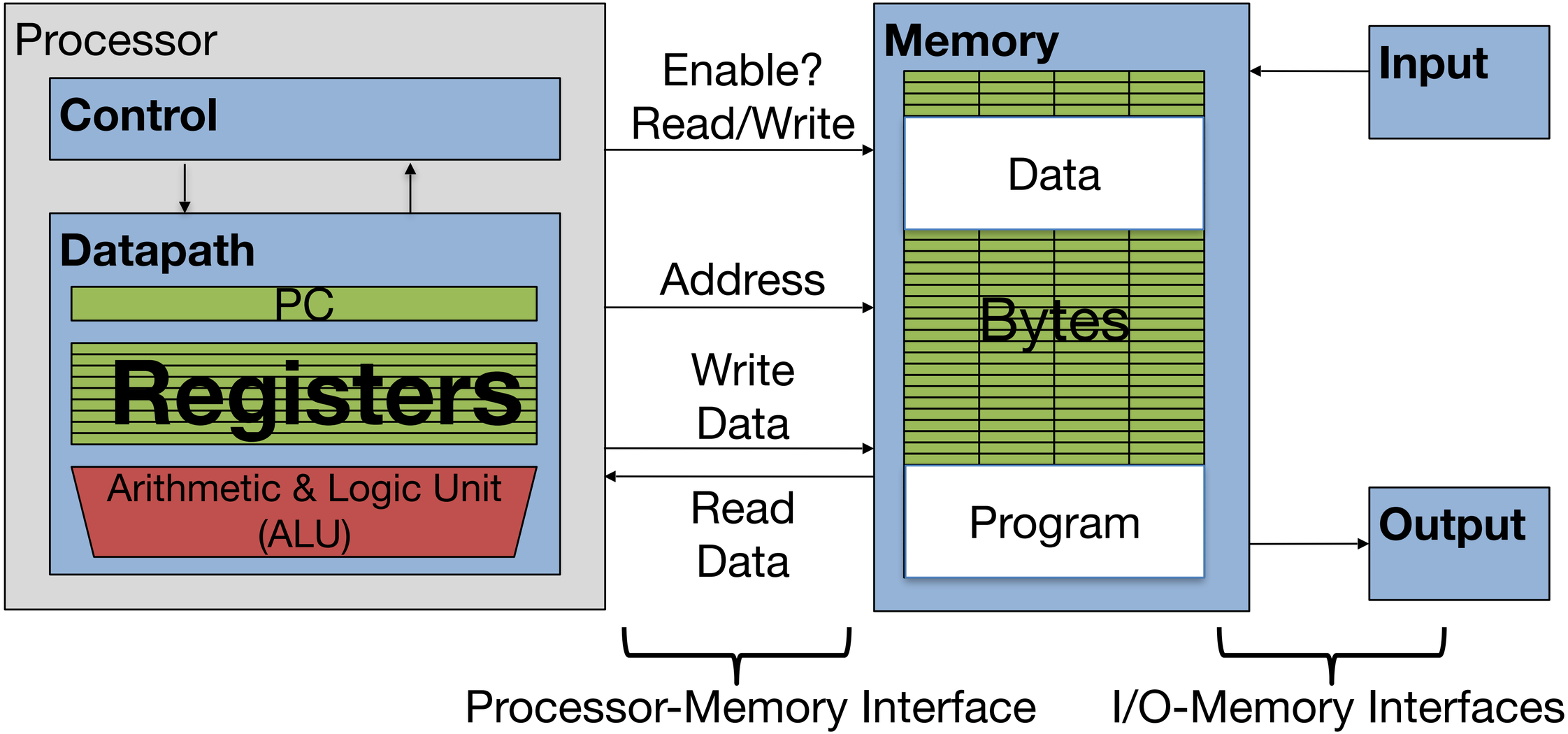
CS61C Fall 2021: Lecture 11

Intro To Digital Circuits
or
Synchronous Digital Systems
Part 2

Computer Hardware Overview



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)

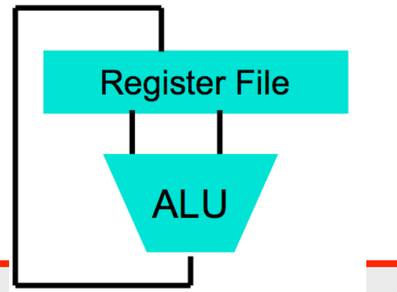
Architecture Implementation

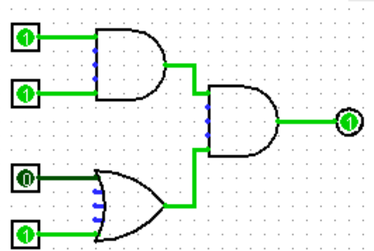
Logic Circuit Description (Circuit Schematic Diagrams)

```
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!

4-bit Combinational-Logic 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

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

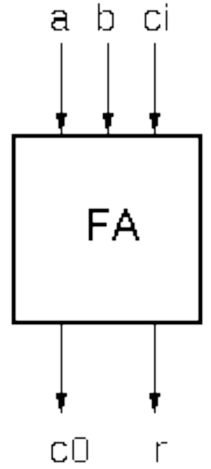
4-bit Adder Example

- In general:

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

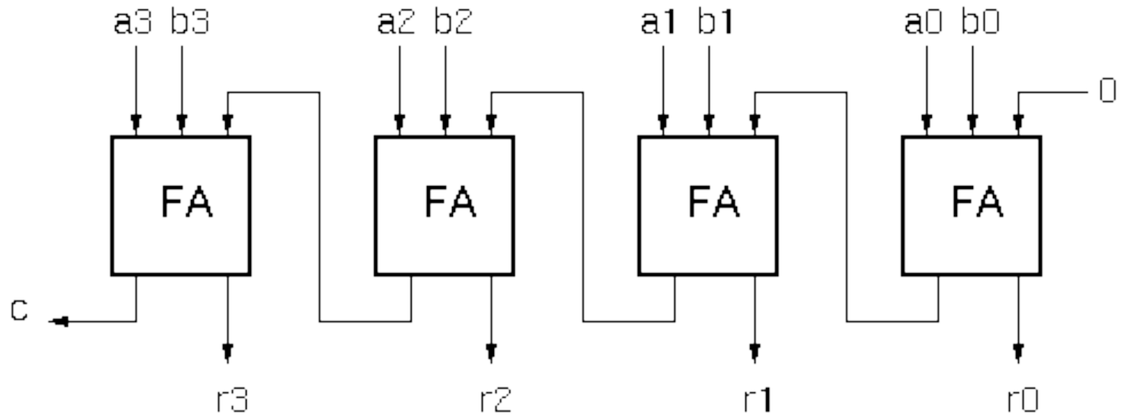
$c_{out} = a_i c_{in} + a_i b_i + b_i c_{in} = c_{in}(a_i + b_i) + a_i b_i$

"Full adder cell"



- Now, the 4-bit adder:

"ripple" adder



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

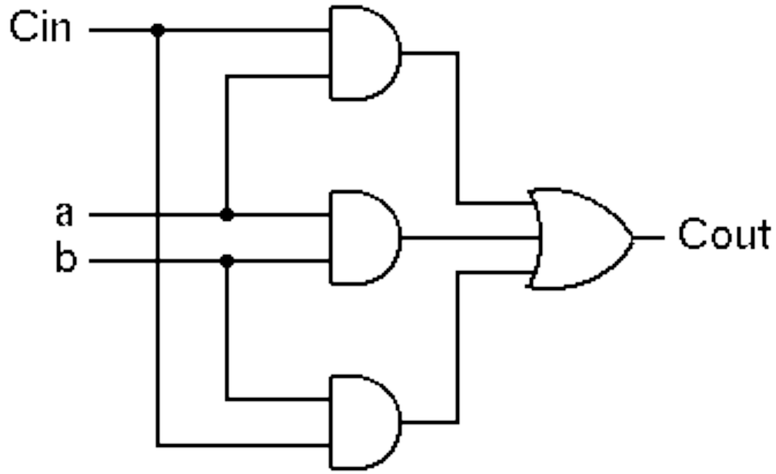
Note: the same circuit works for both unsigned and signed (2's complement)

4-bit Adder Example

- Graphical Representation of FA-cell

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

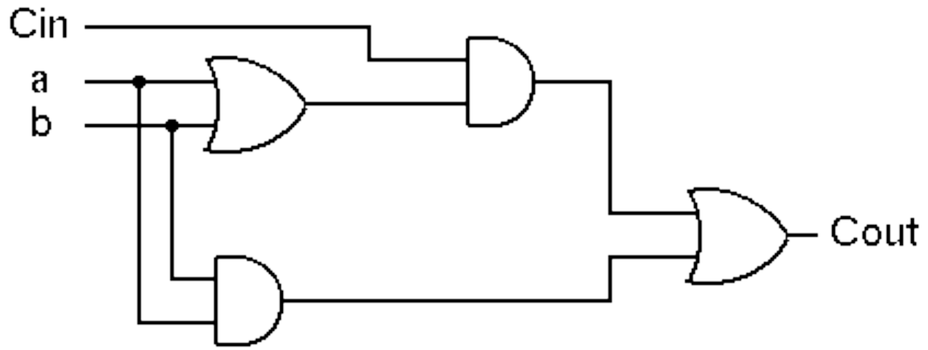
$c_{out} = a_i c_{in} + a_i b_i + b_i c_{in}$

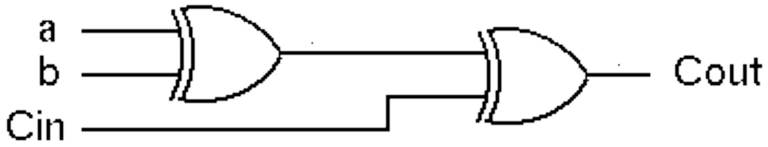


