Chapter 1: The digital abstraction

Computer Structure
&
Intro. to Digital Computers

Dr. Guy Even

Tel-Aviv Univ.

Digital Circuits vs. Analog Devices

Property	Digital Circuit	Analog Device
values	$\{0,1\}$	$\mathbb{R}$
description	simple (Boolean function)	complicated (differential eq.)
real?	abstract model	very real

Conclusion: much easier to use the digital abstraction than the realistic, complete, complicated analog model.

Preliminary questions

- what is an analog device? (components, behavior)
- in what way does a digital circuit model an analog device?
  - can every analog device be modeled as a digital circuit?
  - what type of digital circuits do we want?
  - why is one inverter better than another?
- how can we tell if an analog device is a gate (say, an inverter)?

Transistors

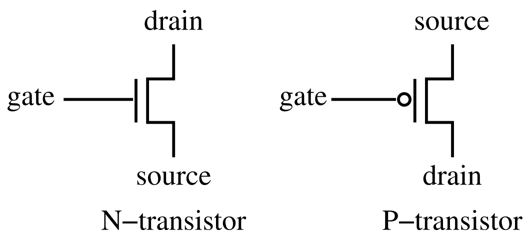
Computers ⇒ VLSI chips ⇒ gates & flip-flops ⇒ transistors
Transistors are the basic components.
Most common VLSI technology is called CMOS.
In CMOS: only two types of transistors:

- N-transistor
- P-transistor

in case you are curious:
VLSI = Very Large Scale Integration

CMOS = Complementary Metal-Oxide Semiconductor

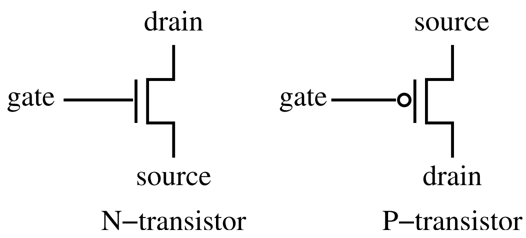
N-transistor & P-transistor



Inputs: gate & source
Output: drain

(not accurate! just for the sake of this discussion)

N-transistor & P-transistor



Functionality of N-transistor:

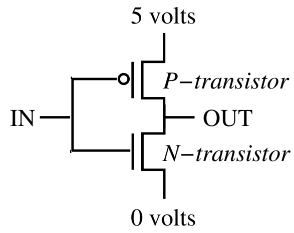
- If $v(gate) = high$, then $resistance(source, drain) = 0$ (and then $v(drain) \leftarrow v(source)$)
- If $v(gate) = low$, then $resistance(source, drain) = \infty$

Functionality of P-transistor:

- If $v(gate) = high$, then $resistance(source, drain) = \infty$
- If $v(gate) = low$, then $resistance(source, drain) = 0$

Story true if: $v(s) = high$ in P & $v(s) = low$ in N.

A CMOS inverter



IN $= low$:

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

$\Rightarrow v(OUT) = high$

IN $= high$:

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

$\Rightarrow v(OUT) = low$

Qualitative Analysis vs. Quantitative Analysis

Qualitative analysis:

- gives an idea about “how an inverter works”.
- no idea about actual voltages of output as a function input voltage.
- no idea about how long it takes the output to stabilize.

Quantitative analysis:

- based on precise modeling of transistor.
- computes precise input-output relationship.
- requires a lot of work (usually done with the aid of a computer program called SPICE).

Analog signals

An analog signal is a real function

$$f : \mathbb{R} \to \mathbb{R},$$

where $f(t) =$ voltage as a function of the time.

Assumption: wires have zero resistance, zero capacity, and signals propagate through wires without delay.
$\Rightarrow$ voltage along a wire is identical at all times.

Since a signal describes the voltage (i.e. derivative of energy as a function of charge), we also assume that a signal is a continuous function.

Digital signals

A digital signal is a function

$$g : \mathbb{R} \to \{0, 1, \text{non-logical}\}.$$

The value of a digital signal describes the logical value carried along a wire as a function of time.

- zero & one : logical values.
- non-logical: indicates that the signal is not digital.

Interpreting analog signals as digital signals

Q: How does one interpret an analog signal as a digital signal?

naive answer: define a threshold voltage V'.
Consider an analog signal $f(t)$.

The digital signal *dig*$(f(t))$ is defined as follows.

$$\textit{dig}(f(t)) \triangleq \begin{cases} 0 & \text{if } f(t) < V' \\ 1 & \text{if } f(t) > V' \end{cases}$$

Q: is this a useful definition?

problems with definition of *dig*$(f(t))$

- All devices in a circuit must use exactly the same threshold V'. This is impossible due to manufacturing tolerances.
- Perturbations of $f(t)$ around V' lead to unexpected values of *dig*$(f(t))$.
  example: measure weight w using a spring. Wish to compare of length ℓ of spring with ℓ_t. Spring length oscillates around ℓ. If $\ell \approx \ell_t$, then comparison requires a long time.

$\Rightarrow$ must separate thresholds for 0 and for 1.

Interpreting analog signals as digital signals

Q: How does one interpret an analog signal as a digital signal?

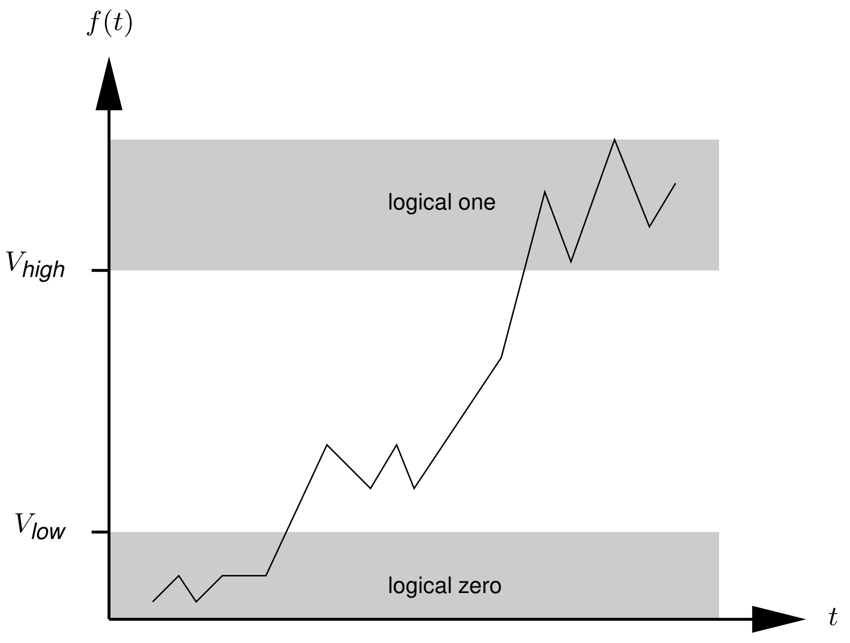
A: Two voltage thresholds are defined: $V_{low} < V_{high}$.

Consider an analog signal $f(t)$.

The digital signal $dig(f(t))$ is defined as follows.

$$dig(f(t)) \triangleq \begin{cases} 0 & \text{if } f(t) < V_{low} \\ 1 & \text{if } f(t) > V_{high} \\ \text{non-logical} & \text{otherwise.} \end{cases}$$

digital interpretation of an analog signal



did we solve the problems of a single threshold?

- manufacturing requirements: a low output must be $\leq V_{low}$ & a high output must be $\geq V_{high}$.
- fluctuations of $f(t)$ around V_{low} still cause fluctuations of *dig*$(f(t))$.
  However, these fluctuations are between 0 and "non-logical" (not between 0 and 1). This is still a problem, but not as bad...

Will noise cause a problem?

Noise = undesired changes to $f(t)$. Back to the example of a weight hanging from a spring: wind causes changes in the spring length and disturbs measurement of spring length.

An inverter

Q: define an inverter.

A:

$$\textit{dig}(OUT(t)) \triangleq \begin{cases} 0 & \text{if } \textit{dig}(IN(t)) = 1 \\ 1 & \text{if } \textit{dig}(IN(t)) = 0 \\ \text{non-logical} & \text{otherwise.} \end{cases}$$

We will see shortly that: noise $\Rightarrow$ cannot use these definitions to build correct circuits.

Before we can answer that we need to discuss transfer functions...

Transfer functions

The voltage at an output of a gate depends on the voltages of the inputs of the gate. This dependence is called the transfer function.

Example: an inverter with an input x and an output y. The value of the signal $y(t')$ at time t' is a function of the signal $x(t)$ in the interval $(-\infty, t']$.

Static transfer function: if the input $x(t)$ is stable for a sufficiently long period of time and equals x_0, then the output $y(t)$ stabilizes on a value y_0 that is a function of x_0.

If this were not the case, then we have a memory device not a logical gate.

Static transfer function

Let G denote a gate with one input x and one output y.

Definition: A function $f : \mathbb{R} \rightarrow \mathbb{R}$ is a static transfer function of a gate G if

$$\exists \Delta > 0 \;\; \forall x_0 \;\; \forall t_0 \;\; : \quad \forall t \in [t_0 - \Delta, t_0] \;\; x(t) = x_0 \quad \Longrightarrow \quad y(t_0) = f(x_0).$$

- Δ - propagation delay (time required for stable output)
- x_0 - stable input voltage
- t_0 - time in which $y(t)$ is measured

Static transfer function - remarks

(1) Since circuits operate over a bounded range of voltages, static transfer functions are usually only defined over bounded domains and ranges (say $[0, 5]$ volts).

Static transfer function - remarks

(2) Allow perturbations of $x(t)$ and $y(t)$.

$$\forall \epsilon \;\; \exists \delta, \Delta > 0 \;\; \forall x_0, t_1, t_2 \;\; :$$
$$\forall t \in [t_1, t_2] \;\; : \;\; |x(t) - x_0| \leq \delta$$
$$\Longrightarrow$$
$$\forall t \in [t_1 + \Delta, t_2] \;\; : \;\; |y(t) - f(x_0)| \leq \epsilon.$$

- δ - measures stability of input $x(t)$
- ϵ - measures stability of output $y(t)$
- $[t_1, t_2]$ - interval during which $x(t)$ is ϵ-stable.
- $[t_1 + \Delta, t_2]$ - interval during which $y(t)$ is δ-stable.

Propagation delay Δ depends only on ϵ (which is fixed and the same for all gates).

back to the definition of an inverter

$$dig(OUT(t)) \triangleq \begin{cases} 0 & \text{if } dig(IN(t)) = 1 \\ 1 & \text{if } dig(IN(t)) = 0 \\ \text{non-logical} & \text{otherwise.} \end{cases}$$

or equivalently,

$$IN(t) < V_{low} \Longrightarrow OUT(t) > V_{high}$$
$$IN(t) > V_{high} \Longrightarrow OUT(t) < V_{low}$$

Q: Define a NAND-gate.

Noise

A ——— *wire* ——— B

Noise signal: the difference $B(t) - A(t)$. (reference signal = $A(t)$).

Q: what causes noise?

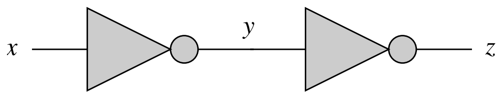
A: The main source of noise is heat. Heat causes random movement of electrons. These random movements do not cancel out perfectly, and random currents are created. These random currents create perturbations in the voltage of a wire.

Bounded noise model

- Bounded noise model - the noise signal along every wire has a bounded absolute value.
- Uniform bounded noise model - $\exists \epsilon > 0$ such that the absolute value of all noise signals is bounded by ϵ.
- Justification - noise is a random variable whose distribution has a rapidly diminishing tail. If the ϵ is sufficiently large, then

$$Prob[|\text{noise}| > \epsilon] \approx 0.$$

The digital abstraction in the presence of noise



Assume that:

- $x > V_{high}$, so *dig*$(x) = 1$,
- $y = V_{low} - \epsilon'$, for a very small $\epsilon' > 0$.
- $\Rightarrow dig(z) = 1$.

What if input to 2nd inverter equals $y(t) + n_y(t)$?

If $n_y(t) > \epsilon'$, then *dig*(y) = non-logical, and can't deduce that $dig(z) = 1$.

$\Rightarrow$ must strengthen the digital abstraction!

Redefining the digital interpretation of analog signals

Deal with noise: interpret input signals and output signals differently.

Input Signal: a signal measured at an input of a gate.

Output Signal: a signal measured at an output of a gate.

Redefining the digital interpretation (cont.)

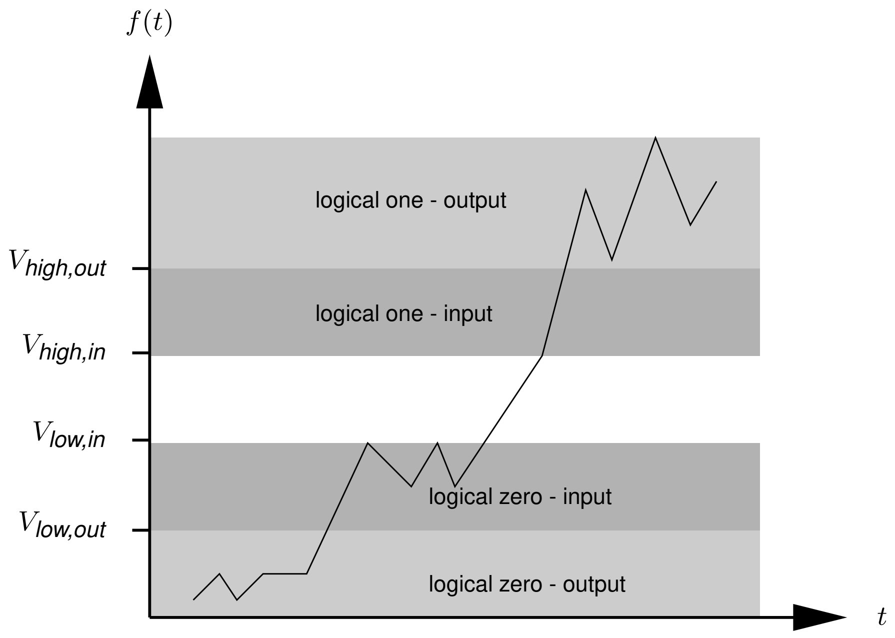
Instead of two thresholds, V_{low} and V_{high}, we define the following four thresholds:

- $V_{low,in}$ - an upper bound on a voltage of an input signal interpreted as a logical zero.
- $V_{low,out}$ - an upper bound on a voltage of an output signal interpreted as a logical zero.
- $V_{high,in}$ - a lower bound on a voltage of an input signal interpreted as a logical one.
- $V_{high,out}$ - a lower bound on a voltage of an output signal interpreted as a logical one.

These four thresholds satisfy the following equation:

$$V_{low,out} < V_{low,in} < V_{high,in} < V_{high,out}.$$

Redefining the digital interpretation (cont.)



Digital interpretation of input & output signals

Consider an input signal $f_{in}(t)$. The digital signal *dig*$(f_{in}(t))$ is defined as follows.

$$\mathit{dig}(f_{in}(t)) \triangleq \begin{cases} 0 & \text{if } f_{in}(t) < V_{\mathit{low,in}} \\ 1 & \text{if } f_{in}(t) > V_{\mathit{high,in}} \\ \text{non-logical} & \text{otherwise.} \end{cases}$$

Consider an output signal $f_{out}(t)$. The digital signal *dig*$(f_{out}(t))$ is defined analogously.

$$\mathit{dig}(f_{out}(t)) \triangleq \begin{cases} 0 & \text{if } f_{out}(t) < V_{\mathit{low,out}} \\ 1 & \text{if } f_{out}(t) > V_{\mathit{high,out}} \\ \text{non-logical} & \text{otherwise.} \end{cases}$$

Noise margins

The differences

$$V_{\textit{low,in}} - V_{\textit{low,out}} \quad \text{and} \quad V_{\textit{high,out}} - V_{\textit{high,in}}$$

are called noise margins.

Claim: If the absolute value of the noise is less than the noise margin, then the logical value of an output signal is unchanged.
Proof: If the absolute value of the noise $n(t)$ is bounded by the noise margins, then an output signal $f_{out}(t) < V_{\textit{low,out}}$ will result with an input signal
$f_{in}(t) = f_{out}(t) + n(t) < V_{\textit{low,in}}$. □

Inverter - revisited

We are now ready to define an inverter.

Definition: Let G denote a device with one input x and one output y. The device G is an inverter if its static transfer function $f(x)$ satisfies:

$$x(t) < V_{\textit{low,in}} \Longrightarrow y(t) > V_{\textit{high,out}}$$
$$x(t) > V_{\textit{high,in}} \Longrightarrow y(t) < V_{\textit{low,out}}.$$

Q: can you define a NAND-gate?

Logical & stable analog signals

back to the zero-noise model (to simplify the discussion)...

logical signal: $f(t)$ is logical at time t if *dig*$(f(t)) \in \{0, 1\}$.

stable signal: $f(t)$ is stable during the interval $[t_1, t_2]$ if $f(t)$ is logical for every $t \in [t_1, t_2]$.

Claim: If an analog signal $f(t)$ is stable during the interval $[t_1, t_2]$ then one of the following holds:

1. *dig*$(f(t)) = 0$, for every $t \in [t_1, t_2]$, or
2. *dig*$(f(t)) = 1$, for every $t \in [t_1, t_2]$.

Proof: Continuity of $f(t)$ & $V_{low} < V_{high}$. $\square$

Logical & stable digital signals

Let $x(t)$ denote a digital signal.

logical signal: $x(t)$ is logical at time t if $x(t) \in \{0, 1\}$.

stable signal: $x(t)$ is stable during the interval $[t_1, t_2]$ if $x(t)$ is logical for every $t \in [t_1, t_2]$.

Summary

- Signals - analog & digital
- Noise - bounded noise model & zero noise model
- Digital interpretation of analog signals
- Transfer functions
- Definition of gate (e.g. inverter) using transfer function
- Stable & logical signals

Chapter 2: Foundations of combinational structures

Computer Structure

&

Intro. to Digital Computers

Dr. Guy Even

Tel-Aviv Univ.

Goals

- define combinational circuits.
- prove that every Boolean function can be implemented by a combinational circuit.
- prove that every combinational circuit implements a Boolean function.
- present an algorithm for simulating a combinational circuit.
- present an algorithm for analyzing the delay of a combinational circuit.

Boolean functions

$\{0,1\}^n$ - the set of n-bit strings.

A Boolean function - a function $f : \{0,1\}^n \to \{0,1\}^k$.

n: input length

k: output length

Gates

- A gate is a device with inputs and outputs. Inputs/outputs often called terminals, ports, or even pins.
- For simplicity, assume a gate G with two inputs x_1, x_2 and one output y.
- The functionality of a gate G is specified by a Boolean function $f : \{0,1\}^2 \rightarrow \{0,1\}$. (must allow gate time to compute f! recall definition of static transfer function.)
- Number of inputs is called fan-in of the gate.

Def: The gate G is consistent with a Boolean function f at time t if the input values are digital at time t and

$$y(t) = f(x_1(t), x_2(t)).$$

Gates - cont.

propagation delay t_{pd} -the amount of time that elapses till G becomes consistent.

Def: A gate G implements a Boolean function $f : \{0,1\}^2 \rightarrow \{0,1\}$ with propagation delay t_{pd} if:

$$x_1, x_2 \text{ stable in } [t_1, t_2]$$
$$\Rightarrow$$
$$G \text{ is consistent with } f \text{ in the interval } [t_1 + t_{pd}, t_2].$$

remarks:

- If $t_2 < t_1 + t_{pd}$, then definition is empty.
- If $x_i(t)$ is not stable in (t_2, t_3), then $y(t)$ need not remain stable after t_2.

Contamination delay

Contamination delay - the amount of time that the output of a consistent gate remains stable after its inputs stop being stable.

Contamination delay tells us how fast an output "reacts" to a change in the input

We we will assume that the contamination delay is zero.

Combinational circuits - building blocks

Combinational circuits are built of gates and wires & nets.

- Gates: inverter (NOT-gate), OR-gate, NOR-gate, AND-gate, NAND-gate, XOR-gate, NXOR-gate, multiplexer (MUX). (all have fan-in ≤ 3)
- Wires & Nets: connect between terminals of gates.

Wires & Nets

Wires connect points to each other. Very often we need to connect several terminals (i.e. inputs and outputs of gates) together.

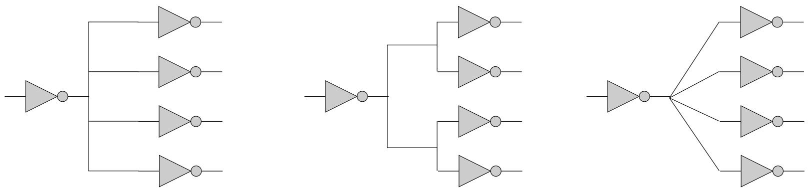
Ignore how connections are actually made.

Net - subset of terminals that are connected by wires. In the digital abstraction we assume that the signals all over a net are identical (why?).

fan-out of a net N - the number of input terminals that are connected by N.

Drawing nets

Three different drawings of the same net (of fan-out 4). We may draw a net in any way that we find convenient or aesthetic. The interpretation of the drawing is that terminals that are connected by lines or curves constitute a net.



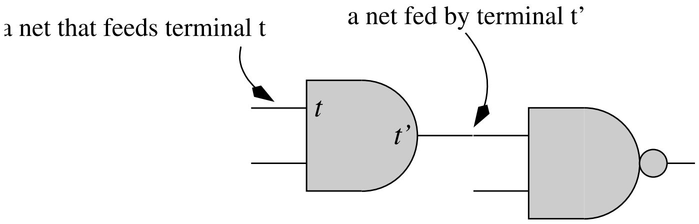
Digital signals for nets

We would like to define the digital signal $N(t)$ for a whole net N.
Noise creates different analog signals along the net.
Define $N(t)$ to logical only if there is a consensus among all the digital interpretations of analog signals at different terminals of the net.
In other words:

- $N(t)$ is zero if the digital values of all the analog signals along the net are zero.
- $N(t)$ is one if the digital values of all the analog signals along the net are one.
- If there is no consensus, then $N(t)$ is non-logical.

Directions in nets

A net N feeds an input terminal t if the input terminal t in N.
A net N is fed by an output terminal t if t is in N.



Information is "supplied" by output terminals and is "consumed" by input terminals.

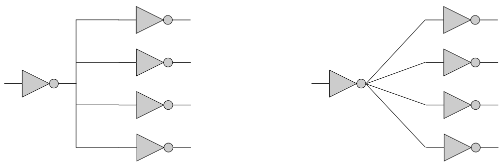
In "pure" CMOS gates, output terminals are connected via resistors either to the ground (low voltage) or to the power (high voltage). Input terminals are connected only to capacitors.

Simple nets

Def: A net N is simple if:

1. N is fed by exactly one output terminal, and
2. N feeds at least one input terminal.

- Consider a simple net $N = \{t_{out}, t_1, t_2, \ldots, t_k\}$, where t_{out} is an output terminal, and $\{t_i\}_{i=1}^k$ are input terminals.
- N can be modeled by a "star" of wires $\{w_i\}_{i \in I}$. Each wire w_i connects t_{out} and t_i. We may regard each wire w_i as a directed edge $t_{out} \rightarrow t_i$.

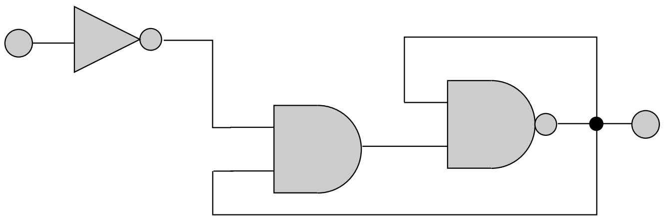


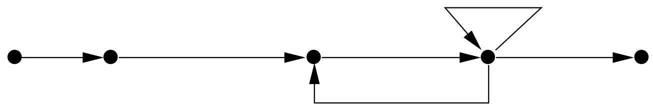
Directed graph corresponding to simple nets

If every every net N in a circuit C is simple, then we can model C by a directed graph.

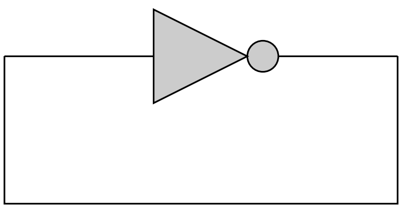
- $DG(C)$ - a directed graph.
- Nodes - gates of C.
- Directed edges - directed edge $u \rightarrow v$ if there is a net N such that: (i) an output terminal of gate u feeds N, and (ii) an input terminal of v is fed by N.

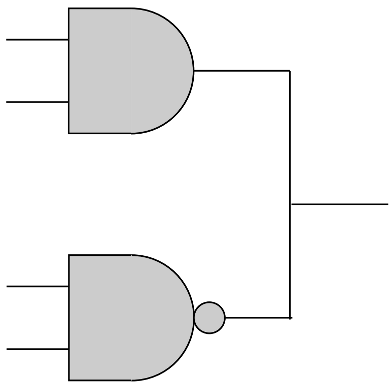
Example of a circuit C and a directed graph $DG(C)$





Are these circuits combinational circuits?





Input gates & output gates

Input and output gates model communication with the “external world”. Solve the problem of “hanging” wires.

Input Gate Output Gate

- input gate - a gate with zero inputs and a single input.
- output gate - a gate with one input and zero outputs.

Syntactic definition of combinational circuits

Def: A combinational circuit is a pair $C = \langle \mathcal{G}, \mathcal{N} \rangle$ that satisfies the following conditions:

1. $\mathcal{G}$ is a set of gates.
2. $\mathcal{N}$ is a set of nets over terminals of gates in $\mathcal{G}$.
3. Every terminal t of a gate $G \in \mathcal{G}$ belongs to exactly one net $N \in \mathcal{N}$.
4. Every net $N \in \mathcal{N}$ is simple.
5. The directed graph $DG(C)$ is acyclic.

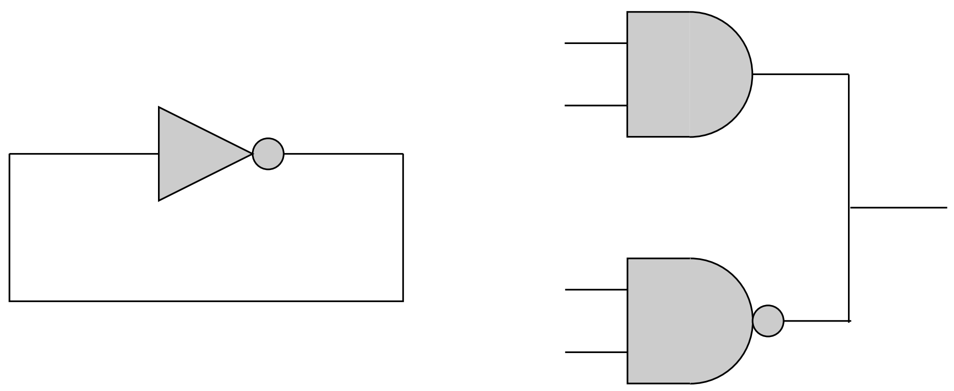
Syntactic definition - remarks

Definition of combinational circuits is independent of the gate types (e.g. inverter, NAND-gate, etc.). The question of whether a circuit is combinational is a purely topological question (i.e. are the interconnections between gates legal?).

syntax - “grammar” rules for forming compound circuits from simple circuits.

Back to “bad” examples...

Which conditions in the syntactic definition of combinational circuits are violated by the “bad” circuits?



homework: design an efficient algorithm to check if a given circuit is combinational.

Combinational circuits: Syntax ⇒ Semantics

- Completeness: for every Boolean function f, there exists a combinational circuit that implements f (exercise).
- Soundness: every combinational circuit implements a Boolean function. (NP-Complete to decide if a given combinational circuit ever outputs a 1.)
- Simulation: given the digital values of the inputs of a combinational circuit, one can simulate the circuit in linear time.
- Delay analysis: given the propagation delays of all the gates in a combinational circuit, one can compute in linear time the propagation delay of the circuit (upper bound).

Simulation theorem of combinational circuits

- $C = \langle \mathcal{G}, \mathcal{N} \rangle$ - a combinational circuit with k input gates.
- $\{x_i\}_{i=1}^k$ - digital input signals
- $[t_1, t_2]$ - a sufficiently long interval of time.

Theorem: If the digital signals $\{x_i(t)\}_{i=1}^k$ are stable during the interval $[t_1, t_2]$, then, for every net $N \in \mathcal{N}$ there exist:

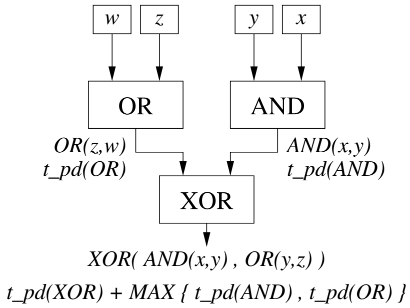
1. a Boolean function $f_N : \{0,1\}^k \rightarrow \{0,1\}$, and
2. a propagation delay $t_{pd}(N)$

such that

$$N(t) = f_N(x_1(t), x_2(t), \ldots, x_k(t)),$$

for every $t \in [t_1 + t_{pd}(N), t_2]$.

Example - simulation of combinational circuit



- process nets according to topological order (i.e. u before v if there is an edge $u \to v$ in $DG(C)$).
- assign Boolean function to each net.
- assign t_{pd} to each net.

Proof of Simulation Theorem

Notation:

- $\vec{x}(t)$ - the vector $x_1(t), \ldots, x_k(t)$.
- $v_1, v_2, \ldots, v_n$ - topological order of vertices (gates) in $DG(C)$.
- WLOG: $v_1, \ldots, v_k$ are the input gates.
- $x_i(t)$ is the digital signal output by v_i (for $1 \le i \le k$).
- $\mathcal{N}_i$ - subset of nets in $\mathcal{N}$ that are fed by gate v_i.
- $e_1, e_2, \ldots, e_m$ - ordering of the nets in $\mathcal{N}$ such that nets in $\mathcal{N}_i$ precede nets in $\mathcal{N}_{i+1}$.
- Note that e_1 is fed by v_1, $\ldots$, e_k is fed by v_k.

Proof - Induction hypothesis

For every $i \le m'$ there exist:

1. a Boolean function $f_{e_i} : \{0,1\}^k \to \{0,1\}$, and
2. a propagation delay $t_{pd}(e_i)$

such that the network e_i implements the Boolean function

$$f_{e_i} : \{0,1\}^k \to \{0,1\}$$

with propagation delay $t_{pd}(e_i)$.

Proof - Induction basis

Instead of proving for $m' = 1$, we prove for $m' = k$.

Consider an $i \le k$. The net e_i is fed by v_i, and the digital signal corresponding to e_i is $x_i(t)$.

$\Longrightarrow$ define

$$f_{e_i}(\sigma_1, \ldots, \sigma_k) = \sigma_i.$$
$$t_{pd}(e_i) = 0.$$

now to induction step...

Proof - Induction step

Ind. Hyp.(m') $\implies$ Ind. Hyp.$(m'+1)$.

Focus on $e_{m'+1}$:

- let v_i denote the gate that feeds $e_{m'+1}$.
- for simplicity: assume that v_i has 2 inputs fed by the nets e_j & e_k, respectively.
- Topological ordering $\Rightarrow j, k \leq m'$.
- Ind. Hyp. $\Rightarrow$:
  - e_j implements a Boolean function f_{e_j} with $t_{pd}(e_j)$.
  - e_k implements a Boolean function f_{e_k} with $t_{pd}(e_k)$.
- $\Rightarrow$ both inputs to gate v_i are stable during the interval

$$[t_1 + \max\{t_{pd}(e_j), t_{pd}(e_k)\}, t_2].$$

Proof - Ind. step - cont.

- Gate v_i implements a Boolean function f_{v_i} with propagation delay $t_{pd}(v_i)$.
- $\Rightarrow$ the output of gate v_i equals

$$f_{v_i}(f_{e_j}(\vec{x}(t)), f_{e_k}(\vec{x}(t)))$$

during the interval

$$[t_1 + \max\{t_{pd}(e_j), t_{pd}(e_k)\} + t_{pd}(v_i), t_2].$$

- Define

$$f_{e_{m'+1}}(\vec{\sigma}) = f_{v_i}(f_{e_j}(\vec{\sigma}), f_{e_k}(\vec{\sigma})).$$
$$t_{pd}(e_{m'+1}) = \max\{t_{pd}(e_j), t_{pd}(e_k)\} + t_{pd}(v_i).$$

QED

Simulation theorem - Corollaries

- simulation algorithm
- timing analysis algorithm
- may regard a combinational circuit as a “macro-gate”. All instances of the same combinational circuit implement the same Boolean function and have the same propagation delay.

very simple algorithms...

Simulation and timing-analysis algorithm

- construct the directed graph $DG(C)$.
- sort gates in topological order .
- order the nets $e_1, e_2, \ldots, e_m$.
- For $i = 1$ to m do:
  - Let v_j denote the gate that feeds e_i.
  -

$$val(e_i) \leftarrow f_{v_j}\left(\{val(e_k)\}_{e_k \text{ feeds } v_j}\right)$$

$$t_{pd}(e_i) \leftarrow t_{pd}(v_j) + \max\{t_{pd}(e_k)\}_{e_k \text{ feeds } v_j}.$$

Complexity: linear if each gate has a single output terminal.

(why?)

Quality measures of combinational gates

- Suppose C_1 and C_2 are combinational circuits that compute the same Boolean function. How do we decide which one is better? We use two criteria:
- Cost
- Propagation delay

Cost

We associate a cost with every gate. We denote the cost of a gate G by $c(G)$.

Def: The cost of a combinational circuit $C = \langle \mathcal{G}, \mathcal{N} \rangle$ is defined by

$$c(C) \triangleq \sum_{G \in \mathcal{G}} c(G).$$

Propagation delay

We associate a propagation delay with every gate. We denote the propagation delay of a gate G by $t_{pd}(G)$.

Def: The propagation delay of a combinational circuit $C = \langle \mathcal{G}, \mathcal{N} \rangle$ is defined by

$$t_{pd}(C) \triangleq \max_{N \in \mathcal{N}} t_{pd}(N).$$

We often refer to the propagation delay of a combinational circuit as its depth or simply its delay.

Delays of paths

- path - a sequence $p = \{v_0, v_1, \ldots, v_k\}$ of gates that form a path in the directed graph $DG(C)$.
- delay of a path p -

$$t_{pd}(p) = \sum_{v \in p} t_{pd}(v).$$

Claim:

$$t_{pd}(C) = \max\{t_{pd}(p) : \text{paths } p\}.$$

critical path - a path p that satisfies $t_{pd}(p) = t_{pd}(C)$.

Q: Number of paths can be exponential. How were we able to compute $max\{t_{pd}(p) : \text{paths } p\}$?

Example: gate costs and delays

Müller and Paul compiled the following costs and delays of gates. These figures were obtained by considering ASIC libraries of two technologies and normalizing them with respect to the cost and delay of an inverter.

Gate	Motorola		Venus	
	cost	delay	cost	delay
INV	1	1	1	1
AND,OR	2	2	2	1
NAND, NOR	2	1	2	1
XOR, NXOR	4	2	6	2
MUX	3	2	3	2

Syntax & Semantics

- semantics - function that a circuit implements. Also called functionality or even the behavior of the circuit.

  In general, formal description that relates

  $$\text{input values} \longmapsto \text{output values}.$$

  In non-combinational circuits, the output depends not only on the current inputs, so semantics cannot be described simply by a Boolean function.

- syntax - a formal set of rules that govern how "grammatically correct" circuits are constructed from smaller circuits (just as sentences are built of words).
  - syntax $\not\Longrightarrow$ useful circuit (e.g. adder).
  - syntax $\Longrightarrow$ well defined functionality, simple simulation, & simple timing analysis.

Summary

- gates - implement simple Boolean functions
- nets & wires - used to connect terminals of gates
- formal (syntactic) definition of combinational gates
- combinational gates are easy to:
  - recognize
  - simulate
  - analyze (propagation delay)
- quality criteria: cost & delay

Chapter 3: Trees

Computer Structure
&
Intro. to Digital Computers

Dr. Guy Even

Tel-Aviv Univ.

Goals

- define associative Boolean functions (and classify them).
- trees - combinational circuits that implement associative Boolean funcs.
- analyze delay & cost of trees.
- prove optimality.

Associative dyadic boolean functions

Def: A Boolean function $f : \{0,1\}^2 \rightarrow \{0,1\}$ is associative if

$$f(f(\sigma_1, \sigma_2), \sigma_3) = f(\sigma_1, f(\sigma_2, \sigma_3)),$$

for every $\sigma_1, \sigma_2, \sigma_3 \in \{0,1\}$.

Q: List all the associative Boolean functions

$$f : \{0,1\}^2 \rightarrow \{0,1\}.$$

"A": There are 16 dyadic Boolean functions, only need to list them and check...

f_n : repeating $f : \{0,1\}^2 \to \{0,1\}$

Def: Let $f : \{0,1\}^2 \to \{0,1\}$ denote a Boolean function. The function $f_n : \{0,1\}^n \to \{0,1\}$, for $n \geq 2$ is defined by induction as follows.

