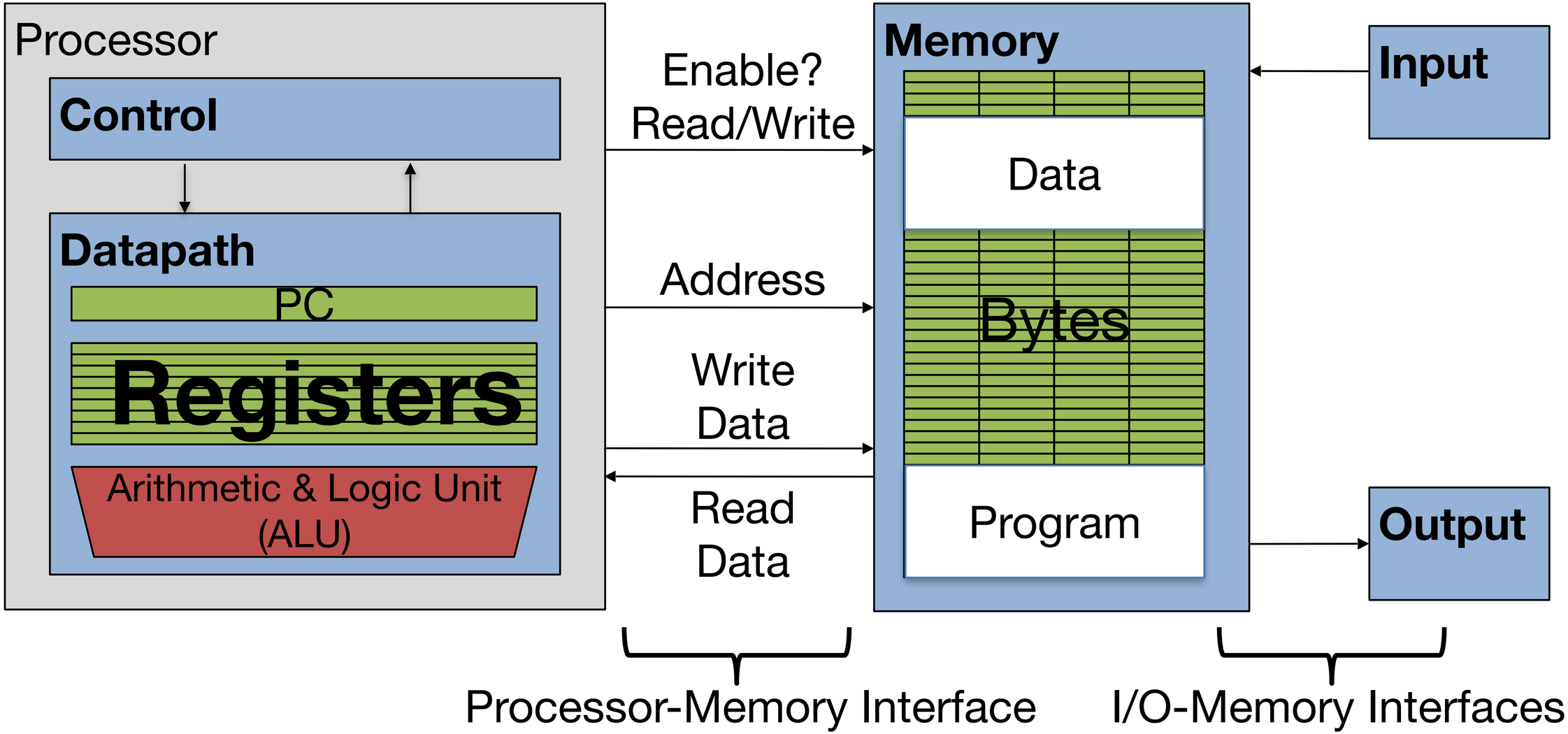
CS61C Fall 2021: Lecture 10

Intro To Digital Circuits
or
Synchronous Digital Systems

Nick's Reminders

- For Project 2: Remember the "campground/frat-house" rule...
  - Callee saved registers are like a campground: always leave it in the same state you got it
    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
  - Caller saved registers are like a frat-house: You can trash them, but if you call another function, they can hold a covid-FRAT-party too...
    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 all the caller-saved registers

Computer Hardware Overview



Hardware Design

- *Next few weeks: how a modern processor is built, starting with basic elements as building blocks*
- Tops reasons to study hardware design:
  1. Knowing how to do hardware design gives you special powers! Go beyond just programming.
  2. To really understand how computers work, need to understand at the physical level (the complete stack)
  3. Understand capabilities and limitations of HW in general and processors in particular. Why is my computer so slow? Why does my battery run down?
  4. There is only so much you can do with standard processors: you may need to design own custom HW for extra performance
  5. Might help you get a job. In addition to tradition hardware companies like Apple, even traditionally software companies Google, Amazon, Facebook do their own hardware design!
  6. Background for more in-depth HW courses (EECS 151, CS 152)
- While it hard to know what you'll need for next 30 years. The principles we teach now will most likely still apply in 30 years when the base technology is different.

Levels of Representation/Interpretation


```
lw      $t0, 0($2)
lw      $t1, 4($2)
sw      $t1, 0($2)
sw      $t0, 4($2)
```

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., RISC-V)

Assembler

Machine Language Program (RISC-V)

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

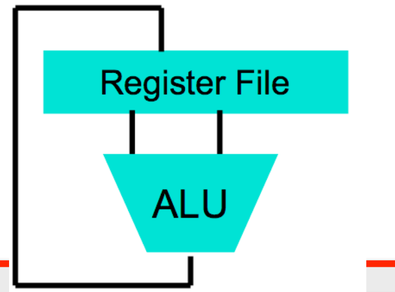
Architecture Implementation

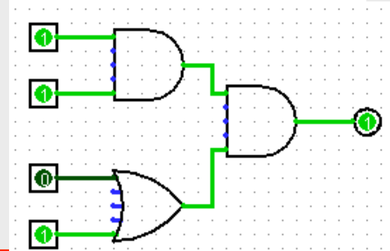
Logic Circuit Description (Circuit Schematic Diagrams)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Anything can be represented as a *number*, i.e., data or instructions

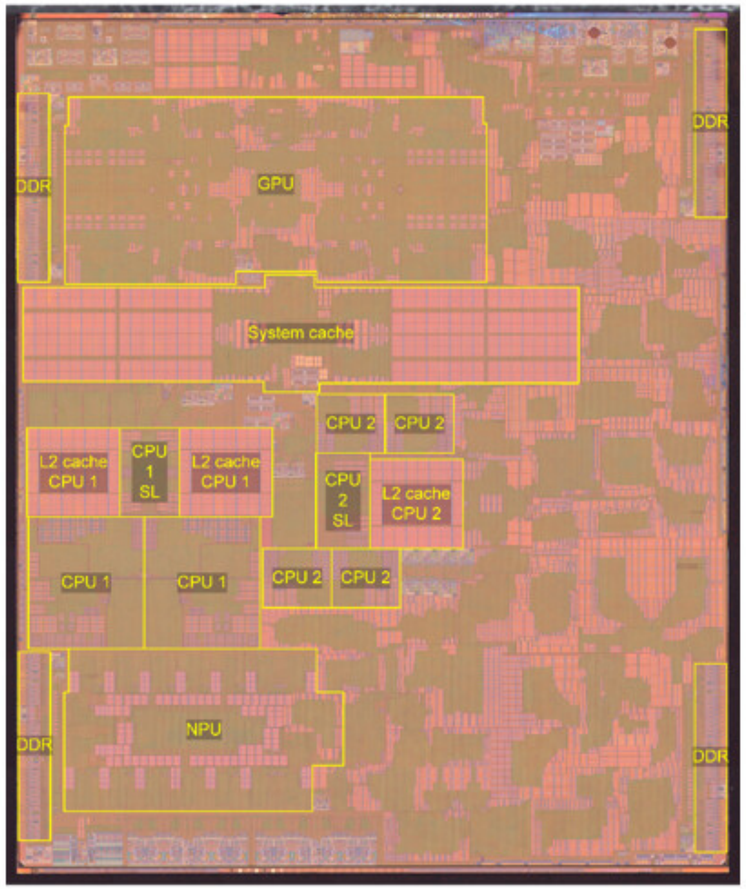
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



