Chapter 1: The digital abstraction

Computer Structure - Spring 2007

©Dr. Guy Even

Tel-Aviv Univ.

Preliminary questions

- Can you justify or explain the saying that “computers use only zeros and ones”?
- Can you explain the following anomaly? The design of an adder is a simple task. However, the design and analysis of a single electronic device (e.g., a single gate) is a complex task.

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.

More 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)?

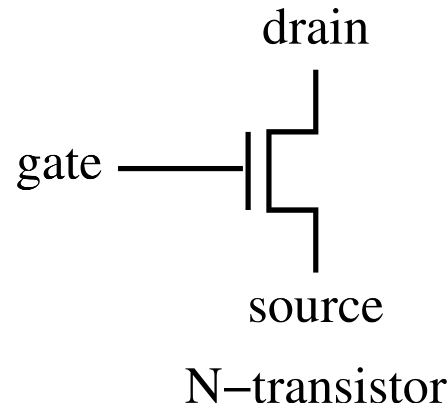
Transistors

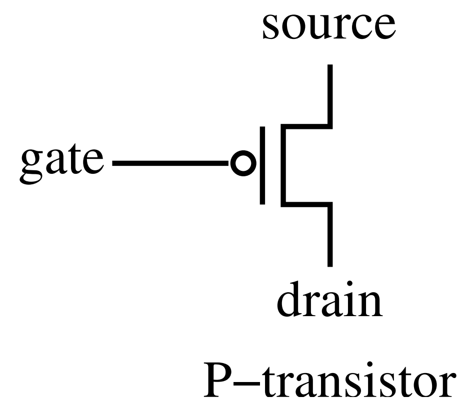
Computers $\Leftarrow$ VLSI chips $\Leftarrow$ gates & flip-flops $\Leftarrow$ transistors
Transistors are the basic components.
Most common VLSI technology is called CMOS.
In CMOS: only two types of transistors:

- N-transistor
- P-transistor

in case you are curious:
VLSI = Very Large Scale Integration (which means "millions of transistors placed on one small chip")
CMOS = Complementary Metal Oxide Semiconductor (which means that both NMOS and PMOS transistors are used).

N-transistor & P-transistor

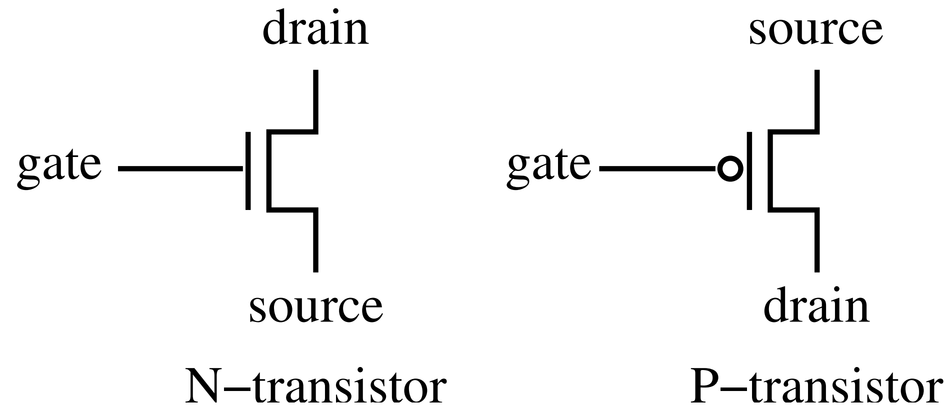




Inputs: gate & source
Output: drain

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

N-transistor & P-transistor



Functionality of N-transistor:

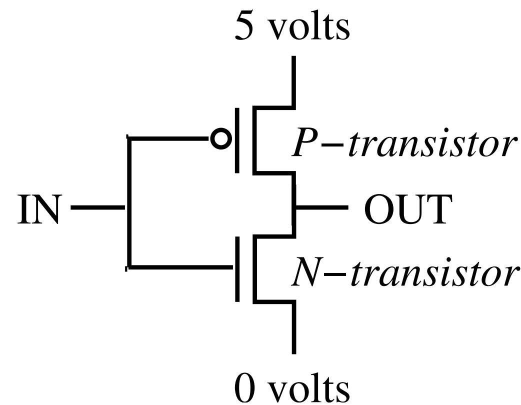
- If $v(gate) = high$, then $resistance(source, drain) = 0$ (and then $v(drain) \leftarrow v(source)$)
- If $v(gate) = low$, then $resistance(source, drain) = \infty$

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.

Example: 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

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

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 neither zero or one.

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 the threshold V' lead to unexpected values of *dig*$(f(t))$.
  Example: Measure weight w by measuring the length ℓ of a spring. Suppose we wish to know if $w > w'$. This can be done by checking if $\ell > \ell'$. However, spring length oscillates around ℓ. If $\ell \approx \ell'$, then comparison requires a long time.

$\Rightarrow$ must use 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?

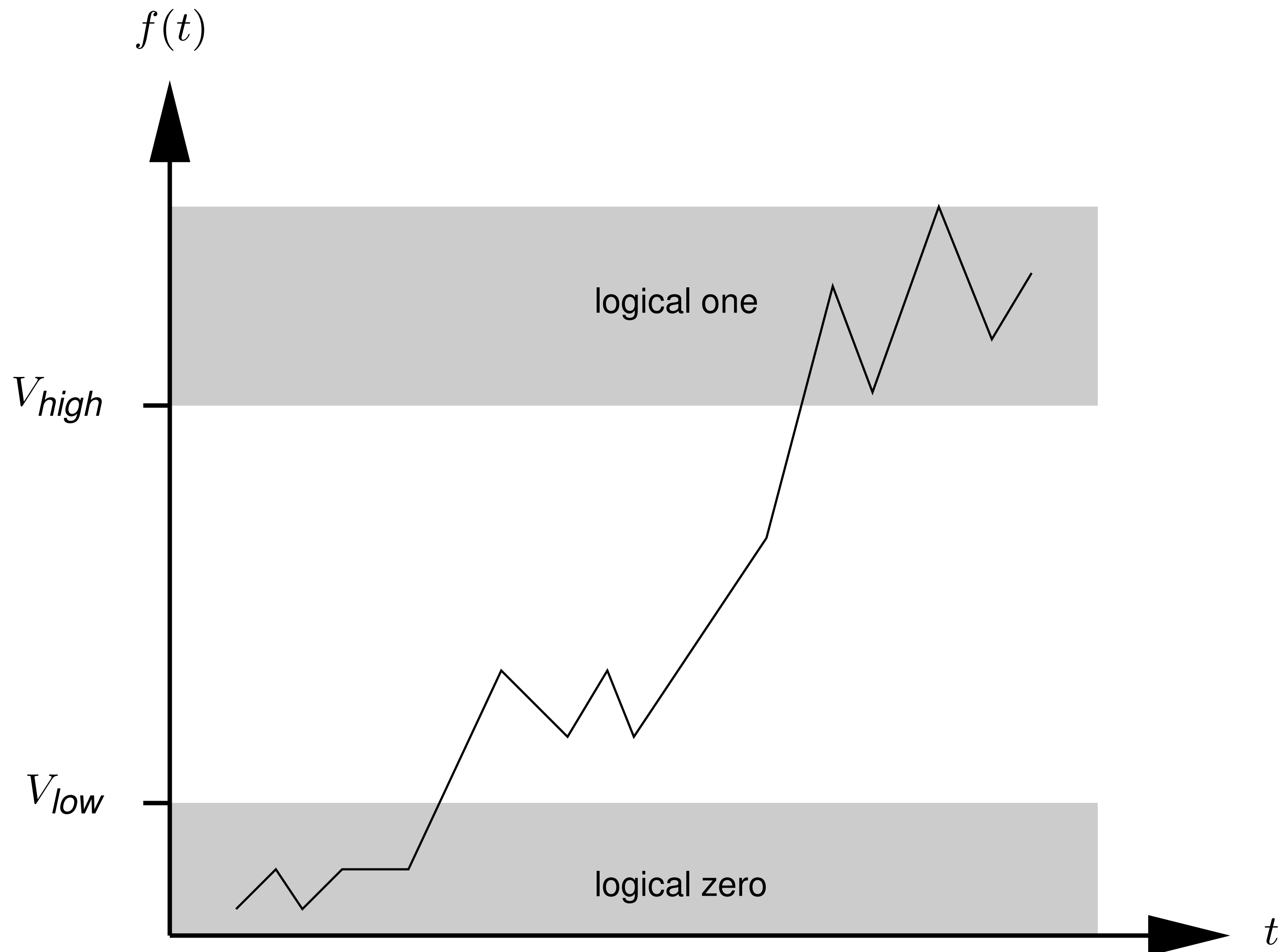
A: Two voltage thresholds are defined: $V_{low} < V_{high}$.

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:

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

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

Before we discuss these issues, we must introduce transfer functions and noise...

Transfer functions

DEF: transfer function - the relation between the voltage at an output of a gate and the voltages of the inputs of the gate.

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.

history vs. present: if a device does not have a static transfer function, then the device is a memory device not a logical gate.

Static transfer function

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

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

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

- 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 δ-stable.
- $[t_1 + \Delta, t_2]$ - interval during which $y(t)$ is ϵ-stable.

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

back to the definition of an inverter

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

or equivalently,

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

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

Q: Define a NAND-gate.

Noise

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.

Bounded noise model

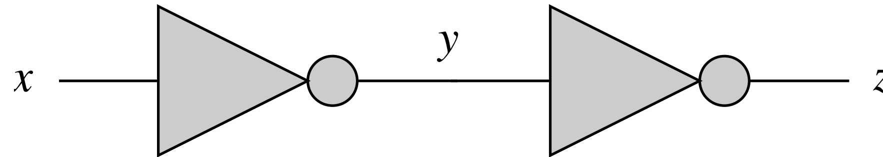
- Bounded noise model - the noise signal along every wire has a bounded absolute value.
- Uniform bounded noise model:

$$\exists \epsilon > 0 \text{ such that : } |\ \textit{noise}\ | \leq \epsilon.$$

- Justification - noise is a random variable whose distribution has a rapidly diminishing tail. If the ϵ is sufficiently large, then

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

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*$(y(t) + n_y(t)) =$ 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.)

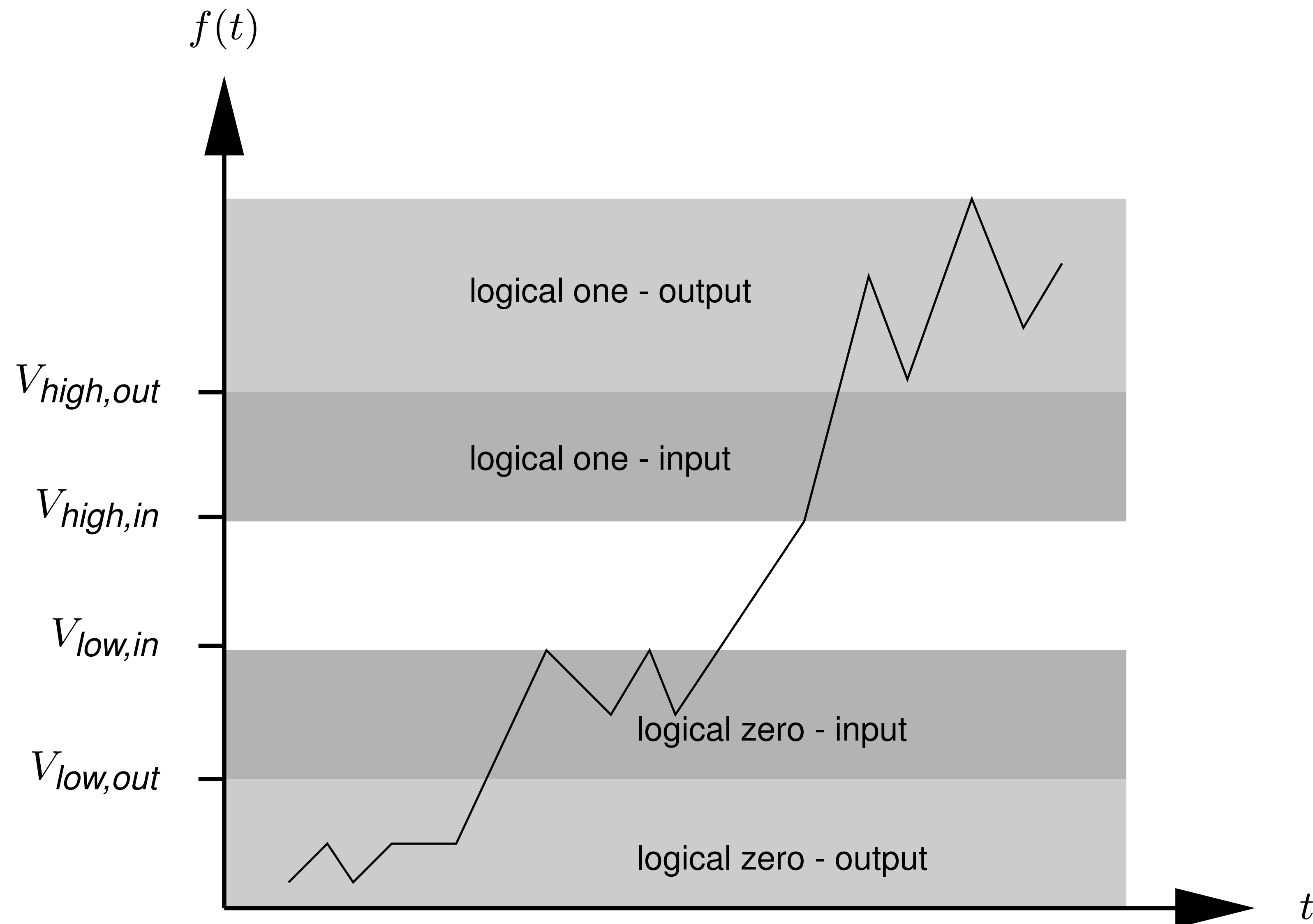
Instead of two thresholds, V_{low} and V_{high}, we define the following four thresholds:

- $V_{low,in}$ - 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.

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

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

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

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: Suppose that a wire transmits an output signal $f_{out}(t)$ to an input signal $f_{in}(t)$. Suppose that $|n(t)|$ is less than the noise margin. If *dig*$(f_{out})(t) \in \{0, 1\}$, then *dig*$(f_{in}(t)) =$ *dig*$(f_{out}(t))$.
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 $f(t)$ & $V_{low} < V_{high}$. □

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 circuits

Computer Structure - Spring 2007

©Dr. Guy Even

Tel-Aviv Univ.

Preliminary Questions

- Does every collection of gates and wires constitute a combinational circuit?
- Which of these tasks is easy?
  - Check if a circuit is combinational.
  - Simulate a combinational circuit.
  - Estimate the propagation delay of a combinational circuit.
- Suggest criteria for comparing functionally equivalent combinational circuits.

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 & static transfer functions

DEF: A gate is a device whose functionality is specified by a static transfer function.

$$\exists \Delta > 0$$

$$\forall x_0$$

$$\forall t \in [t_1, t_2] : x(t) = x_0 \Rightarrow \forall t \in [t_1 + \Delta, t_2] : \; y(t) = f(x_0).$$

This means that *output = func (input)* if the input did not change for a while.

This does not mean that the output must be logical (even if the input is fixed).

Extension of *dig*(x) to vectors

- Suppose $\vec{y} \in \mathbb{R}^n$, where $\vec{y} = (y_1, y_2, \cdots, y_n)$.
- The function *dig*$_n : \mathbb{R}^n \to \{0, 1, \text{non-logical}\}^n$ is defined by

$$\textit{dig}_n(y_1, y_2, \cdots, y_n) \triangleq (\textit{dig}(y_1), \textit{dig}(y_2), \cdots, \textit{dig}((y_n))).$$

- To simplify notation, we denote *dig*$_n$ simply by *dig* when the length n of the vector is clear.

Def: combinational gate

DEF: Consider a gate G with n inputs (denoted by $\vec{x}$) and k outputs (denoted by $\vec{y}$). The gate G is a combinational gate if there exists a $\Delta > 0$, such that, for all $\vec{x}(t) \in \mathbb{R}^n$,

$$\forall t \in [t_1, t_2] : \textit{dig}(\vec{x}(t)) \in \{0,1\}^n$$
$$\Rightarrow \quad \forall t \in [t_1 + \Delta, t_2] : \textit{dig}(\vec{y}(t)) \in \{0,1\}^k.$$

Remark:

- Logically stable input $\Rightarrow$ logical output.
- Static transfer function satisfies:

$$\textit{dig}(\vec{x}) \in \{0,1\}^n \Rightarrow \textit{dig}(f(\vec{x})) \in \{0,1\}^k.$$

- $\vec{x}$ may fluctuate but must remain logically stable.

Boolean functionality of a combinational gate

Suppose $f : \mathbb{R}^n \to \mathbb{R}^k$ is a static transfer function of a combinational gate G.
Define a Boolean function $B_f : \{0,1\}^n \to \{0,1\}^k$ as follows.
Given a Boolean vector $(b_1, \cdots, b_n) \in \{0,1\}^n$,

$$x_i \triangleq \begin{cases} V_{\textit{low}} - \varepsilon & \text{if } b_i = 0 \\ V_{\textit{high}} + \varepsilon & \text{if } b_i = 1. \end{cases}$$

The Boolean function B_f is defined by

$$B_f(\vec{b}) \triangleq \textit{dig}(f(\vec{x})).$$

G combinational circuit $\Rightarrow$ *dig*$(f(\vec{x}))$ is logical $\Rightarrow$ B_f is a Boolean function.

Boolean functionality of a combinational gate - cont.

Since

$$B_f(\vec{b}) \triangleq \textit{dig}(f(\vec{x})).$$

we can rephrase

$$\textit{dig}(\vec{x}) \in \{0,1\}^n \Rightarrow \textit{dig}(f(\vec{x})) \in \{0,1\}^k.$$

by

$$\textit{dig}(\vec{x}) \in \{0,1\}^n \Rightarrow \textit{dig}(f(\vec{x})) = B_f(\textit{dig}(\vec{x})).$$

$\Rightarrow$ Claim: In a combinational gate, the relation between the logical values of the inputs and the logical values of the outputs is specified by a Boolean function.

A consistent combinational gate

propagation delay - upper bound on the amount of time that elapses from the moment that the inputs (nearly) stop changing till the moment that the output (nearly) equals the value of the static transfer function.

DEF: A combinational gate G with inputs $\vec{x}(t)$ and outputs $\vec{y}(t)$ is consistent at time t if *dig*$(\vec{x}(t)) \in \{0,1\}^n$ and $\vec{y}(t) = B_f($*dig*$(\vec{x}(t)))$.

propagation delay - upper bound on time that elapses from logically stable inputs till gate is consistent.

brief roundup

static transfer func $\Rightarrow$ gate $\Rightarrow$ comb. gate where

- gate: *o*utputs = func(inputs)
- combinational gate: *l*og. stable inputs $\Rightarrow$ logical outputs
- consistency : when *dig*$(\vec{y}) = B_f($*dig*$(\vec{x}))$.
- propagation delay: upper bound on time needed to reach consistency.

Very helpful if you need to deal with the following question: Is a device G a good candidate for an AND-gate?

Not helpful if you are given a library of gates to work with. In this case one prefers not to deal with analog signals...

Back to the digital world

- digital signals - refer to input and output signals as digital signals.
- goals for combinational gates:
  - specification - specify functionality using a Boolean function.
  - consistency - define when a gate satisfies the specification.
  - performance - quantify how fast it takes a gate to satisfy the specification.
- propagation delay - loosen definition (allow analog inputs to change as long as they are logically stable)

Specification & Consistency

- Consider a combinational gate G with 2 inputs, denoted by x_1, x_2, and a single output, denoted by y.
- $x_1(t), x_2(t)$ - the digital signals corresponding to inputs.
- $y(t)$ - the digital signal corresponding to the output.
- $B : \{0,1\}^2 \rightarrow \{0,1\}$ - a binary function (specification)

DEF: G is consistent with the Boolean function B at time t if the input values are digital at time t and

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

Propagation delay of comb. gate

DEF: A combinational gate G implements a Boolean function $B : \{0,1\}^2 \rightarrow \{0,1\}$ with propagation delay t_{pd} if the following holds.
For every $\sigma_1, \sigma_2 \in \{0,1\}$, if $x_i(t) = \sigma_i$, for $i = 1,2$, during the interval $[t_1, t_2]$, then

$$\forall t \in [t_1 + t_{pd}, t_2] \ : \ y(t) = B(\sigma_1, \sigma_2).$$

Equivalently,

x_1, x_2 stable in $[t_1, t_2]$

$\Rightarrow$

G is consistent with B in the interval $[t_1 + t_{pd}, t_2]$.

Propagation delay - remarks

- If $t_2 < t_1 + t_{pd}$, then the statement in the above definition is empty.
- Propagation delay is an upper bound. The actual amount of time that passes till a combinational gate is consistent is very hard to compute (there is also randomness involved). We may always be overly pessimistic (i.e., using a propagation delay that is larger than the actual delay will not introduce errors).

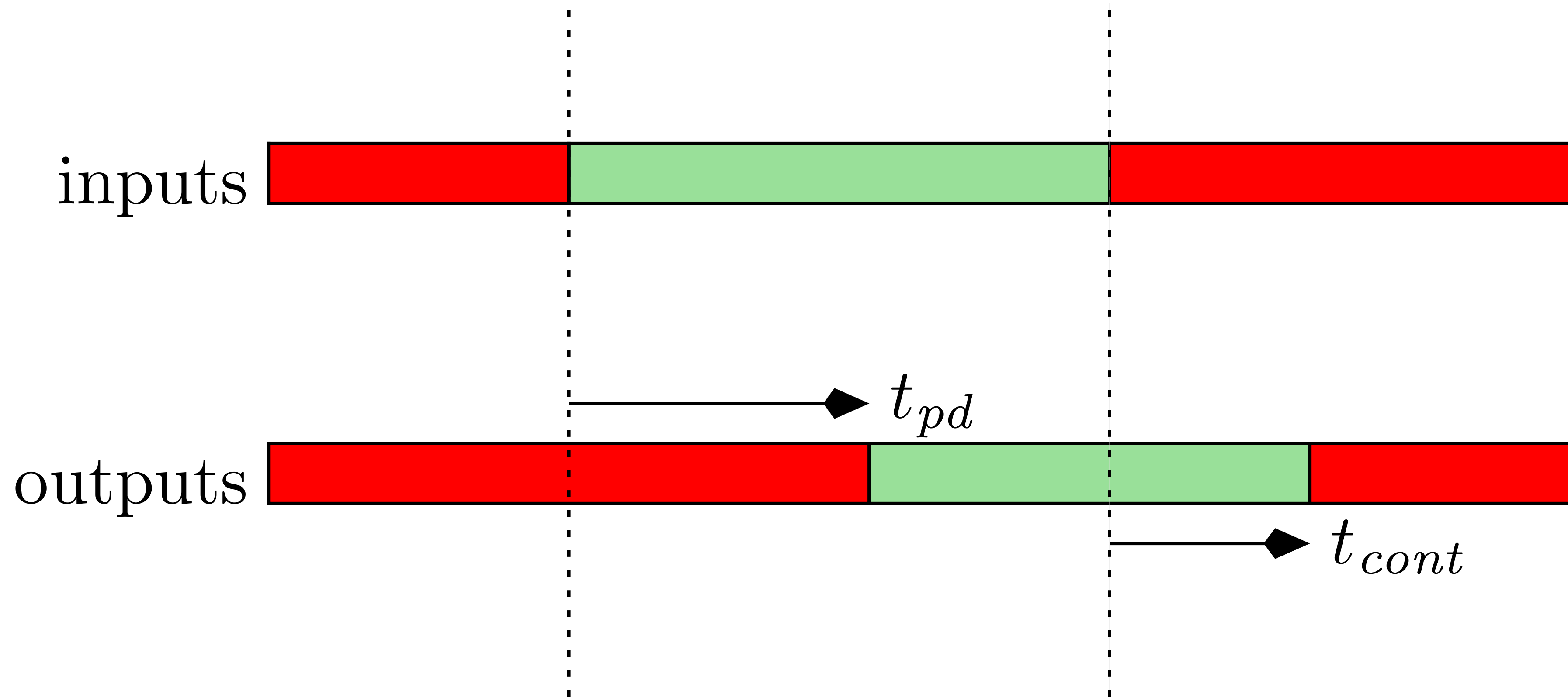
Contamination delay

Contamination delay - a lower bound on 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 can “react” to a change in the input

We we will assume that the contamination delay is zero.

Example: propagation delay and contamination delay



- x-axis: time.
- red segments: signal is not guaranteed to be logical.
- green segments: signal is guaranteed to be stable.

Combinational circuits - building blocks

Combinational circuits are built of combinational gates and wires & nets.

Combinational gates

- Implement a Boolean function.
- Since we consider only combinational gates, we refer to a combinational gate, in short, as a gate.
- Typical gates: inverter (NOT-gate), OR-gate, NOR-gate, AND-gate, NAND-gate, XOR-gate, NXOR-gate, multiplexer (MUX).
- fan-in : number of input terminals (typically, at most 3).

Input ports denoted by the set $\{in(G)_i\}_{i=1}^{n}$, where n denotes the fan-in of G.

Output ports denoted by the set $\{out(G)_i\}_{i=1}^{k}$, where k denotes the number of output ports of G.

Wires & Nets

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

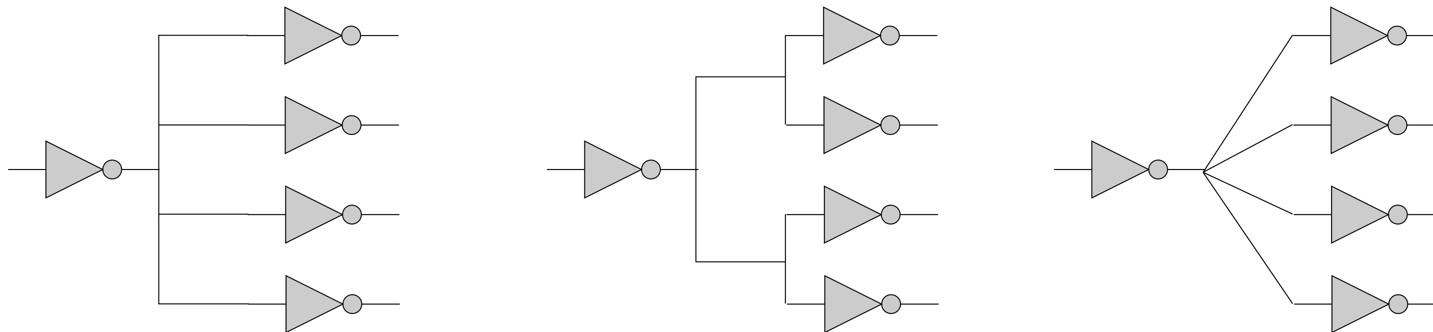
Ignore how connections are actually made.

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 to 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.



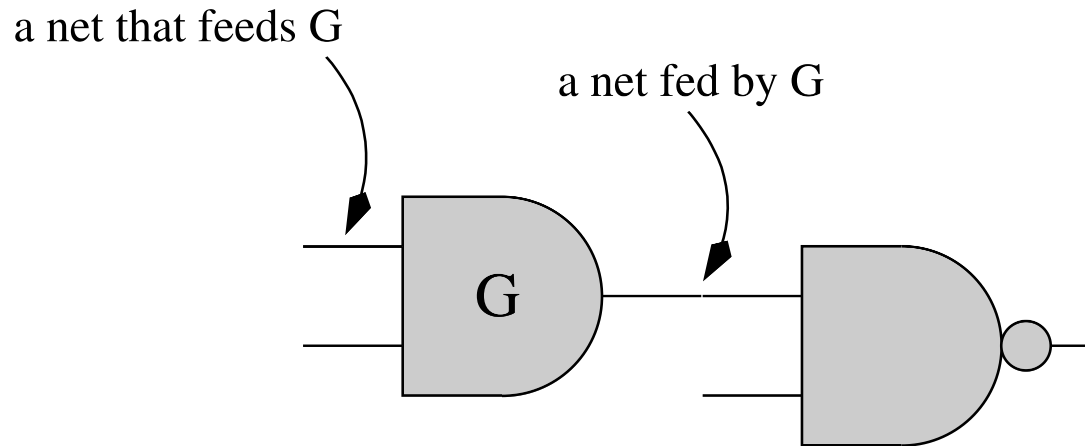
Digital signals for nets

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.

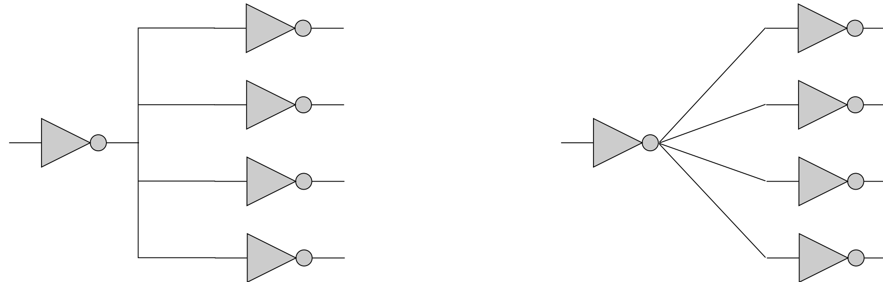
Simple nets

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 $t_{out} \rightarrow t_i$.

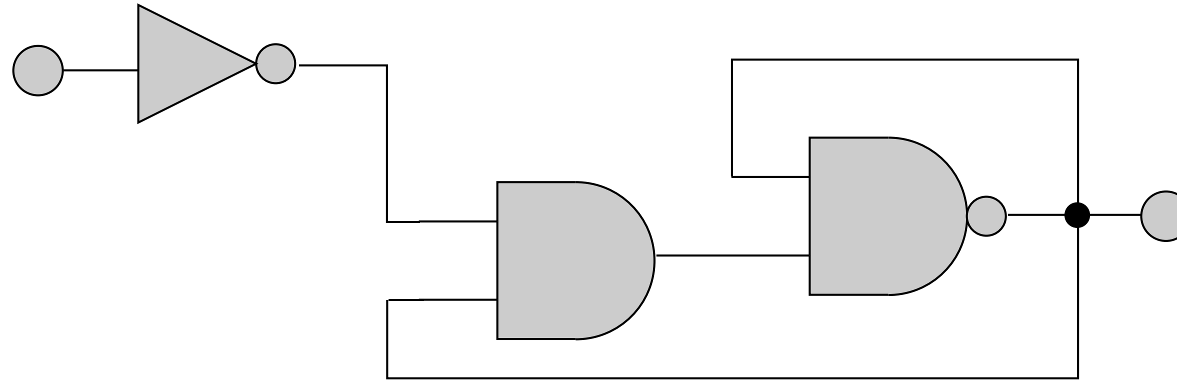


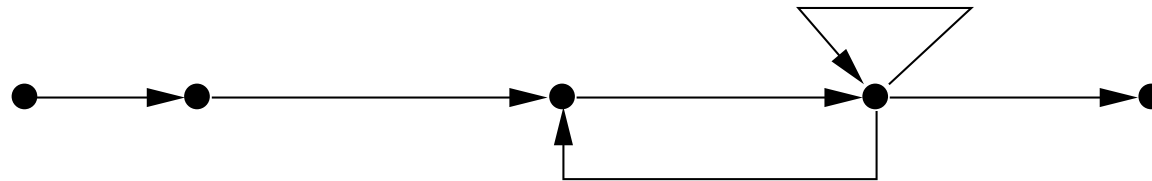
Directed graph corresponding to simple nets

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

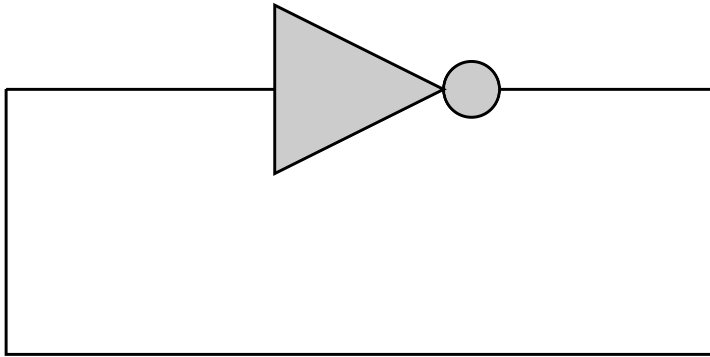
- $DG(C)$ - a directed graph.
- Nodes - gates of C.
- Directed edges - directed edge $u \longrightarrow v$ if there is a net N such that: (i) an output terminal of gate u feeds N, and (ii) an input terminal of v is fed by N.

