Digital Logical Systems

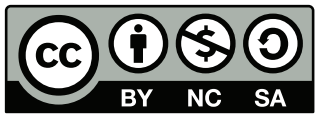
Notes from TAU Course with Additional Information

Lecturer: Guy Even

Gabriel Domingues

August 8, 2020

Contents



1 Discrete Mathematics

1.1 Set Theory (Succinctly)

There will be no definition of a set. Instead, we postulate the existence of a relation $\in$ (read as "is in").

Definition 1.1.1 (Empty)**.** *It is axiom the existence of the empty set $\varnothing$, that is, $\exists \varnothing : \forall x\,,\, x \notin \varnothing$*

Definition 1.1.2 (Principle of Double Inclusion)**.** *We define the following symbols:*

Inclusion: $A \subseteq B \Leftrightarrow \forall x\,,\, x \in A \Rightarrow\ x \in B$
Equality: $A = B \Leftrightarrow A \subseteq B$ and $B \subseteq A \Leftrightarrow \forall x\,,\, x \in A \Leftrightarrow\ x \in B$
Proper Inclusion: $A \subsetneq B \Leftrightarrow A \subseteq B$ and $A \neq B$

Remark 1.1.3 (Contrapositive)**.** *For two propositions P and Q. Then the proposition $P \Rightarrow Q$ is equivalent to $\neg Q \Rightarrow \neg P$, where $\neg P$ is the oppositive statement.*

Definition 1.1.4 (PRC)**.** *We can create new sets by the Principle of Restricted Comprehension: Let A be a set and P a predicate (given an object, it is either True or False), then the following is a set: $\{x \in A \mid P(x)\}$*

Example 1.1.5 (Set Difference)**.** *Let A and B be any two sets. Construct: $A \setminus B = \{x \in A \mid x \notin B\}$ (which is a set cf. 1.1.4)*

Definition 1.1.6 (Complement)**.** *Let $A \subseteq U$. Define $\overline{A} = U \setminus A$.*

Lemma 1.1.7 (Double Complement)**.** *For any $A \subseteq U$, $\overline{\overline{A}} = A$.*

Proof. Double Inclusion:
($\subseteq$) For any $x \in U$, $x \in \overline{\overline{A}} \Rightarrow x \notin \overline{A} \Rightarrow x \in A$
($\supseteq$) For any $x \in U$, $x \notin A \Rightarrow x \in \overline{A} \Rightarrow x \notin \overline{\overline{A}}$, by contrapositive (cf. 1.1.3). □

Definition 1.1.8 (Power Set)**.** *It is also axiom the existence of the power set: Given a set A, there is a set $\mathcal{P}(A)$ so that: $x \in \mathcal{P}(A) \Leftrightarrow x \subseteq A$*

Remark 1.1.9. $\forall A\,,\, \varnothing \in \mathcal{P}(A)$

Definition 1.1.10 (Set Operations)**.** *For sets A and B these are sets (by axiom):*

Union : $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$

Intersection : $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$

Cartesian Product : $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$

Note we use Universal Comprehension, but Zermelo-Frankel axioms guarantee those are sets (Axiom of Pairing).

Example 1.1.11. *If we define $(a, b) := \{\{a\}, \{a, b\}\}$, then the comparision $(a', b') = (a, b) \Leftrightarrow a' = a$ and $b' = b$ is imediatelly satisfied. This is a construction that only use the Axiom of Pairing and PRC (cf. 1.1.4) to build the Cartesian Product.*

Definition 1.1.12 (Cartesian Products)**.** *We define recursively:*

Base Case: $A^0 = \{\varnothing\}$, $A^1 = A$

Reduction: $A^n = A \times A^{n-1}$

Remark 1.1.13. *We can construct the sets we will mainly use:*

Bits $\{0, 1\} = \{0, 1\}$

Natural Numbers $\mathbb{N} = \{0, 1, 2, \cdots\}$

Integer Numbers $\mathbb{Z} = \{\cdots, -2, -1, 0, 1, 2, \cdots\}$

Note: $\mathbb{N}$ *is formally defined using* $0 := \varnothing$*, the successor:* $\text{Succ}(n) = \{n\}$*, so for example,* $3 := \{\{\{\varnothing\}\}\}$.

Definition 1.1.14 (Equivalence Relation)**.** *An equivalence relation on a set X is a relation, denoted $x \sim y$, that has these three properties:*

Reflexivity	$\forall\, x \in X\,,\ x \sim x$
Symmetry	$\forall\, x, y \in X\,,\ x \sim y \ \Leftrightarrow\ y \sim x$
Transitivity	$\forall\, x, y, z \in X\,,\ x \sim y$ and $y \sim z \Rightarrow x \sim z$

Further, we define: $[x] = \{y \in X \mid x \sim y\}$ *and* $X/\sim = \{[x] \mid x \in X\}$

Definition 1.1.15 (Binary Relation)**.** *A subset $R \subseteq A \times B$ is called a relation.*

Example 1.1.16. *$A = B = \{0, 1\}$, we define $R = \{(0, 0), (0, 1), (1, 0)\} \subseteq A \times B$. Why? Because we can.*

Definition 1.1.17 (Function)**.** *A relation $f \subseteq A \times B$ is called a function iff:*

$$\forall\, a \in A\,,\, \exists !\, b \in B : (a,b) \in f$$

Instead of writing $(a,b) \in f$, we write $f(a) = b$. Moreover, $A = D_f$ is called the domain, $B = R_f$ the range and $f \subseteq A \times B$ (sometimes denoted G_f) the table/graph. We denote $f : A \to B$.

Definition 1.1.18 (Composition)**.** *Let $f : A \to B$ and $g : B \to C$ be two functions (sufficient $R_f \subseteq D_g$), the composition $g \circ f$ is the function:*

$$\begin{aligned} h : A &\to C \\ a &\mapsto g\big(f(a)\big) \end{aligned}$$

Definition 1.1.19 (Operation)**.** *An operation $*$ on a set A is a function:*

$$\begin{aligned} * : A \times A &\to A \\ (a,b) &\mapsto a * b \end{aligned}$$

Notice we denote $a * b$ instead of $*(a,b)$. An operation can have (or lack) multiple properties. These are the most important:

Properties	Definition	Structure
Associative	$\forall\, a,b,c \in A\,,\, (a*b)*c = a*(b*c)$	Semigroup
Neutral Element	$\exists\, e \in A : \forall\, a \in A\,,\, a*e = e*a = a$	Monoid
Inverse Element	$\forall\, a \in A\,,\, \exists\, b \in A : a*b = b*a = e$	Group
Commutative	$\forall\, a,b \in A\,,\, a*b = b*a$	Abelian Group

where we named a structure if it has all previous properties. That is, a monoid has an associative operation with identity element.

Lemma 1.1.20. *Composition is associative. That is, $f \circ (g \circ h) = (f \circ g) \circ h$.*

Proof. For any $a \in D_f$, $[f \circ (g \circ h)](a) = f([g \circ h](a)) = f(g(h(a))) = [f \circ g](h(a)) = [(f \circ g) \circ h](a)$. Then, $f \circ (g \circ h)$ and $(f \circ g) \circ h$ are the same function (equality of functions). □

Remark 1.1.21. *$\mathcal{F}(A) = \{f \in \mathcal{P}(A \times A) \mid f : A \to A\}$ is a monoid with composition. The identity is: $\mathrm{id}_A : A \to A$ where $\forall\, a \in A\,,\ \mathrm{id}_A(a) := a$.*

Definition 1.1.22 (Restriction and Extension)**.** *For $f : A \to B$ a function:*
(a) $C \subseteq A$, we define $f|_C : C \to B$ as the restriction of f to C. Formaly, we write $f|_C = \{(a, b) \in f \mid a \in C\} = (C \times B) \cap f$.
(b) $D \supseteq A$, $g : D \to B$ is an extension if $f = g|_A$. Equivalently, $f \subseteq g$.

Definition 1.1.23. *A function $f : A \to B$ is:*
- *Injective if $\forall\, x, y \in A\,,\ f(x) = f(y) \Rightarrow x = y$*
- *Surjective if $\forall\, b \in B\,,\ \exists\, a \in A : f(a) = b$*
- *Bijective if Injective & Surjective*

Lemma 1.1.24. *A function $f : A \to B$ iff bijective iff $\exists\, g : B \to A : g \circ f = \mathrm{id}_A$ and $f \circ g = \mathrm{id}_B$.*

Proof. Observe, by definition f is bijective iff $\forall\, b \in B\,,\ \exists\,!\, a \in A : f(a) = b$. Hence, the relation $g = \{(b, a) \in B \times A \mid f(a) = b\}$ is a function. □

Definition 1.1.25 (Cardinality)**.** *We define $\#A = n$ if there is a bijection (cf. 1.1.23) $\rho : A \to \{1, \cdots, n\}$.*

Remark 1.1.26. *This definition relies on the fact there another no bijections from $\{1, \cdots, n\} \to \{1, \cdots, m\}$ if $n \neq m$.*

Lemma 1.1.27. *We write $A \sqcup B = A \cup B$ if $A \cap B = \varnothing$. Then:*

$$\#(A \sqcup B) = \#A + \#B$$

Proof. Let $\rho_A : A \to \{1, \cdots, n\}$ and $\rho_B : B \to \{1, \cdots, m\}$ be bijections (for $\#A = n$ and $\#B = m$). Then: $\rho : A \sqcup B \to \{1, \cdots, n + m\}$ defined:

$$\rho(c) = \begin{cases} \rho_A(c) & \text{if } c \in A \\ \rho_B(c) + n & \text{if } c \in B \end{cases}$$

which is a bijection, since (cf. 1.1.24) it has an inverse:

$$\rho^{-1}(k) = \begin{cases} \rho_A^{-1}(k) & \text{if } k \leq n \\ \rho_B^{-1}(k - n) & \text{if } k > n \end{cases}$$

□

1.2 Recursion and Induction

Theorem 1.2.1 (Peano Induction)**.** *Let $P \subseteq \mathbb{N}$. Then, if:*

- *Base Case:* $0 \in P$
- *Inductive Step:* $\forall\, n \in \mathbb{N}\,,\, n \in P \Rightarrow n+1 \in P$

Then, $P = \mathbb{N}$. Further, $n \in P$ is called the induction hypothesis.

Remark 1.2.2. *If we want to prove a certain property P is valid for all natural numbers (or after n_0), we let $P = \{n \in \mathbb{N} \mid P(n)\}$ and use induction.*

Corollary 1.2.3 (Stepped Induction)**.** *Let $P \subseteq \mathbb{N}$. Then, if:*

- *Base Case:* $n_0 \in P$
- *Inductive Step:* $\forall\, n \in \mathbb{N}\,,\, n \in P \Rightarrow n+1 \in P$

Then, $\{n \in \mathbb{N} \mid n \geq n_0\} \subseteq P$.

Corollary 1.2.4 (Strong Induction)**.** *Let $P \subseteq \mathbb{N}$. Then, if:*

- *Base Case:* $0 \in P$
- *Inductive Step:* $\forall\, n \in \mathbb{N}\,,\, \{0, \cdots, n\} \subseteq P \Rightarrow n+1 \in P$

Then, $P = \mathbb{N}$.

Theorem 1.2.5 (Counting)**.** *Let $f : A \to B$ be a function*

1. *f is injective $\Rightarrow \#A \leq \#B$*
2. *f is surjective $\Rightarrow \#A \geq \#B$*
3. *f is bijective $\Rightarrow \#A = \#B$*

Proof. We prove each one by induction on $\#A$

1.
   - Base Case: $\#A = 0$, is true that $0 \leq \#B$
   - Induction Step: Let $\#A = n+1$ and $A = A' \sqcup \{a\}$ for $a \in A$ and $B = B' \sqcup \{f(a)\}$. Then $f \cap (A' \times B')$ is injective, so $\#A' \leq \#B'$, by inductive hypothesis. Hence, $\#A = \#A' + 1 \leq \#B' + 1 = \#B$ (cf. 1.1.27).
2.
   - Base Case: $\#A = 0$, is true that $0 \geq 0 = \#B$, since the empty relation is functional iff the image is empty.
   - Induction Step: Let $\#A = n+1$ and $A = A' \sqcup \{a\}$ for $a \in A$ and $B = B' \sqcup \{f(a)\}$. Then $f \cap (A' \times B')$ is surjective, so $\#A' \geq \#B'$, by inductive hypothesis. Hence, $\#A = \#A' + 1 \geq \#B' + 1 = \#B$ (cf. 1.1.27).
3. f is bijective $\Rightarrow \#A \leq \#B$ and $\#A \geq \#B$.

□

Corollary 1.2.6 (Pigeonhole Principle)**.** *For a function $f : A \to B$, if $\#A > \#B$, then $\exists\, x, y \in A : f(x) = f(y)$ and $x \neq y$ (i.e. f is not injective).*

Definition 1.2.7 (Recursion)**.** *A function $f : \mathbb{N} \to \mathbb{N}$ is called recursive if:*

- *Base Cases: $\forall\, k \in \{0, \cdots, n_0\}$, $f(k)$ is defined.*
- *Reduction Rule: $\forall\, n > n_0$, $f(n)$ is an expression involving (not necessarily all) $\{f(n-k) \mid k \in \{0, \cdots, n_0\}\}$.*

Further, f is well-defined by given only these propositions.

1.3 Binary Representation

Theorem 1.3.1 (Modulo and Quotient)**.** *Let $a, b \in \mathbb{N}$, then,*

$$\exists\, ! \, q, r \in \mathbb{N} : a = q \cdot b + r \text{ and } r < b$$

We define $r = \mathrm{mod}(a, b) \in \{0, \cdots, b-1\}$.

Proof. We work by complete induction on a.

- Base Case: $a \in \{0, \cdots, b-1\} : a = b \cdot 0 + a$
- Inductive Step: $a + 1 = (a - b + 1) + b = (q \cdot b + r) + b = (q + 1) \cdot b + r$

Essentially, we compute $q = \lfloor \frac{a}{b} \rfloor$ (rounding down). Further, it is also valid for $a \in \mathbb{Z}$. □

Definition 1.3.2. *Given $n \in \mathbb{N}_{\geq 2}$, we define the following equivalence relation in $\mathbb{Z}$ (cf. 1.1.14):*

$$a \equiv b \mod n \Leftrightarrow \mathrm{mod}(a, n) = \mathrm{mod}(b, n)$$

With $[a]_n = \{b \in \mathbb{Z} \mid a \equiv b \mod n\}$, we define addition and multiplication:

$$[a]_n + [b]_n = [a + b]_n \text{ and } [a]_n \cdot [b]_n = [a \cdot b]_n$$

Which we can check is well-defined (i.e. does not depend on the choice of representative). On the future, we'll just write $a + b \mod n$.

Definition 1.3.3 (Binary String and Concatenation)**.** *We define a n-bit binary string $\vec{A} = A[0\colon n-1] \in \{0,1\}^n$. The set of all string is $\{0,1\}^* = \bigcup_{n \in \mathbb{N}} \{0,1\}^n$. It is a monoid with concatenation operation: $\circ : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ such that $\vec{C} = \vec{A} \circ \vec{B}$, then:*

$$C[i] = \begin{cases} A[i] & \text{if } 0 \leq i < n \\ B[i-n] & \text{if } n \leq i < n + m \end{cases}$$

Moreover, $(\{0,1\}^*, \circ)$ is a monoid.

Remark 1.3.4. *There is a bit of ambiguity on writing $A[0 : n-1]$ and $A[n-1 : 0]$.*

- *For $A[i : j]$, the LSB (least significant bit) is $A[\min\{i, j\}]$*
- *For $A[j : i]$, the MSB (most significant bit) is $A[\max\{i, j\}]$*

All binary represention mathematics heretofore we'll use $A[n-1 : 0]$

Definition 1.3.5. *For $A[n-1:0] \in \{0,1\}^*$, we define the number represented by $\vec{A}$ as:*

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

Remark 1.3.6. $\langle A[n-1:0] \rangle = \langle A[k-1:0] \rangle + 2^k \cdot \langle A[n-1:k] \rangle$

Lemma 1.3.7. *The mapping $\langle \cdot \rangle : \{0,1\}^n \to \{x \in \mathbb{N} \mid 0 \leq x \leq 2^n - 1\}$ is a bijection.*

Proof. Let $x \in \mathbb{N}$, use algorithm 1 or 3. □

Corollary 1.3.8. *The mapping $\langle \cdot \rangle : \{0,1\}^* \to \mathbb{N}$ is a surjection.*

Algorithm 1 BR

```
function BR(k, x ∈ ℕ)
    if x ≥ 2^k then
        return Nothing
    else if k = 1 then
        return x
    else if x ≥ 2^(k−1) then
        return 1 ∘ BR(k − 1, x − 2^(k−1))
    else
        return 0 ∘ BR(k − 1, x)
    end if
end function
```

Algorithm 2 BR'

```
function BR'(k, x ∈ ℕ)
    if x ≥ 2^k then
        return Nothing
    else if k = 1 then
        return x
    else if x is even then
        return BR'(k − 1, x/2) ∘ 0
    else if x is odd then
        return BR'(k − 1, (x − 1)/2) ∘ 1
    end if
end function
```

Lemma 1.3.9. *The following are true:*
(i) $0 \leq \langle A[n-1:0] \rangle \leq 2^n - 1$
(ii) *If* $\langle A[n-1:0] \rangle = \langle B[m-1:0] \rangle$ *with* $m \geq n$, *then* $B = 0^{m-n} \circ A$.

Proof. We prove:
(i) $0 = \sum_{i=0}^{n-1} 0 \cdot 2^i \leq \sum_{i=0}^{n-1} A[i] \cdot 2^i \leq \sum_{i=0}^{n-1} 1 \cdot 2^i = 2^n - 1$
(ii) By definition: $\langle A[n-1:0] \rangle = \langle B[n-1:0] \rangle + 2^n \cdot \langle B[m-1:n] \rangle \Rightarrow 2^n \cdot \langle B[m-1:n] \rangle \leq 2^n - 1 \Rightarrow \langle B[m-1:n] \rangle = 0 \Rightarrow B[m-1:n] = 0^{m-n}$
□

Definition 1.3.10 (Two's Complement)**.** *For* $A[n-1:0] \in \{0,1\}^*$, *we define the number represented by* $\vec{A}$ *in two's complement as:*

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

Algorithm 3 TCR

```
function TCR(n, x ∈ ℕ)
    if x ≥ 2^{n-1} ∨ x ≤ -2^{n-1} - 1 then
        return Nothing
    else if x ≥ 0 then
        return 0 ∘ BR(n - 1, x)
    else if x < 0 then
        return BR(n, x + 2^n)
    end if
end function
```

Lemma 1.3.11. *The following are true:*
(i) $-2^{n-1} \leq \big[A[n-1:0]\big] \leq 2^{n-1} - 1$
(ii) $-\big[A[n-1:0]\big] = \big[\text{INV}(A[n-1:0])\big] + 1$

Proof. We prove:
(i) $-2^{n-1} + 0 \leq -2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle \leq 0 + 2^{n-1} - 1$
(ii) $\big[A[n-1:0]\big] + \big[\text{INV}(A[n-1:0])\big] = -\big(A[n-1] + \text{INV}(A[n-1])\big)\, 2^{n-1} + \sum_{i=0}^{n-2} \big(A[i] + \text{INV}(A[i])\big) \cdot 2^i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i = 1$
□

Lemma 1.3.12 (Sign Extension)**.** *The following are true:*
(i) $\big[A[n-1:0]\big] < 0 \Leftrightarrow A[n-1] = 1$

(ii) $A[n] = A[n-1] \Rightarrow \big[A[n:0]\big] = \big[A[n-1:0]\big]$

Proof. We prove:

(i) $A[n-1] = 0 \Rightarrow \big[A[n-1:0]\big] = \langle A[n-2:0]\rangle \geq 0$ and $A[n-1] \Rightarrow -\big[A[n-1:0]\big] = \big[\textsc{inv}(A[n-1:0])\big] + 1 \geq 1 \Rightarrow \big[A[n-1:0]\big] \leq -1$.

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

□

Corollary 1.3.13. $\big[A[n-1]^k \circ A[n-1:0]\big] = \big[A[n-1:0]\big]$ *we denote the sign extension to* $n+k$ *bits:* $sext_{n+k}(A[n-1:0]) := A[n-1]^k \circ A[n-1:0]$.

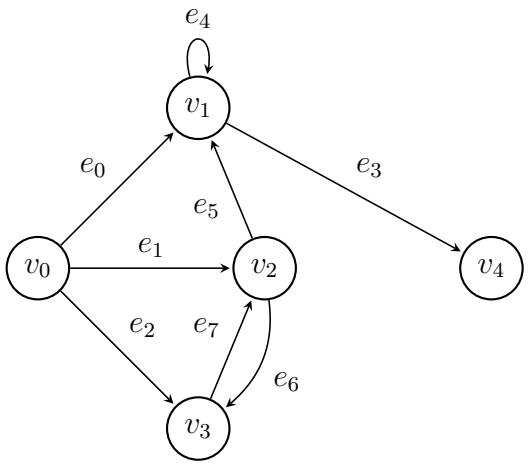
1.4 Graph Theory

Definition 1.4.1 (Directed Graph)**.** *Let V be a finite set and $E \subseteq V \times V$. The pair $G = (V, E)$ is called a directed graph. We say $v \in V$ is a vertex/node and $e = (u, v) \in E$ is an (directed) edge.*

Example 1.4.2. *The set $V = \{0, 1, 2, 3, 4\}$ and*

$$E = \{(0,1), (0,2), (0,3), (1,4), (1,1), (2,1), (2,3), (3,2)\}$$

which we can draw as follows:



Notice we allow for self-loops like $(1, 1)$ and 2-cycles like $(2, 3)$ and $(3, 2)$, but we do not allow repeated edges.

