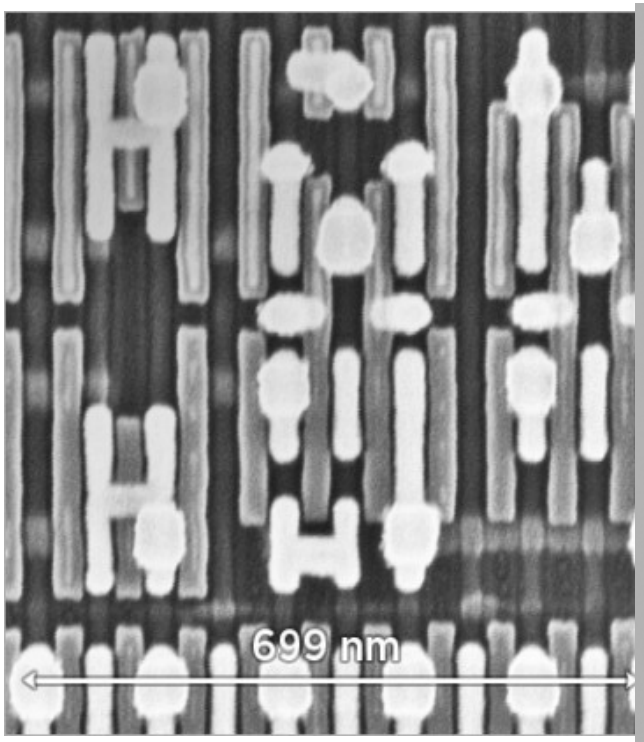
#2: Moore's Law



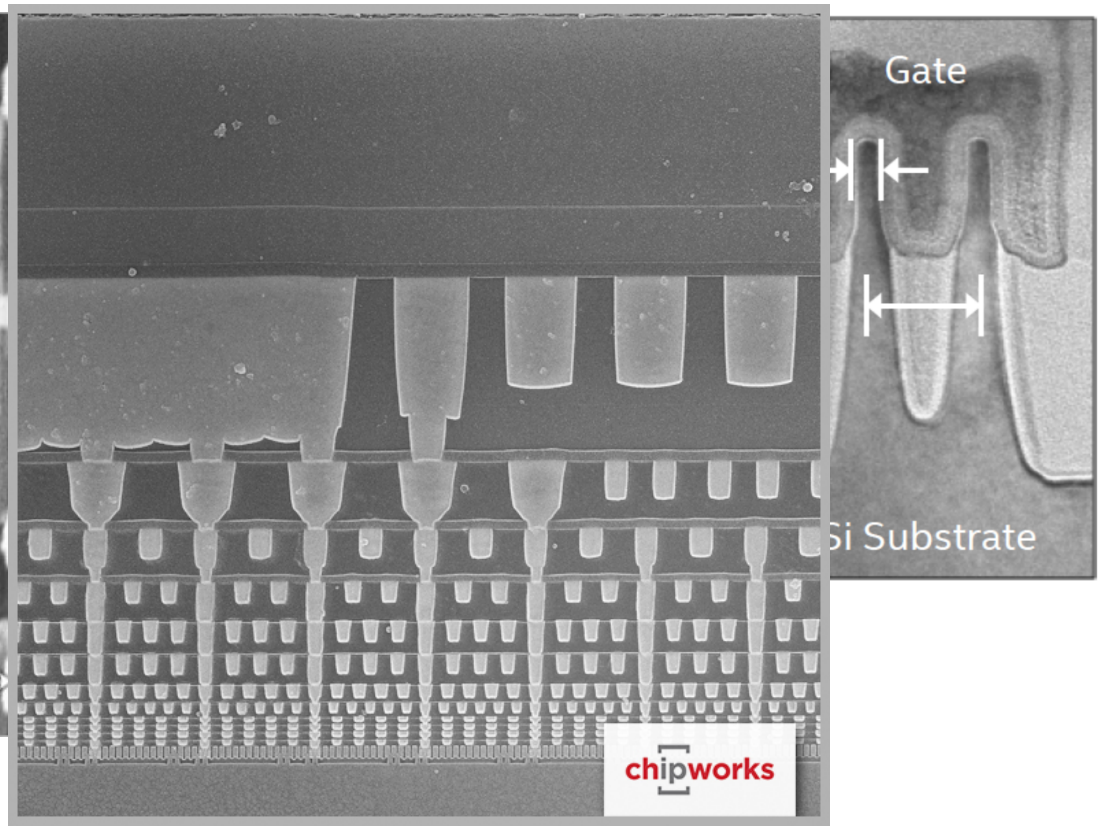
Modern microprocessor chips include several billion transistors

Gordon Moore
Cofounder
Cal 1950!

Intel 14nm Technology

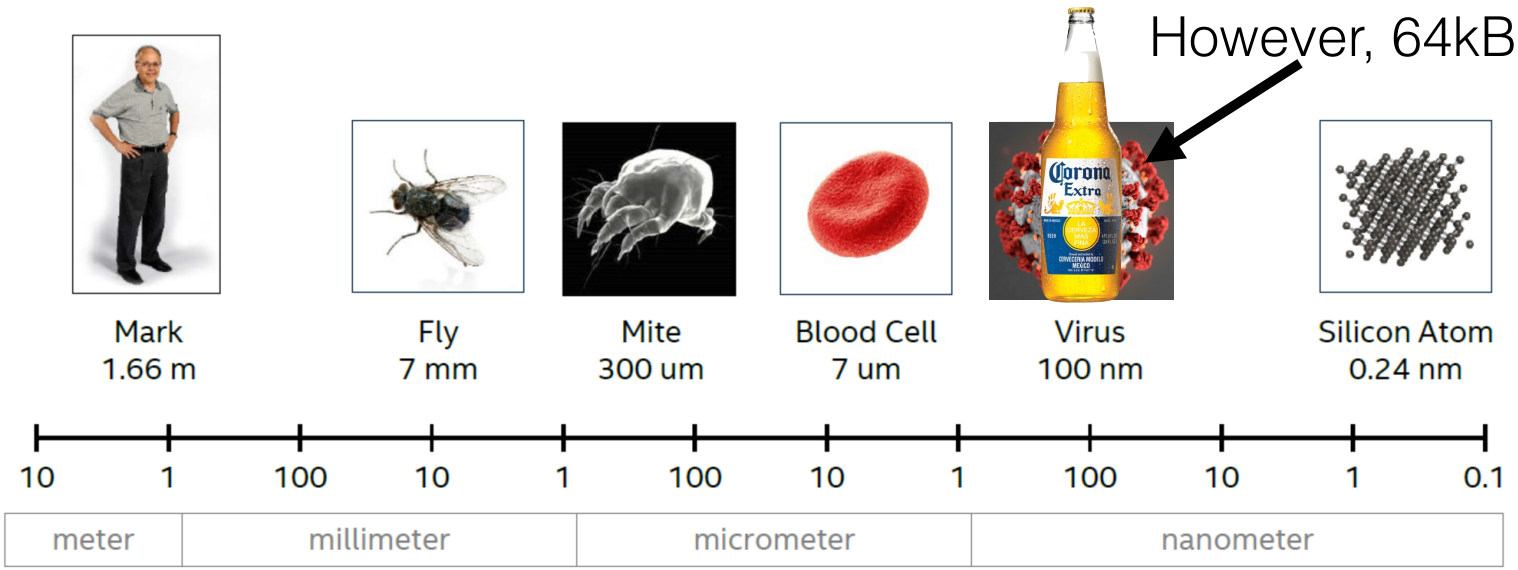


Plan view of transistors



Side view of wiring layers

Sense of Scale



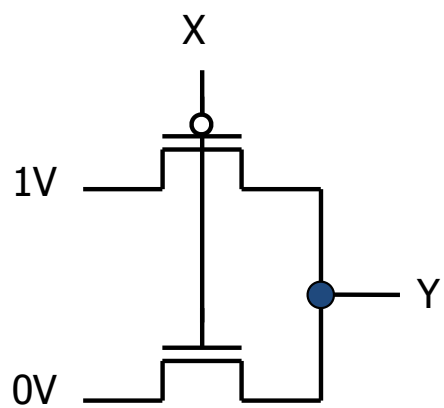
Source: Mark Bohr, IDF14

CMOS Circuit Rules

- ***Don't pass*** weak values => Use Complementary Pairs
  - N-type transistors pass weak 1's (V_{dd} - V_{th})
  - N-type transistors pass strong 0's (ground)
  - Use N-type transistors only to pass 0's (N for negative)
  - Converse for P-type transistors: Pass weak 0s, strong 1s
    - Pass weak 0's (V_{th}), strong 1's (V_{dd})
    - Use P-type transistors only to pass 1's (P for positive)
  - Use pairs of N-type and P-type to get strong values
- ***Never*** leave a wire undriven (in this class)
  - Make sure there's always a path to Vdd or GND
- ***Never*** create a path from Vdd to GND (ground)
  - This would short-circuit the power supply!

