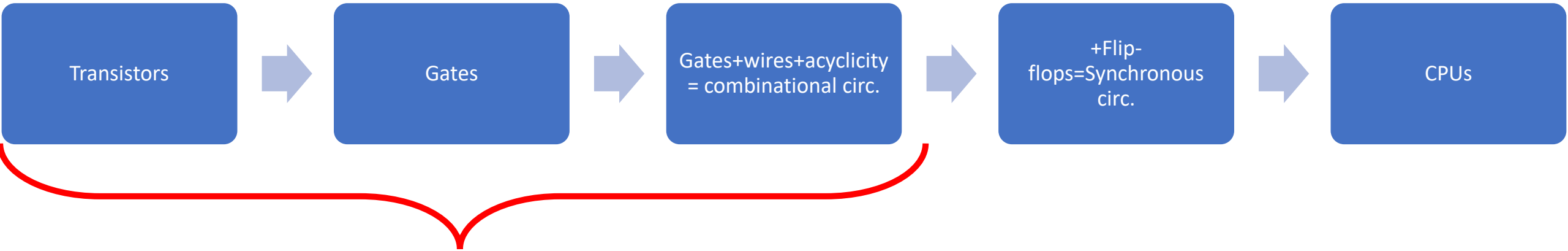
How to Clock Your Computer: Digital Logic Design 1/3

Moti Medina

Slides, slides material, and figures are taken from “Digital Logic Design : a rigorous approach” lecture slides and book by Guy Even and **M** (Chapters: 10, 11, 12, 14).

Our Journey: From Transistors to Computers



Today + what is the best we can do?!

Picture taken from here.

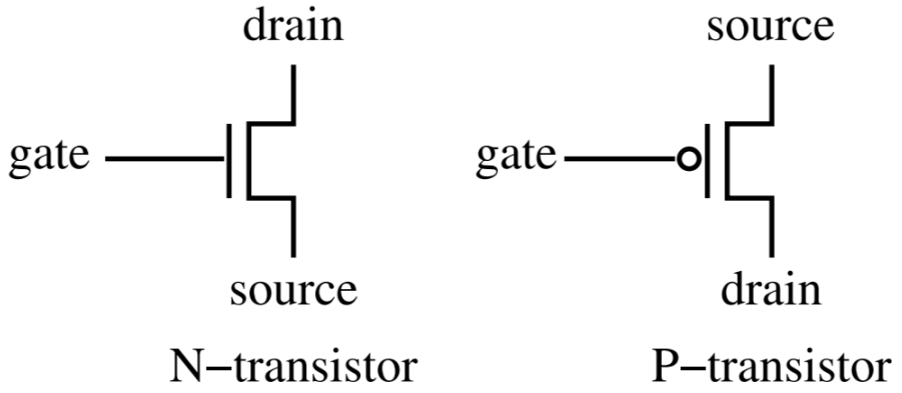
Today's plan

- **40 min lecture session (~pages 1-9, 15-16 in the "reading material").**
  - Transistors
  - From transistors to gates
  - Combinational gates
  - Wires
  - Combinational circuits
  - Equivalence to Boolean functions.

- **Rest of the lecture:**
  - Discuss, solve and present Question 1, Question 2 &3, or Question 4.
  - Questions appear on Page **17** in the reading material.

Transistors

- A lot to say about these devices.
- Let's keep it simple in the 1st 3 sessions.
- Let's view transistors as "switches"
- **N**-transistor:
  - "turn it on" → drain=source (i.e., conducting).
  - "turn off" → drain and source are not connected (i.e., not conducting)
- **P**-transistor:
  - "turn it off" → drain=source.
  - "turn on" → drain and source are not connected.
- "Turn on/off" = depends on the voltage at the gate.

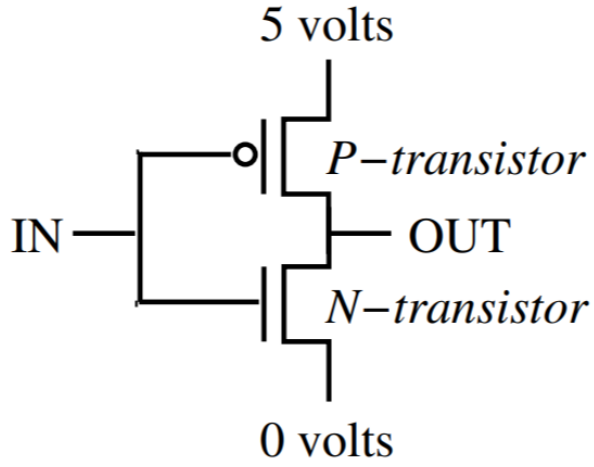


Inputs: gate & source

Output: drain

From Transistors to Gates

- How does an inverter look like?
  - $\forall b \in \{0,1\}: INV(x) = 1 - x$
- Every gate has its propagation delay
  - t_{pd}
  - Time from stable logical inputs to stable logical outputs.
  - There are more parameters…
- Gates are “analog”
  - We interpret voltages as 0 and 1 according to some thresholds.
  - There are noises in any system.
  - → Thresholds has to be separate
  - → there is a “non-logical” area.
  - More about this later in the course.



If IN = *low*, then:

- P-transistor is conducting
- N-transistor is not conducting

$\Rightarrow v(OUT) = high$

If IN = *high*, then:

- P-transistor is not conducting
- N-transistor is conducting

$\Rightarrow v(OUT) = low$

Combinational Gates

- Computes a simple Boolean function
  - AND, OR, XOR, NXOR, NOT,…
  - Takes t_{pd} time so that the output equal to the stable inputs.
  - What happens when the inputs stop being stable?
    - t_{cont} time after that the gate's output becomes "garbage"
- Let's throw in MUX as a gate as well.
- All our gates have a single output.
- Inputs & outputs of a gate aka
  - Terminals, ports, and pins.

The outputs become stable at most t_{pd} time units after the inputs become stable. The outputs remain stable at least t_{cont} time units after the inputs become instable.

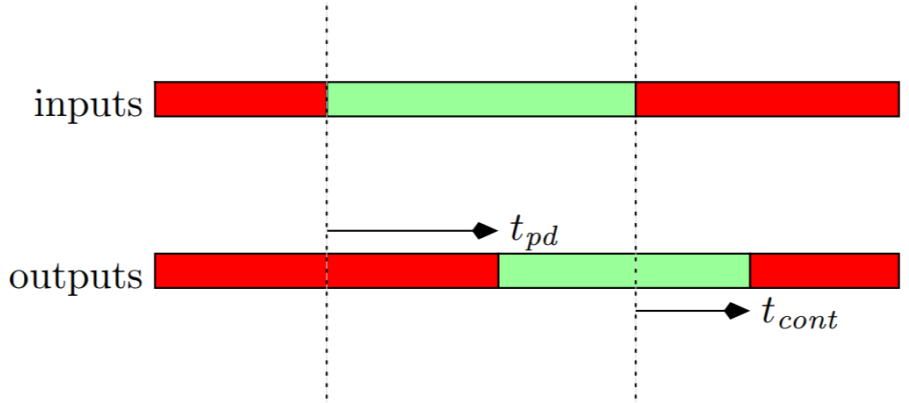


Figure: The x-axis corresponds to time. The red segments signify that the signal is not guaranteed to be logical; the green segments signify that the signal is guaranteed to be stable.

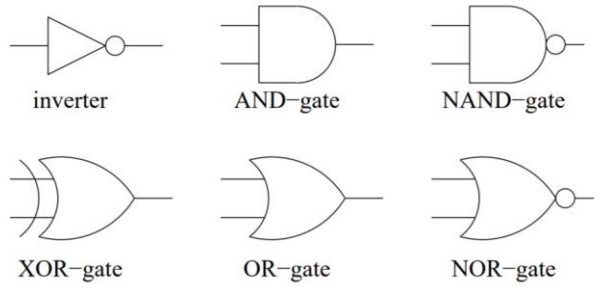


Figure 11.6: Symbols of common gates. Inputs are on the left side, outputs are on the right side.

Combinational Gates

- All out gates have a single output.
- Inputs & outputs of a gate aka
  - Terminals, ports, and pins.
- Fan-in of a gate **=**
  - number of input ports **=**
  - number of bits in the gate's Boolean function **=**
  - in-degree of a node that represents the gate.
- Our "basic" gates have fan-in 1-3.
- Let's add input and output gates
  - Get the "inputs" from the outer world and feed the output back to it.
    - Fan-in is 0-3

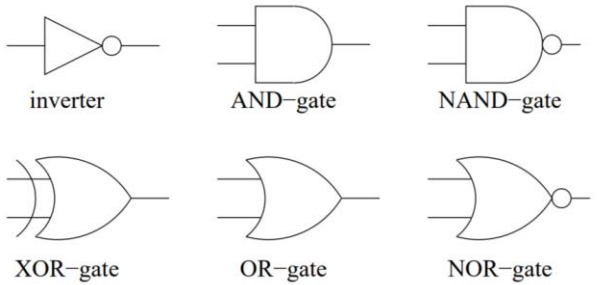


Figure 11.6: Symbols of common gates. Inputs are on the left side, outputs are on the right side.

Wires

- Simply a connection between two terminals.
- Fan-out of a gate = number of input ports it feeds.

Combinational Circuits

- **Wires "have direction": from an output port to an input port.**
- **Each input port is fed by a single output port.**
- Map gates to vertices, wires to arcs = Directed (labeled) Graph.
- **Directed graph is acyclic (DAG) = Combinational Circuit.**

The combinational circuit $C = (G, \pi)$ is called a Half-Adder.

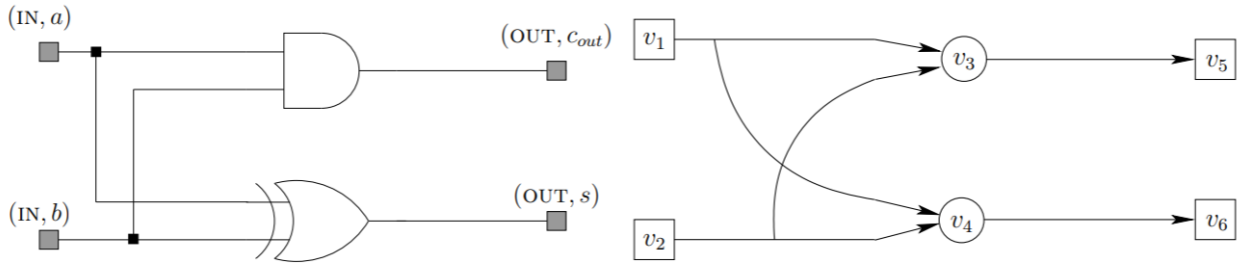


Figure: A Half-Adder combinational circuit and its matching DAG.

The set of the combinational gates in this example is $\Gamma = \{\text{AND}, \text{XOR}\}$. The labeling function $\pi : V \to \Gamma \cup IO$ is as follows.

$$\pi(1) = (\text{IN}, a), \qquad \pi(2) = (\text{IN}, b),$$
$$\pi(3) = \text{AND}, \qquad \pi(4) = \text{XOR},$$
$$\pi(5) = (\text{OUT}, c_{out}), \qquad \pi(6) = (\text{OUT}, s)\,.$$

Bad Circuits

Can you explain why these are not valid combinational circuits?

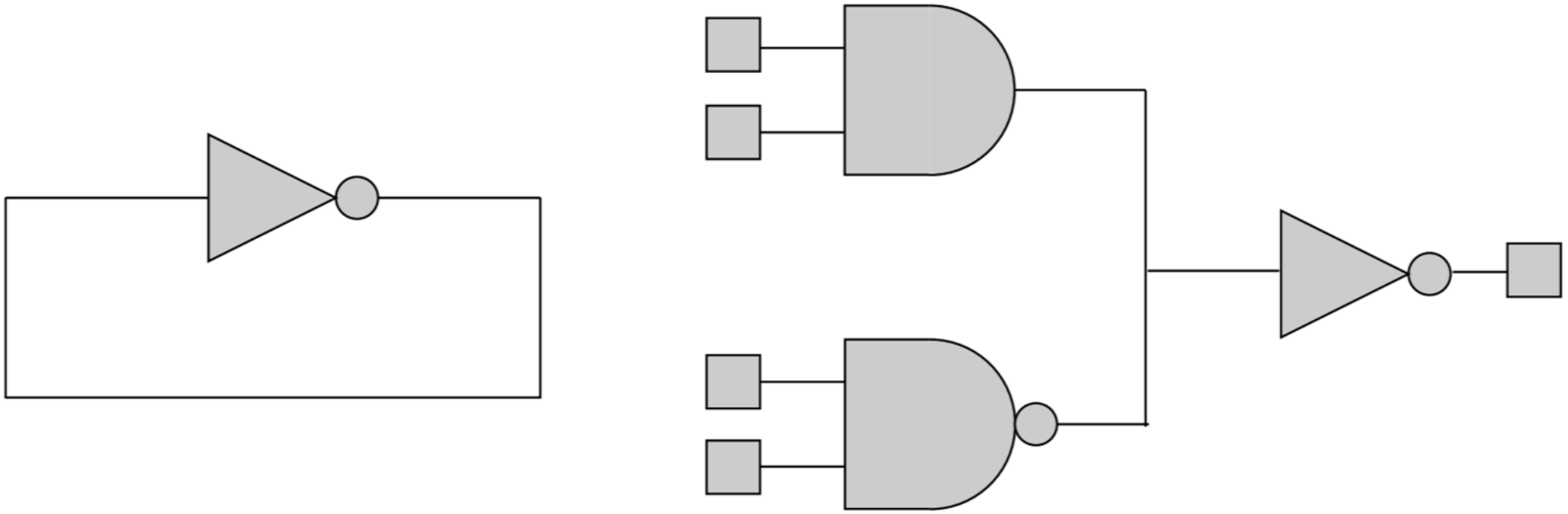


Figure: Two examples of non-combinational circuits.