Definition 1.4.3 (Subgraph)**.** *For $G = (V, E)$ a graph, let $U \subseteq V$, we denote $G[U] = (U, \{(u, v) \in E \mid u, v \in U\})$ the induced subgraph.*

Definition 1.4.4 (Path)**.** *A path of length ℓ (that is, $\ell = \#\Gamma$) in $G = (V, E)$ is a pair $\Gamma \in V^{\ell+1}$ such that $\forall\, i \in \{0, \cdots, \ell - 1\}$, $(v_i, v_{i+1}) \in E$. We denote Γ by: $v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} v_2 \cdots v_{\ell-1} \xrightarrow{e_{\ell-1}} v_\ell$ or $v_0 \xrightarrow{\Gamma} v_\ell$*

Remark 1.4.5. *There is always a path of length 1 from v to v, $\Gamma = (v, \varnothing)$.*

Definition 1.4.6 (Closed and Simple Paths)**.** *A path is closed (called a cycle) if $v_0 = v_\ell$ and is open otherwise. An open path is simple if every element of $\mathcal{V}$ are distinct. Likewise, a cycle is simple if every element of $\mathcal{V} \setminus (v_\ell)$ are distinct.*

Lemma 1.4.7. *A non-simple path contains a cycle.*

Proof. If Γ is a cycle, we are done. Else, if Γ is open and non-simple, let $v_i = v_j \in \mathcal{V}$. Then, $v_i \xrightarrow{\Gamma} v_j = v_i$ is a cycle. □

Definition 1.4.8 (DAG)**.** *A directed acyclic graph (DAG) is a directed graph that has no cycles.*

Corollary 1.4.9. *Every path in a DAG is simple.*

Definition 1.4.10 (Degrees, Sources and Sinks)**.** *Given a graph $G = (V, E)$ and $v \in V$, we define:*

$$\deg_{in}(v) = \#\{e \in E \mid \exists\, u \in V : e = (u, v)\}$$
$$\deg_{out}(v) = \#\{e \in E \mid \exists\, u \in V : e = (v, u)\}$$

*Further, if $\deg_{in}(v) = 0$, v is called a **source**. If $\deg_{out}(v) = 0$, v is called a **sink**.*

Theorem 1.4.11. *Every DAG has at least one sink.*

Proof. By contrary, suppose G has no sink, that is, $\forall\, v \in V$, $\deg_{\text{out}}(v) > 0$. Take $v_0 \in V$. Since $\deg_{\text{in}}(v_0) > 1$, take $v_1 \in V : e_0 = (v_0, v_1) \in E$. We can continue to create a path $v_0 \xrightarrow{\Gamma} v_\ell$ of arbitrary length. If Γ is simple, take $\ell = \#V$, hence, the path has more vertices than there are in V, contradiction. If Γ is not simple, then G has a cycle, contradiction. □

Corollary 1.4.12. *Every DAG has at least one source.*

Proof. Take $G' = (V, E')$ where

$$E' = \{(v, u) \in V \times V \mid (u, v) \in E\}$$

the reversed graph. Notice G' is also acylic, since a cycle in G' implies a cycle in G with the edges and the order reversed. By the theorem, there is a sink u. Since $\forall\, v \in V$, $\deg'_{\text{in}}(v) = \deg_{\text{out}}(v)$, we get: $\deg_{\text{out}}(u) = \deg'_{\text{in}}(u) = 0$. □

Lemma 1.4.13. *An induced subgraph (cf. 1.4.3) of a DAG is still a DAG.*

Proof. If there is a cycle in $G[U]$, then the same cycle exists in G, since $U \subseteq V$ and $E_U \subseteq E$. Hence, $G[U]$ has no cycles. □

Definition 1.4.14 (Topological Ordering)**.** *Let $G = (V, E)$ be a graph. A topological ordering is map $\pi : V \to \{0, \cdots, \#V - 1\}$ such that:*

$$\forall\, u, v \in V\,,\ (u, v) \in E \Rightarrow \pi(u) < \pi(v)$$

Remark 1.4.15 (Topological Sorting)**.** *Since $\pi : V \to \{0, \cdots, \#V - 1\}$ is a bijection, we can write $V = (v_0, v_1, \cdots, v_{\#V-1})$ where $v_i = \pi^{-1}(i)$.*

Algorithm 4 Kahn's Algorithm

```
function TO(G = (V, E))
    if #V = 1 then
        return π(v_0) = 0
    else
        s ← sink(V, E)
        return TO(G[V \ {s}]) with π(s) = #V − 1
    end if
end function
```

Theorem 1.4.16. *A graph is a DAG iff it has a topological ordering.*

Proof. We prove both directions:

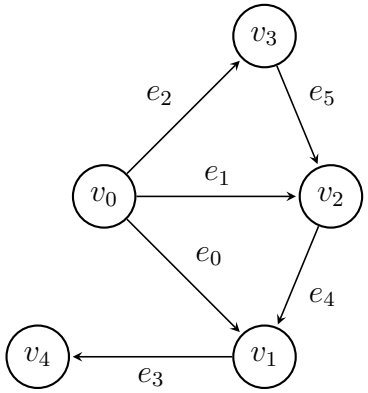
($\Leftarrow$) Suppose G has a topological ordering π. Take a path $\Gamma : v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} v_2 \cdots v_{\ell-1} \xrightarrow{e_{\ell-1}} v_\ell$, then $\pi(v_0) < \pi(v_1) < \cdots < \pi(v_{\ell-1}) < \pi(v_\ell)$. Hence, for any path $\pi(v_0) \neq \pi(v_\ell)$, so G has no cycles.

($\Rightarrow$) We'll prove that algorithm 4 produces a topological ordering by induction. The base case is true by construction of the function. Let $\#V = n + 1$. Now, suppose $\textsc{to}(G[V \setminus \{s\}])$ is a topological ordering $\pi : V \setminus \{s\} \to \{0, \cdots, n - 1\}$, where $s = \text{sink}(V, E)$, which exists by theorem. Since π is a bijection, extending to $\pi(s) = \#V - 1 = n$ is still a bijection. Notice if $(v, s) \in E \Rightarrow v \neq s \Rightarrow \pi(v) < \pi(s) = n$, further, $\nexists\, v \in V : (s, v) \in E$, since s is a source. Therefore, since π is already a topological ordering of $G[V \setminus \{s\}]$, the extension is also a topological ordering.

□

Remark 1.4.17. *Observe that algorithm 4 relies on the fact that there is a sink in G (cf. 1.4.11) and that the induced subgraph is also a DAG (cf. 1.4.13).*

Example 1.4.18. *The following is the result after applying the algorithm:*



v	$\pi(v)$
v_0	0
v_1	3
v_2	2
v_3	1
v_4	4

Definition 1.4.19 (Longest Path)**.** *A path Γ is a longest path of G if*

$$\forall\, \Gamma' \text{ path in } G\,,\ \#\Gamma' \leq \#\Gamma$$

Theorem 1.4.20. *If $G = (V, E)$ is a DAG, then $\forall\, v \in V$, there is some longest path that ends in v. Further, there exists some longest path in G.*

Proof. Since any path of length bigger than $\#V$ is not simple (which G does not have), we get:

$$\forall\, \Gamma \text{ path in } G\,,\ \#\Gamma \leq \#V - 1$$

Since there are a finite number of paths end in $v \in V$, there must be some biggest element. The same argument follows for any path in G. □

Definition 1.4.21 (Delay Function)**.** *Given $G = (V, E)$ a DAG, we define $d : V \to \mathbb{N}$, so that $\forall\, v \in V$, $d(v) =$* length of longest path ending in v*, which exist cf. 1.4.20.*

Algorithm 5 Longest Path Algorithm

$(v_0, \cdots, v_{n-1}) \leftarrow \text{TS}(V, E)$
for $0 \leq j \leq n-1$ **do**
  if v_j is a source **then**
    $d(v_j) \leftarrow 0$
  else
    $d(v_j) \leftarrow 1 + \max\{d(v_i) \mid i < j \text{ and } (v_i, v_j) \in E\}$
  end if
end for

Theorem 1.4.22. *The algorithm 5 produces the delay function, as in 1.4.21.*

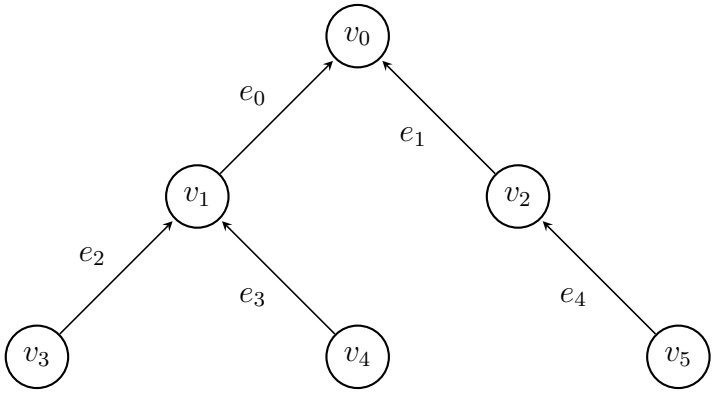
Proof. We prove by induction on j. By topological sorting, v_0 is a source, so $d(v_0) = 0$. Suppose $\forall\, i \in \mathbb{N} : 0 \leq i \leq j$, $d(v_i) =$ length of longest path ending in v_i. We'll prove the inductive step. If v_{j+1} is a source, we are done. Else, there are v_i such that $(v_i, v_j) \in E$. Since they are sorted, $\forall\, i \in \mathbb{N} : (v_i, v_j) \in E$, $0 \leq i \leq j$, so the longest path that ends in v_{j+1} must pass through some v_i for $0 \leq i \leq j$, and we know how to calculate each delay $d(v_i)$ (by hypothesis). Therefore, $1 + \max\{d(v_i) \mid i \leq j \text{ and } (v_i, v_j) \in E\}$ computes the desired delay of v_{j+1}. □

Definition 1.4.23 (Rooted Tree)**.** *A rooted tree $G = (V, E)$ is a DAG s.t.:*

1. *There only one sink r in G*
2. $\forall\, v \in V \setminus \{r\}$, $\deg_{out}(v) = 1$

We call the sink $r = r(G)$ the root of G.

Example 1.4.24. $V = \{0, 1, 2, 3, 4, 5\}, E = \{(1, 0), (2, 0), (3, 1), (4, 1), (5, 2)\}$ *such that $r = 0$:*



Lemma 1.4.25. *If G is a rooted tree, $\#E = \#V - 1$.*

Proof. We shall do induction on $\#V$. The base case is $\#V = 1$, so $\#E = 0$. For the step, suppose any rooted tree with n vertices has $n - 1$ edges. Take a rooted tree G with $\#V = n + 1$ and a source s of G. Since $\deg_{\text{in}}(s) = 0$ and $\deg_{\text{out}}(s) = 1$, there is only on edge in E that contains s. Hence, since $G[V \setminus \{v\}]$ is a rooted tree with with n vertices (because we only removed one edge and one vertex), it has $n - 1$ edges. Therefore G has n edges. □

Theorem 1.4.26. *In a rooted tree $G = (V, E)$, for every vertex $v \in V$, there exists an unique path from v to $r(G)$.*

Proof. The case $v = r$ is trivial. To prove there must exist a path for any $v \in V \setminus \{r\}$, we take a path $v = v_0 \xrightarrow{\Gamma} v_\ell$ of arbitrary length ℓ. If v_ℓ is a sink, we are done since $v_\ell = r$. If v_ℓ is not a sink, by the definition of a rooted tree, there is a vertex $v_{\ell+1}$ such that $(v_\ell, v_{\ell+1}) \in E$, so that $\deg_{\text{out}}(v) = 1$. Therefore, we can continue extending the path. Take $\ell = \#V - 1$ at the limiting case, the path has all the vertices, so $v_\ell = r$. To prove uniqueness, if we take two paths $v = v_0 \xrightarrow{\Gamma} v_\ell = r$ and $v = u_0 \xrightarrow{\Gamma'} u_\ell = r$. By definition, $\forall\, v \in V \setminus \{r\}$, $\deg_{\text{out}}(v) = 1$, so $v_1 = u_1$, $v_2 = u_2, \cdots$, $v_\ell = u_\ell \Rightarrow \Gamma = \Gamma'$. □

Definition 1.4.27. *Let $\{r_i\}_{i=1}^k = \{r_i \in V \mid (r_i, r) \in E\}$, where $k = \deg_{in}(r)$. Define:*

$$V_i = \left\{ v \in V \;\middle|\; \exists\, \Gamma \text{ path in } G : v \xrightarrow{\Gamma} r_i \right\}$$

and $G_i = (V_i, E_i) = G[V_i]$.

Remark 1.4.28. *$r_i \in V_i$ (cf. 1.4.5)*

Lemma 1.4.29. *For $G = (V, E)$ a rooted tree:*

1. *$V = \{r\} \sqcup \left(\bigsqcup_{i=1}^k V_i\right)$*
2. *The graph G_i is a rooted tree with $r(G_i) = r_i$ for every $1 \le i \le k$*

Proof. We prove each one:

1. For $i \neq j$, if $\exists\, v \in V_i \cap V_j$, then there are two different paths from v to r: $v \xrightarrow{\Gamma} r_i \xrightarrow{e_i} r$ and $v \xrightarrow{\Gamma'} r_j \xrightarrow{e_j} r$. Contradiction (cf. 1.4.26). Further since $\forall\, v \in V \setminus \{r\}$, there is a path to r, that path must go through one of the r_i. Hence, $v \in V \Rightarrow v \in V_i$ for some i or $v = r$.
2. We need to prove the only sink is r_i, since G_i is a DAG (cf. 1.4.13). Now, r_i is a sink, since it's only out-edge was $(r_i, r) \notin E_i$. Further, by definition, $\forall\, v \in V \setminus \{r_i\}$, $\deg_{\text{out}}(v) > 0$, so r_i is the only sink.

□

Definition 1.4.30 (Leaf)**.** *In a rooted tree, a source is called a leaf. Any other vertice is called an interior vertex.*

1.5 Asymptotics

Definition 1.5.1. *For $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$, define the following notation (symbolic equality):*

(i) $f(n) = \mathcal{O}(g(n)) \Leftrightarrow \exists\, A \in \mathbb{R}_{>0}, N \in \mathbb{N} : \forall\, n \geq N\,,\ f(n) \leq A \cdot g(n)$

(ii) $f(n) = \Omega(g(n)) \Leftrightarrow \exists\, A \in \mathbb{R}_{>0}, N \in \mathbb{N} : \forall\, n \geq N\,,\ f(n) \geq A \cdot g(n)$

(iii) $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \mathcal{O}(g(n))$ *and* $f(n) = \Omega(g(n))$

The intuition is:

(i) $f(n) = \mathcal{O}(g(n))$: f *does not grow faster than* g

(ii) $f(n) = \Omega(g(n))$: f *grows at least as fast as* g

(iii) $f(n) = \Theta(g(n))$: f *grows as fast as* g

Remark 1.5.2. *The functions f and g may not be defined for finite number of natural and the definition should still work.*

Lemma 1.5.3. *$f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$:*

$$g(n) = \Omega(f(n)) \Leftrightarrow f(n) = \mathcal{O}(g(n))$$

Proof. By definition (cf. 1.5.1), $f(n) \geq A \cdot g(n) \Leftrightarrow g(n) \leq \frac{1}{A} \cdot f(n)$, the result follows. □

Corollary 1.5.4. $g(n) = \Theta(f(n)) \Leftrightarrow f(n) = \Theta(g(n))$

Lemma 1.5.5. *For $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$:*

(i) $f(n) = \mathcal{O}(g(n)) \Rightarrow \exists\, A \in \mathbb{R}_{>0}, B \in \mathbb{R}_{\geq 0} : \forall\, n \in \mathbb{N}\,,\ f(n) \leq A \cdot g(n) + B$

(ii) $f(n) = \Omega(g(n)) \Leftrightarrow \exists\, A \in \mathbb{R}_{>0}, B \in \mathbb{R}_{\geq 0} : \forall\, n \in \mathbb{N}\,,\ f(n) \geq A \cdot g(n) + B$

Moreover, if $\forall\, n \in \mathbb{N}\,,\ g(n) \geq 1$, the converse of (i) is true.

Proof. We prove:

(i) $f(n) = \mathcal{O}(g(n)) \Leftrightarrow \exists\, A \in \mathbb{R}_{>0}, N \in \mathbb{N} : \forall\, n \geq N\,,\ f(n) \leq A \cdot g(n) \Rightarrow$ define $B = \max\{f(n) \mid 0 \leq n < N\}$, so $\forall\, n \in \mathbb{N}\,,\ f(n) \leq A \cdot g(n) + B$

(ii) $f(n) = \Omega(g(n)) \Leftrightarrow \exists\, A \in \mathbb{R}_{>0}, N \in \mathbb{N} : \forall\, n \geq N\,,\ f(n) \geq A \cdot g(n) \Rightarrow$ define $B = \min\left\{A, \min\left\{\frac{f(n)}{g(n)} \;\middle|\; 0 \leq n < N \text{ and } g(n) \neq 0\right\}\right\}$, so that we have: $\forall\, n \in \mathbb{N}\,,\ f(n) \geq B \cdot g(n)$

The converse is given by:

(i) $f(n) \leq A \cdot g(n) + B \leq (A + B) \cdot g(n) \Rightarrow f(n) = \mathcal{O}(g(n))$ (supposing $g(n) \geq 1$)

(ii) $f(n) \geq A \cdot g(n) + B \geq A \cdot g(n) \Rightarrow f(n) = \Omega(g(n))$

□

Theorem 1.5.6. *Let f and g be both monotonically non-decreasing. Then if* $\exists\, \rho \in \mathbb{R}_{>0} : \forall\, k \in \mathbb{N}$,

(i) $\rho \geq \dfrac{g(2^{k+1})}{g(2^k)} \Rightarrow [f(2^k) = \mathcal{O}(g(2^k)) \Rightarrow f(n) = \mathcal{O}(g(n))]$

(ii) $\rho \leq \dfrac{g(2^{k+1})}{g(2^k)} \Rightarrow [f(2^k) = \Omega(g(2^k)) \Rightarrow f(n) = \Omega(g(n))]$

Moreover, if $\dfrac{g(2^{k+1})}{g(2^k)}$ *is bounded, then* $[f(2^k) = \Theta(g(2^k)) \Rightarrow f(n) = \Theta(g(n))]$.

Proof. Suppose

(i) $f(2^k) = \mathcal{O}(g(2^k)) \Leftrightarrow \exists\, A \in \mathbb{R}_{>0}, K \in \mathbb{N} : \forall\, k \geq K\,,\ f(2^k) \leq A \cdot g(2^k)$. For $n \geq 2^K$, $2^k \leq n < 2^{k+1}$ for some $k \geq K$. Since f and g are monotonically non-decreasing:

$$f(n) \leq f(2^{k+1}) \leq A \cdot g(2^{k+1}) \leq \rho \cdot A \cdot g(2^k) \leq \rho \cdot A \cdot g(n)$$

Hence, $\forall\, n \geq 2^K\,,\ f(n) \leq \rho \cdot A \cdot g(n)$, so, $f(n) = \mathcal{O}(g(n))$.

(ii) $f(2^k) = \Omega(g(2^k)) \Leftrightarrow \exists\, A \in \mathbb{R}_{>0}, K \in \mathbb{N} : \forall\, k \geq K\,,\ f(2^k) \geq A \cdot g(2^k)$. For $n \geq 2^K$, $2^k \leq n < 2^{k+1}$ for some $k \geq K$. Since f and g are monotonically non-decreasing:

$$f(n) \geq f(2^k) \geq A \cdot g(2^k) \geq \frac{A}{\rho} \cdot g(2^{k+1}) \geq \frac{A}{\rho} \cdot g(n)$$

Hence, $\forall\, n \geq 2^K\,,\ f(n) \geq \dfrac{A}{\rho} \cdot g(n)$, so, $f(n) = \Omega(g(n))$.

If $\dfrac{g(2^{k+1})}{g(2^k)}$ is bounded, both conditions apply. □

2 Boolean Logic

2.1 Prepositional Logic

Definition 2.1.1 (Boolean Functions)**.** *A function $B : \{0,1\}^n \to \{0,1\}^k$, is called a Boolean function.*

Definition 2.1.2 (Four Basis Boolean Functions)**.** *Define:*

x	$\text{NOT}(x)$
0	1
1	0

x	y	$\text{AND}(x,y)$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$\text{OR}(x,y)$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$\text{XOR}(x,y)$
0	0	0
0	1	1
1	0	1
1	1	0

With that, we can define $\text{NAND} = \text{NOT} \circ \text{AND}$, $\text{NOR} = \text{NOT} \circ \text{OR}$, $\text{XNOR} = \text{NOT} \circ \text{XOR}$. *Further, we write the operations:*

Negation: $\neg x := \text{NOT}(x)$
Conjunction: $x \wedge y := \text{AND}(x,y)$
Disjunction: $x \vee y := \text{OR}(x,y)$

Lemma 2.1.3. $\forall\, x \in \{0,1\}$,

- $x \wedge 1 = x$
- $x \wedge 0 = 0$
- $x \vee 1 = 1$
- $x \vee 0 = x$

Proof. Follows directly from the definition, looking at the truth tables. □

Lemma 2.1.4 (Boolean Monoids)**.** *We have:*

All: $(\{0,1\}, \wedge)$ *is a monoid with identity* 1
Any: $(\{0,1\}, \vee)$ *is a monoid with identity* 0

Proof. For associativity, observe $x \wedge (y \wedge z) = (x \wedge y) \wedge z = 1 \Leftrightarrow x = y = z = 1$ and $x \vee (y \vee z) = (x \vee y) \vee z = 0 \Leftrightarrow x = y = z = 0$. Hence, the truth tables are the same. For the identities, we use the relation in 2.1.3. □

Definition 2.1.5. *Since the formulas are associate, we define:*

- $\text{AND}_n(x_1, \cdots, x_n) = \text{AND}(\text{AND}_{n-1}(x_1, \cdots, x_{n-1}), x_n) = x_1 \wedge \cdots \wedge x_n$
- $\text{OR}_n(x_1, \cdots, x_n) = \text{OR}(\text{OR}_{n-1}(x_1, \cdots, x_{n-1}), x_n) = x_1 \vee \cdots \vee x_n$

Definition 2.1.6 (Boolean Connectives)**.** *A boolean formula has a set U of variables and a set $\mathcal{C}$ of connectives. A connective is an operation to build larger formulas. The arity of the connective is the number of arguments it receives.*

Remark 2.1.7. *Every connective γ is exacty one boolean function $B_\gamma : \{0,1\}^k \to \{0,1\}$. There is only one (non-trivial) unary connective* NOT *(cf. 2.1.2), also denoted $\neg$. Further there a many binary connectives, e.g.* AND *(denoted $\wedge$),* OR *(denoted $\vee$), material implication ($\to$) etc.*

Definition 2.1.8 (Parse Tree)**.** *A parse tree is a pair (G, π) where $G = (V, E)$ is a rooted tree and $\pi : V \to \{0,1\} \cup U \cup \mathcal{C}$ is a labeling function such that:*

1. *If v is a leaf (cf. 1.4.30), $\pi(v) \in \{0,1\} \cup U$*
2. *If v is not a leaf, $\pi(v) \in \mathcal{C}$ and $\deg_{in}(v) =$ arity of $\pi(v)$.*

Algorithm 6 In Order

```
function INORDER(G = (V, E), π)
    n ← deg_in(r(G))
    if #V = 1 then
        return π(v_0)
    else
        G_i = (V_i, E_i) : subtrees hanging from r(G)
        π_i ← π restricted to V_i.
        φ_i ← INORDER(G_i, π_i)
        if n = 1 then return (π(r(G)) φ_1)
        else if n = 2 then return (φ_1 π(r(G)) φ_2)
        else return (π(r(G))(φ_1 , ··· , φ_n))
        end if
    end if
end function
```

Definition 2.1.9. *A Boolean Formula is the resulting string for the algorithm 6. The set of all boolean formula is with variables U and connectives $\mathcal{C}$ is denoted $\mathcal{BF}(U, \mathcal{C})$.*

Remark 2.1.10. *Algorithms that return as parse tree for a string are called parsing algorithms. We suppose those algorithms are intuitive and we need not explain how to parse. We may denote* $G_\varphi = (V_\varphi, E_\varphi), \pi_\varphi$ *for the parse tree of* $\gamma \in \mathcal{BF}(U, \mathcal{C})$ *and* $\#\varphi = \#V_\varphi$.

Lemma 2.1.11. $\mathcal{BF}(U, \mathcal{C})$ *is the smallest set that contains* $\{0, 1\} \cup U$ *(called the atoms) and is closed under* $\mathcal{C}$*. That is, building any two formulas in* $\mathcal{BF}(U, \mathcal{C})$ *using a connective* $\mathcal{C}$ *is still in* $\mathcal{BF}(U, \mathcal{C})$.

