

## **Comprehensive CS**

**Nathan Warner**



**Northern Illinois  
University**

Computer Science  
Northern Illinois University  
United States

## Contents

<b>1 Theory of Computation</b>	<b>4</b>
1.1 Natural Languages, Formal languages: Definitions and theorems . . . . .	4
1.2 Regular languages . . . . .	7
1.2.1 Finite Automata . . . . .	7
1.2.2 Finite Automata: More examples . . . . .	14
1.2.3 nondeterministic Finite automata (NFA) . . . . .	15
1.2.4 Regular expressions . . . . .	26
1.2.5 Properties of regular languages . . . . .	40
1.2.6 Applications of finite automata . . . . .	43
1.3 Nonregular languages . . . . .	44
1.3.1 Pumping lemma examples . . . . .	48
1.4 Context free grammars . . . . .	50
1.4.1 Pushdown automata (PDA) . . . . .	53
1.4.2 Chomsky Normal Form . . . . .	56
1.5 Equivalence of Context-Free Grammars and Pushdown Automata . . . . .	60
1.6 Non context free languages . . . . .	64
1.7 Turing machines . . . . .	70
1.7.1 Decidability and Languages Accepted by Turing Machines . . . . .	77
1.7.2 TM Variations . . . . .	84
1.7.3 Encoding TMs . . . . .	87
1.7.4 Revisiting Recursively Enumerable Languages . . . . .	90
1.7.5 Universal turing machine . . . . .	92
1.7.6 The halting problem . . . . .	97
1.8 Complexity theory . . . . .	100
1.8.1 Time complexity . . . . .	100

<b>2 DSA</b>	102
2.1 C++ Stuff . . . . .	102
2.1.1 Type declarations . . . . .	102
2.1.2 G++ . . . . .	104
<b>3 Databases</b>	106
3.1 Introduction to databases (db concepts) . . . . .	106
3.1.1 Definitions and theorems . . . . .	106
3.2 Conceptual Modeling and ER Diagrams . . . . .	112
3.2.1 Definitions and theorems . . . . .	112
3.3 The Relational Model . . . . .	121
3.4 Relational Model Normalization . . . . .	123
3.5 ERD to Relations (Conceptual to logical) . . . . .	131
3.6 MariaDB, SQL . . . . .	139
3.6.1 DDL . . . . .	140
3.6.2 DML except SELECT . . . . .	145
3.6.3 DML SELECT . . . . .	148
3.7 Frontend: Html . . . . .	162
3.8 PHP . . . . .	171
3.9 SQL via PHP: PDO . . . . .	180
3.10 Transactions and concurrency control . . . . .	187
3.11 Mariadb in C++ . . . . .	193
<b>4 Software engineering</b>	200
4.1 Introduction . . . . .	200
<b>5 Assembler</b>	204
5.1 Introduction to the mainframe, assist, and TSO/ISPF . . . . .	204
5.2 Using ISPF . . . . .	207
5.3 Number systems and computer storage . . . . .	212
5.4 Basic concepts . . . . .	223
5.5 Decimal arithmetic . . . . .	242
5.6 Internal Subroutines . . . . .	255
5.7 Standard linkage and external subroutines . . . . .	259

5.8	Tables . . . . .	263
<b>6</b>	<b>Computer Architecture and Syst Org</b>	<b>265</b>
6.1	Chapter 1: History of computers . . . . .	265
6.2	Chapter 3: Boolean algebra, circuits, registers, and memory . . . . .	271
6.2.1	Boolean algebra . . . . .	271
6.2.2	Boolean identities . . . . .	274
6.2.3	Logic gates . . . . .	276
6.2.4	Combinatorial circuits . . . . .	280
6.2.5	Sequential Circuits . . . . .	286
6.3	Chapter 2: Data representation, integer and fixed point . . . . .	295
6.3.1	Binary Fractions . . . . .	295
6.3.2	Twos complement . . . . .	300
6.3.3	Overflow . . . . .	303
6.3.4	Binary math . . . . .	304
6.3.5	Endianness (byte ordering) . . . . .	305
6.3.6	Ascii . . . . .	307
6.3.7	Unicode . . . . .	309
6.3.8	UTF . . . . .	312
6.3.9	Floating point concepts . . . . .	319
6.4	Chapter 4: Overview of computer architecture . . . . .	340
6.4.1	The Von Neumann Model . . . . .	340
6.4.2	CPU basics and the Bus . . . . .	342
6.4.3	Marie . . . . .	345
6.5	Chapter 5: Instruction set architecture . . . . .	366
6.5.1	Pipelining . . . . .	366
6.5.2	Instruction set architecture (ISA) . . . . .	369
6.6	Chapter 8: Software . . . . .	374
6.6.1	Assemblers . . . . .	374
6.6.2	Binding . . . . .	379
6.6.3	Compilers . . . . .	382
6.6.4	Java case study . . . . .	386

# Theory of Computation

## 1.1 Natural Languages, Formal languages: Definitions and theorems

- **Gödel's incompleteness theorem:** Gödel's Incompleteness Theorems are two fundamental results in mathematical logic that state:

- Proved that for some axiomatic systems that there is no algorithm that will generate all true statements from those axioms.
- No such system can prove its own consistency.

This was the first indication that there are inherent limits on algorithms

- **Turing:** Alan Turing later provided formalism to the concepts of an "algorithm" and "computation", he invented definition for an abstract machine called the "universal algorithm machine", he provided means to formally (i.e., with mathematical rigor) explore the boundaries of what algorithms could, and could not, accomplish. Turing's model for a universal abstract machine was the basis for the first computer - in fact, Turing was involved in the construction of the first computer.
- **Natural languages:** We communicate via a *natural language*, Although we don't often think about it, our language is guided by rules; spelling, grammar, punctuation
- **Formal language:** Formal languages, which are not intended for human-to-human communication, are similar to natural languages in that they too have rules that define "correct" words and statements, but they are also different than natural languages in two key ways;
  - The rules that define a formal language are strictly enforced. There is no tolerance for misspellings, bad grammar, etc.
  - For the purpose of determining if a word or statement is acceptable in a formal language, meaning is ignored. Determining if something is (or is not) part of a language is determined by the language's defining rules which do not attach meaning (i.e., no definitions of words like in natural languages)

In short, formal languages is a game of symbols, not meaning

- **Formal Language terminology:**

- **Symbol:** it is an abstract entity that is not formally defined - like a point or a line in geometry - but think of it as a single character like a letter, numerical digit, punctuation mark, or emoticon
- **String (or Word):** A finite sequence (i.e., order matters) of zero or more symbols
- **Length:** The length of a string  $w$  is denoted by  $\text{length}(w)$  or  $|w|$  and is the number of symbols composing the string. Because strings, by definition, are finite then a string's lengths is always defined (sometimes zero).
- **Prefix, suffix:** Any number of leading/trailing symbols of the string.
- **Concatenation:** The concatenation of two strings  $w$  and  $x$  is formed by writing the first string  $w$  then the second string  $x$

**Note:** For any string  $w$ ,  $\Lambda w = w\Lambda = w$

- **Alphabet:** A finite set of symbols, typically denoted by the Greek capital letter sigma  $\Sigma$ , for example

$$\Sigma = \{a, b, c\} \quad \Sigma = \{0, 1\} \quad \Sigma = \emptyset \quad (\text{special case}).$$

- **The empty string:** A string with zero symbols is called the empty string and is denoted by the capital Greek letter lambda  $\Lambda$ , or sometimes lower case Greek letter epsilon  $\epsilon$ , where  $\Lambda$  and  $\epsilon$  are **not** symbols

Thus,

$$|\Lambda| = 0.$$

- **Formal language definition:** A formal language is a set of strings from some **one** alphabet. Given an alphabet we generally define a formal language over that alphabet by specifying rules that either;

1. Tell us how to test a candidate word, or
2. Tell us how to construct all words.

For example, Given  $\Sigma_1 = \{x\}$ , we can define languages

$$L_1 = \text{any non empty string} = \{x, xx, xxx, \dots\}$$

$$L_2 = \{X^n : x = 2k + 1, k \in \mathbb{Z}\} = \{x, xxx, xxxx, xxxxxx, \dots\} \quad L_3 = \{x, xxxxxxxx\}.$$

- **The empty language:** The empty language  $L = \emptyset$  is typically denoted with the capital greek letter phi  $\Phi$ . Thus,  $L = \emptyset = \Phi$

- **Notes on formal languages:**

- All languages are defined over some alphabet; cannot define a language without an alphabet.
- Some languages are finite, some languages are infinite (remember, alphabets are always finite).
- Some languages include the empty string  $\Lambda$ , some do not.
- Some languages are defined by rules, some are simply written completely (e.g.,  $\Sigma_1 = \{x\}$ ,  $L_3 = \{x, xxxxxxxx\}$ ).
- No matter what the alphabet  $\Sigma$  (even  $\Sigma = \emptyset$ ), you can always define at least two languages;  $L_1 = \{\Lambda\}$  and  $L_2 = \emptyset$ .

- **Closure of an alphabet (closure of  $\Sigma$ ) (Kleene closure):** The language defined by the set of all strings (including the empty string  $\Lambda$ ) over a fixed alphabet  $\Sigma$ .

- **Examples:**

$\Sigma = \{a\}$	$\Sigma^* = \{\Lambda, a, aa, aaa, aaaa, \dots\}$
$\Sigma = \{0, 1\}$	$\Sigma^* = \{\Lambda, 0, 1, 00, 01, 10, 11, 000, \dots\}$
$\Sigma = \emptyset$	$\Sigma^* = \{\Lambda\}$

**Note:** If  $\Sigma = \emptyset$  then  $\Sigma^*$  is finite and  $\Sigma^* = \{\Lambda\}$ , otherwise  $\Sigma^*$  is infinite.

- **Positive closure:**  $\Sigma^+ = \Sigma^* - \{\Lambda\}$ , you just take the empty string out of the kleene closure
- **Recall: Power set:** The power set of any set  $S$ , written  $\mathcal{P}(S)$  is the set of all subsets of  $S$ , including the empty set and the set  $S$  itself.

In other words, given a set  $S$ , then its power set  $\mathcal{P}(S)$  is a set of sets

- **Note:**

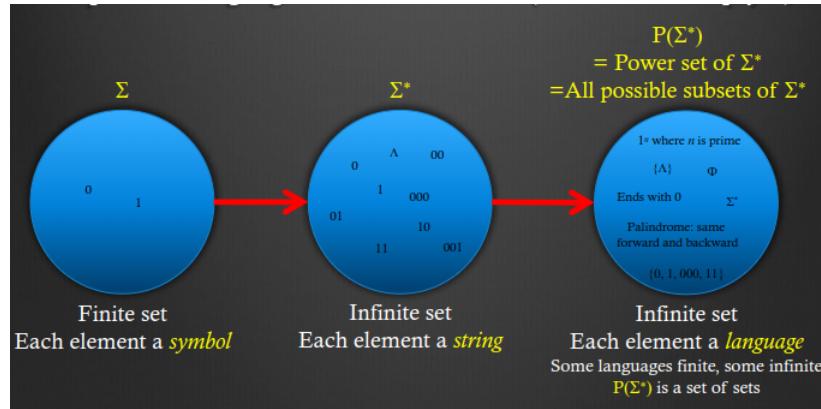
- \* If  $S = \emptyset$ , then  $\mathcal{P}(S) = \mathcal{P}(\emptyset) = \{\emptyset\} = \{\emptyset\}$  = a set with one element =  $\emptyset$ .
- \* If  $S$  is non-empty and finite with  $n$  elements, then  $\mathcal{P}(S)$  will be finite with  $2^n$  elements.
- \* If  $S$  is infinite, then  $\mathcal{P}(S)$  will be infinite.

- **Example:**

If  $S = \{x, y, z\}$ , then  $\mathcal{P}(S)$  will have the following  $2^3 = 8$  elements (each a set):

$$\mathcal{P}(S) = \{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$$

- **Power set of the kleene closure  $\mathcal{P}(\Sigma^*)$ :** Given some alphabet  $\Sigma$  we can construct the set of all possible languages from  $\Sigma$  as follows (assume non-empty  $\Sigma$ ):



- **From formal languages to computers:**

- Given an alphabet  $\Sigma$  we can define many formal languages - the range of which is captured by  $\mathcal{P}(\Sigma^*)$ .
- We can define many formal languages verbally, but is there a way to define/express every language in any  $\mathcal{P}(\Sigma^*)$  with some formal system or abstract machine?
- We search for a formal system or abstract machine with enough "power" to define any language in any  $\mathcal{P}(\Sigma^*)$ .
- **KEY POINT**  
The abstract machines we discover along our search to cover  $\mathcal{P}(\Sigma^*)$  turn out to be *the theoretical basis for all computing*.
- In other words, by understanding the power (and limitations) of abstract machines that cover  $\mathcal{P}(\Sigma^*)$ , we are simultaneously discovering the same limits about all computing.

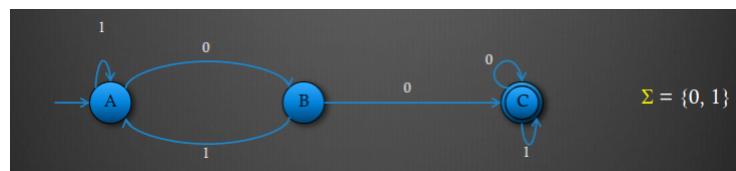
## 1.2 Regular languages

**Preface.** The first few subsubsections will be in the world of regular languages. In the context of computation theory, regular languages are a class of formal languages that can be recognized by finite automata. These languages are important because they are the simplest class of languages that can be described by a computational model. The characteristics of regular languages are as follows,

- **Finite Automata:** Regular languages can be recognized by deterministic or non-deterministic finite automata (DFA or NFA).
- **Regular Expressions:** Regular languages can be described using regular expressions.
- **Closure Properties:** Regular languages are closed under several operations, including:
  - **Union:** The union of two regular languages is also regular.
  - **Concatenation:** The concatenation of two regular languages is also regular.
  - **Kleene Star:** The Kleene star operation, which involves repeating a regular language any number of times (including zero), results in a regular language.
  - **Intersection and Difference:** Regular languages are also closed under intersection and difference.
- **Decision Problems:** Certain decision problems are decidable for regular languages. For example, it is possible to determine whether a given string belongs to a regular language (membership problem), whether two regular languages are equivalent, or whether a regular language is empty.

### 1.2.1 Finite Automata

- **Informal definition:** Described informally, a finite automaton (FA) is always associated with some alphabet  $\Sigma$  and is an abstract machine which has
  1. A non-empty finite number of states, exactly one of which is designated as the "start state" and some number (possibly zero) of which are designated as "accepting states".
  2. A transition table that shows how to move from one state to another based on symbols in the alphabet  $\Sigma$
- **A simple example of a FA:**



- Defined over alphabet  $\Sigma = \{0, 1\}$ .
- States are circles; transitions are directed edges (i.e., arrows) between states.
- Has exactly three states; **A**, **B**, and **C**.
- Every FA must have exactly one start state. In this example, the start state is **A** and denoted as the only state that has an edge coming to it from no other state.

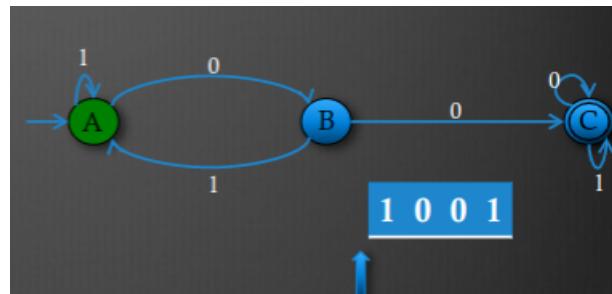
- There is only one accepting state, **C**, and it is denoted by its *double circle*. (We could have more than one but in this case we only have one)
- **Very important:**
  - \* Each symbol in the alphabet has exactly one associated edge leaving every state.
  - \* In other words, every state must have exactly one edge leaving it for each symbol in the alphabet.
- **How to use an FA:** The purpose of a FA is to define a language over its alphabet  $\Sigma$ . The FA provides the means by which to test a candidate string from  $\Sigma$  and determine whether or not the string is in the language. It does this by "writing" the candidate string on an fictitious input tape and proceeding as follows:

1. Set the FA to the start state.
2. If end-of-string then halt.
3. Read next symbol on tape.
4. Update the state according to the current state and the last symbol read.
5. Goto step 2.

When the process halts check which state the FA is in. If it is in any accepting state, then the string is in the language defined by the FA, otherwise the string is not in the language

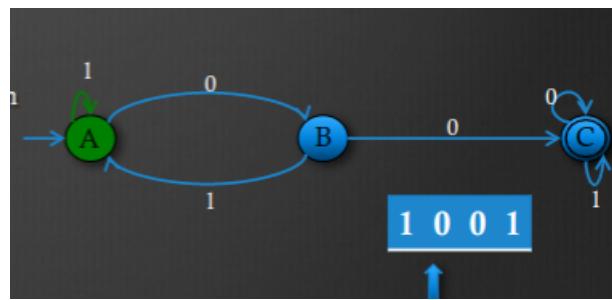
- **Using the previous FA:** Let's now try to use our FA to test whether or not the string 1001 is in the language

We start by writing the string on an input tape, placing the read head at the beginning of the tape, and placing the FA in its initial state, **A**



Since the tape head is not at the end of the tape we

1. Read the next symbol from the tape.
2. Follow the edge from the state we are currently in that corresponds to the symbol we just read to transition to the next state.
3. Move the tape head

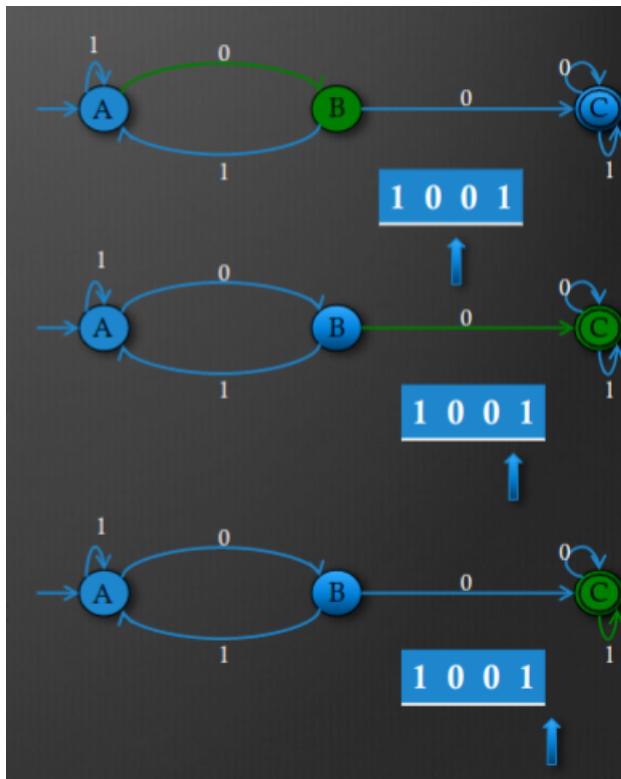


In this case, we started in state  $A$ , read symbol 1, and followed the edge labeled 1 from  $A$  which brought us back to  $A$

We proceed in this way, read, change state, move tape head until we reach the end of the tape

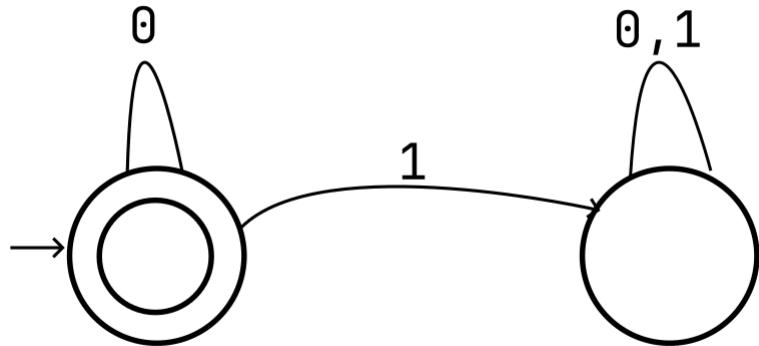
Once the tape head reaches the end of the tape we simply look to see whether or not the FA ended in an accepting state.

In this case it ended in state  $C$ , which is an accepting state, which means that string 1001 is in the language.



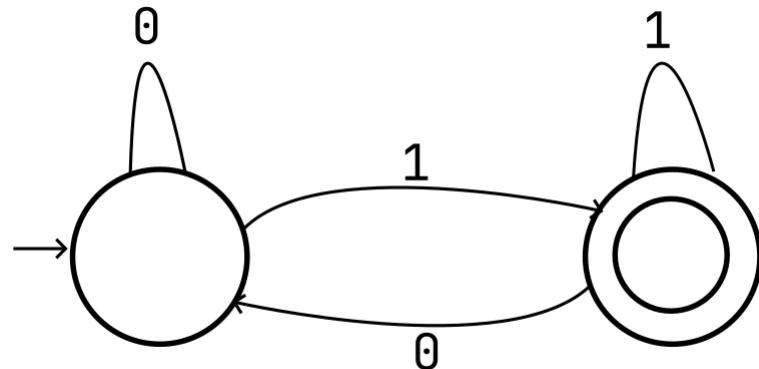
We deduce that the language has only strings with two consecutive zeroes somewhere.

- FA Example Two: The set of all strings that do not contain a one ( $\Sigma = \{0, 1\}$ ):

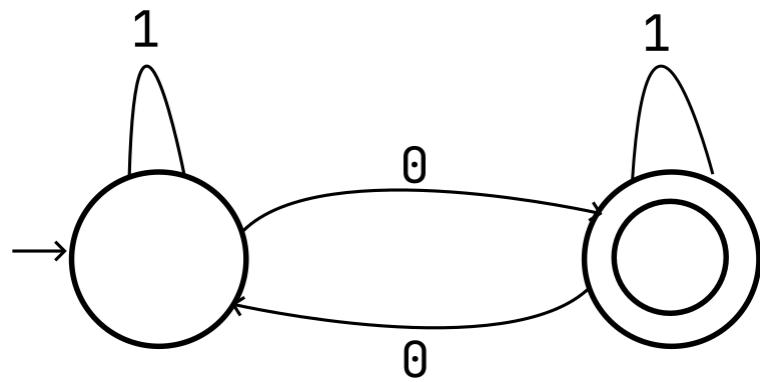


This one is pretty simple. If we have a zero, stay in the accepting state, if we see a one, toss it to the other non-accepting state, its not coming back.

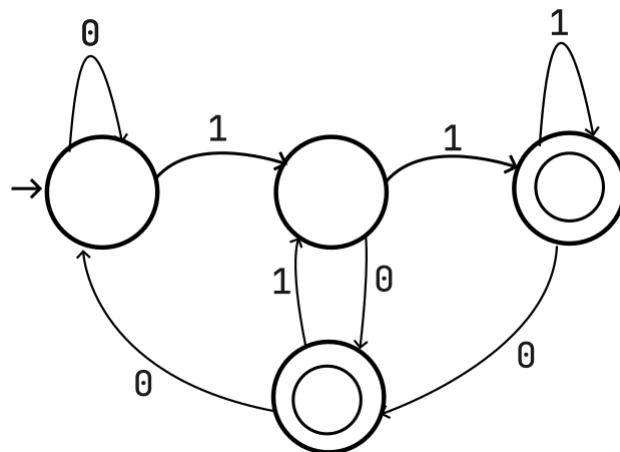
- FA Example Three: The set of all strings that end in one ( $\Sigma = \{0, 1\}$ ):



- FA Example Four: The set of all strings with an odd number of zeros ( $\Sigma = \{0, 1\}$ ):



- **FA Example Five:** The set of all strings where the second to last symbol is one ( $\Sigma = \{0, 1\}$ ):



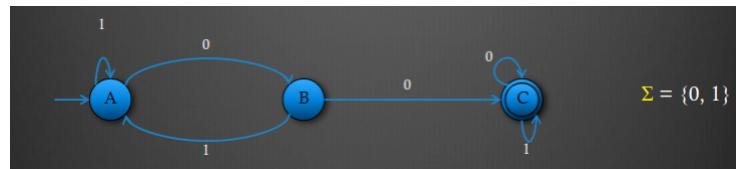
- **States are "memory":** Consider the four FA we just created, in each instance the solution required us to design an FA that remembered at least part of what it had already read from the input tape. The type of memory that an FA has is very different than the RAM we find in contemporary computers, but the FA does have memory. Each time the FA enters a different state it is, in effect, redefining the memory of the entire FA. The FA can only be in a finite number of states, and that number can be arbitrarily large, but (as we will see) that difference in memory has a profound limiting effect in what FAs can compute.
- **Limits of a FA:**

**Limited Memory:**

- **Finite State:** A finite automaton has a finite number of states. This means it can only "remember" a limited amount of information about the input it has processed. Once a finite automaton transitions to a new state, it forgets all previous information except for the current state.
- **No Stack or Tape:** Unlike more powerful models such as pushdown automata (which have a stack) or Turing machines (which have an infinite tape), finite automata cannot use any form of auxiliary memory to keep track of an unbounded number of items or to perform operations that require more complex memory management.

### Inability to Count Unboundedly:

- **No Arbitrary Counting:** Finite automata cannot count occurrences of symbols beyond the number of states they have. For example, a DFA with  $n$  states can only count up to  $n - 1$  occurrences of a symbol reliably. Thus, they cannot recognize languages that require matching counts of different symbols if those counts are unbounded, such as  $\{a^n b^n \mid n \geq 1\}$ , where the number of 'a's must match the number of 'b's.
- **FA Formal Definition:** We formally denote a *finite automaton* by a 5-tuple  $(Q, \Sigma, q_0, T, \delta)$ , where
  - $Q$  is a finite set of *states*.
  - $\Sigma$  is an alphabet of *input symbols*.
  - $q_0 \in Q$ , is the *start state*.
  - $T \subseteq Q$ , is the set of *accepting states*.
  - $\delta$  is the *transition function* that maps a state in  $Q$  and a symbol in  $\Sigma$  to some state in  $Q$ . In mathematical notation, we say that  $\delta : Q \times \Sigma \rightarrow Q$ . With:
    - \*  $Q \times \Sigma$ : The Cartesian product of the set of states  $Q$  and the alphabet  $\Sigma$ . This represents all possible pairs of a state and an input symbol.
    - \*  $\rightarrow Q$ : Indicates that the transition function maps each pair  $(q, \sigma)$  (where  $q \in Q$  and  $\sigma \in \Sigma$ ) to a single state in  $Q$ .
- **Formally Specifying Our First FA:**



Recall our first FA that accepts any string with two consecutive zeros somewhere.

We drew it as a Finite State diagram, but to formally define this FA we must specify each of the elements from the 5-tuple  $(Q, \Sigma, q_0, T, \delta)$ .

- $Q$  is a finite set of *states*:  $Q = \{A, B, C\}$
- $\Sigma$  is an alphabet of *input symbols*:  $\Sigma = \{0, 1\}$
- $q_0 \in Q$ , is the *start state*:  $q_0 = A$
- $T \subseteq Q$ , is the set of *accepting states*:  $T = \{C\}$

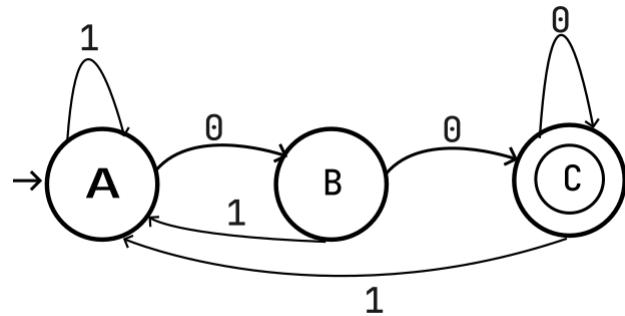
–  $\delta$  is the *transition function*  $\delta : Q \times \Sigma \rightarrow Q$

$\delta$	0	1
A	B	C
B	C	A
C	C	C

- **Unary:** consisting of or involving a single component or element.
- **Unary language:** One where the alphabet has only one symbol.
- **Binary:** Relating to, composed of, or involving two things.
- **Ternary:** Composed of three parts.
- **Dead state (trap state):** This is a state that once entered, can never be left.
- **Deterministic finite automaton (DFA):** The FA's we have looked at thus far have been DFA's. A DFA is a finite automaton where, for each state and each input symbol, there is exactly one transition to a new state. This means that given a current state and an input symbol, the next state is uniquely determined. In the future we will look at nondeterministic finite automaton (NFA). An NFA is a finite automaton where, for each state and input symbol, there can be multiple possible transitions to different states. Additionally, an NFA can have transitions that do not consume any input symbol ( $\epsilon$ -transitions).

### 1.2.2 Finite Automata: More examples

- $\Sigma = \{0, 1\}$ , all strings that start with 00
- $\Sigma = \{0, 1\}$ , all strings that end with 00



With:

- $Q = \{A, B, C\}$
- $\Sigma = \{0, 1\}$
- $q_0 = A$
- $T = C$
- $\delta : Q \times \Sigma \rightarrow Q$  defined by

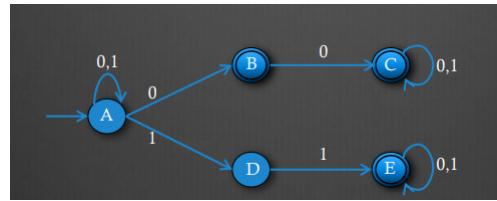
$\delta$	0	1
A	B	A
B	C	A
C	C	A

### 1.2.3 nondeterministic Finite automata (NFA)

- **NFA definition:**

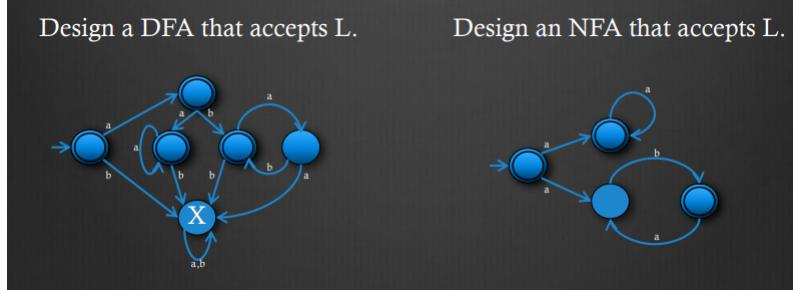
- If an automaton gets to a state where there is more than one possible transition corresponding to the symbol read from the tape, the automaton may choose any of those paths. (nondeterminism) We say it **branches**
- if an automaton gets to a state where there is no transition for the symbol read from the tape, then that path of the automaton halts and rejects the string. We say it **dies**
- the automaton accepts the input string if and only if there exists a choice of transitions that ends in an accept state.

**Example:** Consider this nondeterministic FA (NFA) over  $\Sigma = \{0, 1\}$



- **DFA or NFA?:** Consider the language  $L$  over  $\Sigma = \{a, b\}$  which is defined by

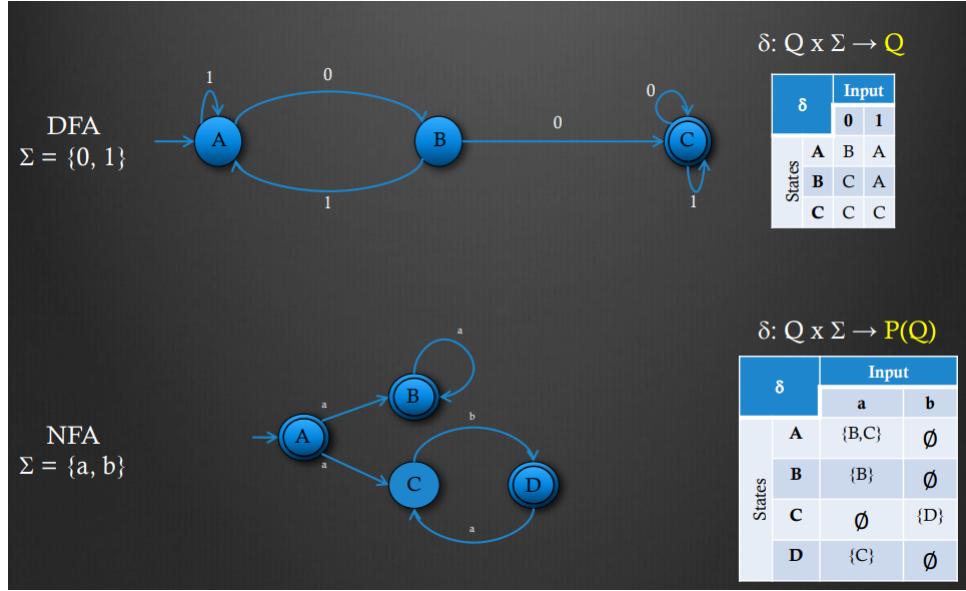
$$L = (a^*) + (ab)^*.$$



- **NFA Formal definition:** We define an NFA  $M(Q, \Sigma, q_0, T, \delta)$

- $Q$  is a finite set of states
- $\Sigma$  is an alphabet of input symbols
- $q_0 \in Q$  is the start state
- $T \subseteq Q$  is the set of accepting states
- $\delta$  is the transition function  $\delta : Q \times \Sigma \rightarrow P(Q)$

- **Transition function, DFA vs NFA:**

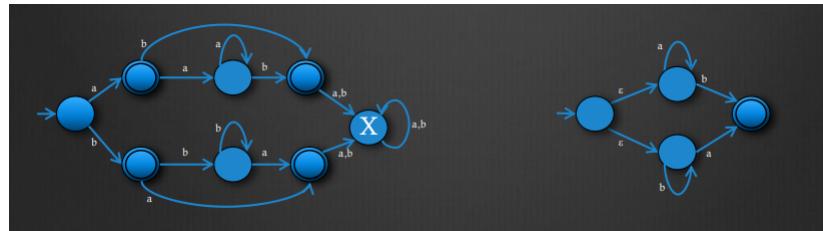


- **NFA with  $\epsilon$ -transitions:**  $\epsilon$ -transitions allow the automaton to change state without consuming an input symbol

Changing states without consuming input symbols can go on arbitrarily long as there are  $\epsilon$ -transitions to traverse.

- **DFA or NFA with  $\epsilon$ -moves?:** Consider the language  $L$  over  $\Sigma = \{a, b\}$  which is

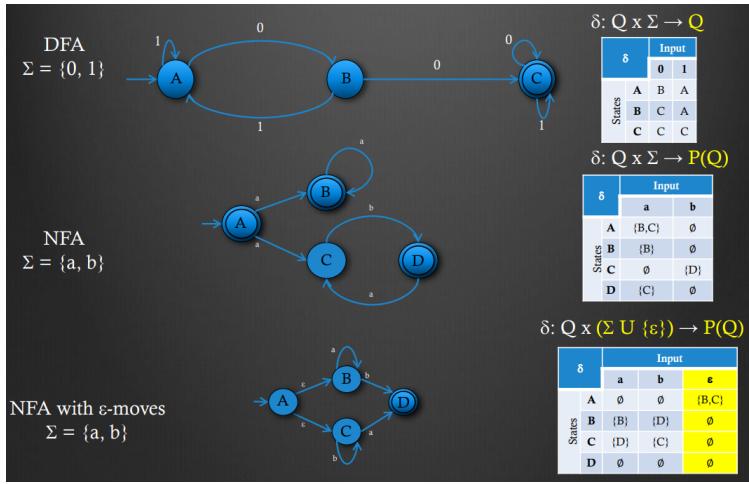
$$L = (b^*a) + (a^*b).$$



- **NFA with  $\epsilon$ -transitions formal definition:** Everything is the same except for the transition function, we now have

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q).$$

- $\delta$  - DFA, NFA, and NFA with  $\epsilon$ -moves:



- **DFA, NFA, or NFA with  $\epsilon$  moves, who can define the most languages?**: We begin by noting, by definition, every DFA is an NFA. This means that any language you can define with a DFA can also be defined by an NFA. Thus,

$$\text{Languages defined by DFA} \subseteq \text{Languages defined by NFA}.$$

Also, by definition, every DFA is an NFA with  $\epsilon$ -moves, an NFA is an NFA with  $\epsilon$  moves, even if it doesn't have any. Thus,

$$\text{Languages defined by DFA} \subseteq \text{Languages defined by NFA with } \epsilon\text{-moves}.$$

But, by definition, every NFA is an NFA with  $\epsilon$ -moves. Thus,

$$\text{Languages defined by NFA} \subseteq \text{Languages defined by NFA with } \epsilon\text{-moves}.$$

This tells us that

- NFAs are at least as powerful in defining languages as DFAs
- NFAs with  $\epsilon$ -moves are at least as powerful in defining languages as DFAs and NFAs.

It turns out that these three are **equally** as powerful. We assert

$$\begin{aligned} & \text{Languages defined by DFA's} \\ &= \text{Languages defined by NFA's} \\ &= \text{Languages defined by NFA's with } \epsilon\text{-moves}. \end{aligned}$$

We prove this by showing an algorithm that converts any NFA with  $\epsilon$ -moves (or any NFA) to a DFA that accepts the exact same language

This means that there does not exist a language that can be defined by an NFA with  $\epsilon$ -moves (or NFA) that cannot also be defined by a DFA.

- **$\epsilon$ -closure**: Before we can look at the algorithm we must first define the  $\epsilon$ -closure of a set of states

Given:

- an NFA with  $\epsilon$ -moves  $M(Q, \Sigma, q_0, T, \delta)$
- Some set of states  $S \subseteq Q$

We define the  $\epsilon$ -closure( $S$ ) as the set of states that are reachable from the set of states  $S$  using only zero or more  $\epsilon$ -moves in  $\delta$ .

Note: it is always the case that  $S \subseteq \epsilon\text{-closure}(S)$

The formal definition is

$$\epsilon\text{-closure}(q) = \{q\} \cup \{p : q \xrightarrow{\epsilon} p\}.$$

- **$\epsilon$ -closure alternate notation.**

$$\epsilon\text{-closure}(\{A\}) = \epsilon(\{A\}) = E(\{A\}).$$

- **$\epsilon$ -closure of the empty set  $\emptyset$ :** The epsilon closure of the empty set is  $\epsilon(\emptyset) = \emptyset$
- **Algorithm: Converting NFA with  $\epsilon$ -moves to DFA:** The algorithm constructs a new DFA  $M'(Q', \Sigma, q'_0, T', \delta')$  From an NFA with  $\epsilon$ -moves  $M(Q, \Sigma, q_0, T, \delta)$ .  $\Sigma$  will remain the same

Things to note about the conversion:

- Same alphabet  $\Sigma$
- Lose column  $\epsilon$
- Lose all nondeterminism
- Lose all empty sets
- Cell values change from sets of states to states

**Example:** Consider the following NFA with  $\epsilon$ -moves  $M(Q, \Sigma, q_0, T, \delta)$  over  $\Sigma = \{0, 1\}$  and its associated transition table  $\delta: Q \times (\Sigma \cup \{\epsilon\}) \rightarrow P(Q)$

	0	1	$\epsilon$
X	{Y}	{Y}	$\emptyset$
Y	{X, Z}	{Z}	{Z}
Z	$\emptyset$	{Y}	$\emptyset$

Start by computing the  $\epsilon$ -closure of the start state in  $\delta$ .

$\delta$		Input			$\delta'$	
States	$\delta$	0	1	$\epsilon$	0	1
		X {Y}	{Y}	$\emptyset$	$\Rightarrow$ x	
Y {X, Z}				{Z}		
Z $\emptyset$		{Y}	$\emptyset$			

$\epsilon\text{-closure } \{X\} = \{X\}$

This becomes the start state for  $\delta'$ .

There is a subtle - but very important - point to be made here ...

we cannot simply take the  $\epsilon$ -closure (a set) and use it to create a row in  $\delta'$  (which needs to be a state). What we do is create a label for the new state in  $\delta'$  that represents the set of states from  $\delta$  and then add that new state to  $\delta'$

In this instance we represented the set of states  $\{X\}$  by a single state whose label is  $X'$

We continue by filling the columns of the start state for each symbol  $\Sigma = \{0, 1\}$

Processing  $\delta'$  state  $X'$  which represents the set of states  $\{X\}$  in  $M$ :

- Processing input symbol 0 (process each state in  $\{X\}$  using  $\delta$ ):

\* Process  $X$

$$\delta(X, 0) = \{Y\}$$

$$\epsilon\text{-closure}(\{Y\}) = \{Y, Z\}$$

Since there are no more states in  $\{X\}$  to process, we have finished processing the symbol 0 and have produced the set of states  $\{Y, Z\}$ .

We create a new state with label  $Y'Z'$  (or  $Z'Y'$ , order does not matter) for  $\delta'$  that represents  $\{Y, Z\}$  in  $M$  and define:

$$\delta'(X', 0) = Y'Z'$$

We note that  $Y'Z'$  is a new state in  $\delta'$  and so we create a new row for it in  $\delta'$ .

We continue this until we reach

$\delta'$	Input	
	0	1
$X'$	$Y'Z'$	$Y'Z'$
$Y'Z'$		

Processing  $\delta'$  state  $Y'Z'$  which represents the set of states  $\{Y, Z\}$  in  $M$ :

- Processing 0:

\* Process  $Y$

$$\delta(Y, 0) = \{X, Z\}, \quad \epsilon\text{-closure}(\{X, Z\}) = \{X, Z\}$$

\* Process  $Z$

$$\delta(Z, 0) = \emptyset, \quad \epsilon\text{-closure}(\emptyset) = \emptyset$$

Here is our first instance of processing a state and symbol where the state in  $\delta'$  represents multiple states in NFA  $M$ . When this happens, the set of states in NFA  $M$  is computed by *taking the union of the  $\epsilon$ -closures*:  $\{X, Z\} \cup \emptyset = \{X, Z\}$ .

This produces a new label  $X'Z'$  which we use to define:

$$\delta'(X'Y', 0) = X'Z'$$

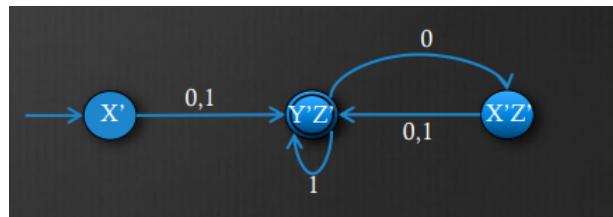
and since  $X'Z'$  is a new state, we add it to  $\delta'$ .

We continue this until we reach

$\delta'$	Input	
	0	1
X'	$Y'Z'$	$Y'Z'$
$Y'Z'$	$X'Z'$	$Y'Z'$
$X'Z'$	$Y'Z'$	$Y'Z'$

A state in  $M'$  is an accepting state iff at least one of the states that it represents in  $M$  is an accepting state ... in this case  $T' = \{Y'Z'\}$ .

We can now draw the new DFA



**Note:** If the closure or union of closures is the empty set, we do this

$\delta'$	Input	
	0	1
A'	$B'C'$	<i>empty</i>
$B'C'$		
<i>empty</i>	<i>empty</i>	<i>empty</i>

This "empty" is a state and represents a garbage state, what goes does not leave.

- **Kleene's theorem revisited:** The following are equivalent for a language  $L$

1. There is a DFA for  $L$
2. There is an NFA for  $L$
3. There is an RE for  $L$

- **Union of two DFA's (cartesian product construction):** The process of finding the union of two deterministic finite automata (DFAs) involves creating a new DFA that accepts the union of the languages accepted by the original DFAs. This is done using a product construction (also called the Cartesian product construction), where you combine the states of both DFAs in a systematic way to ensure the resulting DFA accepts strings from either of the original DFAs.

Let's say we have two DFAs:

$$D_1 = (Q_1, \Sigma, \delta_1, q_1^{\text{start}}, F_1)$$

that recognizes language  $L_1$ .

$$D_2 = (Q_2, \Sigma, \delta_2, q_2^{\text{start}}, F_2)$$

that recognizes language  $L_2$ .

#### Create a New DFA State Set:

- The states of the new DFA are pairs of states, one from each of the original DFAs. The new state set will be the Cartesian product  $Q_1 \times Q_2$ , meaning every possible combination of a state from  $D_1$  and a state from  $D_2$ .
- If  $D_1$  has  $n$  states and  $D_2$  has  $m$  states, the new DFA will have  $n \times m$  states.

#### Define the New Start State:

- The new start state is  $(q_1^{\text{start}}, q_2^{\text{start}})$ , where  $q_1^{\text{start}}$  is the start state of  $D_1$  and  $q_2^{\text{start}}$  is the start state of  $D_2$ .

#### Define the New Transition Function:

- The transition function  $\delta$  for the new DFA operates by taking an input symbol and applying the transition functions of both original DFAs in parallel.
- For each input symbol  $a \in \Sigma$ , the new DFA transitions from state  $(q_1, q_2)$  to state  $(\delta_1(q_1, a), \delta_2(q_2, a))$ .
- In other words, if  $q_1$  moves to  $q'_1$  on input  $a$  in  $D_1$ , and  $q_2$  moves to  $q'_2$  on input  $a$  in  $D_2$ , the new DFA will move from  $(q_1, q_2)$  to  $(q'_1, q'_2)$ .

**Define the New Set of Accepting (Final) States:** The new DFA will accept a string if either of the original DFAs would accept it. Therefore, the set of final states  $F$  in the new DFA is defined as:

$$F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ or } q_2 \in F_2\}$$

This means that if either  $q_1$  is a final state in  $D_1$ , or  $q_2$  is a final state in  $D_2$ , the pair  $(q_1, q_2)$  is a final state in the new DFA.

**Note:** It is possible in the new DFA (constructed as the union of two DFAs) to have states that are unreachable-meaning there are states in the DFA that cannot be reached from the start state. This typically happens because, in the product construction, we generate all possible pairs of states from the two original DFAs, but not all of these pairs are necessarily reachable.

The union of two finite automata (FAs) is useful for constructing a new automaton that recognizes any string accepted by either of the two original automata. This has several practical applications in theoretical computer science and programming:

- **Finding the intersection of two DFA's:** The process is basically the same as finding the union, but it differs in how we define the accepting states in the new machine, the accepting states will be

$$T = \{(q_1, q_2) : q_1 \in T_1 \text{ and } q_2 \in T_2\}.$$

**Note:** The intersection of two DFAs is useful in various practical applications where you need to accept only the strings that satisfy the conditions or rules of both automata

- **Concatenation of two DFA's:** The process is simple

For two machines  $M_1(Q_1, \Sigma, q_{01}, T_1, \delta_1)$ , and  $M_2(Q_2, \Sigma, q_{02}, T_2, \delta_2)$

1. Connect the final states of the first machine to the start state of the second machine (With  $\epsilon$ -transitions)
2. Clear  $T_1$ , There are no more final states in the first machine
3. Convert  $\epsilon$ -NFA to DFA

**Note:** The concatenation of two DFAs has practical uses in many scenarios where the language of interest is the concatenation of two sublanguages. Concatenating two DFAs allows you to recognize strings that can be divided into two parts, where the first part is recognized by one DFA and the second part is recognized by the other.

- **Finding the union of two NFA's:** taking the union of two nondeterministic finite automata (NFAs) involves constructing a new NFA that accepts any string that is accepted by either of the original NFAs. This process can be done by creating a new NFA that combines the two original NFAs.

Given  $M_1(Q_1, \Sigma, q_{01}, T_1, \delta_1)$ , and  $M_2(Q_2, \Sigma, q_{02}, T_2, \delta_2)$

1. **New start state:** Start by defining a new start state  $q'_0$ , this state will have  $\epsilon$  transitions to the start states of both machines.
2. **Define  $Q'$ , the new set of states:** The new set of states will be the set of all states in  $M_1$ , and it will include all the states in  $M_2$ , along with the new start state. Thus,

$$Q' = Q_1 \cup Q_2 \cup \{q'_0\}.$$

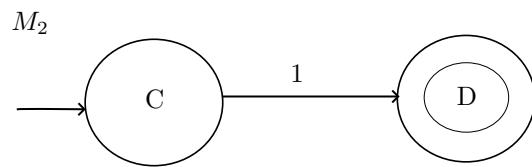
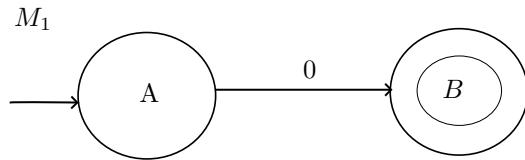
3. **Define the transition function:** The transition function  $\delta'$  of the new NFA will include:
  - All the transitions of  $M_1$  and  $M_2$
  - Two  $\epsilon$  transitions from the new start state to the start states of the two original machines  $q_{01}$  and  $q_{02}$ . Thus,

$$\delta'(q'_0, \epsilon) = \{q_{01}, q_{02}\}.$$

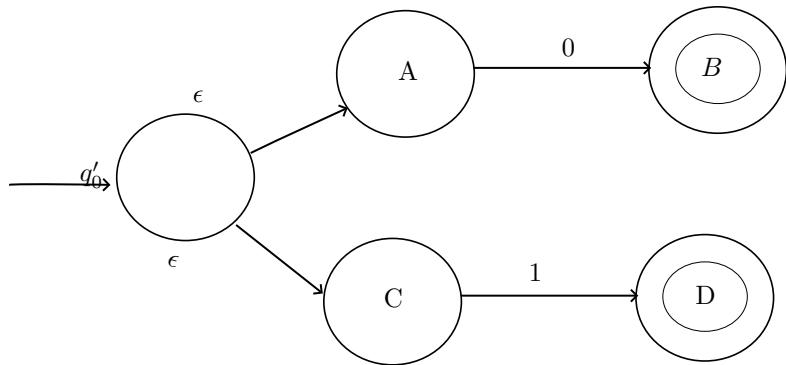
4. **Define the set of accepting states:** The set of accepting states will be

$$T' = T_1 \cup T_2.$$

**Example:**



$M_1 \cup M_2$  is then



- **Finding the intersection of two NFA's:** For NFAs, intersection is more complex because NFAs are nondeterministic and don't handle intersection naturally. Typically, you convert the NFAs to DFAs and then apply the DFA product construction
- **Concatenation of two NFA's:** The process is the same as with two DFA's (see above), but you don't need to convert to a DFA at the end.

- **Properties of union, intersect, and concatenation for two FA's:** the properties of union, intersection, and concatenation for finite automata (FAs) are directly tied to the properties of regular languages.

### Union of two FA

- **Closure:** The class of regular languages (those recognized by FA) is closed under union. This means the union of two regular languages is also regular, and there exists an FA that recognizes the union of the languages.
- **Commutative:** Union is commutative for FA, meaning the order of combining automata does not matter.
- **Associative:** Union is associative, so it doesn't matter how automata are grouped when performing multiple unions.
- **Distributive over Intersection** Union distributes over intersection for regular languages, just as with sets.

### Intersection of two FA

- **Closure:** The class of regular languages is also closed under intersection, meaning there is always an FA (typically constructed as a DFA) that recognizes the intersection of two regular languages.
- **Commutative:** Intersection is commutative, meaning the order of combining automata doesn't matter.
- **Associative:** Intersection is associative, so the grouping doesn't matter.
- **Distributive over Union:** Intersection distributes over union for regular languages, just as with sets.

### Concatenation

- **Closure:** Regular languages are closed under concatenation.
- **Associativity:** Concatenation is associative. This means that the way in which you group the automata when performing concatenation doesn't matter.
- **Identity Element:** The identity element for concatenation is the language that contains only the empty string,

$$L(A) \cdot \{\epsilon\} = L(A).$$

- **Distributivity Over Union:** Concatenation distributes over union. This means:

$$L(A) \cdot (L(B) \cup L(C)) = L(A) \cdot L(B) \cup L(A) \cdot L(C).$$

- **Concatenation with the Empty Set:** Concatenating any language with the empty set results in the empty set. This is because there are no strings to concatenate if one of the languages is empty:

### Not commutative

#### 1.2.4 Regular expressions

- **RE:** A RE corresponds to a set of strings; that is, a RE describes a language
- **RE three operations:**
  1. Union (+)
  2. concatenation (xy)
  3. star (zero or more copies)
- **RE special symbols**

+ \* ( ).

- **Grouping:** The parenthesis are used for grouping,
- **Union:** the plus sign means **union**. Thus, writing

0 + 1.

Means zero or one, we refer to + as "or"

- **Concatenation:** We concatenate simply by writing one expression after the other, with no spaces

(0 + 1)0.

Is the pair of strings 00 and 10

- **Empty string:** We can also use the empty string  $\epsilon$

(0 + 1)(0 +  $\epsilon$ ).

corresponds to 00, 0, 10, and 1

- **Zero or more copies (star):** Using the start indicates zero or more copies, thus

$a^*$ .

corresponds to any string of a's:  $\{\epsilon, a, aa, aaa, \dots\}$

- **More on union:** If you form an RE by the or of two REs, call them  $R$  and  $S$ , then the resulting language is the union of the languages of  $R$  and  $S$ .

Suppose  $R = (0 + 1) = \{0, 1\}$ , and  $S = \{01(0 + 1)\} = \{010, 011\}$ , then  $R + S = (0 + 1) + (01(0 + 1)) = \{0, 1, 010, 011\}$

- **More on concatenation:** If you form an RE by the or of two REs, call them  $R$  and  $S$ , then the resulting language consists of all strings that can be formed by taking one string from the language of  $R$  and one string from the language of  $S$  and concatenating them.

Suppose  $R = (0 + 1) = \{0, 1\}$ , and  $S = \{01(0 + 1)\} = \{010, 011\}$ , then  $RS = (0 + 1)01(0 + 1) = \{0010, 0011, 1010, 1011\}$

- **More on star:** If you form an RE by taking the star of an RE  $R$ , then the resulting language consists of all strings that can be formed by taking any number of strings from the language of  $R$  (they need not be the same and they need not be different), and concatenating them.

Suppose  $R = 01(0+1) = \{010, 011\}$ , then  $R^* = 01(0+1)*\{010, 010010, \dots, 011, 011011, \dots, 010011, \dots\}$

- **Precedence of the operations**

1. Star (\*)
2. Concatenation
3. Union (+)

- **Recursive definition of the kleene star (closure) ( $L^*$ ):**

1.  $\epsilon \in L^*$
2. If  $x \in L^*$  and  $y \in L$ , then  $xy \in L^*$

**Base case:** The first rule provides a starting point by ensuring that the empty string  $\epsilon$  is in  $L^*$ .

**Recursive step:** The second rule allows you to take any string  $x$  already in  $L^*$  and concatenate it with a string  $y \in L$  to produce a new string  $xy \in L^*$ .

After using the second rule once to generate a new string  $xy \in L^*$ , you can apply the rule again by concatenating this new string with another string from  $L$ . This recursive process can continue indefinitely, generating all possible strings that can be formed by concatenating zero or more strings from  $L$ .

- **Recursive definition of the kleene star (other)**

1.  $L^0 = \{\epsilon\}$  (Start with the empty string, always in the closure)
2.  $L^i = LL^{i-1}$  for  $i > 0$  (Start recursively building strings)
3.  $L^* = \bigcup_{i=0}^{\infty} L^i$  (the whole thing)

**Note:** We also define the positive closure of  $L$ , denoted  $L^+$ , as  $L^* - \{\epsilon\}$  or

$$L^+ = \bigcup_{i=1}^{\infty} L^i.$$

- **Closure of the empty language:**  $\Phi^* = \{\epsilon\}$
- **Regular expression for the empty language:**  $\Phi = \emptyset$  is the regular expression for the empty language (empty set)
- **More on language composition operators:** The language composition operators were defined over any language and, in turn, generate new languages. As such, composition operators take any one or two languages from  $P(\Sigma^*)$  and can produce any language in  $P(\Sigma^*)$ .
- **Regular languages (regular sets), regular expression limits:** Although regular expressions are based on language composition operators, their recursive definition (i.e., only regular expressions, therefore only languages defined by regular expressions) limits the languages that they can define.

**Note:** Regular expressions cannot produce all languages in  $P(\Sigma^*)$ .

In fact, the set of languages that regular expressions can define have a special name - they are called regular languages (or sometimes regular sets).

- **Kleene's theorem:** There is an FA for a language if and only if there is an RE for the language
- **Regular expressions order of operations:** From highest to lowest precedence
  1. Parenthesis
  2. Kleene star
  3. Concatenation
  4. Union (+ or |)
- **Properties of regular expressions:**

**Note:** Intersection is a operation not defined for regular expressions

### Union

- **Commutative:**

$$R_1 \cup R_2 = R_2 \cup R_1$$

- **Associative:**

$$(R_1 \cup R_2) \cup R_3 = R_1 \cup (R_2 \cup R_3)$$

- **Identity Element:**

$$R_1 \cup \emptyset = R_1$$

- **Idempotent:**

$$R_1 \cup R_1 = R_1$$

### 2. Concatenation ( $\cdot$ )

- **Non-commutative:**

$$R_1 \cdot R_2 \neq R_2 \cdot R_1$$

- **Associative:**

$$(R_1 \cdot R_2) \cdot R_3 = R_1 \cdot (R_2 \cdot R_3)$$

- **Identity Element:**

$$R_1 \cdot \epsilon = \epsilon \cdot R_1 = R_1$$

- **Concatenation with  $\emptyset$ :**

$$R_1 \cdot \emptyset = \emptyset \cdot R_1 = \emptyset$$

### Kleene Star (\*)

- **Kleene Star of  $\epsilon$ :**

$$\epsilon^* = \{\epsilon\}$$

- **Kleene Star of  $\emptyset$ :**

$$\emptyset^* = \{\epsilon\}$$

- **Idempotent:**

$$(R^*)^* = R^*$$

## Distributive Properties

- Union over Concatenation:

$$R_1 \cdot (R_2 \cup R_3) = (R_1 \cdot R_2) \cup (R_1 \cdot R_3)$$

- Concatenation over Union:

$$(R_1 \cup R_2) \cdot R_3 = (R_1 \cdot R_3) \cup (R_2 \cdot R_3)$$

- **Language of a RE notation:**  $L(RE)$  is the language defined by the regular expression  $RE$ , If we have an RE  $R$ , then the language  $L(R)$  is the language defined by the RE  $R$
- **When a regular expression is the empty set  $\emptyset$ :** When a regular expression (RE) represents the empty set it means that the RE matches no strings at all, not even the empty string.

The language is then

$$L(\emptyset) = \Phi.$$

Where  $\Phi$  denotes the empty language

- **One or more occurrences  $RR^*$ :** We denote this by plus instead of star, ie  $RR^* = R^+$ , but you also must redefine union as  $|$  instead of  $+$
- **Simplifying regular expressions (Some can also be found above in properties):**
  - Concatenation of stars:  $(R^*)^* = R^*$
  - Concatenation of Repeated Expressions:  $R^*R^* = R^*$
  - Idempotence of Union:  $R | R = R$
  - Empty Set in Union and Concatenation:  $R | \emptyset = R, R\emptyset = \emptyset$
  - Empty string in concatenation:  $\epsilon R = R\epsilon = R$
  - Union with the kleene star:  $R^* | R = R^*$
  - Distributive Property:  $R_1(R_2 | R_3) = R_1R_2 | R_1R_3$
  - Absorption:  $R | (RR^*) = RR^* = R^+$
- **The RE operators with the empty language  $\Phi$ :**
  1.  $\emptyset r = r\emptyset = \emptyset\emptyset = \emptyset$  for any regular expression  $r$
  2.  $r + \emptyset = \emptyset + r = r$
  3.  $\emptyset + \emptyset = \emptyset$
  4.  $\emptyset^* = \{\epsilon\}$

These cases can also be represented with language notation

1.  $\Phi L = L\Phi = \Phi\Phi = \Phi \forall L$
2.  $L + \Phi = \Phi + L = L$
3.  $\Phi + \Phi = \Phi$
4.  $\Phi^* = \{\epsilon\}$

- **Convert RE to NFA- $\epsilon$ :** The conversion algorithm starts by defining an NFA with  $\epsilon$ -moves for each of the three base cases from the recursive definition of a regular expressions over an alphabet  $\Sigma$ 
  1.  $\emptyset$  is a regular expression and denotes the empty set (i.e., the empty language  $\Phi$ )
  2.  $\epsilon$  is a regular expression and denotes the set  $\{\epsilon\}$
  3. For each symbol  $x \in \Sigma$ ,  $x$  is a regular expression and denotes the set  $\{x\}$ .

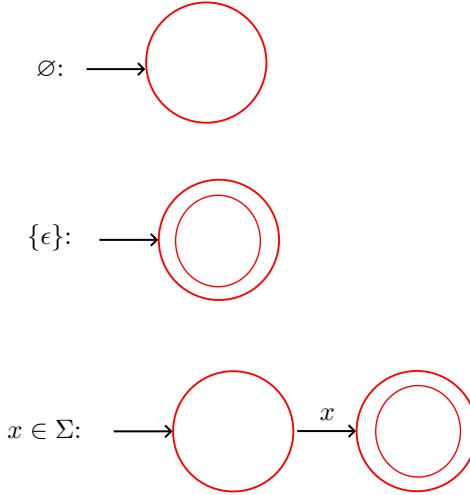
Conditions on the NFAs with  $\epsilon$ -moves for This Algorithm

1. There must be exactly one accepting state.
2. No transitions (not even  $\epsilon$ -moves) may leave the one accepting state.

**Note:** If faced with an NFA with  $\epsilon$ -moves that has more than one accepting state and/or accepting states with transitions leaving it then simply modify the NFA with  $\epsilon$ -moves by

1. Adding a new accepting state.
2. Add an  $\epsilon$ -move from each of the original accepting states to the newly added accepting state.
3. Convert all of the original accepting states to non-accepting states.

The three base cases have the following nfa that satisfy the above criteria



We use those NFAs as the basic building blocks to iteratively build more complex NFA's with  $\epsilon$ -moves (all the while honoring the accepting state conditions for this algorithm) as we apply the recursive part of the regular expression definition. Recall:

If  $r$  and  $s$  are regular expressions denoting the sets  $R$  and  $S$ , respectively, then

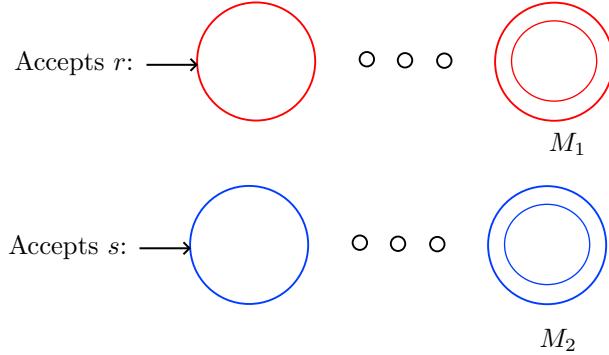
1.  $r + s$  is a regular expression denoting the set  $R + S$ , (i.e., union of languages),
2.  $rs$  is a regular expression denoting the set  $RS$  (i.e., concatenating languages), and
3.  $r^*$  is a regular expression denoting the set  $R^*$  (i.e., Kleene closure of a language).

Once we define an NFA with  $\epsilon$ -moves for each of the base cases (which we have done) then when we address each recursive part of the definition (e.g., union above)

1. We may assume that there already exists NFAs with  $\epsilon$ -moves for each of the regular expressions  $r$  and  $s$  (and that each also satisfies the acceptance state conditions of this algorithm) and
2. Then our job is to use those NFAs with  $\epsilon$ -moves to create a new NFA with  $\epsilon$ -moves that accepts  $r + s$  and that also satisfies the acceptance state conditions of this algorithm.

### The algorithm:

- **Handling union:** We start by assuming there already exists NFAs with  $\epsilon$ -moves  $M_1$  and  $M_2$  that accept regular expressions  $r$  and  $s$ , respectively, and that both  $M_1$  and  $M_2$  satisfy the acceptance state conditions (i.e., one accepting state, no exit) of this algorithm.

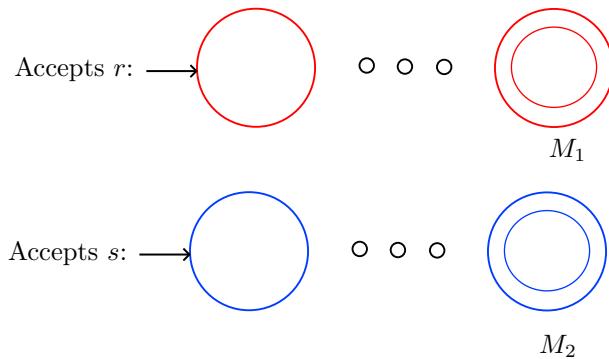
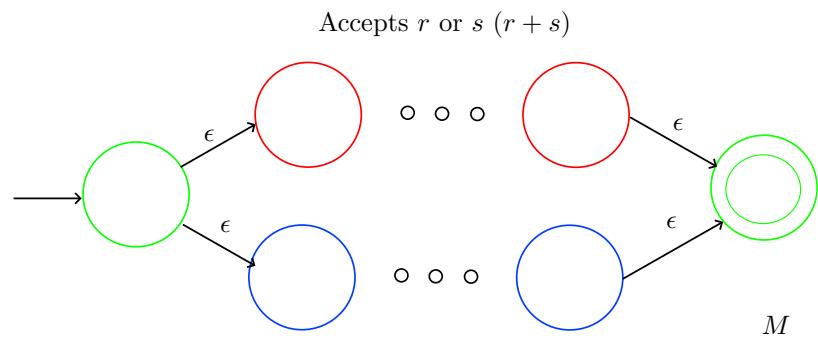


**Note:** The details of the machine aren't important here, all we know is the machine has a start, does whatever else it needs to (represented by the ellipsis), and then accepts strings represented by  $r$  in the top machine and  $s$  in the bottom

We then use these machines  $M_1$  and  $M_2$  to create a new machine  $M$  that accepts  $r + s$

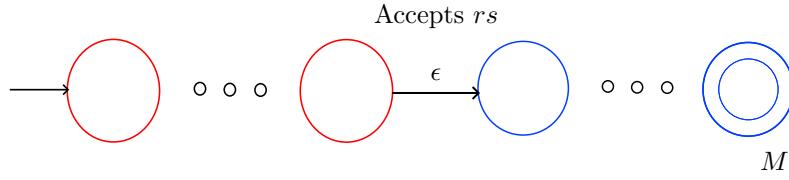
So what did we do here

1. Create new start state and add  $\epsilon$ -moves to the original start states
  2. Create new accepting state and add moves from all the original accepting states.
  3. Change the original accepting states to non-accepting states.
- **Handle Concatenation:** We again start by assuming there already exists NFAs with  $\epsilon$ -moves  $M_1$  and  $M_2$  that accept regular expressions  $r$  and  $s$ , respectively, and that both  $M_1$  and  $M_2$  satisfy the acceptance state conditions (i.e., one accepting state, no exit) of this algorithm.

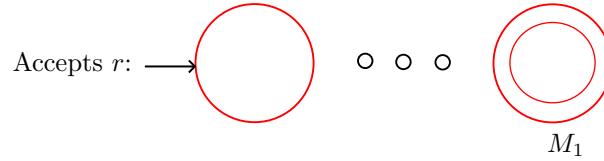


We use  $M_1$  and  $M_2$  to construct new NFA with  $\epsilon$ -move  $M$  that accepts  $rs$ .

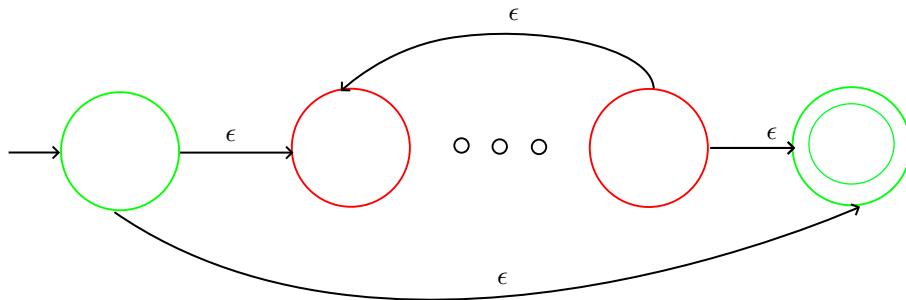
1. Add an  $\epsilon$ -move from  $M_1$ 's accepting state to  $M_2$ 's start state.
2. Change  $M_1$ 's accepting state to a nonaccepting state.



- **Handle Kleene closure:** We again start by assuming there already exists an NFA with  $\epsilon$ -moves  $M_1$  that accepts regular expressions  $r$  and that satisfies the acceptance state conditions (i.e., one accepting state, no exit) of this algorithm.



We use  $M_1$  to construct new NFA with  $\epsilon$ -move  $M$  that accepts  $r^*$

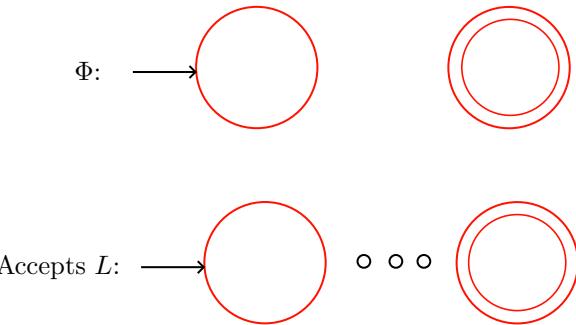


1. Create new start and accepting states.
2. Add  $\epsilon$ -move from new start to  $M_1$  start,  $M_1$  accepting to new accepting, and new start to new accepting.
3. Add  $\epsilon$ -move from  $M_1$  accepting to  $M_1$  start.
4. Change  $M_1$ 's accepting state to a non accepting state.