- Alternative Implementation (with only 2-input gates):

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

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





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

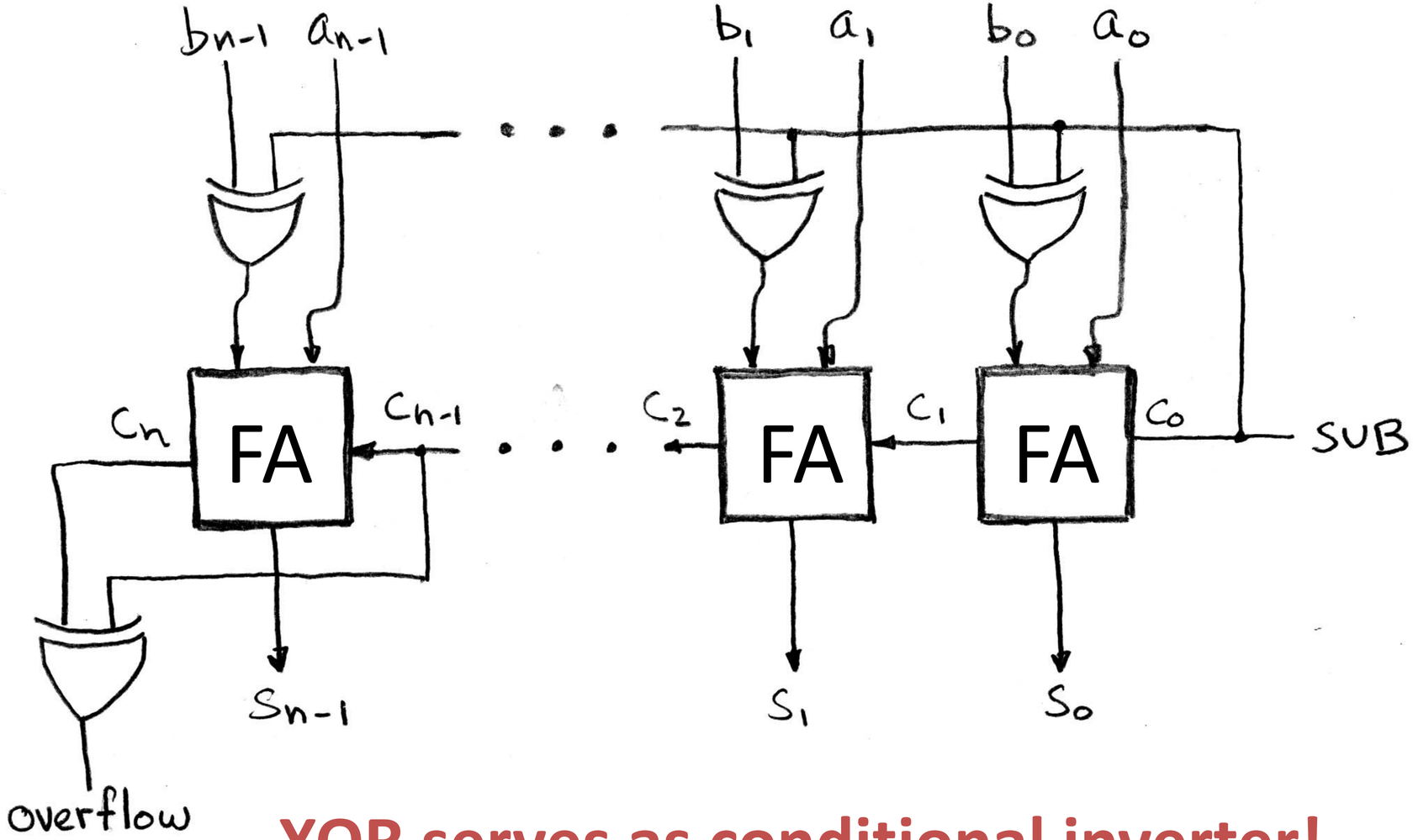
ci	a	b	r	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

aka the "majority function"

Adder/Subtractor: A - B = A + (-B)

To negate B (in 2's complement rep), invert bits and add 1

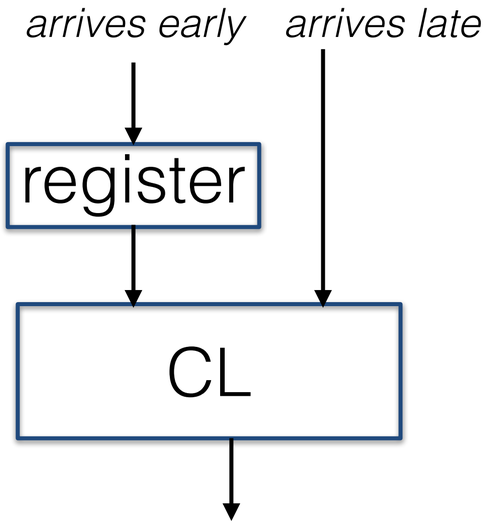
x	y	XOR(x,y)
0	0	0
0	1	1
1	0	1
1	1	0



XOR serves as conditional inverter!

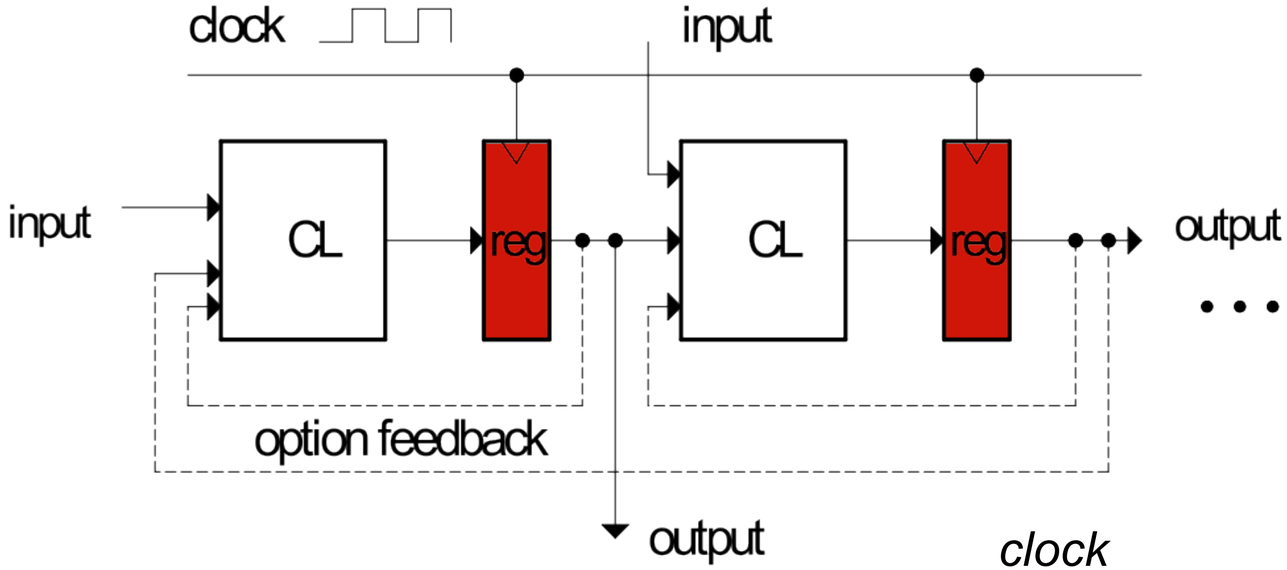
State Elements

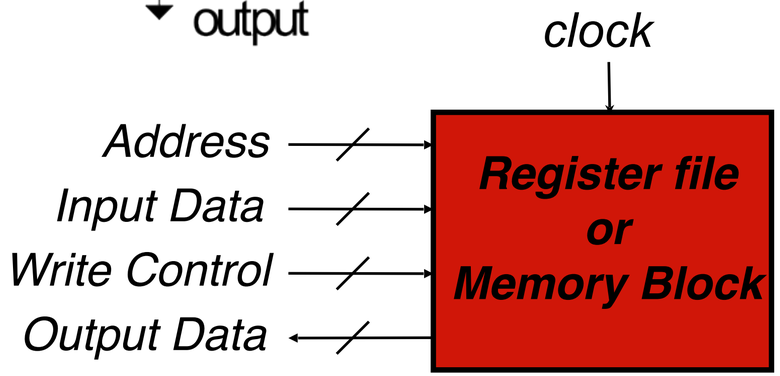
- Combination Logic allow us to implement any discrete valued function. But for complete computing systems (processors and the like), need memory elements.
- Memory elements (aka “state elements”) allow our circuit to “remember” - retain values from one time to the next.
- Examples:
  - RISC-V registers
  - 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)

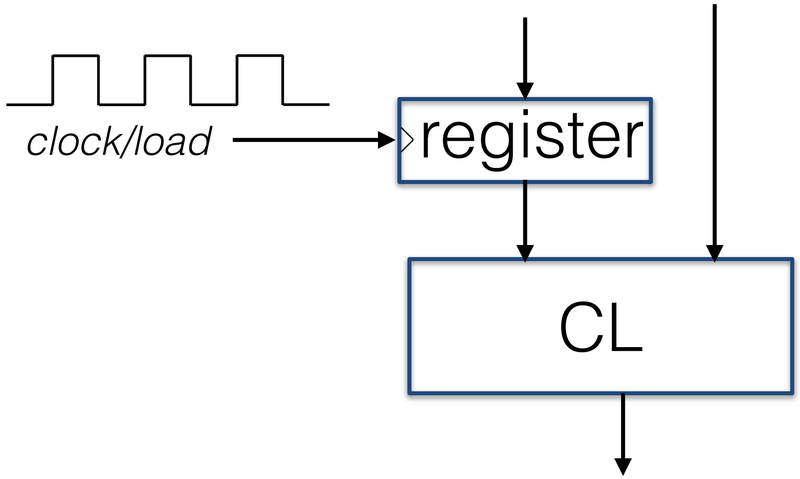




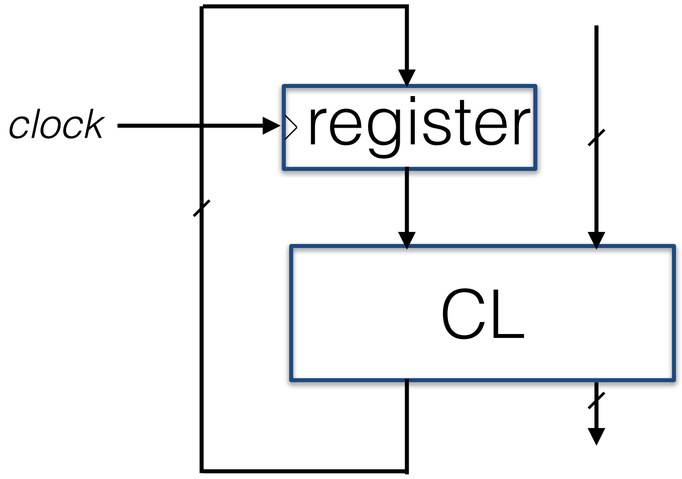
- State elements are mixed in with CL blocks to control the flow of data.
- Sometimes used in large groups by themselves for "long-term" data storage.