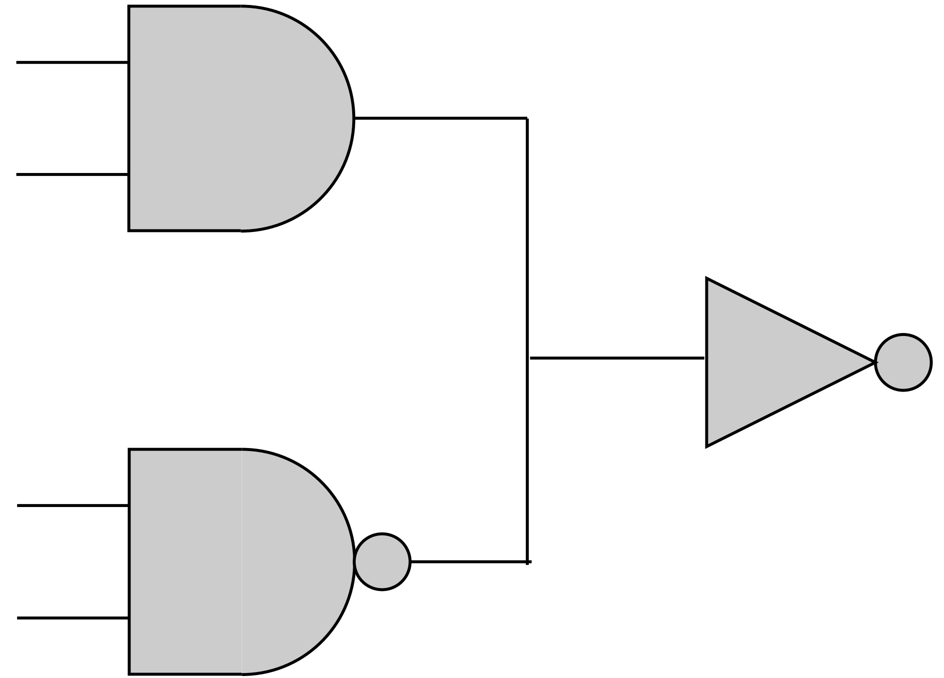
Example of a circuit C and a directed graph $DG(C)$





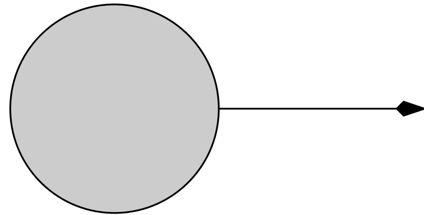
Are these circuits combinational circuits?

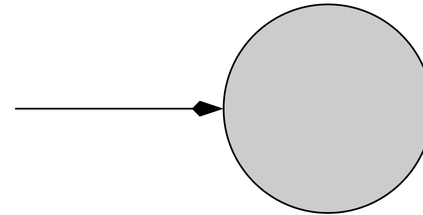




Input gates & output gates

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





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

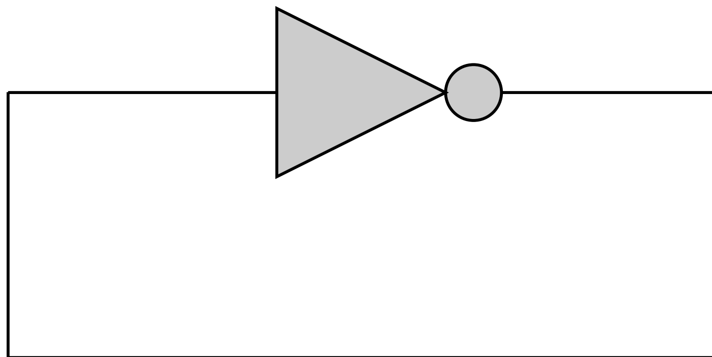
Definition of combinational circuits is independent of the gate types (e.g. inverter, NAND-gate, etc.). The question of whether a circuit is combinational is a purely topological question (i.e. are the interconnections between gates legal?).

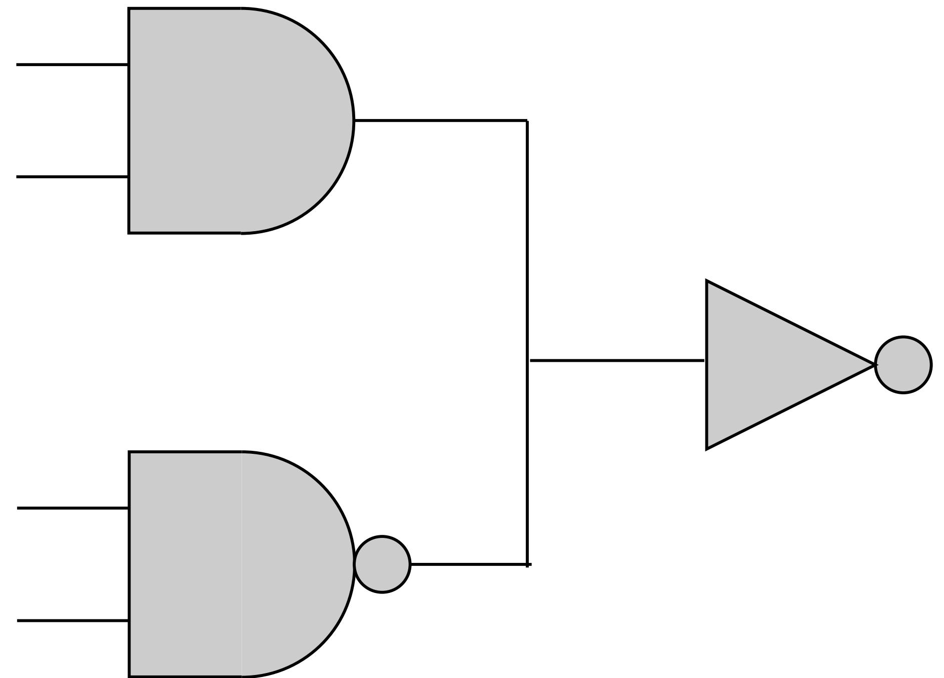
syntax - “grammar” rules for forming compound circuits from

simple circuits.

Back to “bad” examples...

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





Question: Design an efficient algorithm to check if a given circuit is combinational.

Combinational circuits: Syntax $\Rightarrow$ Semantics

- Completeness: for every Boolean function B, there exists a combinational circuit that implements B (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 time that is linear in the circuit's size.
- 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:

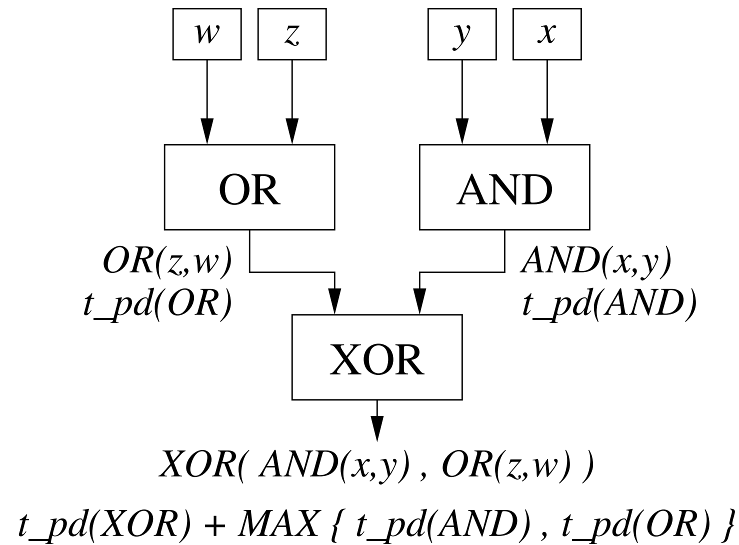
1. a Boolean function $B_N : \{0,1\}^k \rightarrow \{0,1\}$, and
2. a propagation delay $t_{pd}(N)$

such that

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

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 $u \rightarrow v$ 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 $1 \leq i \leq k$).
- e_i - nets fed by gate v_i (assume only one).
- $e_1, e_2, \ldots, e_m$ - ordering of the nets in $\mathcal{N}$.
- Note that e_1 is fed by v_1, ..., e_k is fed by v_k, hence $e_1 = x_1, \ldots, e_k = x_k$.

Proof - Induction hypothesis

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

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

such that the network e_i implements the Boolean function

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

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

Proof - Induction basis

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

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

$\Longrightarrow$ define

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

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. Also, assume that v_i has a single output.
- Topological ordering $\Rightarrow j, k \leq m'$.
- Ind. Hyp. $\Rightarrow$:
  - e_j implements a Boolean function B_{e_j} with $t_{pd}(e_j)$.
  - e_k implements a Boolean function B_{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 B_{v_i} with propagation delay $t_{pd}(v_i)$.
- $\Rightarrow$ the output of gate v_i equals

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

during the interval

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

- Define

$$B_{e_{m'+1}}(\vec{\sigma}) = B_{v_i}(B_{e_j}(\vec{\sigma}), B_{e_k}(\vec{\sigma})).$$

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

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.
  -

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

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

Complexity: linear if each gate has a single output terminal and computing B_{v_j} requires constant time. (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

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

Remark: Naive cost measure - ignores wiring. The more interesting measure is area. However, computing the area requires detailed physical design (which is hard to compute and is very sensitive to technology).

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:

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

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 words form sentences).
  - 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 circuits
- combinational gates are easy to:
  - recognize
  - simulate
  - analyze (propagation delay)
- quality criteria: cost & delay

Chapter 3: Trees

Computer Structure - Spring 2007

©Dr. Guy Even

Tel-Aviv Univ.

Preliminary Questions

- Which Boolean functions are suited for implementation by tree-like combinational circuits?
- In what sense are tree-like implementations optimal?

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

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

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

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

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:

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

Example:

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

Note that NOR is not associative!

f_n : the associative case

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

Claim: If $f : \{0,1\}^2 \to \{0,1\}$ is an associative Boolean function, then for every $k \in [2, n-2]$:

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

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

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

Remark: Only last 4 functions are “interesting”. We focus on OR_n.

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 tree.

   Note that in the tree:

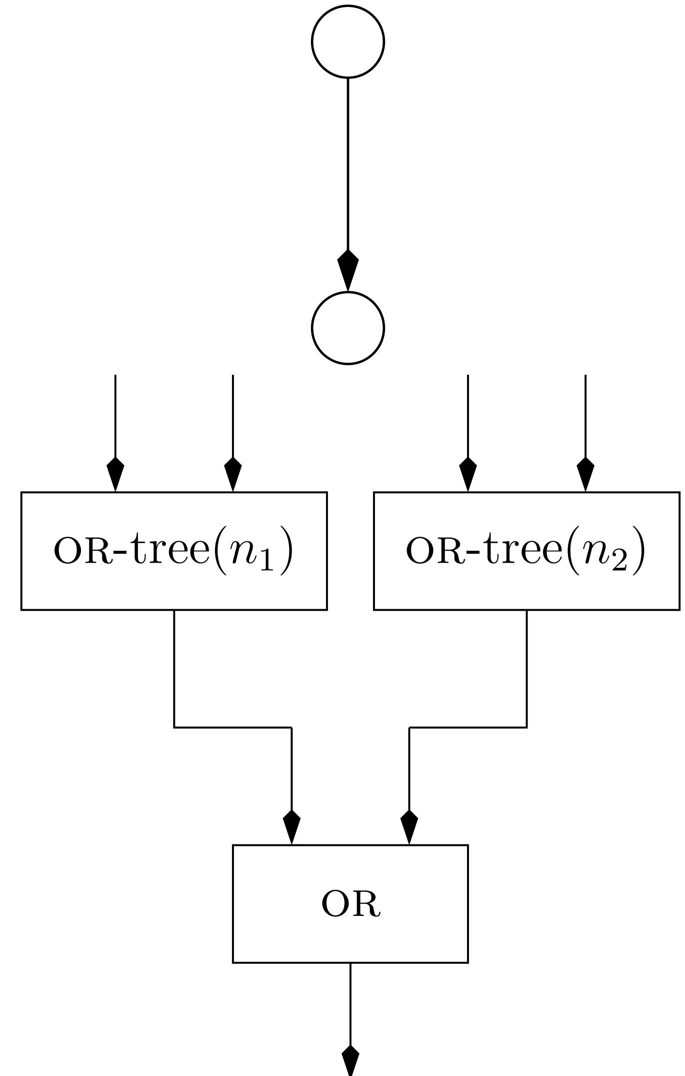
   - leaves correspond to the inputs $x[n-1:0]$ and the output y.
   - interior nodes - OR-gates.
   - Could root the tree, and then the root is the output.

Recursive definition of OR-trees

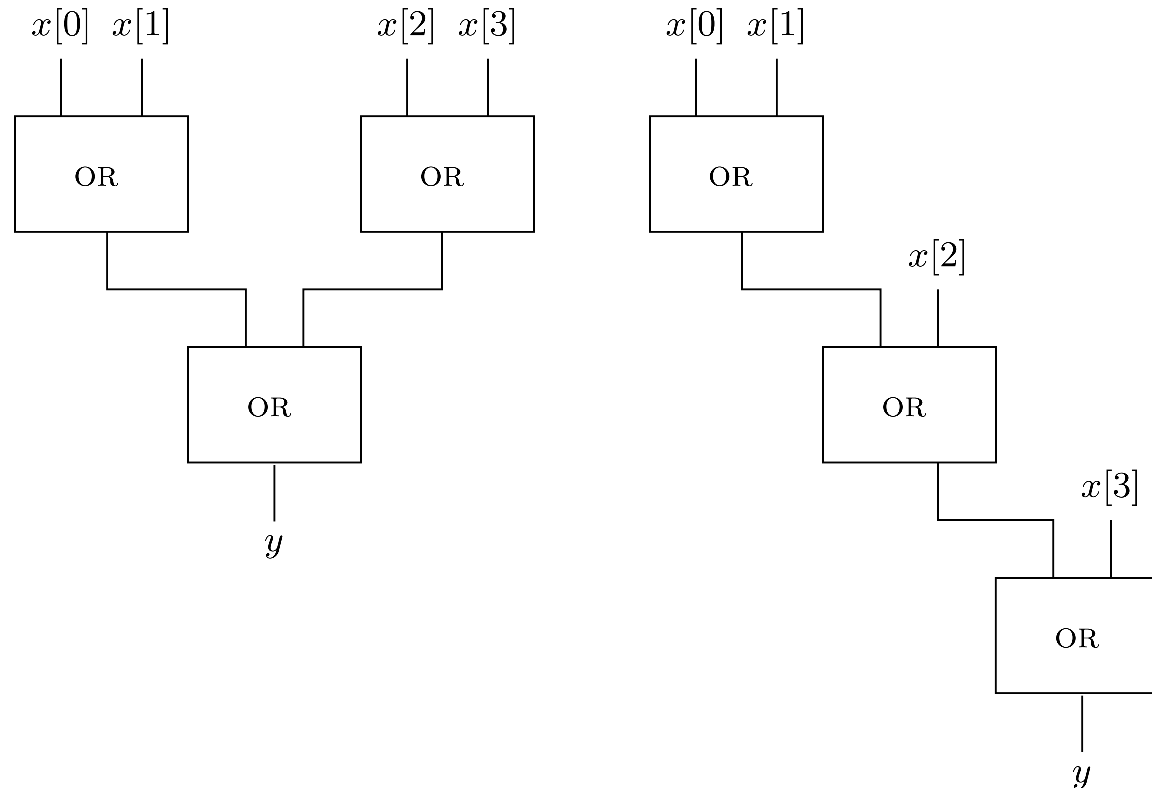
Def: an OR-tree(n) is defined recursively as follows:

basis: $(n = 1)$ - a trivial circuit in which input directly feeds output.

step: $(n > 1)$ - compose two trees T_{n_1} & T_{n_2} by feeding their output to a new gate.



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. What about $n = 1$?

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. $\Rightarrow c(C_1) = (n_1 - 1) \cdot c(\text{OR})$ & $c(C_2) = (n_2 - 1) \cdot c(\text{OR})$.
-

$$\begin{aligned} c(C) &= c(g) + c(C_1) + c(C_2) \\ &= (1 + n_1 - 1 + n_2 - 1) \cdot c(\text{OR}) \\ &= (n - 1) \cdot c(\text{OR}). \end{aligned}$$

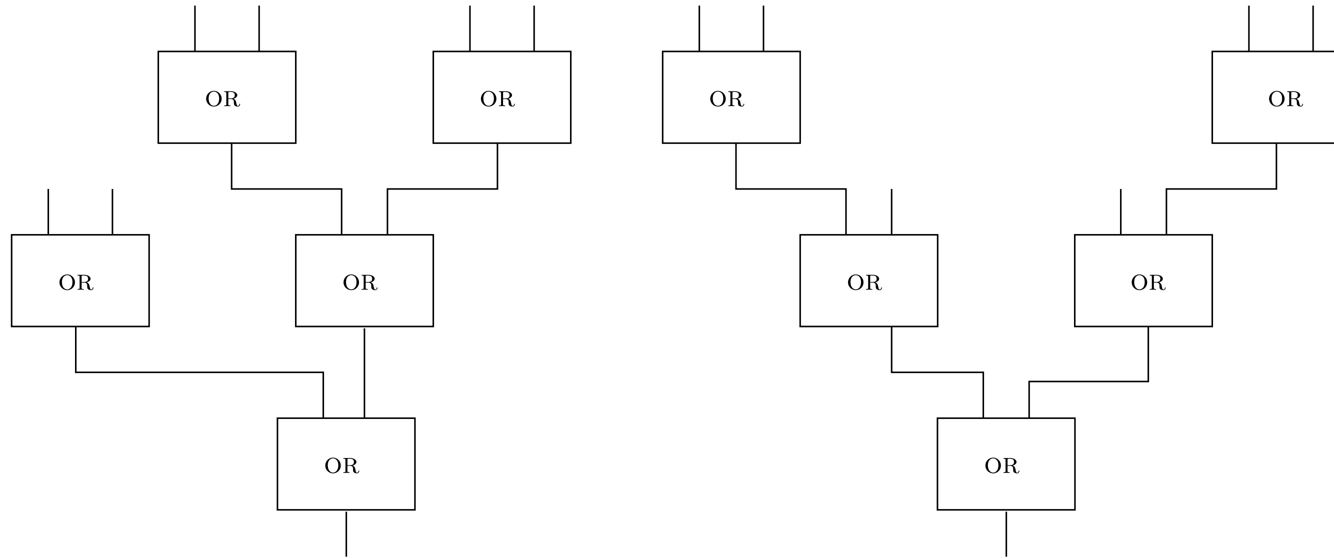
QED

Delay of OR-trees

Since all gates in an OR-tree are identical, delay of tree is proportional to the length of longest path from input to output (i.e., depth of tree).
DEF: A rooted tree is a minimum depth tree if its depth is minimum among all the rooted trees with the same number of leaves.
Since all trees with n inputs have the same cost, we may focus on min. depth trees.

Question: How many min. depth trees are there with $n = 2^k$ inputs?

Example: delay of OR-trees



Question: Are these min. depth trees?

Lower bound on depth

Question: Prove that if T_n is a rooted binary tree with n leaves, then the depth of T_n is at least $\lceil \log_2 n \rceil$.

Upper bound on depth

- Wish to present a simple procedure for constructing T_n with depth $\lceil \log_2 n \rceil$.
- Natural candidate: balanced trees...

DEF: Two positive integers a, b are a balanced partition of n if: (1) $a + b = n$, and (2) $\max\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\} \leq \lceil \log_2 n \rceil - 1$.

Example: Suppose $n = 6$. In this case $\lceil \log_2 n \rceil = 3$. Hence $a = b = 3$ is a balanced partition, but so is $a = 4, b = 2$.

Procedure for constructing “balanced” trees

Question: Consider the following recursive algorithm for constructing a binary tree T_n with $n \geq 2$ leaves. Prove that the depth of T_n is $\lceil \log_2 n \rceil$.

1. The case that $n \leq 2$ is trivial (two leaves connected to a root).
2. If $n > 2$, then let a, b be balanced partition of n.
3. Compute trees T_a and T_b. Connect their roots to a new root to obtain T_n.

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

assumption 2: the cost & delay 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 $f : \{0,1\}^n \rightarrow \{0,1\}$ denote a Boolean function. Let $\sigma \in \{0,1\}$. The Boolean function $g : \{0,1\}^{n-1} \rightarrow \{0,1\}$ 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 C.

Proof: by contradiction,

- assume $i \in$ *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

$$f(w') \neq f(w'')$$
$$w'[i] \neq w''[i]$$
$$.w'[j] = w''[j], \text{for every } j \neq i$$
$$.$$

Proof of Simulation Algorithm
$\Rightarrow C$ 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

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

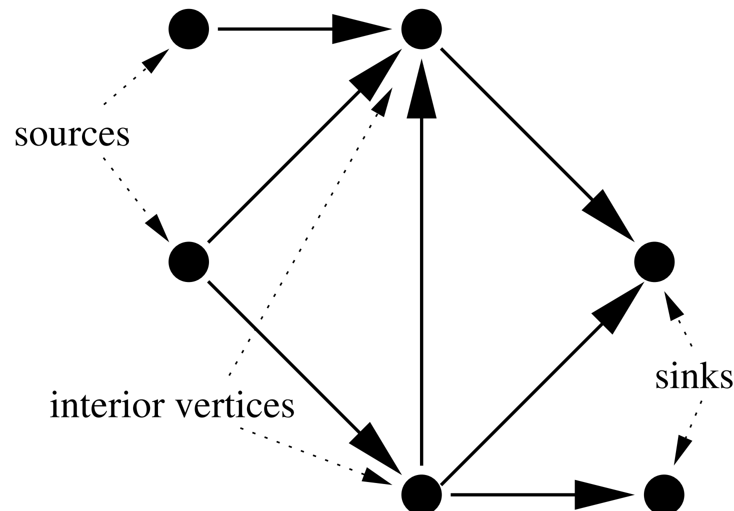
Corollary: If C_n is a combinational circuit that implements OR$_n$, then $c(C_n) \geq n - 1$.

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

DAG terminology

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

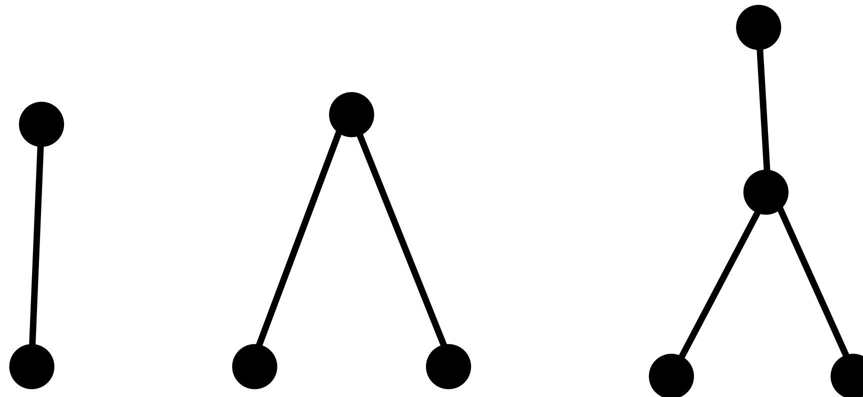
- $\deg_{in}(v)$: in-degree of a vertex v is the number of edges that enter the vertex v.
- $\deg_{out}(v)$: out-degree of a vertex v : is the number of edges that emanate from the vertex v.
- source - a vertex with in-degree zero.
- sink - a vertex with out-degree zero.
- interior vertex - a vertex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



Leaves and interior vertices in trees

- Let $T = (V, E)$ denote a tree with at least two vertices.
- A leaf is a vertex of degree 1.
- An interior vertex is a vertex that is not a leaf.
- leaves(V) - set of leaves in V.
- interior(V) - set of interior vertices in V.
- Claim:
  If the degree of every vertex in T is at most three, then

$$|\text{interior}(V)| \geq |\text{leaves}(V)| - 2.$$



Underlying graph of $DG(C)$

- C - a combinational circuit & $DG(C) = (V, A)$ - a DAG
- underlying graph of $DG(C)$ - undirected graph $G = (V, E)$, where $(u, v) \in E \iff (u \to v) \in A$.
- If fan-in of every gate in C is at most 2 and G is a tree, then degree of every vertex in G is at most 3.
- Leaves in G correspond to input and output gates in C.
- Interior vertices in G correspond to non-trivial gates in C.
- Case of a tree:
  Claim: Assume C has n inputs and a single output. Assume fan-in of all gates is at most 2. If G is a tree, then

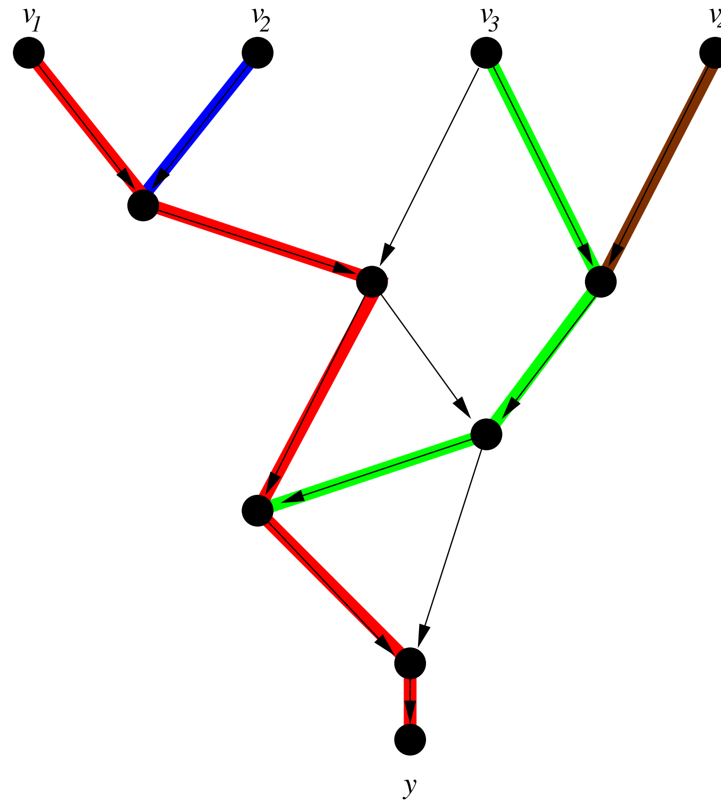
$$c(C) \geq n - 1.$$

Proof of linear cost lower bound theorem

- If underlying graph of $DG(C)$ is a tree, then previous claim proves the theorem.
- 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 |interior nodes| $\geq$ |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 k.

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 $f : \{0,1\}^n \rightarrow \{0,1\}$. If the fan-in of every gate in $\mathcal{G}$ is at most k, then the delay of C is at least $\log_k |\textit{cone}(f)|$.

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

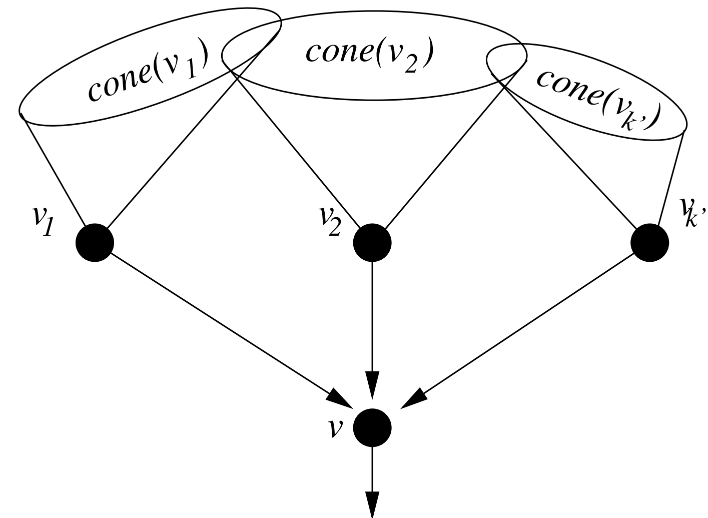
$$t_{pd}(C_n) \geq \lceil \log_k n \rceil .$$

Proof of logarithmic lower bound

- deal only with the graph $DG(C)$.
- show that exists a path with at least $\log_k |\textit{cone}(f)|$ 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 $|\textit{cone}(output)| \geq |\textit{cone}(f)|$.
- $d(v)$ - max number of interior vertices along a path from a source in *cone*(v) to v (including v).
- suffice to prove that $d(v) \geq \log_k |\textit{cone}(v)|$.

Proof: $d(v) \geq \log_k |\boldsymbol{cone}(v)|$

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



$$|\textit{cone}(v)| \leq \sum_{i=1}^{k'} |\textit{cone}(v_i)|$$

$$\leq k' \cdot \max\{|\textit{cone}(v_i)|\}_{i=1}^{k'}\}.$$

Cont. proof: $d(v) \geq \log_k |\textit{cone}(v)|$

Let v' denote a predecessor of v that satisfies

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

The induction hypothesis implies that

$$d(v') \geq \log_k |\textit{cone}(v')|.$$

But,

$$\begin{aligned} d(v) &\geq 1 + d(v') \\ &\geq 1 + \log_k |\textit{cone}(v')| \\ &\geq \log_k k + \log_k |\textit{cone}(v)|/k' \\ &\geq \log_k |\textit{cone}(v)|. \end{aligned}$$

Cont. proof: $d(v) \geq \log_k |\textit{cone}(v)|$

- 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

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

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 $\geq |cone(f)| - 1$.
  - $t_{pd} \geq \log_k |cone(f)|$.

Chapter 4: Decoders & Encoders

Computer Structure - Spring 2007

©Dr. Guy Even

Tel-Aviv Univ.

Preliminary questions

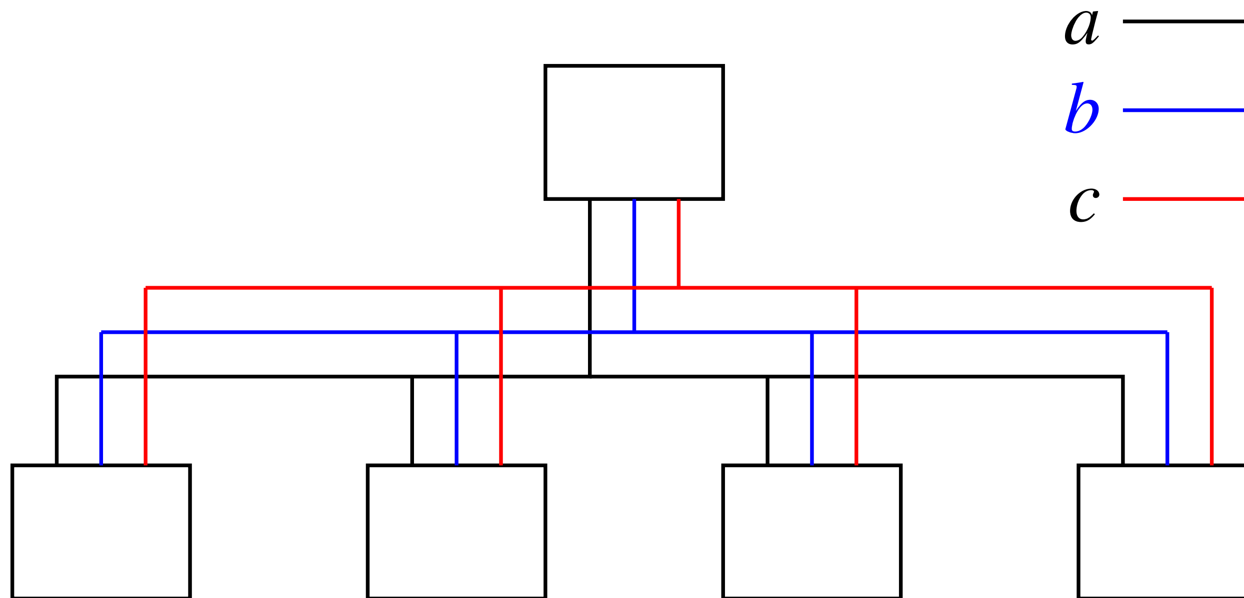
- Suggest methods for designing combinational circuits that implement nontrivial Boolean functions.
- How do we check if a design of a combinational circuit is correct?

