CS3350B Computer Organization
Chapter 2: Synchronous Circuits
Prelude

Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Thursday January 24, 2019

Outline

Radix Representations

Radix is the base number in some numbering system.
In a radix r representation digits (d_i) are from the set $\{0, 1, \ldots, r-1\}$

$$x = d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + \cdots + d_1 \times r^1 + d_0 \times r^0$$

- $r = 10 \implies$ decimal, $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $r = 2 \implies$ binary, $\{0, 1\}$
- $r = 8 \implies$ octal, $\{0, 1, 2, 3, 4, 5, 6, 7\}$
- $r = 16 \implies$ hexadecimal, $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$

Refresh: Decimal to Binary

$$(13)_{10} = (1 \times 10^1) + (3 \times 10^0)$$

$$(1101)_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = (13)_{10}$$

Unsigned Binary Integers

Unsigned Integers $\Longrightarrow$ the normal representation

An n-bit number:

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Has a factor up to 2^{n-1}.
- Has a range: 0 to $(2^n - 1)$
- Example

$$\begin{aligned} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2 \\ = \quad & 0 + \cdots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ = \quad & 0 + \cdots + 8 + 0 + 2 + 1 = 11_{10} \end{aligned}$$

- Using 32 bits: 0 to +4,294,967,295

Signed Binary Integers (1/2)

How to encode a negative sign?

One's Compliment: Invert unsigned representation to get negative.

- Get value by inverting all bits then multiply by -1.
- Leading bit decides if negative or not.
- All positive numbers have the same representation as unsigned.

In one's compliment:

- $(0101)_2 = (0101)_2 = 5$
- $(1101)_2 = -1 \times (0010)_2 = -2$
- $(0000)_2 = (0000)_2 = 0$
- $(1111)_2 = -1 \times (0000)_2 = -0$????

One's compliment is rarely used:

- Signed zero.
- Weird borrowing required in arithmetic.

Signed Binary Integers (2/2)

How to encode a negative sign?

Two's Compliment: Invert all the bits with respect to 2^n

- Same as treating leading bit as negative in expansion.
- Leading bit decides if negative or not.
- All positive numbers have the same representation as unsigned.

In two's compliment:

- $(0101)_2 = (0101)_2 = 5$
- $(1101)_2 = -1 \times 2^3 + (0101)_2 = -8 + 5 =$ -3
- $(0000)_2 = (0000)_2 = 0$
- $(1111)_2 = -1 \times 2^3 + (0111)_2 = -1$

Two's Compliment

Advantages:

- Arithmetic is the same whether positive or negative:

$$\begin{array}{rl} & (0101)_2 = 5 \\ + & (1101)_2 = -3 \\ \hline & (0010)_2 = 2 \end{array} \quad \text{(Throw away carry bit)}$$

- No signed 0.
- One extra value represented with same number of bits.

For an n-bit number:

- Range of values is -2^{n-1} to $2^{n-1} - 1$

Same Bits Different Numbers

It is important to realize that the same bit pattern can represent different numbers.

$$(1001\ 1010)_2 \implies (154)_{10} \quad \text{interpretted as unsigned}$$
$$\implies (-102)_{10} \quad \text{interpretted as two's compliment}$$

Can be disastrous in programming!

```
unsigned int a = (1 << 31); // a = 2147483648
int b = a;                  // b = -2147483648
```

Signed Negation

In two's compliment, **bit-wise complement** then **add 1**.

$$6 = (0110)_2 = (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$$

$$\Downarrow \text{ compliment}$$

$$(1001)_2 = (-1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = -8 + 1$$

$$\Downarrow \text{ add one}$$

$$(1001)_2 + (0001)_2 = (1010)_2 = -8 + 0 + 4 + 0 = -6$$

Also works in reverse! (from negative to positive)

- ↳ Still, compliment then add 1.
- ↳ $-6 = (1010)_2 \Rightarrow (0101)_2 + 1 \Rightarrow (0110)_2 = 6$

Signed Extension

Signed Extension:

- Represent a number using more bits but keep numerical value.
- Very easy in two's compliment!
- Copy the signed bit to the left until desired number of bits.

Examples: 8-bit to 16-bit

- 2: 0000 0010 $\Rightarrow$ 0000 0000 0000 0010
- -2: 1111 1110 $\Rightarrow$ 1111 1111 1111 1110
- -10: 1111 0110 $\Rightarrow$ 1111 1111 1111 0110

Note: Truncation (representing a number using less bits) is tricky and you must know what you're doing.

Logical Shift

Logical Shift:

- Shift the bits left or right a specified number of times.
- Fills the vacancies with 0s on shift left and shift right.
- Throw away any bits that flow out.
- << (shift left) and >> (shift right) in C (unsigned).

Examples (in 8 bits):

- $2 << 3 = (0000\ 0010) << 3 = (0001\ 0000) = 16$.
- $8 >> 2 = (0000\ 1000) >> 2 = (0000\ 0010) = 2$.
- -4 >> $1 = (1111\ 1100) >> 1 = (0111\ 1110) = 126$.
  - ↳ This last one is ambiguous if it is logical or arithmetic shift. In high-level programming languages the right shift operator is usually an *arithmetic shift*...

Arithmetic Shift

Arithmetic Shift:

- Shift the bits left or right a specified number of times.
- Fills the vacancies with 0s on shift left.
- Fills the vacancies with 1s on shift right if number is negative.
- Fills the vacancies with 0s on shift right if number is positive.
- Throw away any bits that flow out.
- << (shift left) and >> (shift right) in C (signed).

Examples (in 8 bits):

- $2 << 3 = (0000\ 0010) << 3 = (0001\ 0000) = 16$.
- $8 >> 2 = (0000\ 1000) >> 2 = (0000\ 0010) = 2$.
- $-4 >> 1 = (1111\ 1100) >> 1 = (1111\ 1110) = -2$.

CS3350B Computer Organization
Chapter 2: Synchronous Circuits
Part 1: Gates, Switches, and Boolean Algebra

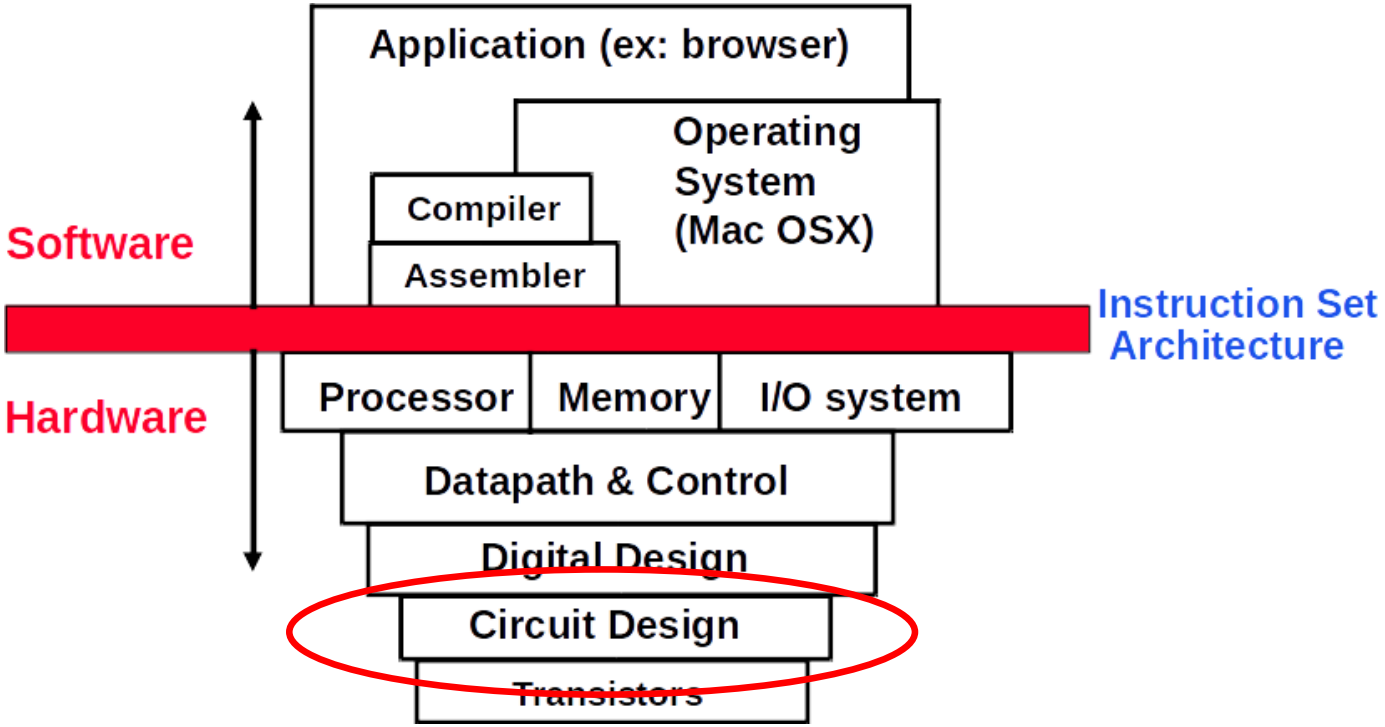
Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Tuesday January 29, 2019

Outline

Layers of Abstraction



After looking at high-level CPU and Memory we will now go down to the lowest level (that we care about).

Circuit Design vs Digital (Logic) Design

↳ Design of individual circuits vs Using circuits to implement some logic.

Circuit Design

Why do we care?

- Appreciate the limitations of hardware.
- Understand why some things are fast and some things are slow.
- Need circuit design to understand logic design.
- Need logic design to understand **CPU Datapath**.

If you are ever working with:

- Assembly, ISAs,
- Embedded Systems and circuits,
- Specialized computer/logic systems,

you will need circuit and logic design.

Digital Circuits

Everything is **digital:** represented by discrete, individual values.

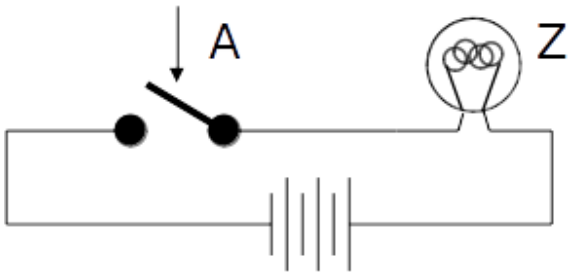
- ↳ No gray areas or ambiguity.

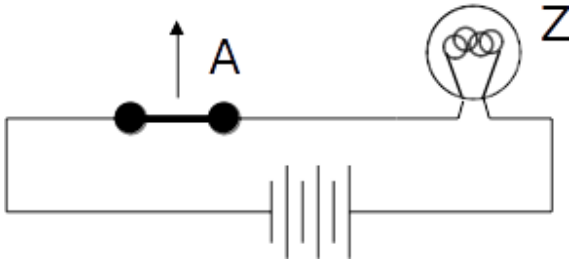
Must convert an **analog** – continuously variable – signal to digital.

For us, the analog signal is electricity (voltage).

- ↳ "High" voltage ⇒ 1
- ↳ "Low" voltage ⇒ 0

Physicality of Circuits





In the end, everything is a switch.

"Input" $\Rightarrow$ A
"Output" $\Rightarrow$ Z

If A is 0/false then switch is open.
If A is 1/true then switch is closed.

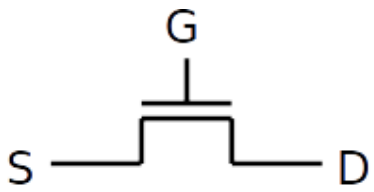
This circuit implements:

$$\mathbf{A} \equiv \mathbf{Z}$$

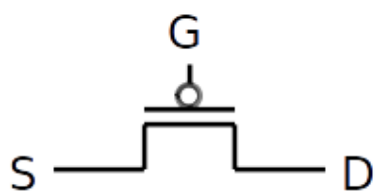
Transistors: Electrically Controlled Switches

MOS-FET: Metal-Oxide-Semiconductor Field-Effect **Transistor**

- Has a source (S), a drain (D), and a gate (G).
- Applying voltage to G allows current to flow between S and D.
- In reality, transistors, logic gates, SRAM, use CMOS (Complimentary-MOS). But we don't care about transistors really...



n-channel
opens when voltage at G is low,
closes when voltage at G is high



p-channel
closes when voltage at G is low,
opens when voltage at G is high

Flipping a transistor is *much faster* than moving a physical switch.

↳ Speed of switching a transistor directly related to speed of a CPU

Outline

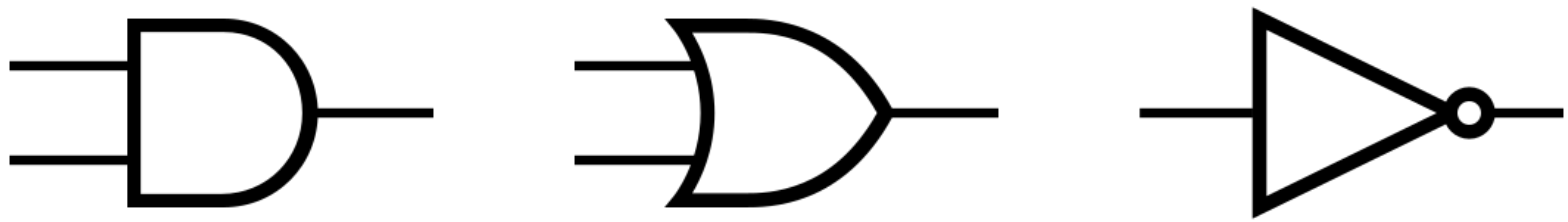
Logic as Circuits

Propositional Logic: A set of propositions (truth values) combined by some logical connectives.

- Truth values ≡ Binary digital signal
- Logical connectives ≡ **Logic gates**

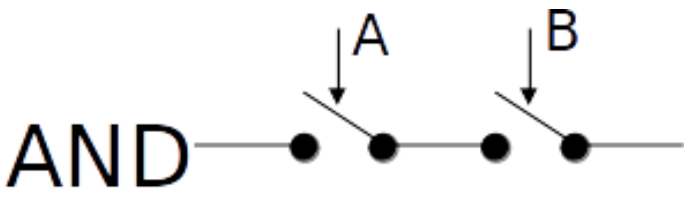
Logic Gate: A circuit implementing some logical expression/function.

The basics: **AND** ($\wedge$), **OR** ($\vee$), **NOT** ($\neg$).

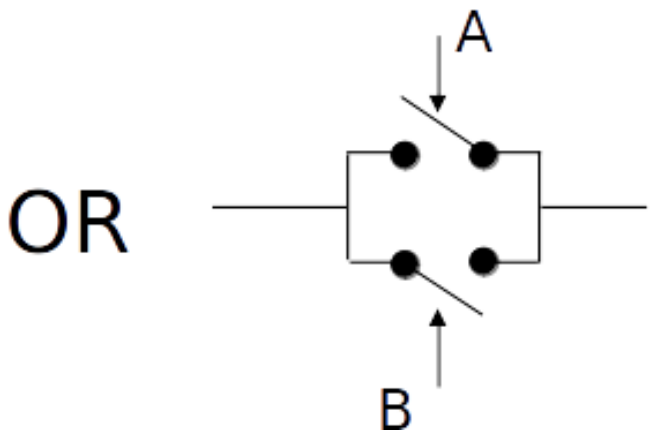


Arity of a function/gate is the number of inputs.

Gates as Switches



- Both A and B must be true/1 to get the circuit to complete.



- Either A or B can be true/1 to get the circuit to complete.