1. If $n = 2$ then $f_2 \equiv f$.
2. If $n > 2$, then f_n is defined based on f_{n-1} as follows:

$$f_n(x_1, x_2, \ldots x_n) \triangleq f(f_{n-1}(x_1, \ldots, x_{n-1}), x_n).$$

Example:

$$\text{NOR}_4(x_1, x_2, x_3, x_4) = \text{NOR}(\text{NOR}(\text{NOR}(x_1, x_2), x_3), x_4).$$

Note that NOR is not associative!

f_n : the associative case

If $f(x_1, x_2)$ is associative, then parenthesis are not important.

Claim: If $f : \{0,1\}^2 \to \{0,1\}$ is an associative Boolean function, then

$$f_n(x_1, x_2, \ldots x_n) = f(f_k(x_1, \ldots, x_k), f_{n-k}(x_{k+1}, \ldots, x_n)),$$

for every $k \in [2, n-2]$.

Q: Show that the set of functions $f_n(x_1, \ldots, x_n)$ that are induced by associative dyadic Boolean functions is

$$\{\text{constant } 0, \text{constant } 1, x_1, x_n, \text{AND}, \text{OR}, \text{XOR}, \text{NXOR}\}.$$

note: only last 4 functions are “interesting”. We focus on OR.

Definition of OR-trees

Def: A combinational circuit $C = \langle \mathcal{G}, \mathcal{N} \rangle$ that satisfies the following conditions is called an OR-tree(n).

1. **Input:** $x[n-1:0]$.
2. **Output:** $y \in \{0, 1\}$
3. **Functionality:** $y = \text{OR}(x[0], x[1], \cdots, x[n-1])$.
4. **Gates:** All the gates in $\mathcal{G}$ are OR-gates.
5. **Topology:** The underlying graph of $DG(C)$ (i.e. undirected graph obtained by ignoring edge directions) is a binary tree.

   Note that in the tree:
   - root - the output y
   - leaves - the inputs $x[n-1:0]$
   - interior nodes - OR-gates.

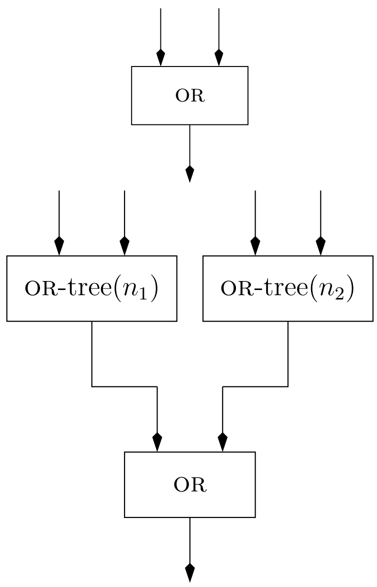
Recursive definition of OR-trees

Def: an OR-tree(n) is defined recursively as follows:

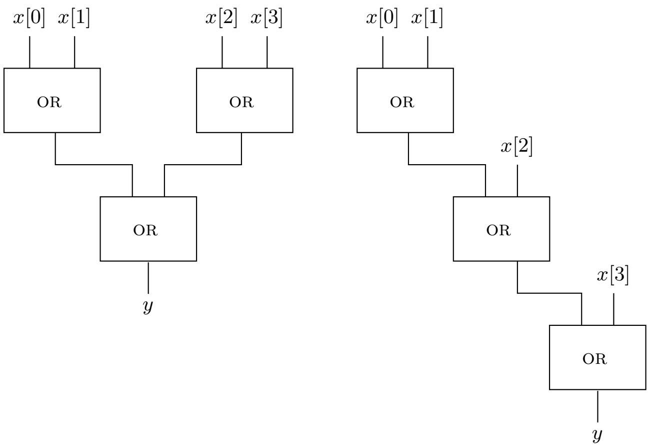
basis: a single OR-gate is an OR-tree(2).

step: an OR(n)-tree is a circuit in which

1. the output is computed by an OR-gate.
2. the inputs of this OR-gate are the outputs of OR-tree(n_1) & OR-tree(n_2), where $n = n_1 + n_2$.



Example: OR-tree(4)



Cost - both trees have 3 gates.

Delay - 2 gates vs. 3.

Cost of OR-trees

Claim: The cost of every OR-tree(n) is $(n-1) \cdot c(\text{OR})$.

Proof: By induction on n.

Induction basis: $n = 2$. In this case, OR-tree(2) contains a single OR-gate.

Cost of OR-trees - Induction step

- let C denote an OR-tree(n).
- let g denote the OR-gate that outputs the output of C.
- g is fed by two wires e_1 and e_2.
- e_1 is the output of C_1 - an OR-tree(n_1)
- e_2 is the output of C_2 - an OR-tree(n_2)
- $n_1 + n_2 = n$
- Ind. Hyp. $\Rightarrow c(C_1) = (n_1 - 1) \cdot c(\text{OR})$ & $c(C_2) = (n_2 - 1) \cdot c(\text{OR})$.
- $$\begin{aligned} c(C) &= c(g) + c(C_1) + c(C_2) \\ &= (1 + n_1 - 1 + n_2 - 1) \cdot c(\text{OR}) \\ &= (n-1) \cdot c(\text{OR}). \end{aligned}$$

QED

Delay of OR-trees

Claim: The delay of a balanced OR-tree(n) is

$$\lceil \log_2 n \rceil \cdot t_{pd}(\text{OR}).$$

Proof: homework. Note that the term “balanced tree” can be interpreted in more than one way if n is not a power of 2...

Are balanced OR-trees optimal?

- What is the best (min. cost & delay) choice of a topology for a combinational circuit that implements the Boolean function OR_n? Is a tree indeed the best topology?
- Could one do better if another implementation is used?

Optimality of balanced OR-trees

Would like to prove that every combinational circuit C that implements OR_n satisfies:

$$c(C) \geq n - 1$$
$$t_{pd}(C) \geq \log_2 n.$$

We need to be more accurate about the model:

Q: what is the cost/delay of an n-input OR-gate?

assumption: the fan-in of every gate ≤ 2, so we have to build big gates from basic gates.

assumption: the cost of every basic gate is ≥ 1. (input/output gates are free)

Optimality of balanced OR-trees

Would like to prove that every combinational circuit C that implements OR_n satisfies:

$$c(C) \geq n - 1$$
$$t_{pd}(C) \geq \log_2 n.$$

Looking for proof also for the case that $DG(C)$ is not a tree!

Restriction of a Boolean function

Def: Let $f : \{0,1\}^n \to \{0,1\}$ denote a Boolean function. Let $\sigma \in \{0,1\}$. The Boolean function $g : \{0,1\}^{n-1} \to \{0,1\}$ defined by

$$g(w_0, \ldots, w_{n-2}) \triangleq f(w_0, \ldots, w_{i-1}, \sigma, w_i, \ldots, w_{n-2})$$

is called the restriction of f with $x_i = \sigma$. We denote it by $f_{\restriction x_i = \sigma}$.

Examples:

$$\text{XOR}_{\restriction x_2=1}(x_1) \triangleq \text{XOR}(x_1, 1)$$

$$\text{MAJORITY}_{\restriction x_n=1}(x_1, \ldots, x_{n-1}) \triangleq \begin{cases} 1 & \text{if } \sum_{i=1}^{n-1} x_i + 1 > n/2 \\ 0 & \text{otherwise.} \end{cases}$$

Cone of a Boolean function

A boolean function $f : \{0,1\}^n \to \{0,1\}$ depends on its ith input if

$$f_{\restriction x_i=0} \not\equiv f_{\restriction x_i=1}.$$

Def: The cone of a Boolean function f is defined by

$$\textit{cone}(f) \triangleq \{i : f \text{ depends on its } i\text{th input}\}.$$

Claim: The Boolean function OR_n depends on all its inputs, namely

$$|\textit{cone}(\text{OR}_n)| = n.$$

Input-Output reachability

Claim: If a combinational circuit C implements a Boolean function f, then there must be a path in $DG(C)$ from every input in *cone*(f) to the output of f.

Proof: by contradiction,

- assume $i \in \textit{cone}(f)$.
- let $g_i \in \mathcal{G}$ denote the input gate that feeds the ith input.
- assume that in $DG(C)$ there is no path from g_i to the output y.
- show that C does not implement f.

Input-Output reachability - cont.

Find vectors $w', w'' \in \{0,1\}^n$ such that

$$f(w') \neq f(w'')$$
$$w'[i] \neq w''[i].$$

Proof of Simulation Theorem
$\Rightarrow C$ outputs the same value in y when input w' and w''.

$\Rightarrow C$ errs either with w' or with w''. QED

Linear Cost Lower Bound Theorem

assumptions:

- fan-in of every gate at most 2.
- cost of trivial gates (i.e. input/output gates) is zero.
- cost of non-trivial gate is at least 1.

Theorem: If C is a combinational circuit that implements a Boolean function f, then

$$c(C) \geq |\textit{cone}(f)| - 1.$$

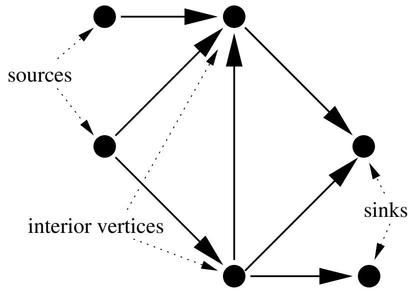
Corollary: If C_n is a combinational circuit that implements OR_n, then $c(C_n) \geq n - 1$.

Easy to prove theorem for trees, but what about arbitrary DAGs?

DAG terminology

Consider the directed acyclic graph (DAG) $DG(C)$.

- in-degree of a vertex v - $\deg_{in}(v)$ is the number of edges that enter the vertex v.
- out-degree of a vertex v - $\deg_{out}(v)$ is the number of edges that emanate from the vertex v.
- source - a vertex with in-degree zero.
- sink - a vertex with out-degree zero.
- interior vertex - a vertex that is neither a source or a sink.



Proof of linear cost lower bound theorem

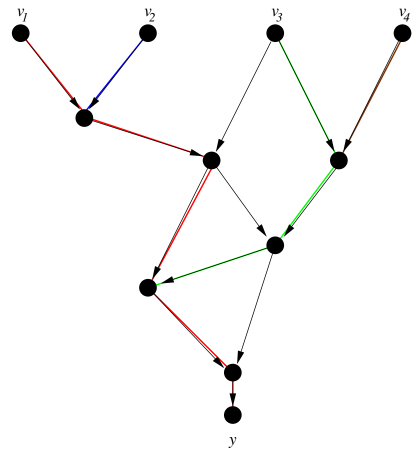
- If underlying graph of $DG(C)$ is a binary tree, then follows from

$$|\text{interior nodes}| \geq |\text{leaves}| - 1.$$

- If $DG(C) = (V, E)$ is not a tree, then construct a directed "binary tree" $T = (V', E')$ such that
  - $V' \subseteq V$ & $E' \subseteq E$
  - sources$(T') = cone(f)$
  - output gate $\in V'$.
- in T' we have $|\text{interior nodes}| \geq |\text{sources}| - 1$.
- But interior nodes of T are also interior in $DG(C)$, and number of sources in T equals $|cone(f)|$. QED.

Left to show how T is constructed...

Construction of T



larger fan-in

Q: Generalize the lower bound on the cost to the case that the fan-in of every gate is bounded by a constant c.

Logarithmic Delay Lower Bound Theorem

Theorem: Let $C = \langle \mathcal{G}, \mathcal{N} \rangle$ denote a combinational circuit that implements a non-constant Boolean function $f\{0,1\}^n \to \{0,1\}$. If the fan-in of every gate in $\mathcal{G}$ is at most c, then the delay of C is at least $\log_c |\textit{cone}(f)|$.

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

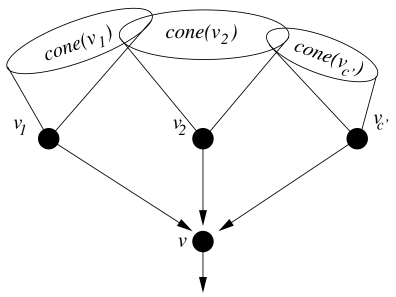
$$t_{pd}(C_n) \geq \lceil \log_c n \rceil .$$

Proof of logarithmic lower bound

- deal only with the graph $DG(C)$.
- show that exists a path with at least $\log_c |\textit{cone}(f)|$ interior vertices in $DG(C)$.
- why interior?
- input/output gates and constants have zero delay $\Rightarrow$ should not be counted.
- only sources & sinks have zero delay $\Rightarrow$ count interior vertices.
- *cone*(v) - set of sources from which v is reachable. Note that $|\textit{cone}(output)| = |\textit{cone}(f)|$.
- $d(v)$ - max number of interior vertices along a path from a source in *cone*(v) to v (including v).
- suffice to prove that $d(v) \geq \log_c |\textit{cone}(v)|$.

Proof: $d(v) \geq \log_c |cone(v)|$

- Proof by induction on $d(v)$.
- Basis: $d(v) = 0$. In this case v is a source, $|\mathit{cone}(v)| = 1$.
- Step: $d(v) = i + 1$. Edges entering v are $v_1 \to v, \ldots, v_{c'} \to v$, for $c' \leq c$.
- by def: $d(v) = \max\{d(v_i)\}_{i=1}^{c'} + 1$.
- $\mathit{cone}(v) = \bigcup_{i=1}^{c'} \mathit{cone}(v_i)$.



$$\begin{aligned} |\mathit{cone}(v)| &\leq \sum_{i=1}^{c'} |\mathit{cone}(v_i)| \\ &\leq c' \cdot \max\{|\mathit{cone}(v_i)|\}_{i=1}^{c'}. \end{aligned}$$

Cont. proof: $d(v) \geq \log_c |cone(v)|$

Let v' denote a predecessor of v that satisfies

$$|\mathit{cone}(v')| = \max\{|\mathit{cone}(v_i)|\}_{i=1}^{c'} \geq |\mathit{cone}(v)|/c'.$$

The induction hypothesis implies that

$$d(v') \geq \log_c |\mathit{cone}(v')|.$$

But,

$$\begin{aligned} d(v) &\geq 1 + d(v') \\ &\geq 1 + \log_c |\mathit{cone}(v')| \\ &\geq \log_c c + \log_c |\mathit{cone}(v)|/c' \\ &\geq \log_c |\mathit{cone}(v)|. \end{aligned}$$

Cont. proof: $d(v) \geq \log_c |\boldsymbol{cone}(v)|$

- Finally, we deal with the case that v is a sink.
- v has a unique predecessor v'.
- we have $d(v) = d(v')$ and *cone*$(v) =$ *cone*(v').
- induction step applies to v', and hence we have

$$d(v) \geq \log_c |\textit{cone}(v)|,$$

  as required.

Summary

- associative Boolean functions.
- extend dyadic functions to functions with n arguments.
- only four non-trivial associative Boolean functions.
- OR-tree(n) - combinational circuits that implement OR$_n$ using a topology of a tree.
- cost(OR-tree) $= n - 1$.
- t_{pd}(balanced OR-tree) $= \log_2 n$.
- Balanced OR-trees optimal cost & delay.
- two lower bounds:
  - cost $\geq |cone(f)| - 1$.
  - $t_{pd} \geq \log_c |cone(f)|$.

Chapter 4: Decoders & Encoders

Computer Structure
&
Intro. to Digital Computers

Dr. Guy Even

Tel-Aviv Univ.

Goals

- vector notation: buses, indexed signals, multiple copies of gates
- representation: binary representation
- Decoders:
  - definition (specification)
  - implementation
  - correctness proof
  - analyze delay & cost
  - optimality
- Encoders: definition (specification), correctness proof, analyze delay & cost, optimality

Bus related definitions

- bus - an indexed set of buses (e.g. $a[1:4]$).
- reversing - reversing the indexes of buses: e.g. if reversing is used, then $b[4:1] \leftarrow a[1:4]$ means $b[1] \leftarrow a[4], \ldots, b[4] \leftarrow a[1]$.
- hardwired shifting - shifting of indexes: e.g. hardwired shifting implies that $b[5:8] \leftarrow a[1:4]$ means $b[5] \leftarrow a[1], \ldots, b[8] \leftarrow a[4]$.

Bus assignment conventions

- Unless stated explicitly, reversing is not used. The assignments:

  $$b[i:j] \leftarrow a[i:j] \quad \& \quad b[j:i] \leftarrow a[i:j]$$

  have the same meaning (so we don't have to worry about ascending/descending indexes).
- Hardwired-shifting is used for shifted index ranges. For example,

  $$b[i+5:j+5] \leftarrow a[i:j]$$

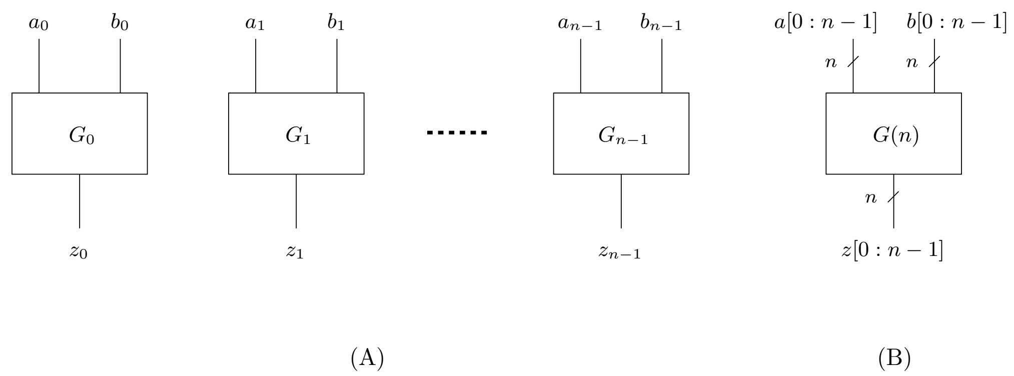
  means

  $$b[i+5] \leftarrow a[i] \quad b[i+6] \leftarrow a[i+1] \quad \cdots \quad b[j+5] \leftarrow a[j]$$

Signals on buses

- Consider a bus $a[5:1]$.
- The signal on $a[3]$ at time t is $a[3](t)$.
- Abbreviate and simply write $a[3]$ as the signal on $a[3]$ after it stabilizes.
- Similarly: $a[5:1]$ refers both to the bus and to the stable signal on the bus.
- $\Longrightarrow a[5:1]$ is both a bus and a binary string.
- Abbreviate $a[5:1]$ and write $\vec{a}$ if index range is clear from context.

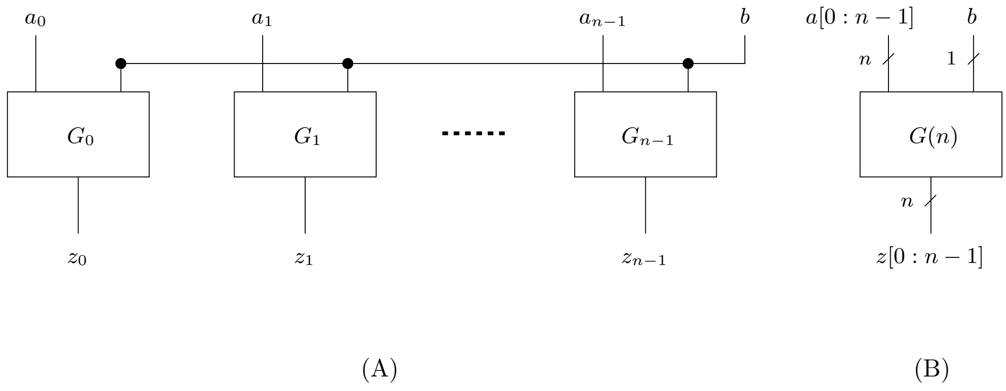
multiple instances of the same gate



- G_i - the ith instance of gate G in $G(n)$.
- a_i, b_i - the two input terminals of G_i.
- z_i - the output terminal of G_i.

Common input in $G(n)$

b is fed to the second input terminal of all the gates.



Fanout of the net b is n. In practice, a large fanout increases the capacity of a net and causes an increase in the delay of the circuit. We usually ignore this phenomena in this course.

Concatenation of binary strings

- assume that a and b are binary strings.
- $a \cdot b$ - the string obtained by concatenating a and b.
- example: if $a = 01$ and $b = 10$, then $a \cdot b = 0110$.
- a^i - the string $\underbrace{a \cdot a \cdots a}_{i \text{ times}}$.
- example: if $a = 01$ and $i = 3$, then $a^i = 010101$.
- If A is a set of strings, then A^i is the set of strings

$$A^i = \{a_1 \cdots \cdot a_i : \forall j \leq i : a_j \in A\}.$$

- example: $\{0,1\}^n$ - set of binary strings of length n.
- $A^* \triangleq \cup_{i=0}^{\infty} A^i$ ($A^0 = \{\Lambda\}$, not the empty set).
- $A^+ \triangleq \cup_{i=1}^{\infty} A^i$.

Values represented by binary strings

Binary strings can be used to represent numbers. Among the many methods for representing natural numbers:

- Binary representation
- Unary representation
- 1-out-of-n (one-hot).

Binary representation

The value represented in binary representation by a binary string $a[n-1:0]$ is denoted by $\langle a[n-1:0]\rangle$. It is defined as follows

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

- Could define a function $\langle\rangle_n : \{0,1\}^n \to [0, 2^n - 1]$.
- Usually omit the parameter n - clear from context.

Inverse function *bin* $: [0, 2^n - 1] \to \{0,1\}^n$.

$$\forall a[n-1:0] \in \{0,1\}^{:} \quad \textit{bin}_n(\langle a[n-1:0]\rangle) = a[n-1:0].$$

Division by 2^k in binary representation

- reminder: division by b with remainder:

$$x = q \cdot b + r, \quad \text{where } r \in [0, b-1].$$

  - q - quotient equals $\lfloor x/b \rfloor$
  - r - remainder equals $\text{mod}(x, b)$.
- Consider a binary string $a[n-1:0]$ and $x = \langle a[n-1:0] \rangle$.
- How do we compute (the binary representation of) $\lfloor x/2^k \rfloor$ & $\text{mod}(x, 2^k)$?
  - $r = \langle a[k-1:0] \rangle$
  - $q = \langle a'[n-k-1:0] \rangle$, where $a'[n-k-1:0] \leftarrow a[n-1:k]$ (with shifting).

Decoders

A decoder with input length n is a combinational circuit specified as follows:

Input: $x[n-1:0] \in \{0,1\}^n$.

Output: $y[2^n - 1:0] \in \{0,1\}^{2^n}$

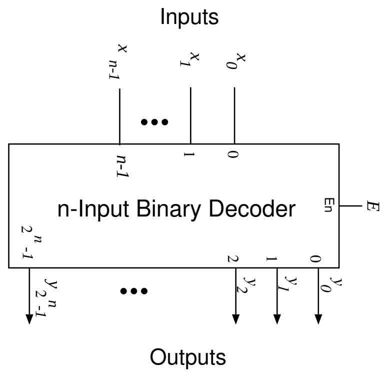
Functionality:

$$y[i] = 1 \iff \langle \vec{x} \rangle = i.$$

- $|$outputs$|$ of a decoder is exponential in $|$inputs$|$.
- exactly one bit of the output $\vec{y}$ is set to one. (called also one-hot encoding or 1-out-of-k encoding.)
- DECODER(n) - a decoder with input length n

example: DECODER(3) - on input $x = 101$, the output y equals 00100000.

Schematic symbol



from: Introduction to Digital Systems, M.D. Ercegovac, T. Lang, and J.H. Moreno, Wiley and Sons, 1998. – p.111

Example: DECODER(3) - a truth table...

EXAMPLE 9.1: 3-INPUT BINARY DECODER

E	x_2	x_1	x_0	x	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
1	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	1	0	0	0	0	0	0	1	0
1	0	1	0	2	0	0	0	0	0	1	0	0
1	0	1	1	3	0	0	0	0	1	0	0	0
1	1	0	0	4	0	0	0	1	0	0	0	0
1	1	0	1	5	0	0	1	0	0	0	0	0
1	1	1	0	6	0	1	0	0	0	0	0	0
1	1	1	1	7	1	0	0	0	0	0	0	0
0	-	-	-	-	0	0	0	0	0	0	0	0

BINARY SPECIFICATION:

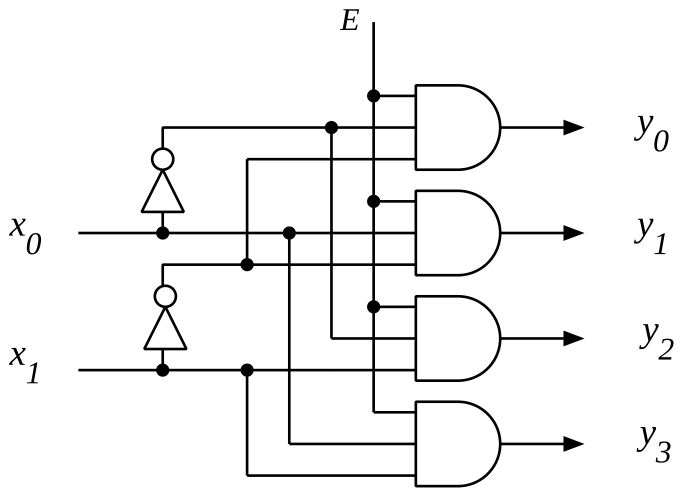
Inputs: $\underline{x} = (x_{n-1}, \ldots, x_0), \quad x_j \in \{0, 1\}$
$E \in \{0, 1\}$

Outputs: $\underline{y} = (y_{2^n-1}, \ldots, y_0), \quad y_i \in \{0, 1\}$

Function: $y_i = E \cdot m_i(\underline{x}) \quad , \quad i = 0, \ldots, 2^n - 1$

from: Introduction to Digital Systems, M.D. Ercegovac, T. Lang, and J.H. Moreno, Wiley and Sons, 1998. – p.112

Example: DECODER(2) - schematic



from: Introduction to Digital Systems, M.D. Ercegovac, T. Lang, and J.H. Moreno, Wiley and Sons, 1998. – p.113

An application

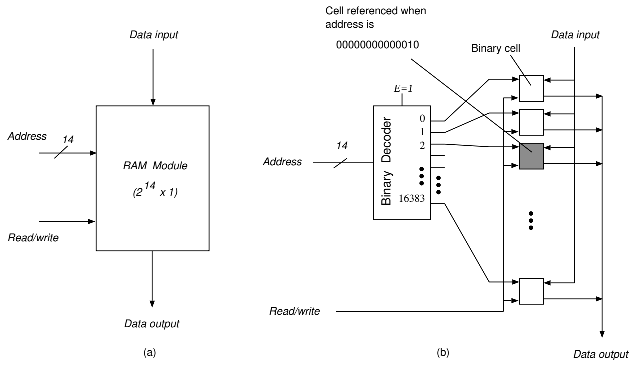
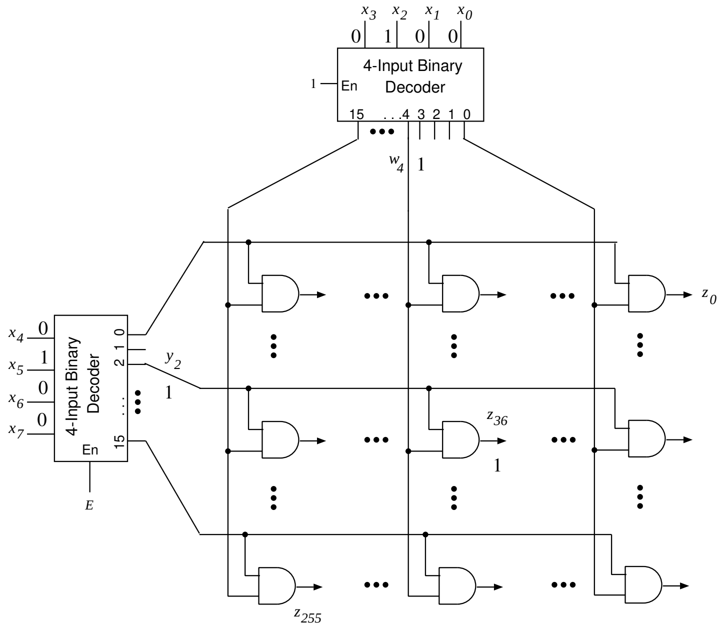


Figure 9.4: RANDOM ACCESS MEMORY (RAM): a) MODULE; b) ADDRESSING OF BINARY CELLS.

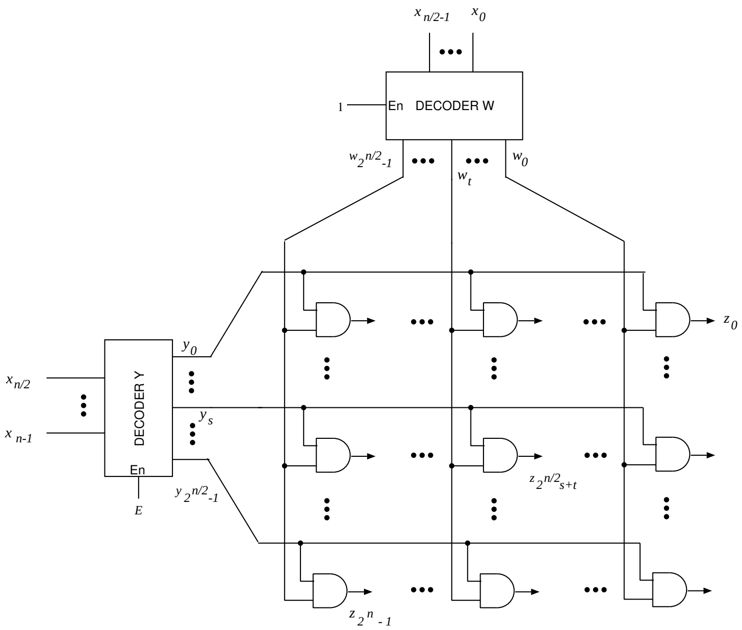
from: Introduction to Digital Systems, M.D. Ercegovac, T. Lang, and J.H. Moreno, Wiley and Sons, 1998. – p.114

Example: DECODER(8) - schematic



from: Introduction to Digital Systems, M.D. Ercegovac, T. Lang, and J.H. Moreno, Wiley and Sons, 1998. – p.115

DECODER(n) - schematic



from: Introduction to Digital Systems, M.D. Ercegovac, T. Lang, and J.H. Moreno, Wiley and Sons, 1998. – p.116

Any questions?

- Why is the design correct?
- Can you prove it?
- Can't prove it unless the design is formally defined...
- Is the design (asymptotically) optimal?
- Why partition into $n/2$ & $n/2$? Is that the best partition?

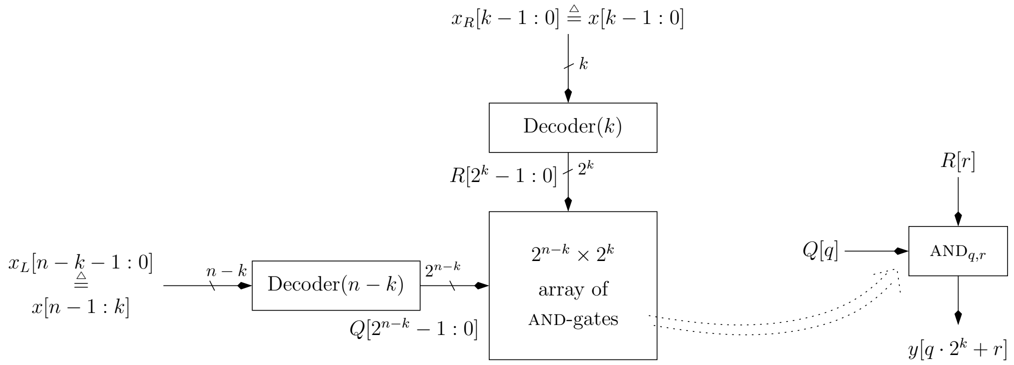
Formal description of DECODER(n)

Use recursion.

Basis: DECODER(1): is simply one inverter where:

$$y[0] \leftarrow \text{INV}(x[0])$$
$$y[1] \leftarrow x[0].$$

Formal description of DECODER(n) - recursive step



Correctness proof

The proof is by induction.

Induction hypothesis for n: if DECODER(n) is input $x[n-1:0]$, then the output $y[2^n-1:0]$ satisfies:

$$\forall 0 \le i < 2^n : \quad y[i] = 1 \quad \Longleftrightarrow \quad \langle x[n-1:0] \rangle = i.$$

Induction basis: $n = 1$: trivial.

Induction step: show that:

$$[\forall n' < n : \quad \text{Ind. Hyp. } (n')] \quad \Longrightarrow \quad \text{Ind. Hyp. } (n).$$

induction step - cont.

- Fix $i \in [2^n - 1 : 0]$.
- Divide by 2^k: $\quad i = q \cdot 2^k + r$.
- $q = \langle x_L[n-k-1:0] \rangle$ & $r = \langle x_R[k-1:0] \rangle$.
- Ind. Hyp. $(k) \Rightarrow$

$$R[r] = 1 \iff \langle x_R[k-1:0] \rangle = r.$$

- Ind. Hyp. $(n-k) \Rightarrow$

$$Q[q] = 1 \iff \langle x_L[n-k-1:0] \rangle = q$$

- $\Longrightarrow$

$$\begin{aligned} y[i] = 1 &\iff R[r] = 1 \text{ and } Q[q] = 1 \\ &\iff \langle x_R[k-1:0] \rangle = r \text{ and } \langle x_L[n-k-1:0] \rangle = q. \\ &\iff \langle x[n-1:0] \rangle = i \quad \text{QED} \end{aligned}$$

Cost analysis

The cost $c(n)$ satisfies the following recurrence equation:

$$c(n) = \begin{cases} c(\text{INV}) & \text{if n=1} \\ c(k) + c(n-k) + 2^n \cdot c(\text{AND}) & \text{otherwise.} \end{cases}$$

It follows that

$$c(n) = c(k) + c(n-k) + \Theta(2^n)$$

- Obviously $c(n) = \Omega(2^n)$.
- Next slide: for every choice of k, $c(n) = O(2^n)$.
- See lecture notes for tight analysis with $k = n/2$.

Cost analysis - cont.

Claim: $c(n) = O(2^n)$.

Proof: Set $\beta = c(\text{AND})$, and
$\alpha = \max\{3\beta, c(1)/2, c(2)/4, c(3)/8\}$.
We prove by induction that $c(n) \leq \alpha \cdot 2^n$. Induction basis follows for $n \leq 3$ by definition of α.

Induction step for $n \geq 4$:

$$\begin{aligned} c(n) &= c(k) + c(n-k) + \beta \cdot 2^n \\ &\leq \alpha \cdot 2^k + \alpha \cdot 2^{n-k} + \beta \cdot 2^n \\ &\leq \alpha \cdot 2^{n-1} + \alpha \cdot 2 + \beta \cdot 2^n \\ &= 2^n \left(\frac{\alpha}{2} + \beta + \frac{2\alpha}{2^n} \right). \end{aligned}$$

But,
$\beta \leq \alpha/3$, and
$n \geq 4 \Rightarrow \frac{2}{2^n} \leq \frac{1}{6}$.
Hence $c(n) \leq \alpha \cdot 2^n$.

Delay analysis

The delay of DECODER(n) satisfies the following recurrence equation:

$$d(n) = \begin{cases} d(\text{INV}) & \text{if n=1} \\ \max\{d(k), d(n-k)\} + d(\text{AND}) & \text{otherwise.} \end{cases}$$

Set $k = n/2$, and it follows that $d(n) = \Theta(\log n)$.

Question: Prove that DECODER(n) is asymptotically optimal with respect to cost and delay.

Weight of binary strings

Def: The weight of a binary string is defined by

$$wt(a[n-1:0]) \triangleq |\{i : a[i] \neq 0\}|.$$

Note that

$$wt(a[n-1:0]) = \sum_{i=1}^{n-1} a[i].$$

example:

$$wt(01101) = 3.$$

Encoders - Definitions

We define the encoder partial function as follows.
Def: The function

$$\text{ENCODER}_n : \{\vec{y} \in \{0,1\}^{2^n} : wt(\vec{y}) = 1\} \rightarrow \{0,1\}^n$$

is defined as follows:

$$wt(y) = 1 \Longrightarrow y[\langle \text{ENCODER}_n(\vec{y}) \rangle] = 1.$$

Examples:

$$\text{ENCODER}_2(0100) = 10$$

$$\text{ENCODER}_2(0101) = \text{undefined}.$$

Encoder - Defs. - cont.

ENCODER(n) - a comb. circuit with 2^n inputs and n outputs that implements the Boolean function ENCODER$_n$.

- Functionality is not defined for all binary strings.
- encoder $=$ decoder^{-1}.
- ENCODER(n) - an encoder with input length 2^n.

Encoders - Defs. - cont.

ENCODER(n) can be also specified as follows:

Input: $y[2^n - 1 : 0] \in \{0,1\}^{2^n}$.

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

Functionality:

$$wt(\vec{y}) = 1 \implies y[\langle \vec{x} \rangle] = 1.$$

If $wt(\vec{y}) \neq 1$, then the output $\vec{x}$ is arbitrary.

examples: Consider an encoder ENCODER(3).
- on input 00100000, the output equals 101.

- on input 01101011, the output is not specified.

Encoder - implementation

Plan:

- Design a simple encoder $\text{ENCODER}'(n)$.
- Prove correctness of $\text{ENCODER}'(n)$.
- Improve $\text{ENCODER}'(n)$ to obtain $\text{ENCODER}^*(n)$.
- Prove functional equivalence

$$\text{ENCODER}^*(n) \equiv \text{ENCODER}'(n),$$

  and hence, correctness of $\text{ENCODER}^*(n)$.
- Prove asymptotic optimality of $\text{ENCODER}^*(n)$.