Goals

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

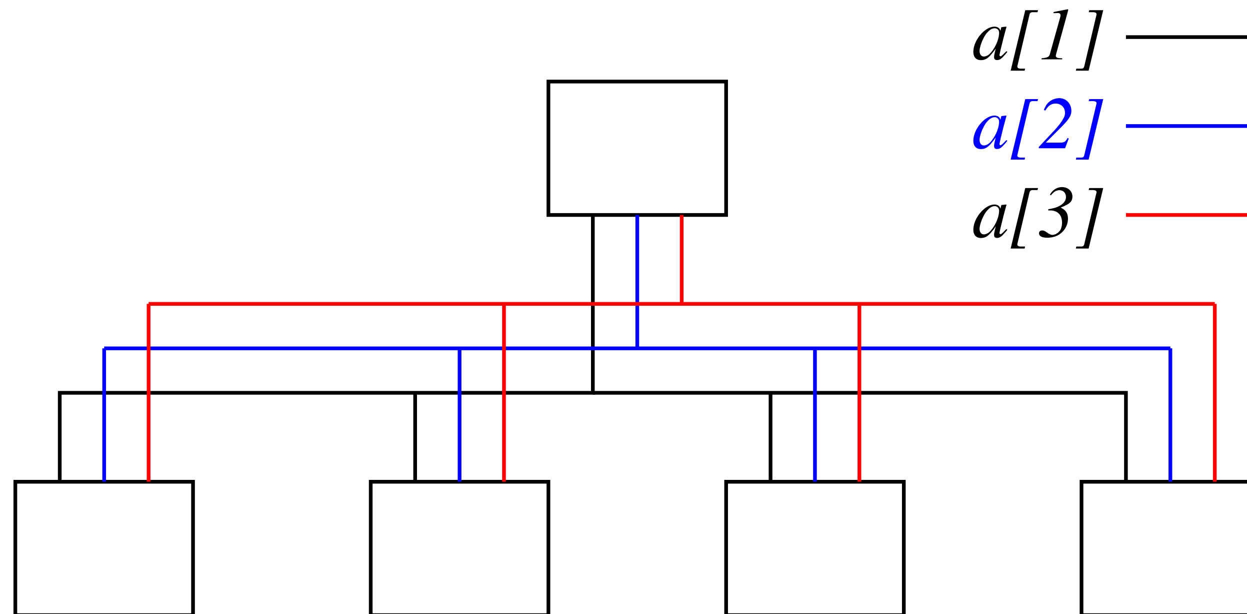
Parallel nets

We could use separate names for each net:



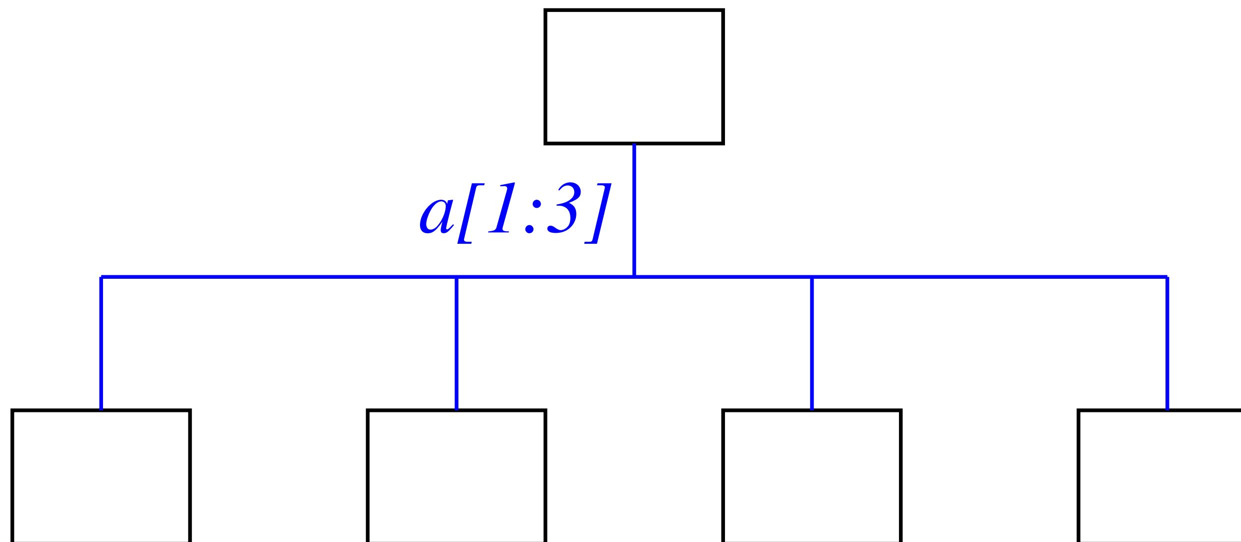
Indexing parallel nets

We could index the names of the nets:



Bus notation

We could refer to the nets as a bus $a[1:3]$:

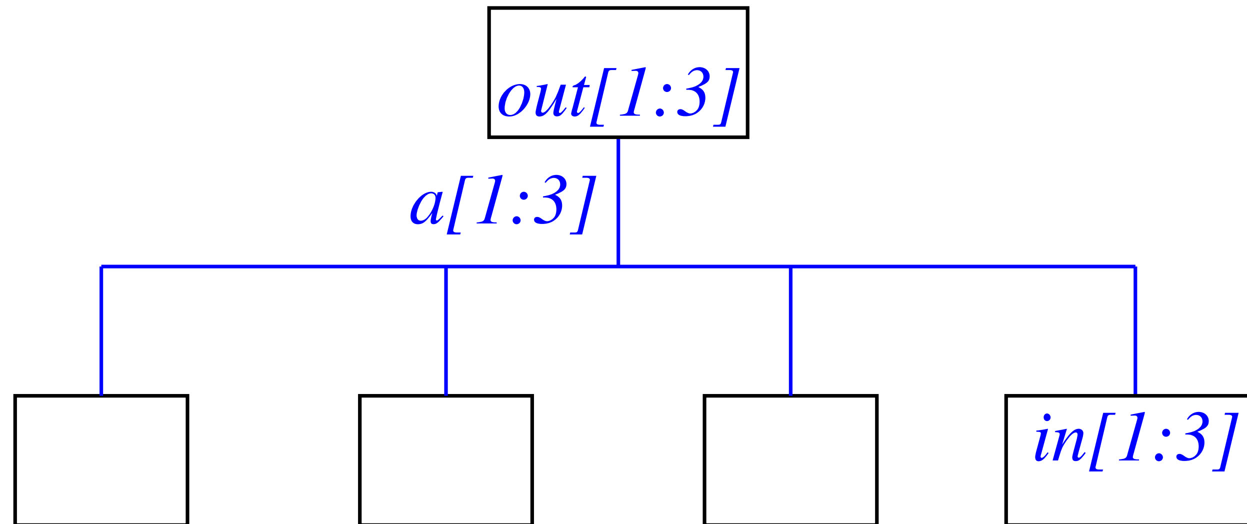


To make sense we would have to give indexed names to inputs/output terminals.

For example: input terminals $in[1:3]$ and output terminals $out[1:3]$. And then $a[i]$ feeds $in[i]$ and is fed by $out[i]$.

Bus notation

Indexes of terminal names match indexes of bus names:



Bus related definitions

- bus - set of nets that are connected to the same modules.
- bus width - number of nets in the bus.

Example: The bus $a[1:3]$ connects the output terminals $out[1:3]$ to the input terminals $in[1:3]$. The width is 3.

Indexed buses

- Assignment $in[1:4] \leftarrow out[1:4]$ means $in[1] \leftarrow out[1], \ldots, in[4] \leftarrow out[4]$.
- Reversing $in[1:4] \leftarrow out[4:1]$ means $in[1] \leftarrow out[4], \ldots, in[4] \leftarrow out[1]$.
- Hardwired shifting $in[i:j] \leftarrow out[i+5:j+5]$ means $in[i] \leftarrow out[i+5], \ldots, in[j] \leftarrow out[j+5]$.

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,

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

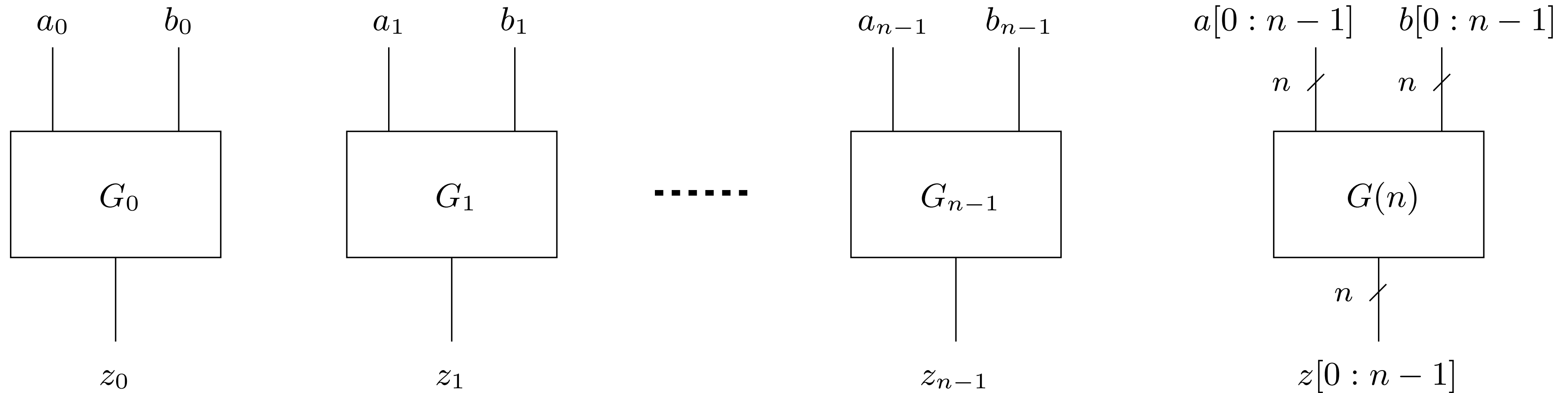
means

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

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.

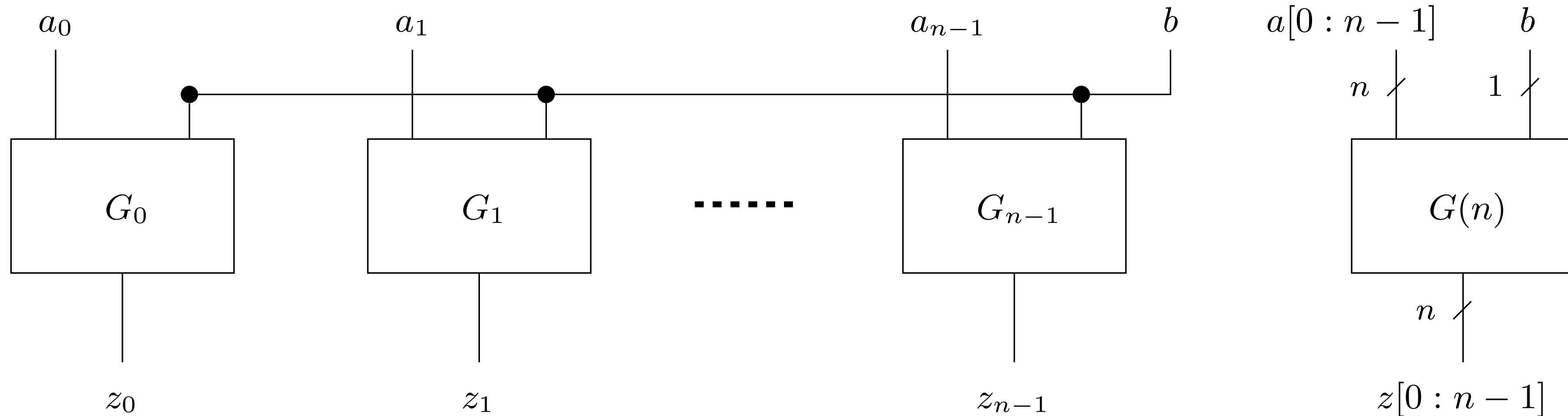
multiple instances of the same gate



- 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.



(A) (B)

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. Ignore this phenomenon 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

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

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

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

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

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:

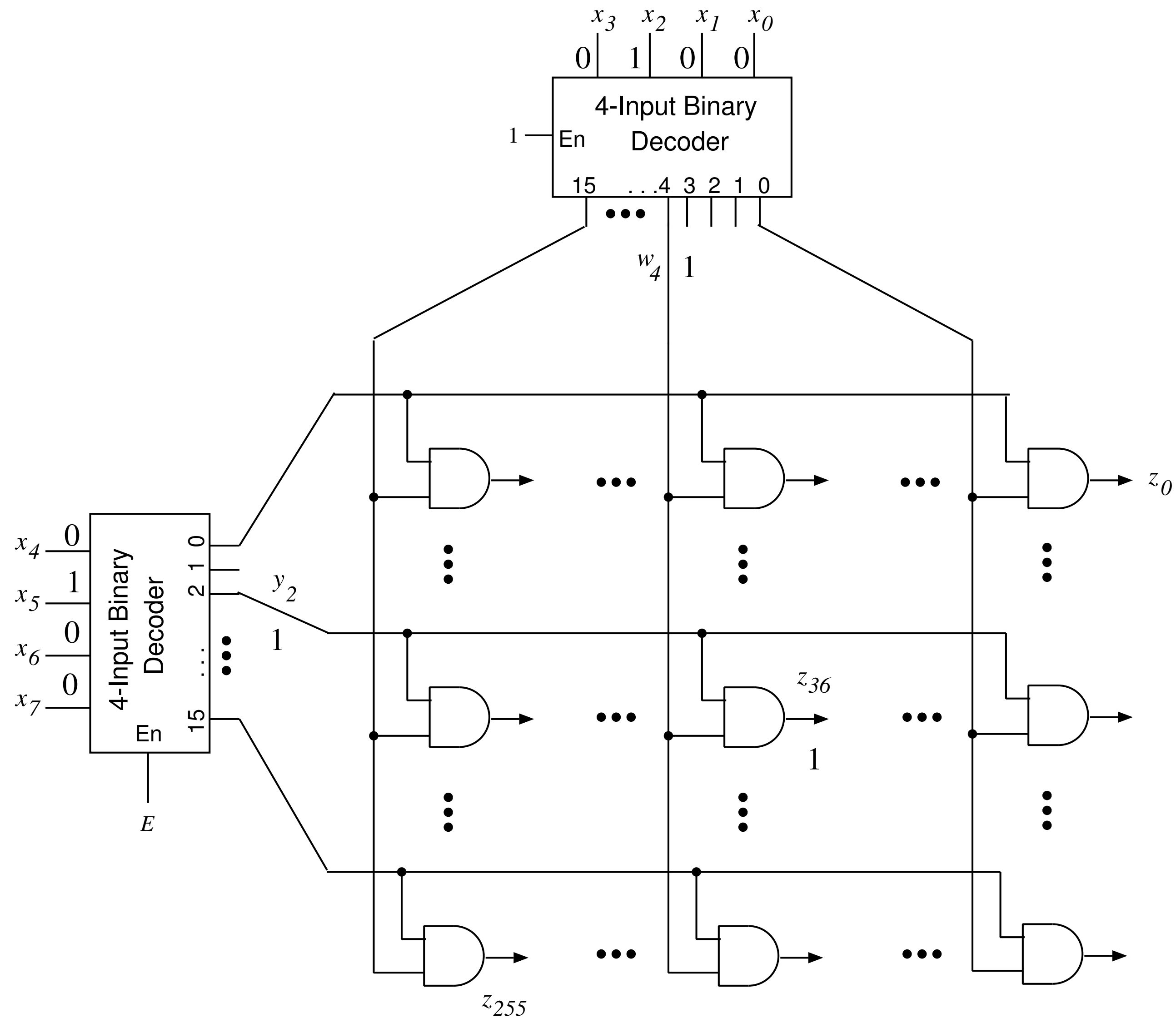
$$y[i] = 1 \Longleftrightarrow \langle \vec{x} \rangle = i.$$

- $|$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

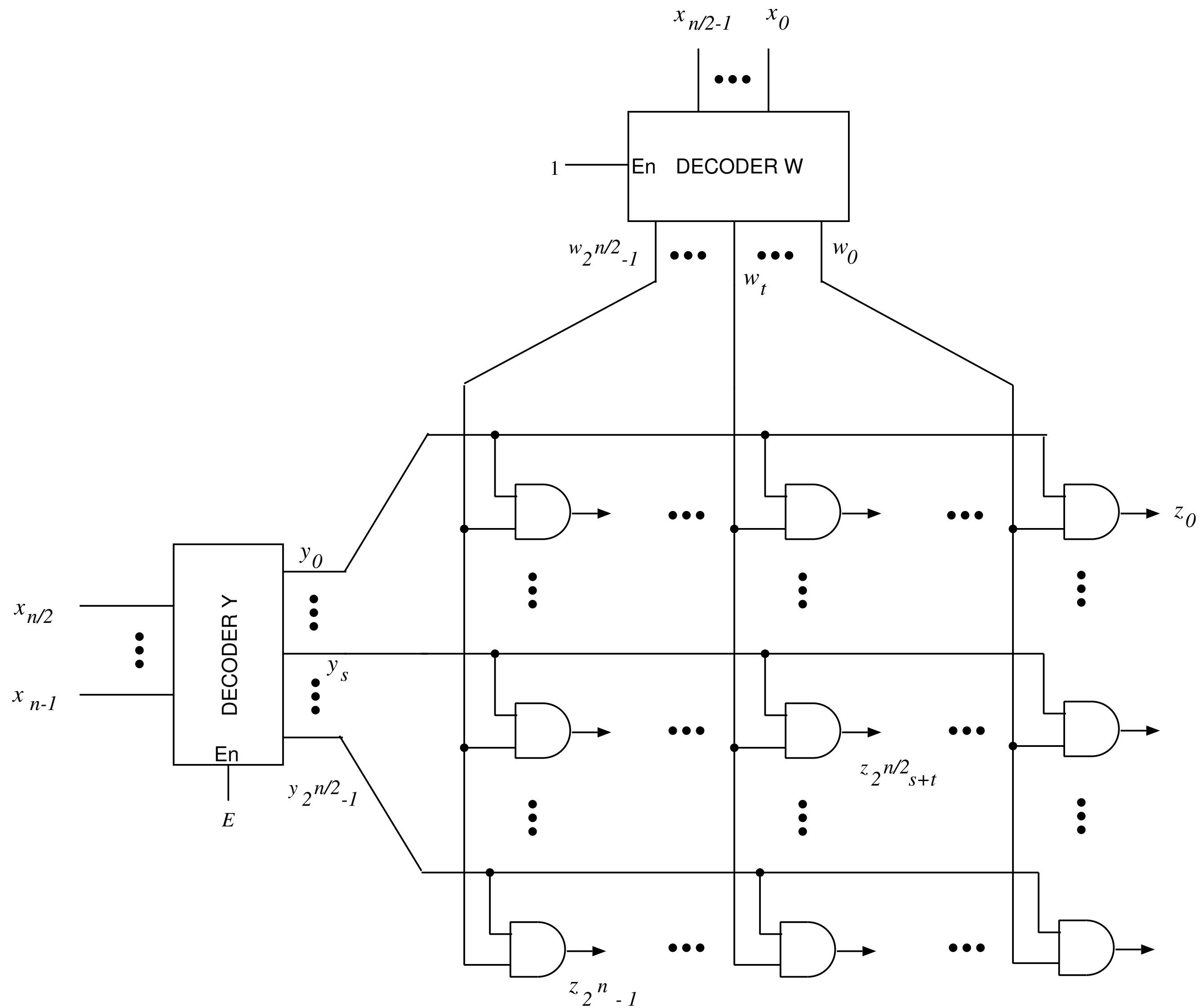
example: DECODER(3) - on input $x = 101$, the output y equals

00100000.

DECODER(8) - schematic



DECODER(n) - schematic



Weaknesses of standard decoder descriptions

- Why is the design correct?
- Can you prove correctness?
- 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?

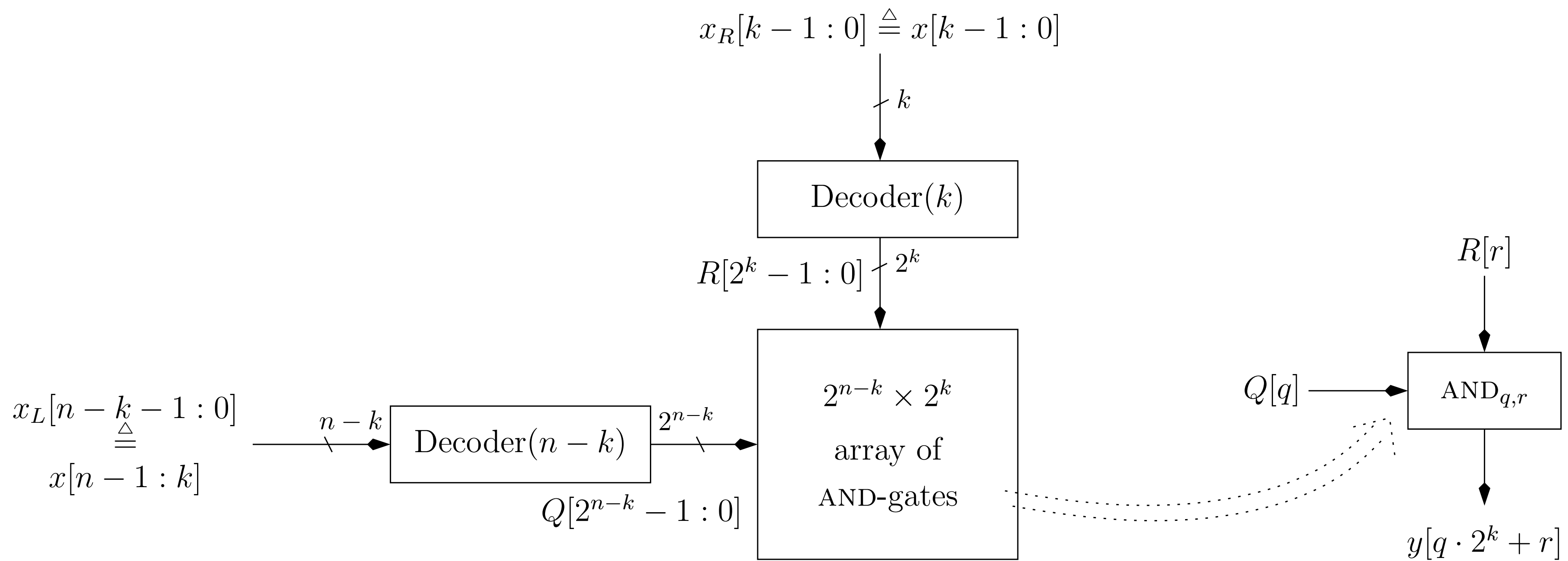
Formal description of $\text{DECODER}(n)$

Use recursion.

Basis: $\text{DECODER}(1)$: 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:

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

Induction basis: $n = 1$: trivial.

Induction step: show that:

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

induction step - cont.

- Fix $i \in [2^n - 1 : 0]$ and prove that $y[i]$ is correct.
- Divide by 2^k: $\quad i = q \cdot 2^k + r$.
- Ind. Hyp. $(k) \Rightarrow$

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

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

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

- Now $y[i] = y[q \cdot 2^k + r]$, hence

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

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 $\text{DECODER}(n)$ 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 $\text{DECODER}(n)$ is asymptotically optimal with respect to cost and delay.

Weight of binary strings

Def: The Hamming weight of a binary string is defined by

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

Note that

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

example:

$$\textit{wt}(01101) = 3.$$

Encoders - Definitions

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

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

is defined as follows:

$$\textit{wt}(y) = 1 \Longrightarrow y[\langle \textsc{encoder}_n(\vec{y}) \rangle] = 1.$$

Examples:

$$\textsc{encoder}_2(0100) = 10$$

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

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}.

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:

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

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 $\textsc{encoder}'(n)$.
- Prove correctness of $\textsc{encoder}'(n)$.
- Improve $\textsc{encoder}'(n)$ to obtain $\textsc{encoder}^*(n)$.
- Prove functional equivalence

$$\textsc{encoder}^*(n) \equiv \textsc{encoder}'(n),$$

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

ENCODER$'(n)$

Use recursion.

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

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

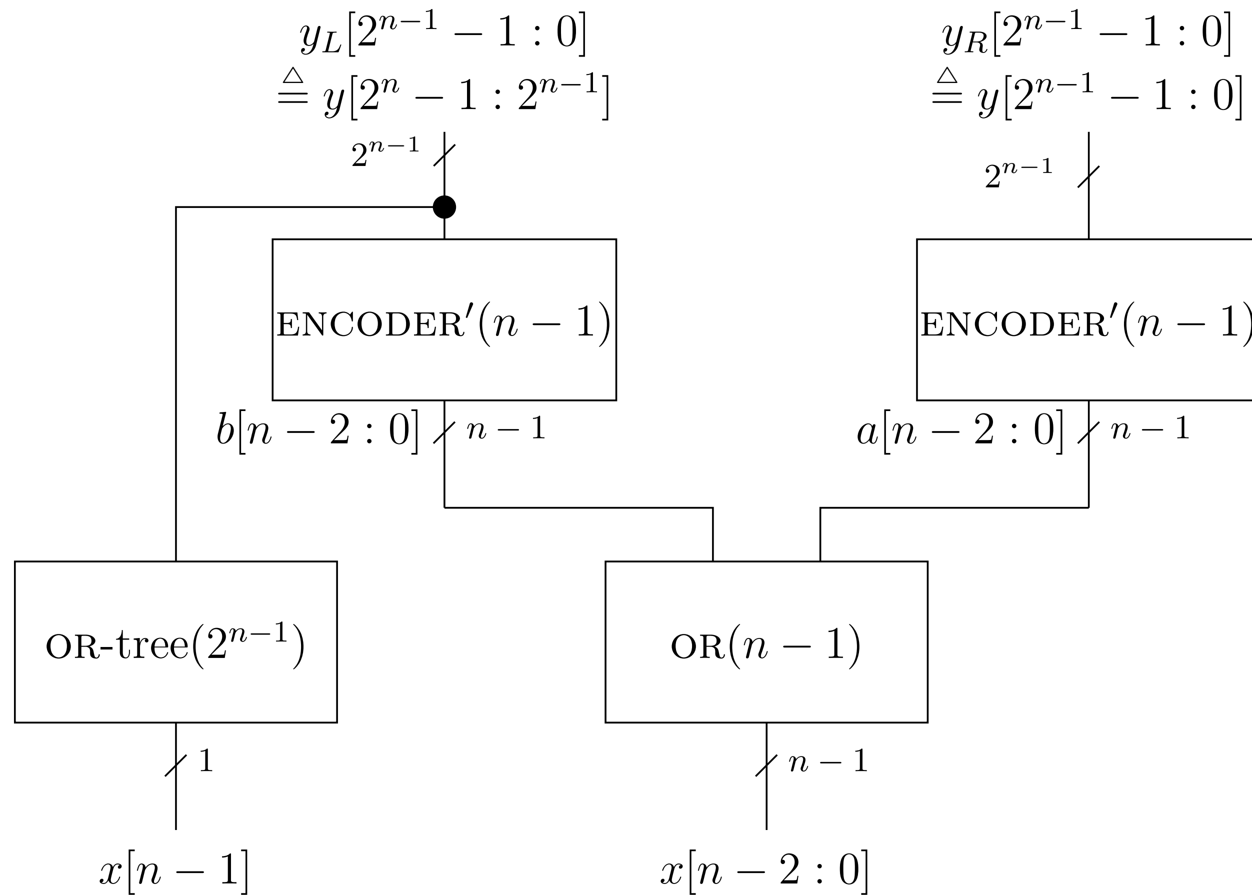
Define $\text{ENCODER}_n(0^{2^n})$

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.

ENCODER$'(n)$ - recursive step



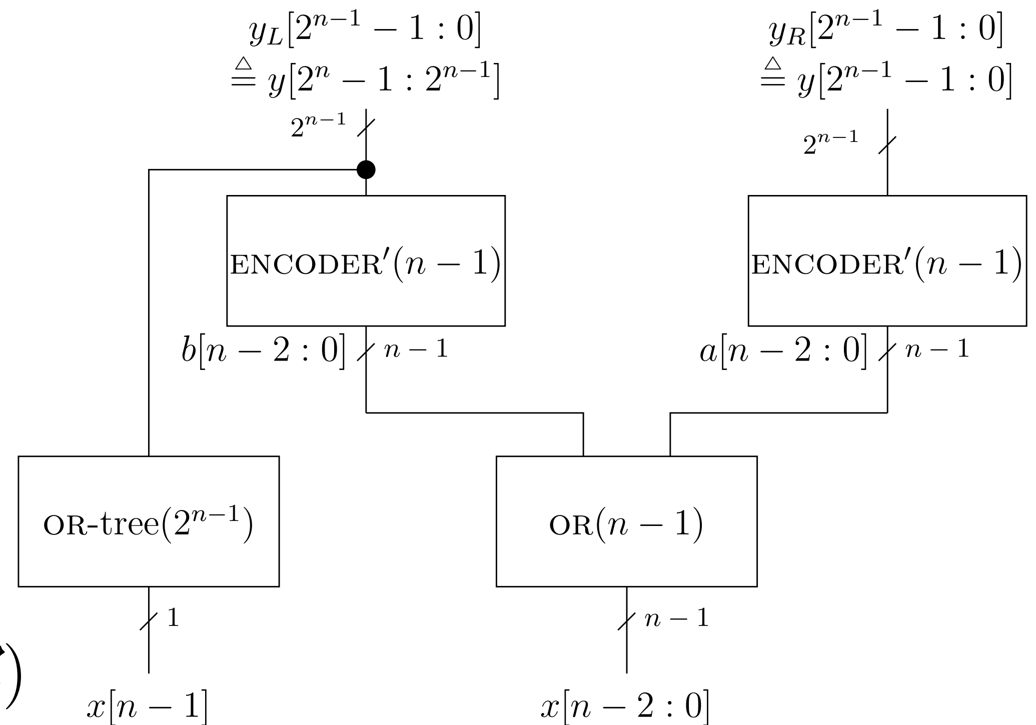
Correctness of ENCODER$'(n)$ - induction step

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

$$\vec{x} = \text{OR-tree}(\vec{y}_L) \cdot \text{OR}(\vec{b}, \vec{a})$$
$$= 0 \cdot \vec{a}.$$



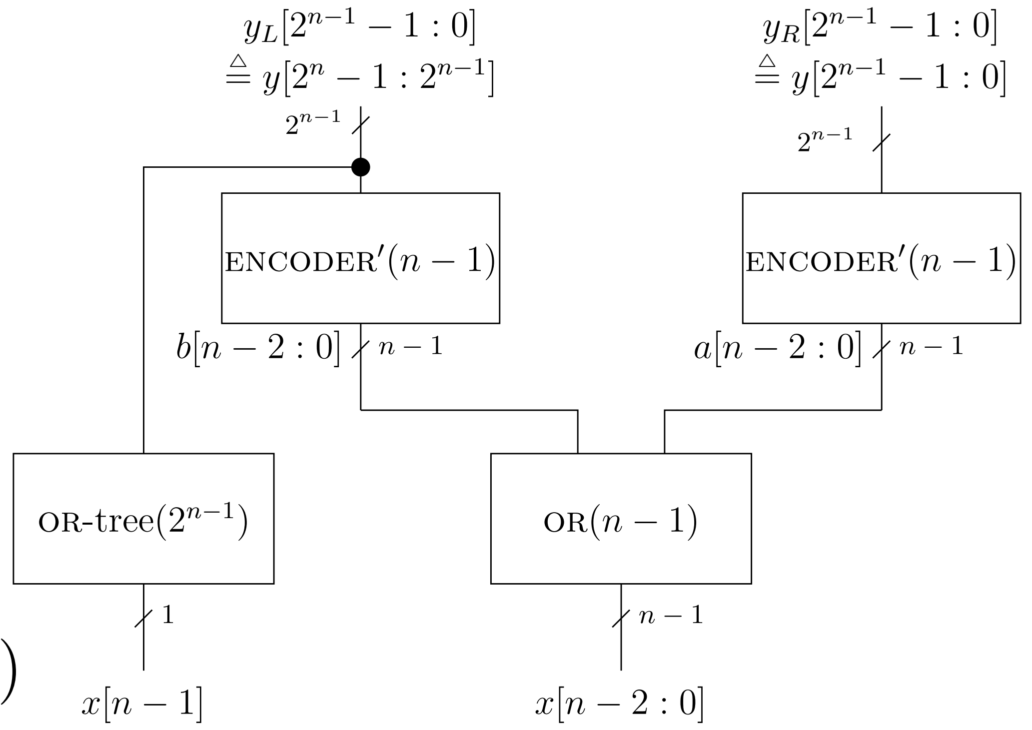
Correctness of ENCODER$'(n)$ - induction step - cont.

(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

$$\vec{x} = \text{OR-tree}(\vec{y}_L) \cdot \text{OR}(\vec{b}, \vec{a})$$

$$= 1 \cdot \vec{b}.$$

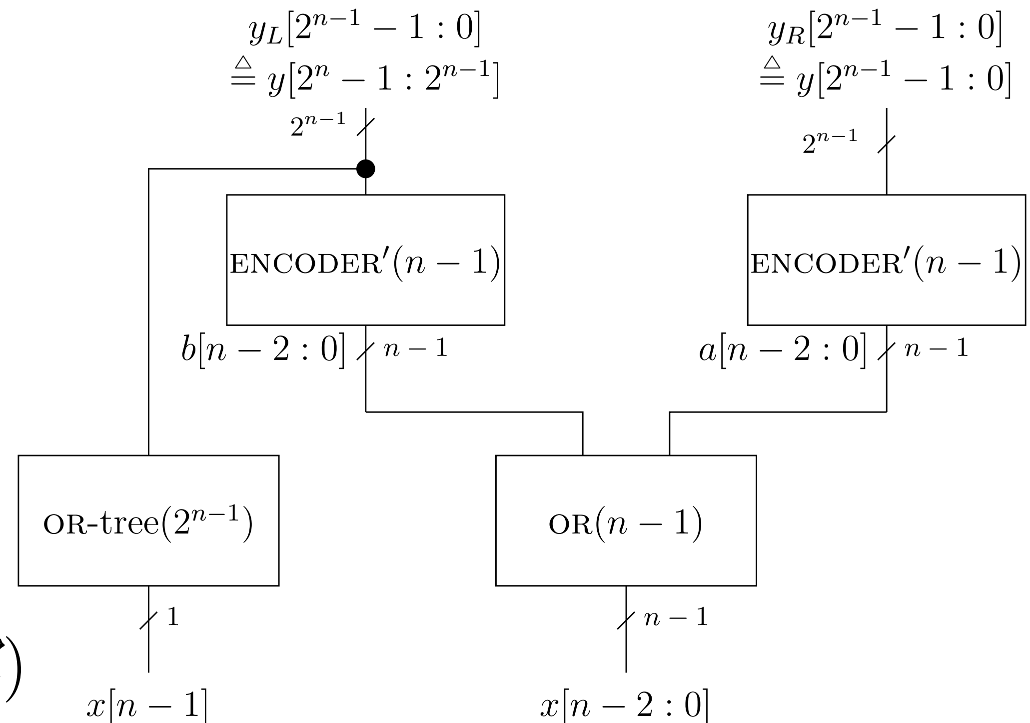


Correctness of ENCODER$'(n)$ - induction step - cont.

(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 $\textsf{ENCODER}'(n)$

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

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

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

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 $O(n)$.