Proof. First, $\mathcal{BF}(U, \mathcal{C})$ contains the atoms, for $x \in \{0, 1\} \cup U$, take $(G_x = (\{v\}, \varnothing), \pi_x)$ where $\pi_x(v) = x$. Now, if we take $\varphi_1, \cdots, \varphi_k \in \mathcal{BF}(U, \mathcal{C})$ and $\gamma \in \mathcal{C}$ with arity k. Then, we can make $\gamma(\varphi_1, \cdots, \varphi_k) \in \mathcal{BF}(U, \mathcal{C})$. First, parse (cf. 2.1.10) φ_i into (G_i, π_i). Take $V = \{r\} \cup \left(\bigcup_{i=1}^k V_i\right)$ and $E = \{(r(G_i), r) \mid 1 \leq i \leq k\} \cup \left(\bigcup_{i=1}^k E_i\right)$. Further, label:

$$\pi(v) = \begin{cases} \gamma & \text{if } v = r \\ \pi_i(v) & \text{if } v \in V_i \end{cases}$$

Then, $(G = (V, E), \pi)$ is the parse tree for $\gamma(\varphi_1, \cdots, \varphi_k)$. Hence, by 2.1.9, $\gamma(\varphi_1, \cdots, \varphi_k) \in \mathcal{BF}(U, \mathcal{C})$. Therefore, $\mathcal{BF}(U, \mathcal{C})$ is exactly all the boolean formulas generated by taking the atoms and building with $\mathcal{C}$. □

2.2 Evaluation and Representation

Remark 2.2.1. *Every connective $\gamma \in \mathcal{C}$ is boolean function, we which we denote $B_\gamma : \{0,1\}^k \to \{0,1\}$, where k is the arity of γ.*

Definition 2.2.2 (Truth Assigment)**.** *A function $\tau : U \to \{0,1\}$ is an assigment on U. Moreover, we extend τ to $\hat{\tau} : \mathcal{BF}(U,\mathcal{C}) \to \{0,1\}$ described in algorithm 7:*

$$\hat{\tau}(\varphi) = \text{EVAL}(G_\varphi, \pi_\varphi, \tau)$$

which we may also write $\hat{\tau}(\varphi) = \text{EVAL}(\varphi, \tau)$.

Algorithm 7 Evaluate

```
function EVAL(G = (V, E), π, τ)
    n ← deg_in(r(G))
    if #V = 1 then
        if π(r(G)) ∈ {0, 1} then return π(r(G))
        else return τ(π(r(G)))
        end if
    else
        G_i = (V_i, E_i) : subtrees hanging from r(G)
        π_i ← π restricted to V_i.
        σ_i ← EVAL(G_i, π_i, τ)
        return B_{π(r(G))}(σ_1, ··· , σ_n)
    end if
end function
```

Definition 2.2.3. *Given an ordered set of variables $U = \{X_1, X_2, \cdots, X_n\}$ and $\varphi \in \mathcal{BF}(U,\mathcal{C})$, we define the Boolean function $B_\varphi : \{0,1\}^n \to \{0,1\}$ as:*

$$B_\varphi(v) = \hat{\tau}_v(\varphi) = \text{EVAL}(\varphi, \tau_v)$$

where τ_v, for $v \in \{0,1\}^n$, is defined so that $\tau_v(X_i) = v[i]$. Of course, every assigment is given by τ_v for some $v \in \{0,1\}^n$.

Lemma 2.2.4. *For $\varphi = \gamma(\varphi_1, \cdots, \varphi_n)$, with $\gamma \in \mathcal{C}$,*

$$\forall\, v \in \{0,1\}^n\,,\; B_\varphi(v) = B_\gamma(B_{\varphi_1}(v), \cdots, B_{\varphi_n}(v))$$

Proof. By 2.2.3, $\forall\, \tau : U \to \{0,1\}\,,\ \hat{\tau}(\varphi) = B_\gamma(\hat{\tau}(\varphi_1), \cdots, \hat{\tau}(\varphi_n))$, directly from the definition in algorithm 7. □

Remark 2.2.5. *We use the same notation as 2.2.1, since for a formula $\varphi = \gamma(X_1, \cdots, X_n)$, the evaluation shall give $B_\varphi = B_\gamma$.*

Definition 2.2.6. *The substitution $\varphi = \phi(\varphi_1, \cdots, \varphi_n)$, with $\varphi_i \in \mathcal{BF}(U, \mathcal{C})$ and $\phi \in \mathcal{BF}(\{X_1, \cdots, X_n\}, \mathcal{C})$, is defined by replacing each leaf node of G_ϕ labeled X_i with a copy of G_{φ_i} to get G_φ. Naturally, π is inherited from the substitutions. By this construction, $\varphi \in \mathcal{BF}(U, \mathcal{C})$.*

Theorem 2.2.7 (Substitution)**.** *The substitution $\varphi = \phi(\varphi_1, \cdots, \varphi_n)$:*

$$\forall\, \tau : U \to \{0,1\}\,,\ \hat{\tau}(\varphi) = B_\phi(\hat{\tau}(\varphi_1), \cdots, \hat{\tau}(\varphi_n))$$

equivalently, $\forall\, v \in \{0,1\}^n\,,\ B_\varphi(v) = B_\phi(B_{\varphi_1}(v), \cdots, B_{\varphi_n}(v))$

Proof. We'll use complete induction on the number of vertices in G_ϕ.

- Base Case: If $\#V_\phi = 1$, then B_ϕ is either a projection ($B_\phi(v) = v[i]$ for some i) or a constant. Further, either $\varphi = \varphi_i$ or $\varphi = \phi$. Either way, the statement holds.
- Inductive Step: Let $\phi = \gamma(\psi_1, \cdots, \psi_k)$. Naturally, it is the case that $\varphi = \gamma(\psi_1(\varphi_1, \cdots, \varphi_n), \cdots, \psi_k(\varphi_1, \cdots, \varphi_n))$. By definition (cf. alg 7), $\hat{\tau}(\varphi) = B_\gamma(\hat{\tau}(\psi_1(\varphi_1, \cdots, \varphi_n)), \cdots, \hat{\tau}(\psi_k(\varphi_1, \cdots, \varphi_n)))$. Moreover, by hypothesis, each ψ_i holds

  $$\forall\, \tau : U \to \{0,1\}\,,\ \hat{\tau}(\psi_i(\varphi_1, \cdots, \varphi_n)) = B_{\psi_i}(\hat{\tau}(\varphi_1), \cdots, \hat{\tau}(\varphi_n))$$

  Hence $\hat{\tau}(\varphi) = B_\gamma(B_{\psi_1}(\hat{\tau}(\varphi_1), \cdots, \hat{\tau}(\varphi_n)), \cdots, B_{\psi_k}(\hat{\tau}(\varphi_1), \cdots, \hat{\tau}(\varphi_n))) = B_\phi(\hat{\tau}(\varphi_1), \cdots, \hat{\tau}(\varphi_n))$, as required (cf. 2.2.4).

□

Definition 2.2.8. *We say $B : \{0,1\}^k \to \{0,1\}$ is representable or expressible with $\mathcal{C}$ if $\exists\, \varphi \in \mathcal{BF}(\{X_1, \cdots, X_k\}, \mathcal{C}) : B \equiv B_\varphi$. The formula φ is a represention of B. Furthermore, A set of connectives $\mathcal{C}$ is complete iff: $\forall\, B : \{0,1\}^k \to \{0,1\}\,,\ B$ is expressible with $\mathcal{C}$.*

Lemma 2.2.9. *If $\mathcal{C}$ is complete and $\forall\, \gamma \in \mathcal{C}\,,\ B_\gamma$ is expressible with $\mathcal{C}'$, then $\mathcal{C}'$ is complete.*

Proof. Follows directly from 2.2.7. □

Theorem 2.2.10. *Every boolean function $B : \{0,1\}^k \to \{0,1\}$ is expressible with* NOT, OR *and* AND. *That is,* $\{\text{NOT}, \text{OR}, \text{AND}\}$ *is complete.*

Proof. We prove by induction on the arity of B.

- Base Case: $n = 1$

Formula $\backslash x$	0	1
$x \wedge (\neg x)$	0	0
x	0	1
$\neg x$	1	0
$x \vee (\neg x)$	1	1

- Induction Step: Observe: For $x \in \{0,1\}$ and $xs \in \{0,1\}^{k-1}$

$$B(x \circ xs) = (x \wedge B_1(xs)) \vee ((\neg x) \wedge B_0(xs))$$

For $B_0(xs) = B(0 \circ xs)$ and $B_1(xs) = B(1 \circ xs)$, which are functions $B_0, B_1 : \{0,1\}^{k-1} \to \{0,1\}$. By induction, they can be represented as a combination of NOT, OR and AND.

□

Remark 2.2.11. *The induction step in 2.2.10 depends on the implementation of the function* $\text{IF} : \{0,1\}^3 \to \{0,1\}$ *where* $\text{IF}(1,x,y) = x$ *and* $\text{IF}(0,x,y) = y$, $\text{IF}(c,x,y)$ *read as "if c then x else y". The truth table is:*

c	x	y	$\text{IF}(c,x,y)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

The example was given by $\text{IF}(c,x,y) = (c \wedge x) \vee ((\neg c) \wedge y)$

Corollary 2.2.12. *Due to 2.3.1, both* $\{\text{NOT}, \text{OR}\}$ *and* $\{\text{NOT}, \text{AND}\}$ *are complete.*

Corollary 2.2.13. $\mathcal{C}$ *is complete iff* NOT *and at least one of* AND *or* OR *is expressible with* $\mathcal{C}$.

Theorem 2.2.14 (Universal Functions)**.** *Define the following boolean functions:* NAND = NOT ∘ AND *and* NOR = NOT ∘ OR*.*

x	y	NAND(x,y)
0	0	1
0	1	1
1	0	1
1	1	0

x	y	NOR(x,y)
0	0	1
0	1	0
1	0	0
1	1	0

Then, {NAND} *and* {NOR} *are both complete.*

Proof. We use 2.2.10:

- NAND: $\text{NAND}(x,x) = \text{NOT}(x)$ & $\text{NOT}(\text{NAND}(x,y)) = \text{AND}(x,y)$
- NOR: $\text{NOR}(x,x) = \text{NOT}(x)$ & $\text{NOT}(\text{NOR}(x,y)) = \text{OR}(x,y)$

□

Definition 2.2.15. *A boolean formula* $\varphi \in \mathcal{BF}(U,\mathcal{C})$ *is a tautology iff*

$$\forall\, \tau : U \to \{0,1\}\,,\ \hat{\tau}(\varphi) = 1$$

Equivalently, iff $B_\varphi \equiv 1$ *(cf. 2.2.3)*

Definition 2.2.16. *A boolean formula* $\varphi \in \mathcal{BF}(U,\mathcal{C})$ *is a satifiable iff*

$$\exists\, \tau : U \to \{0,1\} : \hat{\tau}(\varphi) = 1$$

Lemma 2.2.17. $\varphi \in \mathcal{BF}(U,\mathcal{C})$ *is satifiable iff* $\neg\varphi$ *is not a tautology.*

Proof. By definition, φ is satifiable $\Leftrightarrow \exists\, \tau : U \to \{0,1\} : \hat{\tau}(\varphi) = 1 \Leftrightarrow \exists\, \tau : U \to \{0,1\} : \hat{\tau}(\neg\varphi) = 0 \Leftrightarrow \neg\varphi$ is not a tautology. □

Definition 2.2.18. *We say two boolean formulas* $\varphi, \psi \in \mathcal{BF}(U,\mathcal{C})$ *are logically equivalent iff*

$$\forall\, \tau : U \to \{0,1\}\,,\ \hat{\tau}(\varphi) = \hat{\tau}(\psi)$$

Equivalently, iff $B_\varphi \equiv B_\psi$ *(cf. 2.2.3)*

Definition 2.2.19 (Implication)**.** *Define the following operation:*

x	y	$x \to y$
0	0	1
0	1	1
1	0	0
1	1	1

x	y	$x \leftarrow y$
0	0	1
0	1	0
1	0	1
1	1	1

x	y	$x \leftrightarrow y$
0	0	1
0	1	0
1	0	0
1	1	1

Observe $x \leftrightarrow y = \neg(x \oplus y) = (x \to y) \wedge (x \leftarrow y)$ *and* $x \leftrightarrow y = 1 \Leftrightarrow x = y$*.*

Remark 2.2.20 (Ex Falso Quodlibet)**.** $\forall\, x \in \{0,1\},\ 0 \to x = 1$

Lemma 2.2.21. *Two formulas $\varphi, \psi \in \mathcal{BF}(U, \mathcal{C})$ are logically equivalent iff $\varphi \leftrightarrow \psi$ is a tautology.*

Proof. By definition, $\varphi, \psi \in \mathcal{BF}(U, \mathcal{C})$ are logically equivalent iff:

$$\begin{aligned} &\forall\, \tau : U \to \{0,1\},\ \hat{\tau}(\varphi) = \hat{\tau}(\psi) \\ \Leftrightarrow &\forall\, \tau : U \to \{0,1\},\ \hat{\tau}(\varphi) \leftrightarrow \hat{\tau}(\psi) = 1 \\ \Leftrightarrow &\forall\, \tau : U \to \{0,1\},\ \hat{\tau}(\varphi \leftrightarrow \psi) = 1 \end{aligned}$$

$\Leftrightarrow \varphi \leftrightarrow \psi$ is a tautology. The last equality is from the algorithm 7. □

Corollary 2.2.22. *If φ_i is logically equivalent to ψ_i and ϕ is logically equivalent to θ, then $\varphi = \phi(\varphi_1, \cdots, \varphi_n)$ and $\psi = \theta(\psi_1, \cdots, \psi_n)$ are logically equivalent (cf. 2.2.7).*

Remark 2.2.23. *Due to 2.1.3,for $\mathcal{C} = \{\text{NOT}, \text{OR}, \text{AND}\}$ we may consider only $\pi : V \to U \cup \mathcal{C}$, since, we may consistently simplify any formula.*

2.3 deMorgan Duality

Theorem 2.3.1 (deMorgan's Law)**.** *The following are valid* $\forall\, x, y \in \{0,1\}$ *,*

- $\neg(x \vee y) = (\neg x) \wedge (\neg y)$
- $\neg(x \wedge y) = (\neg x) \vee (\neg y)$

Proof. By direct calculation of truth tables. □

Algorithm 8 deMorgan Dual, for $\varphi \in \mathcal{BF}(U, \{\neg, \wedge, \vee\})$

```
function DM(φ)
    if #φ = 1 then return NOT(φ)
    else if φ = NOT(φ) then
        if #φ = 2 then return φ
        else return NOT(DM(φ))
        end if
    else if φ = AND(φ1, φ2) then return OR(DM(φ1), DM(φ2))
    else if φ = OR(φ1, φ2) then return AND(DM(φ1), DM(φ2))
    end if
end function
```

Remark 2.3.2. *Algorithm 8 used pattern matching with formula substitution (cf. 2.2.6). Equivalently, we could've written with rooted trees and hanging trees (cf. 1.4.23,1.4.27). Further, we adopt the convention in 2.2.23.*

Theorem 2.3.3 (Duality)**.** $\forall\, \varphi \in \mathcal{BF}(U, \{\neg, \wedge, \vee\})$, $\mathrm{DM}(\varphi)$ *is logically equivalent to* $\neg\varphi$*.*

Proof. By induction of $\#\varphi$: due to 2.2.7

- Base Case ($\#\varphi = 1$): $B_{\mathrm{DM}(\varphi)} = B_{\mathrm{NOT}(\varphi)} = \mathrm{NOT} \circ B_\varphi - B_{\neg\varphi}$.
- Induction Step: If $\varphi =$
  - (i) $\mathrm{NOT}(\phi) : \mathrm{DM}(\varphi) = \mathrm{NOT}(\mathrm{DM}(\phi)) \equiv \mathrm{NOT}(\neg\phi) = \mathrm{NOT}(\varphi) = \neg\varphi$
  - (ii) $\mathrm{AND}(\phi_1, \phi_2) : \mathrm{DM}(\varphi) = \mathrm{OR}(\mathrm{DM}(\phi_1), \mathrm{DM}(\phi_2)) = \mathrm{OR}(\neg\phi_1, \neg\phi_2) = \neg(\mathrm{AND}(\phi_1, \phi_2)) = \neg\varphi$
  - (iii) $\mathrm{OR}(\phi_1, \phi_2) : \mathrm{DM}(\varphi) = \mathrm{AND}(\mathrm{DM}(\phi_1), \mathrm{DM}(\phi_2)) = \mathrm{AND}(\neg\phi_1, \neg\phi_2) = \neg(\mathrm{OR}(\phi_1, \phi_2)) = \neg\varphi$

where the last two equation were given by 2.3.1. □

Definition 2.3.4 (NNF)**.** *A formula* $\varphi \in \mathcal{BF}(U, \{\neg, \wedge, \vee\})$ *is in Negation Normal Form if it parse tree* (G, π) *obeys:* $\pi(v) = \neg \Rightarrow v$ *is the parent to a leaf* ℓ *and* $\pi(\ell) \in U$*.*

Lemma 2.3.5. *If φ is in NNF, then so it* $\mathrm{DM}(\varphi)$.

Proof. By induction of $\#\varphi$:

- Base Case ($\#\varphi = 1$): $\#\varphi \in U$, trivially NNF.
- Induction Step: If $\varphi =$
  - (i) $\mathrm{NOT}(\phi)$: then $\phi \in U$, so $\#\varphi = 2$ and $\#\phi = 1$. Hence, $\mathrm{DM}(\varphi) = \phi$, which is in NNF.
  - (ii) $\mathrm{AND}(\phi_1, \phi_2) : \mathrm{DM}(\varphi) = \mathrm{OR}(\mathrm{DM}(\phi_1), \mathrm{DM}(\phi_2))$
  - (iii) $\mathrm{OR}(\phi_1, \phi_2) : \mathrm{DM}(\varphi) = \mathrm{AND}(\mathrm{DM}(\phi_1), \mathrm{DM}(\phi_2))$

  the last two are NNF by substitution.

□

Remark 2.3.6. *Hence, the deMorgan dual is a way to negate a formula, but maintaining NNF.*

Algorithm 9 Negation Normal Form, for $\varphi \in \mathcal{BF}(U, \{\neg, \wedge, \vee\})$

```
function NNF(φ)
    if #φ = 1 then return φ
    else if φ = NOT(φ) then
        if #φ = 2 then return φ
        else return DM(NNF(φ))
        end if
    else if φ = AND(φ1, φ2) then return AND(NNF(φ1), NNF(φ2))
    else if φ = OR(φ1, φ2) then return OR(NNF(φ1), NNF(φ2))
    end if
end function
```

Theorem 2.3.7. *The algorithm 9 produces a formula in NNF, as in 2.3.4, and logically equivalent to φ.*

Proof. By induction on $\#\varphi$, the base cas is trivial and in the recursion, we only need to check $\mathrm{DM}(\mathrm{NNF}(\phi)) \equiv \mathrm{NOT}(\mathrm{NNF}(\phi)) \equiv \mathrm{NOT}(\phi) = \varphi$ □

2.4 Canonical Representations

Definition 2.4.1. *We define:*

(i) A literal is a formula with only are variable (X) or the negation of a variable ($\text{NOT}(X)$), and will be denoted ℓ.

(ii) A product term is the conjunction of n literals: $\pi = \text{AND}_n(\ell_1, \cdots, \ell_n)$

(iii) A product is simple if every variable appears at most once in π, and we denote $\text{vars}(\pi)$ the variable in π. Further, we write $\text{vars}^-(\pi)$ for the variables which are negated and $\text{vars}^+(\pi)$ for the ones which are not.

(iv) A minterm of $\mathcal{BF}(U, \mathcal{C})$ is a simple product with $\text{vars}(\pi) = U$.

(v) A formula φ is a sum of products if $\varphi = \text{OR}_k(\pi_1, \cdots, \pi_k)$ where π_i are product terms.

Lemma 2.4.2. *A minterm π is true ($\hat{\tau}(\pi) = 1$) for exactly one truth assigment.*

Proof. By 2.2.7, $\hat{\tau}(\pi) = 1 \Leftrightarrow \forall\, i \in \{1, \cdots, n\}\,, \hat{\tau}(\ell_i) = 1$, that is, iff the assigment is: $\tau(X) = \begin{cases} 1 & \text{if } X \in \text{vars}^+(\pi) \\ 0 & \text{if } X \in \text{vars}^-(\pi) \end{cases}$. □

Theorem 2.4.3. *For $v \in \{0,1\}^n$, define the literal ℓ_i^v in the variables $U = \{X_1, \cdots, X_n\}$ as: $\ell_i^v = \begin{cases} X_i & \text{if } v_i = 1 \\ \neg X_i & \text{if } v_i = 0 \end{cases}$, then let $\pi_v = \text{AND}_n(\ell_1^v, \cdots, \ell_n^v)$. This definition is exactly such that $\hat{\tau}_v(\pi_v) = 1$. For $B : \{0,1\}^n \to \{0,1\}$, and $B^{-1}(1) = \left\{v^1, \cdots, v^k\right\}$, then $\varphi = \text{OR}_k(\pi_{v^1}, \cdots, \pi_{v^k})$ represents B (i.e. $B_\varphi = B$).*

Proof. $\hat{\tau}_v(\varphi) = \text{OR}_k(\hat{\tau}_v(\pi_{v^1}), \cdots, \hat{\tau}_v(\pi_{v^k}))$ by 2.2.7. If $v \notin B^{-1}(1) \Rightarrow B(v) = 0$, then $\hat{\tau}_v(\varphi) = \text{OR}_k(0^k) = 0 = B(v)$. Else, $v = v^i \Rightarrow \hat{\tau}_{v^i}(\varphi) = \text{OR}_k(0^{i-1} \circ 1 \circ 0^{k-i}) = 1 = B(v^i)$. Hence, $\forall\, v \in \{0,1\}^n\,, \hat{\tau}_v(\varphi) = B(v)$. □

Corollary 2.4.4. *Every boolean function can be represented as a sum of products. Denote $\text{prods}(\varphi)$ the set of products in the SOP φ.*

Definition 2.4.5. *Analogous to 2.4.1, we define:*

(i) A sum term is the disjunction of n literals: $\sigma = \text{OR}_n(\ell_1, \cdots, \ell_n)$

(ii) A sum is simple if every variable appears at most once in σ.

(iii) A maxterm of $\mathcal{BF}(U, \mathcal{C})$ is a simple sum with $\text{vars}(\sigma) = U$.

(iv) A formula φ is a product of sums if $\varphi = \text{AND}_k(\sigma_1, \cdots, \sigma_k)$ where σ_i are sum terms.