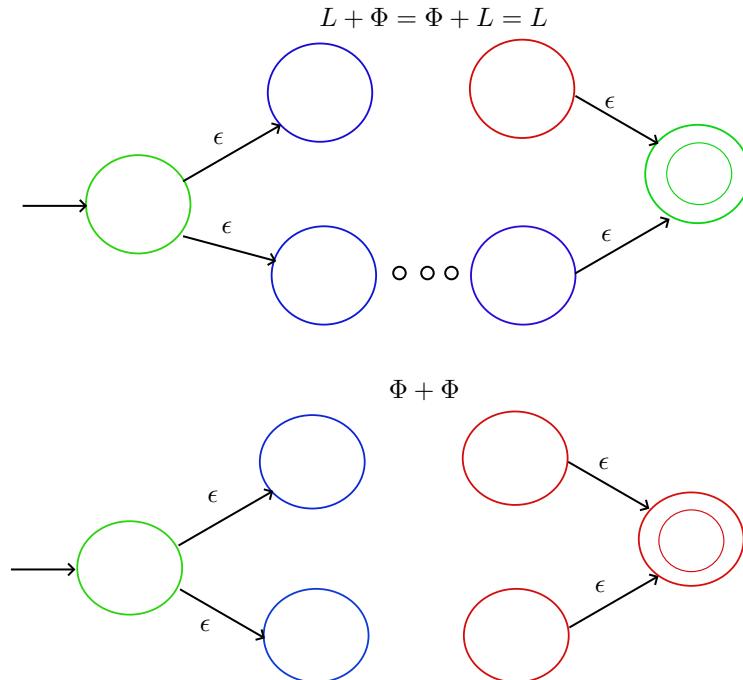
- **RE to NFA conversion: Special cases:** Recall the special case with regular expressions, the empty language  $\Phi$  - and how it behaved with the three regular expression operators;

1.  $\Phi L = L\Phi = \Phi\Phi = \Phi \forall L$
2.  $L + \Phi = \Phi + L = L$
3.  $\Phi + \Phi = \Phi$
4.  $\Phi^* = \{\epsilon\}$

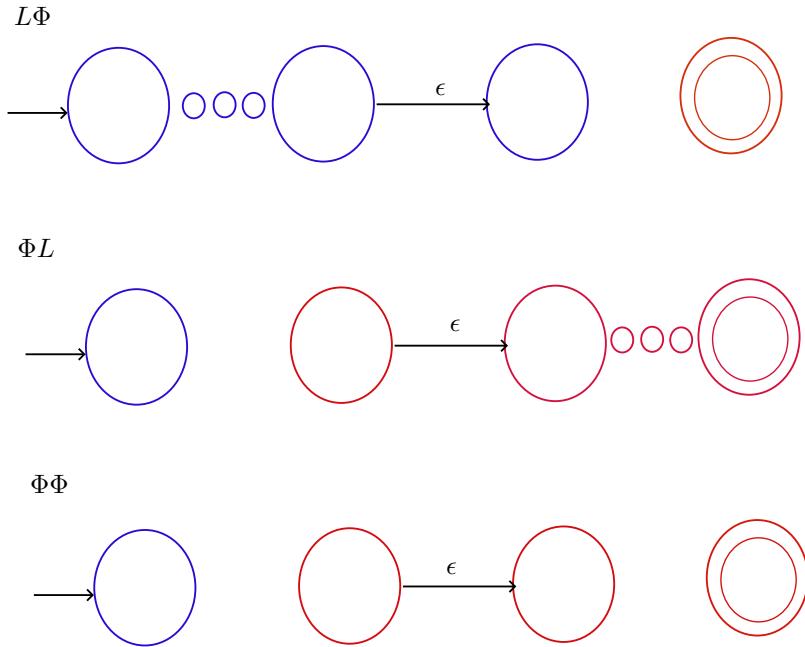
We can now check these operations using the algorithm with  $\Phi$ . First, we define the base case NFA's



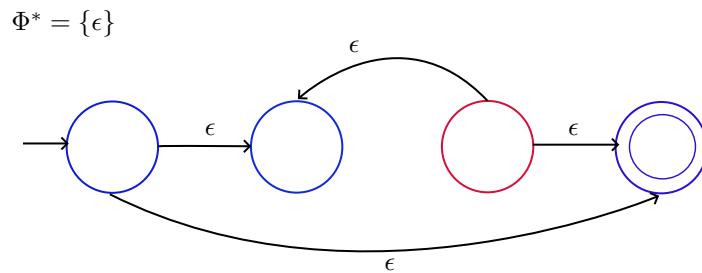
1. Confirming  $L + \Phi = \Phi + L = L$  and  $\Phi + \Phi = \Phi$ :



2. Confirming  $\Phi L = L\Phi = \Phi\Phi = \Phi$



3. Confirming  $\Phi^* = \{\epsilon\}$



- **Convert NFA- $\epsilon$  to RE:** If necessary, first modify the NFA with  $\epsilon$ -moves to satisfy these two conditions (i.e., conditions of this algorithm, not requirements of all NFA's with  $\epsilon$ -moves);
  - No transition may enter the start state - not even a loop.
  - If there exists even one accepting state, then there can be only one accepting state and no transition may leave that accepting state.

**Note:** If there was no accepting state then do not create one, stop the algorithm, and output the regular expression  $\emptyset$  to denote the empty language  $\Phi$ .

Then we start the algorithm. We need to convert the label on each transition to a regular expression until there is only two states, a start state and an accepting state, and all transitions between these two states are regular expressions. The final regular expression will be the union of all transitions.

1. While there are more "middle" states (i.e., states that are neither the start state or accepting state)
  - (i) Select one of the remaining middle states.
  - (ii) Bypass the middle state creating new transitions as necessary annotating each new transition with a regular expression.
  - (iii) Remove the bypassed middle state.
2. If there are any transitions between the start and accepting state, then the regular expression that accepts the same language as the original FA is the the union (i.e., "+") of the regular expressions of all the transitions.

If there are no transitions between the start and accepting state, then output the regular expression  $\emptyset$  to denote the empty language  $\Phi$ .

Recall the following from the definition of regular expressions;

- (i) [base case]:  $L$  is a regular expression and denotes the set  $\{L\}$
- (ii) [base case]: For each symbol  $x \in \Sigma$ ,  $x$  is a regular expression and denotes the set  $\{x\}$
- (iii) [recursive case]:  $r + s$  is a regular expression denoting the set  $R + S$ , (i.e., union of languages).

We use those to covert every  $\epsilon$  to a  $\Lambda$ , every symbol  $x \in \Sigma$  to a regular expression of the same symbol, and every case comma-separated transition label to a regular expression with "+".

After ensuring the FA abides by the start and end state conditions, and we convert every transition to the simple regular expressions, we begin eliminating states.

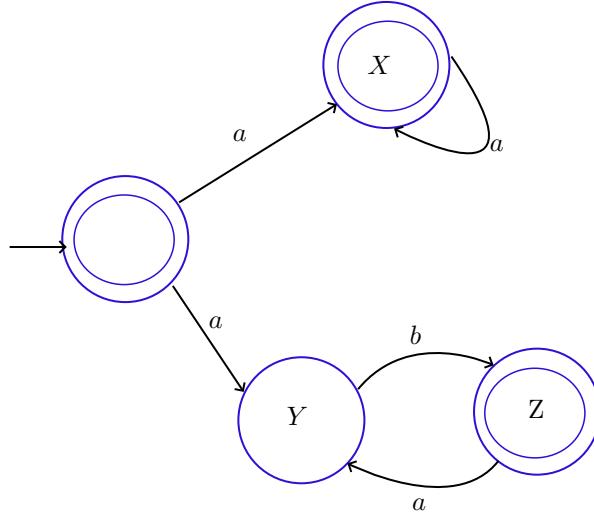
### Some notes:

- (a) Regarding the middle states (states that are neither the start nor accepting state), it doesn't matter in which order we choose to eliminate them.
- (b) For each state we are eliminating, we count the number of incoming and outgoing transitions (loops don't add to the count but we still need to take care of them with the regular expressions), there will be a new regular expression transition for all combinations of outgoing and incoming transitions. Ie pick a state to eliminate, then

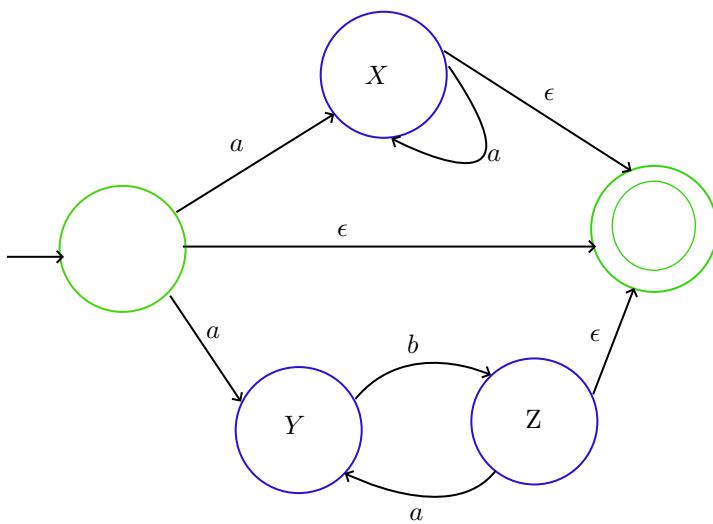
$$\text{New RE transitions} = N(\text{outgoing}) \times N(\text{incoming}).$$

Not including the loops

**Example:** Consider the NFA- $\epsilon$

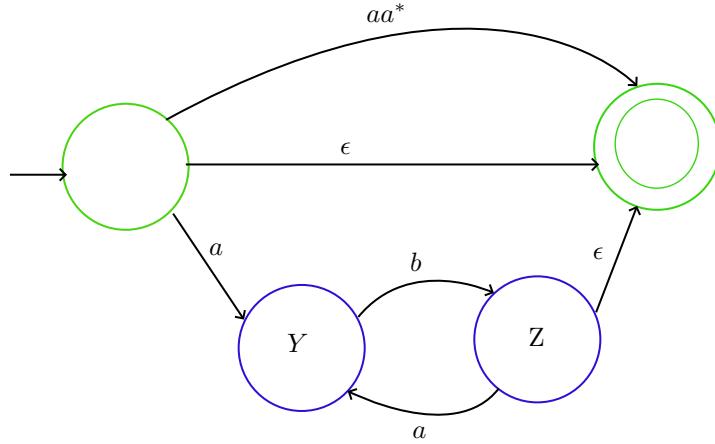


Before we begin eliminating states, we see that this FA does not obey the two constraints described above. So, we create a new accepting state such that there is only one accepting state. Each old accepting state has  $\epsilon$  transitions to this new accepting state. This FA has no incoming transitions to the start state so nothing to fix there.

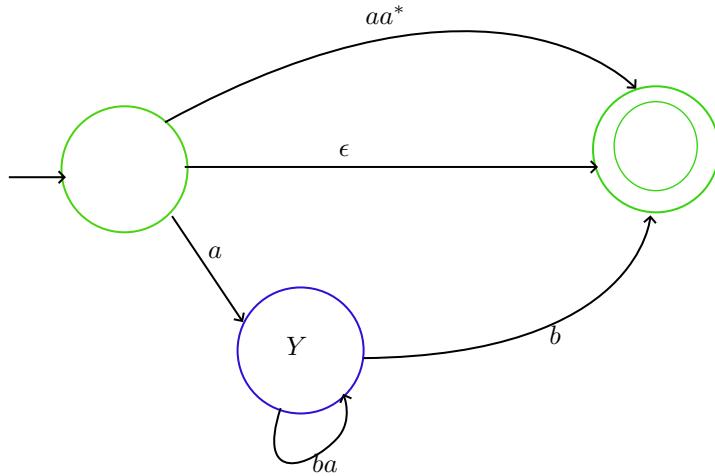


Now, we start eliminating states one at a time. We recall that it does not matter the order in which we eliminate them.

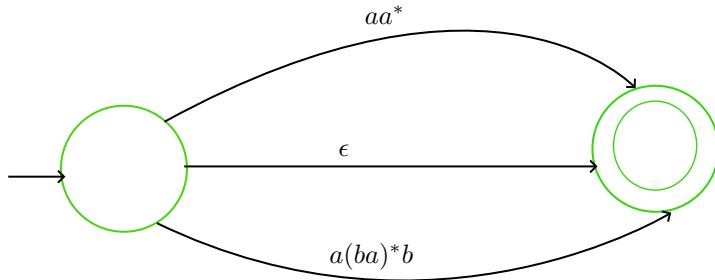
We start by eliminating state  $X$ . There is one incoming transition and one outgoing transition. Thus, there is  $1 \times 1 = 1$  new transition. To get from the start state, through  $X$ , to the accepting state, the regular expression is  $aa^*\Lambda = aa^*$ . Thus,



Next, we choose to eliminate state  $Z$ . We have one incoming and two outgoing transitions. Thus, we have  $1 \times 2 = 2$  new regular expression transitions. To get from  $Y$  through  $Z$  to the accepting state, the RE is  $b\epsilon = b$ . To go from  $Y$  through  $Z$  back to  $Y$ , we have  $ba$ . Thus



Finally, we eliminate  $Y$ . To get from the start state, through  $Y$ , to the accepting state, the regular expression is  $a(ba)^*b$ . Thus,



The final regular expression is then

$$aa^* + \epsilon + a(ba)^*b.$$

■

### 1.2.5 Properties of regular languages

- **Recall: Regular language:** Recall that we call a language a regular language if, and only if, the language is accepted by some regular expression.
- **Recall: Recursive definition of regular expressions:** Recall also our recursive definition of regular expressions over some alphabet  $\Sigma$

Let  $\Sigma$  be an alphabet. The regular expressions over  $\Sigma$  and the sets (i.e., languages) that they denote are defined recursively as follows:

#### Base cases:

1.  $\emptyset$  is a regular expression and denotes the empty set (i.e., the empty language  $\Phi$ ).
2.  $L$  is a regular expression and denotes the set  $\{L\}$ .
3. For each symbol  $x \in \Sigma$ ,  $x$  is a regular expression and denotes the set  $\{x\}$ .

**Recursion:** If  $r$  and  $s$  are regular expressions denoting the sets  $R$  and  $S$ , respectively, then

1.  $r + s$  is a regular expression denoting the set  $R + S$ , (i.e., union of languages),
2.  $rs$  is a regular expression denoting the set  $RS$  (i.e., concatenating languages), and
3.  $r^*$  is a regular expression denoting the set  $R^*$  (i.e., Kleene closure of a language).

- **Relationship between regular languages and the set of all possible languages  $\mathcal{P}(\Sigma^*)$ :** We know that the set of regular languages must be a subset of  $\mathcal{P}(\Sigma^*)$  (it may be equal to  $\mathcal{P}(\Sigma^*)$ ), and so, we can start by creating that subset.

We can then start noting the languages that we know are regular based on the base cases from the definition of regular expressions and for some given alphabet, say  $\Sigma = \{a, b\}$ .

The recursion from the definition tells us that we can take any language, or pair of languages, from the already existing set of regular languages, and use it/them to create a new language this is also a regular language. And the recursion may be applied over and over (i.e., without limit), always taking only regular languages that have been previously created (original base case languages or languages subsequently derived), to create new languages. (i.e., the set of regular languages is infinite).

- **Closure of regular languages and their operations:** Expanding on the item above, formally, we say that the set of regular languages is **closed** under the language composition operations union, concatenation, and Kleene star
- **Closure of complement and intersection:** In addition to the three operations that come from the definition of regular expressions (i.e., union, concatenation, and Kleene star), the set of regular languages is also closed under;
  - Complement
  - Intersection
- **Complement of a Language:** Recall that every language is a set of strings - empty, finite, or infinite - that is always a subset of  $\Sigma^*$

That is,

- For any alphabet  $\Sigma$  we get  $\Sigma^*$
- We define some language  $L$  over that alphabet  $\Sigma$
- Then  $L$  is a subset of  $\Sigma^*$ ;  $L \subseteq \Sigma^*$

We define a new language, the complement of  $L$ , denoted  $L'$  as the set of strings that are not in the language  $L$  (i.e.,  $L' = \Sigma^* - L$ ).

- **Proof: Regular languages are closed under complement:** We assert that if you take the complement of a regular language, the resulting language is then regular

#### **Proof:**

1. A regular language is one that is accepted by some regular expression.
2. By Kleene's Theorem we know that any regular expression can be converted to a NFA with  $\epsilon$ -moves that accepts the same language, and vice versa.
3. We can convert any NFA with  $\epsilon$ -moves to a DFA that accepts the same language and every DFA is, by definition, an NFA with  $\epsilon$ -moves.

So  $L$  is a regular language iff it is accepted by some DFA...

Consider some DFA  $M(Q, \Sigma, q_0, T, \delta)$  that accepts regular language  $L$ .

We construct a new DFA  $M'(Q, \Sigma, q_0, T', \delta)$  from  $M$  by defining  $T' = Q - T$ , that is, every accepting state in  $M$  becomes a non-accepting state in  $M'$ , and vice versa.

Since  $M'$  accepts  $L'$  and  $M'$  is a DFA, then  $L'$  is a regular language.

Since  $M$  was chosen arbitrarily, the complement of any regular language is also a regular language

■

**Note:** Note: The proof must be based on DFA's ... would not have worked for non-deterministic FA's.

- **Intersection of Languages:** Given any two languages,  $L_1$  and  $L_2$ , over some alphabet  $\Sigma$  we can create a new language  $L$  that is the intersection of the two sets  $L_1$  and  $L_2$ .

That is,

$$L = L_1 \cap L_2 = \{x : x \in L_1 \wedge x \in L_2\}.$$

- **Proof: Regular languages are closed under intersection:** This means that when you take the intersection of any two regular languages, the resulting language is always regular.

Assume you have two regular languages  $L_1$  and  $L_2$ .

We define a new language  $L$ , which is the intersection of  $L_1$  and  $L_2$ , namely

$$L = \{x \mid x \in L_1 \wedge x \in L_2\},$$

where “ $\wedge$ ” means “logical and.”

Demorgan’s law:

$$(a \wedge b) = \sim (\sim a \vee \sim b).$$

To prove that  $L = L_1 \cap L_2$  is regular, we use the fact that regular languages are closed under complement and union. This leads us to apply De Morgan’s Law:

$$L_1 \cap L_2 = \sim (\sim L_1 \cup \sim L_2).$$

Here,  $\sim L_1$  and  $\sim L_2$  represent the complements of  $L_1$  and  $L_2$ , respectively. Since  $L_1$  and  $L_2$  are regular, their complements  $\sim L_1$  and  $\sim L_2$  are also regular (because regular languages are closed under complement).

Next, since regular languages are closed under union, the language  $\sim L_1 \cup \sim L_2$  is also regular.

Finally, the complement of  $\sim L_1 \cup \sim L_2$ , i.e.,  $(\sim L_1 \cup \sim L_2)'$ , is also regular because regular languages are closed under complement. But  $(\sim L_1 \cup \sim L_2)'$  is precisely  $L_1 \cap L_2$ .

Thus,  $L = L_1 \cap L_2$  is regular, as required.

$$L_1 \cap L_2 = \sim (\sim L_1 \cup \sim L_2) = \{x \mid x \in L_1 \wedge x \in L_2\}.$$

■

- Addings to regular expression recursive: Thus, we add complement and intersection to recursive cases when building regular languages defined in  $\mathcal{P}(\Sigma^*)$

**Recursion:** If  $r$  and  $s$  are regular expressions denoting the sets  $R$  and  $S$ , respectively, then

1.  $r + s$  is a regular expression denoting the set  $R + S$ , (i.e., union of languages),
2.  $rs$  is a regular expression denoting the set  $RS$  (i.e., concatenating languages), and
3.  $r^*$  is a regular expression denoting the set  $R^*$  (i.e., Kleene closure of a language).

or by taking the complement or intersection of one or two previously created regular languages.

### 1.2.6 Applications of finite automata

- **FA with output:** Thus far we have only considered finite automata as language acceptors (i.e., defining some regular language).

Finite automata can also serve another purpose. They can be used to process an input string to produce some output.

When used in this way, the finite automata do not define a language. In fact, they do not have any accepting states.

Their sole purpose is to process input to generate output

- **Moore machine:** A Moore machine is a deterministic finite automaton and is defined by a 6-tuple  $(Q, \Sigma, q_0, \delta, \Gamma, O)$ , where
  - $\Gamma$  is an alphabet of output symbols.
  - $O$  is the output function:  $O : Q \rightarrow \Gamma$

Each state is annotated with an output symbol. Output "printed" upon entering state

**Note:** start state's output symbol always "printed", even on empty string  $\epsilon$

- **Mealy Machine:** A Mealy machine is a deterministic finite automaton and is defined by a 5-tuple  $(Q, \Sigma, q_0, \delta, \Gamma)$ , where
  - $\Gamma$  is an alphabet of output symbols

Each transition is annotated with an output symbol Output "printed" when traversing edge

**Note:** The number of input and output symbols are always identical.

### 1.3 Nonregular languages

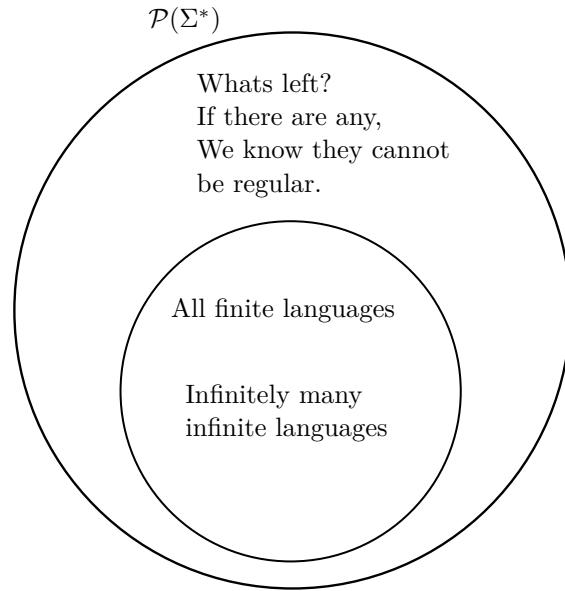
- **Intro to nonregular languages:** Suppose we have some alphabet  $\Sigma$ , we can then find  $\Sigma^*$ , which is the language consisting of all possible strings (including the empty string  $\epsilon$ ) using the symbols from  $\Sigma$ . Then we can take the power set of  $\Sigma^*$  to get the set of all possible subsets of  $\Sigma^*$ . The alphabet  $\Sigma$  is always finite, if  $\Sigma$  is nonempty, then  $\Sigma^*$  is always infinite. (If  $\Sigma = \emptyset$ ,  $\Sigma^* = \{\epsilon\}$ ). Since  $\Sigma^*$  is infinite,  $\mathcal{P}(\Sigma^*)$  is also infinite. If  $\Sigma = \emptyset \implies \Sigma^* = \{\epsilon\} \rightarrow \mathcal{P}(\Sigma^*) = \{\{\epsilon\}, \emptyset\}$ . Note that for any set  $S$ ,  $\emptyset \subset S$ .

What types of languages are contained in  $\mathcal{P}(\Sigma^*)$ ?

1. If a language is finite, it is always regular. This is a consequence of Kleene's theorem, which asserts that if a regular expression expresses a language, the language is regular. If a language is finite, we can **always** make a regular expression for it. Simply take all strings in the language, and take the union. For example, if  $L = \{a, ab, abc\}$ , then a regular expression for the language is simply  $a + ab + abc$  and the language is therefore regular. ■
2. We also know that there are infinitely many regular expressions than can express infinitely many languages from  $\mathcal{P}(\Sigma^*)$ . This is a consequence of the recursive definition of regular expressions. Therefore there are infinitely many regular expressions to describe infinite languages.

So,  $\mathcal{P}(\Sigma^*)$  is the infinite set of all possible languages, finite or infinite, that can be generated from a given language  $\Sigma$ . All Finite languages from this set are regular, and there are also infinitely many infinite languages from this set that are regular. But, are there any languages that are *not* regular? Infinite languages that cannot be expressed as a regular expressions?

It may seem unlikely that nonregular languages exist at all. To claim that a language is nonregular one must prove that no regular expression or FA that accepts the language exists



- **The Pumping Lemma:** Before we continue our conversation about nonregular languages we first look at a lemma about regular languages

**Lemma.** Let  $L$  be an infinite regular language, then

$$\exists x, y, z \in \Sigma^* \mid y \neq \epsilon \wedge xy^n z \in L \forall n > 0.$$

In other words, all regular languages have the property that, there exists some strings  $x, y, z$ , where  $y \neq \epsilon$  such that all the strings of the form  $xy^n z \in L$  for all  $n > 0$

This means that for some  $x, y, z$ , we can infinitely pump in more copies of  $y$ , and the string remains in the language. For example,

$$xyz \quad xyyz \quad xyyyz \quad \dots$$

- **Pumping Lemma Proof:** Assume you have some regular language  $L$  with infinitely many strings. Because  $L$  is regular, there must exist some DFA  $M(Q, \Sigma, q_0, T, \delta)$  that accepts  $L$ . Since FA's are required to have finite states, let's say it has  $n$  states.

Because  $L$  is infinite, we can always find strings that have length greater than  $n$ . Ie  $|w| \geq n$ , where  $w \in L$ . Because  $w$  has at least as many characters as there are states in  $M$ , as we process  $w$  with  $M$  we know that one or more of the states in  $M$  must be revisited. We know that as we process  $w$ , it must traverse what we call the *circuit*, which is a sequence of one or more edges that contains at least one state that is visited more than once.

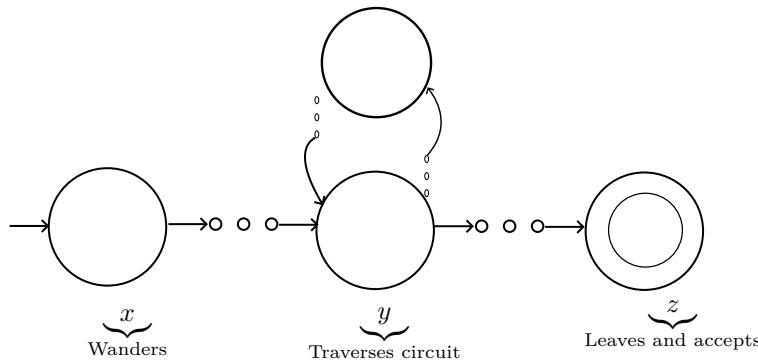
Given that circuit and because we also selected  $w \in L$ , we note that we can modify  $w$  to create a new word  $w'$  pumping into  $w$  as many symbols as are needed and in just the right location in  $w$  that would cause  $M$  to traverse the circuit one more time than it did when processing  $w$ . We note that the new word  $w' \in L$ .

In fact, given DFA  $M$  with  $n$  states and string  $w \in L$ ,  $|w| \geq n$ , we can generate an infinite supply of new words by simply pumping into  $w$ , and at the right location, more and more copies of the string that causes  $M$  to traverse the circuit. We note that the new words created in this way are all in  $L$ .

This gives us the existence of the  $x, y, z$  strings the Pumping Lemma needs as follows:

- $x$  is the prefix of  $w$  that is consumed by  $M$  as the DFA wanders up to the circuit ( $x$  may be  $\Lambda$  and this sequence of states may be empty).
- $y$  is the substring of  $w$  that is consumed by  $M$  as the DFA traverses the circuit (since the circuit must visit at least one state more than once, it must consume some symbols, and so  $y$  cannot be  $\Lambda$ ).
- $z$  is the suffix of  $w$  that is consumed by  $M$  as the DFA leaves the circuit and goes to an accepting state ( $z$  may be  $\Lambda$  and this sequence of states may be empty).

Therefore,  $L$  must contain all the strings of the form  $xy^n z$  for all  $n > 0$ .



- **The value of the pumping lemma:** The Pumping Lemma tells us something that is true of all regular infinite languages.

The real value of The Pumping Lemma is to prove that some infinite language is nonregular, that is a language **cannot** be regular if it does not satisfy the claim in the pumping lemma. Thus, we can prove that a language is non-regular by contradiction.

1. Assume the language is regular
  2. You show that it is not possible to find strings  $x, y, z$  that satisfy the claim in the pumping lemma.
  3. We conclude that our assumption that the original language is a regular language must be false, and therefore, it must be nonregular.
- **Note on the pumping lemma:** The pumping lemma guarantees that for a large enough  $w \in L$  ( $|w| \geq n$ ), where  $n$  is the number of states in the assumed machine) we can find an  $x, y, z$  such that  $y \neq \epsilon$  and  $xy^n z \in L \forall n > 0$ . If the word you select is greater than or equal to  $n$ , and no such  $y$  holds, then the language must be nonregular

The condition is  $\geq n$ , because, for a machine with  $n$  states, there are  $n - 1$  edges that must be traversed to reach the end. If a machine has four states, then there are  $4 - 1 = 3$  edges to reach the end.

- **The pumping language with length: Background:** In the case that you find an  $x, y, z$  such that  $y \neq \epsilon$  and  $xy^n z \in L \forall n > 0$ . This does **not** mean the language must be regular. We know that all regular languages do have this property, but that does not mean that simply possessing this property makes the language regular. In logic theory

$$a \rightarrow b \not\implies b \rightarrow a.$$

In words, if  $a$  implies  $b$ ,  $b$  does not imply  $a$

Thus, the pumping lemma described above is sometimes not enough, and we may need something more powerful.

- **The pumping lemma with length:** Let  $L$  be an infinite language accepted by a FA with  $n$  states. Then, for all  $w \in L$ , where  $|w| \geq n$ , there exists some three strings  $x, y, z$  such that  $w = xyz$ ,  $y \neq \epsilon$ ,  $|xy| \leq n$ , and all the strings of the form  $xy^i z \in L$  for all  $i > 0$

The Pumping Lemma With Length adds that you must always be able to find  $x, y, z$  in all sufficiently long words  $w \in L$ . This means:

- For each word in  $L$  that has length greater than  $n$ , where  $n$  is the number of states in the assumed machine, there must be a composition  $xyz$ , where  $x, y, z \in \Sigma^*$ ,  $y \neq \epsilon$ , and  $|xy| \leq n$ . The  $x, y, z$  need not be the same for every word in  $L$  with length greater than  $n$ , but a pair needs to exist for each word.
- Furthermore, for each word  $w$ , where  $|w| \geq n$ , that has  $x, y, z$  such that  $w = xyz$ . That same  $x, y, z$  needs to have the property  $xy^i z \in L \forall i > 0$ .
- Thus, to show that a language is not regular, we only need to show that such an  $x, y, z$  does not exist for a single word. If we choose a word and  $x, y, z$  exists, we need to keep looking.

If words keep leading to valid  $x, y, z$  at some point we need to stop looking and start trying to prove that the language is actually regular. Whether by creating an RE or an FA. Keep in mind there will be infinite words to check.

### 1.3.1 Pumping lemma examples

- **Pumping lemma example:** Use the pumping lemma to show that the language  $L = a^k b^k \forall k > 0$  over  $\Sigma = \{a, b\}$  is nonregular

Suppose that  $L$  is regular, then we should be able to find some  $x, y, z$ , where  $y \neq \epsilon$  such that  $xy^n z \in L \forall n > 0$ . For simplicity, let's first examine the possible choices for  $y$

1.  $y = a^\ell$  or  $y = b^\ell$  for some  $\ell > 0$
2.  $y = a^\ell b^\lambda$  for some  $\ell, \lambda > 0$

In case one, pumping would lead to an imbalance in the number of  $a$ 's or  $b$ 's. In case two, pumping would lead to more than one  $ab$  boundary.

Thus, no such  $x, y, z$  exists and the language is nonregular ■

- **Pumping lemma example:** Show that the language  $L = \{a^t : t \in \mathbb{P}\}$  over  $\Sigma = \{a\}$  is nonregular.

Suppose  $L$  is regular. Then we will find strings  $x, y, z$  such that  $y \neq \epsilon$  and  $xy^n z \in L \forall n > 0$ .

The only choice of  $y$  in this case is  $y = a^r$ ,  $r > 0$ .

First, define  $w = a^t = xyz$ ,  $t \in \mathbb{P}$ . That is, since  $w$  is a member of  $L$ , we can partition it into the form  $a^t = xyz$ . Furthermore,  $xy^n z \in L \forall n > 0$ . Since this needs to hold **for all**  $n > 0$ , showing that it doesn't work for a single  $n > 0$  breaks the argument. Let  $n = t + 1$ . This leads us to some algebraic manipulations

$$\begin{aligned} xy^n z &= xy^{t+1} z = xyy^t z \\ &= xyzy^t \quad (\text{Everything is } a, \text{ we can commute}) \\ &= a^t y^t \\ &= a^t (a^r)^t \\ &= a^{t+r t} \\ &= a^{t(1+r)}. \end{aligned}$$

This next assertion is one of number theory. We assert that if  $t \in \mathbb{P}$ ,  $t(r + 1) \notin \mathbb{P}$  (Since  $r > 0$ ,  $r + 1 > 0$ ).

Since we only have one choice of  $y$  in this case, and we showed that it does not hold when  $n = t + 1$ , the language must be irregular. ■

- **Using the Pumping Lemma With Length: Palindrome example:** Prove that the language *Palindrome* over  $\Sigma = \{a, b\}$  is a nonregular language.

Assume there exists an FA with  $n$  states that accepts palindrome. Consider  $w = a^{n+1}ba^{n+1} \in \text{Palindrome}$

Since we assume  $L$  is regular, then for  $w = a^{n+1}ba^{n+1}$ , which is clearly a palindrome with length  $\geq n$ , must have the property  $w = xyz$ , for some  $x, y, z \in \Sigma^*$ , where  $y \neq \epsilon$ , and  $|xy| \leq n$ . Thus,  $y$  must be contained within  $a^{n+1}$ , which implies  $xy$  must be contained within  $a^{n+1}$ . This leads to the conclusion that pumping more copies of  $y$  will lead to an  $a$  imbalance to the left of the  $b$ , which is **not** a palindrome. Thus, the language is nonregular. ■.

- **Pumping lemma with length example:** Show with the pumping lemma that  $L = a^n b^m c^{n+m}$ ,  $\forall m, n > 0$  is a nonregular language

The pumping lemma states that for an infinite regular language that has an FA with  $k$  states, then

$$\forall w \in L, |w| \geq n, \exists x, y, z \in \Sigma^* \mid w = xyz, |xy| \leq n \wedge xy^i z \in L \forall i > 0.$$

Let  $m, n = k$ , then  $w = a^k b^k c^{2k}$ . Since  $|xy| \leq n$ ,  $y$  must be  $a^r$ ,  $0 < r \leq k$ . This implies  $w = a^r a^{k-r} b^k c^{2k}$ . Furthermore,  $w' = a^{ir} a^{k-r} b^k c^{2k} \in L \forall i > 0$ . If  $i = 2$ ,  $w' = a^{2r} a^{k-r} b^k c^{2k} = a^{k+r} b^k c^{2k}$ . For  $w'$  to be in  $L$ ,  $2k$  must equal  $k + r + k$ . Since  $2k \neq k + r + k$ , we have a contradiction. Thus, pumping more copies of  $y$  not satisfy  $a^n b^m c^{n+m}$ , as the number of  $c$ 's would not equal the number of  $a$ 's plus the number of  $b$ 's ■.

- **Pumping lemma with length example:** Over the alphabet  $\Sigma = \{a, b, c\}$ , show that the language that houses all strings that are not palindromes is nonregular.

Assume the language has an FA with  $k$  states, then  $\forall w \in L, |w| \leq k, \exists x, y, z \in \Sigma^* \mid w = xyz, |xy| \leq k \wedge xy^i z \in L \forall i > 0$ .

An easy way to prove this is by showing that  $L = \text{Palindrome}$  is nonregular, which is much simpler. Since regular languages are closed under complement, the complement of a regular language must be regular. Thus, the complement of a nonregular language must be nonregular.

Assume  $L = \text{palindrome}$  is infinite and regular defined by an FA with  $k$  states.

Let  $w = a^k b^k a^k$ , then  $y = a^r$ ,  $0 < r \leq k$ . Then  $w = a^r a^{k-r} b^k a^k$  and  $w' = a^{ir} a^{k-r} b^k a^k \in L \forall i > 0$ . If  $i = 2$ ,  $w' = a^{2r} a^{k-r} b^k a^k = a^{k-r+2r} b^k a^k = a^{k+r} b^k a^k$ . Since  $r > 0$ ,  $k + r > k \implies a^{k+r} > a^k$  and  $w'$  is not a palindrome. Thus, we have a contradiction and  $L$  must be nonregular. Since  $L$  is nonregular,  $L'$  must be nonregular. Therefore, the language of all strings that are not palindromes is nonregular. ■.

- **Pumping lemma with length example:** Show that the language  $a^t$ ,  $t \in \mathbb{P}$  over the alphabet  $\Sigma = \{a\}$  is nonregular.

Assume the language is regular defined by a FA with  $n$  states. Choose  $w = a^\ell$ ,  $\ell \in \mathbb{P}$ ,  $\ell \geq n$ . Then,  $y$  must be  $a^r$ ,  $r > 0$ . From this,  $w = xyz = a^r a^{\ell-r}$ , and  $w' = xy^i z \in L \forall i > 0$ . Thus,  $w' = a^{ir} a^{\ell-r} \in L \forall i > 0$ . If we can show that some selection of  $i$  yields a  $w' \notin L$ , then we have a contradiction and the language must be nonregular.

Choose  $i = \ell + 1$ , which implies  $w' = a^{(\ell+1)r} a^{\ell-r} = a^{\ell-r+\ell r+r} = a^{\ell+\ell r} = a^{\ell(1+r)}$ . Since  $\ell \in \mathbb{P}$ , and  $r > 0$ ,  $\ell(1+r)$  cannot be prime. Thus, we have a contradiction and the assumption does not hold for  $L = a^t$ ,  $t \in \mathbb{P}$ .

## 1.4 Context free grammars

- A **Context Free Grammar** (CFG) is a 4-tuple  $(V, \Sigma, S, P)$ , where
  - $V$  is a non-empty finite set of *variables*
  - $\Sigma$  is a finite alphabet of *terminals*  
(we assume  $V$  and  $\Sigma$  are disjoint)
  - $S \in V$ , is the *start variable*
  - $P$  is a finite set of *productions* of the form  $A \rightarrow \alpha$  where
    - \*  $A$  is a variable (i.e.,  $A \in V$ ) and
    - \*  $\alpha$  is a string of symbols from  $(V \cup \Sigma)^*$  (i.e.,  $\alpha \in (V \cup \Sigma)^*$ ).
- **CFG notational convention:** Variables are upper case letters with  $S$  always being the start variable.

Terminals are lower case letters, symbols, or constants, including  $\epsilon$  to denote the empty symbol.

- **A simple CFG:** Consider the following CFG that has two productions

$$\begin{aligned} S &\rightarrow 0S1 \\ S &\rightarrow \epsilon. \end{aligned}$$

- **Derivations:** We say that a finite string  $w$ , consisting only of terminals, is generated by a CFG if, starting with the start variable  $S$ , you can apply a sequence of productions that result in  $w$ .

The sequence is called a derivation of  $w$ .

- **Derivation examples:** All derivations must start with the start variable  $S$ .

As long as there is at least one variable in our string we must continue the derivation by

1. Selecting a variable from our string,
2. Selecting a production whose left side matches the variable we selected from our string, and
3. Replacing the variable in our string with the right side of the production we selected.

With the two productions

$$\begin{aligned} S &\rightarrow 0S1 \\ S &\rightarrow \epsilon. \end{aligned}$$

We select the first production and apply it

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 00\epsilon11 = 0011.$$

This yields the string  $w = 0011$ . Thus, we have derived this string from the grammar. And we also note that there are some other strings for which there is no derivation using our CFG

In short, we observe that a CFG defines a language over its alphabet of terminals  $\Sigma$ .

- **Context Free Languages:** A language is a context free language if it is generated by some context free grammar.
- **Is palindrome context free?**: Recall that the language Palindrome (i.e., the set of all strings that read the same forward as they do backward) is a non-regular language.

We can create a grammar for this language

$$\begin{aligned} S &\rightarrow aSa \\ S &\rightarrow bSb \\ S &\rightarrow a \\ S &\rightarrow b \\ S &\rightarrow \epsilon. \end{aligned}$$

The CFG builds the string from the middle always pushing outward by inserting the same pair of characters each time.

- **Notational convenience:** We can express the grammar above simply as

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon.$$

Where the pipe delimits each production. Note that they are still separate productions, but since they share the same variable  $S$ , we can put them on the same line separated by a pipe.

- **Connection between regular and context free languages:** We assert that every regular language is generated by a context free grammar.

**Proof:** There is a recursive algorithm that converts any regular expression to a context free grammar that accepts the same language.

- If the overall language is the union of two or more pieces, then you can start with  $S \rightarrow A_1|A_2|\dots|A_n$
- If the overall language is the concatenation of two or more pieces, then you can start with  $S \rightarrow A_1A_2\dots A$
- If the overall language is the Kleene star of a piece, say generated by  $E$ , then you can start with  $S \rightarrow ES \mid \epsilon$
- Recursively and similarly generate productions for variables until you are left with only terminals.

**Example:** Consider the regular expression  $(11 + 00)^*11$

At its top level it is a concatenation of  $(11 + 00)^*$  and  $11$ , so we start with the following production:

$$S \rightarrow AB$$

We continue by generating productions for each new variable we introduced  $A$  and  $B$ .

$A$  is a Kleene Star and  $B$  is straightforward so we add the following productions:

$$A \rightarrow CA \mid \varepsilon$$

$$B \rightarrow 11$$

All that remains is production for the variable  $C$  which is a union, so we add this final production.

$$C \rightarrow 11 \mid 00$$

Thus, all regular languages are context free, but not all context free languages are regular, we know that there exist some nonregular languages that are context free, like palindrome.

- **Left and rightmost derivations:**

- **Leftmost derivation:** At each stage one replaces the leftmost variable.
- **Rightmost derivation:** At each stage one replaces the rightmost variable.

- **Derivation (parse) trees:** It is sometimes useful to display derivations as trees called derivation trees or parse trees. Where a parse tree satisfies

- The root of the tree is always the CFG start symbol  $S$
- All of the internal nodes (i.e., non-leaf nodes) are variables.
- All of the leaf nodes are terminals.
- The children of a node are ordered left to right and appear in the same order as they do in the right-hand-side of a production whose left-hand-side matches variable in the parent node.
- The word generated by the derivation tree is the sequence of terminals read from the leaf nodes in left-to-right order.

- **Ambiguous parsing:** If for some words in a language there are more than one parse trees, we say it has ambiguous parsing

- **Ambiguous grammar:** A context free grammar (CFG) defining some language  $L$  is said to be an ambiguous grammar iff there exists some word  $w \in L$  that has two different derivation trees.

Because each derivation tree represents unique leftmost and rightmost derivations;

- a CFG is ambiguous iff there exists some word  $w \in L$  for which there are two different leftmost derivations or two different rightmost derivations.

A CFG that is not ambiguous is said to be an unambiguous grammar

### 1.4.1 Pushdown automata (PDA)

- **Intro to PDA:** PDA's are an abstract machine that is used as an acceptor for CFG's, similar to how FAs were used as acceptors for regular languages.
- **Comparison to NFA:** PDAs are closest to NFA's with  $\epsilon$ -moves in that they both:
  - are nondeterministic (need only one set of choices to accept a string),
  - may crash (i.e., not every symbol must leave each state), and
  - allow for  $\epsilon$ -moves

Recall that NFA with  $\epsilon$ -moves also:

- Have finitely many states, some of which are accepting states.
- Read string from an input tape, must read entire string to accept.
- Reject by reaching the end of string while not in an accepting state.

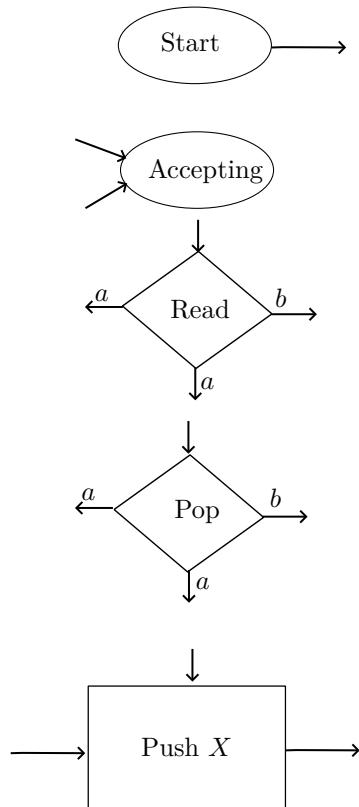
All the same is true for pushdown automata with the following minor revisions and one important addition:

- [revision] There is only one accepting state, once entered accept string (whether the entire input string has been read or not).
  - [revision] There is no notion of reaching the "end of the string".
  - [addition] There is a stack of infinite capacity.
- **The Tape and Stack:** Pushdown automata have a stack of infinite capacity, initially empty.

Because pushdown automata have both an input tape and stack they must distinguish which they are manipulating in each of their states; read, push, or pop.

- **State Symbols:**

- **Start state:** No incoming edges, one outgoing edge
- **Accepting:** state No outgoing edges
- **Read:** (nondeterministic)
- **Pop:** (nondeterministic)
- **Push:** One outgoing edge



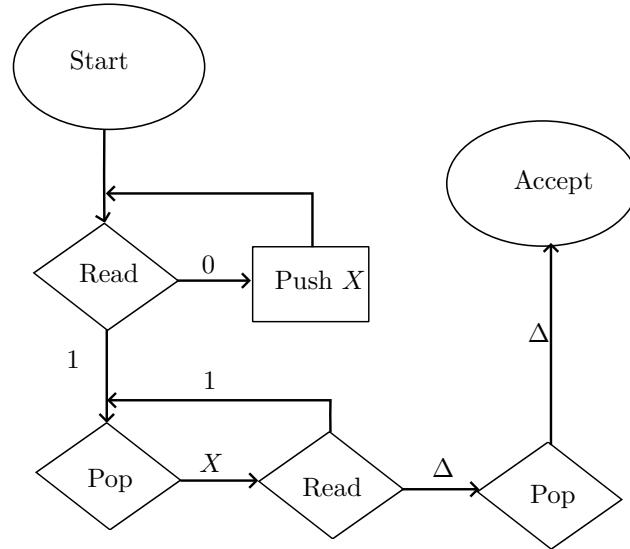
- **Initialization:** Like finite automata, we start pushdown automata by writing the string onto the input tape. Recall that the input tape has a beginning and an infinite capacity.

In pushdown automata, after we write the string onto the input tape we append an infinite sequence of a special symbol  $\Delta$  to denote the end of the string (i.e., assumed that  $\Delta$  is not part of the input alphabet).

We initialize the pushdown automaton's stack with an infinite supply of the special symbol  $\Delta$ .

We place the pushdown automaton in its START state

- **PDA Example:** Consider the PDA that accepts the language  $0^n 1^n \forall n > 0$



It starts by reading all of the leading 0's and using the STACK to "count" how many 0's have been read (i.e., by pushing an  $x$  for each 0 that was read).

Once we encounter our first 1, we enter a different part of the PDA in which we confirm there is an  $x$  on the stack for every 1 we read from the tape.

If we READ a 1 and then pop something other than a  $x$ , then the PDA crashes in the POP state and rejects the string. This happens when there are more 1's than 0's.

Once we encounter our first  $\Delta$  then we have reached the end of the input string and we are assured that there are at least as many 1's as there are 0's.

We still must POP the stack to confirm that there are no  $x$ 's. If there was an  $x$  on the STACK then there were more 0's than 1's.

### 1.4.2 Chomsky Normal Form

- **Intro:** Noam Chomsky showed that it is possible to convert any CFG (i.e., all productions of the form  $A \rightarrow v$  to another CFG that accepts the same language where all the productions are either

$A \rightarrow$  exactly two variables  
or  
 $A \rightarrow$  one terminal (the terminal cannot be  $\epsilon$ ).

Which converts the CFG into Chomsky Normal Form (CNF)

The conversion is done in four steps which must be in this sequence:

1. Eliminate null productions
2. Eliminate unit productions
3. Almost CNF
4. CNF

- **Eliminate Null Productions:** A null production is of the form  $A \rightarrow \epsilon$ .

We start our conversion to Chomsky Normal Form (CNF) by first eliminating all null productions and replacing each eliminated null production with other new production(s).

The basic idea is that the new production(s) we are adding to replace a null production  $N \rightarrow \epsilon$  come from other productions where  $N$  appears on the right.

If such a production exists, then we add new productions that have all possible subsets of  $N$  deleted, for example:

$X \rightarrow aNb$  adds one production:  $X \rightarrow ab$

$X \rightarrow aNbNc$  adds three productions:  $X \rightarrow abNc \mid aNbC \mid abc$

If  $N$  appears  $k$  times in a production then add  $2^k - 1$  new productions (i.e., every possible combination to remove  $N$ ).

How do we remove all null productions from a grammar? One possible strategy is to remove each one at a time ... but that can be problematic.

- **Nullable variable:** The solution is to eliminate all the null productions at the same time, but that requires a new definition.

Given a context free grammar and a variable  $N$  in that grammar, we call  $N$  a nullable variable iff:

1. There is a production  $N \rightarrow \epsilon$  or
  2. There is a derivation that starts with  $N$  and leads to  $\epsilon$
- **Identify all nullable variables:** How do we identify all of the nullable variables in a context free grammar? By essentially taking a transitive closure.
    1. For every null production, "paint" the variable that appears on the left side of the production **"red"**. These are all nullable variables.

2. "Paint" every occurrence of every nullable variable "**red**" throughout the entire grammar, including occurrences that appear on the right side of productions.
3. "Paint" any variable that appears on the left side of a production "**red**" if the right side of the production is all "**red**". These are nullable variables.
4. Repeat steps 2 and 3 until you have painted nothing else "**red**".

- **Eliminate the null productions:** How do we eliminate all null productions from a grammar?

1. Identify all the nullable variables.
2. Remove all the null productions.
3. For every production that has a nullable variable in its right side, add new productions all with the same left side and new right sides with every possible subset of the nullable removed, but do not allow a new null production to be produced, even if the only symbol on the right side of the production is a nullable variable (i.e., do not add  $Y \rightarrow \epsilon$  even if there is production  $Y \rightarrow A$  and  $A$  was found to be nullable).

- **Eliminating null productions example:** Consider the following grammar

$$\begin{aligned}
S &\rightarrow XY \\
X &\rightarrow Zb \\
Y &\rightarrow bW \\
W &\rightarrow Z \\
Z &\rightarrow AB \\
A &\rightarrow aA|bA|\epsilon \\
B &\rightarrow Ba|Bb|\epsilon.
\end{aligned}$$

The nullables are  $W, Z, A, B$  and the two null productions are  $A \rightarrow \epsilon$  and  $B \rightarrow \epsilon$

We have identified the nullables. Now, we remove the null products  $A \rightarrow \epsilon, B \rightarrow \epsilon$

$$\begin{aligned}
S &\rightarrow XY \\
X &\rightarrow Zb \\
Y &\rightarrow bW \\
W &\rightarrow Z \\
Z &\rightarrow AB \\
A &\rightarrow aA|bA \\
B &\rightarrow Ba|Bb.
\end{aligned}$$

Finally, for every production that has a nullable variable in its right side, add new productions to the left side with every possible subset of the nullable removed.

$$\begin{aligned}
S &\rightarrow XY \\
X &\rightarrow Zb \mid b \\
Y &\rightarrow bW \\
W &\rightarrow Z \\
Z &\rightarrow AB \mid A \mid B \\
A &\rightarrow aA \mid bA \mid a \mid b \\
B &\rightarrow Ba \mid Bb \mid a \mid b.
\end{aligned}$$

- **Eliminate unit productions:** A production of the form *one variable*  $\rightarrow$  *one variable* is called a unit production.

Given a context free grammar with no null productions, it is possible to create a new context free grammar that accepts the same language and that has no unit productions.

For every pair of variables  $A$  and  $B$ , if the CFG has a unit production  $A \rightarrow B$  or if there is a derivation from  $A$  to  $B$ ,

$$A \Rightarrow X_1 \Rightarrow \dots \Rightarrow X_n \Rightarrow B$$

where each  $X_i$  is a single variable, then we introduce the following productions according to the following rule:

If all the non-unit productions from  $B$  are

$$B \rightarrow S_1 \mid S_2 \mid \dots \mid S_m$$

then add the productions

$$A \rightarrow S_1 \mid S_2 \mid \dots \mid S_m$$

Conclude by removing all unit productions from the CFG.

Consider the following grammar

$$S \rightarrow A \quad A \rightarrow a \mid B \mid b \mid CB \rightarrow b \mid C \mid AAa \quad C \rightarrow A.$$

Split the grammar into unit productions and non-unit productions as follows:

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow B \mid C \quad A \rightarrow a \mid b \\ B &\rightarrow C \quad B \rightarrow b \mid AAa \\ C &\rightarrow A. \end{aligned}$$

We must consider each unit production, one at a time, and make longer and longer derivation chains that end in a single variable without introducing any cycles (e.g., do not allow  $A \Rightarrow B \Rightarrow \dots \Rightarrow B$ )

New grammar with unit productions removed and new productions added

$$\begin{aligned} S &\rightarrow a|b|AAa \\ A &\rightarrow a|b|AAa \\ B &\rightarrow a|b|AAa \\ C &\rightarrow a|b|AAa. \end{aligned}$$

- **Almost CNF:** If  $L$  is a language generated by some CFG then there is another CFG that generates all the non null words in  $L$  all of whose productions are of one of two basic forms:

$$\text{Variable} \rightarrow \text{string of only variables} \text{ or } \text{Variable} \rightarrow \text{one terminal}$$

Start with a CFG having variables  $S, X_1, \dots, X_n$ .

For each terminal (e.g.,  $a, b, c$ ) add a new production by introducing a new variable:

$$A' \rightarrow a, \quad B' \rightarrow b, \quad C' \rightarrow c$$

In all of the *original* productions (i.e., only those that started with  $S, X_1, \dots, X_n$ ) replace the terminals (*not necessarily every terminal*) with the newly created associated variable.

We only rewrite productions with these new productions if the terminal in the production is not alone. For example  $A \rightarrow Bc$  turns to  $A \rightarrow BC'$ , but  $A \rightarrow c$  stays the same.

- **Chomsky Normal Form:** Going from Almost CNF to CNF is easy

For every production of the form  $A \rightarrow X_1X_2\dots X_n$  where  $n > 2$ :

- Introduce a new variable  $R$
- Add new production  $R \rightarrow X_2\dots X_n$ 
  - \* The new production has  $n - 1$  variables on the right side.
  - \* Because the original production had  $n \geq 2$  variables, this new production will have  $n \geq 2$  variables (i.e., cannot introduce unit productions) on the right side.
- Re-write the original production  $A \rightarrow X_1X_2\dots X_n$  as  $A \rightarrow X_1R$

Repeat Step 1 as necessary, including applying it to any new productions that were generated by Step 1.

**Example:**

$$A \rightarrow BCDE.$$

Becomes

$$\begin{aligned} A &\rightarrow BR_1 \\ R_1 &\rightarrow CR_2 \\ R_2 &\rightarrow DE. \end{aligned}$$

- **Note about CNF:** A CFG converted to CNF will accept the exact same language *except* the empty string  $\epsilon$ . If the original grammar accepted the empty string, the CNF grammar will not. If the original grammar did not accept the empty string, then the grammar in CNF will accept the exact same language.

## 1.5 Equivalence of Context-Free Grammars and Pushdown Automata

- **Recall:** A language is generated by a context free grammar if and only if it is accepted by a pushdown automaton.
- **Equivalence theorem:** The set of languages that can be defined by context free grammars - what we call the set of context free languages - is exactly the same set of languages that can be defined by pushdown automata.

**Claim:** There is no language that can be defined by a CFG for which there is no PDA that accepts the same language.

There is no language that can be defined by a PDA for which there is no CFG that accepts the same language.

- **Equivalence theorem proof:** The proof is in two parts and is similar in structure to the way we proved Kleene's Theorem - by constructive algorithm.
  1. Given any CFG that accepts some language  $L$ , then there is an algorithm to convert the CFG to a PDA that accepts the same language  $L$ .
  2. Given any PDA that accepts some language  $L$ , then there is an algorithm to convert the PDA to a CFG that accepts the same language  $L$ .
- **Converting CFG to PDA:** We can convert any CFG to another CFG that is in Chomsky Normal Form (CNF) that accepts the same language, most of the time; there is one noted exception (the empty string  $\epsilon$ ).

Recall that CNF requires that every production be in one of following two forms:  
*Variable*  $\rightarrow$  *exactly 2 variables* or *Variable*  $\rightarrow$  *one terminal*

Consider a CFG that defines language  $L_o$  and its conversion to another CFG in CNF that defines language  $L_c$ ,

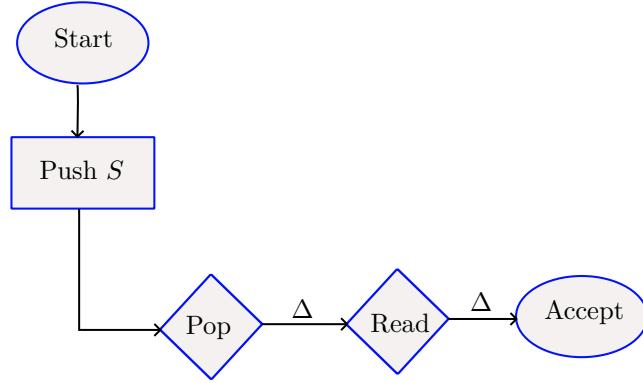
If  $\epsilon \in L_o$  then  $L_c = L_o - \{\epsilon\}$ , otherwise  $L_c = L_o$ .

It is a simple matter to determine if the empty string  $\Lambda$  is in the original language;

$\epsilon \in L_o$  if and only if the start symbol  $S$  in the original CFG is nullable.

To convert any CFG to a PDA that accepts the same language we start by converting the CFG to CNF and taking note if we lost the empty string  $\epsilon$  in the conversion.

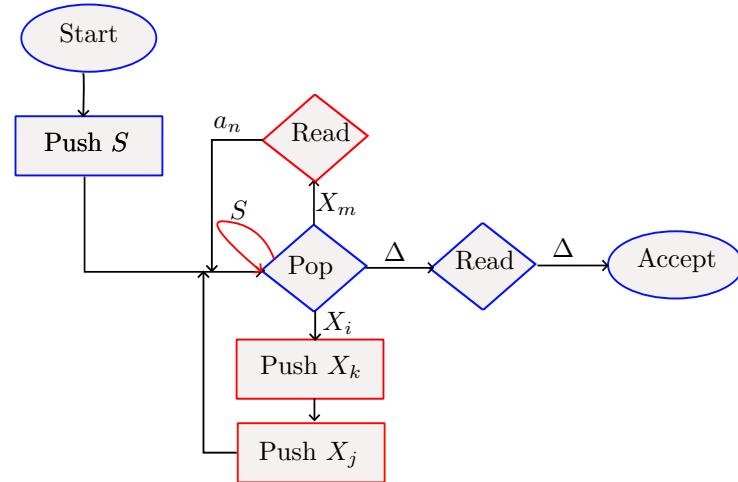
We then start building our PDA, which always starts with the same five states (i.e., no matter what the CFG).



For every production in the CNF CFG of the form  $X_m \rightarrow a_n$  we add a transition from the POP state and a READ state.

For every production in the CNF CFG of the form  $X_i \rightarrow X_j X_k$  we add a transition from the POP state and two PUSH states, first PUSH  $X_k$  and then PUSH  $X_j$ .

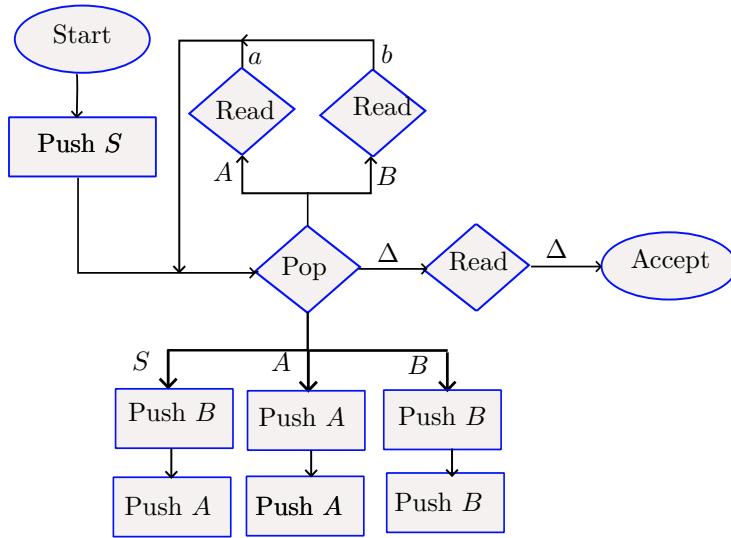
If the language defined by the original CFG accepted the empty string  $L$ , then we add one more transition labeled  $S$  from the POP state back to the POP state.



- **Example: Convert CNF CFG to PDA:** Consider the following CFG in CNF that accepts  $a^n b^m$  for  $n, m > 0$ .

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow AA \mid a \\ B &\rightarrow BB \mid b. \end{aligned}$$

Which converts to the PDA



We start building our PDA with the required five states.

We add the READ states for the two productions that have terminals on their right side.

We add the PUSH states for the three productions that have two variables on their right side.

Since the empty string  $\epsilon$  is not in this language we do not have to add one more transition labeled  $S$  from the POP state back to the POP state, so we are done.

- **Conversion between PDA and CFG:** Given an arbitrary PDA there exists an algorithm to convert it to a CFG that accepts the same language.

The algorithm is complex, and so, we will take it on faith that it exists and that it is correct.

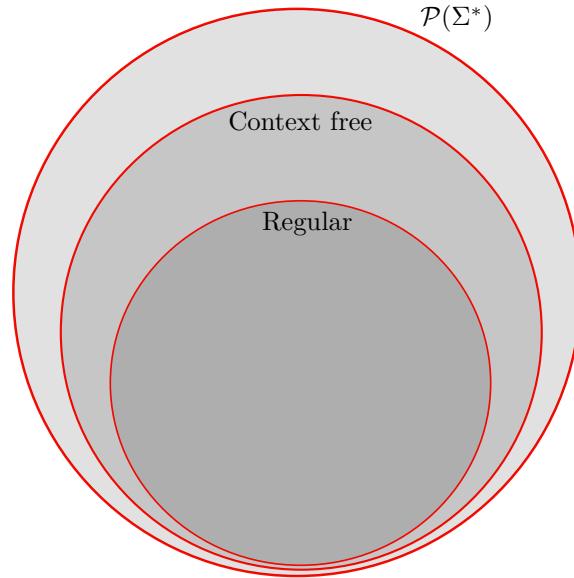
- **Equivalence theorem:** We accept the theorem

A language is generated by a context free grammar if and only if it is accepted by a pushdown automaton.

- **Current state of  $\mathcal{P}(\Sigma^*)$ :** Regular languages are accepted by regular expressions and finite automata.

Context free languages are accepted by context free grammars and pushdown automata.

Don't forget ... CFGs and PDAs can also accept all regular languages.



Where the regular languages are accepted by regular expressions and finite automata, and the context free languages are accepted by context free grammars and pushdown automata

## 1.6 Non context free languages

- **Intro:** We know that set of regular languages has all the finite languages,  $\Sigma^*$ ,  $\Phi$ , and infinitely many infinite languages.

We know that the set of context-free languages contains all the regular languages, some other languages that are not regular, and that all the context-free languages that are not regular languages are infinite languages.

But are there any non-context free languages? if there are any, we know they must all be infinite languages

But what would that mean?

A language is context free if and only if there exists a context-free grammar that accepts it, and

a language is generated by a context-free grammar if and only if it is accepted by a pushdown automaton.

If someone comes to us with a description of a language and asks us to develop a CFG or PDA that accepts the language, we work on it for a long time but fail to come with anything, what does that mean?

It may simply mean that the problem is too hard or that we are not clever enough ... it does not mean that no CFG or PDA exists.

To claim at a language is non-context free one must prove that no CFG or PDA exists (very different than trying for a time and failing).

- **Derivations From CNF CFGs:** Before we continue our conversation about the prospect of noncontext free languages we first look at derivations from context free grammars that are in Chomsky Normal Form (CNF).

Consider the derivations that come from a CFG in CNF

Every derivation starts with the grammar's start symbol (a variable)  $S$  and applies one production at each stage generating a working string until the final stage when the word  $w \in L$  defined by the CFG is produced.

Every application of a live production has the net effect of adding a variable to the working string.

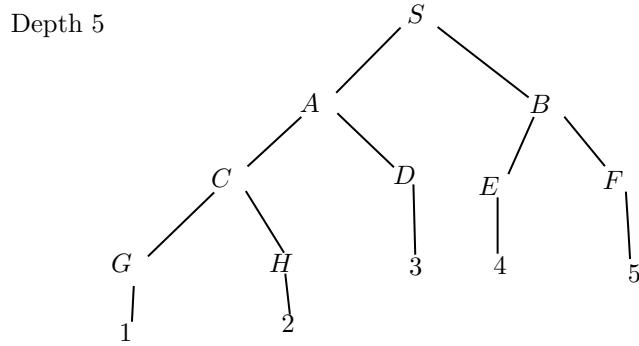
Every application of a dead production has the net effect of replacing a variable (i.e., removing a variable) with a terminal in the working string

Any derivation of a non-empty word  $w \in L$  defined by the CFG in CNF whose length  $|w| = n > 0$  must therefore always have exactly  $2n - 1$  steps, or applications of productions, as follows:

- $n - 1$  applications of *live productions* to convert the single variable (i.e., start symbol)  $S$  to the  $n$  variables needed for the subsequent *dead productions* and
- $n$  applications of *dead productions*; one to convert each variable to a single terminal.

We now turn our attention to derivation trees

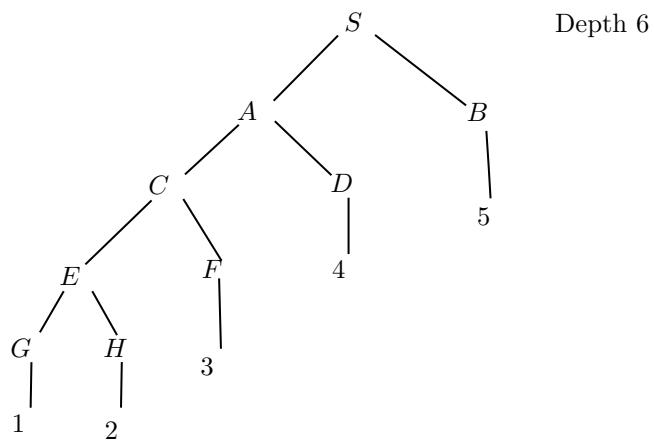
Consider a finite language with exactly one word  $L = \{12345\}$ , the following two CNF CFGs each accepting  $L$ , and the derivation trees using each grammar.



Using

$$\begin{aligned}
 S &\rightarrow AB \\
 A &\rightarrow CD \\
 B &\rightarrow EF \\
 C &\rightarrow GH \\
 D &\rightarrow 3 \\
 E &\rightarrow 4 \\
 F &\rightarrow 5 \\
 G &\rightarrow 1 \\
 H &\rightarrow 2.
 \end{aligned}$$

And



Using

$$\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow CD \\
B &\rightarrow 5 \\
C &\rightarrow EF \\
D &\rightarrow 4 \\
E &\rightarrow GH \\
F &\rightarrow 3 \\
G &\rightarrow 1 \\
H &\rightarrow 2.
\end{aligned}$$

We first note that both derivation trees have

- The expected number of live productions ( $n - 1 = 5 - 1 = 4$ ) and
- The expected number of dead productions ( $n = 5$ )

But we also note

- The shapes of the trees are different
- The depths of the trees are different

But what is the shallowest depth a derivation tree can be to produce a word with length  $n$  from a CFG in CNF?

In a derivation tree from a CFG in CNF,

- **Live productions:** The *branching factor* = 2, children are internal nodes
- **Dead productions:** The *branching factor* = 1, children are leaf nodes

The shallowest tree is one that first (i.e., from the root down) takes the fullest advantage of the live production's higher branching factor by pushing all the live productions to the top of the tree and pushing all the dead productions to the bottom.

So what is the shallowest depth a derivation tree can be to produce a word with length  $k$  from a CFG in CNF? The answer is

$$\begin{aligned}
\lceil \log_2(k) + 2 \rceil &= d \\
\implies 2^{d-2} &= k.
\end{aligned}$$

Where  $d$  is the shallowest depth, with at least one path from root to leaf with  $(\lceil \log_2(k) \rceil + 2) - 1 = \lceil \log_2(k) \rceil + 1$  variables.

- **The Pumping Lemma for Context Free Languages:** Let  $L$  be any context-free language. Then there is a constant  $n$  that depends on  $L$  such that if  $z \in L$  and  $|z| \geq n$ , then we may re-write  $z = uvwxy$  such that
  1.  $|vwx| \leq n$ ,
  2.  $|vx| > 0$ , and
  3. for all  $i \geq 0$ ,  $uv^iwx^i y \in L$ .
- **The pumping lemma for context free languages proof:** Let  $G$  be any context-free grammar that accepts a context-free language  $L$ .