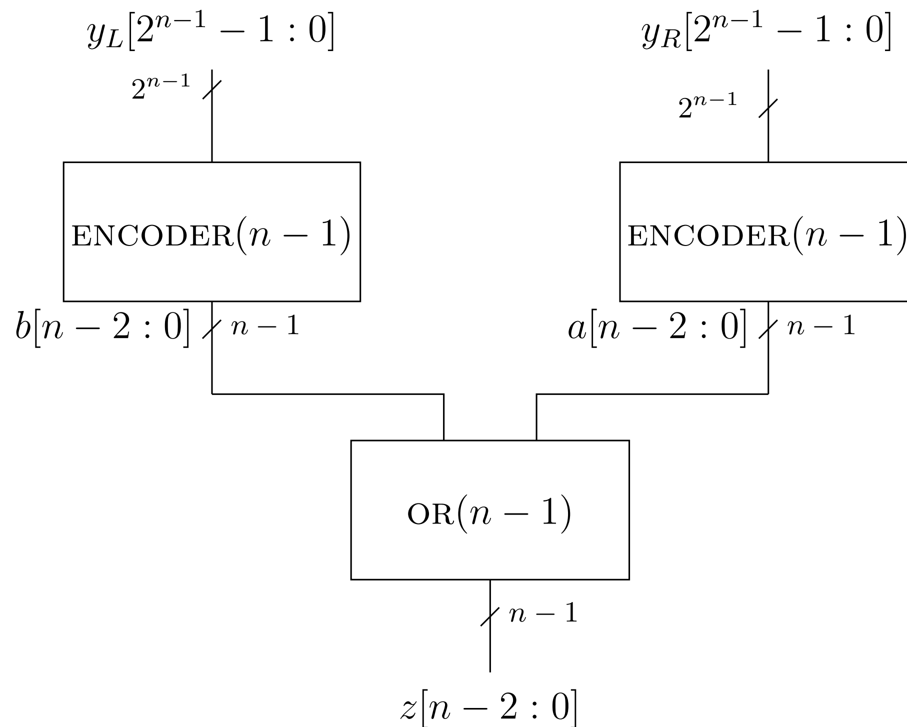
$\Rightarrow$ a trivial design with delay $O(n)$ and cost $O(n \cdot 2^n)$.

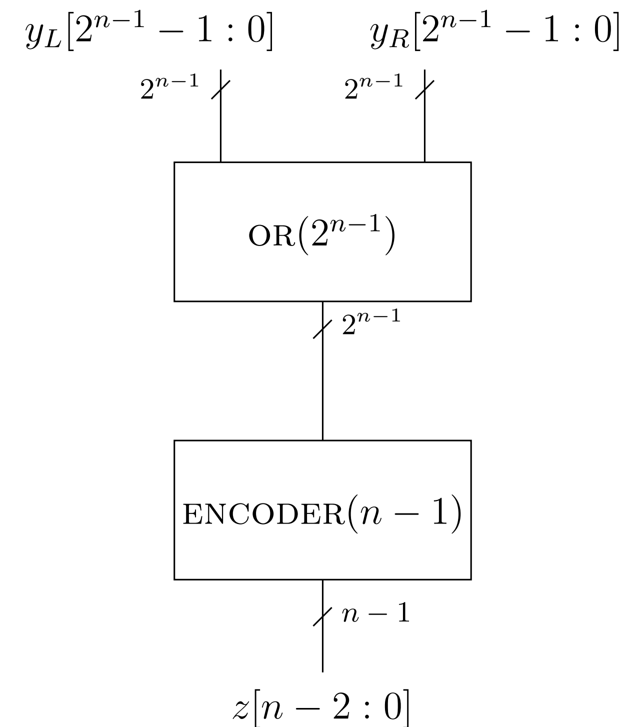
We conclude that $\text{ENCODER}'(n)$ is not a good design... Can we fix it?!

Commuting bitwise-OR and ENCODER$_{n-1}$

claim: If $wt(y[2^n - 1 : 0]) \leq 1$, then

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





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

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

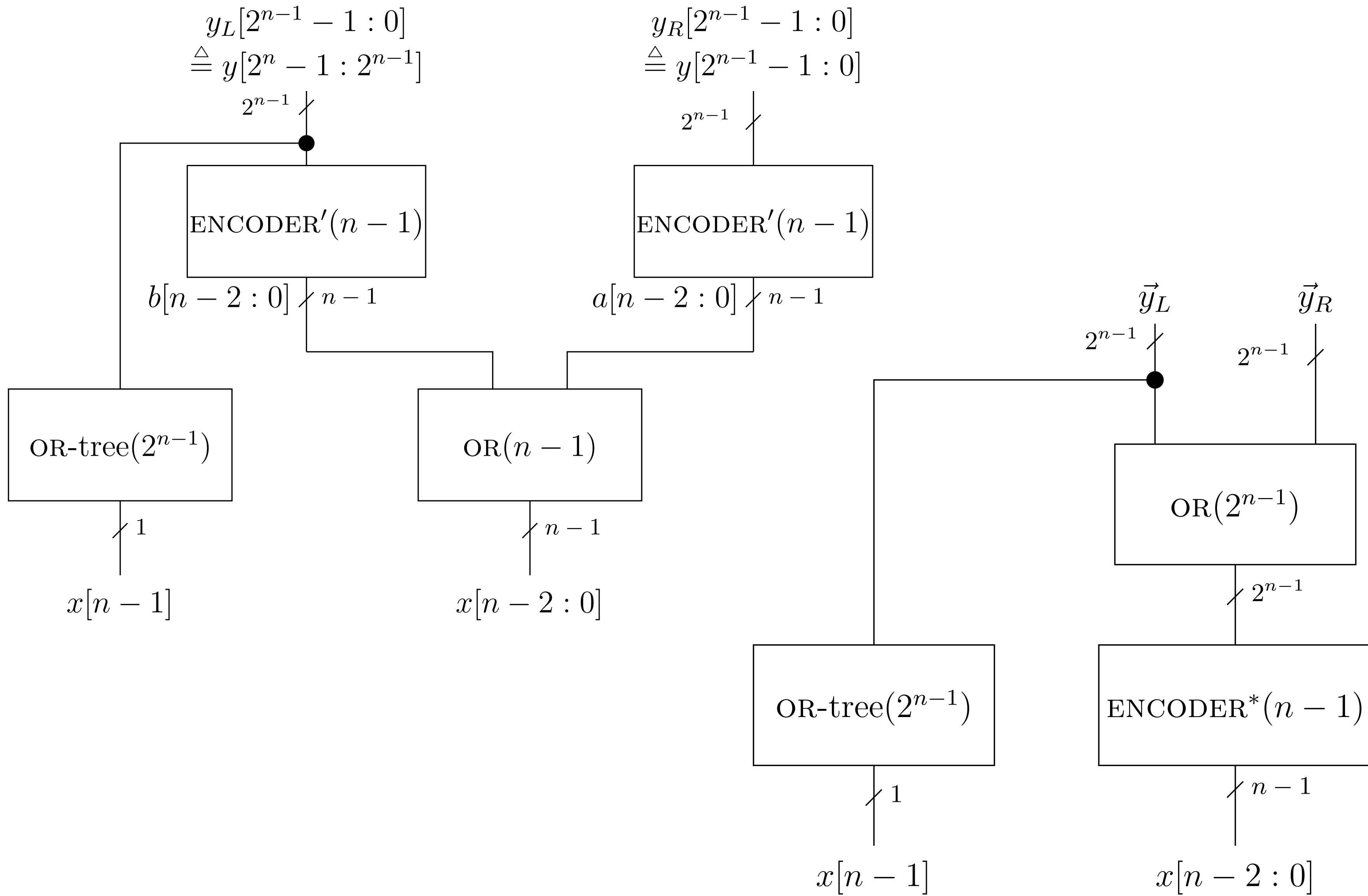
$$\begin{aligned} E_{n-1}(\textsf{OR}(\vec{y}_L, \vec{y}_R)) &= E_{n-1}(\textsf{OR}(0^{2^{n-1}}, \vec{y}_R)) \\ &= E_{n-1}(\vec{y}_R)). \end{aligned}$$

  - However,

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

- (other case is analogous) QED

$\mathsf{ENCODER}'(n) \longrightarrow \mathsf{ENCODER}^*(n)$



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 $\textsf{ENCODER}^*(n)$

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

We expand this recurrence to obtain:

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

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

The delay of $\text{ENCODER}^*(n)$ 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 $\text{ENCODER}^*(n)$ are asymptotically optimal.

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 and Shifters

Computer Structure - Spring 2007

©Dr. Guy Even

Tel-Aviv Univ.

Preliminary questions

- Which types of shifts are you familiar with in your favorite programming language? What is the differences between these shifts? Why do we need different types of shifts?
- How are these shifts executed in a microprocessor?
- Should shifters be considered to be combinational circuits? After all, they simply “move bits around” and do not compute “new bits”.

Goals

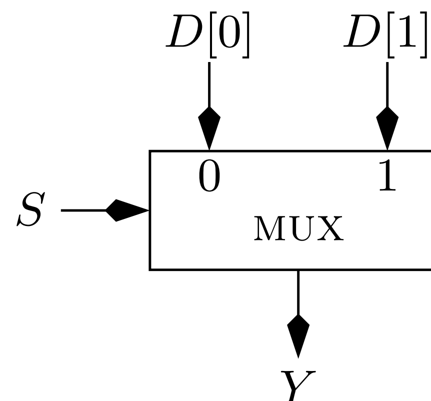
- Selectors:
  - review definition of multiplexer.
  - build $(n:1)$-multiplexers.
- Shifters:
  - Cyclic shifter (Barrel shifter)
  - Logical Shifter
  - Arithmetic Shifter

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 0.

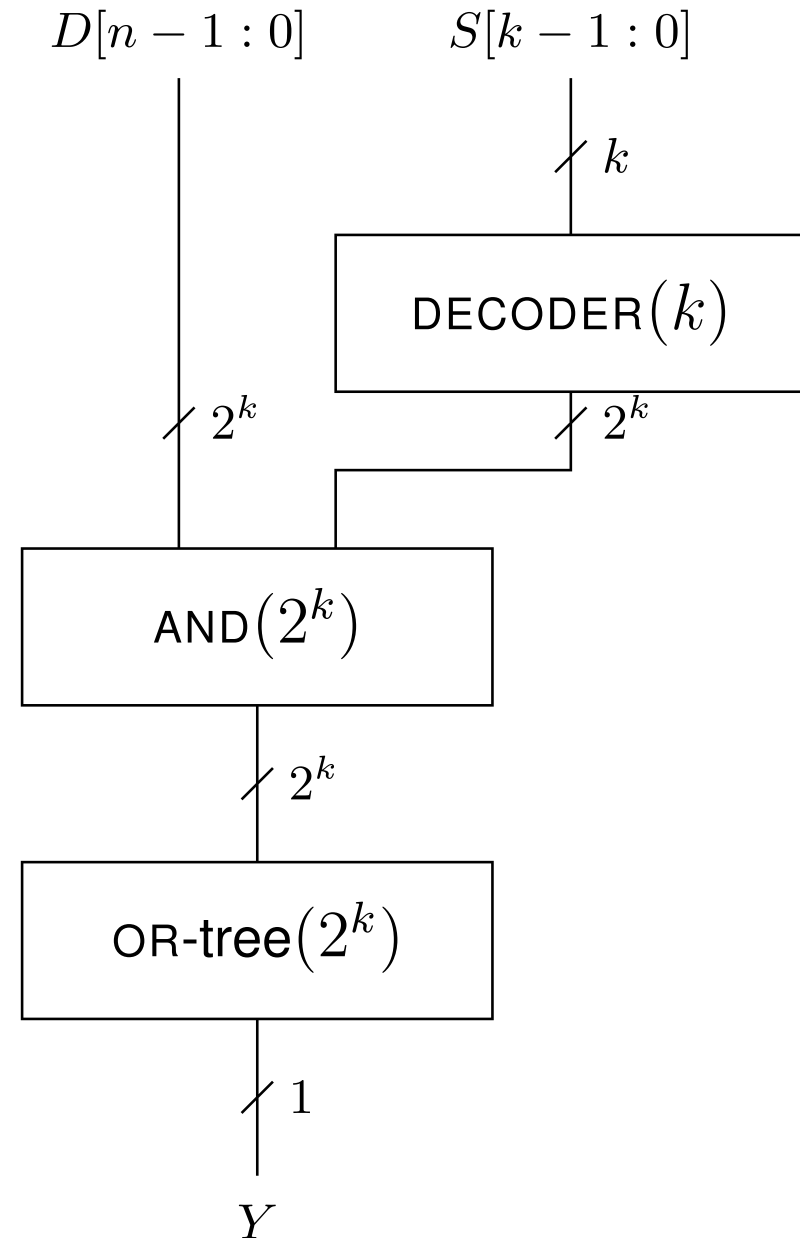
- $\vec{D}$ - data input
- $\vec{S}$ - select input
- simplify: assume that n is a power of 2, namely, $n = 2^k$.

Implementation of (n:1)-MUX

We will present two implementations:

- a decoder based implementation
- a tree-like implementation

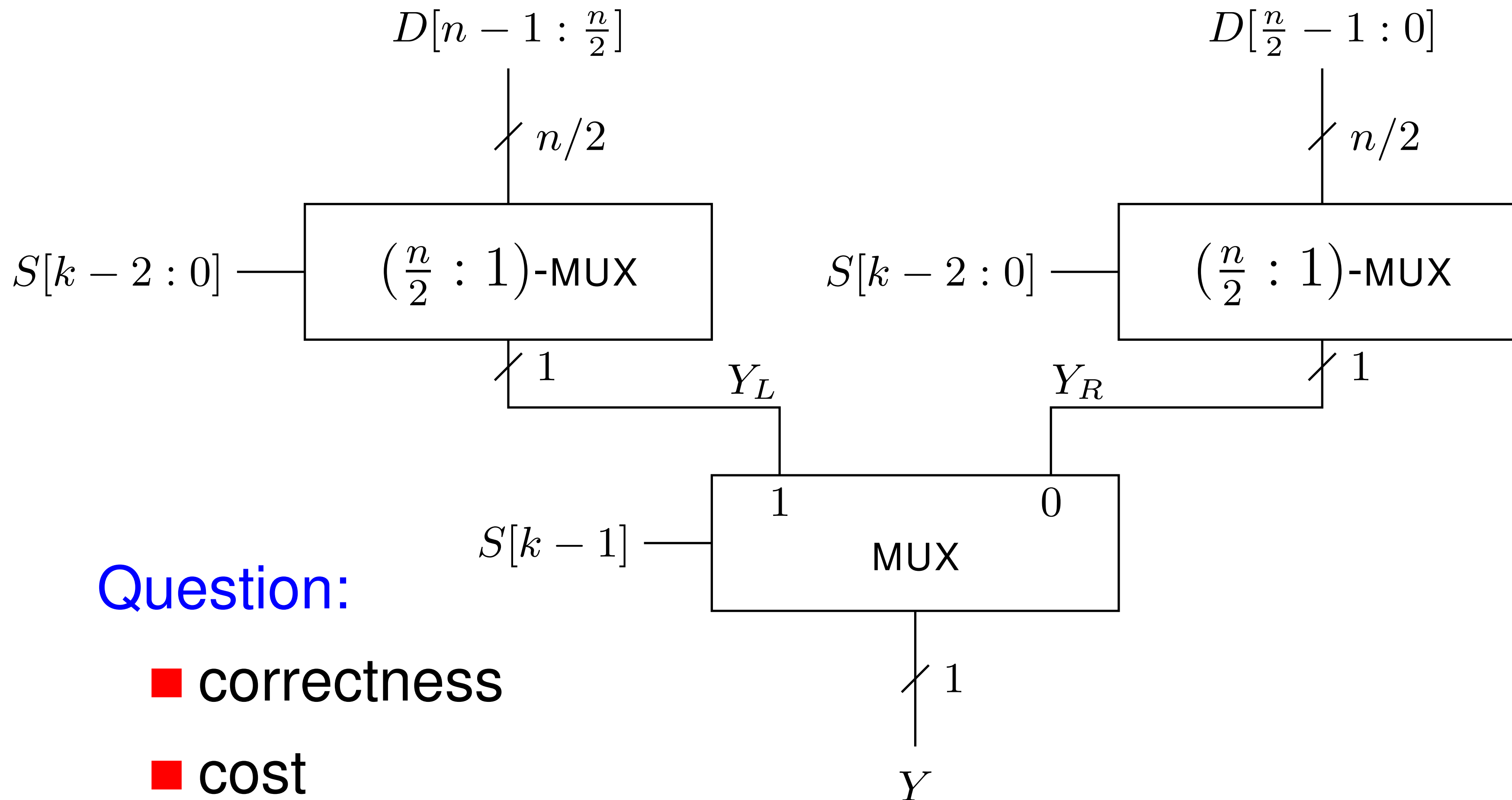
(n:1)-MUX : a decoder based implementation



Question:

- correctness
- cost
- delay
- asymptotic optimality

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



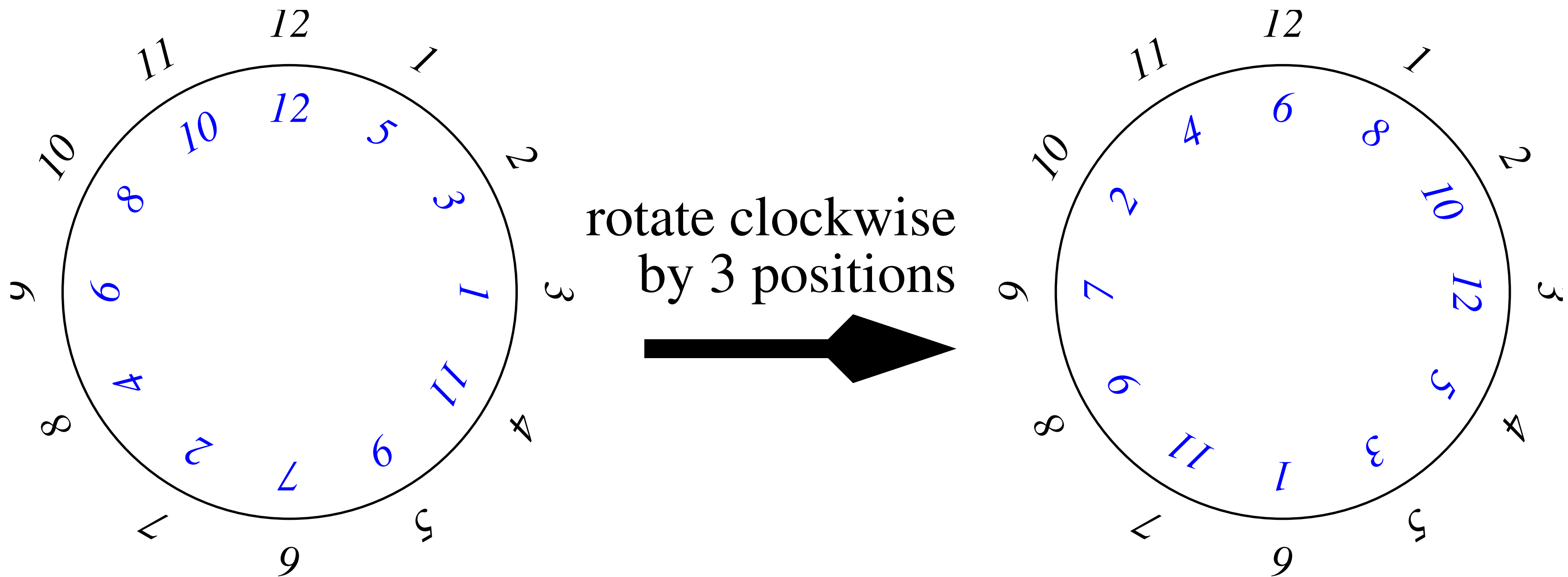
Question:

- 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, $n = 2^k$.

$\textsc{cls}(n, i)$ - Cyclic Left Shift by 2^i positions

DEF: A $\textsc{cls}(n, i)$ is a combinational circuit defined as follows:

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

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

Functionality:

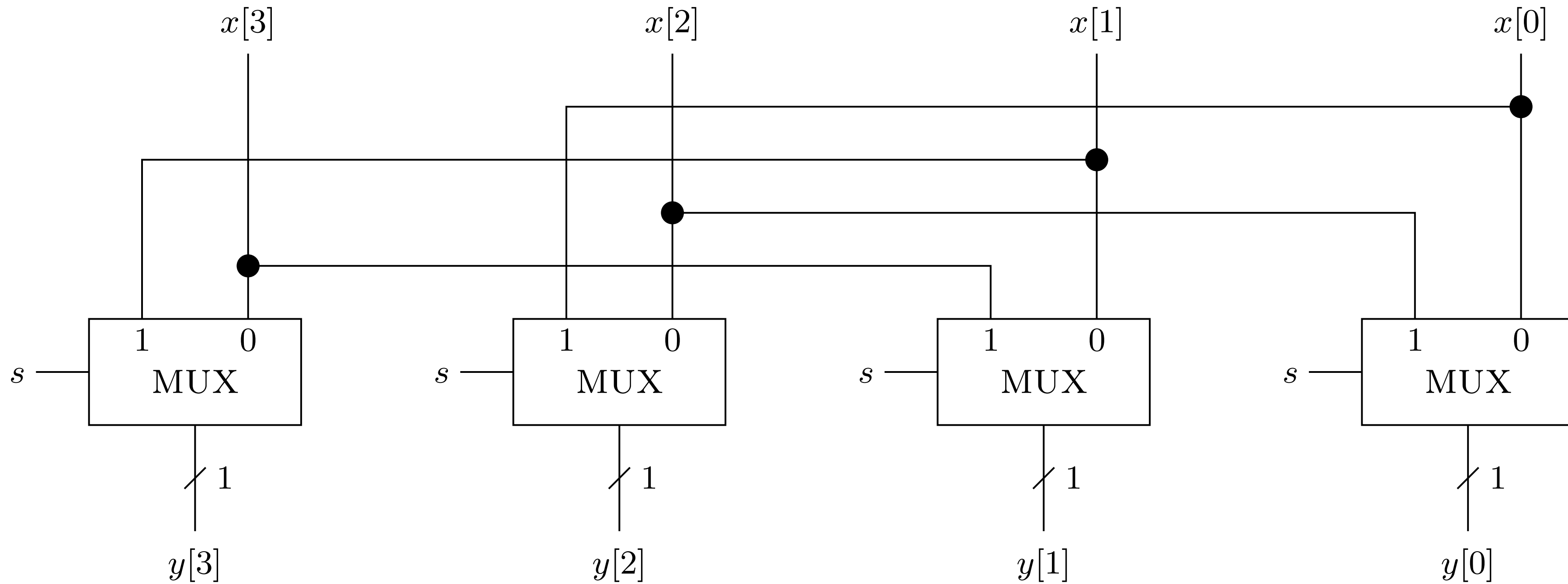
$$\forall j \in [n-1:0]: \quad y[j] = x[\text{mod}(j - s \cdot 2^i, n)].$$

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 $\textsc{cls}(n, i)$ with a row of n $\textsc{mux}$-gates.

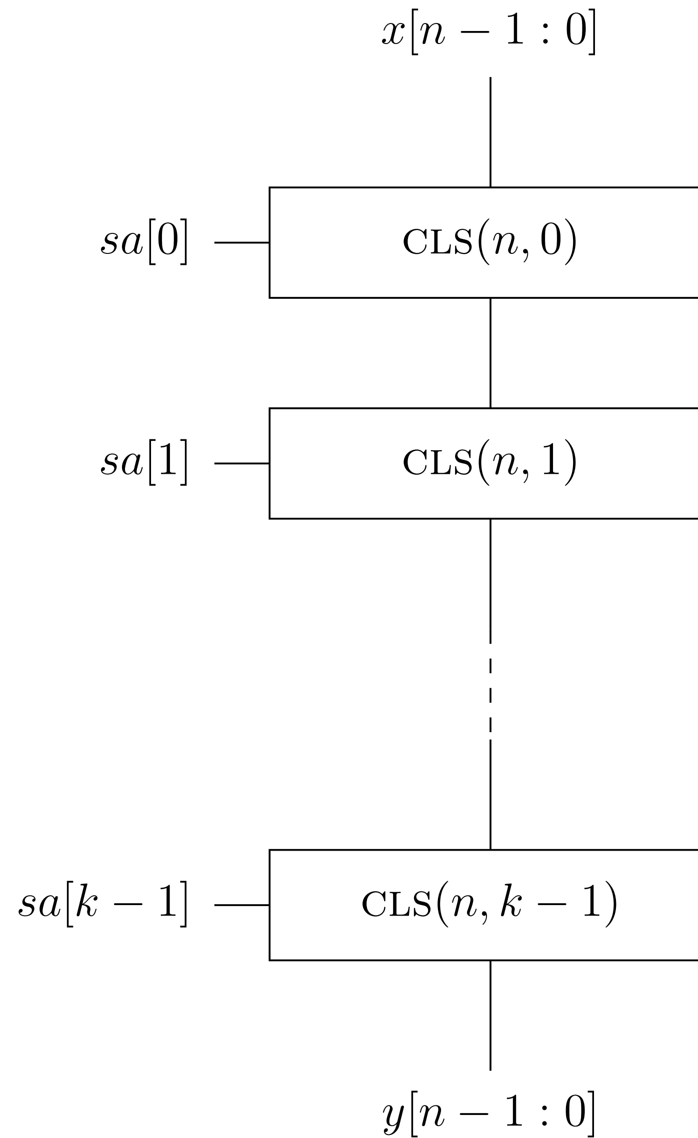
$\textsc{cls}(4, 1)$



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

Our model does not capture routing cost.

BARREL-SHIFTER(n) - a chain of CLS(n, i)



BARREL-SHIFTER(n) - correctness

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

$$y_0[n-1:0] \leftarrow \text{CLS}_{n,0}(x[n-1,0], sa[0])$$

$$y_{i+1}[n-1:0] \leftarrow \text{CLS}_{n,i+1}(y_i[n-1,0], sa[i+1])$$

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 CLS$(n,0)$ shifts by zero/one position.

induction step

-

$$
\begin{aligned}
y_i[j] &= \text{CLS}_{n,i}(y_{i-1}[n-1,0], sa[i])[j] && \text{(by definition of } y_i\text{)} \\
&= y_{i-1}[\text{mod}(j - 2^i \cdot sa[i], n)] && \text{(by definition of CLS}_{n,i}\text{).}
\end{aligned}
$$

- 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.

Logical Shifting - motivation

- Used for shifting binary strings that represent unsigned integers in binary representation.
- Shifting to the left by s positions corresponds to

$$\langle\vec{y}\rangle \leftarrow \text{mod}(\langle\vec{x}\rangle \cdot 2^s, 2^n).$$

- Shifting to the right by s positions corresponds to

$$\langle\vec{y}\rangle \leftarrow \left\lfloor \frac{\langle\vec{x}\rangle}{2^s} \right\rfloor.$$

Bi-Directional Logical Shifter - definition

A LOG-SHIFT(n) is a combinational circuit defined as follows:

Input:

- $x[n-1:0] \in \{0,1\}^n$,
- $sa[k-1:0] \in \{0,1\}^k$, where $k = \lceil \log_2 n \rceil$, and
- $\ell \in \{0,1\}$.

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

Functionality: If $\ell = 1$, then logical left shift as follows:

$$y[n-1:0] \triangleq x[n-1-\langle \vec{sa} \rangle : 0] \cdot 0^{\langle \vec{sa} \rangle}.$$

If $\ell = 0$, then logical right shift as follows:

$$y[n-1:0] \triangleq 0^{\langle \vec{sa} \rangle} \cdot x[n-1 : \langle \vec{sa} \rangle].$$

Bi-Directional Logical Shifter - example

Example: Let $x[3:0] = 0010$. If $sa[1:0] = 10$ and $\ell = 1$, then LOG-SHIFT(4) outputs $y[3:0] = 1000$. If $\ell = 0$, then the output equals $y[3:0] = 0000$.

Bi-Directional Logical Shifter - implementation

- As in the case of cyclic shifters, we break the task of designing a logical shifter into sub-tasks of logical shifts by powers of two.
- Loosely speaking, an LBS(n, i) is a logical bi-directional shifter that outputs one of three possible strings:
  - the input shifted to the left by 2^i positions,
  - the input shifted to the right by 2^i positions, or
  - the input without shifting.

  We now formally define this circuit....

LBS(n,i) - definition

DEF: An LBS(n,i) is a combinational circuit defined as follows:

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

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

Functionality: Define $x'[n-1+2^i:-2^i] \in \{0,1\}^{n+2\cdot 2^i}$ as follows:

$$x'[j] \triangleq \begin{cases} x[j] & \text{if } n > j \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

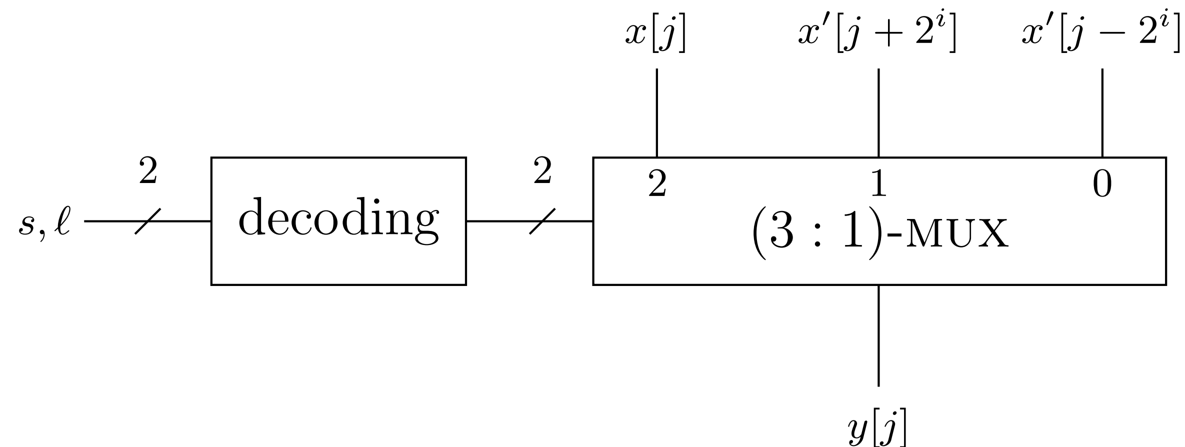
The value of the output $y[n-1:0]$ is specified by

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

$$y[j] = x'[j + (-1)^\ell \cdot s \cdot 2^i]$$

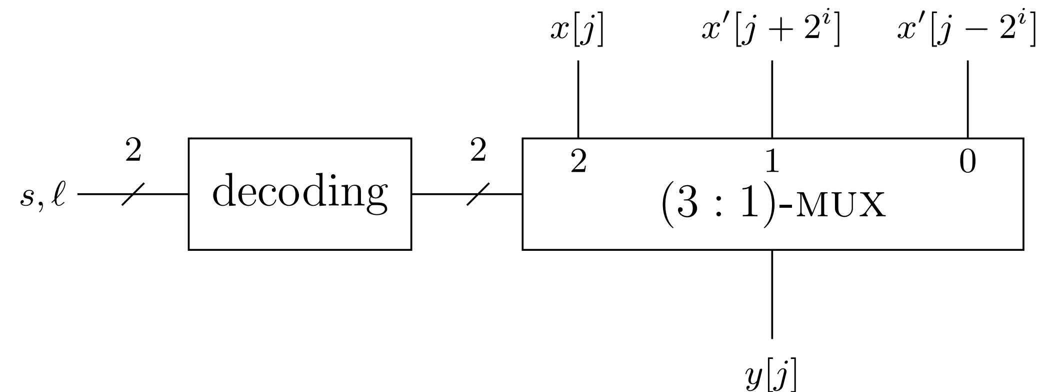
- $x'[n - 1 + 2^i : -2^i] = 0^{2^i} \cdot x[n - 1 : 0] \cdot 0^{2^i}$.
- ℓ - determines if the shift is a left shift or a right shift. If $\ell = 1$ then $(-1)^\ell = -1$, and the shift is a left shift (since increasing indexes from $j - 2^i$ to j has the effect of a left shift).
- s - determines if a shift (in either direction) takes place at all. If $s = 0$, then $y[j] = x[j]$, and no shift takes place.

A bit-slice of an implementation of LBS(n, i)



1. $(3 : 1)$-MUX. Implemented either by a “pruned” tree-like construction or we can simply consider a $(3 : 1)$-MUX as a basic gate. Simple circuit $\Rightarrow$ best option can be easily determined based on the technology at hand.
2. decoding circuit - not a decoder! Decoding of s and ℓ causes the $(3 : 1)$-MUX to select the correct input.

A bit-slice of an implementation of LBS(n, i)



Question: This question deals with various aspects and details concerning the design of a logical shifter.

1. Design a "pruned" tree-like $(3:1)$-MUX.
2. Design the decoding box.
3. Show how LBS(n, i) circuits can be cascaded to obtain a LOG-SHIFT(n).
   Hint: follow the design of a BARREL-SHIFTER(n).

Arithmetic Shifters - motivation

- Used for shifting binary strings that represent signed integers in two's complement representation.
- logical left shifting $=$ arithmetic left shifting.
- Arithmetic right shifting corresponds to dividing by a power of 2 (with sign extension).

Arithmetic right shifter - definition

DEF: An ARITH-SHIFT(n) is a combinational circuit defined as follows:

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

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

Functionality: The output $\vec{y}$ is a (sign-extended) arithmetic right shift of $\vec{x}$ by $\langle \vec{sa} \rangle$ positions. Formally,

$$y[n-1:0] \stackrel{\triangle}{=} x[n-1]^{\langle \vec{sa} \rangle} \cdot x[n-1 : \langle \vec{sa} \rangle].$$

Example: Let $x[3:0] = 1001$. If $sa[1:0] = 10$, then ARITH-SHIFT(4) outputs $y[3:0] = 1110$.