Comb. Circuits = Boolean function

- Given stable input to a Combinational Circuit C (or its graph representation G_C), we can:
  1. Simulate the circuit in linear time (w.r.t. to the size of G_C),
  2. Analyze the delay of C in linear time.
- How? **Topological sorting**.

- Every Combinational circuit C implements a Boolean function.
  - Given that we can simulate, this argument is obvious.
  - Remove the MUX from our "basic gates" to make things easier (i.e., fan-in ≤ 2).

- For every Boolean function $f: \{0,1\}^n \to \{0,1\}$ there is a Comb. Circuit C that implements f
  - That is, $\forall x \in \{0,1\}^n: \mathrm{SIMUL}(C, x) = f(x)$.
  - Question #1 on Page 17.

Complexity Measures: Cost and Delay

- Each gate has its cost.
- We are interested in the asymptotic behavior of our measures.
- Basic gates are of constant cost we treat them as cost=1.
- **The <u>cost</u> of a Comb. Circ. C is the number of its gates.**

- Each gate has its propagation delay.
- Again, we are interested in the asymptotic behavior of our measures.
- Basic gates are of constant delay we treat them as t_{pd}=1.
- **The <u>delay</u> of a Comb. Circ. C is the length of the longest input to output path in C.**


And Now For Something
Completely Different

Lower Bounds: Cost and Depth

- **Reminder I: The depth of a rooted tree with n leaves is $\geq \lceil \log_2 n \rceil$.**
- **Reminder II: The #nodes of every rooted tree with n leaves is $2n - 1$.**
- **Reminder III: Cost of every gate is 1 (input and output gates have 0 cost)**

Definition

Let $flip_i : \{0,1\}^n \to \{0,1\}^n$ be the Boolean function defined by $flip_i(\vec{x}) \stackrel{\triangle}{=} \vec{y}$, where

$$y_j \stackrel{\triangle}{=} \begin{cases} x_j & \text{if } j \neq i \\ \text{NOT}(x_j) & \text{if } i = j. \end{cases}$$

The Cone of a Boolean Function

Definition (Cone of a Boolean function)

The cone of a Boolean function $f : \{0,1\}^n \to \{0,1\}$ is defined by

$$cone(f) \stackrel{\triangle}{=} \{i : \exists \vec{v} \text{ such that } f(\vec{v}) \neq f(flip_i(\vec{v}))\}$$

Example

$cone(\text{XOR}) = \{1,2\}$.

We say that f depends on x_i if $i \in cone(f)$.

An Example

Consider the following Boolean function:

$$f(\vec{x}) = \begin{cases} 0 & \text{if } \sum_i x_i < 3 \\ 1 & \text{otherwise.} \end{cases}$$

Suppose that one reveals the input bits one by one. As soon as 3 ones are revealed, one can determine the value of $f(\vec{x})$. Nevertheless, the function $f(\vec{x})$ depends on all its inputs, and hence, $cone(f) = \{1, \ldots, n\}$.

Composition of Functions & Graphical Cone

Claim

If $g(\vec{x}) \triangleq B(f_1(\vec{x}), f_2(\vec{x}))$, *then*

$$\text{cone}(g) \subseteq \text{cone}(f_1) \cup \text{cone}(f_2)\,.$$

Definition

Let $G = (V, E)$ denote a DAG. The graphical cone of a vertex $v \in V$ is defined by

$$cone_G(v) \triangleq \{u \in V : deg_{in}(u) = 0 \text{ and } \exists \text{path from } u \text{ to } v\}.$$

In a combinational circuit, every source is an input gate. This means that the graphical cone of v equals the set of input gates from which there exists a path to v.

Functional Cone $\subseteq$ Graphical Cone

Claim

Let $H = (V, E, \pi)$ denote a combinational circuit. Let $G = DG(H)$. For every vertex $v \in V$, the following holds:

$$\text{cone}(f_v) \subseteq \text{cone}_G(v) \,.$$

Namely, if f_v depends on x_i, then the input gate u that feeds the input x_i must be in the graphical cone of v.

“Hidden” Rooted Trees

Claim

Let $G = (V, E)$ denote a DAG. For every $v \in V$, there exist $U \subseteq V$ and $F \subseteq E$ such that:

1. *$T = (U, F)$ is a rooted tree;*
2. *v is the root of T;*
3. *$\text{cone}_G(v)$ equals the set of leaves of (U, F).*

The sets U and F are constructed as follows.

1. Initialize $F = \emptyset$ and $U = \emptyset$.
2. For every source u in $cone_G(v)$ do
   (a) Find a path p_u from u to v.
   (b) Let q_u denote the prefix of p_u, the vertices and edges of which are not contained in U or F.
   (c) Add the edges of q_u to F, and add the vertices of q_u to U.

Putting things together! LB on Cost

Theorem (Linear Cost Lower Bound Theorem)

Let $H = (V, E, \pi)$ denote a combinational circuit. If the fan-in of every gate in H is at most 2, then

$$c(H) \geq \max_{v \in V} |\text{cone}(f_v)| - 1.$$

Corollary

Let C_n denote a combinational circuit that implements OR_n. Then

$$c(C_n) \geq n - 1.$$

Lower Bound on Delay

Theorem (Logarithmic Delay Lower Bound Theorem)

Let $H = (V, E, \pi)$ denote a combinational circuit. If the fan-in of every gate in H is at most 2, then

$$t_{pd}(H) \geq \max_{v \in V} \log_2 |\text{cone}(f_v)|.$$

Corollary

Let C_n denote a combinational circuit that implements OR_n. Let 2 denote the maximum fan-in of a gate in C_n. Then

$$t_{pd}(C_n) \geq \lceil \log_2 n \rceil .$$

Effect of fan-in$\leq k$ on Lower Bounds

Theorem (Logarithmic Delay Lower Bound Theorem)

Let $H = (V, E, \pi)$ denote a combinational circuit. If the fan-in of every gate in H is at most k, then

$$t_{pd}(H) \geq \max_{v \in V} \log_k |\text{cone}(f_v)|.$$

Corollary

Let C_n denote a combinational circuit that implements OR_n. Let k denote the maximum fan-in of a gate in C_n. Then

$$t_{pd}(C_n) \geq \lceil \log_k n \rceil .$$

See you on Monday!

Enjoy the discussion ☺

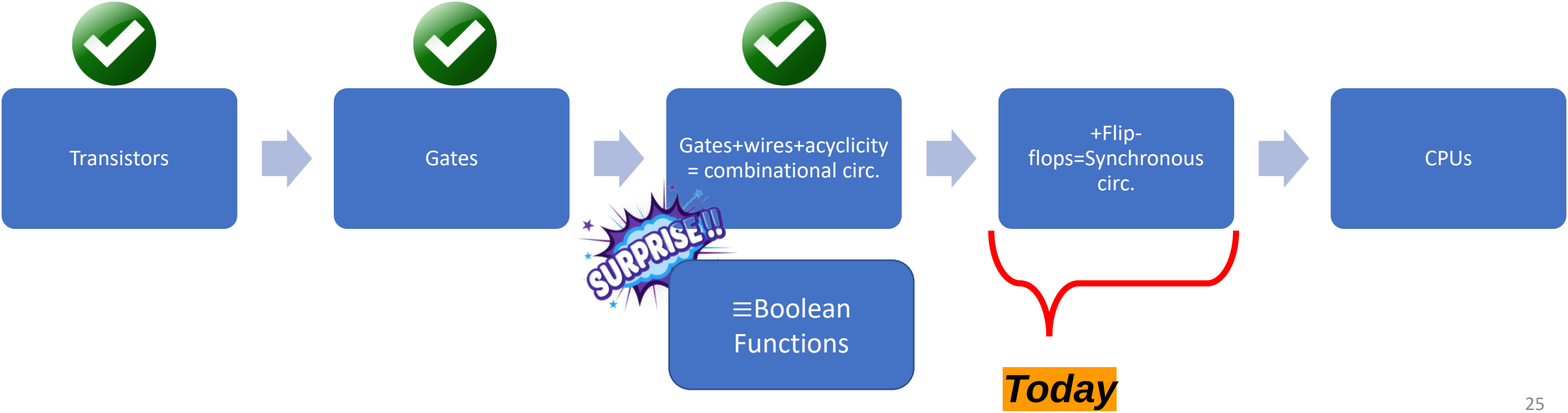
How to Clock Your Computer: Digital Logic Design 2/3

Moti Medina

Slides, slides material, and figures are taken from “Digital Logic Design : a rigorous approach” lecture slides and book by Guy Even and **M** (Chapters: 17-20).

Our Journey: From Transistors to Computers

Picture taken from here.



Today's plan

- **40 min lecture session (~pages 9-17 in the "reading material").**
  - The clock (2 min)
  - Clock cycles (2 min)
  - Flip-flop and Clock enabled Flip-Flops (4 min)
  - The Zero-delay model (2 min)
  - Example: Sequential XOR. (4 min)
  - Canonic form of a synch. Circuit. (5 min)
  - FSMs (4 min)
  - Analysis and Synthesis (4 min)
  - Example 1: analysis of a counter (4 min)
  - Example 2: analysis of shift register (4 mins)
  - Example 3: synthesis of a 2-state FSM (6 min)

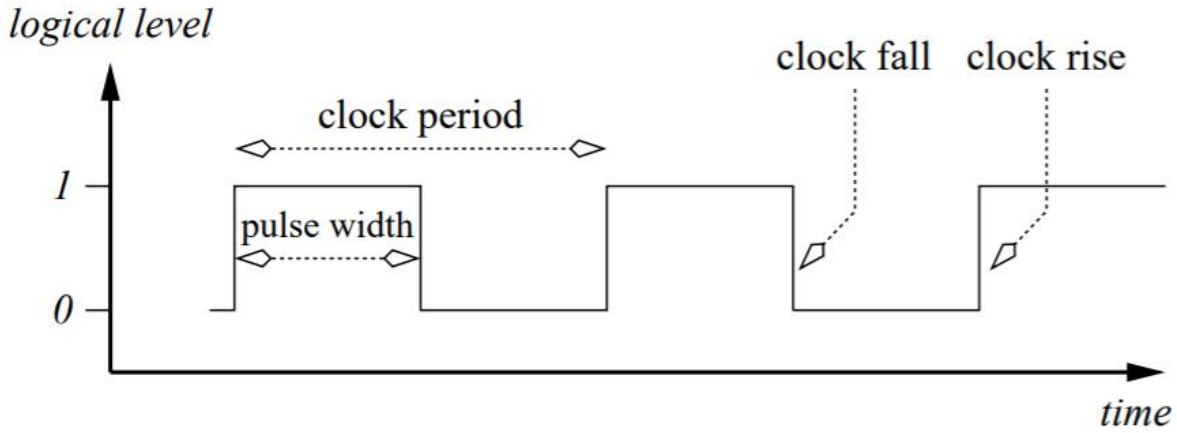
- **Rest of the lecture:**
  - Discuss, solve and present Question 5, Question 6, or Question 10, 12.
  - Questions appear on Page **17** in the reading material.

The Clock

the clock is generated by rectifying and amplifying a signal generated by special non-digital devices (e.g., crystal oscillators).

Definition

A clock is a periodic logical signal that oscillates instantaneously between logical one and logical zero. There are two instantaneous transitions in every clock period: (i) in the beginning of the clock period, the clock transitions instantaneously from zero to one; and (ii) at some time in the interior of the clock period, the clock transitions instantaneously from one to zero.



Clock Cycles

- A clock partitions time into discrete intervals.
- t_i - the starting time of the ith clock cycle.
- $[t_i, t_{i+1})$ -clock cycle i.
- Clock period $= t_{i+1} - t_i$.

Assumption

We assume that the clock period equals 1.

$$t_{i+1} = t_i + 1 \ .$$

Flip-flop

Definition

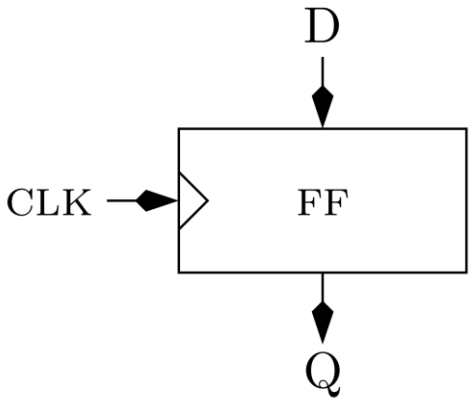
A flip-flop is defined as follows.

Inputs: Digital signals $D(t)$ and a clock CLK.

Output: A digital signal $Q(t)$.

Functionality:

$$Q(t+1) = D(t) \ .$$



t	$D[t]$	$Q[t]$
0	1	?
1	0	1
2	0	0
3	1	0
4	1	1

Clock-enabled Flip-Flops

Definition

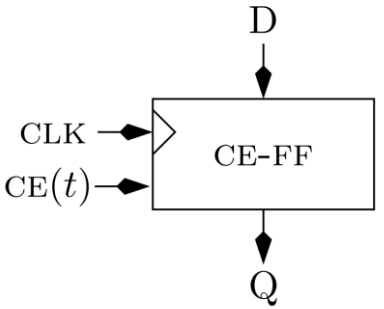
A clock enabled flip-flop is defined as follows.

Inputs: Digital signals $D(t), \mathrm{CE}(t)$ and a clock CLK.

Output: A digital signal $Q(t)$.

Functionality:

$$Q(t+1) = \begin{cases} D(t) & \text{if } \mathrm{CE}(t) = 1 \\ Q(t) & \text{if } \mathrm{CE}(t) = 0. \end{cases}$$



We refer to the input signal $\mathrm{CE}(t)$ as the clock-enable signal. Note that the input $\mathrm{CE}(t)$ indicates whether the flip-flop samples the input $D(t)$ or maintains its previous value.