CMOS Networks

p-channel transistor
on when voltage at Gate is low
off when:
voltage(Gate) > voltage (Threshold)



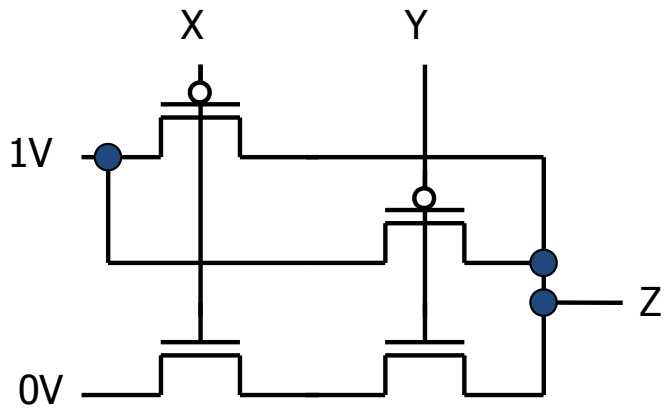
n-channel transitor
off when voltage at Gate is low
on when:
voltage(Gate) > voltage (Threshold)

what is the relationship between x and y?

x	y
0 Volt (GND)	1 Volt (Vdd)
1 Volt (Vdd)	0 Volt (GND)

Called an *inverter* or *not gate*

Two-Input Networks

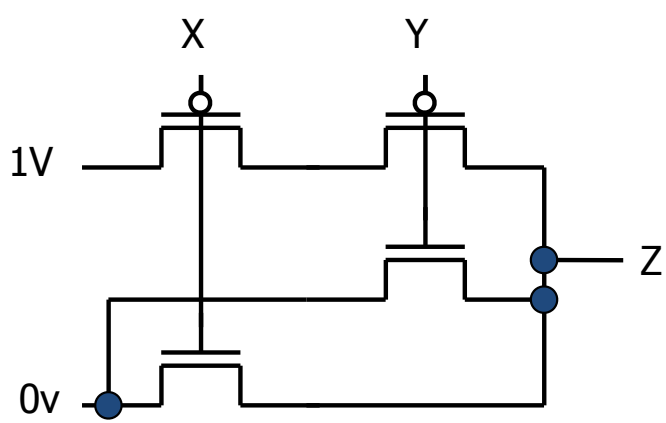


what is the relationship between x, y and z?

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

Called a *NAND gate (NOT AND)*

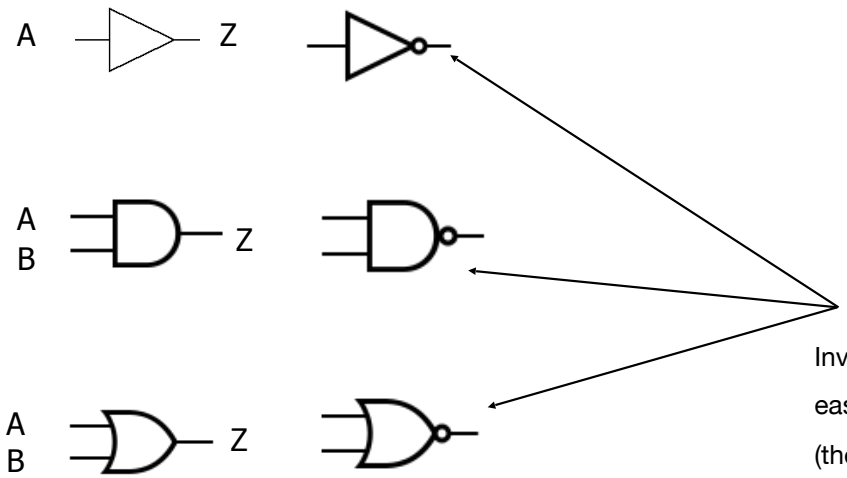
Clickers/Peer Instruction



x	y	z				
		A	B	C	D	
0 Volt	0 Volt	0	0	1	1	Volts
0 Volt	1 Volt	0	1	0	1	Volts
1 Volt	0 Volt	0	1	0	1	Volts
1 Volt	1 Volt	1	1	0	0	Volts

Combinational Logic Symbols

- Common combinational logic systems have standard symbols called logic gates
  - Buffer, NOT
  - AND, NAND
  - OR, NOR

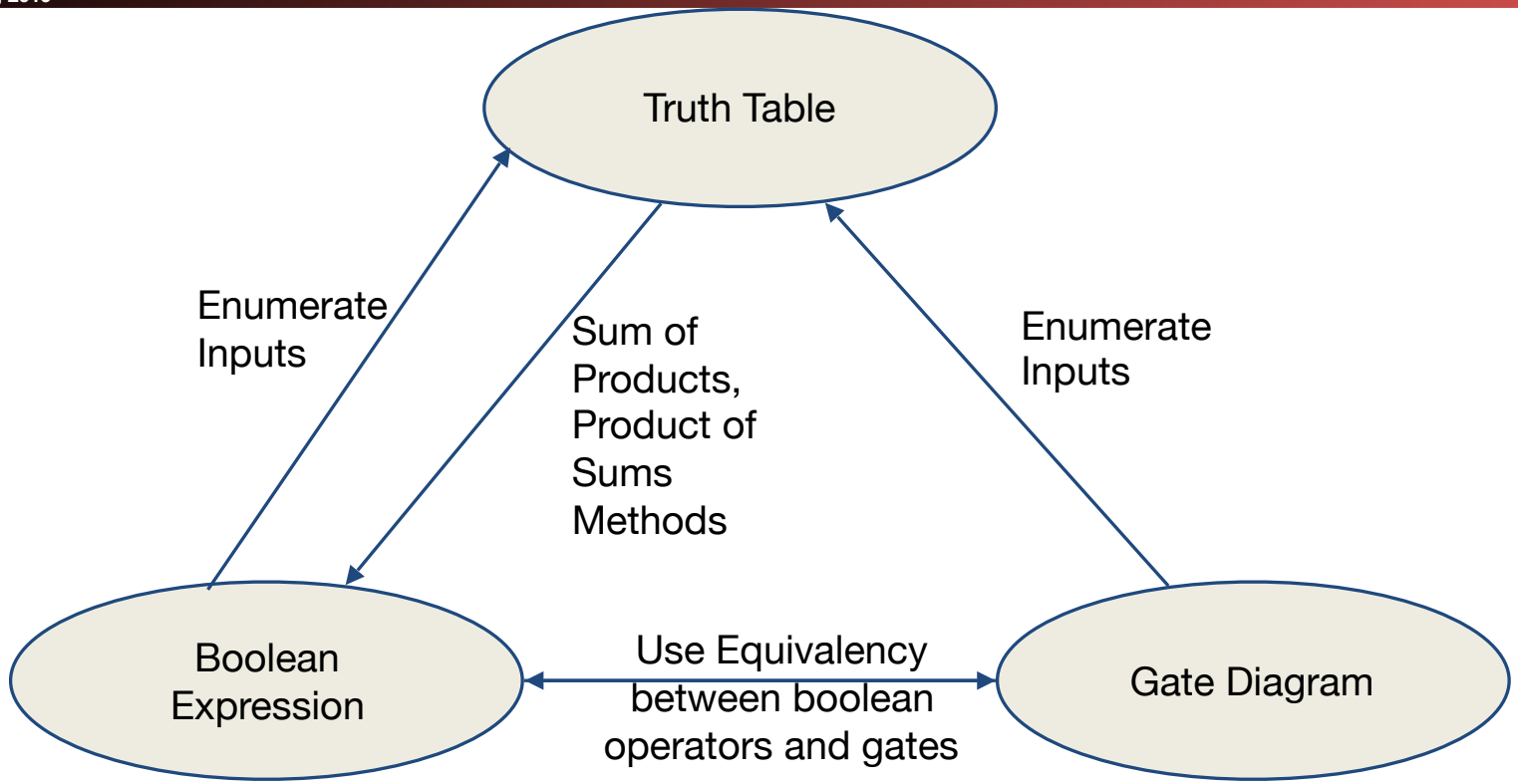


Inverting versions (NOT, NAND, NOR) easiest to implement with CMOS transistors (the switches we have available and use most)