Arithmetic right shifter - implementation

Question: Consider the definitions of $\text{CLS}(n, i)$ and $\text{LBS}(n, i)$. Suggest an analogous definition $\text{ARS}(n, i)$ for arithmetic right shift (i.e., modify the definition of $\vec{x}'$ and use $(2:1)$-MUXs). Suggest an implementation of an arithmetic right shifter based on cascading $\text{ARS}(n.i)$ circuits.

Further questions

Question: Design a bi-directional cyclic shifter. Such a shifter is like a cyclic left shifter but has an additional input $\ell \in \{0,1\}$ that indicates the direction of the required shift. Hint: Consider reducing a cyclic right shift to a cyclic left shifter. To simplify the reduction you may assume that $n = 2^k - 1$ (hint: use one's complement negation). Suggest a simple reduction in case $n = 2^k$ (hint: avoid explicit subtraction!).

Further questions - cont.

Question: CPUs often support all three types of shifting: cyclic, logical, and arithmetic shifting.

1. Write a complete specification of a shifter that can perform all three types of shifts.
2. Propose an implementation of such a shifter.

Summary

- $(n:1)$-multiplexers:
  - definition.
  - two implementations: decoder based & tree-like.
  - both designs are optimal.
- three types of shifts: cyclic, logical, and arithmetic shifts.
- Design method: cascade a logarithmic number of shifters (with parameter i) that either perform a shift by 2^i positions or no shift at all.

Chapter 6: Priority Encoders

Computer Structure - Spring 2007

©Dr. Guy Even

Tel-Aviv Univ.

Preliminary questions

- Suppose that many devices wish to transmit data along a shared bus. How does the bus controller decide which device gets to use the bus?
- Consider the binary fraction 0.000010101. In many cases, an arithmetic unit is supposed to shift the fraction so that its value is in the range $[1/2, 1)$ (this is often called *normalization*). How is the shift amount computed?

Leading One

Consider a binary string $x[0:n-1]$ (ascending indexes!).
DEF: The leading one of a binary string $x[0:n-1]$ is defined by

$$\text{LEADING-ONE}(x[0:n-1]) \triangleq \begin{cases} \min\{i \mid x[i]=1\} & \text{if } x[0:n-1] \neq 0^n \\ n & \text{otherwise.} \end{cases}$$

Example: Consider the string $x[0:6] = 0110100$. The leading one is the index $[1]$. Note that indexes are in ascending order and that $x[0]$ is the leftmost bit.
Claim: For every binary string $x[n-1:0]$

$$\text{LEADING-ONE}(\vec{a}) = \text{LEADING-ONE}(\vec{a} \cdot 1).$$

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: The binary string 01001011 does not represent a number in unary representation. Only a string that is obtained by concatenating an all-ones string with an all-zeros string represents a number in unary representation.

Parallel Prefix Computation

DEF: A parallel prefix computation OR circuit of length n is a combinational circuit specified as follows.

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

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

Functionality:

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

We denote parallel prefix computation OR circuit of length n by PPC–OR(n).

Priority Encoders

- A priority encoder is a combinational circuit that computes the leading one.
- We consider two types of priority encoders:
  - A unary priority encoder - outputs the leading one in unary representation.
  - A binary priority encoder -outputs the leading one in binary representation.

Unary priority encoder

DEF: A unary priority encoder U-PENC(n) is a combinational circuit specified as follows.

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

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

Functionality:

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

Example:

- If $x[0 : 6] = 0110100$, then PPC–OR(7) outputs 0111111. U-PENC(7) outputs 1000000.
- If $\vec{x} \neq 0^n$, then U-PENC(n) outputs $y[0 : n-1] = 1^j \cdot 0^{n-j}$, where $j = \min\{i \mid x[i] = 1\}$.
- If $\vec{x} = 0^n$, then $\vec{y} = 1^n$ and $\vec{y}$ is a unary representation of n.

Binary Priority Encoder

DEF: A binary priority encoder B-PENC(n) is 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:

$$\langle \vec{y} \rangle = \text{LEADING-ONE}(\vec{x})$$

$n = 2^k \implies$ length of the output of a B-PENC(n) is $k+1$ bits; otherwise, the number n could not be represented by the output.

Example: Given input $x[0:5] = 000101$, a U-PENC(6) outputs $y[0:5] = 111000$, and B-PENC(6) outputs $y[2:0] = 011$.

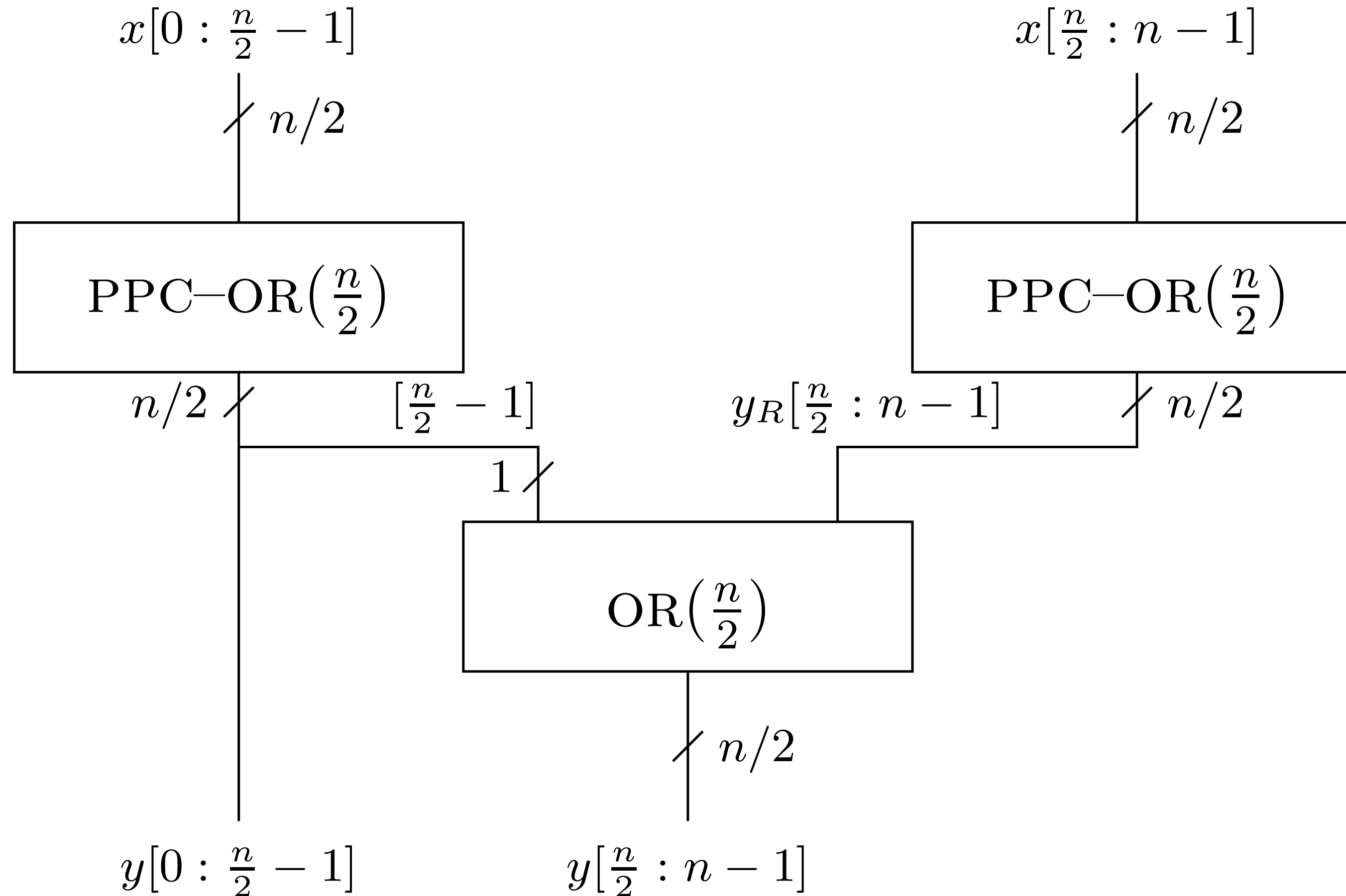
U-PENC(n) - Implementation

- Invert the outputs of a PPC–OR(n).
- Brute force design of PPC–OR(n):
  - separate OR-tree for each output bit.
  - delay = $O(\log n)$
  - cost = $O(n^2)$.
- How can we efficiently combine these trees? To be discussed in detail when we discuss fast addition.
- We now present a (non-optimal) design based on divide-and-conquer.

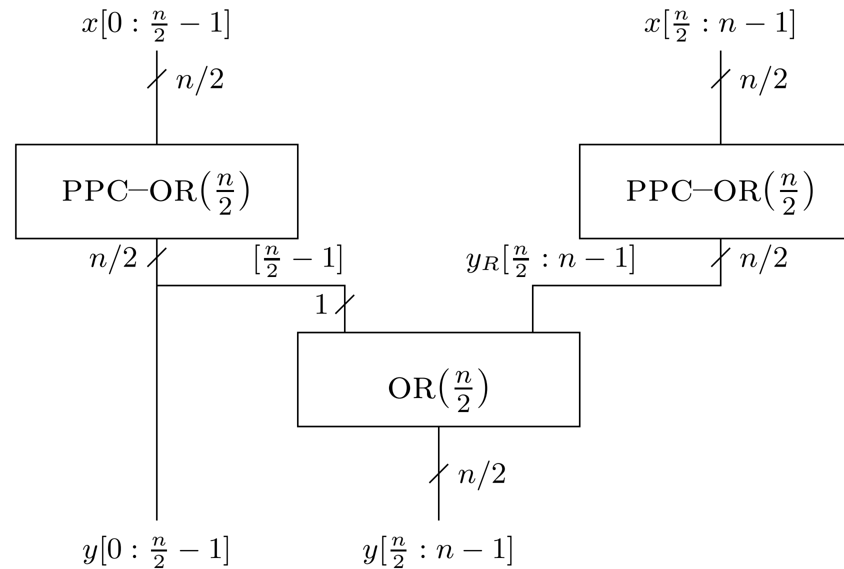
divide & conquer PPC–OR(n)

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

If $n > 1$:



divide & conquer PPC–OR(n) - cont.



Question:

1. Prove the correctness of the design.
2. Extend design for values of n that are not powers of 2.
3. Analyze the delay and cost of the design.
4. Prove the asymptotic optimality of the delay of the design.

divide & conquer PPC–OR(n) - cost analysis

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

It follows that

$$\begin{aligned} c(n) &= 2 \cdot c(\frac{n}{2}) + \Theta(n) \\ &= \Theta(n \cdot \log n). \end{aligned}$$

Question: Prove a lower bound on $c(\text{PPC–OR}(n))$.

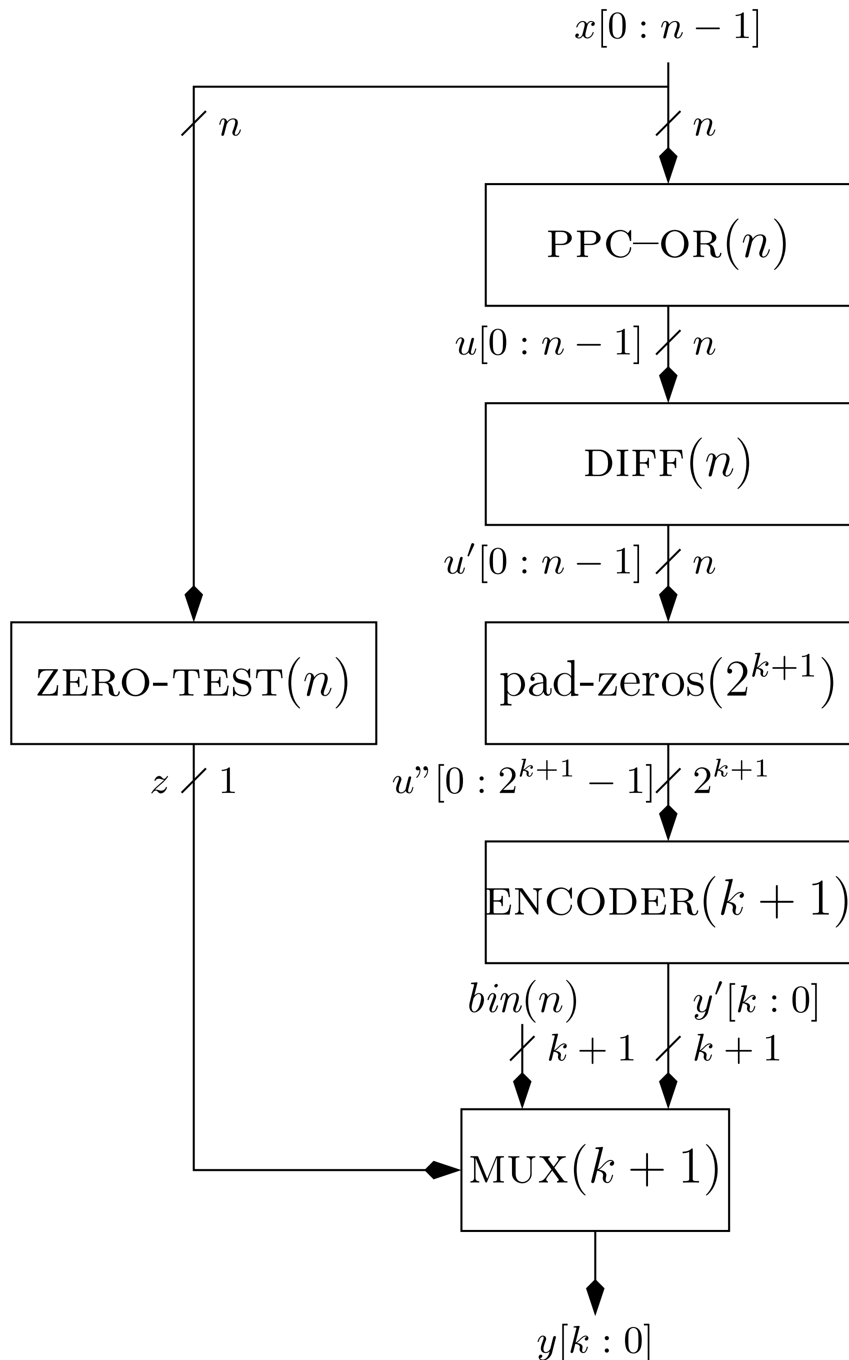
Promise: In the chapter on fast addition we will present a cheaper implementation of PPC–OR(n) (with logarithmic delay).

Implementation of a binary priority encoder

We present two designs for a binary priority encoder.

- design based on a reduction to PPC–OR(n).
- design based on divide-and-conquer

reduction of B-PENC(n) to PPC–OR(n)



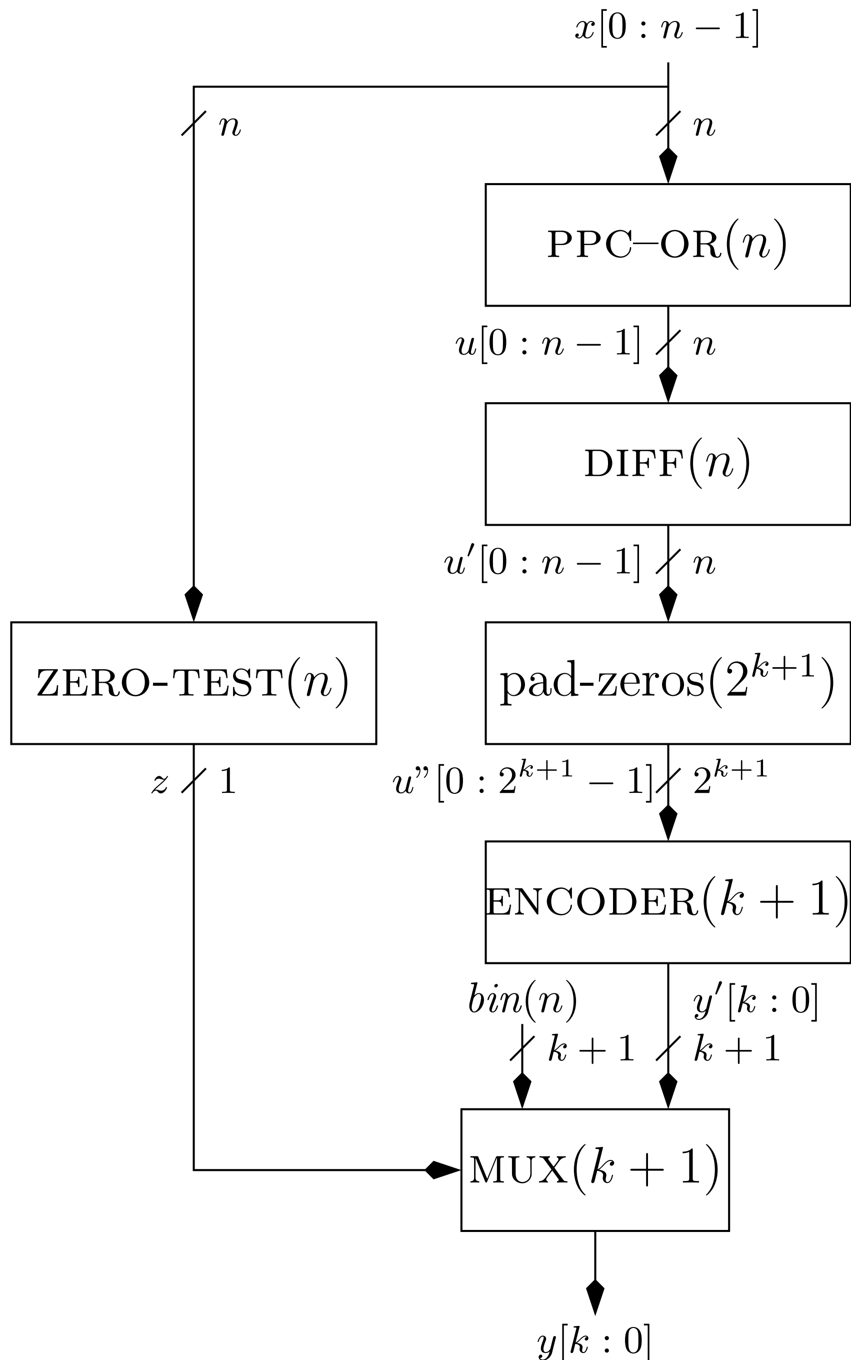
- apply PPC–OR(n)
- difference:

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

  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

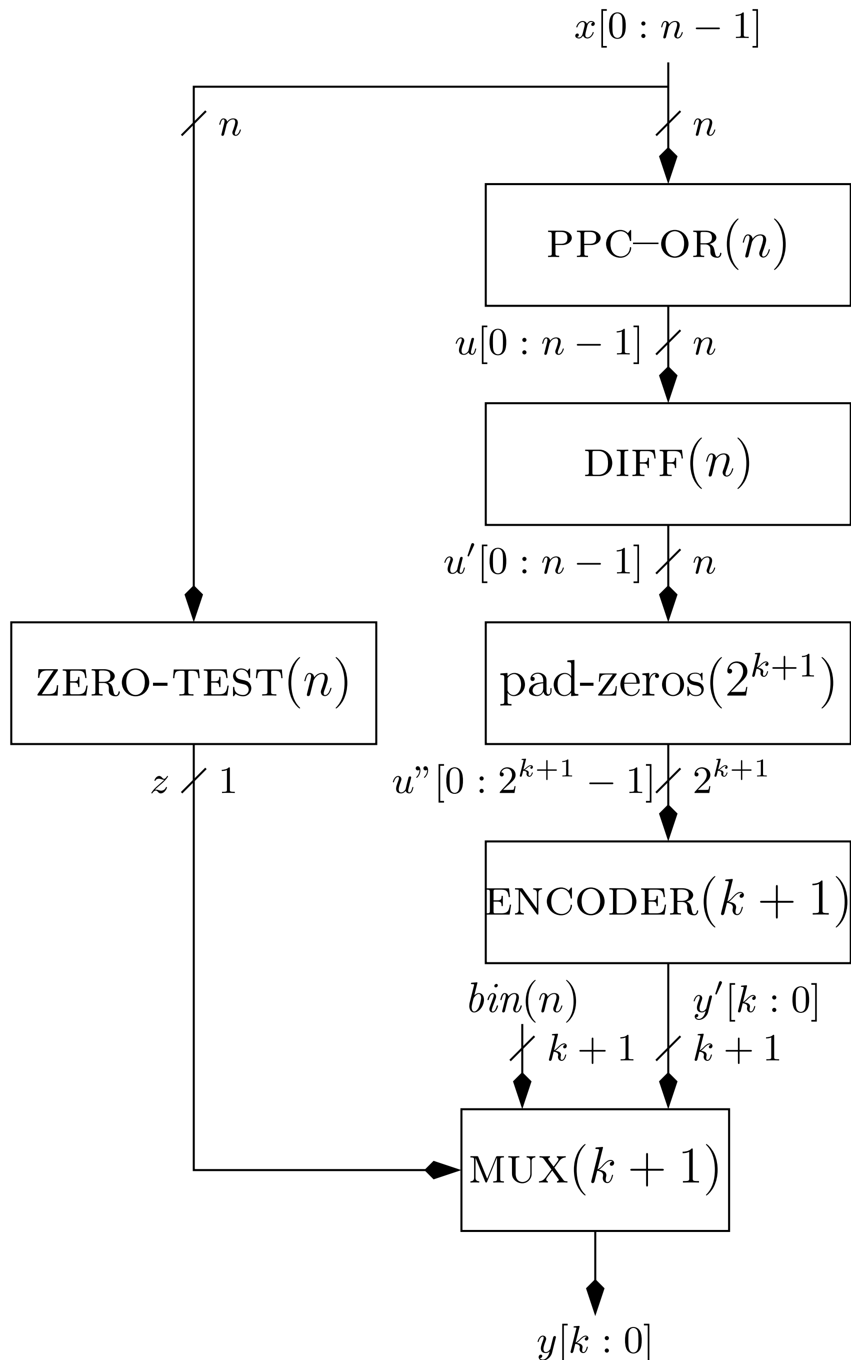


- zero cost: padding with zeros.
- logarithmic cost: multiplexer
- linear cost:
  - difference
  - encoder
  - zero tester
- $\Longrightarrow$

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

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

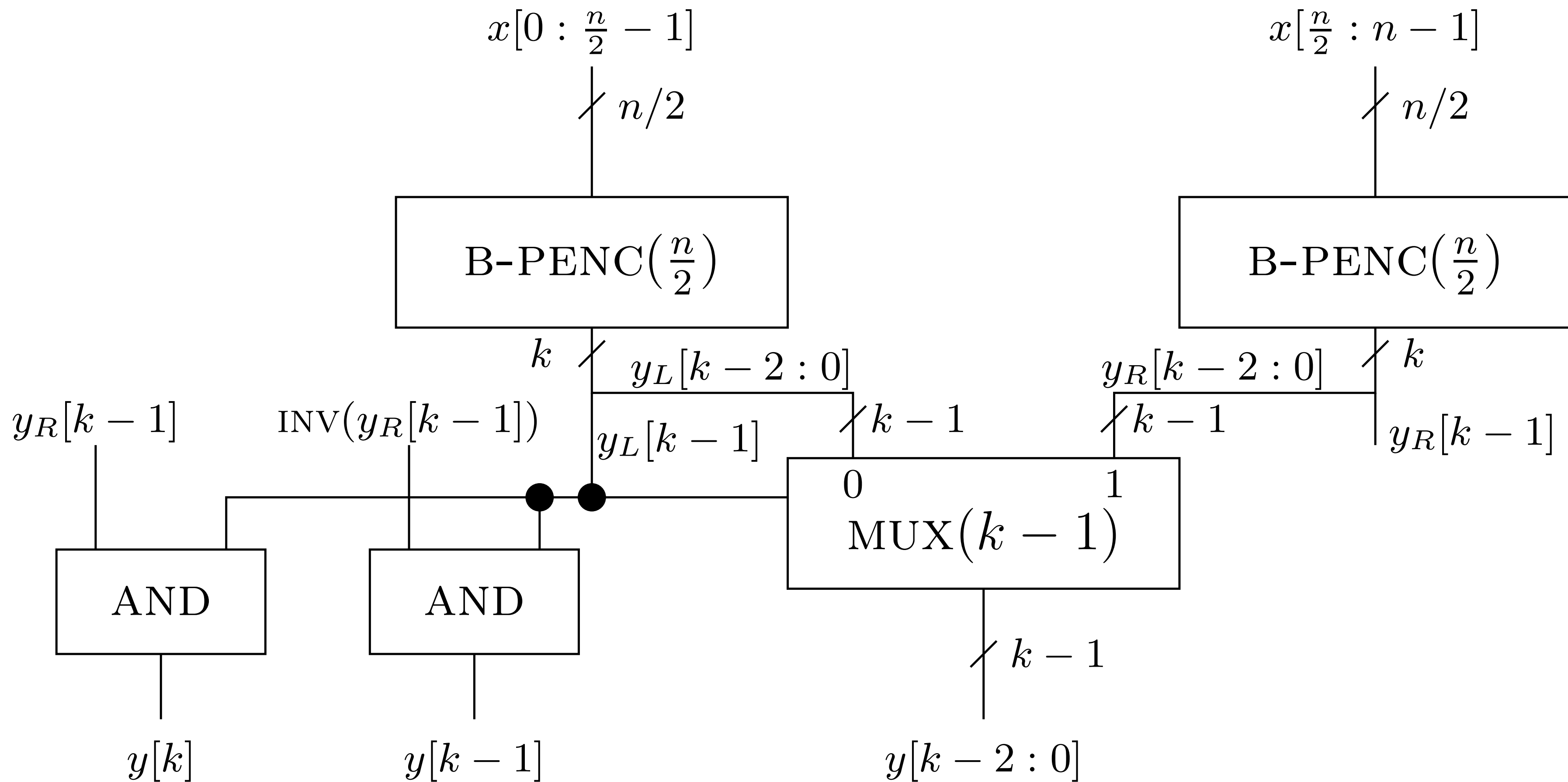
delay analysis



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

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

B-PENC(n): a divide-and-conquer design for $n = 2^k$

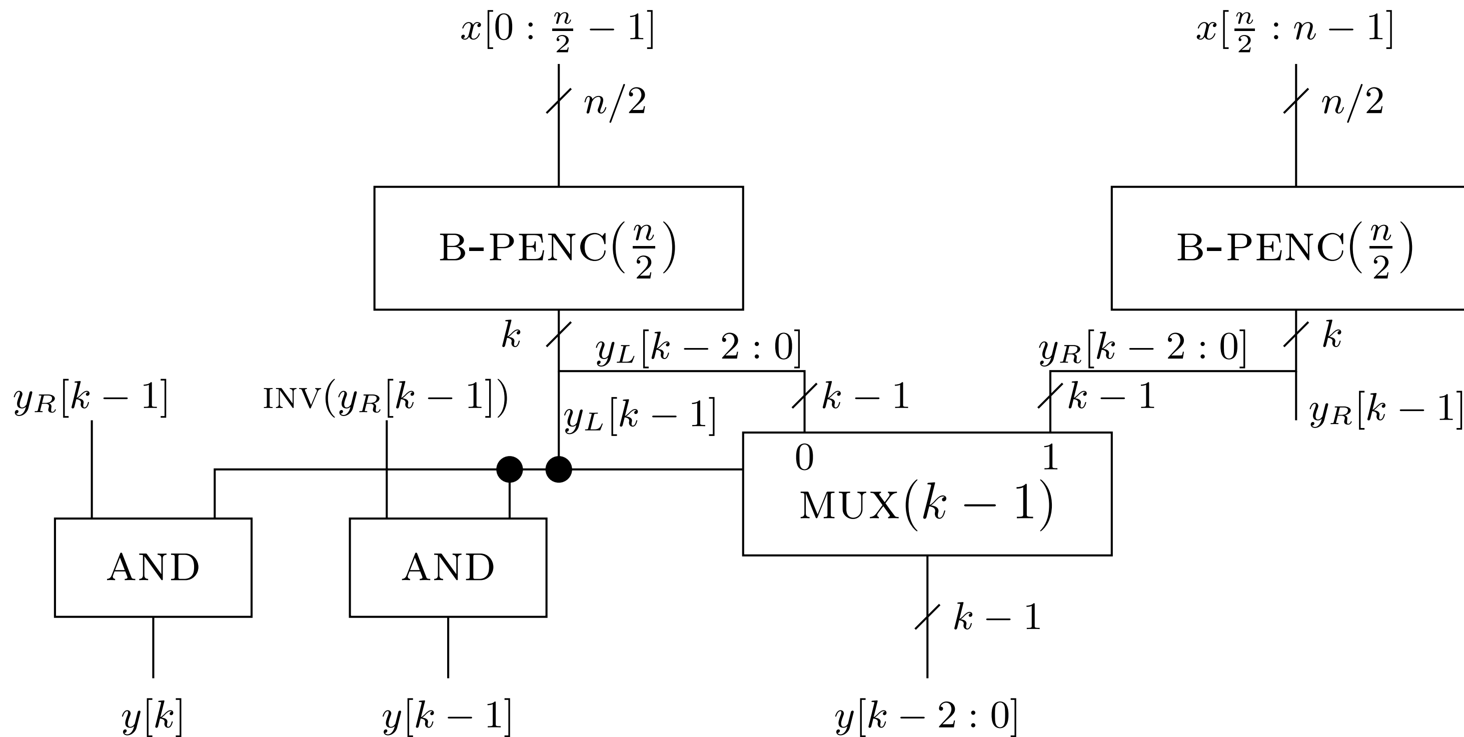


correctness

We prove correctness by induction.

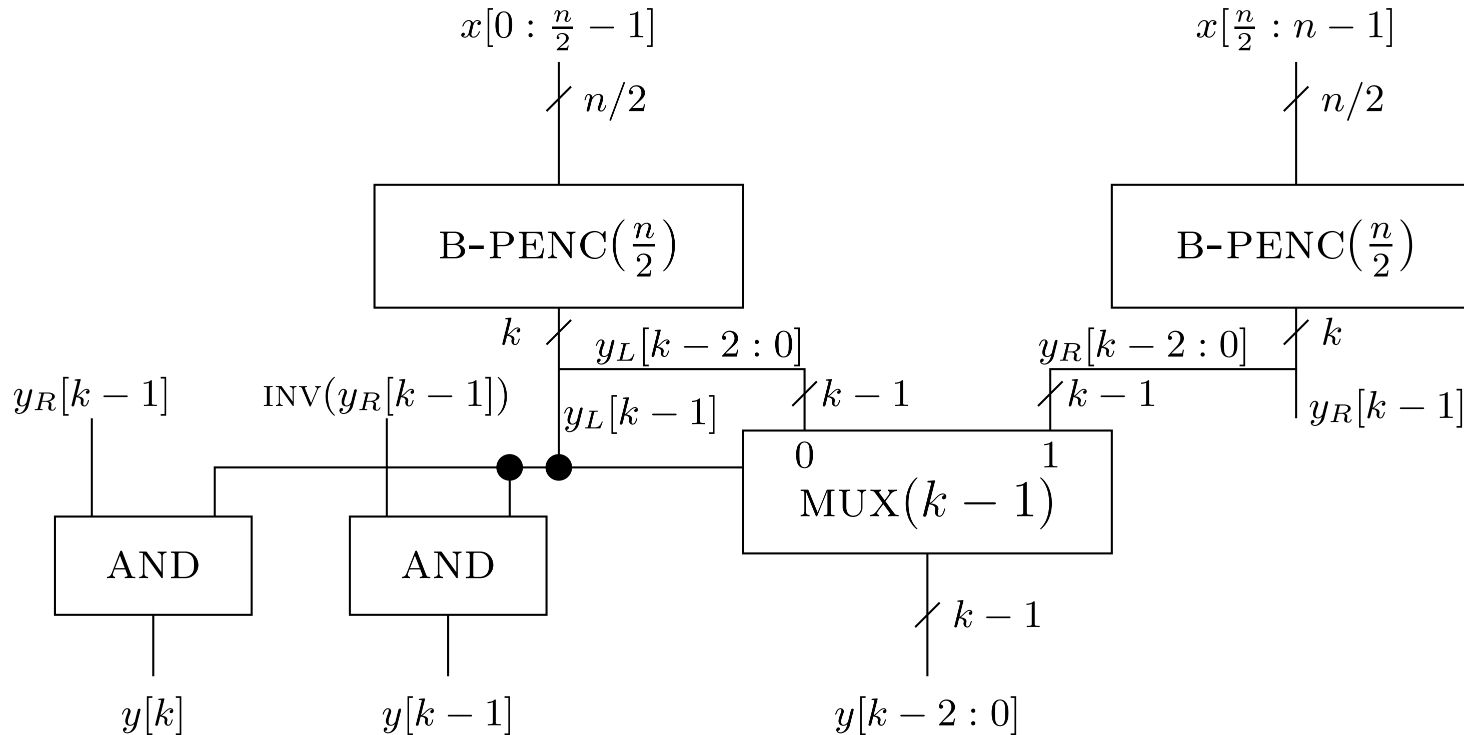
- Induction basis: $n = 1$ is trivial.
- Induction step deals with 3 cases:
  - leading one is in left half $x[0 : \frac{n}{2} - 1]$.
  - leading one is in right half $x[\frac{n}{2} : n - 1]$.
  - $\vec{x} = 0^n$.