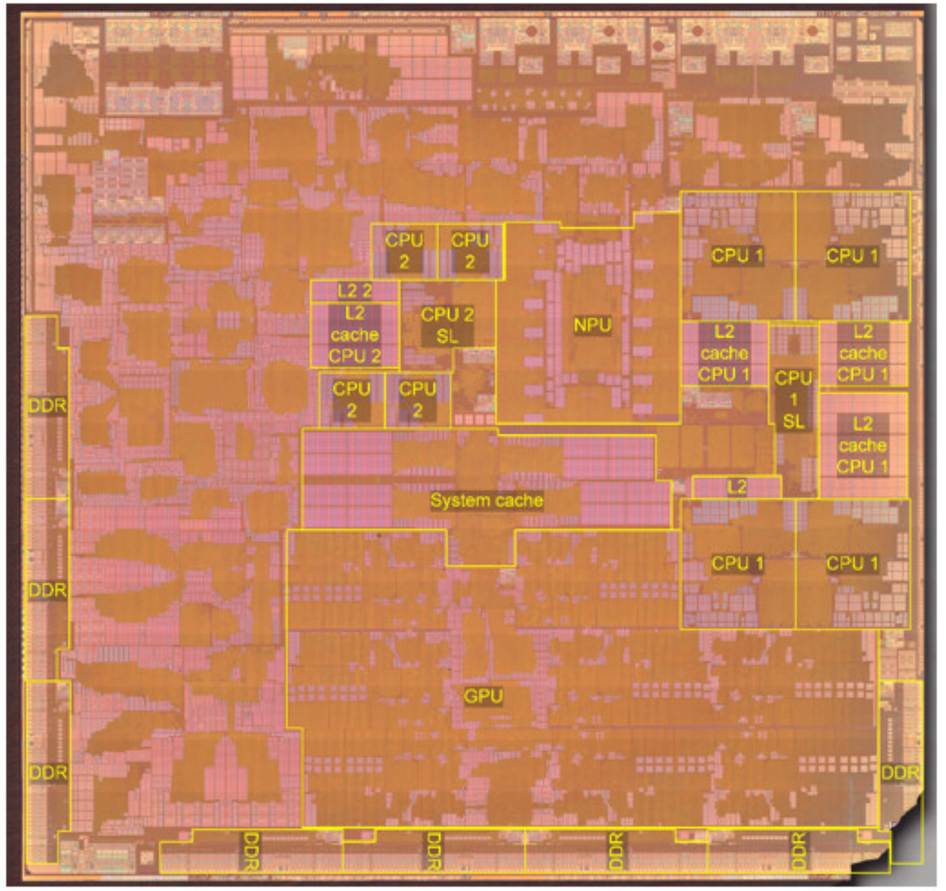


We are here!