Example: $\text{ENCODER}(3)$ with enable signal

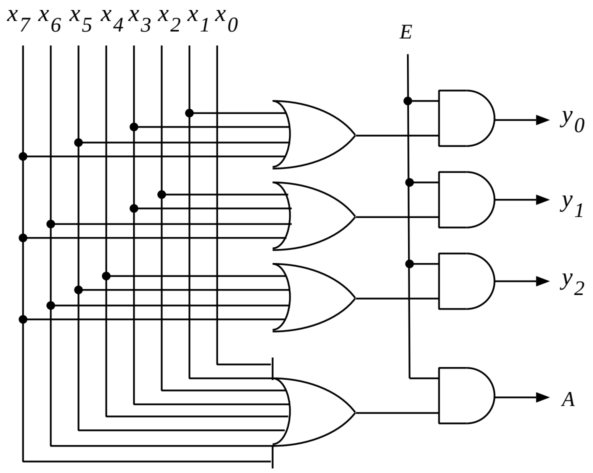


Figure 9.13: IMPLEMENTATION OF AN 8-INPUT BINARY ENCODER.

from: Introduction to Digital Systems, M.D. Ercegovac, T. Lang, and J.H. Moreno, Wiley and Sons, 1998.

$\text{ENCODER}'(n)$

Use recursion.

Basis: $\text{ENCODER}(1)$: is simply $x[0] \leftarrow y[1]$ (zero cost, zero delay).

Recursion step: Partition input $y[2^n - 1 : 0]$

$$y_L[2^{n-1} - 1 : 0] = y[2^n - 1 : 2^{n-1}]$$
$$y_R[2^{n-1} - 1 : 0] = y[2^{n-1} - 1 : 0].$$

Apply $\text{ENCODER}'(n-1)$ to $\vec{y}_L$ and to $\vec{y}_R$.

Problem: if $\vec{y}_L = 0^{2^{n-1}}$, then output of $\text{ENCODER}'(n-1)$ is arbitrary, and design can't be correct...
$\Rightarrow$ Must specify output for an all-zeros input.

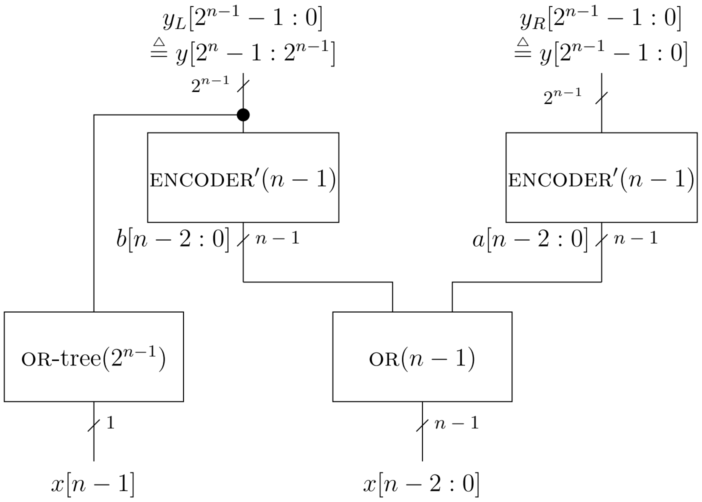
Define $\text{ENCODER}_n(0^{2^n})$

We augment the definition of the ENCODER_n function so that its range also includes the all zeros string 0^{2^n}. We define

$$\text{ENCODER}_n(0^{2^n}) \triangleq 0^n.$$

Note: In $\text{ENCODER}'(1)$: $x[0] \leftarrow y[1]$, so if input is 00 then output is 0. Hence, $\text{ENCODER}'(1)$ also meets this new condition, and the induction basis of the correctness proof holds.

$\textsc{encoder}'(n)$ - recursive step

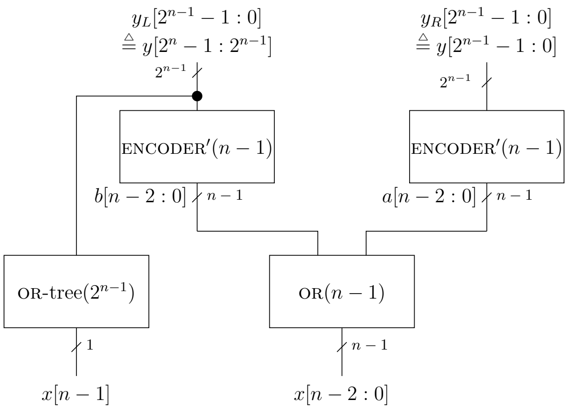


Correctness of $\textsc{encoder}'(n)$ - induction step

Three cases, depending on which "half" of $\vec{y}$ contains a 1.

(1) *wt*$(\vec{y}_L) = 0$ & *wt*$(\vec{y}_R) = 1$:

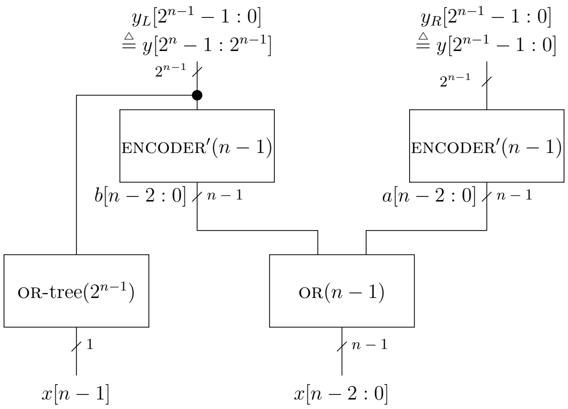
- Ind. Hyp. $\Rightarrow$ $\vec{b} = 0^{n-1}$ & $y_R[\langle\vec{a}\rangle] = 1$.
- $\Rightarrow$ desired output is $\vec{x} = 0 \cdot \vec{a}$.
- actual output is
  $$\vec{x} = \textsc{or}\text{-tree}(\vec{y}_L) \cdot \textsc{or}(\vec{b}, \vec{a})$$
  $$= 0 \cdot \vec{a}.$$



Correctness of ENCODER$'(n)$ - induction step - cont.

(2) *wt*$(\vec{y}_L) = 1$ & *wt*$(\vec{y}_R) = 0$:

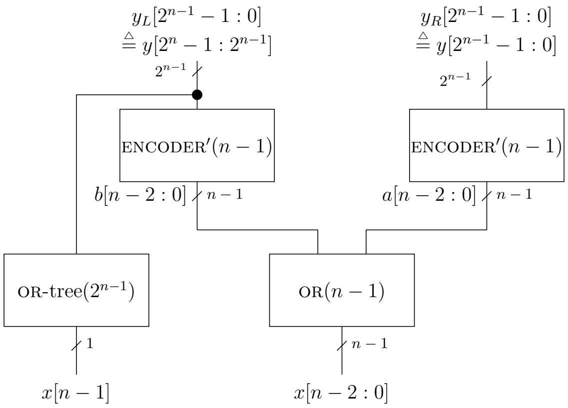
- Ind. Hyp. $\Rightarrow$ $y_L[\langle \vec{b} \rangle] = 1$ & $\vec{a} = 0^{n-1}$.
- $\Rightarrow$ desired output is $\vec{x} = 1 \cdot \vec{b}$.
- actual output is
  $$\vec{x} = \text{OR-tree}(\vec{y}_L) \cdot \text{OR}(\vec{b}, \vec{a})$$
  $$= 1 \cdot \vec{b}.$$



Correctness of ENCODER$'(n)$ - induction step - cont.

(3) *wt*$(\vec{y}_L) = 0$ & *wt*$(\vec{y}_R) = 0$:

- desired output is $\vec{x} = 0^n$.
- Ind. Hyp. $\Rightarrow$ $\vec{b} = 0^{n-1}$ & $\vec{a} = 0^{n-1}$.
- actual output is
  $$\vec{x} = \text{OR-tree}(\vec{y}_L) \cdot \text{OR}(\vec{b}, \vec{a})$$
  $$= 0 \cdot 0^{n-1}.$$



Delay analysis of $\text{ENCODER}'(n)$

Let $d(n)$ denote $t_{pd}(\text{ENCODER}'(n))$.

$$d(n) = \begin{cases} 0 & \text{if n=1} \\ \max\{(n-1)\cdot t_{pd}(\text{OR}), d(n-1) + t_{pd}(\text{OR})\} & \text{otherwise.} \end{cases}$$

Guess the solution: $d(n) = (n-1) \cdot t_{pd}(\text{OR})$.

Cost analysis of $\text{ENCODER}'(n)$

Let $c(n)$ denote $c(\text{ENCODER}'(n))$.

$$c(n) = \begin{cases} 0 & \text{if n=1} \\ 2\cdot c(n-1) + (2^{n-1} - 1 + n - 1)\cdot c(\text{OR}) & \text{otherwise.} \end{cases}$$

Substitute $C(2^n) = c(n)$, and re-write for $N = 2^n$:

$$C(N) = 2 \cdot C(N/2) + \Theta(N).$$

$$\Rightarrow \quad C(N) = \Theta(N \log N)$$

$$\Rightarrow \quad c(\text{ENCODER}'(n)) = \Theta(n \cdot 2^n).$$

Sanity test

$d(\text{ENCODER}'(n)) = \Theta(n)$ & $c(\text{ENCODER}'(n)) = \Theta(n \cdot 2^n)$.

But we can compute each $x[i]$ using a separate OR-tree:

$$x[i] = \text{OR}(\{y[j] : \textit{bin}_n(j)[i] = 1\}).$$

Cost of each tree is $O(2^n)$ and delay is $O(n)$.

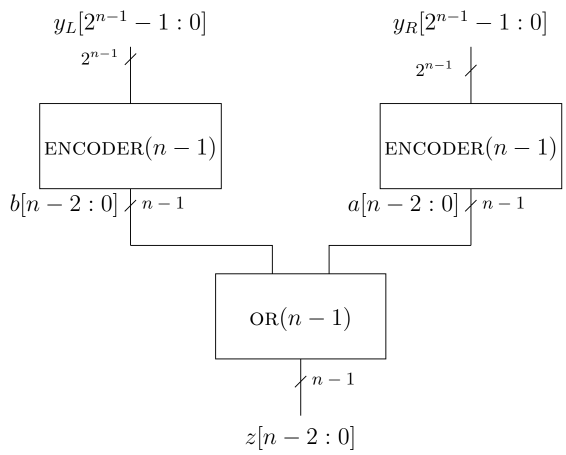
$\Rightarrow$ a trivial design with delay $O(n)$ and cost $O(n \cdot 2^n)$.

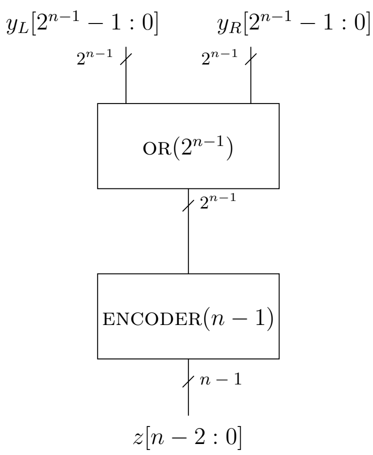
We conclude that $\text{ENCODER}'(n)$ is not a good design... Can we fix it?!

Commuting bitwise-OR and ENCODER_{n-1}

claim: If $\textit{wt}(y[2^n - 1 : 0]) \le 1$, then

$$\text{OR}(\text{ENCODER}_{n-1}(\vec{y}_L), \text{ENCODER}_{n-1}(\vec{y}_R)) = \text{ENCODER}_{n-1}(\text{OR}(\vec{y}_L, \vec{y}_R)).$$





Proof: $\textsc{or}(E_{n-1}(\vec{y}_L), E_{n-1}(\vec{y}_R)) = E_{n-1}(\textsc{or}(\vec{y}_L, \vec{y}_R))$

- Trivial: $\vec{y} = 0^{2^n}$.
- $wt(\vec{y}_L) = 0$ & $wt(\vec{y}_R) = 1$:
  - Assume that $y_R[i] = 1$.
  - $\Rightarrow$

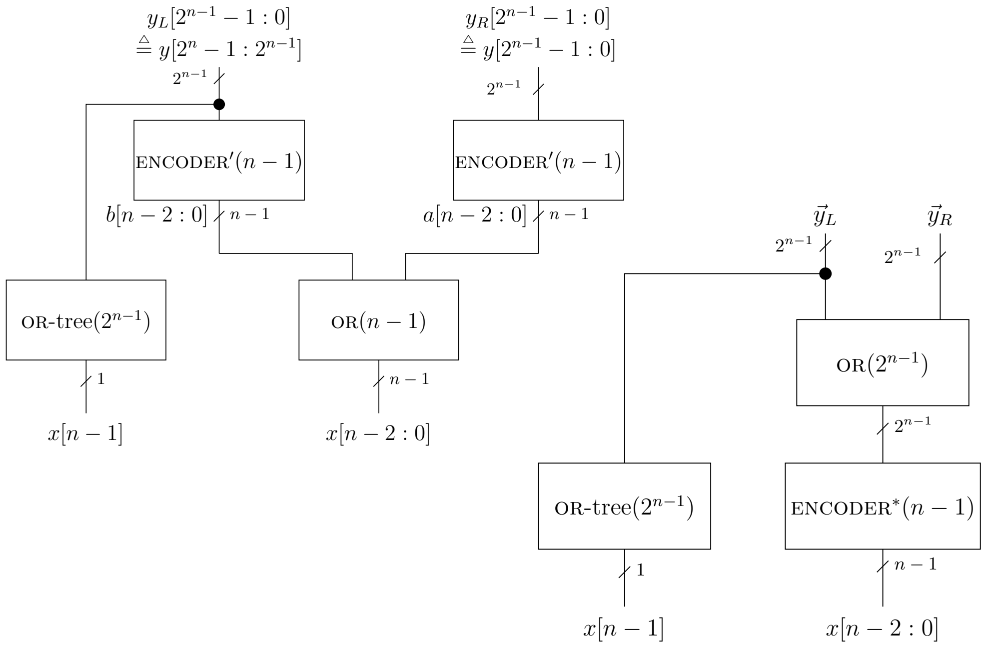
$$\begin{aligned} E_{n-1}(\textsc{or}(\vec{y}_L, \vec{y}_R)) &= E_{n-1}(\textsc{or}(0^{2^{n-1}}, \vec{y}_R)) \\ &= E_{n-1}(\vec{y}_R)). \end{aligned}$$

  - However,

$$\begin{aligned} \textsc{or}(E_{n-1}(\vec{y}_L), E_{n-1}(\vec{y}_R)) &= \textsc{or}(E_{n-1}(0^{2^{n-1}}), E_{n-1}(\vec{y}_R)) \\ &= \textsc{or}(0^{n-1}, E_{n-1}(\vec{y}_R)) \\ &= E_{n-1}(\vec{y}_R), \end{aligned}$$

- (other case is analogous) QED

ENCODER$'(n) \longrightarrow$ ENCODER$^*(n)$



Correctness of $\text{ENCODER}^*(n)$

- No need to prove from "the beginning" (although not a hard task).
- Claim proves that $\text{ENCODER}'(n) \equiv \text{ENCODER}^*(n)$.
- We already proved that $\text{ENCODER}'(n)$ is correct.
- $\Rightarrow$ $\text{ENCODER}^*(n)$ is correct!
- Very useful method for hardware design:
  - start with a naive design (e.g. naive divide & conquer)
  - manipulate design (improve but preserve functionality)

Cost analysis of $\text{ENCODER}^*(n)$

$$c(\text{ENCODER}^*(n)) = \begin{cases} 0 & \text{if n=1} \\ c(\text{ENCODER}^*(n-1)) + 2^n \cdot c(\text{OR}) & \text{otherwise.} \end{cases}$$

We expand this recurrence to obtain:

$$\begin{aligned} c(\text{ENCODER}^*(n)) &= c(\text{ENCODER}^*(n-1)) + 2^n \cdot c(\text{OR}) \\ &= (2^n + 2^{n-1} + \ldots + 4) \cdot c(\text{OR}) \\ &= (2 \cdot 2^n - 3) \cdot c(\text{OR})) \\ &= \Theta(2^n). \end{aligned}$$

Delay analysis of ENCODER$^*(n)$

The delay of ENCODER$^*(n)$ satisfies the following recurrence equation:

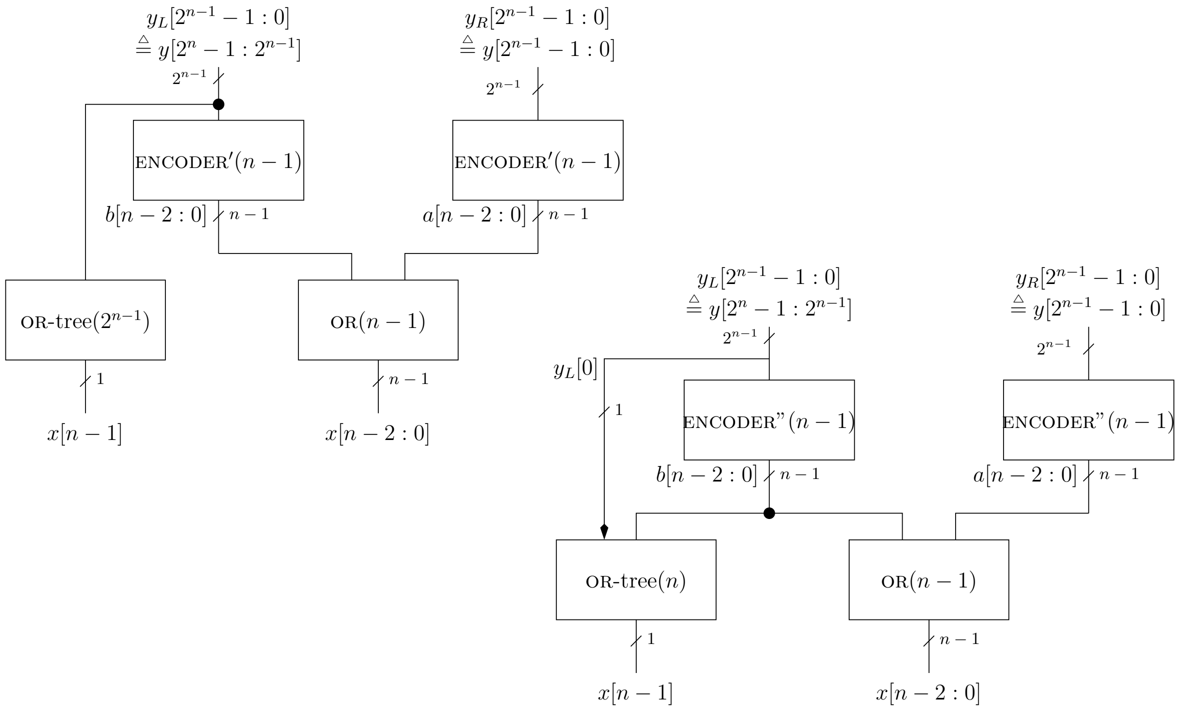
$$d(\text{ENCODER}^*(n)) = \begin{cases} 0 & \text{if n=1} \\ \max\{d(\text{OR-tree}(2^{n-1}), & \\ \quad d(\text{ENCODER}^*(n-1) + d(\text{OR}))\} & \text{otherwise.} \end{cases}$$

Since $d(\text{OR-tree}(2^{n-1}) = (n-1) \cdot d(\text{OR})$, it follows that

$$d(\text{ENCODER}^*(n)) = (n-1) \cdot d(\text{OR}) = \Theta(n).$$

Optimality: Prove that the cost and delay of ENCODER$^*(n)$ are asymptotically optimal.

yet another encoder design



Cost analysis of $\text{ENCODER}''(n)$

We obtain the recurrence equation:

$$c(n) = \begin{cases} 0 & \text{if n=1} \\ 2 \cdot c(n-1) + 2 \cdot (n-1) \cdot c(\text{OR}) & \text{otherwise.} \end{cases}$$

We expand this recurrence to obtain:

$$\begin{aligned} c(n) &= 2 \cdot c(n-1) + 2 \cdot (n-1) \cdot c(\text{OR}) \\ &= c(\text{OR}) \cdot \left(2(n-1) + 2 \cdot 2(n-2) + \cdots 2^{n-i} \cdot 2(i-1) + \cdots + 2^{n-2} \cdot 2\right) \\ &= c(\text{OR}) \cdot 2^n \cdot \left(\frac{1}{2} + \frac{2}{4} + \cdots + \frac{i-1}{2^{(i-1)}} + \cdots + \frac{n-1}{2^{(n-1)}}\right) \\ &\leq c(\text{OR}) \cdot 2^n \cdot 2. \end{aligned}$$

Summary

- vector notation for schematics and binary strings.
- review of binary representation.
- Decoder & Encoder - design, correctness, optimality.
- Techniques:
  - Divide & Conquer
  - Extend specification to make problem easier
  - Evolution - improve naive and correct design while preserving functionality.

Chapter 5: Selectors, Shifters, Priority Encoders & Half-Decoders

Computer Structure

&

Intro. to Digital Computers

Dr. Guy Even

Tel-Aviv Univ.

Goals

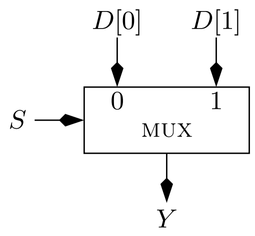
- Selectors:
  - review definition of multiplexer.
  - build $(n:1)$-multiplexers.
- Shifters:
  - Cyclic shifting
  - Barrel shifting
- Priority encoders
  - Unary priority encoder
  - Binary priority encoder

Multiplexer

DEF: A MUX-gate (also known as a $(2:1)$-multiplexer) is a combinational gate that has three inputs $D[0], D[1], S$ and one output Y. The functionality is defined by

$$Y = \begin{cases} D[0] & \text{if } S = 0 \\ D[1] & \text{if } S = 1. \end{cases}$$

Equivalently: $Y = D[S]$



Selectors

DEF: An (n:1)-MUX is a combinational circuit defined as follows:

Input: $D[n-1:0]$ and $S[k-1:0]$ where $k = \lceil \log_2 n \rceil$.

Output: $Y \in \{0,1\}$.

Functionality:

$$Y = D[\langle \vec{S} \rangle].$$

Example: Let $n = 4$, $D[3:0] = 0101$, and $S[1:0] = 11$. The output Y should be 1.

- $\vec{D}$ - data input
- $\vec{S}$ - select input
- simplify: assume that n is a power of 2, namely, $n = 2^k$.

Implementation of (n:1)-MUX

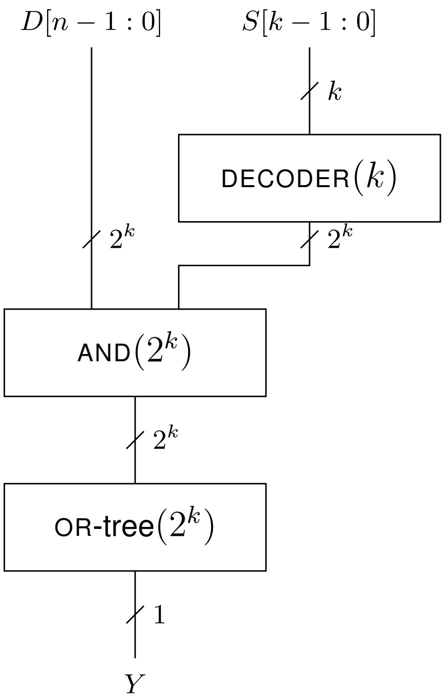
We will present two implementations:

- a decoder based implementation
- a tree-like implementation

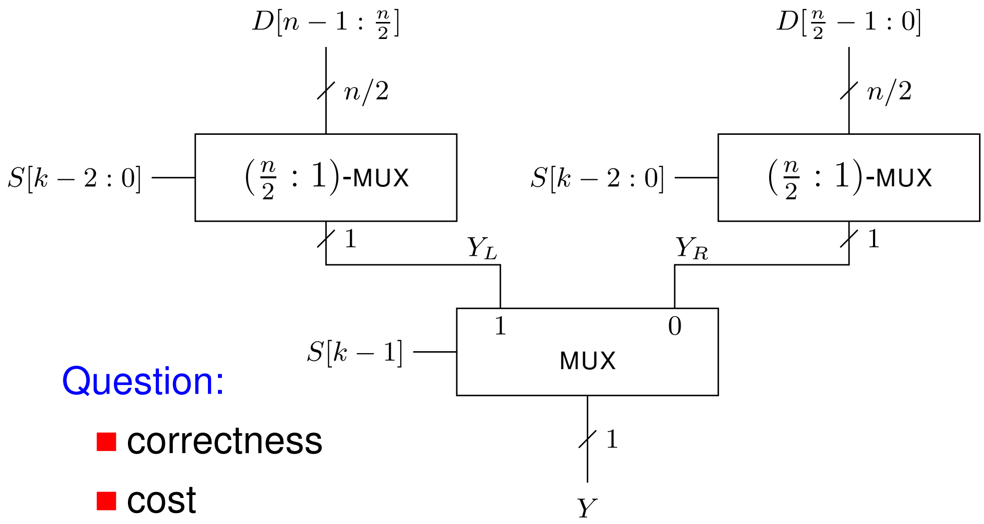
(n:1)-MUX : a decoder based implementation

Question:

- correctness
- cost
- delay
- asymptotic optimality



(n:1)-MUX : a tree-like implementation



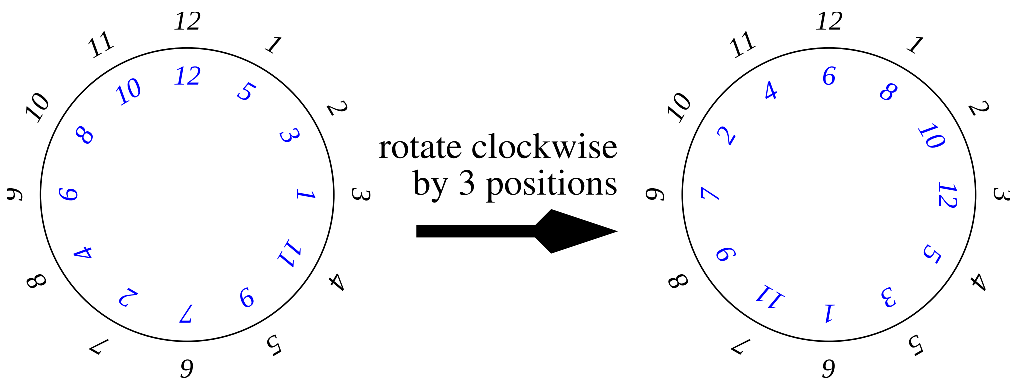
Question:

- correctness
- cost
- delay
- asymptotic optimality

Which design is better?

- both designs are asymptotically optimal.
- based on the tables of Müller & Paul, the tree-like design is better.
- decision is based on specific gate costs in the technology one uses.
- fast MUX-gates in CMOS (transmission gates) do not restore the signals well.
  $\Rightarrow$ long paths consisting only of MUX-gates are not allowed.

Cyclic shift - example



"clock"reads:
5,3,1,11,...,8,10,12

"clock"reads:
8,10,12,...,2,4,6

Cyclic shift - definition

The string $b[n-1:0]$ is a cyclic left shift by i positions of the string $a[n-1:0]$ if

$$\forall j: \quad b[j] = a[\text{mod}(j+i, n)].$$

Example: Let $a[3:0] = 0010$. A cyclic left shift by one position of $\vec{a}$ is the string 0100. A cyclic left shift by 3 positions of $\vec{a}$ is the string 0001.

Barrel Shifter

DEF: A BARREL-SHIFTER(n) is a combinational circuit defined as follows:

Input: $x[n-1:0]$ and $sa[k-1:0]$ where $k = \lceil \log_2 n \rceil$.

Output: $y[n-1:0]$.

Functionality: $\vec{y}$ is a cyclic left shift of $\vec{x}$ by $\langle \vec{sa} \rangle$ positions. Formally,

$$\forall j \in [n-1:0]: \quad y[j] = x[\text{mod}(j + \langle \vec{sa} \rangle, n)].$$

- $\vec{x}$ - data input
- $\vec{sa}$ - shift amount input
- simplify - assume that n is a power of 2, namely, $n = 2^k$.

CLS(n, i) - Cyclic Left Shift by 2^i positions

DEF: A CLS(n, i) is a combinational circuit defined as follows:

Input: $x[n-1:0]$ and $s \in \{0, 1\}$.

Output: $y[n-1:0]$.

Functionality:

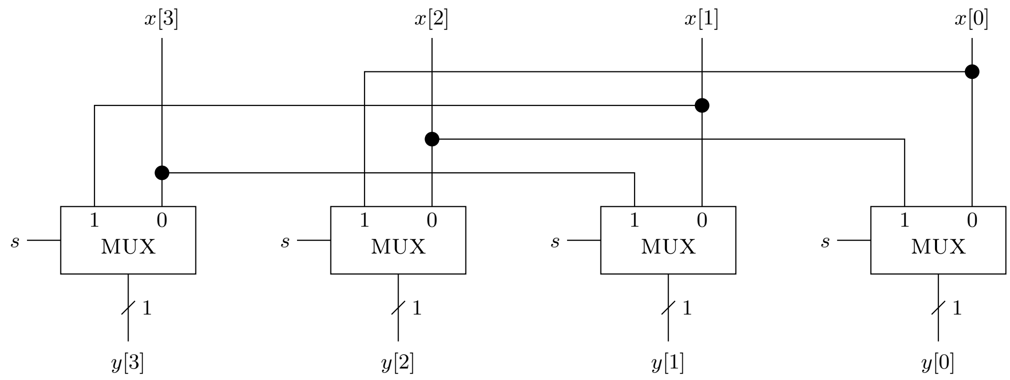
$$\forall j \in [n-1:0]: \quad y[j] = x[\text{mod}(j + s \cdot 2^i, n)].$$

Equivalently,

$$y[j] = \begin{cases} x[j] & \text{if } s = 0 \\ x[\text{mod}(j + 2^i, n)] & \text{if } s = 1. \end{cases}$$

$\Rightarrow$ can implement CLS(n, i) with a row of n MUX-gates.

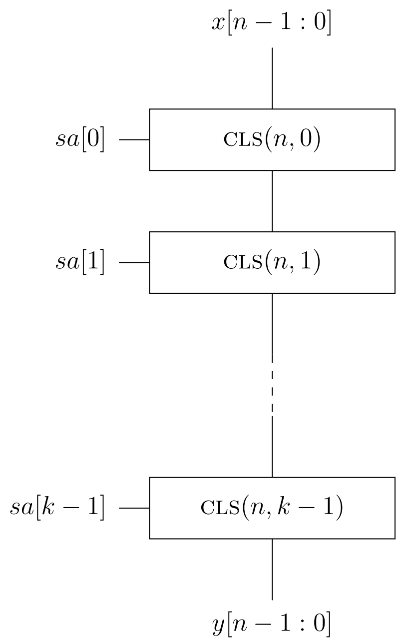
$\text{CLS}(4,1)$



Evident that a $\text{CLS}(n,i)$ requires a lot of area for the wires.

Our model does not capture routing cost.

BARREL-SHIFTER(n) - a chain of $\text{CLS}(n,i)$



$\textsc{Barrel-Shifter}(n)$ - correctness

Define the strings $y_i[n-1:0]$, for $0 \le i \le k-1$, recursively as follows:

$$y_0[n-1:0] \leftarrow \textsc{cls}_{n,0}(x[n-1,0], sa[0])$$
$$y_{i+1}[n-1:0] \leftarrow \textsc{cls}_{n,i+1}(y_i[n-1,0], sa[i+1])$$

Claim: $y_{k-1}[n-1:0]$ is a cyclic left shift of $x[n-1:0]$ by $\langle sa[k-1:0]\rangle$ positions.

Proof: Induction. $k = 0$ - trivial because $\textsc{cls}(n,0)$ shifts by zero/one position.

induction step

- $$\begin{aligned} y_i[j] &= \textsc{cls}_{n,i}(y_{i-1}[n-1,0], sa[i])[j] && \text{(by definition of } y_i\text{)} \\ &= y_{i-1}[\text{mod}(j + 2^i \cdot sa[i], n)] && \text{(by definition of } \textsc{cls}_{n,i}\text{).} \end{aligned}$$
- Let $\ell = \text{mod}(j + 2^i \cdot sa[i], n)$.
- Ind. Hyp. $\Rightarrow y_{i-1}[\ell] = x[\text{mod}(\ell + \langle sa[i-1:0]\rangle, n)$.
- Note that

$$\begin{aligned} \text{mod}(\ell + \langle sa[i-1:0]\rangle, n) &= \text{mod}(j + 2^i \cdot sa[i] + \langle sa[i-1:0]\rangle, n) \\ &= \text{mod}(j + \langle sa[i:0]\rangle, n). \end{aligned}$$

- Therefore $y_i[j] = x[\text{mod}(j + \langle sa[i:0]\rangle, n)]$, and the claim follows.

Priority Encoders

- Priority encoders detect the position of the leading one.
- Example: $x[0:6] = 0110100$. The leading one is $x[1]$.
- Position of leading one can be represented in binary or unary representation.
- DEF: A binary string $x[0:n-1]$ represents a number in unary representation if $x[0:n-1] \in 1^* \cdot 0^*$. The value represented in unary representation by the binary string $1^i \cdot 0^j$ is i.
- Example: 01001011 does not represent a number in unary representation. 111100 represents 4 in unary representation.

Unary priority encoder

U-PENC(n) - a combinational circuit specified as follows.

Input: $x[0:n-1]$.

Output: $y[0:n-1]$.

Functionality:

$$y[i] = \text{OR}(x[0:i]).$$

- Example: $x[0:6] = 0010101 \Longrightarrow y[0:6] = 0011111$.
- $\vec{x} \neq 0^n \quad \Longrightarrow y[0:n-1] = 0^j \cdot 1^{n-j}$, where $j = \min\{i \mid x[i] = 1\}$.
- Hence $\text{INV}(\vec{y})$ is a unary representation of the position of the leading one of the string $\vec{x} \cdot 1$.

Binary priority encoder

B-PENC(n) - a combinational circuit specified as follows.

Input: $x[0:n-1]$.

Output: $y[k:0]$, where $k = \lfloor \log_2 n \rfloor$. (Note that if $n = 2^{\ell}$, then $k = \ell$.)

Functionality:

$$\langle \vec{y} \rangle = \begin{cases} \min\{i \mid x[i] = 1\} & \text{if } \vec{x} \neq 0^n \\ n & \text{if } \vec{x} = 0^n. \end{cases}$$

Example: Given input $x[0:5] = 000101$, a U-PENC(6) outputs $y[0:5] = 000111$, and B-PENC(6) outputs $y[2:0] = 011$.

Claim

If $n = 2^k$, then

$$y[k] = 1 \Longleftrightarrow \vec{x} = 0^n$$

Proof:
If $\vec{x} \neq 0^n$, then $\langle \vec{y} \rangle = \min\{i \mid x[i] = 1\} < n$. Hence, $y[k] = 0$.

If $\vec{x} = 0^n$, then $\langle \vec{y} \rangle = n$. Hence, $y[k] = 1$.

U-PENC(n) - Implementation

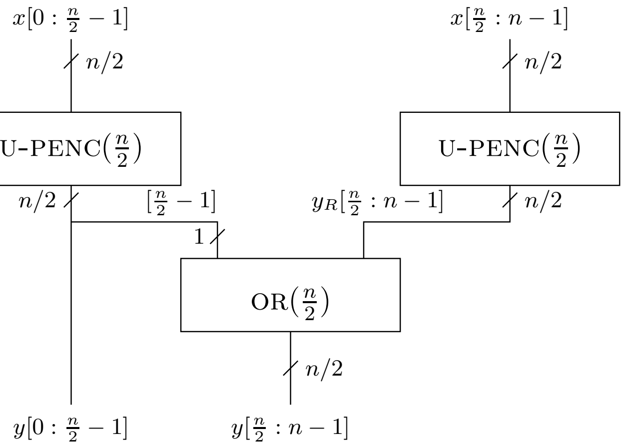
a brute force unary priority encoder:

- separate OR-tree for each output bit.
- delay $O(\log n)$ and cost $O(n^2)$.

- How can we efficiently combine these trees? To be discussed in detail when we discuss parallel prefix computation.
- Now we present a design based on divide-and-conquer.

divide & conquer U-PENC(n)

If $n = 1$, then $y[0] \leftarrow x[0]$.



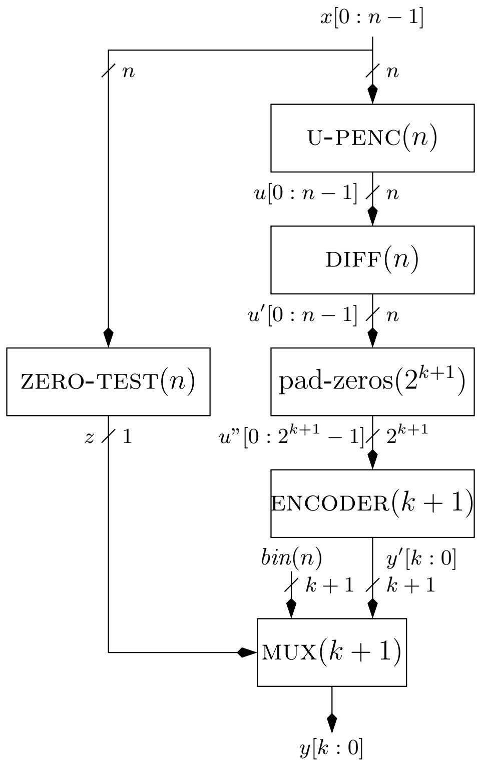
- Question: Prove correctness.
- cost = $O(n \log n)$ (not optimal!)
- delay = $O(\log n)$ (optimal)

Implementation of a binary priority encoder

We present two designs for a binary priority encoder.