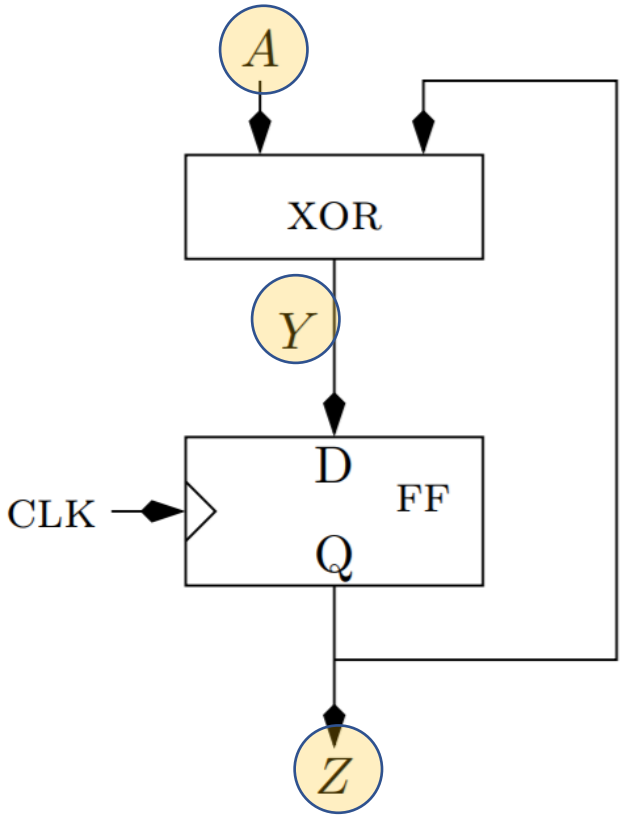
The Zero-delay Model

1. Transitions of all signals are instantaneous.
2. Combinational gates: $t_{pd} = t_{cont} = 0$.
3. Flip-flops satisfy:

$$Q(t+1) = D(t) \ .$$

4. Simplified model for specifying and simulating the functionality of circuits with flip-flops.
5. For a signal X, let X_i denote its value during the ith clock cycle.

Example: Sequential XOR



- How much a comb. circuit that imp. XOR_n costs at best? Delay?
- Sequential version has cost of $O(1)$! How can that be?

i	A_i	Y_i	Z_i
0	0	0	0
1	0	0	0
2	1	1	0
3	0	1	1
4	0	1	1
5	1	0	1
6	0	0	0
7	1	1	0
8	0	1	1

Canonic Form of a Synch. Circuit

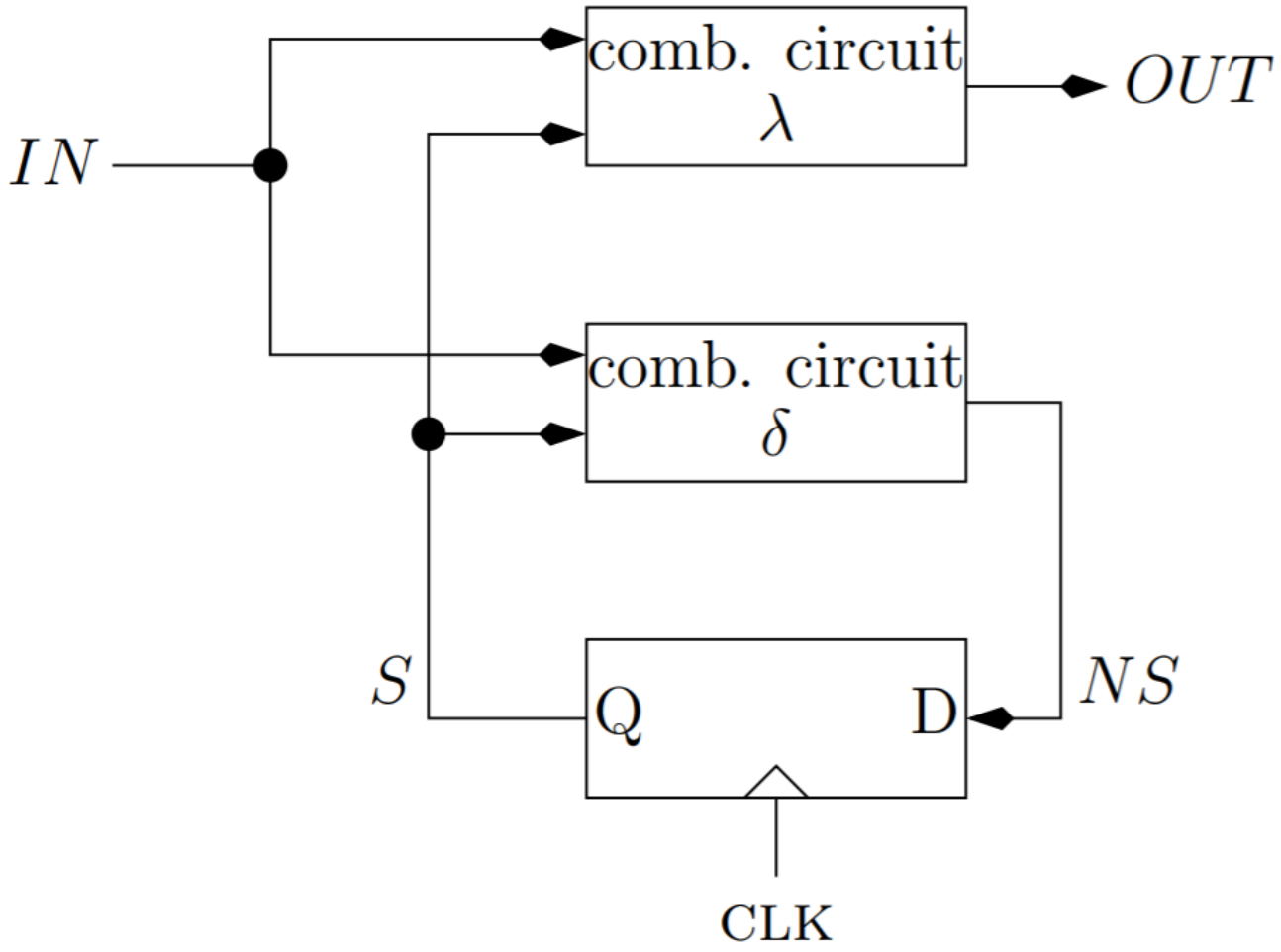


Figure: A synchronous circuit in canonic form.

Functionality (without proof):

- Logical value of a signal X during the ith clock cycle by X_i
- *Claim: For $i \geq 0$*
  - $S_i = NS_{i-1}$
  - $NS_i = \delta(IN_i, S_i)$
  - $OUT_i = \lambda(IN_i, S_i)$

- **Why every Synch. Circuit can be represented in this Canonic Form?**

FSMs

The functionality of a synchronous circuit in the canonic form is so important that it justifies a term called finite state machines.

Definition

A finite state machine (FSM) is a 6-tuple $\mathcal{A} = \langle Q, \Sigma, \Delta, \delta, \lambda, q_0 \rangle$, where

- Q is a set of states.
- Σ is the alphabet of the input.
- Δ is the alphabet of the output.
- $\delta : Q \times \Sigma \to Q$ is a transition function.
- $\lambda : Q \times \Sigma \to \Delta$ is an output function.
- $q_0 \in Q$ is an initial state.

What does an FSM do?

An FSM is an abstract machine that operates as follows. The input is a sequence $\{x_i\}_{i=0}^{n-1}$ of symbols over the alphabet Σ. The output is a sequence $\{y_i\}_{i=0}^{n-1}$ of symbols over the alphabet Δ. An FSM transitions through the sequence of states $\{q_i\}_{i=0}^{n}$. The state q_i is defined recursively as follows:

$$q_{i+1} \stackrel{\triangle}{=} \delta(q_i, x_i)$$

The output y_i is defined as follows:

$$y_i \stackrel{\triangle}{=} \lambda(q_i, x_i).$$

Definition

A finite state machine (FSM) is a 6-tuple $\mathcal{A} = \langle Q, \Sigma, \Delta, \delta, \lambda, q_0 \rangle$, where

- Q is a set of states.
- Σ is the alphabet of the input.
- Δ is the alphabet of the output.
- $\delta : Q \times \Sigma \to Q$ is a transition function.
- $\lambda : Q \times \Sigma \to \Delta$ is an output function.
- $q_0 \in Q$ is an initial state.

Analysis and Synthesis

Two tasks are often associated with synchronous circuits. These tasks are defined as follows.

1. Analysis: given a synchronous circuit C, describe its functionality by an FSM.
2. Synthesis: given an FSM $\mathcal{A}$, design a synchronous circuit C that implements $\mathcal{A}$.

$Synch.Circ. \Rightarrow FSM$

$FSM \Rightarrow Synch.Circ.$

Example 1: Analysis of a Counter

Definition

A *counter*(n) is defined as follows.

Inputs: a clock CLK.

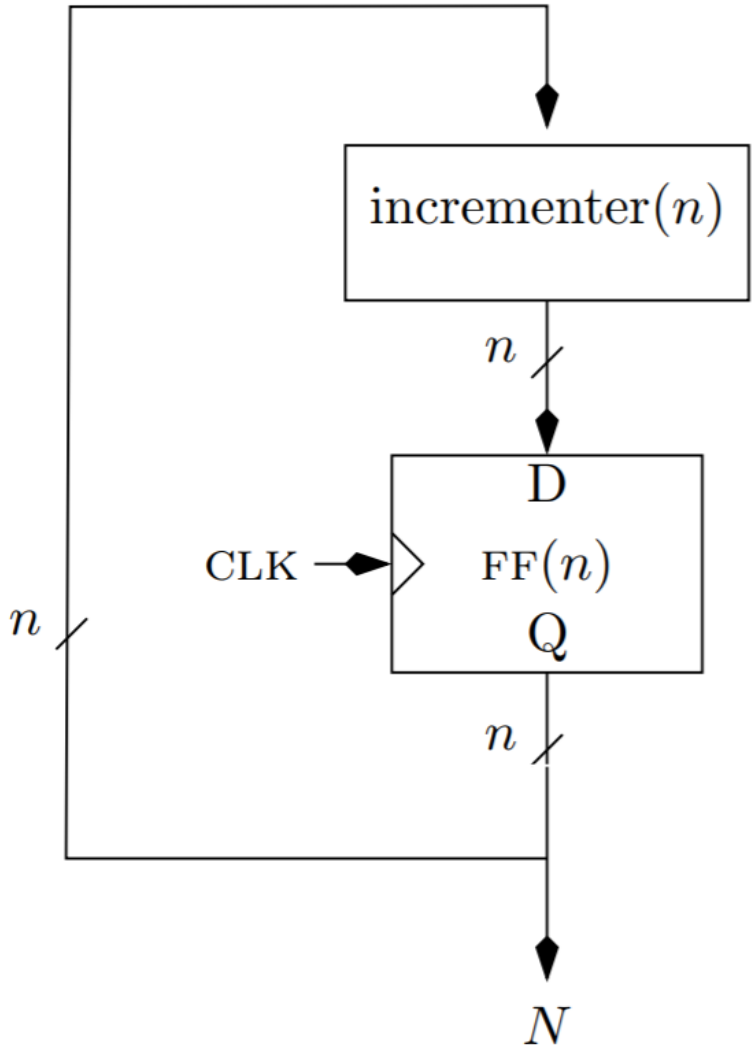
Output: $N \in \{0,1\}^n$.

Functionality:

$$\forall t \ : \ \langle N_t \rangle = t \pmod{2^n}$$

No input?! Input is "implied": it is the (missing) reset signal!

Counter Implementation



Definition

A *counter*(n) is defined as follows.

Inputs: a clock CLK.

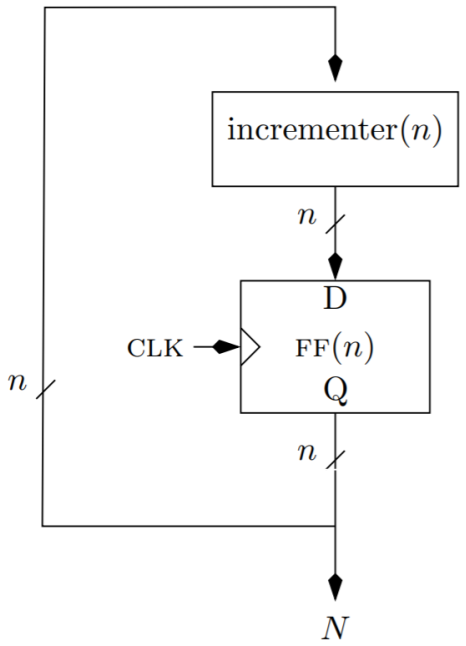
Output: $N \in \{0,1\}^n$.

Functionality:

$$\forall t \ : \ \langle N_t \rangle = t \pmod{2^n}$$

Figure: A synchronous circuit that implements a counter.