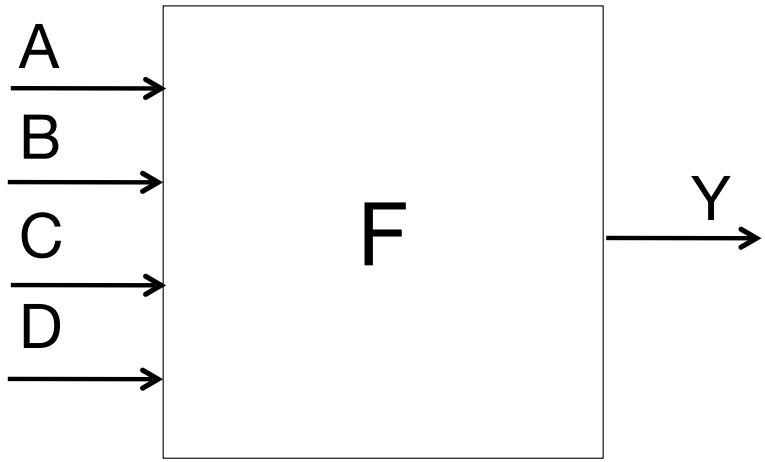
Boolean Algebra

- Use plus "+" for OR
  - "logical sum"
- Use product for AND (a•b or implied via ab)
  - "logical product"
- "Hat" to mean complement (NOT)
- Thus
- $ab + a + \overline{c}$
  $= a \cdot b + a + \overline{c}$
  = (a AND b) OR a OR (NOT c)

Representations of Combinational Logic (groups of logic gates)



Truth Tables for Combinational Logic



a	b	c	d	y
0	0	0	0	F(0,0,0,0)
0	0	0	1	F(0,0,0,1)
0	0	1	0	F(0,0,1,0)
0	0	1	1	F(0,0,1,1)
0	1	0	0	F(0,1,0,0)
0	1	0	1	F(0,1,0,1)
0	1	1	0	F(0,1,1,0)
1	1	1	1	F(0,1,1,1)
1	0	0	0	F(1,0,0,0)
1	0	0	1	F(1,0,0,1)
1	0	1	0	F(1,0,1,0)
1	0	1	1	F(1,0,1,1)
1	1	0	0	F(1,1,0,0)
1	1	0	1	F(1,1,0,1)
1	1	1	0	F(1,1,1,0)
1	1	1	1	F(1,1,1,1)

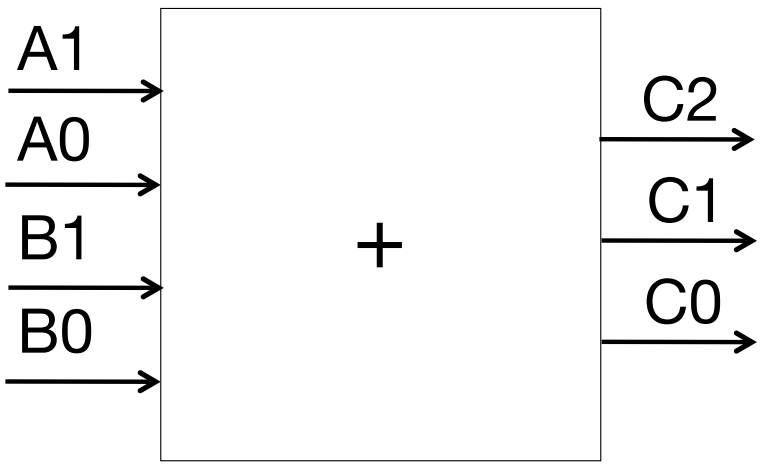
Exhaustive list of the output value generated for each combination of inputs

How many logic functions can be defined with N inputs?

Truth Table Example #1:
y= F(a,b): 1 iff a ≠ b

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

Truth Table Example #2: 2-bit Adder



How Many Rows?

A	B	C
a_1a_0	b_1b_0	$c_2c_1c_0$

Truth Table Example #3: 32-bit Unsigned Adder

A	B	C
000 ... 0	000 ... 0	000 ... 00
000 ... 0	000 ... 1	000 ... 01
.	.	.
.	.	.
.	.	.
111 ... 1	111 ... 1	111 ... 10

How Many Rows?

Truth Table Example #4: 3-input Majority Circuit

a	b	c	y
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

Y = ¯ ;

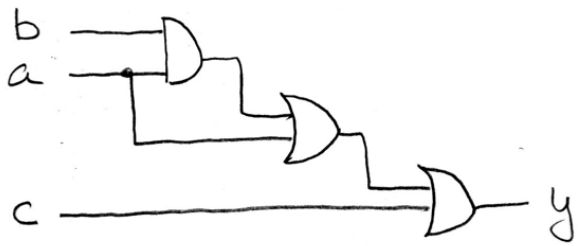
This is called *Sum of Products* form;
Just another way to represent the TT
as a logical expression

More simplified forms
(fewer gates and wires)

And in Conclusion, …

- Multiple Hardware Representations
  - Analog voltages quantized to represent logic 0 and logic 1
  - Transistor switches form gates: AND, OR, NOT, NAND, NOR
  - Truth table mapped to gates for combinational logic design
  - Boolean algebra for gate minimization

Boolean Algebra: Circuit & Algebraic Simplification



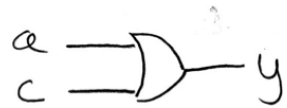
original circuit

$$y = ((ab) + a) + c$$

equation derived from original circuit

$$= ab + a + c$$
$$= a(b + 1) + c$$
$$= a(1) + c$$
$$= a + c$$

algebraic simplification



simplified circuit

Laws of Boolean Algebra

$X\,\overline{X} = 0$	$X + \overline{X} = 1$	Complementarity
$X\,0 = 0$	$X + 1 = 1$	Laws of 0's and 1's
$X\,1 = X$	$X + 0 = X$	Identities
$X\,X = X$	$X + X = X$	Idempotent Laws
$X\,Y = Y\,X$	$X + Y = Y + X$	Commutativity
$(X\,Y)\,Z = Z\,(Y\,Z)$	$(X + Y) + Z = Z + (Y + Z)$	Associativity
$X\,(Y + Z) = X\,Y + X\,Z$	$X + Y\,Z = (X + Y)\,(X + Z)$	Distribution
$X\,Y + X = X$	$(X + Y)\,X = X$	Uniting Theorem
$\overline{X}\,Y + X = X + Y$	$(\overline{X} + Y)\,X = X\,Y$	Uniting Theorem v. 2
$\overline{(X\,Y)} = \overline{X} + \overline{Y}$	$\overline{(X + Y)} = \overline{X}\,\overline{Y}$	DeMorgan's Law