Let  $G'$  be the CNF CFG that accepts  $L - \{\Lambda\}$  having  $m$  variables.

If  $L$  is an infinite language, we can always find a word  $z \in L$  that is longer than some arbitrary length, say the Pumping Lemma's constant  $n$ .

Set  $n = 2^m$  and then select a word  $z \in L$  such that  $|z| \geq n$  (i.e.,  $|z| \geq 2^m$ ).

Consider the derivation tree of  $z$  using the CNF CFG  $G'$ .

Because  $|z| \geq 2^m$ , we know that there is at least one path from root to leaf with at least  $\lfloor \log_2 2^m \rfloor + 1$ , or at least  $m + 1$ , variables.

Because the CNF CFG  $G'$  has only  $m$  variables, we know that same path must repeat at least one variable at least once.

We traverse that path from the leaf node up to the root noting the lowest and second lowest occurrence of the first repeated variable.

Derivation trees for words  $w \in L$  from a CNF CFG have a minimum depth that is based on  $|w|$ , and from that, we also know how many variables there must be in at least one path from the root to a leaf (i.e., terminal).

If you pick a  $w \in L$  with  $|w| \geq 2^m$ , where  $m$  is the number of variables in the CNF CFG, then you are guaranteed that the derivation tree will have a path from root to terminal in which a variable is repeated ... *the repeated variable is our "cycle"*, and

We can use that cycle to generate an infinite number of words that all must also be in the language  $L$ .

- **The Real Value of the Pumping Lemma for Context Free Languages:** The real value of the Pumping Lemma for CFLs is to prove that some infinite language is not context free ... here is the basic strategy - it is a proof by contradiction;

You start with some infinite language that you suspect may be non context free.

You assume that the language is a context free language.

This is a crucial step in a proof by contradiction. You assume something, run the proof from that assumption, come to a contradiction (something that cannot possibly be true), and then that proves that your assumption must be false.

You show that it is not possible to decompose a string into  $uvwxy$  such that ... and so on (i.e., what The Pumping Lemma for CFLs assures us is always possible to do with infinite CFLs).

This gives us our contradiction; we assume the language is a CFL, which means the Pumping Lemma for CFLs is in effect, which means we can always decompose a string into  $uvwxy$  such that ... etc., but we find we cannot decompose the string into  $uvwxy$ , so contradiction.

We conclude that our assumption that the original language is a CFL must be false, and therefore, it must be a non context free language.

- **Pumping lemma example:** Suppose  $L = a^k b^k c^k \forall n \geq 0$

**Proof.** Suppose  $L$  is context free, then  $\exists n \in \mathbb{R} \mid \forall z \in L, |z| \geq n \ z = uvwxy$  for  $u, v, w, x, y \in \Sigma^*$ , with  $|vx| > 0$ ,  $|vwy| \leq n$  and  $z' = uv^iwx^iy \in L \forall i \geq 0$ .

Let  $k = n + 1$ , then  $z = a^{n+1}b^{n+1}c^{n+1}$ . Since the middle portion of the decomposed string  $z = uvwxy$ ,  $vwx$  must be weakly less than  $n$  in length, we have a couple cases for  $v, x$ . The first case is when  $v$  and  $x$  are all a's, all b's, or all c's. In this case, pumping copies of  $v$  and  $x$  leads to an imbalance in one of the symbols. The second case is that  $v$  and  $x$  are homogenous in different symbols, for example  $v$  is all a's, and  $x$  is all b's. In this case, pumping would lead to an imbalance in two of the symbols. The final case is that  $v$  or  $x$  span some boundary. For example,  $v = a^r b^\ell$ . In this case, pumping would lead to more than one boundary, which leads strings not in  $L$ . Thus, there are no choices for  $v, x$  such that  $z' = uv^iwx^iy \in L \forall i \geq 0$ . Since we have a contradiction,  $L$  must not be context free.

- **PL example two:** Show that the language  $L = a^k b^{k+1} a^k \forall n > 0$  is not context free.

**Proof:** Assume  $L$  is context free, then  $\exists n \in \mathbb{Z}^+ \mid \forall z \in L, |z| \geq n \ z = uvwxy$  for  $u, v, w, x, y \in \Sigma^*$ , with  $|vx| > 0$ ,  $|vwy| \leq n$  and  $z' = uv^iwx^iy \in L \forall i \geq 0$ .

Let  $z = a^{n+1}b^{2(n+1)}a^{n+1}$ , for  $k = n+1$ . If  $v, x$  are homogenous substrings, for example  $v = a^r$ ,  $x = b^\ell$  for  $0 \leq r < n+1$ ,  $0 < \ell < n+1-r$ , for all combinations of pairs of  $a, b, a$ . In this case, pumping would lead to one of the symbol sequences unchanged. For example, suppose  $v = a^r$ ,  $x = b^\ell$ . Then,  $v = a^{n+1-r}a^r b^\ell b^{2(n+1)}a^{n+1}$ . Then  $z' = a^{n+1-r+i}b^{2(n+1)-\ell+i}a^{n+1} \in L \forall i \geq 0$ . Let  $i = 2$ , then  $z' = a^{n+1+r}b^{2(n+1)+\ell}a^{n+1}$ , we see that the  $a$  and  $b$  at the beginning grow, while the  $a$  at the end remains unchanged.

The other case is that either  $v$  or  $x$  is non-homogenous, and span across some boundary. For example,  $v = a^r b^\ell$ , and  $x$  is some number of  $b$ 's. In this case, pumping would lead to additional  $ab$  boundaries, which leads to strings not in the language. Since there are no choices for  $v, x$  that satisfies the property of context free languages, we have a contradiction and the language must not be context free. ■

- **PL example three:** Consider  $L = \{ww : w \in (0+1)^*\}$ . In other words, the left half of the string must be the same as the right half. For example,  $z = 00110011 \in L$ .

**Proof:** Assume  $L$  is context free, then  $\exists n \in \mathbb{Z}^+ \mid \forall z \in L, |z| \geq n \ z = uvwxy$  for  $u, v, w, x, y \in \Sigma^*$ , with  $|vx| > 0$ ,  $|vwy| \leq n$  and  $z' = uv^iwx^iy \in L \forall i \geq 0$ .

Consider  $z = 0^n 1^n 0^n 1^n$ . Since  $|vwx| \leq n$ , the only hope of being able to satisfy the criterion is when  $v, x$  span the middle of the string. If  $v, x$  were contained entirely in the left or right half, then its clear that pumping would lead to a string not in  $L$ . Suppose then that  $v, x$  are contained within the middle portion such that  $v$  is  $1^r$ , and  $x = 0^\ell$ , and their lengths satisfy  $|uwx| \leq n$ . Then,  $z = 0^n 1^{n-r} 1^r 0^\ell 0^{n-\ell} 1^n$ , and  $z' = 0^n 1^{n-r+i} 0^{n-\ell+i} 1^n$ , let  $i = 0$ , then  $z' = 0^n 1^{n-r} 0^{n-\ell} 1^n \notin L$ . We note that for any  $i \geq 0$ ,  $z' \notin L$ . To find a  $v, x$  that satisfies the pumping lemma, we would need to be able to select a symbol in the first half as  $v$ , and then reach the same symbol in the second half for  $x$ , which is not possible given the condition  $|vwx| \leq n$ , since the lengths of each symbol are  $n$  characters long. The best we could do is reach the opposite (first) symbol in the second half for  $x$ , with  $v$  being the second symbol in the first half.

Thus, we have a contradiction and the language must not be context free.

- **PL example four:** Suppose  $L = \{a^p : p \in \mathbb{P}\}$ . Choose  $z = a^\ell$  for  $\ell \geq n$ . Then  $v = a^r$ ,  $x = a^s$  for  $0 < r+s \leq n$ , which implies  $z = a^r a^s a^{\ell-r-s}$ , and  $z' = a^{ir} a^{is} a^{\ell-r-s}$ . Let  $i = \ell + 1$ , then  $z' = a^{\ell-r-s+r(\ell+1)+s(\ell+1)} = a^{\ell+r\ell+s\ell} = a^{\ell(1+r+s)}$ . Since  $\ell \in \mathbb{P}$ , and  $r+s > 0$ , we know  $p(1+r+s) \notin \mathbb{P}$ . Thus, we have a contradiction and  $L$  must not be context free.

## 1.7 Turing machines

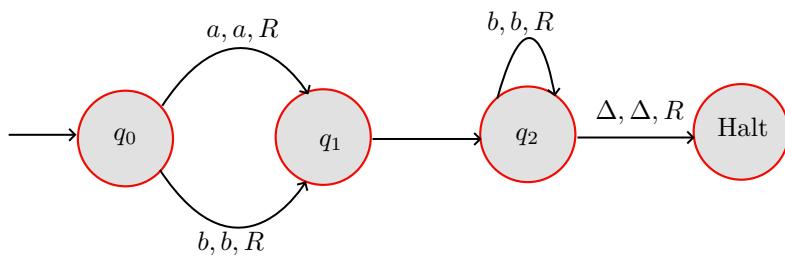
- **Intro:** In 1930's Alan Turing presented an abstract machine called the Turing Machine which serves to this day as the model for all computation (i.e., algorithms) and computing (i.e., computers)
- **TM General Description:** Turing Machines (TM) are similar to deterministic finite automata (DFA) in that a TM has:
  - a tape that is infinite in one direction and a tape head
  - a finite number of states with exactly one that is designated as the start state, and
  - A set of transitions (i.e., directed edges) that deterministically take the TM from one state to the next based on what the tape head reads from the tape
  - Zero or more HALT state(s) that stops the TM and accepts the input string,
  - no requirement that every state must have an outgoing edge for every symbol (i.e., requirement of DFAs), with the same understanding that if the TM is in a state from which there is no edge for the symbol pointed at by the tape head, then the TM crashes and rejects the input string, and
  - no requirement that the entire input string must be read before accepting or rejecting the string.

Turing Machines (TM) differ from FA and PDA in that a TM does the following ordered sequence on each transition:

1. Read the symbol from the tape (same as FA and PDA).
2. Write a symbol to the tape at the same location (may be the same symbol that was read).
3. Move the tape head one slot either left or right with the understanding that an attempt to move left from the leftmost part of the tape crashes the TM (i.e., halts execution and rejects the input string).

Accordingly, each transition in a TM is annotated with the three above

- **Turing machine example:**



In this example the TM

- always wrote the same symbol that it read
- always moved the tape head to the right

That is not always the case with a TM.

- **Another TM example:** Consider the language  $L = a^n b^n \forall n > 0$ . We know that  $L$  is not a regular language and that it is a context free language, so there exists a pushdown automaton (PDA) that accepts  $L$ .

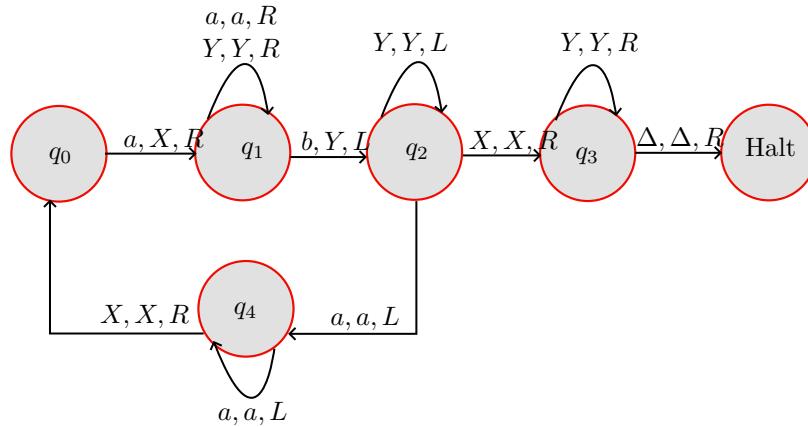
A TM has no stack, but it has a tape head that can write and can move in either direction. What, then, is the strategy in designing a TM that accepts  $L$ ?

As you read each  $a$

- Cross the  $a$  off as an "already read"  $a$  (e.g., replace it with an  $X$ ).
- Move the tape head to the right until we encounter our first  $b$ .
- Cross that  $b$  off as an "already read"  $b$  (e.g., replace it with a  $Y$ ).
- Move the tape head back to the left until we encounter the  $X$  we had just written, then move the tape head one space to the right

At this point the tape head should point either to the next  $a$  or to the first  $Y$  that we wrote.

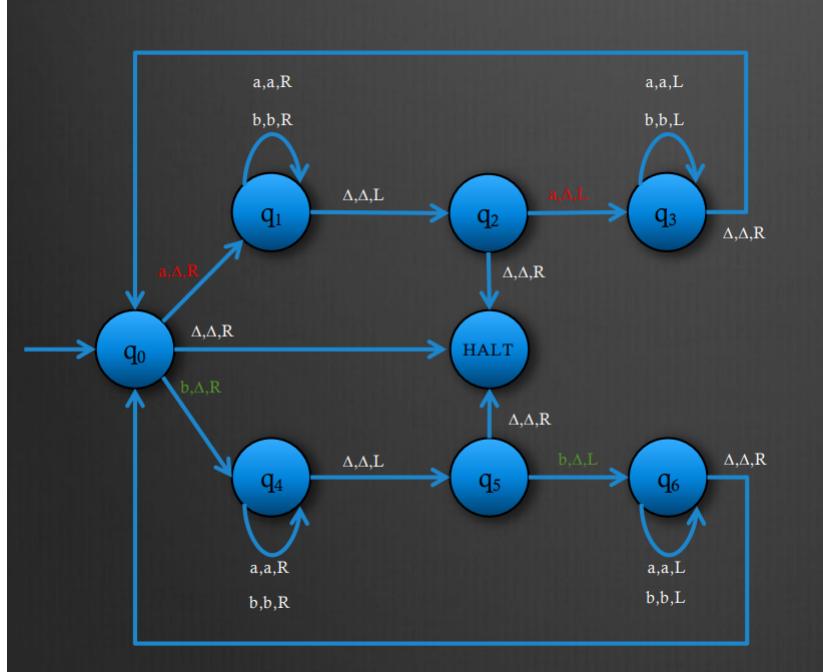
- If the tape head points to the next  $a$ , then simply process the steps above again.
- If the tape head points to the first  $Y$ , then move the tape head to the right making sure that it passes only over  $Y$ 's until it reaches a blank  $\Delta$  (to assure there were not more  $b$ 's than  $a$ 's).



- **Palindrome:** We know that Palindrome is not a regular language and that it is a context free language, so there is a PDA that accepts it. The basic strategy in designing a PDA that accepts Palindrome relies on the PDA's non-determinism.
  - In the first portion of the PDA; read and push the first half of the string onto the stack.
  - Non-deterministically transition to the second portion of the PDA by either;
    - \* reading a single symbol off the tape without pushing it onto the stack (for odd-length palindromes) or
    - \* make a  $\epsilon$ -move (for even-length palindromes).
  - In the second portion of the PDA; read the second half of the string and pop the stack for each symbol read to make sure that the two symbols (read and popped) match.
  - When you reach the end of the tape make sure that there are no more symbols in the stack.

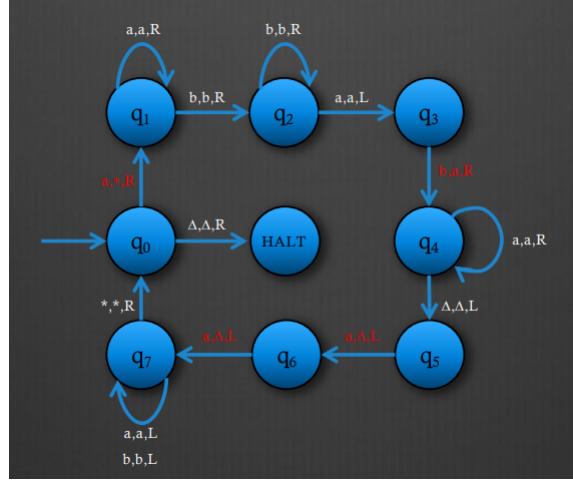
A TM has no stack and it is deterministic. What, then, is your strategy in designing a TM that accepts Palindrome ?

- Process the string by reading and erasing characters from the ends taking care to make sure that the characters at each end of the string always match.
- Take care to account for both even- and odd-length palindromes.



- **Turing machine example:** Consider the language  $L = a^n b^n a^n \forall n \geq 0$ . We know  $L$  is not context free
  - Start by marking the first  $a$  as read with a \*
  - Move tape head to  $ba$  transition and backup to  $b$
  - Replace that  $b$  with an  $a$  as read with a \*

- Move tape head to end of input and backup to  $a$
- Replace last two  $a$ 's with blanks as read with  $a *$
- Back tape head to  $*$  and advance to  $a$
- Repeat



- **Formal Definition TM:** Formally, a Turing Machine (TM) is denoted

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \Delta, F).$$

Where

- $Q$  is a finite set of states
- $\Gamma$  is a finite set of tape symbols
- $\Delta \in \Gamma$  is the blank
- $\Sigma \subset \Gamma$  such that  $\delta \notin \Sigma$ , is a set of input symbols
- $q_0 \in Q$  is the start state
- $\delta$  is the transition function that maps
  - \* a state in  $Q$  and a (read) tape symbol in  $\Gamma$  to
  - \* some (written) tape symbol in  $\Gamma$ , and a tape head move direction (L or R), and some state in  $Q$
  - \*  $\delta : Q \times \Gamma \rightarrow \Gamma \times \{L, R\} \times Q$

**Note:**  $\delta$  is a partial function in that it may be undefined for some values in  $Q \times \Gamma$

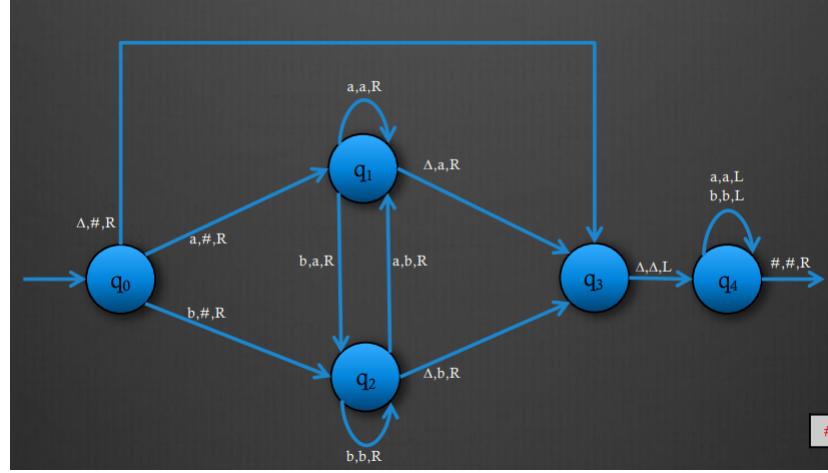
- $F \subseteq Q$  is the (possibly empty) set of HALT state(s)

- **Prepend #:** With a tape that is infinite in only one direction it is sometimes helpful to be able to shift the entire input string to the right one cell and then place in the leftmost cell of the tape a special symbol (e.g.,  $\#$ ) with the understanding that the special symbol will never appear anywhere else on the tape

In this way the rest of the TM can safely move the tape head left always taking care to not move left from  $\#$ .

There is a TM that can be used at the beginning of any TM - it shifts the entire input string, prepends  $\#$ , and positions the tape head at the first symbol of the original input string.

This TM is written for the input alphabet  $\Sigma = \{a, b\}$  but it can be easily modified to work with any input alphabet.



- **Turing machines as transducers:** Thus far we have considered the use of Turing Machines as language acceptors.

Because Turing Machines can write to the tape, we can also use them to perform calculations that transform value(s) that are initially written on the tape (i.e., input) to some other value (i.e., output) on the tape (e.g., the TM implements some function  $f(x)$ ).

- We write the input variables onto the tape, and the TM must reach a HALT state (so the input string is in the language defined by the TM), but what is left on the tape is the output of the function.

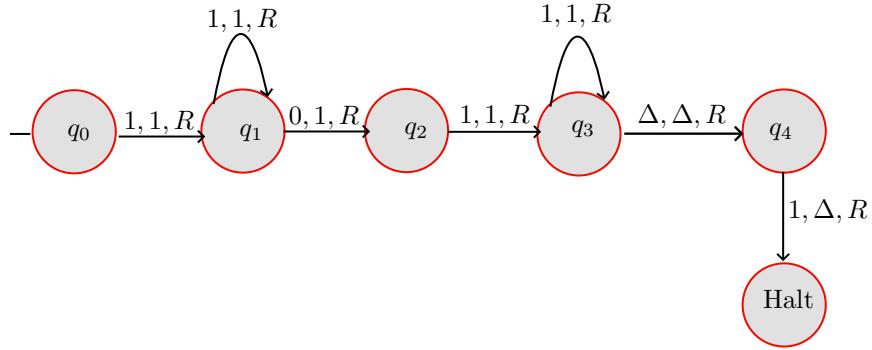
When we use TMs in this way they act as a language acceptor (i.e., the set of valid inputs) but because the output is also valuable we say the TM is a transducer

- **TM transducer example:** Consider the language  $L = 1^m 0 1^n \forall m, n > 0$

We can create a TM that accepts  $L$ , but can we also create a TM that leaves on the tape (i.e., as output) a string of the form  $1^{m+n}?$

In other words, can we create a TM that computes

$$f(m, n) = m + n.$$

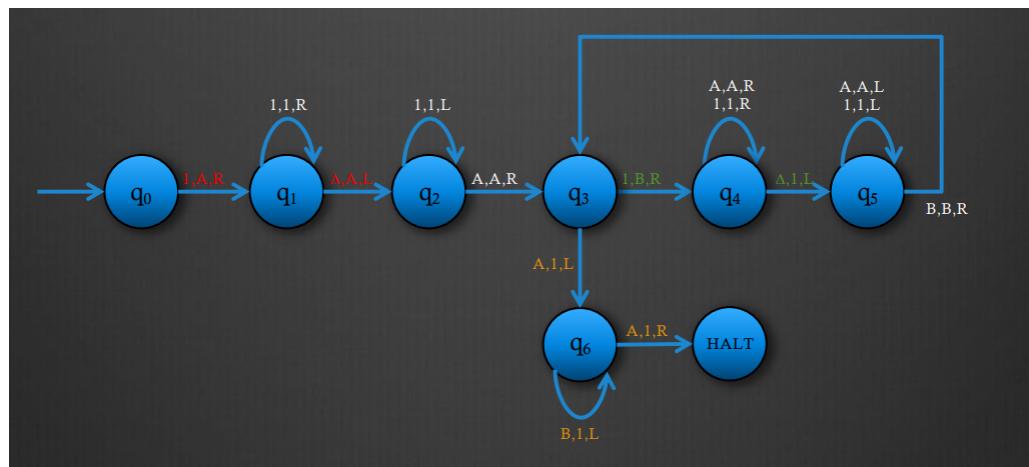


- **Another TM Transducer :** Consider the regular language  $L = 1^n$  for  $n > 0$

Can we create a TM that accepts  $L$  and leaves on the tape a string of the form  $1^{2n}$

In other words, can we create a TM that computes  $f(n) = 2n$ ?

- Start by marking the first 1 as read with a  $A$
- Move tape head to first  $D$ , replace with  $A$
- Back tape head to first  $A$ , then move  $R$  one
- Now enter loop replacing each original 1 with  $B$  and appending a 1 for each, always backing up tape to last  $B$  and then moving  $R$  one
- Eventually we will have processed the last original 1 which will position the tape head on the second  $A$
- All that remains is to move the tape left replacing all the  $A$ 's and  $B$ 's with 1's taking care to stop and move  $R$  when we reach the first  $A$ .



- **Final TM Transducer:**  $f(m, n) = mn$

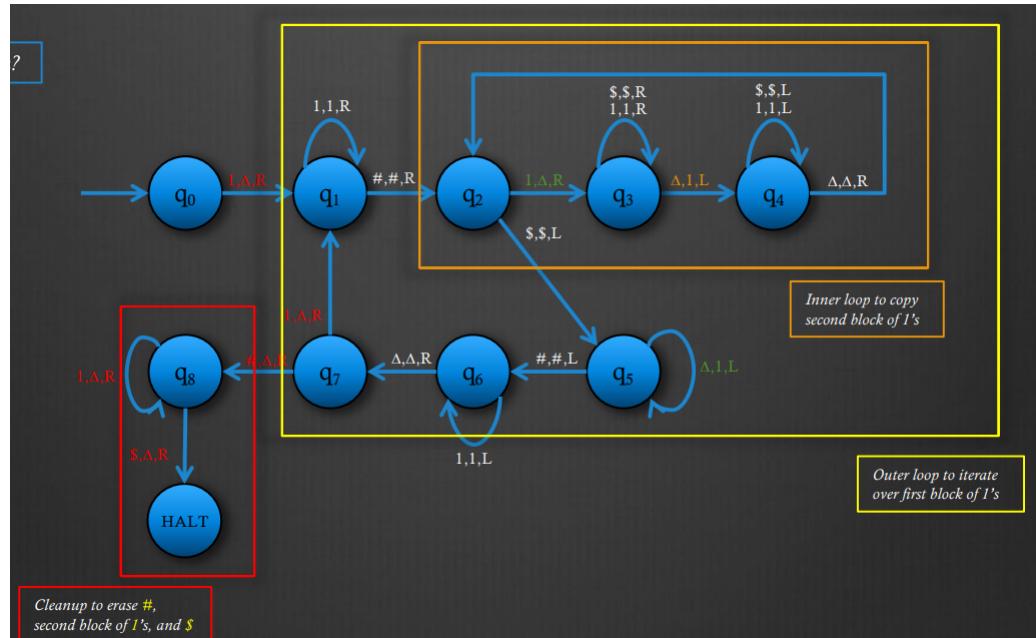
In the previous two examples the output was written left-justified on the tape

What if we temporarily relaxed that restriction by saying that the output may appear anywhere on the tape as long as it is contiguous (i.e., output may be preceded on tape by an arbitrary number of blank cells).

Consider the regular language  $L = 1^m \# 1^n \$$  for  $m, n > 0$

Can we create a TM that accepts  $L$  and leaves somewhere on the tape a string of the form  $1^{mn}$  to compute  $f(m, n) = mn$

1. Start by marking the first 1 as "read" by erasing it and advancing the tape head to the first 1 after the #.
2. Erase the first 1, append a  $\Delta$ , return the tape head to the  $\Delta$  you just wrote, and then move right.
3. Repeat until you have erased and appended a copy of the entire second block of 1's; this will leave the tape head at the \$.
4. Return the tape head to the first  $\Delta$  we wrote and then move right, taking care to restore the second block of 1's.
5. Repeat that process until you have appended a copy of the second block for the last 1 in the first block; this will leave tape at #.
6. Finish by advancing the tape head to \$ and then one more cell right, erasing #, the second block of 1's, and \$.



### 1.7.1 Decidability and Languages Accepted by Turing Machines

- **Membership in Languages:** Given a language and an acceptor for that language (e.g., FA or PDA) one could ask the question... "Is a given string  $w \in \Sigma^*$  in (or not in) the language?"

Or to put the question in different ways

1. For any  $w \in \Sigma^*$  will the acceptor stop and determine whether or not  $w$  is in the language defined by the acceptor?
2. Does the acceptor always stop and partition  $\Sigma^*$  into strings that are in the language and strings that are not in the language? For regular languages the answer is yes. We can see this from the definition of deterministic finite automata (DFA).

In this way for any string  $w \in \Sigma^*$  the DFA always stops and reports whether (or not) the  $w$  is in the language.

In this way for any string  $w \in \Sigma^*$  the DFA always stops and reports whether (or not) the  $w$  is in the language.

For context free languages the answer is also yes. We can see this from the definition of context free grammars (CFG) that are in Chomsky Normal Form (CNF).

Recall that any derivation of a non-empty word  $w \in L$  defined by a CFG in CNF whose length  $|w| = n > 0$  must always have exactly  $2n - 1$  steps ( $n - 1$  live productions plus  $n$  dead productions). Since any CFG has only finitely many productions, then for any string  $w \in \Sigma^*$  with length  $|w| = n > 0$ , there are only finitely many derivations having exactly  $2n - 1$  steps ... simply check them all to determine if any one derivation produces  $w$ .

In this way the CNF CFG always stops and reports whether (or not) the  $w$  is in the language.

In other words, the CNF CFG partitions  $\Sigma^*$  into strings that are in the language and strings that are not in the language.

- **Decidability:** So, for regular and context-free languages we can take any string  $w \in \Sigma^*$  and determine whether or not  $w$  is in the language.

In formal terms, we say that the question of membership for regular and context free languages is *decidable*.

A question whose answer is boolean (i.e., yes or no) is decidable if there exists an effective method (or effective procedure or algorithm) that can determine the answer.

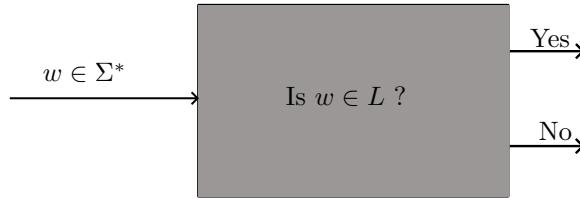
An effective method is one that always halts and produces the correct answer

A question whose answer is boolean and for which there is no algorithm (i.e., stops on all input and correctly answers "yes" or "no") is *undecidable*.

We can, therefore, say that the question of membership for regular and context free languages is decidable because we have an effective procedure that answers the question for both classes of languages.

- **Regular languages:** Use the DFA which always halts and gives correct answer.
- **Context free languages:** Use the CNF CFG which has only finitely many derivations that could possibly produce a given string, simply check them all to see if one does.

It is sometimes convenient to represent a decidable question by depicting its decision procedure (or algorithm) as a "black box".



Note that in this representation the "black box" is not the language acceptor automaton (i.e., it is not a DFA or PDA)

For the decidable question, "Is  $w \in L$ ?"

- **For Regular languages:** The "black box" represents the algorithm that uses the DFA which always halts and gives correct answer.
- **For Context free languages:** The "black box" represents the algorithm that uses the CNF CFG by checking all of the finitely many derivations that have exactly  $2n - 1$  productions.

As we can see by the definitions of decidable/undecidable, the question of whether or not algorithm exists (i.e., a decision procedure that always stops and correctly answers "yes" or "no") is not limited to questions about formal languages (e.g., is membership in regular languages or CFLs decidable).

The question of decidability (i.e., whether an algorithm exists) can be, and is, applied to many domains.

Questions of decidability are at the core of all computability (i.e., does an algorithm exist?).

- **Euclid's conjecture:** Sometimes we answer a "yes/no" question by providing a proof, which answers the question entirely, and therefore, we do not need an algorithm

For example, there is Euclid's Conjecture which states:

*There are infinitely many prime numbers.*

**Proof (by contradiction).** Assume there is a finite list of all the prime numbers  $p_1, p_2, \dots, p_r$

Consider  $m = p_1 \cdot p_2 \cdot \dots \cdot p_r + 1$

We observe that  $m$  cannot be any of the numbers in our original list  $p_1, p_2, \dots, p_r$  but that list was supposed to be all the primes, so contradiction.

If  $m$  is not prime, then it must be evenly divisible by some prime, call it  $p$

We observe that  $p$  cannot be our original list  $p_1, p_2, \dots, p_r$ , but that list was supposed to be all the primes, so contradiction.

- **Is a Number Prime?:** Other times we answer a "yes/no" question by providing an algorithm.

*Given some integer  $n > 1$ , is  $n$  prime?*

Algorithm:

1. Divide  $n$  by every integer  $i$  in the range  $2 \leq i \leq \sqrt{n}$
  2. If any  $i$  in that range evenly divides  $n$  then  $n$  is not prime; otherwise  $n$  is prime
- **Goldbach's Conjecture:** Sometimes we encounter a question for which there is no proof and there is no algorithm (i.e., always halts with "yes/no" answer).

One of the oldest and best-known unsolved problems in number theory was posed on June 7, 1742 by the German mathematician Christian Goldbach in his letter to Leonhard Euler. It states:

*Every even integer greater than two can be expressed as the sum of two primes.*

1690-1764 One approach (not an algorithm):

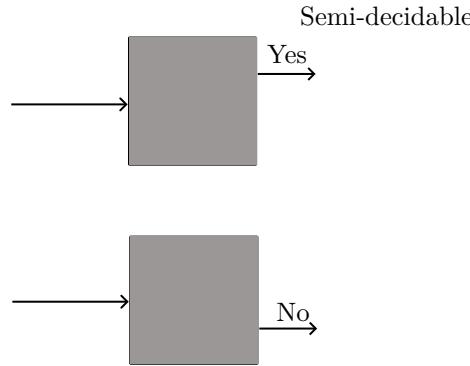
1. Test every even number that is greater than two.
2. For each number try to find two primes whose sum is equal to that number
3. If we find two such primes, then proceed to the next even number; otherwise halt and declare Goldbach's Conjecture false.

If Goldbach's Conjecture is false, then the proposed approach will eventually HALT and provide the answer "no"; otherwise it will run forever

We say that Goldbach's Conjecture is undecidable.

- **Semi-decidable:** While it is true that Goldbach's Conjecture is undecidable (i.e., because there is no algorithm that always halts with "yes/no" answer), the fact that we have a method that always halts and gives one of the "yes/no" answers gives rise to a new definition.

A question whose answer is boolean (i.e., yes or no) is *semi-decidable* if there exists a method (or algorithm) that always halts when the answer is yes or no, but may run forever for the other answer



- **Summary:** Combining our new definition semi-decidable with our previous definition of decidable/undecidable we have the following:
  - **Decidable:** There is an algorithm that always halts with correct "yes/no" answer.
  - **Undecidable:** Any yes/no question that is not decidable.
    - \* **Semi-decidable:** an undecidable question for which there exists an algorithm that always halts with one of the answers "yes" or "no".
- **What About TMs?:** We know that Turing Machines (TMs) can be used as language acceptors.

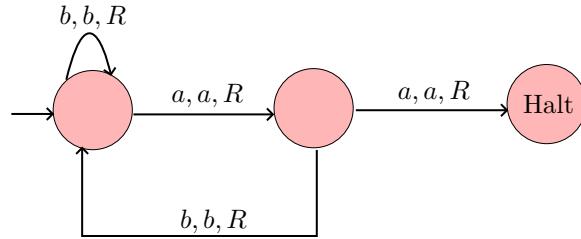
In fact, TMs can accept all regular and context free languages plus other languages that are not context free

But do TM's always partition  $\Sigma^*$  into strings that are in the language and strings that are not in the language?

In other words, is the question of membership in languages accepted by Turing Machines decidable?

The answer is not always

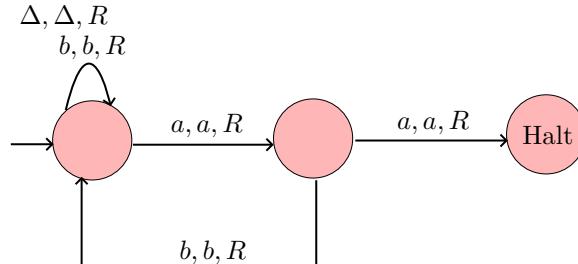
Consider the following TM that accepts all strings with a double a, (i.e., aa):



For all  $w \in \Sigma^*$  this TM:

- For all  $w \in L$ , HALT and accept
- For all  $w \notin L$ , crash and reject

Consider this slightly modified TM that also accepts all strings with a double a, (i.e., aa):



For all  $w \in \Sigma^*$  this TM:

- For all  $w \in L$ , HALT and accept
- For all  $w \notin L$ ,
  - \* sometimes crash and reject (e.g., no double a but ends in a)
  - \* sometimes loop forever (e.g., no double a, ends in b)

You may ask... "If there is an algorithm that always stops, then why not simply use that?".

Because there are some problems for which the only algorithm(s) cannot eliminate the possibility of looping

Every Turing Machine (TM) partitions  $\Sigma^*$  into three classes;

1. HALT and accept
2. crash (i.e., stop) and reject
3. loop forever

For any specific TM one or more of those three classes may be empty, meaning, for any specific TM

- Might always either HALT or crash; might always crash; might always accept.
- [general case]: Sometimes HALT and accept, sometimes crash and reject, or sometimes loop forever

- **A Closer Look at a General TM :** In general, a TM will process strings by;
  - $w \in L$ , always halt and accept any string that is in the language.
  - $w \notin L$ , sometimes halt by crashing, other times loop forever.

In the black box depiction, we might make the no line dashed

Recall, a decidable problem is one for which there exists an algorithm that always halts with the correct "yes/no" answer.

If there is a possibility that the algorithm may sometimes not halt, then that possibility alone renders the problem undecidable

This gives us the definition of two new languages accepted by Turing Machines

- **Languages defined by turing machines:** A language that is accepted by a Turing Machine is said to be a *recursively enumerable language*.

It is convenient to single out a subset of the recursively enumerable languages as follows:

A language that is accepted by at least one Turing Machine that halts on all inputs is said to be a recursive language.

Membership in recursive languages is decidable because, by the definition of recursive languages, there is a TM that always halts.

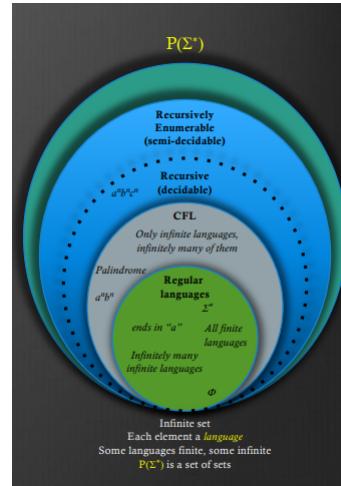
Membership in recursively enumerable languages is not decidable; it is semi-decidable. For recursively enumerable languages (that are not recursive) there are only Turing Machines that

- For any  $w \in L$  always HALT and accept
- But for a  $w \notin L$ , sometimes crash and reject, other times loop forever

The existence of recursively enumerable languages raises the unsettling point ...

If a TM is running for a long time, how do we know if the machine will eventually stop (HALT+accept or crash+reject) or loop forever?

We don't know... Worst yet, we can't know



### 1.7.2 TM Variations

- **Intro:** As we have seen, different classes of automata have different power in their ability to define languages
  - **Finite automata:** Regular languages
  - **Pushdown automata:** Context free languages
  - **Turing Machines:** Recursively enumerable languages

We have also experimented (and we can do other experiments) with different automata to see how such changes might impact the languages each accepts

For Finite automata

- Adding  $\epsilon$ -moves and/or non-determinism didn't make a difference
- Adding a stack makes a difference; equivalent to PDA

For Pushdown automata

- Removing non-determinism (i.e., forcing determinism) makes a difference; reduces the languages accepted
- Adding one or more stacks (i.e.,  $>1$  stack) makes a difference; equivalent to Turing Machine

- **Can we do better?** In the cases that we have studied for which modifications make no difference, we demonstrate equivalence by presenting algorithmic conversions between the automaton with and without the modifications

In such cases we never concern ourselves with efficiency; How much time or how many steps would one automaton take compared to another?

We only concerned ourselves with whether or not both automata always produce the same result; Does the modification render the automaton more restrictive, less restrictive, or keep it exactly the same.

We now consider variations on the Turing Machine with the same question in mind; Will the variation change the set of languages the TM can accept?

We ignore questions of efficiency

- **Stay Option:** What if we allowed the TM to not move the tape head on a transition, instead of just L or R also allow S, which means "stay"?

On the surface this may sound trivial, but it adds the ability to change state without disturbing the tape or tape head

While it is obvious that adding the stay option does not reduce the languages a Turing Machine can accept one might ask ... Does the stay option give Turing Machines any extra real power? The answer is no it does not

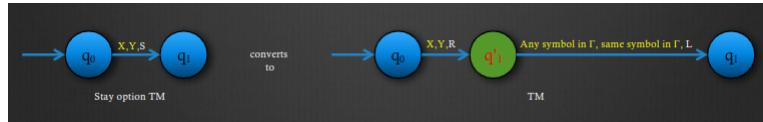
Although the stay option can be useful in reducing the number of states a TM may require, it adds no new power to TMs.

*Theorem:* For any Turing Machine with the stay option there is some Turing Machine that acts the same way on all inputs; looping, crashing, or accepting, while leaving the same data on the tape, and vice versa

First,  $TM \subseteq TM$  with stay option. Every TM is a TM with a stay option that simply does not have any stay option transitions

Part II... TM with stay option  $\rightarrow$  equivalent TM

Construct a new TM from the TM with the stay option by first copying the machine and then converting every transition with a stay option as follows



$\therefore$  Turing machine with stay option = Turing Machine

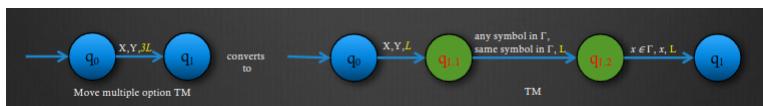
- **Move Multiple:** What if we allowed the TM to move the tape head multiple cells on a transition, instead of just L or R (one cell)?

*Theorem:* For any Turing Machine with the move multiple option there is some Turing Machine that acts the same way on all inputs; looping, crashing, or accepting, while leaving the same data on the tape, and vice versa.

*Proof.* Part I:  $TM \subseteq TM$  with move multiple option. Every TM is a TM with a move multiple option where  $n = 1$  on all move transitions

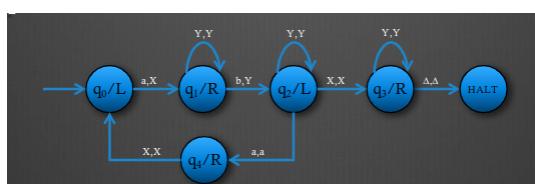
Part II. TM with move multiple option  $\rightarrow$  equivalent TM

Construct a new TM from the TM with the move multiple option by first copying the machine and then converting every transition where  $n > 1$  as in the the following  $3L$  example (do likewise for  $nR$ ):



$\therefore$  Turing machine with move multiple option = Turing Machine

- **Move-in-State:** Consider the following variation to Turing Machines that specifies tape head movement in states rather than on transitions:



Transitions labeled  $p, q$  meaning read  $p$  and replace with  $q$  on tape

States (except HALT) labeled  $q_i/\{L \text{ or } R\}$  meaning move tape head either L or R upon entering state.

Do not move tape head at beginning in  $q_0$  but if you re-enter  $q_0$  then move the tape head as indicated.

In original TMs when we entered a state like  $q_j$  above we moved the tape head, sometimes L other times R, before entering the state. How do we handle this with a move-in-state TM?

Does this make the TM or the move-in-state TM more powerful than the other? ...  
No, they are equally powerful.

*Theorem.* For any Turing Machine with the move-in-state option there is some Turing Machine that acts the same way on all inputs; looping, crashing, or accepting, while leaving the same data on the tape, and vice versa.

*Proof.* Part I: TM with move-in-state  $\rightarrow$  TM

### 1.7.3 Encoding TMs

- **Recall:** Recall that a Turing Machine is formally defined as a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \Delta, F).$$

And that we conveniently depict it as a state diagram.

- **Encoding:** Rather than using the 7-tuple to draw a picture we could, instead, represent the TM as a string of characters (i.e., encode the TM)

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \Delta, F) \rightarrow \text{some string } w \text{ that represents } M.$$

There are many ways to encode a Turing Machine, we present one.

Recall that  $Q$  is a finite set of states where

- $q_0 \in Q$  is the start state
- $F \subseteq Q$  is the (possibly empty) set of HALT state(s)

Without loss of generality we can modify  $M$  by "collapsing" any and all HALT states to a single HALT state.

In other words, if  $F$  was originally non-empty, then  $F$  will be left with a single state (and  $\delta$  modified accordingly), otherwise  $F$  will remain empty and  $\delta$  will be left unchanged.

We continue by assigning numbers to each of the states in  $Q$  as follows:

- The start state  $q_0$  is assigned 1.
- The (now single) HALT state, if one exists, is assigned 2.
- The remaining states are assigned arbitrary numbers other than 1 or 2 and such that each state is assigned a unique number.

Use the numbered states and the transition function  $\delta$  to create a new table that has a row for each transition (i.e., cell in  $\delta$ ).

$\delta: Q \times \Gamma \rightarrow \Gamma \times \{L,R\} \times Q$   
( $\delta$  is a partial function)

$\delta$	Tape Input				
	a	b	X	Y	$\Delta$
1 $q_0$	$\{\text{X}, \text{R}, q_1\}$				
3 $q_1$	$\{\text{a}, \text{R}, q_2\}$	$\{\text{Y}, \text{L}, q_2\}$		$\{\text{Y}, \text{R}, q_3\}$	
4 $q_2$	$\{\text{a}, \text{L}, q_3\}$		$\{\text{X}, \text{R}, q_3\}$	$\{\text{Y}, \text{L}, q_3\}$	
5 $q_3$				$\{\text{Y}, \text{R}, q_4\}$	$\{\Delta, \text{R}, \text{HALT}\}$
2 HALT			$\{\text{X}, \text{R}, q_4\}$		

From	To	Read	Write	Move
1	3	a	X	R
3	3	a	a	R
3	4	b	Y	L
3	3	Y	Y	R
4	6	a	a	L
4	5	X	X	R
4	4	Y	Y	L
5	5	Y	Y	R
5	2	$\Delta$	$\Delta$	R
6	6	a	a	L
6	1	X	X	R

We next, use input symbols in  $\Sigma \subset \gamma$  to create:

- A fixed-length encoding for each tape symbol in  $\Gamma$  and
- An encoding for each direction (L or R)

Note, in order to do this  $\Sigma$  must have at least two symbols

Symbol in $\Gamma$	Fixed-width encoding using only symbols from $\Sigma$	Direction	Fixed-width encoding using only symbols from $\Sigma$
a	aaa	L	a
b	aab	R	b
X	aba		
Y	abb		
$\Delta$	baa		

Encodings may be any length,  
but they must all be the same length

We now encode each row of the new table as follows;

- **Transition from state:**  $i \rightarrow j : a^i b a^j b$
- **Read/Write:** use fixed-width encoding
- **Direction:** use fixed-width encoding

From	To	Read	Write	Move	Encoding
1	3	a	X	R	abaaabaaaaab
3	3	a	a	R	aaabaaabaaaaab
3	4	b	Y	L	aaabaaaababbbb
3	3	Y	Y	R	aaabaaaabbbbbb
4	6	a	a	L	aaaaabaaaaabaaaaa
4	5	X	X	R	aaaaabaaaaabaaab
4	4	Y	Y	L	aaaaabaaaabbbbbb
5	5	Y	Y	R	aaaaabaaaaabbbbbb
5	2	$\Delta$	$\Delta$	R	aaaaabaaaabbaaab
6	6	a	a	L	aaaaanabaaaaabaaaa
6	1	X	X	R	aaaaanababaaaab

Because the *from to* are encoded delimited by 'b' and the rest of the encoding composed of *fixed-length parts*, we can unambiguously decode any encoding from the table:

```

graph TD
    Root[aaaabaaaaaabaaaaaaa] --> From4[From 4]
    Root --> To6[To 6]
    From4 --> ReadA[Read a]
    From4 --> WriteA[Write a]
    ReadA --> MoveL[Move L]
    To6 --> MoveR[Move R]
  
```

The encoding of the TM concludes by concatenating the encoded rows (in any order) to create a single string. (with the understanding that start state = 1 and halt state = 2.)

- **Notes and the code word language (CWL):** Every TM with at least two symbols in  $\Sigma$  can be encoded to a string

Not every string

- Can be decoded to a TM (e.g., strings that begin with 'b' do not represent a TM)
- **Decodes to a valid TM:** Might decode to a TM that is non-deterministic, has transitions from the HALT state, have READ, WRITE, or Move encodings that are invalid, etc.

In fact, using our last example where each symbol in  $\Gamma$  had an encoding length=3, from  $\Sigma^*$  we see that anything outside  $(a^+ba^+b(a+b)^7)^*$  is definitely not a valid TM, and even some of those strings are not a valid TM.

Nonetheless, that regular expression comes pretty close to defining the set of valid TM's so we call it the Code Word Language (CWL)

$$\text{CWL} = (a^+ba^+b(a+b)^7)^*.$$

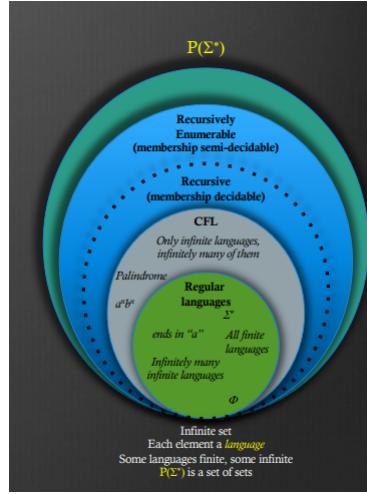
And we note

$$\text{valid TM encodings} \subset \text{CWL} \subset \Sigma^*.$$

#### 1.7.4 Revisiting Recursively Enumerable Languages

- **Recall:** The definitions of recursively enumerable and recursive languages: A language that is accepted by a Turing Machine is said to be a recursively enumerable language

A language that is accepted by at least one Turing Machine that halts on all inputs is said to be a recursive language.



We've seen an example of a recursive language that is not a context free language:  $a^n b^n c^n$

But is there a language that is recursively enumerable that is not recursive? Is there a language that is not even recursively enumerable?

Yes to both

- **Turing Machines and Their Encodings:** Recall that we can take a Turing Machine (having at least two input symbols in  $\Sigma$ ) and encode it using its own input alphabet  $\Sigma$

That means we can use a TM to process its own encoding.

Thinking about this same prospect in a different way, consider the strings in  $CWL = (a^+ba^+b(a+b)^*)^*$ , which we can partition into the following three sets:

- Invalid TMs
- Valid TMs that accept their own encoding
- Valid TMs that do not accept their own encoding

And given that partitioning we can define two new languages,  $\text{Acc}, \text{NotAcc} \subseteq \text{CWL}$ , as follows:

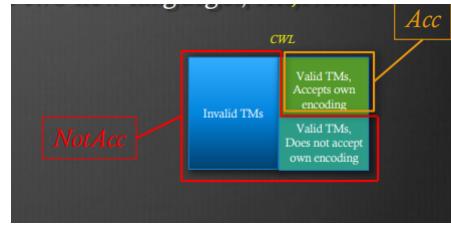
$$\text{Acc} = \begin{cases} w \in \text{CWL} \text{ such that:} \\ w \text{ is a valid TM that is} \\ \text{accepted by the TM it encodes} \end{cases}$$

$$\text{NotAcc} = \begin{cases} w \in \text{CWL} \text{ such that:} \\ 1. w \text{ is not a valid TM or} \\ 2. w \text{ is a valid TM that is not accepted by the TM it encodes} \end{cases} .$$

What kind of language is NotAcc? Regular? Context-free? Recursive? Recursively enumerable? none of the above, meaning, there does not exist a TM that accepts NotAcc.

*Theorem.* There exists a language that is not recursively enumerable.

*Proof (by contradiction).* Assume that the language NotAcc is recursively enumerable.:



This means that there exists some Turing Machine,  $T$ , that accepts NotAcc.

If we have a Turing Machine,  $T$ , then we can create a string  $w$  from  $T$ 's input alphabet  $\Sigma$  that encodes  $T$ . Because  $w$  was built from  $T$ 's input alphabet  $\Sigma$ , we can process  $w$  using  $T$  (i.e., have a TM process its own encoding).

There are only two possibilities; either  $T$  accepts  $w$ , or it does not.

Case I -  $T$  accepts  $w$

Because  $w$  is a valid encoding of  $T$ , and (in this case)  $T$  accepts  $w$ , we see that  $w \in \text{"Valid TM's, Accepts own Encoding"}$ , which means  $w \notin \text{NotAcc}$ . However, (in this case)  $T$  accepts  $w$ , which means  $w \in \text{NotAcc}$ . (contradiction)

Case II -  $T$  does not accept  $w$

Because  $w$  is a valid encoding of  $T$  and (in this case)  $T$  does not accept  $w$ , we see that  $w \in \text{"Valid TM's, Does not accept own Encoding"}$ , which means  $w \in \text{NotAcc}$ .

However,  $T$  (in this case)  $T$  did not accept  $w$ , which means  $w \notin \text{NotAcc}$  (contradiction)

Since the only two possible cases each led to a contradiction, we conclude that our assumption that NotAcc is a recursively enumerable language must be false.

$\therefore$  NotAcc is not a recursively enumerable language.

### 1.7.5 Universal turing machine

- **Turing Machines and Their Encodings:** We have seen that NotAcc is not a recursively enumerable language, meaning one cannot create a TM that accepts NotAcc.

What about the remaining partition?

What kind of language is Acc? Regular? Context-free? Recursive? Recursively enumerable? ... before we can answer that we need to create a special TM.

- **UTM:** We will create a special TM and call it a UTM. We will describe our new UTM in general terms rather than drawing it explicitly.

First, the UTM takes as input some  $w$  which is an encoding of some TM.

The basic function of the UTM is to simulate the TM that is encoded as its input, that is, the UTM should;

- accept strings the encoded TM would accept
- Reject (i.e., crash) strings the encoded TM would reject
- Loop on strings on which the encoded TM would loop

The first thing the UTM does is

1. Shift the input string to the right one cell inserting a # in the leftmost cell
2. Append a \$ to the end of the input string
3. Append a copy of the original input string just after the \$

The general idea is that the UTM will use

1.  $w_1$  to simulate the encoded TM (never changing it)
2.  $w_2$  as the input to the encoded TM (changing often)

Recall that an encoded TM is a concatenation of rows from a table (i.e., one row per transition in the encoded TM).

The UTM next inserts a blank ( $\Delta$ )

1. Before each row in  $w_1$
2. Before each symbol in  $w_2$



To complete its initialization phase, the UTM Inserts a "dummy row" at the beginning of the tape whose only value is its "To" state which must indicate the start state = 1

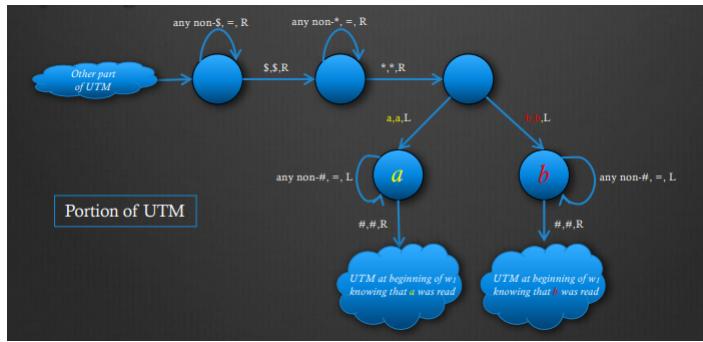
Places an \* before

- the newly inserted "dummy row" (to indicate that was the last transition processed)
- The first character in  $w_2$  to indicate the simulated tape head location

This ends the setup phase for the UTM. It is now ready to enter its next phase in which it simulates the TM encoded by  $w$ .

The UTM runs right down the tape to read the next character from  $w_2$

It next runs left back up the tape down a different branch in the UTM depending on the character that it read.



The UTM runs right down the tape to find the \* in  $w_1$  to discover what state the simulated TM is in (i.e., the To state from the row  $r_i$  last processed).

Then, Find the row  $r_j$  in  $w_1$  whose

- From state matches the simulated TM's current state
- Read character matches the last character read by the simulated TM

If no such row  $r_j$  in  $w_1$  can be found, then the simulated TM crashes, so crash the UTM. Otherwise

- Move the \* in  $w_1$  to precede the matching row  $r_j$
- Update the character in  $w_2$  (i.e., WRITE)
- Move the \* in  $w_2$  (L or R).

If the To state in matching row  $r_j$  is 2 (i.e., HALT), then the simulated TM HALTs (accepts) its input, so the UTM should also HALT (accept).

In this way the UTM accurately simulates the TM that is encoded as its input in that the UTM

1. accepts strings the encoded TM would accept
2. Rejects (i.e., crash) strings the encoded TM would reject
3. Loops on strings on which the encoded TM would loop

In other words, the UTM is a TM that accepts the language Acc. Therefore, Acc is a recursively enumerable language.

Is Acc a recursive language?

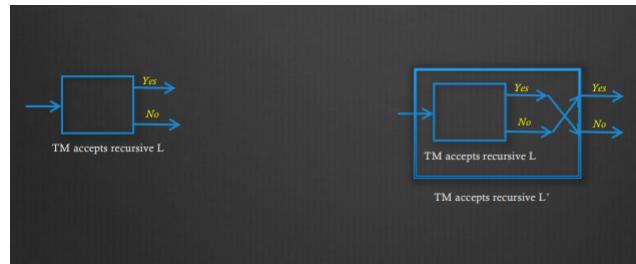
Before we can answer that question, we must first take a closer look at recursive languages by revisiting regular languages.

Recall that if  $L$  is a regular language, then its compliment  $L'$  is also a regular language.

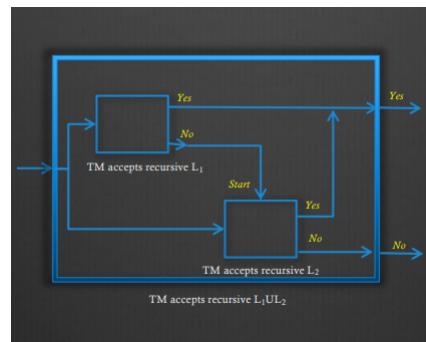
We accomplish this by taking a DFA that accepts  $L$  and changing all accepting states to non-accepting, and vice versa.

We can do something with recursive languages to show that if  $L$  is a recursive language then its compliment  $L'$  is also a recursive language.

We accomplish this by constructing a new TM that swaps "yes/no" decisions.



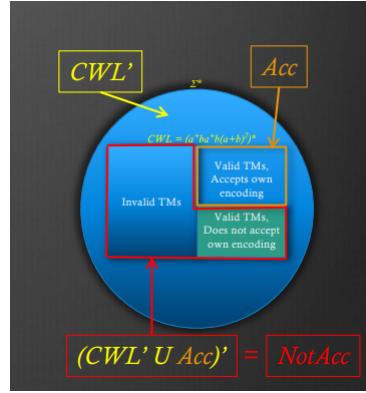
Similarly, the union of two recursive languages is also a recursive language. We accomplish this by constructing a new TM from the two recursive TMs.



Returning to Acc ... we know

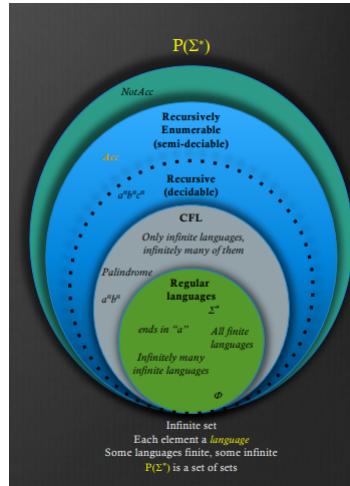
- CWL is a regular language
- CWL' is a regular language
- CWL' is a recursive language

- $(CWL' \cup Acc)$  is a recursive language
- $(CWL' \cup Acc)'$  is a recursive language



However,  $(CWL' \cup Acc)' = NotAcc$ , which we know is not recursively enumerable (much less recursive) ... contradiction.

$\therefore Acc$  is a recursively enumerable language that is not recursive.



- **More on UTM:** Consider the UTM we just described to accept  $Acc$ . UTM took as input an encoding of some TM and started by making a copy of that encoding;
  - $w_1$  - encoding of TM that UTM did not change
  - $w_2$  - copy of encoded TM ( $w_1$ ) that served as input to simulated TM - UTM frequently changed

After the initialization phase (i.e., making a copy of  $w_1$ ) and the UTM entered the simulation phase, it didn't matter that that  $w_2$  was a copy of  $w_1$ .

In other words, we could have placed any input for  $w_2$  and the UTM would have simulated the TM encoded by  $w_1$  processing the arbitrary input we placed as  $w_2$ .

Now the UTM becomes our *first stored-program computer*

The UTM takes as input (a) an encoded TM and (b) input for that encoded TM, and simulates the TM (a) as it processes input (b)

The UTM is a TM that can simulate any TM processing any data, and hence its name, The Universal Turing Machine (Turing, 1936).

The Universal Turing Machine is the foundation of all computing theory and was the conceptual archetype of the early computer.

It is not a computer's operating system, but rather, the logic that guides a computer's instruction fetch and execution cycle (typically implemented in a computer's hardware).

The first stored-program computer was built by John von Neumann and his colleagues within years of Turing's work.

Contemporary computers implement the UTM differently (for efficiency) but they are all based on the UTM.

### 1.7.6 The halting problem

- **Revisiting Turing Machines:** If a TM has been processing some string for a long time, how do we know if this is a case when the TM will be looping forever?

If the TM is going to loop forever, then let's just stop now and declare the input string outside the language accepted by the TM!

- **The halting problem:** There is a problem, The Halting Problem, that states:

*If we are given a Turing Machine T and input string w, can we tell whether T halts on w?*

The Halting Problem question does not care if T accepts/rejects w, only whether T eventually stops (accept or crash/reject) or loops forever.

Obviously, we cannot simply start processing w with T to find the answer.

The UTM we presented cannot answer the question because it will loop forever if T loops on w.

What The Halting Problem is calling for is an algorithm (a TM) that

- Accepts as input an encoded TM T and input string w
- The Halting Problem TM would never loop forever (i.e., must be recursive, not merely recursively enumerable)
- On all possible inputs it would always stop and report either "halts" or "loops" to indicate what T would do when processing w.

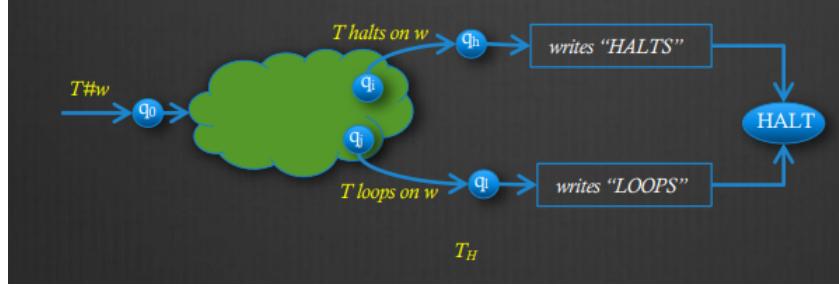
So the question becomes does such an algorithm exist?... No!

- **Halting problem proof:** Assume a TM exists that solves The Halting Problem, call it  $T_H$ .

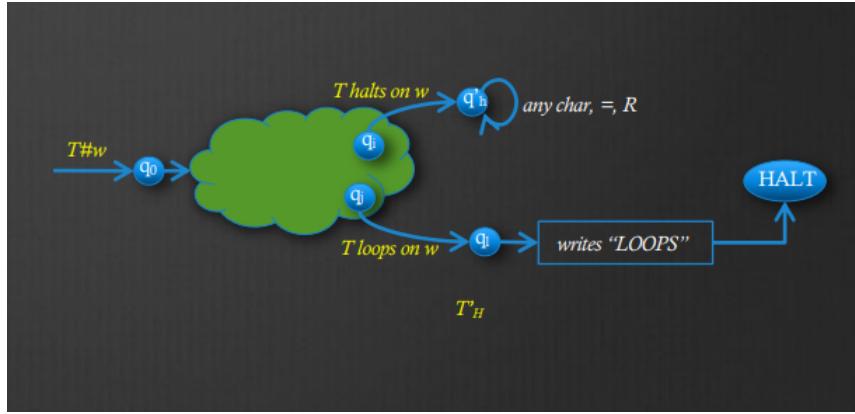
That is,  $T_H$  is a TM that takes as input an encoded TM T, followed by a #, followed by some input string w.

$T_H$  would always terminate and print as its output on its tape

- "HALTS" if T will halt when processing w, or
- "LOOPS" if T will loop forever when processing w



We modify  $T_H$  to create a  $T'_H$  by replacing the TH path from  $q_i$  through its "writes HALTS" states, to HALT with a new state that loops forever.



$T'_H$  will either

- Loop forever if input  $T$  would halt on its input  $w$ , or
- Halt with "LOOPS" written on the tape when input  $T$  would loop forever on its input  $w$

Consider how  $T'_H$  would process input  $T'_H\#T'_H$ , that is, ask  $T'_H$  how it would process an encoding of itself

As we've seen, there are only two possible responses from  $T'_H$ .

**Case I:**  $T'_H$  actually loops forever when processing  $T'_H\#T'_H$  (i.e., enters  $q'_h$ ).  $T'_H$  only loops forever when its input  $T$  would halt on  $T$ 's input  $w$

Since we processed  $T'_H\#T'_H$  that would mean that  $T'_H$  must halt when processing its own input, contradiction.

**Case II:**  $T'_H$  actually halts with "LOOPS" when processing on  $T'_H\#T'_H$  (i.e., enters  $q_1$ ).  $T'_H$  only halts with "LOOPS" when its input  $T$  would loop forever on  $T$ 's input  $w$ .

Since we processed  $T'_H\#T'_H$  that would mean that  $T'_H$  must loop forever when processing its own input, contradiction.

This tells us that  $T'_H$  cannot exist, which in itself is not a proof that there is no solution to The Halting Problem.

However,  $T'_H$  was a legitimate modification of  $T_H$  which we assumed to exist as a solution to The Halting Problem.

Since  $T'_H$  cannot exist and it was based on a legitimate modification of  $T_H$ , we conclude that  $T_H$  cannot exist

. $\therefore$  There is no algorithm that can solve The Halting Problem.

## 1.8 Complexity theory

- **Intro:** The field of computational complexity looks at the resources required to solve problems. Or, more generally, how much resources does it take: time, memory space, number of processors, bandwidth, and so fourth. It is a given that the problem is solvable.
- **$\mathcal{P}$  and  $\mathcal{NP}$ :** Regarding time, we define two sets of languages. The set  $\mathcal{P}$  is those languages that can be solved in polynomial time, and  $\mathcal{NP}$  is the set of those languages that can be solved in polynomial time on a non-deterministic TM. The set  $\mathcal{P}$  is somewhat viewed as the reasonably tractable problems, but many problems of practical interest have been shown to be in  $\mathcal{NP}$ . Thus, the question of whether all of  $\mathcal{NP}$  is in  $\mathcal{P}$  (that is, whether  $\mathcal{P} = \mathcal{NP}$ ) is of fundamental importance. This question remains unsolved.

### 1.8.1 Time complexity

- **Time complexity:** The time complexity of a set of problems is how much time is needed to solve them. The time complexity of a language is how much time is needed to decide membership in it.

The goal is to determine how the resources required depend on the size of the input.  $n$  always denotes the size of the input. Then, the running time is the number of steps as a function of  $n$ .

It is important to note that we analyze the **worst case**. It doesn't matter that some instances can be done quickly, what matters is if *every* instance can be done quickly. A TM is said to run in time  $T(n)$  if for all inputs  $w$ , it halts within  $T(|w|)$  steps.

- **Big-O:** It is important to note that constants do not matter. If a machine runs in  $2n^2, 7n^2$ , or  $1000n^2$  steps is immaterial. In fact, we cannot care about the constants. For, the exact time will depend very heavily on what the atomic operations are. And nobody uses a laptop TM anyway. If the number of steps in a certain TM is proportional to  $n^2$ , we say the TM runs in  $O(n^2)$  time, or "order  $n^2$ " time, meaning there exists some constant  $c$  such that the TM runs in at most  $cn^2$  steps for any input of length  $n$ . The order notation says how the worst case running time grows as  $n$  gets large.
- **Polynomial time:** The collection of all problems that can be solved in polynomial time is called  $\mathcal{P}$ . That is, a language  $L$  is in  $\mathcal{P}$  if there exists a constant  $k$  and a TM that decides  $L$  that runs in time  $O(n^k)$ .
- **Complexity class:** It is common to call a set of related languages a *complexity class*, or just a class. The class  $\mathcal{P}$  roughly captures the collection of practically solvable problems.
- **Polynomially related:** Two models of computation are polynomially related if there is a polynomial  $p$  such that if a language is decidable in  $T(n)$  time on one model, the language is decidable in time  $p(T(n))$  on the other.
- **Nondeterministic time:** The collection of all problems that can be solved in polynomial time by a non-deterministic machine is called  $\mathcal{NP}$ . That is, a language  $L$  is in  $\mathcal{NP}$  if there exists a constant  $k$  and an NTM that decides  $L$  that runs in time  $O(n^k)$ . It should be clear that

$$\mathcal{P} \subseteq \mathcal{NP}.$$

- **Theorem:** Let  $L$  be a recursive language. If there is an NTM for  $L$  that runs in time  $T(n)$ , then there is a deterministic TM for  $L$  that runs in time  $O(C^{T(n)})$  for some constant  $C$
- **Conjecture:**  $\mathcal{P} \neq \mathcal{NP}$ , however, we do not know.

# DSA

## 2.1 C++ Stuff

### 2.1.1 Type declarations

- **Discern any type:** Some rules,
  1. Start with the variable name, we read from inside to out
  2. const, %, \*, and basic types go on the left
  3. const refers to what is immediately on the left (except for `const int*`), but the standard form of this is actually `int const*`. Thus, the exception to this is const is at the very left, then it refers to what is immediately right.
  4. arrays and functions go on the right, function args are type declaration sub-problems

The Algorithm:

- Start with the variable name, or the implied name position
- Read right until end or )
- Read left until end or (
- If something still left to read, move out one level of parenthesis and go to 2, else done.