Counter Analysis



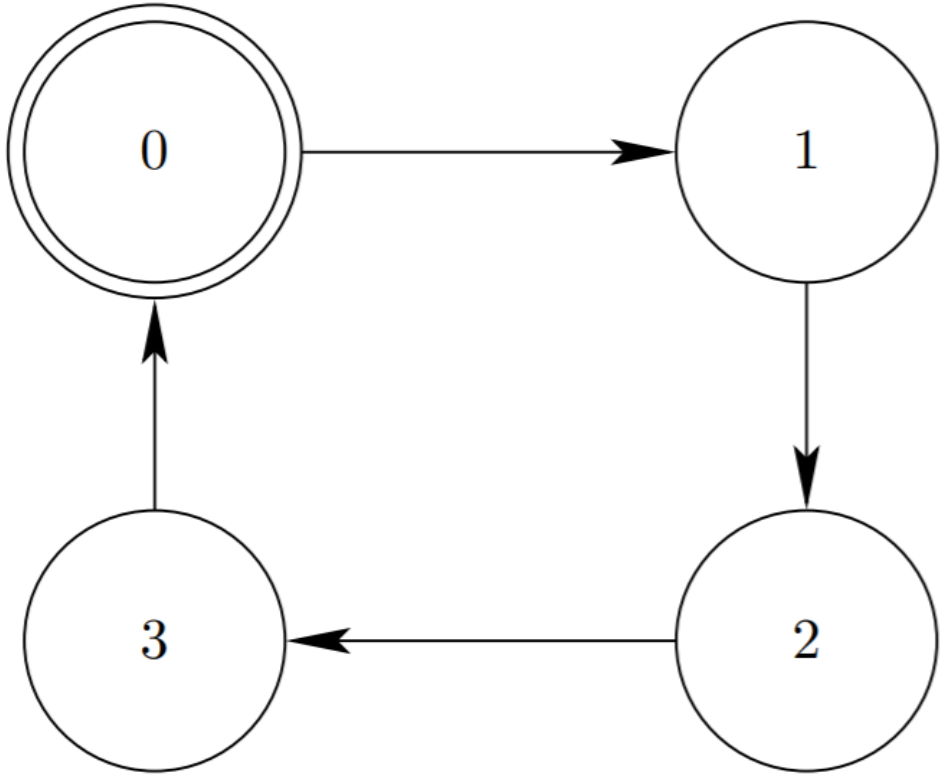


Figure: An FSM of a counter(2). The output always equals binary representation of the state from which the edge emanates.

Example 2: Analysis of Shift Register

Definition

A *shift register* of n bits is defined as follows.

Inputs: $D[0](t)$ and a clock CLK.

Output: $Q[n-1](t)$.

Functionality: $Q[n-1](t+n) = D[0](t)$.

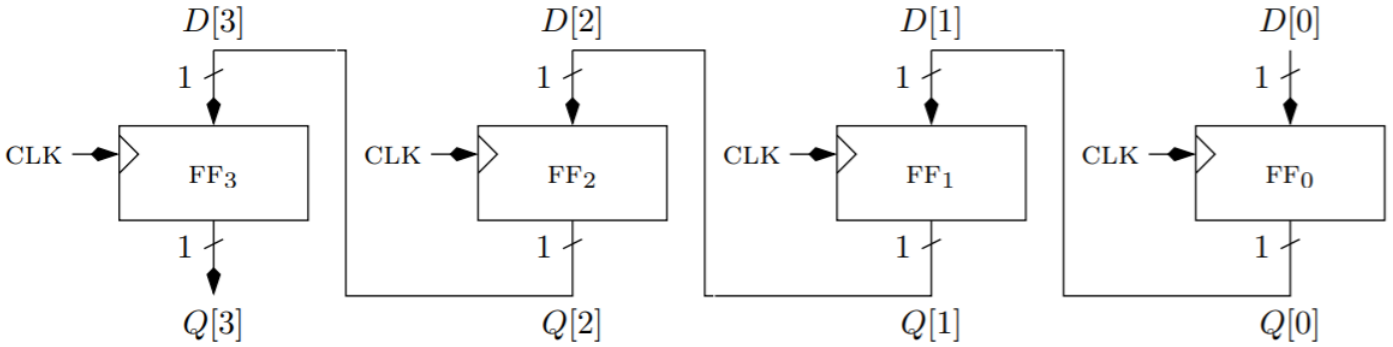
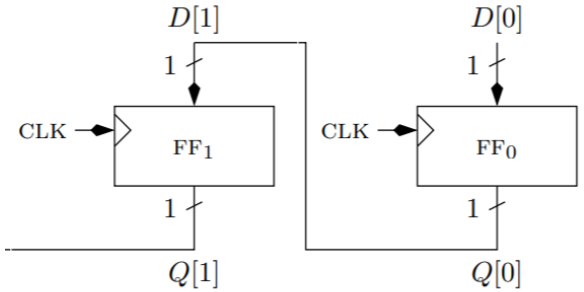


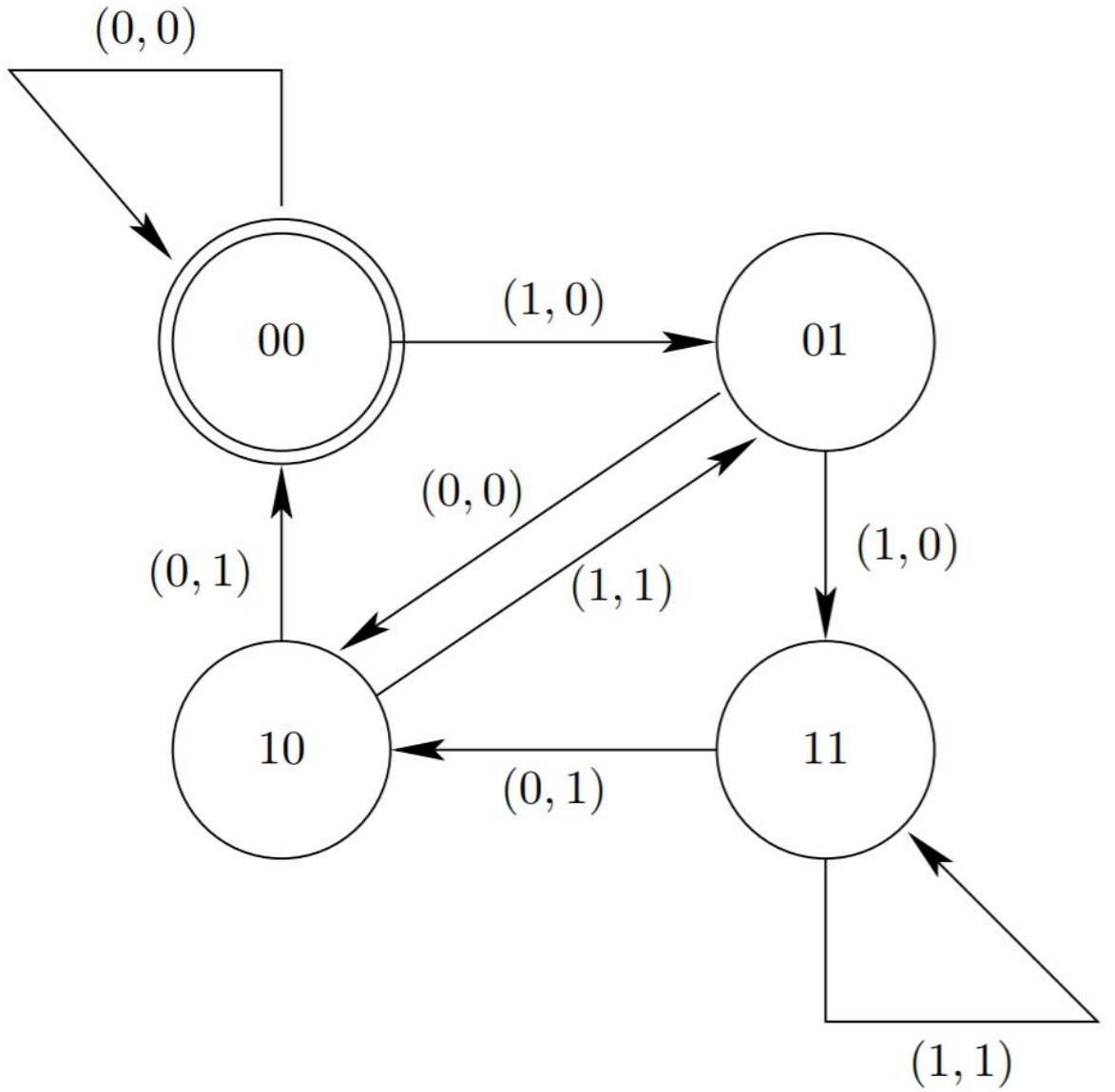
Figure: A 4-bit shift register.

i	$D[0]$	$Q[3:0]$
0	1	0000
1	1	0001
2	1	0011
3	0	0111
4	1	1110

(b) Simulation of shift register

Example 2: Analysis of Shift Register





Example 3: Synthesis of a 2-state FSM

Consider the FSM $\mathcal{A} = \langle Q, \Sigma, \Delta, \delta, \lambda, q_0 \rangle$ depicted in the next figure, where

$$Q = \{q_0, q_1\},$$
$$\Sigma = \Delta = \{0, 1\}.$$

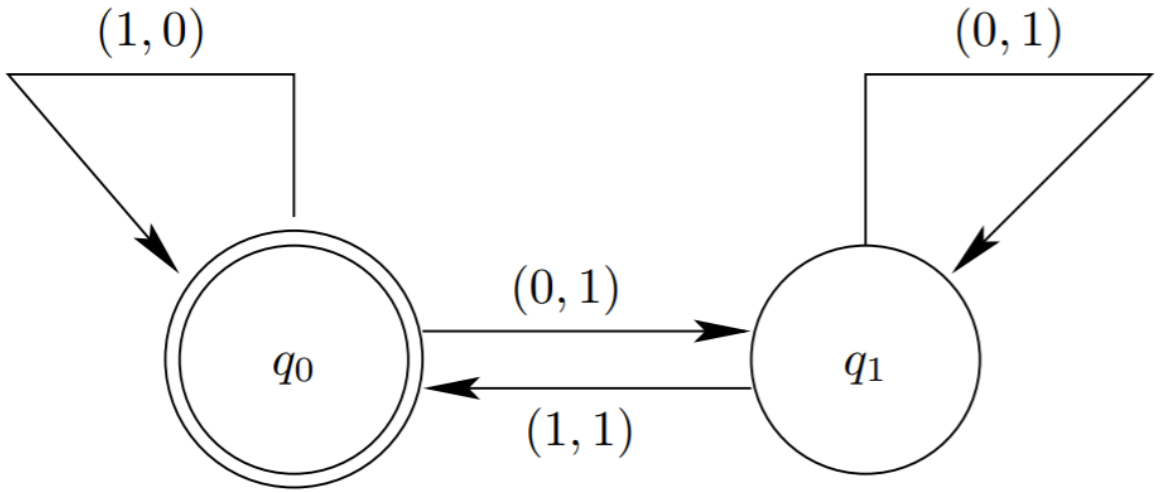
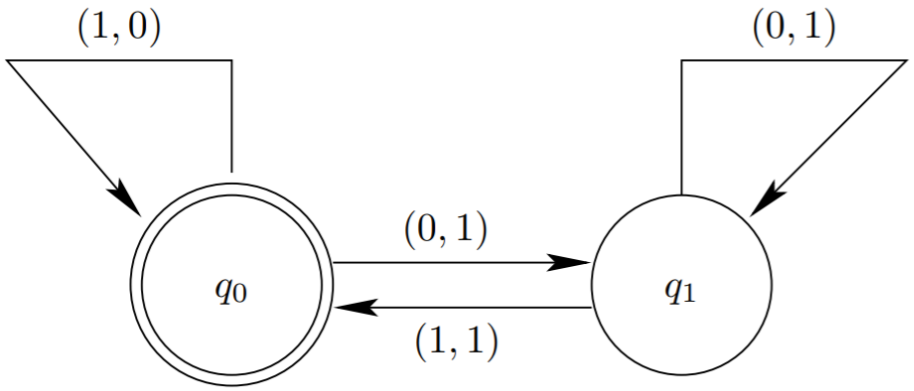


Figure: A two-state FSM.

Example 3: The circuit



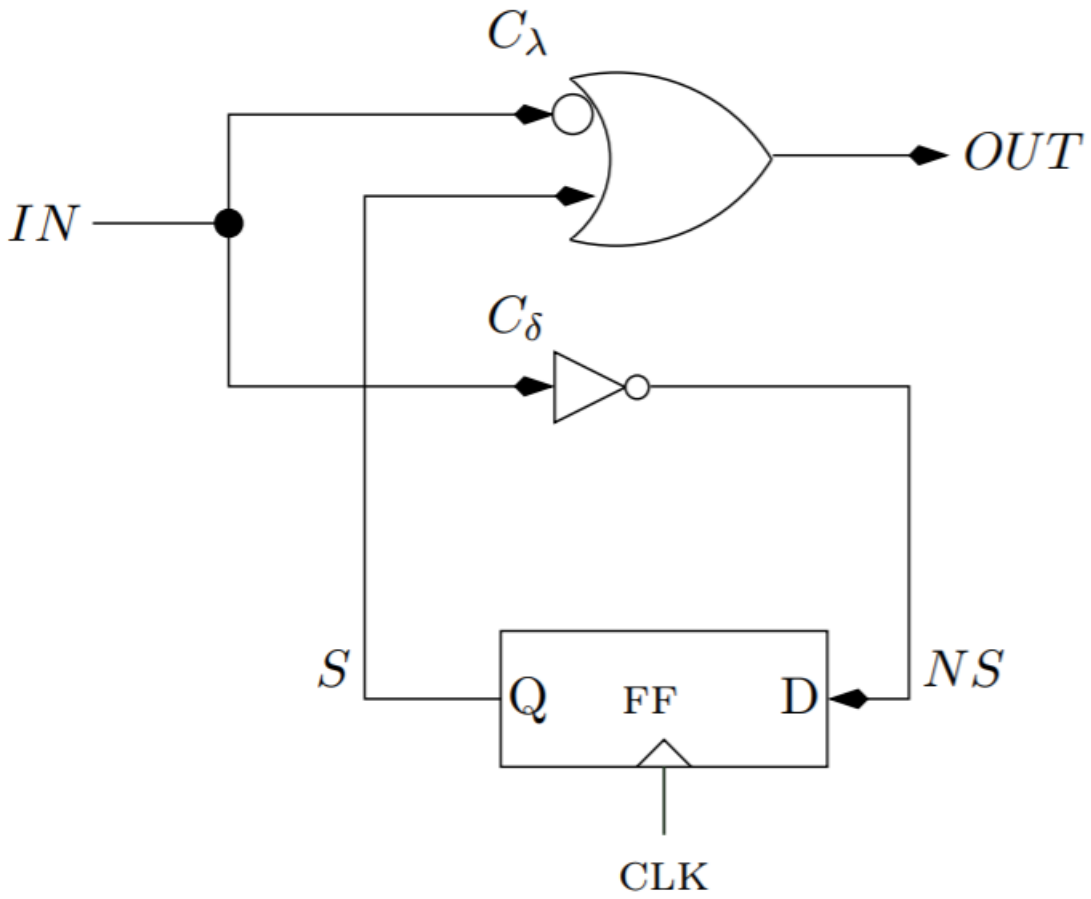


Figure: Synthesis of $\mathcal{A}$.