Logic Gates In Detail: AND

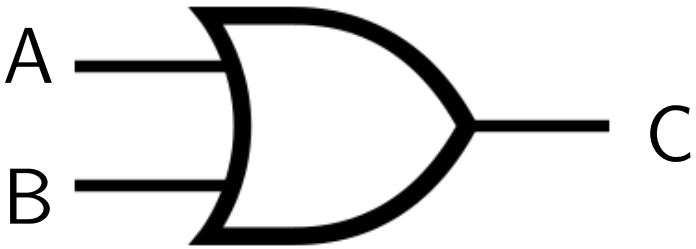
$$A \wedge B \equiv C$$

$$A \cdot B \equiv C$$

Truth Table for AND

A	B	A ∧ B ≡ C
0	0	0
0	1	0
1	0	0
1	1	1

Logic Gates In Detail: OR



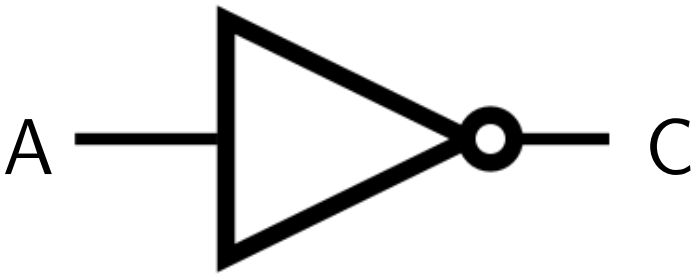
$$A \vee B \equiv C$$

$$A + B \equiv C$$

Truth Table for OR

A	B	A ∨ B ≡ C
0	0	0
0	1	1
1	0	1
1	1	1

Logic Gates In Detail: NOT



Truth Table for NOT

A	$\neg A \equiv C$
0	1
1	0

$$\neg A \equiv C$$

$$\overline{A} \equiv C$$

More Interesting Logic Gates: NAND

Truth Table for NAND

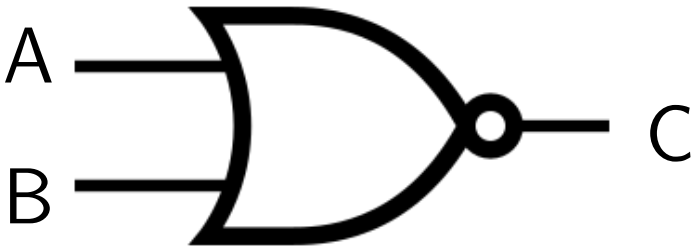
A	B	$\overline{A \cdot B} \equiv C$
0	0	1
0	1	1
1	0	1
1	1	0

$$\neg(A \wedge B) \equiv C$$

$$\overline{A \cdot B} \equiv C$$

$$A \mid B$$

More Interesting Logic Gates: NOR



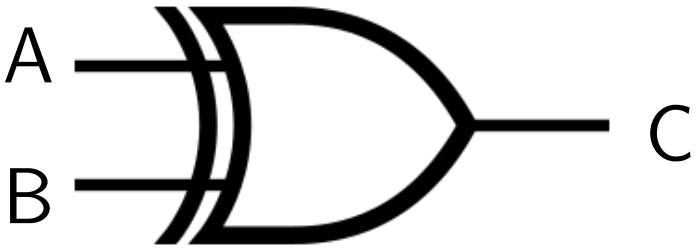
$$\neg(A \vee B) \equiv C$$

$$\overline{A + B} \equiv C$$

Truth Table for NOR

A	B	$\overline{A + B} \equiv C$
0	0	1
0	1	0
1	0	0
1	1	0

More Interesting Logic Gates: XOR (Exclusive OR)



$$A \oplus B \equiv C$$

Truth Table for XOR

A	B	$A \oplus B \equiv C$
0	0	0
0	1	1
1	0	1
1	1	0

Outline

The Algebra of Logic Gates

Due to the equivalence of truth values and binary digital signals, **Boolean Algebra** is heavily used discussing circuitry.

Associativity:

$$(A + B) + C \equiv A + (B + C)$$
$$(A \cdot B) \cdot C \equiv A \cdot (B \cdot C)$$

Identity:

$$A + 0 \equiv A$$
$$A \cdot 1 \equiv A$$

Commutativity:

$$A + B \equiv B + A$$
$$A \cdot B \equiv B \cdot A$$

Annihilation:

$$A + 1 \equiv 1$$
$$A \cdot 0 \equiv 0$$

Distributivity:

$$A + (B \cdot C) \equiv (A + B) \cdot (A + C)$$
$$A \cdot (B + C) \equiv (A \cdot B) + (A \cdot C)$$

Idempotence:

$$A + A \equiv A$$
$$A \cdot A \equiv A$$

Boolean Algebra: More Interesting Laws

Absorption:

$$A \cdot (A + B) \equiv A$$
$$A + (A \cdot B) \equiv A$$

Double Negation

$$\overline{\overline{A}} \equiv A$$

Complementation:

$$A + \overline{A} \equiv 1$$
$$A \cdot \overline{A} \equiv 0$$

De Morgan's Laws:

$$\overline{A + B} \equiv \overline{A} \cdot \overline{B}$$
$$\overline{A \cdot B} \equiv \overline{A} + \overline{B}$$

Look familiar?

↳ Definitions of NOR and NAND.

Proving De Morgan's Laws

Proof by Exhaustion:

- The easiest way to prove something is to write out each expression's truth table.

$$\overline{A+B} \equiv \overline{A} \cdot \overline{B}$$

A	B	A + B	$\overline{A+B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

A	B	$\overline{A}$	$\overline{B}$	$\overline{A} \cdot \overline{B}$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

Simplifying Expressions with Boolean Algebra (1/2)

$$\overline{xyz} + \overline{xy}z$$

$$\begin{aligned} \overline{xyz} + \overline{xy}z &\equiv \overline{xy}(\overline{z} + z) && \text{Factor } \overline{xy} \\ &\equiv \overline{xy}(1) && \text{Complementation of } z \\ &\equiv \overline{xy} && \text{Identity with } \overline{xy} \end{aligned}$$

x	y	z	$\overline{xyz}$	$\overline{xy}z$	$\overline{xyz} + \overline{xy}z$
0	0	0	1	0	1
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	0	0

Simplifying Expressions with Boolean Algebra (2/2)

Sometimes a truth table is too challenging...

↳ For v variables a truth table has 2^v rows.

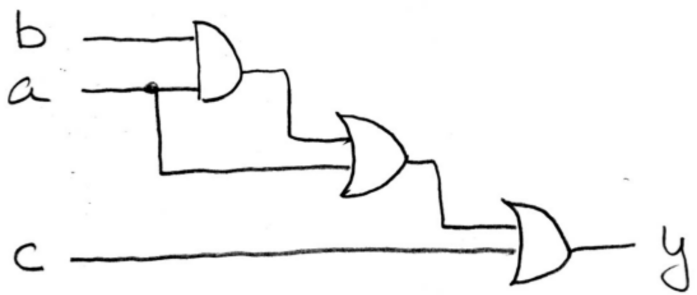
$$\overline{(\overline{x}+\overline{z})}\,(abcd+xz) \implies \text{6 variables, 64 rows}$$

Instead we can simplify using the laws of Boolean algebra:

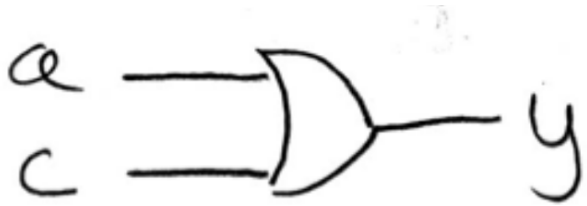
$$\begin{aligned}
\overline{(\overline{x}+\overline{z})}\,(abcd+xz) &\equiv \overline{\overline{xz}}\,(abcd+xz) && \text{De Morgan's Law} \\
&\equiv xz\,(abcd+xz) && \text{Double negation of } x \text{ and } z \\
&\equiv xz && \text{Absorption}
\end{aligned}$$

Simplifying Expressions for Simplified Circuits

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



$$\begin{aligned} y &\equiv (ab + a) + c \\ &\equiv a\,(b + 1) + c && \text{Factor } a \\ &\equiv a\,(1) + c && \text{Annihilaltion} \\ &\equiv a + c && \text{Identity} \end{aligned}$$



Canonical Forms

Different standard or **canonical forms**.

- **Conjunctive Normal Form** (CNF) ⇒ AND of ORs
  - ↳ "Product-of-sums"
- **Disjunctive Normal Form** (DNF) ⇒ ORs of ANDs
  - ↳ "Sum-of-products"

CNF $(a+b)\cdot(\overline{a}+b)\cdot(\overline{a}+\overline{b})$

DNF $ab+\overline{a}b+\overline{a}\overline{b}$

- Every variable should appear in every sub-expression.
  - ↳ Products for DNF, Sums for CNF.
  - ↳ Some authors call this "Full DNF" or "Full CNF".
- Every boolean expression can be converted to a canonical form.
- DNF more useful and practical ⇒ truth tables.

Truth Tables and Disjunctive Normal Forms

We can get a DNF expression directly from a truth table.

- a, b, c are inputs, f is output.
- Create one product term for every entry in the table with $f \equiv 1$.
- Put $\overline{x}$ in product if x is false in that row.
- Put x in product if x is true in that row.
- OR all products together.

a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$$\Longrightarrow \qquad \overline{abc} + \overline{a}b\overline{c} + a\overline{bc} + abc$$

Functional Completeness

Functional Completeness - A set of functions (operators) which can adequately describe every operation and outcome in an algebra.

- For Boolean algebra the classical set of operators: $\{+, \cdot, \neg\}$ is functionally complete but not **minimal**.
- Thanks to De Morgan's Law we only need one of AND or OR.
- The sets $\{+, \neg\}$ and $\{\cdot, \neg\}$ are both functionally complete and minimal.
  - ↳ **minimal** - removing any one of the operators would make the set functionally *incomplete*.
- NAND alone is functionally complete; so is NOR alone.

NAND is Functionally Complete

NAND alone is functionally complete.

- NAND $\equiv$ $|$
- To prove functional completeness simply show that the operators of the set $\{+, \cdot, \neg\}$.

$\neg X \equiv X \mid X$

$X \cdot Y \equiv \overline{X|Y} \equiv (X|Y) \mid (X|Y)$

$X + Y \equiv \overline{\overline{X + Y}} \equiv \overline{\overline{X} \cdot \overline{Y}} \equiv (X|X) \mid (Y|Y)$

X	$\overline{X}$	$X \cdot X$	$\overline{X \cdot X}$
0	1	0	1
1	0	1	0

X	Y	$A \equiv X\|Y$	$A\|A$
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

X	Y	$\overline{X}$	$\overline{Y}$	$\overline{X}\|\overline{Y}$
0	0	1	1	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	1

Summary

Boolean algebra can simplify circuits.

- Remove variables that the output does not depend on.
- Simplifies expression, removing needless gates.
- Space and time complexity improved!

Truth tables, canonical forms, functional completeness.

Help generating truth tables:

- `http://turner.faculty.swau.edu/mathematics/materialslibrary/truth/`

CS3350B Computer Organization
Chapter 2: Synchronous Circuits
Part 2: Stateless Circuits

Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Tuesday February 05, 2019

Outline

Stateless Circuits are Combinational Circuits

- **Stateless** ⇒ No memory.
- **Combinational** ⇒ Output is a combination of the inputs alone.

Combinational circuits are formed from a combination of logic gates and other combinational cirtcuits.

- ↳ Modular Design,
- ↳ Reuse,
- ↳ Simple to add new components.

Sometimes, these are called **functional blocks**, they implement *functions*.

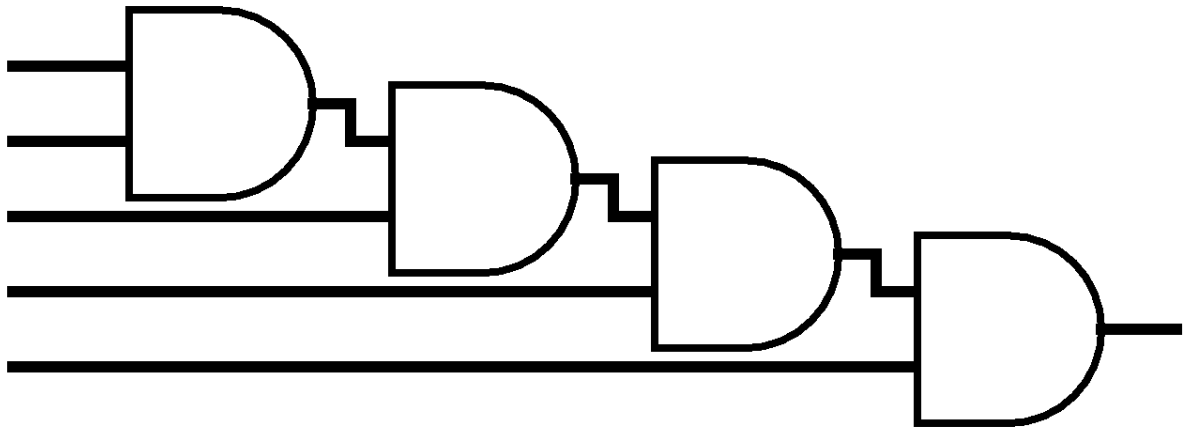
Increasing Arity

Arity: the number of inputs to a gate, function, etc.

How can we build an n-way add from simple 2-input and gates?

↳ Simply chain together $n-1$ 2-way gates.

Example: 5-way AND

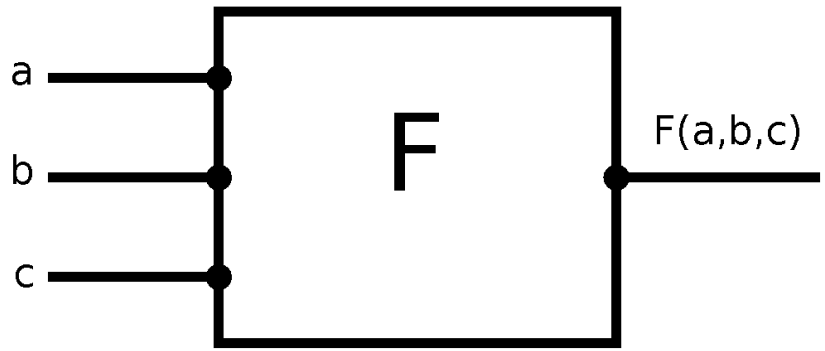


Works for AND, OR, XOR. *Doesn't* work for NAND, NOR.

Block Diagrams

A **block diagram** or **schematic diagram** can use used to express the high-level specification of a circuit.

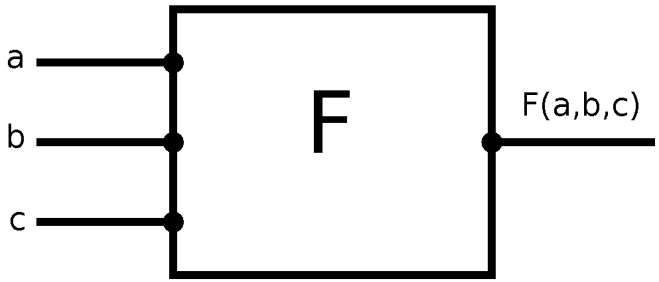
- How many inputs, how many bits for each input?
- How many outputs, how many bits for each output?
- What does the circuit do? Formula or truth table.



$$F \equiv \overline{abc} + abc$$

a	b	c	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

From Blocks to Gates (1/2)



a	b	c	F
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

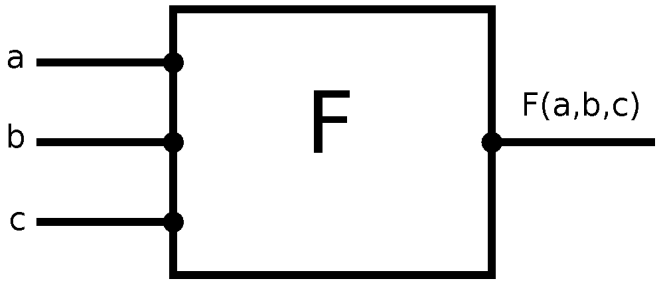
1. Generate truth table.
2. Get canonical form:

$$F \equiv \overline{a}bc + a\overline{b}c + ab\overline{c} + abc$$

3. Simplify if possible:

$$\begin{aligned}
&\overline{a}bc + a\overline{b}c + ab\overline{c} + abc \\
&\equiv \overline{a}bc + a\overline{b}c + ab\overline{c} + abc + abc + abc \\
&\equiv \overline{a}bc + abc + a\overline{b}c + abc + ab\overline{c} + abc \\
&\equiv bc + ac + ab
\end{aligned}$$

From Blocks to Gates (2/2)

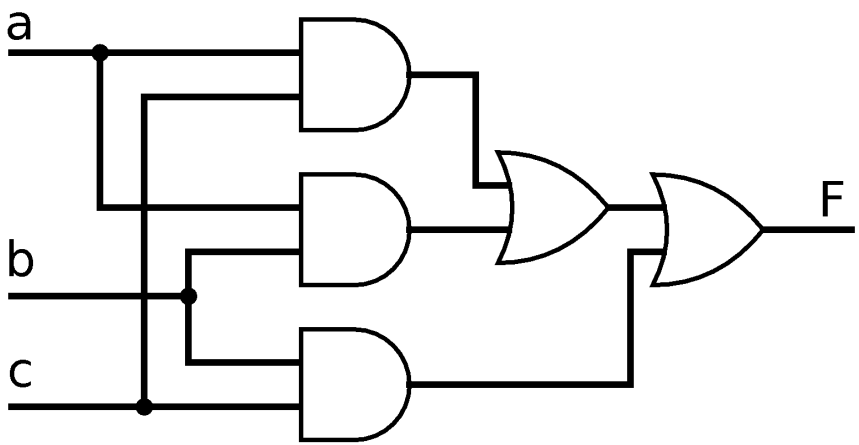


a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

3. Simplify if possible:

$$F \equiv bc + ac + ab$$

4. Draw your circuit from simplified formula.

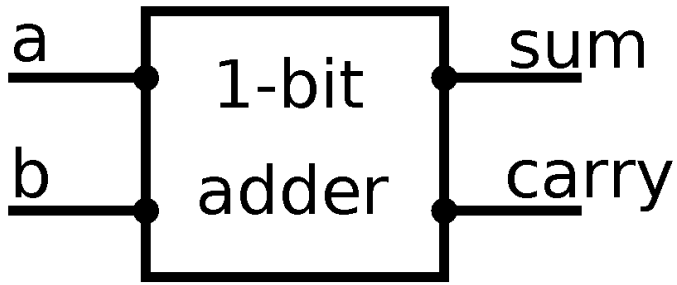


This is called a *majority* circuit. Output is true iff majority of inputs are true.