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**"

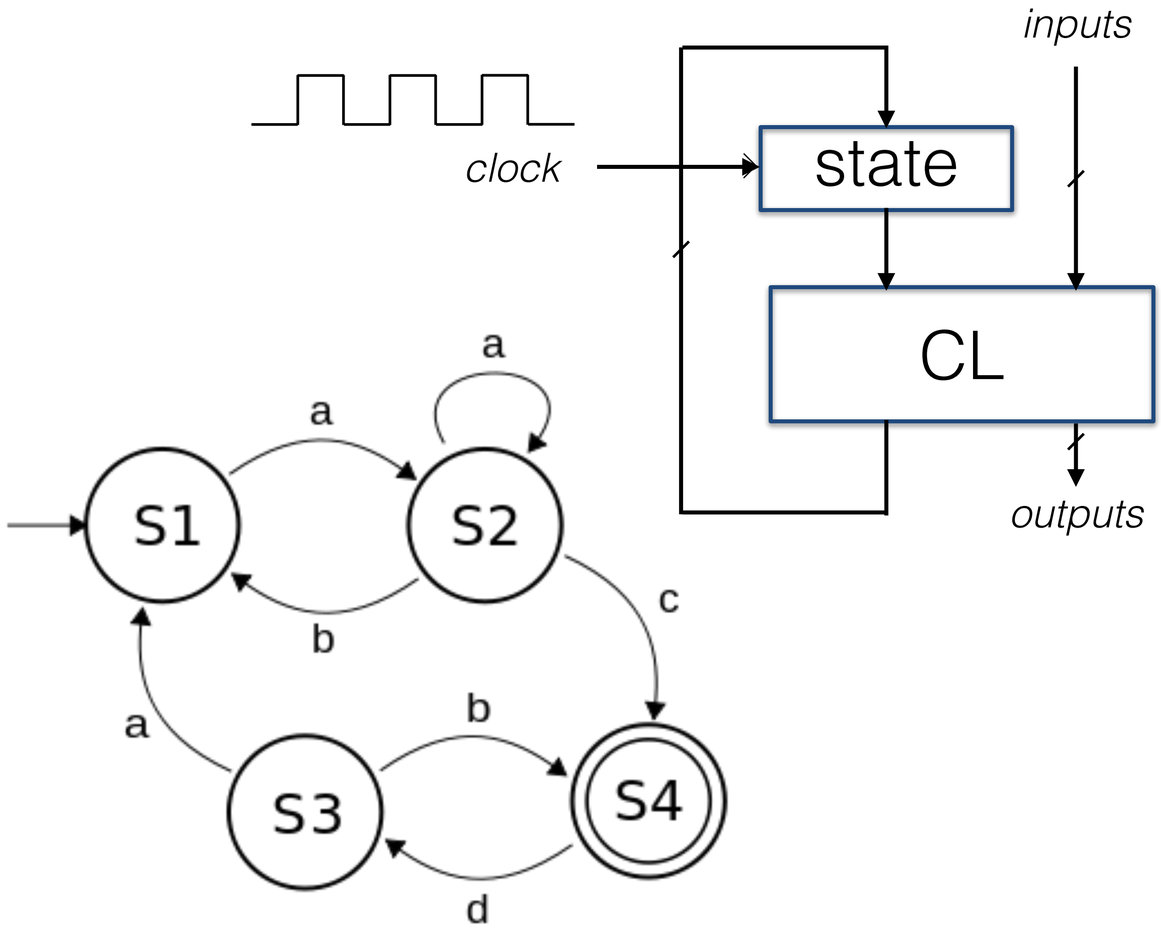


- Sequential circuits usually operate under the control of a clock signal
- On each clock cycle, based on current register value(s) and inputs, register value change, value can change



Finite State Machines

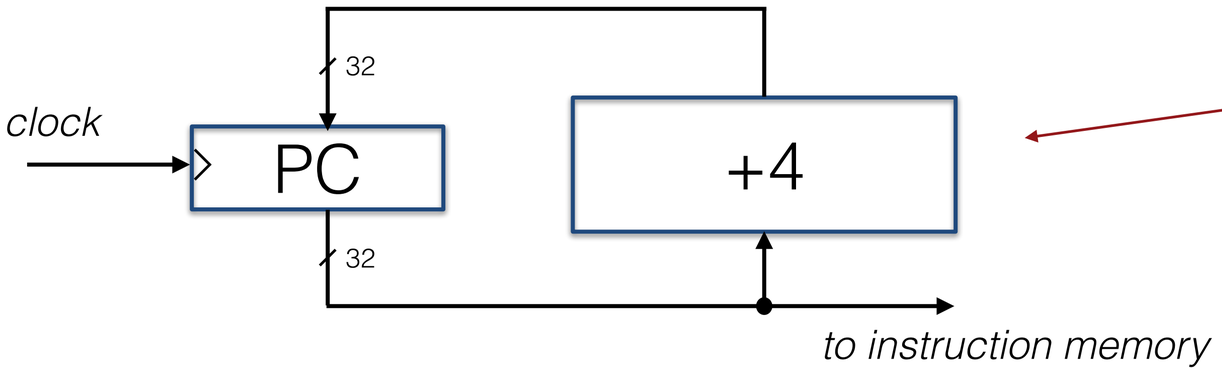
- Sequential circuits with feedback are often modeled as finite state machines (FSMs):
  - "state" of the machine stored in the register
  - On each cycle transition to a new "state" based on input and current state
  - generate output based on input and current state



State Transition Diagram defines behavior of FSM

Example Sequential Circuit

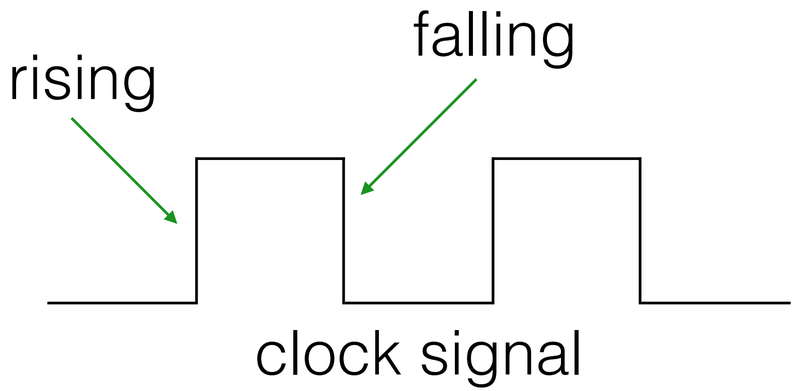
- RISC-V Program Counter (PC) - points to the current instruction being executed
- Except for branches/jumps, at the completion of each instruction:
  $PC \leftarrow PC + 4$
- Assume we design our RISC-V to execute one instruction per clock cycle, then on every cycle $PC \leftarrow PC + 4$



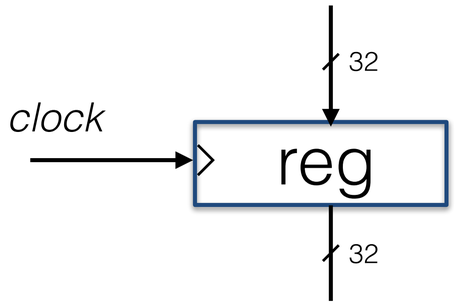
CL incrementer. Could use an adder here, but probably design a simpler circuit

Register Details

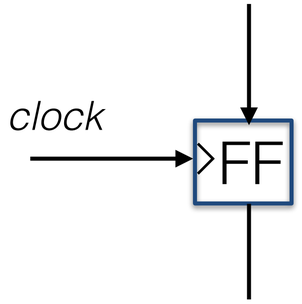
- The CMOS register circuits in common use are “edge-triggered”
  - They take their action based on the rising or falling edge of the clock. We assume rising edge for consistency.



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



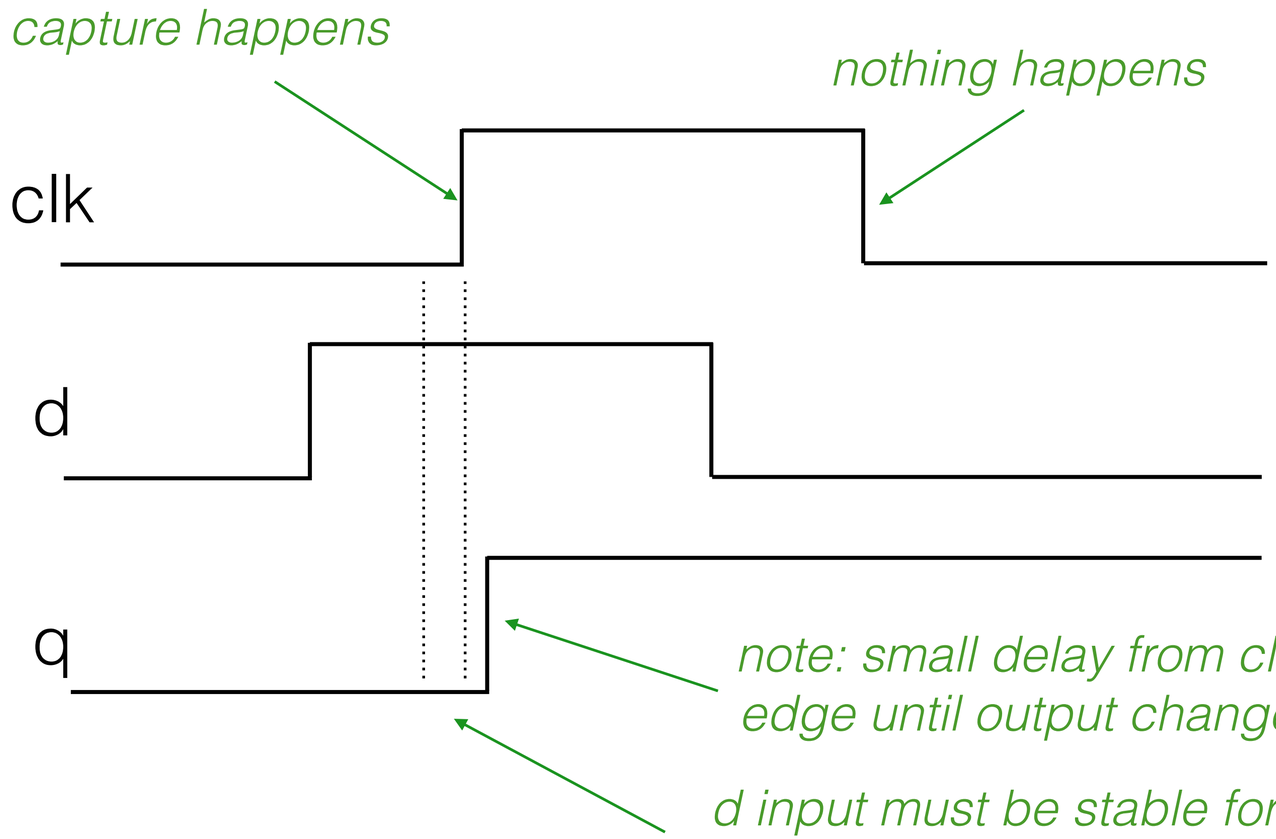
32-bit register

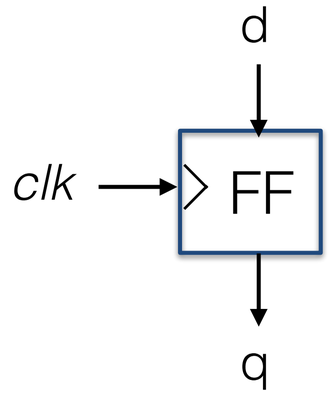


1-bit register, aka flip-flop

Register Timing

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

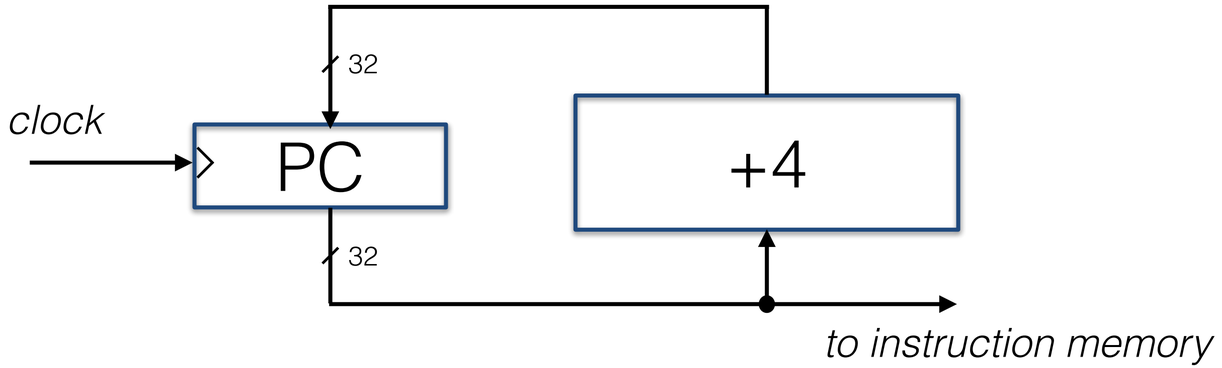




$\tau_{clk-to-q}$

τ_{setup}

Back to the PC register



- Need to modify to accommodate branches and jumps
- Need some sort of circuit "if/else".