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



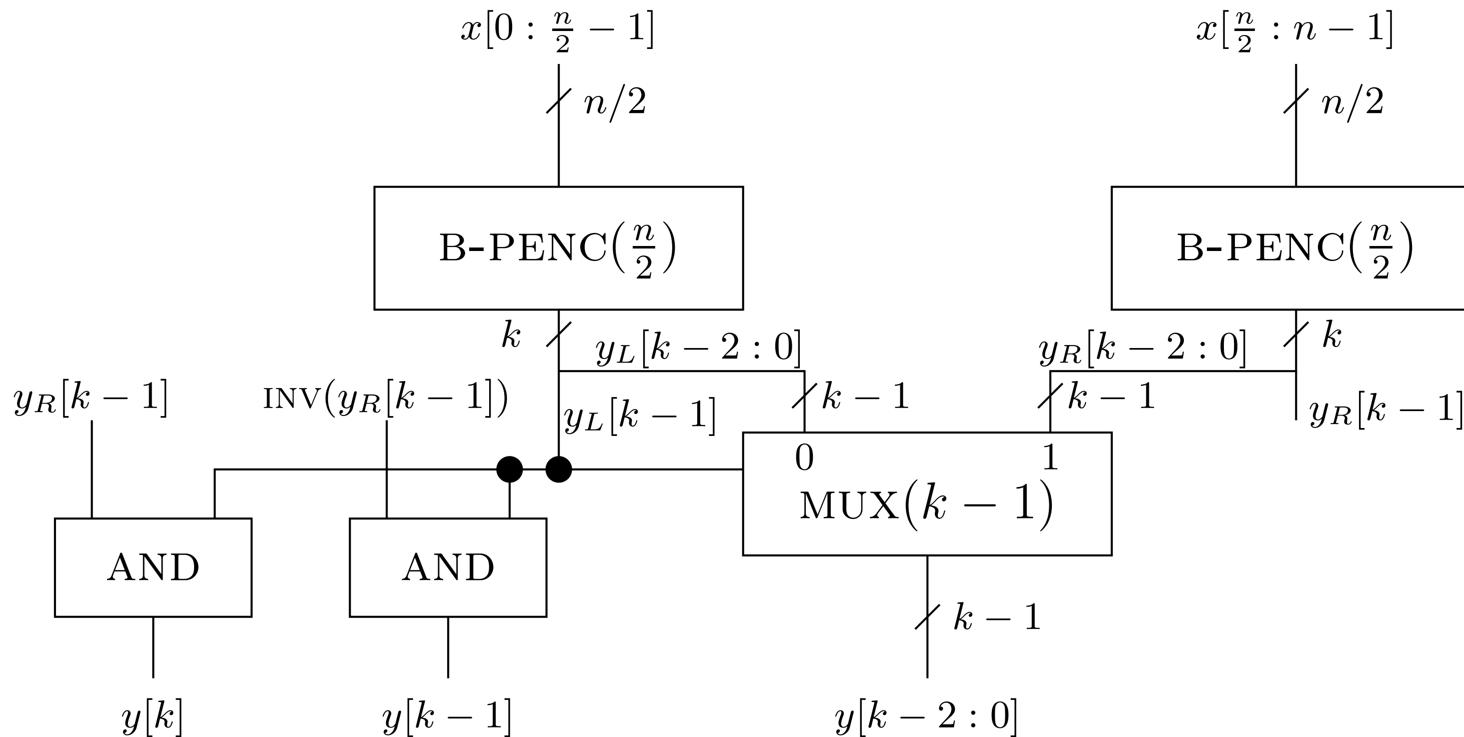
- Ind. Hyp. $\Rightarrow$ required output is $0 \cdot y_L[k-1:0]$.
- index of the leading one $< n/2 \Rightarrow \quad y_L[k-1] = 0$.
- $\Rightarrow \quad y[k] = y[k-1] = 0$ and $y[k-2:0] = y_L[k-2:0]$.
- $\Rightarrow$ output $\vec{y} = 0 \cdot y_L[k-1:0]$.

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



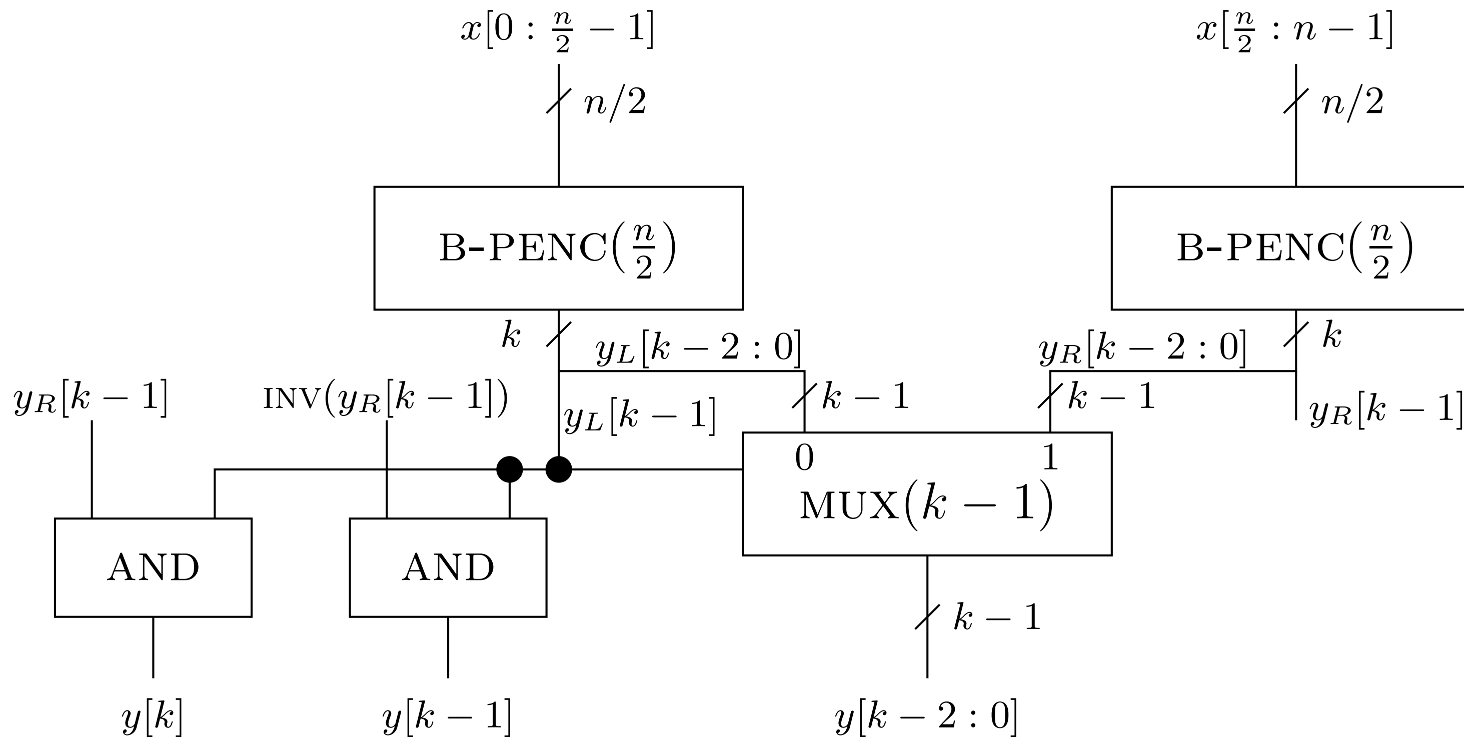
- Ind. Hyp. $\Rightarrow$ index of the leading one is $n/2 + \langle \vec{y_R} \rangle$.
- $\Rightarrow$ required output is $0 \cdot 1 \cdot y_R[k-2:0]$.
- Ind. Hyp. $\Rightarrow$ $y_L[k-1] = 1$ and $y_R[k-1] = 0$.
- $\Rightarrow$ $y[k] = 0$ & $y[k-1] = 1$.
- $y_L[k-1] = 1 \Rightarrow y[k-2:0] = y_R[k-2:0]$.

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

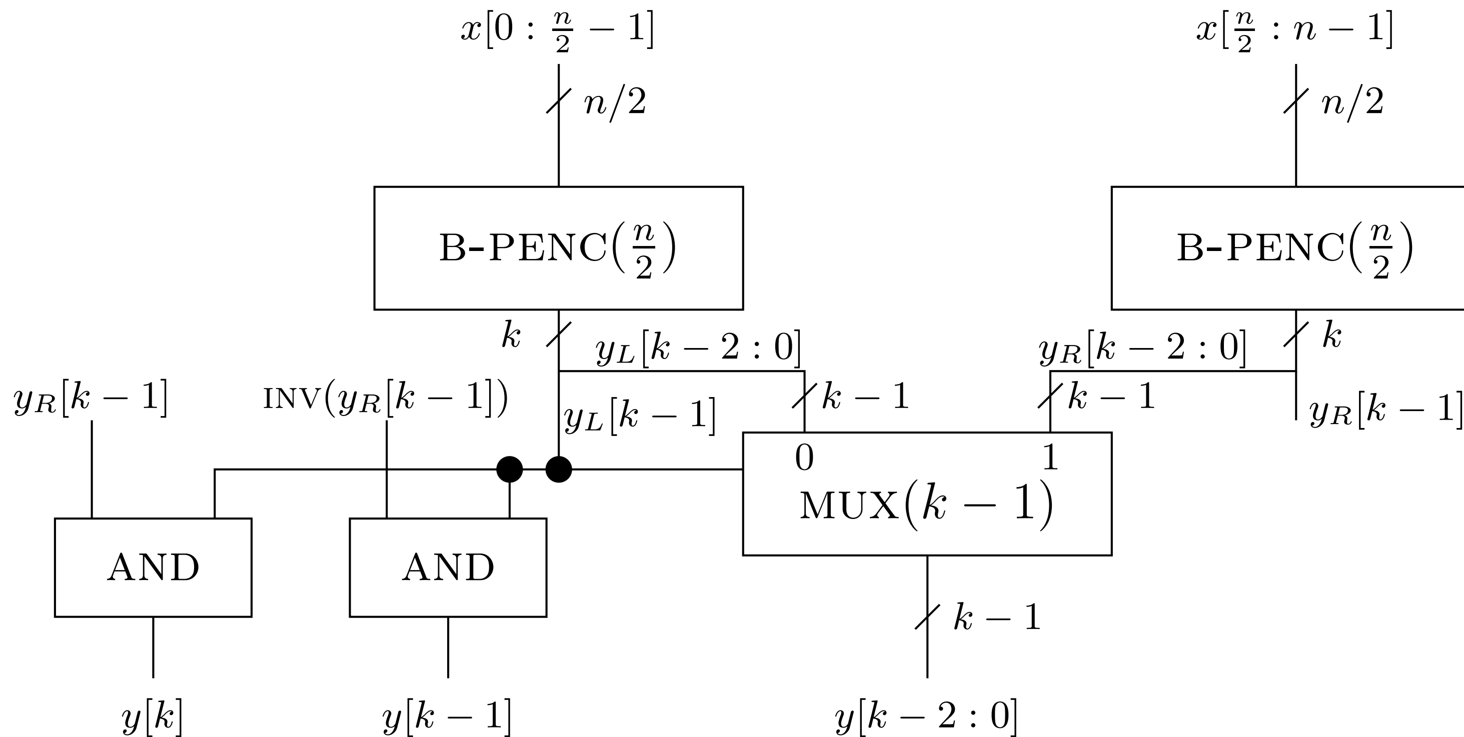
cost analysis



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

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

delay analysis



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

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

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 in cost - $O(n)$ & delay - $O(\log n)$.
  - divide-&-conquer: cost $O(n)$, delay $O(\log n)$.

Chapter 7: Half-Decoders

Computer Structure - Spring 2007

©Dr. Guy Even

Tel-Aviv Univ.

Preliminary questions

- How can we reduce the task of a bi-directional logical shift to a cyclic left shift?
- Is it easy to compare a number x represented in unary representation with a constant i?

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 $\longrightarrow$ 1-out-of-2^n.
- Half-decoder: binary $\longrightarrow$ 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 $\textsf{H-DEC}(n)$ 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

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

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:

$$\textit{wt}(\vec{z}) < i \iff z[i-1] = 0$$

$$\textit{wt}(\vec{z}) > i \iff z[i] = 1$$

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

Example: $z[0 : 5] = 111100$ and *wt*$(\vec{z}) = 4$

$$\textit{wt}(\vec{z}) < 4 \iff z[3] = 0$$

$$\textit{wt}(\vec{z}) > 4 \iff z[4] = 1$$

$$\textit{wt}(\vec{z}) = 4 \iff z[4] = 0 \text{ and } z[3] = 1.$$

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

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

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

Functionality:

$$GT = 1 \text{ if } \textit{wt}(\vec{z}) > i$$

$$EQ = 1 \text{ if } \textit{wt}(\vec{z}) = i$$

$$LT = 1 \text{ if } \textit{wt}(\vec{z}) < i$$

Implementation:

$$GT = z[i]$$

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

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

Claim 3: comparison based on quotient & remainder

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

$$x_L[n-k-1:0] = x[n-1:k] \quad \text{and} \quad x_R[k-1:0] = x[k-1:0].$$

- 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:

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

Implementation: H-DEC(n)

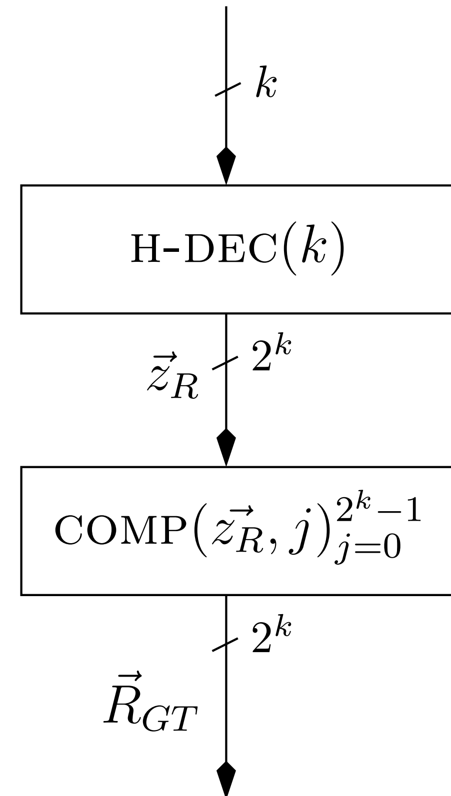
- Recursive divide-and-conquer design.
- Recursion basis: H-DEC(1) - $y[0] \leftarrow x[0]$. (unary representation $\equiv$ binary representation for a single bit).
- Recursion step...

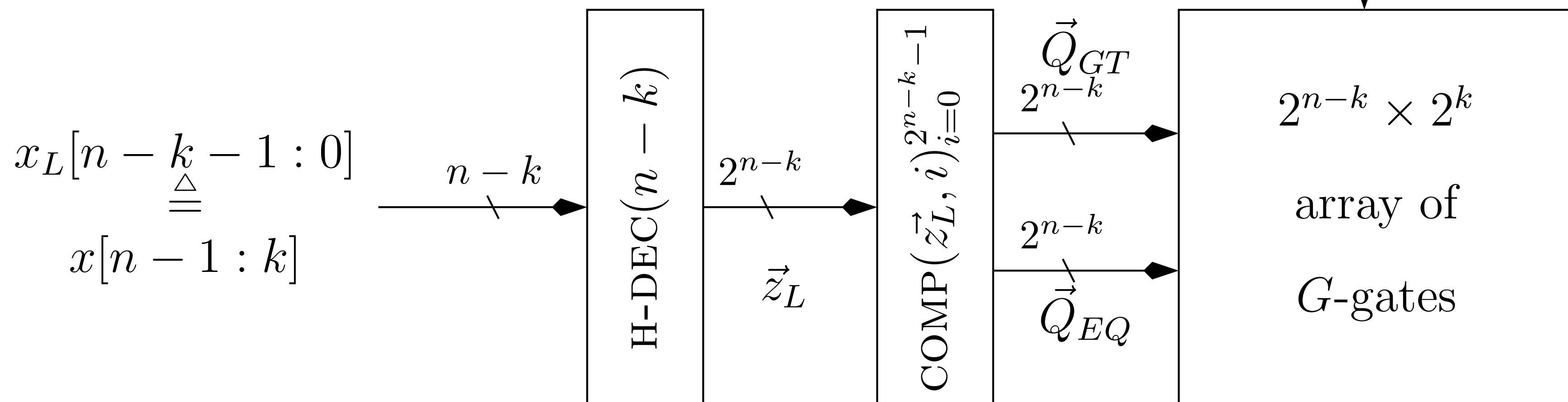
Recursion step: H-DEC(n)

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

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

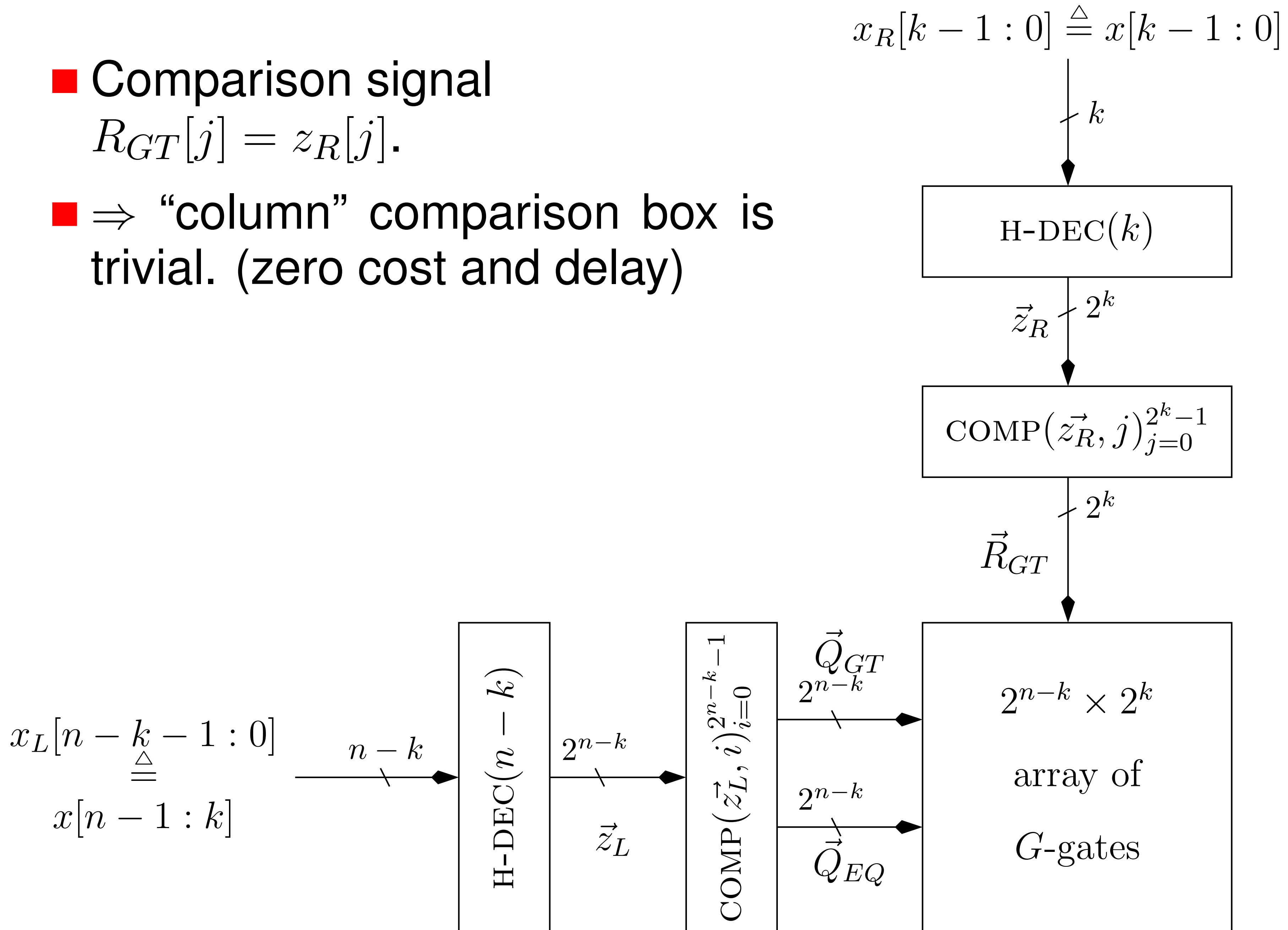
AND$(Q_{EQ}[q], R_{GT}[r]))$





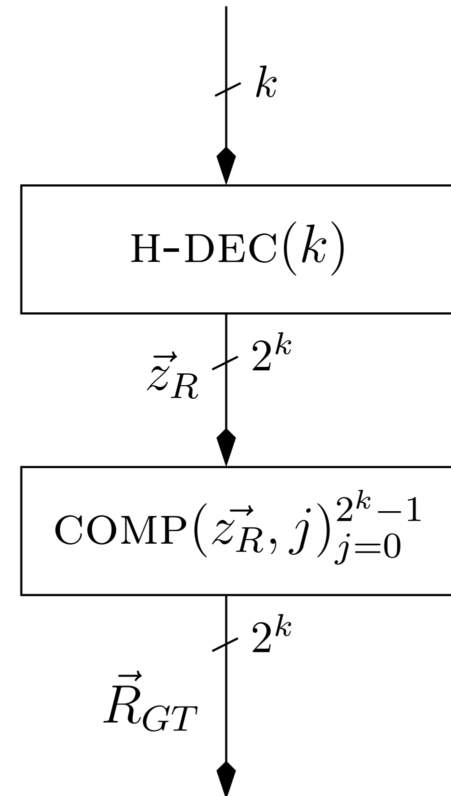
remarks

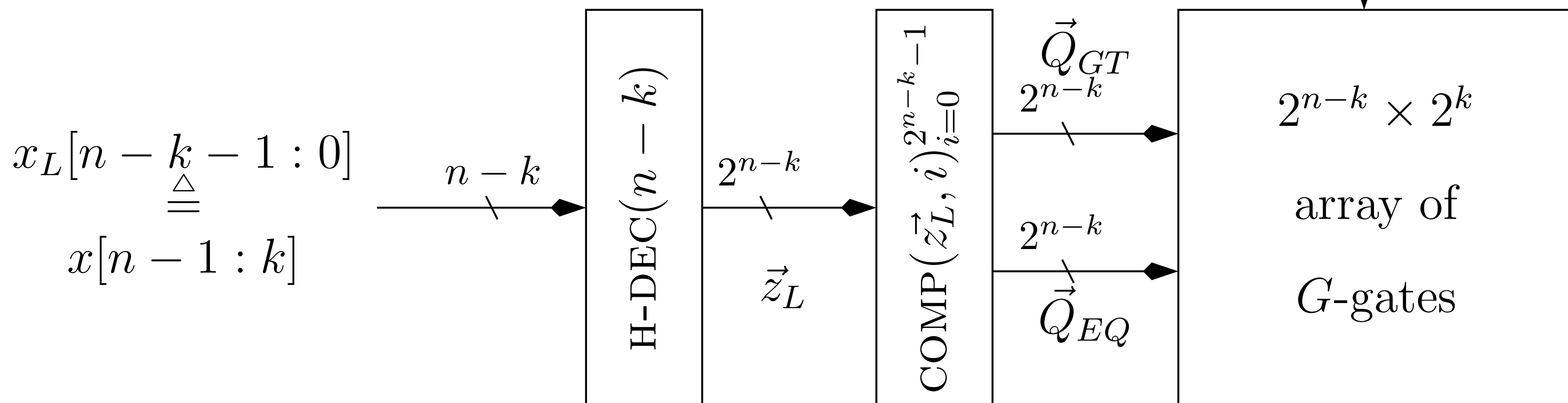
- Comparison signal $R_{GT}[j] = z_R[j]$.
- $\Rightarrow$ "column" comparison box is trivial. (zero cost and delay)



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:

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

Correctness - cont.

- Definition of comparison boxes implies that:

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

- $G_{q,r}$ outputs:

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

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

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

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

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

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

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

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: $\text{H-DEC}(n)$

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

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

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

- Delay of $\text{H-DEC}(n)$ 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 8: Addition

Computer Structure - Spring 2007

© Dr. Guy Even

Tel-Aviv Univ.

Preliminary questions

- What is the definition of an adder?
- What is the smallest possible delay of an adder? Do you know of an adder that achieves this delay?
- Can you prove the correctness of the addition algorithm taught in elementary school?
- Suppose you are given the task of adding very long numbers. Could you share this work with friends so that you could work on it simultaneously to speed up the computation?

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 $\text{ADDER}(n)$:

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

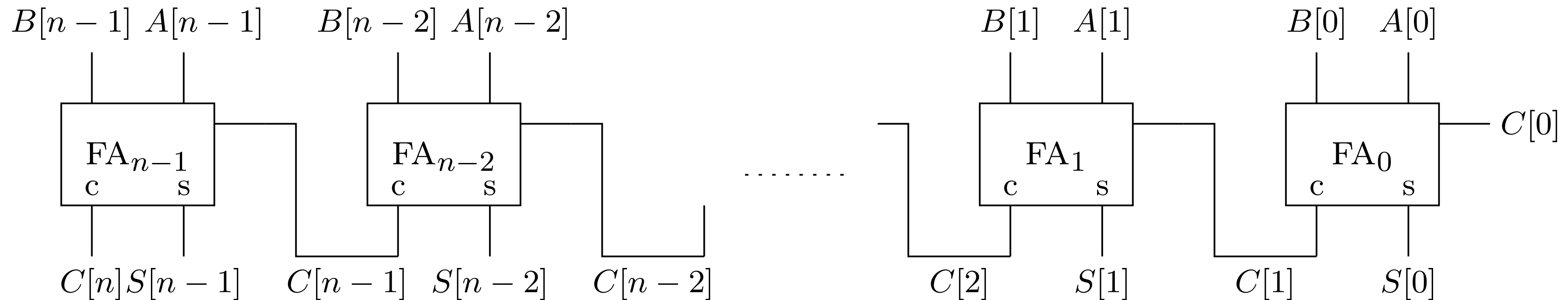
Full Adder

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)



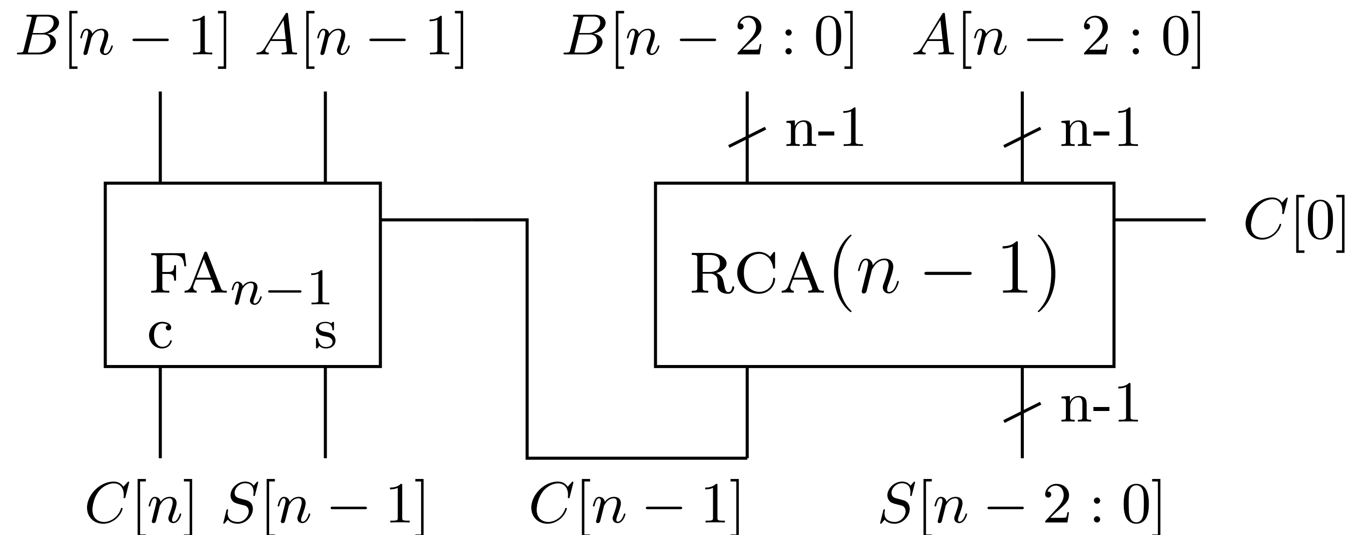
- carry-out output of FA$_i$ is denoted by $c[i+1]$.
- weight of every signal is two to the power of its index.
- RCA(n) - algorithm that we use for adding numbers by hand.

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

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

Full-Adder definition

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

Multiply $\textcolor{red}{(2)}$ by 2^{n-1} to obtain

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

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

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

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 $\textsf{RCA}(n)$ satisfies:

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

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

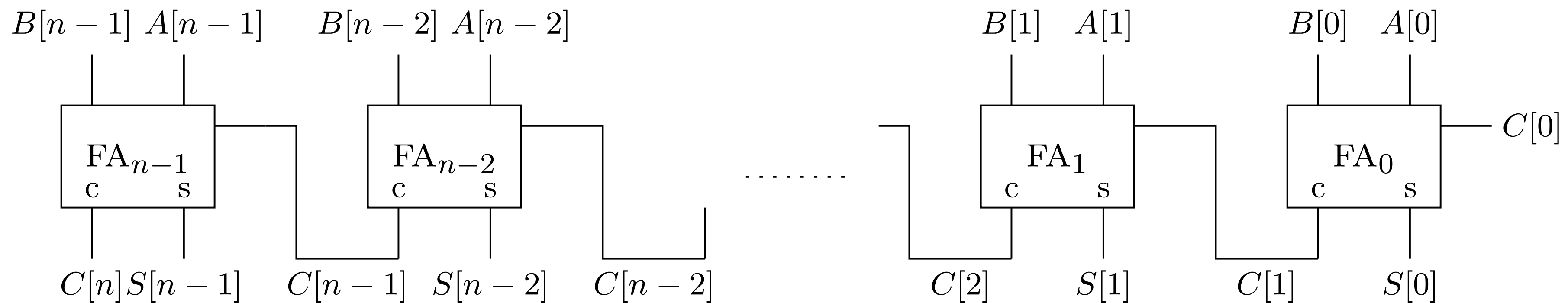
$$d(\textsf{RCA}(n)) = n \cdot d(\textsf{FA}) = \Theta(n).$$

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

- Clock rate $= 1GHz = 10^9 Hz$
  $\Rightarrow$ clock period $= 10^{-9}$sec $= 1ns$.
- Delay of gate $\approx 100ps = 0.1ns$.
- $d(\textsf{FA}) \approx 2 \cdot d(\text{gate}) \approx 0.2ns$.
- $\Rightarrow$ 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 $0 \leq i \leq n-1$,

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

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

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

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

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

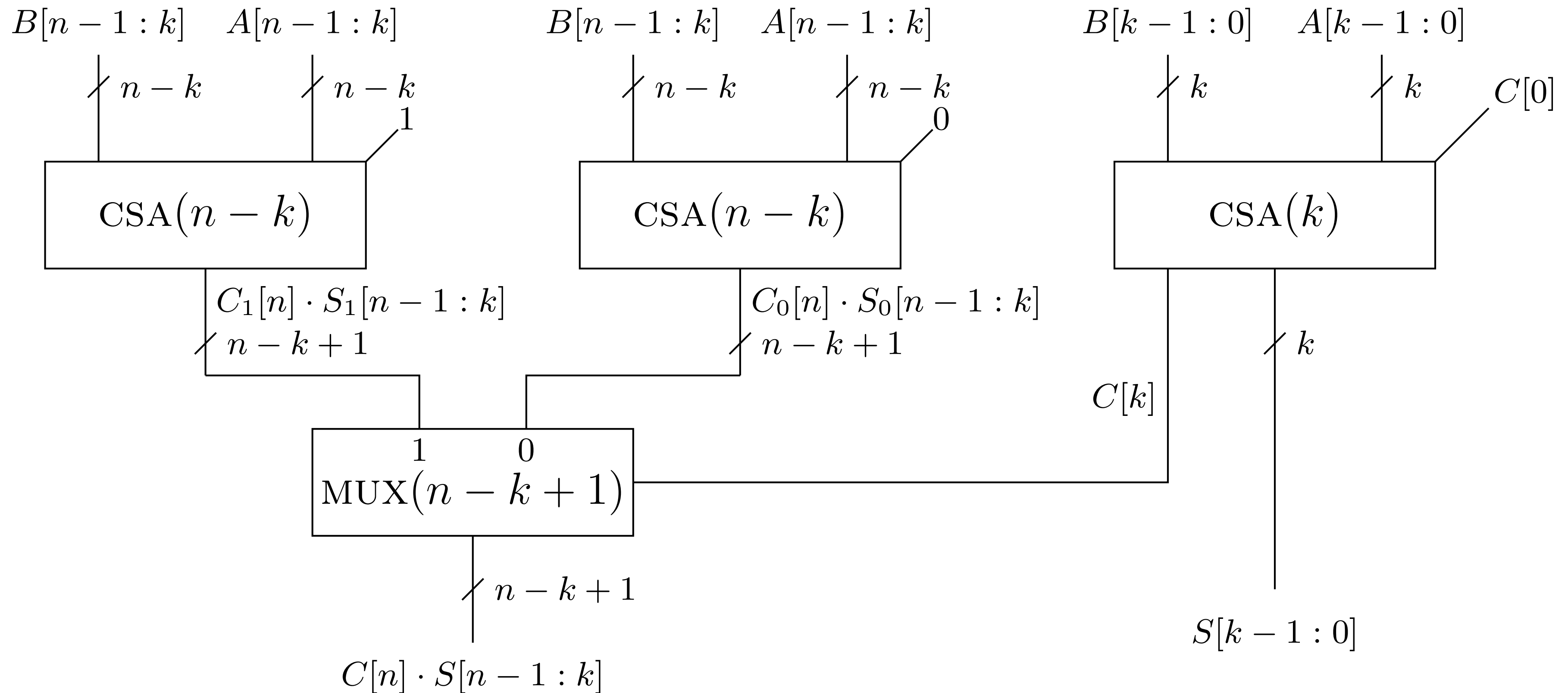
$$C[i] = \text{XOR}(A[i], B[i], S[i]).$$

$\Longrightarrow$ 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 with same asymptotic cost & delay.