- design based on a reduction to a unary priority encoder.
- design based on divide-and-conquer

reduction of B-PENC(n) to U-PENC(n)



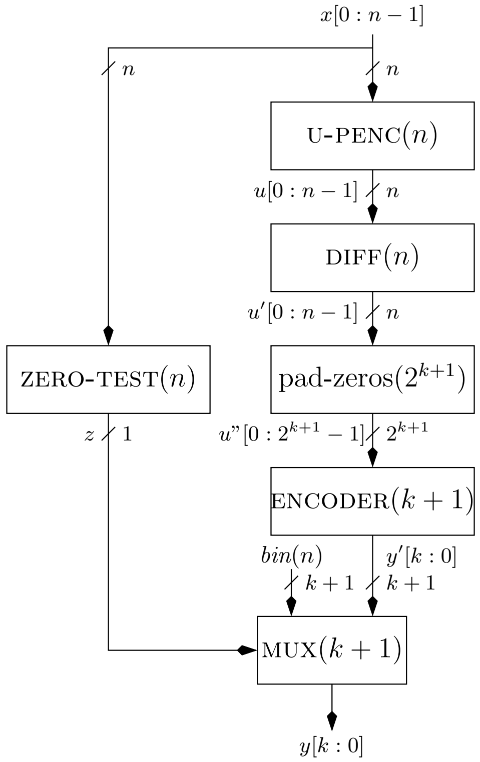
- apply U-PENC(n)
- difference:

$$u'[i] = \begin{cases} u[0] & \text{if } i = 0 \\ u[i] - u[i-1] & \text{otherwise.} \end{cases}$$

  if $\vec{x} \neq 0^n$, then $\vec{u'}$ is a 1-out-of-n representation of leading one's position.

- Pad & Encode.
- Select *bin*(n) if $\vec{x} = 0^n$.

cost analysis

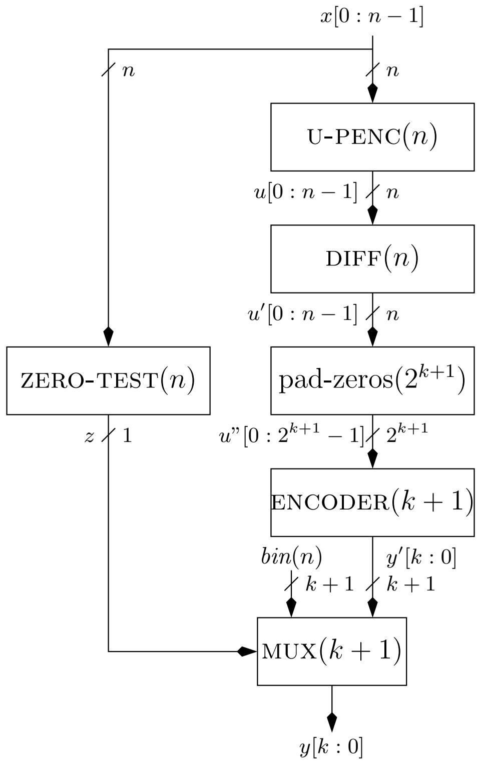


- zero cost: padding with zeros.
- logarithmic cost: multiplexer
- linear cost:
  - difference
  - encoder
  - zero tester
- $\Longrightarrow$

$$c(\text{B-PENC}(n)) = c(\text{U-PENC}(n)) + O(n).$$

- If $c(\text{U-PENC}(n)) = O(n)$, then also $c(\text{B-PENC}(n)) = O(n)$.

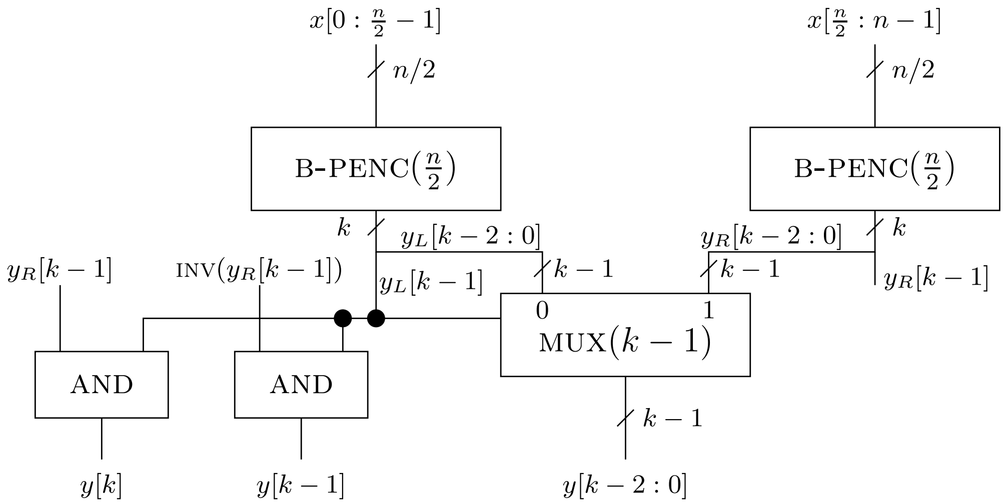
delay analysis



- zero delay: padding with zeros.
- constant delay:
  - difference
  - multiplexer
- logarithmic delay:
  - U-PENC(n)
  - encoder
  - zero tester
- $\Longrightarrow$

$$d(\text{B-PENC}(n)) = O(\log n).$$

a divide-and-conquer design for $n = 2^k$

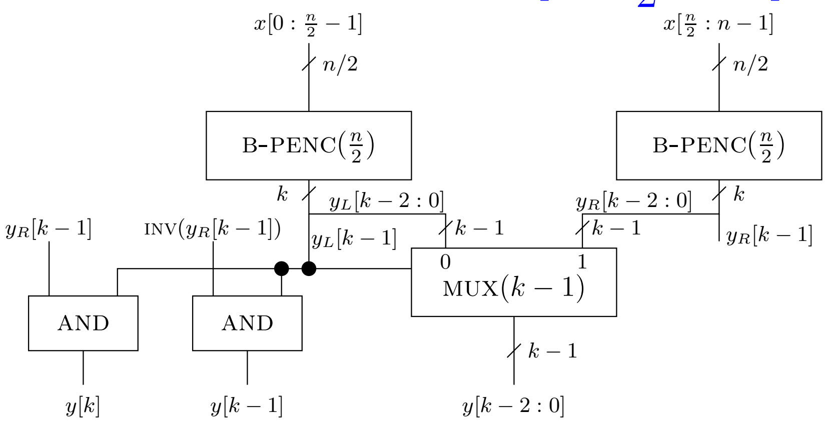


correctness

We prove correctness by induction.

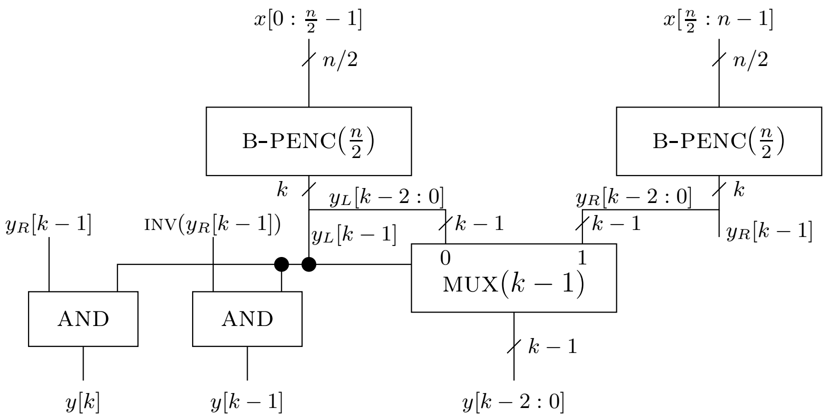
- Induction basis: $n = 1$ is trivial.
- Induction step deals with 3 cases:
  - leading one is in left half $x[0 : \frac{n}{2} - 1]$.
  - leading one is in right half $x[\frac{n}{2} : n - 1]$.
  - $\vec{x} = 0^n$.

correctness: case $x[0 : \frac{n}{2} - 1] \neq 0^{n/2}$



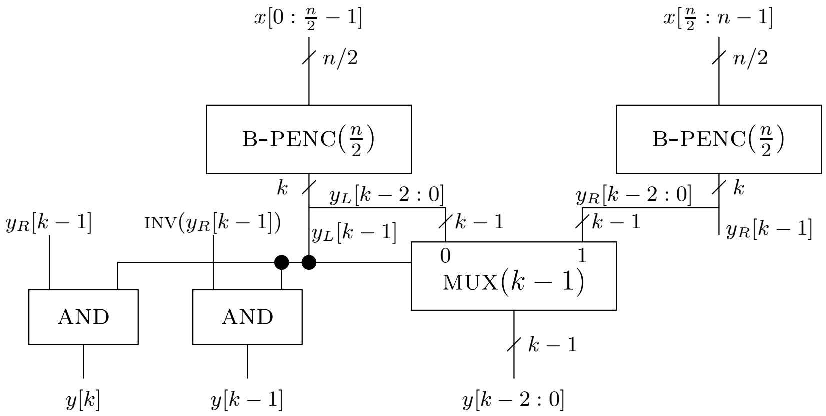
- Ind. Hyp. $\Rightarrow$ required output is $0 \cdot y_L[k-1:0]$.
- index of the leading one $< n/2 \Rightarrow \quad y_L[k-1] = 0$.
- $\Rightarrow \quad y[k] = y[k-1] = 0$ and $y[k-2:0] = y_L[k-2:0]$.
- $\Rightarrow$ output $\vec{y} = 0 \cdot y_L[k-1:0]$.

correctness: case $x[0 : \frac{n}{2} - 1] = 0^{n/2}$ & $x[\frac{n}{2} : n-1] \neq 0^{n/2}$



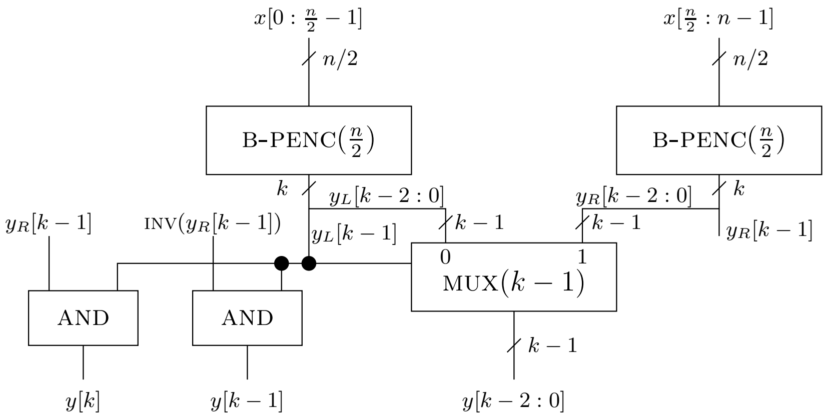
- Ind. Hyp. $\Rightarrow$ index of the leading one is $n/2 + \langle \vec{y_R} \rangle$.
- $\Rightarrow$ required output is $0 \cdot 1 \cdot y_R[k-2:0]$.
- Ind. Hyp. $\Rightarrow \quad y_L[k-1] = 1$ and $y_R[k-1] = 0$.
- $\Rightarrow \quad y[k] = 0$ & $y[k-1] = 1$.
- $y_L[k-1] = 1 \quad \Rightarrow \quad y[k-2:0] = y_R[k-2:0]$.

correctness: case $x[0:\frac{n}{2}-1] = 0^{n/2}$ & $x[\frac{n}{2}:n-1] = 0^{n/2}$



- required output is $1 \cdot 0^k$.
- Ind. Hyp. $\Rightarrow$ $y_L[k-1:0] = y_R[k-1:0] = 1 \cdot 0^{k-1}$.
- $y_L[k-1] = y_R[k-1] = 1 \quad \Rightarrow \quad y[k] = 1$.
- $y_L[k-1] = y_R[k-1] = 1 \quad \Rightarrow \quad y[k-1] = 0$.
- $y_L[k-1] = 1 \quad \Rightarrow \quad y[k-2:0] = y_L[k-2:0] = 0^{k-1}$.

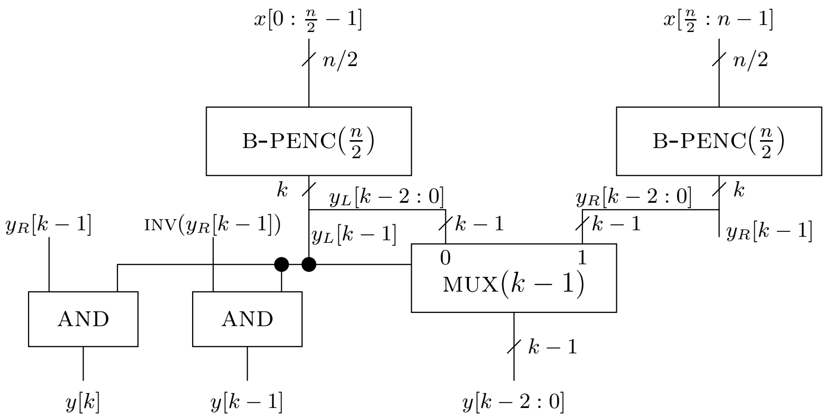
cost analysis



$$c(\text{B-PENC}(n)) = \begin{cases} c(\text{NOR}) & \text{if n=2} \\ 2 \cdot c(\text{B-PENC}(n/2)) + 2 \cdot c(\text{AND}) & \\ +(k-1) \cdot c(\text{MUX}) & \text{otherwise.} \end{cases}$$

Solution is $c(n) = O(n)$, same recurrence as for ENCODER$''(k)$ with substitution $k = \log n$.

delay analysis



$$d(\text{B-PENC}(n)) = \begin{cases} t_{pd}(\text{NOR}) & \text{if n=2} \\ d(\text{B-PENC}(n/2)) + \max\{d(\text{MUX}) + d(\text{AND})\} & \text{otherwise.} \end{cases}$$

$$\implies d(\text{B-PENC}(n)) = O(\log n)$$

Summary - priority encoders

- Two types of priority encoders: U-PENC(n) & B-PENC(n).
- Implementation of U-PENC(n):
  - brute force (separate OR-trees): cost $O(n^2)$, delay $O(\log n)$.
  - divide-&-conquer: cost $O(n \log n)$, delay $O(\log n)$.
  - to be shown: cost $O(n)$ and delay $O(\log n)$.
- Implementation of B-PENC(n):
  - reduction to U-PENC: overhead is cost - $O(n)$ & delay - $O(\log n)$.
  - divide-&-conquer: cost $O(n)$, delay $O(\log n)$.
  - will later show techniques based on adder trees.

Half-decoder

DEF: H-DEC(n) is a combinational circuit defined as follows:

Input: $x[n-1:0]$.

Output: $y[0:2^n-1]$

Functionality:

$$y[0:2^n-1] = 1^{\langle x[n-1:0]\rangle} \cdot 0^{2^n - \langle x[n-1:0]\rangle}.$$

- Decoder: binary $\rightarrow$ 1-out-of-2^n.
- Half-decoder: binary $\rightarrow$ unary.
- Example: $x[2:0] = 101 \quad \Longrightarrow \quad y[0:7] = 11111000.$
- Example: $\vec{x} = 0^n \quad \Longrightarrow \quad \vec{y} = 0^{2^n}.$
- Example: $\vec{x} = 1^n \quad \Longrightarrow \quad \vec{y} = 1^{2^n-1} \cdot 0.$
- Remark: always $y[2^n-1] = 0$.

Try to design H-DEC(n) using known modules

Question: Suggest an implementation of a half-decoder based on a decoder and a unary priority encoder. Analyze the cost and delay of the design. Is it optimal with respect to cost or delay?

Look for a better design...

Claim 1

$$y[i] = 1 \iff i < \langle \vec{x} \rangle.$$

Follows trivially from definition of H-DEC(n):

$$y[0 : 2^n - 1] = 1^{\langle x[n-1:0] \rangle} \cdot 0^{2^n - \langle x[n-1:0] \rangle}.$$

Claim 2

- $z[0 : n-1]$ - represents the number *wt*$(\vec{z})$ in unary representation.
- $i \in [0, n-1]$ - a fixed constant

Easy to compare *wt*$(\vec{z})$ and i:

$$\mathit{wt}(\vec{z}) < i \iff z[i-1] = 0$$
$$\mathit{wt}(\vec{z}) > i \iff z[i] = 1$$
$$\mathit{wt}(\vec{z}) = i \iff z[i] = 0 \text{ and } z[i-1] = 1.$$

Example:

$$z[0:5] = 111100 \Rightarrow \mathit{wt}(\vec{z}) = 4$$
$$\mathit{wt}(\vec{z}) < 4 \iff z[3] = 0$$
$$\mathit{wt}(\vec{z}) > 4 \iff z[4] = 1$$
$$\mathit{wt}(\vec{z}) = 4 \iff z[4] = 0 \text{ and } z[3] = 1.$$

Comparison box $\text{COMP}(\vec{z}, i)$

Input: $z[0 : n-1]$

Output: $GT, EQ, LT \in \{0, 1\}$.

Functionality:

$$GT = 1 \text{ if } \mathit{wt}(\vec{z}) > i$$
$$EQ = 1 \text{ if } \mathit{wt}(\vec{z}) = i$$
$$LT = 1 \text{ if } \mathit{wt}(\vec{z}) < i$$

Implementation:

$GT = z[i]$

$LT = \text{INV}(z[i-1])$

$EQ = \text{AND}(z[i-1], \text{INV}(z[i])) \quad (\mathit{wt}(\vec{z}) < i+1 \text{ and } \mathit{wt}(\vec{z}) > i-1)$

Claim 3: comparison based on quotient & remainder

- partition $\vec{x}$ into $\vec{x}_L$ and $\vec{x}_R$

  $x_L[n-k-1 : 0] = x[n-1 : k]$ and $x_R[k-1 : 0] = x[k-1 : 0]$.

- Binary representation implies that

$$\langle \vec{x} \rangle = 2^k \cdot \langle \vec{x}_L \rangle + \langle \vec{x}_R \rangle.$$

- Divide $i \in [0, 2^n - 1]$ by 2^k:

$$i = 2^k \cdot q + r, \quad \text{where } r \in [0, 2^k - 1].$$

- claim:

$$i < \langle \vec{x} \rangle \iff q < \langle \vec{x}_L \rangle \text{ or } (q = \langle \vec{x}_L \rangle \text{ and } r < \langle \vec{x}_R \rangle)$$

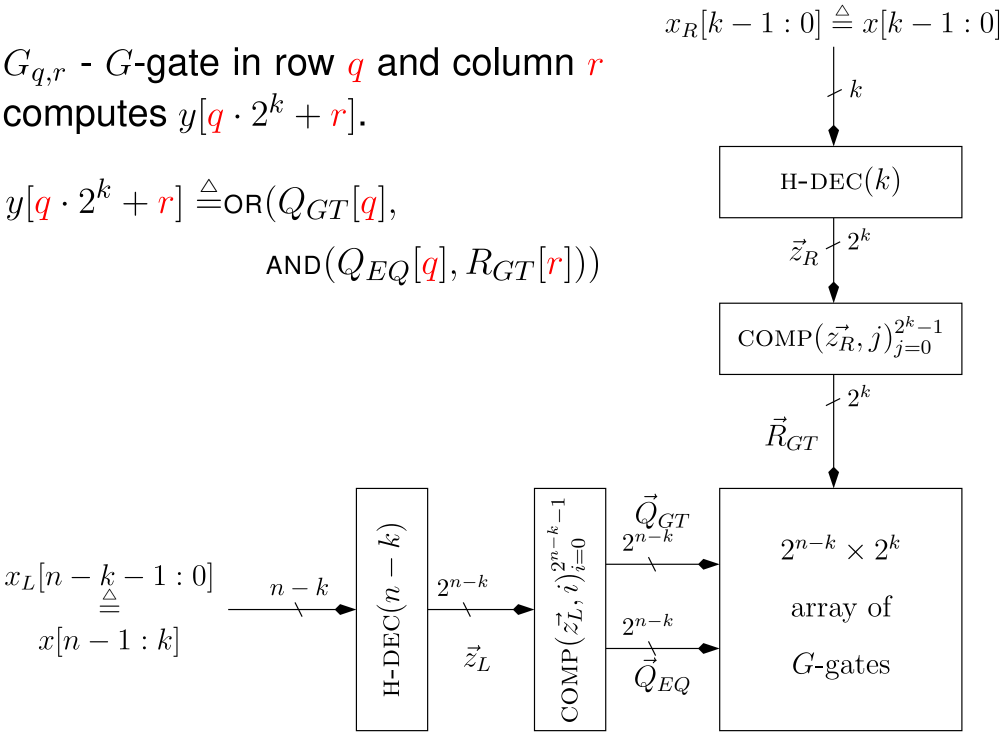
Implementation: H-DEC(n)

- Recursive divide-and-conquer design.
- Recursion basis: H-DEC(1) - $y[0] \leftarrow x[0]$. (unary representation $\equiv$ binary representation for a single bit).
- Recursion step...

Recursion step: H-DEC(n)

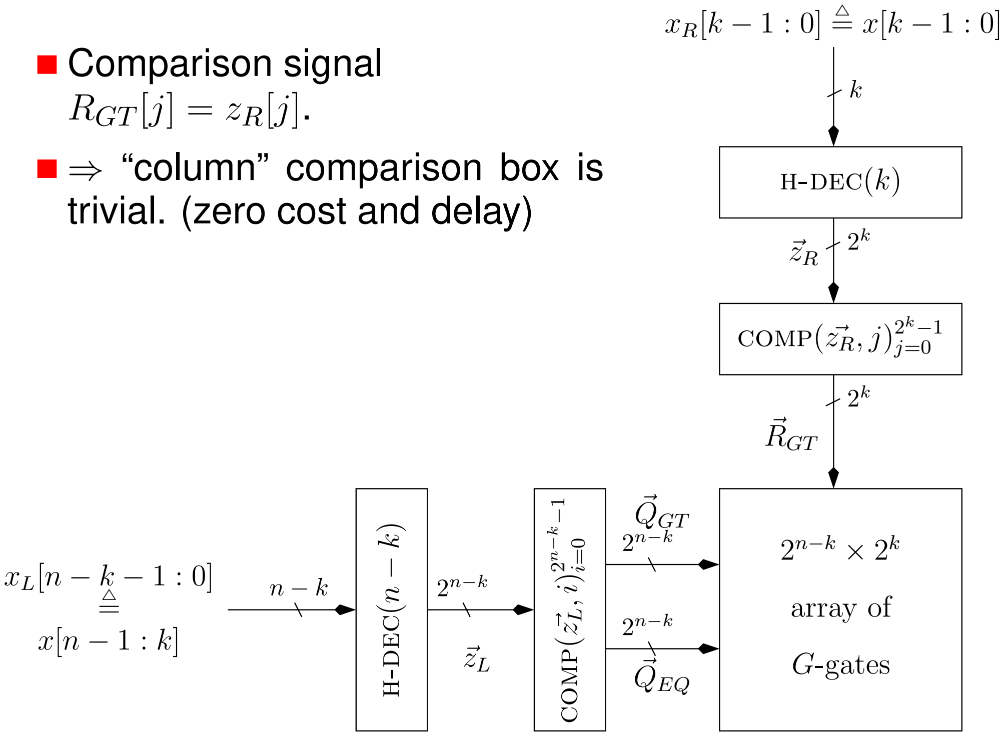
$G_{q,r}$ - G-gate in row q and column r computes $y[q \cdot 2^k + r]$.

$$y[q \cdot 2^k + r] \triangleq \text{OR}(Q_{GT}[q], \text{AND}(Q_{EQ}[q], R_{GT}[r]))$$



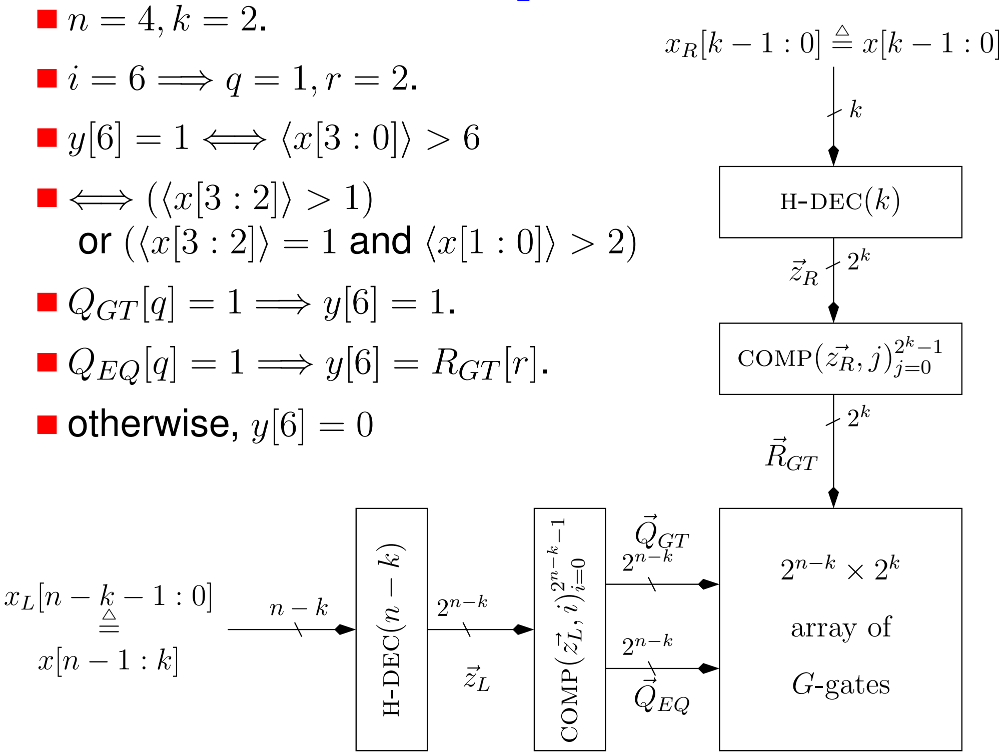
remarks

- Comparison signal $R_{GT}[j] = z_R[j]$.
- $\Rightarrow$ "column" comparison box is trivial. (zero cost and delay)



example

- $n = 4, k = 2$.
- $i = 6 \Longrightarrow q = 1, r = 2$.
- $y[6] = 1 \Longleftrightarrow \langle x[3:0] \rangle > 6$
- $\Longleftrightarrow (\langle x[3:2] \rangle > 1)$ or $(\langle x[3:2] \rangle = 1$ and $\langle x[1:0] \rangle > 2)$
- $Q_{GT}[q] = 1 \Longrightarrow y[6] = 1$.
- $Q_{EQ}[q] = 1 \Longrightarrow y[6] = R_{GT}[r]$.
- otherwise, $y[6] = 0$



Correctness: H-DEC(n)

- Proof by induction on n.
- Induction basis: H-DEC(1) - trivial.
- Induction step: by Claim 1, suffices to show that

$$y[i] = 1 \Longleftrightarrow i < \langle \vec{x} \rangle.$$

- Claim 3:

$$i < \langle \vec{x} \rangle \Longleftrightarrow (q < \langle \vec{x}_L \rangle) \text{ or } ((q = \langle \vec{x}_L \rangle) \text{ and } (r < \langle \vec{x}_R \rangle)).$$

- The induction hypothesis implies that:

$$\begin{aligned} q < \langle \vec{x}_L \rangle &\Longleftrightarrow z_L[q] = 1 \\ q = \langle \vec{x}_L \rangle &\Longleftrightarrow z_L[q] = 0 \text{ and } z_L[q-1] = 1 \\ r < \langle \vec{x}_R \rangle &\Longleftrightarrow z_R[r] = 1. \end{aligned}$$

Correctness - cont.

- Definition of comparison boxes implies that:

$$\begin{aligned} q < \langle \vec{x}_L \rangle &\Longleftrightarrow z_L[q] = 1 \Longleftrightarrow Q_{GT}[q] = 1 \\ q = \langle \vec{x}_L \rangle &\Longleftrightarrow \text{INV}(z_L[q]) \cdot z_L[q-1] \Longleftrightarrow Q_{EQ}[q] = 1 \\ r < \langle \vec{x}_R \rangle &\Longleftrightarrow z_R[r] = 1 \Longleftrightarrow R_{GT}[r] = 1. \end{aligned}$$

- $G_{q,r}$ outputs:

$$y[i] = \text{OR}(Q_{GT}[q], \text{AND}(Q_{EQ}[q], R_{GT}[r])).$$

- $\Longrightarrow y[i]$ is correct. QED

Cost analysis: H-DEC(n)

$$c(\text{H-DEC}(n)) = \begin{cases} 0 & \text{if n=1} \\ c(\text{H-DEC}(k)) + c(\text{H-DEC}(n-k)) & \\ +2^{n-k} \cdot c(EQ) + 2^n \cdot c(G) & \text{otherwise.} \end{cases}$$

- cost of computing the EQ is $c(\text{INV}) + c(\text{AND})$.
- $c(G) = c(\text{AND}) + c(\text{OR})$.
- It follows that

$$c(\text{H-DEC}(n)) = c(\text{H-DEC}(k)) + c(\text{H-DEC}(n-k)) + \Theta(2^n)$$

- same recurrence in the case of decoders.
  $\Rightarrow c(\text{H-DEC}(n)) = \Theta(2^n)$.

H-DEC(n) - lower bound on cost

Question*: Prove that every implementation of a half-decoder design must contain at least $2^n - 2$ non-trivial gates. (Here we assume that every non-trivial gate has a single output, and we do not have any fan-in or fan-out restrictions).

Delay analysis: $\textsf{H-DEC}(n)$

$$d(\textsf{H-DEC}(n)) = \begin{cases} 0 & \text{if n=1} \\ \max\{d(\textsf{H-DEC}(k)), \\ \qquad d(\textsf{H-DEC}(n-k)) + d(EQ)\} \\ +d(G) & \text{otherwise.} \end{cases}$$

- $d(EQ), d(G)$ are constant.
- Set $k = \lceil \frac{n}{2} \rceil$, then the recurrence degenerates to

$$\begin{aligned} d(\textsf{H-DEC}(n)) &= d(\textsf{H-DEC}(n/2)) + \Theta(1) \\ &= \Theta(\log n). \end{aligned}$$

- delay of $\textsf{H-DEC}(n)$ is asymptotically optimal since all the inputs belong to the cone of a half-decoder.

Summary - Half decoders

- half decoder translates binary $\longrightarrow$ unary.
- implementation:
  - variation of decoder design (divide-and-conquer with array of simple gates).
  - based on comparison of a unary number with a constant.
  - asymptotically optimal cost and delay.

Chapter 6: Addition

Computer Structure
&
Intro. to Digital Computers

Dr. Guy Even

Tel-Aviv Univ.

Goals

- Binary addition - definition
- Ripple Carry Adder - definition, correctness, cost, delay
- Carry bits - definition, properties
- Conditional Sum Adder - definition, correctness, cost, delay
- Compound Adder - definition, correctness, cost, delay

Binary Addition

DEF: 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:

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

- $\vec{A}, \vec{B}$ - binary representations of the addends.
- $C[0]$ - the carry-in bit.
- $\vec{S}$ - binary representation of the sum.
- $C[n]$ - the carry-out bit.

Question: is the functionality of ADDER(n) is well defined?

Lower bounds

Prove that for every ADDER(n):

- $c(\text{ADDER}(n)) = \Omega(n)$
- $d(\text{ADDER}(n)) = \Omega(\log n)$

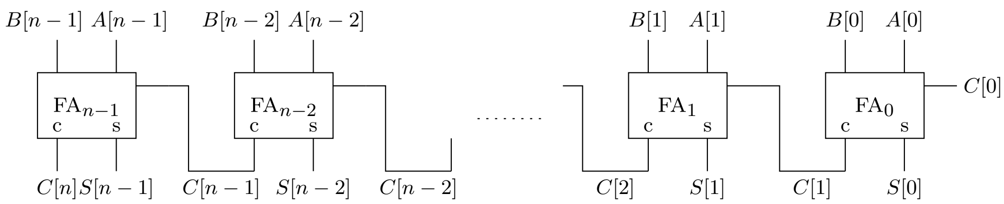
Full Adder

A Full-Adder is a combinational circuit with 3 inputs $x, y, z \in \{0, 1\}$ and 2 outputs $c, s \in \{0, 1\}$ that satisfies:

$$2c + s = x + y + z.$$

- A Full Adder computes a binary representation of the sum of 3 bits.
- s - called the sum output.
- c - called the carry-out output.
- We denote a Full-Adder by FA.

Ripple Carry Adder - RCA(n)



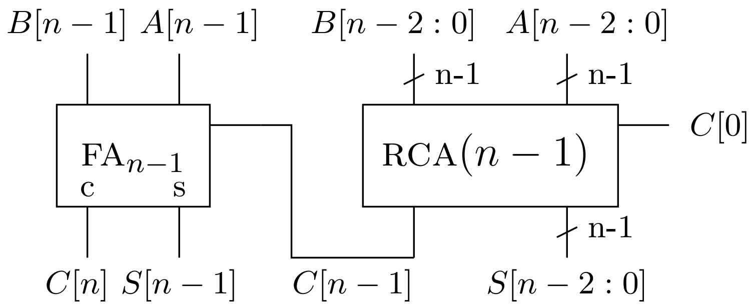
- carry-out output of FA$_i$ is denoted by $c[i+1]$.
- weight of every signal is two to the power of its index.
- RCA(n) - algorithm that we use for adding numbers by hand.

Correctness proof

To facilitate the proof, we use an equivalent recursive definition of RCA(n).

The recursive definition is as follows.

- Basis: an RCA(1) is simply a Full-Adder.
- Step:



Correctness - cont.

The proof is by induction on n.

The induction basis, for $n = 1$, follows directly from the definition of a Full-Adder.

Induction Step

The induction hypothesis, for $n-1$, is

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

Full-Adder definition

$$(2) \quad A[n-1] + B[n-1] + C[n-1] = 2\cdot C[n] + S[n-1].$$

Multiply (2) by 2^{n-1} to obtain

$$(3) \quad 2^{n-1}\cdot A[n-1] + 2^{n-1}\cdot B[n-1] + 2^{n-1}\cdot C[n-1] = 2^{n}\cdot C[n] + 2^{n-1}\cdot S[n-1].$$

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

$$(3) \quad 2^{n-1}\cdot A[n-1] + 2^{n-1}\cdot B[n-1] + 2^{n-1}\cdot C[n-1] = 2^{n}\cdot C[n] + 2^{n-1}\cdot S[n-1].$$

Note that $2^{n-1}\cdot A[n-1] + \langle A[n-2:0]\rangle = \langle A[n-1:0]\rangle$.
(1) + (3) $\Longrightarrow$

$$2^{n-1}\cdot C[n-1] + \langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = 2^{n}\cdot C[n] + 2^{n-1}\cdot C[n-1] + \langle S[n-1:0]\rangle.$$

Cancel out $2^{n-1}\cdot C[n-1]$. QED.

Cost & Delay Analysis

The cost of an $\text{RCA}(n)$ satisfies:

$$c(\text{RCA}(n)) = n \cdot c(\text{FA}) = \Theta(n).$$

The delay of an $\text{RCA}(n)$ satisfies

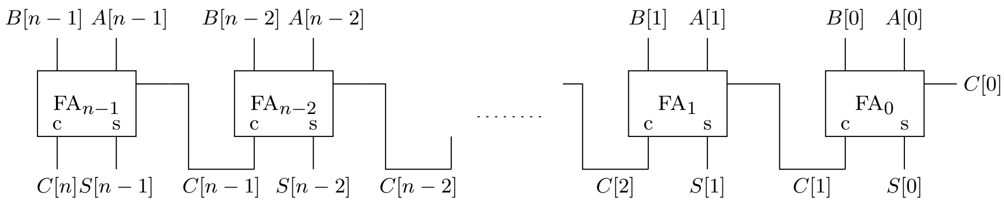
$$d(\text{RCA}(n)) = n \cdot d(\text{FA}) = \Theta(n).$$

Is $\text{RCA}(n)$ good enough?

- Clock rate $= 1GHz = 10^9 Hz$
  $\Rightarrow$ clock period $= 10^{-9}$sec $= 1ns$.
- Delay of gate $\approx 100ps = 0.1ns$.
- $d(\text{FA}) \approx 2 \cdot d(\text{gate}) \approx 0.2ns$.
- $\Rightarrow$ Within a clock period we can only add 5-bit numbers...
- Question: How are > 100 bits added in one clock cycle?