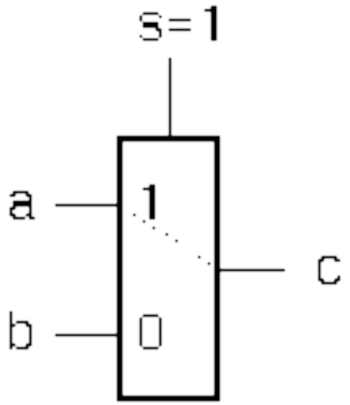
If no branch or branch not taken $PC \leftarrow PC + 4$

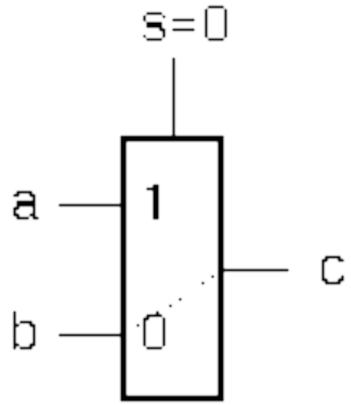
else $PC \leftarrow$ target address

What gets loaded into the PC register at the end of the cycle should depend on the result of branch compare.

Introducing the “multiplexor”, aka “mux”

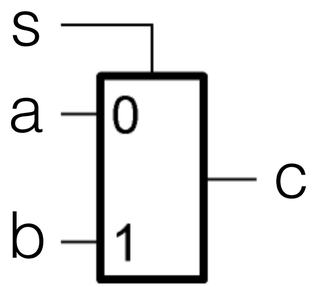
- CL circuit that chooses between multiple inputs and sends to output.
- Example: 2-to-1 mux
  - if s=1 then c=a else c=b





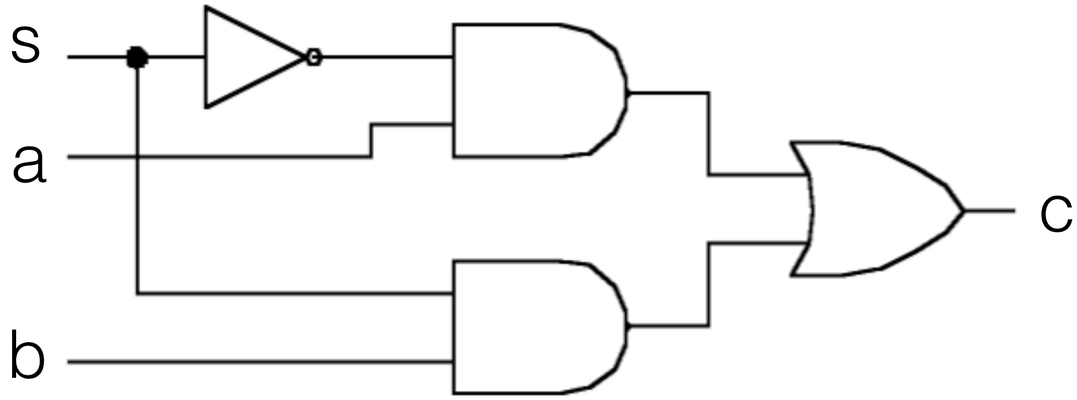
- Muxes come in any width, i.e., n-bit wide means a, b, and c are n-bits
- and any number of inputs, i.e., 4-to-1 mux.
  - Sometimes primitive design elements (built at transistor level)
  - Let’s assume not. Because it is a CL function can build with AND/OR/NOT circuits.

Multiplexor Circuit Implementation



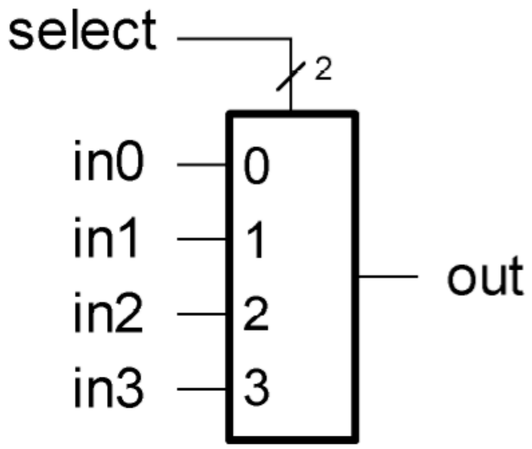
s	a	b	c
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

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

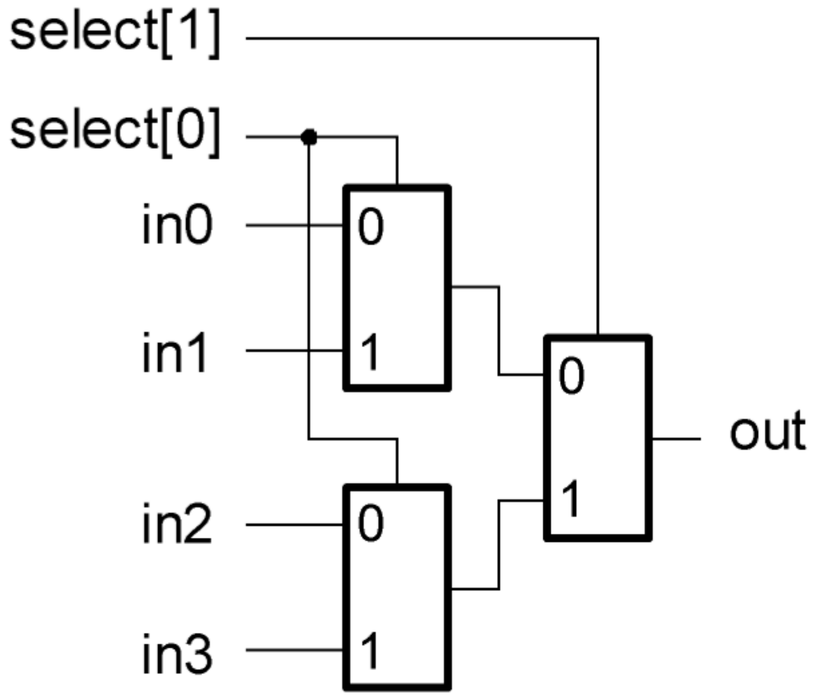


Bigger Muxes

- Example 4-to-1 multiplexor:

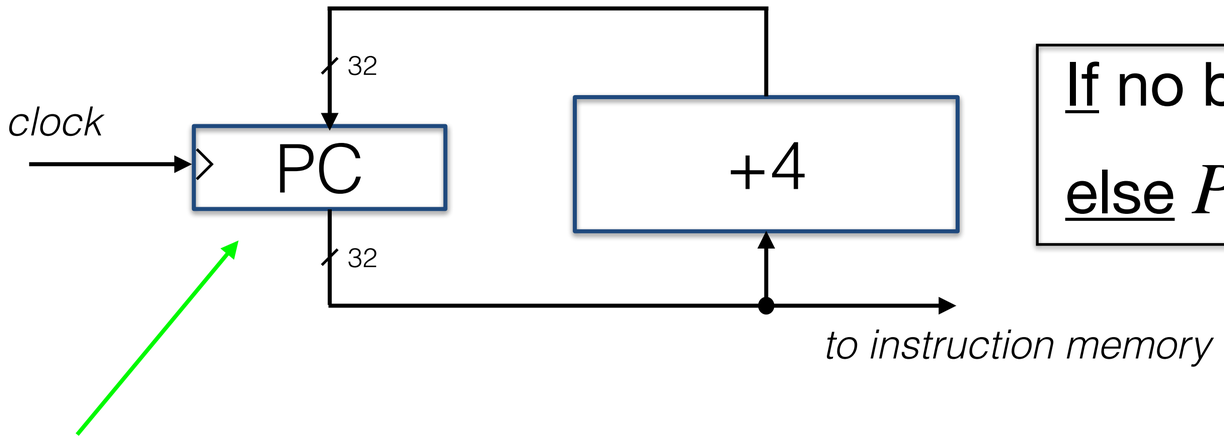


a) 4-input mux symbol