Outline

1 Bit Adder

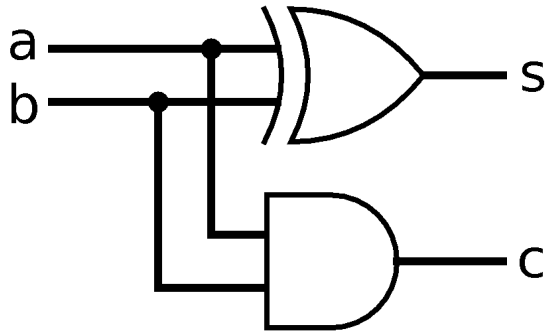
Adder interprets bits as a binary number and does addition.



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

$s = add(a, b)$

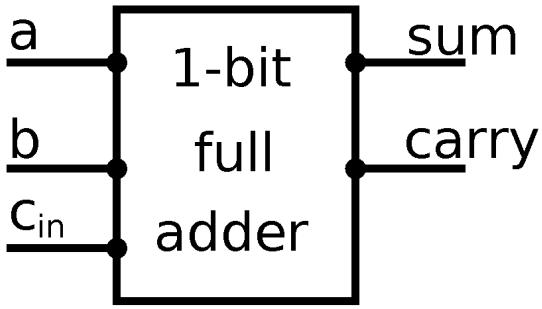
c = carry (overflow) bit

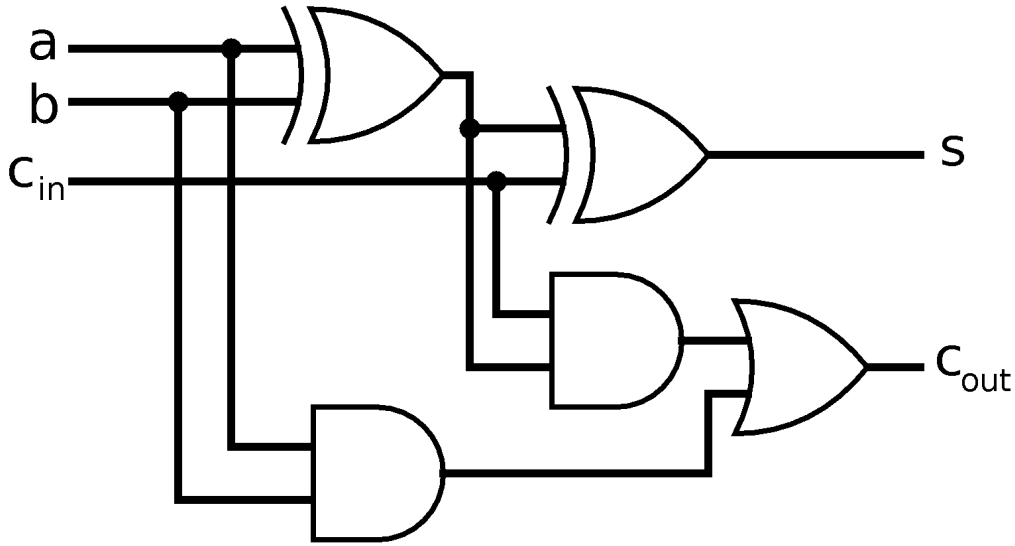


1 Bit Full Adder

Full Adder does addition of 3 inputs: a, b, and carry_{in}.

↳ Previous adder was a *half* adder.





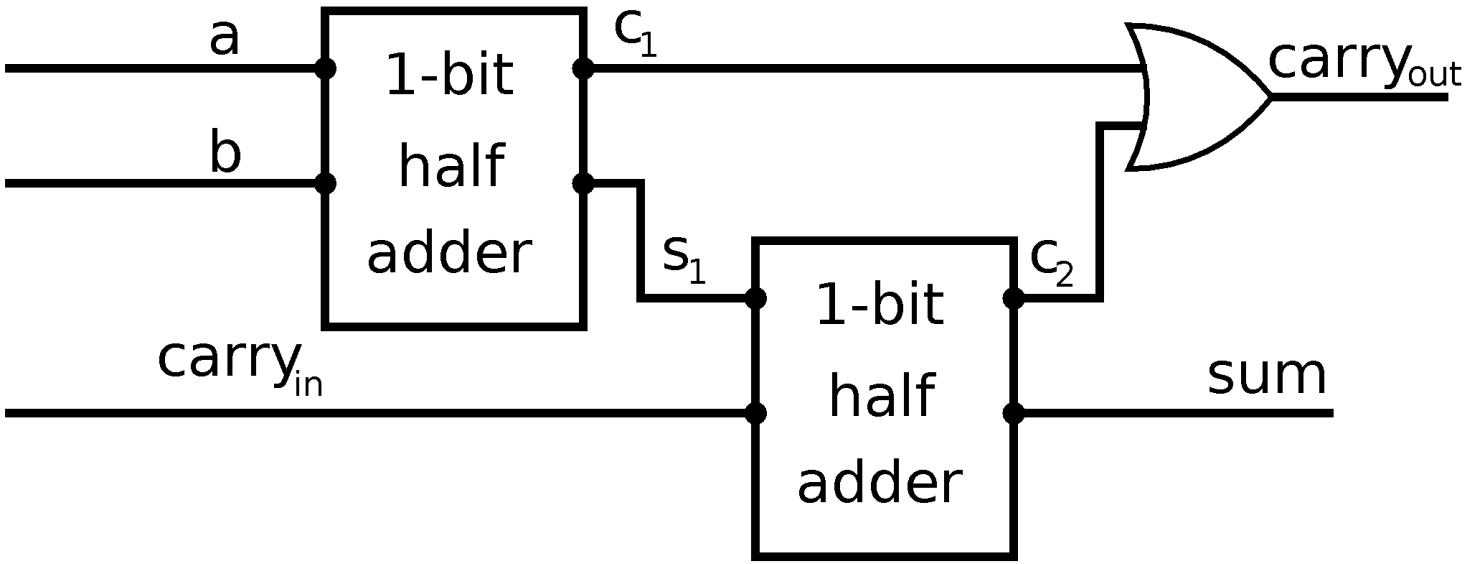
$$s = XOR(a, b, c_{in})$$

$$c = ab + (XOR(a, b) \cdot c_{in})$$

1 Bit Full Adder using Half Adders

A full adder can be built from half adders.

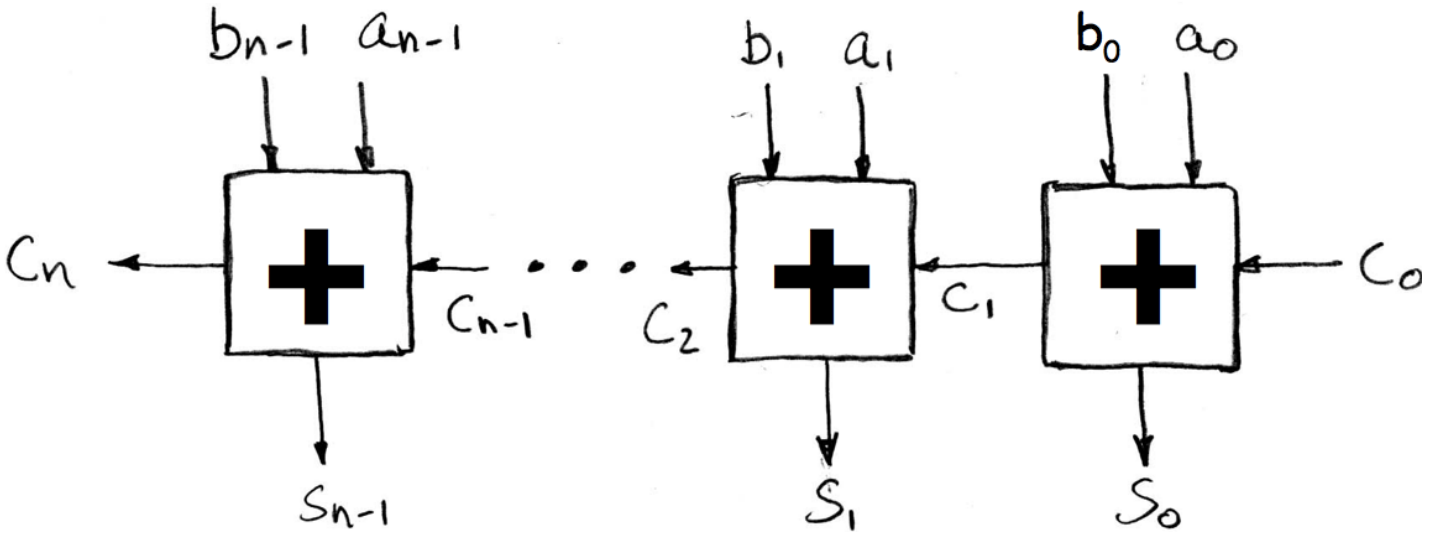
↳ Modular design, reuse, simplified view.



n-Bit Full Adder

n-bit adder: Add n bits with carry.

- Just like long addition done by hand.
- Combine n full adders, adding bit by bit, carrying the carry from lowest-ordered bit to highest-ordered bit.
- Final carry bit is c_n.



Addition Overflow (1/2)

Overflow occurs when arithmetic results in a number strictly larger than can fit in the predetermined number of bits.

- For *unsigned* integers, overflow is detected by c_n being 1.
- For *signed* (i.e. twos-compliment) integers, overflow more interesting.

Example: Addition in 4 bits.

$$\begin{array}{rl} 1000 & \text{(carry bits)} \\ 1101 & \\ +\ 0110 & \\ \hline 10011 & \Rightarrow \quad c_n \text{ is 1. Overflow?} \end{array}$$

Addition Overflow (1/2)

Overflow occurs when arithmetic results in a number strictly larger than can fit in the predetermined number of bits.

- For *unsigned* integers, overflow is detected by c_n being 1.
- For *signed* (i.e. twos-compliment) integers, overflow more interesting.

Example: Addition in 4 bits.

$$\begin{array}{rl} 1000 & \text{(carry bits)} \\ 1101 & \\ +\ 0110 & \\ \hline 10011 & \Rightarrow \quad c_n \text{ is 1. Overflow?} \end{array}$$

$$\begin{array}{rl} -\,3 & \\ +\quad 6 & \\ \hline 3 & \Rightarrow \quad \text{No overflow} \\ & \quad\ \text{Discard last carry bit} \end{array}$$

Addition Overflow (2/2)

In twos-compliment when is there overflow?

- Most significant bit encodes a negative number in two's compliment.
- If both operands are positive *and* $c_{n-1} \equiv 1$ then we have overflow.
- If one positive and one negative, overflow impossible.
  - ↳ Their sum is always closer to zero than either of the operands.
- If both operands are negative *and* $c_n \equiv 1$ then we have overflow.

$$\begin{array}{r} 1000 \\ 0101 \\ +0110 \\ \hline 1011 \end{array} \Rightarrow \text{Overflow}$$

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

$$\begin{array}{r} 1000 \\ 1101 \\ +\ 0110 \\ \hline 10011 \end{array} \Rightarrow \text{No overflow}$$

Overflow in two's compliment: c_n XOR c_{n-1}.

n-bit Subtractor (1/2)

n-bit subtractor: Subtract two n-bit numbers.

- We want $s = a - b$.
- Start with an n-bit adder.
- XOR b with a **control signal** for subtraction.
  - ↳ signal is 1 for subtraction, 0 for addition.

XOR works as conditional inverter.

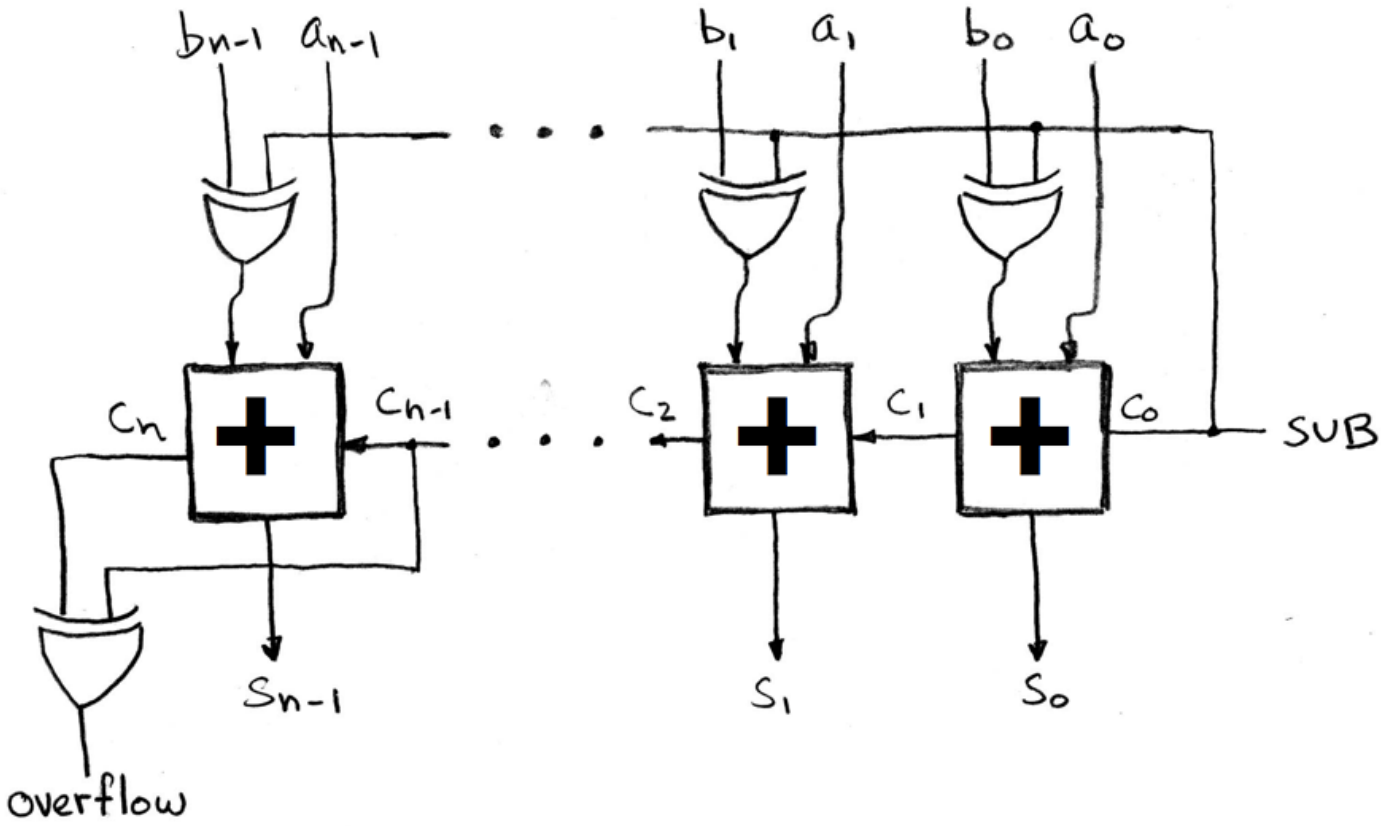
↳ A XOR B ≡ C $\Longrightarrow$ if (B) then $\overline{\text{A}}$ ≡ C else A ≡ C.

A	B	A ⊕ B ≡ C
0	0	0
0	1	1
1	0	1
1	1	0

n-bit Subtractor (2/2)

Control signal SUB acts as c_0.

- Recall: *signed negation*. Invert and add one.
- XOR does invert.



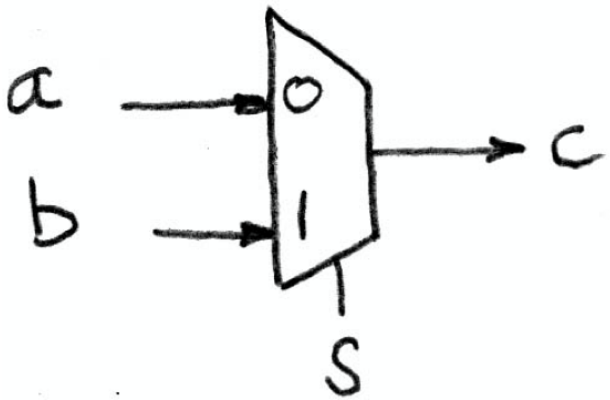
Outline

Multiplexer

A **multiplexer** "mux" conditionally chooses among its inputs to set value of output.

- Uses **control signal** to control which input is chosen.
- Still no state, output depending only on inputs: input bits and control signal.