Carry bits

DEF: The carry bits associated with an addition $\langle\vec{A}\rangle + \langle\vec{B}\rangle + C[0]$ are the signals $C[n:0]$ in an RCA(n).



remark 1: redundant & non-redundant representations

Functionality of an adder:

$$\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = 2^n \cdot C[n] + \langle S[n-1:0]\rangle.$$

- Let $x = \langle\vec{A}\rangle + \langle\vec{B}\rangle + C[0]$.
- x admits two representations (left-hand side, right-hand side)
- $C[n] \cdot S[n-1:0]$ - binary representation of x.
- Binary representation is non-redundant:
  - Every value has a unique representation.
  - $\langle\vec{X}\rangle = \langle\vec{Y}\rangle \Longleftrightarrow X = Y$.

remark 1 - cont

Functionality of an adder:

$$\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = 2^n \cdot C[n] + \langle S[n-1:0]\rangle.$$

- $x = \langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0]$.
- many possible combinations of $\langle \vec{A}\rangle, \langle \vec{B}\rangle$ and $C[0]$. For example: $8 = 4+3+1$, and also $8 = 5+3+0$.
- $\rightarrow$ redundant representation.
- in redundant representation:

$$X \neq Y \not\Longrightarrow \text{value(X)} \neq \text{value}(Y).$$

- $\Rightarrow$ in redundant representation: comparison is complicated.
- ADDER(n) - translates a redundant representation to a non-redundant binary representation.

remark 2: cones

The correctness proof of RCA(n) implies that, for every $0 \leq i \leq n-1$,

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

This equality means that:

$$\textit{cone}(C[i+1]), \textit{cone}(S[i:0]) \subseteq A[i:0] \bigcup B[i:0] \bigcup C[0].$$

Question: Prove that

$$\textit{cone}(S[i]), \textit{cone}(C[i+1]) = A[i:0] \bigcup B[i:0] \bigcup C[0].$$

remark 3

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

$\Longrightarrow$ for every $0 \le i \le n-1$,

$$\langle S[i:0]\rangle = \text{mod}(\langle A[i:0]\rangle + \langle B[i:0]\rangle + C[0], 2^{i+1}).$$

remark 4: reductions sum-bits $\longleftrightarrow$ carry-bits

The correctness of $\textsc{rca}(n)$ implies that, for every $0 \le i \le n-1$,

$$S[i] = \textsc{xor}(A[i], B[i], C[i]).$$

$\Longrightarrow$ for every $0 \le i \le n-1$,

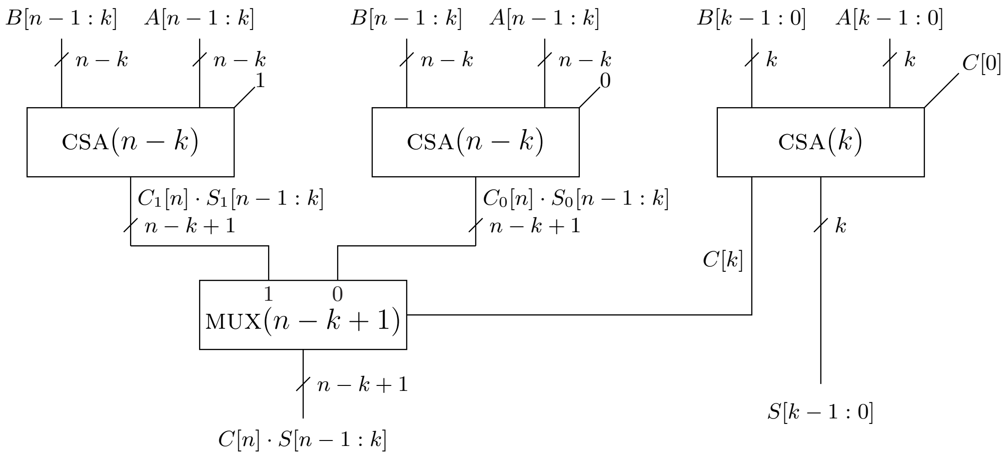
$$C[i] = \textsc{xor}(A[i], B[i], S[i]).$$

$\Longrightarrow$ constant-time linear-cost reductions:

$$S[n-1:0] \longmapsto C[n-1:0]$$
$$C[n-1:0] \longmapsto S[n-1:0]$$

$\Longrightarrow$ if *Circuit* computes $C[n-1:0]$ with $O(n)$ cost and $(\log n)$ delay, then we know how to add asymptotically optimal.

Conditional Sum Adder - $csa(n)$



Question: Prove the correctness of the CSA(n) design.

Delay analysis

- To simplify the analysis we assume that $n = 2^\ell$.
- To optimize the cost and delay, we use $k = n/2$.
- $d(\text{FA})$ - delay of a Full-Adder.
- The delay of a CSA(n) satisfies the following recurrence:

$$d(\text{CSA}(n)) = \begin{cases} d(\text{FA}) & \text{if } n = 1 \\ d(\text{CSA}(n/2)) + d(\text{MUX}) & \text{otherwise.} \end{cases}$$

- It follows that the delay of a CSA(n) is

$$\begin{aligned} d(\text{CSA}(n)) &= \ell \cdot d(\text{MUX}) + d(\text{FA}) \\ &= O(\log n). \end{aligned}$$

Cost Analysis

- $c(\text{FA})$ - cost of a Full-Adder.
- The cost of a $\text{CSA}(n)$ satisfies the following recurrence:

$$c(\text{CSA}(n)) = \begin{cases} c(\text{FA}) & \text{if } n = 1 \\ 3 \cdot c(\text{CSA}(n/2)) + (n/2+1) \cdot c(\text{MUX}) & \text{otherwise.} \end{cases}$$

- master theorem for recurrences - provides a solution. We will solve this recurrence from scratch.
- open two steps of the recurrence:

$$\begin{aligned} c(n) &= 3 \cdot c\left(\frac{n}{2}\right) + c\,(\text{MUX}) \cdot \left(\frac{n}{2} + 1\right) \\ &= 3 \cdot \left(3 \cdot c\left(\frac{n}{4}\right) + c(\text{MUX}) \cdot \left(\frac{n}{4} + 1\right)\right) + c(\text{MUX}) \cdot \left(\frac{n}{2} + 1\right) \\ &= 3^2 \cdot c\left(\frac{n}{4}\right) + c(\text{MUX}) \cdot \frac{n}{2} \cdot \left(1 + \frac{3}{2}\right) + (1+3) \cdot c\,(\text{MUX}) \end{aligned}$$

Cost Analysis - cont.

$$c(n) = 3^2 \cdot c\left(\frac{n}{4}\right) + c(\text{MUX}) \cdot \frac{n}{2} \cdot \left(1 + \frac{3}{2}\right) + (1+3) \cdot c\,(\text{MUX})$$

- good guess (which can be proved by induction):

$$c(n) = 3^\ell \cdot c\left(\frac{n}{2^\ell}\right) + c(\text{MUX}) \cdot \frac{n}{2} \cdot \left(1 + \frac{3}{2} + \cdots \left(\frac{3}{2}\right)^{\ell-1}\right) + (1 + 3 + \cdots + 3^{\ell-1}) \cdot c(\text{MUX}).$$

- Since $\ell = \log_2 n$, it follows that
  - $3^\ell = n^{\log_2 3}$ and $(1 + 3 + \cdots + 3^{\ell-1}) < 3^\ell/2$.
  - $\frac{n}{2} \cdot \left(1 + \frac{3}{2} + \cdots \left(\frac{3}{2}\right)^{\ell-1}\right) < n^{\log_2 3}$.
- We conclude that

$$\begin{aligned} c(n) &< n^{\log_2 3} \cdot \left(c(\text{FA}) + \frac{3}{2} \cdot c(\text{MUX})\right) \\ &\approx \Theta\left(n^{1.58}\right). \end{aligned}$$

Conditional Sum Adder - Discussion

- $d(\text{CSA}(n)) = \Theta(\log n)$.
- $c(\text{CSA}(n)) \approx \Theta\left(n^{1.58}\right)$.
- $\text{CSA}(n)$ is rather costly.. but only adder we know so far whose delay is logarithmic.
- method allows to use three half-sized adders (i.e. addends are $n/2$ bits long) in order to implement a full sized adder (i.e. addends are n bits long).

Conditional Sum Adder - Discussion - cont.

Question: What is the effect of fanout to the delay and cost of $\text{CSA}(n)$?

- The fanout of the carry-bit $C[k]$ is $n/2 + 1$ if $k = n/2$. Suppose that we associate a delay of $\log_2(f)$ with a fanout f. How would taking the fanout into account change the delay analysis of a $\text{CSA}(n)$?
- Suppose that we associate a cost $O(f)$ with a fanout f. How would taking the fanout into account change the cost analysis of a $\text{CSA}(n)$?

Compound Adder

- CSA(n) - uses two adders in the upper part, one with a zero carry-in and one with a one carry-in.
- compound adder - computes both the sum and the incremented sum.

Compound Adder

- DEF: A Compound Adder with input length n is a combinational circuit specified as follows.

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

  Output: $S[n:0], T[n:0] \in \{0,1\}^{n+1}$.

  Functionality:

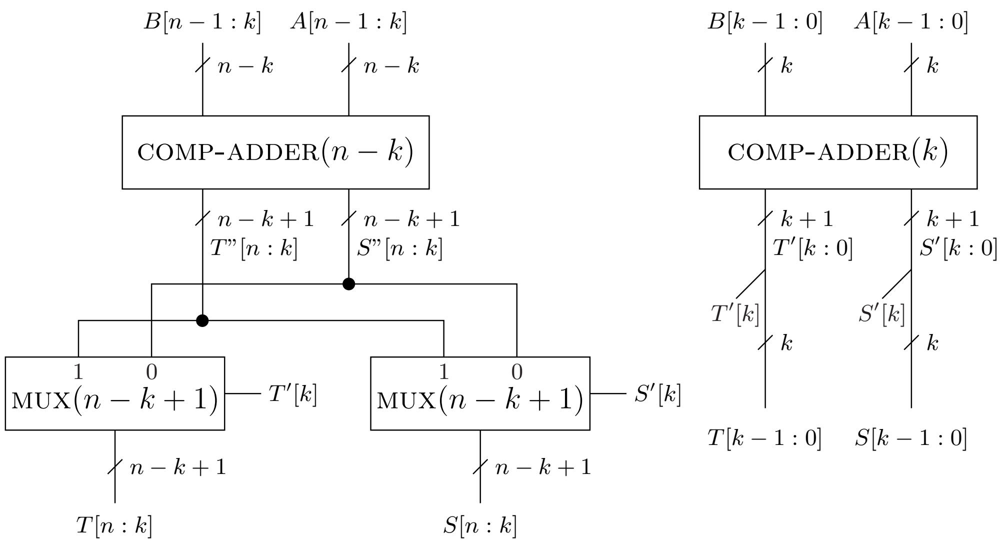
$$\langle \vec{S} \rangle = \langle \vec{A} \rangle + \langle \vec{B} \rangle$$
$$\langle \vec{T} \rangle = \langle \vec{A} \rangle + \langle \vec{B} \rangle + 1.$$

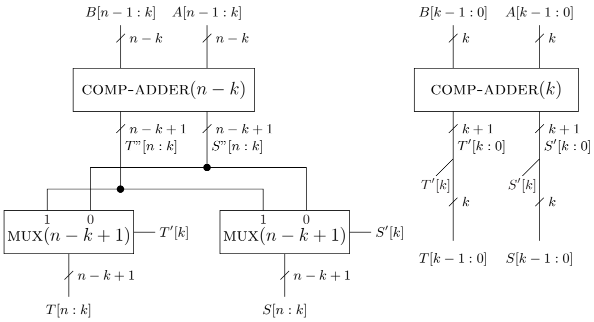
- Compound Adder does not have carry-in input.
- $S[n]$ - carry-out of the sum.
- $T[n]$ - carry-out of the incremented sum.
- COMP-ADDER(n) - a compound adder with input length n.

COMP-ADDER(n) - implementation

- We apply divide-and-conquer to design a COMP-ADDER(n).
- For $n = 1$, we simply use a Full-Adder and a Half-Adder (one could optimize this a bit).
- The design for $n > 1$...

COMP-ADDER(n) - implementation

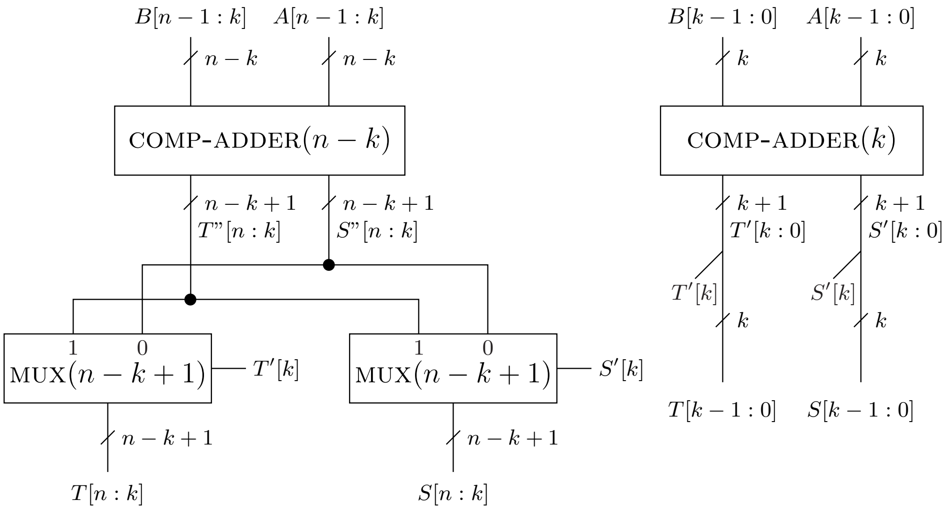




Example: COMP-ADDER(4) with inputs $A[3:0] = 0110$ & $B[3:0] = 1001$.

- $S'[2:0] = 011$ and $T'[2:0] = 100$.
- $S[1:0] = S'[1:0] = 11$ and $T[1:0] = T'[1:0] = 00$.
- The upper part: $S''[4:2] = 011$ and $T''[4:2] = 100$.
- The output $S[4:2] = S''[4:2]$ since $S'[2] = 0$.
- The output $T[4:2] = T''[4:2]$ since $T'[2] = 1$.
- Hence $S[4:0] = 01111$ and $T[4:0] = 10000$.

Compound adder



Question:

- In the example we had $S'[k] = 0$ & $T'[k] = 1$.
- Is it possible to have $S'[k] = 1$ & $T'[k] = 0$?

COMP-ADDER(n) - correctness

- The proof is by induction on n.
- Basis: $n = 1$ follows from the correctness of a Full-Adder and a Half-Adder.
- Induction Step: prove for the output $S[n:0]$; the correctness of $T[n:0]$ is proved in a similar fashion.
- The induction hypothesis implies that

$$\langle S'[k:0]\rangle = \langle A[k-1:0]\rangle + \langle B[k-1:0]\rangle.$$

- Note that:
  - $S[k-1:0] = S'[k-1:0]$
  - $S'[k] = C[k]$, where $C[k]$ is the carry-bit in position $[k]$ corresponding to $\langle A[k-1:0]\rangle + \langle B[k-1:0]\rangle$.

COMP-ADDER(n) - correctness - cont.

We apply the induction hypothesis to the upper part:

$$\langle S''[n:k]\rangle = \langle A[n-1:k]\rangle + \langle B[n-1:k]\rangle$$
$$\langle T''[n:k]\rangle = \langle A[n-1:k]\rangle + \langle B[n-1:k]\rangle + 2^k.$$

Since

$$\langle S'[k:0]\rangle = \langle A[k-1:0]\rangle + \langle B[k-1:0]\rangle.$$

It follows that

$$\langle S''[n:k]\rangle + \langle S'[k:0]\rangle = \langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle$$

COMP-ADDER(n) - correctness - cont.

$$(1) \quad \langle S''[n:k] \rangle + \langle S'[k:0] \rangle = \langle A[n-1:0] \rangle + \langle B[n-1:0] \rangle$$

two cases: $C[k] = 0$ and $C[k] = 1$.

1. If $C[k] = 0$, then $S'[k] = 0$. Equation (1) reduces to

$$\begin{aligned} \langle A[n-1:0] \rangle + \langle B[n-1:0] \rangle &= \langle S''[n:k] \rangle + \langle S'[k-1:0] \rangle \\ &= \langle S[n:k] \rangle + \langle S[k-1:0] \rangle \\ &= \langle S[n:0] \rangle. \end{aligned}$$

2. If $C[k] = 1$, then $S'[k] = 1$. Equation (1) reduces to

$$\begin{aligned} \langle A[n-1:0] \rangle + \langle B[n-1:0] \rangle &= \langle S''[n:k] \rangle + 2^k + \langle S'[k-1:0] \rangle \\ &= \langle T''[n:k] \rangle + \langle S[k-1:0] \rangle \\ &= \langle S[n:0] \rangle. \end{aligned}$$

In both cases: $\langle \vec{S} \rangle = \langle \vec{A} \rangle + \langle \vec{B} \rangle$. QED

COMP-ADDER(n) - Delay analysis

- To simplify the analysis we assume that $n = 2^\ell$.
- To optimize the cost and delay, we use $k = n/2$.
- The delay of a COMP-ADDER(n) satisfies the following recurrence:

$$d(\text{COMP-ADDER}(n)) = \begin{cases} d(\text{FA}) & \text{if } n = 1 \\ d(\text{COMP-ADDER}(n/2)) & \\ +d(\text{MUX}) & \text{otherwise.} \end{cases}$$

- It follows that the delay of a COMP-ADDER(n) is

$$\begin{aligned} d(\text{COMP-ADDER}(n)) &= \ell \cdot d(\text{MUX}) + d(\text{FA}) \\ &= O(\log n). \end{aligned}$$

- As in CSA(n), fanout considerations lead to $\Theta(\log^2 n)$ delay.

COMP-ADDER(n) - Cost Analysis

$$c(\text{COMP-ADDER}(n)) = \begin{cases} c(\text{FA}) & \text{if } n = 1 \\ 2 \cdot c(\text{COMP-ADDER}(n/2)) & \\ +2 \cdot (n-k+1) \cdot c(\text{MUX}) & \text{otherwise.} \end{cases}$$

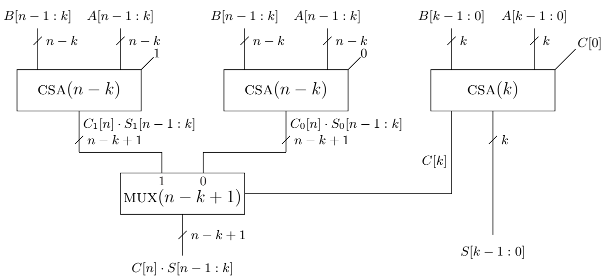
$\Rightarrow$

$$c(n) = 2c(n/2) + \Theta(n).$$

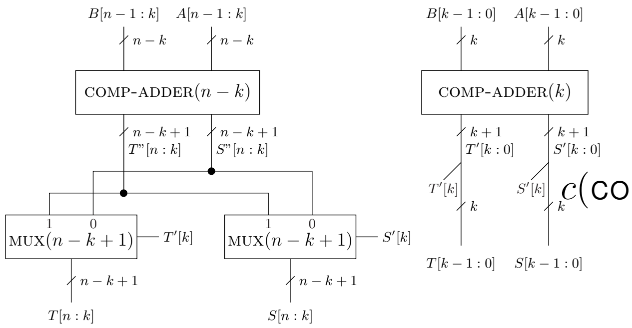
$\Rightarrow$

$$c(n) = \Theta(n \log n).$$

CSA(n) vs. COMP-ADDER(n)



$$c(\text{CSA}(n)) \approx \Theta\left(n^{1.58}\right)$$



$$c(\text{COMP-ADDER}(n)) = \Theta(n \log n)$$

Question: more functionality is cheaper?!

$\textsf{CSA}(n)$ vs. $\textsf{COMP-ADDER}(n)$ - cont.

Solve the recurrences:

$$c(\textsf{CSA}(n)) \approx n^{1.58} \cdot \left(c(\textsf{FA}) + \frac{3}{2} \cdot c(\textsf{MUX}) \right)$$

$$\begin{aligned} c(\textsf{COMP-ADDER}(n)) \approx\ & n \cdot (c(\textsf{FA}) + c(\textsf{HA})) \\ & + c(\textsf{MUX}) \cdot \left(\frac{n}{2} \cdot \log \frac{n}{2} + n - 1 \right). \end{aligned}$$

Assume $c(\textsf{FA}) = c(\textsf{HA}) = c(\textsf{MUX}) = 1$. Then,

$$c(\textsf{CSA}(n)) \approx 2.5 \cdot n^{1.58}$$

$$c(\textsf{COMP-ADDER}(n)) \approx \frac{n}{2} \cdot \log \frac{n}{2} + 3n - 1.$$

$\textsf{CSA}(n)$ vs. $\textsf{COMP-ADDER}(n)$ - cont.

$$c(\textsf{CSA}(n)) \approx 2.5 \cdot n^{1.58}$$

$$c(\textsf{COMP-ADDER}(n)) \approx \frac{n}{2} \cdot \log \frac{n}{2} + 3n - 1.$$

n	$c(\textsf{CSA}(n))$	$c(\textsf{COMP-ADDER}(n))$
4	22.5	15
8	67.5	35
16	202	79
32	607	175
64	1822	383
128	5467	831

Summary

- defined binary addition & RCA(n)
- defined carry bits
- showed some properties of addition
- Conditional Sum Adder:
  - a divide-and-conquer design
  - $\Theta(\log n)$ delay (fanout $\longmapsto \Theta(\log^2 n)$)
  - $\Theta\left(n^{1.58}\right)$ cost
- Compound Adder
  - adds and also adds $+1$
  - a divide-and-conquer design
  - $\Theta(\log n)$ delay (fanout $\longmapsto \Theta(\log^2 n)$)
  - $\Theta\left(n \cdot \log n\right)$ cost
- $c(\text{COMP-ADDER}(n)) \ll c(\text{CSA}(n))$!

Chapter 7: Fast Addition: parallel prefix computation

Computer Structure

&

Intro. to Digital Computers

Dr. Guy Even

Tel-Aviv Univ.

Goals

- Design an adder with $O(\log n)$ delay and $O(n)$ cost.
- Learn some interesting methods along the way...

reminder: reduction sum-bits $\longmapsto$ carry-bits

The correctness of $\text{RCA}(n)$ implies that, for every $0 \leq i \leq n-1$,

$$S[i] = \text{XOR}(A[i], B[i], C[i]).$$

$\Longrightarrow$ constant-time linear-cost reduction:

$$S[n-1:0] \longmapsto C[n-1:0]$$

$\Longrightarrow$ if Circuit computes $C[n-1:0]$ with $O(n)$ cost and $O(\log n)$ delay, then we know how to add asymptotically optimally.

Computing the carry bits - preliminary

Functionality of Full-Adder:

$$C[i+1] = \begin{cases} 0 & \text{if } A[i] + B[i] + C[i] \leq 1 \\ 1 & \text{if } A[i] + B[i] + C[i] \geq 2. \end{cases}$$

Claim:

$$\begin{aligned} A[i] + B[i] = 0 &\implies C[i+1] = 0 \\ A[i] + B[i] = 2 &\implies C[i+1] = 1 \\ A[i] + B[i] = 1 &\implies C[i+1] = C[i] \end{aligned}$$

$\implies$

- if $A[i] + B[i] \in \{0, 2\}$, then easy to compute $C[i+1]$.
- if $A[i] + B[i] = 1$, then "ripple effect" of carry.

definition of $\sigma[n-1:-1]$

DEF: for $i = -1, 0, \ldots, n-1$

$$\sigma[i] \triangleq \begin{cases} 2 \cdot C[0] & \text{if } i = -1 \\ A[i] + B[i] & \text{if } i \in [0, n-1]. \end{cases}$$

Note that $\sigma[i] \in \{0, 1, 2\}$.

Claim: for every $-1 \leq i \leq n-1$,

$$C[i+1] = 1 \qquad \Longleftrightarrow \qquad \exists j \leq i \ : \ \sigma[i:j] = 1^{i-j} \cdot 2.$$

example with $\sigma[n-1:-1]$

$$\sigma[i] \triangleq \begin{cases} 2 \cdot C[0] & \text{if } i = -1 \\ A[i] + B[i] & \text{if } i \in [0, n-1]. \end{cases}$$

Claim: for every $-1 \leq i \leq n-1$,

$$C[i+1] = 1 \qquad \Longleftrightarrow \qquad \exists j \leq i \ : \ \sigma[i:j] = 1^{i-j} \cdot 2.$$

Example: $A[3:0] = 0110, B[3:0] = 0011, C[0] = 0$.

position	4	3	2	1	0	-1
A		0	1	1	0	
B		0	0	1	1	
S		1	0	0	1	
C	0	1	1	0	0	
σ		0	1	2	1	0

Proof: $\sigma[i:j] = 1^{i-j} \cdot 2 \Rightarrow C[i+1] = 1$

- By induction on $i - j$.
- Basis $i - j = 0$: in this case $\sigma[i] = 2$.
  - If $i = -1$, then $C[0] = 1$.
  - If $i \geq 0$, then $A[i] + B[i] = 2$. Hence $C[i+1] = 1$.
- Ind. Step: note that $\sigma[i-1:j] = 1^{i-j-1} \cdot 2$.
- Ind. Hyp. $\Rightarrow C[i] = 1$.
- Since $\sigma[i] = 1$, we conclude that

$$\underbrace{A[i] + B[i]}_{\sigma[i]=1} + C[i] = 2.$$

- Hence, $C[i+1] = 1$.

Proof: $C[i+1] = 1 \Rightarrow \exists j \leq i : \sigma[i:j] = 1^{i-j} \cdot 2$

- By induction on i.
- Basis $i = -1$: in this case $C[0] = 1$, hence $\sigma[-1] = 2$. Set $j = i$.
- Ind. Step: Assume $C[i+1] = 1$. Hence,

$$\underbrace{A[i] + B[i]}_{\sigma[i]} + C[i] \geq 2.$$

  - $\sigma[i] = 0$: contradiction.
  - $\sigma[i] = 2$: set $j = i$.
  - $\sigma[i] = 1$: $\Rightarrow C[i] = 1$.

$$C[i] = 1 \overset{\text{Ind. Hyp.}}{\Longrightarrow} \exists j \leq i : \sigma[i-1:j] = 1^{i-j-1} \cdot 2$$

$$\overset{\sigma[i]=1}{\Longrightarrow} \exists j \leq i : \sigma[i:j] = 1^{i-j} \cdot 2.$$

Corollary: method for computing $C[i+1]$

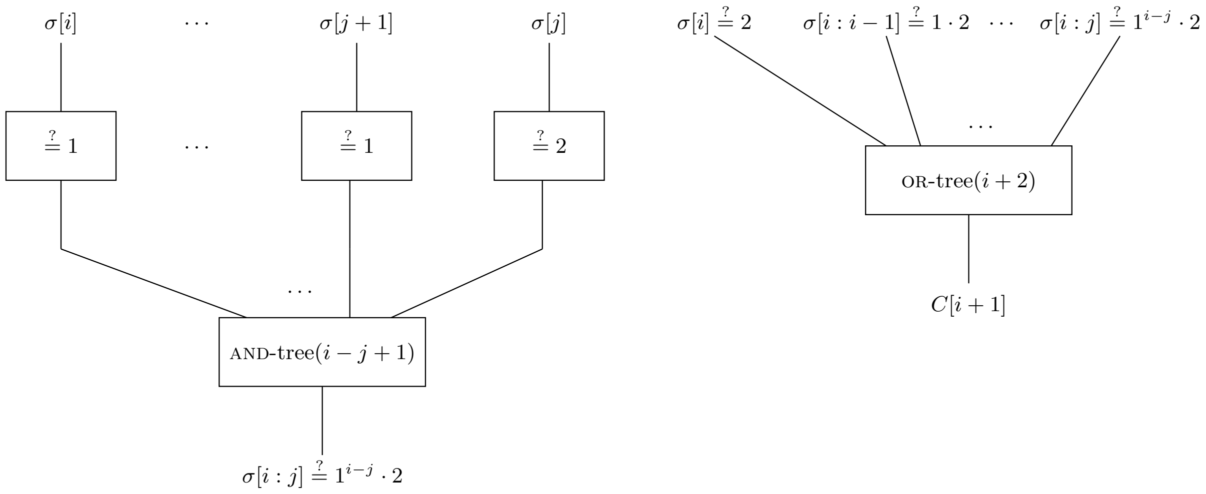
$$C[i+1] = 1 \iff \exists j \leq i \ : \ \sigma[i:j] = 1^{i-j} \cdot 2.$$

Corollary:

$$\begin{aligned} C[i+1] = & (\sigma[i:-1] = 1^{i+1} \cdot 2) \\ & \vee (\sigma[i:0] = 1^{i} \cdot 2) \\ & \vee (\sigma[i:1] = 1^{i-1} \cdot 2) \\ & \vdots \\ & \vee (\sigma[i:i-1] = 1 \cdot 2) \\ & \vee (\sigma[i] = 2) \end{aligned}$$

Carry-Lookahead Circuit

$$C[i+1] = 1 \quad \Longleftrightarrow \quad \exists j \le i \ : \ \sigma[i:j] = 1^{i-j} \cdot 2.$$



Carry-Lookahead Circuit: cost & delay

- constant cost & depth comparison gates for deciding if :
  - $\sigma[i] = 1$
  - $\sigma[i] = 2$
- Use a row of comparison gates for $\sigma[i:j]$.
- Feed outputs of comparison gates to AND-tree$(i-j+1)$.
- Cost of test $\sigma[i:j] = 1^{i-j} \cdot 2$

$$c(\text{AND-tree}(i-j+1)) + (i-j+1) \cdot c(\text{comparison}) = \Theta(i-j).$$

- Delay of test $\sigma[i:j] = 1^{i-j} \cdot 2$

$$d(\text{comparison}) + (\text{AND-tree}(i+j+1)) = \Theta(\log(i-j)).$$

Carry-Lookahead Circuit: cost & delay - cont.

- Test if $\sigma[i:j] = 1^{i-j} \cdot 2$ for $j = -1, 0, \ldots, i$.
- Cost of computing $C[i+1]$:

$$\sum_{j=-1}^{i} c(\text{testing if } \sigma[i:j] = 1^{i-j} \cdot 2) = \sum_{j=-1}^{i} \Theta(i-j) = \Theta(i^2).$$

- Delay of computing $C[i+1]$:

$$\max_{j=-1\ldots i} \Theta(\log(i-j)) = \Theta(\log i).$$

- $\Rightarrow$ cost of computing $C[n:1]$: $\sum_{i=0}^{n-1} \Theta(i^2) = \Theta(n^3)$.
- ...usually applied only to short blocks (e.g. 4 bits)

Carry-Lookahead Adder: typical description

SWITCHING EXPRESSIONS

- INTERMEDIATE CARRIES:

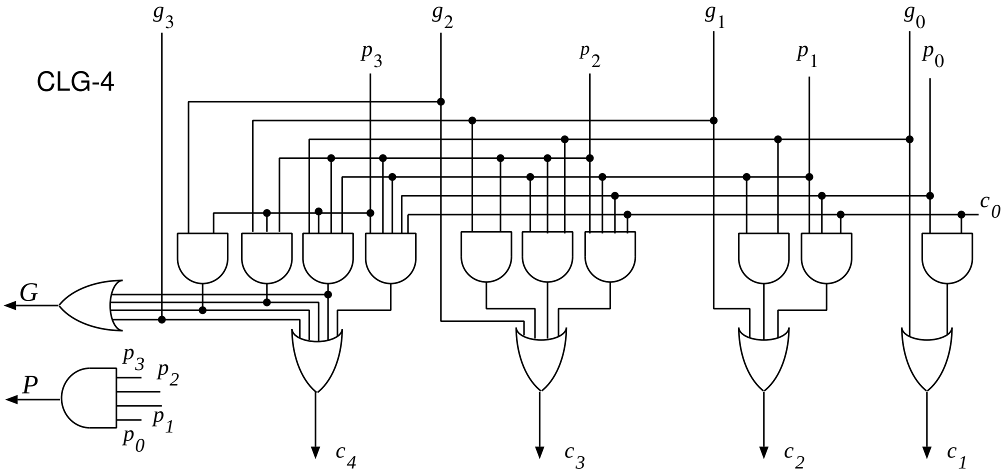
$$c_{i+1} = g_i + p_i \cdot c_i$$

BY SUBSTITUTION,

$$\begin{aligned}
c_1 &= g_0 + p_0c_0 \\
c_2 &= g_1 + p_1c_1 \\
&= g_1 + p_1g_0 + p_1p_0c_0 \\
c_3 &= g_2 + p_2c_2 \\
&= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0 \\
c_4 &= g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0
\end{aligned}$$

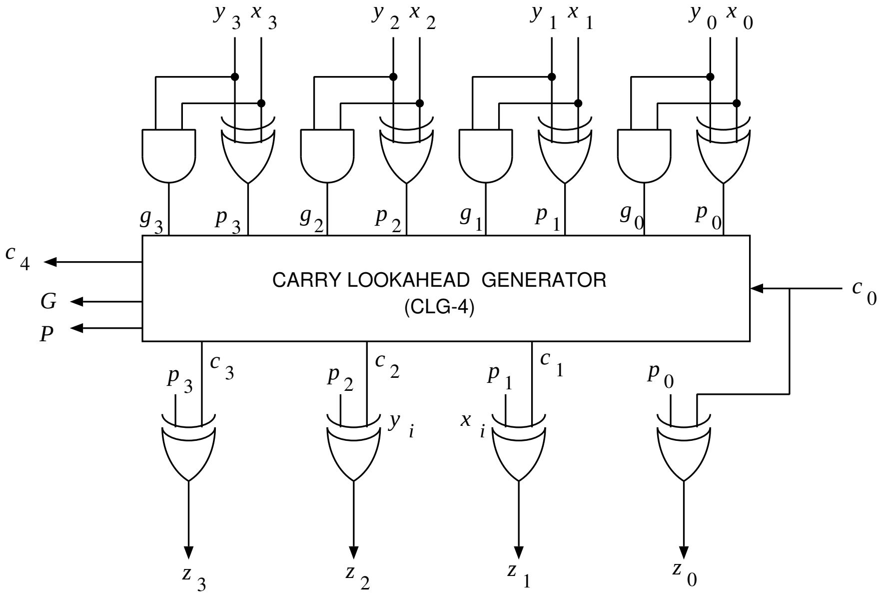
from: Introduction to Digital Systems, M.D. Ercegovac, T. Lang, and J.H. Moreno, Wiley and Sons, 1998.

Carry-Lookahead Adder: typical description



from: Introduction to Digital Systems, M.D. Ercegovac, T. Lang, and J.H. Moreno, Wiley and Sons, 1998. – p.251

Carry-Lookahead Adder: typical description



from: Introduction to Digital Systems, M.D. Ercegovac, T. Lang, and J.H. Moreno, Wiley and Sons, 1998. – p.252

Two-level Carry-Lookahead Adder

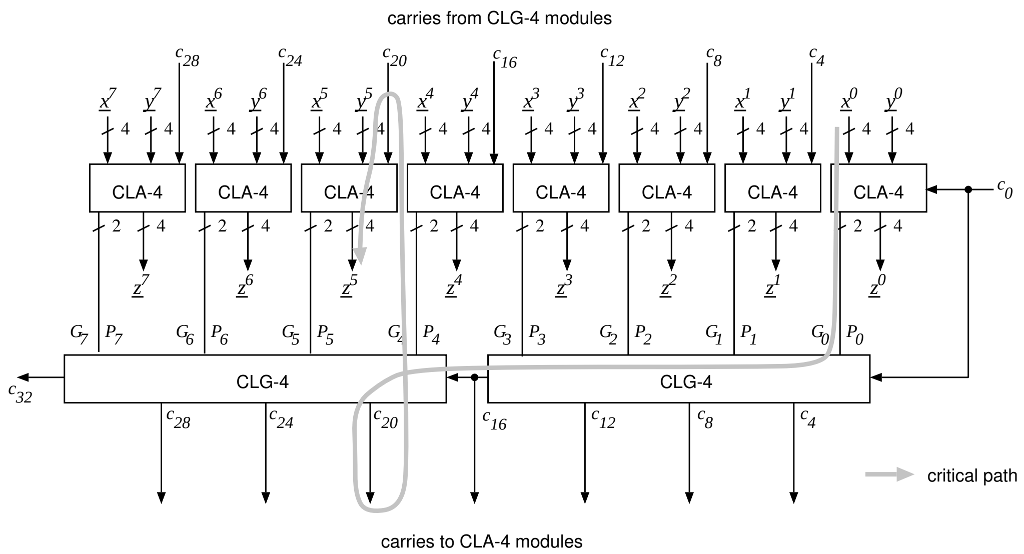


Figure 10.7: 32-BIT CARRY-LOOKAHEAD ADDER USING CLA-4 AND CLG-4 MODULES.

from: Introduction to Digital Systems, M.D. Ercegovac, T. Lang, and J.H. Moreno, Wiley and Sons, 1998. – p.253

Definition of $* : \{0,1,2\} \times \{0,1,2\} \longrightarrow \{0,1,2\}$

$*$	0	1	2
0	0	0	0
1	0	1	2
2	2	2	2

Remark: for every $a \in \{0,1,2\}$:

$$0 * a = 0$$
$$1 * a = a$$
$$2 * a = 2.$$

Claim: (homework) $*$ is an associative function. Namely,

$$\forall a, b, c \in \{0,1,2\} : (a * b) * c = a * (b * c).$$

Question: Is $*$ commutative?