b) 4-input mux implemented with 2-input muxes

Back, again, to the PC register

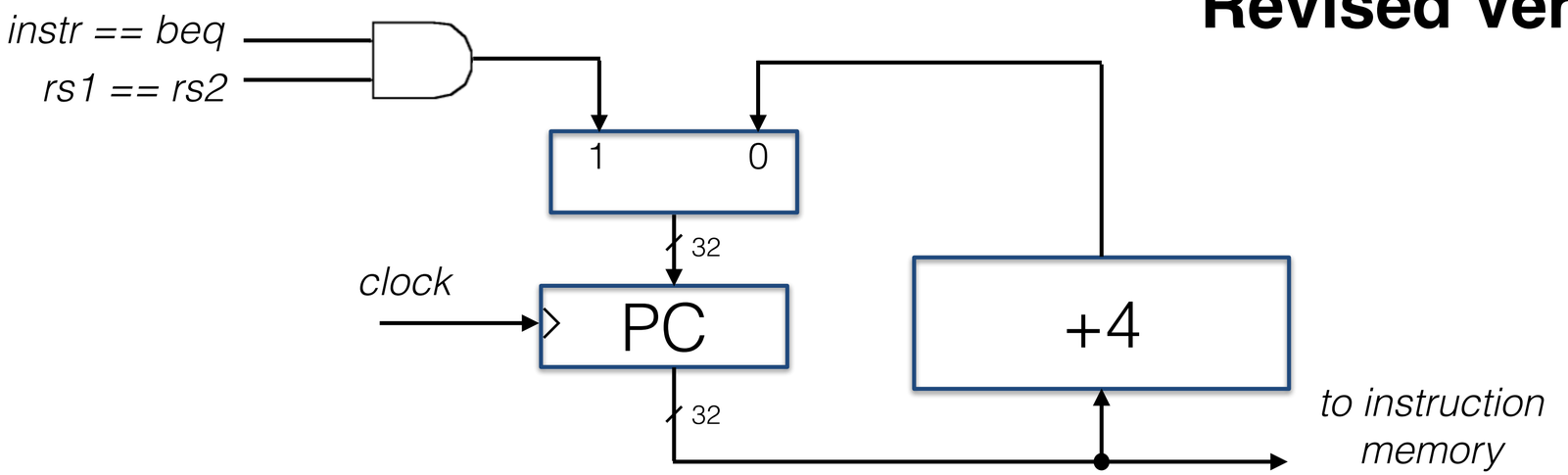


If no branch or branch not taken $PC \leftarrow PC + 4$

else $PC \leftarrow$ target address

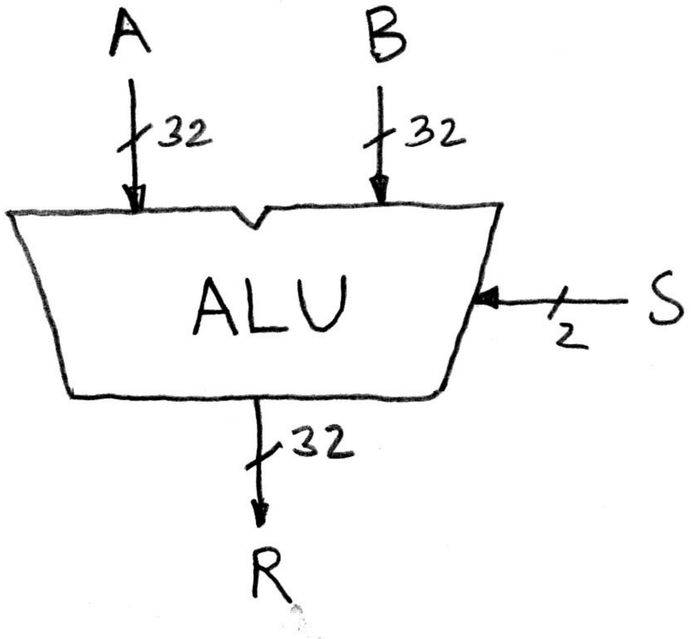
New PC value should depend on if beq and register compare

Revised Version



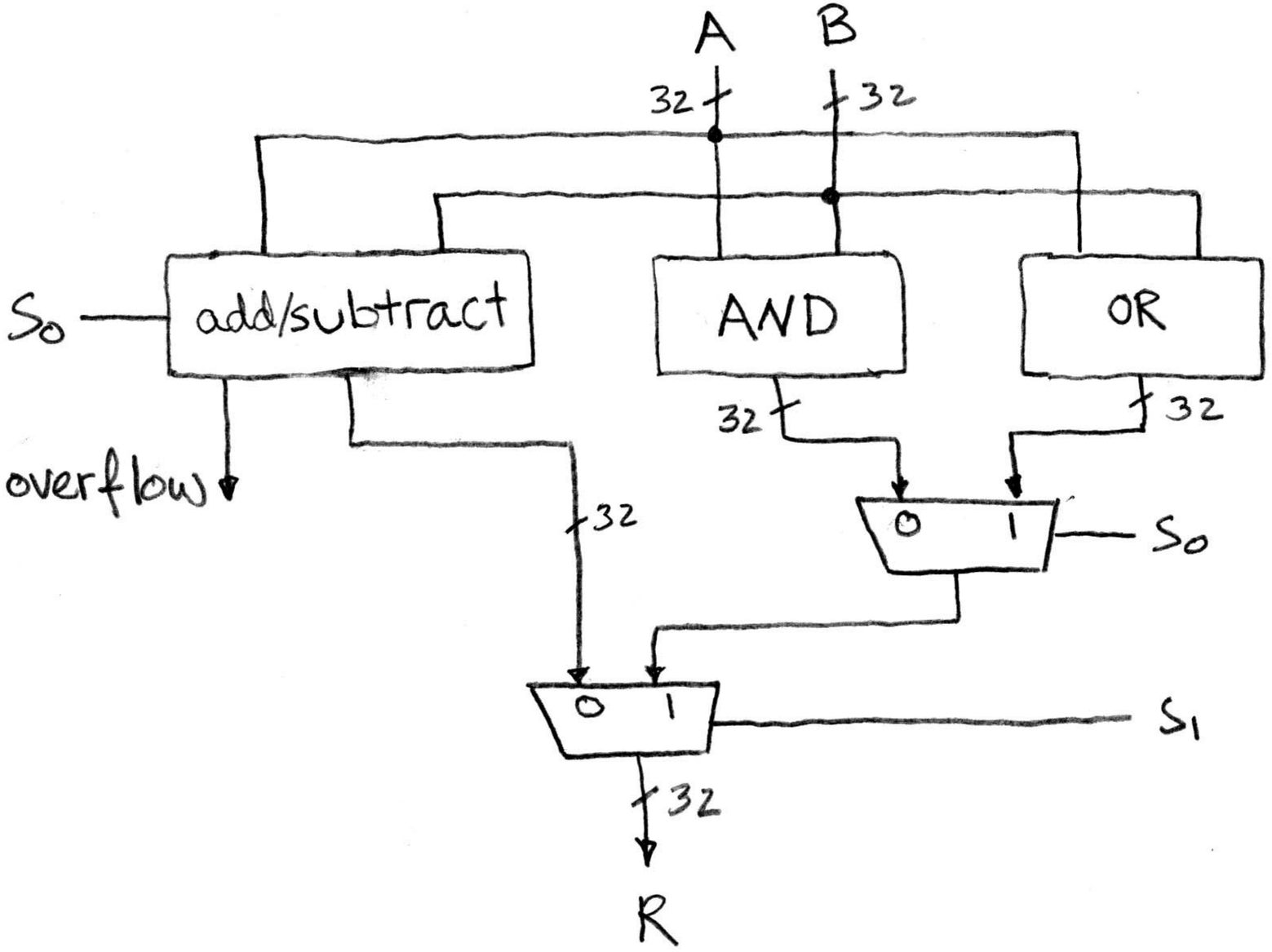
Arithmetic and Logic Unit

- Most processors contain a special logic block called the “Arithmetic and Logic Unit” (ALU)
- We show a simple one that does ADD, SUB, bitwise AND, bitwise OR



when S=00, R=A+B
when S=01, R=A-B
when S=10, R=A AND B
when S=11, R=A OR B

Our simple ALU

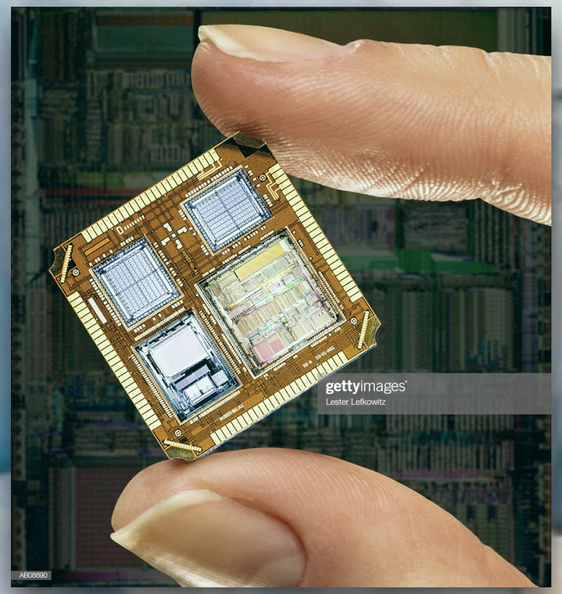


when S=00, R=A+B
when S=01, R=A-B
when S=10, R=A AND B
when S=11, R=A OR B

Note: all options computed all the time and in parallel. Appropriate result is sent to output.

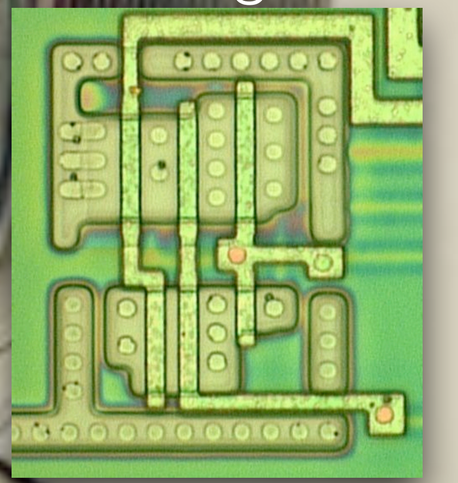
Beneath the Logic Gate Abstraction

CMOS: Complementary metal–oxide–semiconductor



die in package

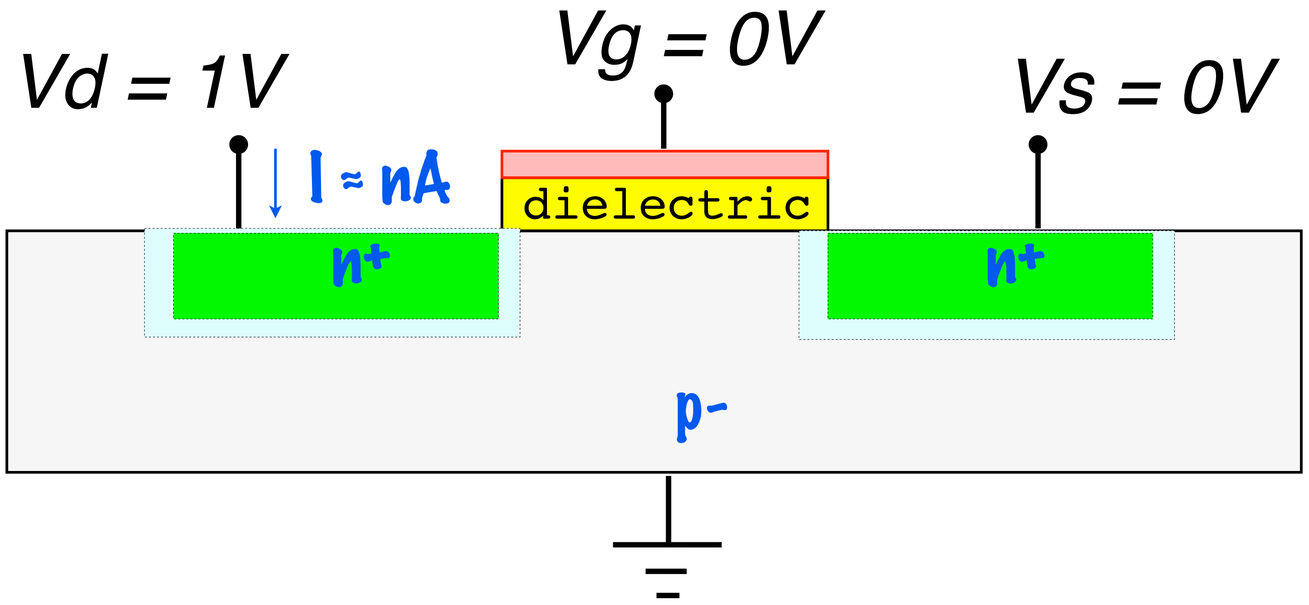
AND gate



- On crystalline silicon wafers, transistors and wires are fabricated, then cut into chips (die) and packaged.