Thus, using parenthesis allows us to change direction, this will come in handy.

#### Examples:

- `a` is an int  $\Rightarrow$  `int a`
- `a` is a pointer to an int  $\Rightarrow$  `int * a`
- `a` is a pointer to a constant int  $\Rightarrow$  `int const * a` (also `const int * a`)
- `a` is a constant pointer to an int  $\Rightarrow$  `int * const a`
- `a` is a constant pointer to a constant int  $\Rightarrow$  `int const * const a` (also `const int * const a`)
- `a` is an array of 5 ints  $\Rightarrow$  `int a[5]`
- `a` is an array of 5 pointers to constant ints  $\Rightarrow$  `int const * a[5]`
- `a` is a pointer to an array of 5 constant ints  $\Rightarrow$  `int const (* a)[5]`

- **Multi dimensional arrays (matrices):** Think of multi-dimensional arrays as arrays of arrays. More indicative of what's happening internally. `float dat [3][4];` can be read as: "dat is an array of 3 arrays of 4 floats" (Using the algorithm from above).

#### Examples:

- `arg1` is a reference to an array of 25 constant pointers to arrays of 8 strings.  $\Rightarrow$  `string (* const (& arg1)[25])[8]`

**Note:** Notice how we use parenthesis to change direction

- **Function Pointers:** Pointers point to bytes, which can be interpreted different ways. Pointers can point to bytes that can be interpreted as code, i.e. a function pointer.

**Examples:**

- $f$  is a pointer to a function which takes an int and returns void.  $\Rightarrow \text{void } (*f) (\text{int})$

### 2.1.2 G++

- **Compilation and linking:** Compilers turn source code into executable code.
  - **Source code → object code (Compilation):** Object code is almost executable. It contains pieces that it provides to other objects, and holes to be filled in. It is a slow process
  - **Object code → executable (Linking):** Connects pieces of object files together. This is a fast process

**Note:** Many "compilers" do both compiling and linking. Most programs are built in two stages:

1. Compile all the source code files
2. Link the object code file into an executable

This is the most efficient way to compile large projects. Changing a single source code file requires a small number of compilations (slow), followed by linking (fast).

- **Standard unix c compiler:** The standard is GNU gcc
- **Standard unix cpp compiler:** The standard is GNU g++
- **g++ Options:** With no options, g++ will go from source to an executable named a.out
  - **-o:** The -o option gives the name of the output file
  - **-c:** The -c option makes the compiler stop after the compilation stage. No linking is done. The name of the object code file is the same as the source with the extension replaced with .o
  - **-W[warning]:** Tell the compiler to look for a specific warning
  - **-Wall (Warning all):** There are many -Wwarning options, which warn of various conditions. -Wall warns about all of them. The compiler keeps going through warnings

**Note:** A compiler warning is usually a bug waiting to happen. Do all you can to get rid of all warnings.

- **-Werror:** The -Werror option turns all warnings into errors. The compiler aborts on an error.
- **-g:** The -g option turns on debugging, and leaves much extra information in an object file. Executable is much larger, possibly slower.
- **-O:** The -O option turns on optimization. There are several different levels of optimization, e.g. -O0, -O1, -O2, -O3.

**Note:** Optimization may break your code, and -O and -g don't always work well together

- **-I[directory]:** The -I option specifies an additional directory to search for include files. No space between -I and directory

Thus,

```
0  #include "./dir/headerfile" // Without -I
1  #include "headerfile" // With -I : g++ -I./dir ...
```

- **-L[directory]:** The -L option specifies an additional directory to search for libraries. No space between -L and directory.

**Note:** This option is meant for linking only. It has no effect in compilation.

- **-l[*libraryname*]**: The -l option specifies a library for linking. No space between -l and library name. The library name is related to the libray file name, but it is not identical. Library names start with "lib" and end with ".so.\*" or ".a". These are removed. For example
  - \* The math library /lib/x86\_64-linux-gnu/libm.so.6 is linked as -lm
  - \* The X11 graphics library /usr/lib/x86\_64-linux-gnu/libX11.so is linked as -lX11

**Note:** This option is for linking only. It has no effect in compilation. Libraries are the last things listed in a linking command.

If you're linking against a library that is located in a non-standard directory (a directory that is not automatically searched by the linker, such as ./libs), then you need to tell the linker where to find that library using the -L option. Thus, -L tells the compiler where to look, -l specifies which one to grab.

# Databases

## 3.1 Introduction to databases (db concepts)

### 3.1.1 Definitions and theorems

- **What is a database?:** A database is a collection of stored operational data used by the application systems of some particular enterprise, better yet a collection of related data.
- **What is an enterprise?:** a generic term for any reasonably large-scale commercial, scientific, technical, or other application. Such as
  - Manufacturing
  - Financial
  - Medical
  - University
  - Government
- **Operational data:** Data maintained about the operation of an enterprise, such as
  - Products
  - Accounts
  - Patients
  - Students
  - Plans

**Note:** Notice that this DOES NOT include input/output data

- **Database Management System (DBMS):** A Database Management System (DBMS) is a collection of programs that enables users to create and maintain a database. Ie a general-purpose software system that facilitates
  - Definition of databases
  - Construction of databases
  - Manipulation of data within a database
  - Sharing of data between users/applications
- **Defining a database:** For the data being stored in the database, defining the database specifies
  - The data types
  - The structures
  - The constraints
- **Constructing a Database:** Constructing a database is the process of storing the data itself on some storage device

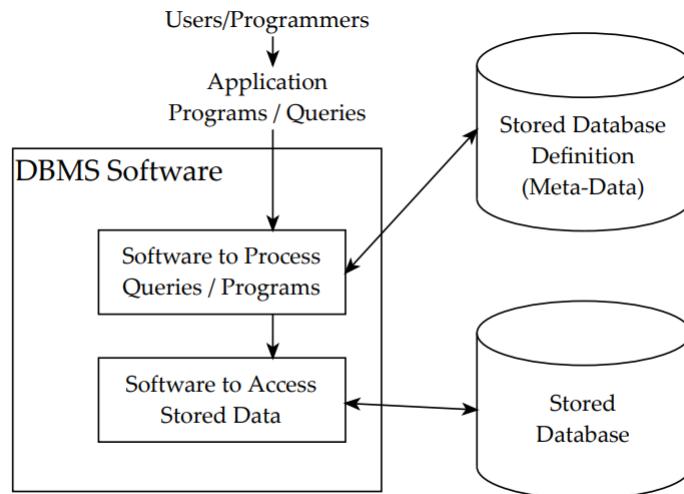
**Note:** The storage device is controlled by the DBMS

- **Manipulating a Database**

- retrieve specific information in a query
- update the database to include changes
- generate reports from the data

Most likely already defined by whatever dbms you choose

- **Sharing a Database:** Sharing a database Allows multiple users and programs to access the database at the same time, any conflicts between applications are handled by the DBMS
- **Other Important Functions of a Database:** Other important functions provided by a DBMS include
  - Protection, system protection, security protection
  - Maintenance, allows updates to be performed easily
- **Simplified Database System Environment:**



- **Main characteristics of a database system are:**

- Self-describing nature of a database system
- Insulation between programs and data, and data abstraction
- Support for multiple views of the data
- Sharing of data and multi-user transaction processing

- **Other Capabilities of DBMS Systems:** Support for at least one data model through which the user can view the data, There is at least one abstract model of data that allows the user to see the "information" in the database, Relational, hierarchical, network, inverted list, or object-oriented

Support for at least one data model through which the user can view the data

- efficient file access which allows us to "find the boss of Susie Jones"
- allows us to "navigate" within the data
- allows us to combine values in 2 or more databases to obtain "information"

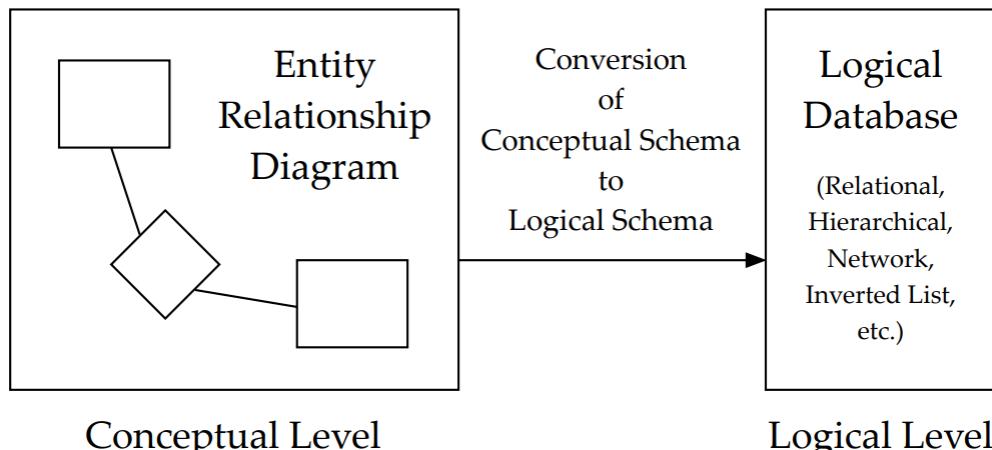
Support for high-level languages that allow the user to define the structure of the data, access that data, and manipulate it

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)
- query language access data
- operations such as add, delete, and replace

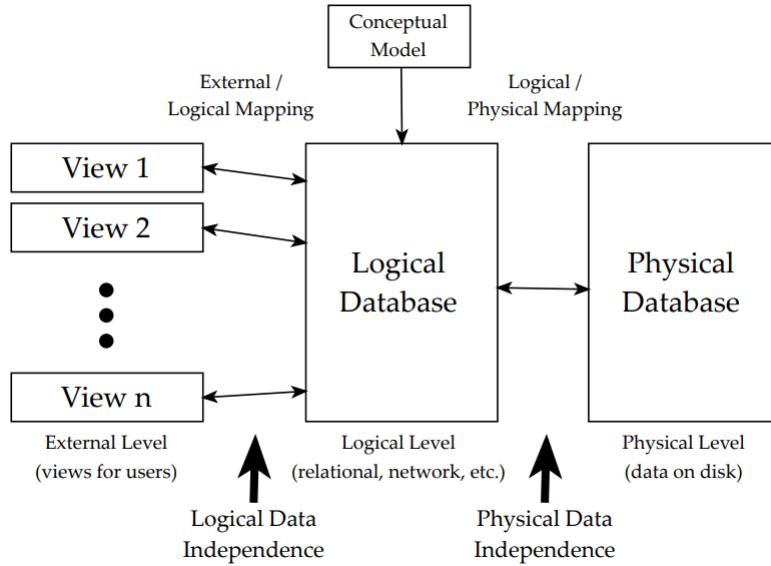
- **Transaction Management:** Transaction management is a feature that provides correct, concurrent access to the database, possibly by many users at the same time, ability to simultaneously manage large numbers of *transactions*
- **Access Control:** Access control is the ability to limit access to data by unauthorized users along with the capability to check the validity of the data. This is to protect against loss when database crashes and prevent unauthorized access to portions of the data
- **Resiliency:** Resiliency is the ability to recover from system failures without losing data, Ideally, should be able to recover from any type of failure, such as
  - sabotage
  - acts of God
  - hardware failure
  - software failure
  - etc.

**Note:** Obviously, some of these would require more than just software - offsite backups, etc

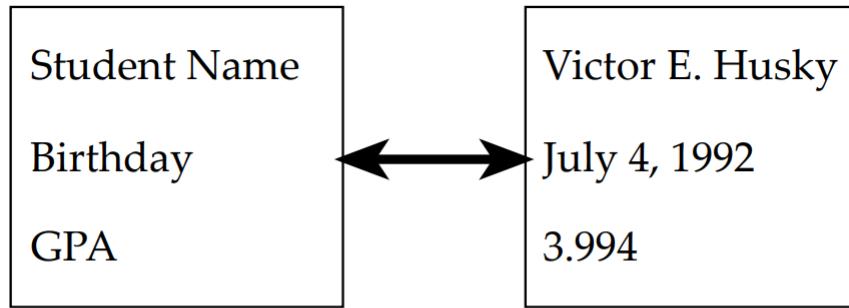
- **Use of Conceptual Modeling:**



- Leveled Architecture of a DBMS:



- **External level:** a view or sub-schema, a portion of the logical database, may be in a higher level language
- **Logical Level:** abstraction of the real world as it pertains to the users of the database. DBMS provides a data definition language (DDL) to describe the logical schema in terms of a specific data model such as relational, hierarchical, network, inverted list, etc.
- **Physical Level:** The collection of files and indices, the collection of files and indices, this is the actual data
- **Instance:** An instance of the database is the actual contents of the data, it could be
  - the extension of the database
  - current state of the database
  - a snapshot of the data at a given point in time
- **Schema:** The schema of a database is the data about what the data represents. Such as,
  - plan of the database
  - logical plan
  - physical plan
  - the intention of the database
- **Schema vs Instance:**



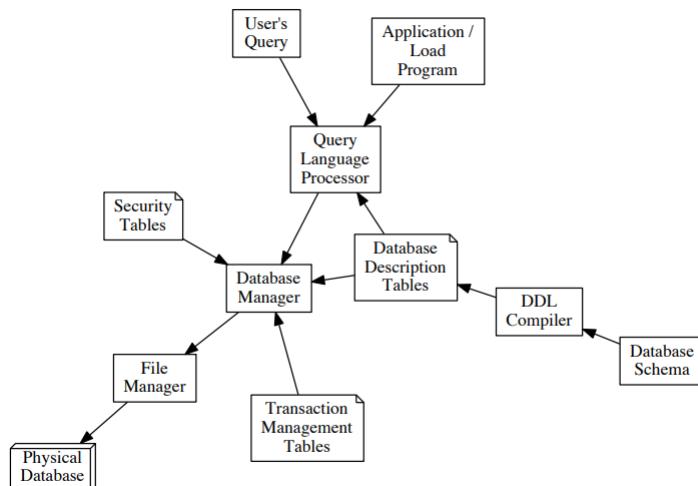
## Schema

description of what data can be stored

## Instance

the actual data that is stored

- **Data Independence:** Data Independence is a property of an appropriately designed database system, it has to do with the mapping of logical level to physical level, and logical to external
  - **Physical data independence:** Physical schema can be changed without modifying logical schema
  - **Logical data independence:** logical schema can be changed without having to modify any of the external views
- **DCL (Control), DDL (Definition), DML (Manipulation):** may be completely separate (example is IMS), may be intermixed (DB2), or may be a host language, for example an application program in which DML commands are embedded such as COBOL or PL/I
- **DBMS Components:**



- **Overall DBMS Usage Scenario:** Database Administrator (DBA) define the conceptual, logical, and physical levels using DDL. DBMS software stores instances of these in schemas. User defines views (External Schema) in DDL. User accesses database using DML

- **Advantages of a Database:**

- Controlled redundancy
- Reduced inconsistency in the data
- Shared access to data
- Standards enforced
- Security restrictions maintained
- Integrity maintained more easily
- Provides capability for backup and recovery
- Permitting inferences and actions using rules

- **Disadvantages of a Database:**

- Increased complexity needed to implement concurrency control
- Increased complexity needed for centralized access control
- Security needed to allow the sharing of data
- Necessary redundancies can cause complexity when updating

- **Data vs Information:**

- **Data:** Data refers to raw, unprocessed facts, figures, and details. It represents basic elements that have not been interpreted or given any meaning.
- **Information:** Information is processed, organized, or structured data that is meaningful and useful. It is data that has been interpreted or analyzed to provide context, relevance, and purpose.

## 3.2 Conceptual Modeling and ER Diagrams

### 3.2.1 Definitions and theorems

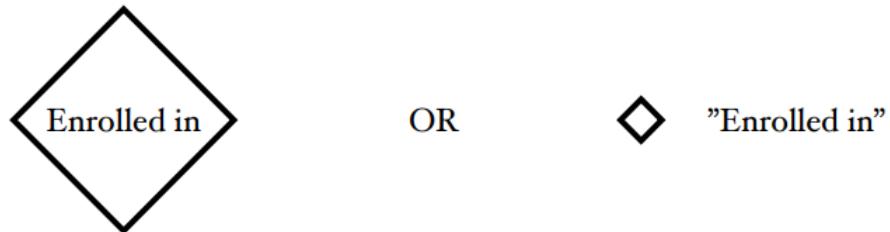
- **Data Models:** A means of describing the structure of data, we typically have A set of operations that manipulate the data (for data models that are implemented)
- **Types of data models:**
  - Conceptual data model
  - Logical data models - relational, network, hierarchical, inverted list, or object-oriented
- **Conceptual Data Model:**
  - Shows the structure of the data including how things are related
  - Communication tool
  - Independent of commercial DBMSes
  - Relatively easy to learn and use
  - Helps show the semantics or meaning of the data
  - Graphical representation
  - Entity-Relationship Model is very common
- **Logical Data Models - Relational:** Data is stored in relations (tables). These tables have one value per cell. Based upon a mathematical model.
- **Logical Data Models - Network:** Data is stored in records (vertices) and associations between them (edges), Based upon a model called CODASYL
- **Logical Data Models - Hierarchical:** Data is stored in a tree structure with parent/child relationships
- **Logical Data Models - Inverted List:** Tabular representation of the data using indices to access the tables, Almost relational, but it allows for non-atomic data values<sup>1</sup>, which are not allowed in relations
- **Logical Data Models - Object Oriented:** Data stored as objects which contain
  - Identifier
  - Name
  - Lifetime
  - Structure
- **Entity-Relationship Model:** Meant to be simple and easy to read. Should be able to convey the design both to database designers and unsophisticated users
- **Entities:** Principle objects about which information is kept - These are the \*things\* we store data about. If you look at the ER Diagram like a spoken language, the entities are nouns - Person, place, thing, event. When drawn on the ER diagram, entities are shown as rectangles with the name of the entity inside.

---

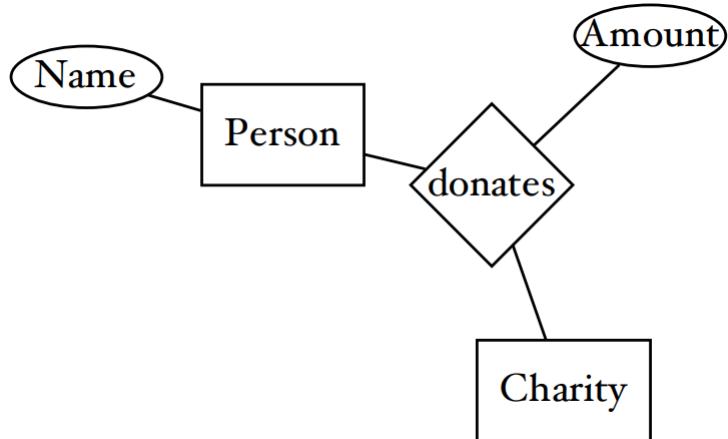
<sup>1</sup>"Non-atomic data values" refer to data structures or values that are composed of multiple components, as opposed to atomic data values, which are indivisible and represent a single value.



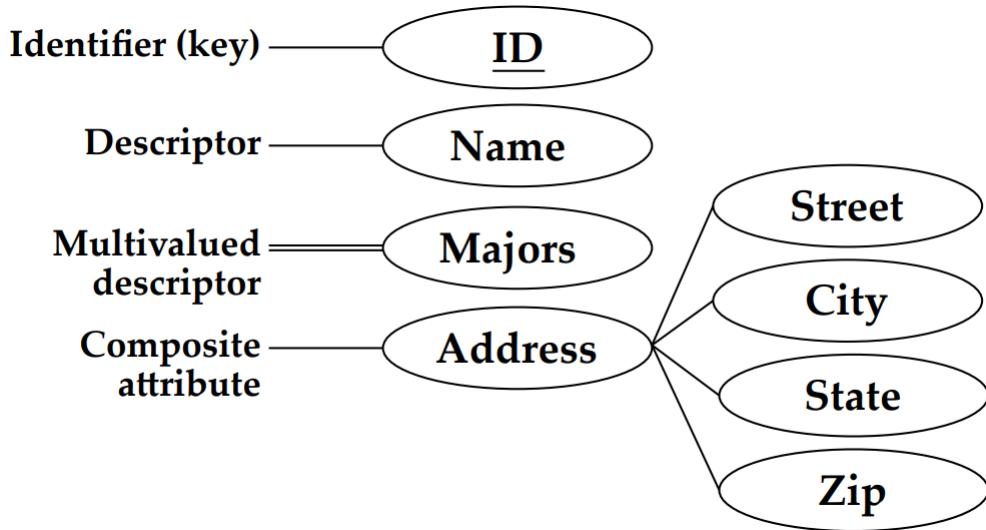
- **Relationships:** Relationships connect one or more entities together to show an association. A relationship *cannot* exist without at least one associated entity. Graphically represented as a diamond with the name of the relationship inside, or just beside it



- **Attributes:** Characteristics of entities **OR** of relationships, Represent some small piece of associated data, Represented by either a rounded rectangle or an oval.

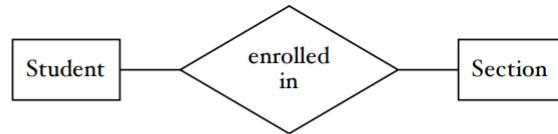


- **Attributes on Entities:** When an attribute is attached to an entity, it is expected to have a value for every instance of that entity, unless it is allowed to be null. For instance, in the diagram above, Name was an attribute of Person. Every person that we store data about will have a value for Name.
- **Attributes on Relationships:** When an attribute is attached to a relationship, it is only expected to have a value when the entities involved in the relationship come together in the appropriate way. In the diagram from before, the Amount attribute is attached to the donates relationship, which connects the Person and Charity entities. Amount will have one value for each time a Person donates to a Charity, denoting how much that person donated to the charity. It will not necessarily have a value for a given person, or a given charity. This can be referred to as the **intersection data**.
- **Types of attributes:**

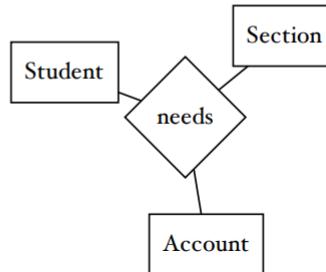


- **Degree of a Relationship:** The degree of a relationship is defined as how many entities it associates. If one entity is associated more than once (such as with a recursive relationship), then the degree counts each time it is referenced.

► binary



► ternary

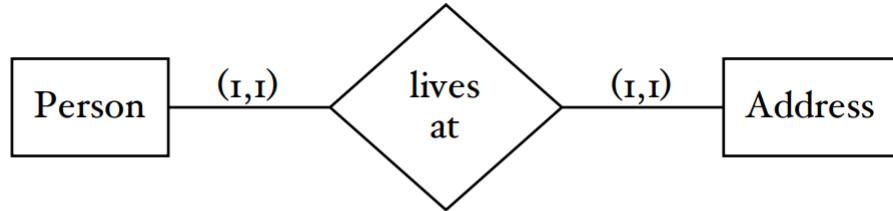


**Note:** There is no limit to how many entities there can be in a relationship. After binary, and ternary, we start to call the relationships  $n$ -ary, where  $n$  is the degree

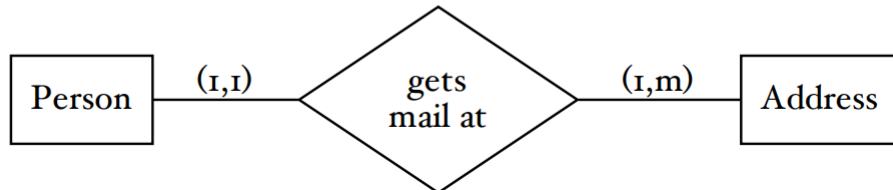
- **Connectivity of a Relationship:**

- A constraint of the mapping of associated entities
- Written as (minimum, maximum).
- Minimum is usually zero or one.
- Maximum is a number (commonly one) or can be a letter denoting many.
- The actual number is called the cardinality.

► one-to-one

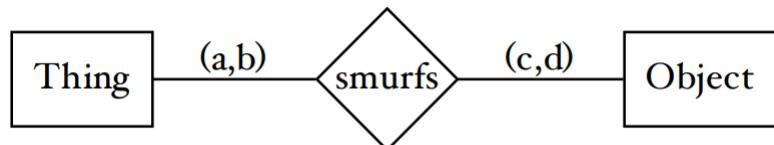


► one-to-many



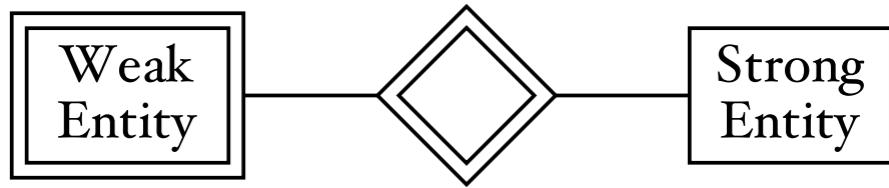
Together (from the image) both sides make up the connectivity, to refer to a single side, we use the term "cardinality", ie the cardinality of a person is (1,1). If we hold Address constant (We know a specific address and are therefore referring to that), how many persons may live at that address, in this case (1,1)

- **Attributes on Relationships (revisited):** Must be on a many-to-many relationship. (1-many and 1-to-1 relationships should have the attribute on one of the entities involved. Someone needs to know all of the associated entities to access the attribute.
- **Reading Cardinalities:** For binary relationships:
  - For each Thing that smurfs, there are a minimum of  $c$ , and a maximum of  $d$  Objects.
  - For each Object that smurfs/is smurfed, there is a minimum of  $a$  and a maximum of  $b$  Things

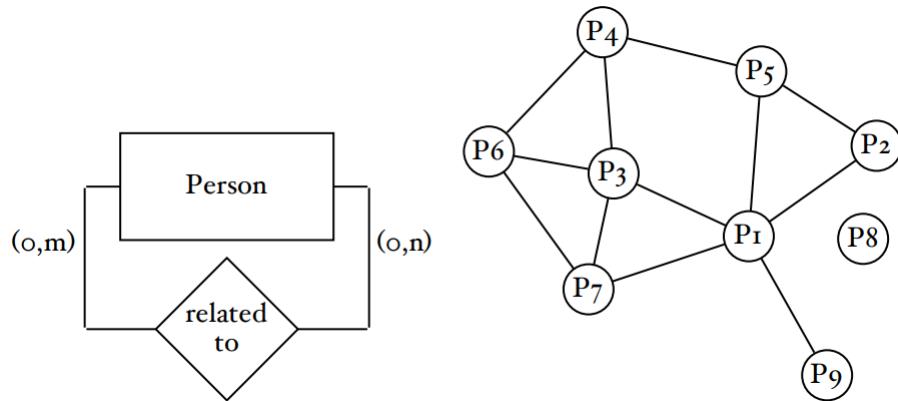


- **Weak Entities:** Sometimes you may run into an entity that depends upon another entity for its existence. The weak entity is a tool you can use to represent this.:w

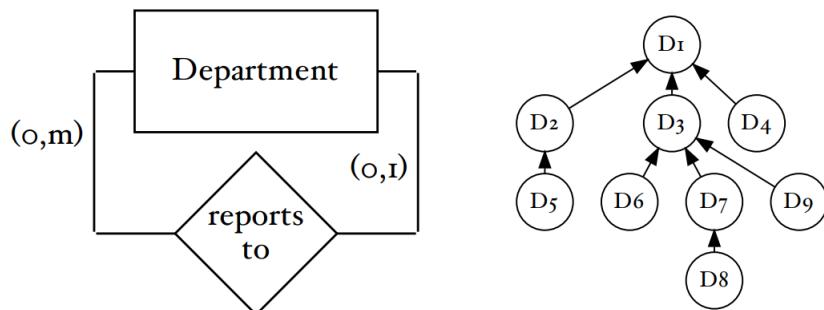
Weak entities are written like normal entities, except that they have a double rectangle outline. The relationship that connects the weak entity to the strong entity it depends upon will be written with a double diamond. This does not mean that the relationship is weak. It is just to indicate upon which entity the weak entity depends.



- **Recursive Relationships:** It is possible for an entity to have a relationship with itself. This is called a recursive relationship. It makes more sense if you think of entities as collections of objects of their appropriate type
- **Recursive Relationships - Many-To-Many:** A many-to-many recursive relationship means that the objects are arranged in a network structure. Notice that the minimum is 0 on both sides. This is important.



- **Recursive Relationships - One-To-Many:** A one-to-many recursive relationship means that the objects are arranged in a tree structure. Notice that the minimum is still 0 on both sides. This is important.

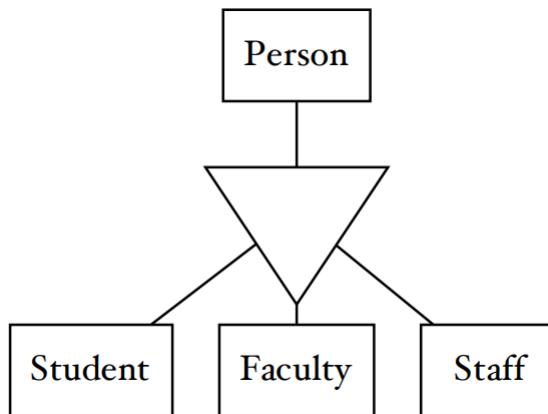


- **Entity or Attribute?:** Sometimes it isn't clear whether something should be an entity or an attribute of some other entity. Usually the decision will come down to how complicated it is to store the data, and how important it is. If it ends up being used in multiple places, it might be a clue that you should use an entity

- **Inheritance:** Two types of inheritance available
  - "is a" inheritance. This shows that the subtype IS a member of the supertype.
  - "is part of " inheritance. This shows that the supertype contains, or is made up of members of the subtypes.

All attributes of the supertype entity are inherited by the subtype entities. The identifier of the subtypes will be the same as the supertype

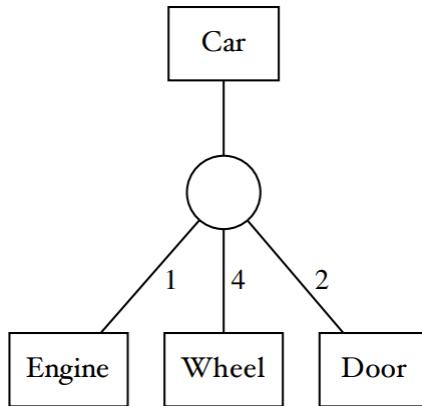
- **IS A Inheritance:** This type of inheritance happens when you have a supertype and one or more subtypes that are members of the supertype. Denoted by an upside-down triangle, with the supertype on top, and the subtypes coming out the bottom.



- **Defining IS-A inheritance:** There are two things you need to choose when using IS-A inheritance:
  - **Generalization (no) vs. specialization (yes):** can the supertype occur without being a member of the specified subtypes?
  - **Overlapped (yes) vs. disjoint subtypes (no):** is it possible for a single occurrence of the supertype to be a member of more than one subtype?

They are mutually exclusive so you need to pick one of each, ie. GO, GD, SO, SD

- **IS-A inheritance - Generalization:** Supertype is the union of all of the subtypes, This means that an instance of the supertype CANNOT EXIST without belonging to at least one subtype.
- **IS-A inheritance - Specialization:** The subtype entities specialize the supertype, This means that an instance of the supertype CAN exist without being related to any of the subtypes
- **IS-A inheritance - Overlapping Subtypes:** It is possible for an instance of the supertype to be related to more than one of the subtypes
- **IS-A inheritance - Disjoint Subtypes:** the subtype entities are mutually exclusive, it is not possible for an instance of the supertype to be related to more than one subtype.
- **IS-PART-OF Inheritance:** "Is part of " inheritance indicates that the supertype is constructed from instances of the subtypes. It is shown on an ER diagram as a circle, with the supertype on the top, and subtypes on the bottom.



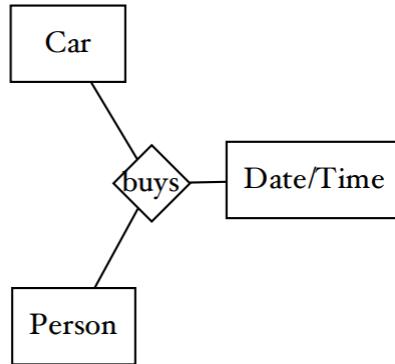
- **Warning about IS-PART-OF:** The IS PART OF inheritance operator does have its uses, but it is not very commonly used. If you see something involving a certain number of things being present, there are several possibilities
  - Sometimes a number is specified that isn't actually important for what we are modeling. This won't even be represented on an ER Diagram. This is the case when changing the number wouldn't have any effect on the necessary structure of a database.
  - If you need a certain number of items for a relationship to hold, you should explore using the connectivity of the relationship to express that.
  - Finally, this IS PART OF inheritance might be useful. It is almost never necessary, however.
- **Are you actually representing what you want to?:** Let's say you're running a business selling used cars. A simple ER diagram for the sales might look like the following:



The resulting database would have one entry for each time a specific person buys a specific car. If the same person buys the same car more than once (obviously selling it to someone else at some point), this model would no longer be appropriate.

The resulting database would have one entry for each time a specific person buys a specific car. If the same person buys the same car more than once (obviously selling it to someone else at some point), this model would no longer be appropriate.

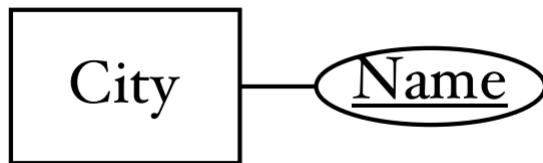
Adding a new entity to the relationship for the date/time of the purchase can fix this problem.



Notice that the connectivities can change when you add new entities to the relationship.

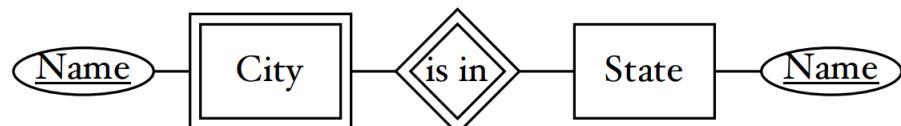
- **Weak Entities - Introduction:** So far, all of the entities we have used have been things that stand on their own. There are some situations where we are modeling an object for which we certainly need to store data, but the items exist only in the context of some other entity. Many of these examples can occur

One example of a time that an entity depends on another would be the idea of a city. Within a state, we can generally be assured that cities will have unique names. If we were working only at that level, the City could be an entity as we saw above. A good identifier for it would be the name of the city, so we would see the following:



In some situations, this would be valid. The Name attribute can serve, in those circumstances, as an appropriate identifier.

To indicate this sort of dependency, we can make the dependent entity a "weak" entity. This is drawn with a double-edged rectangle, shown below.



Notice that the City entity is now drawn as a weak entity, with a double border. The relationship between the weak entity and the strong entity is also drawn with a double border. The relationship is not weak, per se, but it is used to indicate which strong entity the weak entity depends upon.

- **Discriminant (partial key):** The discriminant, also known as the partial key, is an attribute (or a set of attributes) within the weak entity that can uniquely identify the weak entity, but only in combination with the primary key of the strong entity it is associated with. In other words, the discriminant helps to distinguish instances of the weak entity when they are tied to a particular instance of the strong entity.
- **Schema:** In databases, a schema is the structural definition of how data is organized in a database. It outlines the way data is stored

### 3.3 The Relational Model

- **Basic Structure:**

- **Relations:** In the relational data model, our database is made up of one or more **relations** (tables). Each relation should have a unique name.
- **Schema:** The schema of a relation is written as **Relation\_Name**( $A_1, A_2, \dots, A_n$ ), Where  $A_1, A_2, \dots, A_n$  are placeholders for the attribute names
- **Column headers (attributes):** The attributes becomes the column headers of the relation.
- **Instance data, tuples:** When there is instance data, it will come in the form of **tuples** (rows), which have a value for each attribute, as shown below

**Note:** No field may contain than one value.

Relation_Name				
$A_1$	$A_2$	$A_3$	...	$A_n$
$x_1$	$x_2$	$x_3$	...	$x_n$
$y_1$	$y_2$	$y_3$	...	$y_n$
...	...	...		...

- **The domain of an attribute:** Each attribute becomes a column heading

Each attribute (column) also has an associated **domain**. The domain of an attribute is the set of all valid values for it. The domain may be looked at as a data type, but may have additional constraints.

- **The domain of a set of attributes:** The domain of a set of attributes is the set of all possible combinations of values for the attributes in the set.
- **Tuples (Rows):** A tuple is a special type of (mathematical) set containing values for each attribute within the relation. Tuples are shown as rows in the table, with the value for each attribute under the appropriate column
- **Atomic tuples:** The values are required to be atomic; there can be only one value per tuple per attribute
- **Relation vs relationship:** Though they have similar names, A relation (table) and a relationship (from an ER diagram) **ARE NOT** the same thing.
  - **Degree of relation:** The degree of a relation is the number of attributes present.
  - **Cardinality of a Relation:** The cardinality of a relation is the number of tuples present.
- **Keys:** Speaking generally, the purpose of a key is to uniquely identify a tuple in some relation.
  - **Super keys:** A super key within a relation is an attribute or set of attributes whose values can uniquely identify any tuple within that relation
  - **The trivial key:** Every relation has at least one - the set of all attributes in the relation
  - **Candidate Keys:** A candidate key is a minimal super key - the minimum set of attributes necessary to uniquely identify a tuple within the relation

- **Primary Key:** The primary key for a relation is chosen by the database designer from among the relation's candidate keys. It becomes the "official" key that is used to reference tuples within the relation. There can be only one
- **Prime, non-prime attributes:** Once a primary key is chosen, each of the attributes in the relation will be either **prime** or **non-prime** with respect to the relation. A prime attribute is one of the attributes that can be found in any of the candidate keys. A non-prime attribute is one of the attributes not found in any of the candidate keys

Once a primary key is chosen for it, the schema of a relation is written with the primary key's attributes underlined

- **Foreign Keys:** A foreign key is a tool used to link relations within a database. Since every relation has a primary key that uniquely identifies each tuple, the values of those key attributes can be used from another relation to reference individual tuples.

The relation whose primary key is being used is the **home relation**

- **Order Independence:** In relations, the order things appear doesn't matter. There are ways to force them to sort later when we're working with SQL, but the relation itself has no order for either rows or attributes...
- **Order Independence - Attributes:** It doesn't matter what order the attributes appear in, if two relational schemas have the same name, the same attributes, and the same primary key, then they are equivalent.
- **Order Independence - Tuples:** Tuples are stored unordered. If you need to have them appear in some order later, you will be able to sort based on the values inside of them using SQL.
- **Constraints:** Constraints are limits imposed on the domains of various attributes. These can come from the system your database is modeling
- **Entity Integrity Constraint:** The entity integrity constraint applies to all relations. It states that no tuple may exist within a relation that has null value for any of attributes that make up the primary key. This is a consequence of the primary key being a candidate key, which is minimal and cannot do its job with any less data.
- **Referential Integrity Constraint:** It constrains the values of foreign keys in relations to values that actually exist as primary keys for tuples within the home relation. If the foreign key is otherwise allowed to be NULL, then that is also an acceptable value.
- **Summary: Terms:**
  - **Relations:** Tables
  - **Columns:** Attributes
  - **Tuples:** The rows in the relation that holds the instance datae
  - **Domain of an attribute:** Set of all possible values for the attribute
  - **Domain of a set of attributes:** Set of all possible combinations of values for the attributes in the set
  - **Degree of relation:** The degree of a relation is the number of attributes present.
  - **Cardinality of a Relation:** The cardinality of a relation is the number of tuples present.

### 3.4 Relational Model Normalization

- **Designing Relational Databases:** There are a large number of possible ways to represent each problem with using relations. Some choices will perform better than others for various reasons. The option chosen should be the best one, but how do we know which one that is?

We should study:

- Problems that can come up
  - How to avoid them
  - Desirable properties
  - How to guarantee them
- **Basic Example:** If our database is a single relation with schema **SP**(SuppName, SuppAddr, Item, Price) with the instance data:

SuppName	SuppAddr	Item	Price
John	10 Main	Apple	\$2.00
John	10 Main	Orange	\$2.50
Jane	20 State	Grape	\$1.25
Jane	20 State	Apple	\$2.25
Frank	30 Elm	Mango	\$6.00

There are some common things that we might want to do that would cause issues

- **Insertion Anomaly:** Let's say we want to add a new vendor, "Sally", and store her address, "40 Pine", but she is not selling anything yet. Can this be inserted into the relation SP?

**NO.** The primary key is (SuppName, Item), but we only have SuppName. The entity integrity constraint is violated if we try to insert the data as a tuple in this relation. It cannot fit. We call this an insertion anomaly.

- **Deletion Anomaly:** This time, let's say that Frank no longer sells Mango. We want to take that out of the database so nobody can order a mango that is not available. Can this tuple remain in the relation with the Mango information removed?

**NO.** The primary key is (SuppName, Item), and the Item is going away. The entity integrity constraint is violated if we remove the data from the tuple in this relation. We can either keep the whole tuple, advertising fake mango, or delete the whole tuple and lose the information on Frank, which doesn't exist in any other tuples. We call this a deletion anomaly.

- **Update Anomaly:** Next, let's say that John is moving to a different address. We would have to change it once for every item John is selling. This isn't a big deal with only two items, but as John's list of supplied items grows, so does the amount of database work that needs to be done every time he moves. If any of the SuppAddr values for John don't agree, then it may not be clear which is the right address for John. This is an update anomaly.

- **Redundancy:** Redundancy is when values are repeated.

It can be

- \* **Good:** If you have an off-site backup of your entire database, the redundancy is useful, and can be used to restore in case of a failure.
- \* **Bad:** Redundancy on the same physical device is unnecessary. It wastes space and comes with the potential for update anomalies.
- **Note:** The good redundancy is something the DBA/IT department should handle. When we talk about redundancy in the design of our database, we will be talking about the bad kind.
- **Anomalies summarized:**

#### **Insertion anomalies:**

- When a piece of data cannot be inserted because it violates some constraint of the relation.
- Usually this is the entity integrity constraint being violated, but not always. See the Sally example

#### **Deletion anomalies:**

- When deleting some piece of data, a deletion anomaly is when more data is lost than intended
- Usually this is caused when the data removed is part of the primary key, which would cause a violation of the entity integrity constraint. See the Frank example

#### **Update anomalies:**

- When updating a single value requires changes to multiple tuples, this is an update anomaly. See the John example.
- This is caused by unnecessary redundancies in the data.
- These cause inefficiency, and potential inconsistencies.

- **Decomposition:** There is no rule that says that a relational database must be made up of a single relation. The way we will solve these anomalies is to add new relations to our database and change the old ones. This is called decomposition.

Using the example from above, we can remove the anomalies by decomposing the database into two relations.

**SP**(SuppName, Item, Price)

SuppName	Item	Price
John	Apple	\$2.00
John	Orange	\$2.50
Jane	Grape	\$1.25
Jane	Apple	\$2.25

**S**(SuppName, SuppAddr)

SuppName	SuppAddr
John	10 Main
Jane	20 State
Frank	30 Elm
Sally	40 Pine

- **When to decompose:** One way of designing a database could be to list all of the possible anomalies and then decompose to fix each of them. The problem with this is that any anomalies you don't see coming will not be fixed.

We will look at a systematic method of identifying the potential for anomalies. This method is called normalization

- **Normalization:** Normalization involves making sure that each of your relations follows certain rules. Depending on which rules are followed, each of the relations in your database will be in one or more normal forms. These rules are based on functional dependencies
- **Functional Dependencies:** A functional dependency is a statement about which attributes can be inferred from other attributes. If we take  $X$  and  $Y$  as sets of attributes, we can write:

$$X \rightarrow Y.$$

Which means, if, whenever unique values for **all** of the attributes in  $X$  are known, unique values for **each** of the attributes of  $Y$  are guaranteed to be possible to look up or to infer using those values.

This is read either as:

- $X$  functionally determines  $Y$
- $Y$  is functionally dependent upon  $X$

- **Functional Dependencies: Real-life Examples:**
  - **ZID → StudentFirstName, StudentLastName, Birthday:** If I identify a student using their ZID, that student has one first name, last name, and birthday
  - **StudentFirstName ↛ ZID:** The first name is not enough to determine a single ZID, as there are multiple students with the same first name
  - **ZID, CourseID, Semester → Grade:** If I know which student, which course, and which semester, I can find a single grade
- **Functional Dependencies: Keep In Mind:** FDs are constraints present within the operational data your database models. They don't necessarily describe how things work in the real world, but they do have to accurately describe any data you will store in your database

FDs **must** hold for all possible data values. Attempts to add data that does not obey the FDs will result in anomalies.

FDs can be enforced during insertion if the database is set up properly

- **Armstrong's Axioms:** Armstrong's Axioms are a set of rules for operations that are permissible when manipulating functional dependencies
  - **Reflexivity:** If  $Y \subseteq X$ , then  $X \rightarrow Y$
  - **Augmentation:** If  $X \rightarrow Y$ , then  $XZ \rightarrow YZ$  for any  $Z$
  - **Transitivity:** If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$
  - **Decomposition:** If  $X \rightarrow YZ$ , then  $X \rightarrow Y$  and  $X \rightarrow Z$
  - **Composition:** If  $X \rightarrow Y$  and  $A \rightarrow B$ , then  $XA \rightarrow YB$
  - **Union (Notation):** If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow YZ$
  - **Pseudo-transitivity:** If  $X \rightarrow Y$  and  $YZ \rightarrow W$ , then  $XZ \rightarrow W$
  - **Self-determination:**  $I \rightarrow I$  for any  $I$
- **Functional Dependencies: Keys Revisited:** Now that we know about functional dependencies (FDs), we can assert:

The attributes of a superkey must functionally determine all of the attributes of the relation.

Candidate keys and primary keys are superkeys, so this is true of them as well, and they also satisfy additional requirements.

**Example:** As an example, say we have the relation  $\mathbf{R}(a,b,c,d,e,f)$ . We can say

$$\begin{aligned} a &\rightarrow a, b, c, d, e, f \\ \implies a &\rightarrow b, c, d, e, f. \end{aligned}$$

- **First Normal Form (1NF):** You should recall from the introduction to relations that all of the values in a tuple with a relation must be atomic. This means that there is a maximum of one value per attribute per tuple

The requirement for a relation to be in First Normal Form (1NF) is this same requirement that all of the values must be atomic

What this usually looks like is a table with multiple values in a single cell. A non-1NF relation would not even technically count as a relation.

Given the table:

X	Y	Z
x1	y1	z1
		z2
		z3
x2	y2	z4
x3	y2	z5

It looks like  $X$  would have been the primary key, but it's not doing its job of uniquely determining  $Z$ , which is showing as a repeating group so  $X$  can't be a key

What usually causes this is not having the correct primary key

The table above has the following function dependencies:

$$\begin{aligned} X &\rightarrow Y \\ X, Z &\rightarrow Z. \end{aligned}$$

To move this pseudo-relation into an actual relation that doesn't violate 1NF, we need to choose a real primary key that meets the requirements. We do that using the FDs. In this case,  $(X, Z)$  works.

Changing the primary key yields: -  $R(X, Y, Z)$

X	Y	Z
x1	y1	z1
x1	y1	z2
x1	y1	z3
x2	y2	z4
x3	y2	z5

- **Pseudo-relation:** The notation for a "pseudo-relation" like the one above would be to use inner parenthesis on the repeating group, ie.  $\mathbf{R}(X, Y, (Z))$
- **Second Normal Form (2NF):** Second Normal Form (2NF) has to do with the concept of full dependence.

Given two sets of attributes,  $X$  and  $Y$ , we can say that  $Y$  is fully dependent on  $X$ , if (and only if)

$$X \rightarrow Y.$$

And no subset of  $X$  determines  $Y$

A relation is in 2NF if:

- It already meets the requirements of 1NF, and
- All non-prime attributes of the relation are fully dependent upon the entire primary key

What breaks 2NF is when attributes are dependent upon only part of the primary key. To fix 2NF violations once we're in 1NF, decomposition is the solution.

**Example:** Going back to our earlier example: **EmpProj**(EmpID, Project, Supv, Dept, Case)

EmpID	Project	Supv	Dept	Case
e1	p1	s1	d1	c1
e2	p2	s2	d2	c2
e1	p3	s1	d1	c3
e3	p3	s1	d1	c3

### Functional Dependencies:

$$\begin{aligned} \text{EmpID, Project} &\rightarrow \text{Supv, Dept, Case} \\ \text{EmpID} &\rightarrow \text{Supv, Dept} \\ \text{Supv} &\rightarrow \text{Dept} \end{aligned}$$

A quick glance confirms all of the values are atomic, so 1NF is confirmed.

There is a 2NF violation caused by  $(\text{EmpID} \rightarrow \text{Supv, Dept})$  because the primary key is  $(\text{EmpID, Project})$ , but only EmpID is on the LHS.

Observing the instance data, you should easily see that the attributes of the RHS cause update anomalies in this table. We also can't insert a new employee with no project (insertion anomaly), and removing e2 from p2 would remove e2 from the database entirely (deletion anomaly). These are symptoms of the 2NF violation.

**Decomposition Pattern:** There is a pattern to follow for the decomposition. Start with the original relation, and the FD that causes the violation.

$$\begin{aligned} \mathbf{EmpProj}(\underline{\text{EmpID}}, \underline{\text{Project}}, \text{Supv}, \text{Dept}, \text{Case}) \\ \mathbf{EmpID} \rightarrow \text{Supv, Dept}. \end{aligned}$$

The attributes on the RHS of the FD are removed from the original relation and placed into a newly created relation that has the FD's LHS as its primary key. A foreign key links the attribute from the LHS in the original table (the LHS is not removed) to the corresponding tuple in the new table, where it is the primary key.

$$\begin{aligned} \mathbf{EmpProj}(\text{EmpID}, \text{Project}, \text{Case}) \\ \mathbf{Employee}(\text{EmpID}, \text{Supv}, \text{Dept}). \end{aligned}$$

### Instance of 2NF Version:

$$\mathbf{EmpProj}(\text{EmpID}, \text{Project}, \text{Case})$$

EmpID	Project	Case
e1	p1	c1
e2	p2	c2
e1	p3	c3
e3	p3	c3

$$\mathbf{Employee}(\text{EmpID}, \text{Supv}, \text{Dept})$$

EmpID	Supv	Dept
e1	s1	d1
e2	s2	d2
e3	s1	d1

- **Third Normal Form (3NF):** To be in Third Normal Form (3NF), a relation must
  1. already qualify to be in 2NF
  2. none of the non-prime attributes may be transitively dependent upon the primary key

By definition, all non-prime attribute are functionally dependent upon the primary key. What makes a transitive dependency is that there is also some non-prime attribute (which also depends on the key) that also functionally determines the attribute.

To quickly identify the transitive dependencies from the list of FDs, look on the LHS for attributes that are non-prime in the context of the current relation.

**Example:**

$\text{EmpProj}(\text{EmpID}, \text{Project}, \text{Case})$   
 $\text{Employee}(\text{EmpID}, \text{Supv}, \text{Dept})$   
 $\text{EmpID}, \text{Project} \rightarrow \text{Supv}, \text{Dept}, \text{Case}$   
 $\text{EmpID} \rightarrow \text{Supv}, \text{Dept}$   
 $\text{Supv} \rightarrow \text{Dept.}$

In this case, the FD that causes our relations to violate 3NF is ( $\text{Supv} \rightarrow \text{Dept}$ ), and the violation happens in the Employee relation. If you refer back to the instance data of that in the 2NF solution, you can see that the violation can cause anomalies, so we want to fix it.

Just like 2NF, we fix 3NF by decomposing using the FD that causes the violation to occur. **AT NO POINT DO WE CHANGE THE FDs**

**Decomposition Pattern:** We follow the same pattern for decomposition in 3NF as we did in 2NF. Start with the relation that has the violation, and the FD that causes the violation to occur.

$\text{Employee}(\text{EmpID}, \text{Supv}, \text{Dept})$   
 $\text{Supv} \rightarrow \text{Dept.}$

The attributes on the RHS of the FD are removed from the violating relation and placed into a newly created relation that has the FD's LHS as its primary key. A foreign key links the attribute from the LHS in the original table (the LHS is not removed) to the corresponding tuple in the new table, where it is the primary key.

$\text{Employee}(\text{EmpID}, \text{Supv})$   
 $\text{SupvDept}(\text{Supv}, \text{Dept}).$

The RHS (Dept) that was a violation when it was in Employee because the LHS (Supv) was non-prime is no longer there to cause the problem. It is in the new relation where the LHS (Supv) is the primary key, and therefore we don't have a transitive dependency. These two relations no longer have the 3NF violation.

- **Summary of the normalization forms:**

**First Normal Form (1NF):**

- No repeating groups. All values are atomic.
- A primary key must have been chosen, and this primary key must be a proper superkey - it needs to be able to functionally determine every attribute in the relation.

1NF violations are fixed by choosing an appropriate primary key

**Second Normal Form (2NF) - To be in Second Normal Form, a relation must conform to 1NF and:**

- All of the non-prime attributes must be fully dependent upon the entire primary key.
- No non-prime attribute may be functionally determined by any subset of the primary key.
- No partial key dependencies

2NF violations are fixed by decomposition.

**Third Normal Form (3NF) - To be in Third Normal Form, a relation must conform to 2NF and:**

- There may be no transitive dependencies.
- No non-prime attribute may functionally determine another non-prime attribute.

3NF violations are fixed by decomposition.

### 3.5 ERD to Relations (Conceptual to logical)

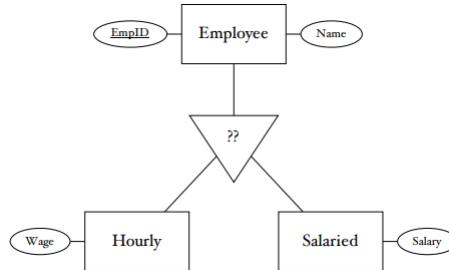
- **The basic outline (steps)**
  1. Handle all of the entities
  2. Handle all of the relationships
- **Entity handling:** We will start with entities, because they can stand on their own, unlike relationships or attributes. In general, each entity will get its own relation. The attributes of the entity will become attributes in the schema of the relation created. There are some special cases to take into account, which will be handled from most independent to least, so:
  - a. Strong (non-weak) entities that are not subtypes
  - b. Strong.(non-weak) entities that are subtypes
  - c. Weak entities
- **Entities like date:** there is no reason to make a relation for a "Date" entity or similar. The single value for the date is enough to determine it, and any other data associated with it is generally happening through a relationship anyway. Think about what data would go into such a table and how little use there would be for storing it separately.
- **Handling strong, non subtype entities:** Make a new relation, whose name will be the same as the name of the entity

The primary key of the relation will be all of the identifier attributes, taken together  
All attributes of the entity become attributes of the relation Every instance of the entity gets the relevant values put into a new tuple in the relation

**Example:** Suppose we had an entity *A* with attributes ID, and other:

Then, we would make a relation *A(ID, other)*

- **Handling strong, subtype entities:** Suppose



**Employee** is a supertype (not subtype) so it gets handled in the previous step

**Employee**(EmpId, name)

**Hourly** and **Salaried** are each strong, but they are subtypes (each is a type of Employee), so they are handled here

This type of inheritance means that the subtypes are types of the supertype, so they are identified by **Employee's** EmpID

There are two methods of handling these.

1. **Big table:** The first method involves putting the attributes of the subtypes into the relation made for the supertype. So, the original relation:

**Employee**(EmpID, Name)

Would become something like:

**Employee**(EmpID, Name, Wage, Salary)

but it would need to be modified to indicate which subtypes a given employee belongs to. Let's examine that on the next page.

The big table method needs a way to know which of the subtypes the current instance of the supertype belongs to, which is handled differently depending on the IS-A's configuration.

For **disjoint subtypes**, where an instance of the supertype can only be one of the subtypes at a time, we can add an attribute, EmpType that has a value indicating which type this employee is.:

**Employee**(EmpID, Name, EmpType, Wage, Salary)

For generalization, EmpType would not allow NULL. For specialization, it would be allowed.

For **overlapping subtypes**, it is possible to be more than one at a time, so we need an individual true/false answer for each type:

**Employee**(EmpID, Name, IsHourly, Wage, IsSalaried, Salary)

In this case, nothing about the schema would indicate generalization vs. specialization

2. **New relation:** Method 2 involves creating a new relation for the subtype entity. The name of new relation would be the same as the name of the entity.

The primary key of the new relation would be the same as the primary key for the supertype's relation.

The primary key is also a foreign key to the existing table.

An instance of the supertype entity will only have a tuple in the subtype relation if it is a member of that subtype, so we will not need any extra attributes like we did in method 1.

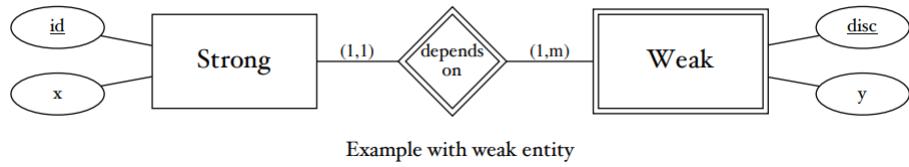
The foreign key can be used to look up any of the attributes that are being inherited from the supertype

Thus, we would have

**Employee**(EmpID, Name  
**Hourly**(EmpID†, Wage)  
**Salaried**(EmpID†, Salary).

**Note:** The (†) (dagger symbol) will be used in these slides to indicate that the attribute is part of a foreign key (and, in this example, the whole thing).

- **Handling weak entities:** Suppose



The strong entity would already have a relation.

**Strong**(id, x)

The weak entity gets its own relation. The primary key will be the concatenation of the weak entity's discriminator with the strong entity's identifier. The other attributes of the entity are brought in as non-prime attributes.

**Weak**(id†, disc, y)

The id portion is a foreign key to the Strong relation

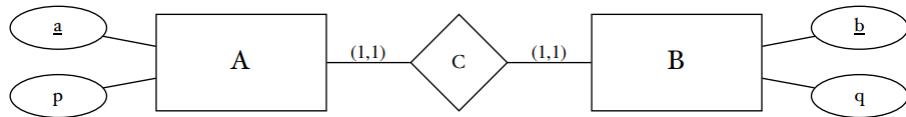
- **Entities: Functional Dependencies:** The only functional dependencies introduced by the entities of an ER diagram are the ones introduced when the identifiers become primary keys. Remember that a primary key has to functionally determine all of the other attributes in a relation
- **Handle relationships:** The relationships will be handled in order from lowest degree to highest degree, and within that, from simplest cardinality (one-to-one) to more complicated cardinalities (many-to-many, etc.).

The purpose of a relationship is to form connections between entities. We know that we are using relations to represent our entities, so we will need to use a tool that can link those relations to each other.

The tool best suited to linking tuples from relations together is the foreign key.

Every relationship we model in the relational model will have one or more foreign key involved. Where we put these foreign keys will depend on the cardinality, and the decisions are motivated by the normal forms we discussed.

1. **Binary one-to-one Relationships:** In a binary relationship, we will already have made a relation for each of the entities involved.



Here, C is a binary, one-to-one relationship between A and B.

$$\mathbf{A}(\underline{a}, p) \quad \text{and} \quad \mathbf{B}(\underline{b}, q)$$

Since each instance of B will have one of A, and each instance of A will have one of B through C, we can represent this one-to-one relationship by putting a new foreign key into the entity for either side. Choose either:

$$\mathbf{A}(\underline{a}, p, b\dagger) \quad \text{or} \quad \mathbf{B}(\underline{b}, q, a\dagger)$$

The relationship implies the functional dependencies:

$$a \rightarrow b$$

$$b \rightarrow a.$$

2. **Binary one-to-many Relationships:** In a binary relationship, we will already have made a relation for each of the entities involved.

$$\mathbf{A}(\underline{a}, p) \quad \text{and} \quad \mathbf{B}(\underline{b}, q)$$

For this one-to-many relationship, there can be many instances of B for each of A, so we can't have the foreign key in the A table (wouldn't be atomic, so 1NF would be violated). We still do have the option of putting a foreign key in the B table pointing to the corresponding A, so our only option is:

$$\mathbf{B}(\underline{b}, q, a\dagger)$$

The only FD is

$$b \rightarrow a.$$

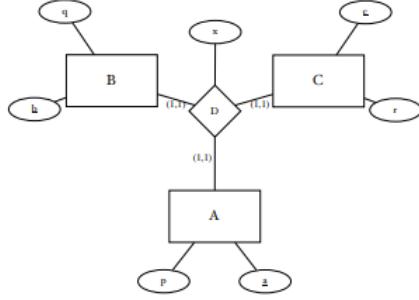
3. **Binary many-to-many Relationships:** In a binary relationship, we will already have made a relation for each of the entities involved.

$$\mathbf{A}(\underline{a}, p) \quad \text{and} \quad \mathbf{B}(\underline{b}, q)$$

There are no new functional dependencies introduced by the relationship, and putting a foreign key into either relation would not be atomic (1NF violation). The many-to-many relationship requires a new relation. Its foreign key will be the concatenation of the primary keys of each of the entity relations, which will be used as foreign keys to the corresponding tables. Any intersection data is put into this new relation as a non-prime attribute.

$$\mathbf{C}(a\dagger, b\dagger, x)$$

4. **Relationships Greater than Binary: one-to-one-to-one:**



So we have

$$\mathbf{A}(\underline{a}, p) \text{ and } \mathbf{B}(b, \underline{q}) \text{ and } \mathbf{C}(\underline{c}, r)$$

Each of the "one legs" represents a functional dependency, and each of them gives us a potential relation to choose from for our relation.

**Note:** If we have say only two ones, like a one to one to many relationship, we

Functional Dependency	Potential Relation for D
$a, b \rightarrow c$	$\mathbf{D}(\underline{a}^\dagger, b^\dagger, c^\dagger, x)$
$b, c \rightarrow a$	$\mathbf{D}(a^\dagger, b, c^\dagger, x)$
$a, c \rightarrow b$	$\mathbf{D}(a^\dagger, b, c^\dagger, x)$

would just have less functional dependencies and therefore less options to choose from (see table above)

5. **Greater than Binary without any "ones":** No functional dependencies are implied by this relationship. To stay in 3NF, the relation we must use is:

$$\mathbf{D}(\underline{a}^\dagger, \underline{b}^\dagger, \underline{c}^\dagger, x)$$

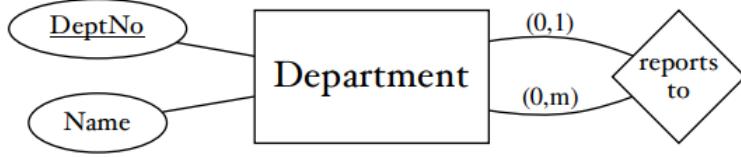
6. **Date entities (and similar):** For relationships that have a "Date" entity (or the equivalent), recall that we did not make a relation for that entity. The only change necessary for your relationship involving that entity is that the date value is used instead of a foreign key, and that attribute will not be a foreign key, because the home relation would not exist

As an example, if the C entity in the ternary relationship with no "ones" ER diagram were a Date entity, we would not create the C relation for it, and the relation to represent the relationship would be modified. Notice that the  $c$  attribute is still part of the primary key, but no longer a foreign key.

$$\begin{aligned} \text{From: } & \mathbf{D}(\underline{a}^\dagger, \underline{b}^\dagger, \underline{c}^\dagger, x) \\ \text{To: } & \mathbf{D}(\underline{a}^\dagger, \underline{b}^\dagger, \underline{c}, x) \end{aligned}$$

7. **Recursive Relationships: one-to-many:** Recursive relationships will be handled as if they were normal relationships of the same degree and cardinality. The practical difference is that the entity that is linked multiple times will still only have one relation, so multiple foreign keys might go to the same table.

Suppose:



There should obviously only be one relation for the entity Department, because it is only a single entity.

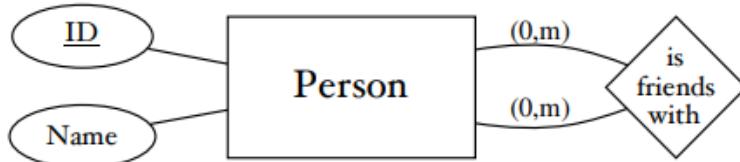
**Department**(DeptNo, Name)

With a non-recursive one-to-many binary relationship, we would have put a foreign key to the relation for the one side into the relation on the one side. In this version, we only have one table, so the decision is easy. We will need to come up with another name for the foreign key, as we cannot have two attributes with the same name inside the same relation. Thus, we grow Department into the following:

**Department**(DeptNo, Name, ReportsToDept†)

Where the home relation for the new attribute, ReportsToDept, is that same relation, Department. The tuple of the department that the current department reports to will have a DeptNo that equals the ReportsToDept in the current tuple. Alternatively, ReportsToDept can be NULL if the department does not report to another.

8. **Recursive Relationships, many-to-many:** Suppose



Like non-recursive many-to-many relationships, we will need to create a new relation. Unlike the non-recursive version, we only have one home relation for our two foreign keys. As in the one-to-many version, we will need to choose a new name for at least one copy of the foreign key, since they can't share the same name. The relation for our Person entity would be **Person**(ID, Name)

The new relation created to represent the relationship would be in the following form:

**Friends**(activeFriend†, passiveFriend†)

ActiveFriend and PassiveFriend are foreign keys to the tuple in Person with data for the person that is taking part in the relationship. This can be done in a directed or undirected way, and you probably want to put a comment somewhere about which way you intend to use it.

directed: (Person1, Person2) would not imply (Person2, Person1)

undirected: (Person1, Person2) does imply (Person2, Person1)

ID	Name
1	Regina George
2	Karen Smith
3	Cady Heron
4	Gretchen Wieners
5	Janis Ian

ActiveFriend	PassiveFriend
2	1
3	5
3	2
3	4
4	2
4	3
5	3

- **Summary:**

1. Strong, non-subtype entities
  - New relation, PK is entities identifiers
2. Sub-type entities
  - New relations, PK is supertype identifiers, which are foreign keys to supertype relation
3. Weak entities
  - New relation, PK is concatenation of strong identifier and discriminator.  
Strong Id from the concat is FK to strong relation.
4. **Relationships: Binary 1-1**
  - Put foreign key in either side
5. **Binary 1-m**
  - Put foreign key to one side in the many side
6. **Binary m-m**
  - New relation, PK is concatenation of both entities keys, which also serves as foreign keys to entities.
7. **n-ary 1-1-...-1 (all ones)**
  - New relation, choose n-1 entities for PK, put remaining entity ID as non-prime, but foreign.
8. **n-ary 1-1-...-m (Two ones)**
  - New relation, choose all many legs and one of the one legs for PK, put remaining one leg as non-prime but foreign
9. **n-ary 1-m-...-m (Single one)**
  - New relation, choose all many legs for PK, put remaining one leg as non-prime but foreign

10. **n-ary m-m-...-m (No ones)**
  - New relation, all legs are PK
11. **Handle date entities, and things of that nature**
  - Since we do not create relations for these types of entities, we cannot make them foreign keys, because the home relation will not exist. They can still be part of the PK.
12. **Recursive relationships**
  - We handle these the same, but the foregin key will link to the same relation. Make sure to put a comment somewhere to specify directed or undirected.

### 3.6 MariaDB, SQL

- **DDL:**
  - CREATE TABLE
  - ALTER TABLE
  - DROP TABLE
- **DML:**
  - INSERT
  - UPDATE
  - DELETE
- **MariaDB navigation:**
  - **USE <x>**: select the database <x>
  - **SHOW TABLES** list all of the tables in the current database
  - **DESCRIBE <x>** show the properties of each column of table <x>
  - **SHOW CREATE TABLE <x>** show a CREATE TABLE statement that can be used to reconstruct table <x>

### 3.6.1 DDL

- **Creating a new table with CREATE TABLE:** The basic format of a CREATE TABLE statement. []'s and <>'s are not to be typed. [] indicates that the contents are optional, and the <>'s indicate placeholders:

```
1  CREATE TABLE <table_name> (
2      <attribute> <type> [NOT NULL] [UNIQUE] [PRIMARY KEY], [
3          ↪ ... ]
4      [PRIMARY KEY(<pk attrs>),]
5      [FOREIGN KEY(<attr_here>) REFERENCES
6          ↪ <home_table>(<attr_home>)]
7  );
```

- <table\_name> name of the table
- <attribute> name of the current attribute
- <type> data type of the current attribute
- <pk attrs> comma-separated list of the attributes making up the table's primary key
- <attr\_here> comma-separated list of attributes in the current table forming a foreign key
- <home\_table> name of the home table
- <attr\_home> comma-separated list of attributes in the home table, matching the attributes in <attr\_here>

- **Table / Column names:** When choosing a name for a table or a column, we can use the following characters:

- any of the normal upper or lower case letters (regexp: [A-Za-z])
- an underscore - \_
- a dollar sign - \$
- digits, but only after the first character

The following limits are in place:

- Table names must be unique within the database. They share the same namespace with views.
- Attribute/column names must be unique with each table.
- Unless quoted properly with backticks, reserved keywords cannot be used as identifier

**Note:** These identifiers may or may not be case sensitive, depending on the locale setting of the server.

Generally, the maximum length of an identifier is 64 characters.