$*$-products

For $j \geq i$:

$$\prod[j:i] \triangleq \sigma[j] * \cdots * \sigma[i].$$

Associativity of $*$ implies that for every $i \leq j < k$:

$$\prod[k:i] = \prod[k:j+1] * \prod[j:i].$$

A stronger claim

Claim: For every $-1 \leq i \leq n-1$,

$$C[i+1] = 1 \qquad \Longleftrightarrow \qquad \prod[i:-1] = 2.$$

Corollary: Can compute $C[i+1]$ using a $*$-tree$(i+2)$.

- $\Rightarrow$

$$c(\text{compute } C[i+1]) = O(i)$$
$$d(\text{compute } C[i+1]) = O(\log i).$$

- $\Rightarrow$

$$c(\text{compute } C[n:1]) = \sum_{i=1}^{n} O(i) = O(n^2)$$
$$d(\text{compute } C[n:1]) = O(\log n).$$

explains carry-lookahead generator... still too expensive!

Proof: $C[i+1] = 1 \iff \prod[i:-1] = 2$

From previous claim, it suffices to prove that

$$\exists j \leq i \ : \ \sigma[i:j] = 1^{i-j} \cdot 2 \qquad \iff \qquad \prod[i:-1] = 2.$$

Proof: $\sigma[i:j] = 1^{i-j} \cdot 2 \Rightarrow \prod[i:-1] = 2$

- Assume that $\sigma[i:j] = 1^{i-j} \cdot 2$.
- $\Rightarrow$
$$\prod[i:j] = 2.$$
- If $j = -1$ we are done.
- Otherwise,
$$\begin{aligned}\prod[i:-1] &= \underbrace{\prod[i:j]}_{=2} * \prod[j-1:-1] \\ &= 2.\end{aligned}$$

Proof: $\prod[i : -1] = 2 \Rightarrow \exists j \leq i \;:\; \sigma[i : j] = 1^{i-j} \cdot 2$

- Assume that $\prod[i : -1] = 2$.
- If, for every $\ell \leq i$, $\sigma[\ell] \neq 2$, then $\prod[i : -1] \neq 2$, a contradiction. Hence
$$\{\ell \in [-1, i] \;:\; \sigma[\ell] = 2\} \;\neq\; \emptyset.$$
- Let
$$j \triangleq \max\{\ell \in [-1, i] \;:\; \sigma[\ell] = 2\}.$$
- $\prod[j : -1] = 2$ (since $2 * a = 2$).
- We claim that $\sigma[\ell] = 1$, for every $j < \ell \leq i$.

Proof: $\prod[i : -1] = 2 \Rightarrow \exists j \leq i \;:\; \sigma[i : j] = 1^{i-j} \cdot 2$

Let
$$j \triangleq \max\{\ell \in [-1, i] \;:\; \sigma[\ell] = 2\}.$$
We claim that $\sigma[\ell] = 1$, for every $j < \ell \leq i$.

- max. of j $\Rightarrow$ for every $j < \ell \leq i$: $\sigma[\ell] \neq 2$.
- if $\sigma[\ell] = 0$, for $j < \ell \leq i$, then $\prod[i : \ell] = 0$.
  $\Rightarrow$
  $$\prod[i : -1] = \prod[i : \ell] * \prod[\ell - 1 : -1] = 0,$$
  a contradiction.
- since $\sigma[i : j + 1] = 1^{i-j}$, we conclude that $\sigma[i : j] = 1^{i-j} \cdot 2$, QED.

Prefix Computation Problem

DEF: Let Σ denote a finite alphabet. Let $\textsc{op} : \Sigma^2 \longrightarrow \Sigma$ denote an associative function. A prefix computation over Σ with respect to $\textsc{op}$ is defined as follows.

Input $x[n-1:0] \in \Sigma^n$.

Output: $y[n-1:0] \in \Sigma^n$ defined recursively as follows:

$$y[0] \leftarrow x[0]$$
$$y[i+1] = \textsc{op}(x[i+1], y[i]).$$

Note that $y[i]$ can be also expressed simply by

$$y_i = \textsc{op}_{i+1}(x[i], x[i-1], \ldots, x[0]).$$

Reduction: $C[n:1] \longmapsto$ Prefix Computation Prob.

The Claim

$$C[i+1] = 1 \qquad \Longleftrightarrow \qquad \prod[i:-1] = 2$$

implies a reduction to the Prefix Computation Problem:

- $\Sigma = \{0, 1, 2\}$
- $\textsc{op} = *$
- input: $\sigma[-1:n]$
- output: $y[i] = \prod[i:-1]$.

Prefix Computation Problem - example

- $\Sigma = \{0, 1\}$
- OP $=$ OR

$\Longrightarrow$ Unary Priority Encoder U-PENC(n).

Parallel Prefix Circuit

DEF: A Parallel Prefix Circuit, PPC–OP(n), is a combinational circuit that computes a prefix computation. Namely, given input $x[n-1:0] \in \Sigma^n$, it outputs $y[n-1:0] \in \Sigma^n$, where

$$y_i = \text{OP}_{i+1}(x[i], x[i-1], \ldots, x[0]).$$

- representation of values in Σ - not addressed.
- assume: some fixed representation is used.
- OP-gate: given representations of $a, b \in \Sigma$, outputs a representation of OP(a, b).

PPC–OP(n) - questions

Question: Design a PPC–OP(n) circuit with linear delay and cost.

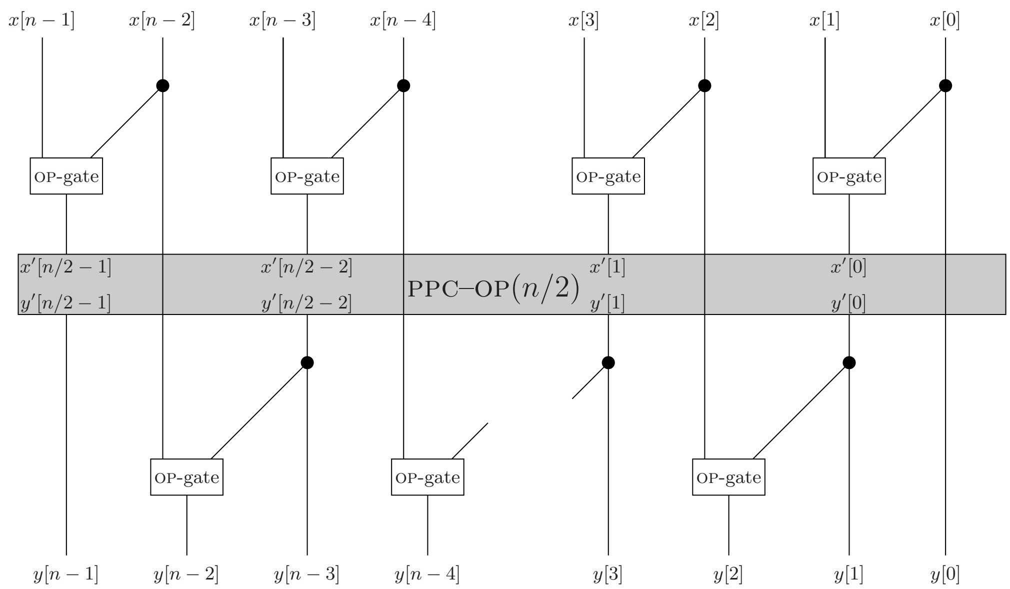
Question: Design a PPC–OP(n) circuit with logarithmic delay and quadratic cost.

Question: Assume that a design $C(n)$ is a PPC–OP(n). This means that it is comprised only of OP-gates and works correctly for every alphabet Σ and associative function OP : $\Sigma^2 \to \Sigma$. Can you prove a lower bound on its cost and delay?

PPC–OP - implementation

- A recursive design.
- We already saw a divide-and-conquer design for U-PENC(n) with cost $\Theta(n \cdot \log n)$.
- Aim for $O(n)$ cost.
- "odd-even" divide-and-conquer (as opposed to left/right side divide-and-conquer).
- basis $n = 2$: an OP-gate.
- recursion step...

PPC–OP(n) - recursion step



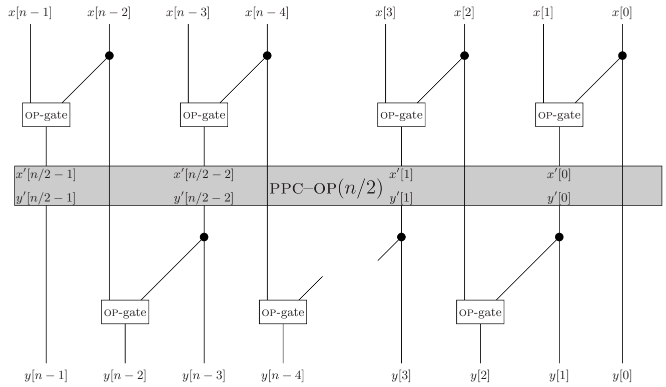
PPC–OP(n) - correctness

- By induction. Basis: holds trivially for $n = 2$. We now prove the induction step.
- $x'[n/2-1:0], y'[n/2-1]$ - inputs/outputs of PPC–OP$(n/2)$.
- $x'[i] \leftarrow \text{OP}(x[2i+1], x[2i])$.
- Induction hypothesis:

$$\begin{aligned} y'[i] &= \text{OP}_{i+1}(x'[i], \ldots, x'[0]) \\ &= \text{OP}_{2i+2}(x[2i+1], \ldots, x[0]). \end{aligned}$$

- $y[2i+1] \leftarrow y'[i] \Rightarrow$ odd indexed outputs $y[1], y[3], \ldots, y[n-1]$ are correct.
- $y[2i] \leftarrow \text{OP}(x[2i], y'[i-1]) \Longrightarrow y[2i] = \text{OP}(x[2i], y[2i-1])$.
- $\Rightarrow$ even indexed outputs are also correct. QED

PPC–OP(n) - delay analysis $(n = 2^k)$

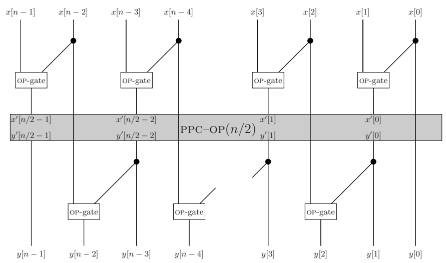


$$d(\text{PPC–OP}(n)) = \begin{cases} d(\text{OP-gate}) & \text{if } n = 2 \\ d(\text{PPC–OP}(n/2)) + 2 \cdot d(\text{OP-gate}) & \text{otherwise.} \end{cases}$$

If follows that

$$d(\text{PPC–OP}(n)) = (2\log n - 1) \cdot d(\text{OP-gate}).$$

PPC–OP(n) - cost analysis $(n = 2^k)$



$$c(\text{PPC–OP}(n)) = \begin{cases} c(\text{OP-gate}) & \text{if } n = 2 \\ c(\text{PPC–OP}(n/2)) + (n-1) \cdot c(\text{OP-gate}) & \text{otherwise.} \end{cases}$$

It follows that

$$\begin{aligned} c(\text{PPC–OP}(n)) &= \sum_{i=2}^{k} (2^i - 1) \cdot c(\text{OP-gate}) + c(\text{OP-gate}) \\ &= (2n - 4 - (k-1) + 1) \cdot c(\text{OP-gate}) \\ &= (2n - \log n - 2) \cdot c(\text{OP-gate}). \end{aligned}$$

PPC–OP(n) - corollary

Corollary: If the delay and cost of an OP-gate is constant, then

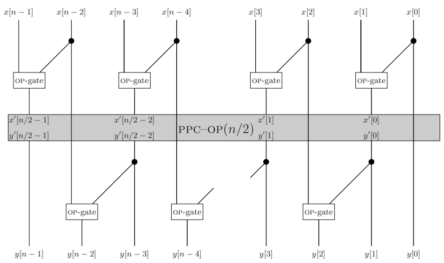
$$d(\text{PPC–OP}(n)) = \Theta(\log n)$$
$$c(\text{PPC–OP}(n)) = \Theta(n).$$

$\Rightarrow$

- $\Sigma = \{0, 1\}$ & OP $=$ OR
  $\Rightarrow$ asymptotically optimal U-PENC(n).
- $\Sigma = \{0, 1, 2\}$ & OP $= *$
  $\Rightarrow$ compute carry-bits $C[n : 1]$ with $O(n)$ cost and $O(\log n)$ delay.

PPC–OP(n) - fanout

- Insert a buffer in every branching point of the PPC–OP(n) design.
- $\Rightarrow$ constant fanout.
- Question: What is the maximum fanout in the PPC–OP(n) design. Analyze the effect of inserting buffers to the cost and delay of PPC–OP(n).



putting it all together

Compute $\sigma[n-1:-1]$: Cost & delay are constant per $\sigma[i]$.
$\Rightarrow$ total cost is $O(n)$ & the total delay is $O(1)$.

PPC$-*(n)$: Compute product $\prod[i:-1]$ from $\sigma[i:-1]$, for every $i \in [n-1:0]$.
The cost $O(n)$ and delay $O(\log n)$.

Extraction of $C[n:1]$: Recall $C[i+1]=1$ iff $\prod[i:-1]=2$.
Compare each $\prod[i:-1]$ with 2. The result of this comparison equals $C[i+1]$.
The cost and delay is constant per carry-bit $C[i+1]$.
Total cost of this step is $O(n)$ and the delay is $O(1)$.

Computation of sum-bits: The sum bits are computed by

$$S[i] = \text{XOR}_3(A[i], B[i], C[i]).$$

Cost of this step is $O(n)$ and the delay is $O(1)$.

Fast Addition

By combining the cost and delay of each stage we obtain the following result.

Theorem: The adder based on parallel prefix computation is asymptotically optimal; its cost is linear and its delay is logarithmic.

Summary

- Presented an adder with asymptotically optimal cost and delay.
- Design based on two reductions:
  - reduction of the task of computing the sum-bits to the task of computing the carry bits.
  - reduction of the task of computing the carry bits to a prefix computation problem.
- A prefix computation problem is the problem of computing $\text{OP}_i(x[i-1:0])$, for $0 \leq i \leq n-1$, where OP is an associative operation.
- PPC–OP(n) - a linear cost logarithmic delay circuit for the prefix computation problem.
- Can use PPC–OP(n) for asymptotically optimal U-PENC(n).

Chapter 8: Signed Addition

Computer Structure

&

Intro. to Digital Computers

Dr. Guy Even

Tel-Aviv Univ.

Goals

- represent negative numbers
- two’s complement representation
- add & subtract two’s complement numbers
- identify overflow and negative result

Signed numbers

- unsigned numbers - non-negative integers
- signed numbers - positive/negative numbers
- Many ways to represent signed numbers

Representation of signed numbers

- The number represented in sign-magnitude representation by $A[n-1:0] \in \{0,1\}^n$ and $S \in \{0,1\}$ is

$$(-1)^S \cdot \langle A[n-1:0] \rangle .$$

- The number represented in one's complement representation by $A[n-1:0] \in \{0,1\}^n$ is

$$-(2^{n-1}-1) \cdot A[n-1] + \langle A[n-2:0] \rangle .$$

- The number represented in two's complement representation by $A[n-1:0] \in \{0,1\}^n$ is

$$-2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle .$$

Two's complement - examples

- We denote the number represented in two's complement representation by $A[n-1:0]$ as follows:

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

- Examples:
  - $[0^n] = 0$.
  - $[0 \cdot x[n-2:0]] = \langle x[n-2:0] \rangle$.
  - $[1 \cdot x[n-2:0]] = -2^{n-1} + \langle x[n-2:0] \rangle < 0$.
  - $\Rightarrow$ MSB indicates the sign.
  - $[1^n] = -1$.
  - $\left[1 \cdot 0^{n-1}\right] = -2^{n-1}$.

Two's complement - story

- The most common method for representing signed numbers is two's complement.
- Why? adding, subtracting, and multiplying signed numbers represented in two's complement representation is almost as easy as performing these computations on unsigned (binary) numbers.
- We will discuss addition & subtraction.

Question: Suppose that the string A represents the value x.

Negation means computing the string B that represents $-x$.

Suggest circuit for negation with respect to sign-magnitude representation and one's complement representation.

Two's complement - notation

T_n - the set of signed numbers that are representable in two's complement representation using n-bit binary strings.

Claim:

$$T_n \triangleq \left\{-2^{n-1}, -2^{n-1}+1, \ldots, 2^{n-1}-1\right\}.$$

Question: Prove the claim.

Two's complement - negation

Claim:

$$-[A[n-1:0]] = [\text{INV}(A[n-1:0])] + 1.$$

Proof: Note that $\text{INV}(A[i]) = 1 - A[i]$. Hence,

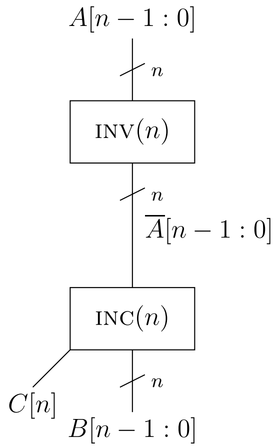
$$\begin{aligned}
[\text{INV}(A[n-1:0])] &= -2^{n-1} \cdot \text{INV}(A[n-1]) + \langle \text{INV}(A[n-2:0]) \rangle \\
&= -2^{n-1} \cdot (1 - A[n-1]) + \sum_{i=0}^{n-2} (1 - A[i]) \cdot 2^i \\
&= \underbrace{-2^{n-1} + \sum_{i=0}^{n-2} 2^i}_{=-1} + \underbrace{2^{n-1} \cdot A[n-1] - \sum_{i=0}^{n-2} A[i] \cdot 2^i}_{=-[A[n-1:0]]} \\
&= -1 - [A[n-1:0]] \,.
\end{aligned}$$

QED.

A circuit for negating a two's complement number

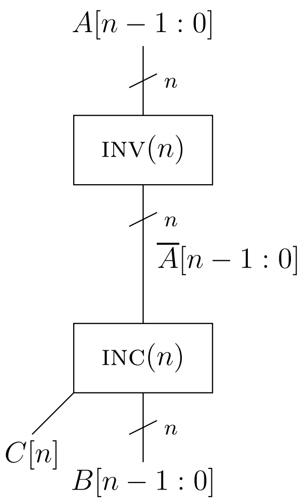
Claim:

$$-[A[n-1:0]] = [\text{INV}(A[n-1:0])] + 1.$$



Question: $[B[n-1:0]] \stackrel{?}{=} -[A[n-1:0]]$

A circuit for negating a two's complement number - cont.



The increment circuit computes:

$$\langle \overline{A}[n-1:0] \rangle + 1.$$

However, we should compute

$$\left[\overline{A}[n-1:0]\right] + 1.$$

We know that

$$\langle C[n] \cdot B[n-1:0] \rangle = \langle \overline{A}[n-1:0] \rangle + 1.$$

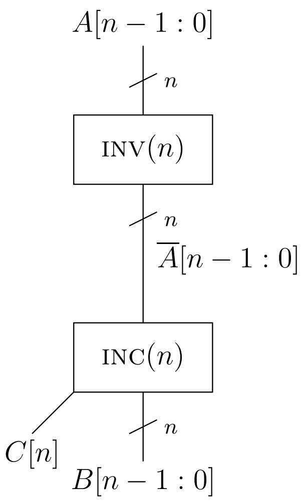
Suppose we are "lucky" and $C[n] = 0$.

$$\langle B[n-1:0] \rangle = \langle \overline{A}[n-1:0] \rangle + 1.$$

Why should this imply that

$$[B[n-1:0]] = \left[\overline{A}[n-1:0]\right] + 1?$$

A circuit for negating a two's complement number - cont.



Counter example:

$$A[n-1:0] = 1 \cdot 0^{n-1}.$$

$$\overline{A}[n-1:0] = 0 \cdot 1^{n-1}.$$

Increment yields $C[n] = 0$ and

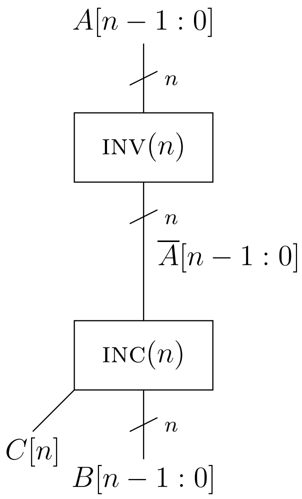
$$B[n-1:0] = 1 \cdot 0^{n-1} = A[n-1:0].$$

$$\Longrightarrow \left[\vec{B}\right] \neq -\left[\vec{A}\right].$$

Reason? binary increment is not a two's complement increment.

Had to err: $-\left[\vec{A}\right] \notin T_n$.

A circuit for negating a two's complement number - cont.



We will prove a theorem that will help us formulate and prove the correctness of the negation circuit.

Two's complement - mod 2^n property

Claim: For every $A[n-1:0] \in \{0,1\}^n$

$$\text{mod}(\langle \vec{A} \rangle, 2^n) = \text{mod}(\left[\vec{A}\right], 2^n).$$

Note that

$$\langle \vec{A} \rangle \in [0, 2^n - 1]$$
$$\left[\vec{A}\right] \in [-2^{n-1}, 2^{n-1} - 1].$$

Remark: Alternative definition of two's complement representation based on Claim. Namely, represent $x \in [-2^{n-1}, 2^{n-1} - 1]$ by $x' \in [0, 2^n - 1]$, where $\text{mod}(x, 2^n) = \text{mod}(x', 2^n)$.

Claim: $\text{mod}(\langle \vec{A} \rangle, 2^n) = \text{mod}(\left[\vec{A}\right], 2^n)$

Proof:

$$\begin{aligned}\text{mod}(\langle \vec{A} \rangle, 2^n) &= \text{mod}(2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle, 2^n) \\ &= \text{mod}((2^{n-1} - 2^n) \cdot A[n-1] + \langle A[n-2:0] \rangle, 2^n) \\ &= \text{mod}(-2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle, 2^n) \\ &= \text{mod}(\left[\vec{A}\right], 2^n).\end{aligned}$$

$\square$

Two's complement - sign extension

Claim: If $A[n] = A[n-1]$, then

$$[A[n:0]] = [A[n-1:0]] \,.$$

Proof:

$$\begin{aligned}[A[n:0]] &= -2^n \cdot A[n] + \langle A[n-1:0] \rangle \\ &= -2^n \cdot A[n] + 2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle \\ &= -2^n \cdot A[n-1] + 2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle \\ &= -2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle \\ &= [A[n-1:0]] \,.\end{aligned}$$

QED

Two's complement - sign extension

Claim: If $A[n] = A[n-1]$, then

$$[A[n:0]] = [A[n-1:0]] .$$

Corollary:

$$[A[n-1]^* \cdot A[n-1:0]] = [A[n-1:0]] .$$

sign-extension - duplicating the most significant bit does not affect the value represented in two's complement representation. This is similar to padding zeros from the left in binary representation.

Theorem: signed addition $\longmapsto$ binary addition

- Binary addition: assume that

$$\langle C[n] \cdot S[n-1:0] \rangle = \langle A[n-1:0] \rangle + \langle B[n-1:0] \rangle + C[0].$$

- $C[n-1]$ - carry-bit in position $[n-1]$ associated with this binary addition.
-

$$z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0].$$

$\Longrightarrow$

$$\begin{aligned}
C[n-1] - C[n] = 1 \quad &\Longrightarrow \quad z > 2^{n-1} - 1 \\
C[n] - C[n-1] = 1 \quad &\Longrightarrow \quad z < -2^{n-1} \\
z \in T_n \quad &\Longleftrightarrow \quad C[n] = C[n-1] \\
z \in T_n \quad &\Longrightarrow \quad z = [S[n-1:0]] .
\end{aligned}$$

Theorem - proof

functionality of FA_{n-1} in $\text{RCA}(n) \Longrightarrow$

$$A[n-1] + B[n-1] + C[n-1] = 2C[n] + S[n-1]$$
$$\Rightarrow A[n-1] + B[n-1] = 2C[n] - C[n-1] + S[n-1].$$

We now expand z as follows:

$$\begin{aligned}
z &= [A[n-1:0]] + [B[n-1:0]] + C[0] \\
&= -2^{n-1} \cdot (A[n-1] + B[n-1]) \\
&\quad + \langle A[n-2:0]\rangle + \langle B[n-2:0]\rangle + C[0] \\
&= -2^{n-1} \cdot (2C[n] - C[n-1] + S[n-1]) + \langle C[n-1] \cdot S[n-2:0]\rangle \\
&= -2^{n-1} \cdot (2C[n] - C[n-1] - C[n-1]) + [S[n-1] \cdot S[n-2:0]] \\
&= -2^{n} \cdot (C[n] - C[n-1]) + [S[n-1:0]]\,.
\end{aligned}$$

Theorem - proof - cont

$$z = -2^{n} \cdot (C[n] - C[n-1]) + [S[n-1:0]]\,.$$

We distinguish between three cases:

1. If $C[n] - C[n-1] = 1$, then
$$\begin{aligned} z &= -2^n + [S[n-1:0]] \\ &\le -2^n + 2^{n-1} - 1 = -2^{n-1} - 1. \end{aligned}$$
2. If $C[n] - C[n-1] = -1$, then
$$\begin{aligned} z &= 2^n + [S[n-1:0]] \\ &\ge 2^n - 2^{n-1} = 2^{n-1}. \end{aligned}$$
3. If $C[n] = C[n-1]$, then $z = [S[n-1:0]]$, and obviously $z \in T_n$.

QED

Overflow

DEF: Let $z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0]$. The signal OVF is defined as follows:

$$\text{OVF} \triangleq \begin{cases} 1 & \text{if } z \notin T_n \\ 0 & \text{otherwise.} \end{cases}$$

- overflow - sum is either too large or too small.
- better term - out-of-range - not the common term.
- By Theorem

$$\text{OVF} = \text{XOR}(C[n-1], C[n]).$$

Detecting Overflow

- The signal $C[n-1]$ may not be available if one uses a "black-box" binary-adder (e.g., a library component in which $C[n-1]$ is an internal signal).
- In this case we detect overflow based on the following claim.

Claim:

$$\text{XOR}(C[n-1], C[n]) = \text{XOR}_4(A[n-1], B[n-1], S[n-1], C[n]).$$

Proof: Recall that

$$C[n-1] = \text{XOR}_3(A[n-1], B[n-1], S[n-1]).$$

$\square$

Determining the sign of the sum

- How do we determine the sign of the sum z?
- Obviously, if $z \in T_n$, then the sign-bit $S[n-1]$ indicates whether z is negative.
- What happens if overflow occurs?

Question: Provide an example in which the sign of z is not signaled correctly by $S[n-1]$.

We would like to be able to know whether z is negative regardless of whether overflow occurs.

Determining the sign of the sum - cont.

DEF: The signal NEG is defined as follows:

$$\text{NEG} \triangleq \begin{cases} 1 & \text{if } z < 0 \\ 0 & \text{if } z \geq 0. \end{cases}$$

Theorem implies that:

$$\text{NEG} = \begin{cases} S[n-1] & \text{if no overflow} \\ 1 & \text{if } C[n] - C[n-1] = 1 \\ 0 & \text{if } C[n-1] - C[n] = 1. \end{cases}$$

An even simpler method...

Claim: NEG $= \text{XOR}_3(A[n-1], B[n-1], C[n])$.

Proof:
The proof is based on playing the following "mental game":

- "extend" the computation to $n+1$ bits.
- $\Longrightarrow$ overflow does not occur in extended precision.
- $\Longrightarrow$ the sum bit in position n indicates correctly the sign of the sum z.
- express this sum bit using n-bit addition signals.

Proof: NEG $= \text{XOR}_3(A[n-1], B[n-1], C[n])$ - cont.

Sign extension to $n+1$ bits:

$$\tilde{A}[n:0] \triangleq A[n-1] \cdot A[n-1:0]$$

$$\tilde{B}[n:0] \triangleq B[n-1] \cdot B[n-1:0]$$

$$\langle \tilde{C}[n+1] \cdot \tilde{S}[n:0] \rangle \triangleq \langle \tilde{A}[n:0] \rangle + \langle \tilde{B}[n:0] \rangle + C[0].$$

Since sign-extension preserves value, it follows that

$$z = \left[\tilde{A}[n:0]\right] + \left[\tilde{B}[n:0]\right] + C[0].$$

Proof: NEG $= \text{XOR}_3(A[n-1], B[n-1], C[n])$ **- cont.**

We claim that $z \in T_{n+1}$. This follows from

$$\begin{aligned} z &= [A[n-1:0]] + [B[n-1:0]] + C[0] \\ &\leq 2^{n-1} - 1 + 2^{n-1} - 1 + 1 \\ &\leq 2^n - 1. \end{aligned}$$

Similarly $z \geq 2^{-n}$.

Since sign-extension preserves value and $z \in T_{n+1}$:

$$z \overset{\text{sign-ext}}{=} \left[\tilde{A}[n:0]\right] + \left[\tilde{B}[n:0]\right] + C[0] \overset{\text{no OVF}}{=} \left[\tilde{S}[n:0]\right].$$

$$\Longrightarrow \quad \text{NEG} = \tilde{S}[n].$$

Proof: NEG $= \text{XOR}_3(A[n-1], B[n-1], C[n])$ **- cont.**

$$\begin{aligned} \text{NEG} &= \tilde{S}[n] \\ &= \text{XOR}_3(\tilde{A}[n], \tilde{B}[n], \tilde{C}[n]) \\ &= \text{XOR}_3(A[n-1], B[n-1], C[n]). \end{aligned}$$

QED

More on NEG

Question: Prove that $\text{NEG} = \text{XOR}(\text{OVF}, S[n-1])$.

A two's-complement adder - S-ADDER(n)

DEF:

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 $\text{NEG}, \text{OVF} \in \{0,1\}$.

Functionality: Define z as follows:

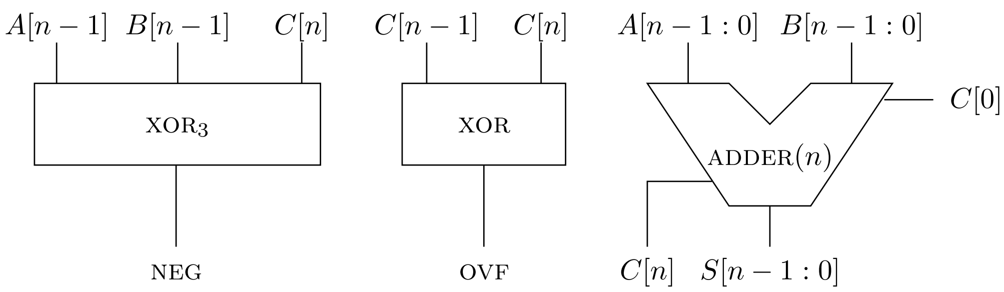
$$z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0].$$

The functionality is defined as follows:

$$\begin{aligned} z \in T_n &\implies [S[n-1:0]] = z \\ z \in T_n &\iff \text{OVF} = 0 \\ z < 0 &\iff \text{NEG} = 1. \end{aligned}$$

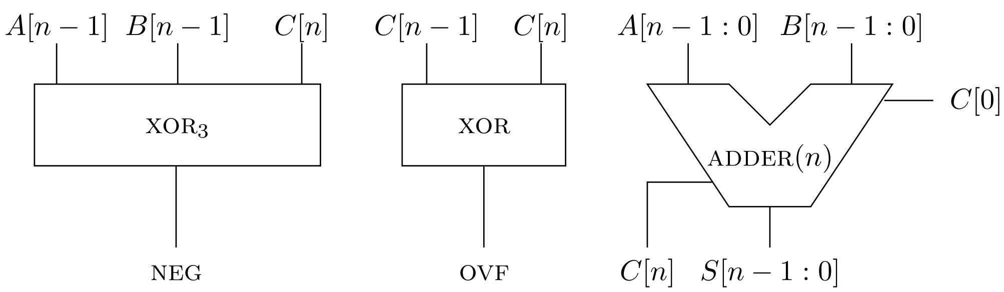
- Note that no carry-out $C[n]$ is output.

S-ADDER(n) - implementation



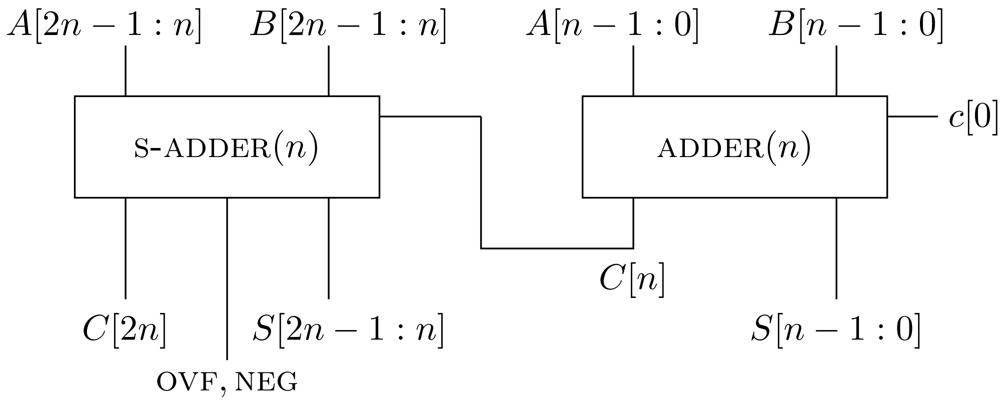
- a two's complement adder is identical to a binary adder except for the circuitry that computes the flags OVF and NEG.
- in an arithmetic logic unit (ALU), the same circuit is used for signed addition and unsigned addition.

S-ADDER(n) - correctness



Question: Prove that this design is correct.

Concatenating adders



Question: Is this a correct S-ADDER$(2n)$?

Question: How about a partition k and $2n-k$?

two's-complement adder/subtracter - ADD-SUB(n)

DEF:

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

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

Functionality: Define z as follows:

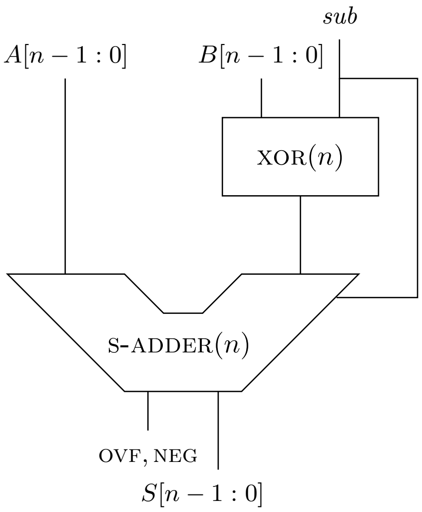
$$z \triangleq [A[n-1:0]] + (-1)^{\textit{sub}} \cdot [B[n-1:0]] .$$

The functionality is defined as follows:

$$\begin{aligned} z \in T_n &\implies [S[n-1:0]] = z \\ z \in T_n &\iff \text{OVF} = 0 \\ z < 0 &\iff \text{NEG} = 1. \end{aligned}$$

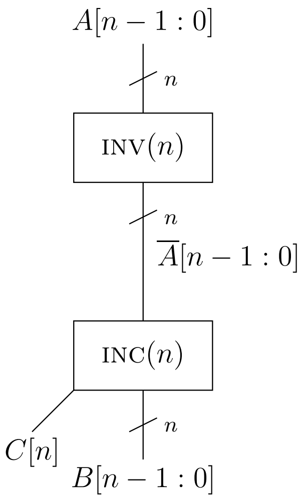
- *sub* - indicates if the operation is addition or subtraction.
- no carry-in bit $C[0]$ is input & no carry-out $C[n]$ is output.

ADD-SUB(n) - implementation



Question: Is this implementation correct?

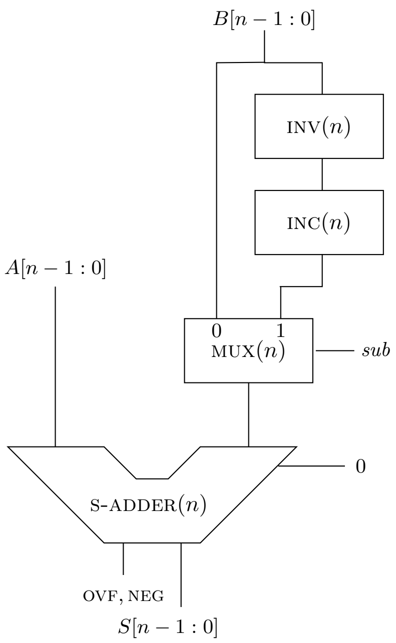
back to the negation circuit



Question:

1. When is the circuit correct?
2. Suppose we wish to add a signal that indicates whether the circuit satisfies $[\vec{B}] = -[\vec{A}]$. How should we compute this signal?
3. Does $C[n]$ indicate whether $[\vec{B}] \neq -[\vec{A}]$?

wrong implementation of ADD-SUB(n)



Question: Why is this design wrong?

OVF and NEG flags in high level programming

Question: High level programming languages such as C and Java do not enable one to see the value of the OVF and NEG signals (although these signals are computed by adders in all microprocessors).

1. Write a short program that deduces the values of these flags. Count how many instructiond are needed to recover these lost flags.
2. Short segements in a low level language (Assembly) can be integrated in C programs. Do you know how to see the values of the OVF and NEG flags using a low level language?