Some IC photos



A14 Bionic

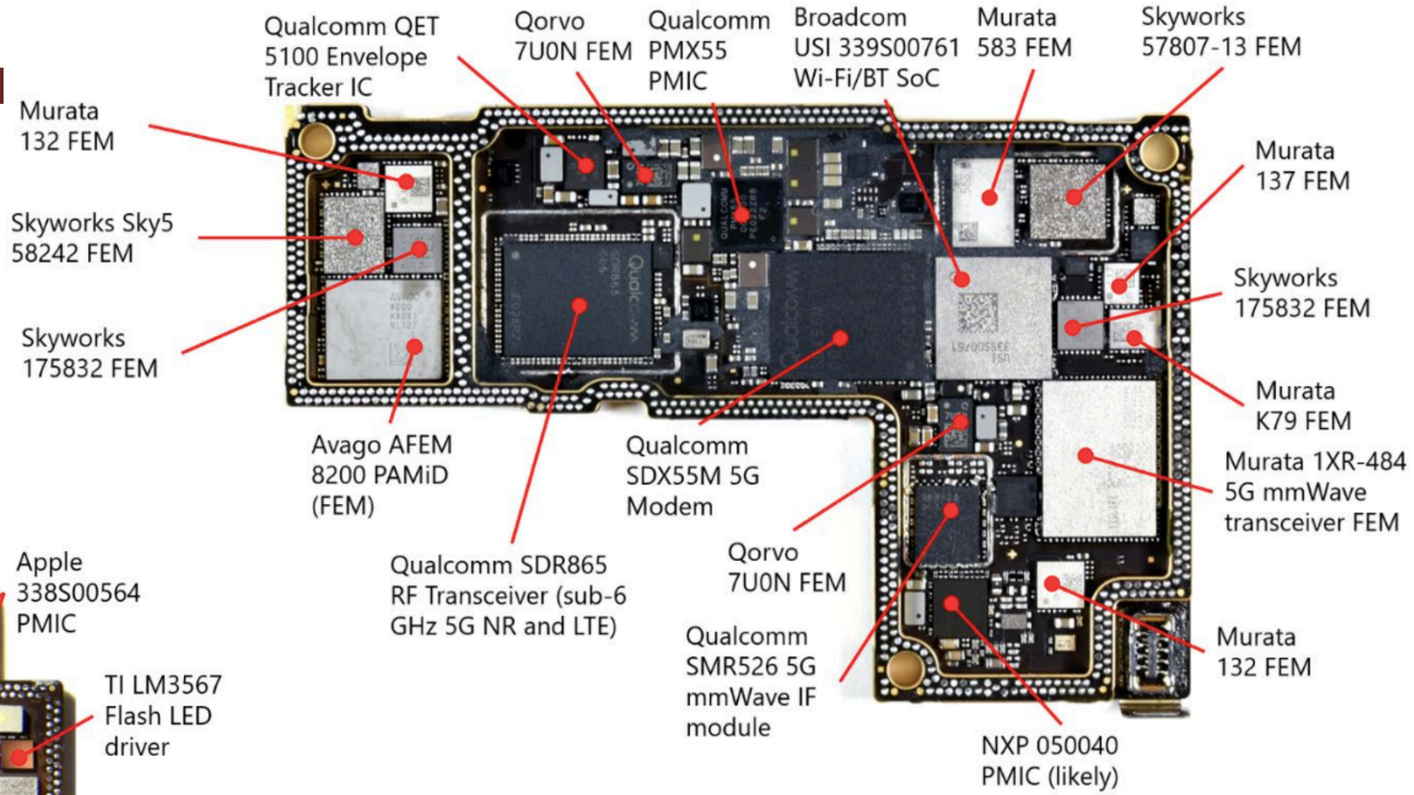


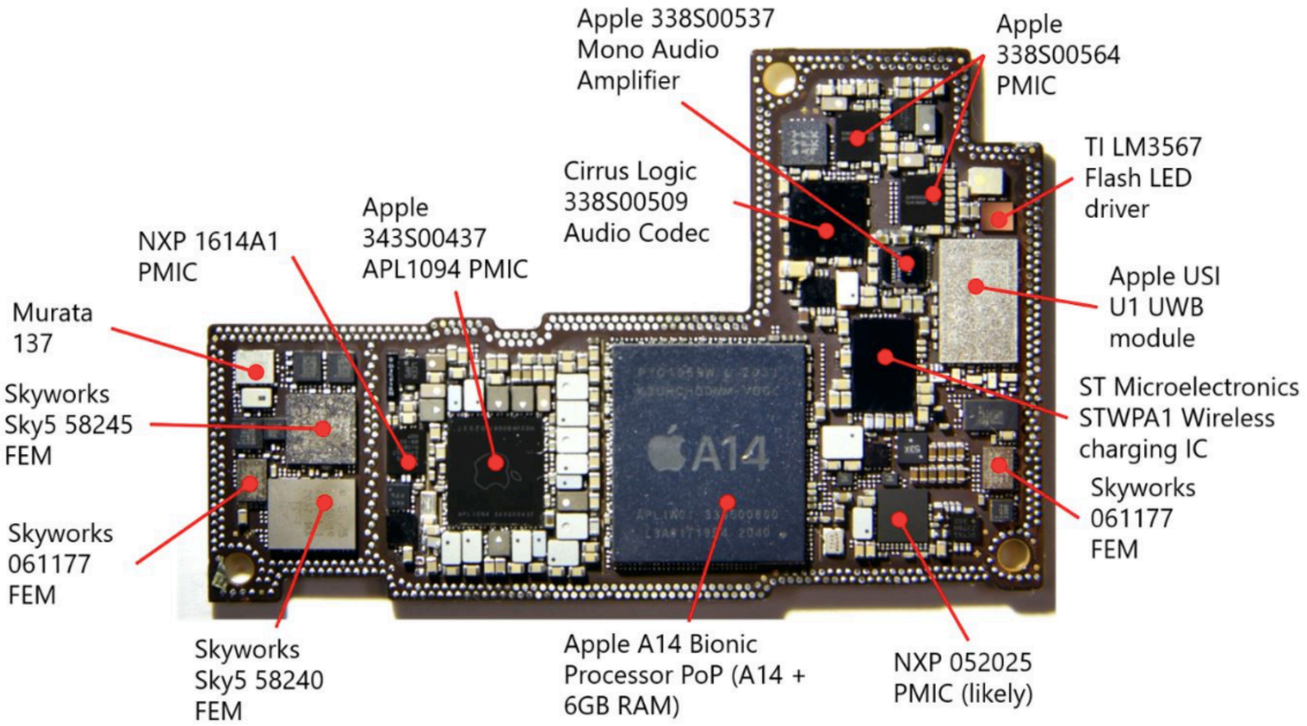
M1

- Apple Implementation of ARM v8.1-a
- 16M Transistors!
- 5W power consumption
- 5 nm process technology
- eight cores divided into two clusters
- four cores working at 3.2 GHz
- four cores working at 2.0 GHz
- CPU supports 64-bit data
- GPU for working with graphical data

On chip, all circuits are made from transistors and wires (unfortunately we also get "parasitic" resistors, capacitors, and inductions.)

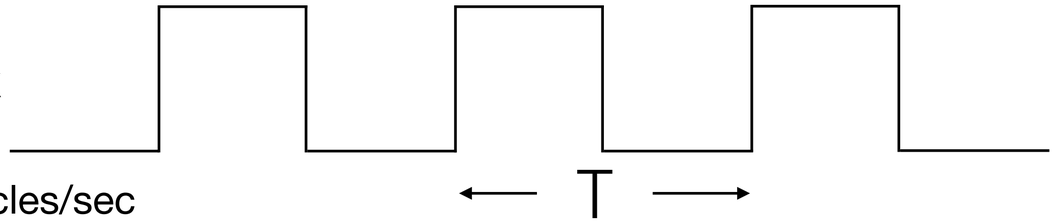
PC boards


Qualcomm QET 5100 Envelope Tracker IC
Qorvo 7U0N FEM
Qualcomm PMX55 PMIC
Broadcom USI 339S00761 Wi-Fi/BT SoC
Murata 583 FEM
Skyworks 57807-13 FEM
Murata 132 FEM
Skyworks Sky5 58242 FEM
Skyworks 175832 FEM
Murata 137 FEM
Skyworks 175832 FEM
Murata K79 FEM
Avago AFEM 8200 PAMiD (FEM)
Qualcomm SDR865 RF Transceiver (sub-6 GHz 5G NR and LTE)
Qualcomm SDX55M 5G Modem
Qorvo 7U0N FEM
Murata 1XR-484 5G mmWave transceiver FEM
Qualcomm SMR526 5G mmWave IF module
Murata 132 FEM
NXP 050040 PMIC (likely)


Apple 338S00537 Mono Audio Amplifier
Apple 338S00564 PMIC
Cirrus Logic 338S00509 Audio Codec
Apple 343S00437 APL1094 PMIC
NXP 1614A1 PMIC
TI LM3567 Flash LED driver
Apple USI U1 UWB module
Murata 137
Skyworks Sky5 58245 FEM
ST Microelectronics STWPA1 Wireless charging IC
Skyworks 061177 FEM
Skyworks 061177 FEM
Skyworks Sky5 58240 FEM
Apple A14 Bionic Processor PoP (A14 + 6GB RAM)
NXP 052025 PMIC (likely)

Synchronous Digital Systems: Almost Every Processor etc...

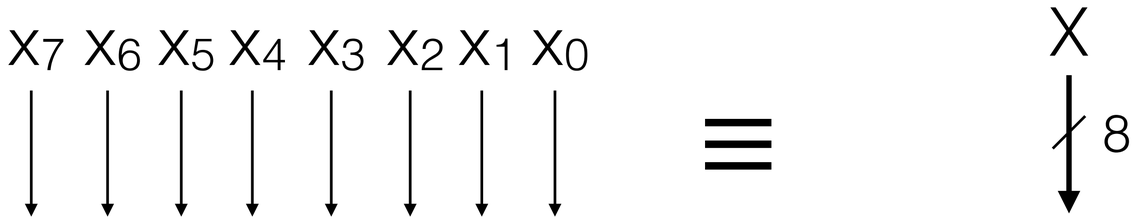
- ***Synchronous***:
  - All operations and communication coordinated by a central clock
    - "Heartbeat" of the system!
    - Typical T=1ns = 10^{-9} sec, F = 1/T = 1GHz = 10^{9} cycles/sec, Billions of cycles/sec
    - ***Anynchronous*** systems: actions and communications between components is locally coordinated. Much harder to design & debug



- ***Digital***:
  - Represent all values by finite set of numerals/symbols, "discrete" representation of values
  - ***Analog*** computing: can represent a continuous values
- ***Also:***
  - Two 2 digits (binary): **1** and **0.** All symbols/numerals made up of sets of 1/0
  - Electrical signals used to represent 1's and 0's
  - Use electrical signals to represent 1's and 0's.
    - High voltage for 1, low voltage for 0
    - Other (older) technologies have used currents instead of voltages.
  - In Analog circuits, voltage or current used to represent continuous ranges of values (ex: in temperature sensor, 0-1v might represent temperature values from freezing to boiling point)

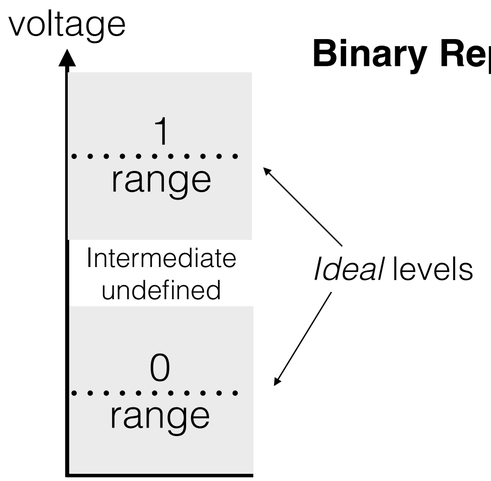
Binary Representation

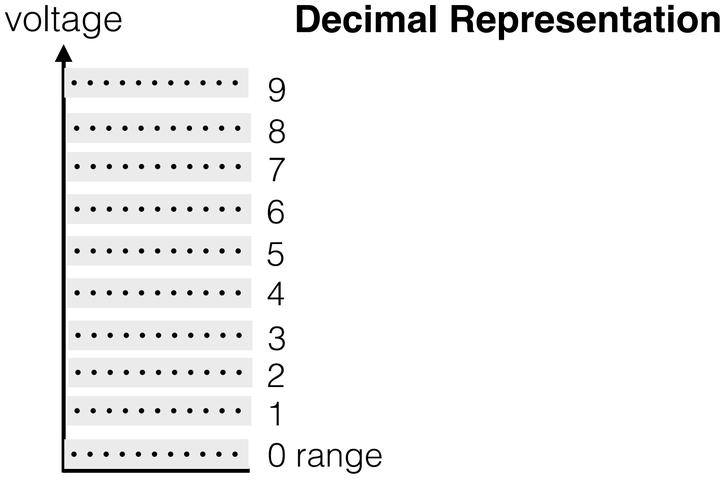
- Wires (electrical nodes) on chip or PC board provide electrical signals and used to represent variables. A wire can take on different values (0 or 1) at different points in time.
- Voltage levels are use to signal 0 or 1. For CMOS technology, 0 volts means 0 and higher voltage (around 0.8v) means 1.
- Collections of bits (nibbles, bytes, words) used to represent a finite set of things.
- For instance, some 8-bit value X:



Why Binary Representation?