- **Data types**

- **INT/INTEGER:** integer values
- **FLOAT:** single precision floating point numbers

- **DOUBLE/REAL**: double precision floating point numbers
- **DECIMAL(i,j)**: decimal numbers, i digits total, j after the decimal point .
- **CHAR(n)**: character string exactly n characters long
- **VARCHAR(n)**: variable-length character string up to n characters long
- **DATE**: date in 'YYYY-MM-DD' format
- **TIME**: time in 'HH:MM:SS' format
- **DATETIME**: date/time in 'YYYY-MM-DD HH:MM:SS' format, no timezone conversion
- **TIMESTAMP**: date/time in 'YYYY-MM-DD HH:MM:SS' format, timezone conversion
- **Column Options**: Here are some common options that can be applied to a column/attribute. They are written right after the type when defining a new column in a CREATE TABLE statement
  - **NULL**: allows NULL to be stored as the value for this attribute (default)
  - **NOT NULL**: prevents NULL from being stored as the value for this attribute
  - **UNIQUE**: ensures that no two tuples have the same value for this attribute
  - **PRIMARY KEY**: declares this attribute to be the entire primary key
  - **AUTO\_INCREMENT**: next-available value auto-assigned for this attribute when not provided
  - **DEFAULT <x>**: sets the default value of the attribute to <x> when not supplied
- **Setting the Primary Key**: There are two ways to set the primary key:
  1. For single-attribute primary keys, you can use the PRIMARY KEY column option. The option may only be used once, and proclaims that the single attribute is the entirety of the primary key.
  2. If you have multiple attributes in the primary key, the only way is to add the separate constraint:

`PRIMARY KEY(<x>, <y>, <z>, <etc>)`

This can also be used for single attribute primary keys.

**Note:** It should be obvious that only one primary key can be set.

- **Comments**: MariaDB supports the following comment syntax
  1. **Pound (#)**:
  2. **Double hyphen (--)**: This is the standard style
  3. **C-style multiline comments (\\* ... \\*)**
- **Quotes**: There are two types of quotes that you may encounter in SQL.
  1. **Quotes for values - single quotes 'value'**: not necessary for numeric values, but can be used without breaking them, always required for string values. If it is ambiguous whether something is a value or an identifier, use these quotes
  2. **Quotes for identifiers - backticks `identifier`**: not necessary for identifiers that follow the rules from above, but can be used anyway, can allow identifier names to contain characters not otherwise allowed. Can allow identifiers to use names that would normally be reserved keywords

**Note:** Notice that identifier in the SQL context is a different thing than an identifier in an ER diagram. Here, identifier will mean the name of some table, column, variable, etc.

- **An example of CREATE TABLE:** Let's go ahead and make the SQL CREATE TABLE statement to create a table for the relation:

**Person(SSN, FNAME, LNAME, PHONE)**

```
1  CREATE TABLE Person(
2      SSN CHAR(9) PRIMARY KEY, # SSN BAD IDEA, PK on same line
3          ↳ (1)
4      FNAME CHAR(20) NOT NULL, # First name
5      LNAME CHAR(20) NOT NULL, # Last name
6      PHONE CHAR(10) # Phone number
7  );
```

The relational schema we started with does not have information on data types or column options other than PRIMARY KEY, so we choose them while creating the table.

- **Setting up a foreign key:** A foreign key links the current table to another table, which we call the home relation.

1. The foreign key must contain all of the attributes of the primary key of the home relation.
2. They may have different names in each of the tables, but there needs to be a match for each.
3. Each of these attributes must have the exact same data type as its counterpart in the home table.

If a table is to contain a foreign key, we include a constraint in our CREATE TABLE statement like the following:

```
1  FOREIGN KEY (<localnames>) REFERENCES
   ↳ <home_table>(<homenames>)
```

This can be done for multiple foreign keys, filling in the placeholders <localnames>, <home\_table>, and <homenames> appropriately for each.

- **Table with foreign key example:** Let's make a table for a subtype of Person, Student:

**Student(SSN†, CLSYEAR, GPA, TOTALHRS)**

```

1 CREATE TABLE Student (
2     SSN CHAR(9) NOT NULL, -- SSN is BAD IDEA
3     CLSYEAR CHAR(9), -- fresh/soph/junior/senior
4     GPA DECIMAL(4,3), -- 4.000, we hope
5     TOTALHRS INT,
6
7     PRIMARY KEY (SSN), -- set up the primary key separately
8         -- (2)
9     FOREIGN KEY (SSN) REFERENCES Person(SSN) -- a Student is
10        -- a Person
11 );

```

**Note:** We need to use SHOW CREATE TABLE to show the get information of the foreign keys of a table.

- **Change existing table schema: ALTER TABLE:** An ALTER TABLE statement will allow you to have the DBMS make changes to the schema of a table that has already been created. It works with various subcommands. The three we will cover are:
  1. ALTER TABLE ADD
  2. ALTER TABLE MODIFY
  3. ALTER TABLE DROP
- **ALTER TABLE ADD:** The ALTER TABLE ADD command can be used to add a new column or new columns to the schema of an existing table.

To add a single column/attribute

```

1 ALTER TABLE <table_name> ADD <attribute> <type>;

```

To add multiple columns/attributes:

```

1 ALTER TABLE <table_name> ADD (<attribute> <type>, ...);

```

- **ALTER TABLE MODIFY:** The ALTER TABLE MODIFY command can be used to change properties of a column/attribute (including type, length, and other column options) in a table that already exists.

```

1 ALTER TABLE <table_name> MODIFY <col_name> <new_options>;

```

- **ALTER TABLE DROP:** The ALTER TABLE DROP command can be used to remove a column/attribute from the schema of a table.

```

1 ALTER TABLE <table_name> DROP <col_name>;

```

- **SHOW TABLES:** In MariaDB/MySQL, if you want to see a list of the tables present in the current database, you can use the command:

```
1 SHOW TABLES;
```

- **DROP TABLE:** To remove a table from the database, we can use the DROP TABLE command.

```
1 DROP TABLE <table_name>;
```

- **Termination of commands (;**): Notice in all sql code examples we have a semi colon after the command / line, this is needed to execute the command.

### 3.6.2 DML except SELECT

- **DML Introduction:** The Data Manipulation Language (DML) is the language used to work with the instance data. In SQL, this means doing things with the rows contained by tables, rather than to the tables themselves. We have
  - **INSERT:** Add a new row to a table
  - **UPDATE:** Change values in an existing row
  - **DELETE:** Remove rows from the table
  - **SELECT:** Display the data stored in rows (In the next subsection)
- **INSERT:**

```
1  INSERT INTO <table_name>
2      VALUES (<value_list>);
3
4  INSERT INTO <table_name>
5      (<attr_list>)
6      VALUES (<value_list>);
7
8  INSERT INTO <table_name>
9      <another_query>;
```

Where

- **<table\_name>**: The name of the table where the row should be added.
- **<value\_list>**: A list of values for the new row. If no **<attr\_list>** is given, then the values are for each of the columns of the table, in order.
- **<attr\_list>**: A list of names of attributes that match up with the values in **<value\_list>**. This allows us to omit optional columns or change the order.
- **<another\_query>**: A query that returns rows, like a **SELECT** statement. The rows returned are inserted into the table.

**Notes:** Without the attribute list, there must be a value in the **VALUES()** for every column, and they have to be in the same order as they had in the table.

Columns not in the attribute list are set to their default value if possible. This is why **PHONE** is **NULL**. This version of the **INSERT** statement is better if you're making SQL that needs to be in a script that is to be run later, as it tolerates more changes to the table schema than the other version.

- **The WHERE clause:**

```
1  ... WHERE <expression> ...
```

When working with DML statements, it will be desirable to be able to work only with specific rows. This can be accomplished using a **WHERE** clause.

The **WHERE** clause is the keyword **WHERE** followed by an expression that evaluates to either true or false. It is included in an SQL query to control which rows are affected by the query

The expression after WHERE is evaluated one time per row. Rows where the expression evaluates as true are included in the operation. Rows where the expression evaluates to false are excluded from the operation

WHERE clauses are generally used in UPDATE, DELETE, and SELECT statements.

- **UPDATE:**

```
1 UPDATE <table_name>
2     SET <attr> = <value> [, <attr> = <value> ...]
3     [ WHERE <expression> ];
```

Where

- <attr>: name of a column to change
- <value>: value to assign to <attr>
- <expression>: expression evaluated for each row to determine if the row is affected

- **DELETE:** To delete the rows without getting rid of the table, use a DELETE statement.

```
1 DELETE FROM <table_name>
2     [ WHERE <expression> ];
```

It is important to realize that all rows are affected by default, so if a WHERE clause is not supplied, all of the rows will be deleted.

- **Views in SQL:** A view in SQL is a virtual table. It does not store its own data, but rather derives it from the other tables (or views) via a query that is a part of its definition.

Views do not contain their own data. They dynamically grab their data from the base tables on demand. Thus, changes to the data in the base tables will be reflected in the views that derive from them automatically

- **CREATE VIEW:**

```
1 CREATE VIEW <view_name>
2     [( <view_col_name> [, <view_col_name>]...)] # can rename
3         ↪ columns here
4     AS SELECT <attr_name> [, <attr_name>] ...
5             FROM <source_table_or_view> [, ...]
6                 WHERE <condition>;
```

The portion after the AS keyword is a SELECT statement, part of the DML that is used to ask the DBMS to show portions of instance data (rows from tables).

Once the view is created, it supports DML queries in most of the same ways a non-virtual table can be. Writing to a view is sometimes possible, but depends on how the SELECT statement that constructed it was formulated. It is generally a better idea to write directly to the base tables.

**Example:**

```
1 CREATE VIEW dekalb_people
2   (SSN, first_name, last_name) # control the names of the
3   ↪ columns as seen in the view
4 AS SELECT SSN, FNAME, LNAME # control which columns are
5   ↪ returned by SELECT
6   FROM Person # get rows from the Person table
7   WHERE ZIP = '60115'; # control which rows make it
8   ↪ into the view
```

- **DROP VIEW:** Although tables and views share the same namespace (so it is not possible to have a view and a table with the same name) and work the same in a lot of queries, DROP TABLE is one of the exceptions and will not work to delete a view. It will give you an error message

Instead, use DROP VIEW, which has generally the same syntax:

```
1 DROP VIEW <viewname>;
```

- **Advantages of Views:**

- Base tables should always be designed in Third Normal Form or better. Views allow us to access them in possibly more convenient ways while still having the benefits of 3NF.
- Views can free users from complicated DML operations, such as joins.
- Users can be denied direct access to base tables, but given access to portions of them through the views. This enhances security

### 3.6.3 DML SELECT

- **SELECT Statement Format:** Two versions of the basic format of a SELECT statement follow.

```

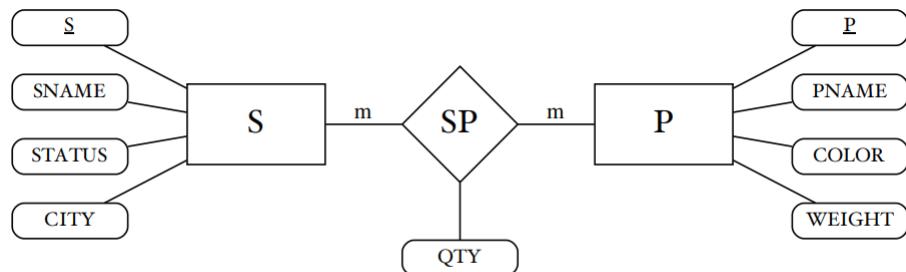
1  SELECT [DISTINCT|ALL] <column_list> # most common, show row
   → data
2   FROM <table_list>
3   [ WHERE <where_exp> ]
4   [ GROUP BY <group_key> ]
5   [ HAVING <having_exp> ]
6   [ ORDER BY <sortcols> ] ;
7
8  SELECT <anyexpression> ; # show results of the supplied
   → expression

```

Where

- **<column\_list>**: comma separated list of the columns to show in the results,  
\* for all columns
- **<where\_exp>**: boolean expression evaluated once per row to determine whether the row is included
- **<group\_key>**: comma-separated list of the columns to use when grouping the rows
- **<having\_exp>**: boolean expression evaluated once per group to determine whether the group is included
- **<sortcols>**: comma-separated list of the columns to sort by (most important comes first)
- **<anyexpression>**: the expression whose results should be displayed

- **Example data:** Here we have a simple database to use for the examples that follow. It tracks suppliers and the parts they supply.



ER diagram representing the example database.

Supplier Info S(\$, SNAME, STATUS, CITY)

Part Info P(P<sub>1</sub>, PNAME, COLOR, WEIGHT)

Supplied Parts SP(S<sup>†</sup>, P<sup>†</sup>, QTY).

The S table contains the information on the suppliers themselves.

S	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

The P table contains information on parts.

P	PNAME	COLOR	WEIGHT
P1	Nut	Red	12
P2	Bolt	Green	17
P3	Screw	Blue	17
P4	Screw	Red	14
P5	Cam	Blue	12
P6	Cog	Red	19

The SP table contains information on which suppliers supply which parts, and how many.

S	P	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

- **Example query:**

Get supplier numbers and status for suppliers in Paris.

```
1  SELECT S,STATUS  
2    FROM S  
3   WHERE CITY = 'paris';
```

Get part numbers for all parts supplied

```
1  SELECT P FROM SP;
```

Adding the DISTINCT keyword can prevent duplicate output rows from being shown.

```
1  SELECT DISTINCT P  
2      FROM SP;
```

List the full details of all suppliers.

```
1  SELECT * FROM S;
```

List supplier numbers for all suppliers in Paris with a STATUS greater than 20.

```
1  SELECT * FROM S  
2      WHERE CITY = 'Paris' AND  
3          STATUS > 20;
```

- **Relational Operators in SQL:**

- = is equal to
- < less than
- <= less than or equal to
- > greater than
- >= greater than or equal to
- <> or != not equal to

- **Compound Logical Operators:**

- AND
- OR
- NOT

- **The ORDER BY clause:** Adding the ORDER BY clause allows us to enforce a sorting order upon our results.

```
1  ORDER BY <attrs>
```

Where <attrs> is a comma-separated list of the attributes to base our sorting upon.

After each attribute, you have the option to add either DESC (for descending) or ASC (for ascending) to affect the sort direction for each attribute. The default sort direction is ascending, if not specified.

The first attribute listed is the most important, and any subsequent attributes is only sorted upon if there are multiple rows in which the values for the previous attributes before them were all the same.

- **ORDER BY example:** List the supplier numbers and status for suppliers in Paris in descending order of status.

```

1   SELECT S,STATUS FROM S
2       WHERE CITY = 'Paris'
3           ORDER BY STATUS DESC;

```

- **Cartesian Product in SQL:** For two sets  $A = \{a, b, c\}$ , and  $B = \{d, e, f\}$

$$A \times B = \{(a, d), (a, e), (a, f), (b, d), (b, e), (b, f), (c, d), (c, e), (c, f)\}.$$

This is relevant because the Cartesian Product is used in SQL when we SELECT from multiple tables. When this happens, the sets (like  $A$  and  $B$ ) to be combined are the tables, and the items inside of them are the tuples/rows they contain.

When the Cartesian Product is done on two tables,

- The width of the result is the sum of the widths (in columns) of both of the tables.
- The length (in rows) of the result will be the product of the lengths of both of the tables.

**Note:** The Cartesian Product is an associative operation

- **Aliases and the dot operator:** When we select certain relations, we can give aliases to them and then reference their attributes with a dot (similar to how you access C++ member functions)

This is important because if we take the cartesian product of the same relations, we need to give them aliases (they would otherwise have the same name)

- **Aliases with AS:** We can also use AS to assign aliases

```

1   SELECT col AS c1
2       FROM relation1 AS r1
3           ...

```

- **Using the dot in general:** In general, even without aliases, we can use the dot to refer to relations attributes, this is crucial if we select on two relations with matching attribute names.
- **Cartesian Product Example:** S has 5 rows of 4 columns. The Cartesian product, `SELECT * FROM S T1, S T2;` returns 25 rows, each with 2 sets of the columns in S, for a total of 8 columns. They don't all fit on the page here.

T1.S	T1.SNAME	T1.STATUS	T1.CITY	T2.S	T2.SNAME	T2.STATUS	T2.CITY
S1	Smith	20	London	S1	Smith	20	London
S2	Jones	10	Paris	S1	Smith	20	London
S3	Blake	30	Paris	S1	Smith	20	London
S4	Clark	20	London	S1	Smith	20	London
S5	Adams	30	Athens	S1	Smith	20	London
S1	Smith	20	London	S2	Jones	10	Paris
S2	Jones	10	Paris	S2	Jones	10	Paris
S3	Blake	30	Paris	S2	Jones	10	Paris
S4	Clark	20	London	S2	Jones	10	Paris
S5	Adams	30	Athens	S2	Jones	10	Paris
S1	Smith	20	London	S3	Blake	30	Paris
S2	Jones	10	Paris	S3	Blake	30	Paris
S3	Blake	30	Paris	S3	Blake	30	Paris
S4	Clark	20	London	S3	Blake	30	Paris
S5	Adams	30	Athens	S3	Blake	30	Paris

- **Cartesian product example:** For each part supplied, get the part number and names of all the cities supplying the part. (This is a join which pulls together data from multiple tables.)

```

1  SELECT DISTINCT P, CITY
2    FROM SP,S
3   WHERE SP.S = S.S

```

List the supplier numbers for all pairs of suppliers such that two suppliers are located in the same city.

```

1  SELECT T1.S, T2.S /* one S from each side */
2    FROM S T1, S T2 /* cartesian product of S with S, giving
3      ↳ name to each side */
4   WHERE T1.CITY = T2.CITY /* same city for both suppliers
5      ↳ */
6        AND T1.S < T2.S; /* avoid duplicate pairs; lower S on
7      ↳ left */

```

- **Multiple-row subqueries:** Multiple-row subqueries are nested queries that have the potential to return more than one row of results to the parent query. Most commonly used in WHERE and HAVING clauses

**Note:** Must be used with multiple-row operators.

- **SQL Sets:** In an SQL statement, we can denote a set with a list of values inside parentheses.
- **Multiple Row Subqueries: IN Set Operator:** IN is a set operator used to test membership.

The IN operator will have value on its left, and a set on its right. It will evaluate to true if the value from the left hand side is present in the set provided on the right.

Example	Evaluates to
'S1' IN ('S2','S3','S1')	true
'S1' IN ('S2','S3','S4')	false
4 IN (2,1,6,4,5)	true
3 IN (1,5,6,10)	false

When a multiple-row subquery is evaluated, its results are inserted into its parent query as a set. We can use set operations like IN to fit those results into our query

- **Multiple-row subqueries: Example:** List the supplier names for suppliers who supply part P2. (This time using a subquery.)

```

1  SELECT SNAME
2    FROM S
3    WHERE S IN          # IN operator used to check
      ↪ current S against the list
4        ( SELECT S      # this is the subquery
5          FROM SP         # which returns a list (set)
6          WHERE P = 'P2' ); # containing all the suppliers
      ↪ that supply part P2.

```

**Note:** The innermost subqueries are the first to run. It returns

S
—
S1
S2
S3
S4

Which is inserted into the parent query as ('S1', 'S2', 'S3', 'S4'), in the position where the subquery that returned the results was found.

Thus, after the subquery is run, the outer query effectively becomes:

```

1  SELECT SNAME
2    FROM S
3    WHERE S IN          # IN operator used to check
      ↪ current S against the list
4        ('S1', 'S2', 'S3', 'S4'); # <-- results of subquery
      ↪ inserted in place

```

Which would have the following results:

SNAME
Smith
Jones
Blake
Clark

- **Set Operators: ALL and ANY:** The ALL and ANY operators modify the normal relational (in the comparison sense) operators to work on sets.

If we want to compare a value with every item in the set and reduce the answers to a single true/false using the AND operation, we can use ALL

```
1 <value> <relop> ALL (set)
```

If we want to compare a value with every item in the set and reduce the answers to a single true/false using the OR operation, we can use ANY.

```
1 <value> <relop> ANY (set)
```

- **Set Operator: EXISTS:** The EXISTS operator is a unary operator working on sets that is used to determine whether the set supplied is non-empty. Once again <set> is either an explicitly written set or a multi-row subquery

```
1 EXISTS (set)
```

- Evaluates to true if the set is non-empty (contains at least one element)
- Evaluates to false if the set is empty (no elements inside)

- **Set operator: NOT EXISTS:** EXISTS, when used in conjunction with the logical inversion operator, NOT, enables two types of queries that were difficult before
  - Queries involving the set difference operation  $\{a, b, c, d, e\} - \{b, c\} = \{a, d, e\}$
  - Queries that involve the concept of every

only include rows where the subquery is EMPTY

- **Union:** The UNION operator causes two sets to be merged, the set union.

```
1 SELECT P
2   FROM P
3 WHERE WEIGHT > 18 # first SELECT returns only P6
4 UNION
5 SELECT P
6   FROM SP
7 WHERE S = 'S2'; # second query returns P1, P2
```

- **Union caveat:** You should be careful in situations where the domain of a column matters, as UNION will put rows together whether the columns match in type/purpose or not.
- **Group Functions:** Group functions are sometimes referred to as aggregate or multiple-row functions. They take a list of columns as an argument, with an optional DISTINCT or ALL inside before those columns are listed.
  - **SUM(<x>):** add up the value of column <x> in all of the rows of each group
  - **AVG(<x>):** find the average value of column for each group
  - **COUNT(<x>):** count how many rows there are (usually <x> is a \* here.)

- **MAX(<x>)**: returns the maximum value of column <x> for each group
- **MIN(<x>)**: returns the minimum value of column <x> for each group
- **STDDEV(<x>)**: returns the standard deviation of column <x> for each group
- **VARIANCE(<x>)**: returns the variance of column <x> for each group

All of these functions will return a single value for each group present.

If no GROUP BY clause is included, then there is only a single group, which contains all the rows of the query. The GROUP BY clause will allow that to be divided into subgroups.

- **Group function example: COUNT**: Find out the number of suppliers

```
1 SELECT COUNT(*) FROM S;
```

- **DISTINCT with group functions**: Get the total number of suppliers currently supplying parts

If you want to count only distinct values, we can do that with DISTINCT

```
1 SELECT COUNT(DISTINCT S) FROM SP;
```

- **WHERE clause with group functions**: The WHERE clause is evaluated BEFORE any groups are formed.

```
1 SELECT COUNT(*)
2   FROM SP
3 WHERE P = 'P2'; # the value of P is known before
      ↳ grouping, so WHERE works
```

- **The GROUP BY clause**: The GROUP BY clause in a SELECT statement takes the following form:

```
1 GROUP BY <attrs>
```

It will cause the SELECT statement to examine the rows in its result set, and gather the ones that match on their values for the columns in <attrs> into subgroups.

```
1 SELECT SUM(QTY) FROM SP
2     GROUP BY P; # make a subgroups for each part
3
4 SELECT P, SUM(QTY) FROM SP # added P to be shown
5     GROUP BY P; # make a subgroup for each part
```

- **Group by caveat**: However, if we try to display columns that aren't part of the <attrs> of the GROUP BY and aren't calculated by a group function, we begin to have problems.

```

1   SELECT P, S, QTY, SUM(QTY) FROM SP GROUP BY P; # P is good,
      ↵ but look at S and QTY

```

P	S	QTY	SUM(QTY)
P1	S1	300	600
P2	S1	200	1000
P3	S1	400	400
P4	S1	200	500
P5	S1	100	500
P6	S1	100	100

What do the values of S and QTY mean in this grouped context? Nothing! They are not relevant or correct. Is S1 the only supplier for all of the groups? The SP table indicates no. Is the QTY there valid for P1? No, the correct answer for total P1 supplied is in the SUM(QTY).

There is a distinct value of S and a value of QTY for every row in each subgroup. That is many values, and only one place to show them in - it's not atomic. Unfortunately the DBMS is just choosing one to show anyway, but it has no meaning, and such situations should be avoided.

- **HAVING clause:** Just as the WHERE clause could be used to filter individual rows based on whether they evaluated true for its expression, the HAVING clause allows us to filter out groups based on values that pertain to the group.

```

1   HAVING <expr>

```

For each group in the results, the HAVING expression, <expr> is evaluated, and only groups where <expr> is true will be included in the final output.

The reason HAVING is necessary is that the WHERE clause is evaluated BEFORE the groups are formed, and is not able to work with values that don't exist until after it has already finished.

- **Example with HAVING:** List the part numbers for all parts supplied by more than one supplier.

```

1   SELECT P
2     FROM SP
3   GROUP BY P
4   HAVING COUNT(*) > 1;

```

- **Single-Row Subqueries:** Single-row subqueries are subqueries that return a single value (ONE ROW with ONE COLUMN).

Like the multiple-row subqueries, they are evaluated and then their results are used in the parent query that contained them.

They don't need to use the multiple-row operators to work.

- **Single-Row Subquery as a column:** SELECT Title, Retail, (SELECT AVG(Retail) FROM Books) # third column will have result 'Overall Average' # with a changed title

Title	Retail	Overall	Average
The Princess Bride	39.99	42.00	
The Life of Pi	3.14	42.00	
The Hitchhiker's Guide	29.50	42.00	
...	...	...	...

Having the call to the group function AVG would normally reduce the results to a single row per group, but it happened inside a subquery, so it did not change the outer query. This can be useful when you really want to know an aggregate value but don't want to condense your rows.

- **Single-Row Subquery in a WHERE clause:** Let's use a bookstore as an example. If you knew the ISBN of a book and wanted to run a query to find all of the books that are more expensive than it, you could use a subquery to find out the cost of the book with that ISBN and then compare that value with its result.

```
1  SELECT Title, Cost
2    FROM Books
3    WHERE Cost > # compare the cost of current row with
4        ↪ result of subquery
5        (SELECT Cost # only the Cost returned -- single
6            ↪ column
7            FROM Books
8            WHERE ISBN = '1328948854'); # ISBN is PK -- single
9            ↪ row
```

- **Single-Row Subquery in a HAVING clause:** Since the result of the subquery is inserted in place, it will work anywhere a single value makes sense. This includes use as part of a HAVING clause. Using the same book database from the previous slide:

```
1  SELECT Category,
2      AVG(Retail - Cost) 'Average Profit' # calculate average
3          ↪ profit of all books, change the label
4    FROM Books
5    GROUP BY Category
6    HAVING AVG(Retail - Cost) > # compare cost of each group
7        ↪ with result of the subquery
8        ( SELECT AVG(Retail - Cost) # finds the average
9            ↪ profit for books in LIT
10           FROM Books
11           WHERE Category = 'LIT' );
```

- **Single-Row Subquery example 1:** List the supplier numbers for suppliers who are located in the same city as supplier S1

```

1   SELECT S
2     FROM S
3      WHERE CITY = # compare each row with result of subquery
4          ( SELECT CITY # find out which city S1 is in
5            FROM S
6            WHERE S = 'S1' );

```

- **The LIKE operator:** So far, all of the string comparisons we've done have been with the `=` operator, which tests for strict equality. (Locale settings determine whether it's a case sensitive or case insensitive match.)

Using just `=`, we'd have to have a lot of OR's strung together to have any kind of flexibility.

If we have a pattern to be matched, we generally won't use `=`, but rather the LIKE operator.

```

1   <val> LIKE <pattern>

```

The LIKE operator will return true when `<val>` matches the pattern specified in `<pattern>`.

- **Patterns with LIKE:** The patterns that LIKE uses to check your values against are defined using these special characters.

- `%` any zero or more characters can fit here without breaking the match
- `_` any single character can fit here without breaking the match
- `\` escape the next character
- `\%` escaped `%`, so only match the actual `%` character here
- `\_` escaped `\_`, so only match the actual `_` character here
- `\\\` escaped `\`, so only match the actual `\` character here

Any characters not in this list will only match themselves.

- **LIKE: Character classes and union (or):** You can specify a list or range of characters with square brackets.

```

1   SELECT ...
2     WHERE ... LIKE "_[abc]%""
3     WHERE ... LIKE "_[a-z]%""

```

- **Negating character classes:** To invert a character class, we can use `!`

```

1   SELECT ...
2     WHERE ... LIKE "_[!abc]%""

```

- **List suppliers whose name starts with the letter 'S':**

```

1  SELECT * FROM S
2  WHERE SNAME LIKE `S%`;

```

- **Single-valued (non-group) functions:** Unlike the aggregate functions, these functions won't make your results collapse based on groups. They are evaluated, and their value is inserted in place.

- **LOWER(<str>):** Returns copy of <str> but all lowercase
- **UPPER(<str>):** Returns a copy of <str> but all uppercase
- **SUBSTR(<str>, <pos>, <len>):** Returns a copy of the substring of <str> starting at its <pos>th position, <len> characters long
- **LENGTH(<str>):** Returns the length in characters of the string, <str>
- **LPAD(<str>, <len>, <sp>):** Returns <str> fit into <len> characters, padding with <sp> on the left if necessary
- **RPAD(<str>, <len>, <sp>):** Returns <str> fit into <len> characters, padding with <sp> on the right if necessary
- **ROUND(<num>, <pos>):** Returns the number <num>, rounded to <pos> digits after the decimal point
- **CONCAT(<str>, [...]):** Returns the concatenation of the strings <str>, in order.
- **SOUNDEX(<str>):** Returns a string containing a code that can be used to compare how <str> sounds like other strings.

These functions can be nested however you'd like. Just like C++, they're evaluated from the inside out.

- **JOINS:** We've seen joins in some of our examples already.

A join is an operation that takes information from separate tables and combines it into one set of results.

There are two basic types of join:

1. **Inner join**
2. **Outer join**

For either of these types of join, it is possible to join a table with itself, in which case we call it a self join.

- **Change to S:** To make things interesting in these joins, let's add a new supplier, S7, to our S table. They won't supply anything yet so leave P and SP unchanged.

We have already done a few inner joins in the earlier examples,

- **Inner join:** With an inner join, only lines that match up with each other in both tables will be a part of the result. Earlier, we accomplished this by putting together two tables with the Cartesian Product, and then using a WHERE clause to make sure only things that matched on the foreign key were retained.

```

1  SELECT S.S,P,SNAME,QTY
2    FROM S,SP # Cartesian product of S with SP
3    WHERE S.S = SP.S; # only keep rows matching S=S

```

Can be written as

```

1  SELECT S.S,P,SNAME,QTY
2    FROM S JOIN SP # replace the comma with the keyword JOIN
3      ON S.S = SP.S; # WHERE becomes ON for the foreign key

```

- **Outer Join:** An outer join can be more flexible than an inner join. It will contain everything that the inner join contained, but one or both of the two tables involved will be special, and will have at least one row in the results whether it matched the other side or not. The values for the missing side will be filled with NULL since there is no relevant value.

Here we have the same query from before, but as an outer join instead of an inner join.

```

1  SELECT S.S,P,SNAME,QTY
2    FROM S LEFT JOIN SP # LEFT means table on LHS of JOIN is
3           ↪ the strong one
4    ON S.S=SP.S;

```

LEFT means the table on the left-hand side of the JOIN keyword is the strong one. RIGHT would mean the RHS is strong. In some dialects of SQL, you can use FULL to make both strong, but this does not work in MariaDB. You can accomplish something similar with a UNION if needed.

- **The LIMIT Clause:** the LIMIT clause is used to restrict the number of rows returned by a query. When you specify LIMIT 50, it tells the database to return only the first 50 rows of the result set.

```

1  SELECT * from sometable LIMIT 50; // Queries the first 50
   ↪ rows

```

**Note:** Redundant if the number of rows in the relation is less than the limit restriction

- **IS NULL and IS NOT NULL:** A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

It is not possible to test for NULL values with comparison operators, We will have to use the IS NULL and IS NOT NULL operators instead.

```
1  SELECT ...
2    WHERE ... IS NULL;
3
4  SELECT ...
5    WHERE ... IS NOT NULL;
```

- **BETWEEN operator:** the BETWEEN operator is used to filter the result set within a specified range. It works for numbers, dates, and text values.

```
1  SELECT ...
2    WHERE ... BETWEEN val1 AND val2;
```

We can of course negate this with NOT

```
1  SELECT ...
2    WHERE ... NOT BETWEEN val1 AND val2;
```

### 3.7 Frontend: Html

- **Why HTML for databases?**: HTML is a relatively easy way to provide a graphical user interface to a user. This interface can solicit from the user the data that is needed to perform operations on the database, as well as present to the user the data that is returned in a structured way
- **HTML vs. XHTML**: As a web browser goes through evaluating all of the HTML code it encounters, it has several possible modes of operation.
  - The HTML-only parser mode is very loose. If you make a mistake, it will try to guess what you meant and continue anyway. This can cause weird behavior with no warning that anything was wrong.
  - The XHTML parser performs more checks, which can help you locate and fix your problem areas more quickly.
- **Tree-Like Structure**: A well-formed HTML document should have a structure based upon the data structure known as a tree. This tree is formed by the elements present in the document, which can be used as containers of one or more other elements.
- **Elements**: There are three basic types of elements that you will see in HTML.
  1. **Plain Text**: This is just normal text, in whatever encoding (ASCII, EBCDIC, Unicode) the document is using. Note that HTML-specific characters will need to be properly escaped.
  2. **Comments**: These start with <!-- and end with -->
  3. **HTML tags**: These are special keywords, surrounded by angle brackets ( < and >). For each tag element, there will be a start tag and an end tag

The start, or open tag is, in its most simple form, just the keyword in angle brackets, eg. <tag>.

The end, or close tag, when written separately, has a forward slash / after the first angle bracket, eg. </tag>.

- **Plain Text**: Plain text elements will be displayed (or not displayed) in a manner appropriate to where they occur in the document.
- **A Note on Comments**: Comments are nice, and should still be used when they would be helpful. However, since the comments are sent to the user from the server every time the page is loaded, it might be beneficial, from a performance standpoint, to keep them to a minimum.
- **HTML Tags**: They can be written separately, usually when they are surrounding something, such as <div>Text Here</div> - note the / at the beginning of the end tag.

If there is nothing between the start and end tag, there is a shorthand, <meta/> - the / at the end of the tag means that the end tag is included with the opening tag, so it is equivalent to <meta></meta>, with nothing inside.

If the element is generally expected to have content, you should not use this shortcut.

- **Attributes**: It is possible to specify attributes of an HTML tag. These get written as part of the start, or opening tag, and they can either be name=value pairs (attrname1 or attrname2 below), or just the name of an attribute if it is boolean if it is desired to set it to true (boolattr below).

```
1 <tag attrname1="attr1value" attrname2="attr2value" boolattr>
```

Each type of tag will have a different set of applicable attributes. The values specified for these attributes will control the behavior of the element.

- **Basic HTML Document:**

```
1 <html>
2   <head>
3     <title>Page Title Here</title>
4     <meta charset="UTF-8"/>
5   </head>
6   <body>
7     <h1>Big Heading Here</h1>
8     <p>A paragraph here.</p>
9   </body>
10 </html>
```

- **<html> Element:** The <html> element should be the one of the first elements to appear, and will contain all of the other elements in your document.

There should be only one per document.

- **<head> Element:** The <head> element is meant to contain header information. Among other things, this can include:

- <title> The title that will be displayed for this document in either a tab button or the window caption. The title is the text between the start and the end tag, not an attribute.
- <meta> Various information about the document that you would like to make available to search engines, etc.

Notice that any text you put in the <head> tag will not show up when the document is rendered.

- **<body> Element:** The <body> element will contain the body of your document. Any text that you would like to display to the user should be contained by this element
- **Working with Text:** Before we start working with text, there are a few things we should address.

1. whitespace (spaces, tabs) is mostly ignored after the first space
2. newline character in the source do not get shown as newlines when the document is displayed
3. characters that could be interpreted as part of the HTML markup may need to be escaped to show up as text-only

- **Special Characters and Escape Codes:**

- &gt;; greater than symbol (because used to identify tags)
- &lt;; less than symbol (because used to identify tags)
- &amp;; ampersand symbol (because used for these codes)

- &nbsp; non-breaking space (this is a space that will not be ignored)
- &copy; copyright symbol, © (not a part of ASCII alphabet)
- &#9999; Unicode character with decimal value 9999
- &#xffff; Unicode character with hexadecimal value ffff
- **Headings - <h1> to <h6>**: The tags, <h1> through <h6>, provide various levels of headings. <h1> is the first heading, and will have the largest text, with the others getting progressively smaller in importance.
- **Paragraphs - <p>**: The <p> element is used to indicate that the text it contains is meant to be a paragraph. In general, this means that it will be printed as a single block of text, wrapping automatically as its text reaches the end of the line.

What if we wanted to format an address, with the name on one line, the street address on another, and the city/state/zipcode below that?

```

1  <p>Northern Illinois University</p>
2  <p>1425 West Lincoln Highway</p>
3  <p>DeKalb, IL 60115</p>

```

- **Line breaks:** There is a special tag, <br />, the line break, that can help in this situation.

```

1  <p>Northern Illinois University<br />
2      1425 West Lincoln Highway<br />
3      DeKalb, IL 60115</p>

```

**Note:** Some purists will argue that the <br /> tag should not be used, because it's purely visual, and the purpose of HTML is to denote structure. They are not entirely wrong, but it's more work to do it other ways, so it will work in a pinch.

- **Unordered Lists - <ul>**: It is possible to create unordered (bulleted) lists with <ul> and <li> elements.

```

1  <ul>
2      <li>No</li>
3      <li>Particular</li>
4      <li>Order</li>
5  </ul>

```

- **Ordered Lists - <ol>**: It is possible to create ordered (numbered) lists with <ol> and <li> elements.

```

1  <ol>
2      <li>First</li>
3      <li>Second</li>
4      <li>Third</li>
5  </ol>

```

The type of numeral used for ordered lists can be controlled by CSS, or via the type attribute of the `<ol>` tag. Its possible values are (1, A, a, I, and i).

- **Images - `<img>`:** It is possible to embed images into your HTML documents. This is done with the `<img>` tag. On its own, the `<img>` element cannot do much, because it needs more information. It will be necessary to supply the path to the file containing the image to display as the `src` attribute.

In HTML, paths can be specified in any of three ways:

- **relative paths** - these are paths starting from the directory the current HTML document is being loaded from. These will never begin with a forward slash, `/`.
- **absolute paths (local)** - these are paths starting from the root of the webserver that is serving the current document, these will always begin with a forward slash, `/`.
- **absolute paths (remote)** - these are paths that are on other servers. They will begin with the protocol specifier of the server. (`http://`, `https://`, `ftp://`, etc.)

With all three of these types of paths, the steps along the path will be separated by a forward slash, `/`.

- **Images - Useful Attributes:** Some useful attributes for the `<img>` element:

- **height** - height of image (in pixels) or as a percentage of screen (with a `%` at the end)
- **width** - width of image (in pixels) or as a percentage of screen (with a `%` at the end)
- **alt** - alternate text, which will show when hovering, or may be read aloud for accessibility purposes

The `height` and `width` attributes can be used to scale the image, but the same image file is downloaded either way. It is best to make the image exactly the size you intend to display it using some image editing tool, rather than resizing it on the page. The best use for `height` and `width` is to allow the browser to know the size of the image before downloading it, so the page doesn't have to resize as it loads.

- **Links:** One of the features that originally made HTML popular was the idea of the hyperlink. Now we just call them links.

They are portions of the document that can be clicked to navigate to somewhere else.

They are added to the HTML using the anchor tag, `<a>`. The path for the destination of the link is specified with the `href` attribute, and the text that forms the link will be whatever is contained by the anchor element.

```
1 <a href="other.html">Relative Path Link</a>
2 <a href="/some.html">Absolute Path Link</a>
3 <a href="http://www.niu.edu"> Link on another server</a>
```

- **Tables:** Something that comes up very often when dealing with relational databases is the table. There is support for drawing them in HTML. This is done with a set of four elements that work together:

- `<table>` - This element starts a table, the others should only occur inside one of these.
- `<tr>` (table row) - This element is a row within a table. It will contain one or more of either of the following two.
- `<th>` (table heading) - This element is a table heading cell. The text to display as a column label should go inside.
- `<td>` (table data) - This element is a table data cell. The text inside should be the data to be shown in the cell.

Tables in HTML are always specified in row-major order. The cells go from left to right inside of rows. There is not a way to specify it columnwise.

```

1  <table>
2    <tr>
3      <th>Topping</th>
4      <th>Price</th>
5    </tr>
6    <tr>
7      <td>Sausage</td>
8      <td>$0.90</td>
9    </tr>
10   <tr>
11     <td>Pepperoni</td>
12     <td>$1.00</td>
13   </tr>
14 </table>

```

- **Forms:** In HTML, forms are used to collect user input and submit it to a server for processing. They are fundamental to creating interactive web applications, allowing users to input information, like login credentials, feedback, or search queries

```

1  <form action="submit_page.php" method="post">
2    <!-- Form elements like input fields, buttons, etc. go
     -->
3  </form>

```

- **action:** Specifies the URL where form data should be submitted.
- **method:** Defines the HTTP method for submitting the form. Common values are:
  1. **GET:** Sends form data as URL parameters (query strings), typically used for non-sensitive data.
  2. **POST:** Sends form data in the request body, more secure for sensitive data.
- **Input:** The `<input>` tag in HTML is used to create interactive fields in forms that collect user input. The `<input>` tag is versatile and supports various types of inputs by setting the `type` attribute to specify the kind of data the field should accept.
  1. **Text Input:** Collects single-line text input, like names or emails.

```
1 <input type="text" name="username" placeholder="Enter  
→ your name">
```

2. **Password Input:** Similar to text but hides characters as they're typed.

```
1 <input type="password" name="password"  
→ placeholder="Enter your password">
```

3. **Radio Buttons:** Allow users to select a single option from a set.

```
1 <input type="radio" name="gender" value="male"> Male  
2 <input type="radio" name="gender" value="female"> Female
```

4. **Checkboxes:** Let users select multiple options independently.

```
1 <input type="checkbox" name="subscribe"  
→ value="newsletter"> Subscribe to newsletter
```

5. **Submit Button:** Sends the form data to the server.

```
1 <input type="submit" value="Submit">
```

6. **Color Picker:** Allows users to pick a color from a color wheel or enter a color code.

```
1 <input type="color" name="favcolor" value="#ff0000">
```

7. **Date Picker:** Provides a calendar interface for selecting a date.

```
1 <input type="date" name="birthdate">
```

8. **Email Input:** Accepts and validates email addresses.

```
1 <input type="email" name="email" placeholder="Enter your  
→ email" required>
```

9. **File Upload:** Allows users to upload files from their device.

```
1 <input type="file" name="profilePicture">
```

10. **Hidden Input:** Stores data that isn't visible to the user but is submitted with the form.

```
1 <input type="hidden" name="userID" value="12345">
```

11. **Image Button:** Acts as a submit button but uses an image instead of text.

```
1 <input type="image" src="submit_button.png" alt="Submit"
→ width="50" height="50">
```

12. **Month Picker:** Allows users to select a specific month and year.

```
1 <input type="month" name="startmonth">
```

13. **Number Input:** Accepts numeric values, optionally within a specified range.

```
1 <input type="number" name="quantity" min="1" max="10"
→ step="1">
```

14. **Range Slider:** Allows users to select a number within a specified range by sliding a handle.

```
1 <input type="range" name="volume" min="0" max="100"
→ step="10">
```

15. **Reset Button:** Clears all input fields in the form to their default values.

```
1 <input type="reset" value="Reset">
```

16. **Search Field:** Provides a field for entering search queries.

```
1 <input type="search" name="query"
→ placeholder="Search... ">
```

17. **Telephone Input:** Used for entering telephone numbers, with optional pattern for specific formats.

```
1 <input type="tel" name="phone"
→ placeholder="123-456-7890"
→ pattern="[0-9]{3}-[0-9]{3}-[0-9]{4}">
```

18. **URL Input:** Accepts and validates URLs.

```
1 <input type="url" name="website"
→ placeholder="https://example.com">
```

- **Form validation:** HTML5 provides several attributes for form validation:

- **required:** Ensures a field must be filled out.

```
1 <input type="email" name="email" required>
```

- **pattern:** Specifies a regular expression pattern for input validation.

```
1 <input type="text" name="username"  
→ pattern="[A-Za-z]{3,}>
```

- **min and max:** Define minimum and maximum values for number fields.

```
1 <input type="number" name="age" min="1" max="100">
```

- **Dropdown List (Select):** Allows users to select from a list of options.

```
1 <select name="country">  
2   <option value="usa">USA</option>  
3   <option value="canada">Canada</option>  
4 </select>
```

- **Textarea:** For multi-line text input, such as comments or descriptions.

```
1 <textarea name="comments" placeholder="Enter your  
→ comments"></textarea>
```

- **Buttons in html:** We can create buttons with the button tag

```
1 <button> </button>
```

- **Nav tags:** The `<nav>` tag in HTML is used to define a section of navigation links on a web page. It's a semantic element introduced in HTML5 that helps improve the structure and accessibility of a website by explicitly marking sections dedicated to navigation. This tag is typically used for primary navigation elements like menus, tables of contents, or other links that users can use to navigate a site.

```
1 <nav>  
2   <ul>  
3     <li><a href="index.html">Home</a></li>  
4     <li><a href="about.html">About Us</a></li>  
5     <li><a href="services.html">Services</a></li>  
6     <li><a href="contact.html">Contact</a></li>  
7   </ul>  
8 </nav>
```

- **Div containers:** The most commonly used container in HTML, `<div>` stands for "division."

It's a generic container that can hold any other HTML elements.

Used for grouping elements to apply CSS styles or JavaScript functionality.

```
1 <div class="container">
2     <h1>Welcome to My Website</h1>
3     <p>This is a sample container.</p>
4 </div>
```

- **Section container:** A semantic container used to group related content.

Useful for sections like articles, services, or product listings.

```
1 <section class="features">
2     <h2>Features</h2>
3     <p>Our product offers the following features:</p>
4 </section>
```

- **<header>, <footer>, and <aside> Tags:**
  - **<header>:** For header content, like logos or navigation.
  - **<footer>:** For footer content, like copyright or contact information.
  - **<aside>:** For content related to the main content, like sidebars or ads.
- **Span:** The  element, can be used to group inline elements only. So, if you have a part of a sentence or paragraph which you want to group together

### 3.8 PHP

- **Components of web services:**
  - **Web browser:** formats and displays Web pages
  - **Web server:** sends Web pages to browsers and lets site visitors enter and request information
  - **Information server:** accepts requests from the Web server and uses its stored data to respond appropriately
- **Database server:** A database server is a kind of information server, it stores information in databases

Web servers, etc. connect to the database server to send queries or update data

- **Web pages and HTML:** An HTML document is used to create the format and structure of a Web page

HTTP is a communication protocol that specifies how two or more things are expected to interact on the Web

- **Web server:** a computer system that responds to requests for Web pages by processing the requests and returning a new Web page to the browser
- **Preparing to use php:** The php language was designed to help developers create dynamic and data-driven web pages

php interacts with one main external tool, the MySQL database management system, to access data stored in a database

MySQL must be installed on a functional Web server to interact with php, but this is a relatively easy step in setting up the php environment

php is a server-side scripting language that you can embed into HTML documents. You can also embed HTML in php scripts

php scripts are parsed and interpreted on the server side of a Web application

php is popular with web developers and web designers alike, and is powerful and easy to use

To start working with php, you can create a script that contains HTML code

- **Methods of using php in html:** There are two ways

```
1  <?php
2      ...
3  ?>
4
5  <SCRIPT LANGUAGE="php"
6      ...
7  </SCRIPT>
```

- **PHP example:** We can create a file called example.php, that is essentially an html file, but can contain php code in <?php ... ?> blocks. For example,

```

1 <body>
2   <?php
3     print("Hello world");
4   ?>
5 </body>

```

Note that lines in php are terminated with a semicolon

- **Displaying php Output:** php has two functions that allow display: echo and print

The only difference between echo and print is that the print function returns a 1 or 0 integer (denoting success or failure, respectively), for the contents of the function being displayed

Also, be aware that if you want to send php reserved characters (such as double quotations) to the Web browser within the echo command, you must use the backslash character

- **Defining php Variables:** Variables in php are preceded with a dollar sign (\$) and contain either letters or numbers

php is called a loosely typed programming language, meaning that you don't have to predefine your variables; you can define and use them as needed

You do have to follow certain rules for naming a variable:

- Precede the variable name with a dollar sign
- Assign the variable a meaningful name that you can remember in the future
- Name the variable with uppercase or lowercase letters, numbers, or the underscore (\_) character
- Do not allow the first character after the (\$) to be a number
- Variable names are case sensitive
- Assign the variable an initial value with a single equals (=) sign

```

1 <?php
2   $x = 3.14;
3   echo $x;
4   echo "<br/>";
5   print($x);
6 ?>

```

- **Variable scope:** If a variable is defined at the start of a php file, it stays in memory until the end of that file. This is known as the variable's scope

If a variable is assigned a value of 5 in one php file, and that file calls another php file that has a variable of the same name, then the first variable is terminated and its value is lost

One major distinction that relates to a variable's scope involves the processing of web-based forms

Any variables that are defined within a php/HTML form and sent to the server with the form's Post method are automatically sent with the called Post action and named in php by the same name used in the HTML form

- **Variable data types:**
  - **Boolean:**
  - **Integer:**
  - **Float or double:**
  - **String :**
  - **Object:** An instance of a class
  - **Array:** Ordered set of keys and values
  - **Resource:** Reference to a thirdparty resource (a database for example)
  - **NULL:** An uninitialized variable
- **Gettype function:** Test the type of a variable by using the built-in php function `gettype()`.
- **Functions for numbers:** There are also many functions you can use with numbers. Two nice ones are `round()` and `number_format()`.
  - **round()**: rounds a decimal to either the nearest integer or to a specified number of digits. `Round($n,2)` will give 2 digits to the right of the decimal point.
  - **number\_format()**: makes a number appear in the more commonly written format (adding commas where appropriate) and you can specify digits to the right of the decimal point.
- **Super Global Variables:** Super Global Variables - pre defined in php, these are always present and their values available to all your scripts
  - **`$_GET`:** contains any variables provided to a script through the GET method
  - **`$_POST`:** contains any variables provided to a script through the POST method
  - **`$_COOKIE`:** contains any variables provided to a script through a cookie
  - **`$_FILES`:** contains any variables provided to a script through file uploads
  - **`$_SERVER`:** contains information such as
  - **`$_ENV`:** contains any variables provided to a script as part of the server environment
  - **`$_REQUEST`:** contains any variables provided to a script via any user input mechanism
  - **`$_SESSION`:** contains any variables that are currently registered to a session
- **php operators:**
  - **Arithmetic Operators** (used to perform mathematical calculations)
    - \* **+** (Addition): Adds two values, e.g., `5 + 3` results in `8`.
    - \* **-** (Subtraction): Subtracts one value from another, e.g., `5 - 3` results in `2`.
    - \* **\*** (Multiplication): Multiplies two values, e.g., `5 * 3` results in `15`.
    - \* **/** (Division): Divides one value by another, e.g., `15 / 3` results in `5`.
    - \* **%** (Modulus): Returns the remainder of division, e.g., `5 % 2` results in `1`.
    - \* **\*\*** (Exponentiation): Raises a number to the power of another, e.g., `2 ** 3` results in `8`.

- **Assignment Operators** (used to assign values to variables)
  - \* `=` (Basic assignment): Assigns a value to a variable, e.g., `$x = 5`.
  - \* `+=` (Addition assignment): Adds and assigns a value, e.g., `$x += 5` is equivalent to `$x = $x + 5`.
  - \* `-=` (Subtraction assignment): Subtracts and assigns a value, e.g., `$x -= 5` is equivalent to `$x = $x - 5`.
  - \* `*=` (Multiplication assignment): Multiplies and assigns a value, e.g., `$x *= 5` is equivalent to `$x = $x * 5`.
  - \* `/=` (Division assignment): Divides and assigns a value, e.g., `$x /= 5` is equivalent to `$x = $x / 5`.
  - \* `%=` (Modulus assignment): Takes modulus and assigns a value, e.g., `$x %= 5` is equivalent to `$x = $x % 5`.
- **Comparison Operators** (used to compare two values)
  - \* `==` (Equal): Checks if values are equal, e.g., `5 == 5` results in `true`.
  - \* `===` (Identical): Checks if values and types are identical, e.g., `5 === "5"` results in `false`.
  - \* `!=` (Not equal): Checks if values are not equal, e.g., `5 != 3` results in `true`.
  - \* `<>` (Not equal): Alternative to `!=`.
  - \* `!==` (Not identical): Checks if values and types are not identical, e.g., `5 !== "5"` results in `true`.
  - \* `>` (Greater than): Checks if one value is greater than another, e.g., `5 > 3` results in `true`.
  - \* `<` (Less than): Checks if one value is less than another, e.g., `3 < 5` results in `true`.
  - \* `>=` (Greater than or equal to): Checks if one value is greater than or equal to another, e.g., `5 >= 5` results in `true`.
  - \* `<=` (Less than or equal to): Checks if one value is less than or equal to another, e.g., `3 <= 5` results in `true`.
  - \* `<=>` (Spaceship operator): Returns `-1` if left is less, `0` if equal, `1` if greater, e.g., `5 <=> 3` results in `1`.
- **Logical Operators** (used to combine boolean expressions)
  - \* `&&` (And): Returns `true` if both operands are true, e.g., `true && false` results in `false`.
  - \* `||` (Or): Returns `true` if either operand is true, e.g., `true || false` results in `true`.
  - \* `!` (Not): Inverts the boolean value, e.g., `!true` results in `false`.
  - \* `xor` (Exclusive or): Returns `true` if only one operand is true, e.g., `true xor false` results in `true`.
- **Increment/Decrement Operators** (used to increase or decrease a variable's value by 1)
  - \* `++` (Increment): Increases the variable's value by 1, e.g., `$x++`.
  - \* `--` (Decrement): Decreases the variable's value by 1, e.g., `$x--`.
- **String Operators** (used to work with strings)
  - \* `.` (Concatenation): Joins two strings, e.g., `"Hello" . "world"` results in `"Hello world"`.
  - \* `.=` (Concatenation assignment): Appends a string to a variable, e.g., `$x .= "world"`.
- **Array Operators** (used to work with arrays)

- \* + (Union): Combines two arrays, e.g., `$a + $b` merges arrays `$a` and `$b`.
- \* == (Equality): Checks if arrays have the same key-value pairs, e.g., `$a == $b`.
- \* === (Identity): Checks if arrays have identical key-value pairs in the same order and of the same types.
- \* != (Inequality): Checks if arrays do not have the same key-value pairs.
- \* !== (Non-identity): Checks if arrays are not identical in either key-value pairs, order, or types.
- **Type Operators** (used to check or specify types)
  - \* instanceof: Checks if an object is an instance of a specific class, e.g., `$object instanceof ClassName`.
- **Error Control Operator** (used to suppress errors)
  - \* @ (Error suppression): Suppresses error messages for an expression, e.g., `@file_get_contents("nonexistentfile.txt")`.
- **Comments:** Like most computer languages, php allows you to add explanations to the code in the form of comments. These comments are ignored by the php parser
 

Comments should be added whenever necessary to explain code that is hard to follow

To insert a comment in a single line of php code, you preface the comment with either a pound symbol (#) or two forward slashes (//)

  - **Define():** use php's builtin define() function

```
1 define("YOUR_CONSTANT_NAME", value)
```

You can set your constant to a number, a string, or a boolean.

By convention, use all caps for name of a constant.

You don't use a \$ when accessing a constant.

  - **If statements:** If statements in php are the same as c++

```
1 if (expression) {
2     ...
3 } else if (expression) {
4     ...
5 } else {
6     ...
7 }
```

- **Switch-Case:** Switch statements are also the same as c++

```

1  switch (expression) {
2      case 1:
3          ...
4          break;
5      case 2:
6          ...
7          break;
8      default: // If no break was encountered
9          ...
10         break;
11 }

```

- **For Loop:** Same as c++

```

1  for (init expression; test expression; modification
2      → expression) {
3          //code to execute
4      }

```

**Note:** zero, an undefined variable or an empty string will all evaluate to false, all others will evaluate to true.

- **False evaluations:** In PHP, several values evaluate to false when used in a boolean context.
  - **Boolean:** false itself.
  - **Integer:** 0 (zero).
  - **Float:** 0.0 (zero as a floating-point number).
  - **String:** An empty string "" and the string "0".
  - **Array:** An empty array [].
  - **NULL:** The NULL type.
  - **Objects without methods or properties:** Objects that do not have any methods or properties defined in them may also evaluate to false when checked with empty().

- **While and do while:** Same as c++

```

1  while (expression) {
2      ...
3  }
4
5  do {
6      ...
7  } while(expression);

```

- **Arrays:** There are two ways to define an array in php.

```

1 $colors = array("red", "green", "blue");
2
3 $colors[0] = "red";
4 $colors[1] = "green";
5 $colors[2] = "blue";

```

These are both numerically indexed arrays.

- **Associative arrays:** You can also have associative arrays which have named keys.

```

1 $character = array(
2     "name" => "Monk",
3     "occupation" => "detective"
4 );

```

You access an element of an associative array by using the key name rather than a number.

- **Array functions:** There are approximately 60 array functions built into php. Some important ones are

- count() and sizeof() return the number of elements in the array.
- foreach() steps through an array
- each() and list() usually appear together in the context of stepping through an array and returning keys and values
- reset() rewinds the pointer to the beginning of the array
- array\_push() adds elements at the end of an existing array
- array\_pop() removes and returns the last element in an existing array
- array\_merge() combines two or more existing arrays
- shuffle() randomizes the elements of a given

- **Including Files:** When developing more than a single home page for the Internet, you probably want the pages to have a common look and feel

To make this possible, php has provided a method called include files

These files let you incorporate common artwork, contact information, and menu and link options into your Web pages with a minimum of code

```

1 // vars.php
2 <?php
3     $color = 'green';
4     $fruit = 'apple';
5 ?>
6
7 // test.php
8 <?php
9     echo "A $color $fruit"; // Output - A
10    include 'vars.php';
11    echo "A $color $fruit"; // Output - A green apple
12 ?>

```

- **Try except:** Same as c++

```

0  try {
1      ...
2  } catch (exceptionname e) {
3      ...
4 }
```

- **Creating objects with the new keyword:** In PHP, the new keyword is used to create an instance of a class, which is known as an object. When you use new, PHP allocates memory for the object and calls its constructor method (if defined) to initialize it

```

1 $object = new ClassName();
```

- **Scope resolution operator (::):** In PHP, the :: operator is called the Scope Resolution Operator, also known as the Paamayim Nekudotayim (Hebrew for "double colon"). It is used to access static, constant, or overridden properties and methods of a class, without needing to instantiate the class.
- **arrow operator (->):** In PHP, the -> operator is used to access properties and methods of an object instance. It allows you to call non-static methods and access non-static properties on an instantiated object.
- **print\_r:** The print\_r function in PHP is used for outputting human-readable information about a variable. It's particularly useful for displaying the contents of arrays and objects. Unlike echo or print, which are mainly used to display strings, print\_r provides a structured format for complex data types, making it easier to visualize the nested structure of an array or object.

```

0 print_r(mixed $expression, bool $return = false): mixed
```

```

0 $array = array("name" => "John", "age" => 30, "city" => "New
    York");
1 print_r($array);
```

- **foreach():**

```

0 $a = array(1 => 1 ,2,3);
1 foreach($a as $item) {
2     echo $item;
3
4 }
5
6 foreach($a as $key => $value) {
7     echo "key: $key      value: $value";
8     echo "<br/>";
9 }
```

- **Functions:** Functions in php are made with the *function* keyword

```
0 function functionName($param1, $param2) {  
1     // Code to execute  
2     return $result; // Optional  
3 }
```

- **Anonymous functions:** Functions with no name, often used as variables or passed as arguments to other functions.

```
0 $sayHello = function($name) {  
1     return "Hello, $name!";  
2 };  
3  
4 echo $sayHello("Bob"); // Outputs: Hello, Bob!
```

- **Arrow Functions:** Shorter syntax for one-liner anonymous functions.

```
1 $multiply = fn($a, $b) => $a * $b;  
2 echo $multiply(2, 3); // Outputs: 6
```

- **Optional Parameters:** Parameters with default values, which are used if no argument is provided.

```
0 function greet($name = "Guest") {  
1     return "Hello, $name!";  
2 }  
3 echo greet(); // Outputs: Hello, Guest!
```

- **Passing by reference:** By default, arguments are passed by value, meaning changes within the function don't affect the original variable. You can pass by reference using &.

```
0 function addOne(&$number) {  
1     $number += 1;  
2 }  
3 $num = 5;  
4 addOne($num);  
5 echo $num; // Outputs: 6
```

### 3.9 SQL via PHP: PDO

- **Why SQL via PHP?**: Can provide an interface for the user that does not require them to worry about database design specifics.

Although there are other ways to provide this interface, the web-based interface is very common, and PHP is a common and relatively easy way of making it work.

- **Application Programming Interface (API)**: In order for our PHP application to interface with our DBMS, we will need to use an appropriate API (Application Programming Interface) An API is the set of function calls and other resources that are provided to allow you to interface with a given application via your program code.
- **Which API?**: Even for a given DBMS, there can be many API's present. PHP has been around for a while, so many things have evolved and died out. Many of the API's still work. Some work but are considered deprecated, and others are no longer supported at all. In this class, we will be working with the PDO library.
- **PDO Library**: The PHP Data Objects (PDO) library is an object oriented API to connect PHP to SQL servers. It allows you to use a common interface to interact with many different DBMS programs.

It supports most of the popular relational DBMSes, including MySQL, Postgresql, and SQLite, in a fairly transparent way, so it is more portable than using the other, DBMS specific API's. **Note:** Because PDO is object oriented, it requires at least version 5 of PHP. If you need to use a lower version, you'll need to look into the other API's available.

- **PHP Data Objects**: The PDO library is object oriented. That means that our interactions with the database will be done using objects and their members. There are three basic objects that we will be concerned with:
  - **PDO**: this object handles the connection to the database
  - **PDOStatement**: this object handles prepared statements, and is used to work with result sets
  - **PDOException**: this object is used to store information on errors that have occurred
- **Specifying a Database with a DSN**: Although, once properly initialized, PDO should function the same for any DBMS, you need to properly initialize it by telling it which type of server you are connecting to. To do this, you need to make a DSN string.

DBMS	DSN Format
MySQL	mysql:host=HOSTNAME;dbname=DBNAME
Postgresql	pgsql:host=HOSTNAME;dbname=DBNAME
SQLite 3	sqlite:PATHTODB
SQLite 2	sqlite2:PATHTODB

MariaDB will use the MySQL interface.

- **Initializing PDO**: Once you've chosen the DBMS you'll be using and you've chosen an appropriate DSN string, you can use that DSN to construct an instance of the PDO class. This object represents a connection to the database specified by the DSN

```

1  try { // if something goes wrong, an exception is thrown
2      $dsn = "mysql:host=courses;dbname=z123456";
3      $pdo = new PDO($dsn, $username, $password);
4  }
5  catch(PDOException $e) { // handle that exception
6      echo "Connection to database failed: " .
    $e->getMessage();
7  }

```

- **Using PDO to Talk to DBMS:** The PDO library provides three basic ways of running queries for a database, once connected:
  - the PDO::exec() function is used to execute an SQL query that does not return a result (INSERT, UPDATE, etc.)
  - the PDO::query() function is used to execute an SQL query that will return a result (SELECT)
  - the PDO::prepare() function should be used when constructing a query from user input.
- **Using PDO::exec():** PDO::exec() is used to run a query that does not return any results. Instead of returning a result set, it will tell you how many rows were affected by your query.

```

1  // Three examples follow.
2  $n = $pdo->exec("INSERT INTO Students (FName) VALUES
    ('Victor');");
3  $n = $pdo->exec("UPDATE Students SET LName='Husky' WHERE
    FName='Victor');");
4  $n = $pdo->exec("DELETE FROM OldJunk;");

```

- **Using PDO::query():** PDO::query() is used to run a query that does return a result. The result set is returned as a PDOStatement object.

```

1  # Defining sql as the query you'd like to run; here's one from classicmodels
2  $sql = "SELECT phone FROM Customer;";
3

```

- **Using PDO::prepare():** The third option is to use the PDO::prepare() command. This is useful for situations where the same query is run multiple times in the same script, and can also help you to avoid some SQL Injection issues. Once prepare() succeeds, you run the query with the execute() method of the PDOStatement returned by prepare()

You can use a colon before a value name in your query to denote where the execute statement will insert the value of the given name:

```

1  <?php
2      # Notice that we use :color below in our SQL template
3      $sql = 'SELECT name, color, calories
4          FROM fruit
5          WHERE calories < :calories AND color = :color';
6
7      $prepared = $pdo->prepare($sql, array(PDO::ATTR_CURSOR
8          => PDO::CURSOR_FWDONLY));
9      # The value associated with the :color key will be used
10     # when executing
11
12     $success = $prepared->execute(array(':calories' => 150,
13         ':color' => 'red'));
14     # if(success == true) then prepared will be ready to be
15     # used as a result set
16     # with fetch() or fetchAll() -- just like the object
17     # returned by query()
18
19 ?>

```

It is also possible to use a ? in your query as a positional parameter.

```

1  <?php
2      # Execute a prepared statement by passing an array of
3      # values
4      $prepared = $pdo->prepare('SELECT name, color, calories
5          FROM fruit
6          WHERE calories < ? AND color =
7              ?');
8
9      # Here we execute the query twice with different
10     # parameters.
11     # The ?'s will be replaced with the values in the array
12     # specified,
13     # in the order they are specified.
14     $prepared->execute(array(150, 'red'));
15     $red = $prepared->fetchAll();
16     $prepared->execute(array(175, 'yellow'));
17     $yellow = $prepared->fetchAll();
18
19 ?>

```

- **Named placeholders:** In PDO (PHP Data Objects), named placeholders are a way to represent dynamic values in SQL statements using identifiable names instead of anonymous ? placeholders. Named placeholders make SQL statements easier to read and maintain, especially when there are multiple variables or parameters.

A named placeholder is written as :name in an SQL query, where name is an identifier you choose. When you prepare the SQL statement, you can bind actual values to these placeholders by their names, allowing you to execute the query with specific data values.

- **anonymous placeholders:** In PDO (PHP Data Objects), anonymous placeholders (also called positional or unnamed placeholders) are represented by question marks

? in SQL statements. These placeholders are used to represent values that will be dynamically bound to the SQL query at execution time.

When using anonymous placeholders, each placeholder corresponds to a specific value based on its position in the SQL query. When you prepare and execute the query, you provide an array of values that will replace each ? in the order they appear.

- **Execute:** In PHP's PDO (PHP Data Objects) extension, the `execute()` method is used to run a prepared SQL statement with specific values for placeholders. It's part of the `PDOStatement` class and is essential for safely executing queries that include dynamic data, protecting against SQL injection attacks.

After preparing a statement with `PDO::prepare()`, you call `execute()` to supply any values for placeholders (either named or anonymous) and run the query.

- **Dealing with result sets:** Once you have a result set (stored in a `PDOStatement` from `query()`) you can use its `PDOStatement::fetch()` or `PDOStatement::fetchAll()` methods to get the data returned.

If you would like to work on one row at a time, as if using the `mysql_fetch_array()` function from the original MySQL API, use `fetch()`.

To grab all of the rows at the same time, use `fetchAll()`.

```
1 <?php
2     # FETCH_BOTH means that you will get both position
3     # indices and the column names
4     # as keys in the array returned
5     $row = $result->fetch(PDO::FETCH_BOTH);
6
7     # this returns all of the rows at once in a
8     # two-dimensional array
9     $allrows = $result->fetchAll();
10    ?>
```

- **Handling Errors:** As of the time of this writing, there are three modes PDO can use to handle errors.
  - **PDO::ERRMODE\_SILENT:** the default mode. PDO will set the error code for you to inspect using the `errorCode` and `errorInfo` methods on your PDO and `PDOStatement` objects
  - **PDO::ERRMODE\_WARNING:** in addition to setting the error code, emits a traditional `E_WARNING` message. Use for debugging/testing when you just want to see what problems occurred without interrupting the flow of the application.
  - **PDO::ERRMODE\_EXCEPTION:** in addition to setting the error code, also throw `PDOException` when an error occurs.