Boolean Algebraic Simplification Example

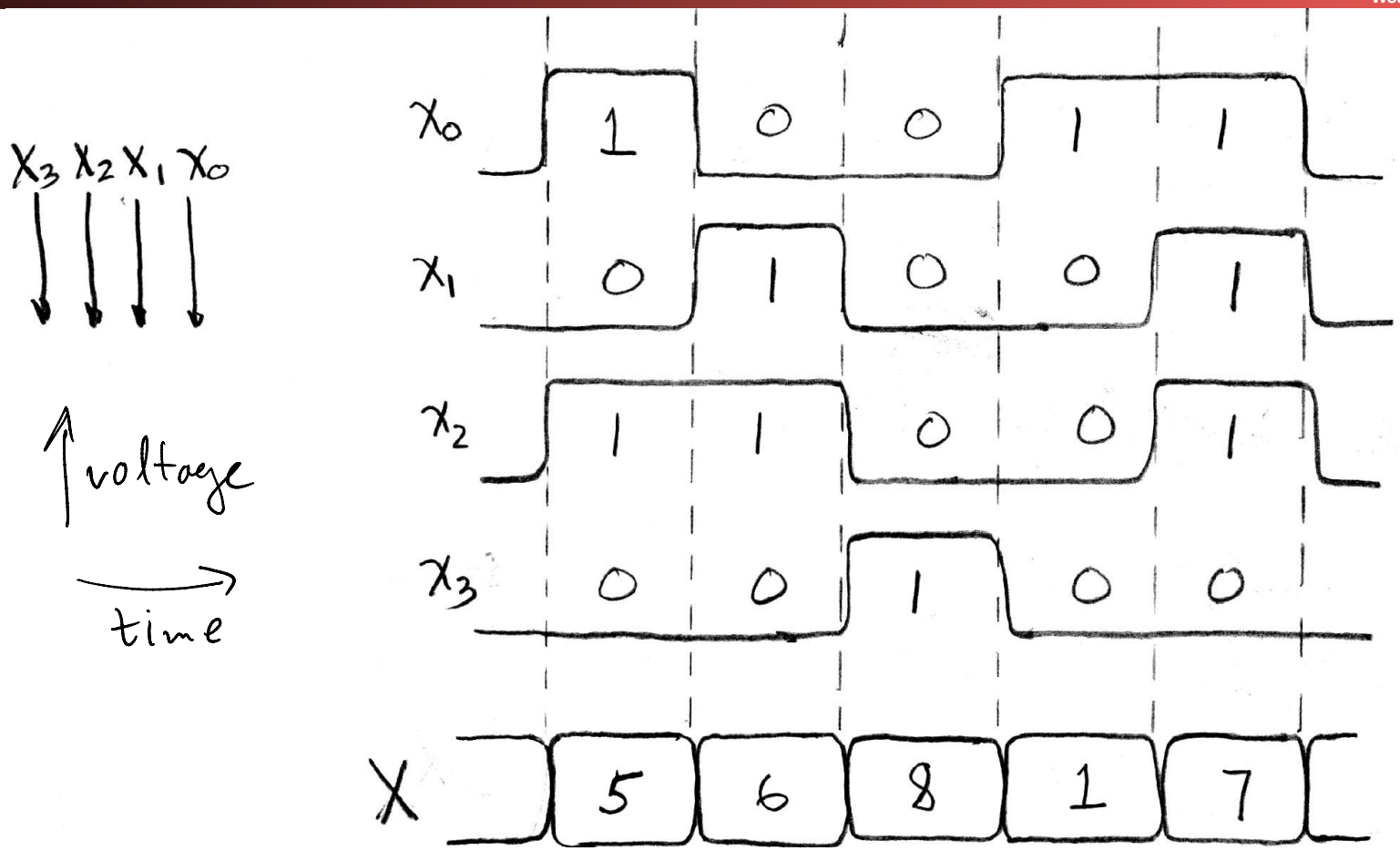
- $$y = ab + a + c$$

Boolean Algebraic Simplification Example

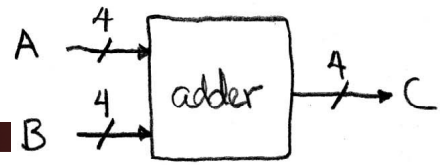
a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$
\begin{aligned}
y &= ab + a + c \\
&= a(b+1) + c && \textit{distribution, identity} \\
&= a(1) + c && \textit{law of 1's} \\
&= a + c && \textit{identity}
\end{aligned}
$$