See you on Thursday!

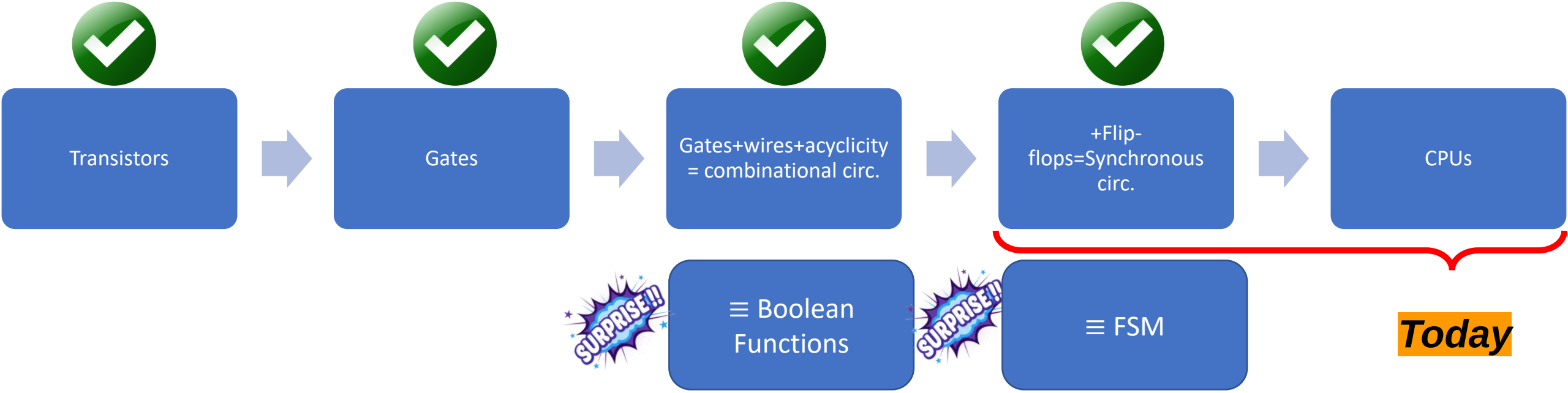
Enjoy the discussion ☺

How to Clock Your Computer: Digital Logic Design 3/3

Moti Medina

Slides, slides material, and figures are taken from “Digital Logic Design : a rigorous approach” lecture slides and book by Guy Even and **M** (Chapters: 15, 17-22).

Our Journey: From Transistors to Computers



Picture taken from here.

Today's plan

- **45 min lecture session (~pages 9-17 in the "reading material").**
  - Sequential Adder (4 min)
  - Binary Adder (2 min)
  - Ripple Carry Adder (RCA) (4 min)
  - Relation between Ripple Carry Adder and Seq. Adder (2 min)
  - Recursive Def. of RCA(n) (2 min)
  - Registers**:** parallel load and serial load (aka Shift registers) (6 min)
  - Random Access Memory (RAM) (10 min)
  - A simplified CPU example – the DLX. (14 min)
  - Recap (1 min)

- **Rest of the lecture:**
  - Discuss, solve and present Question 7, Question 8, or Question 9.
  - Questions appear on Page **17** in the reading material.

Sequential Adder

Definition

A *sequential adder* is defined as follows.

Inputs: A, B and a clock signal CLK, where $A_i, B_i \in \{0,1\}$.

Output: S, where $S_i \in \{0,1\}$.

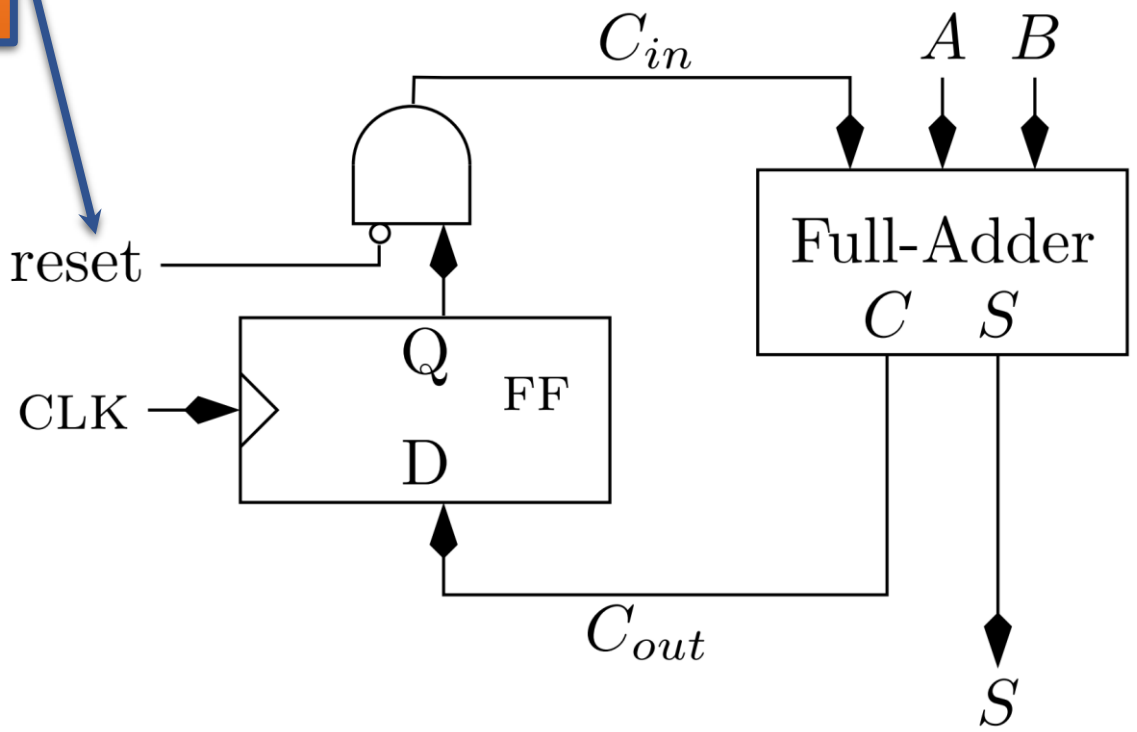
Functionality: Then, for every $i \geq 0$,

$$\langle A[i:0] \rangle + \langle B[i:0] \rangle = \langle S[i:0] \rangle \pmod{2^{i+1}}.$$

$$\langle A[n-1:0] \rangle \triangleq \sum_{i=0}^{n-1} A[i] \cdot 2^i.$$

Sequential Adder: Implementation

Assume $reset_i = 1 \leftrightarrow i = 0$



Definition

A *sequential adder* is defined as follows.

Inputs: A, B and a clock signal CLK, where $A_i, B_i \in \{0, 1\}$.

Output: S, where $S_i \in \{0, 1\}$.

Functionality: Then, for every $i \geq 0$,
$\langle A[i:0] \rangle + \langle B[i:0] \rangle = \langle S[i:0] \rangle \pmod{2^{i+1}}$.

Sequential Adder: Correctness

Theorem

$$\sum_{j=0}^{i} A_j \cdot 2^j + \sum_{j=0}^{i} B_j \cdot 2^j = \sum_{j=0}^{i} S_j \cdot 2^j + c_{out}(i) \cdot 2^{i+1} .$$

Proof.

The proof is by induction on i.
The induction basis for $i = 0$ follows from the functionality of the full-adder:

$$A_0 + B_0 + C_{in}(0) = 2 \cdot C_{out}(0) + S_0 .$$

This requires that $C_{in}(0) = 0$! Namely, that the FF is initialized to zero.

□

Complete the Pf. (Ex. 7)

Binary Adder

Definition

$\text{ADDER}(n)$ - a binary adder with input length n is a combinational circuit specified as follows.

Input: $A[n-1:0], B[n-1:0] \in \{0,1\}^n$, and $C[0] \in \{0,1\}$.

Output: $S[n-1:0] \in \{0,1\}^n$ and $C[n] \in \{0,1\}$.

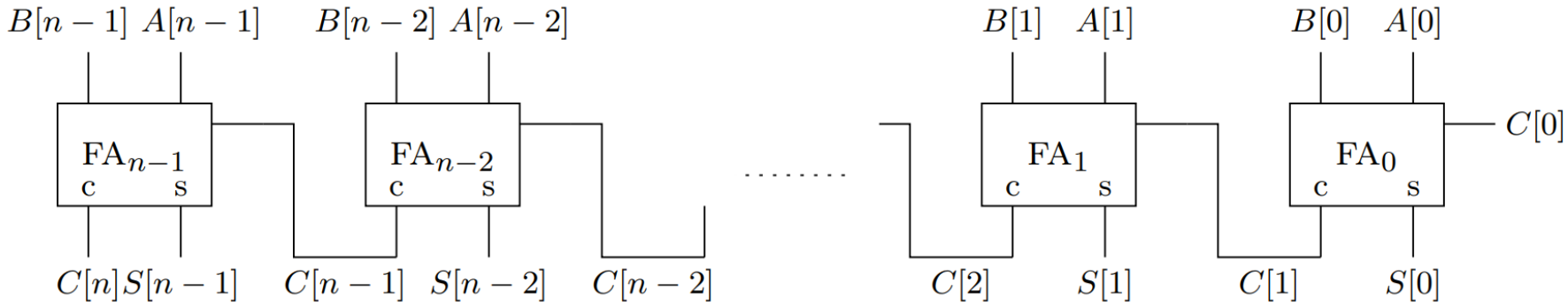
Functionality:

$$\langle \vec{S} \rangle + 2^n \cdot C[n] = \langle \vec{A} \rangle + \langle \vec{B} \rangle + C[0]. \quad (1)$$

Addition terminology:

- addends: $\langle \vec{A} \rangle = \sum_{i=1}^{n-1} A[i] \cdot 2^i$, and $\langle \vec{B} \rangle = \sum_{i=1}^{n-1} B[i] \cdot 2^i$
- carry-in bit : $C[0]$
- sum: $\langle \vec{S} \rangle$
- carry-out bit: $C[n]$

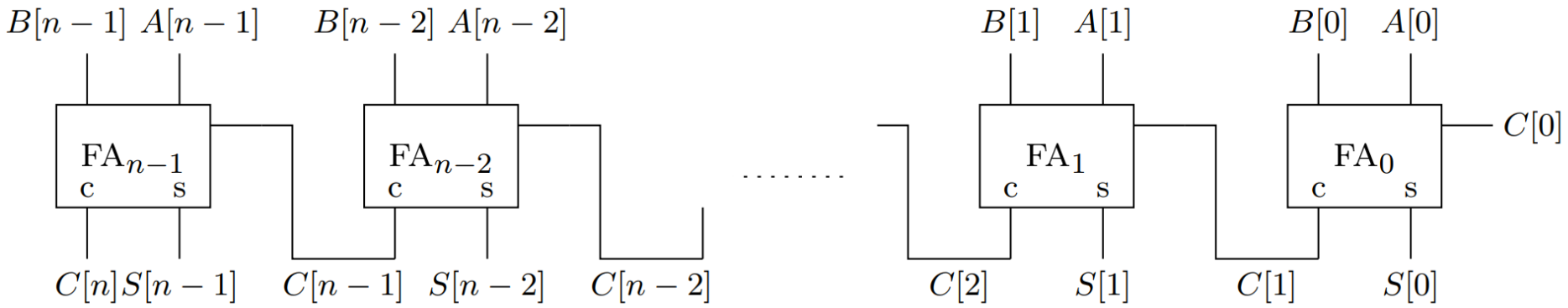
Ripple Carry Adder RCA(n)



- same addition algorithm that we use for adding numbers by hand.
- row of *n* Full-Adders connected in a chain.
- the weight of every signal is two to the power of its index. (Do not confuse weight here with Hamming weight. Weight means here the value in binary representation.)