- Two types of (complementary) transistors possible

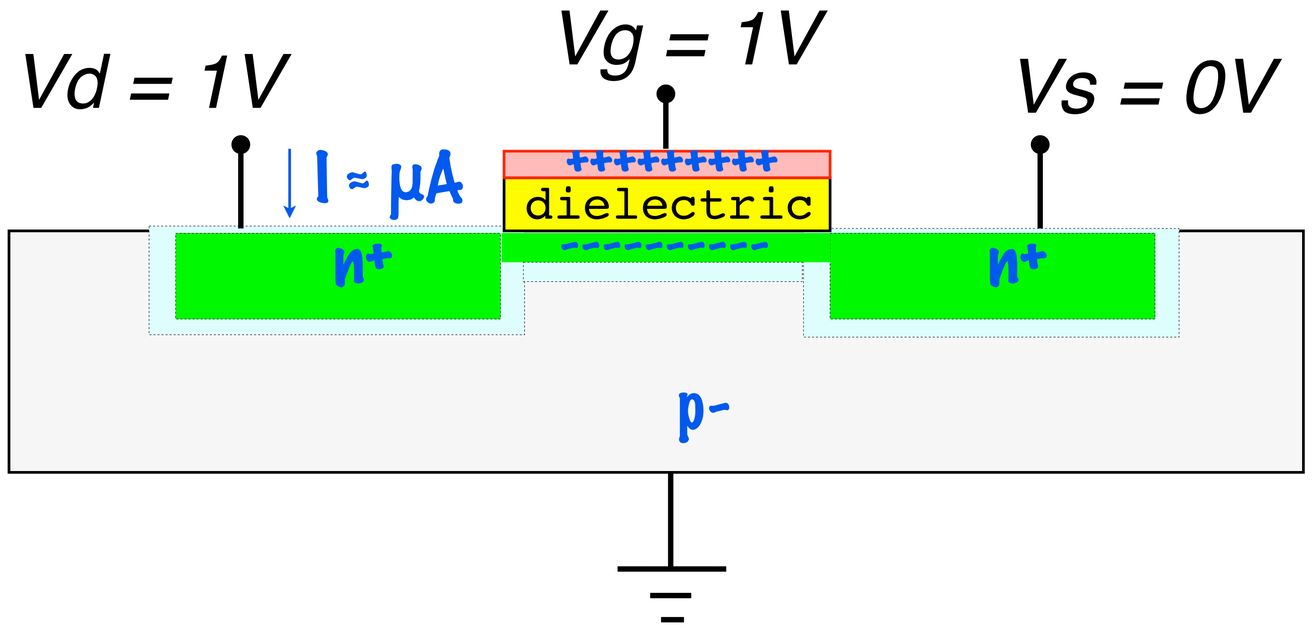
Cross-section of n-type MOS transistor: nFET (in planar process)

- Three electrical terminals: gate, source, drain
- *Vg, Vd, Vs* indicates voltages on each node



Polysilicon gate, dielectric, and substrate form a capacitor.

nFet is off
(I is “leakage”)

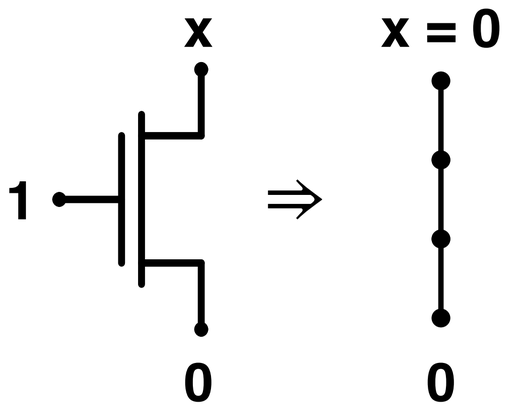


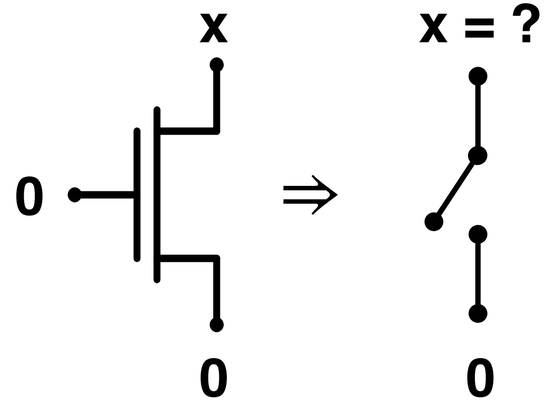
Vg = 1V, small region near the surface turns from p-type to n-type.

nFet is on.

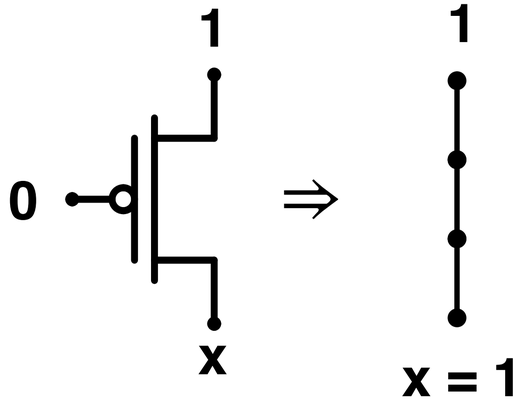
Switches made in CMOS process

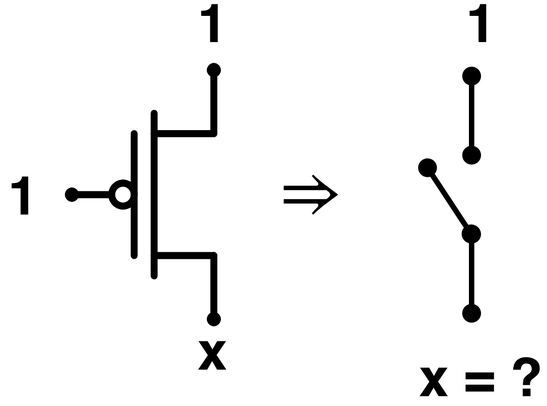
- nFET (n-type Field Effect Transistor)





- pFET

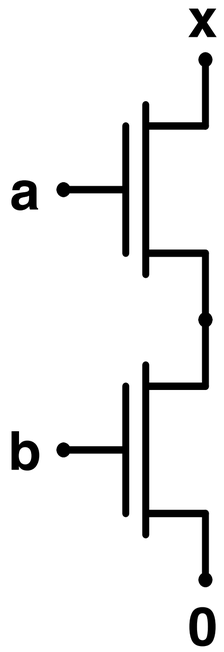




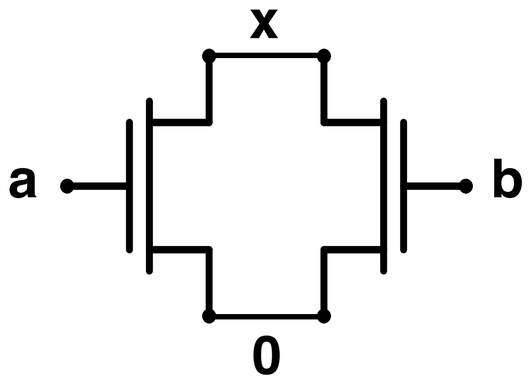
nFET used for passing 0, pFET used for passing 1

Multiple switches implement logic operations

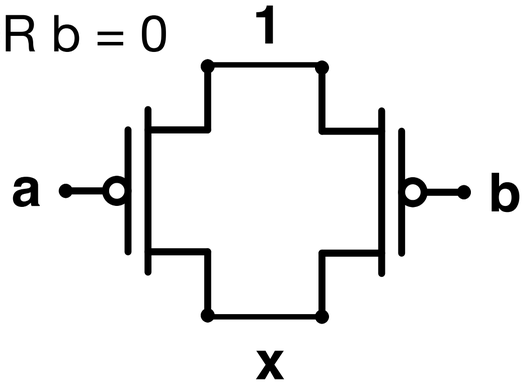
- nFET series connection:
  - x=0 iff a AND b = 1



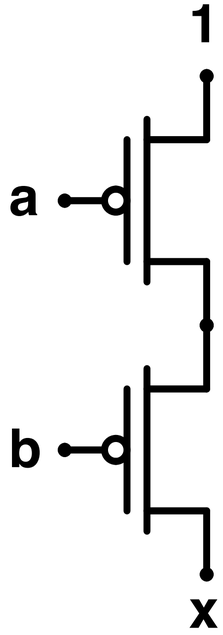
- nFET parallel connection:
  - x=0 if a OR b = 1



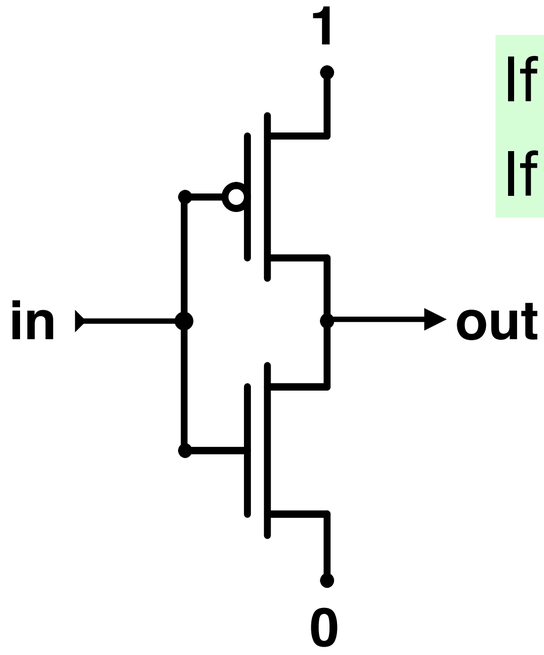
- pFET parallel connection:
  - x=1 if a OR b = 0



- pFET series connection:
  - x=1 iff a AND b = 0



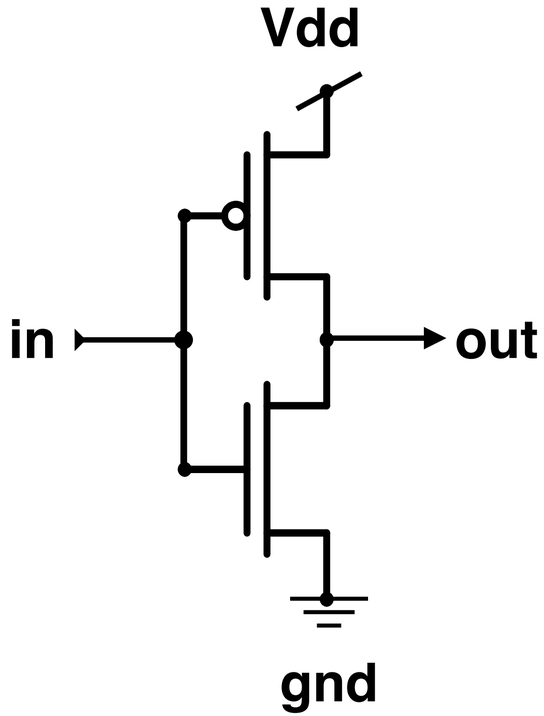
CMOS inverter



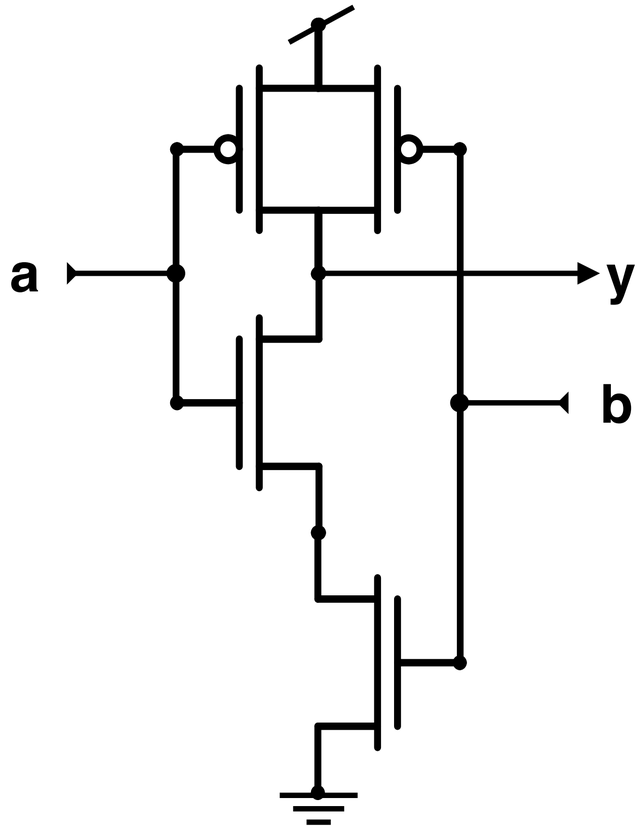
If in=0 out=1

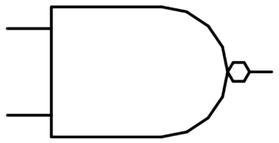
If in=1 out=0

- *High voltage is source of logic 1's, Vdd*
- *Low voltage is source of logic 0's, gnd*



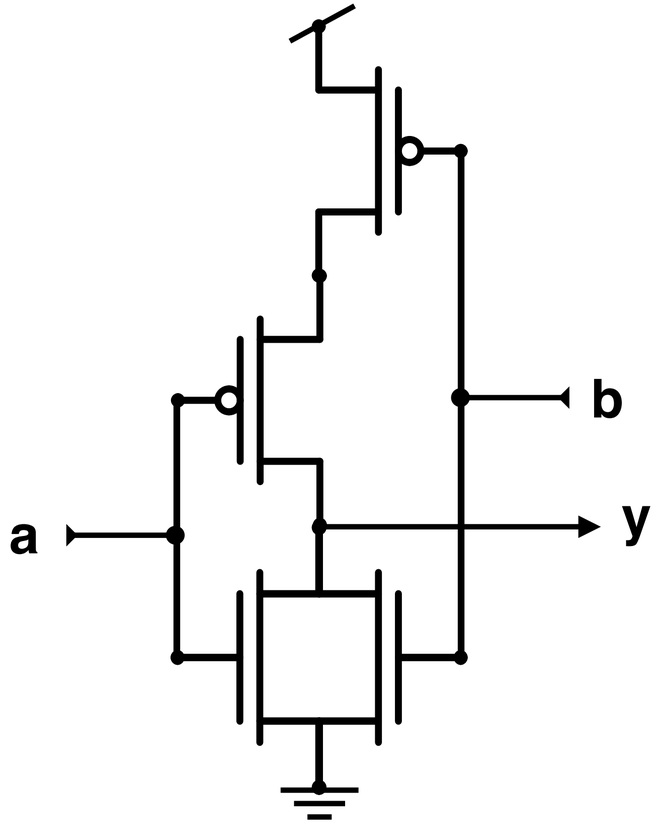
CMOS logic gates

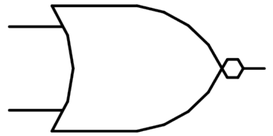




NAND

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