Theorem 2.4.6 (Duality)**.** *Let φ be a product of sums $\text{AND}_k(\sigma_1, \cdots, \sigma_k)$ where $\sigma_i = \text{OR}_n(\ell_1, \cdots, \ell_n)$. Let $\overline{\ell}_i = \text{DM}(\ell_i)$, let $\pi_i = \text{AND}_n(\overline{\ell}_1, \cdots, \overline{\ell}_n)$ and $\psi = \text{OR}_k(\pi_1, \cdots, \pi_k)$, a sum of products. Then $\neg\varphi$ and ψ are logically equivalent.*

Proof. A more direct formula is: $\psi = \text{DM}(\varphi)$. □

Corollary 2.4.7. *A maxterm σ is false ($\hat{\tau}(\sigma) = 0$) for exactly one truth assigment.*

Corollary 2.4.8. *Every boolean function can be represented as a products of sums. Take a sum of product representation for $\text{NOT} \circ f$, and use* DM*. Denote sums(φ) the set of products in the POS φ.*

Definition 2.4.9 (Galois Field)**.** *The field $\mathbb{F}_2$, also denoted GF(2), is the set $\{0, 1\}$ together with the operation* XOR *for addition (denoted $\oplus$) and* AND *for multiplication (denoted $\odot$). It is a field (cf. Linear Algebra). Further GF(2)[U] denotes the polynomial ring of that field.*

Lemma 2.4.10. $\forall\, X \in GF(2)$,

(i) $X \oplus X = 0$

(ii) $\forall\, k \in \mathbb{N}_{\geq 1}$, $X^k = \underbrace{X \odot \cdots \odot X}_{n \text{ times}} = X$

Proof. Direct calculation:

(i) $1 \oplus 1 = 0 \oplus 0 = 0$

(ii) $X^1 = X$ and $X^2 = X \odot X = X$, since $1 \odot 1 = 1$ and $0 \odot 0 = 0$. Hence, by induction, it follows.

□

Corollary 2.4.11. *In the monomial $\mu = \prod_{X \in U} X^{p_X}$ of GF(2)[U], only needs $p_X \leq 1$, up to logical equivalence of the polynomial functions.*

Theorem 2.4.12. *Every boolean function $B : \{0,1\}^k \to \{0,1\}$ can be represented as a polynomial in $GF(2)[X_1, \cdots, X_k]$.*

Proof. By induction on k:

- Base Case: $k = 1$

Formula $\backslash x$	0	1
0	0	0
x	0	1
$x \oplus 1$	1	0
1	1	1

- Induction Step: Observe: For $x \in \{0,1\}$ and $xs \in \{0,1\}^{k-1}$

$$B(xs \circ x_n) = B_0(xs) \oplus B_1(xs) \oplus x \odot B_1(xs)$$

where $B_0(xs) = B(xs \circ 0)$ and $B_1(xs) = B(xs \circ 1)$, which are functions $B_0, B_1 : \{0,1\}^{k-1} \to \{0,1\}$. By induction, they can be represented as a polynomial in $\mathrm{GF}(2)[X_1, \cdots, X_{k-1}]$.

□

3 Digital Devices and Circuits

3.1 Digital Abstraction

Definition 3.1.1 (Digital Interpretation)**.** *Define* $dig : \mathbb{R}^n \to \{0, 1, \text{NL}\}$*, which gives the digital interpretation of a signal (*NL *is non-logical). Usually, it is implemented using setting a low and high voltage* $V_{low} < V_{high}$*, so that:*

$$dig(x) = \begin{cases} 0 & \text{if } x < V_{low} \\ 1 & \text{if } x > V_{high} \\ \text{NL} & \text{otherwise} \end{cases}$$

We can extend to $dig_n : \mathbb{R}^n \to \{0, 1, \text{NL}\}^n$*, where*

$$dig_n(y_1, \cdots, y_n) = (dig(y_1), \cdots, dig(y_n))$$

Definition 3.1.2 (Noise Margins)**.** *By setting a high and low voltage of input and output* $V_{out,low} < V_{in,low} < V_{in,high} < V_{out,high}$*, so we can define:*

$$dig_{in}(x) = \begin{cases} 0 & \text{if } x < V_{in,low} \\ 1 & \text{if } x > V_{in,high} \\ \text{NL} & \text{otherwise} \end{cases} \quad \text{and} \quad dig_{out}(x) = \begin{cases} 0 & \text{if } x < V_{out,low} \\ 1 & \text{if } x > V_{out,high} \\ \text{NL} & \text{otherwise} \end{cases}$$

and the differences $V_{in,low} - V_{out,low}$ *and* $V_{out,high} - V_{in,high}$ *are called the noise margins.*

Lemma 3.1.3 (Bounded Noise)**.** *If* $f(t) = g(t) + n(t)$ *and* $|n(t)|$ *is less than the noise margins, then,* $dig_{out}(g(t)) \in \{0, 1\} \Rightarrow dig_{out}(g(t)) = dig_{in}(f(t))$

Proof. If $\text{dig}_{\text{out}}(g(t)) = 0$, that is, $g(t) < V_{\text{out,low}}$, then: $f(t) = g(t) + n(t) < V_{\text{out,low}} + (V_{\text{in,low}} - V_{\text{out,low}}) = V_{\text{in,low}}] \Rightarrow \text{dig}_{\text{in}}(f(t)) = 0$. The other case is analogous. □

Definition 3.1.4 (Stability)**.** *An analog signal* $f(t)$ *is logical at* $[t_1, t_2]$ *if* $\forall\, t \in [t_1, t_2]\,,\ dig(f(t)) \in \{0, 1\}$*. Moreover, it is stable at* $[t_1, t_2]$ *if either* $\forall\, t \in [t_1, t_2]\,,\ dig(f(t)) = 0$ *or* $\forall\, t \in [t_1, t_2]\,,\ dig(f(t)) = 1$*. Further, we can extend definition to vector signals.*

Definition 3.1.5 (Gate)**.** *A device G with input $\vec{x}$ and output $\vec{y}$ is a gate if there is a function $g : \mathbb{R}^n \to \mathbb{R}^k$ such that:*

$$\exists\, \Delta > 0 : \forall\, x_0, t_0 \in \mathbb{R}\,,\, [\forall\, t \in [t_0 - \Delta, t_0]\,,\, x(t) = x_0] \Rightarrow y(t_0) = g(x_0)$$

in that case, g is called the static transfer function of G. We also require continuity of g (cf. Calculus I).

Definition 3.1.6 (Combinational Gates)**.** *A gate G with input $\vec{x}$ and output $\vec{y}$ is a combinational gate if:*

$$\begin{aligned}\exists\, \Delta > 0 : \forall\, \vec{x}(t) \in \mathbb{R}^n, t_1, t_2 \in \mathbb{R}\,,\\ [\exists\, \vec{b_x} \in \mathbb{R}^n : \forall\, t \in [t_1, t_2]\,,\ dig_{in}(\vec{x}(t)) = \vec{b_x}] \Rightarrow\\ [\exists\, \vec{b_y} \in \mathbb{R}^n : \forall\, t \in [t_1 + \Delta, t_2]\,,\ dig_{out}(\vec{y}(t)) = \vec{b_y}]\end{aligned}$$

Observe the static transfer function g satisfies: $\forall\, \vec{x} \in \mathbb{R}^n\,,\ dig_{in}(\vec{x}) \in \{0,1\}^n \Rightarrow dig_{out}(g(\vec{x})) \in \{0,1\}^k$. Moreover, since we need to maintain stability, there is a boolean function $B_g : \{0,1\}^n \to \{0,1\}^k$ such that $\forall\, \vec{x} \in \mathbb{R}^n\,,\ dig_{in}(\vec{x}) \in \{0,1\}^n \Rightarrow dig_{out}(g(\vec{x})) = B_g(dig_{in}(\vec{x}))$, which we call the gate G consistent with B_g.

Definition 3.1.7 (Propagation Delay)**.** *For a gate G, the propagation delay t_{pd} is such that: If $\vec{x}$ is stable in $[t_1, t_2]$, G is consistent with B_g in $[t_1 + t_{pd}, t_2]$. Analogously, the contamination delay t_{cont} is such that: If $\vec{x}$ is stable in $[t_1, t_2]$, G is consistent with B_g in $[t_2, t_2 + t_{cont}]$.*

3.2 Combinational Circuits

Remark 3.2.1. *We'll consider a library* Γ *of commutative* $n:1$ *gates.*

Definition 3.2.2 (Terminals)**.** *For a gate* G*, define the set of terminals:* $terminals(G) = \{in(G)_i\}_{i=1}^{\mathrm{IN}(g)} \cup \{out(G)_i\}_{i=1}^{\mathrm{OUT}(g)}$*, where* $in(G)_i$ *are input ports of* G *and* $out(G)_i$ *are output ports. We define an input gate as a gate with zero inputs and one output, denoted* $(\textsc{in}, x_i)$*, where* x_i *is the name of the output signal. Analogously,an output gate as a gate with zero outputs and one input, denoted* $(\textsc{out}, y_i)$*, where* y_i *is the name of the input signal. There are ways to communicate with the external world and modularize our designs. We call the set of input and output gates the library* IO*.*

Definition 3.2.3 (Wires and Nets)**.** *A wire is a connection between two terminals (of different gates), which in the zero noise model, makes the signals are both ends equal. A net is a subset of terminals which are connected by wires. If* N *is connected to an input gate of* G*, then* N *feeds* G*, and if it's connected to output,* N *is fed by* G*. Moreover, a net is called simple if:*

(i) N *is fed by exactly one output terminal*

(ii) N *feeds at least one input terminals*

Definition 3.2.4 (Netlist)**.** *A netlist is a tuple* $H = (V, \pi, N_s)$*, where* V *is a set of nodes,* $\pi : V \to \Gamma \cup \mathrm{IO}$ *is a labeling function, and* N_s *is a set of nets over* $terminals(V, \pi) = \{(v, t) \mid v \in V\ ,\ t \in terminals(\pi(v))\}$*, and we require they are pairwise disjoint (its a partition). This gives all the information to construct a circuit.*

Remark 3.2.5. *If all nets in a netlist* $H = (V, \pi, N_s)$ *are simple, we can construct a directed graph* $\mathrm{DG}(H) = (V, E_H)$ *as follows, for* $N = \{t, t_1, \cdots, t_k\} \in N_s$*, where* t *is input terminal of* v *and* t_i *are output terminals of* v_i*, define* $\{(v, v_i)\}_{i=1}^{k} \subseteq E_H$*.*

Lemma 3.2.6. *Given a directed graph* $G = (V, E)$ *and a labeling* $\pi : V \to \Gamma \cup \mathrm{IO}$*, where* Γ *is commutative, there is a netlist* $H_G = (V, \pi, N)$ *(with every net simple) such that* $\mathrm{DG}(H_G) = G$*.*

Proof. For $u \in V$ define

$$N_u = \{(u, out(\pi(u)))\} \cup \left\{ \begin{matrix} (v, \pi(v)) & \text{if } \pi(v) \in \mathrm{IO} \\ (v, in(\pi(v))_i) & \text{otherwise} \end{matrix} \;\middle|\; (u, v) \in E \right\}$$

and i is whatever since Γ is commutative, but we can be systematic by using topological ordering, for example (if it exists). Then, $N = \{N_u \mid u \in V\}$ is a set of nets over *terminals*(V, π). □

Definition 3.2.7. *A netlist $H = (V, \pi, N_s)$ is a combinational circuit if:*
(i) Every net in N_s is simple;
(ii) Every terminal in terminals(V, π) belongs to exactly one net in N_s;
(iii) The directed graph $\mathrm{DG}(H)$ is acyclic.
We denote a combinational circuit by $\mathbb{C}$.

Algorithm 10 Simulation, for $\mathbb{C} = (V, \pi, N_s)$ with input $\vec{x}$

$(v_1, v_2, \cdots, v_n) \leftarrow \mathrm{TS}(\mathrm{DG}(\mathbb{C}))$
for $1 \leq i \leq n$ **do**
  if $\deg_{\mathrm{in}}(v_i) = 0$ **then**
    $(\textsc{in}, x_j) \leftarrow \pi(v_i)$
    return $B_{v_i}(\vec{x}) = x_j$ and $t_{\mathrm{pd}}(v_i) = 0$
  else if $\deg_{\mathrm{in}}(v_i) = 1$ **then**
    $v_j : (v_j, v_i) \in E_{\mathrm{DG}(\mathbb{C})}$
    if $\pi(v_i) = (\textsc{out}, y)$ **then**
      return $B_{v_i}(\vec{x}) = B_{v_j}(\vec{x})$ and $t_{\mathrm{pd}}(v_i) = t_{\mathrm{pd}}(v_j)$
    else
      return $B_{v_i}(\vec{x}) = B_{\pi(v_i)}(B_{v_j}(\vec{x}))$ and $t_{\mathrm{pd}}(v_i) = t_{\mathrm{pd}}(v_j) + t_{\mathrm{pd}}(\pi(v_i))$
    end if
  else $d = \deg_{\mathrm{in}}(v_i) \geq 2$
    $v_{j_k} : (v_{j_k}, v_i) \in E_{\mathrm{DG}(\mathbb{C})}$
    return $B_{v_i}(\vec{x}) = B_{\pi(v_i)}(B_{v_{j_1}}(\vec{x}), \cdots, B_{v_{j_d}}(\vec{x}))$
    and $t_{\mathrm{pd}}(v_i) = \max\{t_{\mathrm{pd}}(v_{j_k}) \mid 1 \leq k \leq d\} + t_{\mathrm{pd}}(\pi(v_i))$
  end if
end for

Theorem 3.2.8. *Given a boolean formula $\varphi \in \mathcal{BF}(\{X_1, \cdots, X_n\}, \mathcal{C})$, there is a combinational circuit $\mathbb{C}_\varphi$ that is consistent with B_φ.*

Proof. Define $\mathrm{DG}(\mathbb{C}_\varphi)$:

- For each $1 \leq i \leq n$ merge all sources in G_φ labeled X_i into a new source u_i with $\pi_{\mathbb{C}}(u_i) = (\textsc{in}, x_i)$.
- Add a vertex y with the arc $(r(G_\varphi), y)$ and label $\pi_{\mathbb{C}}(y) = (\textsc{out}, y)$.

Since, we only merged vertices and added one outgoing arc to the root, the resulting directed graph is acyclic. Moreover, we can the reconstruct of the combinational circuit (cf. 3.2.6). □

Theorem 3.2.9 (Simulation Theorem)**.** *Suppose* $\{x_i(t)\}_{i=1}^k$ *are stable at* $[t_1, t_2]$ *and are inputs to* $\mathbb{C} = (V, \pi, N_s)$*. Then* $\forall\, v \in V$ *,*

$$\exists\, t_{pd}(v) \in \mathbb{R}, B_v : \{0,1\}^k \to \{0,1\} : \forall\, t \in [t_1 + t_{pd}(v), t_2]\,,\ v(t) = B_v(\vec{x}(t))$$

Moreover, they are given by algorithm 10.

Proof. Follows from the algorithm by induction and the definition of combinational gate on the propagation delay. □

Remark 3.2.10. *The algorithm 10 does not depend on the topological sorting.*

3.3 Cost and Delay Analysis

Definition 3.3.1 (Cost and Delay of Combinational Circuit)**.** *For* $\mathbb{C} = (V, \pi, N_s)$*. For* $c : \Gamma \to \mathbb{R}^{\geq 0}$ *a cost function of each gate (inputs and outputs usually have zero cost), define:*

- *Cost:* $c(\mathbb{C}) = \sum_{v \in V} c(\pi(v))$
- *Delay:* $d(\mathbb{C}) = t_{pd}(\mathbb{C}) = \max_{v \in V} t_{pd}(v) = \max\{t_{pd}(v) \mid v \in V\,,\, \deg_{out}(v) = 0\}$

Definition 3.3.2. *For* $B : \{0,1\}^n \to \{0,1\}$*, let* $B_{\restriction x_i = \sigma} : \{0,1\}^{n-1} \to \{0,1\}$ *such that:* $B_{\restriction x_i = \sigma}(X) = B(X[0:i] \circ \sigma \circ X[i:n-2])$*. Define:*

$$\operatorname{cone}(B) = \{i \in \{0, \cdots, n-1\} \mid B_{\restriction x_i = 0} \not\equiv B_{\restriction x_i = 1}\}$$

Moreover, if we write $\mathit{flip}_i : \{0,1\}^n \to \{0,1\}^n$*:*

$$\mathit{flip}_i(X)[j] = \begin{cases} X[j] & \text{if } j \neq i \\ \textsc{not}(X[j]) & \text{if } j = i \end{cases}$$

then $\operatorname{cone}(B) = \{i \in \{0, \cdots, n-1\} \mid \exists\, v \in \{0,1\}^n : B(v) \neq B(\mathit{flip}_i(v))\}$

Lemma 3.3.3. $\operatorname{cone}(B) = \varnothing$ *iff* B *is a constant function.*

Proof. We prove both directions.

($\Rightarrow$) By contrapositive, $\exists\, u, v \in \{0,1\}^n : f(u) = 0$ and $f(v) = 1$. Take a path $u \to u_1 \to \cdots u_k \to v$ that scans from LSB to MSB, and flips that bits of u to match v. Then, there are u_i, u_{i+1} such that $f(u_i) = 0$ and $f(u_{i+1}) = 1$, and, by definition, $u_{i+1} = \mathit{flip}_j(u_i)$, for some j.

($\Leftarrow$) Trivial from the definition.

□

Theorem 3.3.4 (Linear Cost)**.** *Let* $\mathbb{C}$ *be a combinational circuit that implements* $B : \{0,1\}^n \to \{0,1\}$*. If the fan-in of every gate in* $\mathcal{C}$ *is at most* 2*, then:*

$$c(\mathbb{C}) \geq \#\operatorname{cone}(B) - 1$$

Proof. We omit the proof for it is a bit long, and you can find it in the book. □

Theorem 3.3.5 (Logarithmic Delay)**.** *Let $\mathbb{C}$ be a combinational circuit that implements $B : \{0,1\}^n \to \{0,1\}$. If the fan-in of every gate in $\mathcal{C}$ is at most k, then:*

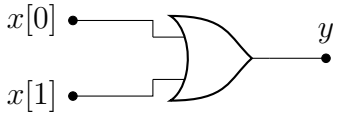
$$d(\mathbb{C}) \geq \log_k \Big(\# \operatorname{cone}(B) \Big)$$

Proof. We omit the proof for it is a bit long, and you can find it in the book. □

3.4 Tree Circuit Designs

Definition 3.4.1. *A combinational circuit* $\mathbb{C} = (G, \pi)$ *is* OR-TREE(n) *if:*
(i) G *is a rooted tree with* n *sources;*
(ii) If v *is not a source nor a sink,* $\pi(v) = \text{OR}$.

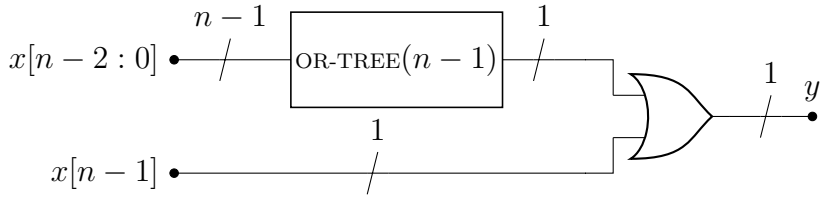
Example 3.4.2. *The only* OR-TREE(2) *is:*
With cost $c(2) = 1$ *and delay* $d(2) = 1$.



Lemma 3.4.3. *Every* OR-TREE(n) *implements* OR_n.

Proof. Let G_1 and G_2 be trees hanging from v such that $(v, r(G)) \in E$ (remember, $\pi(r(G))$ is an output gate). By induction G_i implements OR_{n_i} and by substitution, $y = \text{OR}(\text{OR}_{n_1}(x[n_1 - 1 : 0]), \text{OR}_{n_2}(x[n - 1 : n_1])) = \text{OR}_n(x[n - 1 : 0])$ since OR is associative. □

Design 3.4.4 (Tree Linear Recursion)**.** *We design an* OR-TREE(n) *recursively as follows:*



(i) Cost: $c(n) = (n - 1) \cdot c(\text{OR})$
(ii) Delay: $d(n) = (n - 1) \cdot d(\text{OR})$

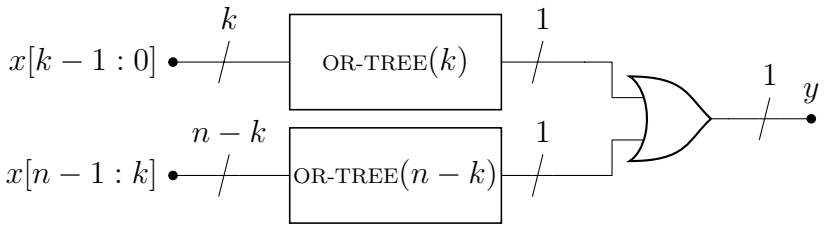
Proof. We write the recurence equations:
(i) Cost: for $n > 2 : c(n) = c(n - 1) + c(\text{OR}) = (n - 1) \cdot c(\text{OR})$, follows by induction.
(ii) Delay: for $n > 2 : d(n) = d(n - 1) + d(\text{OR}) = (n - 1) \cdot d(\text{OR})$, follows by induction.
(iii) Correctness: 3.4.3.

□

Lemma 3.4.5. $c(\text{OR-TREE}(n)) = n - 1$ *for any design.*

Proof. As before, define G_i and n_i. Then, by definition, $c(\text{OR-TREE}(n)) = c(v) + c(\text{OR-TREE}(n_1)) + c(\text{OR-TREE}(n_2))$. By induction, $c(\text{OR-TREE}(n)) = 1 + (n_1 - 1) + (n_2 - 1) = (n_1 + n_2) - 1 = n - 1$. □

Design 3.4.6 (Tree Divide and Conquer)**.** *We design an* OR-TREE(n) *recursively, where* $k = \lfloor n/2 \rfloor$*, as follows:*



(i) Cost: $c(n) = (n-1) \cdot c(\text{OR})$
(ii) Delay: $d(n) = d(\text{OR}) \cdot \lceil \log_2(n) \rceil$

Proof. First, notice: $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. We write the recurence equations:

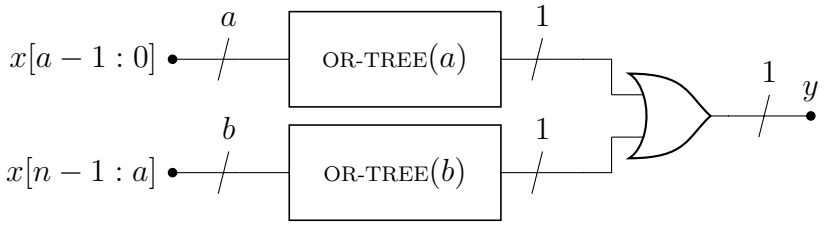
(i) Cost: for $n > 2 : c(n) = c(\lfloor n/2 \rfloor) + c(\lceil n/2 \rceil) + c(\text{OR}) = (n-1) \cdot c(\text{OR})$, follows by induction.

(ii) Delay: for $n > 2 : d(n) = \max\left\{d(\lfloor n/2 \rfloor), d(\lceil n/2 \rceil)\right\} + d(\text{OR}) = d(\text{OR}) \cdot \lceil \log_2(n) \rceil$, follows by induction and:
- $n = 2k : d(n) = d(\text{OR}) \cdot \lceil \log_2(k) \rceil + d(\text{OR}) = d(\text{OR}) \cdot \lceil \log_2(2k) \rceil = d(\text{OR}) \cdot \lceil \log_2(n) \rceil$
- $n = 2k+1 : d(n) = d(\text{OR}) \cdot \lceil \log_2(k+1) \rceil + d(\text{OR}) = d(\text{OR}) \cdot \lceil \log_2(2k+2) \rceil = d(\text{OR}) \cdot \lceil \log_2(n) \rceil$, since n is odd $n \leq 2^d \Rightarrow n+1 \leq 2^d$.