Relation between RCA(n) and Seq. Adder

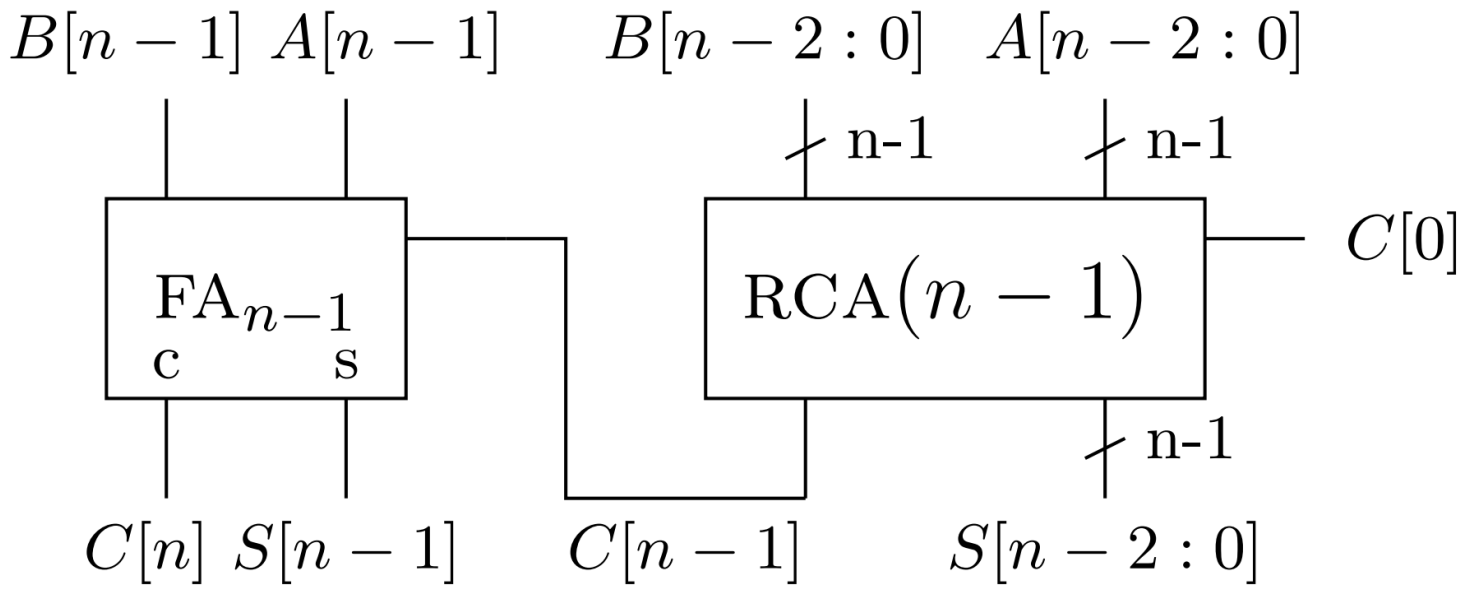
1. FA_i is "simulated" by the FA (in Seq. Adder) in the i'th clock cycle.
2. We can view $\text{RCA}(n)$ as an "unrolling" of the Seq. Adder.



Recursive Definition of RCA(n)

Basis: an $\mathrm{RCA}(1)$ is simply a Full-Adder.

Reduction Step:



Can We Do Better?

- Cost and Delay of RCA(n) are $\Theta(n)$.

- Lower bound of Binary Adder $\Omega(\log n)$ for the delay and $\Omega(n)$ for the cost
  - What is the cone of the $S[n-1]$?
  - What is the cone of $C[n]$?
  - Again: How did we manage to get a const. size seq. adder then?

- Can you somehow "break" RCA(n) and make it more "parallel"?
  - Almost optimal design CSA(n)
    - Cost is $\Theta(n^{\log_2 3})$
  - Optimal design uses Parallel-Prefix Comp.
    - Next semester.

Definition

$\text{ADDER}(n)$ - a binary adder with input length n is a combinational circuit specified as follows.

Input: $A[n-1:0], B[n-1:0] \in \{0,1\}^n$, and $C[0] \in \{0,1\}$.

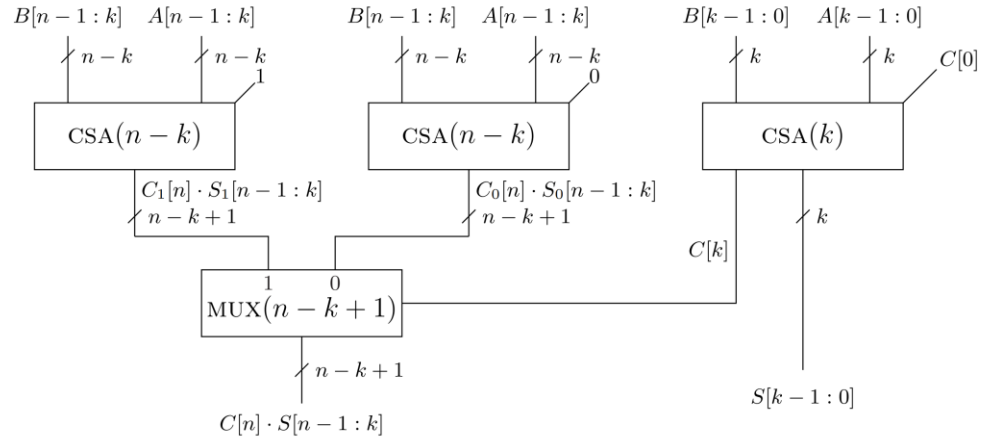
Output: $S[n-1:0] \in \{0,1\}^n$ and $C[n] \in \{0,1\}$.

Functionality:

$$\langle \vec{S} \rangle + 2^n \cdot C[n] = \langle \vec{A} \rangle + \langle \vec{B} \rangle + C[0]. \quad (1)$$

basis: A $\text{CSA}(1)$ is simply a Full-Adder.

reduction step:




And Now For Something
Completely Different

Registers

A term register is used to define a memory device that stores a bit or more. There are two main types of register depending on how their contents are loaded.

1. Parallel Load Register
2. Shift Register (also called a serial load register)

Parallel Load Register - specification

Definition

An n-bit *parallel load register* is specified as follows.

Inputs:
- $D[n-1:0](t)$,
- $\mathrm{CE}(t)$, and
- a clock CLK.

Output: $Q[n-1:0](t)$.

Functionality:

$$Q[n-1:0](t+1) = \begin{cases} D[n-1:0](t) & \text{if } \mathrm{CE}(t) = 1 \\ Q[n-1:0](t) & \text{if } \mathrm{CE}(t) = 0. \end{cases}$$

Parallel Load Register - design

i	$D[3:0]$	CE	$Q[3:0]$
0	1010	1	0000
1	0101	1	1010
2	1100	0	0101
3	1100	1	0101
4	0011	1	1100

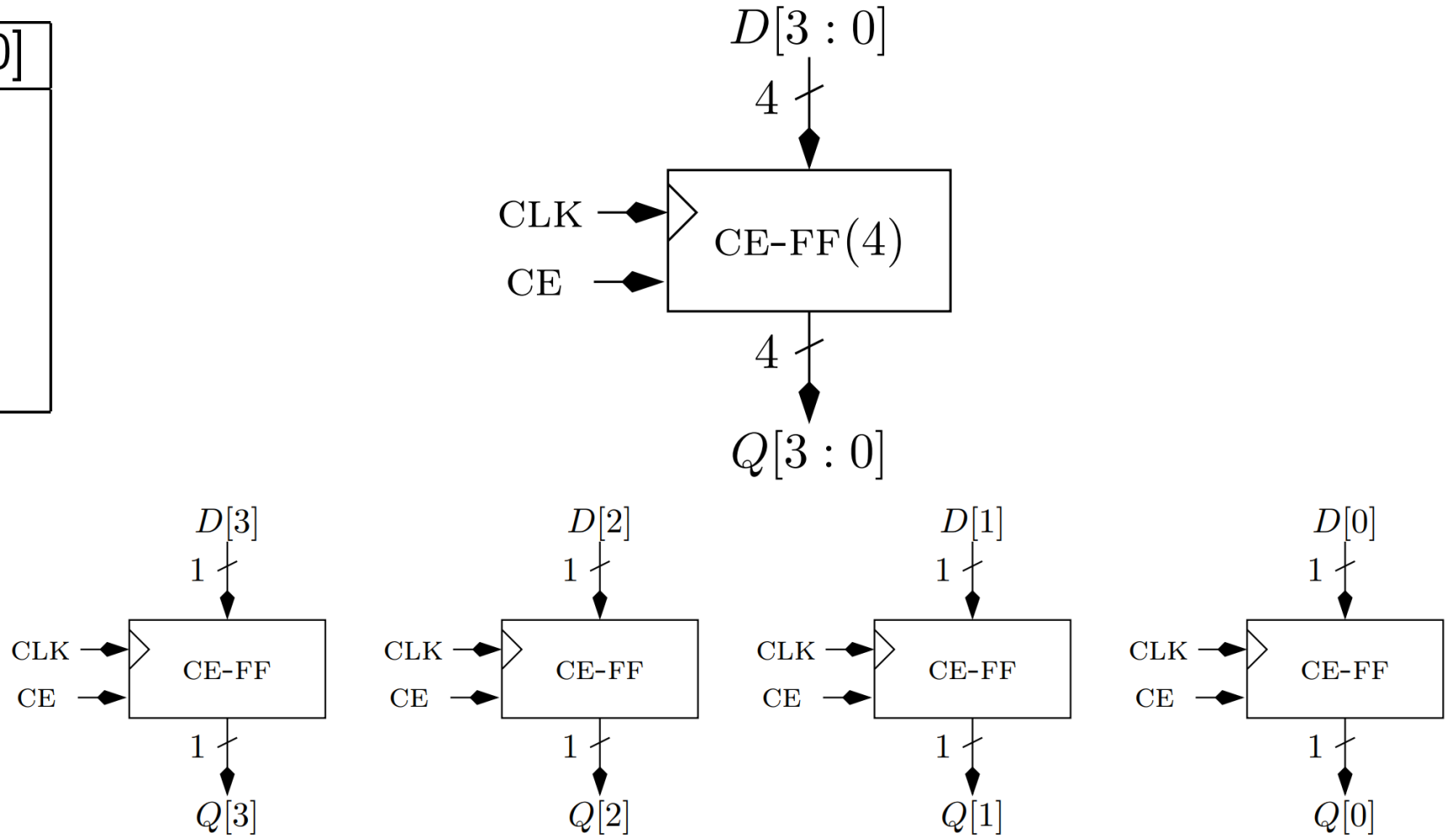


Figure: A 4-bit parallel load register

Random Access Memory (RAM)

RAM - definition

Definition

A $\text{RAM}(2^n)$ is specified as follows.

Inputs: $Address[n-1:0](t) \in \{0,1\}^n, D_{\text{in}}(t) \in \{0,1\}$, $R/\overline{W}(t) \in \{0,1\}$ and a clock CLK.

Output: $D_{\text{out}}(t) \in \{0,1\}$.

Functionality :

1. data: array $M[2^n - 1 : 0]$ of bits.
2. initialize: $\forall i : M[i] \leftarrow 0$.
3. For $t = 0$ to ∞ do
   1. $D_{\text{out}}(t) = M[\langle Address \rangle](t)$.
   2. For all $i \neq \langle Address \rangle$: $M[i](t+1) \leftarrow M[i](t)$.
   3.

$$M[\langle Address \rangle](t+1) \leftarrow \begin{cases} D_{\text{in}}(t) & \text{if } R/\overline{W}(t) = 0 \\ M[\langle Address \rangle](t) & \text{else.} \end{cases}$$

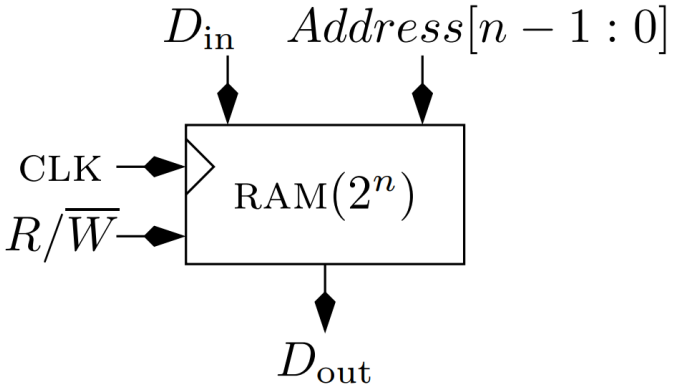


Figure: A schematic of a $\text{RAM}(2^n)$.

Memory Cell - specification

- Wait a minute…how do we do this for a single bit?

Definition

A single bit *memory cell* is defined as follows.

Inputs: $D_{\text{in}}(t)$, $R/\overline{W}(t)$, $sel(t)$, and a clock CLK.

Output: $D_{\text{out}}(t)$.

Functionality:

Assume that D_{out} is initialized zero, i.e., $D_{\text{out}}(0) = 0$. The functionality is defined according to the following cases.

$$D_{\text{out}}(t+1) \leftarrow \begin{cases} D_{\text{in}}(t) & \text{if } sel(t) = 1 \text{ and } R/\overline{W}(t) = 0 \\ D_{\text{out}}(t) & \text{otherwise.} \end{cases}$$

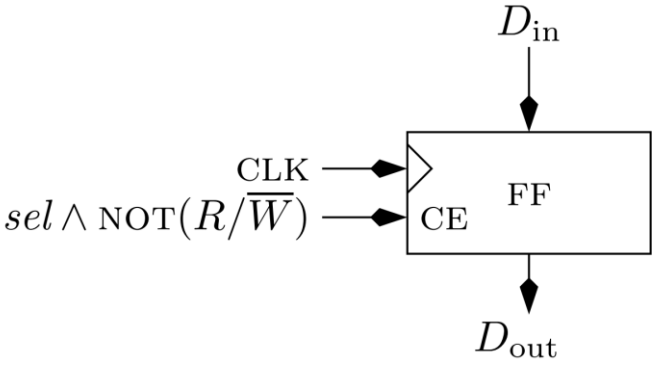


Figure: An implementation of a memory cell.