2-way multiplexer



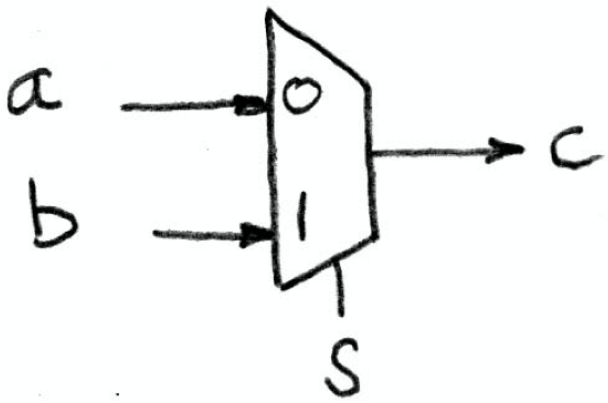
If $s \equiv 0$ then $c \equiv a$.
If $s \equiv 1$ then $c \equiv b$.

Notice actual value of a and b have no effect on decision.

↳ 0 and 1 in multiplexer is not the *value* of a or b but the "index".

2-way Multiplexer

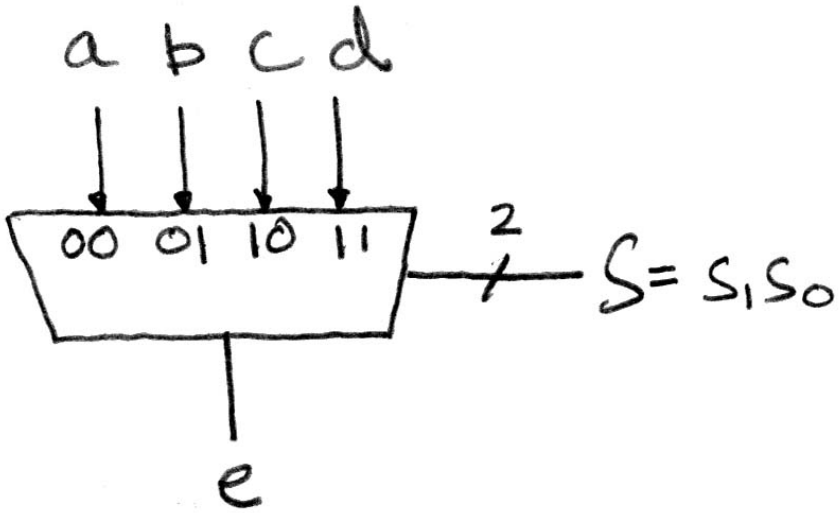
How to encode this "if-then" behaviour without actual conditionals?



$$
\begin{aligned}
c &\equiv MUX(a, b, s) \\
&\equiv \overline{s}a(b + \overline{b}) + sb(a + \overline{a}) \\
&\equiv \overline{s}a + sb
\end{aligned}
$$

Note: $X \cdot (Y + \overline{Y})$ encodes "X independent of what the value of Y is".

4-way Multiplexer

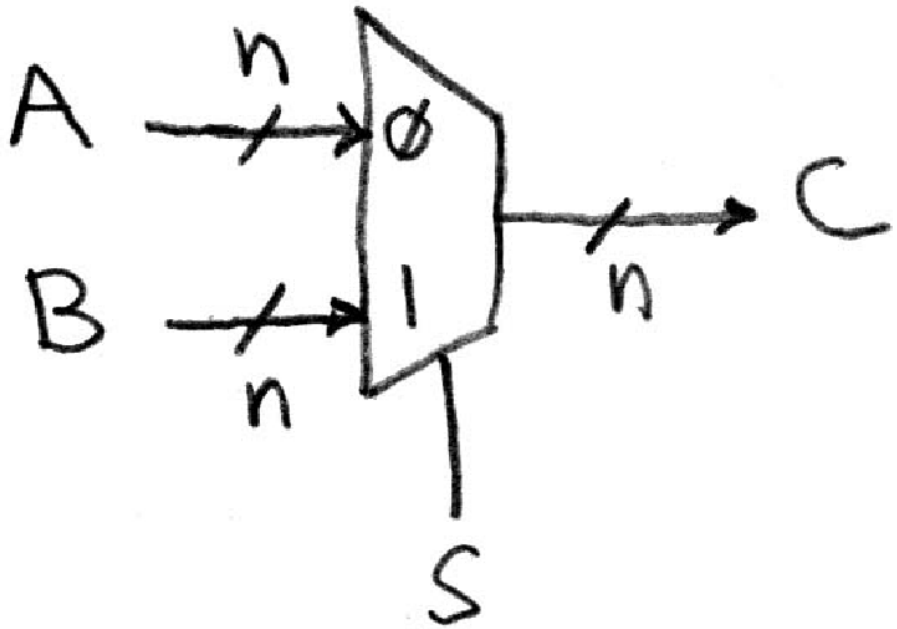


$$
\begin{aligned}
e &\equiv MUX(a, b, c, d, S) \\
&\equiv \overline{s_1 s_0} a + \overline{s_1} s_0 b \\
&\quad + s_1 \overline{s_0} c + s_1 s_0 d
\end{aligned}
$$

The index of each input is now 0 through 3.

- Need 2 bits to choose among 4 inputs.
- Control signal's **bit-width** is now 2.

Big Data Multiplexer

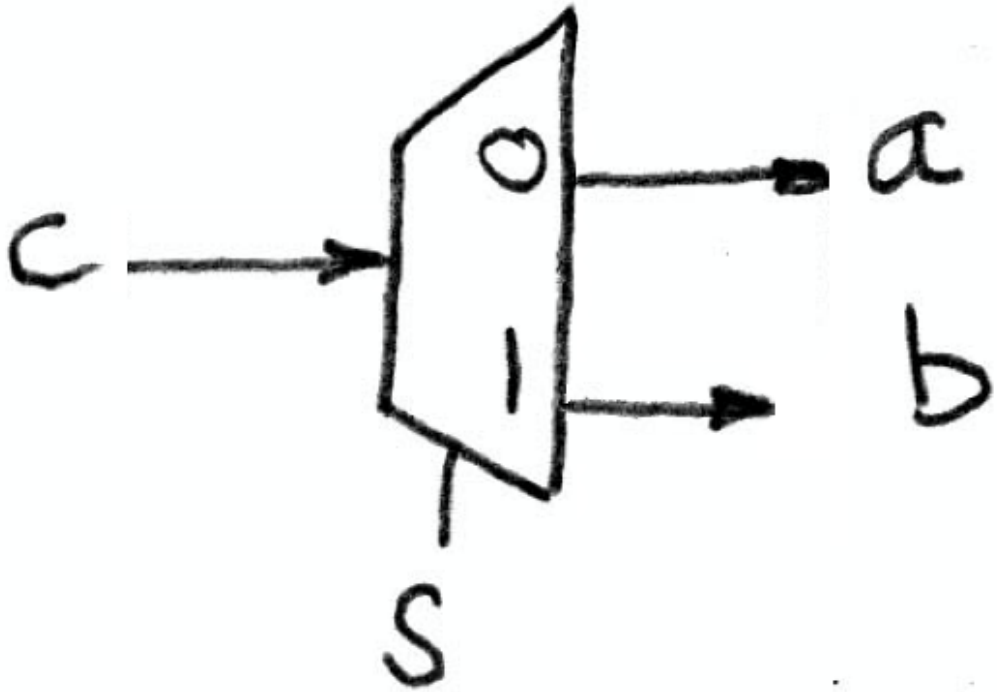


Bit-width of input and output must match, but bit-width of *control signal* only determined by number of inputs to choose from.

Demultiplexer

Demultiplexer "demux" conditionally chooses among its outputs.

- Opposite of MUX.
- Un-selected outputs are set to 0.

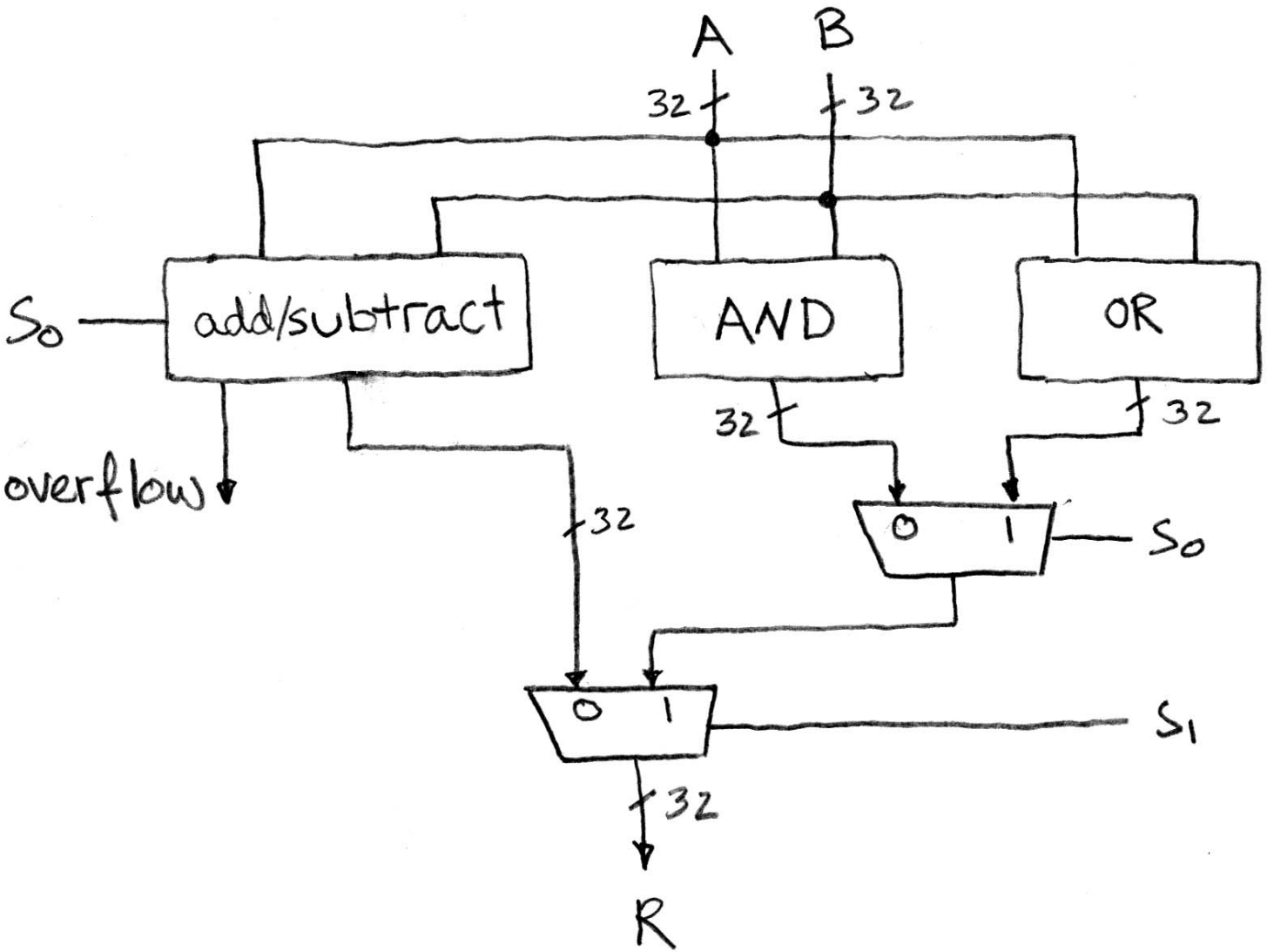


Outline

Arithmetic Logic Unit

- An **ALU** is a black-box type circuit which can do many *different* arithmetic or logic operations on its inputs.
  - ↳ Not many at one time, but selectively acts as many.
- Depending on the implementation can do addition, subtraction, multiplication, division, logical AND, logical OR, shifting, etc.
- Use a control signal to choose which operation to perform.
- Essentially a big collection of MUX and combinational blocks for each operation.

Simple ALU Circuit



Optimizing ALU

Remember, every additional gate increases delay and space. Instead, optimize via the normal four step process:

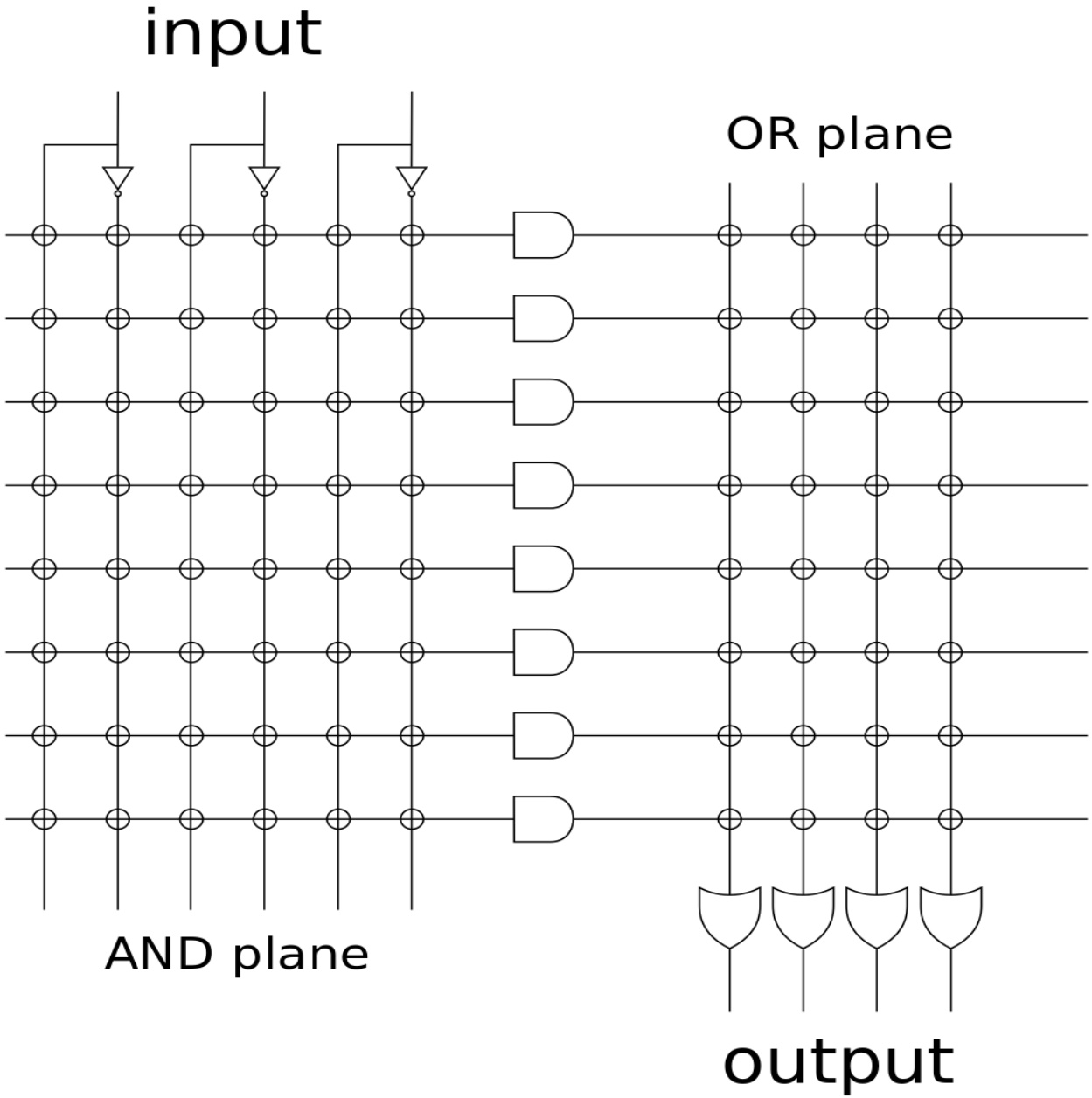
1. Generate a truth table,
2. Get canonical from from truth table,
3. Simplify expression,
4. Make circuit.

Another option: programmable logic array.

Programmable Logic Array

A **programmable logic array** (PLA) directly implements a truth table/canonical disjunctive normal form.

- Each AND row is a truth table row.
- Each OR column is one output bit.
- Each $\oplus$ is a *programmable* (i.e. optional) join of the input to the circuit.