Summary

- representation of signed numbers: sign-magnitude, one's complement, two's complement.
- negation of two's complement numbers.
- reduction: two's complement addition $\longmapsto$ binary addition.
- Computation of OVF and NEG flags.
- two's complement adder and adder/subtracter.
- all these issues are important in: designing an ALU, DSP programming, and even regular programming (signed vs. unsigned int).

Chapter 9: Flip-Flops

Computer Structure
&
Intro. to Digital Computers

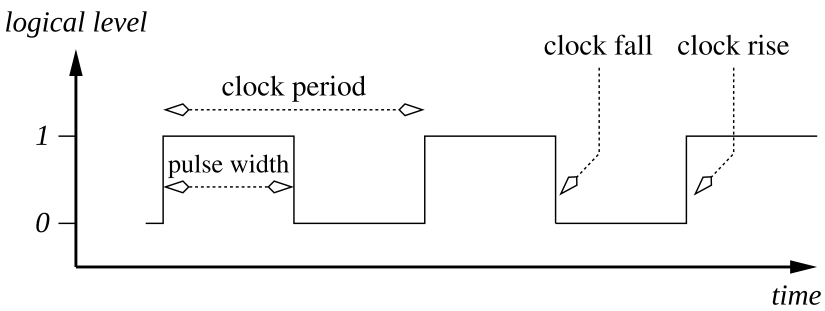
Dr. Guy Even

Tel-Aviv Univ.

Goals

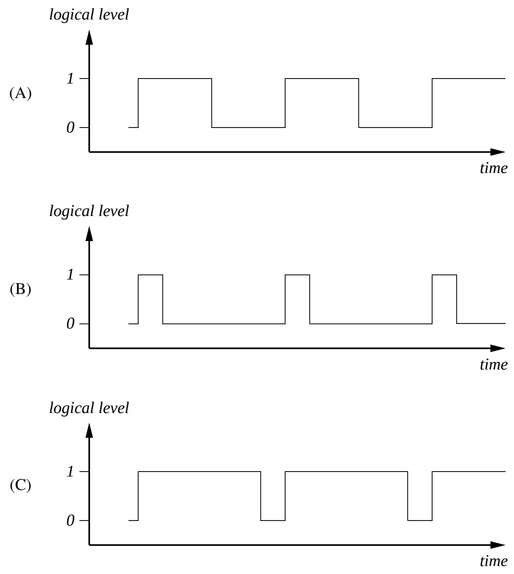
- introduce clock signal.
- define edge-triggered flip-flops.
- discuss parameters of flip-flops: setup time, hold time, contamination delay, propagation delay.
- explain importance of critical segment.
- understand timing of a flip-flop.
- other memory devices.

The Clock



- digital signal with periodic oscillations between 0 and 1.
- oscillations are instantaneous.
- each clock period starts with a $0 \rightarrow 1$ transition.
- $1 \rightarrow 0$ transition in the interior of the clock period.
- we denote the clock signal by CLK.

Clock terminology



- clock period - denoted by $\varphi(\text{CLK})$.
- clock pulse - interval during which $\text{CLK}(t) = 1$.
- CLK_{pw} - duration of clock pulse.
- symmetric clock - if $\text{CLK}_{pw} = \varphi(\text{CLK})/2$.
- narrow pulses - if $\text{CLK}_{pw} < \varphi(\text{CLK})/2$.
- wide pulses - if $\text{CLK}_{pw} > \varphi(\text{CLK})/2$.

Clock cycles

A clock partitions time into discrete intervals as follows:

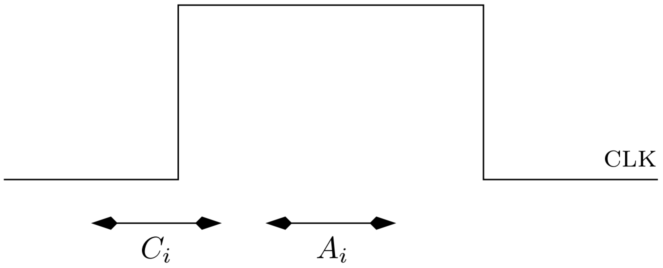
- Let t_i denote the starting time of the ith clock period.
- We refer to the half-closed interval $[t_i, t_{i+1})$ as *clock cycle* i.

Parameters of an Edge-triggered Flip-Flop

- Setup-time denoted by t_{su},
- Hold-time denoted by t_{hold},
- Contamination-delay denoted by t_{cont},
- Propagation-delay denoted by t_{pd}.

These parameters satisfy $-t_{su} < t_{hold} < t_{cont} < t_{pd}$.
Notation:

- critical segment: $C_i = [t_i - t_{su}, t_i + t_{hold}]$.
- instability segment: $A_i = [t_i + t_{cont}, t_i + t_{pd}]$.

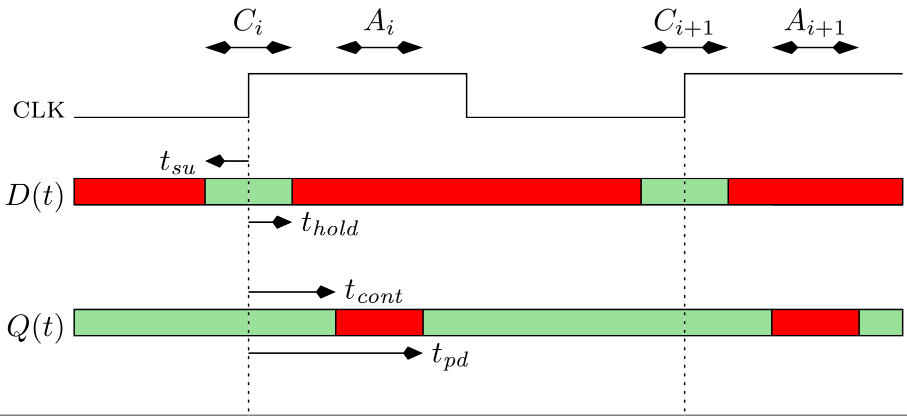


Definition: Edge-triggered Flip-Flop

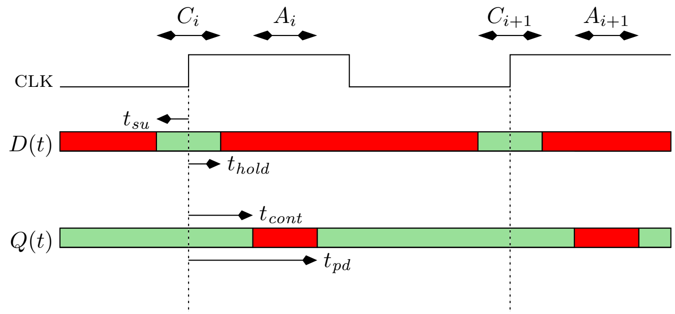
Inputs: A digital signal $D(t)$ and a clock CLK.

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

Functionality: If $D(t)$ is stable during the critical segment C_i, then $Q(t) = D(t_i)$ during the interval $(t_i + t_{pd}, t_{i+1} + t_{cont})$.

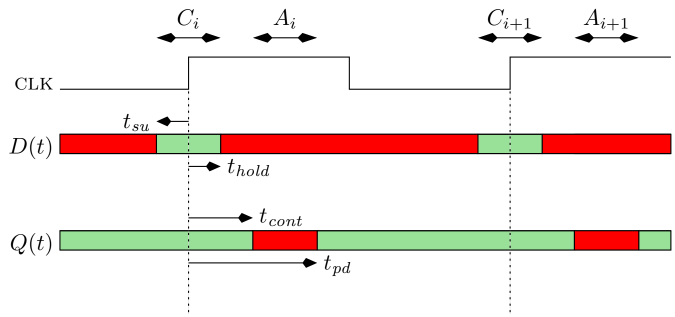


Remarks on definition of flip-flop



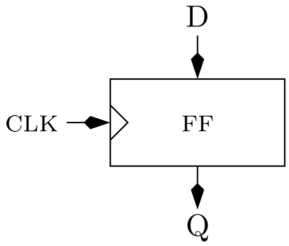
- $-t_{su} < t_{hold} < t_{cont} < t_{pd} \implies C_i \cap A_i = \emptyset$.
- Stability of $D(t)$ during $C_i \Rightarrow$ digital value of $D(t)$ during the critical segment C_i is logical and equals $D(t_i)$.
- Flip-flop samples $D(t)$ during C_i. The sampled value $D(t_i)$ is output during the interval $[t_i + t_{pd}, t_{i+1} + t_{cont}]$.
- Sampling is successful only if $D(t)$ is stable while it is sampled. This is why we refer to C as a critical segment.

Remarks on definition of flip-flop - cont.



- If the input $D(t)$ is stable during the critical segments $\{C_i\}_i$, then the output $Q(t)$ is stable in between the instability segments $\{A_i\}_i$.
- The stability of the input $D(t)$ during the critical segments depends on the clock period. We will later see that slowing down the clock (i.e. increasing the clock period) helps in achieving a stable $D(t)$ during the critical segments.

schematic of an edge triggered flip-flop



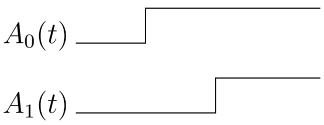
- clock port is marked by an “arrow”.
- we abbreviate and refer to an edge-triggered flip-flop simply as a flip-flop.

Question: Prove that an edge-triggered flip-flop is not a combinational circuit.

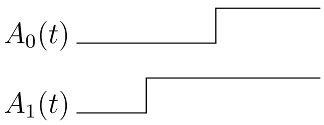
Arbitration

Arbitration is the problem of deciding which event occurs first.

Focus on the task of determining which of two signals reaches 1 first.



$A_0(t)$ reaches 1 first



$A_1(t)$ reaches 1 first

Definition: arbiter

Inputs: Non-decreasing analog signals $A_0(t), A_1(t)$ defined for every $t \geq 0$.

Output: An analog signal $Z(t)$.

Functionality: Assume that $A_0(0) = A_1(0) = 0$. Define T_i, for $i = 0, 1$, as follows:

$$T_i \triangleq \inf\{t \mid \mathit{dig}(A_i(t)) = 1\}.$$

Let $t' \triangleq 10 + \max\{T_0, T_1\}$. The output $Z(t)$ must satisfy, for every $t \geq t'$,

$$\mathit{dig}(Z(t)) = \begin{cases} 0 & \text{if } T_0 < T_1 - 1 \\ 1 & \text{if } T_1 < T_0 - 1 \\ 0 \text{ or } 1 & \text{otherwise.} \end{cases}$$

Arbiter - remarks

- $$T_i \triangleq \inf\{t \mid \mathit{dig}(A_i(t)) = 1\}.$$

  If T_0 or T_1 equals infinity, then $t' = \infty$, and there is no requirement on the output $Z(t)$.
- Arbiter circuit is given 10 time units starting from $\max\{T_0, T_1\}$ to determine if $T_0 < T_1$ or $T_1 < T_0$.
- tie: the case that $|T_0 - T_1| \leq 1$.
- In the case of a tie, the arbiter is free to decide, but must decide. $Z(t)$ is stable in the interval $[t, \infty)$.

Arbiters - an impossibility result

Claim: There does not exist a circuit C that implements an arbiter.

- Inherent limitation - not just a weakness of the digital abstraction.
- Use the claim to show that flip-flops must have critical segments.

Proof: every circuit C is not an arbiter

- Define $A_0(t)$ so that $T_0 = 100$ as follows:

$$A_0(t) \triangleq \begin{cases} \frac{t}{100} \cdot V_{\textit{high,in}} & \text{if } t \in [0, 100] \\ V_{\textit{high,in}} & \text{if } t > 100. \end{cases}$$

- Fix a parameter $x \in [-2, 2]$ and define $A_1(t)$ so that $T_1 = 100 + x$ as follows:

$$A_1(t) \triangleq \begin{cases} \frac{t}{100+x} \cdot V_{\textit{high,in}} & \text{if } t \in [0, 100 + x] \\ V_{\textit{high,in}} & \text{if } t > 100 + x. \end{cases}$$

- Define the function $f(x)$ by $f(x) \triangleq Z(200)$.
- We study the function $f(x)$ in the interval $x \in [-2, 2]$.

Proof: every circuit C is not an arbiter - cont.

- $x = -2 \Rightarrow T_1 = 100 + x = 98$. It follows that $A_1(t)$ "wins", and *dig*$(Z(200)) = 1$. Hence $f(-2) \geq V_{\textit{high,out}}$.
- $x = 2 \Rightarrow T_1 = 100 + x = 102$. It follows that $A_0(t)$ "wins", and *dig*$(Z(200)) = 0$. Hence $f(2) \leq V_{\textit{low,out}}$.
- claim: $f(x)$ is continuous (will prove this later).
- Mean Value theorem $\Rightarrow$

$$\forall y \in [V_{\textit{low,out}}, V_{\textit{high,out}}] \;\; \exists x \in [-2, 2] \; : \; f(x) = y.$$

- Pick y such that *dig*$(y) =$ non-logical.
- $\Rightarrow$ There exist valid inputs $A_0(t), A_1(t)$ with $t' \leq 112$, such that *dig*$(Z(200)) =$non-logical.
- $\Rightarrow$ C is not an arbiter. QED.

Proof: $f(x)$ is continuous

Rely on the assumption that an infinitesimal change in the energy of input signals causes an infinitesimal change in the energy of the output. Otherwise, noise would cause uncontrollable changes in $Z(t)$ and the circuit C would not be useful anyhow.

The output $Z(200)$ depends on the following:

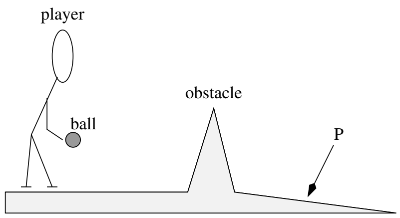
1. The initial state of the device C at time $t = 0$. We assume that the device C is in a stable state and that the charge is known everywhere.
2. The signal $A_i(t)$ in the interval $[0, 200]$, for $i = 0, 1$.

Proof: $f(x)$ is continuous - cont.

- Consider an infinitesimal change in x. This change affects $A_1(t)$ but does not affect $A_0(t)$ and the initial state.
- infinitesimal change of x $\Rightarrow$ infinitesimal difference in energy of $A_1(t)$.
- infinitesimal difference in energy of $A_1(t)$ $\Rightarrow$ infinitesimal difference in $Z(200)$.
- $\Rightarrow$ $f(x)$ is continuous.

Discussion: Arbiters - an impossibility result

- Claim is counter-intuitive.
- For every judge in a 100-meter dash, there exist two runners whose running times are such that the judge still hangs after an hour.
- Implies that there does not exist a perfect judge who can determine the winner in a 100-meters dash even if:
    1. high speed cameras located at the finish line and runners run very slowly.
    2. we allow the judge several hours to decide.
    3. we allow the judge to decide arbitrarily if the running times of the winner and runner-up are within a second.

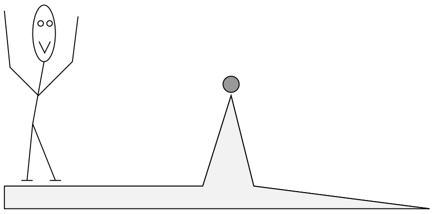


- Player - rolls a ball. Judge - announces decision if ball passes point P one day after.
- If speed of ball is above v', then ball passes the obstacle and then rolls past point P.
- If speed of ball is below v', then ball does not pass the obstacle.

Judge is in trouble:

- If speed$= v'$, then the ball reaches the tip of the obstacle and may remain there indefinitely long!
- If the ball remains on the obstacle's tip 24 hours past the throw, then the judge cannot announce her decision.

Meta-stability



- Meta-stability - a state of equilibrium (i.e. zero force) which is not a local minimum of energy (i.e. a slight force causes a movement away from the state).
- Inclined to say that the “probability of meta-stability occurring is very small”. This requires a probability distribution over the rolling speed v where

$$\lim_{\varepsilon \to 0} Pr(|v - v'| < \varepsilon) = 0.$$

Lessons learned

- Certain tasks are not achievable with probability 1.
  - coin toss might end up with the coin standing on its perimeter.
  - noise could be big enough to cause the digital value of a signal to flip from zero to one. (increase noise margin to reduce the probability of such an event.)

Reducing the probability of meta-stability

- Increase length of segment of instability.
  Increasing the delay of the arbiter (significantly) decreases the chances of meta-stability. E.g., ball resting on the tip of the obstacle is likely to fall to one of the sides.
- Increase the slope of the transfer function in the range of non-logical values. Similar to sharpening the tip of the obstacle.
- However, increasing the clock rate means that “decisions” must be made faster (i.e. within a clock period) and the chance of meta-stability increases.

Question

Does the proof of the Claim hold only if the signals $A_i(t)$ rise gradually?

Prove the claim with respect to "fast" non-decreasing signals $A_i(t)$. Namely, the length of the interval during which *dig*$(A_i(t))$ is non-logical equals ε.

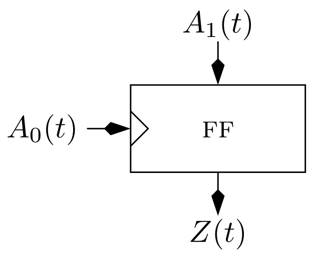
Flip-flops: necessity of critical segments

DEF: A flip-flop without a critical segment is a flip-flop in which the setup-time and hold-time satisfy $t_{\mathit{su}} = t_{\mathit{hold}} = 0$. The functionality is defined as follows:

- For every i, $Q(t)$ is logical (either zero or one) during the interval $t \in (t_i + t_{\mathit{pd}}, t_{i+1} + t_{\mathit{cont}})$ regardless of whether $D(t_i)$ is logical.
- If $D(t_i)$ is logical, then $Q(t) = D(t_i)$ during the interval $t \in (t_i + t_{\mathit{pd}}, t_{i+1} + t_{\mathit{cont}})$.

Just as the arbiter's decision is free if a tie occurs, the flip-flop is allowed to output zero or one if $D(t_i)$ is not logical. However, the output of the flip-flip must be logical once the instability segment ends.

An arbiter based on a flip-flop without a critical segment



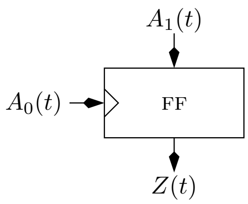
Assumptions:

- flip-flop is without a critical segment.
- $t_{cont}, t_{pd} \approx 10^{-9}$ time unit.
- intervals during which the inputs $A_0(t)$ and $A_1(t)$ are non-logical are also very short (e.g. 10^{-9} time unit).

Claim: The circuit above is an arbiter.

CORO: There does not exist a flip-flop without a critical section.

Remarks



- the signal $A_0(t)$ is input as a clock to the flip-flop, but $A_0(t)$ is not a clock.
- requirements from $A_0(t)$ are weaker than the requirements from a clock. Instead of periodic instantaneous transitions from zero to one and back, $A_0(t)$ is non-decreasing.
- the claim assumes only one "tick of the clock", so we may regard $A_0(t)$ as a clock with a very long period.
- proof of claim does not rely on $A_0(t)$ rising slowly; the claim holds regardless of the rate of change of $A_0(t)$.

Proof that circuit is an arbiter

We consider three cases:

- $|T_1 - T_0| \leq 1$: flip-flop's output $Z(t)$ is always logical at time $T_0 + t_{pd}$, so circuit functions properly.
- $T_1 < T_0 - 1$: if $T_1 < T_0 - 1$, then *dig*$(A_1(T_0)) = 1$. Hence sampled value equals 1, and hence, *dig*$(Z(t)) = 1$, for every $t \geq T_0 + t_{pd}$.
- $T_0 < T_1 - 1$: we claim that *dig*$(A_1(T_0)) = 0$, and hence, *dig*$(Z(t)) = 0$, for every $t \geq T_0 + t_{pd}$.

Proof that circuit is an arbiter - cont.

We need to show that $T_0 < T_1 - 1 \Rightarrow$ *dig*$(A_1(T_0)) = 0$.

- $T_0 < T_1 \Rightarrow$ *dig*$(A_1(T_0)) \in \{0,$ non-logical$\}$.
- assumption on the fast transition of *dig*$(A_1(t))$ implies:

$$dig(A_1(T_0)) = \text{non-logical} \;\Rightarrow\; dig(A_1(T_0 + 10^{-9})) = 1.$$

Hence, $T_1 \leq T_0 + 10^{-9}$ contradicting $T_1 > T_0 + 1$.

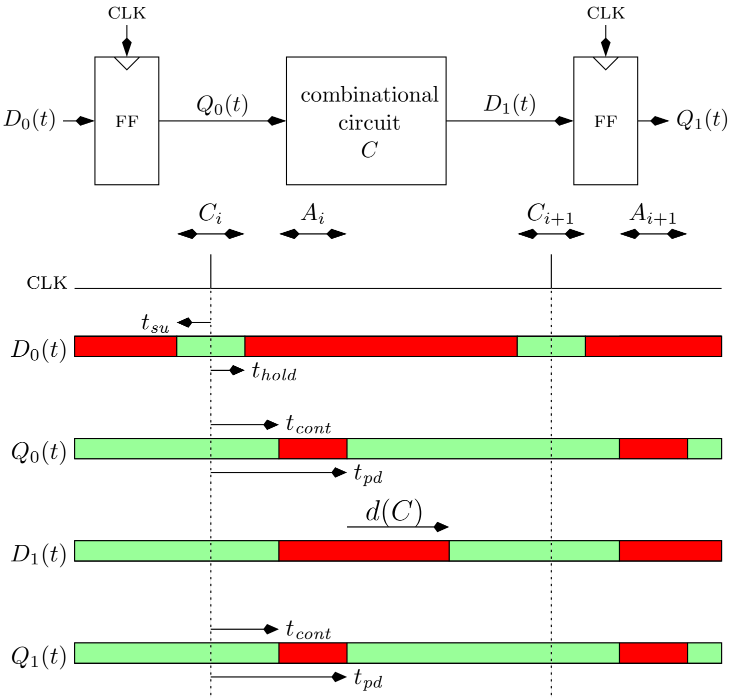
It follows that if $T_0 < T_1 - 1$, then *dig*$(A_1(T_0)) = 0$. QED

Corollary: conclusion

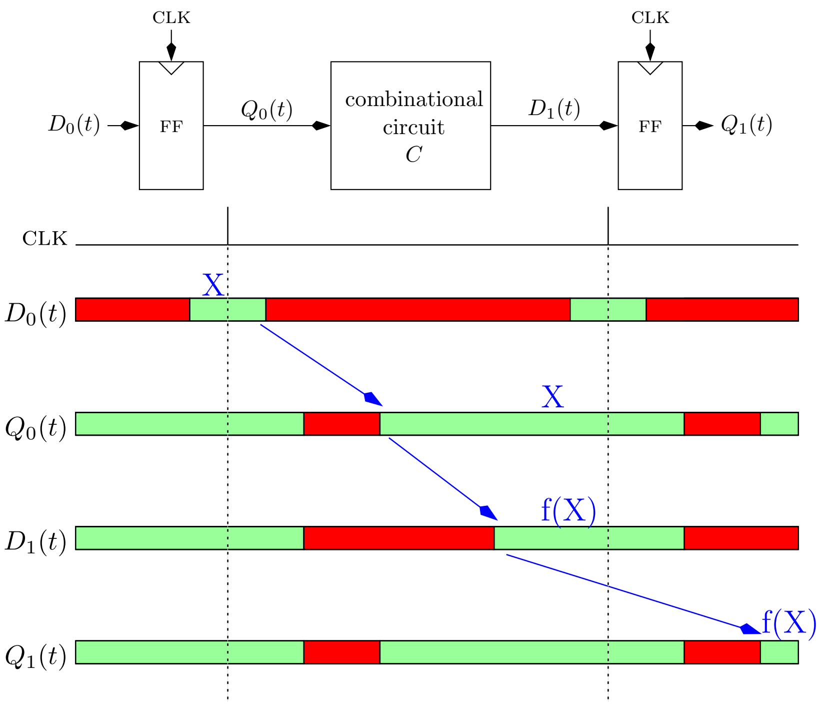
Critical segment is required to avoid meta-stability of the flip-flop.

Without critical segment, flip-flop's output can be non-logical even after $t_i + t_{pd}$.

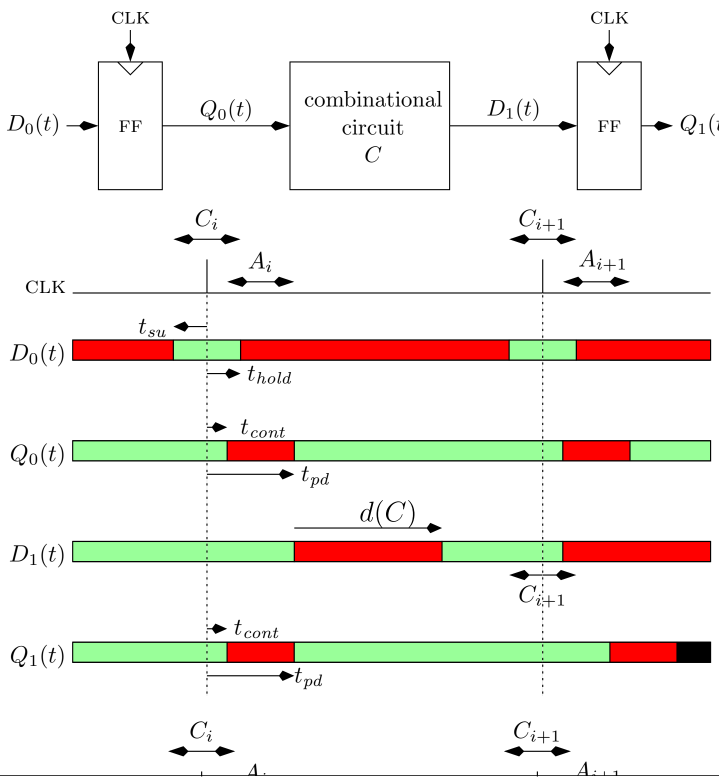
An example: timing



An example: functionality



Non-disjoint segments: $A_i \cap C_i \neq \emptyset$



What if $A_i \cap C_i \neq \emptyset$?

Stability interval of $D_1(t)$ is:

$$[t_i + t_{\textit{pd}} + d(C), t_{i+1} + t_{\textit{cont}}].$$

If $t_{\textit{cont}} < t_{\textit{hold}}$, then $D_1(t)$ is not stable during

$$C_{i+1} = [t_{i+1} - t_{\textit{su}}, t_{i+1} + t_{\textit{hold}}].$$

In this case, we need to rely on the contamination delay *cont*(C) of the combinational circuit C.

Now $D_1(t)$ is stable during the interval

$$[t_i + t_{\textit{pd}} + d(C), t_{i+1} + t_{\textit{cont}} + \textit{cont}(C)].$$

If $t_{\textit{cont}} + \textit{cont}(C) > t_{\textit{hold}}$, then the signal $D_1(t)$ is stable during the critical segment C_{i+1}, and correct functionality is obtained.

Contamination delay of combinational circuits

- Can help in obtaining stability during the critical segment.
- Many combinational gates have a positive contamination delay. But some don't.
- Relying on the contamination delay of combinational circuits complicates timing analysis.
- We use a strict assumption that *cont*$(C) = 0$, for every combinational circuit C. This does not cause incorrect circuits even if *cont*$(C) > 0$.

Fixing $A_i \cap C_i \neq \emptyset$

Question: Assume that we have an edge-triggered flip-flop FF in which $t_{hold} > t_{cont}$. Suppose that we have an inverter with a contamination delay $cont(\text{INV}) > 0$.

- Suggest how to design an edge-triggered flip-flop FF′ that satisfies $t_{hold}(\text{FF}') < t_{cont}(\text{FF}')$.
- What are the parameters of FF′?

D-Latch: parameters

- characterized by two parameters t_{su}, t_{hold}
- the critical segment is defined with respect to the falling edge of the clock.
- t'_i - time of the falling edge of the clock during the ith clock cycle.
- critical segment of a D-latch is

$$[t'_i - t_{su}, t'_i + t_{hold}].$$

- d - combinational delay of the D-latch.

D-Latch: definition

- During the interval $[t_i + d, t'_i)$, the output $Q(t)$ satisfies: $Q(t) = D(t)$, provided that $D(t)$ is stable during the interval $[t - d, t]$. We say that the D-latch is transparent during the interval $[t_i + d, t'_i)$.
- During the interval $(t'_i + t_{hold}, t_{i+1})$, if $D(t)$ is stable during the critical segment $[t'_i - t_{su}, t'_i + t_{hold}]$, then $Q(t) = D(t'_i)$. We say that the D-latch is opaque during the interval $(t'_i + t_{hold}, t_{i+1})$.

D-Latch : story

- D-latches are very important devices.
- D-latches are cheaper than flip-flops, and in fact, D-latches are the building blocks of flip-flops (e.g. master/slave designs).
- using D-latches wisely leads to faster designs.
- designs based on D-latches require multiple clock phases (or at least a clock CLK and its negation $\overline{\text{CLK}}$).
- Although timing with multiple clock phases is an important and interesting topic, we do not deal with it in this course.

Definition : clock enabled flip-flips

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

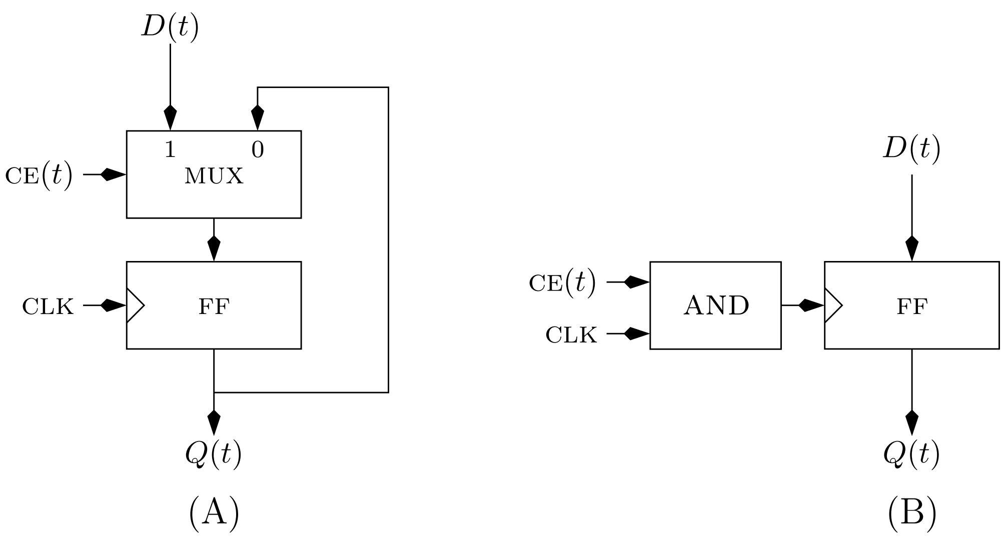
Output: A digital signal $Q(t)$.

Functionality: If $D(t)$ and $\text{CE}(t)$ are stable during the critical segment C_i, then for every $t \in (t_i + t_{pd}, t_{i+1} + t_{cont})$

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

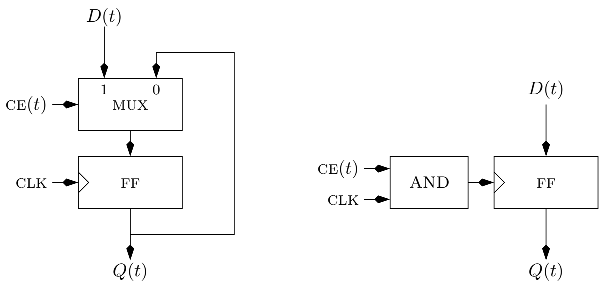
- $\text{CE}(t)$ - clock-enable signal.
- $\text{CE}(t)$ indicates whether the flip-flop samples the input $D(t)$ or maintains its previous value.

Clock enabled flip-flips : implementation



Question: Which design is correct?

Clock enabled flip-flips : implementation - cont

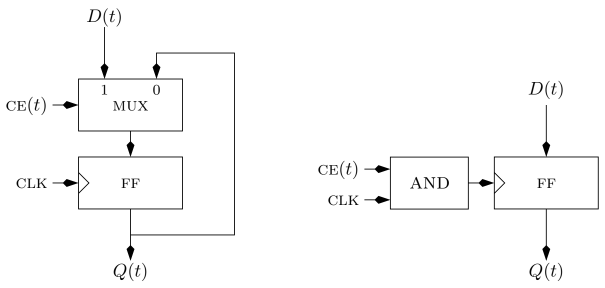


(A) (B)

Design (B) is wrong because:

- output of the AND-gate is not a clock signal (glitches).
- slow transitions of the output of the AND-gate (increase hold time)
- in some technologies, the flip-flop does not retain the stored bit forever. $\Rightarrow$ if CE$(t) = 0$ for a long period, then the flip-flop's output may become non-logical.

Clock enabled flip-flips : implementation - cont



(A) (B)

Question: Compute the parameters of the clock-enabled flip-flop depicted in part (A) in terms of the parameters of the edge-triggered flip-flop and the MUX.

Summary

- clock signal - definition, terminology
- define edge-triggered flip-flops
- prove that critical segments are crucial:
  - arbitration - the problem of deciding “whose first”
  - prove that arbiters do not exist
  - use this proof to show that critical segments are crucial
- a timing example
- other memory devices: D-latch & clock-enabled flip-flop

Chapter 10: Synchronous Circuits

Computer Structure
&
Intro. to Digital Computers

Dr. Guy Even

Tel-Aviv Univ.

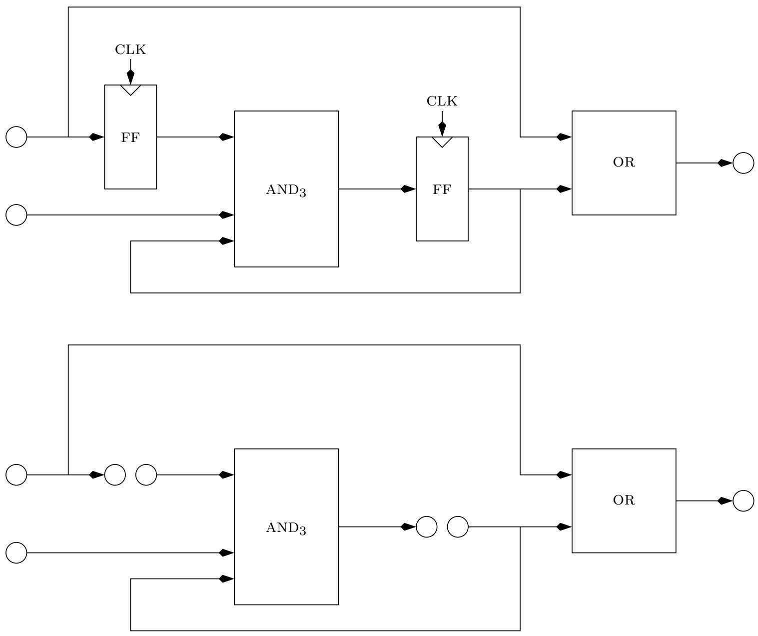
Goals

- define synchronous circuits.
- analyze timing (start with simple case...).
- define: timing constraints.
- find out if timing constraints are feasible.
- define: minimum clock period.
- algorithm: check if timing constraints are feasible.
- algorithm: compute minimum clock period.

Striping flip-flops away

- C - a circuit composed of combinational gates, nets, and flip-flops with a clock net called CLK.
- C' - a circuit obtained from C by:
  1. deleting the CLK net,
  2. deleting the input gate that feeds the CLK net, and
  3. replacing each flip-flip with an output gate (instead of the port D) and an input gate (instead of the port Q).

Striping flip-flops away - example



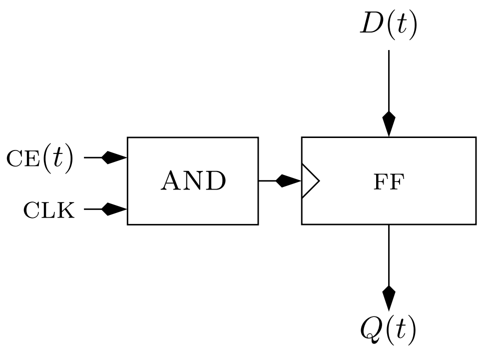
Definition: Synchronous Circuit

A synchronous circuit is a circuit C composed of combinational gates, nets, and flip-flops that satisfies the following conditions:

1. There is a net called CLK that carries a clock signal.
2. The CLK net is fed by an input gate.
3. The set of ports that are fed by the CLK net equals the set of clock-inputs of the flip-flops.
4. The circuit C' obtained from C by stripping away flip-flops is combinational.