Signals and Waveforms: Showing Time & Grouping

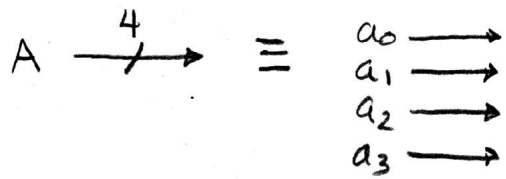


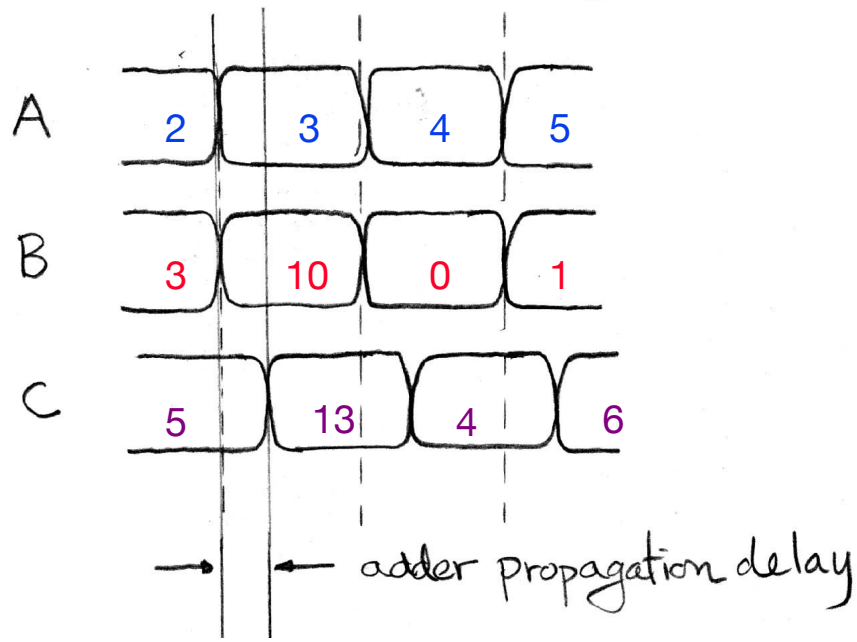
Signals and Waveforms: Circuit Delay



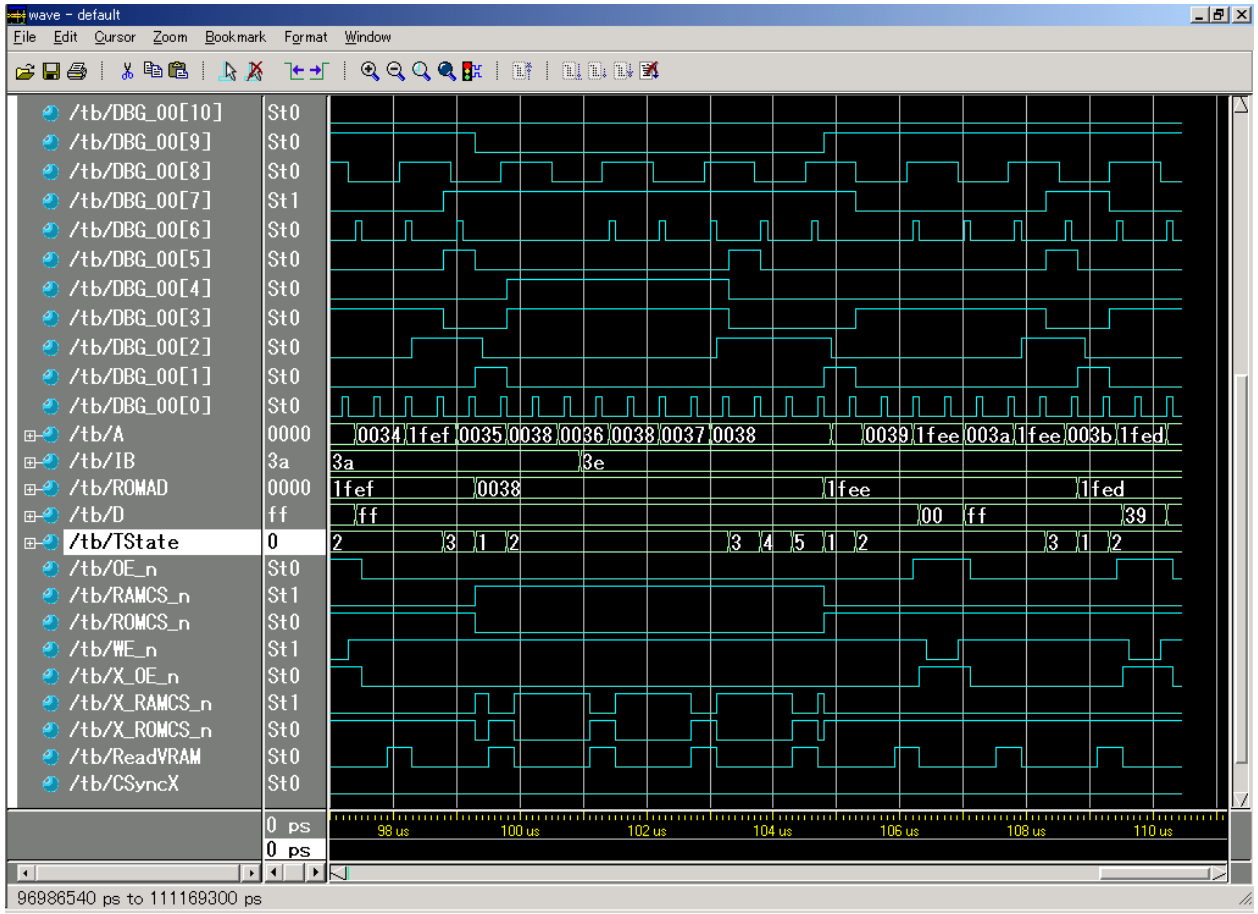
$A = [a_3, a_2, a_1, a_0]$

$B = [b_3, b_2, b_1, b_0]$





Sample Debugging Waveform



Type of Circuits

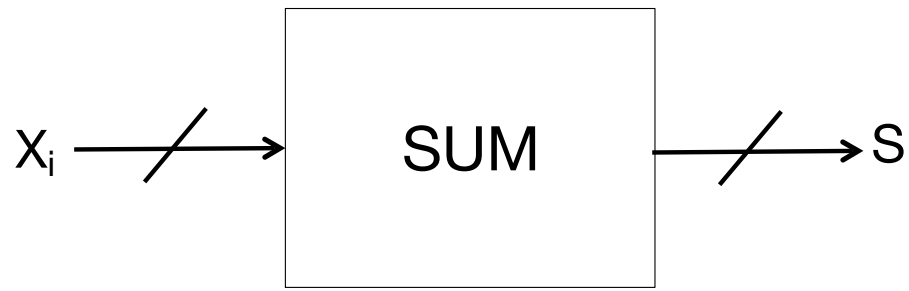
- *Synchronous Digital Systems* consist of two basic types of circuits:
  - Combinational Logic (CL) circuits
    - Output is a function of the inputs only, not the history of its execution
    - E.g., circuits to add A, B (ALUs)
  - Sequential Logic (SL)
    - Circuits that "remember" or store information
    - aka "State Elements"
    - E.g., memories and registers (Registers)

Uses for State Elements

- Place to store values for later re-use:
  - Register files (like x1-x31 in RISC-V)
  - Memory (caches and main memory)
- *Help control flow of information between combinational logic blocks*
  - State elements hold up the movement of information at input to combinational logic blocks to allow for orderly passage

Accumulator Example

Why do we need to control the flow of information?



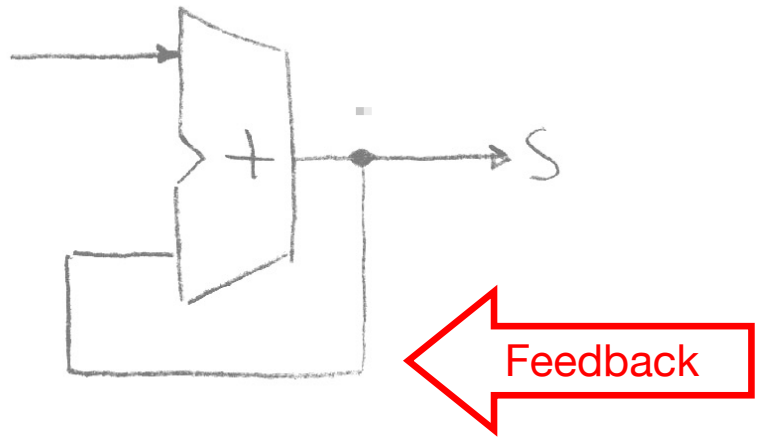
Want:

```
S=0;
for (i=0;i<n;i++)
        S = S + Xi
```

Assume:

- Each X value is applied in succession, one per cycle
- After n cycles the sum is present on S

First Try: Does this work?



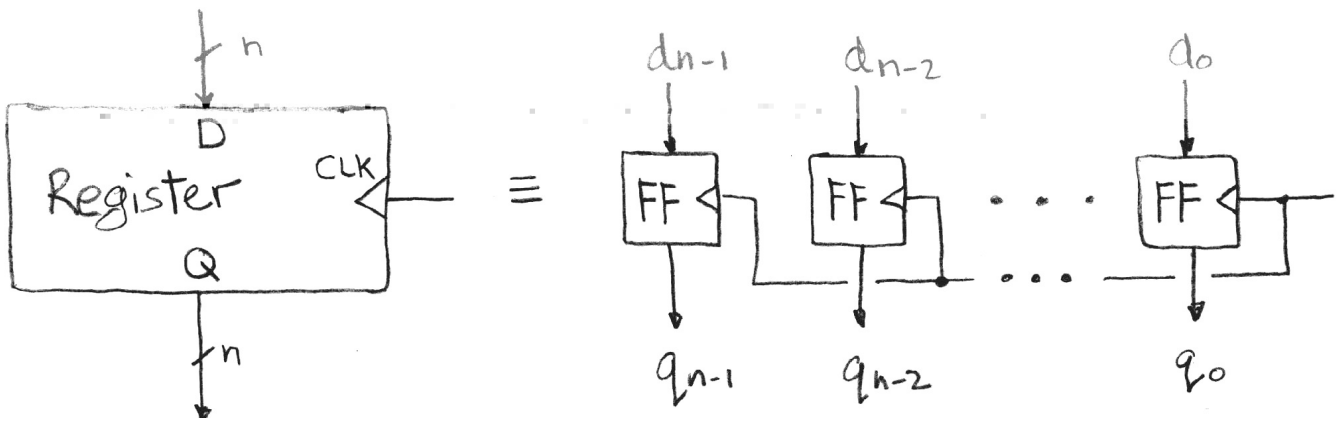
No!

Reason #1: How to control the next iteration of the ‘for’ loop?

Reason #2: How do we say: ‘S=0’?

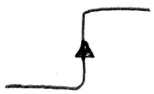
Register Internals

- n instances of a "Flip-Flop"
- Flip-flop name because the output flips and flops between 0 and 1
- D is "data input", Q is "data output"
- Also called "D-type Flip-Flop"

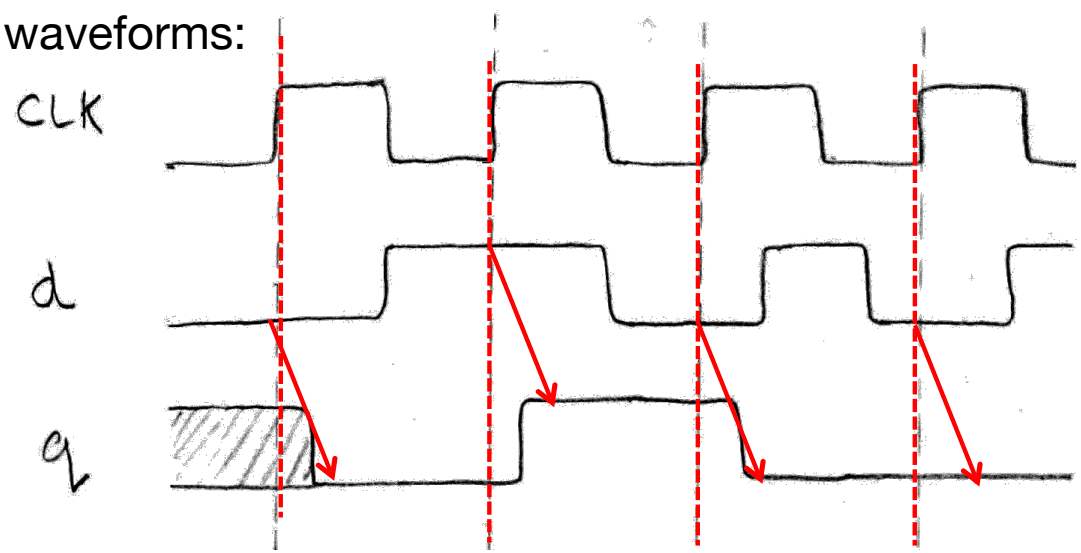


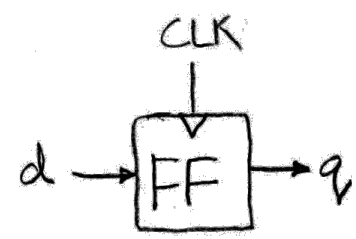
Flip-Flop Operation

- Edge-triggered d-type flip-flop
  - This one is "positive edge-triggered"