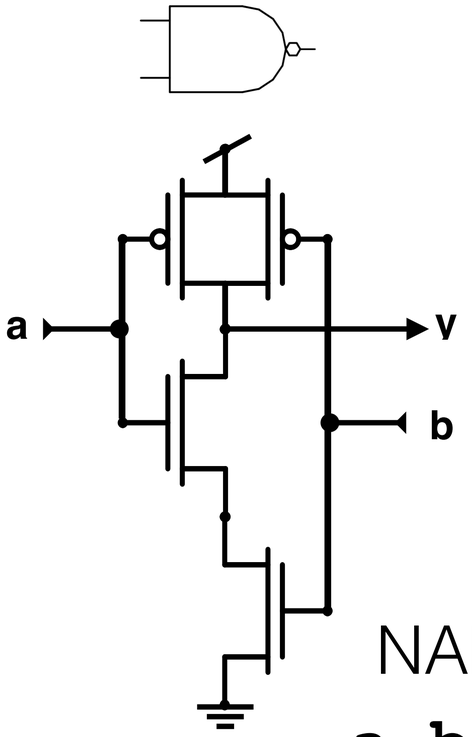




NOR

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

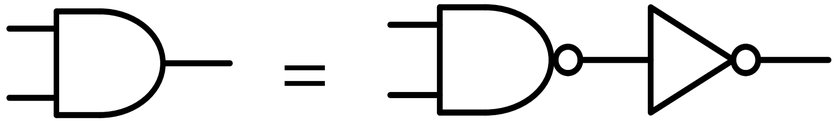
CMOS logic gates



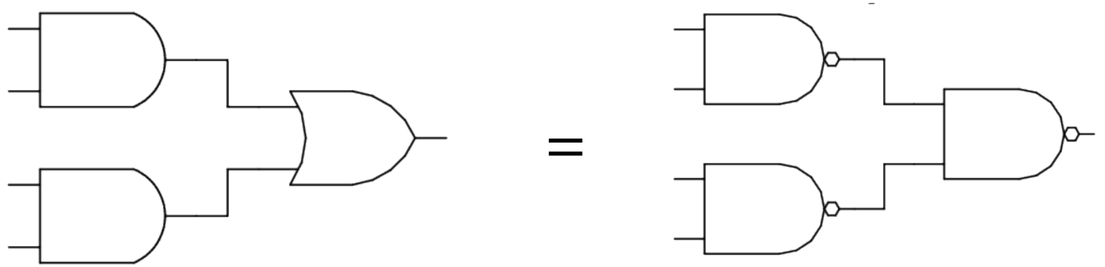
NAND

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

- *CMOS gates are always inverting!*
- *If we really want an AND, could use NAND plus inverter*



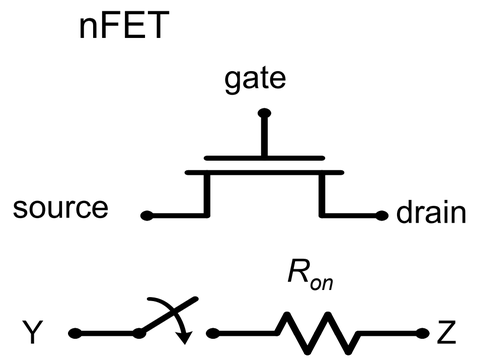
- *However, AND/OR/NOT circuits can be converted to NAND/NOR circuits using Boolean Algebra (primarily DeMorgan's law), with approximately the same number of gates.*
- *Ex: AND/OR = NAND/NAND*



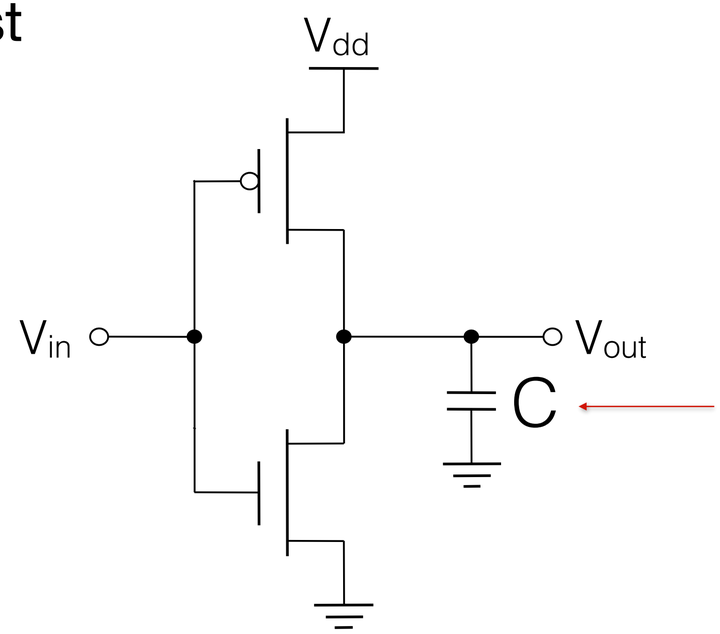
Nasty Realities: Delays in CMOS circuits

More physically realistic model:

1. Transistors are not perfect switches
   A. They leak when off
   B. They have finite resistance when on
2. All circuit nodes have capacitance
   - To change their voltage level must displace charge



When on, resistance between Y and Z.
Likewise for pFET

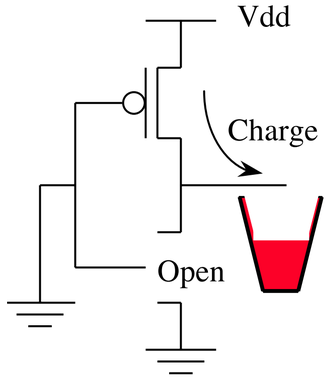


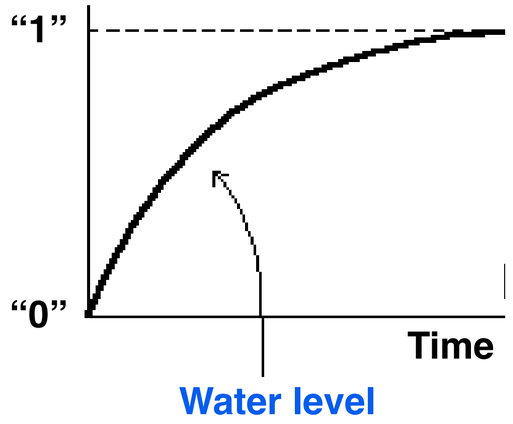
Represents the sum of all the capacitance at the output of the inverter and everything to which it connects: (drains, wires, transistor-gate capacitance of next gate(s))

Transistors as water valves

If electrons are water molecules, transistor resistance like pipe diameters, and capacitors are buckets ...

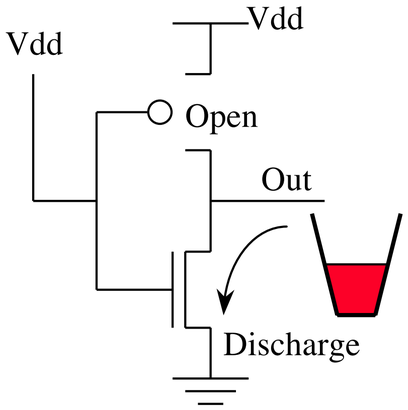
A "on" p-FET fills up the capacitor with charge.

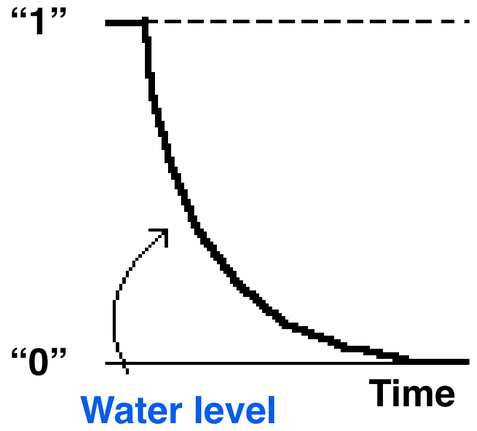




$$\tau \propto R \cdot C$$

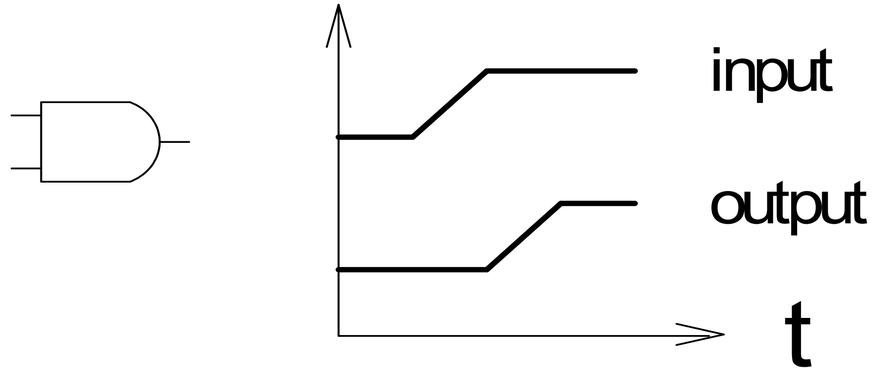
A "on" n-FET empties the bucket.