You can set which one you'd like PDO to use with the `setAttribute` method of the PDO object you're using, as below:

```

1  $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_SILENT);
2  $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
3  $pdo->setAttribute(PDO::ATTR_ERRMODE,
4    PDO::ERRMODE_EXCEPTION);

```

- Members of the PDO object

```

0  PDO {
1      // constructor has a bunch of optional parameters, check
2      // reference if needed
3      public __construct ( string $dsn [, string $username [,,
4          string $password [,,
5          array $options ]]] )
6      public bool beginTransaction ( void )
7      public bool commit ( void )
8      public mixed errorCode ( void )
9      public array errorInfo ( void )
10     public int exec ( string $statement )
11     public mixed getAttribute ( int $attribute )
12     public static array getAvailableDrivers ( void )
13     public bool inTransaction ( void )
14     public string lastInsertId ([ string $name = NULL ] )
15     public PDOStatement prepare ( string $statement [, array
16         $driver_options ] )
17     public PDOStatement query ( string $statement )
18     public string quote ( string $string [, int
        $parameter_type = PDO::PARAM_STR ] )
19     public bool rollBack ( void )
20     public bool setAttribute ( int $attribute , mixed $value
        )
21 }

```

- Members of the PDOStatement object:

```

0  PDOStatement implements Traversable {
1      /* Properties */
2      readonly string $queryString;
3      /* Methods */
4      public bool bindColumn ( mixed $column , mixed &$param
5          ↵ [, int $type [, int $ maxlen [, mixed $driverdata ]]]
6          ↵ )
7      public bool bindParam ( mixed $parameter , mixed
8          ↵ &$variable [, int $data_type = PDO::PARAM_STR [, int $length [, mixed
9              ↵ $driver_options ]]] )
10     public bool bindValue ( mixed $parameter , mixed $value
11         ↵ [, int $data_type = PDO::PARAM_STR ] )
12     public bool closeCursor ( void )
13     public int columnCount ( void )
14     public void debugDumpParams ( void )
15     public string errorCode ( void )
16     public array errorInfo ( void )
17     public bool execute ([ array $input_parameters ] )
18     public mixed fetch ([ int $fetch_style [, int
19         ↵ $cursor_orientation = PDO::FETCH_ORI_NEXT [, int $cursor_offset = 0 ]]] )
20     public array fetchAll ([ int $fetch_style [, mixed
21         ↵ $fetch_argument [, array $ctor_args = array() ]]] )
22     public mixed fetchColumn ([ int $column_number = 0 ] )
23     public mixed fetchObject ([ string $class_name =
24         ↵ "stdClass" [, array $ctor_args ]] )
25     public mixed getAttribute ( int $attribute )
26     public array getColumnMeta ( int $column )
27     public bool nextRowset ( void )
28     public int rowCount ( void )
29     public bool setAttribute ( int $attribute , mixed $value
30         ↵ )
31     public bool setFetchMode ( int $mode )
32 }

```

- Members of the PDOException object

```
0 PDOException extends RuntimeException {
1     /* Properties */
2     public array $errorInfo ;
3     protected string $code ;
4     /* Inherited properties */
5     protected string $message ;
6     protected int $code ;
7     protected string $file ;
8     protected int $line ;
9     /* Inherited methods */
10    final public string Exception::getMessage ( void )
11    final public Throwable Exception::getPrevious ( void )
12    final public mixed Exception::getCode ( void )
13    final public string Exception::getFile ( void )
14    final public int Exception::getLine ( void )
15    final public array Exception::getTrace ( void )
16    final public string Exception::getTraceAsString ( void )
17    public string Exception::__toString ( void )
18    final private void Exception::__clone ( void )
19 }
```

### 3.10 Transactions and concurrency control

- **Transaction:** A logical unit of work, a unit of program execution that access and possibly updates various data items

is initiated by user program written in a high-level data-manipulation language like SQL, COBOL, C, C++, or Java

Transactions are delimited by statements such as *begin transaction end transaction* or *COMMIT* or *ROLLBACK*

- **Sample Transaction:**

```
0 BEGIN TRANSACTION;
1     UPDATE ACC 123 { BALANCE := BALANCE - 100 };
2     IF (ANY ERROR) THEN
3         GO TO UNDO;
4     ENDIF;
5     UPDATE ACC 456 { BALANCE := BALANCE + 100 };
6     UPDATE ACC 456 { BALANCE := BALANCE + 100 };
7     IF (ANY ERROR) THEN
8         GO TO UNDO;
9     ENDIF;
10    COMMIT;
11    GO TO FINISH;
12 UNDO:
13     ROLLBACK;
14 FINISH:
15     RETURN;
```

- **ACID properties of transactions:**

- **Atomicity:** either all operations of the transaction are implemented properly or none are.
- **Consistency:** execution of a transaction in isolation preserves the consistency of the database (with no other transaction executing concurrently).
- **Isolation:** each transaction is unaware of other transactions executing concurrently in the system.
- **Durability:** after a transaction completes successfully, the changes it has made to the database persist, even if the system fails.

- **Transaction States:** In the absence of any failures, all transactions complete successfully.

However, there is not always an absence of any failures.

Thus we have different "states" in which a transaction may reside.



- **Active:** the initial state and one in which the transaction stays while it is executing.
- **Partially committed:** a state after the final statement has been executed.
- **Failed:** the state after the discovery that normal execution can no longer proceed.
- **Aborted:** the state after the transaction has been rolled back and the database restored to its condition prior to the start of the transaction.
- **Committed:** the state after successful execution.
- **Committed Transaction:** A transaction is considered committed when it has performed updates that transforms the database into a new consistent state.

Once a transaction is committed, its effects cannot be undone by a system failure.

Only way to undo a committed transaction is to execute a compensating transaction.

However it is not always possible to create a compensating transaction.

- **Failed & Aborted Transaction:** When a transaction cannot be completed (due to some kind of system failure), the transaction must be "rolled back".

It also enters the aborted state where the system has two options:

- Restart the transaction (but only if aborted due to some hardware or software error).
- Kill the transaction which is usually done due to some internal logical error.

- **Concurrent Executions of Transactions:** Concurrent executions are good:

- Improved throughput and resource utilization
- Reduced waiting time

In transaction processing multiple transactions are allowed to run concurrently.

Updating within concurrent transactions causes several complications with consistency of the data.

- **Execution sequences:** Represent the chronological order in which instructions are executed in the system
- **Serial schedules:** consists of a sequence of instructions from various transactions where the instructions belonging to one single transaction appear together in that schedule
- **Concurrency control:** Concurrency control: the task of ensuring that any schedule that gets executed will leave the database in a consistent state

The concurrency-control component of the DBMS carries out setting up the schedules to ensure consistency during the execution of multiple transactions

- **Serializability:** a schedule that is equivalent to a serial schedule.

In discussing serializability, only two operations are important

- read(Q)
- write(Q)

We also assume that the transaction may perform an arbitrary sequence of operations on the copy of Q between the read and write operations

- **Concurrency control introduction:** Fundamental Property of a transaction is isolation

To ensure that isolation property is preserved when several transactions are running concurrently, the system controls the interaction among the concurrent transactions

These schemes are called concurrency control

- **Lock-Based Protocols:** One way to ensure serializability

- Require data items be accessed in a mutually exclusive manner
- That is when one transaction is accessing the data item, no other transaction can modify that data item
- Common method to implement is that transaction must hold a lock on an item

- **Modes in which data item may be locked:**

- **Shared:** If a transaction T1 has obtained a shared-mode lock on item Q, then T1 can read, but cannot write, Q
- **Exclusive:** If a transaction T1 has obtained an exclusivemode lock on item Q, then T1 can both read or write Q

Every transaction is required to request a lock in an appropriate mode on each data item

Request is made to the concurrencycontrol manager

Concurrency-control manager must grant the lock to the transaction before it can proceed.

- **Compatibility Function:** Given set of lock modes can define compatibility function

- A & B represent arbitrary lock modes
- Transaction T1 request a lock of mode A on item Q
- Transaction T2 currently holds a lock of mode B
- If T1 can be granted a lock on Q immediately in spite of the presence of the mode B lock, then mode A is compatible with mode B

Can represent with a Lock-compatibility matrix

	shared	exclusive
shared	true	false
exclusive	false	false

When a transaction requests a lock that is incompatible, it enters a wait state until all incompatible locks have been released

Transactions cannot execute until concurrency-control manager grants the requested locks

- **Deadlock:** Locking can lead to an undesirable situation where no transaction can proceed with normal execution. This situation is called deadlock

When this occurs

- The system must roll back one of the transactions
- Unlocking transactions until execution can be continued

If locking is not used, or if a data item is unlocked as soon as possible after reading or writing, inconsistent states may occur

On the other hand, if a data item is not unlocked before requesting a lock on some other data item deadlocks may occur

In general, deadlocks are a necessary evil associated with locking which is necessary to avoid inconsistent states

Deadlocks are preferable to inconsistent states

- since they can be handled via rolling back transactions
- inconsistent states cannot be handled by database

- **Locking Protocol:** a set of rules that each transaction must follow

Indicates when a transaction may lock or unlock each data item

A schedule is legal under a given locking protocol if it follows the rules

A locking protocol ensures conflict serializability if and only if all legal schedules are conflict serializable

- **Granting of Locks:** Grant can take place if

- a transaction requests a lock on a data item in some mode
- and no other transaction has a lock on the same data item in a conflicting mode

Care must be taken to avoid certain situations

Suppose transaction T2 has a shared-mode lock on a data item

Transaction T1 requests an exclusive mode lock on the data item

Clearly, T1 has to wait for T2 to release the shared-mode lock

Meanwhile, transaction T3 may request a shared-mode lock on the same data item

The lock request is compatible with the lock granted to T2 so T3 may be granted the shared-mode lock

At this point, T2 releases the lock but T1 has to still wait for T3 to finish

But again, there are other transactions  $T_i$ , that requests a shared-mode lock

In fact it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item and T1 NEVER gets the exclusive-mode lock

T1 is then said to be starved

- **Avoiding starvation of transactions:** T1 requests a lock on a data item Q in some mode M. the lock is granted provided that
  - there is no other transaction holding a lock on Q in a mode that conflicts with M
  - there is no other transaction that is waiting for a lock on Q and that made its lock request before T1
- **Two-Phase Locking Protocol:** A protocol that ensures serializability is the two-phase locking protocol

Each transaction issues lock and unlock requests in two phases

1. **Growing phase:** a transaction may obtain locks but may not release any lock
2. **Shrinking phase:** a transaction may release locks but may not obtain any new locks

Initially a transaction is in the growing phase

Once the transaction releases a lock, it enters shrinking phase and cannot request more locks

The Two-phase locking protocol

1. Ensures conflict serializability
2. Does not ensure freedom from deadlock

- **Variations of two-phase locking protocol:**
  1. **Strict two-phase locking protocol:** requires not only that locking be two phase, but that all exclusive-mode locks taken by a transaction be held until that transaction commits
  2. **Rigorous two-phase locking protocol:** requires that all locks be held until the transaction commits
- **Refinement of basic two-phase locking protocol:** lock conversions are allowed
  - a mechanism is allowed for upgrading a shared lock to an exclusive lock
  - a mechanism is allowed for downgrading an exclusive lock to a shared lock

Strict two-phase and rigorous twophase locking (with lock conversions) are extensively used in DBMSs

A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction on the basis of read and write requests

When a transaction T1 issues a read(Q), the system issues a lock-S(Q) instruction followed by the read(Q) instruction

When T1 issues a write(Q) operation, the system checks to see whether T1 already holds a shared lock on Q.

- If yes, system issues an upgrade(Q) followed by a write(Q)
- If no, system issues a lock-X(Q) followed by a write(Q)

All locks obtained by a transaction are unlocked after that transaction commits or aborts

- **Other Locking Protocols:**

- **Graph-based protocols**
- **Timestamp-based protocols**
- **Validation-based protocols:** majority of transactions are read-only
- **Multiversion schemes:** each write(Q) creates a new version of Q

- **Deadlock Handling:** Deadlock state... there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set

Two principal methods for dealing with deadlocks

1. deadlock prevention
2. deadlock detection and deadlock recovery

- **Deadlock prevention:** two approaches

1. **one approach:** ensure that no cyclic waits can occur by
  - ordering the requests for locks
  - or requiring all locks be acquired together
2. **second approach:** impose an ordering of all data items and require that a transaction lock data items only in a sequence consistent with the ordering

- **Deadlock detection and recovery:** an algorithm that examines the state of the system is invoked periodically to see if a deadlock has occurred, if one has, then the system must recover

To recover from a deadlock the system must

- maintain information about the current allocation of data items to transactions as well as any outstanding data item requests
- provide an algorithm that uses this information to determine whether the system has entered a deadlock state
- recover from the deadlock when the detection algorithm determines that a deadlock exists.

### 3.11 Mariadb in C++

- **Motivation:** We have already looked at how to interact with a MariaDB Database from PHP using the PHP Data Objects (PDO) API. Now we will look at the API you can use to do the same thing in C/C++ programs.

The MariaDB API is implemented in C, so it is provided as a set of functions to call, as opposed to a set of classes, as may be found in more object oriented programming

- **Header file:** To use any of these functions, you need to include the appropriate header file.

```
o #include <mysql/mysql.h>
```

This file provides declarations for the functions and new types that make up the MariaDB API. Later, during the linking stage, you will specify the mariadb library, where the implementations of the functions are found.

- **Some new variable types:** The MariaDB library defines some new types. Some of them are data structures that store more complicated data, which is dealt with using the library functions, others are just a different way of referring to types you already know
  - **MYSQL:** a data structure containing data about a connection to a MariaDB server
  - **MYSQL\_RES:** a data structure that contains data about a result set
  - **MYSQL\_ROW** a data structure that contains data about a single row from a result set
  - **my\_ulonglong:** used to store large unsigned integers
- **Handling errors:** Various functions in the MariaDB API have specific return codes for when an error has occurred. In other situations, it may not be clear.

When an error occurs, there are two functions you can use to find out what happened:

```
o unsigned int mysql_errno(MYSQL *mysql);
1 const char *mysql_error(MYSQL *mysql);
```

- mysql\_errno - returns a numeric error code that identifies the last error that occurred
- mysql\_error - returns a human-readable error message describing the last error that occurred.

The mysql parameter is a pointer to the connection object you would like to check the error for. We will discuss that later.

- **Library Initialization:**

```
o int mysql_library_init(int argc, char **argv, char **groups)
```

This function initializes the MySQL library. It is generally only needed if you want your application to be threadsafe, but you should call it in your programs for this class anyway.

- **argc**: argument count, use 0 unless you have a reason not to
- **argv**: argument vector, use NULL unless you have a reason not to
- **groups**: NULL-terminated array of strings listing which options to read, use NULL unless you have a reason not to

- **Library Deinitialization:**

```
o void mysql_library_end(void)
```

This function cleans up any memory allocated during the use of the MySQL library. You should make sure to call it after you are completely finished using the library to communicate with the DBMS.

- **Initializing a Connection Object:**

```
o MYSQL *mysql_init(MYSQL *mysql)
```

This function initializes (allocating memory as well, if needed) a MYSQL object, which is suitable for use with mysql\_real\_connect(). If mysql is NULL, then function will dynamically allocate memory for the object. If it is a pointer to an already allocated MYSQL object, it will set it up in that place. Either way, it will return a pointer to where the initialized object is. If there is an error, NULL will be returned.

- **mysql**: a pointer to a MYSQL object to initialize, or NULL

- **Establishing a DB Connection:**

```
o MYSQL *mysql_real_connect(MYSQL *mysql, const char *host,
 1   const char *user, const char *passwd,
 2   const char *db, unsigned int port,
 3   const char *unix_socket, unsigned long client_flag)
```

Establishes a connection to the specified MySQL database.

- **mysql**: pointer to an existing MYSQL object, can be used to set options
- **host**: the hostname of the server to connect to item user - the username to authenticate with
- **passwd**: the password for the specified user
- **db**: the name of the database to use on that MySQL server
- **port**: set to zero unless you want to use a non-default port
- **unix\_socket**: set this to NULL
- **client\_flag**: set this to zero (can be used for special flags)

Returns a pointer to a MYSQL connection object on success, NULL on failure.

- **Closing a DB Connection:**

```
o void mysql_close(MYSQL *mysql)
```

This function closes a previously opened connection. It also deallocates the connection handler pointed to by mysql if the handler was allocated automatically in either mysql\_init() or mysql\_connect().

- **mysql:** a pointer to the MYSQL object for the connection to be closed

- **Running SQL Queries:** After you have initialized the API and connected to the server, you can begin to run SQL queries on it. There are a couple of functions that are designed to do this.

- **mysql\_query():** allows you to send a query as a null-terminated string

- **mysql\_real\_query():** allows you to send a query as a string of a specified length

- **Queries using null-terminated string:**

```
o int mysql_query(MYSQL *mysql, const char *stmt_str)
```

This function executes the SQL statement stored in the null-terminated string pointed to by stmt\_str. Normally the string must consist of a single SQL statement without a terminating semicolon. If you explicitly enable multiple-statement execution, it can contain several statements separated by semicolons. It is recommended not to do this, normally, because of the potential for SQL injection.

- **stmt\_str:** a pointer to the beginning of a null-terminated string containing the SQL statement(s) to run

Returns zero on success. Non-zero if an error has occurred.

- **Queries using string with length:**

```
o int mysql_real_query(MYSQL *mysql,
1           const char *stmt_str,
2           unsigned long length)
```

This function executes the SQL statement pointed to by stmt\_lstr, which is interpreted to be length bytes long. The use of a length as opposed to a terminating null character is what distinguishes this from the mysql\_lquery() function from before. This allows binary data, which may contain null characters as valid data, to be sent.

- **mysql:** pointer to the MySQL connection to run the query through

- **stmt\_lstr:** pointer to the beginning of the string containing the SQL statement(s)

- **length:** the length, in bytes, of the string, stmt\_lstr

It returns zero on success, non-zero if an error has occurred.

- **Get information on a query's results:** After a query is run, there may be a result set, which you would expect from a query with, for example, a SELECT statement. There are a couple of ways to access the results of such a query.

Other queries might not generate a result set, such as INSERT, UPDATE, or DELETE. There are some things you may want to know about them even though no data is returned

For queries that don't generate a result set, you may want to know some or all of the following

- If you want to know how many rows were returned in a result set, you can use mysql\_ lnum\_lrows().
- If you want to know how many attributes (columns) exist in a result set, you can use mysql\_ lfield\_lcount().
- If you want to know what value was chosen for an AUTO\_INCREMENT field, you can use mysql\_ linsert\_lid().

- **How many rows were affected?:**

```
o my_ulonglong mysql_affected_rows(MYSQL *mysql)
```

This function can be called immediately after running a query on the server.

- **mysql:** a pointer to the server connection object you ran the query through

Greater than zero indicates number of rows affected or retrieved. Zero indicates no rows. Errors are indicated with a -1 return code.

- **How many rows were returned?:**

```
o my_ulonglong mysql_num_rows(MYSQL_RES *result)
```

This function will return the number of rows in the specified result set. If you want to use this before doing things with your data, you will need to use mysql\_ lstore\_lresult() instead of mysql\_ luse\_lresult() to retrieve the result set.

- **result:** a pointer to the result set object to inquire about

- **How many columns in the result?:**

```
o unsigned int mysql_field_count(MYSQL *mysql)
```

This function will return the number of columns were in the result set of the most recent query on the specified connection.

- **mysql:** a pointer to the connection object you just queried

- **AUTO\_INCREMENT → What is the key of inserted row?:**

```
o my_ulonglong mysql_insert_id(MYSQL *mysql)
```

This function returns the value generated for an AUTO\_INCREMENT column by the previous INSERT or UPDATE statement.

- **mysql**: a pointer to the connection object we ran the query through

Will return zero unless a value has been stored in an AUTO\_INCREMENT field. If multiple rows were affected, only the first one will be returned.

- **Getting info from result sets**: There are two basic functions that you can use to retrieve the data from a result set.

- The mysql\_lstore\_lresult() will function similarly to the way PDOStatement::fetchAll() function did, in that it will retrieve all the results at once.
- The mysql\_luse\_lresult() works more similarly to the PDOStatement::fetch() function, grabbing the results one row at a time.
- Neither of these two will directly give you the values in the row; that can be done with the mysql\_lfetch\_lrow() function.
- You should make a call to mysql\_lfree\_lresult() after finishing with the result sets, to free up memory used to store the results.

- **Download all rows of the result set immediately**

```
o MYSQL_RES *mysql_store_result(MYSQL *mysql)
```

This function is used to retrieve the result set generated by a query. It will download the whole result set from the server immediately and store it in your program's memory. The values can then be obtained row-by-row with mysql\_lfetch\_lrow() or you can use mysql\_lrow\_lseek() to jump to specific rows.

- **mysql**: pointer to the connection object we used to run the query

Returns a pointer to a result set structure if successful, NULL on error.

- **Set up to download one row at a time**:

```
o MYSQL_RES *mysql_use_result(MYSQL *mysql)
```

This function sets up a result set that will fetch its rows one at a time from the server. Individual rows are obtained with calls to mysql\_lfetch\_lrow(). This is more memory efficient than using mysql\_lstore\_lresult() would be.

- **mysql**: a pointer to the connection object that ran the query

Returns a pointer to the result set structure on success. NULL is returned on failure.

- **Get the row data**:

```
o  MYSQL_ROW mysql_fetch_row(MYSQL_RES *result)
```

This will fetch the next row in the result set as a MYSQL\_IROW structure, if there is one to fetch. It will start at the beginning and advance by one row each time it is called. If you call this after you've fetched the last row, it will return NULL. It will also return NULL if an error has occurred.

- **result:** a pointer to the result set object to fetch rows from

If you have any binary data in any of your fields, you will need to use mysql\_lfetch\_llengths() to get the lengths, as they may contain the null character as part of their valid data. Otherwise, you can treat a MYSQL\_IROW as an array of null-terminated strings. The fields can be addressed as the elements of the array, in the order they appear in the result set, starting from element 0.

- **Get byte-lengths for the fields in a row:**

```
o  unsigned long *mysql_fetch_lengths(MYSQL_RES *result)
```

This function returns an array of integers that contains the lengths of each of the fields in a row returned by a previous call to mysql\_lfetch\_lrow(). The integer in a given element of the array returned is the length of the corresponding element in the MYSQL\_IROW

- **result:** a pointer to the result set you just got a row from

Returns NULL on error. The most common error will be that you haven't fetched a row, or the last fetch failed.

- **Compiling without Linking:** To compile without linking, you can specify the -c flag to gcc or g++. This will take your source code, program.cc or program.c in this example, and make an object file from the code within it.

```
o  g++ -c -I/usr/include/mariadb program.cc
  1  gcc -c -I/usr/include/mariadb program.c
```

Either of these statements will yield a new file called program.o, which is the object file compiled from the original source.

- **Compilation Errors:** The compilation stage is concerned with declarations. If you have an error that says something is undeclared, that failure is happening in the compilation stage.

If you have an error during compilation, it is usually one of the following types

- You forgot to include a header file that contains necessary declarations
- You made a syntax error in your code, which the compiler’s error message should help you correct
- You made a typo somewhere (misspelled identifiers, etc.)

- **Linking Alone:** After your object files have been created through compilation, you can perform the linking stage with one of the following commands.

```
0 g++ -o program program.o -lmariadb  
1 gcc -o program program.o -lmariadb
```

- The `-o` flag say to name the program whatever the next word is. In this case, your `program.o` would be linked to make an executable file named `program`. If your project had multiple source code files, you’d include all of the object files’ names, separated by spaces, where `program.o` is now.
- the `-l` flag is used to specify the name of the library to link in.

- **Linking Errors:** The linking stage is concerned with finding the compiled code in object files/libraries. If you see an error about an “undefined reference”, then the failure is happening during the linking stage. If an error occurs during the linking phase, it is usually one of the following

- You failed to list one of the object files that contains the implementations of your functions
- You failed to tell the linker to include a library that is needed
- The linker path does not include the directory your library is located in.
- Your header file has a different declaration than its implementation uses.

- **Compilation and Linking Together:** It is possible to perform compilation and linking with the same command. This may be easier to type, but doing it separately allows you to skip recompiling files that haven’t changed. The following commands are an example of how to do both stages together.

```
0 g++ -o program -I/usr/include/mariadb -lmariadb program.cc  
1 gcc -o program -I/usr/include/mariadb -lmariadb program.c
```

# Software engineering

## 4.1 Introduction

- **Software engineering:** Software engineering is concerned with theories, methods, and tools for professional software, expenditure on software represents a significant fraction of GNP in all developed countries

Software engineering is concerned with cost effective software development

SWE is an engineering discipline that is concerned with all aspects of software product from the early stages of system specification through to maintaining the system after it has gone into use

- **Engineering discipline:** Using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints
- **All aspects of software product:** Not just technical process of development. Also project management and the development of tools, methods, etc to support software production.
- **Software costs:** Software costs dominate computer system costs, the costs of software on a PC are often greater than the hardware cost

Software costs more to maintain than it does to develop, maintenance costs may be several times development costs

It is usually cheaper to use SWE methods for software systems rather than just write the programs as if it was a personal programming project. Majority of costs are the costs of changing the software after its gone into use.

- **Software project failure:** New software eng techniques help us build larger, more complex systems. Although the demands are changing, larger more complex systems are required.

Software that does not use effective software eng techniques are often more expensive and less reliable than it should be.

- **Good software:** Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and useable
- **Fundamental software engineering activities:** Software specification, software development, software validation and software evolution,
  - **Software specification:** Customers and engineers define the software to be produced and the constraints on its operation.
  - **Software development:** The software is designed and programmed.
  - **Software validation:** The software is checked to ensure it meets customer requirements.
  - **Software evolution:** The software is modified to reflect changing customer and market requirements.

- **CS vs software engineering:** CS focuses on theory and fundamentals, SWE is concerned with the practicalities of developing and delivering useful software
- **SWE vs system engineering:** System engineering is concerned with all aspects of computer based systems development including hardware, software and process engineering. SWE is part of this more general process.
- **Key challenges for SWE:** Coping with increasing diversity, demands for reduced delivery times, and developing trustworthy software.
- **What are the costs of software engineering:** 60% of software costs are development costs, 40% are testing costs.
- **What differences has the web made to software engineering:** The web has led to the availability of software services and the possibility of developing highly distributed service based systems. Web based systems development has led to important advances in programming languages and software reuse.
- **Software products:**
  - **Generic products:** Standalone systems that are marketed and sold to any customer who wants to buy them.  
The specification of what the software should do is owned by the software developer and decisions on software change are made by the dev
  - **Customized products:** Software that is commissioned by a specific customer to meet their own needs.  
Specification of what the software should do is owned by the customer for the software and they make decisions on software changes that are required.
- **Good software:**

Product Characteristic	Description
Maintainability	Software should be written to evolve with changing customer needs, as software change is a critical requirement in a dynamic business environment.
Dependability and Security	Includes reliability, security, and safety. Dependable software prevents physical or economic damage and restricts access from malicious users.
Efficiency	Avoids wasteful use of resources like memory and processor cycles, emphasizing responsiveness, processing time, and memory utilization.
Acceptability	Software must be understandable, usable, and compatible with systems used by its intended audience.

- **General issues that affect software:**
  - **Heterogeneity:** Systems must operate as distributed networks across various types of devices, including computers and mobile devices.
  - **Business and Social Change:** Rapid societal and economic changes, driven by emerging technologies, require adaptable and quickly developed software solutions.
  - **Security and Trust:** Software is deeply integrated into our lives, making it essential to trust its reliability and security.

- **Scale:** Software must accommodate a vast range of applications, from small embedded systems in portable devices to large-scale, cloud-based systems serving global communities.

- **Application types:**

- **Stand-alone applications:** These are systems that run on a local computer, such as a PC, with all necessary functionality and no network connection required.
- **Interactive transaction-based applications:** Applications executed on a remote computer, accessed by users through their PCs or terminals. Examples include web applications like e-commerce systems.
- **Embedded control systems:** Software control systems managing hardware devices. These are numerous and found in a wide variety of hardware applications.
- **Batch processing systems:** Business systems designed to process large amounts of data in batches, converting multiple inputs into corresponding outputs.
- **Entertainment systems:** Primarily for personal use, these systems are intended to entertain users.
- **Systems for modeling and simulation:** Developed by scientists and engineers to model physical processes or situations, involving multiple separate and interacting objects.
- **Data collection systems:** Systems that collect data from their environment using sensors and send it to other systems for processing.
- **Systems of systems:** These are composed of multiple software systems working together.

- **Web services (internet SWE):** Web services allow application functionality to be accessed over the web

- **Cloud computing:** An approach to the provision of computer services where applications run remotely on the 'cloud'. Users do not buy software but pay to use

- **Web based SWE:** Web-based systems are complex distributed systems but the fundamental principles of SWE are as applicable to them as they are to any other types of system

- **Software reuse:** The dominant approach for constructing web-based systems. It involves assembling systems from pre-existing software components and systems.
- **Incremental and agile development:** Web-based systems should be developed and delivered incrementally. It is now recognized as impractical to specify all requirements in advance.
- **Service-oriented systems:** Implemented using service-oriented software engineering, where components are standalone web services.
- **Rich interfaces:** Technologies such as AJAX and HTML5 enable the creation of rich interfaces within a web browser.

- **SWE ethics:**

- **Confidentiality:** Engineers must respect the confidentiality of their employers or clients, regardless of whether a formal confidentiality agreement has been signed.
- **Competence:** Engineers should not misrepresent their level of competence and must avoid knowingly accepting work beyond their expertise.
- **Intellectual property rights:** Engineers should be aware of local laws related to intellectual property, including patents and copyrights. They must ensure the intellectual property of employers and clients is protected.
- **Computer misuse:** Engineers must not use their technical skills to misuse others' computers. Misuse can range from trivial actions, such as playing games on an employer's machine, to serious offenses like disseminating viruses.

- **ACM/IEEE code of ethics:**

- The professional societies in the US have collaborated to create a code of ethical practice.
- Members of these organizations agree to abide by the code of practice upon joining.
- The code includes eight principles guiding the behavior and decisions of professional software engineers, including practitioners, educators, managers, supervisors, policymakers, trainees, and students of the profession.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing, and maintenance of software a beneficial and respected profession.

In accordance with their commitment to the health, safety, and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. **Public:** Software engineers shall act consistently with the public interest.
2. **Client and Employer:** Software engineers shall act in the best interests of their client and employer, consistent with the public interest.
3. **Product:** Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. **Judgment:** Software engineers shall maintain integrity and independence in their professional judgment.
5. **Management:** Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. **Profession:** Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. **Colleagues:** Software engineers shall be fair to and supportive of their colleagues.
8. **Self:** Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

# Assembler

## 5.1 Introduction to the mainframe, assist, and TSO/ISPF

- **The mainframe:** A mainframe is a powerful, high-performance computer system designed for large-scale data processing and critical applications. It is widely used in industries that require high reliability, scalability, and security, such as banking, healthcare, government, retail, and telecommunications.
  - **High Reliability (Fault Tolerance):** Mainframes are designed for continuous operation, often running without interruption for years. They feature redundant components and sophisticated error detection and correction systems to minimize downtime.
  - **Massive Processing Power:** Mainframes can process vast amounts of data and handle thousands to millions of transactions per second, making them ideal for enterprise-scale applications like banking systems or airline reservations.
  - **Scalability:** Mainframes can handle an increasing workload by adding more processing power or storage without needing major architectural changes.
  - **Virtualization and Multitasking:** They support multiple operating systems and can run thousands of virtual servers simultaneously. This capability enables diverse workloads and maximizes resource utilization.
  - **Data Security and Compliance:** Mainframes have robust security features, including encryption, access control, and auditing, making them suitable for industries with stringent compliance requirements (e.g., financial regulations like PCI DSS).
  - **Batch and Online Processing:** They excel in both batch processing (processing large data sets at scheduled times) and online transaction processing (OLTP) for real-time applications.
- **Examples of mainframes:**
  - **IBM Z Series:** A popular series of mainframes developed by IBM, including the IBM z15 and z16, known for advanced encryption and AI integration.
  - **UNIVAC and BULL:** Earlier examples of mainframes used for enterprise computing.
- **z/OS:** z/OS is IBM's mainframe operating system designed to handle large-scale computing for enterprise applications. It is part of IBM's Z series mainframe platform and is known for its high reliability, scalability, and security. z/OS supports both legacy and modern workloads, making it the backbone for many mission-critical operations in industries like banking, healthcare, and government.
- **Assembly and assembler:** Assembly, or assembly language, is a low-level programming language that is closely tied to a computer's hardware architecture. It serves as a human-readable representation of the machine language instructions executed by a computer's central processing unit (CPU).

Assembly is one step above machine code, making it hardware-specific. Each instruction corresponds closely to a machine code instruction

Assembly uses mnemonics (e.g., MOV, ADD, SUB) to represent instructions, making it easier to understand compared to binary machine code.

Assembly code is specific to a particular CPU architecture (e.g., x86, ARM, IBM System/360). Code written for one architecture typically won't work on another without modification.

Provides direct access to hardware components such as CPU registers, memory addresses, and I/O ports, allowing precise control over the system.

An assembler converts the assembly code into machine code (binary) that the CPU can execute. The machine code is linked with libraries and loaded into memory for execution.

- **ASSIST:** ASSIST (Assembler System for Student Instruction and Systems Teaching) is an educational assembler designed to help students learn IBM Assembly Language programming (also known as System/360 or System/370 Assembly Language). It is a simplified, user-friendly tool tailored for academic settings, offering an approachable environment for understanding low-level programming concepts and mainframe assembly.
- **TSO (Time sharing option):** TSO, or Time Sharing Option, is an interactive environment in IBM's z/OS mainframe operating system. It allows users to access and interact with the system in a time-shared manner, as opposed to batch processing, where jobs are submitted and processed sequentially. TSO enables users to:
  - Enter commands interactively.
  - Run programs or scripts directly.
  - Edit and manage datasets.
  - Submit batch jobs for execution.
- **ISPF (Interactive System Productivity Facility):** ISPF, or Interactive System Productivity Facility, is a user-friendly, menu-driven interface that runs on top of TSO. It is essentially an application within TSO that provides a structured and more intuitive way to perform tasks on the z/OS system. ISPF offers:
  - **Menu Navigation:** A hierarchical menu system for accessing different functionalities.
  - **Dataset Management:** Create, edit, delete, and view datasets (files).
  - **Utilities:** Tools for sorting, copying, and comparing data.
  - **Programming Support:** Edit and compile source code, debug programs, and manage jobs.
  - **Custom Applications:** Organizations can add their own ISPF applications.

ISPF makes working with TSO more accessible by abstracting many complexities into guided menus and forms.

- **Datasets:** In the context of IBM mainframes and z/OS, a dataset is a structured collection of data stored on a storage medium. Datasets are the main way data is organized, stored, and accessed on mainframe systems. They are similar to files in other operating systems, but with more specific formats, attributes, and access methods.

Datasets in z/OS are categorized by how they are organized and accessed:

1. **Sequential Datasets (PS - Physical Sequential):**

- Data is stored in a linear fashion, similar to text files in other systems.

- Access is sequential (record-by-record in order).
- Example: Log files or batch job outputs.

**2. Partitioned Datasets (PDS) or Extended PDS (PDSE):**

- Contains multiple members (like a directory of files).
- Each member acts as a separate file.
- Commonly used for source code, JCL, and configuration data.
- Example: A PDS might contain different COBOL program files as separate members.

**Note:** A PDSE is sometimes called a "library." This is only because a PDSE, unlike a sequential data set, or "flat file," is separated into different members which are, in themselves, sequential files. Each of these members of a PDSE is somewhat like a book on a bookshelf in a library, hence the alias "library." So, a PDSE is a collection of members

## 5.2 Using ISPF

- **Logging off TSO/ISPF:** To properly sign off TSO/ISPF, press F3 while in the ISPF Primary Option Menu. If you have made changes while signed on, you will be presented the following screen:

The screenshot shows a terminal window titled "Vista TN3270 Session A". The main title is "Specify Disposition of Log Data Set". Below it, the text reads "Log Data Set (KC02322.S0W1.SPFL0G1.LIST) Disposition:" followed by a list of four options: 1. Print data set and delete, 2. Delete data set without printing, 3. Keep data set - Same (allocate same data set in next session), and 4. Keep data set - New (allocate new data set in next session). There are fields for "Batch SYSOUT class . . ." and "Local printer ID or writer-name . . . :". Below these, a message says "List Data Set Options not available". At the bottom, there are instructions: "Press ENTER key to complete ISPF termination." and "Enter END command to return to the primary option menu.". A "Job statement information" section follows, with a command line starting with "====> Command ==>". The bottom of the screen shows standard terminal keys: F1=Help, F2=Split, F3=Exit, F7=Backward, F8=Forward, F9=Swap, F12=Cancel. The status bar at the bottom right shows "0.0 06/16/15.167 12:56PM zos.kctr.marist.edu a 4.25".

Type the number 2 and press Enter and you will then be presented with a screen with a red-lettered READY displayed.

Type the word logoff or LOGOFF and press Enter and you will now be logged off.

- **Making a PDSE:** To allocate a PDSE, enter 3 for Utilities and press Enter. On this screen, enter 2 (Data Set) and press Enter

Move the cursor by tabbing or with your mouse to the line to the right of Project under ISPF Library: Type your project name. Tab again to the line to the right of Group and type your group name. Tab again to the line to the right of Type and type your type name. Your library will be *projectname.groupname.typename*

Move your cursor to the option line and type *a*

Only change or fill in the specific fields mentioned here: First, tab or use your mouse to move the cursor to the line to the right of Space units and enter TRKS for tracks. Tab twice and enter 10 for Primary Quantity. Tab again and enter 10 for Secondary quantity. Tab again and enter 5 for Directory blocks. Tab again and enter FB for Record format. Tab again and enter 80 for Record length. Tab again and enter 880 for Block size. Tab again and enter LIBRARY for Data set name type.

After pressing Enter and, if the data set does not already exist (and it shouldn't!), you will be presented the Data Set Utility panel again with the message Data set allocated in white lettering in the upper right hand corner of the panel: This indicates success! At this point, F3 back to the option page.

- **Naming things on the mainframe:** It is first important to know that names of entities on the mainframe can have 1 to 8 characters. They can only contain letters A-Z (upper case only), digits 0-9, and international characters \$, # and @ (dollar sign, pound sign/hash tag, and at sign). They can only begin with a letter or one of the three international characters

- **Creating PDSE members:** Enter 2 (Edit) at the Option ==> line and press Enter. The first time you enter this panel, you will need to fill in some fields that will be pre-filled the next time you come back to it.

Fill in the project, group, and type of the created PDSE. Enter a name for the member you would like to create and begin editing. If you type an existing member, you can begin editing that as well.

- **Editing PDSE members:** Follow the same instructions from above (creating pdse members) or you can press Enter in the Edit Entry Panel (with the member field blank) and move the cursor to the dot across from the name of the member you wish to edit and type either s, S, e, or E and press Enter.

**To insert a line while editing:** Move the cursor with the tab key or your mouse to the line numbers on the left hand side of the screen and anywhere within those 6 digits type the letter i or I. Press Enter to have the line inserted

To insert multiple lines while editing, do the above but follow the letter i or I with an integer between 2 and n. It will insert the number of lines you have requested but it will not scroll to show all of your inserted lines. It will fit as many on the panel as it can depending on where you began inserting the lines.

**To delete a line while editing:** Move the cursor with the tab key or your mouse to the line numbers on the left hand side of the screen and anywhere within those 6 digits type the letter d or D

**To move a line while editing:** Move the cursor with the tab key or your mouse to the line numbers on the left hand side of the screen and anywhere within those 6 digits type the letter m or M. Then, move the cursor with the tab key or your mouse to where you want the line moved and type either a or A for inserting the line you are moving after the line where your cursor is or type either b or B for inserting the line you are moving before the line where your cursor is. note that, when you start a move, you must complete it before you can go on editing. In other words, if you change your mind, you will still have to move the line but you can then delete it if necessary.

**To copy a line while editing:** Move the cursor with the tab key or your mouse to the line numbers on the left hand side of the screen and anywhere within those 6 digits type the letter c or C. Then, move the cursor with the tab key or your mouse to where you want the line copied and type either a or A for inserting the line you are moving after the line where your cursor is or type either b or B for inserting the line you are moving before the line where your cursor is.

**Deleting, moving or copying blocks, or multiple lines while editing:** To delete a block of contiguous lines, type dd or DD on the first line you want to delete and type dd or DD on the last line you want to delete. These two lines and every line in between will be deleted.

To move a block, use mm or MM on both the first line and the last line you want to move and the a or A for after or b or B for before as in item K above. The block of contiguous lines will be deleted from its original place.

To copy a block, use cc or CC on both the first line and the last line you want to copy and the a or A for after or b or B for before as in item K above. The block of contiguous lines will remain in its original place and a copy will be inserted where you indicated it to be inserted.

**To split a line of text:** To move the end of a line to the next line, or split the text, move the cursor with the tab key or your mouse to the line numbers on the left hand side of the screen and anywhere within those 6 digits type ts or TS for 'text split'. Then, before you press Enter, move your cursor to the character where you want to begin the split Now, press Enter. Everything from the character you indicated as the beginning of the split will be pushed and inserted two lines down with a new line inserted in between

**To collapse (hide) lines while editing:** To collapse one or more lines, move the cursor with the tab key or your mouse to the line numbers on the left hand side of the screen and anywhere within those 6 digits type x or X to collapse one line or xn or Xn to collapse n lines.

To collapse a block, type xx or XX on the first line you want to collapse and scroll to the last line you want to collapse and type xx or XX. Press Enter and a dashed line will appear telling you how many lines are collapsed.

**To uncollapse (reveal) lines while editing:** Type res or RES or reset or RESET on the command line and press enter and all of the lines will be uncollapsed or revealed.

To uncollapse or reveal some lines but not others, go to the dashed line and type f or F to reveal the first collapsed line or fn or Fn to reveal the first n collapsed lines. You can also type l or L to reveal the last collapsed line or ln or Ln to reveal the last n collapsed lines.

#### **Useful Primary Editing Primary Commands:**

- **LOCATE line-number:** Moves to the indicated line.
- **FIND string:** Finds the first occurrence of *string*, starting from the current line.

To find the next occurrence, press **PF5**.

- **CHANGE string-1 string-2 [ALL]:** Finds the first occurrence of *string-1*, starting from the current line, and changes it to *string-2*.

To find the next occurrence, press **PF5**.

To change the next occurrence, press **PF6**.

To change all occurrences, include the **ALL** option.

- **COPY member-name:** Retrieves data from the specified member; use an **A** or **B** line command to specify where the data should be placed.
- **PROFILE:** Displays the profile settings for the edit session.
- **RECOVERY [ON | OFF]:** Determines whether edit recovery mode is on.
- **UNDO:** Reverses the last editing change.
- **SAVE:** Saves changes and continues the edit session.

- **END (PF3/15)**: Saves changes and returns to the Edit Entry panel.
- **RETURN (PF4/16)**: Saves changes and returns to the Primary Option Menu.
- **CANCEL**: Returns to the Edit Entry panel without saving changes.
- **Saving members**: Type the word save or SAVE anywhere on the command line
- **Setting the Scroll ==> value**: It is strongly suggested that you change Scroll ==> PAGE to Scroll ==> CSR on every panel that you can in ISPF. ISPF will retain this setting if you exit the panel normally.
- **SDSF**: SDSF stands for System Display and Search Facility. It is a powerful interactive tool used on IBM z/OS mainframes for monitoring, managing, and controlling system resources and jobs. SDSF provides users with a user-friendly interface to view and interact with the output of batch jobs, system logs, and other critical system information.

Here are some typical commands used within SDSF

- **? or H**: Help.
- **OWNER \***: Displays all jobs owned by the user.
- **PREFIX jobname**: Filters jobs by their name or prefix.
- **FIND string**: Searches for a string in the output.
- **SORT column**: Sorts the list by the specified column (e.g., jobname, priority).
- **P jobname**: Purges a job from the queue.
- **H jobname**: Holds a job in the queue.
- **R jobname**: Releases a held job.
- **Jobs**: In the context of IBM mainframes and z/OS, a job refers to a unit of work submitted to the system for processing. Jobs are typically associated with batch processing, where multiple tasks or programs are executed without direct user interaction. Jobs are defined and controlled using Job Control Language (JCL), which specifies the tasks to be performed and the resources required.
- **Batch processing**: Batch processing is a method of executing a series of tasks or jobs on a computer without requiring user interaction during the process. It is commonly used in environments where large volumes of data need to be processed or repetitive tasks need to be automated. Batch processing is a fundamental concept in mainframe systems like IBM z/OS but is also used in other computing environments.
- **Viewing results in SDSF**: To review output in the output queue in SDSF on TSO/ISPF at Marist, enter SD (for SDSF) from the ISPF Primary Option Menu command line. From the SDSF Primary Option Menu enter ST for status on the command line. This will display the queue of completed jobs, both successful and unsuccessful.

Be sure not to let these completed jobs pile up. To get rid of jobs in the queue, put a P on the line in the margin just to the left of the job to be purged and press enter. A P can be entered on multiple jobs at once but the user may have to press enter a few times to get the jobs to roll off.

By the way, the user can enter SD.ST from the ISPF Primary Option Menu to go directly to the status queue. If somewhere else within TSO, the user can enter =SD.ST to go directly to the status queue. For example, if editing a PDS member and it has been saved, the user can submit his or her job by typing SUB on the command line. A red message will pop up if a successful submission has been made. Press enter again and then enter =SD.ST and press enter to go to the SDSF Status queue to see the results of the recently submitted job.

Once in SDSF status queue, select the job to be reviewed in the queue by typing a letter S in the margin just to the left of the job. It is important to review at least the first few 'pages' of output.

#### Action Characters Used in SDSF Status

- ?: Displays a list of the output data sets for a job.
- S: Displays one or more output data sets.
- H: Holds a job.
- A: Releases a held job.
- O: Releases held output and makes it available for printing.
- C: Cancels a job.
- P: Purges a job and its output

#### Action Characters Used in the SDSF Held Output Queue

- ?: Displays a list of the output data sets for a job.
- S: Displays one or more output data sets.
- O: Releases output and makes it available for printing.
- P: Purges output data sets.

### 5.3 Number systems and computer storage

- **The Decimal number system:** In the decimal system, any natural number  $m$  can be represented by use of the ten symbols 0, 1, 2, 3, ...9. These are the decimal digits,  $m$  can be represented as

$$d_n d_{n-1} d_{n-2} \dots d_1 d_0$$

Where  $m \geq 0$ . This same number  $m$  can be represented as

$$d_n \times 10^n + d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \dots + d_1 \times 10^1 + d_0 \times 10^0$$

For example, the natural number 123 can be represented as

$$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

The decimal system is also called the base ten system since ten digits are utilized in the number representations.

There is, however, nothing sacred about the base ten since the notion of a positional number system can be easily generalized to any given base  $b$  where  $b$  is a natural number greater than or equal to two.

- **Binary:** The binary number system is a base two number system, where any natural number  $m$  can be represented with the digits 0 and 1. Observe since

$$123 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

The number 123 would be represented in the binary system as

1111011

- **Hexadecimal:** The hexadecimal number system is a base 16 number system, which uses 0, 1, 2, ...9, A, B, C, ...F, where A, B, C, ..., F represent 10, 11, 12, ..., 15. Observe

$$123 = 7 \times 16^1 + 11 \times 16^0$$

Thus 123 has the decimal representation 7B

- **Numbers 1-30 in each system:** The following table gives the representations of the numbers zero through thirty-two in each of these three number systems.

Decimal	Hexadecimal	Binary	Decimal	Hexadecimal	Binary
0	0	0	17	11	10001
1	1	1	18	12	10010
2	2	10	19	13	10011
3	3	11	20	14	10100
4	4	100	21	15	10101
5	5	101	22	16	10110
6	6	110	23	17	10111
7	7	111	24	18	11000
8	8	1000	25	19	11001
9	9	1001	26	1A	11010
10	A	1010	27	1B	11011
11	B	1011	28	1C	11100
12	C	1100	29	1D	11101
13	D	1101	30	1E	11110
14	E	1110	31	1F	11111
15	F	1111	32	20	100000
16	10	10000			

- **Base notation:** For clarity, when it is not clear which system a given number is represented in, a subscript with the base of the system will be used. For example,

$$(123)_{10} = 7B_{16} = 1111011_2$$

- **Binary and hex to decimal:** Let  $a_n a_{n-1} \dots a_2 a_1 a_0$  be the representation of a number  $m$  in base  $b$ . Then, the decimal representation of  $m$  is given by the sum

$$d_n b_n + d_{n-1} b_{n-1} + \dots + d_2 b_2 + d_1 b_1 + d_0 b_0$$

Where each  $d_i$  is the decimal equivalent of the corresponding  $a_i$

For example, consider the binary number  $1011_2$ , then

$$1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 = 11$$

Next, consider the hex number  $A61$ . Observe

$$\begin{aligned} 1 \cdot 16^0 + 6 \cdot 16^1 + A \cdot 16^2 &= 1 \cdot 16^0 + 6 \cdot 16^1 + 10 \cdot 16^2 \\ &= 2657 \end{aligned}$$

- **Decimal to binary or hex:** Obtain the representation of a number  $n$  in a given base  $b$  from the representation of  $n$  in the decimal system by using the following steps

1. Divide  $n$  by  $b$ , giving a quotient  $q$  and remainder  $r$
2. Write the representation of  $r$  in the base  $b$  as the rightmost digit or as the digit to the immediate left of the one last written.
3. If  $q$  is zero, stop. Otherwise set  $n$  equal to  $q$  and go to step 1

For example, consider  $123_{10} \rightarrow h_{16}$ , where  $h$  is the hexadecimal representation. We follow the above procedure.

$$\begin{aligned} 123 &= 16(7) + 11 : B_{16} \\ 7 &= 16(0) + 7 : 7_{16} \end{aligned}$$

Since we hit a  $q = 0$ , we stop. The hexadecimal representation is then  $7B_{16}$ . Next, consider  $123 \rightarrow b_2$

$$\begin{aligned} 123 &= 2(61) + 1 : 1_2 \\ 61 &= 2(30) + 1 : 1_2 \\ 30 &= 2(15) + 0 : 0_2 \\ 15 &= 2(7) + 1 : 1_2 \\ 7 &= 2(3) + 1 : 1_2 \\ 3 &= 2(1) + 1 : 1_2 \\ 1 &= 2(0) + 1 : 1_2 \end{aligned}$$

Thus,  $123_{10} = 111011_2$

- **Conversion between binary and hex:** Because 16 digits are required in the hexadecimal system and  $2^4 = 16$ , a very simple algorithm exists for converting binary representations to hexadecimal representations, and vice versa.

The algorithm may be stated as follows

- Starting at the right of a binary representation  $n$ , separate the digits into groups of four. If there are fewer than four digits in the last (leftmost) group, add as many zeros as may be necessary to the left of the leftmost digit to fill out the group. For example, if  $n = 101101$ , the digits should be separated into two groups depicted as follows: 10 1101. Since the leftmost group does not contain four digits, two leading zeros are added to give 0010 1101.
- Convert each group of four binary digits to a hexadecimal digit. The result is the hexadecimal representation of  $n$ .

Consider  $n = 101101$ . Splitting into groups of four, we get the two groups

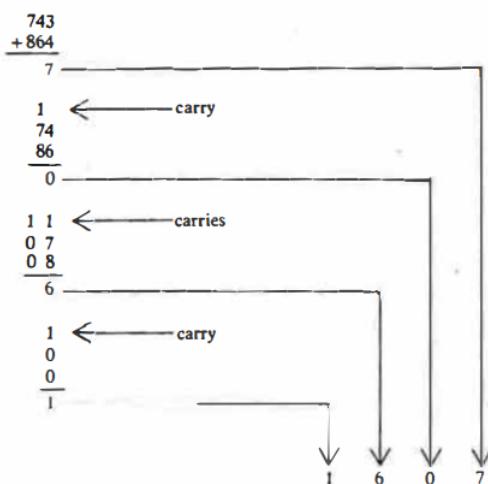
$$\begin{array}{r} 0010 \quad 1101 \\ \end{array}$$

Since  $0010 = 2_{10}$ , and  $1101 = 13_{10} = D_{16}$ , we get  $0010_2 = 2D_{16}$

- Addition of binary and hexadecimal numbers:** The algorithm for the addition of unsigned integer numbers is as follows

- Write the two addends one above the other with the rightmost digits of these numbers aligned.
- Add the two rightmost digits; if a 1 appears above these digits, indicating a carry, add 1 to the result. Write the integer portion of this result to the immediate left of the last recorded digit in the sum; If a carry is part of the result, write a 1 above the next higher order pair of digits. (If one or both of the digits do not exist, assume a value of 0 for the missing digits.)
- Delete the rightmost digits of the two addends. If the digits of the addends are exhausted, stop; otherwise, go to Step 2.

We start with a decimal system example. Suppose  $743_{10}$  and  $864_{10}$  are to be added. Then,



Collecting the results yields  $1607_{10}$ . The carry table for binary is simple, since there are only two digits involved.

+	0	1
0	0	1
1	1	$0 + c$

The result of using this table and the algorithm to find the sum of 10110 and 1011 is shown below



$$\text{Thus, } 10110_2 + 1011_2 = 100001_2$$

The carry table for hexadecimal addition is a bit more complex.

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0+c
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0+c	1+c
3	3	4	5	6	7	8	9	A	B	C	D	E	F	0+c	1+c	2+c
4	4	5	6	7	8	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c
5	5	6	7	8	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c
6	6	7	8	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c
7	7	8	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c
8	8	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c
9	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c
A	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c
B	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c	A+c
C	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c	A+c	B+c
D	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c	A+c	B+c	C+c
E	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c	A+c	B+c	C+c	D+c
F	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c	A+c	B+c	C+c	D+c	E+c

Let's add  $FCDE$  and  $9A05$



Thus, the result is  $196E3_{16}$

- **Subtraction of binary and hexadecimal numbers:** In the subtraction algorithm it is assumed that for  $a - b$ ,  $a \geq b$ .

1. Write the minuend above the subtrahend with the rightmost digits of these numbers aligned.

2. (a) If the rightmost digit in the minuend is greater than or equal to the corresponding digit in the subtrahend, subtract the digit in the subtrahend from the corresponding digit in the minuend and write the result to the immediate left of the last recorded digit in the difference; otherwise
  - (b) If the rightmost digit  $d$  in the minuend is less than the corresponding digit in the subtrahend, replace  $d$  by  $d + c$ , decrease the next-higher-order nonzero digit in the minuend by 1, replace any intervening zero digits in the minuend by the digit corresponding in value to the base minus 1. Then subtract the rightmost digit in the subtrahend from  $d + c$  and write the result to the immediate left of the last recorded digit in the difference.
3. Delete the rightmost digits in the minuend and subtrahend. If the digits of these two numbers are exhausted, stop; otherwise, go to Step 2.

- **Main storage (RAM):** A computer's main storage is typically RAM (Random Access Memory). It is the primary, volatile memory that temporarily stores data and instructions that the CPU needs while a computer is running. RAM is fast and allows quick access to data, but it loses its content when the computer is turned off.

Main storage can also refer to primary storage components, which include:

- **RAM:** Used for active processes and running applications.
- **Cache Memory:** High-speed memory located near or on the CPU for frequently accessed data.
- **Registers:** Small storage areas directly within the CPU for immediate operations.

It is distinct from secondary storage, such as hard drives (HDDs) or solid-state drives (SSDs), which provide long-term, non-volatile storage for data and programs.

- **Main storage organization:** The computers memory contains a certain amount of bits, say  $2^{30}$  bits. These bits are organized into groups of eight contiguous bits, called a *byte*. Bytes are assigned consecutive increasing hexadecimal numbers starting with the number zero. For example,

$\underbrace{11111111}_{\text{Byte 0}} \quad \underbrace{11111111}_{\text{Byte 1}} \quad \dots$

The number given to a byte is called the absolute *address* of the byte. It is often necessary to reference *fields* that contain several bytes. The address of such a field is considered to be the address of the first (leftmost) byte in the field.

If all of the bytes in storage are thought of as being grouped into consecutive non-overlapping pairs beginning with byte zero, the resulting pairs of bytes are referred to as *halfwords*. From this, we note that each address that is an even number is the address of a halfword, and only even numbers are halfword addresses. For this reason, each even address is called a *halfword boundary*.

Storage is also thought of as being organized into groups of four bytes, called *fullwords* (*or just words*), and groups of eight bytes, called *doublewords*

- **Binary representation of integer numbers:** Let  $n$  be the binary representation of any integer. Then, the *one's complement* of  $n$  is the result of replacing each 0 with a 1, and each 1 with a 0. For example, consider

Then, the one's complement is

0110

Further, if  $n$  is the binary representation of any integer, then the *two's complement* of  $n$  is formed by doing both of the following

1. Find the one's complement of  $n$
2. Add 1 to the result

Consider the binary number  $0101_2 = 5_{10}$ . The one's complement is 1010, adding one gives  $1010 + 1 = 1011_2$ . Thus, 1011 is the two's complement of 0101.

The 32-bit fullword is the unit of storage that was chosen in IBM computers for representing integer numbers. The method used for encoding integer numbers in fullwords is as follows

1. Any integer in the range 0 to  $2^{31} - 1$ , where  $2^{31} - 1 = 2,147,483,647$ , is represented in a fullword in its exact binary representation. Note that each 32-bit binary number in this range has a value of 0 in the leftmost bit position. This bit is called the *sign bit*, and all positive integer representations are characterized by having a sign bit value of 0.

If we have 32 bits, where the leftmost is the sign bit, then

$$1 + 2 + 4 + \dots + 2^{30} = 2^{31} - 1$$

Since

$$1 + 2 + 4 + \dots + 2^n = 2^{n+1} - 1$$

And a  $k$ -bit binary number has  $n = k - 1$ .

2. Any integer in the range  $-1$  to  $-2^{31} + 1$  is encoded by taking the two's complement of the encoded form of its absolute value. A representation of  $-2^{31}$  is also allowed, but this representation (a 1 followed by 31 0's) is not the two's complement of the representation of any positive integer. The sign bit in the encoded form of each negative integer has a value of 1.

For example, the encoded form of +1 is

00000000 00000000 00000000 00000001

The encoded form of  $-1$  is then found by taking the two's complement of +1. That is,

$$\begin{aligned} & (11111111 \ 11111111 \ 11111111 \ 11111110) + 1 \\ & = 11111111 \ 11111111 \ 11111111 \ 11111111 \end{aligned}$$

There are two integers with encoded forms that are identical to their two's complement. These integers are 0 and  $-2^{31}$ .

Some rather remarkable things are true of the binary representations of integers in this scheme

- The two's complement of the representation of a negative integer (with the exception of  $-2^{31}$ ) is the representation of the absolute value of that integer
- When binary addition is performed on the representations of integer numbers (whether the signs are mixed or not), the result has the correct value in the sign bit (provided the result is in the range  $-2^{31}$  to  $2^{31} - 1$ )

**Note:** The range of a fullword is  $-2^{31}$  to  $2^{31} - 1$ .  $-2^{31}$  is represented as

10000000 00000000 00000000 00000000

- **Integer addition and subtraction:** Integer addition is performed by simply performing the usual binary addition. Integer subtraction is performed by first taking the two's complement of the subtrahend and then adding the result to the minuend. There is no need for the sign bits of the integer representations of the numbers involved in these operations to be checked before the operations are performed, for the sign bit of the result will be correct as noted above.

To conserve space and since 32-bit binary numbers are all but impossible to read at a glance, the printouts of the conditions of memory locations are always given in hexadecimal form. The printed forms of the representations of 1 and  $-1$  are therefore

00000001

and

*FFFFFFF*

- **Overflow:** Overflow occurs when operations are performed on the representations of integer numbers with the effect that the carry into the sign bit of the result differs from the carry out of that position. Consider the following addition operations on integers coded in five-bit binary numbers

$$\begin{array}{r}
 0 \quad 1 \\
 0 \quad 1 \quad 0 \quad 0 \quad 0 \\
 + \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \\
 \hline
 1 \quad 0 \quad 0 \quad 0 \quad 1
 \end{array}$$

Notice that the carry into the sign bit differs from the carry out of the sign bit... *overflow!* If these carries matched, the result would be valid.

- **Understanding signed numbers in the two's complement system:** Consider an 8-bit integer. Recall that for an  $n$ -bit signed integer, the range is

$$-2^{n-1} \text{ to } 2^{n-1} - 1$$

The maximum occurs when the sign bit is zero, and all other bits are one. Observe

$$01111111_2 = 127_{10}$$

The minimum occurs when the sign bit is one and all other bits are zero

$$10000000_2 = -128_{10}$$

If we have 8 bits to play with, then the total number of 8-bit permutations is  $2^8 = 256$ . Dividing this by two gives 128 non-negative numbers and 128 negative numbers. Let's consider these sets.

$$\text{Non-negative } \{0, 1, 2, \dots, 127\} \qquad \text{Negative: } \{-1, -2, -3, \dots, -128\}$$

If we consider the cardinalities of these sets,

$$\begin{aligned}
 |\{0, 1, 2, \dots, 127\}| &= 128 \\
 |\{-1, -2, -3, \dots, -128\}| &= 128
 \end{aligned}$$

The union of these two sets give us our 256 total 8-bit permutations.

**Note:** To interpret a signed hexadecimal value as an integer in a two's complement system, you must first convert it to binary. This is because the two's complement representation inherently operates on binary numbers

If the leftmost hex digit is  $8 - F$ , the value is negative, whereas  $0 - 7$  is positive.

- ***n-bit addressing:***  $n$ -bit addressing refers to a memory addressing system where each memory address is represented using 24 bits. This addressing scheme allows a system to address up  $2^n$  to unique memory locations.

The IBM System/370 uses 24-bit addressing. Thus, each memory address is represented using 24 bits (3 bytes/6 hex digits), and there are  $2^{24} = 16,777,216$  unique memory locations (16mb)

The first byte of memory has the hex address 00 00 00, the second has 00 00 01, up to the address with value  $2^{24} - 1$ , which is *FF FF FF*

- ***General purpose registers and relative addressing:*** Main storage (memory) was introduced earlier as a hardware unit into which data could be stored in the form of binary numbers. Modern IBM computers also contain 16 general purpose registers as units for storing and manipulating data. Each register is a 32-bit storage unit whose contents can be altered or accessed in much less time than it would take to alter or access a field in main storage. Because of the difference in response time, the registers are often used as storage for frequently referenced data items or as operands in arithmetic operations.

The 16 registers are numbered 0, 1, 2, ..., 15 and will be referred to as *R0, R1, ..., R15*

One important use of registers involves the concept of addressing. Every byte of storage has associated with it an absolute address, and every field is addressed by the address of its leftmost byte. Recall that the number given to a byte is called the absolute *address* of the byte. It is often necessary to reference *fields* that contain several bytes. The address of such a field is considered to be the address of the first (leftmost) byte in the field.

Whenever a program is run, it must be stored somewhere in main storage. Thus, each instruction and each item of data in the program will have an absolute address. In a program, when a reference is made to a data item or to an instruction, the computer must have an absolute address to find what is referred to.

A difficulty arises immediately: The writer of a program does not know where in storage his program will be stored when it is run and therefore does not know any of the relevant absolute addresses. Some other means has to be provided for referring to data or instructions within the program.

The means that is provided is called *base-displacement addressing* or *explicit addressing*. The idea is that within a program the relative positions of any two statements are fixed, such that if the absolute address of any one statement in the program were known, then the address of any other statement could be calculated. To do this, one would compute the distance between the statement whose absolute address is known and the one whose address is to be found. Such a distance, measured as a number of bytes, is called a *displacement*.

But how, you ask, can the absolute address of even one statement in the program be known? For now, it is sufficient to know, when a program is executed, such an address will be contained in at least one of the general purpose registers. A register that holds an absolute address, from which the addresses of other statements can be calculated, is called a *base register*.

The standard way, then, to refer to any data item or instruction in a program is to specify a base register and a displacement. When the instruction containing the reference is executed, the sum of the displacement and the absolute address in the base register will be calculated to obtain the absolute address of the item or instruction.

As an example, if the displacement specified were 4, the base register specified were  $R1$ , and  $R1$  contained 0000007C, the absolute address formed would be 000080

Note that in a 24-bit environment (with 24-bit addressing), the contents of the registers will always be 32-bits, but the absolute addresses will remain 24-bit.

Base-displacement addresses are often specified in the format

$$D(B)$$

Where  $D$  is the displacement, expressed as a decimal number in the range 0 to 4095, and  $B$  is the number of the base register.

For example, consider the base-displacement addresses.

$$4(1) \quad 20(13) \quad 0(11)$$

In the first example  $4_{10} = 4_{16}$  is added to the contents of  $R1$ . In the second example,  $20_{10}$  is converted to  $14_{16}$  and added to the contents of  $R13$ . In the third example, the contents of  $R11$  give the desired address without modification.

Occasionally, it is convenient to express an address as the sum of a displacement, the contents of a base register, and the contents of a second register as well. The additional register is called an *index register*; the format for an address that includes an index register is

$$D(X, B)$$

Where  $D$  and  $B$  have the same meaning as above, and  $X$  is the number of the index register. Consider

$$4(7, 1)$$

Is a  $D(X, B)$  address, using  $R7$  as the index register,  $R1$  as the base register, and 4 as the displacement. If  $R1$  contains 0000007C and  $R7$  contains 00000010, the address calculated will be the sum of the contents of the two registers plus the displacement, or 000090

It should be noted that  $D(B)$  or  $D(X, B)$  addresses never stand alone, they are always used as part of a particular instruction. The rules for the instruction always specify which of the two addressing formats should be used.

There are three details that qualify the above remarks

1. When the contents of an index register or a base register are used to calculate an address, only the value represented by the rightmost 24 bits of the register is used in the calculation. Thus, for the purpose of calculating an address, the value added from a register must always be in the range 000000 to FFFFFF. As an example, suppose that R1 contained FFFFFFFF (the encoding of -1). Then the effective address derived from 0(R1) is not negative; rather it is FFFFFF, an extremely large absolute address.
2. In either the D(B) or the D(X,B) address format, the registers may be omitted. This means that when a D(B) address is required, the programmer may use either of the following forms:

Form	Example
D(B)	42(15)
D	42

Similarly, when a D(X,B) address is required, the programmer may use:

Form	Example
D(X,B)	42(1,14)
D(X)	42(1)
D(,B)	42(,14)
D	42

Note that if only one register is specified in the D(X,B) address format, the presence or absence of a comma determines whether the register given is considered an index register or a base register.

When registers are omitted from a D(B) or D(X,B) address, the calculation of the corresponding absolute address is performed by using 0 in place of the contents of the omitted registers. Thus, either 42(14) or 42(,14) results in the same absolute address. 42(,14) is converted to an absolute address by adding the binary equivalent of decimal 42 to the binary number represented by the rightmost three bytes of the contents of R14.

3. *R0* should not ordinarily be used as an index register or as a base register. If *R0* is specified in an address, it will be taken to mean that the corresponding register has been omitted. (As an example, both 4(,15) and 4(0,15) refer to the same absolute address.)
4. **Note about register zero as index or base:** In assembly language (e.g., for IBM System/360 or z/Architecture), the address specification *D(X,B)* refers to a displacement *D* an index register *x*, and a base register *B*. When an index or base register is omitted, it is implicitly treated as 0.

Register 0 (R0) is often special in many architectures. When used as an index register (x) or base register (B), it is ignored in the address calculation, effectively contributing 0 to the address.

## 5.4 Basic concepts

- **The function of a program:** The modern computer can perform millions of operations per second. If these operations occur in the proper sequence, meaningful results will be obtained that might have been totally inaccessible otherwise. The function of a program is to direct the order in which operations are executed within the computer.

The creation of programs that correctly direct the sequence of operations is, therefore of fundamental importance if computers are to be successfully utilized.

- **Type of assembly on IBM mainframes:** The language used in this text is a symbolic language called *Basic Assembler Language*, this is the assembler language developed for use on IBM mainframe computers.

Note that the computer cannot directly carry out or execute assembler language instructions. Each of these instructions must be translated into a machine-language instruction before the computer can begin to perform the task that the program was intended to accomplish.

The machine language instructions that the computer can execute directly are strings of binary digits that are difficult for a human to decipher. A program consisting entirely of machine-language instructions is called an *object program*.

- **The Job Control Language (JCL):** After an assembler language program has been prepared, a few additional instructions in JCL are added to the program and the resulting program is presented to the computer through an input device, such as a card reader or a terminal. The JCL instructions invoke a program called an *assembler*, which translates the source program instructions into machine-language instructions and thus produces an object program.

If no syntax errors are detected, the object program is loaded into the main storage of the computer and execution of the program begins.

- **Execution of the program:** The actual execution of the program is depicted by the following algorithm

1. Initially, the absolute address of the first instruction of the program to be executed is inserted into a special pointer called the **Program Status Word (PSW)**.
2. The machine retrieves from storage the instruction that is pointed to by the PSW.
3. The machine then updates the contents of the PSW to point to the next instruction.
4. The machine executes the operation indicated by the previously retrieved instruction. If the instruction did not cause a branch to occur (a branch is caused by a basic operation analogous to a **GOTO** statement in a higher-level language), then go to **Step 2**. Otherwise, put the absolute address that is to be branched to into the PSW, and go to **Step 2**.

- **R14 and R15** In ASSIST, general purpose register 15 (R15) is used to hold the address of the first byte of the program. R14 holds the return address, the address to return to after the program ends. In ASSIST virtual memory, it is simulated that the program starts at address 0

- **Explicit addressing:**  $D(X, B)$  displacement addressing is *explicit* addressing
- **Types of Instructions:** The process of encoding instructions can be clarified by considering some specific instructions in their symbolic and encoded forms. Instructions are encoded according to five distinct formats, RR, RX, RS, SI, and SS. The set of instructions encoded in the RR format is referred to as the *RR instructions*.
- **RR Instructions, add register (AR):** An RR instruction is most often used to cause an operation involving two registers.

Consider the format of the Add Register instruction

$$AR \quad r1, r2$$

Execution of this instruction causes the contents of r2 to be added to the contents of r1; the contents of r2 are unaltered, unless r2 is r1. If an overflow occurs in the process of addition, a fixed point overflow condition will exist, this will normally cause termination of the program.