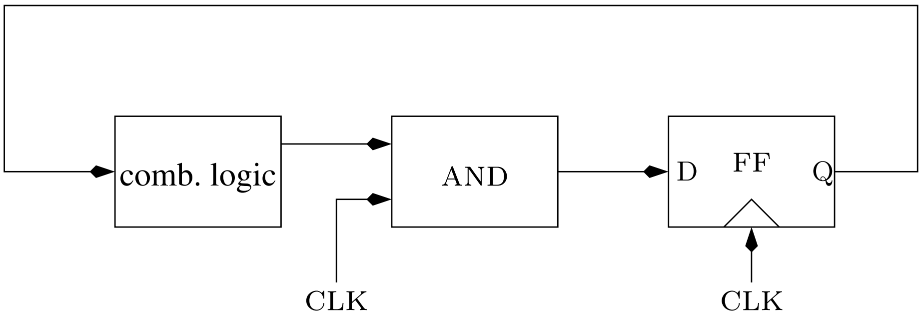
remarks on the definition of synchronous circuits

- CLK connected to all the clock-ports of flip-flops and only to them.
- We already saw that a "bad example" in which CLK feeds a gate:

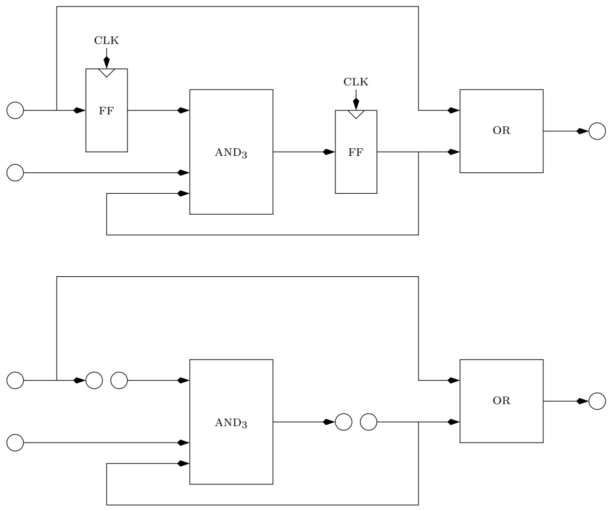


remarks on the definition of synchronous circuits

Question: What is required so that the D-port is stable during the critical segment in this "bad example":



back to the first example



Question: Is this a synchronous circuit?

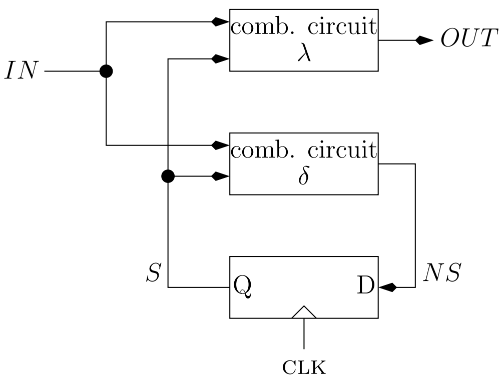
Recognizing a synchronous circuit

Question: Suggest an efficient algorithm that decides if a given circuit is synchronous.

A synchronous circuit is a circuit C composed of combinational gates, nets, and flip-flops that satisfies the following conditions:

1. There is a net called CLK that carries a clock signal.
2. The CLK net is fed by an input gate.
3. The set of ports that are fed by the CLK net equals the set of clock-inputs of the flip-flops.
4. The circuit C' obtained from C by stripping away flip-flops is combinational.

Synchronous Circuits: canonic form

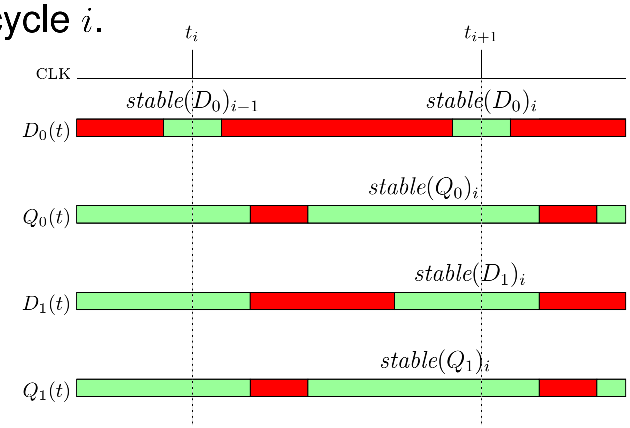


Transform a synchronous to canonic form:

- gather the flip-flops into one group.
- duplicate the combinational circuits to separate between output and next-state.

Stability Interval

- stability interval of signal X - interval during which X is stable.
- *stable*$(X)_i$ - stability interval of X corresponding to clock cycle i.



Timing analysis: the canonic form

Plan:

- Define timing constraints for IN and OUT.
- Define timing constraints for S and NS.
- Find sufficient conditions so that timing constraints are feasible.
- Define minimum clock period.
- Infer functionality from syntax.

Input/output timing constraints

- The input/output timing constraints formulate the timing interface between the the circuit and the “external world”.
- Input timing constraint - tells us when the input is guaranteed to be stable.
- Output timing constraint - tells us when the circuit’s output is required to be stable.
- Usually the external world is also a synchronous circuit. $\Rightarrow IN$ is an output of another synchronous circuit, and OUT is an input of another synchronous circuit.

Input timing constraint

The timing constraint corresponding to IN is defined by two parameters: *pd*$(IN) >$ *cont*(IN) as follows.

$$\forall i : \quad [t_i + \textit{pd}(IN), t_{i+1} + \textit{cont}(IN)] \subseteq \textit{stable}(IN)_i.$$

Remarks:

- t_i - denotes the starting time of the ith clock period.
- Why do we require that *pd*$(IN) >$ *cont*(IN)?
  If *pd*$(IN) \leq$ *cont*(IN), then the stability intervals *stable*$(IN)_i$ and *stable*$(IN)_{i+1}$ overlap. This means that IN is always stable, which is obviously not an interesting case.

Output timing constraint

The timing constraint corresponding to OUT is defined by two parameters: *setup*(OUT) and *hold*(OUT) as follows.

$$\forall i : \quad [t_{i+1} - \textit{setup}(OUT), t_{i+1} + \textit{hold}(OUT)] \subseteq \textit{stable}(OUT)_i.$$

Remark: Note that that timing constraint of OUT is given relative to the end of the ith cycle (i.e. t_{i+1}) .

Remarks

- asymmetry in the terminology regarding IN and OUT. The parameters associated with IN are *pd*(IN) and *cont*(IN), whereas the parameters associated with OUT are *setup*(OUT) and *hold*(OUT).
- this is not very aesthetic if OUT is itself an input to another synchronous circuit.
- useful to regard IN as an output of a flip-flip and OUT as an input of a flip-flop (even if they are not).

Timing constraint of NS

NS is stable during the critical segments. Namely,

$$\forall i \geq 0: \quad C_{i+1} \subseteq \textit{stable}(NS)_i.$$

Remark: Note that, as in the case of the output signal, the timing constraint of NS corresponding to clock cycle i is relative to the end of the ith clock cycle (i.e. the critical segment C_{i+1}).

Remark: If NS satisfies its timing constraint for i , then S satisfies:

$$[t_{i+1} + t_{\textit{pd}}, t_{i+2} + t_{\textit{cont}}] \subseteq \textit{stable}(S)_{i+1}.$$

Stability Intervals of OUT & NS

- We associate a contamination delay $\mathit{cont}(x)$ and a propagation delay $\mathit{pd}(x)$ with each combinational circuit x.
- If $[t_i + t_{\mathit{pd}}, t_{i+1} + t_{\mathit{cont}}] \subseteq \mathit{stable}(S)_i$, then the stability intervals of the signals OUT and NS satisfy:

$$[t_i+\max\{t_{\mathit{pd}}, \mathit{pd}(IN)\}+pd(\lambda), t_{i+1}+\min\{t_{\mathit{cont}}, \mathit{cont}(IN)\}+\mathit{cont}(\lambda)] \subseteq \mathit{stable}(OUT)_i$$

$$[t_i+\max\{t_{\mathit{pd}}, \mathit{pd}(IN)\}+pd(\delta), t_{i+1}+\min\{t_{\mathit{cont}}, \mathit{cont}(IN)\}+\mathit{cont}(\delta)] \subseteq \mathit{stable}(NS)_i.$$

Sufficient conditions: OUT

Claim: If

$$[t_i + t_{\mathit{pd}}, t_{i+1} + t_{\mathit{cont}}] \subseteq \mathit{stable}(S)_i$$
$$\max\{t_{\mathit{pd}}, \mathit{pd}(IN)\} + pd(\lambda) + \mathit{setup}(OUT) \leq t_{i+1} - t_i$$
$$\min\{t_{\mathit{cont}}, \mathit{cont}(IN)\} + \mathit{cont}(\lambda) \geq \mathit{hold}(OUT),$$

then

$$[t_{i+1} - \mathit{setup}(OUT), t_{i+1} + \mathit{hold}(OUT)] \subseteq \mathit{stable}(OUT)_i.$$

Proof: stability interval of OUT satisfies:

$$[t_i+\max\{t_{\mathit{pd}}, \mathit{pd}(IN)\}+pd(\lambda), t_{i+1}+\min\{t_{\mathit{cont}}, \mathit{cont}(IN)\}+\mathit{cont}(\lambda)] \subseteq \mathit{stable}(OUT)_i$$

$\square$

Sufficient conditions: NS

Claim: If

$$[t_i + t_{\textit{pd}}, t_{i+1} + t_{\textit{cont}}] \subseteq \textit{stable}(S)_i$$

$$\max\{t_{\textit{pd}}, \textit{pd}(IN)\} + \textit{pd}(\delta) + t_{\textit{su}} \leq t_{i+1} - t_i$$

$$t_{\textit{hold}} \leq \min\{t_{\textit{cont}}, \textit{cont}(IN)\} + \textit{cont}(\delta),$$

then the signal NS is stable during the critical segment C_{i+1}.

Proof: stability interval of NS satisfies:

$$[t_i+\max\{t_{\textit{pd}}, \textit{pd}(IN)\}+pd(\delta), t_{i+1}+\min\{t_{\textit{cont}}, \textit{cont}(IN)\}+\textit{cont}(\delta)] \subseteq \textit{stable}(NS)_i.$$

$\square$

Timing constraints for $i \geq 0$

CORO: If 4 conditions hold and

$$[t_0 + t_{\textit{pd}}, t_1 + t_{\textit{cont}}] \subseteq \textit{stable}(S)_0,$$

then

1. timing constraints of NS and OUT hold wrt every $i \geq 0$,
2. $\forall i \geq 0 \;:\; [t_i + t_{\textit{pd}}, t_{i+1} + t_{\textit{cont}}] \subseteq \textit{stable}(S)_i$.

Proof: Induction on i.

- Basis: part (1) follows from sufficient conditions for OUT and NS.
- Step: NS is stable during $C_{i+1} \Rightarrow$ part (2).
- $\Rightarrow$ part(1).

$\square$

Simplifying the conditions

- Our goal is to simplify the conditions in the 2 Claims.
- Prefer: lower bounds on the clock period.
- $\Rightarrow$ well defined functionality provided that the clock period is large enough.
- We discuss each of the 4 conditions (2 per claim).

$$\max\{t_{\boldsymbol{pd}}, \boldsymbol{pd}(IN)\} + pd(\lambda) + \boldsymbol{setup}(OUT) \leq t_{i+1} - t_i$$

- condition is a lower bound on $\varphi(\text{CLK})$. Great.

$$\min\{t_{\boldsymbol{cont}}, \boldsymbol{cont}(IN)\} + \boldsymbol{cont}(\lambda) \geq \boldsymbol{hold}(OUT)$$

- condition may not hold $\Rightarrow$ serious problem that can lead to failure to meet the timing constraint of OUT...
- Hope: under reasonable circumstances, condition does hold. Why?
  - Suppose IN is the output of a combinational circuit, all the inputs of which are outputs of flip-flops.
  - Assume that all the flip-flops are identical.
  - It follows that *cont*$(IN) \geq t_{cont}$.
  - By definition: *cont*$(\lambda) \geq 0$.
  - $\Rightarrow \min\{t_{cont}, \mathit{cont}(IN)\} + \mathit{cont}(\lambda) \geq t_{cont}$.
  - Suppose OUT feeds a combinational circuit that feeds a flip-flop.
  - Hence *hold*$(OUT) \leq t_{hold}$.
  - $t_{hold} < t_{cont} \Rightarrow$ condition holds.

$$\max\{t_{\boldsymbol{pd}}, \boldsymbol{pd}(IN)\} + \boldsymbol{pd}(\delta) + t_{\boldsymbol{su}} \leq t_{i+1} - t_i$$

- condition is a lower bound on $\varphi(\textsc{clk})$. Great.

$t_{\textit{hold}} \le \min\{t_{\textit{cont}}, \textit{cont}(IN)\} + \textit{cont}(\delta)$

- As before, if *cont*$(IN) \ge t_{\textit{cont}}$, the condition holds!

Conclusion

Claim: Assume that *cont*$(IN) \ge t_{\textit{cont}}$ and *hold*$(OUT) \le t_{\textit{hold}}$.
If

$$[t_0 + t_{\textit{pd}}, t_1 + t_{\textit{cont}}] \subseteq \textit{stable}(S)_0,$$

$$\varphi(\text{CLK}) \ge \max\{t_{\textit{pd}}, \textit{pd}(IN)\} + \max\{\textit{pd}(\lambda) + \textit{setup}(OUT), \textit{pd}(\delta) + t_{\textit{su}}\},$$

then

1. timing constraints of NS and OUT hold wrt every $i \ge 0$,
2. $\forall i \ge 0 \; : \; [t_i + t_{\textit{pd}}, t_{i+1} + t_{\textit{cont}}] \subseteq \textit{stable}(S)_i$.

Under reasonable assumptions, all we need is initialization and a sufficiently long clock period.

Minimum clock period

DEF: The minimum clock period of a synchronous circuit C is the shortest clock period for which the timing constraints of the output signals and signals that feed the flip-flops are satisfied.
We denote the minimum clock period of a synchronous circuit by $\varphi^*(C)$.

- Minimum clock period does not exist if timing constraints are infeasible.
- "timing constraints are satisfied" - for every value of the delays provided that they are in their range. (i.e. actual propagation delay of λ is in $[0, pd(\lambda)]$.)
- if assumptions hold, then in canonic form

$$\varphi^*(C) = \max\{t_{pd}, pd(IN)\} + \max\{pd(\lambda), pd(\delta)\} + t_{su}.$$

Discussion

- The timing analysis of synchronous circuits in canonic form is overly pessimistic.
- The problem is that each of the combinational circuits λ and δ is regarded as a "gate" with a propagation delay.
- In practice it may be the case, for example, that the accumulated delay from the input IN to the output OUT is significantly different than the accumulated delay from S to the output OUT. The situation is even somewhat more complicated in the case of multi-bit signals.
- We now deal with the general case.

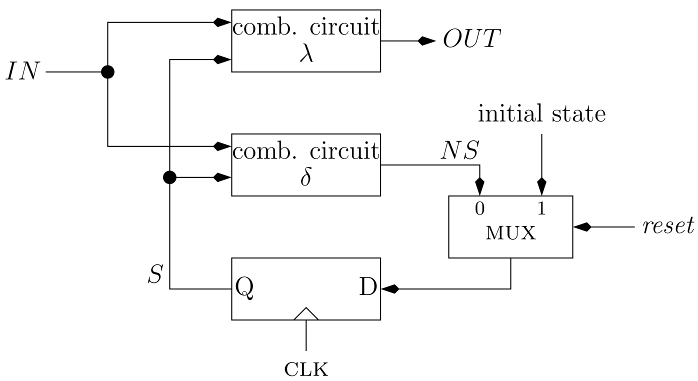
Initialization

We require that

$$[t_0 + t_{\mathit{pd}}, t_1 + t_{\mathit{cont}}] \subseteq \mathit{stable}(S)_0.$$

- after power-up, flip-flop output may be non-logical (and even meta-stable).
- solution: introduce a reset signal.
- boot-strapping problem: How is a reset signal generated?
- no solution to this problem within the digital abstraction (meta-stability). All we can try to do is reduce the probability of such an event.
- reset controller - a special circuit that generates a reset signal.

Synchronous Circuit: canonic form with reset



Remark: NS may not be logical during reset.
Implementation of MUX must output initial-state if *reset* $= 1$.
Implementation based on drivers has this property, while implementation based on combinational gates may not have this property.

Functionality of Synchronous Circuits: canonic form

- X_i - *dig*(X) during *stable*$(X)_i$.
- Assumptions:

$$\begin{aligned} \textit{cont}(IN) &\geq t_{\textit{cont}} \\ \textit{hold}(OUT) &\leq t_{\textit{hold}} \\ [t_0 + t_{\textit{pd}}, t_1 + t_{\textit{cont}}] &\subseteq \textit{stable}(S)_0, \\ \varphi(\text{CLK}) &\geq \max\{t_{\textit{pd}}, \textit{pd}(IN)\} + \max\{\textit{pd}(\lambda), \textit{pd}(\delta)\} + t_{\textit{su}}, \end{aligned}$$

CORO: Assumptions $\Rightarrow \quad \forall i \geq 0$:

$$\begin{aligned} NS_i &= \delta(IN_i, S_i) \\ OUT_i &= \lambda(IN_i, S_i) \\ S_{i+1} &= NS_i. \end{aligned}$$

Finite State Machines

Corollary states that synchronous circuits implement finite state machines.

DEF: 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.
- $\lambda : Q \times \Sigma \to Q$ is an output function.
- $q_0 \in Q$ is an initial state.

Definition of FSM: remarks

- Other terms for a finite state machine are a finite automaton with outputs, transducer, and Mealy Machine.
- Moore Machine - an FSM in which the output function $\lambda : Q \to \Delta$.

What does an FSM do?

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

$$q_{i+1} \triangleq \delta(q_i, x_i)$$

- The output y_i is defined as follows:

$$y_i \triangleq \lambda(q_i, x_i).$$

State Diagrams

FSMs are often depicted using state diagrams.

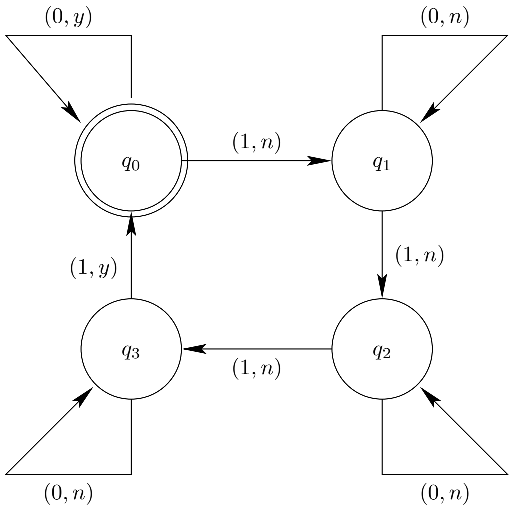
DEF: The state diagram corresponding to an FSM $\mathcal{A}$ is a directed graph $G = (V, E)$ with edge labels $(x, y) \in \Sigma \times \Delta$. The vertex set V equals the state set S. The edge set E is defined by

$$E \triangleq \{(q, \delta(q, x)) : q \in Q \text{ and } x \in \Sigma\}.$$

An edge $(q, \delta(q, x))$ is labeled $(x, \lambda(q, x))$.

State Diagram: example

A state diagram of an FSM that outputs y if the weight of the input so far is divisible by 4, and n otherwise.



Timing analysis: the general case

- Deal with a synchronous circuit that is not in canonic form.
- Algorithm that computes the minimum clock period $\varphi^*(C)$. (if timing constraints are feasible.)
- Algorithm that decides whether the timing constraints are feasible (i.e. conditions used by this algorithm are less restrictive than the conditions used in previous claims).

Review of timing constraints

Input constraints: For every input signal IN, guaranteed:

$$[t_i + \textit{pd}(IN), t_{i+1} + \textit{cont}(IN)] \subseteq \textit{stable}(IN)_i.$$

Output constraints: For every output signal OUT, require:

$$[t_{i+1} - \textit{setup}(OUT), t_{i+1} + \textit{hold}(OUT)] \subseteq \textit{stable}(OUT)_i.$$

Critical segments: For every signal NS that feeds a D-port of a flip-flop, require:

$$C_{i+1} \subseteq \textit{stable}(NS)_i.$$

Algorithm: minimum clock period

- $C' \leftarrow$ combinational circuit obtained by stripping away flip-flops from C.
- For every gate v of C' define $d(v)$ as follows:

$$d(v) \triangleq \begin{cases} \textit{pd}(IN) & \text{if } v \text{ feeds input signal } IN. \\ t_{\textit{pd}} & \text{if } v \text{ corresponds to a } Q\text{-port.} \\ \textit{setup}(OUT) & \text{if } v \text{ is fed by } OUT. \\ t_{\textit{su}} & \text{if } v \text{ corresponds to a } D\text{-port.} \\ \textit{pd}(v) & \text{if } v \text{ is a combinational gate of } C. \end{cases}$$

- Let $DG(C')$ denote the directed acyclic graph (DAG) that corresponds to C'. Let p' denote the longest path in $DG(C')$ with respect to the delays $d(v)$. Return $d(p')$.

Algorithm: correctness

define delays $c(v)$ to non-sink vertices in $DG(C')$ as follows.

$$c(v) \triangleq \begin{cases} \textit{cont}(IN) & \text{if } v \text{ feeds an input signal } IN. \\ t_{\textit{cont}} & \text{if } v \text{ corresponds to a } Q\text{-port of a flip-flop.} \\ \textit{cont}(v) & \text{if } v \text{ is a combinational gate in } C. \end{cases}$$

Lemma: Consider a combinational gate, an input gate, or a flip-flop v in the synchronous circuit C. Let $\mathcal{P}_v$ denote the set of all directed paths in the directed acyclic graph $DG(C')$ that begin at a source and end in v. If the output of every flip-flop is stable in the interval $[t_i + t_{\textit{pd}}, t_{i+1} + t_{\textit{cont}}]$, then every output N of v satisfies

$$[t_i + \max_{p \in \mathcal{P}_v} d(p), t_{i+1} + \min_{p \in \mathcal{P}_v} c(p)] \subseteq \textit{stable}(N)_i.$$

proof: $[t_i + \max_{p\in\mathcal{P}_v} d(p), t_{i+1} + \min_{p\in\mathcal{P}_v} c(p)] \subseteq$ ***stable***$(N)_i$.

- $\{v_0, \ldots, v_{n-1}\}$ - topological sort of vertices of $DG(C')$.
- Let $v = v_j$. Proof by induction on j.
- Basis: two cases: (i) If v is an input gate, then input constraint. (ii) If v is a flip-flop, then assumption on the output of flip-flops.
- Step: same as basis if v is an input gate or a flip flop.
- Assume v is a combinational gate.
- Ind. Hyp. : every input N' of v_{j+1} satisfies equation.
- $\Rightarrow$ every output N of v_{j+1} satisfies: (i) N becomes stable at most $d(v_{j+1})$ time units after its last input becomes stable, and (ii) N remains stable at least $c(v_{j+1})$ time units after its first input becomes instable.

Algorithm: correctness (cont.)

Claim: Suppose that: (i) for every signal fed by a Q-port of a flip-flop, $[t_i + t_{pd}, t_{i+1} + t_{cont}] \subseteq$ *stable*$(S)_i$, (ii) for every input IN, *cont*$(IN) \geq t_{cont}$, and (iii) for every output OUT, *hold*$(OUT) \leq t_{hold}$. Then,

1. For every clock period $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$, the signals feeding D-ports of flip-flops are are stable during the critical segment C_{i+1}.
2. For every clock period $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$, the output timing constraints corresponding to cycle i are satisfied.
3. For every clock period $\varphi(\text{CLK}) < \varphi^*(\text{CLK})$, a violation of the timing constraints is possible.

Proof: $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$ satisfies timing constraints

Let N denote signal that feeds a D-port of a flip-flop v that is fed by u. By Lemma, N is stable during the interval

$$[t_i + \max_{p \in \mathcal{P}_u} d(p), t_{i+1} + \min_{p \in \mathcal{P}_u} c(p)].$$

Since $\varphi(\text{CLK}) \geq \max_{p \in \mathcal{P}_v} d(p) = d(v) + \max_{p \in \mathcal{P}_u} d(p)$ and $d(v) = t_{su}$, we conclude that

$$t_{i+1} - t_i = \varphi(\text{CLK}) \geq t_{su} + \max_{p \in \mathcal{P}_u} d(p).$$

$\Rightarrow$ signal N stable starting at

$$t_i + \max_{p \in \mathcal{P}_u} d(p) \leq t_{i+1} - t_{su}.$$

$\Rightarrow$ setup-time constraint is satisfied.

Proof: $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$ - cont.

N is stable until $t_{i+1} + \min_{p \in \mathcal{P}_u} c(p)$.
However, every path $p \in \mathcal{P}_u$ begins at a source. A source may correspond to an input gate in C or a Q-port of a flip flop. Since *cont*$(IN) \geq t_{cont}$, we conclude that $c(s) \geq t_{cont}$, for every source s.
It follows that

$$\min_{p \in \mathcal{P}_u} c(p) \geq t_{cont} > t_{hold}.$$

Lemma: N is stable until $t_{i+1} + \min_{p \in \mathcal{P}_u} c(p) \geq t_{i+1} + t_{hold}$.
$\Rightarrow$ hold-time constraint is satisfied.

$\Rightarrow$ N is stable during the critical segment C_{i+1}, as required.

Proof: $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$ - cont.

Proof for an output signal OUT is similar.

Proof: $\varphi(\text{CLK}) < \varphi^*(\text{CLK}) \Rightarrow$ violation

- p - longest path in $DG(C')$ with respect to lengths $d(v)$. (p begins at a source and ends in a sink v.)
- Let p' denote the path obtained from p by omitting the sink v. It follows that

$$t_i + d(p') > t_{i+1} - d(v).$$

- If the actual propagation delays along p are maximal, then the signal feeding v is not stable at time $t_{i+1} - d(v)$.
- If v is a flip-flop, then its input is not stable during the critical segment.
- If v is an output gate, then its input does not meet the output constraint. The claim follows.

Corollary

If the circuit is properly initialized, then the clock period computed by the algorithm is the shortest clock period that satisfies all the timing constraints for all clock cycles i, for $i \geq 0$.
Formally...

Corollary - formal

CORO: Suppose that: (i) for every signal S fed by a Q-port of a flip-flop, $[t_0 + t_{pd}, t_1 + t_{cont}] \subseteq$ *stable*$(S)_0$, (ii) for every input IN, *cont*$(IN) \geq t_{cont}$, and (iii) for every output OUT, *hold*$(OUT) \leq t_{hold}$. Then,

1. For every clock period $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$, the signals feeding D-ports of flip-flops are are stable during every critical segment C_{i+1}, for $i \geq 0$.
2. For every clock period $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$, the output timing constraints corresponding to cycle i are satisfied, for every $i \geq 0$.
3. For every clock period $\varphi(\text{CLK}) < \varphi^*(\text{CLK})$, a violation of the timing constraints is possible.

Proof: Proof is by induction on the clock cycle i. □

Algorithm: feasibility of timing constraints

- So far, reasonable assumptions are made so that it is guaranteed that a minimum clock period exists.
- It is possible that these assumptions do not hold although the timing constraints are feasible.
- We now present an algorithm that verifies whether the timing constraints are feasible without relying an any assumptions.

Follow recipe of the lemma

Lemma states that, for every non-sink v in C', the guaranteed stability interval of the signals that are output by v is:

$$[t_i + \max_{p \in \mathcal{P}_v} d(p), t_{i+1} + \min_{p \in \mathcal{P}_v} c(p)].$$

The $\varphi^*(C)$ algorithm deals with making sure that each such interval does not start too late (i.e. satisfy setup-time constraint).

Feasibility means checking that stability intervals do not end too early (i.e. satisfy hold-time constraint).

Follow recipe of the lemma - cont

Recall: signal fed by v is stable during

$$[t_i + \max_{p \in \mathcal{P}_v} d(p), t_{i+1} + \min_{p \in \mathcal{P}_v} c(p)].$$

Check that

1. For every u that feeds a D-port of a flip-flop, require

$$\min_{p \in \mathcal{P}_u} c(p) \geq t_{\textit{hold}}.$$

2. For every u that feeds an output signal OUT, require

$$\min_{p \in \mathcal{P}_u} c(p) \geq \textit{hold}(OUT).$$

violation $\Rightarrow$ timing constraints are infeasible.
no violation $\Rightarrow$ timing constraints are feasible.

Algorithmic Aspect

- All we need to check if timing constraints are feasible is to compute

$$\forall \text{ non-sink } v \ : \ \min_{p \in \mathcal{P}_v} c(p).$$

- Compute shortest path in a DAG (can be done in linear time using depth first search).
- After these values are computed for all the non-sinks, the algorithm simply checks hold-time constraints for every D-port and for every output.
- If a violation is found, then the timing constraints are infeasible.

Recap

- We started with a syntactic definition of a synchronous circuit.
- We then attached timing constraints to the inputs and outputs of synchronous circuit.
- For a given synchronous circuit C with input/output timing constraints, we differentiate between two cases:
  - timing constraints are infeasible $\Rightarrow$ cannot guarantee well defined functionality of C. For example, if the timing constraints are not met, then inputs of flip-flops might not be stable during the critical segments, and then the flip-flop output is not guaranteed to be even logical.
  - timing constraints are feasible $\Rightarrow$ functionality is well defined provided that the clock period satisfies $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$.

Functionality

- Assume that the timing constraints are feasible.
- Introduce a trivial timing model called the zero delay model.
- In this model, time is discrete and in each clock cycle, the circuit is reduced to a combinational circuit.
- Advantage: decouple timing issues from functionality and enables simple logical simulations.

The zero delay model

- In the zero delay model we assume that all the parameters of all the components are zero (i.e. $t_{su} = t_{hold} = t_{cont} = t_{pd} = 0$, $pd(IN) = cont(IN) = setup(OUT) = hold(OUT) = 0$, and $d(G) = 0$, for every combinational gate G). Under this unrealistic assumption, the timing constraints are feasible.
- By Lemma, it follows that, in the zero delay model, the stability interval of every signal is $[t_i, t_{i+1})$.
- Following Corollary (synchronous circuit implements an FSM), we conclude that, for every signal X, X_i is well defined.

Simulation of a synchronous circuit

Simulation during cycles $i = 0, \ldots, n-1$ in the zero propagation model proceeds as follows:
assume: flip-flops are initialized ($\vec{S}_0$ - initial values of FFs).

1. Construct comb. circuit C' that corresponds to C.
2. For $i = 0$ to $n-1$ do:
   (a) Simulate C' with input values $\vec{S}_i$ and $I\vec{N}_i$.
   (b) For every output OUT^j, let y denote the value that is fed to y. We set $OUT_i^j = y$.
   (c) For every D-port NS^j of a flip-flop, let y denote the value that is fed to the flip-flop. We set $NS_i^j = y$.
   (d) For every Q-port S^j of a flip-flop, define $S_{i+1}^j \leftarrow NS_i^j$, where NS^j denotes the signal that feeds the D-port of the flip-flop.

Summary

- define synchronous circuits.
- canonic form of synchronous circuits:
  - definition of timing constraints.
  - formulation of sufficient conditions for satisfying the timing constraints.
  - simplify sufficient conditions by relying on the assumption that the input originates from a flip-flop and the output is eventually fed to a flip-flop.
  - define the minimum clock period.
  - initialization.
  - synchronous circuit implement FSMs.

Summary -cont.

- general case of synchronous circuits (not in canonic form).
  - algorithm: min. clock period.
  - algorithm: feasibility of timing constraints.
- Functionality:
  - zero delay model.
  - simulation.

Chapter 11: Additional Components

Computer Structure
&
Intro. to Digital Computers

Dr. Guy Even

Tel-Aviv Univ.

Goals

- define drivers (3-state buffers).
- define Read Only Memory (ROM).
- define Random Access Memory (RAM).

Drivers

DEF: A driver is a device defined as follows:

Input: IN, *enable* $\in \{0, 1\}$.

Output: $OUT \in \{0, 1\}$.

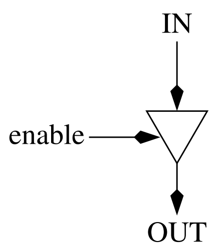
Functionality:

$$OUT = \begin{cases} IN & \text{if } \textit{enable} = 1 \\ \text{not-connected} & \text{otherwise.} \end{cases}$$

Remark:

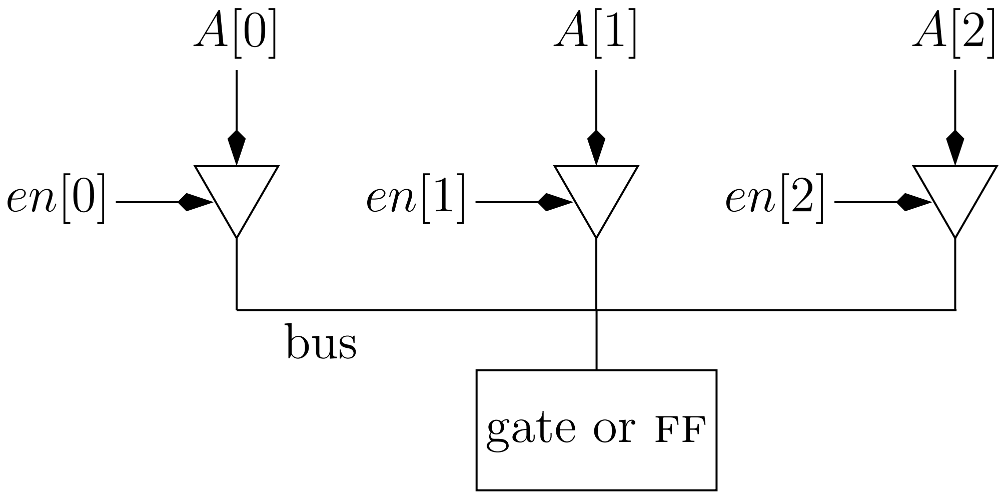
- A driver is not a combinational circuit.
- A driver is in one of two "states":
  - connected - a "wire" between IN and OUT.
  - disconnected - value of IN does not affect value of OUT.

Drivers - cont.

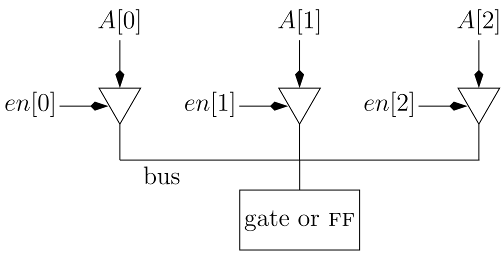


Remark: technology dependent: when connected either uni-directional or bi-directional. (can always force uni-directional by connecting two inverters in series to OUT.)

Bus fed by drivers



Bus fed by drivers - cont



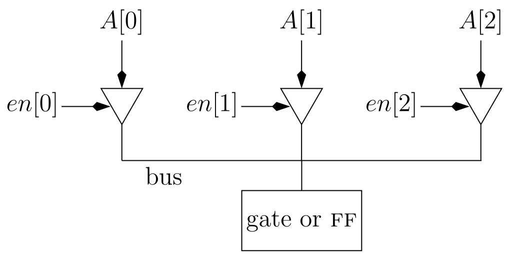
Design rule: If net N is fed by a driver, then every component feeding N must be a driver.

Semantic rule: stable values of $en[0], \ldots, en[k-1]$ satisfy:

$$\sum_{i=0}^{k-1} en[i] = 1.$$

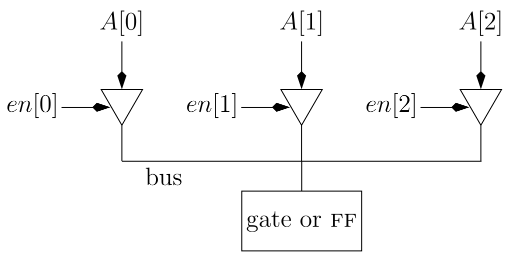
Why use driver fed buses?

- Could use $k : 1$-MUX instead.
- Driver fed bus is simpler in a “distributed” situation:
  - Signals $A[i]$ are computes in different locations, single wire connecting these locations carries the selected signal.
  - Still need to connect separate wires for $en[i]$.



Disadvantage of driver fed buses?

- If sematic rule is violated $\sum_{i=0}^{k-1} en[i] > 1$, then conflicting drivers can damage the circuit.
- $\Rightarrow$ no drivers in lab work.



Read Only Memory (ROM)

two parameters:

- a - $A = 2^a$ number of memory “words” .
- d - number bits per memory word.

DEF: A ROM(a, d) is a combinational circuit defined as follows:

Input: *Address* $\in \{0, 1\}^a$.

Output: $D \in \{0, 1\}^d$.

Functionality:

$$D = M[\langle \textit{Address} \rangle],$$

$M[0..(A-1)]$ is a vector of binary strings $\in \{0, 1\}^d$.

Remark: ROM(a, d) stores the vector $M[0..(A-1)]$, and outputs $M[\langle \textit{Address} \rangle]$.

ROM

- A ROM(a, d) implements a boolean function

$$f : \{0, 1\}^a \rightarrow \{0, 1\}^d.$$

- ROM(a, d) is loaded with the contents of vector $M[0..(A-1)]$ during fabrication.
- Other technologies enable loading of ROM after fabrication or even on-board (EPROM,EEPROM, flash memory).

Random Access Memory (RAM)

- Abstract: stores a vector $M[0..A-1]$ of words.
- Allows reading the contents of vector entry:

$$D_{\text{out}} \leftarrow M[\langle \textit{Address} \rangle].$$

- Allows writing contents into a vector entry:

$$M[\langle \textit{Address} \rangle] \leftarrow D_{\text{in}}.$$

- Complicated devices (not combinational and not synchronous) - designed to be as efficient as possible.
- Communication with external world according to a bus protocol.
- S-RAM slides

RAM as main memory of a processor

- Communication between processor and RAM is done via dedicated registers (MAR, MDR) and by a bus.
- One often puts controllers (FSMs) that controll the bus signals and inform processor when memory access is completed.
- Bus protocol and controller - topic of lab work next semester.