- **Reliability - good noise immunity.** All electrical nodes (wires) subjected to interference and non-idealities. Grows worse at smaller sizes.
- Circuit to discriminate between two possible inputs values are simple to implement and have scaled well with Moores law process scaling.
- Early computers built using decimal representation (each node could take on 1 of 10 possible values).
  - Any electrical disturbance generates error.
  - Complex circuits for combining signals.
- Moore's law only possible because of binary representation.
- Flash is a notable exception. Two bits per storage cell (four levels).





Combining Binary Signals

- In RISC-V processor hardware (and generally hardware design) signals are combined using primitive operators to implement the capabilities we need for executing RISC-V instructions (or for other needs)
- Example primitive operators:
  - Given binary variables a, b:

y = AND (a,b) = 1 iff both a, b are 1

y = OR (a,b) = 0 iff both of a, b are 0

y = NOT(a) = 1 iff a is 0

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

AND

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

OR

Truth Tables (TT) define the function (enumerate each output value for each input combination)

a	y
0	1
1	0

NOT

Binary Operators of 2 variables

- How many functions of 2 variable exist?

a	b	y_0	y_1	y_2	.	.	.	$y_?$
0	0	0	0					
0	1	0	0					
1	0	0	0					
1	1	0	1					

- It can be proven, that a small subset of these is sufficient for implementing any discrete function. For now we consider only AND, OR, and NOT.
- Also, we can define primitive operators on > 2 variables - but because of technology limitations, best to keep number small (usually less than 5).

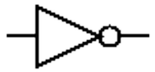
Logic Gates

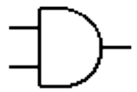
- Transistor-level circuits (gates) implement primitive operators:

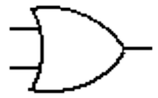
a	b	AND
0	0	0
0	1	0
1	0	0
1	1	1

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

a	NOT
0	0
1	0







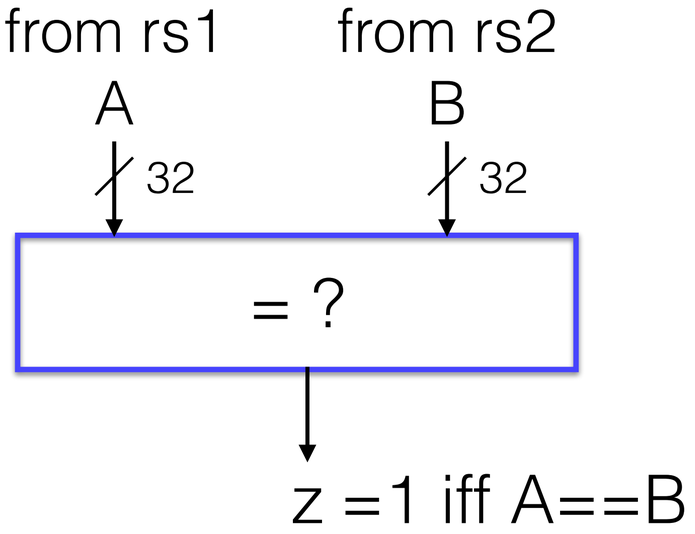
Gate Symbols are used in graphical representation of circuits (schematic diagrams)

- *Next lecture* we see how transistors are combined to make logic gates
- *This lecture* we see how logic gates are combined to make interesting/useful functions

Example of using gates to implement a useful function

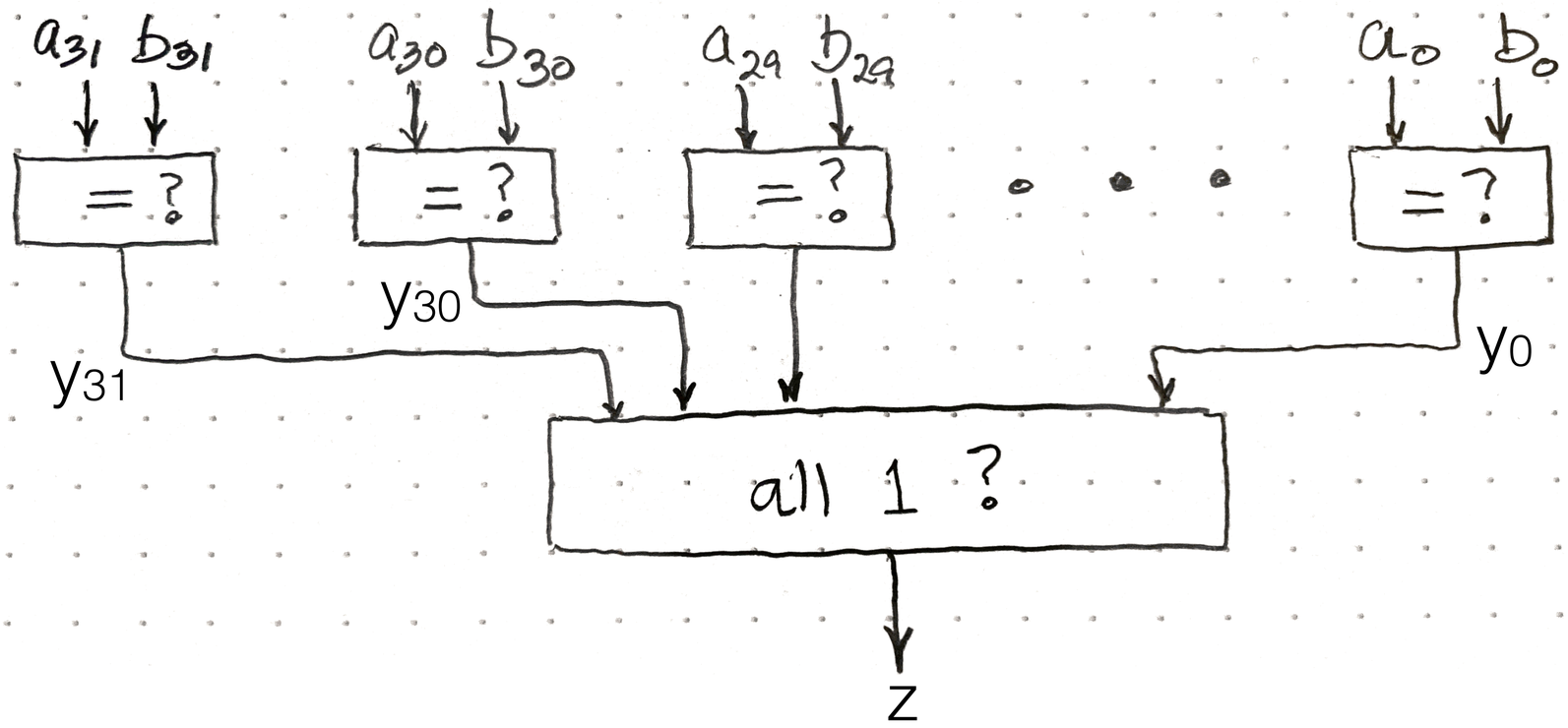
- Recall RISC-V beq instruction: **`beq rs1,rs2,label`**
  - If r1 (value in rs1) == (value in rs2) then go to instruction at label, else go to next instruction
- Somewhere in the processor must be a way (a circuit) to compare the two registers
- Could subtract the two and check for result = 0 (more expensive than necessary)

- Assume a dedicated "equal compare" circuit:



How to implement “= ?” circuit

A == B iff $a_{31}==b_{31}$ and $a_{30}==b_{30}$ and … and $a_0==b_0$



Single bit compare

Check for all 1

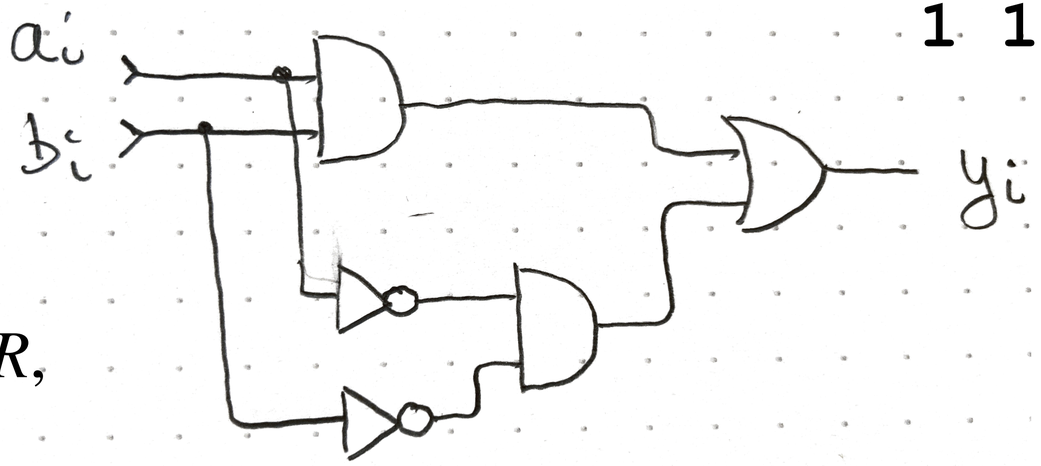
Single bit compare circuit: aka exclusive-nor (xnor)