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 $\text{CSA}(n)$ 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 $\text{CSA}(n)$ is

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

Cost Analysis

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

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

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

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

Cost Analysis - cont.

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

- 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

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

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 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 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 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:

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

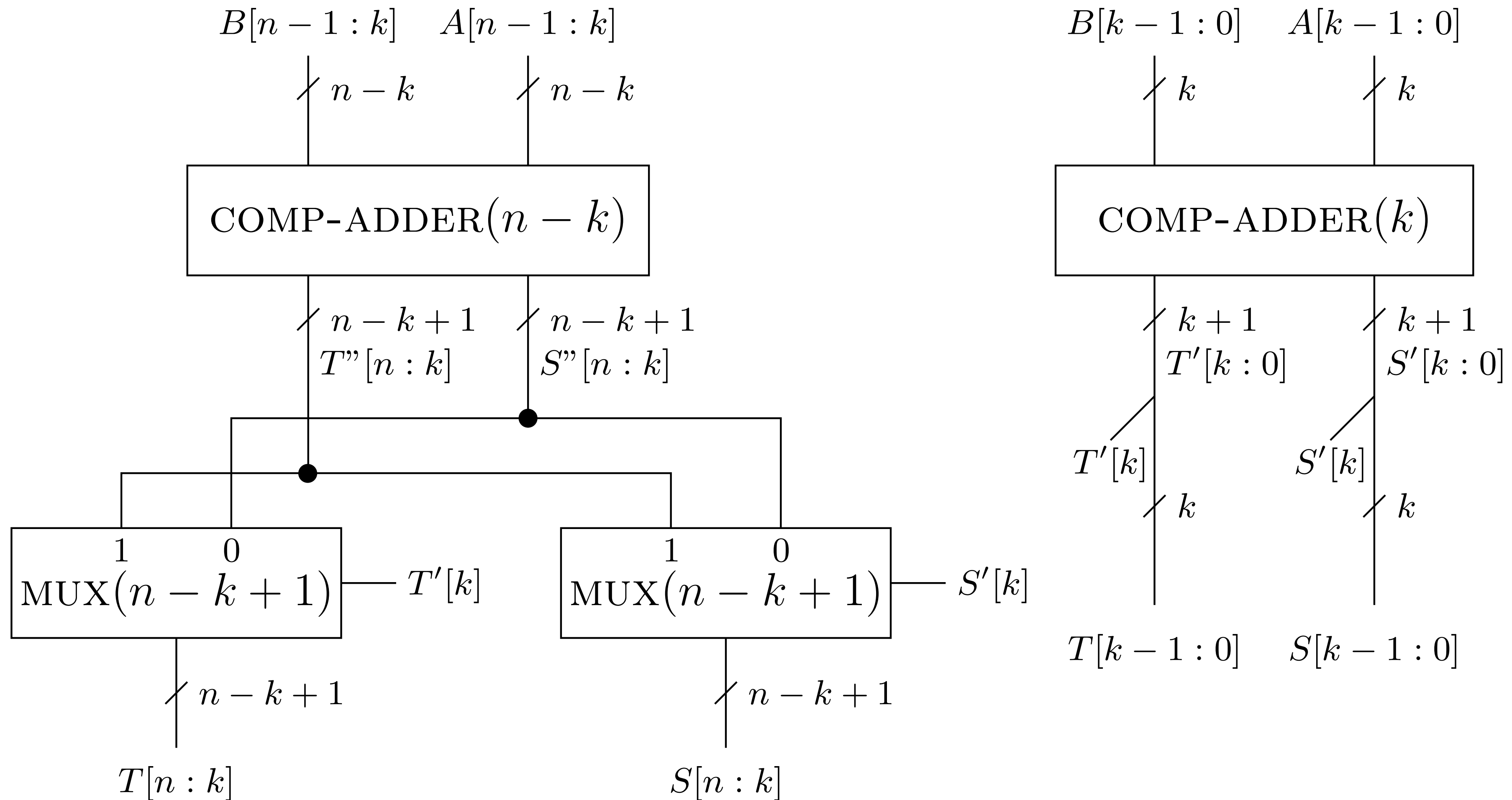
$$\langle \vec{T} \rangle = \langle \vec{A} \rangle + \langle \vec{B} \rangle + 1.$$

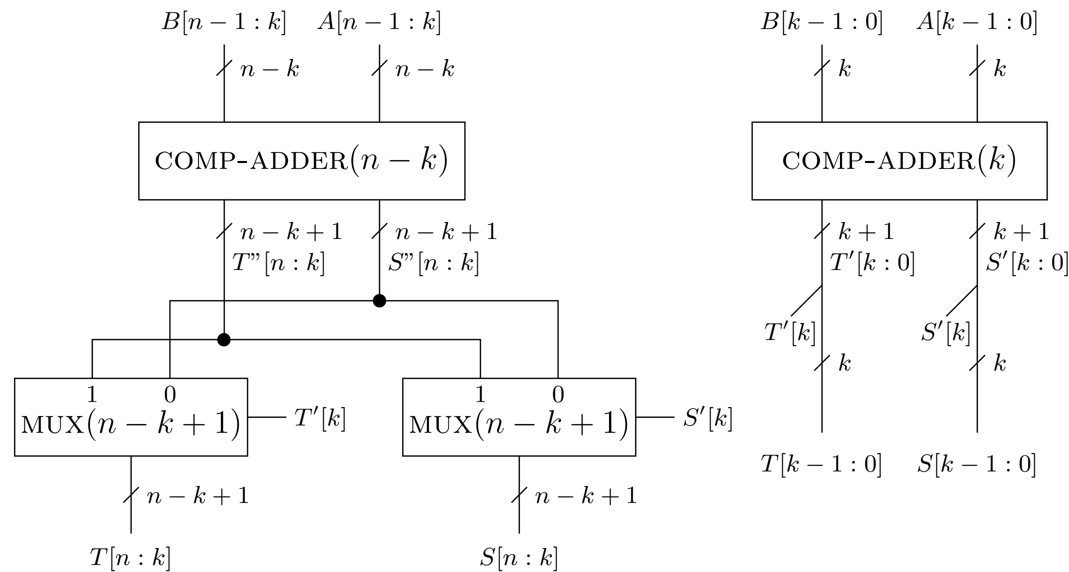
- Compound Adder does not have carry-in input.
- $S[n]$ - carry-out of the sum.
- $T[n]$ - carry-out of the incremented sum.
- COMP-ADDER(n) - a compound adder with input length n.

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

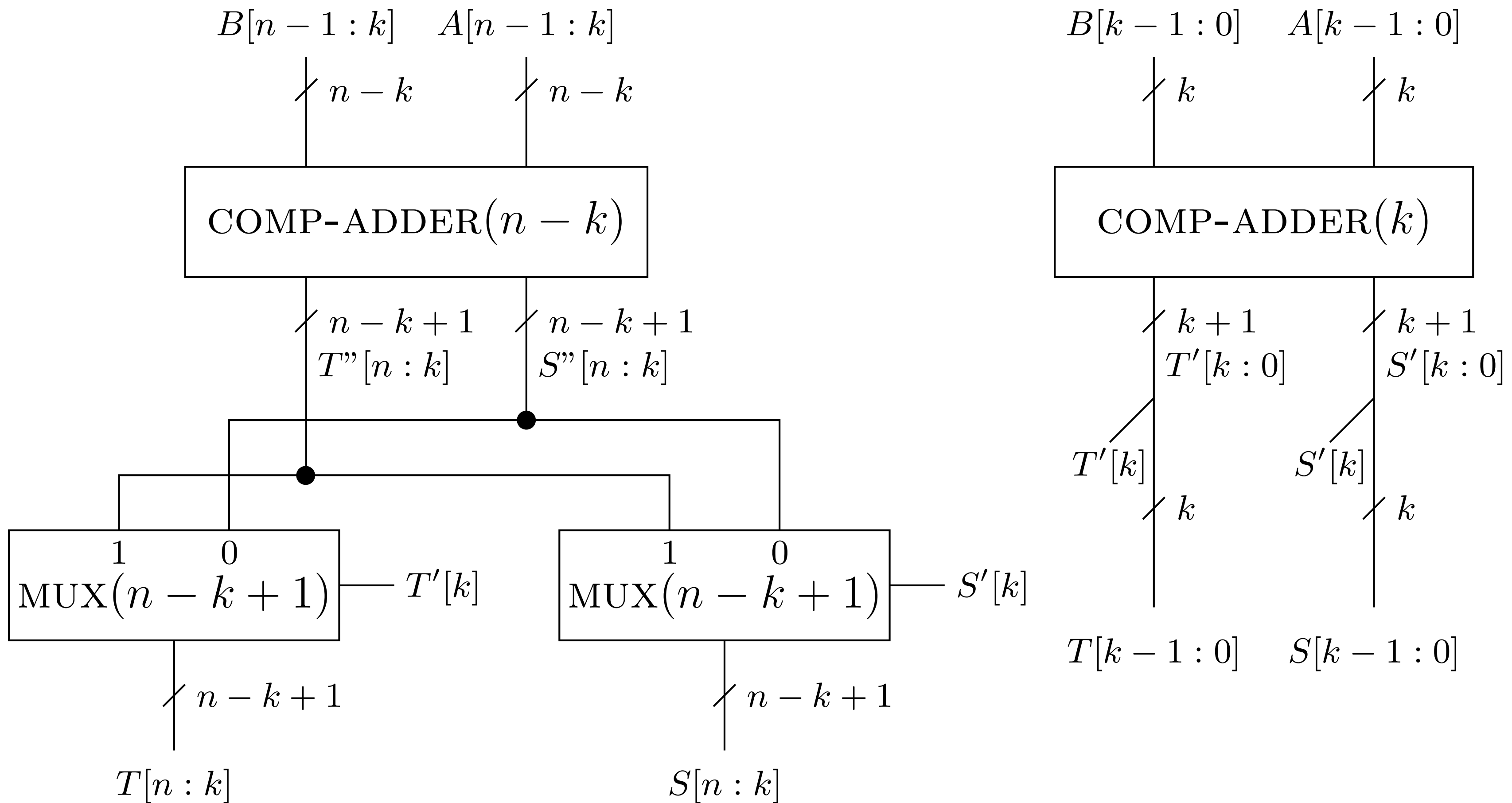




Example: COMP-ADDER(4) with inputs $A[3:0] = 0110$ & $B[3:0] = 1001$.

- $S'[2:0] = 011$ 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

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

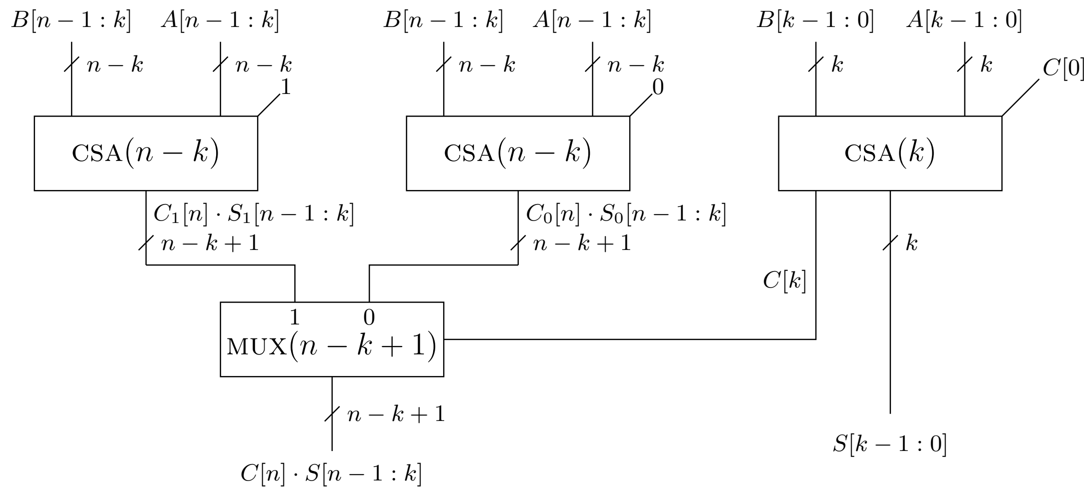
$\Rightarrow$

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

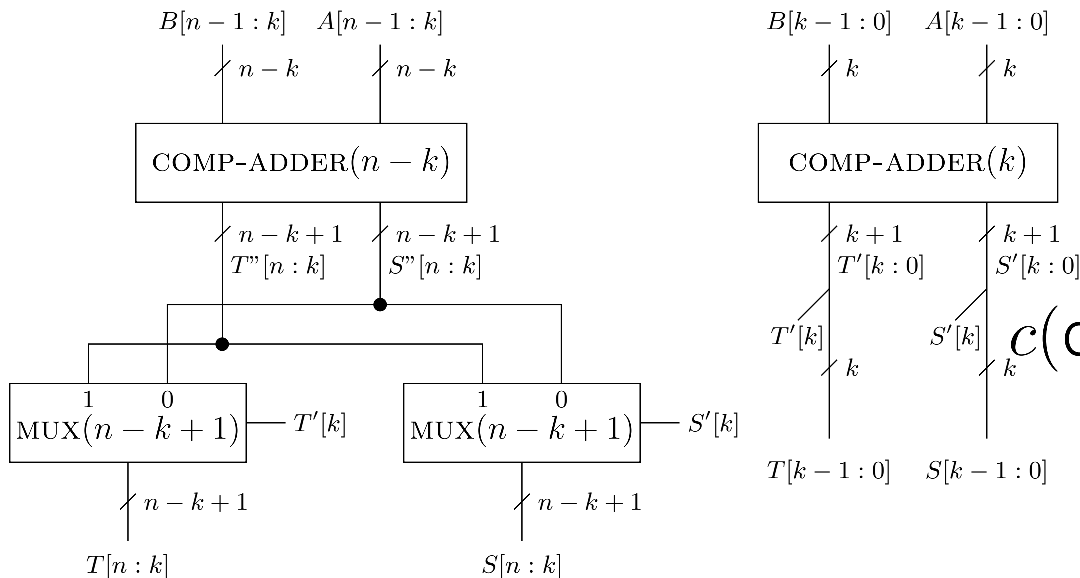
$\Rightarrow$

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

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:

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

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

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

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

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

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

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

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

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

Summary

- defined binary addition & $\text{RCA}(n)$
- 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 9: Fast Addition: parallel prefix computation

Computer Structure - Spring 2007

© Dr. Guy Even

Tel-Aviv Univ.

Preliminary questions

- Is the task of computing the sum bits harder than the task of computing the carry bits?
- Consider a bit-serial adder implemented by a finite state machine with two states. How can one “parallelize” the computation of such a finite state machine? What about a bit serial finite state machine that computes the OR of sequence of bits?

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 (ith FA in RCA(n)):

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

Claim:

$$A[i]+B[i]=0 \implies C[i+1]=0$$
$$A[i]+B[i]=2 \implies C[i+1]=1$$
$$A[i]+B[i]=1 \implies C[i+1]=C[i]$$

$\implies$

- 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 $C[i+1]$

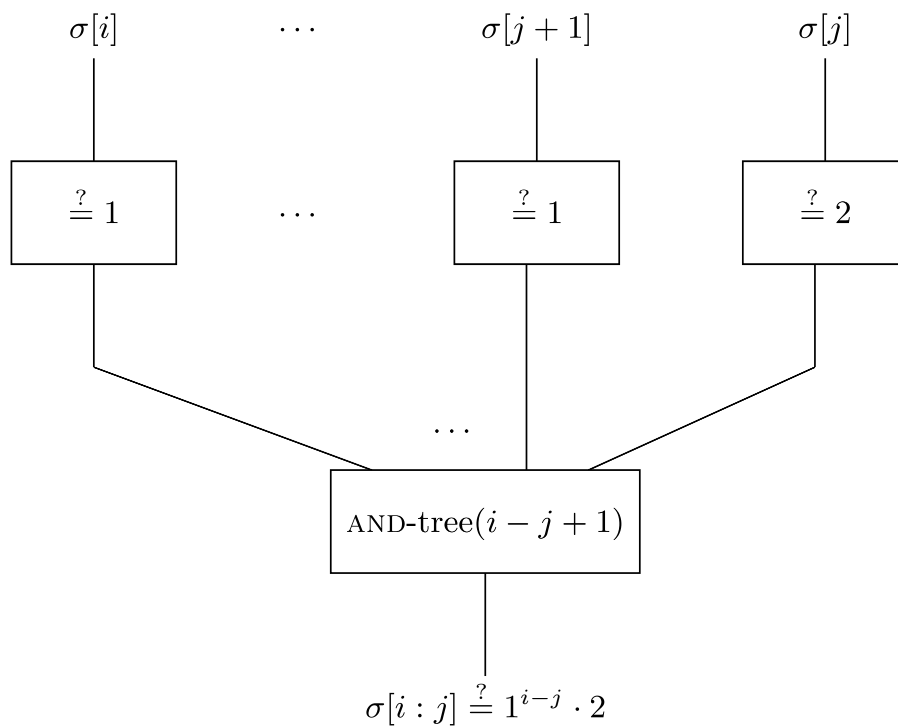
$$C[i+1] = 1 \qquad \Longleftrightarrow \qquad \exists j \leq i \ : \ \sigma[i:j] = 1^{i-j} \cdot 2.$$

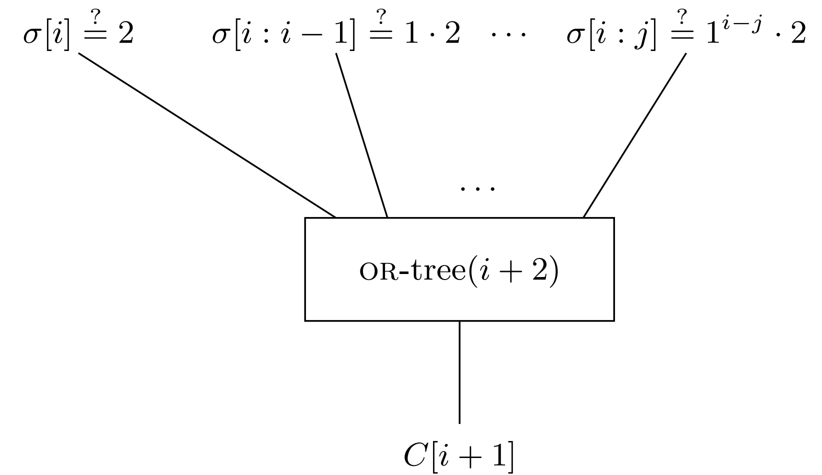
Corollary:

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

Carry-Lookahead Generator

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





Carry-Lookahead Generator: 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 Generator: 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

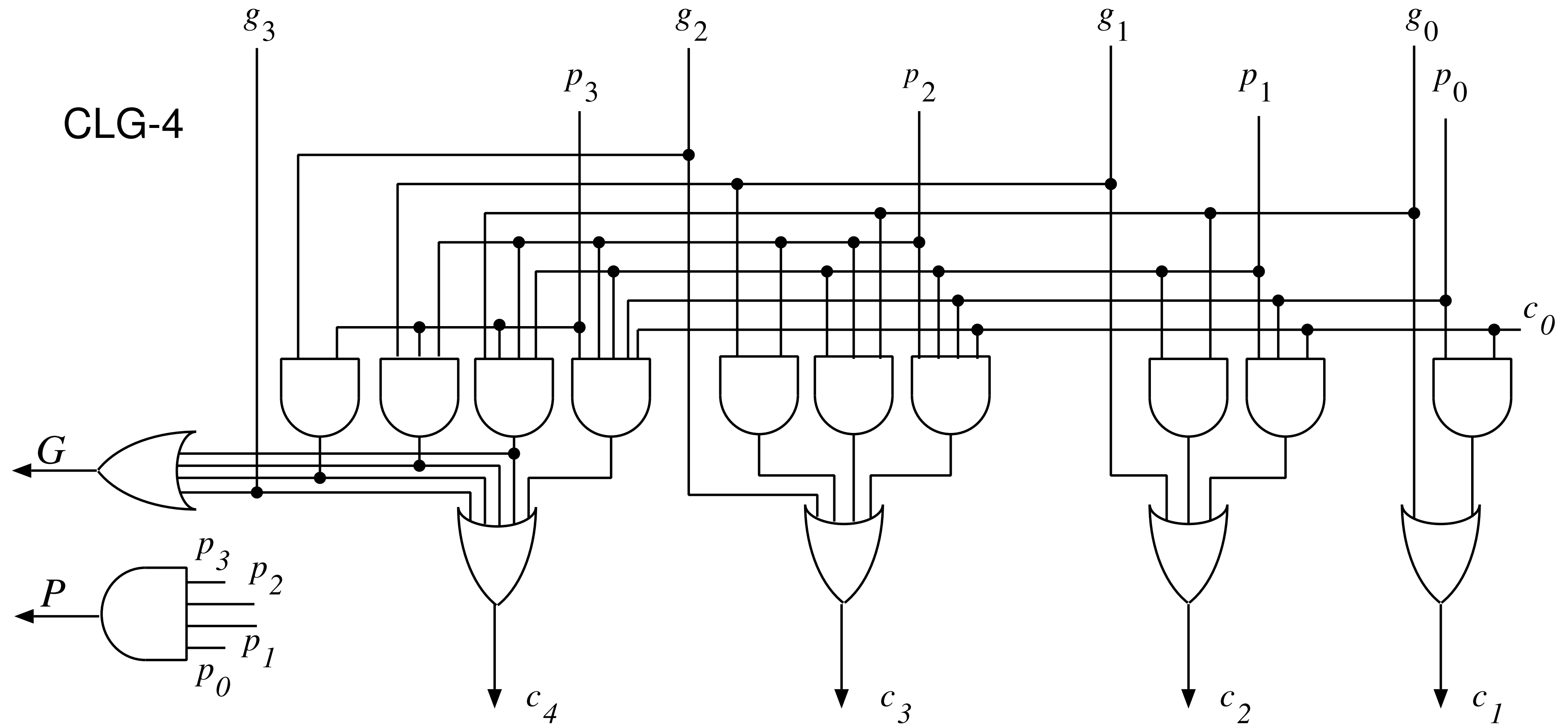
- INTERMEDIATE CARRIES:

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

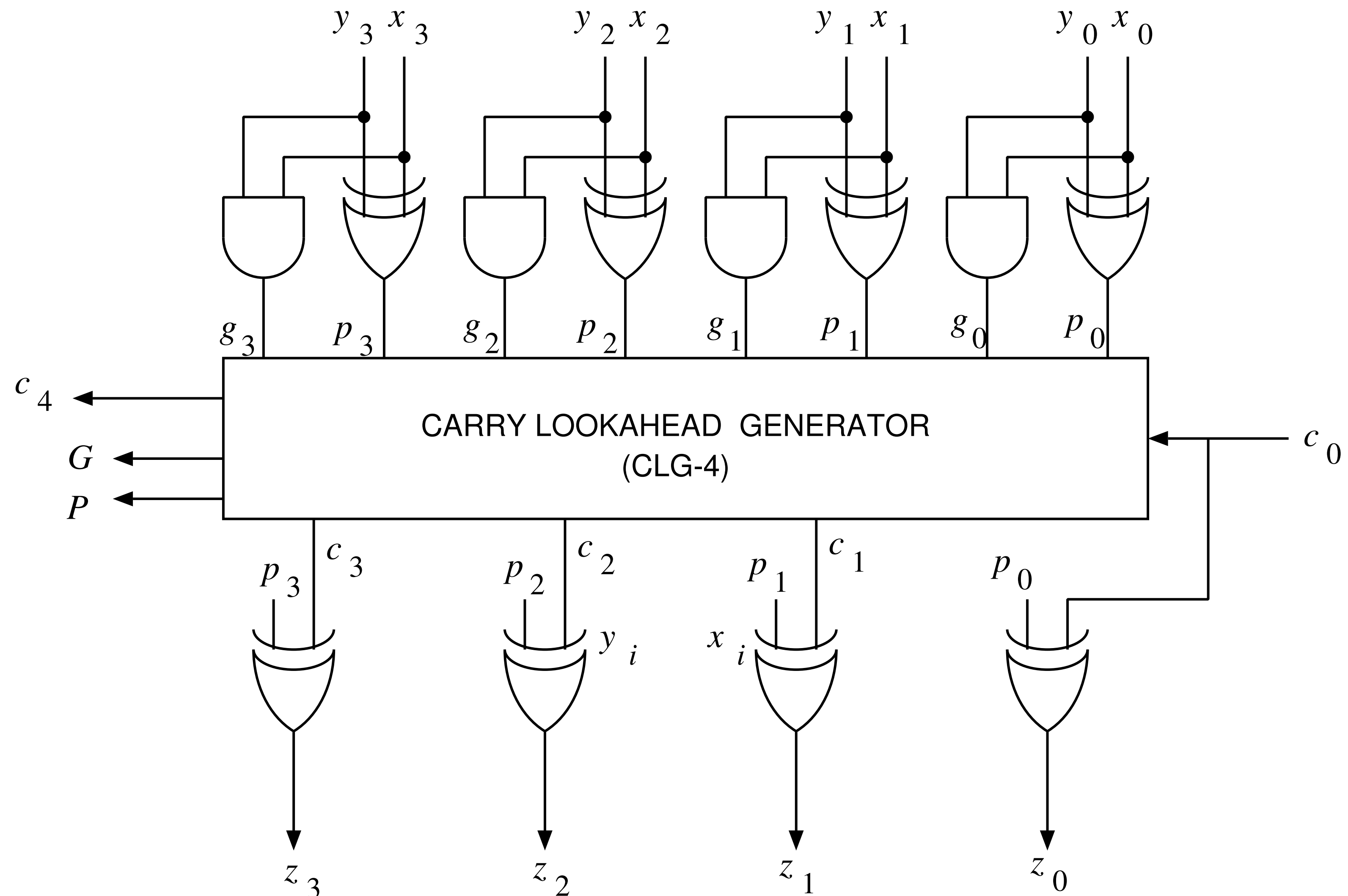
BY SUBSTITUTION,

$$\begin{aligned}
c_1 &= g_0 + p_0 c_0 \\
c_2 &= g_1 + p_1 c_1 \\
&= g_1 + p_1 g_0 + p_1 p_0 c_0 \\
c_3 &= g_2 + p_2 c_2 \\
&= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\
c_4 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0
\end{aligned}$$

Carry-Lookahead Adder: typical description



Carry-Lookahead Adder: typical description



Two-level Carry-Lookahead Adder

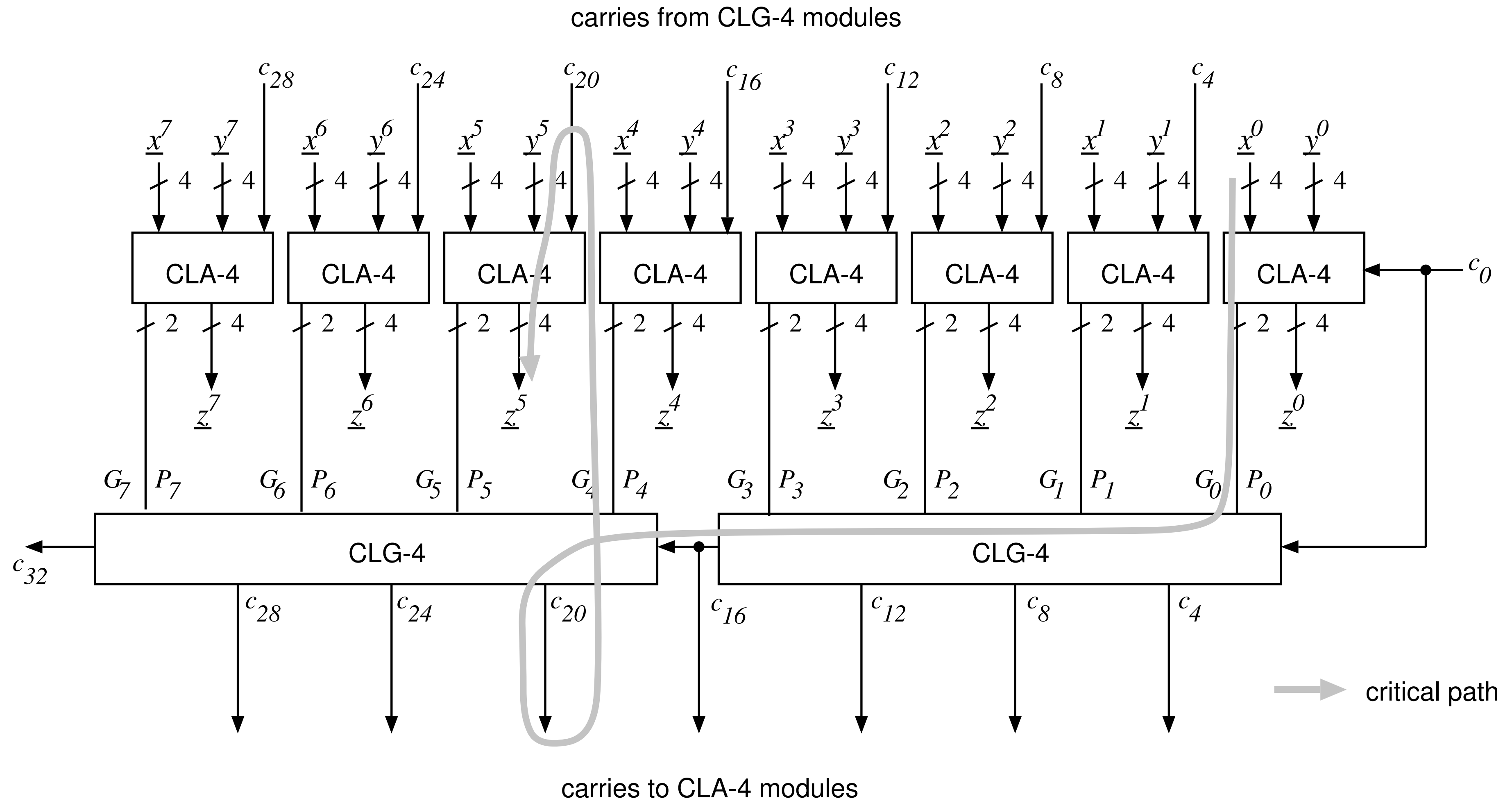


Figure 10.7: 32-BIT CARRY-LOOKAHEAD ADDER USING CLA-4 AND CLG-4 MODULES.

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 $j \geq i$:

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

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

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

A stronger claim

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

$$C[i+1] = 1 \qquad \iff \qquad \pi[i:-1] = 2.$$

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

From previous claim, it suffices to prove that

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

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

- Assume that $\sigma[i : j] = 1^{i-j} \cdot 2$.
- $\Rightarrow$

$$\pi[i : j] = 2.$$

- If $j = -1$ we are done.
- Otherwise,

$$\begin{aligned} \pi[i : -1] &= \underbrace{\pi[i : j]}_{=2} * \pi[j - 1 : -1] \\ &= 2. \end{aligned}$$

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

- Assume that $\pi[i:-1]=2$.
- If, for every $\ell \leq i$, $\sigma[\ell] \neq 2$, then $\pi[i:-1] \neq 2$, a contradiction. Hence

$$\{\ell \in [-1,i] \ : \ \sigma[\ell]=2\} \ \neq \ \emptyset.$$

- Let

$$j \triangleq \max\{\ell \in [-1,i] \ : \ \sigma[\ell]=2\}.$$

- $\pi[j:-1]=2$ (since $2*a=2$).
- We claim that $\sigma[\ell]=1$, for every $j<\ell\leq i$.

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

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 $\pi[i:\ell] = 0$.
  $\Rightarrow$
  $$\pi[i:-1] = \pi[i:\ell] * \pi[\ell - 1 : -1] = 0,$$
  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 OP : $\Sigma^2 \longrightarrow \Sigma$ denote an associative function. A prefix computation over Σ with respect to 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:

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

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

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

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

The Claim

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

implies a reduction of the problem of computing $C[n:1]$ to a Prefix Computation Problem:

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

Prefix Computation Problem - example

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

$\Longrightarrow$ PPC–OR(n) used to design a Unary Priority Encoder U-PENC(n).

Parallel Prefix Circuit

DEF: A Parallel Prefix Circuit, $\text{PPC–OP}(n)$, 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 $\text{OP}(a, b)$.