CS3350B Computer Organization
Chapter 2: Synchronous Circuits
Part 3: State Circuits

Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

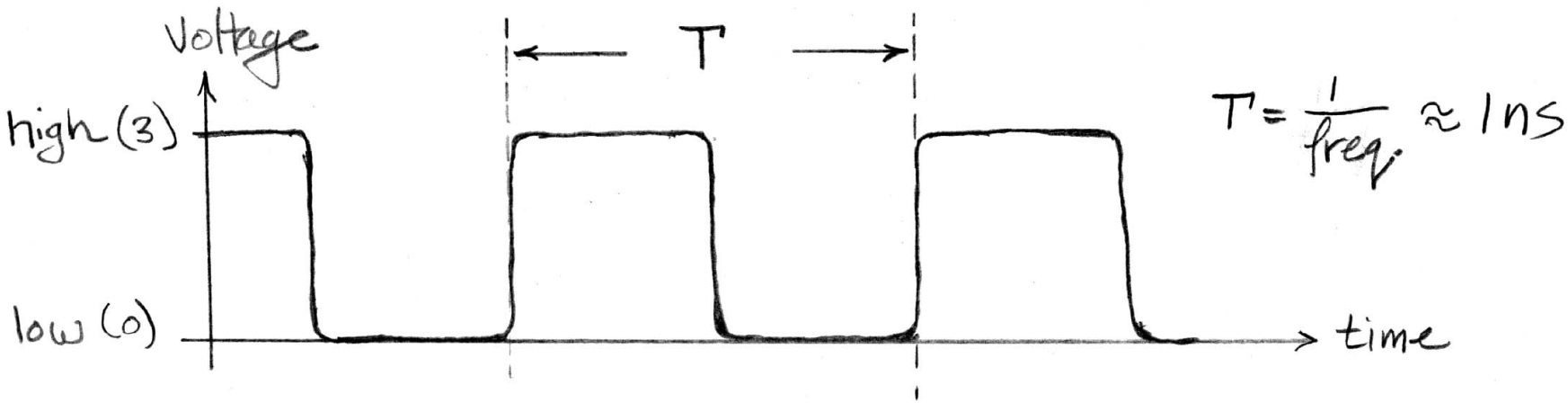
Thursday February 07, 2019

Outline

Digital Signals

We digitalize an analog (voltage) signal to encode binary.

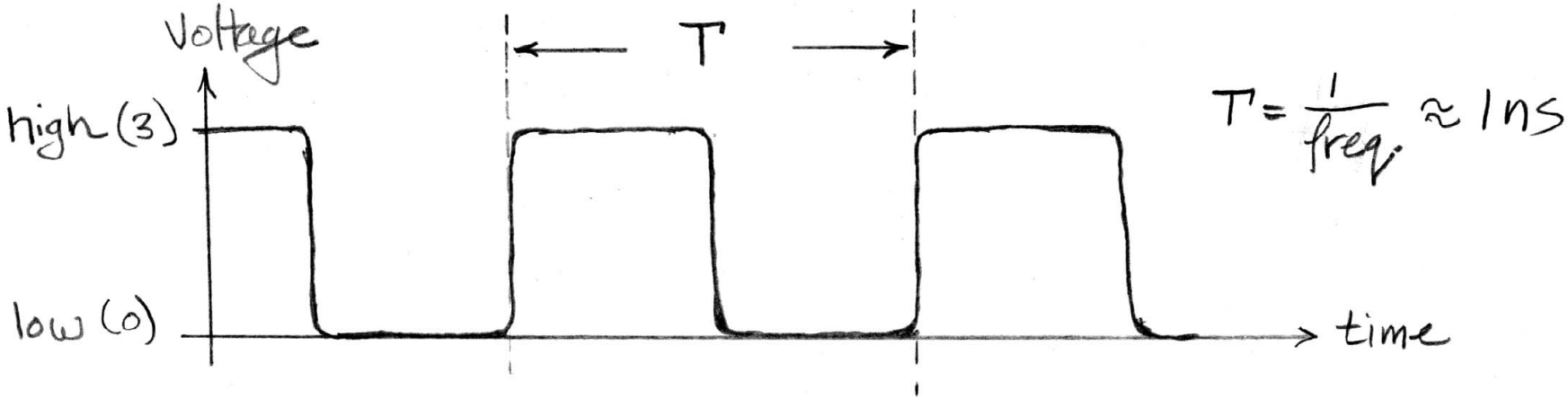
- "High" voltage $\Rightarrow$ 1.
- "Low" voltage $\Rightarrow$ 0.



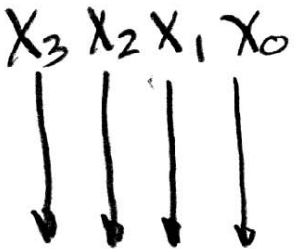
Transmitting Digital Signals

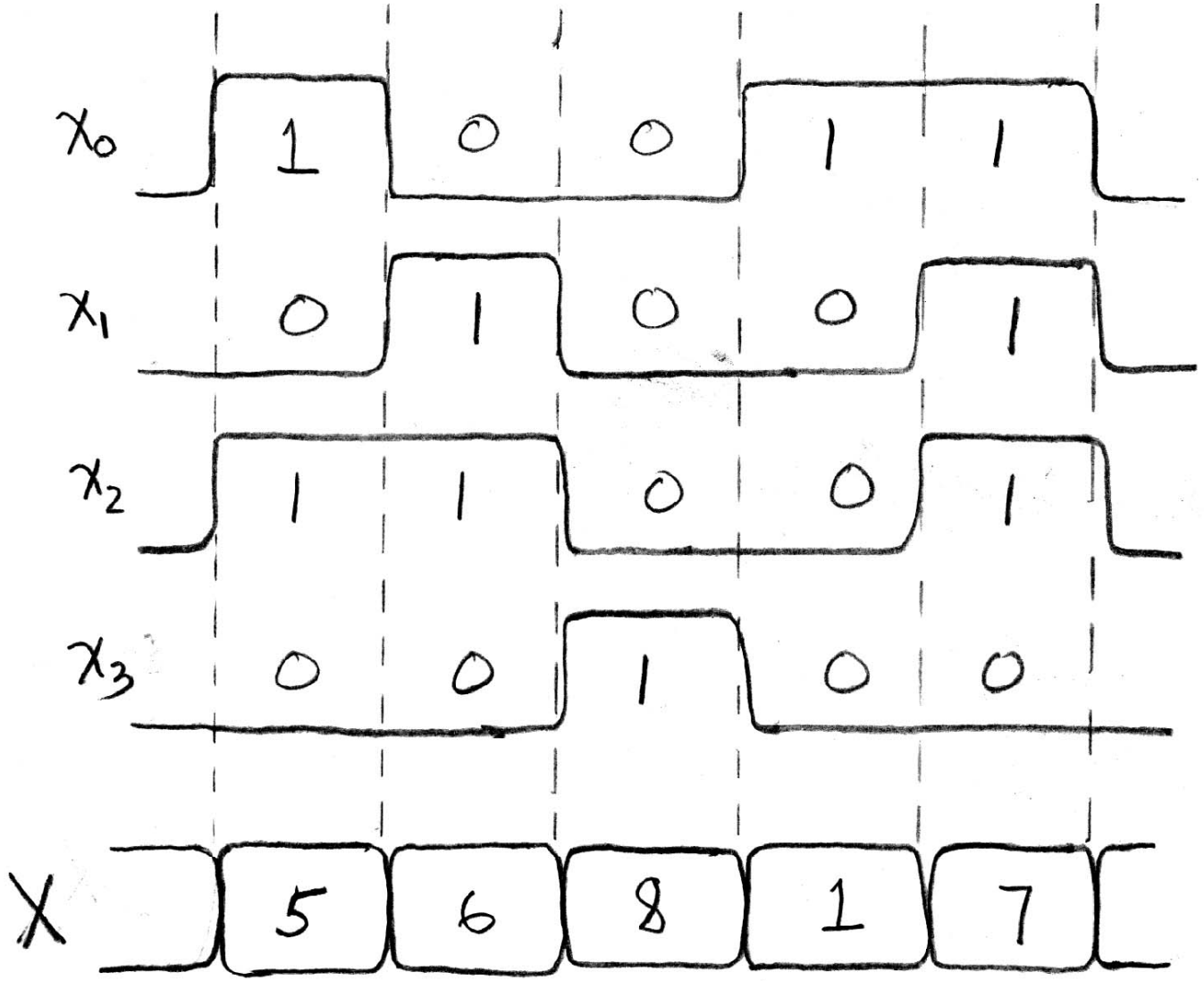
For our purposes:

- Transmission is continuous. There's always something on the wire.
- Transmission/switching is effectively instantaneous.



Grouping Signals To Encode Many Bits

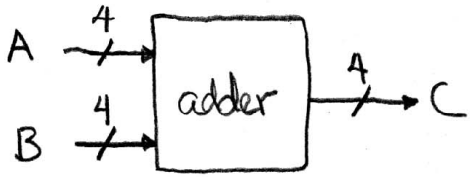




Signals and Circuits

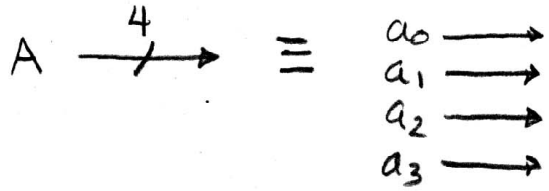
Unfortunately for us, combinational circuits cause **propagation delay**.

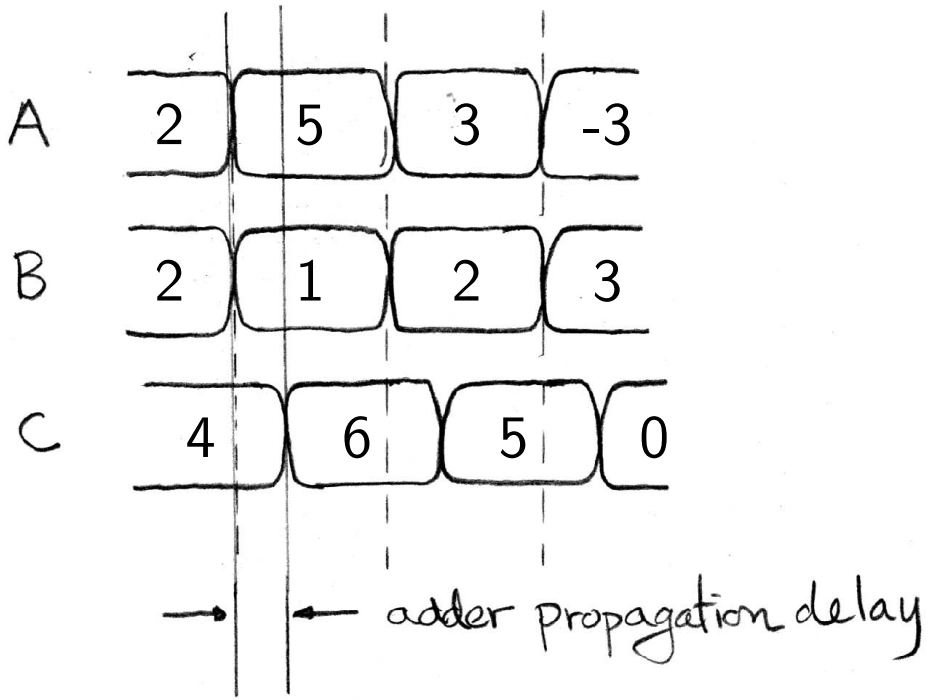
- The more complex the circuit the longer the delay.
- Every gate adds some delay.



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

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





Dealing with Delay

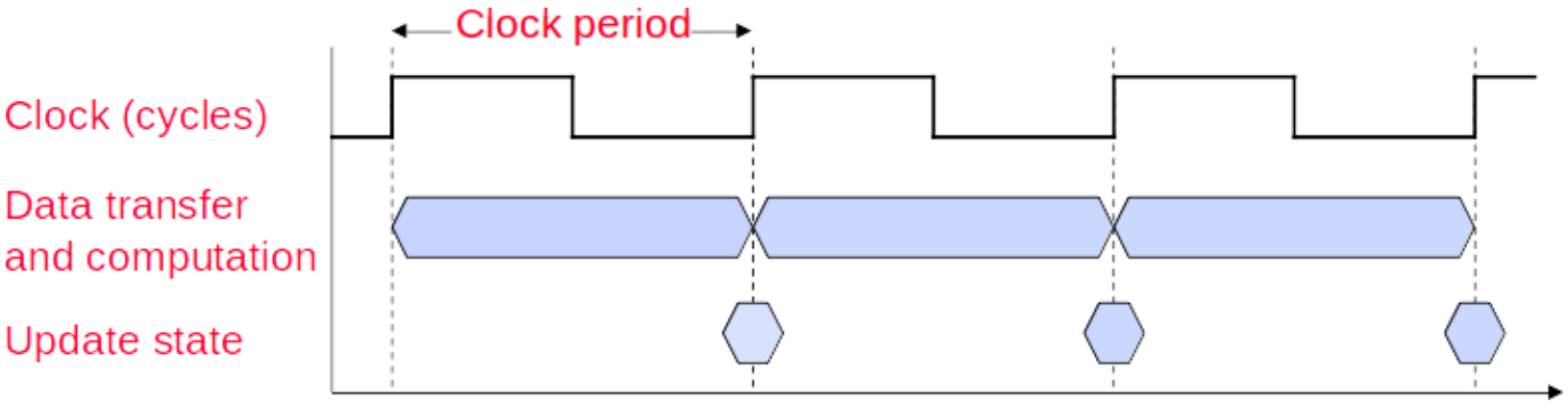
Problems with propagation delay:

- Inputs transmit (change) instantaneously, but *output* does not.
- When can the next circuit read the output and ensure it is getting the correct value?

Synchronize the circuits via a **clock**.

Outline

The Clock Signal



The clock is a digital signal which has a precise timing for switching between 1/0.

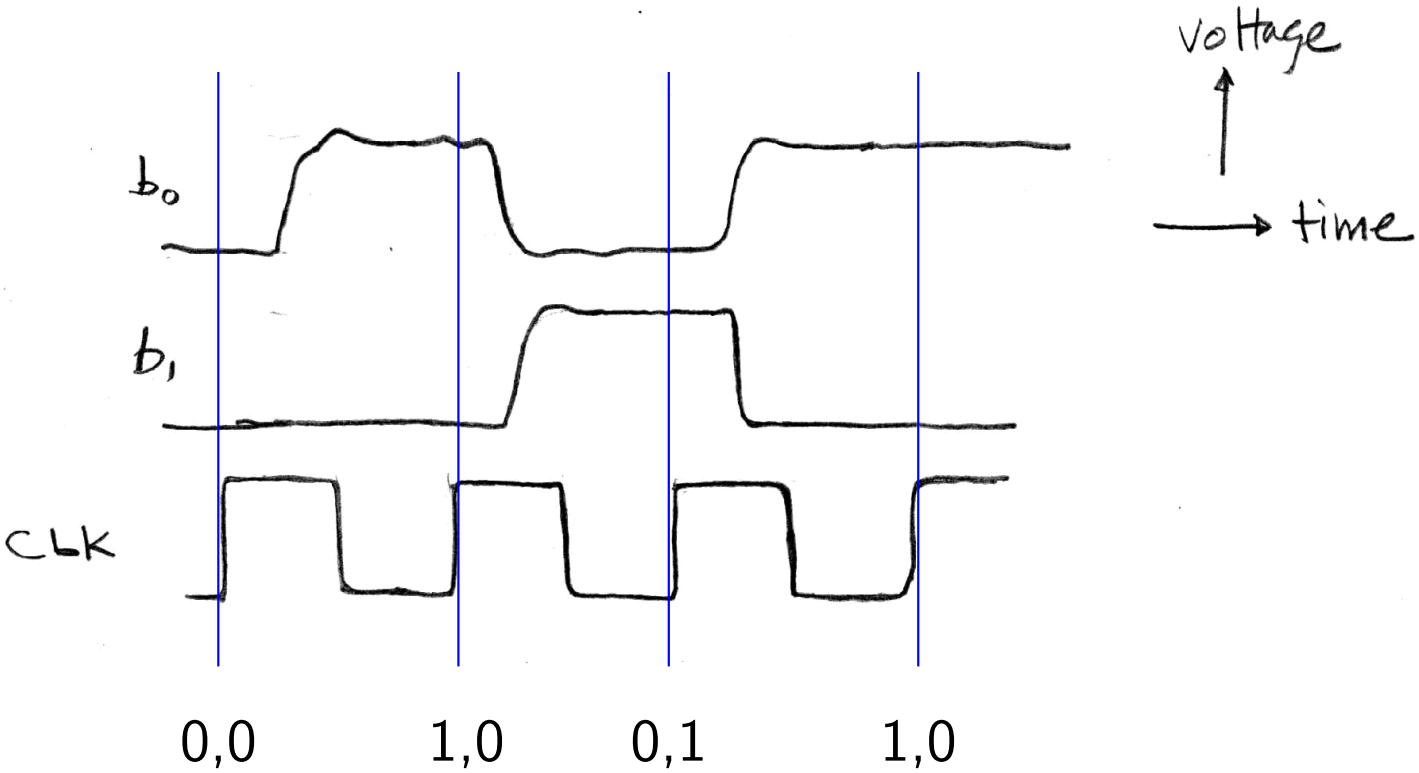
Synchronous circuits use the clock to sync their executions, decide when to read inputs/outputs.

↳ Heartbeat of a synchronous system.

How to Synchronize

Circuits usually synchronize to the **rising edge** of the clock.