- Might be in our technology library. If not, can implement with AND/OR/NOT
- Observe, output is a 1 only when a=b=0 or a=b=1
- Therefore with gates:
- Or algebraically:

$$y_i = (a_i \cdot b_i) + (\overline{a_i} \cdot \overline{b_i})$$

- where $\cdot$ means AND, $+$ means OR, $\bar{x}$ means $NOT(x)$

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



- Often we drop the dot, use x' (prime) instead of x (bar), assume AND has precedence over OR

$$y_i = a_i b_i + a_i' b_i'$$

Check-for-all-1 circuit

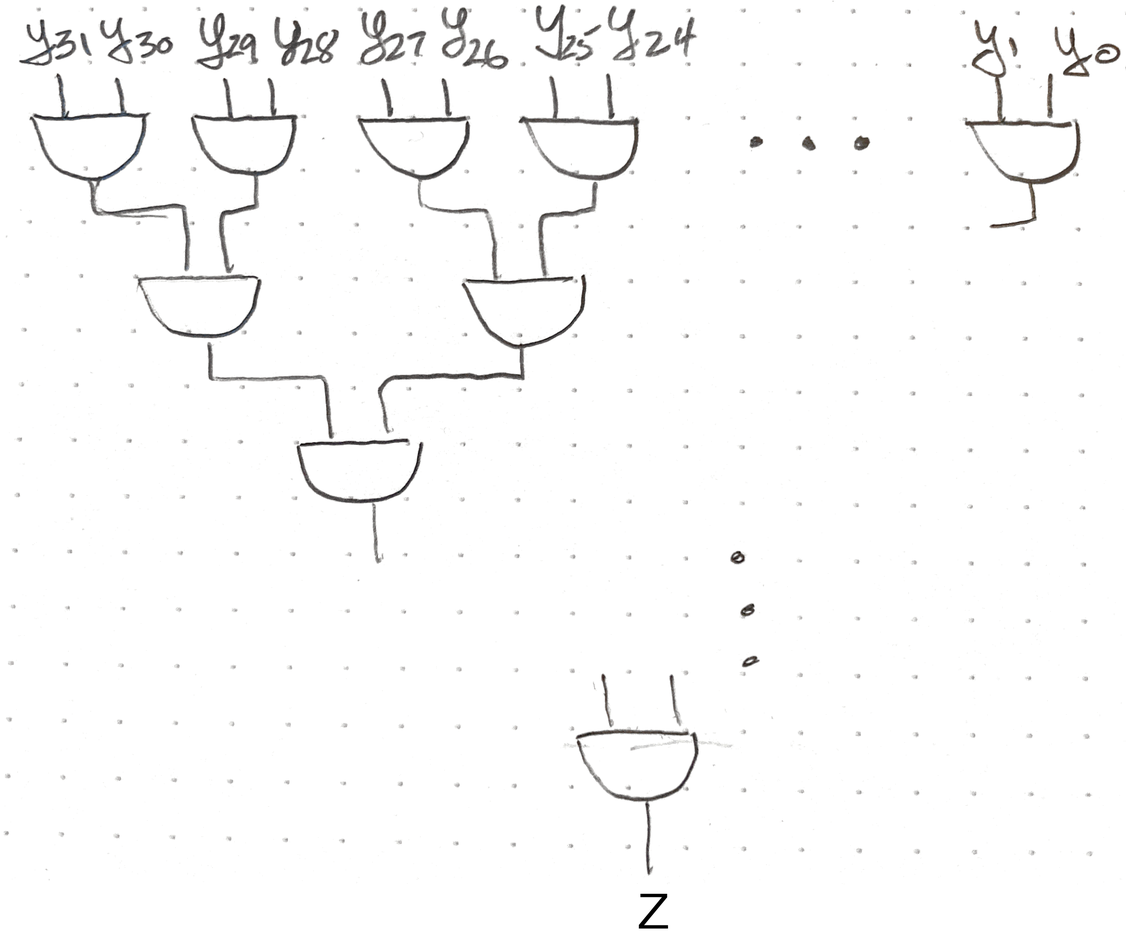
- Functionally, it's a 32 input AND-gate! aka "AND reduction"
- Not practical in our technology, therefore decompose (factor)
- Algebraically:

$$z = y_{31} \cdot y_{30} \cdot y_{29} \cdot \ldots \cdot y_0$$

- AND is *associative*, therefore

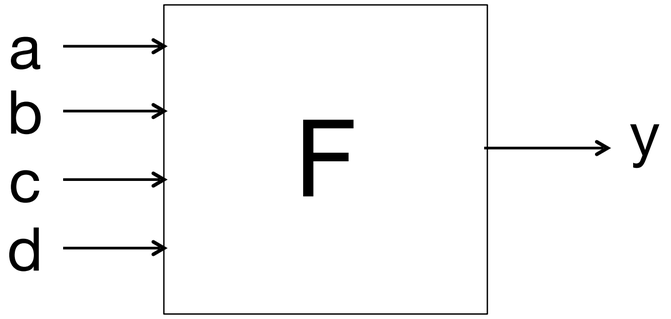
$$z = (y_{31} \cdot y_{30}) \cdot (y_{29} \cdot y_{28}) \cdot \ldots \cdot (y_1 \cdot y_0)$$

- Apply recursively:



How many levels?