(iii) Correctness: 3.4.3.

Hence, the design is asymptotically optimal. □

Remark 3.4.7 (Balanced Tree)**.** *A partition* (a, b) *of* n *is balanced if* $a+b = n$ *and* $\max\{\lceil \log_2(a) \rceil, \lceil \log_2(b) \rceil\} = \lceil \log_2(n) \rceil - 1$*. Then, the design:*



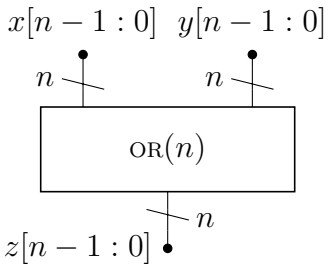
Will give same delay and cost as 3.4.6. The proof is analogous.

Remark 3.4.8. *All definitions and designs above work for any associative function.*

Definition 3.4.9 (Bitwise Gates)**.** *Let γ be a gate with fan-in 2. For simplicity, we'll consider $\gamma = \text{OR}$. Then, we write the circuit as follows, where $\text{OR}(n)$ is the unique combinational circuit with inputs $x[n-1:0]$ and $y[n-1:0]$ and output $z[n-1:0]$ such that $z[i] = \text{OR}(x[i], y[i])$. Furthermore,*

(i) Cost: $c(n) = n \cdot c(\text{OR})$

(ii) Delay: $d(n) = d(\text{OR})$



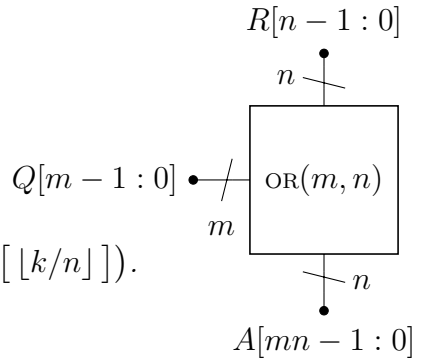
Definition 3.4.10 (Array of Gates)**.** *Let γ be a gate with fan-in 2. For simplicity, we'll consider $\gamma = \text{OR}$. Then, we write the circuit as follows, where $\text{OR}(m, n)$ is the unique combinational circuit with inputs $R[n-1:0]$ and $Q[m-1:0]$ and output $A[mn-1:0]$ such that*

$$A[q \cdot n + r] = \text{OR}(R[r], Q[q])$$

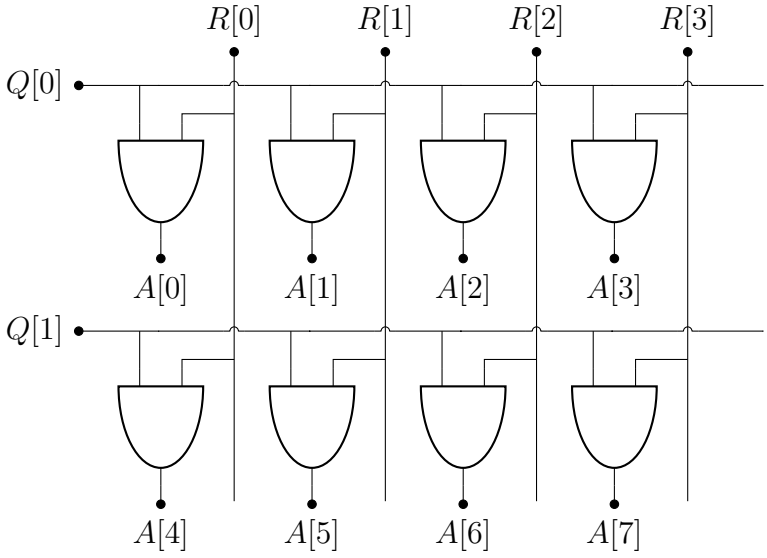
That is, using 1.3.1, $A[k] = \text{OR}\left(R[k \bmod n], Q\left[\,\lfloor k/n \rfloor\,\right]\right)$. Furthermore,

(i) Cost: $c(m, n) = mn \cdot c(\text{OR})$

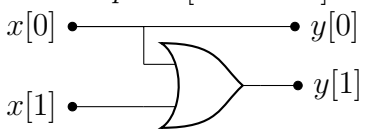
(ii) Delay: $d(m, n) = d(\text{OR})$



Example 3.4.11. *For $m = 2$ and $n = 4$ the array $\text{AND}(2, 4)$ is:*



Definition 3.4.12 (Parallel Prefix Circuit)**.** PPC-OR(n) *(eq. for other associative operations) is a combinational circuit with input* $x[n-1:0]$ *and output* $y[n-1:0]$ *s.t.:* $y[i] = \text{OR}_{i+1}(x[i:0])$.

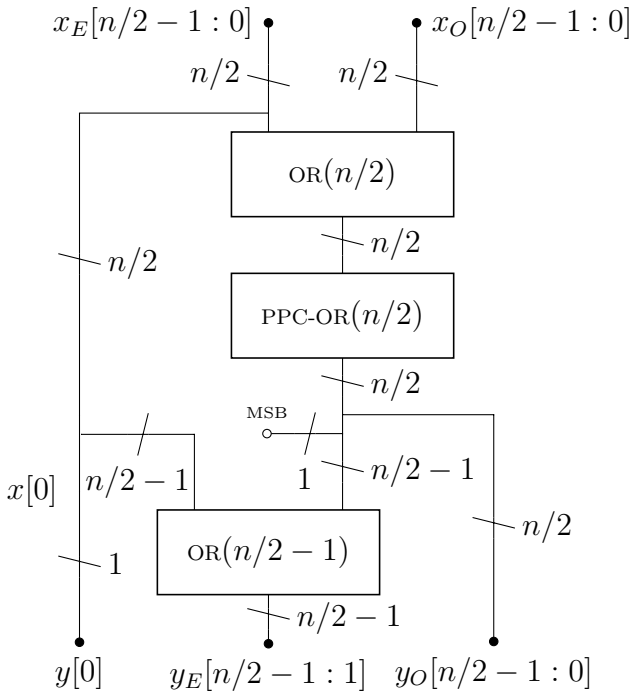


Example 3.4.13. *The only* PPC-OR(2) *is: With cost* $c(2) = 1$ *and delay* $d(2) = 1$.

Lemma 3.4.14. $c(\text{PPC-OR}(n)) = \Omega(n)$ *and* $d(\text{PPC-OR}(n)) = \Omega(\log n)$.

Proof. $\#\,\text{cone}(B_{y[n]}) = n$ by taking $\vec{x} = 0^n$ and flipping each bit, hence they are all in the cone. Follows from 3.3.4 and 3.3.5. □

Design 3.4.15. *We design an* PPC-OR(n) *recursively, where* $n = 2^k$ *and we define:* $x_E[i] = x[2i]$ *and* $x_O[i] = x[2i+1]$*, as follows:*



(i) Cost: $c(n) = \Theta(n)$
(ii) Delay: $d(n) = \Theta(\log n)$

Proof. We write the recurence equations:
(i) Cost: for $n > 1 : c(n) = c(n/2) + n - 1 \leq 2n - \log_2(n) - 2$, by induction, hence, $c(n) = \Theta(n)$.

(ii) Delay: for $n > 2 : d(n) = d(n/2) + 2 = 2\lceil \log_2(n) \rceil - 1$, follows by induction.

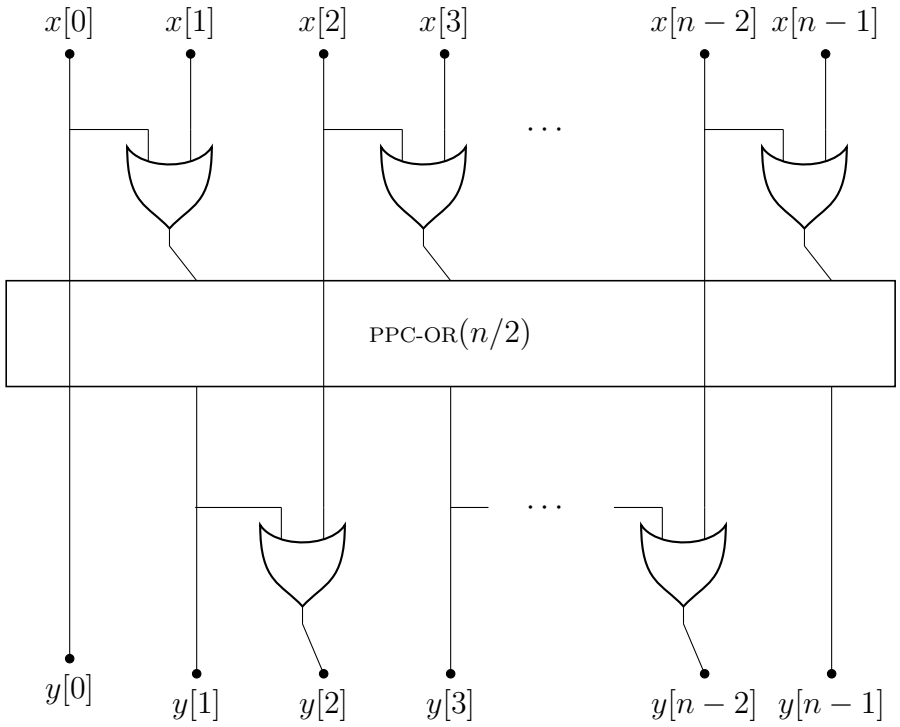
(iii) Correctness: For $1 \leq i \leq n-1$, and let $x'[i] = \text{OR}(x_E[i], x_O[i])$:

(a) $i = 2j+1 : y[i] = y[2j+1] = y_O[j] = \text{OR}_{j+1}(x'[j:0]) = \text{OR}_{2j+2}(x[2j+1:0])$.

(b) $i = 2j : y[i] = y[2j] = y_E[j] = \text{OR}(x[2j], \text{OR}_j(x'[j-1:0])) = \text{OR}_{2j+1}(x[2j:0])$.

Hence, the design is asymptotically optimal. □

We may rewrite the design as follows:

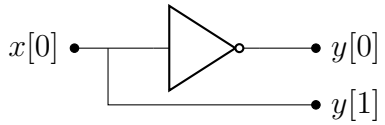


4 Bit Manipulation Designs

4.1 Decoders and Encoders

Definition 4.1.1 (Decoder)**.** DECODER(n) *is a combinational circuit with input* $x[n-1:0]$ *and output* $y[2^n-1:0]$ *s.t.:* $y[i] = \begin{cases} 1 & \text{if } \langle \vec{x} \rangle = i \\ 0 & \text{otherwise} \end{cases}$

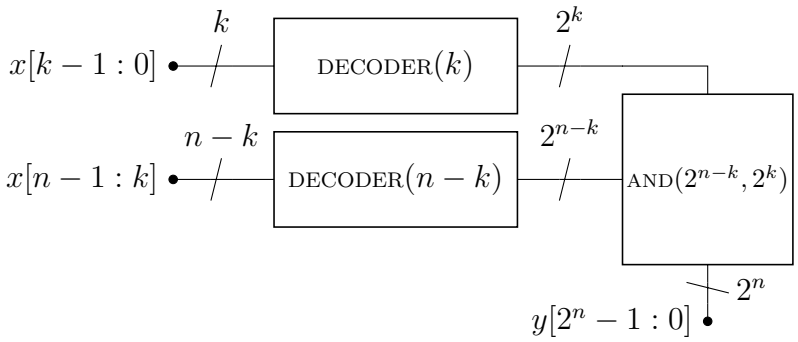
Example 4.1.2. *The basic* DECODER(1) *is:*
With cost $c(1) = 1$ *and delay* $d(1) = 1$.



Lemma 4.1.3. $c(\text{DECODER}(n)) = \Omega(2^n)$.

Proof. Every two bits are distinct, since for any i, j, $y[i] \neq y[j]$ for $\langle \vec{x} \rangle = i$. Further, every output requires at least one non-trivial gate, since each $y[i] = 1$ for a unique input. □

Design 4.1.4 (Decorder Divide and Conquer)**.** *We design an* DECODER(n) *recursively, where* $k = \left\lceil \frac{n}{2} \right\rceil$*, as follows:*



(i) Cost: $c(n) = \Theta(2^n)$
(ii) Delay: $d(n) = d(\text{OR}) \cdot \lceil \log_2(n) \rceil$

Proof. We write the recurence equations:

(i) Cost: for $n > 1 : c(n) = c(\lfloor n/2 \rfloor) + c(\lceil n/2 \rceil) + 2^n \Rightarrow 2^n \leq c(n) \leq 2^{n+1}$, follows by induction: $2^{\lfloor n/2 \rfloor} + 2^{\lceil n/2 \rceil} \leq 2^{\lceil n/2 \rceil + 1} \leq 2^n$, hence, $c(n) = \Theta(2^n)$.

(ii) Delay: for $n > 2 : d(n) = \max\left\{d(\lfloor n/2 \rfloor), d(\lceil n/2 \rceil)\right\} + d(\text{OR}) = d(\text{OR}) \cdot \lceil \log_2(n) \rceil$, follows by induction.

(iii) Correctness: For $i = q \cdot 2^k + r : y[i] = 1 \Leftrightarrow R[r] = 1$ and $Q[q] = 1 \Leftrightarrow \langle x[k-1:0] \rangle = r$ and $\langle x[n-1:k] \rangle = q \Leftrightarrow \langle x[n-1:0] \rangle = q \cdot 2^k + r = i$, by induction.

Hence, the design is asymptotically optimal. □

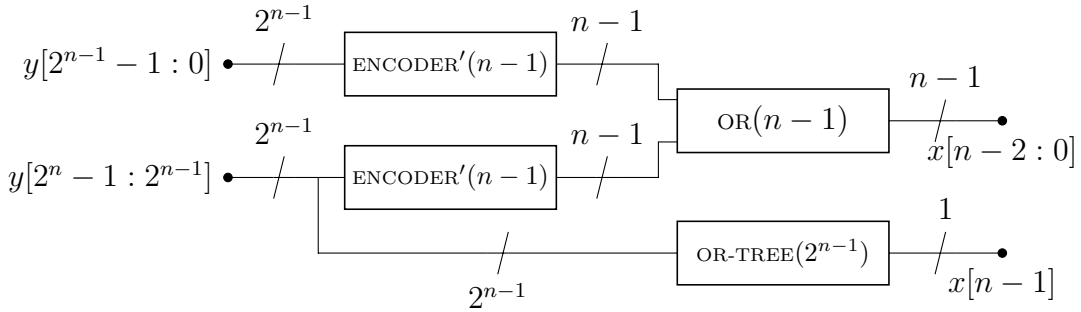
Definition 4.1.5 (Encoder)**.** ENCODER(n) *is a combinational circuit with input* $y[2^n - 1 : 0]$ *(with exactly one bit equal to 1) and output* $x[n-1:0]$ *s.t.:* $y[\langle \vec{x} \rangle] = 1$ *and all other bits of* y *are zero. We denote* ENCODER$_n$ *be boolean function implemented by* ENCODER(n)*. For further simplicity, we also require the output to be all zeros if the input is.*

Example 4.1.6. *The basic* ENCODER(1) *is:*
With cost $c(1) = 0$ *and delay* $d(1) = 0$.

Lemma 4.1.7. $c(\text{ENCODER}(n)) = \Omega(2^n)$ *and* $d(\text{ENCODER}(n)) = \Omega(n)$.

Proof. $\#\,\text{cone}(B_{x[0]}) \geq 2^{n-1}$ by taking $\vec{y} = 0^{2^n}$ and flipping odd-indexed bits, hence they are in the cone. Follows from 3.3.4 and 3.3.5. □

Design 4.1.8 (Encorder Divide and Conquer I)**.** *We design an* ENCODER$'(n)$ *recursively, as follows:*



(i) Cost: $c(n) = n\,(2^{n-1} - 1) \cdot c(\text{OR})$
(ii) Delay: $d(n) = (n-1) \cdot d(\text{OR})$

Proof. We write the recurence equations:

(i) Cost: for $n > 1 : c(n) = 2\,c(n-1) + (2^{n-1} - 1) \cdot c(\text{OR}) + (n-1) \cdot c(\text{OR}) = n\,(2^{n-1} - 1) \cdot c(\text{OR})$, follows by induction.

(ii) Delay: for $n > 2 : d(n) = \max\{d(n-1) + d(\text{OR}), \lceil \log_2(n) \rceil \cdot d(\text{OR})\} = (n-1) \cdot d(\text{OR})$, follows by induction.

(iii) Correctness: We consider, by induction:
   (a) $\vec{y} = 0^{2^n}$, then, by induction hypothesis, $x = 0^n$.
   (b) $\vec{y}_L = 0^{2^{n-1}}$, we get: $x[n-2:0] = \text{ENCODER}_{n-1}(\vec{y}_R)$ and $x[n-1] = 0$. Hence, $y[\langle\vec{x}\rangle] = y_R[\langle x[n-2:0]\rangle] = 1$.
   (c) $\vec{y}_R = 0^{2^{n-1}}$, we get: $x[n-2:0] = \text{ENCODER}_{n-1}(\vec{y}_L)$ and $x[n-1] = 1$. Hence, $y[\langle\vec{x}\rangle] = y[2^{n-1} + \langle x[n-2:0]\rangle] = y_L[\langle x[n-2:0]\rangle] = 1$.

where $\vec{y}_R = y[2^{n-1}-1:0]$ and $\vec{y}_L = y[2^n-1:2^{n-1}]$.

□

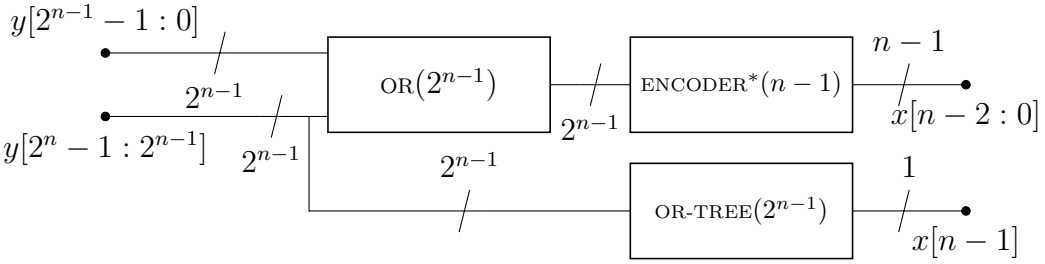
Lemma 4.1.9. *If $\vec{y}$ has no more than one bit equal* 1*, then:*

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

Proof. We consider:
1. $\vec{y} = 0^{2^n}$, trivial.
2. $\vec{y}_L = 0^{2^{n-1}}$, we get: $\text{ENCODER}_{n-1}(\text{OR}(0^{2^{n-1}}, \vec{y}_R)) = \text{ENCODER}_{n-1}(\vec{y}_R) = \text{OR}(\text{ENCODER}_{n-1}(0^{2^{n-1}}), \text{ENCODER}_{n-1}(\vec{y}_R))$
3. $\vec{y}_R = 0^{2^{n-1}}$, we get analogous.

□

Design 4.1.10 (Encorder Divide and Conquer II)**.** *We design an* $\text{ENCODER}^*(n)$ *recursively, as follows:*

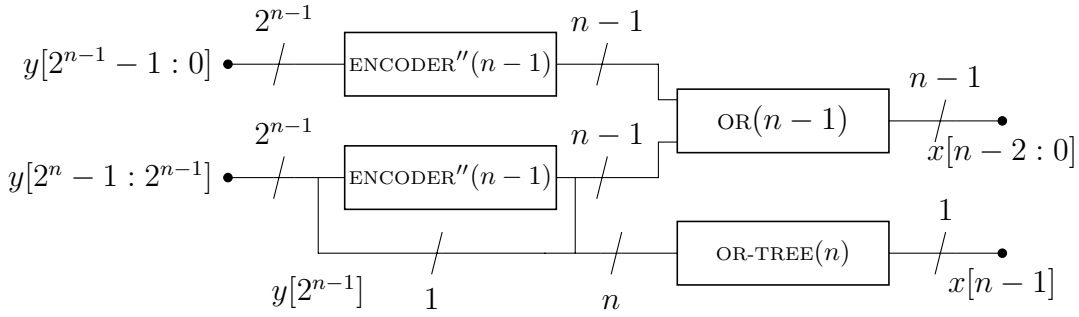


(i) Cost: $c(n) = (2^{n+1} - n - 3) \cdot c(\text{OR})$
(ii) Delay: $d(n) = (n-1) \cdot d(\text{OR})$

Proof. We write the recurence equations:
(i) Cost: for $n > 1 : c(n) = c(n-1) + (2^{n-1}-1) \cdot c(\text{OR}) + 2^{n-1} \cdot c(\text{OR}) = (2^{n+1} - n - 3) \cdot c(\text{OR})$, follows by induction.
(ii) Delay: for $n > 2 : d(n) = \max\{d(n-1) + d(\text{OR}), \lceil \log_2(n) \rceil \cdot d(\text{OR})\} = (n-1) \cdot d(\text{OR})$, follows by induction.

(iii) Correctness: Follows from the previous design and lemma.
Hence, the design is asymptotically optimal. □

Design 4.1.11 (Encorder Divide and Conquer III)**.** *We design an* ENCODER$''(n)$ *recursively, as follows:*



(i) Cost: $c(n) = \Theta(2^n)$
(ii) Delay: $d(n) = \Theta(n \log n)$

Proof. We write the recurence equations:

(i) Cost: for $n > 1 : c(n) = 2\,c(n-1) + 2(n-1) \Rightarrow c(n) \leq 2^{n+1}$, follows by induction.

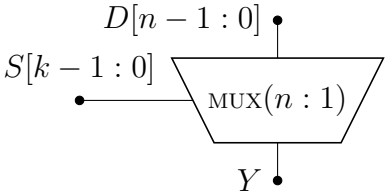
(ii) Delay: for $n > 2 : d(n) = \max\{d(n-1)+1, d(n-1)+\lceil \log_2(n) \rceil\} = d(n-1)+\lceil \log_2(n) \rceil \Rightarrow (n-1)\left[\log_2(n) - 2\right] \leq d(n) \leq 2n \log_2(n)$, follows by induction.

(iii) Correctness: Follows from previous design and special case $y[2^{n-1}] = 1$.

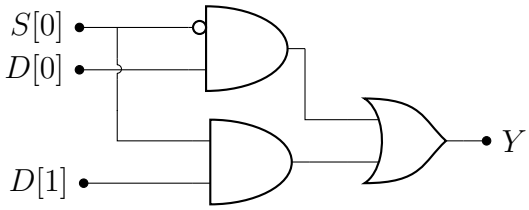
□

4.2 Selectors and Shifters

Definition 4.2.1 (Multiplexers)**.** MUX$(n : 1)$ *is a combinational circuit with inputs* $D[n-1:0]$, $S[k-1:0]$ *(where* $n = 2^k$*) and output* Y *s.t.:* $Y = D[\langle \vec{S} \rangle]$.