This symbolic form must be translated into an encoded form before the instruction is actually executed. The encoded form of any RR instruction is a 16-bit binary number that will occupy a halfword of storage in the machine when the program is executed. The encoded instruction can therefore be represented as a four digit hexadecimal number. The precise format of an encoded RR instruction is

$$h_0 h_0 h_{r1} h_{r2}$$

Where

- $h_0 h_0$  is a two-digit operation code specifying the purpose of the instruction.
- $h_r$  is the number of the register (0-F) that is to be used as the first operand.
- $h_r$  is the number of the register that is to be used as the second operand.

The operation code (**op code**) for an **AR** instruction is 1A. Therefore, the encoding of:

$$AR \ 14, 0$$

is 1AE0. The 1A means that the encoded instruction is an **AR**, the E indicates that the first operand is R14, and the 0 indicates that R0 is the second operand.

- **RR Instructions, SR (Subtract register) and LR (Load register):** Consider the instruction

$$SR \quad r1, r2$$

Execution causes the number represented by the contents of r2 to be subtracted from the number represented by the contents of r1. Just as for AR, r2 is unaltered. Fixed-point overflow can occur. Next, consider

$$LR \quad r1, r2$$

Causes the replacement of the original contents of r1 by the contents of r2, r2 remains unaltered.

The opcode for SR is 1B, and the opcode for LR is 18

- **RX instructions: L (load):** RX instructions usually cause an operation that involves a register and main storage to be performed. For example, consider the Load instruction

$$r, D(X, B)$$

Which causes the fullword in storage starting at the effective address derived from  $D(X, B)$  to be loaded into  $r$ . The original contents of  $r$  are replaced, while the fullword in storage remains unaltered

Three errors might occur during the execution of a load instruction

1. If the absolute address calculated from  $D(X, B)$  is not a multiple of 4 (not a fullword boundary), a *specification exception* occurs.
2. If  $D(X, B)$  is an address greater than any actual storage address, an *addressing exception* will occur. This exception can happen only if either the  $X$  or the  $B$  register contains an excessively large number.
3. If  $D(X, B)$  is the actual address of an area, but the address is not within the area of storage allocated to the program, a *protection error* will occur.

The encoded form of an RX instruction occupies two halfwords and conforms to the following format:

$$h_0 h_0 h_r h_X \quad h_B h_D h_D$$

where

- $h_0 h_0$  is the op code indicating the particular instruction
- $h_r$  is the number of the register used as the first operand
- $h_X$  is the number of the index register (0 if omitted)
- $h_B$  is the number of the base register (0 if omitted)
- $h_D h_D h_D$  is the displacement

Thus, since the op code of L is 58,

$$L \quad 2, 12(1, 10)$$

is encoded as 5821A00C. Note that the displacement used in a relative address must be in the range 0 to 4095 simply because  $FFF_{16} = 4095_{10}$  is the largest three-digit hexadecimal number (and only three digits are provided in the encoded form of the instruction).

- **RX: ST (store):** The ST instruction performs the inverse of the operation of the L instruction:

$$ST \quad r, D(X, B) \quad (\text{Store})$$

Execution of this instruction causes the contents of  $r$  to replace the fullword at the location determined by  $D(X, B)$ ; the condition of  $r$  remains unaltered. The absolute address corresponding to  $D(X, B)$  must, therefore, be on a fullword boundary. The same exceptions (errors) that can occur for an L instruction can also occur for an ST instruction.

The opcode of *ST* is 80

- **A complete program:** Given a choice, a programmer would naturally choose to write programs using the symbolic instructions. The role of an assembler program is to accept as input a source program written using symbolic notations and to encode the instructions into executable format. The precise format used to represent a symbolic instruction that can be submitted to the assembler is as follows.

1. A label may start in column 1. A label is a string of from one to eight characters such that:
  - (a) The first character is either a letter from A to Z, a \$, a #, or an @.
  - (b) Each of the characters following may be any of the above or any one of the decimal digits (0, 1, 2, ..., 9).

Thus,

```
WORD1
LOOP
MYROUTNE
BRANCH01
```

are valid labels.

2. A mnemonic for the operation code, such as AR for Add Register, starts in column 10. (Appendix C contains a list of valid op codes and the corresponding mnemonics.)
3. The operands start in column 16.
4. Any comment that the user wishes to append to the instruction can occur after allowing at least one blank following the last character of the operands. The only restriction on the comment is that it cannot extend past column 71 of the input record.

The following example of a symbolic instruction, with indications of where the fields would appear, illustrates the above points:

```
1      10      16
LOADUP    L      1,0(2,3)      LOAD THE WORD INTO R1
```

Up to this point, the only symbolic instructions that have been discussed are those that are to be encoded into executable instructions. There are, however, several instructions, called *assembler instructions*, that are not used to generate machine instructions. Rather, these instructions are used to communicate to the assembler information about how the user's program is to be processed and to direct the generation of constants and storage areas. The following assembler instructions are of particular importance:

1. The CSECT instruction is used to begin a program and must appear before any executable instructions in the program. The format of the CSECT instruction is:

```
label      CSECT
```

2. The end of a program is signified by an END statement. The END statement has as an operand the label of the place in the program where execution of the program should begin. Normally this label is the label of the CSECT statement. For example, the CSECT and END statements would typically occur as:

```

    MYPROG      CSECT
    .
    .
    .
    END  MYPROG

```

3. A DC statement is used to define constants. The only form of a DC statement that is needed at this point is:

```

label      DC      mF'n'

```

where

- $m$  is a nonnegative decimal integer (duplication factor)
- $n$  is a decimal integer

If  $m$  is omitted, it is assumed to be 1. The effect of this DC statement is to cause  $m$  consecutive fullwords, each containing the number  $n$ , to be generated. Thus,

```

TWO      DC      F'2'

```

causes one fullword containing 00000002 to be generated and makes it accessible to the program by its symbolic name TWO. DC statements can appear anywhere between the CSECT and END statements in the program. The fullwords that are generated as the result of such DC statements will always begin on fullword boundaries.

4. A DS statement is used to set aside areas of storage for fullwords that require no initial values. The format of the DS statement is:

```

label      DS      mF

```

The above statement causes the assembler to set aside the next  $m$  fullwords as a storage area. An area of  $n$  bytes, which may or may not begin on a fullword boundary, can be set aside by use of a DS statement in the following format:

```

label      DS      CLn

```

Here, CL $n$  stands for Character, Length  $n$ . Again, if more than one such area is required, a duplication factor can be used. For example, the statement:

```

label      DS      mCLn

```

causes  $m$  fields, each of which has a length of  $n$  bytes, to be set aside.

Before looking at a sample program, two additional details should be mentioned:

- When execution of a program begins, R15 will always contain the absolute address of the beginning of the program. For now, this can be assumed to mean that R15 contains the address of the first generated instruction in the program. This fact is significant since it allows all relative addresses to be created using R15 as a base register
- When execution of a program begins, R14 contains the absolute address of the routine that should be branched to when execution of the program has been completed. That is, when execution of a program terminates, branch to the address in R14 should be effected. This is referred to as *exiting* from the program. Although branching instructions are not covered until later in this text, the format of the branch instruction that causes the exit is

label BCR B'1111',14

**Note:** The encoding of BCR B'1111',14 is 07FE (important when we look at dumps)

- **Sample program:**

```
ADD2      CSECT
          L      1,16(,15)
          L      2,20(15)
          AR     1,2
          ST      1,24(15)

          BCR    B'1111',14
          DC     F'4'
          DC     F'6'
          DS     F
          END    ADD2
```

Note that any input record with an asterisk in column 1 is a comment. The comment can be anywhere from column 2 through column 71.

The actual instructions in this program that will be translated into executable code are

```
L      1,16(,15)
L      2,20(15)
AR    1,2
ST    1,24(15)
```

The assembler will convert these into machine code and produce a listing of the following form.

LOC	OBJ CODE	SOURCE	STATEMENT
000000	5810 F010	L	1,16(15)
000004	582F 0014	L	2,20(15)
000008	1A12	AR	1,2
00000A	501F 0018	ST	1,24(15)
00000E	07FE	BCR	B'1111',14

In the first instruction, R15 is used as a base register. In the other RX instructions no base register is used, and R15 is used as an index register. When only one of the two registers is specified in a D(X,B) address, it makes no difference in the execution of the program whether it is used as the X or the B register.

The constants are generated starting at location 000010 and the listing produced from the DC and DS statements is:

LOC	OBJ CODE	SOURCE	STATEMENT
000010	00000004	DC	F'4'
000014	00000006	DC	F'6'
000018		DS	F

Note that the DS statement generates no object code.

The END statement marks the end of the program and contains the label of the starting point as the operand. Normally, this is the label that is used in the CSECT statement.

- **The using statement:** The USING directive is used in IBM's System/360 Assembly Language (including the ASSIST simulator) to tell the assembler which register should be used as a base register for address resolution. It enables the assembler to generate base-displacement addressing automatically.

```
PROGRAM1    CSECT
            USING  PROGRAM1,15
```

Sets us *addressability* off of R15. In this way, we can use named vars instead of  $D(X, B)$

- **Intro to LTORG and using named variables:** The LTORG directive forces the assembler to dump the current literal pool (a set of constants stored in memory) at that point in the program.

If we wrote the sample program above as

```
ADD2      CSECT
          USING ADD2,15

          L      1,NUM1
          L      2,NUM2
          AR     1,2
          ST      1,RESULT

          BCR   B'1111',14

LTORG

          NUM1   DC    F'4'
          NUM2   DC    F'4'
          RESULT DS    F

END      ADD2
```

Then we see we can use the named variables instead of directly calculating the  $D(X, B)$  address. This form of addressing is called *implicit* addressing. In doing this, any additional instructions that may shift the bytes of our constants won't force us to recalculate the  $D(X, B)$  addresses

- **Note about LTORG:** The LTORG directive needs to start at a double word boundary. If LTORG does not naturally start at a double word boundary, it will go to the next available. This means we might get "slack bytes", which are unused bytes from the end of the instruction before the LTORG up until the double word boundary where the LTORG begins.
- **Implicit and explicit addressing:** A  $D(X, B)$  displacement (also called relative) address is termed *explicit*, whereas using a label is *implicit*

- **The location counter (LOC):** The location counter is a hidden variable maintained by the assembler that tracks the current address where instructions and data are assembled in memory.

In ASSIST, the LOC starts at 000000, it increments as instructions and data are defined.

- **Literals in IBM Z/OS and ASSIST:** Consider the add RX instruction, we could write

```
o A R,D(X,B)
```

Adds the contents found at the relative address  $D(X, B)$  to the contents of the given register  $R$ . But can also replace the relative address with a *literal*. Consider

```
o A 5,=F'3'
```

$F'3'$  specifies a fullword (4-byte) integer constant with the value 3. The assembler automatically places literals in a literal pool (usually at the end of a program block). The instruction fetches the value from memory, not from a register.

If you need to use the same literal multiple times, you have two options:

1. Use the Same Literal Again (Assembler Handles It) The assembler will store only one instance of  $=F'3'$  in the literal pool, even if you use it multiple times:

```
o A 6,=F'3' ; Adds 3 to Register 6
1 A 7,=F'3' ; Adds 3 to Register 7 (same literal, no
              → duplicate in memory)
```

2. If you need direct access to the constant, define it explicitly with a DC statement

**Note:** We note that the  $F$  in a DC statement like  $DC F'...'$  guarantees that the full word will be placed at the beginning of a fullword boundary

- **A few basic RR and RX instructions**

Suppose we have the named variable

```
o NUM1    DC F'36'
```

### RR (2 Byte instructions):

- **LR (Load register):** LR 1,2 load contents of register two into register one
- **AR (Add register):** AR 1,2 Add contents of R2 into and over contents of R1
- **SR (Subtract register)** SR 1,2 Add contents of R2 into and over contents of R1

### RX (4 Byte instructions):

- **L (Load)**: L 7,NUM1 Load contents of NUM1 into register 7
- **ST (Store)**: ST 10,NUM1 Store contents of R10 into NUM1
- **LA (Load address)**: LA 3,NUM1 Load address of the first byte of NUM1 into R3
- **A (Add)**: A 4,NUM1 add contents of NUM1 to contents of R4
- **S (Subtract)**: S 9,NUM1 subtract NUM1 from R9
- **Note about opcodes**: If the opcode has rightmost hex digit
  - 0-3: 2 byte instruction
  - 4-B: 4 byte instruction
  - C-F: 6 byte instruction
- **Note about fullword boundaries**: An address is at the start of a fullword boundary if the last digit is 0, 4, 8, or C
- **Note about double word boundaries**: A double word boundary is an address that ends with either 0 or 8.

CSECTS (program start) and LTORGs always begin on a dword boundary. In AS-SIST CSECT will begin at a dword boundary no matter what because our program starts virtually at address zero (which is a doubleword boundary).

- **A note about literals**: Consider the instruction

o L 3,=F'235'

This takes the decimal literal 235 and loads it into register 3. However, this is a sloppy way to do it. This instruction uses 8 bytes, 4 for the encoded instruction, and 4 for the fullword literal. Instead, we can do

o LA 3,235

Since LA needs a  $D(X, B)$  address, 235 is shorthand for  $235(0, 0)$ . Since we ignore R0 in the address calculation, the above instruction loads the "address"  $235_{10} = EB_{16}$  (plus nothing because we ignore R0) into R3. In this way, we skip needing to store the literal 235 in storage. Therefore, the above instruction only uses 4 bytes instead of 8.

Further, consider a situation where we need to "zero out" a register. We could write

o L 8,=F'0'

Which replaces the contents of R8 with zero, effectively "zeroing out" the registers contents. Again, the above instruction uses 8 bytes. Instead, we could write

o SR 8,8

Which subtracts the contents of register 8 from register 8. This instruction therefore only uses two bytes... The two bytes required to encode the instruction.

- **A note about define storage:** If we specify a value in a DS instruction, the value won't be placed in storage. It will behave just like a regular DS.
- **Note about instructions Load and Store:** The instructions L and ST require arguments at fwb's (full word boundarys), failure to supply a value at a fwd results in a ABEND (SOC6), which is a *specification exception*
- **Note about assist:** The ASSIST assembler fills all bytes in storage with a default value F5, and non special registers with value F4F4F4
- **XDUMP:** If we write an XDUMP instruction with no operands, we get a output dump of the contents of all the registers. If we specify a location in memory, and a number of bytes after that location, we get those requested bytes but they will be buried within the 32 byte line in which they are contained. (Dump gives us 32 byte line starting at previous 32 byte boundary).
- **Assist and ABENDS:** In ASSIST, ABENDS are called *assist completion dumps*

Note that a return code of zero means that assist finished successfully, the return code will always be zero even if the program abends

- **Assembler (compilation) errors and execution (runtime errors):** An abend is always an execution (runtime) error, we cannot get abends during compilation.

The first pass of the assembler looks for syntax errors and addressability errors. The program will not compile if everything in the program is not addressable.

- **Dumps and the PSW:** A dump of storage will begin by specifying the address of the first byte of the dumped storage

An ABEND dump will dump the registers contents, the storage that our program occupies, and the PSW (program status word). The PSW is 8 bytes. There is one PSW per cpu and each instruction changes the contents.

The first two bytes of the PSW we should not worry about. The last two bytes of the first four bytes contain the interruption code, the ones we should encounter at this point are the following

- **SOC 1 (Operation exception):** Invalid opcode / instruction
- **SOC 4 (Protection exception):** Good address, outside the scope of the program
- **SOC 5 (Addressing exception):** Bad address
- **SOC 6 (Specification exception):** Happens for a number of reasons, most commonly when you an instruction requires storage on a fullword boundary, but we give storage that is not on a fullword boundary. Load and store are examples of instructions that require the second operand to be on a fullword boundary
- **SOC 9 (Fixed-point divide exception):** divide by zero
- **SOC B (Decimal-divide exception):** divide by zero

The next piece of information that is of concern to us is the first 4 bits of the fifth byte of the PSW. The first two bits are ILC (instruction length count), which is the length of the instruction that caused the abend (in halfwords). Therefore, to get the length of the instruction in bytes, we multiply by two. The last two bits is the condition code set by the last successful instruction that sets the condition code.

To process the information of the first four bits of the 5th byte of the PSW, we take the hex number and convert to binary, the split into two different 2 bit binary numbers. Since the ILC has max value 11, the biggest an instruction is therefore  $3(2) = 6$  bytes. Suppose we our four bits are

1000

We then split into two different 2-bit binary numbers

10 00

The ILC is therefore 10, which means instruction length  $2(2) = 4$  bytes. And, the condition code is 0.

The last 3 bytes of the 8 byte PSW is the next instruction address, which is the address of the instruction following the one that caused the abend.

- **Uses a register as a counter:** Suppose we want to register 3 as a counter variable. To begin, we clear out its contents

o SR 3,3 // SR R,R to clear contents of R

Then, to increment the count. That is, to add one to its contents, we do

o LA 3,1(,3) // General: LA R,1(,R)

Suppose this is the first time we increment the registers contents. Thus, before the LA, its contents will be zero. LA 3,1(,3) loads the "address" one byte off of register 3. Since the contents are zero, one byte off of register three is 1.

After the first increment, the registers contents are now two. A second LA will load the "address" 1 byte off of 1 (R3s contents), which is two.

- **Condition codes and the bitmasks in branches:** Some instructions set the condition code. Recall from analysis of the PSW that the condition code can either be 0,1,2, or 3 (cc is 2 bits). When we see something like B'0111' in a branch, what we are looking at is a *bitmask*. The bitmask B'0111' deals with the condition code, it says that if the condition code is zero, don't branch. If the condition code is 1,2, or 3, branch.

$B'$	$\underbrace{0}$	$\underbrace{1}$	$\underbrace{1}$	$\underbrace{1}$	'
	Don't branch	Branch if cc if cc is one	Branch if cc is two	Branch if cc is three	

- **Read from files and process data:** In ASSIST assembly, to read data from files we use three X type (assist) instructions

- **XREAD**
- **XPRNT**
- **XDECI**

Consider the following assembler program that reads data from a file and outputs

```

0      XREAD RECORD,80
1  LOOP1   BC    B'0111', ENDLOOP1
2      XPRNT DETAIL,133
3      XREAD RECORD,80
4      BC    B'1111',LOOP1
5  ENDLOOP1
6      LTORG
7  RECORD  DS    CL80

```

Note that XREAD sets the condition code. Zero if successful, 1,2, or 3 if we are at the end of the file (no more reading to do)

We begin by doing a preliminary read, This ensures our file is not empty, if the file is empty, we don't loop. The condition code will be nonzero, and we take the branch to ENDLOOP1 immediately

The LOOP1 is a label, which allows us to execute branch instructions to that label. Notice that we defined storage named record, which is character length 80, each character is one byte, the total buffer is therefore 80 bytes (or 80 characters) long. For each XREAD, this buffer reads 80 bytes of the file, and overwrites the contents of the RECORD buffer. At the beginning of each loop iteration, we check to see if the condition code set by the last XREAD was zero. If it was, we do not branch to ENDLOOP1 (thus continuing the loop).

If the loop enters, we XPRNT (more on xprnt later), then read again, and branch unconditionally to the loop condition check

After each XREAD, our RECORD buffer is filled with space separated data. We use XDECI to process the data in the RECORD buffer. XDECI allows us to read numbers from an input record and put the numbers into registers.

```
o  XDECI 2,RECORD
```

XDECI starts at the specified address (RECORD in this case), and reads digits until it finds a space character. It then converts what it found to binary and stores in the specified register (R2 in this case). It also puts the address of the space it found into R1. Thus, to continue, we would do

```
o  XDECI 3,0(1)
```

Start the next XDECI zero bytes off of R1 (the address of the space it found), and store findings in R3 this time.

- **XDECO and XPRNT:** We use XDECO and XPRNT to create *print lines*. Consider the following example

```

0          L      3,NUM1
1          L      4,NUM2
2          XDECO  3,ONUM1
3          XDECO  4,ONUM2
4          AR     4,3
5          XDECO  4,OSUM1
6          XPRNT  DETAILED,133
7          *
8          LTORG
9  DETAIL  DC    C'0'           Carriage control char
10 ONUM1   DS    CL12          Character length 12 storage
11          DC    5C'  '         Repeat 5 spaces
12 ONUM2   DS    CL12
13          DC    5C'  '
14 OSUM    DS    CL12
15          DC    86C'  '        Fill the remaining of 133 byte
                           → print line with space

```

Note that print lines should always occupy 133 bytes.  $1 + 12 + 5 + 12 + 5 + 12 + 86 = 133$ , note that the first byte is the carriage control. A carriage control character zero is to double space the print lines, two blanks between each line

#### Carriage controls:

- **C' ':** Single space
- **C'0':** Double space
- **C'-' :** Triple space
- **C'1':** Top of next page

Let's first discuss the storage. The first byte specifies the carriage control. We define 12 character bytes for each output variable, with 5 space characters in between each output variable. After the 12 character bytes are allocated for OSUM1, and the 5 padding bytes after OSUM1, we have 86 remaining bytes to fill, so we use spaces.

Regarding the top instructions assume NUM1, NUM2 are defined, we load NUM1 and NUM2 into registers 3 and 4. Then, we XDECO R3 into ONUM1, R4 into ONUM2, and after AR modifies R4, we XDECO the new contents of 4 into OSUM1. What XDECO does is converts the contents to decimal, and puts them into the character buffer right justified.

XPRNT then starts at the carriage control (DETAILED), and prints a total of 133 bytes formatted.

We could also define headers and column names

```

0 XPRNT HEADER1,133 PRINT COLUMN HEADER 1
1 XPRNT COLHDR1,133 PRINT HYPHENS
2 XPRNT HYPHENS1,133 PRINT HYPHENS
3 XPRNT DETAIL,133 PRINT DETAIL LINE
4
5 HEADER1 DC C'1'
6 DC 57C' '
7 DC C'HERE IS MY REPORT'
8 DC 58C' '
9 *
10 COLHDR1 DC C'0' CARRIAGE CONTROL CHARACTER
11 DC C' NUM1' OUTPUT AREA FOR NUM1
12 DC 5C' ' SPACES
13 DC C' NUM2' OUTPUT AREA FOR NUM2
14 DC 5C' ' SPACES
15 DC C' SUM' OUTPUT AREA FOR THE SUM
16 DC 86C' ' SPACES
17 *
18 HYPHENS1 DC C' CARRIAGE CONTROL CHARACTER
19 DC C'-----' OUTPUT AREA FOR NUM1
20 DC 5C' ' SPACES
21 DC C'-----' OUTPUT AREA FOR NUM2
22 DC 5C' ' SPACES
23 DC C'-----' OUTPUT AREA FOR THE SUM
24 DC 86C' ' SPACES

```

**Note:** XDECO requires 12 bytes.

- **Even odd pairs:** An even odd pair of registers are two adjacent registers in which the number of the first register is even and the second is odd, we have the even odd pairs

0, 12, 34, 56, 78, 910, 1112, 1314, 15

Even odd pairs are used in some instructions in order to 64-bit "Register" in which the two registers are combined

- **Multiplication and division:** We have the instructions

```

0 MR R,R
1 M R,D(X,B)
2 DR R,R
3 D R,D(X,B)

```

Note that the first operand in all the above instructions are even odd pair registers. Thus, the left operand is an even register in 0 – 14. The second operand is either an fullword boundary address (M and D), or any register (MR, DR)

Let's first consider the MR instruction.

```

0 LA 3,4
1 LA 7,3
2 MR 2,7 Multiply 4 by 3

```

First to note that the even odd pair register R2 used as the first operand to the MR instruction takes R2 and R3, and combines them to make a 64-bit register in which to store the result of the multiplication. Thus, even though the target register is R2, the result will be completely contained within R3 unless the result was big enough to also need to use R2. If the result was not big enough to use R2, the contents of R2 are zeroed out.

Next, we consider division. Let's divide 24 by 6

```

o  LA  5,24
1  LA  9,6
2  M   4,=F'1'  Need to use M to extend the sign bit of R5
    ↳ into R4
3  DR  4,9

```

The remainder in this case will be stored in the even register, and the quotient in the odd register.

- **Note about multiplication and division:** In multiplication, the even register can be junk. But, with division, the even reg must have extension of the sign bit of the odd register. Thus, the even register must be filled with all zeros or all ones. This is why we multiply the even register by one before we divide.
- **Defining label that does not take up storage:**

```

o  LB    DS  0H

```

We define storage of zero halfwords... Ie it will not take up any storage and will not increase the LOC. Note that LB is simply the name of the label

you're not actually reserving any memory; you're simply associating the label LB with the current location counter value. This label is stored in the assembler's symbol table. The symbol table holds the address (which is the current location counter at the time of definition) for use during assembly and linking, but it does not correspond to any allocated memory in the final object code.

- **Compares and branching:** We have the two instructions
  - **RX: C (compare):**

```

o  C    R,D(X,B)

```

Compares the contents of R and the storage at address D(X,B)

Note that D(X,B) must be on a fullword boundary (FWB)

- **RR: CR (compare register)**

```

o  CR  R,R

```

Compares the contents of the specified registers

The condition code will be

- **Zero:** if  $a = b$
- **One** if  $a < b$
- **Two** if  $a > b$

Note that the condition code can be at most 3, but C and CR never set the CC to 3.

Suppose we want to compare two registers, and branch if the condition code is note zero (values were not equal), we would write

0	CR	2,3	Compare registers 2 and 3
1	BC	B'0111',NOTEQ	
2			
3	NOTEQ	DS	0H

Or, using BCR, we could write

0	LA	7,NOTEQ	
1	CR	2,3	Compare registers 2 and 3
2	BCR	B'0111',7	
3			
4	NOTEQ	DS	0H

- **Defining or decimaling character strings:** We have

0	NAME	DC	CLn'...'	String of length n
1	NAME	DC	C'...'	Single char
2	NAME	DC	C'...'	Char str, size is the size of → the specified string
3	NAME	DC	nC'...'	Repeat Single char n times

And of course we also have their DS counterparts

The encoding of these strings follows EBCDIC char encodings. The string *smith* is encoded to E2 D4 C9 E3 C8

- **Copying and comparing strings:** We do not use C or CR on character strings, instead we use the 6 byte SS (storage to storage) instructions

- **MVC (Move characters):** Copies the characters from source to destination

0	MVC	D(L,B),D(B)
1	MVC	DEST(len),SRC

Where the second operand is the source, and the first is the destination. Note that the provided length is used for **both** operands

**Note:** In this form it is assumed that the length of both operands is the same, more on the other forms later.

- **CLC (Compare logical characters):** Byte by byte lexicographical compare based on the EBCDIC HEX encodings, sets the condition code same as C and CR

o CLC	D(L,B), D(B)
1 CLC	LABEL1(length), LABEL2

- **Encoding of MVC and CLC:** The encoding of the forms described above is

$$h_0 h_0 h_L h_L \quad h_{B1} h_{D1} h_{D1} h_{D1} \quad h_{B2} h_{D2} h_{D2} h_{D2}$$

Where

- $h_0 h_0$  is the opcode
- $h_L h_L$  is the length minus one
- $h_{B1} h_{D1} h_{D1} h_{D1}$  Base and displacement of the first operand
- $h_{B2} h_{D2} h_{D2} h_{D2}$  Base and displacement of the second operand

- **Important EBCDIC encodings to remember:**

- **Space character:** 40

- **LTR (load and test register):** The LTR instruction loads into a register and sets the condition code based on its value. The CC will be

1. **Zero:** If the loaded value is zero
2. **One** If the loaded value is negative.
3. **Two:** If the loaded value is positive
4. **Three:** If there is overflow

Thus, we can test if a register has a value of zero with

o LTR R,R
-----------

Then, we can test the condition code accordingly.

- **If-endif statements:** We can write an "if-endif" statement by using branches

0	BC	B'bbbb', OVER
1		...
2		...
3		...
4	OVER	DS OH

This code branches if register 8 contains zero.

- **If-else-endif statements:** Consider

```
0      BC      B'bbbb',ELSE
1      ...
2      ...
3      ...
4      BC      B'1111',ENDIF
5  ELSE    DS      OH
6      ...
7      ...
8      ...
9  ENDIF   DS      OH
```

- **Extended mnemonics:** Shorten common instructions. Note that these are not real instructions, but translate to equivalent instructions. A few common ones are

```
0  BCR      B'1111',14
1  BR       14
```

Branch non-zero:

```
0  BC      B'0111',LABEL
1  BNZ     LABEL
```

Branch zero:

```
0  BC      B'1000',LABEL
1  BZ     LABEL
```

Branch:

```
0  BC      B'1111'LABEL
1  B      LABEL
```

- **For loop:** We can use BCTR or BCT...

- **Decrementing a register:** We use bctr

```
0  BCTR  R,0
```

Decrements R by 1

- **EQU:** EQU, or EQUates, assigns a value to a label

```
o    label      EQU      Expression
```

gives a label the value of the expression. Every occurrence of label will be treated as if it was the expression.

Equates are either typed above the CSECT or below the END

```
o    LOAD    EQU    L
```

Then, a load instruction can be written as

```
o    LOAD    3,NUM1
```

Before assembling the code, the Assembler replaces the label of the equates with the expression.

## 5.5 Decimal arithmetic

- **Intro:** We are now going to learn to do arithmetic in decimal instead of binary arithmetic using registers and fullwords in storage

To do this, we use a numeric format called packed decimal which is also referred to as "packed numbers" or simply "decimal" for short

Numbers are in storage in decimal so numbers in dumps are easier to read. We can use larger numbers - up to 31 decimal digits!

Also, both operands are in storage so we don't need to use registers. Storage to storage arithmetic means fewer instructions to do arithmetic, i.e., no loading values into registers, storing them, etc.

- **Two decimal formats:**

- **Zoned Decimal Format:** The number is represented in almost character format (EBCDIC), similar to what we get after an XDECO

Used for getting input and for formatting output

For example, 4-byte zoned decimal number: F1F2F8F9 = 1289<sub>10</sub>. One zoned decimal digit per byte. Within the byte, the left hex digit is the zone digit, and the right hex digit is the numeric digit.

The zone digit of the rightmost byte is used to represent the sign

- \* If the sign digit is A, C, E or F, the number is positive.
- \* If the sign digit is B or D, the number is negative

F3F8F4F3 is equivalent to 3843. A zoned decimal number looks a lot like the character, or EBCDIC, representation of the number. The difference is the sign zone digit in the first hex digit of the last byte.

**Note:** You cannot do arithmetic with zoned decimal numbers! Zoned Decimal Numbers

- **Packed Decimal Format:** the format used for doing arithmetic in decimal.

- **Declaring zoned decimal numbers**

```
o  label    dc    mZLn'p'
```

Where  $n$  is the length in bytes, and  $m$  is how many times to repeat (1 if left blank).

Declaration	Generates
NUM1 DC 3ZL3'123'	F1F2C3 F1F2C3 F1F2C3
NUM2 DC 3Z'123'	F1F2C3 F1F2C3 F1F2C3
NUM3 DC ZL3'-123'	F1F2D3
NUM4 DC ZL6'-123'	F0F0F0F1F2D3
NUM5 DC ZL3'1.23'	F1F2C3

- **Packed decimal format:** Each number translates to a digit in the packed number, with the sign represented by the hex digit at the end of the representation.

For example, *F1F2C3* in zoned format translates to *123C* in packed format. For the bytes leading up to the byte with the sign digit, we simply drop the zoned digit. Then, we reverse the order of the last byte (sign digit + number)

Packed decimal numbers take fewer bytes to store than zoned. Packed decimal numbers are actually stored as decimal numbers.

In a dump, packed decimal numbers appear as decimal numbers.

For example, the positive number 4657 would be 04657C. The sign digit of a packed decimal number is the last digit (highlighted in red above).

A sign digit of A, C, E or F indicates a positive packed decimal number

We often get numbers from an input data set or from some other source in a quasi-zoned decimal format like EBCDIC.

We can take the zoned decimal numbers as input and convert them to packed decimal numbers. Packed decimal allows us to do arithmetic. We can even do real number division in packed decimal!

**Note:** Leading zeros are fine, they stay.

- **Declaring packed decimal variables:**

o    `label DC mPLn'p'`

Declaration	Generates
NUM1 DC 3PL3'123'	00123C 00123C 00123C
NUM2 DC 2PL4'123'	0000123C 0000123C
NUM3 DC PL3'-1.23'	00123D (decimal not stored!)
NUM4 DC PL6'-123'	0000000123D
NUM5 DC PL2'12345'	345C (truncated on left!)

- **Default sign digits:**

- **Positive:** C
- **Negative:** D

- **Note about decimal points:** Decimal points are not stored, it is up to the developer to know where the decimal should be placed.
- **EBCDIC vs zoned decimal:** As long as we have a positive number, the EBCDIC representation is the same as the zoned decimal representation.
- **The case of packed decimals and extra leading zeros:** Consider the zoned decimal to packed decimal conversion

*F0F0F0F3F9F7F1F2* → 000039712F.

Notice the extra leading zero that appeared in the packed decimal. If not for this extra added zero, there would be an odd number (9) bytes. There is no such thing as an empty halfbyte.

- **Determine how many bytes of packed decimal storage you will need to declare to hold a converted zoned decimal number:** We take

$$\lfloor \text{len(zoned decimal bytes)} / 2 \rfloor + 1.$$

So, to store an 8-byte (8 digit) zoned decimal number in packed decimal, you would need a minimum of 5 bytes of packed decimal storage

It is a good general rule to declare packed decimal fields no larger than is absolutely necessary

- **Instructions involving packed decimals:**

- **Pack (F2) (SS) (Zoned to packed conversion):** A little like XDECI, It translates numbers in character format into a format which can be used for arithmetic

```
o  label PACK D1(L1,B1),D2(L2,B2)
1  label PACK label1(L1),label2(L2)
```

PACK converts zoned decimal numbers into packed decimal numbers. Consider the zoned decimal F1F2C3, It does the following

- \* The rightmost byte of the second operand is placed in the rightmost byte of the first operand, with zone and numeric digits reversed
- \* The zone digits are stripped away and the remaining numeric digits from the second operand are moved to the first operand, right to left

If the length of the first operand, the target packed decimal field, is not long enough, the number is truncated on the left.

If the length of the first operand, the target packed decimal field, is too long, the number is padded on the left with zeros.

The maximum length for a packed decimal field is 16 bytes. 16 bytes can hold a number with 31 digits!

PACK does not verify that the second operand holds a valid zoned decimal number.

PACK does not verify that a valid sign is converted into the packed decimal field.

PACK does not cause a Data Exception (S0C7).

If you code the length on the second operand incorrectly, the resulting packed decimal number will definitely NOT have the correct sign.

- **UNPK (F3) (SS) (Packed to zoned conversion):** A little like XDECO, It translates numbers in a format which can be used for arithmetic to numbers in character format

```
o  label UNPK D1(L1,B1),D2(L2,B2)
1  label UNPK label1(L1),label2(L2)
```

UNPK converts packed decimal numbers into zoned decimal numbers. Consider the number 123C. It does the following to this number, as an example

- \* The rightmost byte of the second operand is placed in the rightmost byte of the first operand, with the zone (sign) and numeric digits reversed
- \* Zone digit F is added to the remaining digits from right to left, each now taking a byte.
- **AP (FA) (SS) (Add)**: Add one packed decimal field to another

```
o  AP PFIELD1(5),PFIELD2(2)
```

adds the packed decimal number in PFIELD2 to the packed decimal number in PFIELD1. PFIELD2 is unchanged

```
o  label AP D1(L1,B1),D2(L2,B2) Explicit addr.
1  label AP PNUM1(L1),PNUM2(L2) Implicit addr.
```

L1 and L2 represent lengths coded on both operands, respectively, with 16 being the greatest

- **SP (FB) (SS) (Subtract)**: Subtract one packed decimal field from another

```
o  SP PFIELD1(5),PFIELD2(2)
```

subtracts the packed decimal number in PFIELD2 from the packed decimal number in PFIELD1. PFIELD2 is unchanged

```
o  label SP D1(L1,B1),D2(L2,B2) Explicit addr.
1  label SP PNUM1(L1),PNUM2(L2) Implicit addr
```

L1 and L2 represent lengths coded on both operands, respectively, with 16 being the greatest.

- **ZAP (F8) (SS) (Zero and add packed)**: Copy one packed decimal field to

```
o  ZAP PFIELD1(5),PFIELD2(2)
```

copies the packed decimal number in PFIELD2 to the packed decimal field named PFIELD1

In reality, PFIELD1 is zero'd out and PFIELD2 is added into PFIELD1. PFIELD2 is unchanged

Often used to copy a number into a larger field preparing for arithmetic.

```
0  label ZAP D1(L1,B1),D2(L2,B2) Explicit addr.  
1  label ZAP PNUM1(L1),PNUM2(L2) Implicit addr
```

L1 and L2 represent lengths coded on both operands, respectively, with 16 being the greatest.

- **MP (FC) (SS) (Multiply)**: multiply one packed decimal field by another.

```
0  label MP D1(L1,B1),D2(L2,B2) Explicit addr.  
1  label MP PNUM1(L1),PNUM2(L2) Implicit addr
```

L1 and L2 represent lengths coded on both operands, respectively, with 16 being the greatest.

Does not set the condition code

```
0  MP PNUM1(6),PNUM2(3)
```

Where

```
0  PNUM1 DC PL6'12233'      00 00 00 12 23 3C  
1  PNUM2 DC PL3'15'        00 01 5C
```

Becomes

```
0  PNUM1 DC PL6'12233'      00 00 01 83 49 5C  
1  PNUM2 DC PL3'15'        00 01 5C
```

- **DP (FD) (SS) (Divide)**: divide one packed decimal field by another

```
0  label DP D1(L1,B1),D2(L2,B2) Explicit addr.  
1  label DP PNUM1(L1),PNUM2(L2) Implicit addr.
```

Both quotient and remainder are stored in the first operand. Also does not set the condition code.

**Note:** Remainder will be in the last  $n$  bytes, where  $n$  is the size of the second operand (divisor).

```
0  DP PNUM1(5),PNUM2(2)
```

Where

```

0  PNUM1 DC PL5'27'    00 00 00 02 7C
1  PNUM2 DC PL2'5'      00 5C

```

Becomes

```

0  PNUM1 DC PL5'27'    00 00 5C 00 2C
1  PNUM2 DC PL2'5'      00 5C

```

Observe that the last two bytes (002C) is the remainder. Thus, 00005C is the quotient.

- **CP (F9) (SS) (Compare)**: compare one packed decimal field with another.

```

0  label CP D1(L1,B1),D2(L2,B2) Explicit addr.
1  label CP PNUM1(L1),PNUM2(L2) Implicit addr.

```

Always use CP when comparing one packed decimal field with another and not some other compare instruction.

Does a numeric comparison so the value of the field is important, not the lengths of the operands.

Does NOT change either operand.

```

0  CP PNUM1(5),PNUM2(2)

```

Where

```

0  PNUM1 DC PL5'27'    00 00 00 02 7C
1  PNUM2 DC PL2'5'      00 5C

```

condition code is set to two.

- **SRP (F0) (SS) (Shift and round)**: Shift a packed decimal by decimal digits left or right

```

0  label SRP D1(L,B1),D2(B2),i Explicit addr.
1  label SRP PNUM1(L),D2(B2),i Implicit addr.

```

Used to multiply and divide packed decimal numbers by factors of 10.

Often used to add extra decimal places to a number preparing it for division and then to get rid of one or more decimal places and round.

o SRP PNUM2(11),4,0

The first operand is the packed decimal field with the number being shifted, the second indicates a left shift by 4 digits and the third, the rounding factor, is set to 0 (rounding is unnecessary when shifting left).

Equivalent to multiplying PNUM2 for 11 bytes by  $10^4$ .

o SRP PNUM3(10),64-3,5

The first operand is the packed decimal field with the number being shifted, the second indicates a right shift by 3 digits and the third, the rounding factor, is set to 5. 5 is standard rounding. In other words, if the number being rounded off is 5, 6, 7, 8 or 9, round up by adding 1 to the digit to the left.

Shifting right by 3 digits with standard rounding is like dividing the packed decimal number by  $10^3$

**Note:** The mainframe knows it is a shift left if the second operand is between 1 and 31 and, conversely, knows it is a shift right if the second operand is between 32 and 63.

The second operand of the previous example could have been coded as 61 but 64-3 is far more easy to understand and is self-documenting

The sign of the number shifted is not changed unless the result becomes 0, in which case a negative sign, B or D, is made positive.

- **CVB (4F) (RX) (Convert to binary):** converts a packed decimal number in a doubleword of storage on a doubleword boundary to its binary equivalent and stores it in a register

o label CVB R1,D2(X2,B2) Explicit addr.  
1 label CVB R1,DWORD Implicit addr.

- **CVD (4E) (RX) (Convert to decimal):** converts a binary number in the first operand register to its packed decimal equivalent in a doubleword on a doubleword boundary

o label CVD R1,D2(X2,B2) Explicit addr.  
1 label CVD R1,DWORD Implicit addr

- **More SRP examples:** Example of a left shift by 3, equivalent to a multiply by  $10^3$

o SRP PNUM1(5),3,0

Where

```
o PNUM1 DC PL5'3227'      00 00 03 22 7C
```

Becomes

```
o PNUM1 DC PL5'3227'      00 32 27 00 0C
```

Example of a right shift by 2, equivalent to a divide by  $10^2$ , with standard rounding:

```
o SRP PNUM1(5),64-2,5
```

Where

```
o PNUM1 DC PL5'3277'      00 00 03 27 7C
```

Becomes

```
o PNUM1 DC PL5'3277' 00 00 00 03 3C
```

- **Recall an important fact in arithmetic:** When multiplying two decimals, you add together the number of digits after the decimal point in each number. For example, if one number has 2 decimal places and the other has 3, the product will have  $2 + 3 = 5$  decimal places (ignoring any trailing zeros that might be dropped).

Dividing decimals works a bit differently. There isn't a direct rule like "add" or "subtract" the decimal places. Instead, you often adjust the numbers by multiplying both the dividend and divisor by the same power of 10 to make the divisor a whole number, then perform the division. The number of decimal places in the quotient depends on the specific numbers and the precision you need (sometimes resulting in a terminating decimal, other times in a repeating or rounded decimal).

- **Real number divide example:** Now, let's do an example of a divide pack that provides us a real number result instead of just a quotient and remainder.

Note the following fields defined in storage and their initial values:

```
o PDEPAMT DC PL7'456929.87'      00 00 04 56 92 98 7C
  1 PSHRPRC DC PL3'12.35'          01 23 5C
  2 PCALCSHR DC PL10'0'           00 00 00 00 00 00 00 00 0C
```

We want to divide PDEPAMT by PSHRPRC and get a result with three decimal places just like we would get if using a calculator to do the divide

But, to do so, we will have to "fake" real number division using the DP instruction which gives us only integer division results.

So, as long as the number being divided, the dividend, has the same number of decimal places as the divisor, no pre-shifting necessary

To prepare for the division, we first need to ZAP the number being divided into a larger field.

How many bytes long should that larger field be? A simple way to determine that is to add the length of the divisor to the length of the number being divided. In this case it is  $7 + 3 = 10$

PCALCSHR is defined as a 10-byte packed decimal field initialized to 0.

Here are the instructions to accomplish the task

```

0  ZAP PCALCSHR(10),PDEPAMT(7) COPY TO LARGER FIELD
1  SRP PCALCSHR(10),3,0 ADD 3 FAKE DECIMAL PLACES
2  DP PCALCSHR(10),PSHRPRC(3) DIVIDE DEP BY PRC
3  SRP PCALCSHR(7),64-1,5 SHIFT AND ROUND QUOTIENT TO TWO PLACES

```

The result is rounded to two decimal places and is in the quotient part of PCALCSHR, i.e., the first 7 bytes. We can ignore the last three bytes of PCALCSHR, the remainder

- **Encoding of the SS instructions above:** We use the SS encoding format (there are three)

$$h_0 h_0 h_{L_1} h_{L_2} \quad h_{B_1} h_{D_1} h_{D_1} h_{D_1} \quad h_{B_2} h_{D_2} h_{D_2} h_{D_2}.$$

where

- $h_0 h_0$ : Specifies the opcode
- $h_{L_1} h_{L_2}$ : Specifies the length-1 of the arguments lengths
- $h_{B_1} h_{D_1} h_{D_1} h_{D_1}$ : Address of the first operand
- $h_{B_2} h_{D_2} h_{D_2} h_{D_2}$ : Address of the second operand

- **Decker's Rules for Packed Decimal Instructions:**

- Begin the label, or name, of ALL packed decimal fields with the letter P, for Packed
- Declare ALL packed decimal fields with a DC, a specific length in bytes and initialized to 0 (unless to some other value)
- NEVER let lengths default! Code a length on every packed operand where it CAN be coded!

- **Errors with packed decimals:**

- **Data Exception (S0C7):** At least one of the operands is not a valid packed decimal representation
- **Decimal-overflow Exception (S0CA):** The result is too large for the receiving field

- **Errors that occur with MP:**
  - **Specification Exception (S0C6):** - if length of second operand is greater than 8 or if length of the second operand, the multiplier, is greater than length of the first operand, the multiplicand
  - **Data Exception - (S0C7):** if the first  $n$  bytes of the first operand are not all zeros where  $n$  is the length of the second operand or if at least one of the operands is not a valid packed decimal number
- **Errors that occur with DP:**
  - **Specification Exception (S0C6):** if length of second operand is greater than 8 or if length of the second operand is greater than or equal to the length of the first operand.
  - **Decimal-divide Exception (S0CB):** if quotient will not fit in  $n$  bytes where  $n$  is the length of the first operand minus the length of the second operand.
  - **Data Exception (S0C7):** if at least one of the operands is not a valid packed decimal number
- **Errors that occur with SRP:**
  - **Decimal-overflow Exception (S0CA):** if a left shift results in losing nonzero digits.
- **Errors that occur with CVB:**
  - **Specification Exception - S0C6 :-** if the second operand is not on a double-word boundary.
  - **Data Exception - S0C7 :-** if the doubleword does not hold a valid packed decimal number.
  - **Fixed-point Divide Exception - S0C9 -** if the packed decimal number at the D(X,B) address is too large to be represented in 32 SIGNED bits in the register.
- **Errors that occur with CVD:**
  - **Specification Exception - S0C6 :-** if the second operand is not on a double-word boundary.
- **Packed decimal literals:** Literals can be used in the instructions, for example
  - o AP PNUM1(5),=PL2'35'

Don't let lengths default in literals either

- **Formatting Numeric Output:** There are two instructions that can be used to format and print packed decimal numbers

The formatting can - as desired - include a floating dollar sign, a floating positive sign, a floating negative sign, commas between every three digits and/or a decimal point.

Two instructions can be used to accomplish this editing:

- ED - Edit Instruction
- EDMK - Edit and Mark Instruction
- **ED (DE) (SS) (Edit):** converts a packed decimal number to its printable EBCDIC equivalent.

```
0  label ED D1(L1,B1),D2(B2) Explicit addr.  
1  label ED ONUM1(15),PNUM1 Implicit addr.
```

Takes the packed decimal number at D2(B2), converts it to EBCDIC and places it at D1(L1,B1) according to a pre-placed edit pattern.

Note that we do not code a length for the second operand.

It can edit the number inserting special characters as desired.

A hexadecimal edit pattern must be moved into the output field prior to executing the ED instruction.

The edit pattern can insert commas and/or a decimal point as desired.

The edit pattern can also supply a character with which we can suppress leading zeros in the number displayed in the print line.

```
0  MVC ODEPAMT(10),=X'4020206B2021204B2020'  
1  ED ODEPAMT(10),PDEPAMT
```

Consider the storage (print line)

```
0  PDEPAMT DC PL4'9435.75' 09 43 57 5C  
1  DETAIL DC C'0' DOUBLE SPACING  
2  ODEPAMT DS CL10 OUTPUT SPACE FOR PDEPAMT  
3  DC 122C' ' SPACES
```

Remember that each byte consists of two hexadecimal digits, if looking at storage in hex, that is.

This edit pattern will fill 10 bytes, or columns, of the print line.

The first character in the edit pattern is almost always a fill character.

In this example, the fill character is X'40', a space in EBCDIC.

X'20' and X'21' are digit selectors.

X'21' is a digit selector but it is different from X'20'.

X'21' sets significance on in the byte, or column, following it.

X'6B' represents a comma in EBCDIC.

X'4B' represents a decimal point in EBCDIC

The result in EBCDIC in print line:  $\bar{b}9,435.75$  ( $\bar{b}$  = a space)

If significance is off, the character to the right of the 20 will be suppressed if there are no nonzero digits to the left. This is not the case for significance on (21)

- **EDMK (DF) (SS):** Edit and Mark (EDMK) is used exactly the same as Edit (ED). The only difference is that EDMK places the address of the first non-zero digit - from left to right in the output field - in register 1

If EDMK reaches a significance on digit selector (X'21') before reaching a non-zero digit, the address of the byte following the X'21' is placed into register 1.

The address in register 1 can then be used to place a dollar sign, positive sign, negative sign or some other character to the immediate left of the first non-zero digit or where significance is turned on.

```
0  MVC ODEPAMT(10),=X'4020206B2021204B2020'
1  EDMK ODEPAMT(10),PDEPAMT
2  BCTR 1,0 DECREMENT REGISTER 1 BY 1
3  MVI 0(1),C'$' PLACE THE DOLLAR SIGN
```

```
0  PDEPAMT DC PL4'9435.75' 09 43 57 5C
1  PRTLINE DC C'0' DOUBLE SPACING
2  ODEPAMT DS CL10 OUTPUT SPACE FOR PDEPAMT
3  DC 122C' ' SPACES
```

ODEPAMT after the above four instructions and XPRNT:  $\bar{b}\$9,435.75$  ( $\bar{b}$  = a space)

Note that, if PDEPAMT is 0, register 1 is not changed.

Therefore, we need to add a level of insurance to print \$0.00 if PDEPAMT is 0 by presetting register 1 to point to the byte immediately to the left of the decimal point byte:

```
0  LA 1,ODEPAMT+6
1  MVC ODEPAMT(10),=X'4020206B2021204B2020'
2  EDMK ODEPAMT(10),PDEPAMT
3  BCTR 1,0 DECREMENT REG 1 BY 1
4  MVI 0(1),C'$' PLACE THE DOLLAR SIGN
```

- **Notes about ED and EDMK:** Digits are moved from the source field, one at a time, and from left to right.

If not enough digit selectors, the result will be truncated on the right

A Data Exception - S0C7 - can occur if the source field is not a valid packed decimal number.

- **Packed instruction condition codes**

- **AP, SP, ZAP:**

- \* 0 = the result is zero
    - \* 1 = the result is negative
    - \* 2 = the result is positive
    - \* 3 = overflow

**Note:** Note that overflow does not automatically occur if the first operand is shorter than the second, only when the result of the arithmetic operation does not fit into the first operand

- **CP:**

- \* 0 = the two values compared are equal
    - \* 1 = the first operand's value is less than the second operand's
    - \* 2 = the first operand's value is greater than the second operand's
    - \* 3 - not used

- **SRP:**

- \* 0 = result is zero
    - \* 1 = result is negative
    - \* 2 = result is positive
    - \* 3 = overflow has occurred

- **ED, EDMK:**

- \* 0 = source field is zero
    - \* 1 = source field is negative
    - \* 2 = source field is positive
    - \* 3 = unused

## 5.6 Internal Subroutines

- STM (90) (RS) (Store Multiple) Instruction

```
o    label STM R1,R2,D(B)
```

Stores all registers from R1 through R2 to contiguous fullwords in that order in storage starting at D(B).

If R2 is less than R1 , then R1 through register 15 and register 0 through R2 are stored in that order in storage starting at D(B).

- LM (98) (RS) (Load Multiple) Instruction:

```
o    label LM R1,R2,D(B)
```

Loads all registers from R1 through R2 from contiguous fullwords in that order from storage starting at D(B).

If R2 is less than R1 , then R1 through register 15 and register 0 through R2 are loaded in that order from storage starting at D(B)

- **RS instructions and encodings:** RS instructions are registers to storage, they have the form

```
o    NAME    R1,R2,D(B)
```

With encoding

$$h_0 h_0 h_{r_1} h_{r_2} h_B h_D h_D h_D.$$

- **Why subroutines?:** subdividing our programs into functions makes a complicated program easier to understand, enhance and debug. They reduce the amount of inline repeated code; we can put that repeated code in a subroutine and simply call and execute that subroutine when the code is needed again.
- **Internal vs external subroutines:** An internal subroutine in Assembler is one that is located within a single control section, or CSECT.

### External subroutines

- It is located outside of the caller's CSECT.
- It has its own CSECT.
- Is actually a subprogram (although it's often referred to as a subroutine).
- External subprograms can be written in C++, Java, etc., on the mainframe
- **Notes about internal subroutines:** There are specific standards that Assembler programmers worldwide must follow to call subroutines and subprograms.

- **Internal subroutine standards:**
  - On entrance to a subroutine, register 1 holds the address of a parameter list (if parameters need to be passed in)
  - On entrance to a subroutine, the initial value in any register that will be altered by the subroutine should be saved using ST and/or STM as necessary
  - Before exiting a subroutine, the initial value in any register that was altered by the subroutine should be restored using L and/or LM as necessary
- **Defining an Internal Subroutine:** The following puts a label in storage without moving the location counter or actually declaring storage:

```
o  rtnName DS 0H
```

Note that the name of the subroutine can also be placed on the subroutine's first instruction.

```
o  rtnName STM 3,7,SUBSAVE
```

- **Parameter list:** Standards dictate that a parameter list must be declared in a very specific manner in Assembler.

Parameters are only passed by reference in Assembler and NEVER by value.

A parameter list is a set of contiguous fullwords, each containing the address of a parameter, or variable, to be passed.

To declare a parameter list that allows the addresses in the fullwords in the parameter list to be virtual, we use Address Constants, or ADCONs, defined as

```
o  label DC A(expression)
```

If **expression** is a non-negative integer, the generated fullword will contain the binary representation of that integer, which is the same as declaring

```
o  label DC F'expression'
```

If expression is a *label* or *label + n*, the generated fullword will contain the address of *label* or *label + n*.

```
o  PARM DC A(5)
```

declares a fullword at label PARM in storage as:

00000005

o PARMLIST DC A(FIELD1)

declares a fullword at label PARMLIST in storage as: 00000148, if FIELD1 is declared in storage and is at location counter value 000148 after assembly

o 000148 FIELD1 DC F'34220'

o PARMLIST DC A(45,FIELD2)

declares two fullwords at label PARMLIST in storage as:

0000002D00000274

if the following is declared in storage and is at location counter value 000274 after assembly

o 000274 FIELD2 DC F'2301.00'

Another example declaration:

o PARMLIST DC A(FIELD3)  
1 DC A(34)

Standards dictate that, if the internal subroutine needs parameters passed into it, set up the parameter list similarly to what is shown immediately above

Then load the address of PARM or PARMLIST into register 1 before calling the subroutine.

By the way, it's the same for passing parameters to external subprograms!

- **BAL (45) (RX) (Branch and Link) Instruction:**

o label BAL R1,D2(X2,B2)

Would be more appropriately named the Link and Branch Instruction.

– **Link Part of the Instruction:** Puts the last three bytes of the PSW into register R1 and zeros out the first byte of R1 . Why?

Remember that the last three bytes of the PSW hold the address of the next instruction, i.e., where to return to after the subroutine.

- **Branch Part of the Instruction:** After the link part described immediately above, the BAL instruction then takes an unconditional branch to the D(X,B) address, i.e., the address of the subroutine.
- **Passing control to subroutine:** Here is the sequence of instructions calling an internal subroutine using a parameter lis

```
o  LA 1,PARMLIST POINT R1 AT PARMLIST  
1  BAL 11,SUBRTN BRANCH AND LINK TO SUBRTN
```

Then, when the subroutine is finished, branch back to the instruction in the caller immediately following the call, or BAL:

```
o  BR 11 RETURN TO CALLER
```

- **Using parameters:** In the subroutine, after we save the caller's registers, we dereference the parms

```
o  STM  2,4,SAVEREGS  STORE REGS TO BE USED  
1  LM   2,4,0(1)
```

- **Returning control to caller:** When the subroutine completes its task and is ready to return to the caller, it must:

- Restore the caller's registers using either a Load (L) if only a single register needs to be restored or Load Multiple (LM) if more than one
- Return to the caller using: BR 11

## 5.7 Standard linkage and external subroutines

- **External subprograms:** Subprograms are similar to internal subroutines but they are outside, or external, to our program.

In Assembler, they have their own CSECT

An external subprogram is a type of subroutine but

- It is located outside of the caller's CSECT.
- It has its own CSECT.
- Is actually a subprogram (although it's often mistakenly referred to as a subroutine).
- External subprograms can be written in COBOL, C, C++, Metal C, Java, etc., on the mainframe

Just like defining internal subroutines, there are standards and conventions that must be followed.

In ASSIST, subprograms are not really "external". Simulated by adding a new CSECT below the storage of the CSECT above

Must have full standard entry and exit linkage. One END statement refers to first CSECT at the top of the file

Standards dictate that a parameter list must be declared in a very specific manner in Assembler. Parameters are only passed by reference in Assembler and NEVER by value. A parameter list is a set of contiguous fullwords, each containing the address of a parameter, or variable, to be passed.

Here is the sequence of instructions calling an external subprogram using a parameter list:

```
0  LA  1,PARMLIST    POINT R1 AT PARMLIST
1  L   15,=V(SUBPGM) LOAD 15 WITH ADDR OF SUBPGM
2  BALR 14,15        BRANCH AND LINK TO SUBRTN
```

Then, when the subprogram is finished, branch back to the instruction in the caller immediately following the call, or BALR:

```
0  BR 14 RETURN TO CALLER
```

When the subprogram is completes its task and is ready to return to the caller, it must:

1. The standard exit linkage restores the caller's registers
2. Return to the caller using: BR 14

- **DROP Statement:**

```
o  DROP R
```

Or

```
o  DROP R1,R2,...,RN
```

It ends the "domain" of a USING statement. The DROP informs the Assembler that register R or registers R1,R2,...,Rn are no longer to be associated with label.

Or, that the specified register is no longer (for instructions below) supposed to be used to convert implicit addresses to explicit addresses for encoding instructions.

- **Dummy SECTions, or DSECTs:** A dummy section is used to specify a format that can be associated with a particular area in storage without producing any object code.

The end of a dummy section is signaled by the occurrence of a CSECT statement, another DSECT statement or an END statement

An example DSECT definition:

```
o  $TABELEM DSECT
  1  $STCKNUM DS      F
  2  $ARTIST   DS     CL24
  3  $TITLE    DS     CL24
  4  $INSTOCK  DS      F
  5  $PRICE    DS      F
```

specifies the format of a table element. The labels \$STCKNUM, \$ARTIST, etc., can be used rather than displacements into the element itself.

Note the convention to use the \$ to denote the name of a DSECT and its fields

Before a DSECT can be used, a USING statement must be coded

```
o  USING $TABELEM,3
```

- **Standard Entry and Exit Linkage Conventions:**

- Conventions about how to call a subprogram and return from it were standardized many years ago by a group of Assembler developers. What follows is a description of those conventions.
- Standards dictate that, when control is passed to an external subprogram, register 15 contains the address of the subprogram.

- Standards dictate that register 14 contains the address of the next instruction to execute in the caller program, i.e., the one following the call to the subprogram, i.e., the instruction.
- Standards dictate that register 13 contains the address of an 18-fullword save area in the caller's storage in which its own registers will be saved by a called subprogram(!).
- Just as was presented in the previous chapter when calling internal subroutines, parameters are passed to an external subprogram in exactly the same manner. Standards dictate that register 1 contains the address of the beginning fullword of the parameter list if parameters are passed.
- Return codes are passed back to the caller in register 15.
- A simple calculated value can be passed back to the caller in register 0 (rare, and should be avoided).
- As hinted above while talking about register 13, the subprogram is responsible for storing the contents of the caller's registers upon entry to the routine and restoring those values before returning control to the caller.

- **Format of 18F save area**

Unused
Backward Pointer
Forward pointer
R14
R15
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12

Notice that we do not include R13

- **Standard entry linkage:** The following code should be included as the first lines of each CSECT

```

0  pgmName    CSECT
1          STM   14,12,12(13)
2          LR    12,15
3          USING pgmName,12
4          LA    14,name_of_18F_save_area_in_pgmName
5          ST    13,4(,14)
6          ST    14,8(,13)
7          LR    13,14

```

```
o STM 14,12,12(13)
```

Saves all of the caller's registers, except for register 13, in the caller's 18-fullword register save area.

```
o LR 12,15  
1 Using pgmName,12
```

Puts the address of pgmName in R12, then establishes R12 as the base register for pgmName

```
o LA 14,name_of_18F_save_area_in_pgmName
```

points register 14 to an 18-fullword register save area in the current program, pgmName's, storage where its own registers will be saved if it calls a subprogram itself.

```
o ST 13,4(,14)
```

stores the address of the caller's 18-fullword register save area in the current program, pgmName's, own 18- fullword register save area. This value in register 13 is known as the **backward pointer**.

```
o ST 14,8(,13)
```

stores the address of the current program, pgmName's, 18-fullword register save area in the caller's 18- fullword register save area. This value in register 14 is known as the **forward pointer**.

```
o LR 13,14
```

now points register 13 to the current program, pgmName's, 18-fullword register save area in case it calls a subprogram itself.

- **Standard Exit Linkage:** The following code should be included as the last lines of executable code in each CSECT if no return code is being passed back in register 15 (and no calculated value is being returned in register 0):

```
o L 13,4(,13)  
1 LM 14,12,12(13)  
2 BR 14
```

```
o L 13,4(,13)
```

## 5.8 Tables

- **Tables:** tables in assembler are like arrays in higher-level languages. Like arrays, Assembler tables are defined in storage - and given a name - to store related data items.

Examples of "related" data items or data items of "similar character"

- the ages of each individual student in class
- the account balances of checking accounts of bank customers
- the daily average temperature in New York City for the calendar year
- the names of the students in an Assembler class

In an array, we call the storage for an individual data item an element. In Assembler, this storage is called an entry. In arrays, we can only store one data item per element and, unless an array of objects, all data items must be of the same primitive data type. In Assembler tables, we have complete freedom to store any combination of data items and types in a single entry of a table.

In Assembler, we can define a table in storage using any storage class. We only need to be sure that the table is big enough in bytes to hold all of the data we need it to hold.

Here is an example of an Assembler table definition that can hold integer test scores for up to 50 students

```
o SCORES DC 50F'0'
```

Each of the 50 fullwords can hold a single student's test score

We could store the grades and then process the table like an array to calculate an average and/or find the minimum and maximum scores.

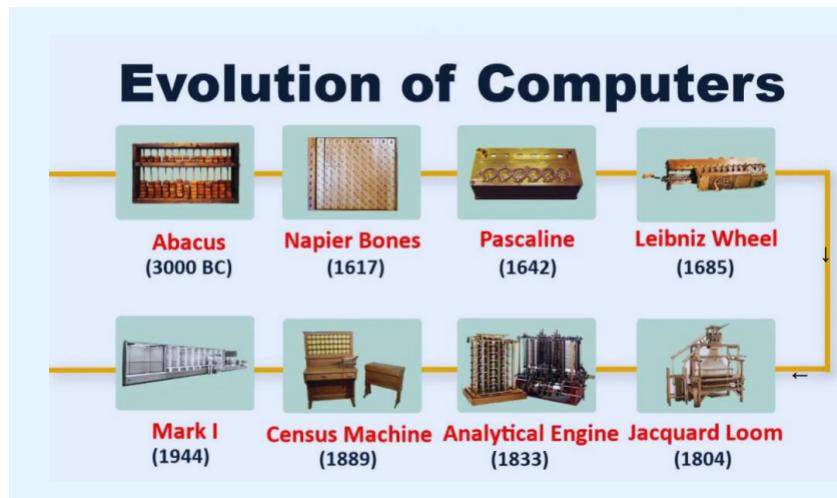
With the table declared on the previous slide, the following slide displays the code for a read loop that reads records from a file with up to 50 records

```
0      LA 2,SCORES R2 -> SCORES TABLE (OUR TABLE POINTER)
1      SR 3,3 R3 = 0 (OUR COUNT OF FILLED TABLE ENTRIES)
2      *
3      XREAD RECORD,80 READ FIRST RECORD
4      *
5      LOOP1 BNZ ENDLOOP1 BRANCH TO ENDLOOP1 IF NO MORE RECORDS
6      *
7      LA 3,1(,3) ADD 1 TO FILLED ENTRY COUNTER
8      XDECI 4,RECORD GET SCORE FROM RECORD
9      ST 4,0(2) STORE SCORE INTO TABLE ENTRY
10     *
11     LA 2,4(,2) R2 -> NEXT FULLWORD TABLE ENTRY
12     *
13     XREAD RECORD,80 READ NEXT RECORD
14     B LOOP1 BRANCH BACK TO TOP OF LOOP1
15     *
16     ENDLOOP1 DS 0H
```

# Computer Architecture and Syst Org

## 6.1 Chapter 1: History of computers

- Evolution of computers:



Although computers have become faster and more reliable, the same principal components have been present since the beginning of the stored program computer.

- **Stored program computer:** A stored program computer stores the program so that you can change the program without changing the hardware.
- **Mechanical calculators:** The opposite of a stored program computer is a device like a calculator where you have to input each step one at a time.

Mechanical calculators have existed since the 1600's. Development limited by ability to machine the parts.

The progress of mechanical calculators (in the 1600s and later) depended on how precisely people could manufacture and shape the physical components (gears, levers, cams, wheels, screws, etc.) using the tools and machining techniques of the time.

Early lathes, milling machines, and other metalworking tools weren't advanced enough to create very small, precise, and reliable parts. Without that precision, the calculators would jam, wear down, or give incorrect results.

Handheld electronic calculators were invented in the 1970's, overlapping stored program technology

- **The Pascaline (first commercially produced mechanical calculator):** The Pascaline (1640s) was a mechanical calculator invented by Blaise Pascal to help his father with tax work.

It could perform addition and subtraction directly, and multiplication/division by repeated operations.

It used a series of interlocking gears and wheels. Each wheel represented a digit (0–9). Turning one wheel past 9 automatically carried over to the next wheel (like an odometer).

It was accurate but expensive, fragile, and limited to basic arithmetic.

the Pascaline was the first commercially produced mechanical calculator, using gear-driven wheels to automate carrying in arithmetic.

- **Jacquard loom:** The Jacquard loom (1804, Joseph-Marie Jacquard) was a mechanical loom that revolutionized weaving:

It used punched cards to control the raising and lowering of warp threads.

This let weavers produce complex, repeatable textile patterns automatically, rather than moving threads by hand.

It's considered an early form of programmable machine, directly inspiring later computing ideas (like punch-card programming in Babbage's engines and early computers).

**Note:** Possibly the first stored program computer. It is Still used today in less industrialized parts of the world

- **Four generations:** The evolution of the stored program computer is usually classified into four generations according to the underlying technology.
  1. **First Generation: 1940s-1950s:** Vacuum tubes
  2. **Second Generation: 1950s-1960s:** Transistors
  3. **Third Generation: 1960s-1980s:** Integrated circuits
  4. **Fourth Generation: 1980s-present:** Microprocessors
- **Generation zero (Card tabulator) (Punched card tabulating machines):** The punched card tabulating machine (1880s–1890s, invented by Herman Hollerith) was an early data-processing device:

Built for the 1890 U.S. Census, it sped up counting and sorting huge amounts of population data.

Information was encoded as holes in cards (each position represented a data field). The machine had electrical contacts—when a pin passed through a hole, it completed a circuit, advancing a counter or sorter.

It cut census processing from nearly a decade (1880) to just a couple of years.

Hollerith's company later became part of IBM, making this a direct ancestor of modern computing.

The cards used EBCDIC format which is still used in IBM machines today. The cards had 80 columns, each row represents an 80 character input line.

- **Vacuum tubes:** Vacuum tubes (early 1900s) are electronic devices used to control the flow of electricity in a vacuum-sealed glass tube.

A typical tube has a cathode (heated filament that releases electrons), an anode (plate that collects electrons), and often one or more grids to control the flow.

Functions:

- Amplification (make weak signals stronger)
- Switching (on/off, like early transistors)
- Rectification (convert AC to DC)

They formed the basis of the first generation of computers (1940s–1950s), but they were large, power-hungry, produced heat, and failed often.

They were eventually replaced by transistors in the late 1950s, which were smaller, faster, and more reliable.

- **John Atanasoff and Clifford Berry of Iowa State University:** Created the Atanasoff Berry Computer (1937 - 1938), which solved systems of linear equations.

It was the first totally electronic machine, destroyed during department cleanup

- **John Mauchly and J. Presper Eckert (ENIAC):** Created the Electronic Numerical Integrator and Computer (ENIAC) at the University of Pennsylvania, 1946

Lawsuits over whether Atanasoff or Mauchly and Eckert should get credit for inventing the computer

The ENIAC wasn't the first machine ever, but it was the first large-scale electronic digital machine officially named a "computer." Before ENIAC, the term "computer" referred to people who did calculations by hand.