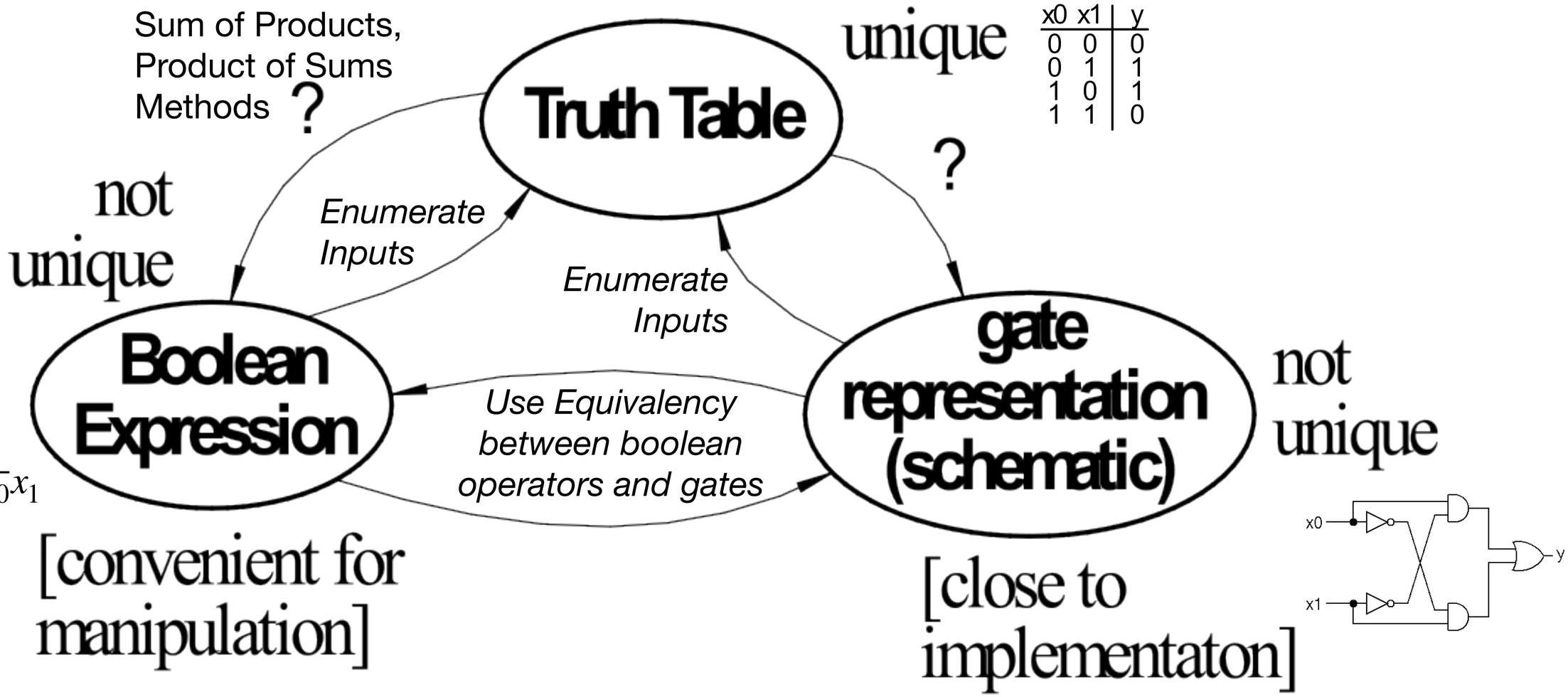
Summary, so far



- These circuits that combine binary signals are called “Combinational Logic Circuits”
- The output, y is a function only of the inputs (a, b, c, …)
- Changes to inputs are reflected immediately at the output (after some technology dependent delay)
- CL circuits have no memory of the past
- Such circuits can always be defined by truth tables (but might be too big in practice), and alternatively represented algebraically, or with gate diagrams.

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)

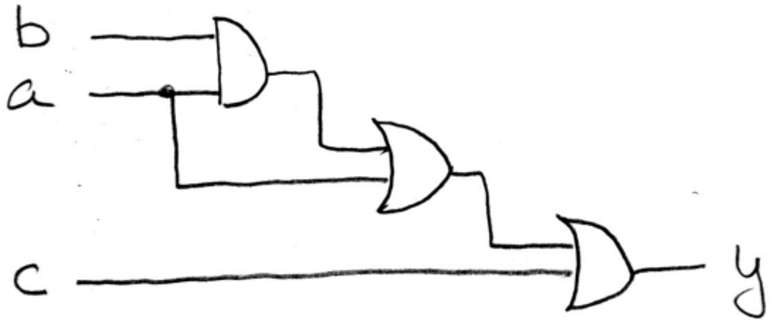
Representations of Combinational Logic Circuits



Historical Note on Boolean Algebra

- 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 (Logic) George Boole
  - Called it "Boolean Algebra" in his honor
- Could apply math to give theory to hardware design, minimization, …

Boolean Algebra: Circuit & Algebraic Simplification



original circuit

$$\downarrow$$

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

equation derived from original circuit

$$\downarrow$$

$$\begin{aligned} &= ab + a + c \\ &= a(b + 1) + c \\ &= a(1) + c \\ &= a + c \end{aligned}$$

algebraic simplification

$$\downarrow$$

simplified circuit

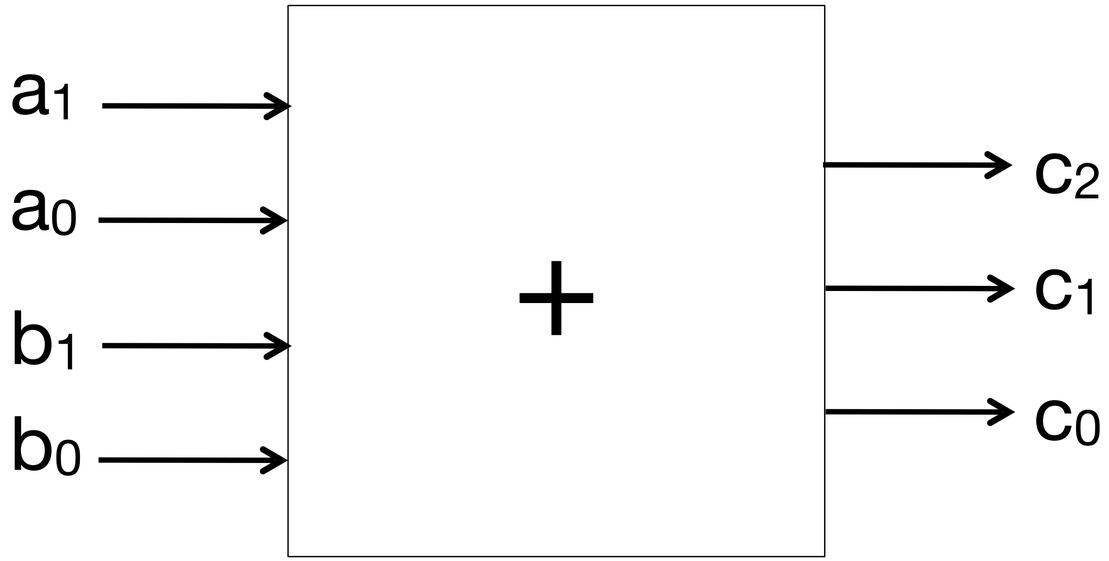
Some Useful Laws of Boolean Algebra

$X \overline{X} = 0$	$X + \overline{X} = 1$	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

Exhaustive proof (perfect induction) good way to prove these.

Combinational-logic circuit Example #2: Adder

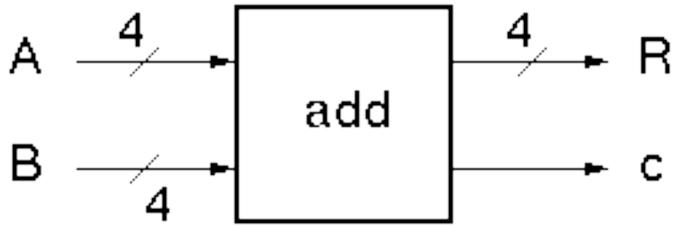
- Start simple: 2-bit wide unsigned adder (forms sum of 2 2-bit numbers)



A a_1a_0	B b_1b_0	C $c_2c_1c_0$
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

- Based on TT, could find Boolean equations, simplify, and convert to logic gates.
- Wider adders?

4-Bit Adder - where decomposition helps



In general: 2^n rows for n inputs. If n=32!

Is there a more efficient (compact) way to specify this function?

- Truth Table Representation:

a3	a2	a1	a0	b3	b2	b1	b0	r3	r2	r1	r0	c
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	0	1	1	0	0	1	1	0
0	0	0	0	0	1	0	0	0	1	0	0	0
⋮												
0	0	1	0	0	0	1	0	0	1	0	0	0
0	0	1	0	0	0	1	1	0	1	0	1	0
⋮												
0	0	0	1	1	1	1	1	0	0	0	0	1

256 rows!

4-bit Adder Example

- Motivate the adder circuit design by hand addition:

```
   a3 a2 a1 a0
+  b3 b2 b1 b0
-------------
c  r3 r2 r1 r0
```

```
   a3 a2 a1 a0
+  b3 b2 b1 b0
-------------
c  r3 r2 r1 r0
```

- Add a0 and b0 as follows:

a	b	r	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

carry to next stage

r = a XOR b = a ⊕ b
c = a AND b = ab

- Add a1 and b1 as follows:

ci	a	b	r	co
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

$r = a \oplus b \oplus c_i$
$co = ab + ac_i + bc_i$

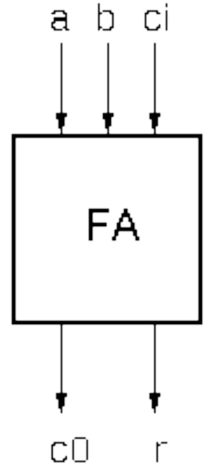
4-bit Adder Example

- In general:

$r_i = a_i \oplus b_i \oplus c_{in}$

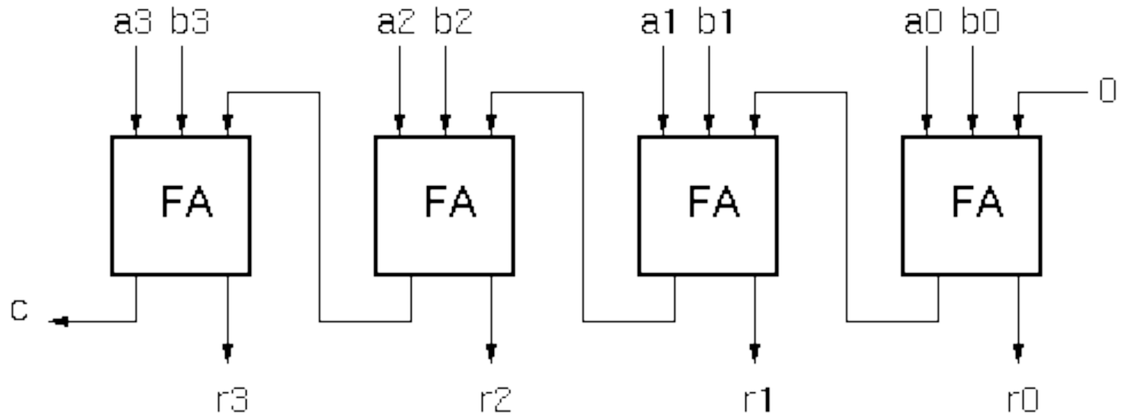
$c_{out} = a_i c_{in} + a_i b_i + b_i c_{in} = c_{in}(a_i + b_i) + a_i b_i$

"Full adder cell"



- Now, the 4-bit adder:

"ripple" adder



Can extend to any number of bits: "n-bit adder"

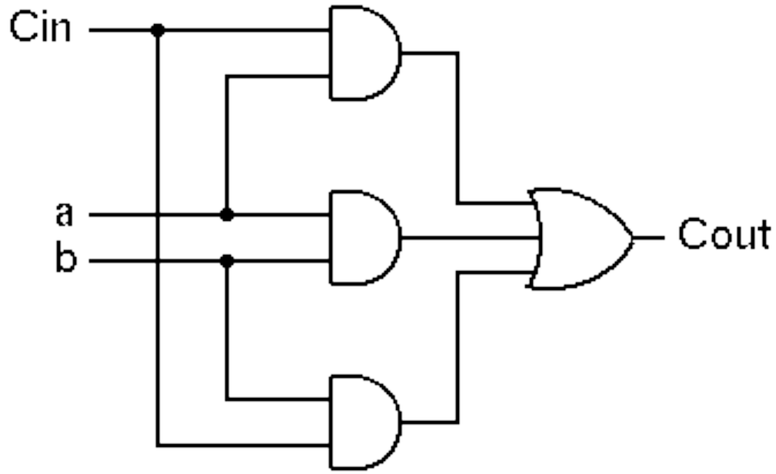
Note: the same circuit works for both unsigned and signed (2's complement)

4-bit Adder Example

- Graphical Representation of FA-cell

$r_i = a_i \oplus b_i \oplus c_{in}$

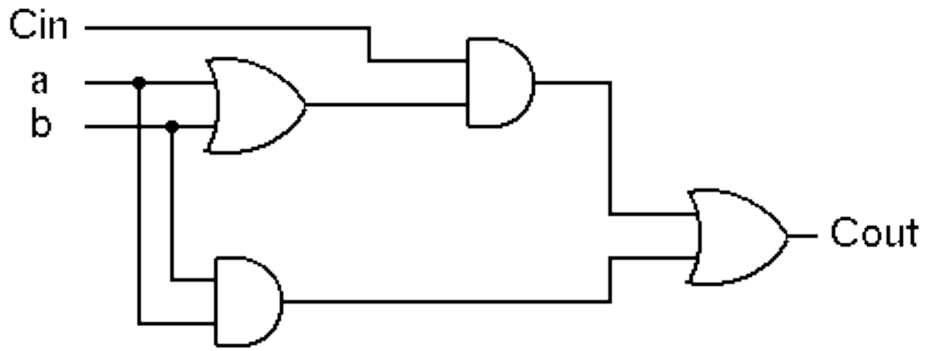
$c_{out} = a_i c_{in} + a_i b_i + b_i c_{in}$

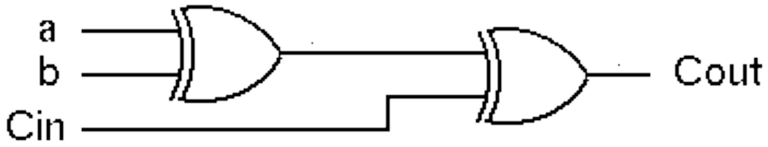


- Alternative Implementation (with only 2-input gates):

$r_i = (a_i \oplus b_i) \oplus c_{in}$

$c_{out} = c_{in}(a_i + b_i) + a_i b_i$





Using Algebraic Simplification for c_{out}

Start by writing sum-of-products "canonical" form. It enumerates all the ways the function can be equal to 1.

Cout = a'bc + ab'c + abc' + abc

= a'bc + ab'c + abc' + abc + abc

= a'bc + abc + ab'c + abc' + abc

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

= (1)bc + ab'c + abc' + abc

= bc + ab'c + abc' + abc + abc

= bc + ab'c + abc + abc' + abc

= bc + a(b' +b)c + abc' +abc

= bc + a(1)c + abc' + abc

= bc + ac + ab(c' + c)

= bc + ac + ab(1)

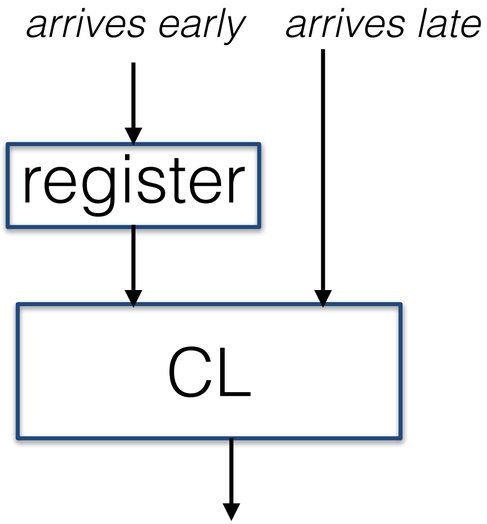
= bc + ac + ab

ci	a	b	r	co
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

aka the "majority function"