Example 4.2.2. *The basic* MUX$(2:1)$ *(also called a* MUX *gate) is:*

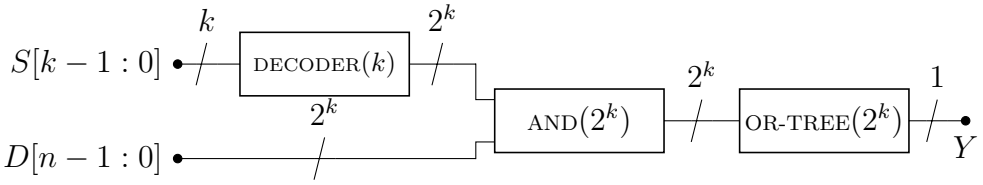


(the circ $\circ$ *indicates a negation) With cost* $c(2) = 4$ *and delay* $d(1) = 3$*. However, one later analysis, we'll consider it a gate, so we normalize* $c($MUX$)$ *and* $d($MUX$)$*. Notice this implements* IF$(S[0], D[1], D[0])$ *(cf. 2.2.11).*

Lemma 4.2.3. $c($MUX$(n:1)) = \Omega(n)$ *and* $d($MUX$(n:1)) = \Omega(\log n)$.

Proof. $\#\,\text{cone}($MUX$_n(\vec{D}, \vec{S})) \geq n$ by taking $\vec{D} = 0^n$ and flipping bit $\langle S \rangle$, hence all bits in D are in the cone. Follows from 3.3.4 and 3.3.5. □

Design 4.2.4 (Multiplexer by Decoding)**.** *We design a* MUX$(n:1)$*:*



(i) Cost: $c(n) = \Theta(n)$
(ii) Delay: $d(n) = \Theta(\log n)$

Proof. We write the recurence equations:

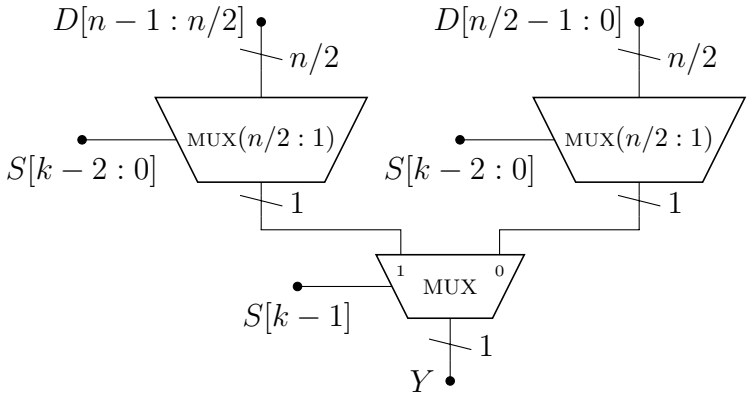
(i) Cost: for $n > 1$: $c(n) = c($DECODER$(k)) + c($AND$(2^k)) + c($OR-TREE$(2^k)) = \Theta(2^k) + \Theta(2^k) + \Theta(2^k) = \Theta(2^k) = \Theta(n)$.

(ii) Delay: for $n > 2$: $d(n) = d($DECODER$(k)) + d($AND$(2^k)) + d($OR-TREE$(2^k)) = \Theta(\log k) + \Theta(1) + \Theta(k) = \Theta(k) = \Theta(\log n)$.

(iii) Correctness: The output of the decoder is $A[i] = 1 \Leftrightarrow i = \langle \vec{S} \rangle$, after bitwise AND: $B[\langle S \rangle] = D[\langle S \rangle]$ and zeros otherwise. Taking OR with all the bits, we retrieve $D[\langle S \rangle]$ since 0 is neutral element of OR.

Hence, it is asymptotically optimal. □

Design 4.2.5 (Tree Multiplexer)**.** *We design an* MUX$(n : 1)$ *recursively, where* $n = 2^k$*, as follows:*



(i) Cost: $c(n) = (n-1) \cdot c(\text{MUX})$

(ii) Delay: $d(n) = \lceil \log_2(n) \rceil \cdot c(\text{MUX})$

Proof. We write the recurence equations:

(i) Cost: for $n > 2 : c(n) = 2c(n/2) + c(\text{MUX}) = (n-1) \cdot c(\text{MUX})$, follows by induction.

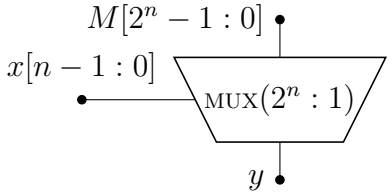
(ii) Delay: for $n > 2 : d(n) = d(n/2) + d(\text{MUX}) = \lceil \log_2(n) \rceil \cdot d(\text{MUX})$, follows by induction.

(iii) Correctness: Let Y_L and Y_R be output of the left and right MUXs respectively. By inductive hypothesis, $Y_L = D[n/2 + \langle S[k-2:0] \rangle]$ and $Y_R = D[\langle S[k-2:0] \rangle]$. If:

- $S[k-1] = 0$, then $Y = Y_R = D[\langle S[k-2:0] \rangle] = D[\langle S[k-1:0] \rangle]$.
- $S[k-1] = 1$, then $Y = Y_L = D[n/2 + \langle S[k-2:0] \rangle] = D[\langle S[k-1:0] \rangle]$.

Hence, the design also is asymptotically optimal. □

Design 4.2.6 (Brute Force Design)**.**
Every boolean function $B : \{0,1\}^n \to \{0,1\}$ *can be implemented by a* MUX$(2^n : 1)$ *as follows: Let* $M[\langle \vec{x} \rangle] = B(\vec{x})$ *preset values and*



connect $\vec{x}$ to the selector. Of course, can also implement $B : \{0,1\}^n \to \{0,1\}^k$ taking bitwise (cf. 3.4.9).

Corollary 4.2.7. *For any boolean function $B : \{0,1\}^n \to \{0,1\}^k$, the optimal cost and delay: $c_{opt}(\mathbb{C}_B) = \mathcal{O}(k\,2^n)$ and $d_{opt}(\mathbb{C}_B) = \mathcal{O}(n)$.*

Definition 4.2.8 (Shifter)**.** *A combinational circuit $\mathbb{C}$ is a shifter with inputs $x[n-1:0]$, $S[k-1:0]$ (where $n = 2^k$) and output $y[n-1:0]$ s.t.: $y[n-1:\langle S\rangle] = x[n-\langle S\rangle-1:0]$ if it's a left shifter and $y[n-\langle S\rangle-1:0] = x[n-1:\langle S\rangle]$ if it's a right shifter. A bidirectional shift has another input $\ell \in \{0,1\}$, such that it's a left shifter if $\ell = 1$ and a right shifter if $\ell = 0$.*

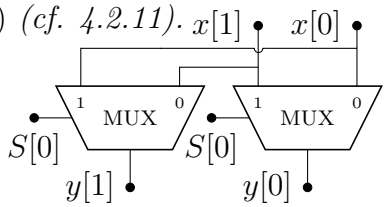
Theorem 4.2.9 (Peppinger Yao)**.** *Let $\mathbb{C}$ be a shifter, then $c(\mathbb{C}) = \Omega(n \log n)$*

Remark 4.2.10. *For a fixed $\vec{S}$, shifting can be done for free (zero cost, zero delay) by relocating wires.*

Definition 4.2.11 (Cyclic Left Shift)**.** *Define $\text{CLS} : \{0,1\}^n \to \{0,1\}^n$ as $\text{CLS}(\vec{x})[j] = x[\text{mod}(j-1,n)]$. We can extend to operations by composition: $\text{CLS}^k : \text{CLS}^k(\vec{x})[j] = x[\text{mod}(j-k,n)]$ (we also get associativity between the operations).*

Definition 4.2.12 (Cyclic Shifters)**.** *A cyclic (barrel) shifter, denoted* BARREL-SHIFTER(n)*, is a combinational circuit with inputs $x[n-1:0]$, $S[k-1:0]$ (where $n = 2^k$) and output $y[n-1:0]$ s.t.: $\vec{y}$ is a cyclic left shift of $\vec{x}$ by $\langle\vec{S}\rangle$ positions, $y[j] = x[\text{mod}(j - \langle\vec{S}\rangle, n)]$, that is, $\vec{y} = \text{CLS}^{\langle\vec{S}\rangle}(\vec{x})$ (cf. 4.2.11).*

Example 4.2.13.
The basic BARREL-SHIFTER(2) *is as follows, with cost $c(1) = 2$ and delay $d(1) = 1$.*



Design 4.2.14 (Brute Force Design)**.** *We can implement the cyclic shift by manually changing the wires and selecting in a* MUX *as follows: Let $M_i[j] = \text{CLS}^j(\vec{x})[i]$ (cf. 4.2.10,4.2.11) and use the design 4.2.6 with selector $\vec{S}$.*

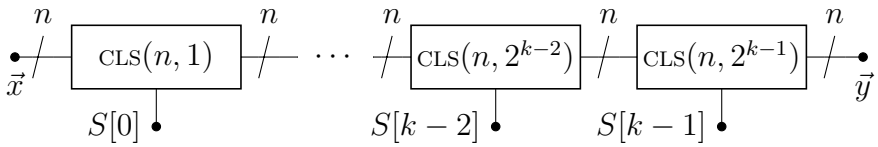
(i) Cost: $c(n) = \Theta(n^2)$

(ii) Delay: $d(n) = \Theta(\log n)$

Lemma 4.2.15. *$c($*BARREL-SHIFTER$(n)) = \Omega(n)$ and $d($*BARREL-SHIFTER$(n)) = \Omega(\log n)$.*

Proof. $\#\operatorname{cone}(\textsc{cls}^{\langle S\rangle}) = n$ by taking $\vec{x} = 0^n$ and flipping each bit, hence they are in the cone. Follows from 3.3.4 and 3.3.5. □

Design 4.2.16 (Barrel Shifter in Binary)**.** *Let* $\textsc{cls}(n, j)$ *be a combinational circuit with inputs* $x[n-1:0]$*,* $s \in \{0,1\}$ *and output* $y[n-1:0]$ *such that* $\vec{y} = \begin{cases} \textsc{cls}^j(\vec{x}) & \text{if } s = 1 \\ \vec{x} & \text{if } s = 0 \end{cases} = \textsc{cls}^{sj}(\vec{x})$*, which we can implement with* $M_i[1] = \textsc{cls}^j(\vec{x})[i]$ *(cf. 4.2.10,4.2.11) and* $M_i[0] = x[i]$ *and use the design 4.2.6 with selector* s*. Then, we design a* BARREL-SHIFTER(n) *recursively as follows:*



(i) Cost: $c(n) = n\log_2(n) \cdot c(\textsc{mux})$
(ii) Delay: $d(n) = \log_2(n)$

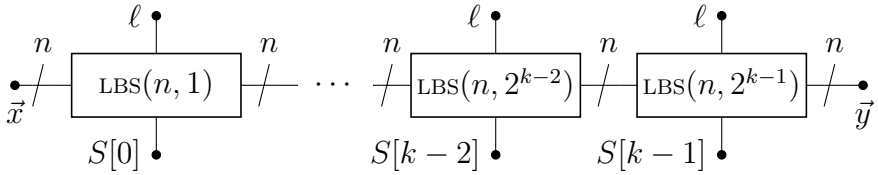
Proof. We write the recurence equations:
(i) Cost: $c(n) = \sum_{j=0}^{k-1} c(\textsc{cls}(n, 2^j)) = \sum_{j=0}^{k-1} n = k \cdot n = n\log_2(n)$.
(ii) Delay: $d(n) = \sum_{j=0}^{k-1} d(\textsc{cls}(n, 2^j)) = \sum_{j=0}^{k-1} 1 = k = \log_2(n)$.
(iii) Correctness: Composing: $\vec{y} = \textsc{cls}^{\sum_{j=0}^{k-1} S[j]\,2^j}(\vec{x}) = \textsc{cls}^{\langle \vec{S}\rangle}(\vec{x})$.
Hence, it is asymptotically optimal. □

Definition 4.2.17 (Logical Shift)**.** *Define the logical left shift:* $\textsc{lls} : \{0,1\}^n \to \{0,1\}^n$ *as* $\textsc{lls}(\vec{x}) = x[n-2:0] \circ 0$*. We can extend:* $\textsc{lls}^k : \textsc{lls}^k(\vec{x}) = x[n-k-1:0] \circ 0^k$*). Similarly, we define the logical right shift:* $\textsc{lls} : \{0,1\}^n \to \{0,1\}^n$ *as* $\textsc{lrs}(\vec{x}) = 0 \circ x[n-1:1]$ *and* $\textsc{lrs}^k(\vec{x}) = 0^k \circ x[n-1:k]$.

Definition 4.2.18 (Logical Shifters)**.** *A logical shifter, denoted* L-SHIFT(n)*, is a combinational circuit with inputs* $x[n-1:0]$*,* $S[k-1:0]$ *(where* $n = 2^k$*),* $\ell \in \{0,1\}$ *and output* $y[n-1:0]$ *s.t.:* $\vec{y}$ *is a logical left shift (*$\ell = 1$*) or right shift (*$\ell = 0$*) of* $\vec{x}$ *by* $\langle \vec{S}\rangle$ *positions,* $\vec{y} = \begin{cases} \textsc{lls}^{\langle \vec{S}\rangle}(\vec{x}) & \text{if } \ell = 1 \\ \textsc{lrs}^{\langle \vec{S}\rangle}(\vec{x}) & \text{if } \ell = 0 \end{cases}$ *(cf. 4.2.17).*

Design 4.2.19 (Logical Shifter in Binary)**.** *Let* $\textsc{lbs}(n, j)$ *be a combinational circuit with inputs* $x[n-1:0]$*,* $s \in \{0,1\}$*,* $\ell \in \{0,1\}$ *and output* $y[n-1:0]$

such that $\vec{y} = \begin{cases} \text{LLS}^j(\vec{x}) & \text{if } s = 1 \text{ and } \ell = 1 \\ \text{LRS}^j(\vec{x}) & \text{if } s = 1 \text{ and } \ell = 0 \\ \vec{x} & \text{if } s = 0 \end{cases} = \begin{cases} \text{LLS}^{sj}(\vec{x}) & \text{if } \ell = 1 \\ \text{LRS}^{sj}(\vec{x}) & \text{if } \ell = 0 \end{cases}$, *which we can implement by 4.2.10 and two multiplexers, with selectors* ℓ *and* s*. Then, we design a* L-SHIFT(n) *recursively as follows:*



(i) Cost: $c(n) = 2n \log_2(n)$
(ii) Delay: $d(n) = 2 \log_2(n)$

Proof. We write the recurence equations:
(i) Cost: $c(n) = \sum_{j=0}^{k-1} c(\text{LBS}(n, 2^j)) = \sum_{j=0}^{k-1} 2n = 2k \cdot n = 2n \log_2(n)$.
(ii) Delay: $d(n) = \sum_{j=0}^{k-1} d(\text{LBS}(n, 2^j)) = \sum_{j=0}^{k-1} 2 = 2k = 2 \log_2(n)$.
(iii) Correctness: Composing:

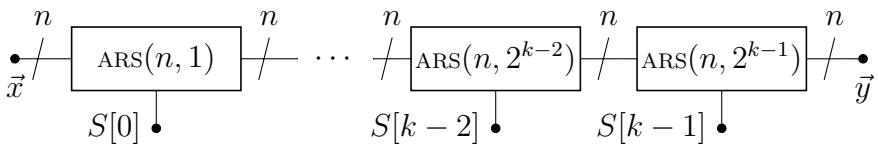
$$\vec{y} = \begin{cases} \text{LLS}^{\sum_{j=0}^{k-1} S[j]\, 2^j}(\vec{x}) & \text{if } \ell = 1 \\ \text{LRS}^{\sum_{j=0}^{k-1} S[j]\, 2^j}(\vec{x}) & \text{if } \ell = 0 \end{cases} = \begin{cases} \text{LLS}^{\langle \vec{S} \rangle}(\vec{x}) & \text{if } \ell = 1 \\ \text{LRS}^{\langle \vec{S} \rangle}(\vec{x}) & \text{if } \ell = 0 \end{cases}$$

Hence, it is asymptotically optimal. □

Definition 4.2.20 (Arithmetic Right Shift)**.** *Define* ARS : $\{0,1\}^n \to \{0,1\}^n$ *as* ARS$(\vec{x}) = x[n-1] \circ x[n-1:1]$*. We can extend:* ARS$^k(\vec{x}) = x[n-1]^k \circ x[n-1:k]$*.*

Definition 4.2.21 (Arithmetic Shifters)**.** *A cyclic (barrel) shifter, denoted* ARITH-SHIFT(n)*, is a combinational circuit with inputs* $x[n-1:0]$*,* $S[k-1:0]$ *(where* $n = 2^k$*) and output* $y[n-1:0]$ *s.t.:* $\vec{y}$ *is a arithmetic right shift of* $\vec{x}$ *by* $\langle \vec{S} \rangle$ *positions, that is,* $\vec{y} = \text{ARS}^{\langle \vec{S} \rangle}(\vec{x})$ *(cf. 4.2.20).*

Design 4.2.22 (Arithmetic Shifter in Binary)**.** *Analogous to 4.2.16, we define* ARS(n, j) *and design:*



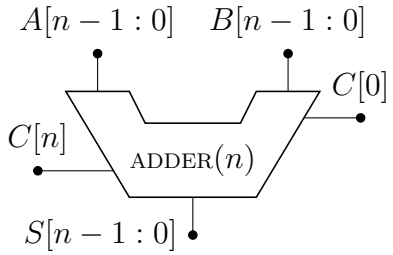
(i) *Cost:* $c(n) = n \log_2(n)$
(ii) *Delay:* $d(n) = \log_2(n)$

Proof. Analogous to 4.2.16. Hence, it is asymptotically optimal. □

4.3 Addition

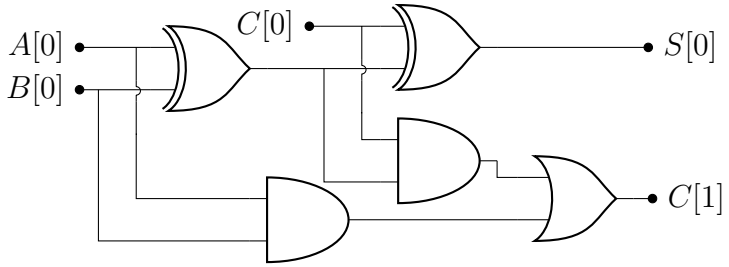
Definition 4.3.1 (Binary Adder)**.**
A binary adder, denoted $\textsc{adder}(n)$*, is a combinational circuit with inputs* $A[n-1:0]$*,* $B[n-1:0]$*,* $C[0] \in \{0,1\}$ *and outputs* $S[n-1:0]$*,* $C[n] \in \{0,1\}$ *s.t.:*

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

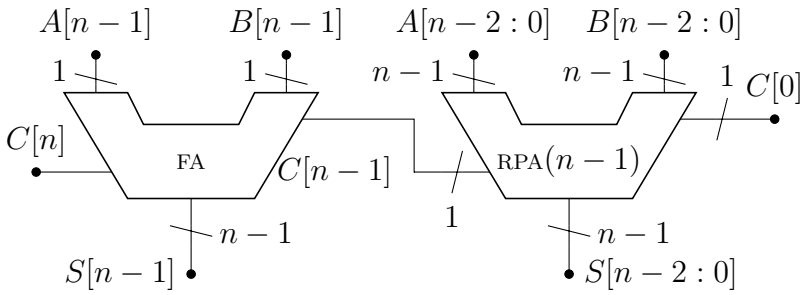


Further, $C[0]$ *is called the carry-in bit and* $C[n]$ *the carry-out.*

Example 4.3.2. *The basic adder is the full adder* $\textsc{fa} = \textsc{adder}(1)$ *with the following design:*



Design 4.3.3 (Ripple Carry Adder)**.** *Denote this design by* $\textsc{rpa}(n)$*. We design it recursively as follows:*



(i) Cost: $c(n) = n \cdot c(\textsc{fa})$
(ii) Delay: $d(n) = n \cdot d(\textsc{fa})$

Proof. We write the recurence equations:

(i) Cost: $c(n) = c(\text{FA}) + c(n-1) = n \cdot c(\text{FA})$.
(ii) Delay: $d(n) = d(\text{FA}) + d(n-1) = n \cdot d(\text{FA})$.
(iii) Correctness: $\langle\vec{A}\rangle+\langle\vec{B}\rangle+C[0] = 2^{n-1}\Big(A[n-1]+B[n-1]\Big)+\langle A[n-2:0]\rangle+$ $\langle B[n-2:0]\rangle + C[0] = 2^{n-1}\Big(A[n-1] + B[n-1]\Big) + 2^{n-1}C[n-1] +$ $\langle S[n-2:0]\rangle = 2^n C[n] + 2^{n-1}S[n-1] + \langle S[n-2:0]\rangle = 2^n C[n] + \langle\vec{S}\rangle$, by induction.

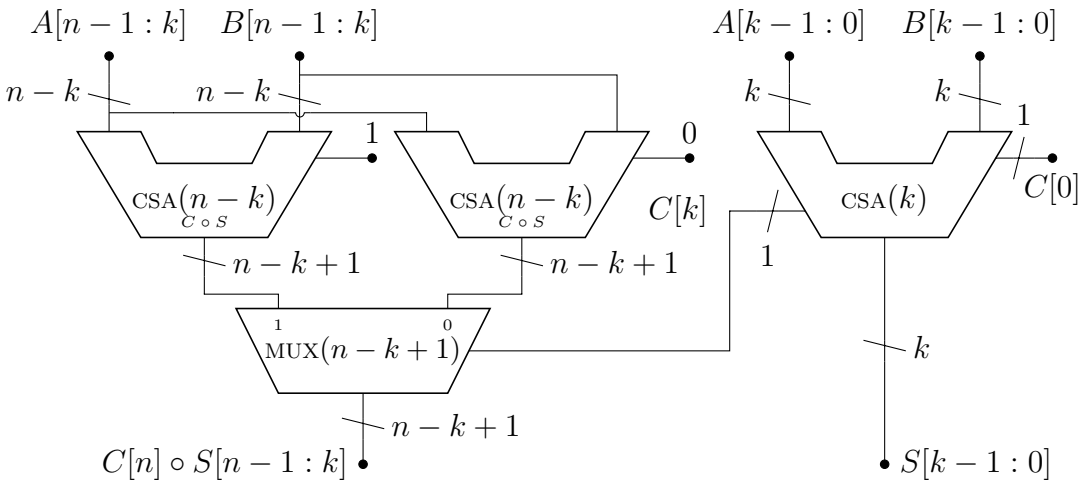
□

Lemma 4.3.4. *$c(\text{ADDER}(n)) \geq 2n$ and $d(\text{ADDER}(n)) \geq \log_2(2n+1)$.*

Proof. $\#\,\text{cone}(\text{ADDER}) = 2n+1$ by taking $\vec{A} = 0^n$ and flipping each bit of B, and vice-versa, hence they are in the cone and so it the carry-in bit. Follows from 3.3.4 and 3.3.5. □

Corollary 4.3.5. *$c(\text{ADDER}(n)) = \Omega(n)$ and $d(\text{ADDER}(n)) = \Omega(\log n)$.*

Design 4.3.6 (Conditional Sum Adder)**.** *Let $k = \lceil n/2 \rceil$. We design it recursively as follows:*



(i) Cost: $c(n) = \Theta(n^{\log_3 2})$
(ii) Delay: $d(n) = d(\text{MUX}) \cdot \lceil \log_2(n) \rceil + d(\text{FA})$

Proof. We write the recurence equations:
(i) Cost: $c(n) = 2c(n-k)+c(k)+(n-k+1)\cdot c(\text{MUX})$ follows from induction.