RAM -design

Definition

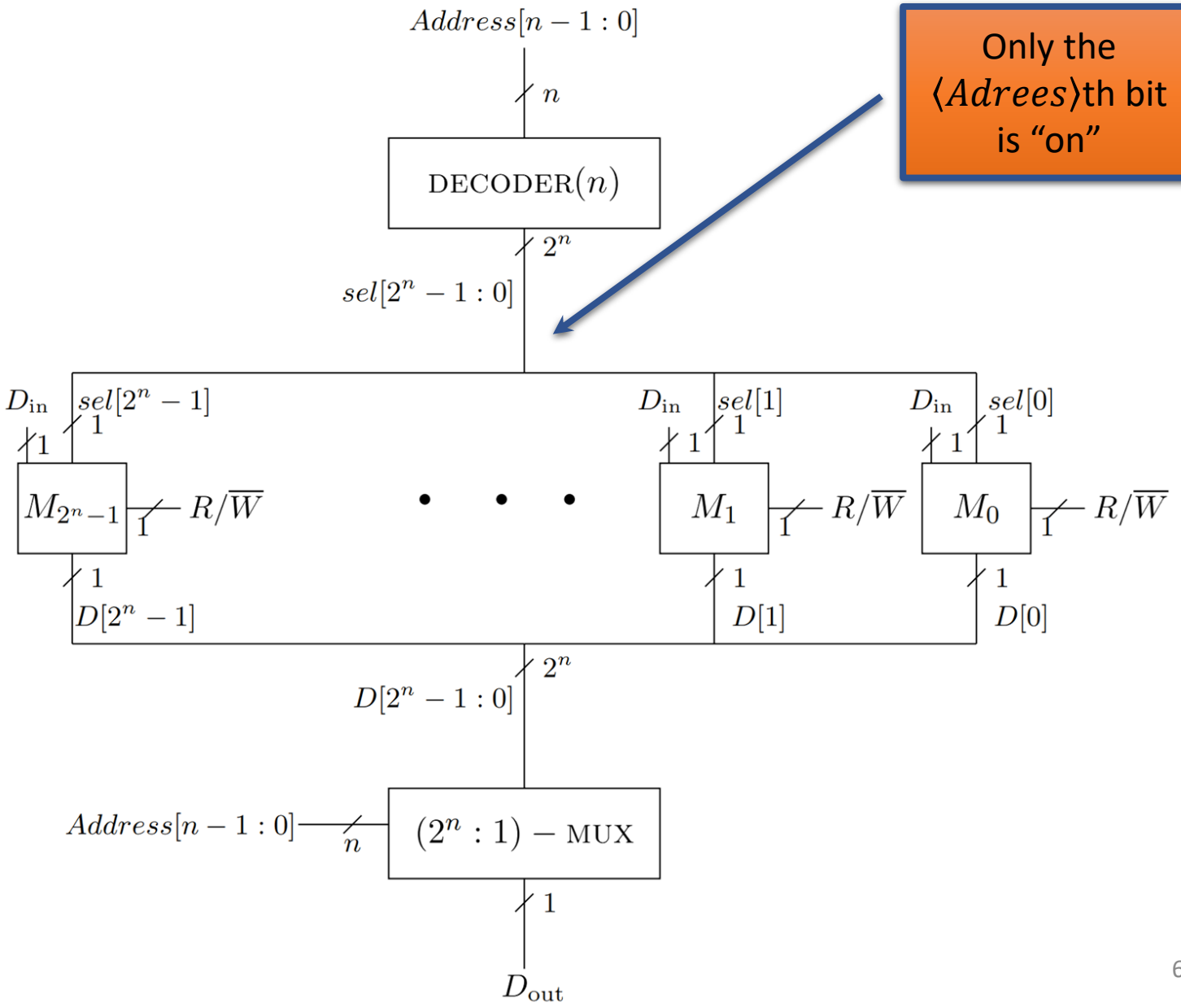
A $\text{RAM}(2^n)$ is specified as follows.

Inputs: $Address[n-1:0](t) \in \{0,1\}^n, D_{\text{in}}(t) \in \{0,1\}$, $R/\overline{W}(t) \in \{0,1\}$ and a clock CLK.

Output: $D_{\text{out}}(t) \in \{0,1\}$.

Functionality :

1. data: array $M[2^n-1:0]$ of bits.
2. initialize: $\forall i : M[i] \leftarrow 0$.
3. For $t = 0$ to ∞ do
   1. $D_{\text{out}}(t) = M[\langle Address \rangle](t)$.
   2. For all $i \neq \langle Address \rangle$: $M[i](t+1) \leftarrow M[i](t)$.
   3. $$M[\langle Address \rangle](t+1) \leftarrow \begin{cases} D_{\text{in}}(t) & \text{if } R/\overline{W}(t) = 0 \\ M[\langle Address \rangle](t) & \text{else.} \end{cases}$$



Only the $\langle Adrees \rangle$th bit is “on”

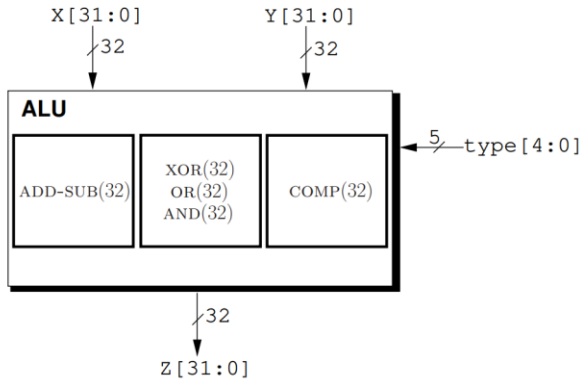

AND NOW FOR SOMETHING
COMPLETELY DIFFERENT.

We are ready…

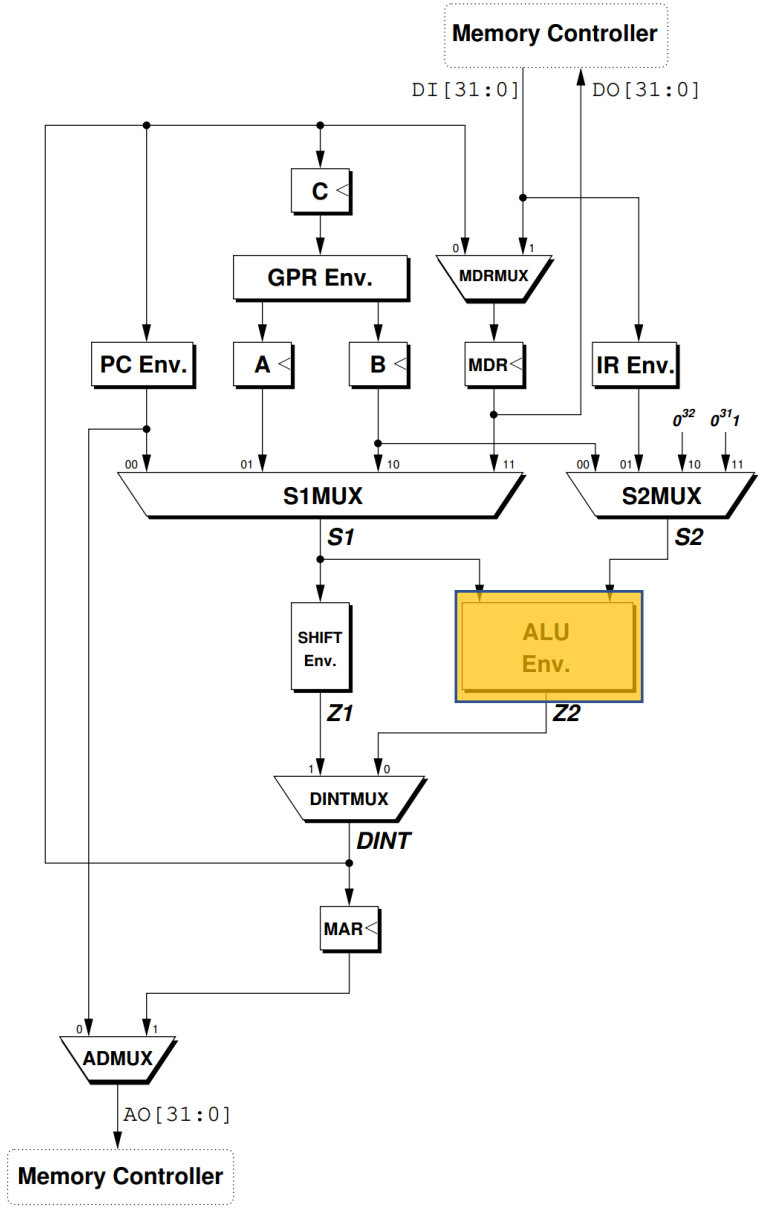
- We are now ready to **start** talking about a **high-level** implementation of a **simplified** processor (don't get confused – this is a huge achievement).
- We know how to implement FSMs and how to implement any Boolean function.
- We also have some (more than) rough idea on how a RAM looks like.
- Let us assume a processor that interacts with a RAM which stores both instructions and data that the processor operates on.
- These instructions have some syntax and semantics, that we won't discuss here.
- Nevertheless, we will see how to combine all the things that we saw in this series of lectures to implement a "system" that reacts to the bits that encode instructions and we shall see how that data flows in the processor.

The Datapath of the Simplified DLX Machine

The ALU is a combinational circuit that supports: addition and subtraction, bitwise logical instructions, and comparison instructions.



The main three subcircuits of the ALU are: (1) 32-bit adder/subtractor, ADD-SUB(32), (2) bitwise logical operations, XOR, OR, AND, and (3) a comparator, COMP(32). Note that the comparator is fed by the outputs of the adder/subtractor circuit.



ALU Environment

Definition

An ALU environment is a combinational circuit specified as follows:

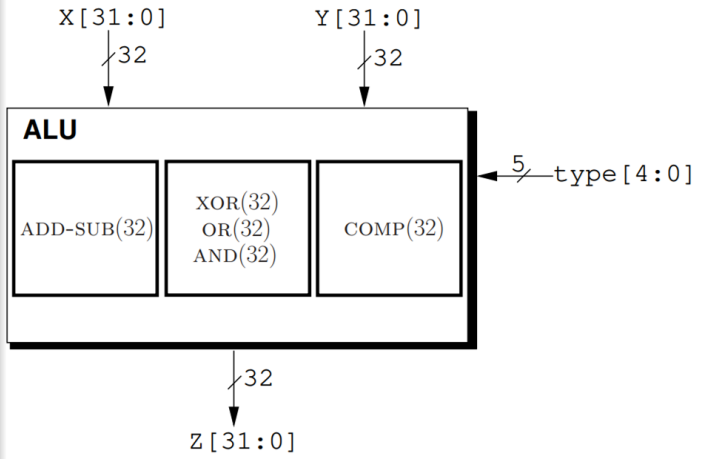
Input: $x[31:0], y[31:0] \in \{0,1\}^{32}$, $type \in \{0,1\}^{5}$.

Output: $z[31:0] \in \{0,1\}^{32}$.

Functionality:

$$\vec{z} \stackrel{\triangle}{=} f_{type}(\vec{x}, \vec{y}) \,,$$

We now need to describe how the ALU functions are encoded...



Shifter Environment

- The shifter is a 32-bit bi-directional logical shifter by one position.

Definition

The shifter environment is a combinational circuit defined as follows:

Input:

- $x[31:0] \in \{0,1\}^{32}$,
- $\texttt{shift} \in \{0,1\}$, and
- $\texttt{right} \in \{0,1\}$.

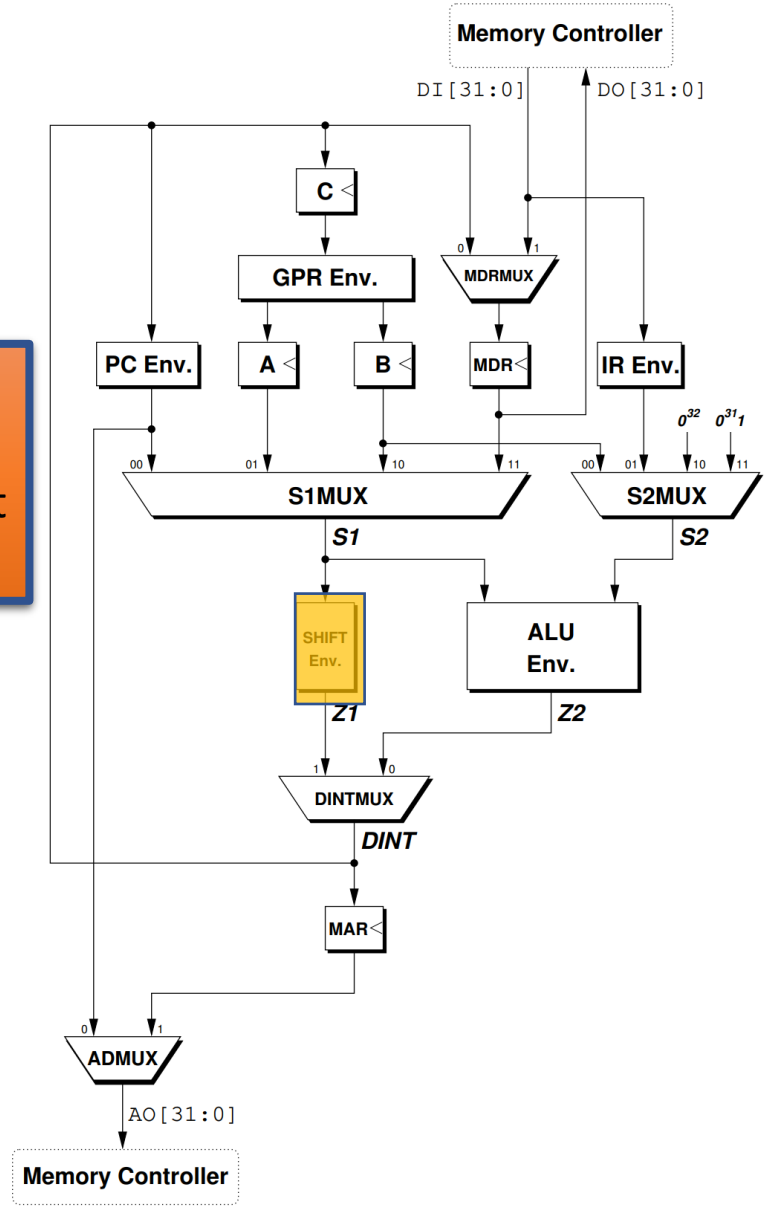
Output: $y[n-1:0] \in \{0,1\}^{32}$.