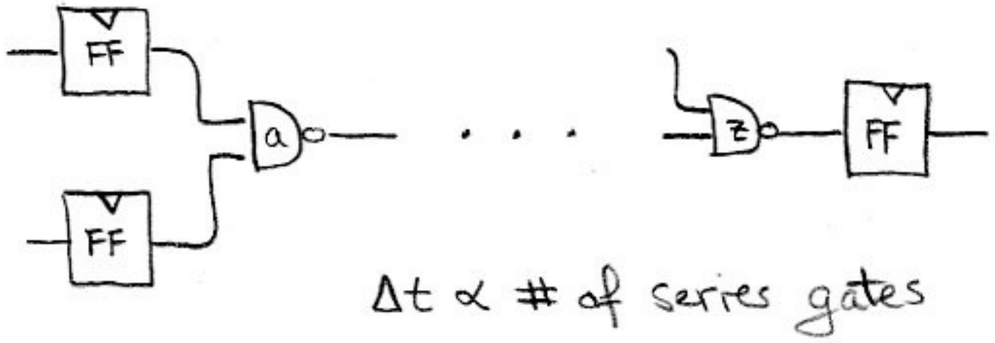




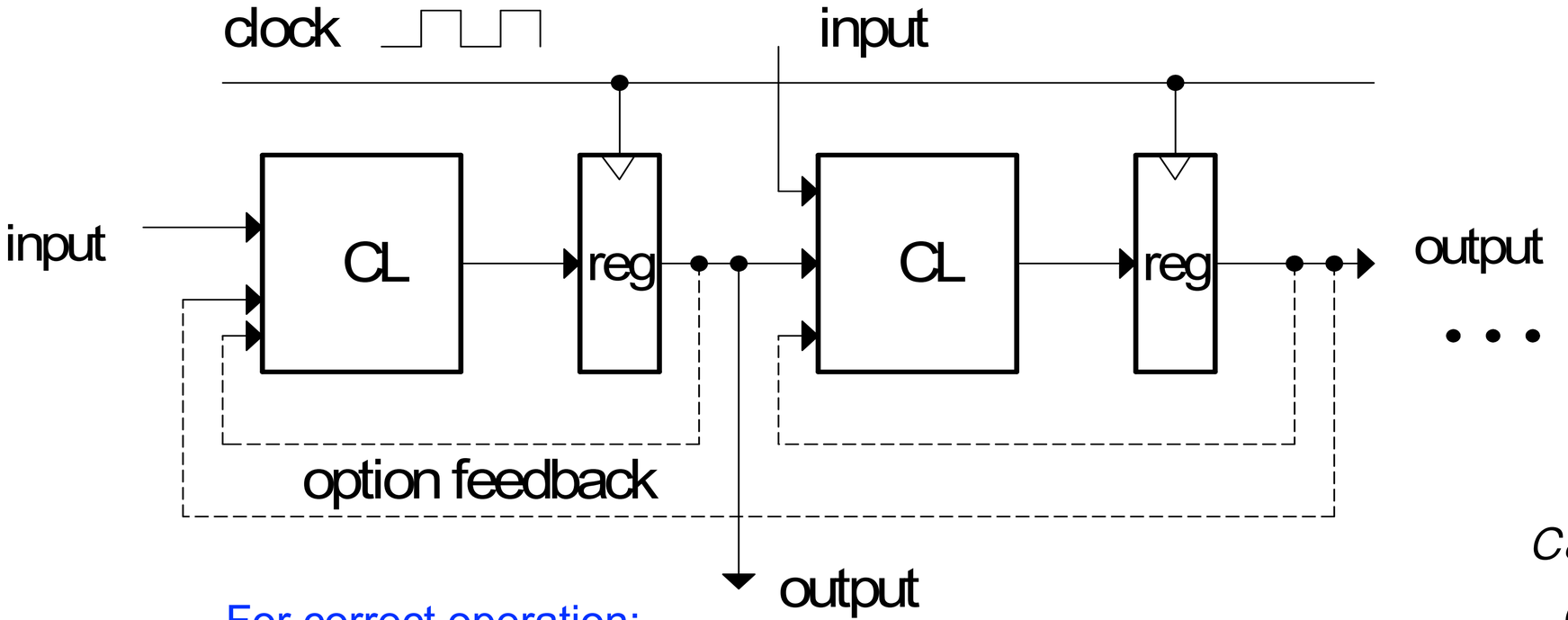
Consequences

- For every logic gate, delay from input change to output change
- The exact amount of the delay depends on:
  - type of gate, how many other gates it's output connects to, IC process details
- For cascaded gates, delay accumulates
- Remember, flip-flops also have details and timing constraints: $\tau_{clk-to-q}$ and τ_{setup}





Therefore, in General ...



For correct operation:

$$T \geq \tau_{clk \rightarrow Q} + \tau_{CL} + \tau_{setup}$$ for all paths.

The worst case path is called the "critical path"

What can we do to reduce T (increase frequency)?

More nasty realities: CMOS circuits use electrical energy (consume power)

Energy is the ability to do work (joules).

Power is rate of expending energy (watts).

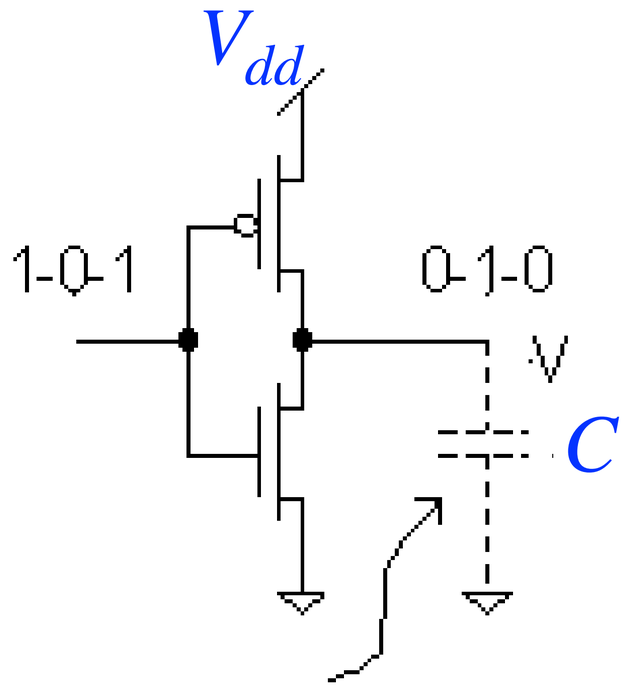
Energy Efficiency: energy per operation

$$P = \frac{dE}{dt}$$

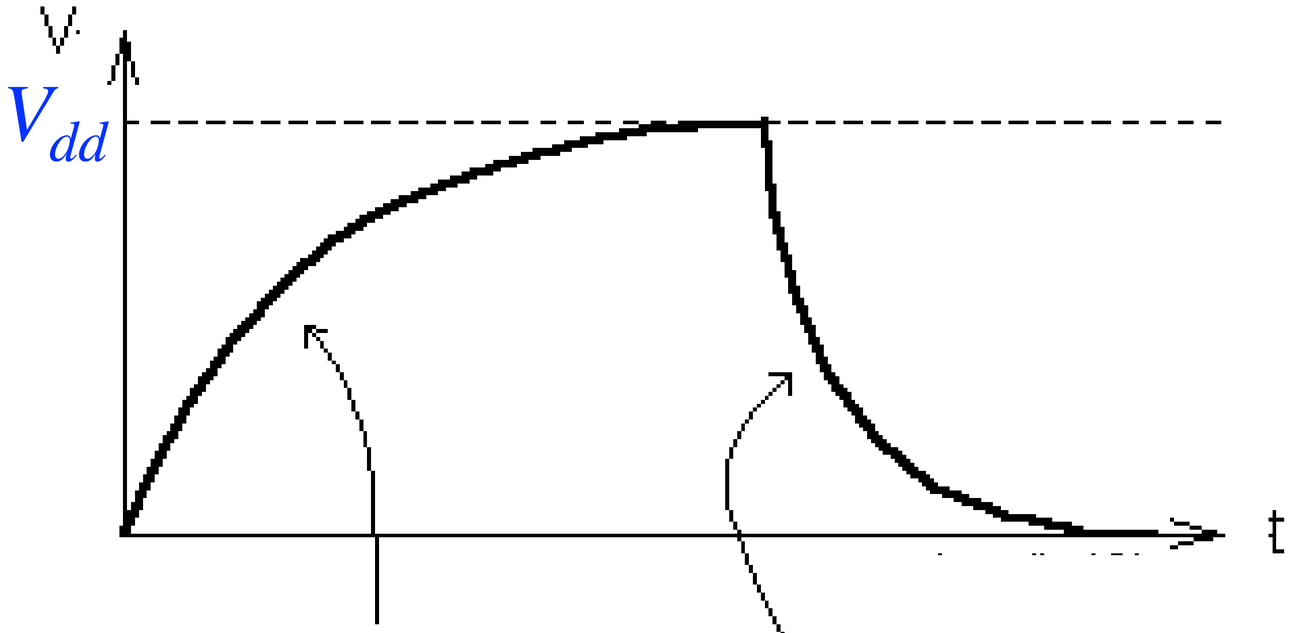
- ***Handheld and portable*** (battery operated):
  - Energy Efficiency - limits battery life
  - Power - limited by heat
- ***Infrastructure and servers*** (connected to power grid):
  - Energy Efficiency - dictates operation cost
  - Power - heat removal contributes to TCO

Switching Energy: Fundamental Physics

Every logic transition dissipates energy.



Models inputs to other gates & wire capacitance



$$E_{0\to1} = \frac{1}{2} \cdot C \cdot V_{dd}^{\,2} \qquad E_{1\to0} = \frac{1}{2} \cdot C \cdot V_{dd}^{\,2}$$

Chip-Level “switching” Power

$$P = dE / dt$$

$$P_{sw} = 1/2 \; \alpha \; C \; V_{dd}^{2} \; F$$

“activity factor”, average percentage of capacitance switching per cycle (~ number of nodes to switch)

Total chip capacitance to be switched

Clock Frequency

Reducing power consumption or improving energy efficiency

$$P_{sw} = 1/2 \; \alpha \; C \; V_{dd}^{\;2} \; F$$

- Power proportional to F. Can reduce power by reducing frequency. But that doesn't improve energy efficiency (just spreads computation over longer time)
- Energy efficiency:
  - $E_{SW} \propto V_{dd}^{\;2}$ but $\tau_{logic} \propto V_{dd}$
  - Therefore can improve energy efficiency by lowering supply voltage and making up for less performance by using parallelism
  - Main driver of the move towards multi-core processors (Ex: Apple M1 had 8 cores)