(ii) Delay: $d(n) = \max\{d(n-k)+d(\text{MUX}), d(k)\} = d(\text{MUX})\cdot\lceil\log_2(n)\rceil+d(\text{FA})$.

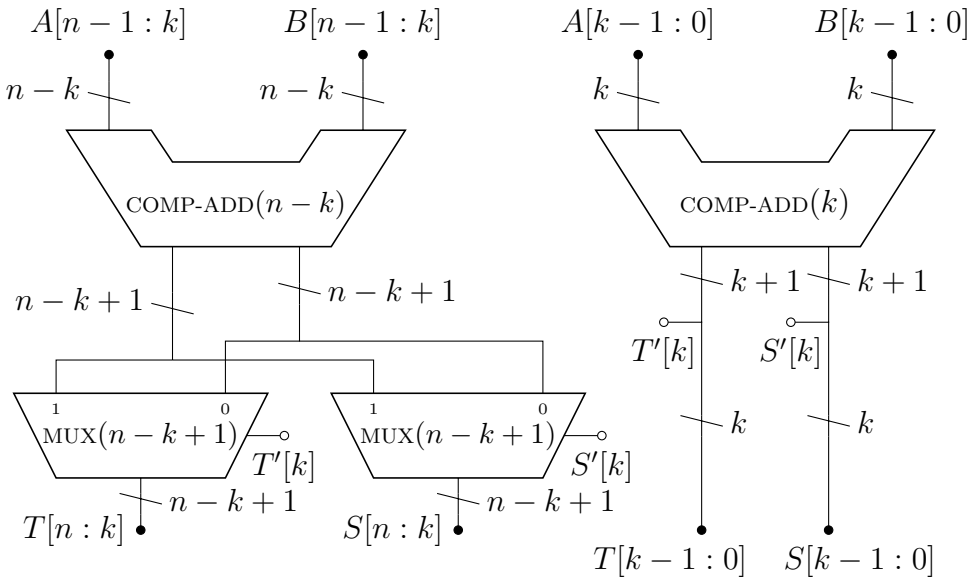
(iii) Correctness: $\langle\vec{A}\rangle + \langle\vec{B}\rangle + C[0] = 2^k\Big(\langle A[n-1:k]\rangle + \langle B[n-1:k]\rangle\Big) + \langle A[k-1:0]\rangle+\langle B[k-1:0]\rangle+C[0] = 2^k\Big(\langle A[n-1:k]\rangle+\langle B[n-1:k]\rangle\Big)+ 2^kC[k] + \langle S[k-1:0]\rangle \;=\; 2^nC[n] + 2^k\langle S[n-1:k]\rangle + \langle S[k-1:0]\rangle \;= 2^nC[n] + \langle\vec{S}\rangle$, by induction.

□

Design 4.3.7 (Compound Adder)**.** *A compound adder, denoted* COMP-ADDER(n)*, is a combinational circuit with inputs $A[n-1:0]$, $B[n-1:0]$ and outputs $S[n:0]$, $T[n:0]$ s.t.:*

$$\langle\vec{S}\rangle = \langle\vec{A}\rangle + \langle\vec{B}\rangle \;\; \textit{and} \;\; \langle\vec{T}\rangle = \langle\vec{A}\rangle + \langle\vec{B}\rangle + 1$$

We design it recursively as follows:



(i) Cost: $c(n) = \Theta(n \log n)$

(ii) Delay: $d(n) = d(\text{MUX}) \cdot \lceil\log_2(n)\rceil + d(\text{FA})$

Proof. We write the recurence equations:

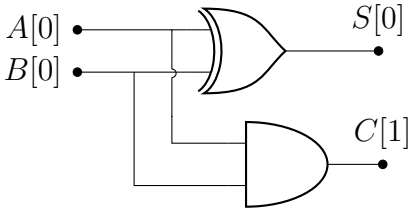
(i) Cost: $c(n) = c(n-k)+c(k)+2(n-k+1)\cdot c(\text{MUX})$ follows from induction.

(ii) Delay: $d(n) = \max\{d(n-k)+d(\text{MUX}), d(k)\} = d(\text{MUX})\cdot\lceil\log_2(n)\rceil+d(\text{FA})$.

(iii) Correctness: Analogous to the previous one.

□

Example 4.3.8 (Half Adder)**.**
The half adder HA *implements* ADDER(1) *with carry-in* $C[0] = 0$*, we get the following design with* $c(\text{HA}) = 2$ *and* $d(\text{HA}) = 1$*, we'll consider it a gate, so we normalize* $c(\text{HA})$ *and* $d(\text{HA})$*.*

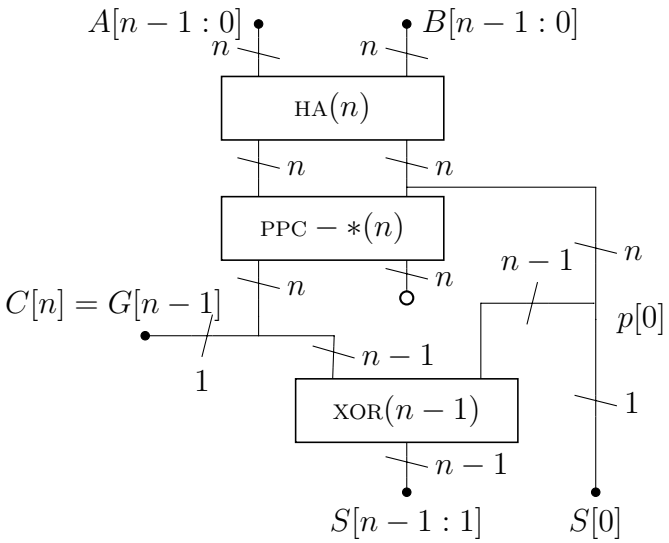


Lemma 4.3.9 (Carry Lookahead)**.** *If we can calculate* $C[n:1]$ *in with linear cost and logarithmic delay, we can implement an* ADDER(n) *with same asymptotics.*

Proof. Observe $S[i] = (A[i] \oplus B[i]) \oplus C[i]$ and $c(n) = c_{\text{carry}}(n) + 2n \cdot c(\text{XOR})$ and $d(n) = d_{\text{carry}}(n) + 2 \cdot d(\text{XOR})$. That is, the cost of the postprocessing is $\Theta(n)$ and the delay is $\Theta(1)$. □

Design 4.3.10 (Parallel Prefix Adder)**.** *In view of the previous lemma, we define an encoding* $\sigma : \{0,1\}^2 \to \{0,1,2\}$ *with* $\sigma(a,b) = a + b$*.*
We will implement using a half-adder HA*, that is, let* $\delta(a,b) = (a \wedge b, a \oplus b)$ *(MSB first) and hence:* $\sigma(a,b) = \langle \delta(a,b) \rangle$*. Define an associative operation* $* : \{0,1,2\}^2 \to \{0,1,2\}$ *with the table as follows. Using the previous encoding, this can be implemented by* $(g_1,p_1) * (g_2,p_2) = (g_1 \vee (p_1 \wedge g_2), p_1 \wedge p_2)$*. We design as follows:*

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



(i) Cost: $c(n) = \Theta(n)$
(ii) Delay: $d(n) = \Theta(\log n)$

Proof. We write the recurence equations:
(i) Cost: $c(n) = n \cdot c(\text{HA}) + c(\text{PPC} - *(n)) + (n-1) \cdot c(\text{XOR}) = \Theta(n)$.
(ii) Delay: $c(n) = d(\text{HA}) + d(\text{PPC} - *(n)) + d(\text{XOR}) = \Theta(\log n)$.
(iii) Correctness: By the previous lemma, we show $G[n-1:0] = C[n:1]$. This follows from the definition of $*$ and interpreting the output of PPC.
Hence, it is asymptotically optimal. □

Design 4.3.11 (Carry Saver)**.** *A* $3:2$ *carry save adder, denoted* CARRYADD(n)*, is a combinational circuit with inputs* $A[n-1:0]$*,* $B[n-1:0]$ *and outputs* $S[n-1:0]$ *and* $C[n:0]$ *such that* $\langle\vec{A}\rangle + \langle\vec{B}\rangle = \langle\vec{S}\rangle + \langle\vec{C}\rangle$*. We can implement it very simply by taking* FA(n) *to give* $S[n-1:0]$ *and* $C[n:1]$*, where* $C[0] = 0$*.*

Corollary 4.3.12. *We can implement an adder of* k *strings, by "saving" the carry until the end.*

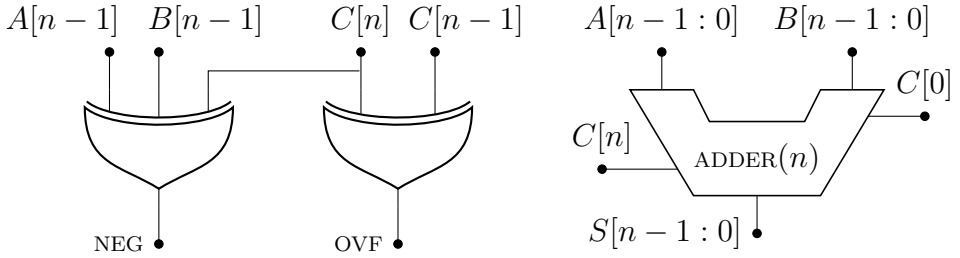
Definition 4.3.13 (Signed Adder)**.** *A signed binary adder, denoted* S-ADDER(n)*, is a combinational circuit with inputs* $A[n-1:0]$*,* $B[n-1:0]$*,* $C[0] \in \{0,1\}$ *and outputs* $S[n-1:0]$*,* $\text{OVF} \in \{0,1\}$*,* $\text{NEG} \in \{0,1\}$ *s.t. given* $\sigma = \left[\vec{A}\right] + \left[\vec{B}\right] + C[0]$*, let:*

(i) $$\text{OVF} = \begin{cases} 0 & \text{if } -2^{n-1} \le \sigma \le 2^{n-1} - 1 \\ 1 & \text{otherwise} \end{cases}$$

(ii) If $\text{OVF} = 0 \Rightarrow \left[\vec{S}\right] = \sigma$

(iii) $$\text{NEG} = \begin{cases} 0 & \text{if } \sigma \ge 0 \\ 1 & \text{if } \sigma < 0 \end{cases}$$

Design 4.3.14 (Two's Complement Adder)**.** *First, note:* $C[i] = (A[i] \oplus B[i]) \oplus S[i]$*, so we have access to* $C[n-1]$*, and we suppose we use a Carry-Lookahead adder, so we have to do no extra calculation.*



The asymptotics are the same as the adder, that is, linear cost and logarithmic delay.

Proof. Correctness: We know, from correctness of the adder, that

$$\begin{aligned}\langle S[n-2:0]\rangle + 2^{n-1}\cdot C[n-1] &= \langle A[n-2:0]\rangle + \langle B[n-2:0]\rangle + C[0]\\ &= \sigma + 2^{n-1}\cdot\Big(A[n-1]+B[n-1]\Big)\end{aligned}$$

$\Rightarrow \sigma = -2^{n-1}\cdot\Big(A[n-1]+B[n-1]-C[n-1]\Big)+\langle S[n-2:0]\rangle$. Observe, taking the last bit in the adder: $2\cdot C[n]+S[n-1] = A[n-1]+B[n-1]+C[n-1]$ we get: $\sigma = \big[S[n-1:0]\big]-2^n\Big(C[n]-C[n-1]\Big)$. Hence $\sigma \geq 2^{n-1}$ or $\sigma < -2^{n-1}$ occur iff $C[n] \neq C[n-1]$, therefore, $\text{OVF} = C[n]\oplus C[n-1]$. Now, notice if $\text{OVF} = 0$, $\text{NEG} = S[n-1]$ and if $\text{OVF} = 1$, then $\text{NEG} = \begin{cases}1 & \text{if } (C[n-1],C[n]) = (0,1)\\ 0 & \text{if } (C[n-1],C[n]) = (1,0)\end{cases} = \text{INV}(S[n-1])$ by considering the last bit in the adder, as before. Therefore, we get: $\text{NEG} = \text{OVF}\oplus S[n-1] = C[n]\oplus\big(A[n-1]\oplus B[n-1]\big)$. □

Remark 4.3.15. *In either case, ignoring the overflow,* $\big[\vec{S}\big] = \sigma \pmod{2^n} = \big[\vec{A}\big]+\big[\vec{B}\big]+C[0] \pmod{2^n}$ *where we choose the remainder between* $-2^{n-1} \leq r \leq 2^{n-1}-1$.

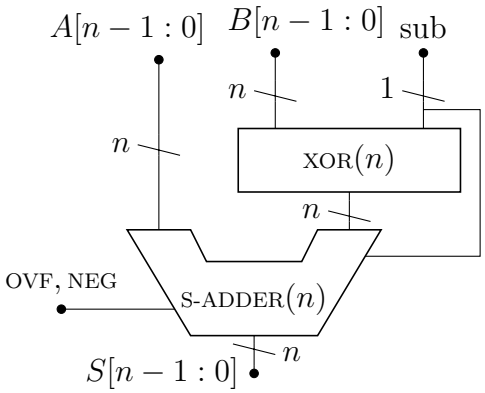
Definition 4.3.16 (Adder-Subtractor)**.** *A adder/subtractor, denoted* ADD-SUB(n)*, is a combinational circuit with inputs* $A[n-1:0]$*,* $B[n-1:0]$*,* $sub \in \{0,1\}$ *and outputs* $S[n-1:0]$*,* $\text{OVF} \in \{0,1\}$*,* $\text{NEG} \in \{0,1\}$ *s.t. given* $\sigma = \begin{cases}\big[\vec{A}\big]+\big[\vec{B}\big] & \text{if } sub = 0\\ \big[\vec{A}\big]-\big[\vec{B}\big] & \text{if } sub = 1\end{cases}$*, let:*

(i) $\text{OVF} = \begin{cases}0 & \text{if } -2^{n-1} \leq \sigma \leq 2^{n-1}-1\\ 1 & \text{otherwise}\end{cases}$

(ii) If $\text{OVF} = 0 \Rightarrow \big[\vec{S}\big] = \sigma$

(iii) $\text{NEG} = \begin{cases}0 & \text{if } \sigma \geq 0\\ 1 & \text{if } \sigma < 0\end{cases}$

Design 4.3.17. *We use the previous design in the following manner:*

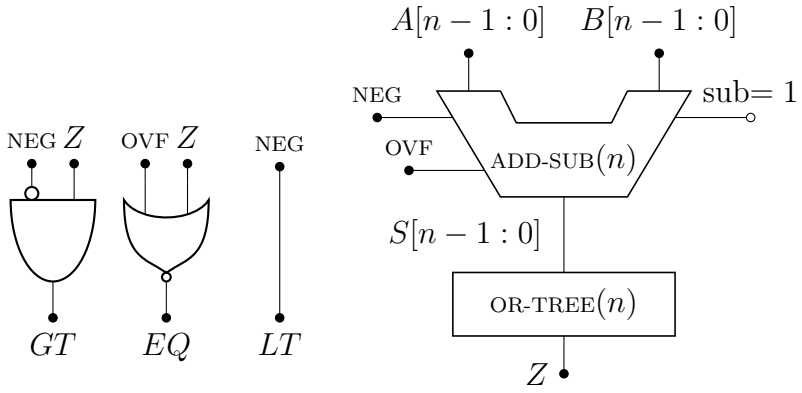


Proof. Follows from 1.3.11 □

Definition 4.3.18 (Comparator)**.** *A comparator, denoted* COMPARE(n)*, is a combinational circuit with inputs* $A[n-1:0]$*,* $B[n-1:0]$*, and output* (GT, EQ, LT)*, s.t.*

$$(GT, EQ, LT) = \begin{cases} (1,0,0) & \text{if } [\vec{A}] > [\vec{B}] \\ (0,1,0) & \text{if } [\vec{A}] = [\vec{B}] \\ (0,0,1) & \text{if } [\vec{A}] < [\vec{B}] \end{cases}$$

Design 4.3.19 (Comparator by Addition)**.** *We use subtraction, as desgined above as follows:*



Proof. Notice $Z = 0 \Leftrightarrow \vec{S} = 0^n$:

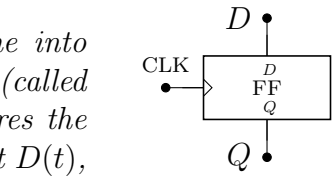
$$(GT, EQ, LT) = \begin{cases} (1,0,0) & \text{if } \sigma > 0 \\ (0,1,0) & \text{if } \sigma = 0 \\ (0,0,1) & \text{if } \sigma < 0 \end{cases} = \begin{cases} (1,0,0) & \text{if } \textsc{neg} = 0 \text{ and } Z = 1 \\ (0,1,0) & \text{if } Z = 0 \text{ and } \textsc{ovf} = 0 \\ (0,0,1) & \text{if } \textsc{neg} = 1 \end{cases}$$

□

5 Syncronous Circuits

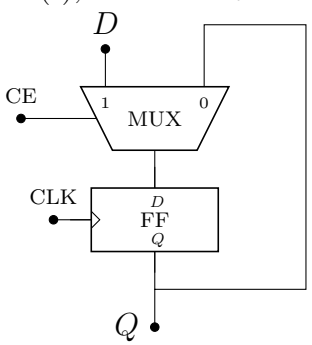
5.1 Flip Flops and Memory

Definition 5.1.1 (Clocks and Flip Flops)**.** *Define a global input* CLK *that partition the time into steps, where we denote $t \in \mathbb{N}$ the t-th partition (called clock cycles). A flip-flop is a circuit* FF *that stores the input for the next clock cycle. That is, given input $D(t)$, the output obeys: $Q(t+1) = D(t)$.*



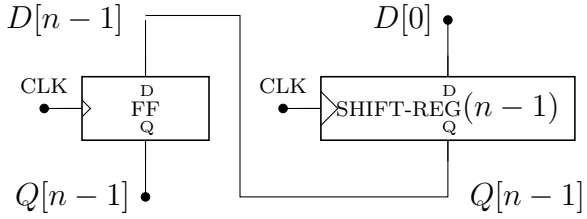
Design 5.1.2 (Clock-Enabled Flip Flop)**.** *A clock-enabled flip flop, denoted* CE-FF*, is a combinational circuit with inputs $D(t)$, $\text{CE}(t)$, and output*

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



Remark 5.1.3 (Parallel Load Register)**.** *We can bitwise extend a clock enabled flip flop* CE-FF(n)*, what we call a parallel load register.*

Design 5.1.4 (Shift Register)**.** *A shift register* SHIFT-REG(n) *is a (synchronous) circuit with inputs $D[0](t)$ and clock* CLK *and output $Q[n-1](t)$ such that $Q[n-1](t+n) = D[0](t)$. We design recursively as follows:*



Design 5.1.5 (Memory Cell)**.** *A memory cell* MEM *is a (synchronous) circuit with inputs $D_{in}(t)$, $R/\overline{W}(t)$, $\text{SEL}(t)$ and clock* CLK *and output $D_{out}(t)$ such that: Let $M(t)$ denote the state at cycle t (initialized as $M(0) = 0$), then:*

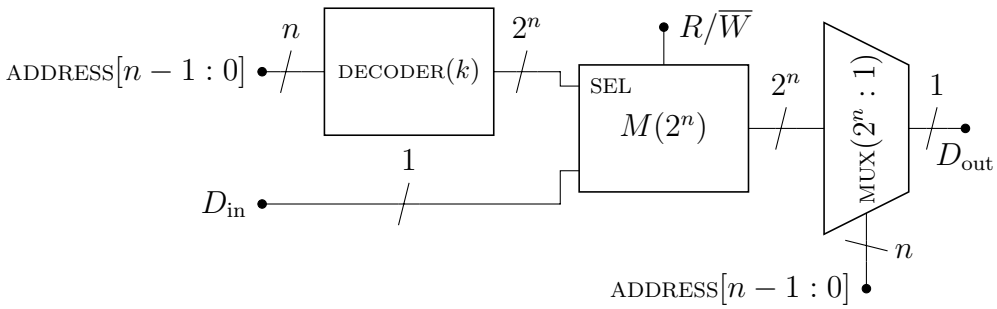
$$D_{out}(t) = M(t) \text{ and } M(t+1) = \begin{cases} D_{in}(t) & \text{if } \text{SEL}(t) = 1 \text{ and } R/\overline{W}(t) = 0 \\ M(t) & \text{otherwise} \end{cases}.$$

We design by using a CE-FF *with* $\text{CE} = \text{SEL} \wedge \neg R/\overline{W}$.

Definition 5.1.6 (RAM)**.** *A Random Access Memory* $\text{RAM}(2^n)$ *is a (synchronous) circuit with inputs* $D_{in}(t)$, $R/\overline{W}(t)$, $\text{ADDRESS}[n-1:0](t)$ *and clock* CLK *and output* $D_{out}(t)$ *such that: Let* $M[2^n-1:0](t)$ *denote the state at cycle* t *(initialized as* $M(0) = 0^{2^n}$*), then:* $D_{out}(t) = M[\langle \text{ADDRESS} \rangle](t)$ *and*

$$M[i](t+1) = \begin{cases} D_{in}(t) & \text{if } i = \langle \text{ADDRESS} \rangle \text{ and } R/\overline{W}(t) = 0 \\ M[i](t) & \text{otherwise} \end{cases}$$

Design 5.1.7 (RAM with Cells)**.** *We use* 2^n *memory cells, which we denote* $M(2^n)$*, to design* $\text{RAM}(2^n)$ *as follows:*



designing it similar to 4.2.4.