Functionality: The output $\vec{y}$ satisfies

$$\vec{y} \triangleq \begin{cases} \vec{x}, & \text{if } \texttt{shift} = 0, \\ \mathrm{LLS}(\vec{x}, 1), & \text{if } \texttt{shift} = 1, \texttt{right} = 0, \\ \mathrm{LRS}(\vec{x}, 1), & \text{if } \texttt{shift} = 1, \texttt{right} = 1. \end{cases}$$

LLS: Left shift
LRS: Right shift

The shifter also implements the identity function: route a word through the shifter in the execution of some instructions.

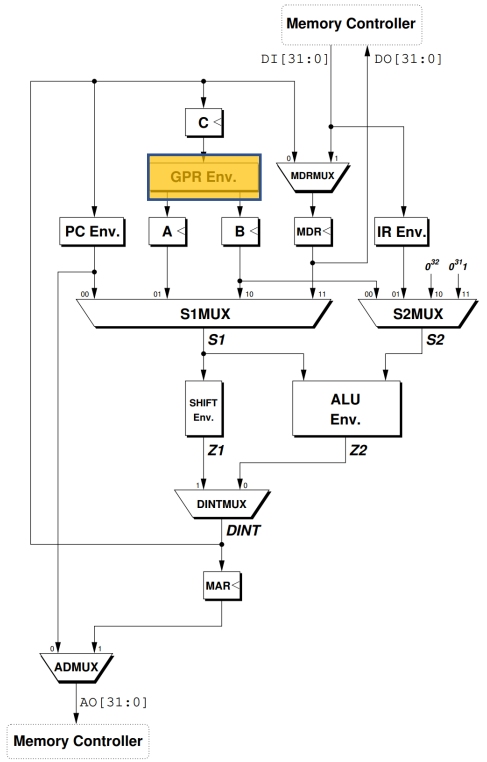


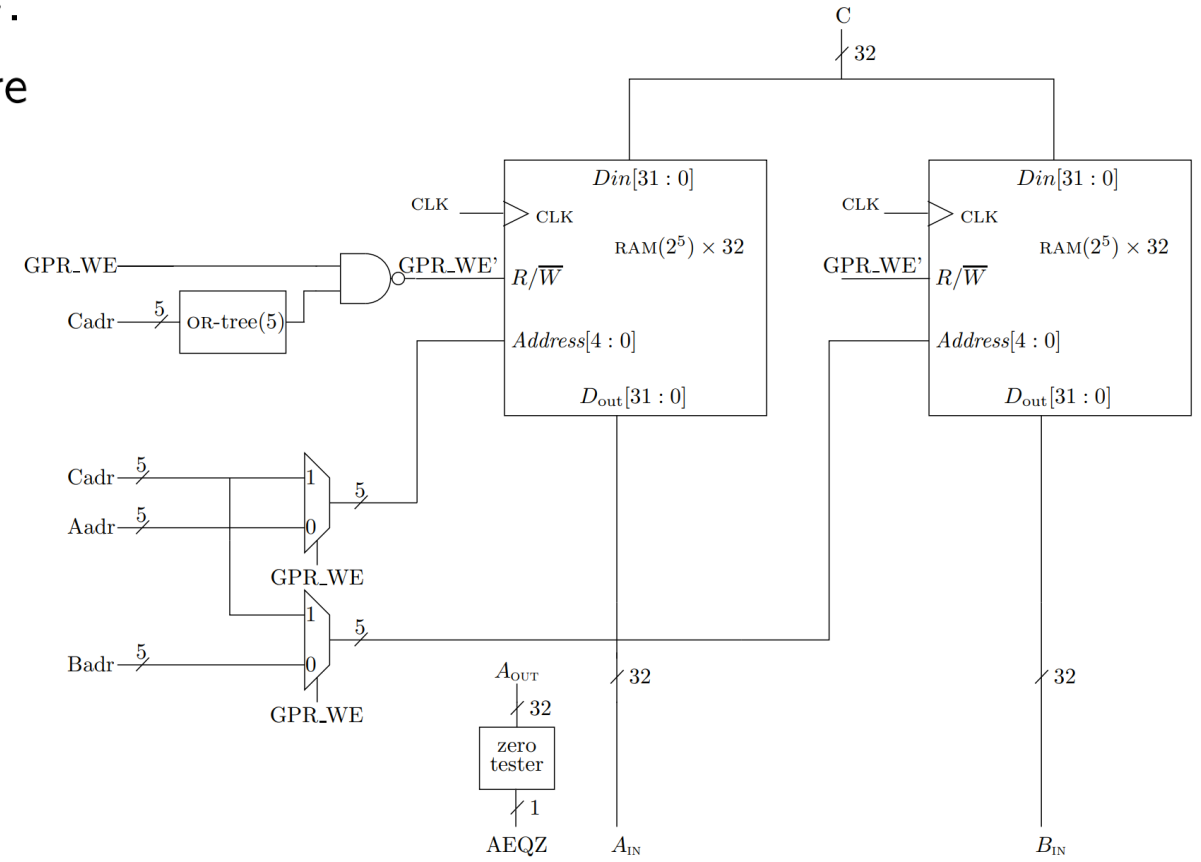
The GPR Environment

There are 32 registers in the GPR Environment, called $\mathrm{R0}, \mathrm{R1}, \ldots, \mathrm{R31}$. The GPR Environment (or GPR, for short) can support one of two operations in each clock cycle.

1. Write the value of input C in $\mathrm{R}i$, where $i = \langle \texttt{Cadr} \rangle$.
2. Read the contents of the registers $\mathrm{R}i$ and $\mathrm{R}j$, where $i = \langle \texttt{Aadr} \rangle$ and $j = \langle \texttt{Badr} \rangle$.

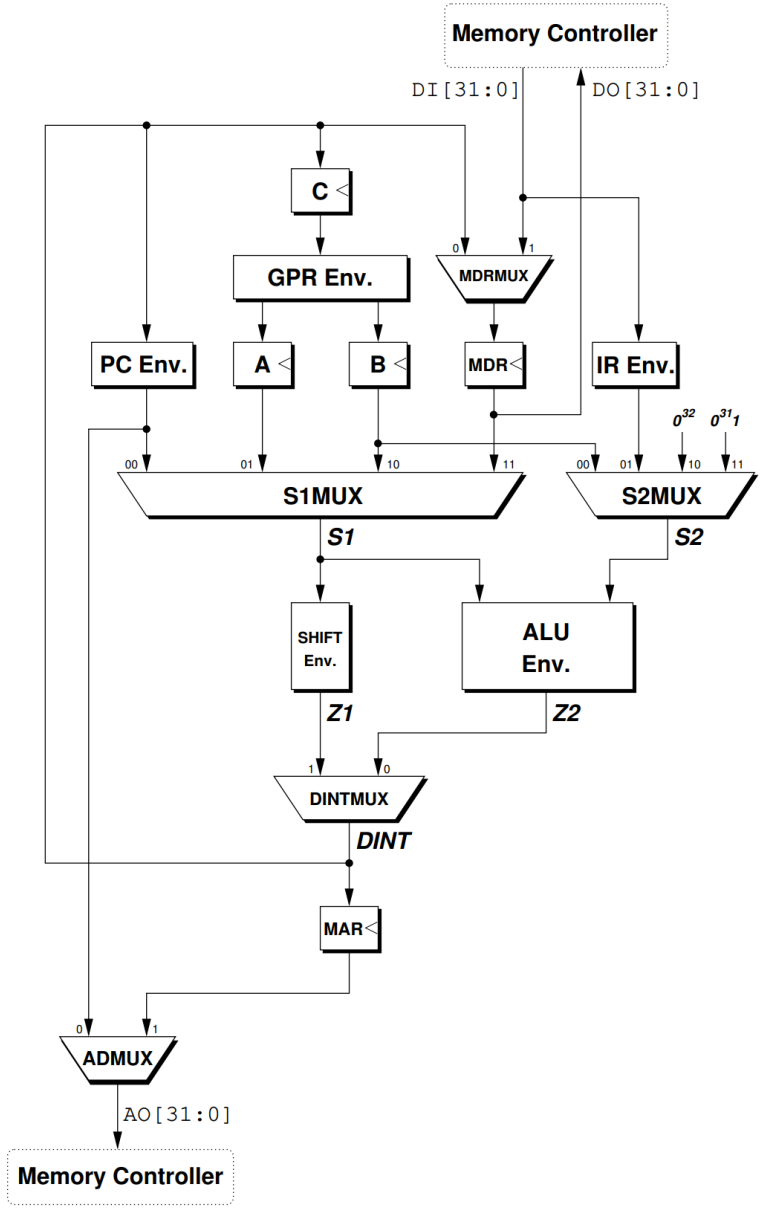
Addresses are coming from the instruction





Special Registers

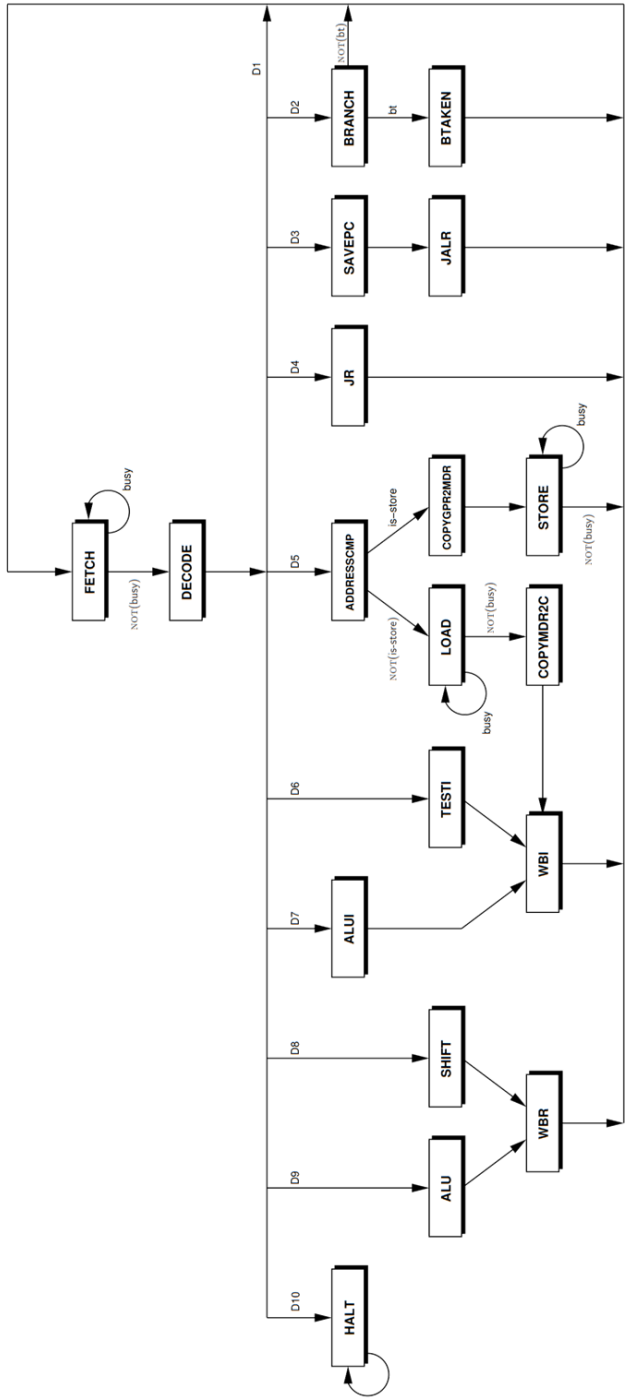
- **A, B, C, MAR, MDR, PC**, and **IR** are (ce-) registers
  - With some additional logic (e.g., MUXes) for reg's with env's.
- **IR** (Instruction Reg.) holds the instruction brought from memory.
- **A,B,C** are used as "interface" registers to the GPR env.
- **PC** (Program Counter) simply holds a "pointer" to memory which (unless the programmer affects it) advances by 1 after bringing an instruction from memory (e.g., PC env is a CE Counter).
- **MAR, MDR** (Mem. Add. Reg, Mem. Data. Reg.) holds a memory address that some data should be brought from and MDR holds data that is brought from memory.
- All registers in this processor are 32 bit wide



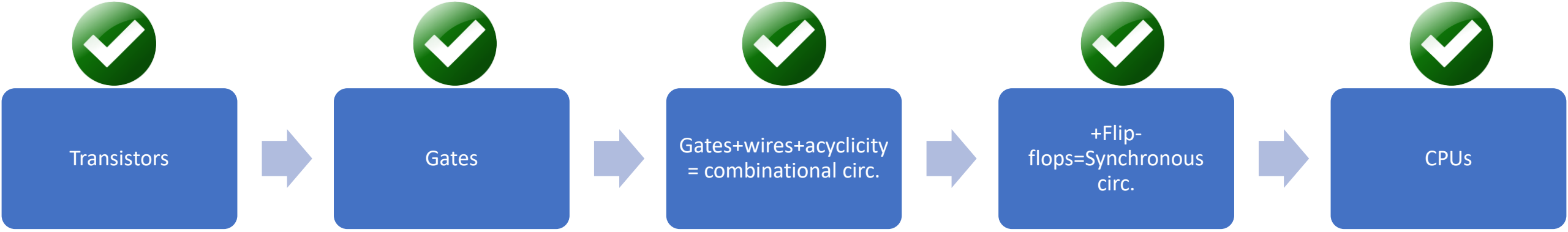
DLX Control

- The control is an **FSM** that interprets the DLX instructions.
- For every DLX instruction the control output a sequence of control signals that are input to the data-path (clock enables, MUX select bits, etc.)
- These control signals are output according to the state in which the FSM is at and according to the executed instruction…as in every FSM.
- Essentially, these control signals control the way the bits are routed between the registers.
- Let us stop here.

- By now, you know that you **can** implement any FSM. **The complete processor consists of the Control and the DATA path.**



Our journey is over: From Transistors to Computers



Picture taken from here.

Good Luck!

Enjoy the discussion ☺