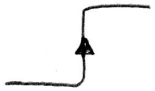
- "On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored."
- Example waveforms:



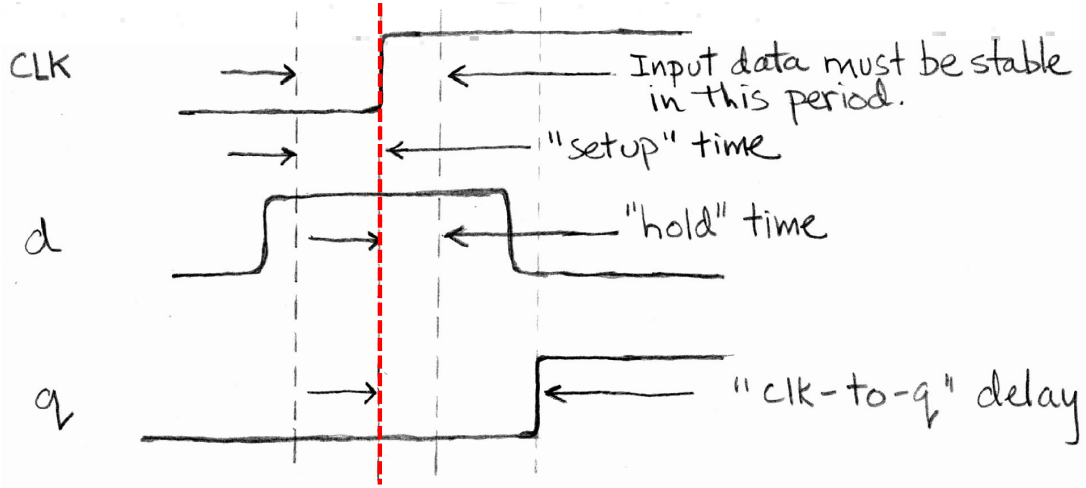


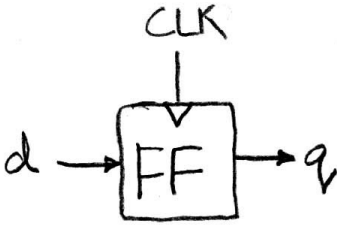
Flip-Flop Timing

- Edge-triggered d-type flip-flop
  - This one is "positive edge-triggered"



- "On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored."
- Example waveforms (more detail):





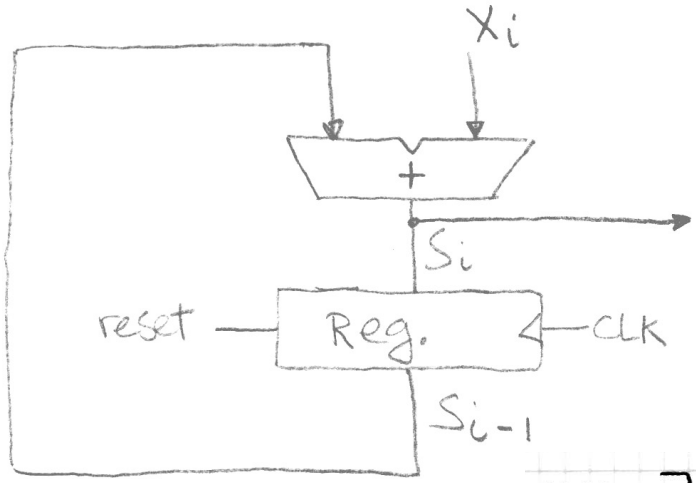
Camera Analogy Timing Terms

- Want to take a portrait – timing right before and after taking picture
- *Set up time* – don't move since about to take picture (open camera shutter)
- *Hold time* – need to hold still after shutter opens until camera shutter closes
- *Time click to data* – time from open shutter until can see image on output (viewscreen)

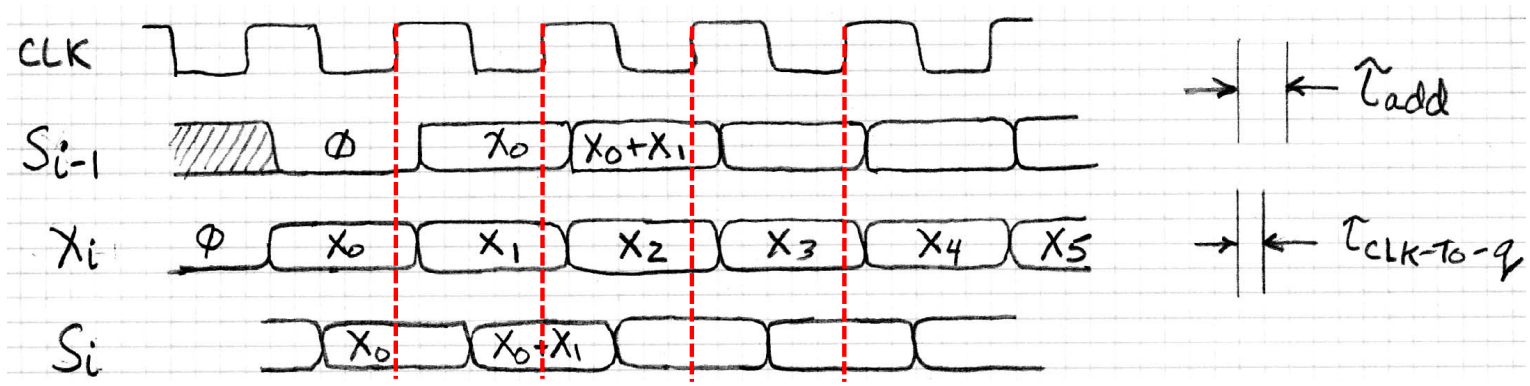
Hardware Timing Terms

- Setup Time: when the input must be stable *before* the edge of the CLK
- Hold Time: when the input must be stable *after* the edge of the CLK
- "CLK-to-Q" Delay: how long it takes the output to change, measured from the edge of the CLK