Proof. Correctness: SEL input will filter the operations given in the memory cell due to the decoder. At the end, the MUX will only output the address into D_{out}. The final MUX is not facultative, since the read operation comprises it's precise use. □

5.2 Finite State Machines

Definition 5.2.1 (FSM)**.** *An FSM is a tuple* $\mathcal{A} = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ *where:*
(i) Q *is the set of all possible states;*
(ii) Σ *is the input alphabet;*
(iii) Δ *is the output alphabet;*
(iv) $\delta : \Sigma \times Q \to Q$ *is the transition function;*
(v) $\lambda : \Sigma \times Q \to \Delta$ *is the output function;*
(vi) $q_0 \in Q$ *is the initial state.*
Given an input sequence $\{x_i\}_{i=0}^{n-1}$ *(over the alphabet* Σ*), the sequence of states is defined:* $q_{i+1} = \delta(x_i, q_i)$ *and the outputs are:* $y_i = \lambda(x_i, q_i)$ *(which are over the alphabet* Δ*).*

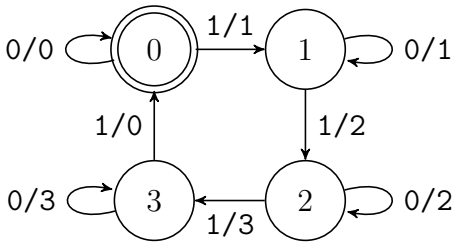
Definition 5.2.2 (State Diagram)**.** *Given an FSM* $\mathcal{A} = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ *we define a directed graph* $G_{\mathcal{A}} = (V_{\mathcal{A}}, E_{\mathcal{A}})$*, where* $V_{\mathcal{A}} = Q$ *and*

$$E_{\mathcal{A}} = \{(q, \delta(q, x)) \mid q \in Q \text{ and } x \in \Sigma\}$$

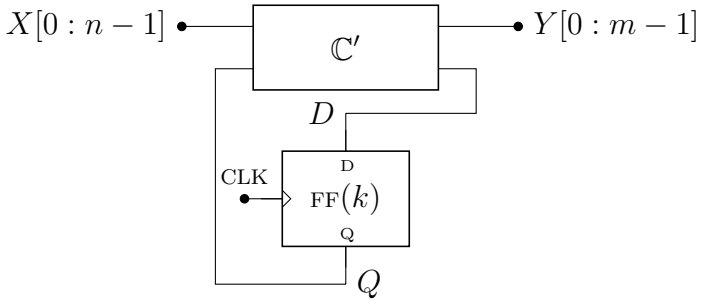
Further, we label the edges $(x, \lambda(q, x))$ *(sometimes we denote* $x/\lambda(q, x)$*). Moreover, we denote* q_0 *by a double circle.*

Example 5.2.3.
This a state diagram for a counter modulo 4*: First we define* $Q = \Delta = \{0, 1, 2, 3\}$ *and* $\Sigma = \{0, 1\}$*. And the function are:*

$$\delta(x, q) = \lambda(x, q) = (q + x) \mod 4$$

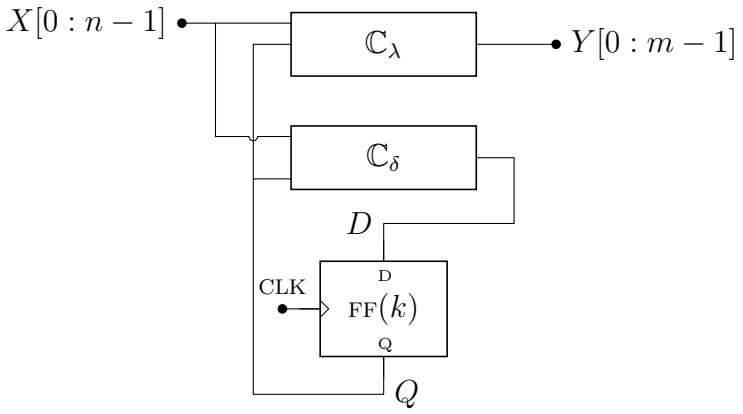


Definition 5.2.4. *A synchronous circuit is a circuit* $\mathbb{C}$ *in the following form:*



where $\mathbb{C}'$ *is a combinational circuit. That is, the* CLK *input only feeds flip flops and if we remove the flip flops, replacing* D *ports by output gates and* Q *ports by input gates, we get a combinational circuit.*

Definition 5.2.5. *The canonical form of a combinational circuit is as follows:*



which is similar to the definition, with maybe some duplication necessary.

Theorem 5.2.6. *Every synchronous circuit implements a FSM (synthesis) and every FSM can be implemented as a synchronous circuit (analysis).*

Proof. $\mathbb{C}_\lambda$ implements λ and $\mathbb{C}_\delta$ implements δ in the following way:

(Synthesis) Since the sets Q, Σ and Δ are finite, there are injective functions: $\rho : Q \to \{0,1\}^k$, $\sigma : \Sigma \to \{0,1\}^n$ and $\tau : \Delta \to \{0,1\}^m$. Define:

$$B_\delta(\rho(q), \sigma(x)) = \rho(\delta(q,x)) \text{ and } B_\lambda(\rho(q), \sigma(x)) = \tau(\lambda(q,x))$$

Notice this may not define $B_\delta : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^k$ and $B_\lambda : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^m$ uniquely.

(Analysis) Let $Q = \{0,1\}^k$, $\Sigma = \{0,1\}^n$ and $\Delta = \{0,1\}^m$, further, let δ be the function implemented by $\mathbb{C}_\delta$ and λ be the function implemented by $\mathbb{C}_\lambda$. Lastly, q_0 is the value which the flip flops are initialized.

Together with 11, we get the states of the flip flops $S_i = \phi(q_i)$. □

Algorithm 11 Simulation, for $(\mathbb{C}_\delta, \mathbb{C}_\lambda, \vec{S}_0)$ with inputs $\{\vec{x}_i\}_{i=1}^n$

for $0 \le i \le n-1$ **do**
  $\vec{y_i} \leftarrow \text{SIM}(C_\lambda, \vec{x}_i, \vec{S}_i)$
  $\vec{NS}_i \leftarrow \text{SIM}(C_\delta, \vec{x}_i, \vec{S}_i)$
  $\vec{S}_{i+1} \leftarrow \vec{NS}_i$
end for

Design 5.2.7 (Parallelization)**.** *Suppose we have a synchronous circuit that takes inputs* $\{x_i\}_{i=1}^n$ *and outputs* $\{y_i\}_{i=1}^n$*, and we want to implement a combinational circuit that taking in* $x[0:n-1]$ *and outputs* $y[0:n-1]$ *with the same functionality. We apply the following:*

- *Let* $\delta_i(q) = \delta(q, x_i)$ *partial application (currying). Then,* $\forall\, 0 \leq j \leq n-1$*, we get:* $q_{j+1} = \delta_j \circ \delta_{j-1} \circ \cdots \circ \delta_0(q_0)$*.*
- *Let* $\mathcal{F} \subseteq \{f : \{0,1\}^k \to \{0,1\}^k\}$*, where* k *is the number of flip flops in the synchronous circuit, such that* $\{\delta_0, \delta_1, \cdots, \delta_{n-1}\} \subseteq \mathcal{F}$ *and* $\mathcal{F}$ *is closed under composition.*
- *We encode* $\varphi : \mathcal{F} \to \{0,1\}^f$*. Let* $\star$ *be an (associative) operation on* $\{0,1\}^f$ *such that:* $\varphi(f \circ g) = \varphi(f) \star \varphi(g)$*.*
- *Then, let* $\pi_j = \varphi(\delta_j) \star \varphi(\delta_{j-1}) \star \cdots \star \varphi(\delta_0)$*. We can implement this by Parallel Prefix Calculation of* $\star$*.*
- *Lastly, we implement a function applicator:* $\psi(\varphi(f)) = f(q_0)$*. Then,* $y_i = \lambda(\psi(\pi_i), x_i)$*.*

6 Simplified DLX

6.1 Instruction Set Architecture

Definition 6.1.1. *We define the structures:*

- *Registers:*
  1. *32 General Purpose Registers (GPRs), denoted* $R0 - R31$*. All registers are directly manipulated by the program, however, we require* $R0 = 0^{32}$ *always (cannot be altered).*
  2. *Program Counter (PC)*
  3. *Instruction Register (IR)*
  4. *Memory Address Register (MAR)*
  5. *Memory Data Register (MDR)*
- *Memory: An array* $M[0 : 2^{32} - 1]$ *of words (*32 *bits or* 4 *bytes), such that each element* $M[i]$ *holds a word. We implement it as* RAM*, so we allow two operations:*
  1. *Write:* $M[\langle MAR \rangle] \leftarrow MDR$
  2. *Read:* $MDR \leftarrow M[\langle MAR \rangle]$

Definition 6.1.2 (Instructions)**.** *We define two types of instructions in (our dialect of) Assembly:*

Type	Instruction	Encoding
I-type	op RD RS1 imm	6 | 5 | 5 | 16 opcode | RS1 | RD | imm
R-type	op RD RS1 RS2	6 | 5 | 5 | 5 | 5 | 6 opcode | RS1 | RS2 | RD | X | func

where "op" is a mnemonic we'll define for each operation and imm is a 16 bit integer. After every instruction (except some special ones we'll define) we increment the program counter by $1 \pmod{32}$*.*

Definition 6.1.3 (Load and Write)**.** *We define the instructions:*

Instruction				Semantics
lw	RD	RS1	imm	RD:=M[sext(imm)+RS1]
sw	RD	RS1	imm	M[sext(imm)+RS1]:=RD

Where sext is sign extension (cf. 1.3.12). The calculation is (technically) done by effective address: $ea = \langle RS1 \rangle + \left[imm \right] \mod 2^{32}$*. The effective address is store on* MAR *(after) the calculation. The data is then loaded in* MDR *before the transfers.*

Definition 6.1.4 (Arithmetic Instructions)**.** *We define the instructions:*

Instruction				Semantics
addi	RD	RS1	imm	RD:=RS1+sext(imm)
slti	RD	RS1	imm	RD:=(RS1 < sext(imm))
seqi	RD	RS1	imm	RD:=(RS1 = sext(imm))
sgti	RD	RS1	imm	RD:=(RS1 > sext(imm))
slei	RD	RS1	imm	RD:=(RS1 $\leq$ sext(imm))
snei	RD	RS1	imm	RD:=(RS1 $\neq$ sext(imm))
sgei	RD	RS1	imm	RD:=(RS1 $\geq$ sext(imm))
sll	RD	RS1		RD:=RS1 << 1
srl	RD	RS1		RD:=RS1 >> 1
add	RD	RS1	RS2	RD:=RS1 + RS2
sub	RD	RS1	RS2	RD:=RS1 − RS2
and	RD	RS1	RS2	RD:=RS1 $\wedge$ RS2
or	RD	RS1	RS2	RD:=RS1 $\vee$ RS2
xor	RD	RS1	RS2	RD:=RS1 $\oplus$ RS2

Definition 6.1.5 (Branch and Jump)**.** *The instruction that is executed by the hardware is at line number stored by* PC*. Hence, we define the instructions:*

Instruction			Semantics
beqz	RS1	imm	PC=$\begin{cases}\text{PC+1+sext(imm)} & \text{if } RS1 = 0\\ \text{PC+1} & \text{if } RS1 \neq 0\end{cases}$
bnez	RS1	imm	PC=$\begin{cases}\text{PC+1+sext(imm)} & \text{if } RS1 \neq 0\\ \text{PC+1} & \text{if } RS1 = 0\end{cases}$
jr	RS1		PC=RS1
jalr	RS1		R31=PC+1 ; PC=RS1

we take those expressions $\bmod 2^{32}$*. This allows us to jump between program instructions and create if statements and loop.*

Design 6.1.6 (For Loop)**.** *We implement the following C code in DLX:*

```
1   int sum = 0, N = 10;
2   for(int i = 0; i < N; i++){
3     sum += i;
4   }
```

We use the followng data binding:

Variable	*sum*	*i*	*N*	*temp*	$i < N$
Register	R1	R2	R3	R4	R5

The assembly code becomes:

```
1   addi R1 R0 0   ; sum = 0
2   addi R3 R0 10  ; N = 10
3   addi R2 R0 0   ; i = 0
4   sub  R4 R3 R2  ; temp = N - i
5   sgti R5 R4 0   ; i < N
6   beqz R5  3     ; for loop
7   add  R1 R1 R2  ; sum += i
8   addi R2 R2 1   ; i++
9   beqz R0 -6     ; for loop
10  halt
```

Design 6.1.7 (Array Indexing). *We implement the following C code in DLX:*

```
int sum = 0, N = 10, *A;
for(int i = 0; i < N-1; i++){
  if (A[i] < A[i+1]) {
    sum++;
  }
}
```

Say the begging of the array A is stored at address indexed 30. *We use the following data binding:*

Variable	sum	i	N	A	$temp$	$comp$	$A+i$	$A[i]$	$A[i+1]$
Register	R1	R2	R3	R4	R5	R6	R7	R8	R9

The assembly code becomes:

```
addi  R1  R0   0 ; sum = 0
addi  R3  R0  10 ; N = 10
addi  R4  R0  30 ; A (ptr) = 30
addi  R2  R0   0 ; i = 0
sub   R5  R3  R2 ; temp = N - i
sgti  R6  R5   1 ; comp = i < N - 1
beqz  R6   9     ; for loop
add   R7  R4  R2 ; A + i
lw    R8  R7   0 ; A[i]
lw    R9  R7   1 ; A[i+1]
sub   R5  R9  R8 ; temp = A[i+1] - A[i]
sgti  R6  R5   0 ; comp = A[i] < A[i+1]
beqz  R6  1      ; if statement
addi  R1  R1   1 ; sum++
addi  R2  R2   1 ; i++
beqz  R0 -12     ; for loop
halt
```

6.2 Implementation

Definition 6.2.1. *We define the structures:*

- *Datapath, with Environments: ALU, Shifter, IR, GPRs, and Parallel Load Registers: PC, MAR, MDR, plus the registers A,B, and C.*
- *Control (DLX)*
- *Memory Controller*

Definition 6.2.2 (Memory Controller)**.** *The Memory Controller is a synchronous circuit with inputs* $\text{IN}[31:0]$, $\text{ADDRESS}[31:0]$, $\texttt{MR}, \texttt{MW} \in \{0,1\}$ *and output* $\text{OUT}[31:0]$ *and* $\texttt{busy} \in \{0,1\}$*, such that:*
(Busy) The input is stable while $\texttt{busy}(t) = 1$
(Free) If $\texttt{busy}(t) = 0$ *and* $\texttt{busy}(t-1) = 1$*, then:*
(Read) If $\texttt{MR}(t-1) = 1 : \text{OUT}(t) \leftarrow M[\langle \text{ADDRESS}(t-1) \rangle](t-1)$
(Write) If $\texttt{MW}(t-1) = 1 : M[\langle \text{ADDRESS}(t-1) \rangle](t-1) \leftarrow \text{OUT}(t)$

Definition 6.2.3. *The Datapath defines the following environments:*

1. *ALU: A combinational circuit with inputs* $x[31:0]$, $y[31:0]$, $\texttt{type} \in \{0,1\}^5$ *and output* $z[31:0]$ *such that:* $z = B_{\texttt{type}}(x,y)$.

2. *Shifter: A bidirectional shifter by* 1 *position.*

3. *IR: A synchronous circuit with inputs* $\text{DI}[31:0]$, $\texttt{IRce}, \texttt{JLINK}, \texttt{Itype} \in \{0,1\}$ *and output* $\texttt{Inst}[31:0]$ *(instruction),* $\texttt{Imm}[31:0]$ *(immediate) and the GPR addresses* $\texttt{Aadr}[4:0], \texttt{Badr}[4:0], \texttt{Cadr}[4:0]$*, that is:*

$$\texttt{Inst}(t+1) = \begin{cases} \texttt{Inst}(t) & \text{if } \texttt{IRce}(t) = 0 \\ \text{DI}(t) & \text{if } \texttt{IRce}(t) = 1 \end{cases}$$

$$\texttt{Imm}[31:0] = sext(imm)$$

$$\texttt{Aadr}[4:0] = \texttt{Inst}[25:21] \text{ and } \texttt{Badr}[4:0] = \texttt{Inst}[20:16]$$

$$\texttt{Cadr}[4:0] = \begin{cases} 1^5 & \text{if } \texttt{JLINK} = 1 \\ \texttt{Inst}[20:16](t) & \text{if } \texttt{Itype}(t) = 1 \text{ and } \texttt{JLINK} = 0 \\ \texttt{Inst}[15:11](t) & \text{otherwise} \end{cases}$$

4. *PC: Simply a 32-bit (CE) parallel load register, initialized at* 0^{32}.

5. *GPR: A synchronous circuit with inputs* $\texttt{Aadr}[4:0], \texttt{Badr}[4:0], \texttt{Cadr}[4:0]$*, a data input* $C[31:0]$ *(from C register),* $\texttt{GPR}_{\texttt{W}}\texttt{E} \in \{0,1\}$ *(write*

enable) and outputs a flag $\texttt{AEQZ} \in \{0,1\}$*, and* $A_{in}[31:0]$*,* $B_{in}[31:0]$ *(inputs to A and B registers), such that: the environment holds* 32 32*-bit registers* $R[31:0]$*, each initialized at* 0^{32} *and performs:*

(a) If $\texttt{GPR}_\texttt{W}\texttt{E} = 1$ *and* $\texttt{Cadr} \neq 0^5$*,* $R[\langle \texttt{Cadr} \rangle](t+1) \leftarrow C(t)$
(b) If $\texttt{GPR}_\texttt{W}\texttt{E} = 0$*,* $A_{in}(t) \leftarrow R[\langle \texttt{Aadr} \rangle](t)$ *and* $B_{in}(t) \leftarrow R[\langle \texttt{Badr} \rangle](t)$
(c) Further, $\texttt{AEQZ}(t+1) = \text{NOT}(\text{OR}(A_{out}))$*, where* A_{out} *is the output of the register A.*

These are connected between themselves and the memory controller are:

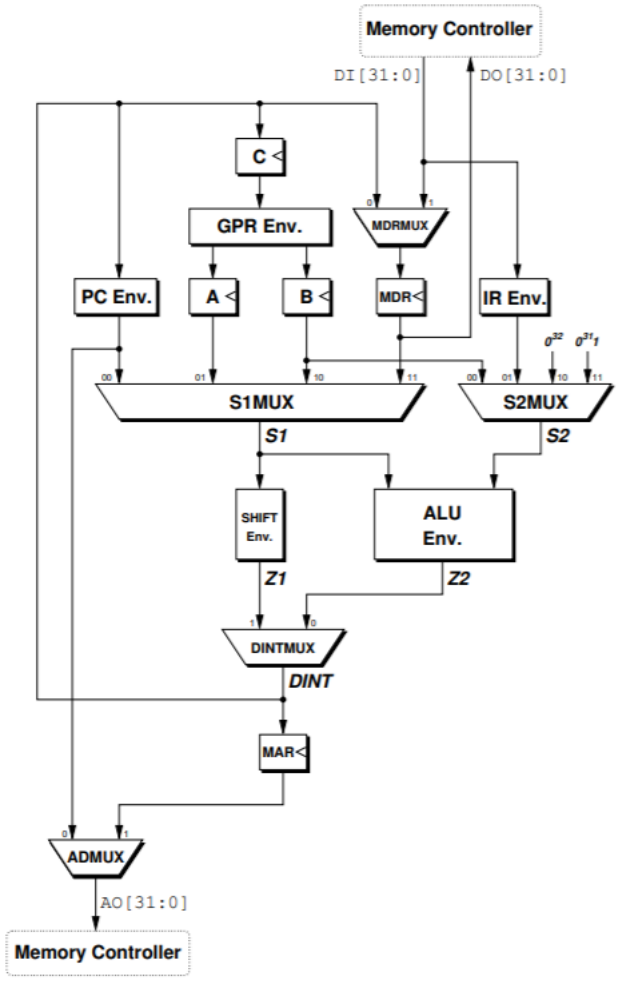


Figure 1: Datapath

Remark 6.2.4. *The IR env. and GPR env. are connected by the addresses* $\texttt{Aadr}[4:0], \texttt{Badr}[4:0], \texttt{Cadr}[4:0]$.

Definition 6.2.5. *The DLX controller is* 19 *state FSM with state diagram depicted below:*

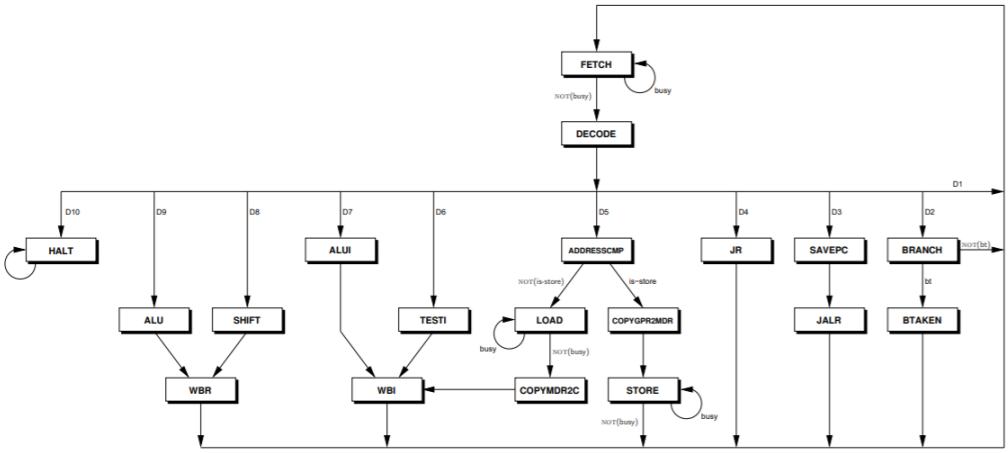


Figure 2: Controller

and here is a table with the RTL instructions (meaning Register Transfer Level) performed at each state. Each DLX instruction is broken down into a FETCH-DECODE-EXECUTE (which may contain many states) to perform it. Further, notice all executes, apart from HALT, DECODE, BRANCH, and instructions with busy flags, have $\deg_{out} = 1$.

State	RTL Instruction	Active Output
Fetch	$IR = M[PC]$	MR, IRce
Decode	A = RS1, B = RS2, PC=PC+1	Ace, Bce, $S2sel[1]$, $S2sel[0]$, add, PCce
Alu	$C = A$ op B	$S1sel[0]$, Cce, some in $ALUF[2:0]$
TestI	$C = (A$ rel imm)	$S1sel[0]$, $S2sel[0]$, Cce, $test$, Itype, some in $ALUF[2:0]$
AluI(add)	$C = A + imm$	$S1sel[0]$, $S2sel[0]$, Cce, add, Itype
Shift	$C = A$ shift 1	$S1sel[0]$, Cce, $DINTsel$, $shift, right$
Adr.Comp	$MAR = A + imm$	$S1sel[0]$, $S2sel[0]$, MARce, add
Load	$MDR = M[MAR]$	MDRce, $ADsel$, MR, $MDRsel$
Store	$M[MAR] = MDR$	$ADsel$, MW
CopyMDR2C	$C = MDR(>> 0)$	$S1sel[0]$, $S1sel[1]$, $S2sel[1]$, $DINTsel$, Cce
CopyGPR2MDR	$MDR = B(<< 0)$	$S1sel[1]$, $S2sel[1]$, $DINTsel$, MDRce
WBR	$RD = C$(R-type)	GPR_WE
WBI	$RD = C$(I-type)	GPR_WE, Itype
Branch	$bt =$ AEQZ $\oplus$ Inst[26]	
Btaken	$PC = PC + imm$	$S2sel[0]$, add, PCce
JR	$PC = A$	$S1sel[0]$, $S2sel[1]$, add, PCce
SavePC	$C = PC$	$S2sel[1]$, add, Cce
JALR	$PC = A;\ R31 = C$	$S1sel[0]$, $S2sel[1]$, add, PCce, GPR_WE, JLINK
HALT		

References

[EvenMedina] Guy Even and Moti Medina, "Digital Logic Design: A Rigorous Approach", Cambridge University Press, 2019. Available online at `http://hyde.eng.tau.ac.il/Even-Medina/master.pdf`

[Even04] Guy Even, "Computer Structure; Spring 2004 Lecture Notes", manuscript, 2004. Available online at `http://hyde.eng.tau.ac.il/Computer_Structure04/Notes/master.pdf`

[Even06] Guy Even, "On teaching fast adder designs: revisiting Ladner & Fischer"

[LadnerFischer] R. Ladner and M. Fischer, "Parallel prefix computation", J. Assoc. Comput. Mach., 27, pp. 831–838, 1980.