- The transition from 0 to 1.
- Depending on the system, can instead sync on the **falling edge**.



Clock Multipliers

We know that CPU and memory operate at difference speeds. So how do they synchronize?

- One central clock used.
- Central clock is as slow as the slowest component.
- Faster components use a **clock multiplier**.

A clock multiplier multiplies the central clock frequency so that a component has many **internal cycles** for a single clock cycle of the *entire system*.

Note: this is simply a technicality of implementation. Generally, we still discuss speeds based on frequency the CPU experiences. Our old metrics still work as they always have.

Outline

Circuits that Remember

Sometimes values on a wire (i.e. a bit) cannot be maintained indefinitely on that wire. Values must change.

- Computer memory is circuits which *remember*.
- Circuits implement memory but are also used within other circuits to hold state.
  - ↳ Modular design.

Flip-flop: a circuit which implements a single bit of memory.

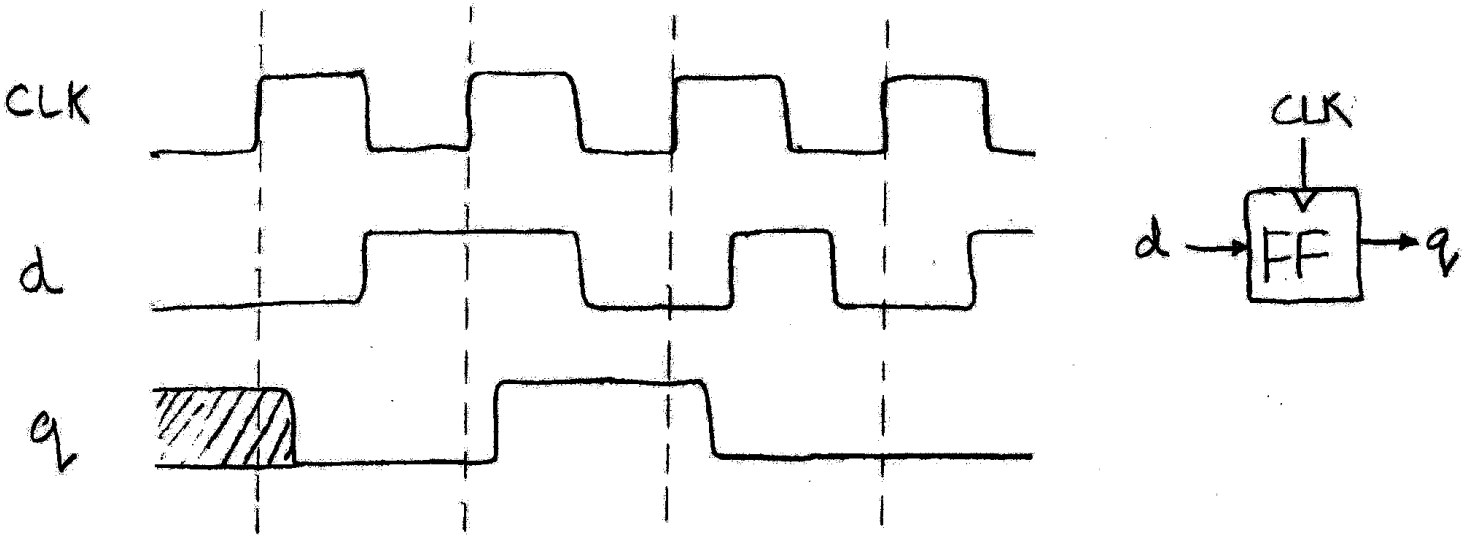
- ↳ All flip-flops based on a simple design: inputs, combined with current state, give next state.
- ↳ Essentially, the implementation of static RAM (SRAM).

Register: a storage for multiple bits of memory.

Edge-Triggered Flip-Flop

A flip-flop which looks at its input on the edge of clock.

- Rising edge or positive edge (usually), or
- Falling edge or negative edge.



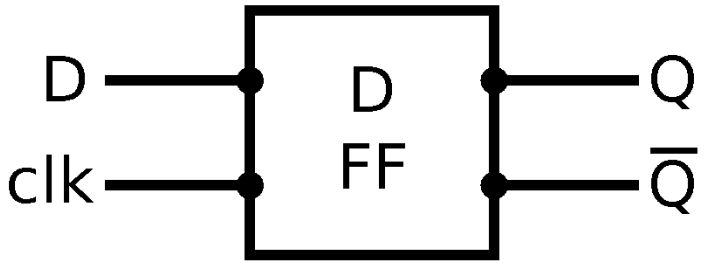
This is a *delay* flip-flop

D Flip-Flop

Delay flip-flop: takes input and, with some delay, sets output equal to the input.

- Simplest (conceptually) flip-flop.
- Requires constant updating to maintain state.
- Grabs input on rising edge and outputs that until next clock cycle.
- Current state does not affect next state.

D	Q	Q_{next}
0	-	0
1	-	1



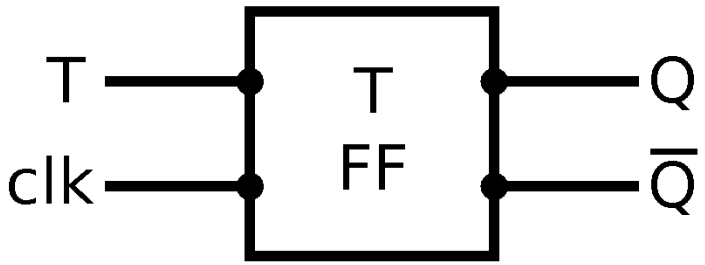
Flip-flops usually produce next state and negation of next state simultaneously.

T Flip-Flop

Toggle flip-flop: if input is 1, toggle current state.

- Uses current state to determine next state.
- $T \equiv 0 \Rightarrow$ "Hold". Next state is same as current.
- $T \equiv 1 \Rightarrow$ "Toggle". Next state is opposite of current.

T	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	0

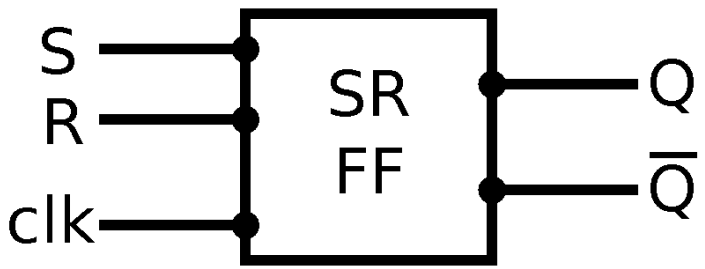


SR Flip-Flop

Set-Reset flip-flop

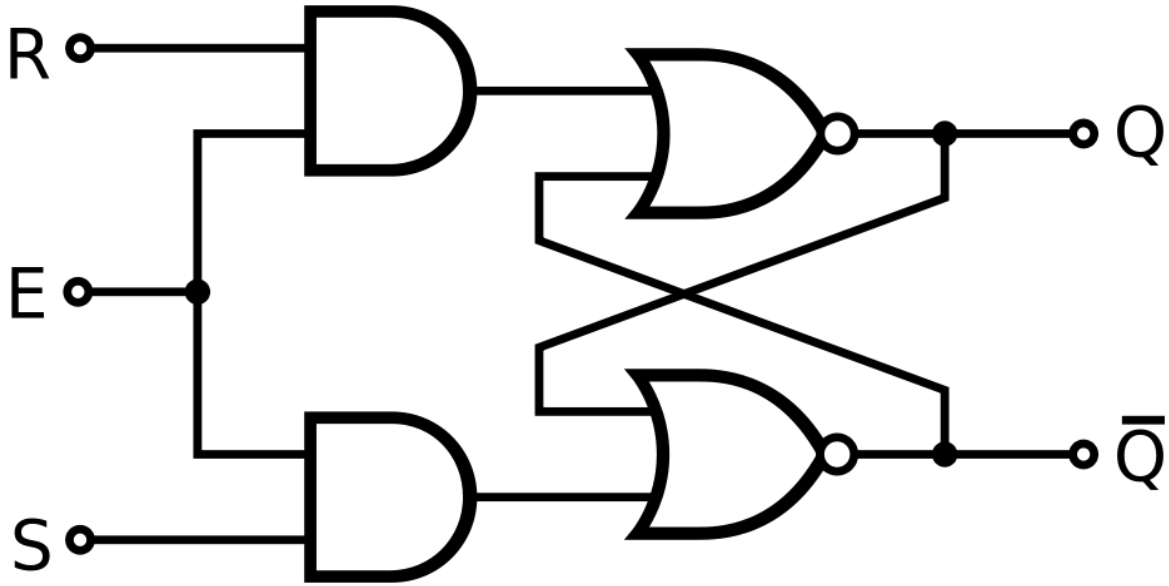
- Two inputs, S (set), R (reset), synchronized by a clock.
- $S \equiv 1 \Rightarrow$ "Set". Next state is 1.
- $R \equiv 1 \Rightarrow$ "Reset". Next state is 0.
- $S \equiv 0 \wedge R \equiv 0 \Rightarrow$ "Hold".

S	R	Q	Q_{next}
0	0	-	Q
0	1	-	0
1	0	-	1
1	1	-	-



Can **not** have both S and R set to 1...

SR Technicalities



E "enable" $\iff$ clock

$$S \equiv R \equiv E \equiv 1 \implies \overline{1 + \overline{Q}} \equiv 0 \equiv \overline{1 + Q} \implies Q \equiv \overline{Q} \; ???$$

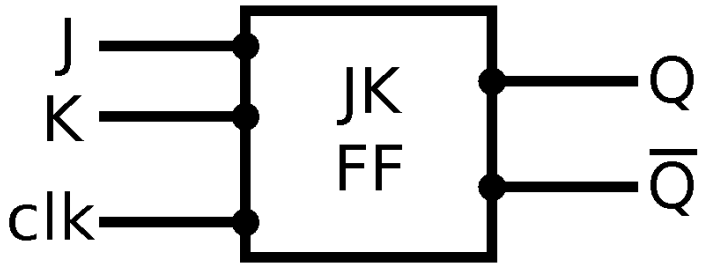
We get undefined behaviour. This is weird and can destabilize the system.

JK Flip-Flop

JK flip-flop

- Two inputs, J (set), K (reset), synchronized by a clock.
- Same as SR except with toggle.
- $J \equiv 1 \wedge K \equiv 1 \Rightarrow$ "Toggle".

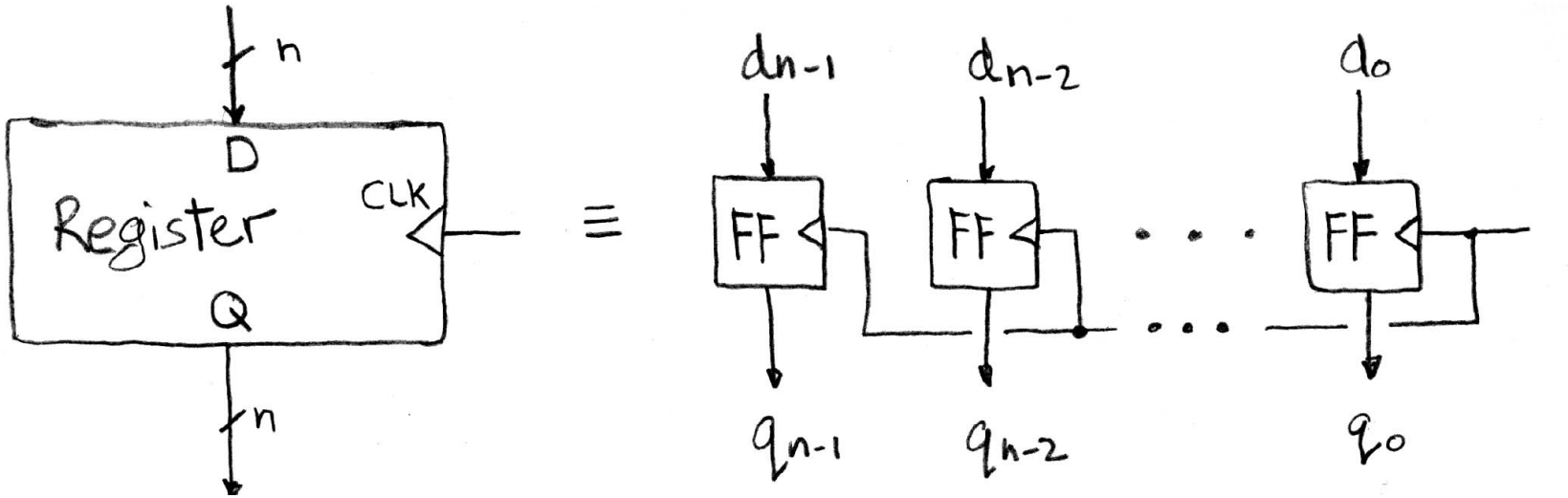
J	K	Q	Q_{next}
0	0	-	Q
0	1	-	0
1	0	-	1
1	1	-	$\overline{Q}$



Registers

A register is just a collection of flip-flops.

- Technically, this is a *shift register*.
- n-bits $\implies$ n flip-flops.
- Clock pulse connected to all flip-flops.
- Can be encoded using any type of flip-flop.

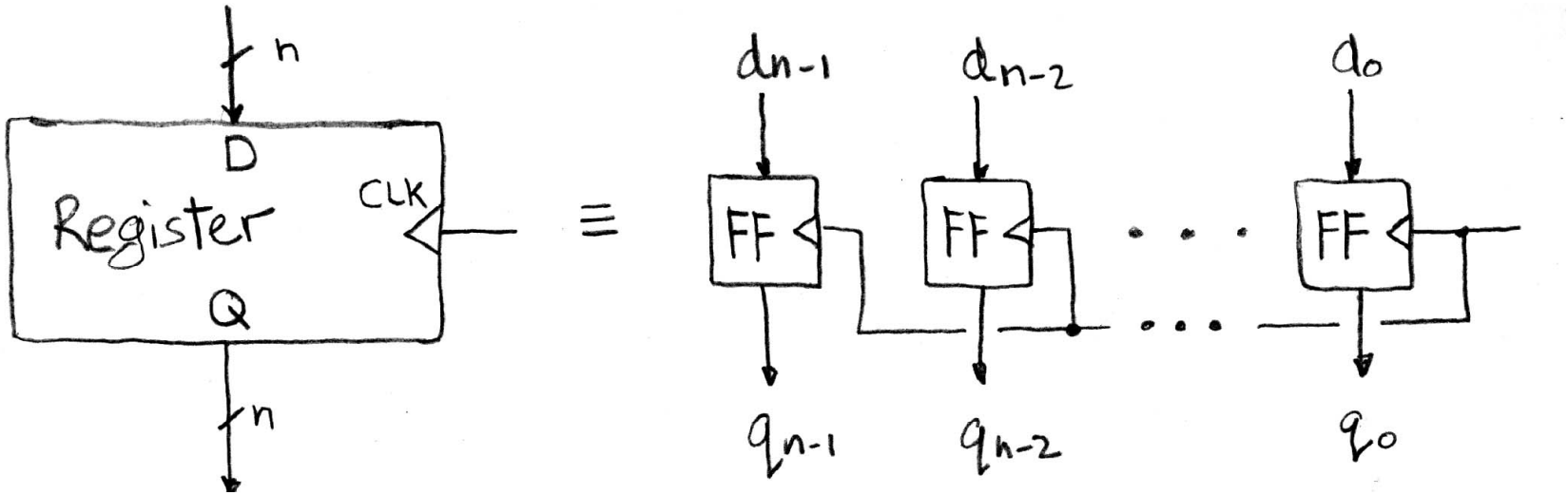


This example is a *parallel in, parallel out* register.

PIPO Registers

Parallel In, Parallel Out Register: All inputs bits come in in parallel, and output bits get output in parallel.

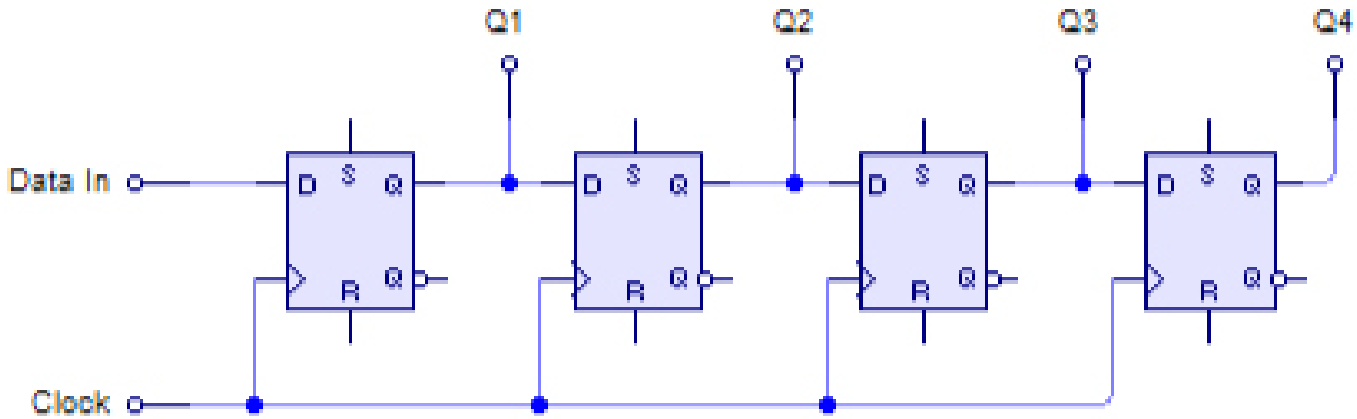
- Most common.
- Input/output of each flip-flop is independent.
- Can be encoded using any type of flip-flop.