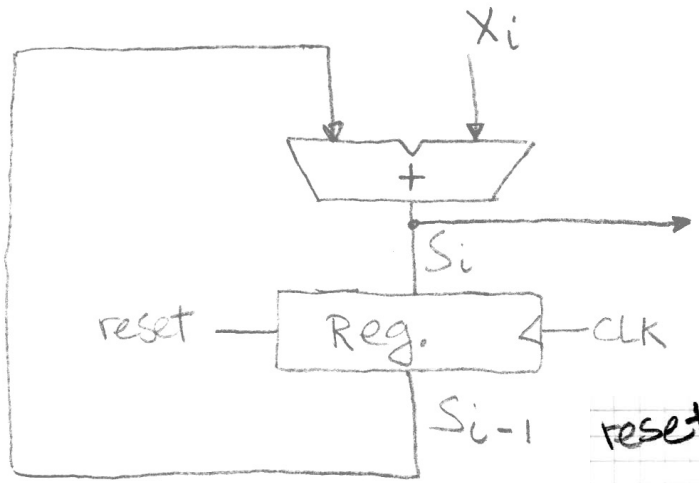
Accumulator Timing 1/2



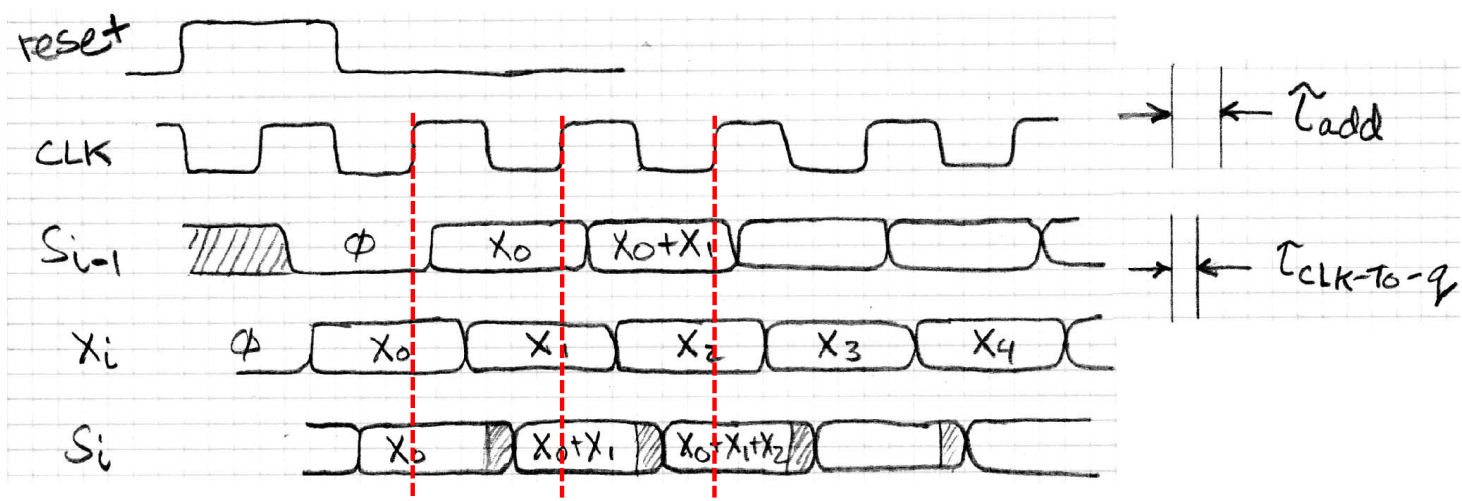
- Reset input to register is used to force it to all zeros (takes priority over D input).
- S_{i-1} holds the result of the ith-1 iteration.
- Analyze circuit timing starting at the output of the register.



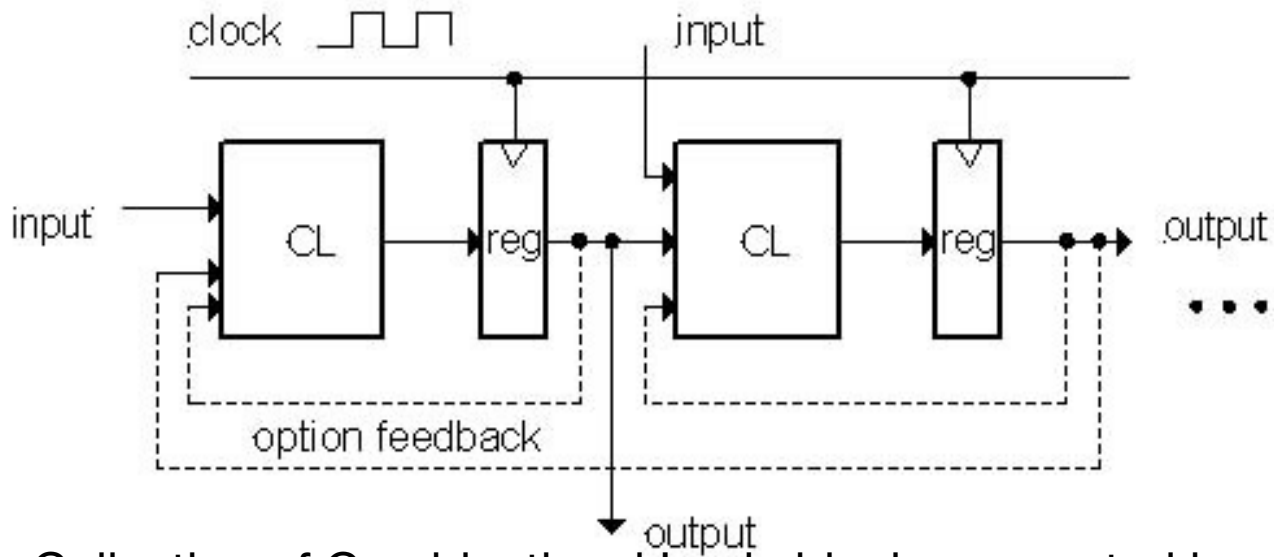
Accumulator Timing 2/2



- reset signal shown.
- Also, in practice X might not arrive to the adder at the same time as S_{i-1}
- S_i temporarily is wrong, but register always captures correct value.
- In good circuits, instability never happens around rising edge of clk.



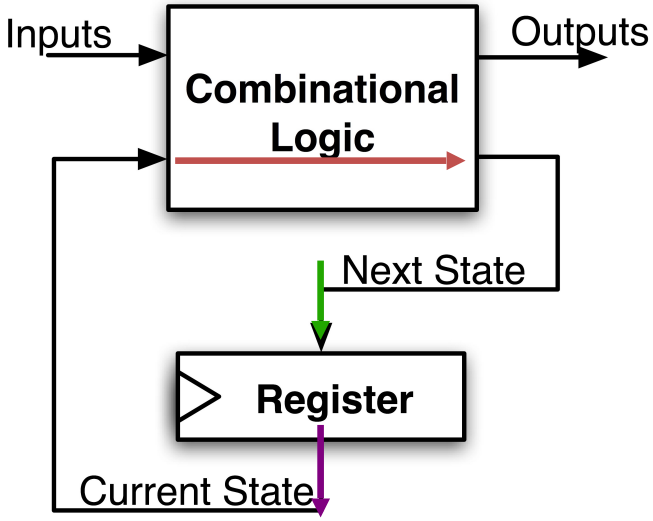
Model for Synchronous Systems



- Collection of Combinational Logic blocks separated by registers
- Feedback is optional
- Clock signal(s) connects only to clock input of registers
- Clock (CLK): steady square wave that synchronizes the system
- Register: several bits of state that samples on rising edge of CLK (positive edge-triggered) or falling edge (negative edge-triggered)

Maximum Clock Frequency

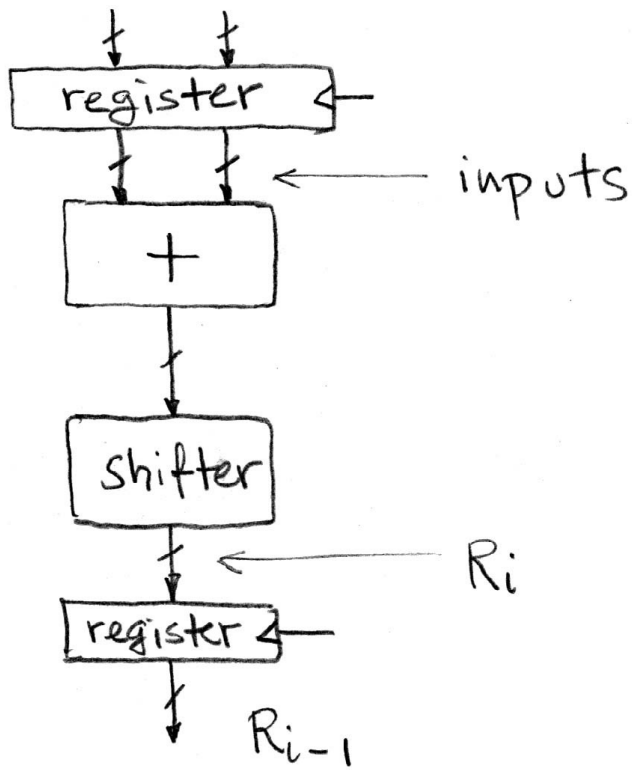
- What is the maximum frequency of this circuit?