PPC–OP(n) - questions

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

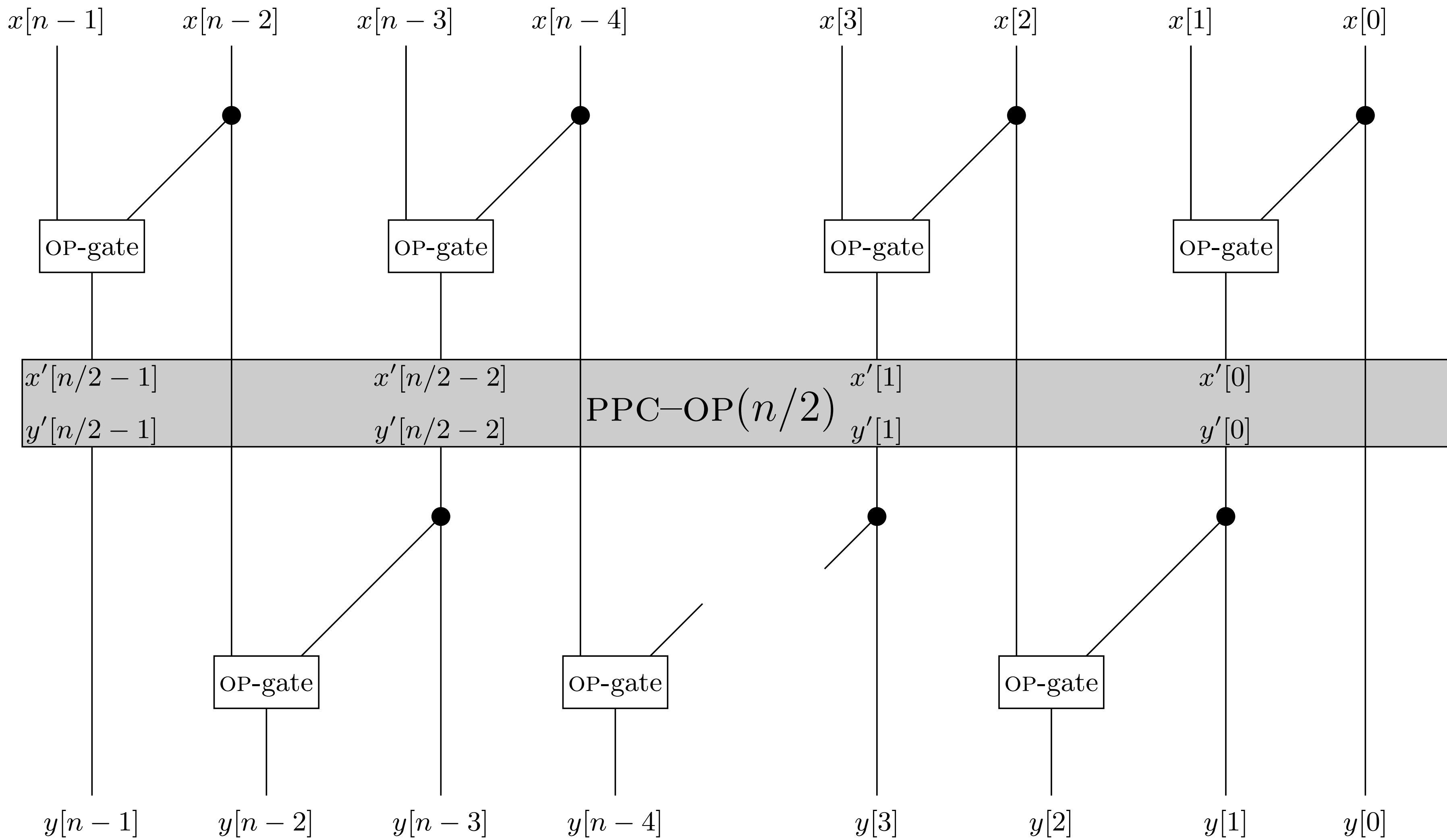
Question: Design a PPC–OP(n) circuit with logarithmic delay and quadratic cost.

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 : $\Sigma^2 \rightarrow \Sigma$. 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 PPC–OR(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



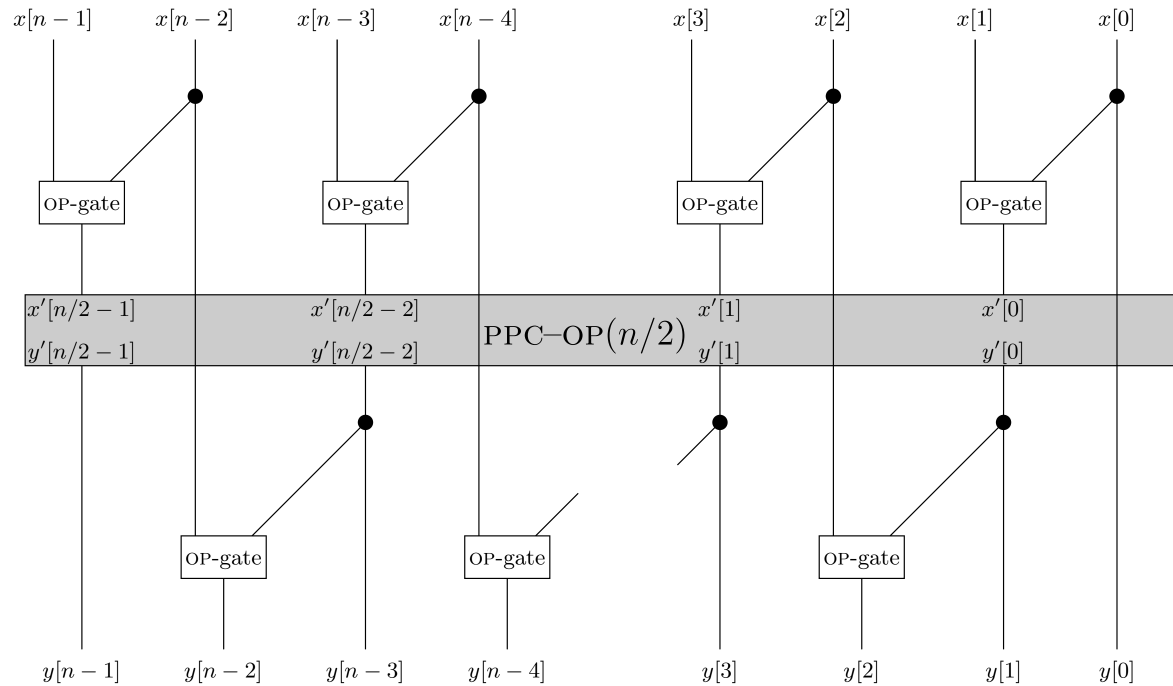
PPC–OP(n) - correctness

- 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 $(n = 2^k)$

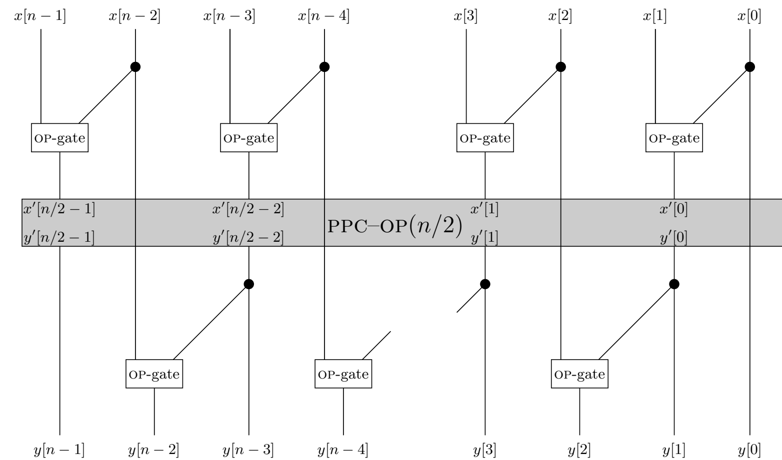


$$d(\text{PPC–OP}(n)) = \begin{cases} d(\text{OP-gate}) & \text{if } n = 2 \\ d(\text{PPC–OP}(n/2)) + 2 \cdot d(\text{OP-gate}) & \text{otherwise.} \end{cases}$$

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

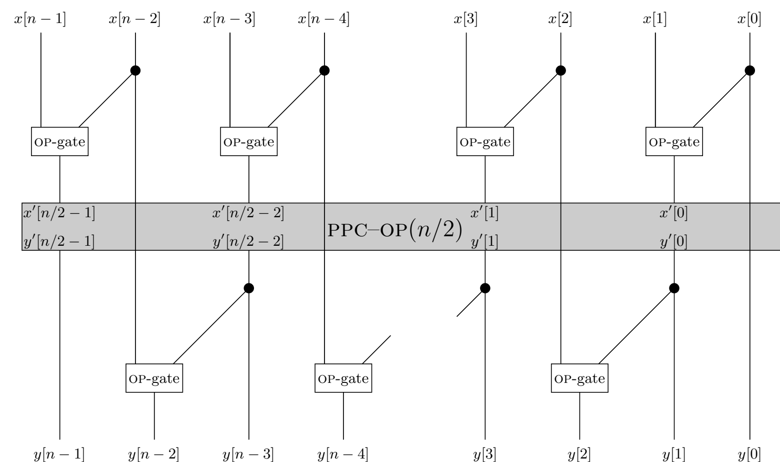
$$d(\text{PPC–OP}(n)) = \Theta(\log n)$$
$$c(\text{PPC–OP}(n)) = \Theta(n).$$

$\Rightarrow$

- $\Sigma = \{0, 1\}$ & 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 $\pi[i:-1]$ 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 $\pi[i:-1] = 2$.
Compare each $\pi[i:-1]$ 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 10: Signed Addition

Computer Structure - Spring 2007

©Dr. Guy Even

Tel-Aviv Univ.

Preliminary questions

- How are signed integers represented in a computer?
- How are signed integers added and subtracted in a computer?
- Can we use the same circuitry for adding unsigned and signed integers?

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.

DEF: Suppose that the string A represents the value x. Negation means computing the string B that represents $-x$.

Question: 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.

Remark: T_n is not closed under negation: $-2^{n-1} \in T_n$ but $2^{n-1} \notin T_n$.

Two's complement - negation

Claim:

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

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

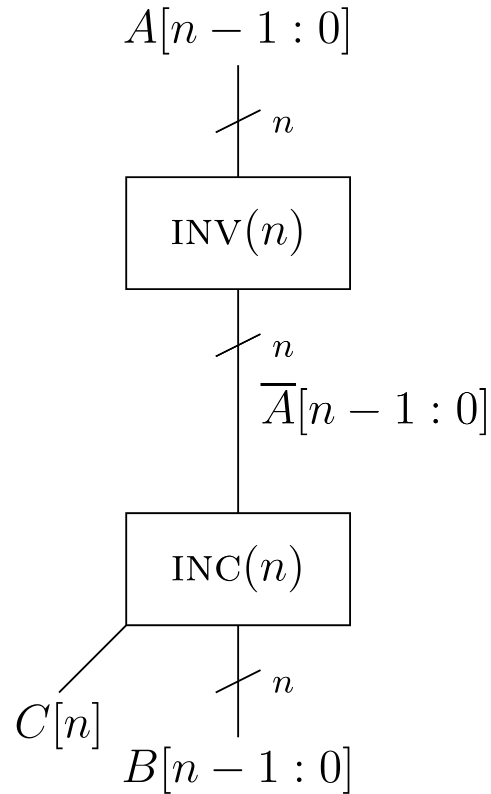
$$\begin{aligned}
[\text{INV}(A[n-1:0])] &= -2^{n-1} \cdot \text{INV}(A[n-1]) + \langle \text{INV}(A[n-2:0]) \rangle \\
&= -2^{n-1} \cdot (1 - A[n-1]) + \sum_{i=0}^{n-2} (1 - A[i]) \cdot 2^i \\
&= \underbrace{-2^{n-1} + \sum_{i=0}^{n-2} 2^i}_{=-1} + \underbrace{2^{n-1} \cdot A[n-1] - \sum_{i=0}^{n-2}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

QED.

A circuit for negating a two's complement number

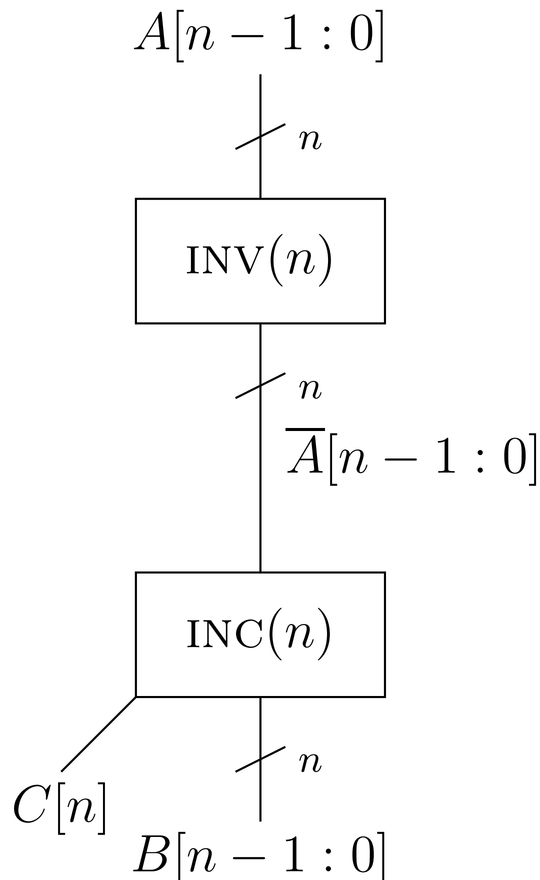
Claim:

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



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

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

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 $C[n] = 0$.

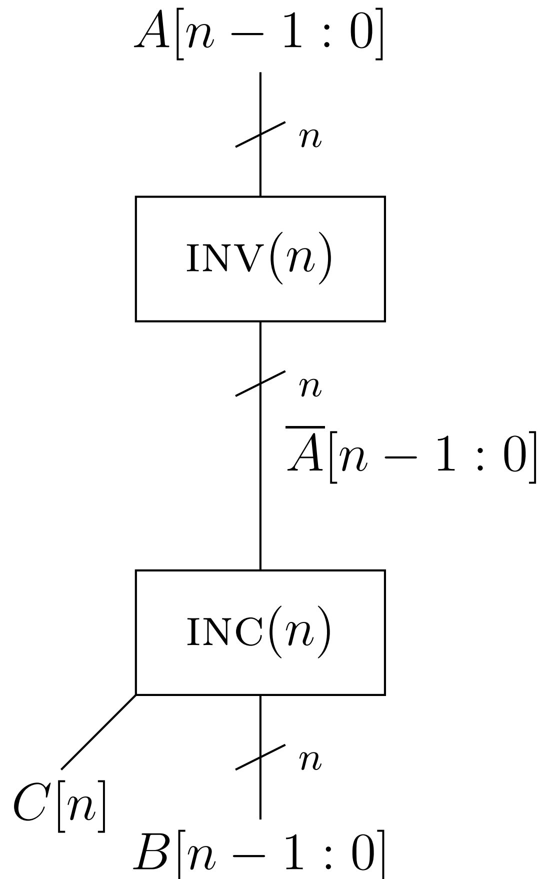
$$\langle B[n-1:0] \rangle = \langle \overline{A}[n-1:0] \rangle + 1.$$

Why should this imply that

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

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

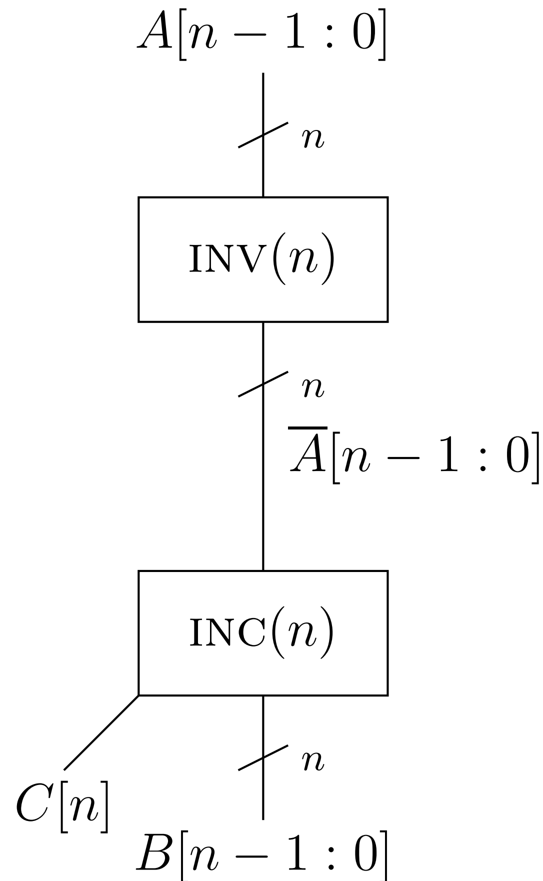
$$B[n-1:0] = 1 \cdot 0^{n-1} = A[n-1:0].$$

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

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.

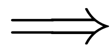
Theorem: signed addition $\longmapsto$ binary addition

- 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.

$$z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0].$$



$$\begin{aligned}
C[n-1] - C[n] = 1 \quad &\Longrightarrow \quad z > 2^{n-1} - 1 \\
C[n] - C[n-1] = 1 \quad &\Longrightarrow \quad z < -2^{n-1} \\
z \in T_n \quad &\Longleftrightarrow \quad C[n] = C[n-1] \\
z \in T_n \quad &\Longrightarrow \quad z = [S[n-1:0]]\,.
\end{aligned}$$

Theorem - proof

functionality of $\textsf{FA}_{n-1}$ in $\textsf{RCA}(n) \Longrightarrow$

$$A[n-1] + B[n-1] + C[n-1] = 2C[n] + S[n-1]$$
$$\Rightarrow A[n-1] + B[n-1] = 2C[n] - C[n-1] + S[n-1].$$

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
$$\begin{aligned} z &= -2^n + [S[n-1:0]] \\ &\leq -2^n + 2^{n-1} - 1 = -2^{n-1} - 1. \end{aligned}$$

2. If $C[n] - C[n-1] = -1$, then
$$\begin{aligned} z &= 2^n + [S[n-1:0]] \\ &\geq 2^n - 2^{n-1} = 2^{n-1}. \end{aligned}$$

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

$$C[n-1] = \text{XOR}_3(A[n-1], B[n-1], S[n-1]).$$

□

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: $\text{NEG} = \text{XOR}_3(A[n-1], B[n-1], C[n])$.

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: $\text{NEG} = \text{XOR}_3(A[n-1], B[n-1], C[n])$ - 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: $\text{NEG} = \text{XOR}_3(A[n-1], B[n-1], C[n])$ - 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 $z \in T_{n+1}$:

$$z \overset{\text{sign-ext}}{=} \left[\tilde{A}[n:0]\right] + \left[\tilde{B}[n:0]\right] + C[0] \overset{\text{no OVF}}{=} \left[\tilde{S}[n:0]\right].$$

$$\implies \quad \text{NEG} = \tilde{S}[n].$$

Proof: $\textsf{NEG} = \textsf{XOR}_3(A[n-1], B[n-1], C[n])$ - cont.

$$\begin{aligned}\textsf{NEG} &= \tilde{S}[n] \\ &= \textsf{XOR}_3(\tilde{A}[n], \tilde{B}[n], \tilde{C}[n]) \\ &= \textsf{XOR}_3(A[n-1], B[n-1], C[n]).\end{aligned}$$

QED

More on NEG

Question: Prove that 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 NEG, OVF $\in \{0,1\}$.

Functionality: Define z as follows:

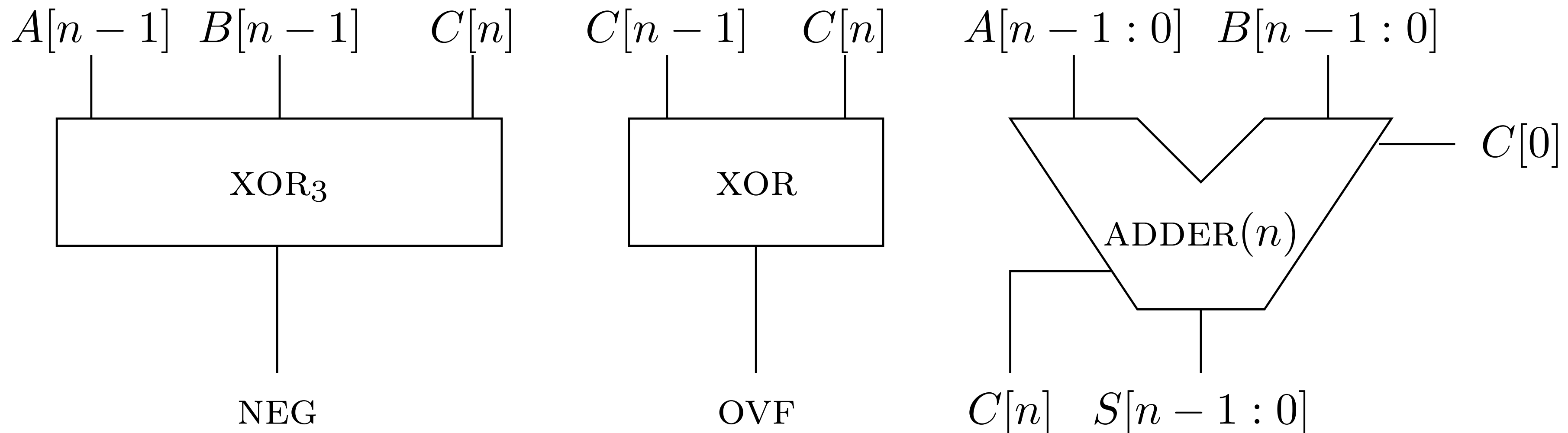
$$z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0].$$

The functionality is defined as follows:

$$\begin{aligned} z \in T_n &\implies [S[n-1:0]] = z \\ z \in T_n &\iff \text{OVF} = 0 \\ z < 0 &\iff \text{NEG} = 1. \end{aligned}$$

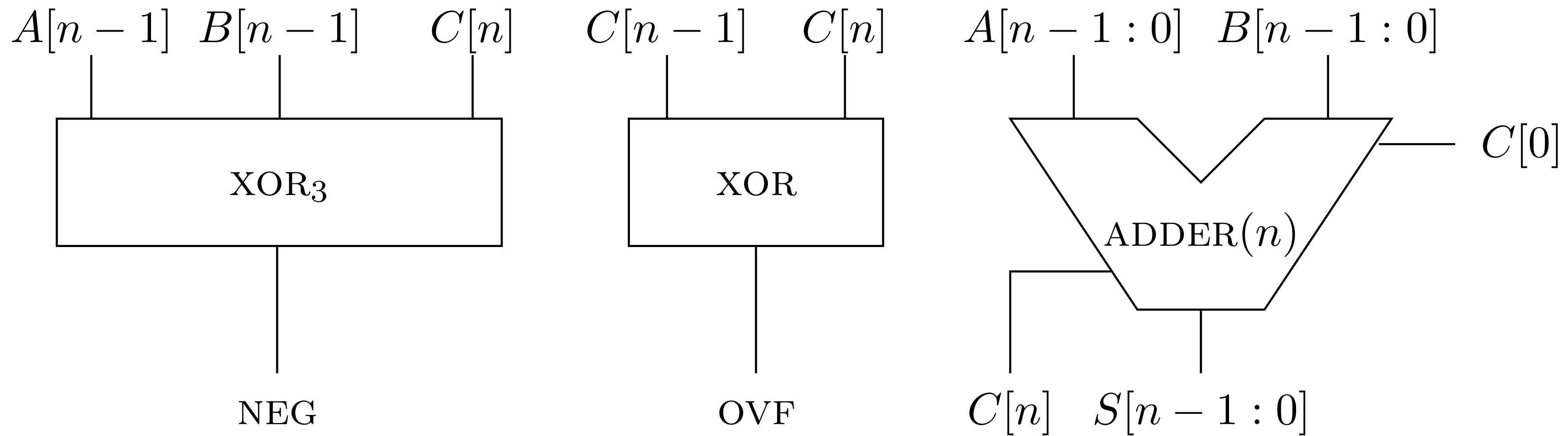
- Note that no carry-out $C[n]$ is output.

S-ADDER(n) - implementation



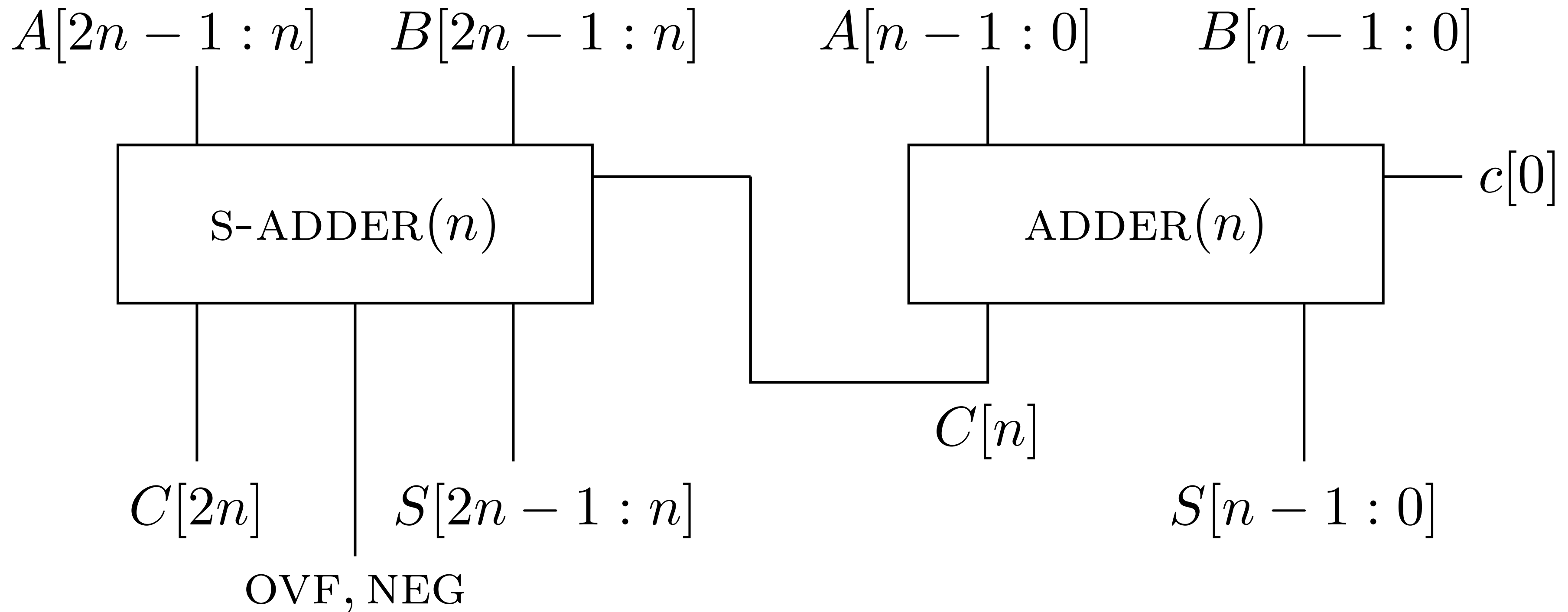
- a two's complement adder is identical to a binary adder except for the circuitry that computes the flags OVF and NEG.
- in an arithmetic logic unit (ALU), the same circuit is used for signed addition and unsigned addition.

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:

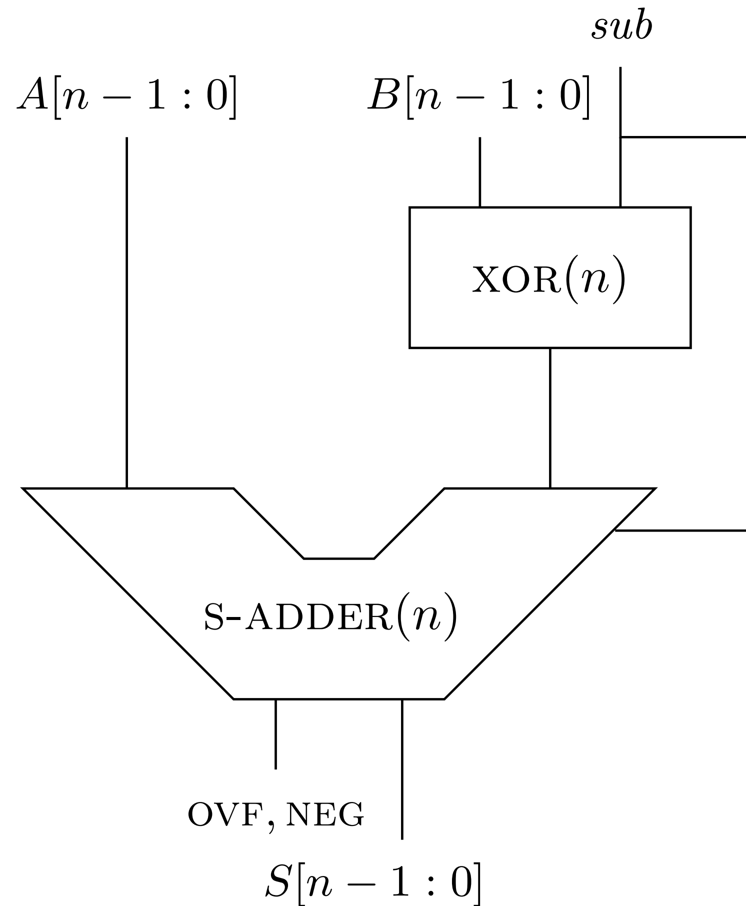
$$z \triangleq [A[n-1:0]] + (-1)^{\textit{sub}} \cdot [B[n-1:0]] .$$

The functionality is defined as follows:

$$\begin{aligned} z \in T_n &\implies [S[n-1:0]] = z \\ z \in T_n &\iff \text{OVF} = 0 \\ z < 0 &\iff \text{NEG} = 1. \end{aligned}$$

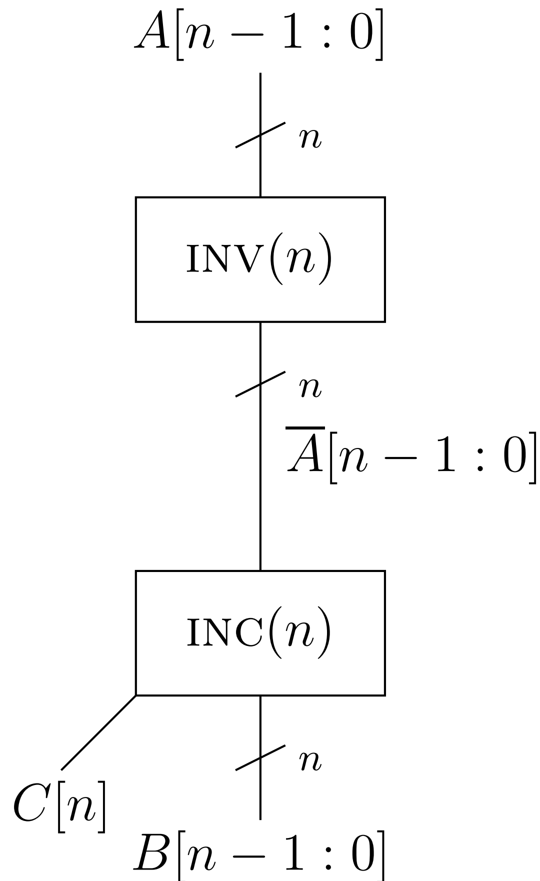
- *sub* - indicates if the operation is addition or subtraction.
- no carry-in bit $C[0]$ is input & no carry-out $C[n]$ is output.

ADD-SUB(n) - implementation



Question: Is this implementation correct?

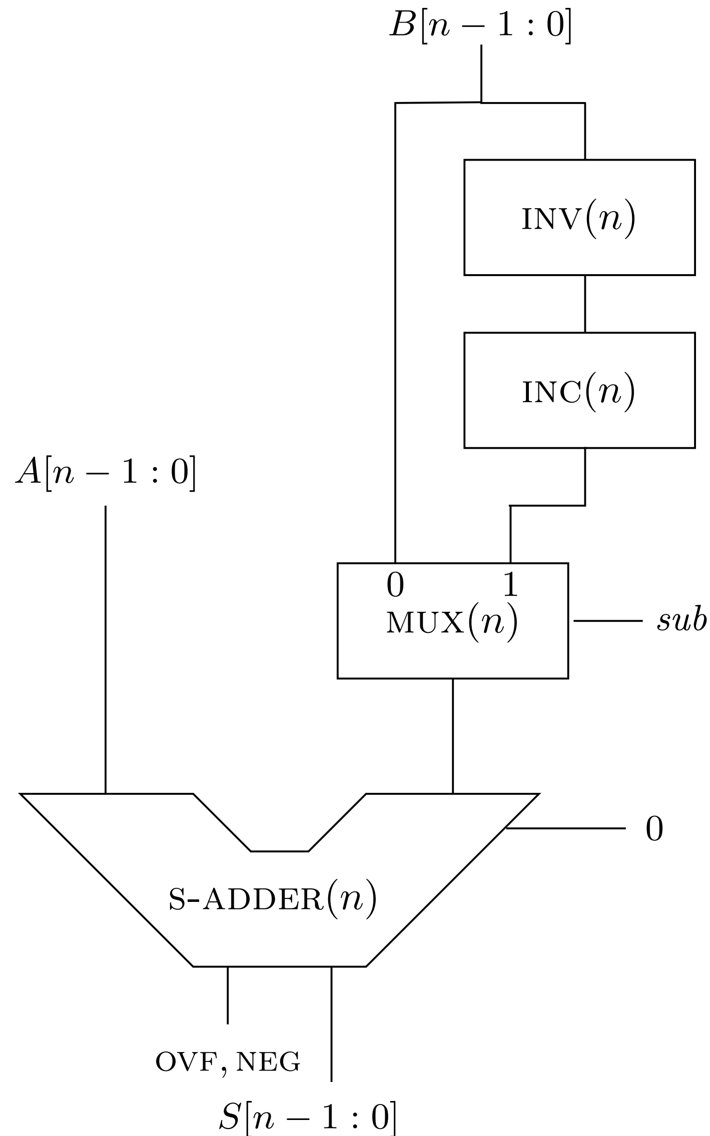
back to the negation circuit



Question:

1. When is the circuit correct?
2. Suppose we wish to add a signal that indicates whether the circuit satisfies $\left[\vec{B}\right] = -\left[\vec{A}\right]$. How should we compute this signal?
3. Does $C[n]$ indicate whether $\left[\vec{B}\right] \neq -\left[\vec{A}\right]$?

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).