SIPO Registers

Serial In, Parallel Out Register: One input bit at a time, output all bits at once.

- Input bit moves through chain of flip-flops.
- Transitions at each clock.



- This example uses D flip-flops.
- Sometimes it is useful to clear the entire register without waiting n cycles for n bits of data to shift out.
- Additional control signals can be used to set all flip-flops to 1 (S) or all flip-flops to 0 (R).

SISO/PISO Registers

Serial In, Serial Out Register: A linear chain of flip-flops.

- Output of one flip-flop is the input of the next.
- One input bit and one output bit.
- Kind of like a conveyor belt of bits.

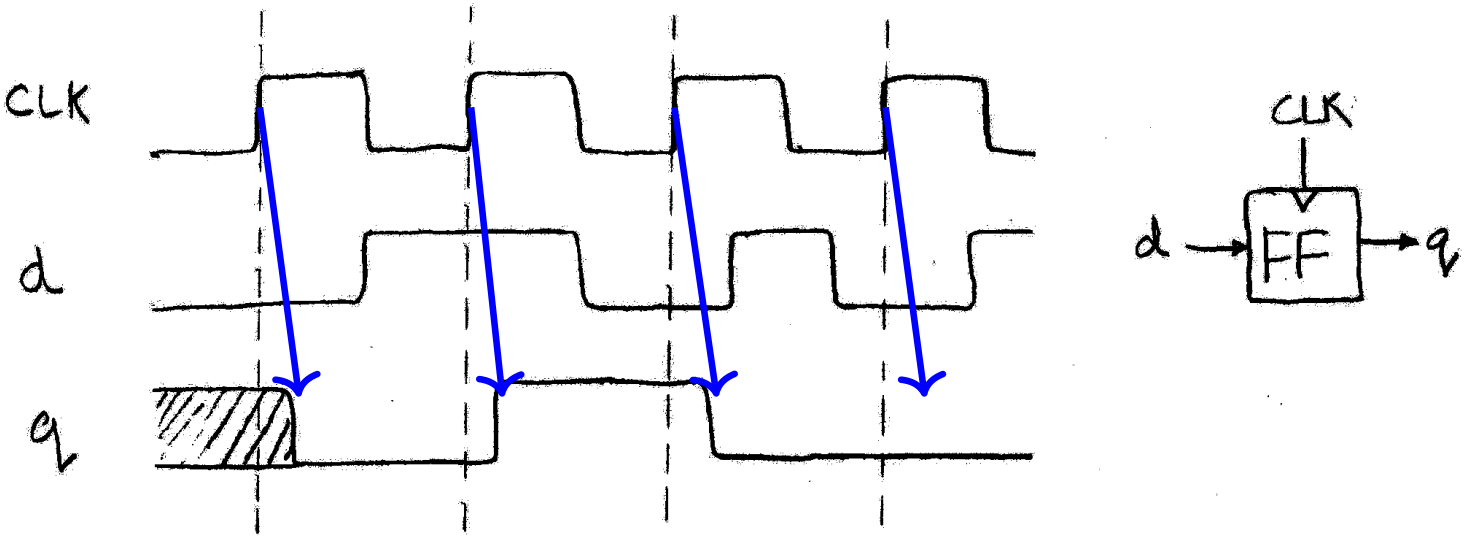
Parallel In, Serial Out Register: A linear chain of flip-flops + control circuits.

- Data *loaded* in parallel: n flip-flops load n bits at once.
- Data *output* in serial: Acts as SISO for output.
  - ↳ Output one bit at a time.
  - ↳ Bits are shifted one over on each output.
- Requires clock and additional write/shift control signal.

Timing a Flip-Flop

All gates/circuits introduce propagation delay.

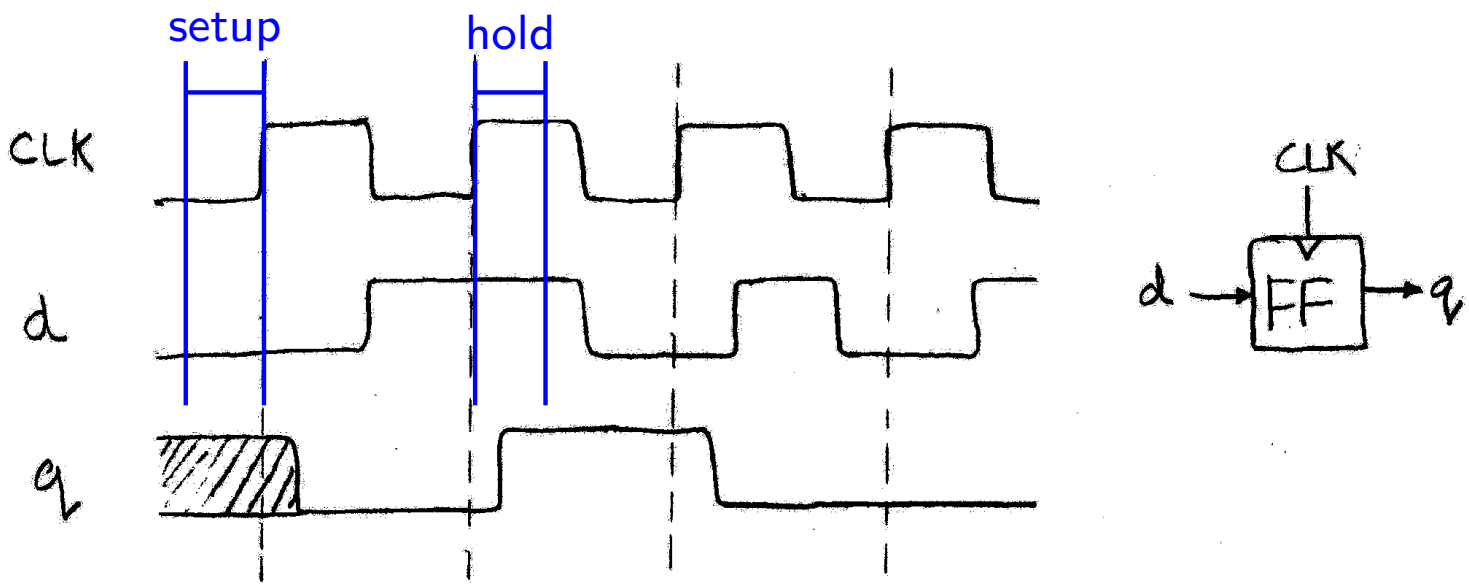
For flip-flops this propagation delay is called **clk-to-q delay**.



Timing a Flip-Flop: Data Stability

Input to a flip-flop must have a stable value around the rising edge of the clock.

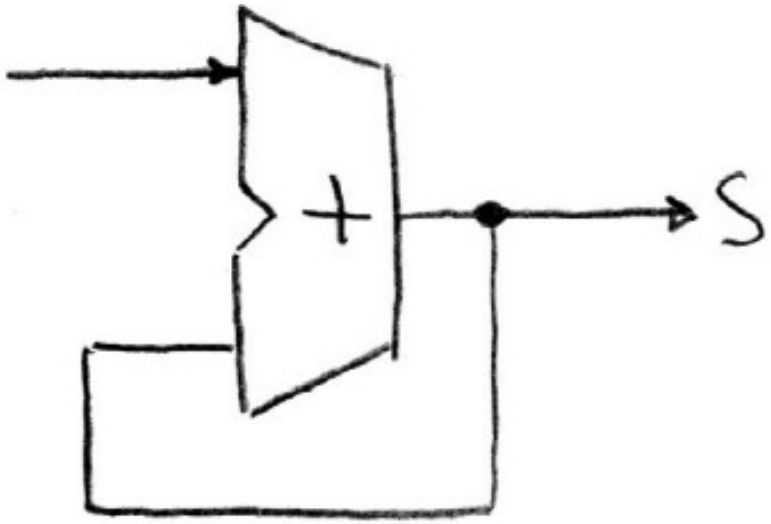
- Before the rising edge: **setup time**.
- After the rising edge: **hold time**.



Despite how it's shown here, hold time is less than clk-to-q delay.

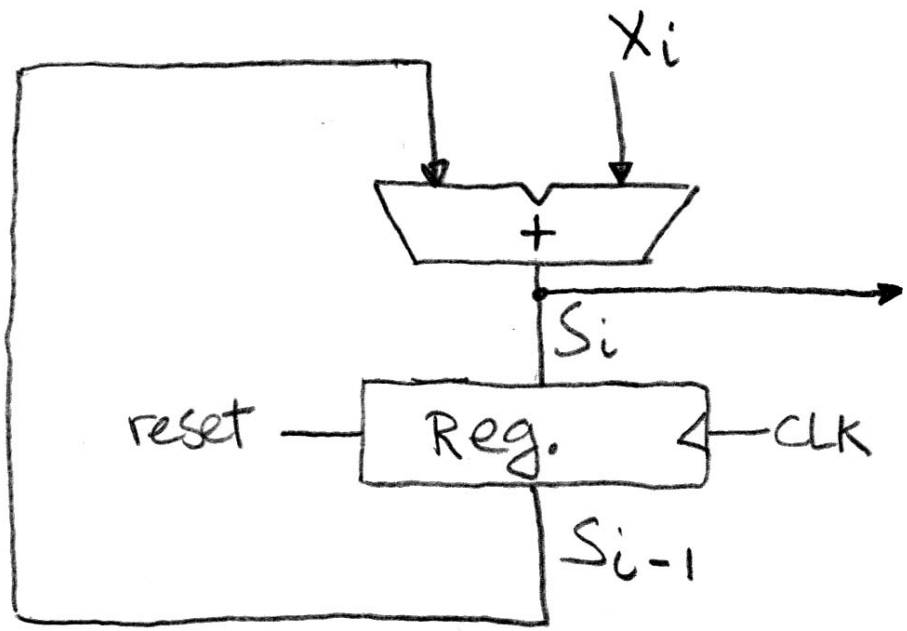
Putting it all Together: Accumulator

An accumulator: continually adds input value to its stored value.



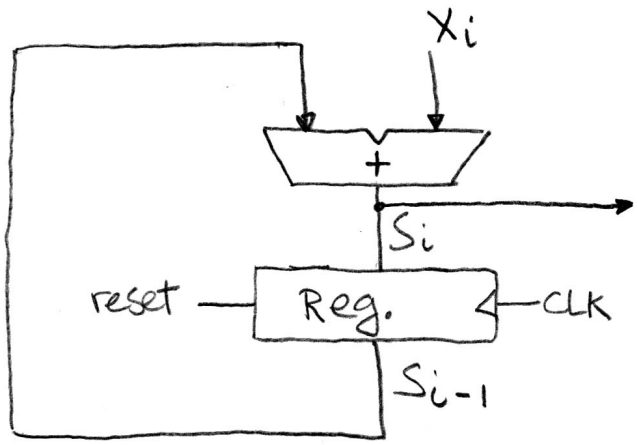
- This doesn't work.
- Would spin once per circuit's propagation delay, not once per input.
- Need clock to **synchronize** reading from input.

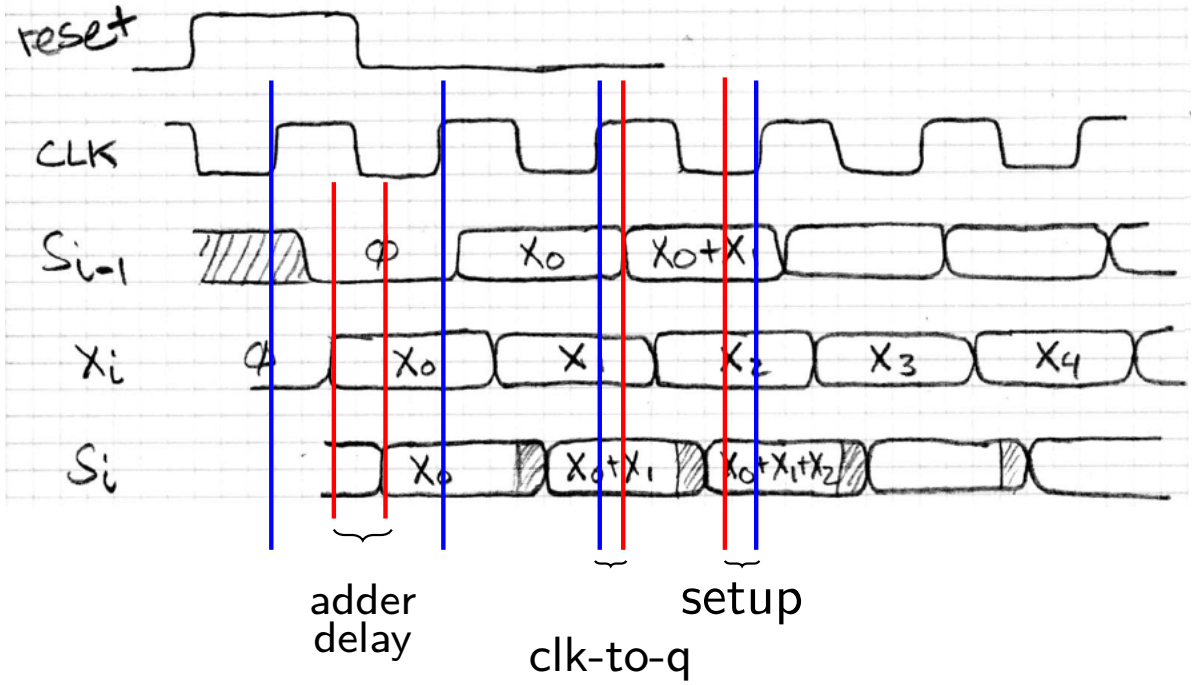
Clocked Accumulator



- Insert register to store output.
- Only need to clock the register, not the combinational circuit.
- Clock on register determines when output of circuit actually gets stored.

Timing the Accumulator

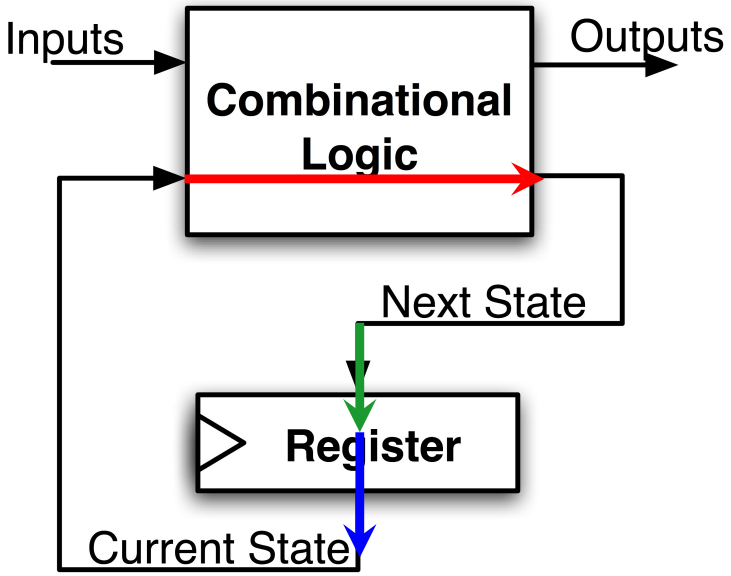




Clock must be slow enough to include:

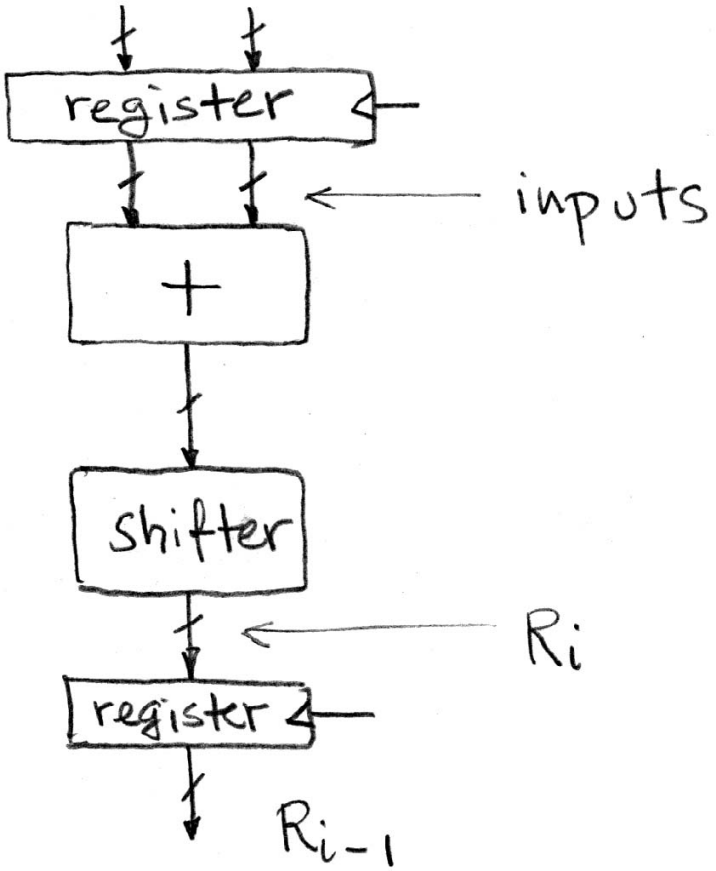
- Adder delay,
- Clk-to-q,
- Setup time.