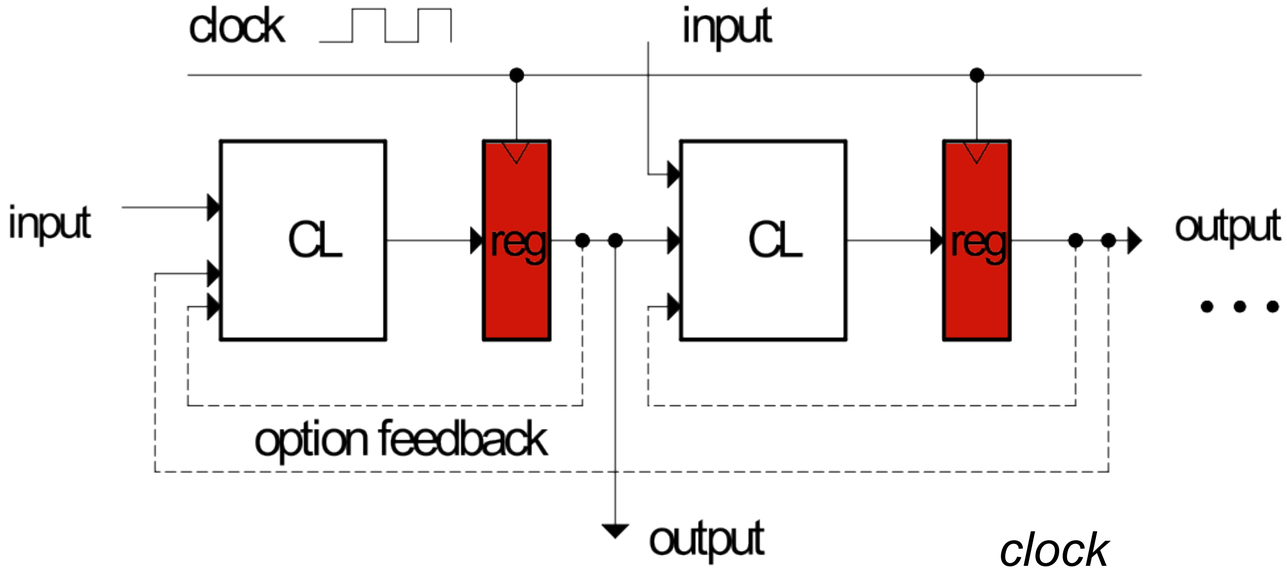
State Elements

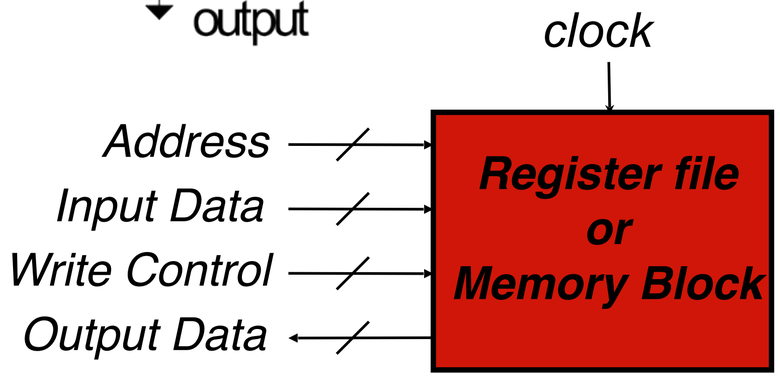
- Combination Logic allow us to implement any discrete valued function. But for complete computing systems (processors and the like), need memory elements.
- Memory elements (aka “state elements”) allow our circuit to “remember” - retain values from one time to the next.
- Examples:
  - RISC-V registers
  - Main memory
  - Other registers used by “micro-architecture” to control and synchronize movement of data through CL blocks



Only Two Types of Circuits Exist

- Combinational Logic Blocks (CL)
- State Elements (registers, memories)

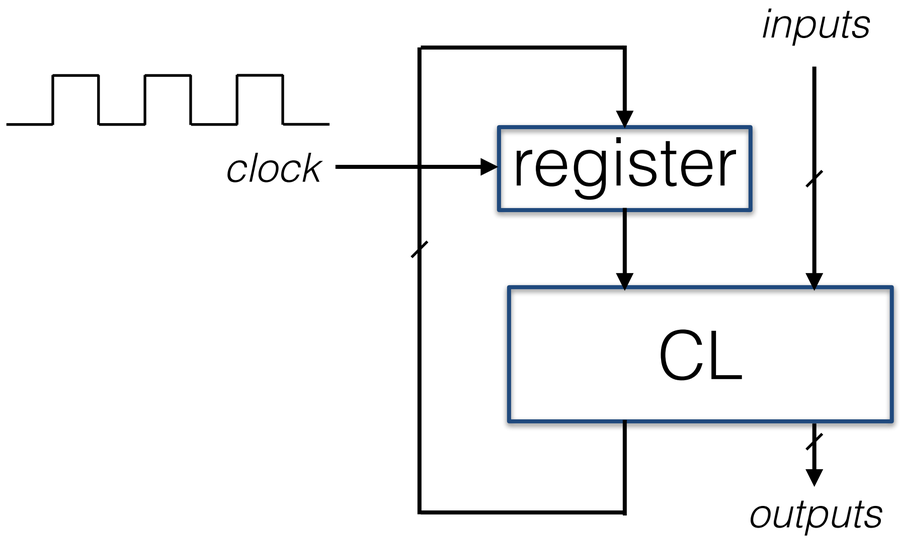




- State elements are mixed in with CL blocks to control the flow of data.
- Sometimes used in large groups by themselves for "long-term" data storage.

Adding registers to CL

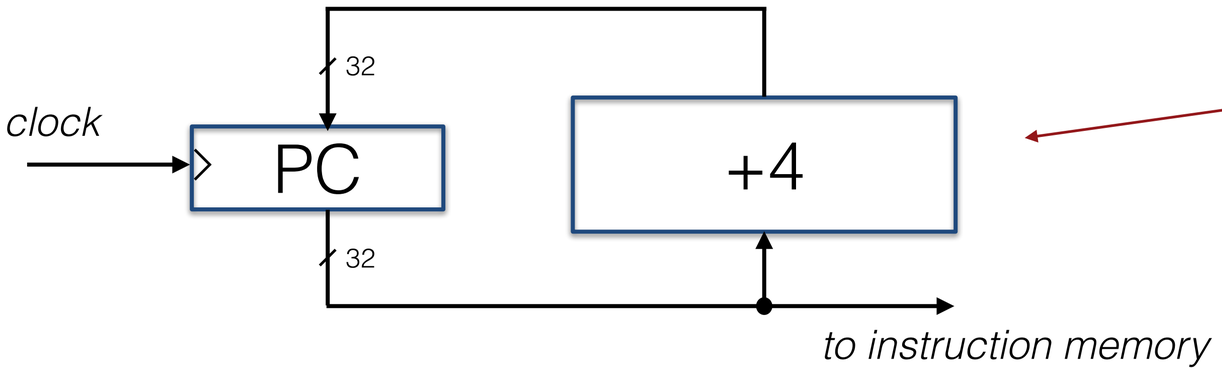
- Circuits that contain both CL blocks and state-elements cannot be abstracted by truth tables
- The output not just a function of the inputs - but also a function of the past history (value in the state element)
- Call “sequential circuits”
- Often modeled as Finite State Machine (FSM)



- Sequential circuits operate under the control of a clock signal
- On each clock cycle, based on current register value(s) and inputs, register value change, value can change

Example Sequential Circuit

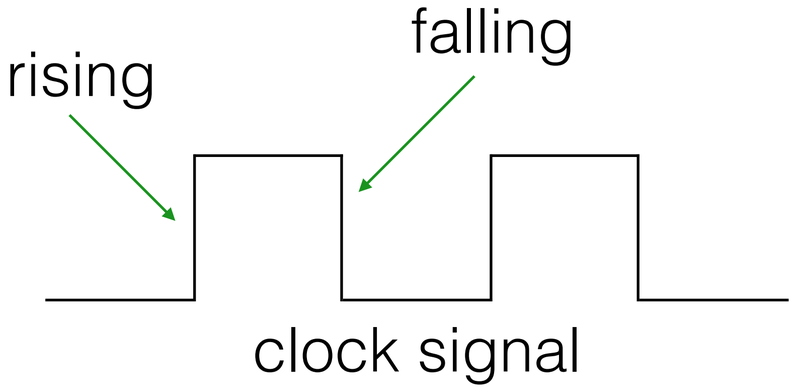
- RISC-V Program Counter (PC) - points to the current instruction being executed
- Except for branches/jumps, at the completion of each instruction:
  $PC \leftarrow PC + 4$
- Assume we design our RISC-V to execute one instruction per clock cycle, then on every cycle $PC \leftarrow PC + 4$



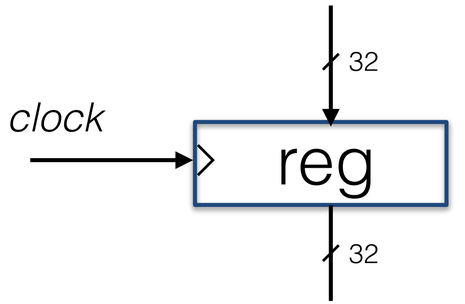
Could use an adder here, but probably design a simpler circuit

Register Details

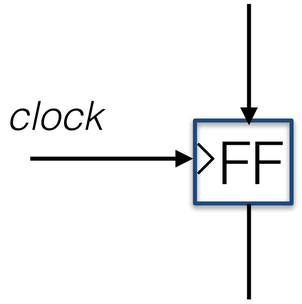
- The cMOS register circuits in common use are “edge-triggered”
  - They take their action based on the rising or falling edge of the clock. We assume rising edge for consistency.



- All state elements have clock signal connection (sometimes called “enable”)
- 1-bit register is called “flip-flop”
- N-bit wide register is a parallel collection of n flip-flops



32-bit register



1-bit register, aka flip-flop

Register Timing

- On rising-edge of clock (signal), FF captures input value, stores it internally, and transfers it to the output.