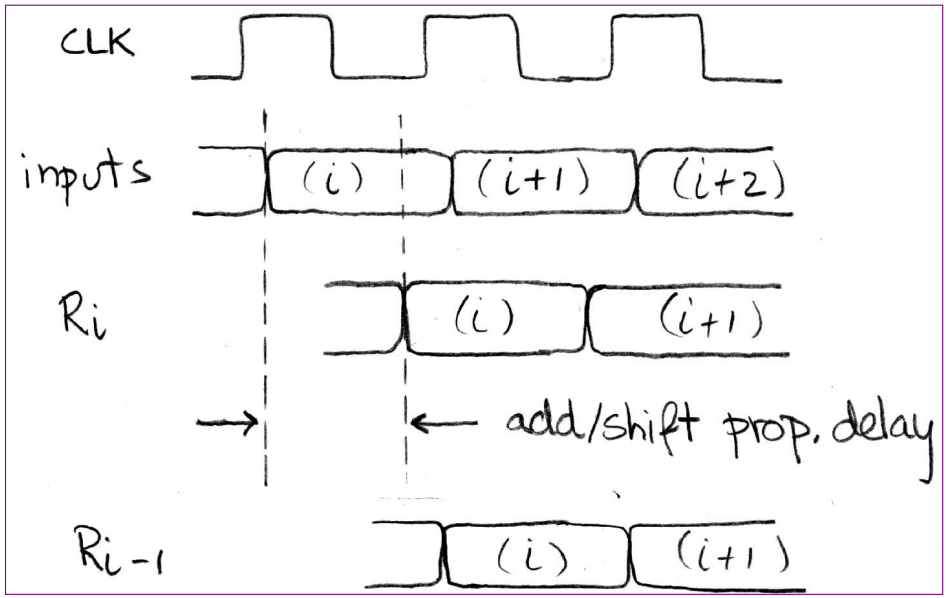
Hint:
Frequency = 1/Period

Period = Max Delay = CLK-to-Q Delay + CL Delay + Setup Time

Critical Paths

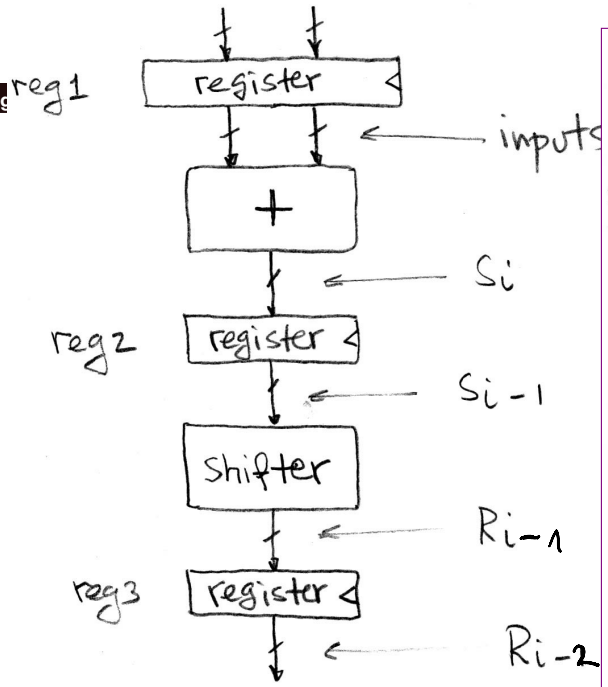


Timing…

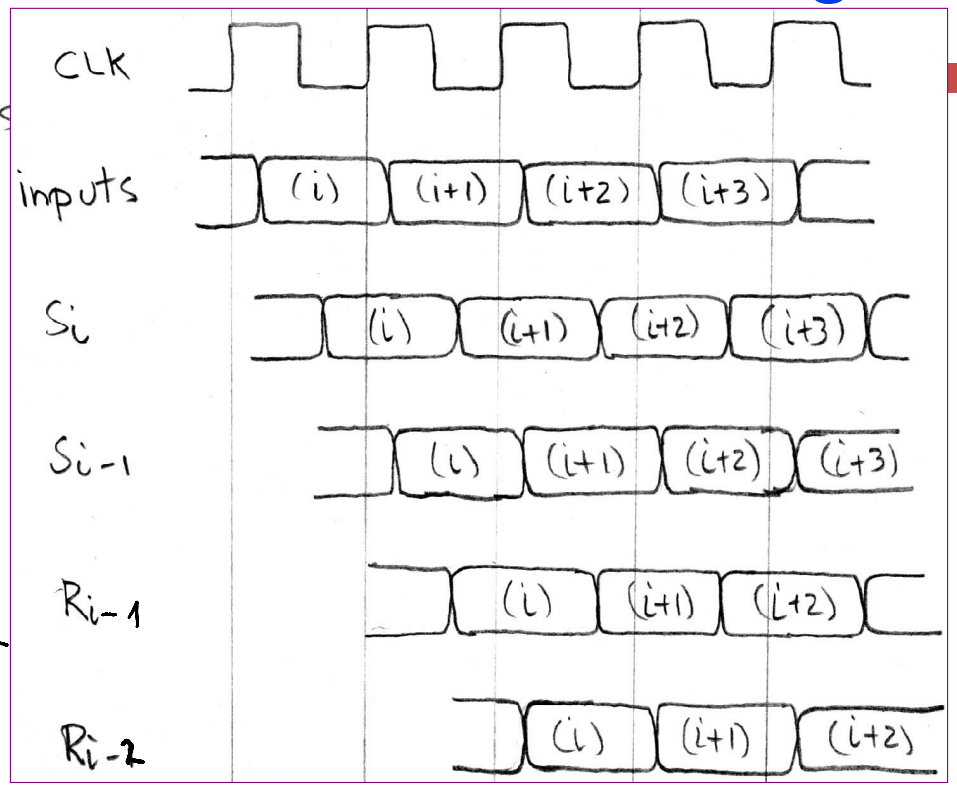


Note: delay of 1 clock cycle from input to output.
Clock period limited by propagation delay of adder/shifter.

Pipelining to improve performance



Timing…



- Insertion of register allows higher clock frequency
- More outputs per second (higher bandwidth)
- But each individual result takes longer (greater latency)

Recap of Timing Terms

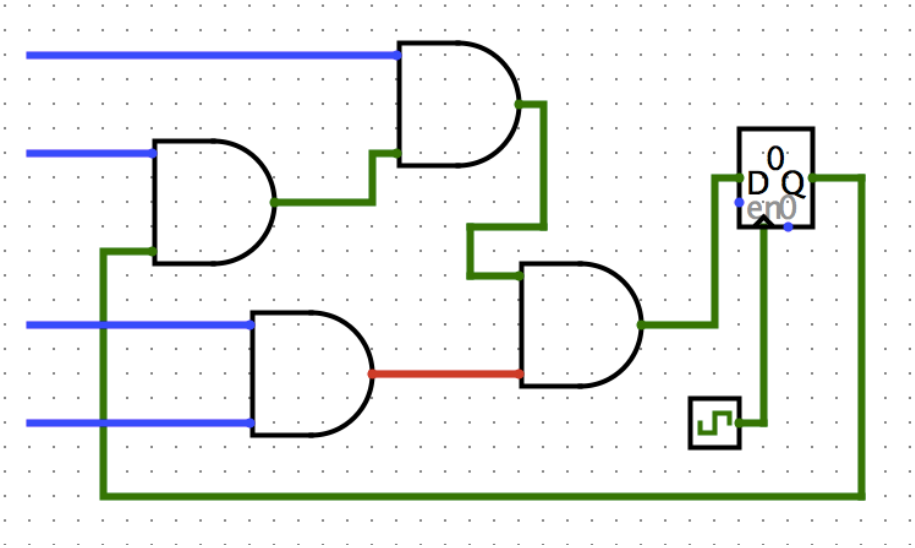
- Clock (CLK) - steady square wave that synchronizes system
- Setup Time - when the input must be stable before the rising edge of the CLK
- Hold Time - when the input must be stable after the rising edge of the CLK
- "CLK-to-Q" Delay - how long it takes the output to change, measured from the rising edge of the CLK

- Flip-flop - one bit of state that samples every rising edge of the CLK (positive edge-triggered)
- Register - several bits of state that samples on rising edge of CLK or on LOAD (positive edge-triggered)

Clickers/Peer Instruction

What is maximum clock frequency? (assume all unconnected inputs come from some register)

- A: 5 GHz
- B: 200 MHz
- C: 500 MHz
- D: 1/7 GHz
- E: 1/6 GHz



Clock->Q 1ns
Setup 1ns
Hold 1ns
AND delay 1ns