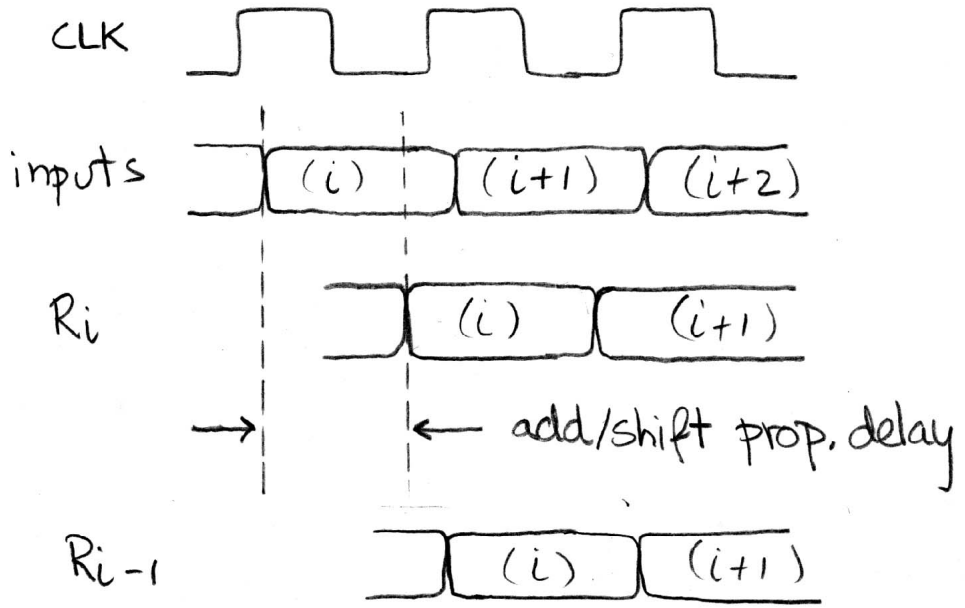
Synchronous Circuits: Clock Frequency



(Max Clock Freq.)
Min. Clock Period = Combinational Circuit Propagation Delay
+ Setup Time
+ Clk-To-Q

Pipeline for Performance (1/2)

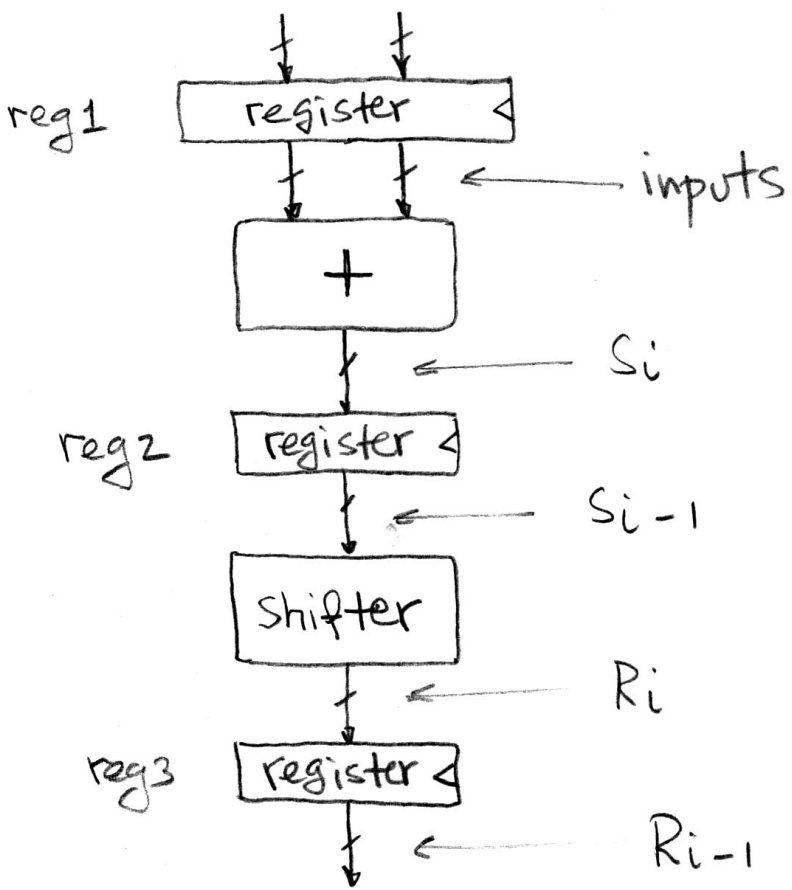


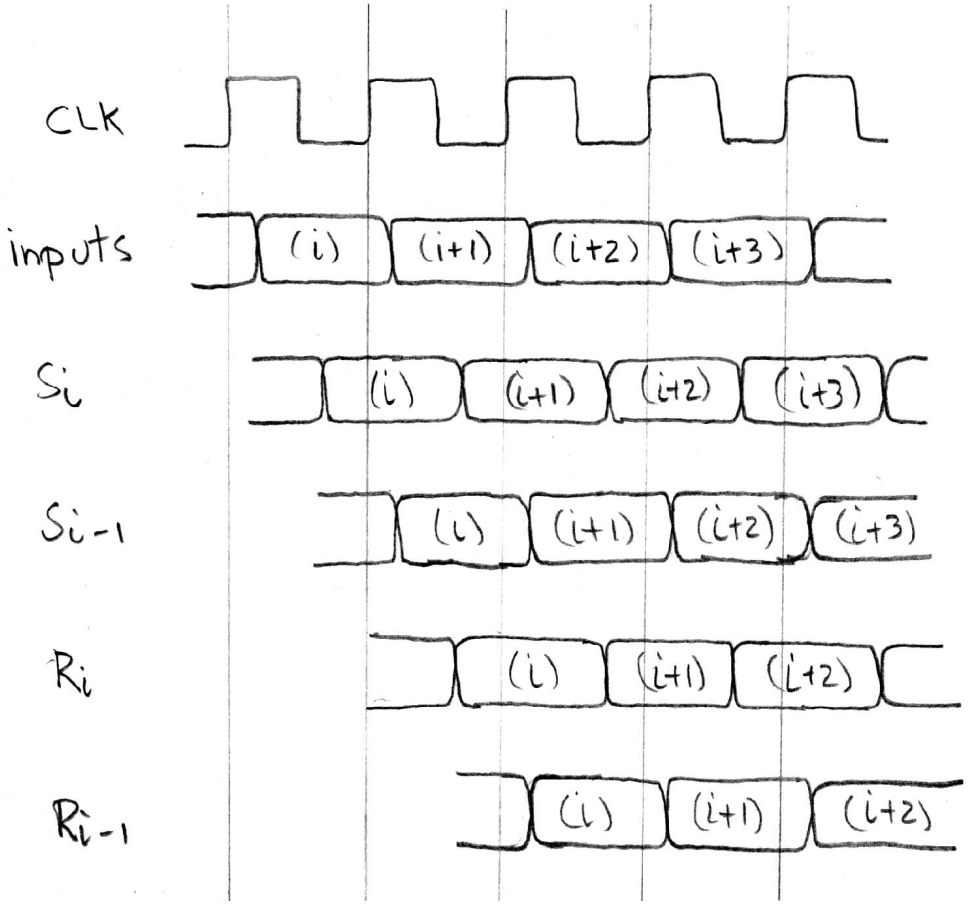


Delay of adder and shifter is very long.

- Forces clock cycle to be very long.
- Slows down other circuits in this synchronous system.

Pipeline for Performance (2/2)

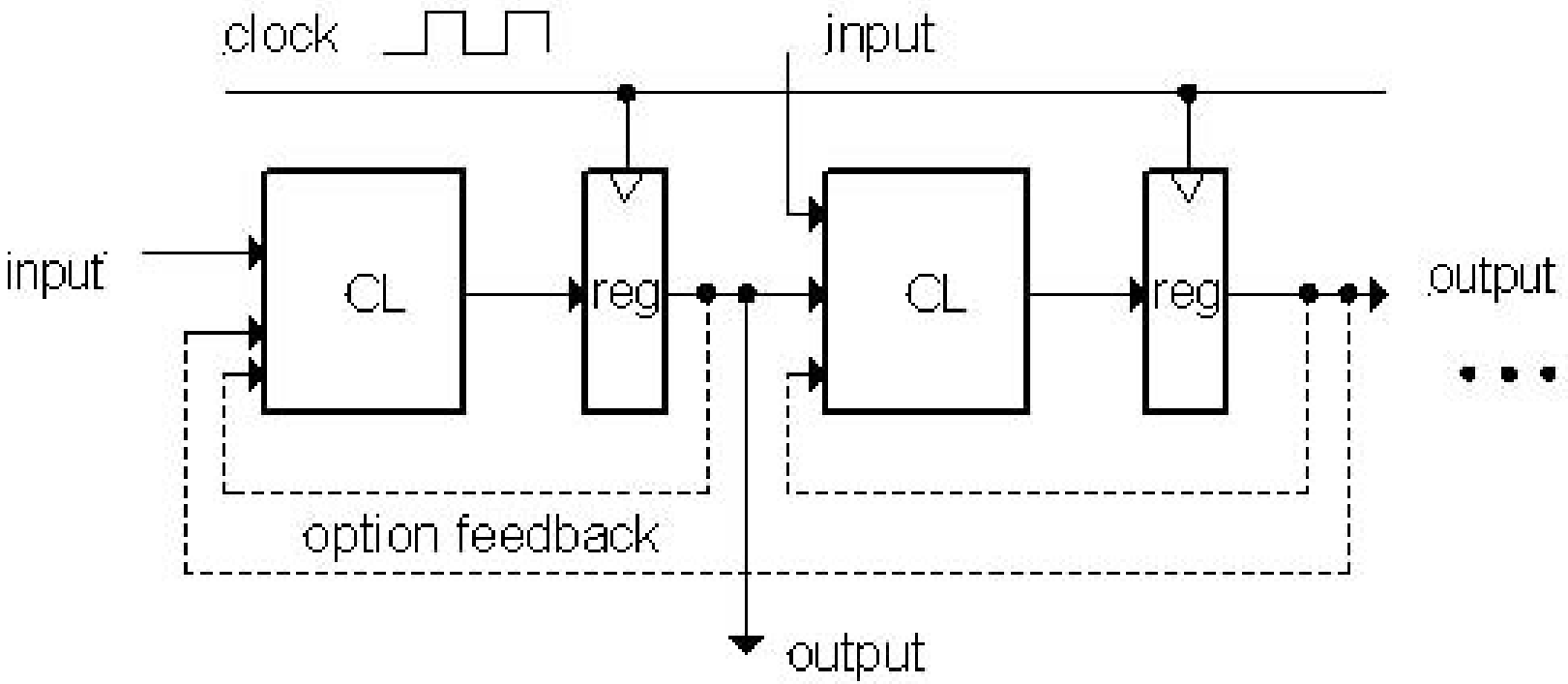




- Split add and shift into two different tasks.
- Insert register between to store results temporarily.
- **Increase clock frequency.**

General Synchronous Systems

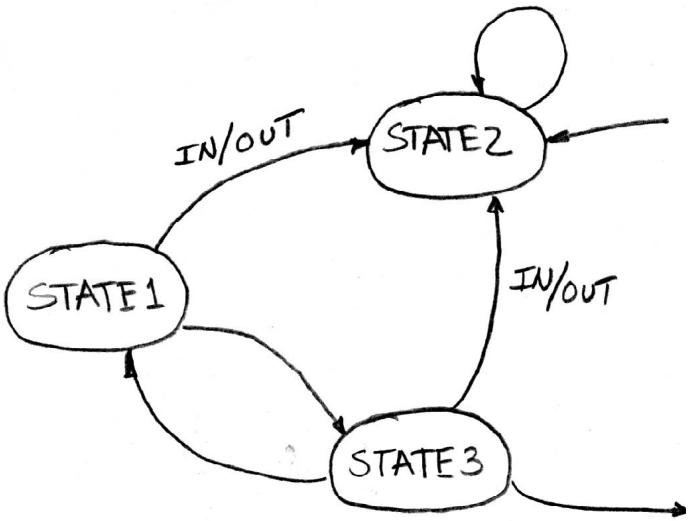
- All systems follow a general pattern:
- A chain of logic circuit blocks, separated by registers, controlled by a single clock.
- Foreshadowing for MIPS pipeline.



Outline

Finite State Machines: Introduction

We know **FSMs** from logic, formal languages, complexity.

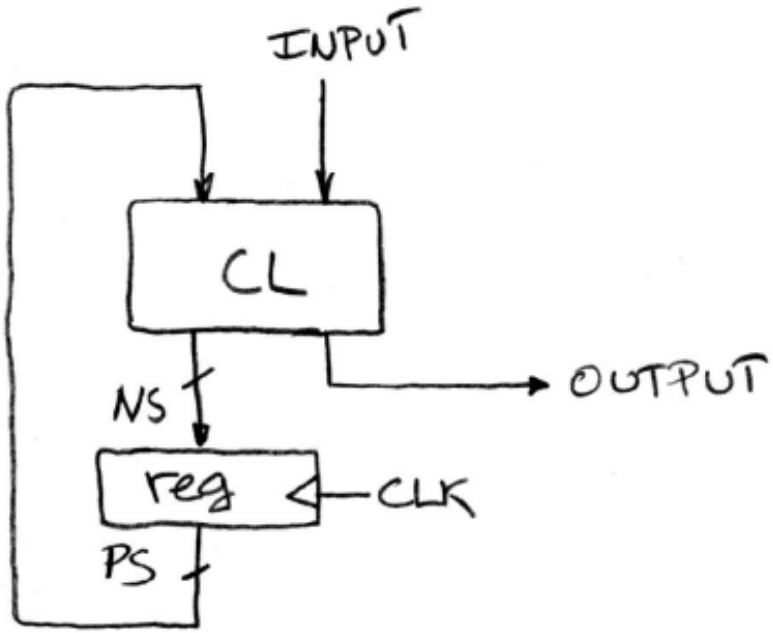


- Each state of the machine is a node.
- Inputs trigger change of state and an output.
- This is a Mealy machine: outputs occur on transitions.
- Moore machines are equivalent.
  - ↳ Output is based on current state.

Finite State Machines: As Circuits

FSMs have three components: state, input, output.

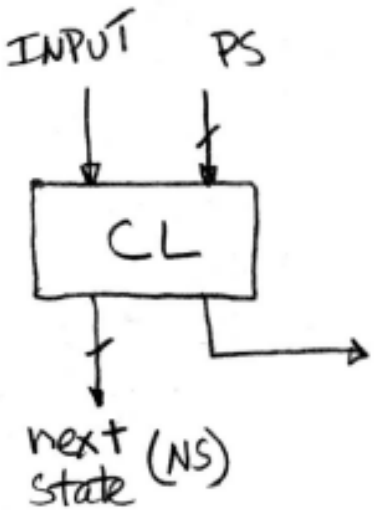
- Just like synchronous circuit.
- Registers, input bits, output bits.
- Clock controls when inputs read $\Rightarrow$ transitions.
- PS: present state, NS: next state.



Finite State Machines: Implementing The Logic

Next state and output is always just some Boolean combination of input and output. Use our normal 4-step process:

1. Build a truth table,
2. Get canonical form,
3. Simplify,
4. Draw circuit.



PS	In	NS	Out
00	0	01	0
00	1	01	1
01	-	10	0
10	0	10	1
10	1	11	1
11	0	10	0
11	1	11	1

$\Longleftrightarrow$ FSM state diagram

FSMs are Synchronous Systems are FSMs

- Essentially every synchronous system can be modelled by an FSM.
  - ↳ Would become absurdly large in most circumstances.
- A valid design strategy for integrated circuits and specialized hardware includes:
  1. Turn problem into FSM.
  2. Turn FSM into truth table.
  3. Turn truth table into circuit.
- Full Example: An elevator-controlling circuit.
  - ↳ `https://www.cs.princeton.edu/courses/archive/spr06/cos116/FSM_Tutorial.pdf`