The ENIAC was the first general-purpose computer, it had 17000 vacuum tubes which made up two large rooms (one for air-conditioning). It weighed roughly 30 tons. A few vacuum tubes burnt out every day

It has approximately 125 bytes of memory

- **IBM 650:** The IBM 650 was the first mass-produced computer (1955). It had a magnetic drum, not a disk. It used decimal, not binary. Phased out in 1969.
- **The Transistor:** A transistor (invented in 1947 at Bell Labs) is a tiny semiconductor device that can amplify or switch electronic signals, just like a vacuum tube but without the drawbacks.

A transistor functions as an electronic switch, controlling the flow of electrical current through a circuit or amplifying a weak signal into a stronger one.

- **Switching:** Transistors can be turned "on" or "off" by their input signal, creating a digital "1" or "0". This on-off behavior is essential for logic gates, which perform calculations in computer processors
- **Amplification:** A small input signal can control a larger output signal, increasing its power or amplitude. This is crucial in applications like radio receivers and audio systems, where weak signals need to be strengthened for detection or output.

### Why transistors replaced vacuum tubes:

- **Size:** Transistors are extremely small compared to bulky glass tubes.
- **Power & Heat:** They consume far less power and generate much less heat.
- **Reliability:** Transistors are solid-state (no fragile filaments or vacuum glass), so they rarely burn out.
- **Speed:** They can switch on and off much faster, enabling quicker computations.
- **Cost:** Mass production on silicon chips made them cheaper over time.

**Note:** Solid-state means that a device's components are made entirely from solid materials (like semiconductors) and do not rely on moving parts or vacuum tubes to function.

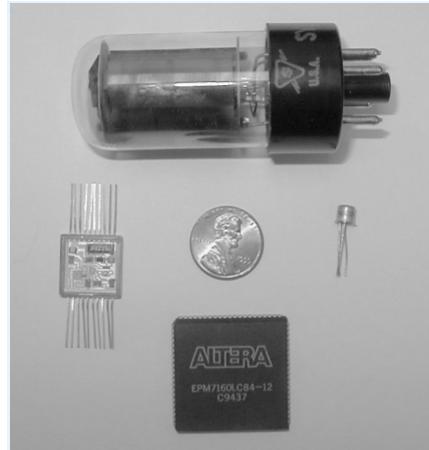
- **Second generation (Transistorized computers) ((1954-1965)):** These computers used transistors, which are far more reliable than vacuum tubes

Examples include the IBM 7094 (scientific) and 1401 (business), and the Digital Equipment Corporation (DEC) PDP-1

Each of these computers had a different architecture, and less than 1MB of memory.

- **Size comparison:** Relative sizes (clockwise from top)

- Vacuum tube
- Transistor
- Chip with 2000 NAND gates
- Integrated circuit package



- **Integrated circuit:** An integrated circuit (IC) is a tiny chip of semiconductor (usually silicon) that contains many electronic components-transistors, resistors, capacitors, and wiring-all fabricated together on a single piece of material.

### **Why ICs were an improvement (over vacuum tubes and even discrete transistors):**

- **Miniaturization:** Instead of wiring thousands of individual transistors by hand, many could be built directly onto one chip → much smaller devices.
- **Reliability:** Fewer soldered connections meant fewer points of failure compared to vacuum tubes or discrete transistor circuits.
- **Speed:** Signals travel much faster across a tiny chip than between separate components on a circuit board.
- **Power efficiency:** ICs consume very little power compared to vacuum tubes (which required heating filaments).
- **Cost:** Mass production made ICs cheap to manufacture, lowering the cost of electronics.
- **The Third Generation: Integrated Circuit Computers (1965-1980):** An integrated circuit computer is a computer built using integrated circuits (ICs) instead of vacuum tubes or discrete transistors.

Thousands of transistors, resistors, and capacitors were fabricated on small silicon chips, then connected together to form CPUs and memory systems.

Examples of third generation computers include the IBM 360/370, DEC PDP-8 and PDP-11, and the Cray-1 supercomputer

Beginning of the modern era, System software (operating systems and compilers), and Application software

- **Why IBM succeeded:** By using the same architecture, IBM provided an upgrade path

Same architecture but not necessarily same hardware design

Scientific and business machines. Scientists need floating point. Business people don't want roundoff error

IBM was the Microsoft of its day, but it no longer is

- **Fourth generation VLSI computers:** Very large scale integrated circuits (VLSI) have more than 10,000 components per chip

Enabled the creation of microprocessors.

8080, 8086, and 8088 chips used in first personal computers in early 80's

Earliest PCs had no hard disk they had a 360KB per floppy disk (later 1.4MB) which ran only one application at a time

- **Backwards compatibility:** Backwards compatibility is the ability of a new machine to run software intended for the previous generation

Essential to modern software industry because users want to spend their money on their core business, not on rewriting or converting software

Current PC architecture became near-universal not because it was a good one but because it was created at the right time

Newer machines needed backward compatibility, so we are stuck with that architecture (and its successors) forever

IBM has also provided backwards compatibility for its mainframe line

- **Moore's Law:** Gordon Moore (founder of Intel) in 1965: The density of transistors in an integrated circuit will double every year.

The more recent version: Density of silicon chips doubles every 18 months.

This is just an approximation, not a real law like that law of gravity.

This can't last forever, eventually you'd have transistors smaller than an atom!

- **Summary:**

First Generation (1940s-1950s)

- **Technology:** Vacuum tubes
- Characteristics
  - \* Large, slow, and expensive
  - \* High power consumption
- **Example:** ENIAC

Second Generation (1950s-1960s)

- **Technology:** Transistors
- Characteristics
  - \* Smaller, faster, and more reliable than vacuum tubes
  - \* Reduced power usage
- **Example:** IBM 1401

Third Generation (1960s-1980s)

- **Technology:** Integrated Circuits (ICs).
- Characteristics:
  - \* Increased speed and efficiency.
  - \* Cost-effective and compact design.
- **Example:** IBM System/360

Fourth Generation (1980s-present)

- **Technology:** Microprocessors
- Characteristics
  - \* Entire processing unit on a single chip
  - \* Further miniaturization and mass production
- **Example:** Personal Computers (PCs)

## 6.2 Chapter 3: Boolean algebra, circuits, registers, and memory

### 6.2.1 Boolean algebra

- **Boolean expressions:** Boolean expressions are built by using Boolean operators to combine Boolean variables

Most people consider the fundamental Boolean operators to be AND, OR, and NOT.

- **Not:** Unary operator, we can write NOT  $A$
- **And, Or:** Binary operators
- **And:** AND plays the role of "multiply" in Boolean algebra, so it can be written using any symbol for "times". It is called the Boolean product.

$$AB = A \text{ AND } B.$$

Can also be written with \*,  $\wedge$ ,  $\times$

- **Or:** OR plays the role of "add" in Boolean algebra, so it can be written using any symbol for "plus". It is called the Boolean sum.

$$A + B = A \text{ OR } B.$$

Can also be written with  $\vee$

- **NOT:** NOT plays the role of negation, so it can be written with any symbol for negation

Can be written with a minus sign, overbar, prime mark, elbow ( $\neg$ ) or the exclamation mark (!)

- **Order of operations:**
  - Parentheses have top priority.
  - NOT has the next priority.
  - AND is next.
  - NAND, NOR is lower than and but higher than xor and or
  - XOR is lower than AND,NAND, and NOR but higher than OR
  - OR has the lowest priority
- **AND binds tighter than OR:** This is another way of saying that AND has priority over OR
- **Boolean function:** A Boolean function is a function of zero or more Boolean variables.

The output of a Boolean function is also a Boolean value, i.e., 0 or 1.

- **Canonical form:** There are numerous ways of stating the same Boolean expression.
  - These synonymous forms are logically equivalent.
  - Logically equivalent expressions have identical truth tables.

To make it easier to compare Boolean expressions, we use standardized forms called canonical forms.

The most useful canonical form for Boolean expressions is the sum-of-products form

In the sum-of-products form, terms are connected by OR.

The terms used to make up sum-of-products form are called minterms

For a function of  $n$  variables, each minterm contains exactly  $n$  items AND'ed together..

For example, the minterms with 2 variables are

$$(-x)(-y), (-x)(y), (x)(-y), (x)(y).$$

Some or all of these terms are OR'd together to create the sum-of-products form for any function of two variables.

To make the result unique, we need a way to ensure that the terms always come in the same order.

In this class we will use the following rule:

1. Alphabetize the variables in each term.
2. Alphabetize the terms, considering the negative form of a variable to precede the positive one, i.e.,  $-x$  comes before  $x$ .

So whichever of the four minterms

$$(-x)(-y), (-x)(y), (x)(-y), (x)(y).$$

are used, they will always be listed in that order.

To convert a function to sum-of products form, we start with its truth table

Every row that results in a 1 contains a combination of values of variables that makes the function true.

So we AND together that set of values.

The function is true if any one of those sets of values occurs, so we OR the sets together

Consider  $F(x, y, z) = x\bar{z} + y$ , where  $\bar{z}$  denotes the negation of  $z$

The truth table is

$x$	$y$	$z$	$x\bar{z} + y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

This function has 5 rows that return true, so our sum-of-products representation will have 5 terms

Each term will show the values of  $x$ ,  $y$  and  $z$  that are needed to make that term true. Remember that if  $x$  is false (0), then  $\bar{x}$  is true.

Thus,

$$F(x, y, z) = \bar{x}y\bar{z} + \bar{x}yz + x\bar{y}\bar{z} + xy\bar{z} + xyz.$$

You can check that you have correctly generated the sum-of-products form by building a truth table for the sum-of-products form. The result column should match the result column of the original function.

### 6.2.2 Boolean identities

- **Intro:** The simpler that we can make a Boolean function, the smaller the circuit that will result.

Simpler circuits are cheaper to build, consume less power, and run faster than complex circuits.

However, there are other criteria involved

1. How the circuit involved fits with other circuits in the machine, e.g., can common activities be factored out.
2. General criteria such as providing redundancy, a safety margin of error, and others.
3. Consistency with other machines in the same line.

Still, we often want to reduce our Boolean functions to a simpler form

We can use Boolean identities (equations) for this.

In a real-world situation, this would be accomplished with the help of a hardware compiler that would also enforce standards.

- **Boolean identities in one variable** Most Boolean identities have an AND (product) form as well as an OR (sum) form.

The first group contains identities involving only one variable. They enable you to reduce the number of terms in an expression, and sometimes also the number of variables.

Identity Name	AND form	OR form
Identity Law	$1x = x$	$0 + x = x$
Null Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$x\bar{x} = 0$	$x + \bar{x} = 1$

The AND form and the OR form are duals. The dual of a law is created as follows:

1. Switch AND and OR.
2. Switch 0 and 1

If a Boolean statement is valid (its truth table contains all TRUEs), its dual is valid also. This is different from ordinary arithmetic, it is a characteristic feature of Boolean algebra

Note that laws are good for variables and expressions. The laws are expressed in terms of variables like  $X$ ,  $Y$  and  $Z$ .

You can substitute any Boolean expression in place of a variable.

- **Identities:**

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null (or Dominance) Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$x\bar{x} = 0$	$x + \bar{x} = 1$
Commutative Law	$xy = yx$	$x + y = y + x$
Associative Law	$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$
Distributive Law	$x + yz = (x + y)(x + z)$	$x(y + z) = xy + xz$
Absorption Law	$x(x + y) = x$	$x + xy = x$
DeMorgan's Law	$\overline{(xy)} = \bar{x} + \bar{y}$	$\overline{(x + y)} = \bar{x}\bar{y}$
Double Complement Law	$\bar{\bar{x}} = x$	

- **Example:** Simplify the following function

$$F(x, y, z) = (x + y)(x + \bar{y})(\bar{x}\bar{z})$$

$$\begin{aligned}
 F(x, y, z) &= (x + y)(x + \bar{y})(\bar{x}\bar{z}) \quad (\text{Given}) \\
 &= (x + y)(x + \bar{y})(\bar{x} + \bar{z}) \quad (\text{DeMorgan}) \\
 &= (x + y)(x + \bar{y})(\bar{x} + z) \quad (\text{Double Complement}) \\
 &= (xx + x\bar{y} + xy + y\bar{y})(\bar{x} + z) \quad (\text{Distributive (OR)}) \\
 &= (x + x\bar{y} + xy + y\bar{y})(\bar{x} + z) \quad (\text{Idempotent}) \\
 &= (x + x\bar{y} + xy)(\bar{x} + z) \quad (\text{Inverse}) \\
 &= (x + x(y + \bar{y}))(\bar{x} + z) \quad (\text{Distributive}) \\
 &= (x + x)(\bar{x} + z) \quad (\text{Inverse}) \\
 &= x(\bar{x} + z) \quad (\text{Idempotent}) \\
 &= x\bar{x} + xz \quad (\text{Distributive}) \\
 &= xz \quad (\text{Inverse})
 \end{aligned}$$

### 6.2.3 Logic gates

- **Gate:** A gate is an electronic device that produces a result based on two or more input values

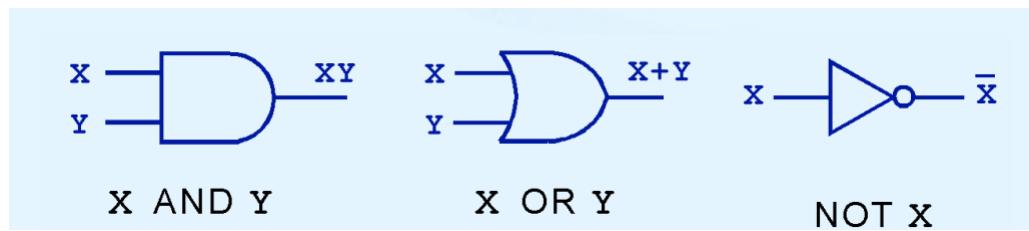
Technically, one or more input values, since a NOT gate only has one input.

In reality, gates consist of one to six transistors, but digital designers think of them as a single unit.

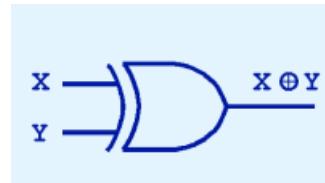
Integrated circuits contain collections of gates suited to a particular purpose.

Because of the associative law, AND-gates and OR-gates can also take in 3 or more inputs, but NAND and NOR are not associative

- **And, or, and not gates:**



- **XOR gate:** Another very useful gate is the exclusive OR (XOR) gate



The XOR gate is defined as

$$x\bar{y} + \bar{x}y.$$

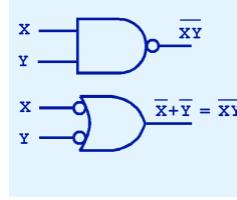
- **Properties of XOR**

- **Commutative:**  $A \oplus B = B \oplus A$
- **Associative:**  $(A \oplus B) \oplus C = A \oplus (B \oplus C)$
- **Identity element (0):**  $A \oplus 0 = A$
- **Self inverse:**  $A \oplus A = 0$
- **Inverse element (1):**  $A \oplus 1 = \bar{A}$
- **Distributive over AND (but not over OR):**

$$A(B \oplus C) = AB \oplus AC.$$

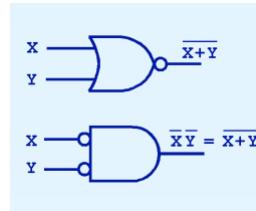
- **Non-Idempotent:**  $A \oplus A = 0 \neq A$
- **Relation to equality:**
  - \*  $A \oplus B = 1$  means  $A = B$
  - \*  $A \oplus B = 0$  means  $A \neq B$

- **NAND Gate:**



$$x \text{ NAND } y = \bar{xy} = \bar{x} + \bar{y}.$$

- **NOR Gate:**



$$x \text{ NOR } y = \bar{x+y} = \bar{x}\bar{y}.$$

- **Notation for NAND and NOR:**

- **NAND (Sheffer stroke):**  $x \uparrow y$
- **NOR (Peirce arrow):**  $x \downarrow y$

- **NAND, NOR - Universal:** NAND and NOR are universal gates.

A universal gate means that any Boolean function can be constructed using all NAND gates or all NOR gates

This makes NAND and NOR different from gates like AND and OR, which are not universal gates

Since NAND and NOR are inexpensive to manufacture, this is a useful property

To show that NAND is a universal gate, we need to show that the basic gates AND, OR and NOT can be built from NANDs

- **Building AND, OR, and NOT from NAND:** First, we remark on a key property of the NAND gate

$$x \uparrow x = \bar{x}.$$

Thus, we have our first gate: the NOT gate.

The two remaining gates are built in the following way

$$(x \uparrow y) \uparrow (x \uparrow y) = xy.$$

**Proof.** Since  $x \uparrow x = \bar{x}$ , and  $x \uparrow y = \overline{xy}$ , we have

$$(x \uparrow y) \uparrow (x \uparrow y) = \overline{(\overline{xy})(\overline{xy})} = \overline{\overline{xy}} = xy.$$

Finally, the OR gate is

$$(x \uparrow x) \uparrow (y \uparrow y) = x + y.$$

**Proof.** Since  $x \uparrow x = \bar{x}$ , and  $y \uparrow y = \bar{y}$ , we have

$$(x \uparrow x) \uparrow (y \uparrow y) = \overline{\bar{x}\bar{y}} = \overline{\overline{x+y}} = x + y.$$

■

- **Building AND, OR, and NOT from NOR:** For NOR, we have the same property,

$$x \downarrow x = \bar{x}.$$

So, the NOT gate is  $x \downarrow x$ . The AND and OR gates are built with

$$\begin{aligned} (x \downarrow x) \downarrow (y \downarrow y) &= xy \\ (x \downarrow y) \downarrow (x \downarrow y) &= x + y. \end{aligned}$$

Proofs left as an exercise to the reader.

- **Properties of NAND gates**

- **Commutative:**  $A \uparrow B = B \uparrow A$
- **Non-Associative:**  $(A \uparrow B) \uparrow C \neq A \uparrow (B \uparrow C)$
- **Non-Idempotent:**  $A \uparrow A = \bar{A} \neq A$
- **Absorption like behavior**

$$\begin{aligned} A \uparrow 1 &= \bar{A} \\ A \uparrow 0 &= 1. \end{aligned}$$

- **Universal**

- **Properties of NOR gates**

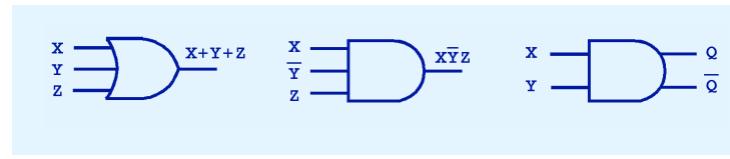
- **Commutative:**  $A \downarrow B = B \downarrow A$
- **Non-Associative:**  $(A \downarrow B) \downarrow C \neq A \downarrow (B \downarrow C)$
- **Non-Idempotent:**  $A \downarrow A = \bar{A} \neq A$
- **Absorption like behavior**

$$\begin{aligned} A \downarrow 0 &= \bar{A} \\ A \downarrow 1 &= 0. \end{aligned}$$

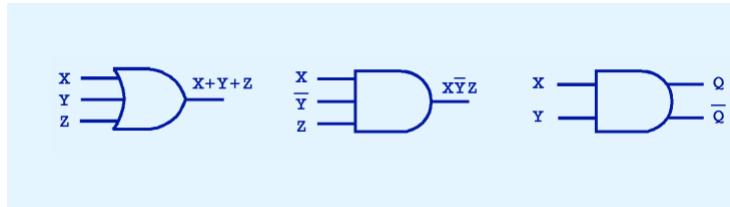
- **Universal:**

- **Generalizing Gates:** Gates can have multiple inputs and more than one output

A gate showing 3 or more inputs is really an abbreviation for a series of gates. This is made possible by the associative and commutative laws for AND and OR



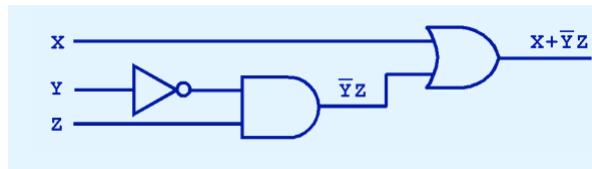
A gate with more than one output is an abbreviation for a set of gates with the same input, where each of them provides one of the outputs.



- **Implementing Functions:** Boolean functions are implemented with combinations of gates.

The circuit below implements:

$$F(x, y, z) = x + \bar{y}z.$$



#### 6.2.4 Combinatorial circuits

- **Combinatorial circuit:** We have designed a circuit to implement the Boolean function

$$F(x, y, z) = x + \bar{y}z$$

This circuit is an example of a *combinational* or *combinatorial logic* circuit

Combinational logic circuits produce a specified output at the instant when input values are applied.

- **Half Adder:** Combinational logic circuits give us many useful devices.

One of the simplest is the half adder, which finds the sum of two bits.

To build a half adder, look at its truth table, shown at the right

It has two inputs and two outputs.

$x$	$y$	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

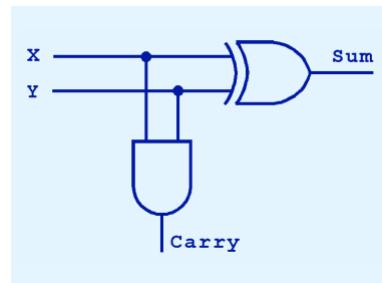
The sum can be calculated with XOR

$$\text{SUM}(x, y) = x \oplus y$$

The carry can be calculated with AND

$$\text{CARRY}(x, y) = xy$$

The half adder circuit is



**Note:** The dot denotes intersection of wires, no dot denotes no intersection.

We have used gates to build an arithmetic function. We needed to use the binary number system to do so. Every aspect of building a computer derives from this ability to use gates to build arithmetic functions.

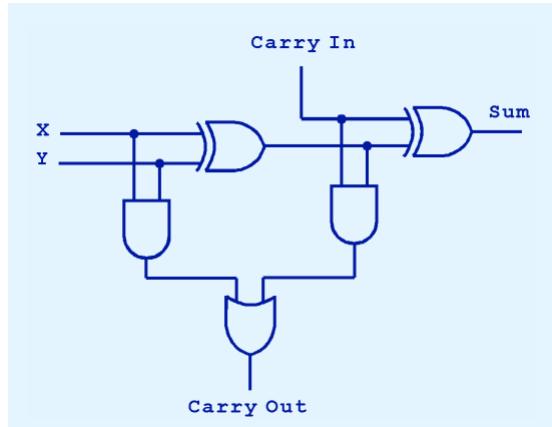
- **Full Adder:** A full adder handles the complete calculation of one column of a binary addition.

It has 3 inputs and 2 outputs.

The truth table for a full adder is shown below

Inputs				Outputs	
X	Y	Carry In		Sum	Carry Out
0	0	0		0	0
0	0	1		1	0
0	1	0		1	0
0	1	1		0	1
1	0	0		1	0
1	0	1		0	1
1	1	0		0	1
1	1	1		1	1

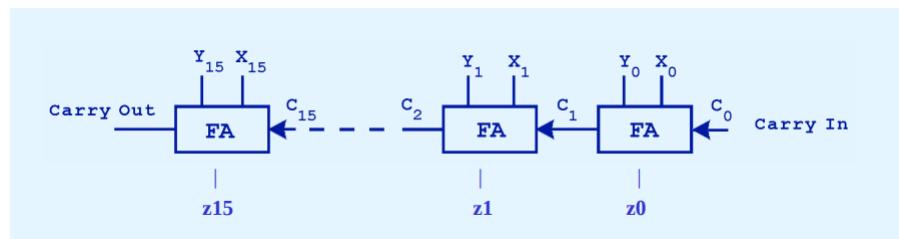
$$\text{SUM-FULL}(x, y, c) = \text{SUM}(\text{SUM}(x, y), c) = (x \oplus y) \oplus c$$



If either sums has a carry, we output one. If neither sums has a carry, we output zero.

$$\begin{aligned}\text{CARRY-OUT}(x, y, c) &= \text{CARRY}(x, y) \text{ OR } \text{CARRY}(\text{SUM}(x, y), c) \\ &= xy + (x \oplus y)c\end{aligned}$$

- **Ripple-Carry Adder:** Full adders can be connected in a series. The carry bit "ripples" from one adder to the next; hence, this circuit is called a ripple-carry adder.



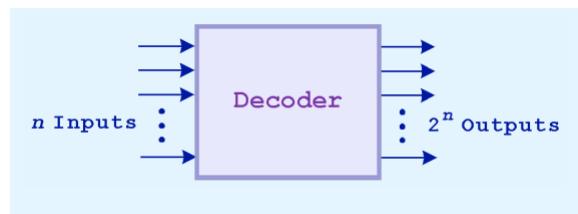
The right-most full adder does not need a carry-in. However, from the point of view of manufacturing efficiency, it does not hurt to use a full adder with the carry-in set to 0

You can do that by having no input to that item. Or you could replace the right-most full adder by a halfadder

- **Block diagrams:** Describe the external features of a circuit, the inputs and outputs.
- **Decoders:** Decoders are another important type of combinational circuit

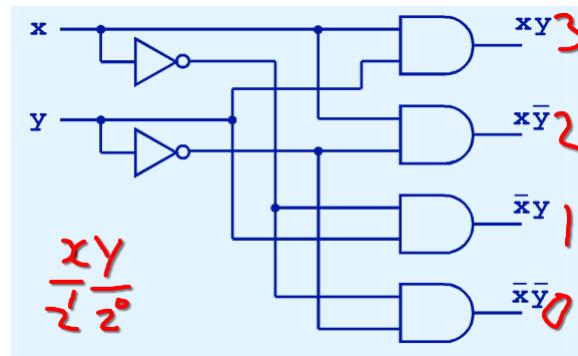
Among other things, they are useful in selecting a memory location according a binary value placed on the address lines of a memory bus

Address decoders with  $n$  inputs can select any of  $2^n$  locations.



**Figure 1:** This is a block diagram for a decoder

This diagram shows the internals of a 2-to-4 decoder



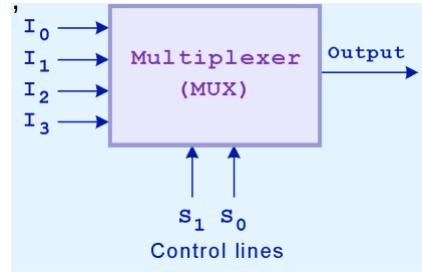
For example, the  $xy$  output is only true if  $x$  is true and  $y$  is true.

A decoder is used to convert a binary number ( $B'xy'$ ) to a value (0th, 1st, 2nd, 3rd output is true, counting from the bottom of the diagram).

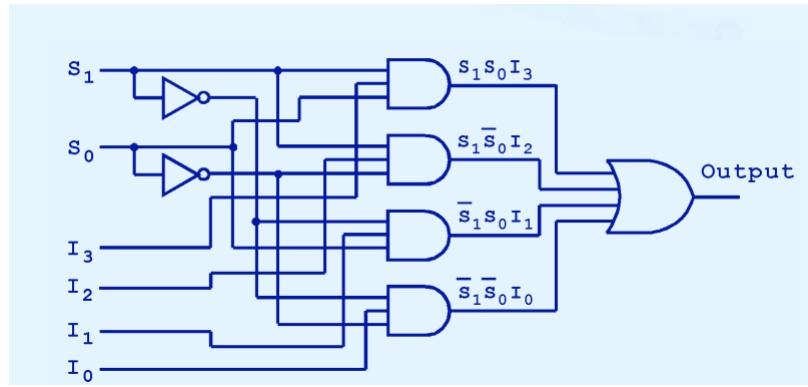
Given an address, a decoder can be used to access the memory cell at that address.

- **Multiplexers:** A multiplexer, abbreviated mux, is the opposite of a decoder. It selects a single output from several inputs. The particular input chosen for output is determined by the value of the multiplexer's control lines

To be able to select among  $n$  inputs,  $\log_2(n)$  control lines are needed.



This diagram shows the internals of a 4-to-1 multiplexer.



Note that only one of the AND-gates can be true at any given time.

Each input  $I_n$  is only connected to one of the AND-gates. That AND-gate is only true for a specific pair of values of  $S_0$  and  $S_1$ . For example, the top one is true when  $S_1 S_0 = 0b11$ .

Input  $I_n$  is connected to the AND-gate where the binary value  $S_1 S_0 = n$ . From top to bottom, the AND-gates are connected to inputs # 3, 2, 1 and 0

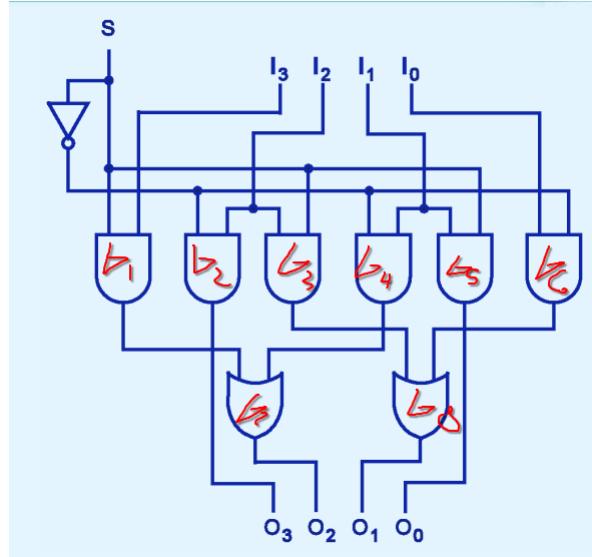
Each control input,  $S_0$  and  $S_1$ , is provided in its original and negated form to make it easy to build the four AND-gates

Since only one of the AND-gates is true at any one time, we can OR together their values to get the value of the input ( $I_0$ ,  $I_1$ ,  $I_2$ , or  $I_3$ ) selected by the two  $S$  lines.

A multiplexer can be used to send data from a given memory cell to a register. The memory address is given in  $S_1 S_0$ .

- **Shifter:** This shifter moves the bits of a nibble one position to the left or right.

If  $S = 1$ , we have a right shift. If  $S = 0$ , we have a left shift.



$$G_1 = I_3S$$

$$G_2 = I_2\bar{S}$$

$$G_3 = I_2S$$

$$G_4 = I_1\bar{S}$$

$$G_5 = I_1S$$

$$G_6 = I_0\bar{S}$$

$$G_7 = G_1 + G_4 = I_3S + I_1\bar{S}$$

$$G_8 = G_3 + G_6 = I_2S + I_3S$$

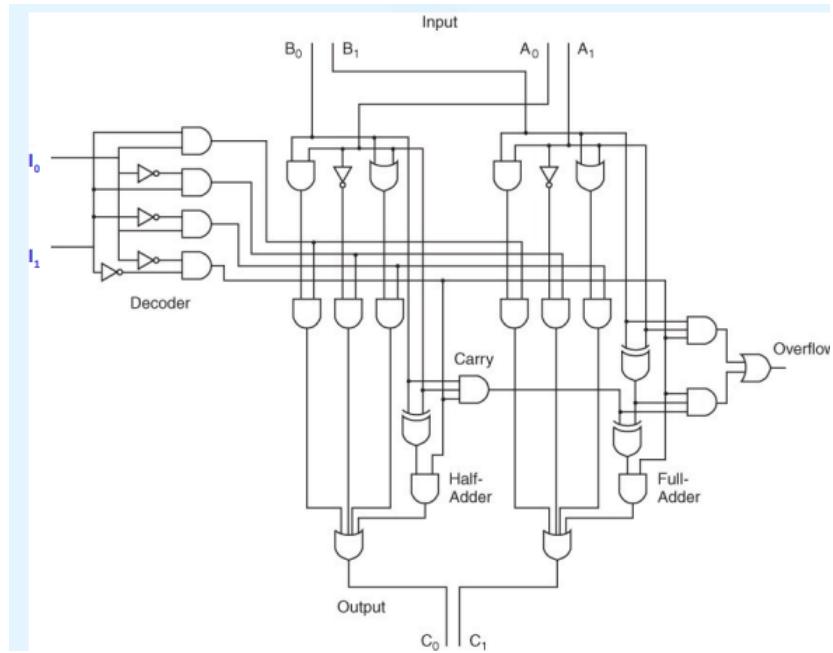
- **Two bit ALU:** The ALU, or arithmetic-logic unit, is a core part of the CPU that handles arithmetic and logic

We can use the combinational circuits we have learned to build an ALU. Our sample ALU will handle four operations - AND, OR, NOT and addition - for two-bit numbers.

The inputs are  $A_1A_0$  and  $B_1B_0$ . The output is  $C_1C_0$ .

The control lines  $I_0I_1$  determine which operation will be done

- 00 = Addition
- 01 = Not
- 10 = Or
- 11 = And



### 6.2.5 Sequential Circuits

- **Intro:** Combinational logic circuits are perfect for situations when we require the immediate application of a Boolean function to a set of inputs

There are other times, however, when we need a circuit to change its value with consideration to its current state as well as its inputs.

These circuits have to "remember" their current state.

*Sequential logic circuits* provide this functionality for us.

- **Sequential circuits:** As the name implies, sequential logic circuits require a means by which events can be sequenced.

State changes are controlled by clocks.

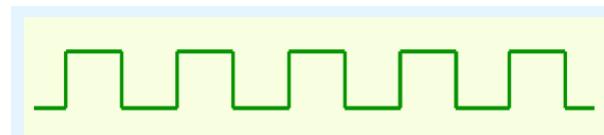
Sequential circuits are used anytime that we have a "stateful" application.

A stateful application is one where the next state of the machine depends on the current state of the machine and the input

A stateful application requires both combinational and sequential logic.

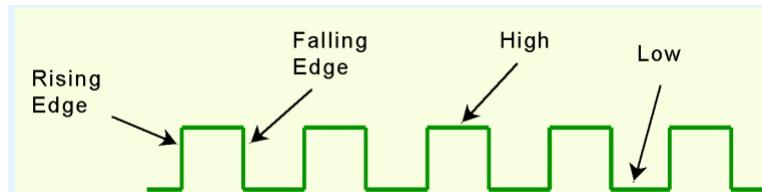
- **Clock:** A "clock" is a special circuit that sends electrical pulses through a circuit.

Clocks produce electrical waveforms such as the one shown below



- **State changes:** State changes occur in sequential circuits only when the clock ticks

Circuits can change state on the rising edge, falling edge, or when the clock pulse reaches its highest voltage



Circuits that change state on the rising edge, or falling edge of the clock pulse are called edge-triggered

Level-triggered circuits change state when the clock voltage reaches its highest or lowest level.

In general, the circuits that we will look at will get their inputs on the rising edge.

By the time of the falling edge, the output will be stable and we can use it.

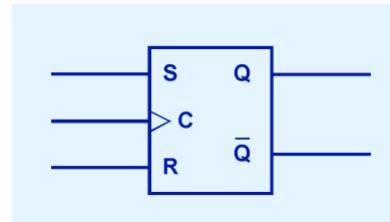
- **Feedback:** To retain their state values, sequential circuits rely on feedback.

Feedback in digital circuits occurs when an output is looped back to the input

A simple example of this concept is shown below.

- **SR flipflop (Set/Reset):** You can see how feedback works by examining the most basic sequential logic components, the SR flipflop

The block diagram for an SR flipflop is shown below along with its characteristic table.

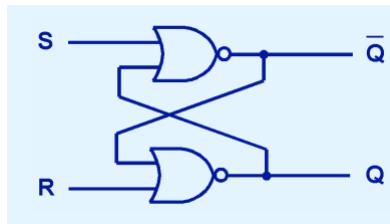


$S$	$R$	$Q(t + 1)$
0	0	$Q(t)$ (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	undefined

The characteristic table describes its behavior.

$Q(t)$  means the value of the output at time  $t$ .  $Q(t + 1)$  is the value of  $Q$  after the next clock pulse.

The internals are



We see that the top gate has output  $(\bar{Q}(t))$ , and the bottom has output  $Q(t)$ . The top gate gives

$$\bar{Q}(t+1) = \overline{S + Q(t)}$$

and the bottom gives

$$Q(t+1) = \overline{R + \bar{Q}(t)}.$$

At the clock tick, the top gets input  $S$  and  $Q(t)$ , and the bottom gate gets input  $R$  and  $\bar{Q}(t)$ .

For example, if  $S = 1$ ,  $R = 0$ , the outputs for the top and bottom gates, respectively are

$$\begin{aligned}\bar{Q}(t+1) &= \overline{S + Q(t)} = \overline{1 + Q(t)} = \bar{1} = 0, \\ Q(t+1) &= \overline{R + \bar{Q}(t)} = \overline{0 + \bar{Q}(t)} = \overline{\bar{Q}(t)}.\end{aligned}$$

But, we have that the complement of  $Q(t+1)$ ,  $\bar{Q}(t+1)$  is zero, so  $Q(t+1)$  is one.

If  $S = R = 0$ , then the gates give (top to bottom)

$$\begin{aligned}\bar{Q}(t+1) &= \overline{0 + Q(t)} = \overline{Q(t)}, \\ Q(t+1) &= \overline{0 + \bar{Q}(t)} = \overline{\bar{Q}(t)}.\end{aligned}$$

From here we can deduce that if  $Q(t) = 0$  or  $Q(t) = 1$ ,  $Q(t+1) = 0$ , which is expected.

**Note:** To understand basic computer architecture, we can consider flipflops as fundamental units, just as we treat gates, i.e., we don't need to know their internals.

The SR flipflop actually has three inputs:  $S$ ,  $R$ , and its current output,  $Q$ .

Thus, we can construct a truth table for this circuit, as shown at the right.

Notice the two undefined values. When both  $S$  and  $R$  are 1, the SR flipflop is unstable

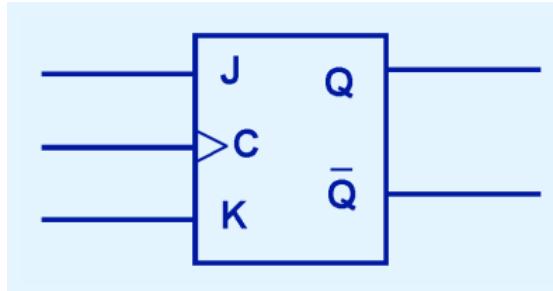
S	R	Q(t)	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	undefined
1	1	1	undefined

- **JK Flipflop:** If we can be sure that the inputs to an SR flipflop will never both be 1, we will never have an unstable circuit

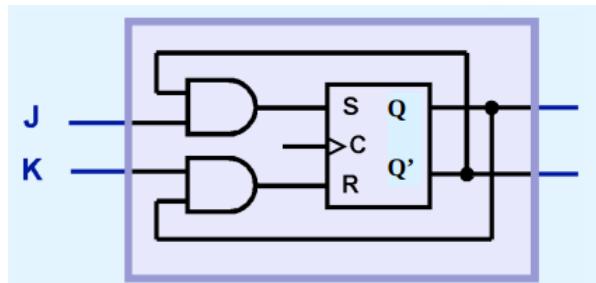
The SR flipflop can be modified to provide a stable state when both inputs are 1.

This modified flipflop is called a JK flipflop, shown at the right.

**Note:** The "JK" is named after *Jack Kilby*.



At the right, we see how an SR flipflop can be modified to create a JK flipflop



The characteristic table indicates that the flipflop is stable.

$J$	$K$	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\bar{Q}(t)$

When  $J$  and  $K$  are both 1, the JK flipflop changes its output from the preceding value.

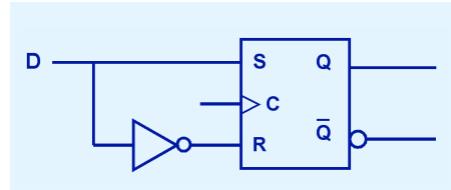
- **D Flipflop:** The output of the D flipflop remains the same during subsequent clock pulses. The output changes only when the value of  $D$  changes.

A  $D$  flipflop is built by sending  $D$  and  $D'$  into the  $S$  and  $R$  inputs of an SR flipflop.

The characteristic table is

$D$	$Q(t+1)$
0	0
1	1

With externals

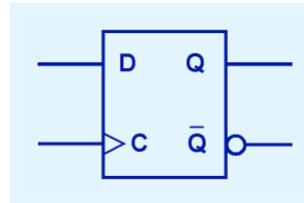


Since its value is constant unless the input changes, the  $D$  flipflop is the fundamental circuit of computer memory.

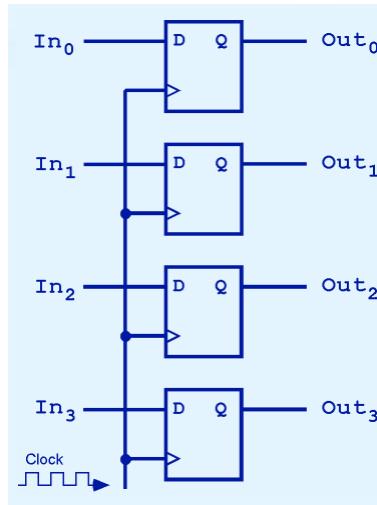
$D$  flipflops are usually illustrated using the block diagram shown below.

In addition to  $D$ , the clock input is needed as for any sequential circuit.

It is convenient but not required to output both  $Q$  and  $Q'$ .



- **Register:** This illustration shows a 4-bit register consisting of  $D$  flipflops. You will usually see its block diagram (below) instead.



This register maintains its state as the clock ticks because the value of  $In_n$  is equal to the value of  $out_n$ .

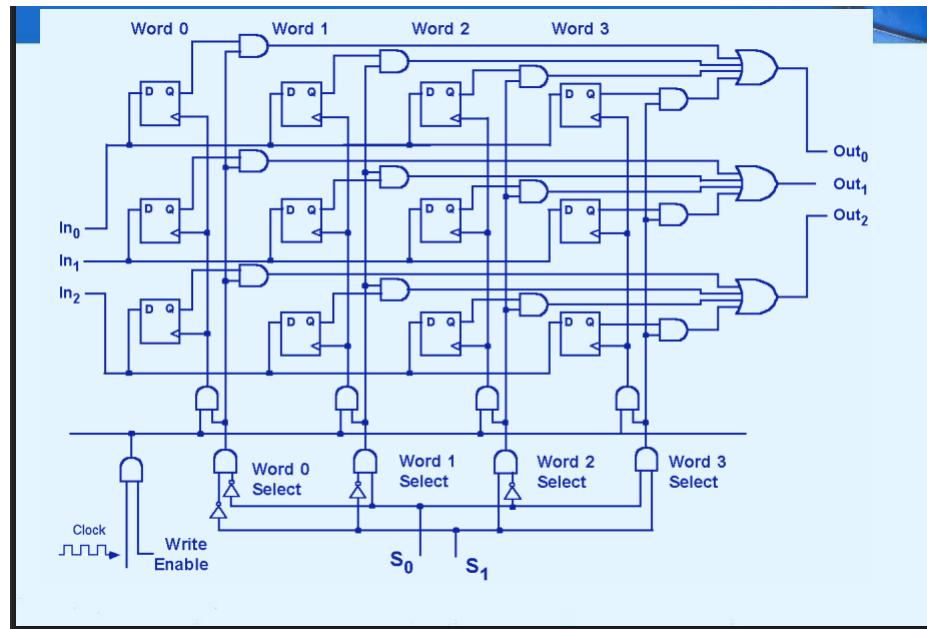
- **Memory:** The following image represents four words of memory. Each word has 3 bits, represented in one column

A 4-word memory needs 2 selector lines to address the four words. We need to know which word the data is going to or from

Note that  $S_1$  is the high-order bit even though it's shown on the right. Look carefully at the AND gates  $S_0$  and  $S_1$  feed into

When data is being written into memory, it comes from the 3 input bits on the left.

$In_0$  is connected to bit 0 of each word, but the update only happens if the corresponding selector line is 1 (and the write enable bit is 1) when the clock ticks.



- **Memory read:** When data is being read from memory (i.e., output), it goes into the 3 output bits on the right.

Each output bit contains the ‘OR’ of the corresponding output bits from the four words.

Only one of those bits can be true at any one time. That’s the output bit from the word that is selected, i.e., the word whose number (address) is represented by the selector lines.

Memory read (i.e., output) does not require a clock tick. The output is always there. To get the data from the  $Q$  signals for each memory cell to Out only requires combinational logic. The clock isn’t needed for combinational logic.

- **Memory Read example:** We will use bit 0 as an example again. Bit 0 of each word is AND-ed to the selector for that word number

So bit 0 of a given word is only output from the  $D$  flipflop it's stored in when the selector for that word is on.

All of the bit 0's from the different words are OR'd together to form output bit  $\text{Out}_0$ , but only one of them (the one from the selected word) will have a 1 in it.

So the value of  $\text{Out}_0$  will contain the value of bit 0 of the selected word, i.e., whatever is in the  $D$  flipflop. If the word had been updated during this clock cycle,  $\text{Out}_0$  will contain the new value. Otherwise it will contain the old value. Again, that's what we expect from a computer memory.

- **Memory write**

memory update (i.e., input) only happens when the clock ticks. That's the only time the clock bit in the lower left is on. Input is sampled on the rising edge and the output is stable by the time of the falling edge.

Also, memory update (i.e., input) only happens when the write enable bit is on.

Memory can only be updated when the "write enable" bit is on. There is a write enable line that is AND-ed with the clock

The clock ticks regularly, and there is always some value on the input lines (every bit is either 0 or 1), but we don't want to update memory every time the clock ticks.

So we only turn the write enable bit on when we want to update memory

- **Memory write example:** We will use bit 0 as an example. All 3 bits act the same way

Input  $\text{In}_0$  is connected to bit 0 of each of the words, but the data in bit  $\text{In}_0$  does not flow to bit 0 of each word. It only flows to the word represented by the selector.

Furthermore, the new value from  $\text{In}_0$  is only written when the clock ticks and write enable is on. (You can see the clock line and the write enable line AND-ed together, and that result AND-ed together with the selector line for the column.)

If write enable is off or the particular word is not selected, the data in the memory remains the same. That's how a  $D$  flipflop works, and it's also what you'd expect from a computer memory. We want values to be stable unless updated.

- **Implementing  $y = x$ :** Which is `LOAD x; STORE y`

First, read the value of  $x$  from memory:

1. Load the address of  $x$  on the selector lines
2. Now the value of  $x$  will be available on the Out lines.

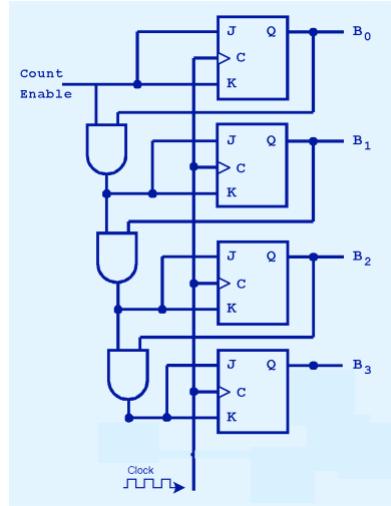
Then, write the value of  $y$  into memory:

1. Load the address of  $y$  on the selector lines
2. Load the new value of  $y$  (i.e., the value of  $x$ ) on the In lines.
3. Turn on the write enable bit
4. The next time the clock ticks, the new value of  $y$  will be loaded into the cells pointed to by the address of  $y$

**Note:** To implement something like  $y = x + 1$ , we basically do the same thing as above, but between outputting the value of  $x$  and loading that value int  $y$ , we feed the output of  $x$  into an ALU to get  $x + 1$

- **Counter:** A binary counter is another a sequential circuit. Among other things, it can be used to keep track of how many cycles the clock has ticked.

This is a 4-bit counter. It represents the 4-bit binary number  $B_3B_2B_1B_0$ .



It counts  $B'0000'$ ,  $B'0001'$ , etc. Note that the high-order bits are at the bottom of the diagram.

After it hits  $B'1111'$ , it returns to  $B'0000'$ . Now it can count forever.

When the count enable line is on, a pulse is produced every time the clock ticks.

The input is sampled on the rising clock edge and the output is stable at the time of the falling clock edge.

The count enable line feeds into both  $J$  and  $K$  of the top flip-flop.

When  $J = 1$  and  $K = 1$  at a clock tick, the JK flip-flop changes state, i.e., the output is the inverse of the previous output. Assuming the original value of  $B_0$  was 0, it is now 1. The value of  $B_0$  is also sent to the AND gate below it. So  $B_3B_2B_1B_0$  now equals  $B'0001'$ .

When will  $B_1$  flip to 1? On the next clock pulse, the  $J = 1$ ,  $K = 1$  input causes  $B_0$  to flip from 1 to 0. At the same time, the 1 from  $B_0$  is AND-ed with the 1 from the count enable line and sent to  $B_1$ , causing it to flip from 0 to 1. So  $B_3B_2B_1B_0$  now equals  $B'0010'$ .

When will  $B_2$  flip to 1? Its input comes from an AND gate. One of the inputs is  $B_1$ , and the other is the AND of the clock enable line and  $B_0$ .

The clock enable is 1 during counting, so the relevant part is  $B_0$ .

In other words,  $B_2$  will become 1 when  $B_1 = 1$  and  $B_0 = 1$ , as we expect from a counter. So the value after  $B'0010'$  is  $B'0011'$ , and the following value is  $B'0100'$ .

When  $B_3B_2B_1B_0 = B'1111'$ , the largest value that the counter can hold has been reached.

The next cycle will reset  $B_3B_2B_1B_0$  to  $B'0000'$  (i.e., all four bits will flip).

You can use a combinational circuit to add 1 to a number, but you need a sequential circuit, i.e., one that knows its previous value, to implement a counter.

## 6.3 Chapter 2: Data representation, integer and fixed point

### 6.3.1 Binary Fractions

- **Intro:** Fractions do not necessarily have exact representations in all bases, i.e., with a fixed number of digits.

In base 10, if the denominator contains powers of 2 and/or 5, the fraction can be exactly represented.

If a fraction cannot be exactly represented, it can always be represented by a fixed part plus a repeating part

In binary, only sums of powers of 2 can be exactly represented.

$$\begin{aligned}\frac{1}{2} &= 0b.1 \\ \frac{1}{4} &= 0b.01 \\ \frac{1}{8} &= 0b.001 \\ \frac{3}{4} &= \frac{1}{2} + \frac{1}{4} = 0b.11 \\ \frac{1}{2^{20}} &= 0b.0000000000000000000001\end{aligned}$$

And here are some fractions that can't be exactly represented:

$$\frac{1}{3} = 0b.010101\dots \frac{1}{12} = 0b.00010101\dots \frac{1}{5} = 0b.00110011\dots \frac{1}{10} = 0b.000110011\dots$$

Again, there is a repeating part, possibly preceded by a non-repeating part.

The fact that most fractions can't be exactly represented in a given number of bits is extremely important in computer science.

- **Truncation error:** Since any field has a finite width, we need to round off or truncate after that width is reached.

Usually we truncate, since that is easier for computer systems to do

That means that binary fractions are dangerous for representing money, for example, where we need an exact representation

For non-financial systems, we generally just need to know what the maximum possible error is.

- **Fractional representation:** Digits to the right of the radix point represent negative powers of the radix.

The radix point is called the decimal point or binary point in decimal and binary, respectively.

In base 10,

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$

In base 2,

$$\begin{aligned} 0.11_2 &= 1 \times 2^{-1} + 1 \times 2^{-2} \\ &= \frac{1}{2} + \frac{1}{4} = 0.75 \end{aligned}$$

- **Decimal fractions to binary (subtraction method):** For converting decimal fractions to binary, there is a subtraction method and a multiplication method.

We always start with the largest value first,  $2^{-1}$ , which is .5. Then we subtract  $2^{-2}$ , which is .25, etc

For example, suppose we want to convert 0.8125 to binary. We subtract  $\frac{1}{2}$ , then  $\frac{1}{4}$ , etc. If the number is too small to subtract the current fraction, that bit gets a zero.

$$\begin{array}{r} 0.8125 \\ - 0.5000 = 2^{-1} \times 1 \\ \hline 0.3125 \\ - 0.2500 = 2^{-2} \times 1 \\ \hline 0.0625 \\ - 0 = 2^{-3} \times 0 \\ \hline 0.0625 \\ - 0.0625 = 2^{-4} \times 1 \\ \hline 0 \end{array}$$

Our result, reading from top to bottom is

$$0.8125_{10} = 0.1101_2$$

We stop when we reach 0, or when we reach the desired number of bits.

For example, if we were converting .8126 to a binary number with a width of 4 bits, we'd still get 0b.1101, with an error term of .0001<sub>10</sub> (because there would be a 1 instead of a 0 at the end.)

**Note:** The subtraction method is not always convenient because it requires a table of negative powers of 2: .5, .25, etc.

- **Inverses of powers of two (first 8)**

$$\begin{aligned}
 2^{-1} &= \frac{1}{2} = 0.5 = 0b.1 \\
 2^{-2} &= \frac{1}{4} = 0.25 = 0b.01 \\
 2^{-3} &= \frac{1}{8} = 0.125 = 0b.001 \\
 2^{-4} &= \frac{1}{16} = 0.0625 = 0b.0001 \\
 2^{-5} &= \frac{1}{32} = 0.03125 = 0b.00001 \\
 2^{-6} &= \frac{1}{64} = 0.015625 = 0b.000001 \\
 2^{-7} &= \frac{1}{128} = 0.0078125 = 0b.0000001 \\
 2^{-8} &= \frac{1}{256} = 0.00390625 = 0b.00000001
 \end{aligned}$$

- **Number of output bits:** To know the number of output bits, you must look at the desired output data structure

Number of bits requested to the right of the point is often required to be a multiple of 2, 4 (nibble) or 8 (byte).

You can add as many 0's to the right as desired.

- **Decimal fractions to binary (multiplication method):** In this method, we repeatedly multiply by 2

Suppose we again want to convert decimal 0.8125, we begin by multiply by two.

$$0.8125 \times 2 = 1.6250$$

Whenever we get a part that's no longer a fraction (like the 1 in the example), we have finished converting that part, so we drop it from our calculation and save it for the binary result

continue multiplying each fractional part by 2.

We drop the 1, then multiply the remaining fraction (.6250) by 2.

$$\begin{array}{r}
 .8125 \\
 \times 2 \\
 \hline
 1.6250
 \end{array}$$
  

$$\begin{array}{r}
 .6250 \\
 \times 2 \\
 \hline
 1.2500
 \end{array}$$
  

$$\begin{array}{r}
 .2500 \\
 \times 2 \\
 \hline
 0.5000
 \end{array}$$
  

$$\begin{array}{r}
 .5000 \\
 \times 2 \\
 \hline
 1.0000
 \end{array}$$

We are finished when the remaining piece of the input is zero, as in this example, or until we have reached the desired number of binary places.

Our result, reading from top to bottom is

$$0.8125_{10} = 0.b.1101$$

- **Decimal fraction to hex:** Convert it to a binary fraction with the desired number of bits

Even if the conversion terminates earlier, fill out the binary fraction on the right with 0's to a full number of bytes.

Convert each nibble to hex

For example,  $\frac{7}{8} = 0b.11100000 = 0x.e0$

- **Converting binary fractions to decimal:** Converting binary fractions to decimal is a two-step process:

1. Convert the binary number to the right of the point to decimal.
2. Divide it by the appropriate power of 2

For example, if we wanted to convert  $0b.1101$  to decimal,  $1101 = 13$ , and there are four bits after the binary point, so the denominator is  $2^4$ . So, the result is  $\frac{13}{2^4} = \frac{13}{16}$

- **Hex fractions to decimal:** Hex fractions can be converted to decimal in the same way

For example,  $0x.d1$ ,  $0xd1 = 209$ , and there are two hex digits after the decimal, so we divide by  $16^{-2}$

You can also convert the hex fraction to binary first

### 6.3.2 Twos complement

- **Signed integers:** The conversions we have so far presented have involved only positive numbers.

There are three representation systems for signed integers:

1. Signed magnitude
2. One's complement
3. Two's complement

We only use the last one. We only look at the first two to show their disadvantages.

For signed numbers, it is essential to know the width of the field, e.g., 8, 16, 32 or 64 bits. Other widths are occasionally used.

- **Signed magnitude:** The signed magnitude system uses the highorder bit to indicate the sign of a value.

The high-order bit is the leftmost bit in a byte.

It is also called the most significant bit (MSB) because in an unsigned number, it represents the largest value.

We will occasionally also use the term LSB, least significant bit, for the rightmost bit, or the bit in the 1's position.

An 8-bit signed magnitude number contains:

- **bit 1:** sign bit (leftmost bit)
- **bits 2-8:** absolute value of the number (7 bits)

For example, in 8-bit signed magnitude,

$$3 = 0000\ 0011$$

$$-3 = 1000\ 0011$$

Signed magnitude has two significant disadvantages.

- It requires multiple rules to do arithmetic with signed magnitude numbers.
- There are two different reps. for zero, e.g., 0000 0000 and 1000 0000 in an 8-bit system.

But we want each number to have a unique representation so we can do comparisons.

For these reasons many computer systems employ complement systems to represent numeric values.

- **One's complement:** To calculate the one's complement of a number, we flip each bit.

For example, in 8-bit one's complement:

$$\begin{aligned} +3 &= 0000\ 0011 \\ -3 &= 1111\ 1100 \end{aligned}$$

In an 8-bit one's complement system, the largest absolute value uses 7 bits.

$$\begin{aligned} +127 &= 0111\ 1111 \quad (\text{Largest}) \\ -127 &= 1000\ 0000 \quad (\text{Smallest}) \end{aligned}$$

The high-order bit is the sign bit. As in the examples above, it identifies the sign and is not part of the magnitude.

- Positive numbers always have a 0.
- Negative numbers always have a 1.

- **Propagating the sign bit:** Again, notice the importance of the field width. In 8-bit one's complement:

$$\begin{aligned} +3 &= 00000011 \\ -3 &= 11111100 \end{aligned}$$

In 16-bit one's complement:

$$\begin{aligned} +3 &= 00000000000000011 \\ -3 &= 1111111111111100 \end{aligned}$$

So you can't left fill a negative number with 0's to move it to a larger field. You have to left fill negative numbers with 1's. This is called propagating the sign bit

- **Bit flipping:** Suppose you have a number  $b$  in binary, flipping all the bits of  $b$  is equivalent to computing

$$(2^N - 1) - b$$

For an  $N$  bit binary number, the maximum value is  $2^N - 1$ , which would be all  $N$  bits set to one. Take each bit of  $b$

- If the bit is 0, subtracting it from a 1 leaves 1.
- If the bit is 1, subtracting it from a 1 leaves 0.

That's exactly bit flipping

- **One's complement addition:** In one's complement addition, the carry bit is "carried around" and added to the sum.

$$\begin{array}{r}
 \begin{array}{c} 1 \\ 1 \\ 00110000 \\ 11101100 \\ \hline 00011100 \\ + 1 \\ \hline 00011101 \end{array}
 \end{array}$$

- **Why does end-around carry work?:** Suppose  $a$  and  $b$  are positive numbers. How would you add  $a + (-b)$ ? You would calculate  $a - b$

Now suppose  $b$  is represented in one's complement as  $(2^N - 1) - b$ .

Then doing unsigned arithmetic would give  $a + (2^N - 1 - b) = 2^N - 1 + (a - b)$ .

But the correct value is  $2^N + (a - b)$ , so we need to add 1

Although end-around carry adds some complexity, one's complement addition is simpler to implement than signed magnitude.

But it still has the disadvantage of having two different representations for zero:

$$\begin{aligned} 00000000 &= +0 \\ 11111111 &= -0 \end{aligned}$$

Two's complement solves this problem.

- **Two's complement:** To express a value in two's complement:
  - If the number is positive, just convert it to binary.
  - If the number is negative, find the one's complement of the number and then add 1.
- **MSB in two's complement:**
  - **Non-negative number:** MSB = 0
  - **Negative number:** MSB = 1
- **Range of signed (two's complement) numbers:** For an  $N$  bit number, where  $n = N - 1$ , the range is

$$\begin{aligned} \text{Range} &= -2^{N-1} \text{ to } 2^{N-1} - 1 \\ &= -2^n \text{ to } 2^n - 1. \end{aligned}$$

For example, the range of an 8-bit number is  $-2^7$  to  $2^7 - 1 = -128$  to 127. Thus,  $127 + 128 + 1 = 256$  possible numbers. Observe that we add one to include the value zero.

**Note:** The asymmetry is the price of having a unique zero.

- **Some representations (8-bit):**
  - 0 and -0: 00000000
  - -1: 11111111
  - +127: 01111111
  - -127: 10000001
  - -128: 10000000

### 6.3.3 Overflow

- **Intro:** What happens when the result of a calcuation doesn't fit in the given field width?

When we detect overflow, we can compensate for it and/or notify the user.

In complement arithmetic, an overflow condition is easy to detect

- **Rule for detecting signed two's complement overflow:** When the "carry in" and the "carry out" of the sign bit differ, overflow has occurred.

Adding two numbers of opposite sign never causes overflow.

- **Twos complement to decimal:** To convert from two's complement to decimal:
  - **If the sign bit is zero:** Just convert to decimal the unsigned way.
  - **If the sign bit is one:** There are two methods
    1. Subtract one, flip bits, convert to decimal, add minus sign
    2. Flip bits, add one, convert to decimal, add minus sign

#### 6.3.4 Binary math

- **Binary multiplication:** Can do binary multiplication the way people multiply by hand.

Easy to do in hardware because the algorithm is made up of adds and shifts.

$$\begin{array}{r} 0110 \\ \times \underline{0101} \\ 0110 \\ 0000 \\ \underline{0110} \\ 011110 \end{array}$$

- **Binary multiplication with Booth's algorithm:** Research into finding better arithmetic algorithms has continued for over 50 years.

The methods we use by hand are not the most efficient possible.

Booth's algorithm carries out multiplication faster and more accurately than our usual algorithm.

The general idea is to replace arithmetic operations with bit shifting to the extent possible.

Booth's algorithm is based on this idea:

$$\begin{aligned} n \times 1001 &= n \times (1000 + 1) \\ n \times 999 &= n \times (1000 - 1). \end{aligned}$$

In binary, this involves thinking of 1111 in the middle of a number as 10000 – 1 in the correct position.

### 6.3.5 Endianness (byte ordering)

- **Endianness:** Byte ordering, or endianness, is another architectural consideration. There are two options for storing an integer:
  - **Little endian machines:** store the least significant byte before the most significant byte, i.e., the least significant byte is stored at the lower address. Intel x86 machines (PCs) are little endian
  - **Big endian machines:** store the most significant byte before the least significant byte, i.e., the most significant byte is stored at the lower address. IBM mainframes are big endian

Let  $x \in \mathbb{R}$ ,  $b = \text{bin}(x)$ ,  $b = b_n b_{n-1} \dots b_2 b_1 b_0$ . In a big endian machine, suppose  $b$  was stored at address zero, then

$$\frac{b_n}{a_0} \frac{b_{n-1}}{a_1} \dots \frac{b_2}{a_{n-2}} \frac{b_1}{a_{n-1}} \frac{b_0}{a_n}.$$

In a little endian machine,

$$\frac{b_0}{a_0} \frac{b_1}{a_1} \frac{b_2}{a_2} \dots \frac{b_{n-1}}{a_{n-1}} \frac{b_n}{a_n}.$$

- **Advantages:** The big endian architecture has certain advantages
  - Regardless of the length of the number, the sign of the number can be determined by looking at the byte at address offset 0.
  - Strings and integers are stored in the same order.

The little endian architecture also has certain advantages

- Makes it easier to place values on non-word boundaries.
- Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic
- It can be considered a historical development due to changes in field length

- **Other places where endianness is important:**

- Endianness is an issue when creating a file to send to another machine
- Endianness is also an issue in networking. Some networking protocols take care of endianness for the user
- Endianness is an issue in Unicode. As long as each character is represented with one byte, there's no problem, but as soon as characters are represented by multiple bytes, the order must be determined. Unicode uses a special character, the BOM (byte order mark), to indicate endianness.

- **BOM:** The BOM (Byte Order Mark) is a special Unicode character used at the beginning of a text file or data stream to indicate:

- Which Unicode encoding is being used (UTF-8, UTF-16, UTF-32, etc.)
- The endianness of the encoding (for UTF-16 and UTF-32: little endian vs big endian)

Encoding	BOM Bytes (Hex)	Endianness
UTF-8	EF BB BF	N/A (byte order irrelevant)
UTF-16	BE FE FF	Big Endian
UTF-16	LE FF FE	Little Endian
UTF-32	BE 00 00 FE FF	Big Endian
UTF-32	LE FF FE 00 00	Little Endian

### 6.3.6 Ascii

- **Intro:** Characters also need to be represented as bit patterns

Types of characters:

- Upper case letters
- Lower case letters (were not in the original schemes)
- Numerals 0-9: not the same as binary values!
- Control characters

Different coding schemes have evolved over time. IBM mainframes vs. rest of computing world

- **ASCII:** ASCII = American Standard Code for Information Interchange. 7 bit scheme = 128 characters, originally based on telecommunications (Telex) codes

**Note:** Used everywhere in computer world except IBM

In ASCII the capital letters are in consecutive positions, 0x41 through 0x5a. Lower case letters also have a consecutive range, 0x61 through 0x7a. Numerals are 0x30 through 0x39. This is Different from EBCDIC, where alphabet is not consecutive

Some of the low-order ASCII characters are non-printing control characters. Some, such as line feed and carriage return, are used to control printing, others are used to control data transfer

- **Extended ASCII:** They have 8 bits = 256 characters. They have the same first 128 characters (characters with a 0 in the high-order (left) bit), but they have different characters for the second 128 characters (which have a 1 in the high-order bit)

"Extended ASCII" is not a standard, can't be a standard when there are multiple versions! Each 256-character set is called a "code page". Effectively, each code page is a standard. Different code pages were used for different languages

English users have it easy because we do not have diacritics (accent marks), but each western European language (French, Spanish, Portuguese, German, etc.) has a slightly different set of accented characters

- **Code page system:** With the code page system, users whose language used a non-Roman alphabet had even more problems. For example, there were multiple possible code pages for Russian, so a Russian user receiving an email or a file from another Russian user might see garbage

The issue isn't the code page per se; it's the encoding system, i.e., which character is attached to which ASCII position. Obviously, the code page system didn't work for anyone who wanted to write multilingual text

Also, Windows and Mac had different default code pages, this caused quote marks and accented characters to get mangled when sending a file from one computer to another. People tended to call these files "ASCII" when they really just meant byte-oriented

Adding to the confusion, code pages don't have standard names. ISO-8859-1 is an international standard code page for English (ISO = International Standards Organization), ISO-8859-1 is also called Latin-1 because it is a common code page for the Roman alphabet.

The Windows standard code page for English is CP-1252

- ISO-8859-1 and CP-1252 are similar but not identical
- ISO-8859-1 has control characters at positions 0x80 through 0x9f where CP-1252 has various symbols

Therefore Latin-1 is not the same as CP-1252 either, although all three names are frequently used incorrectly

Another point of confusion is that there are multiple ways to spell each of these code page names (underscores, hyphens, etc.)

- **Symptoms of wrong code page use:** When you see a file with a few special characters mangled, it's because the file was written with one code page but you are reading it with another. There is no way to know which code page was used to write a file because a file is just a sequence of bytes, so there is no place to put additional information like the name of the code page

This type of garbled text is often called mojibake, which is Japanese for "character transformation"

### 6.3.7 Unicode

- **Unicode intro:** Unicode is a newer system that can encode the characters of most languages in the world, newer than ASCII, but not new! The Unicode Consortium was established in 1991, it grew out of earlier work by Xerox and Apple
- **Advantages:** Unicode has many advantages
  - Unicode has room for the large numbers of characters needed by Chinese and Japanese
  - Unicode can handle languages like Korean that build each syllable as a single compound character
  - Unicode can handle 93 alphabets, which together cover most languages of the world
  - It can also handle emoji, historical alphabets such as hieroglyphics, and other special characters

Many programming languages now use Unicode as their default character encoding, Java, Python 3, etc.

Most common software supports Unicode

- gcc compiler
- XML web standard
- Word processors and spreadsheets
- Databases

Most operating systems today support Unicode, Linux, Windows, and Mac

- **A major exception:** C++ does not support Unicode, C++ still thinks the length of a string is the number of bytes it contains. That's valid for some purposes (e.g., networking) but not for others (e.g., string processing). For example, for names the length of a string should be the number of characters in the name, even if a character requires more than one byte
- **Aside: C++ character encodings:** C++ does not mandate a single character encoding. Instead, it depends on the platform, compiler, and the specific kind of character/string you are using. The C++ standard doesn't force UTF-8, ASCII, or any specific encoding, it just says your source code is in some "implementation-defined encoding."

In practice, Modern compilers (GCC, Clang, MSVC) assume your source files are in UTF-8 by default (unless told otherwise). Older systems may assume ASCII or Windows-1252.

So `std::string` is NOT automatically UTF-8 or ASCII - it's just a buffer of bytes. You decide what encoding goes inside.

When you print using `std::cout`, the raw bytes in `std::string` are sent to console/-file, whether they display correctly depends on your terminal encoding:

- Linux/macOS terminals: typically UTF-8
- Windows older consoles: CP437 or Windows-1252
- Windows Terminal / PowerShell now support UTF-8

- **Code point:** A Unicode character is called a code point, a code point is written as U+nnnn, where nnnn is a hex number. There are

$$17 \cdot 2^{16} - 8 \cdot 2^8.$$

Mostly numbered from U+0000 to U+10FFFF, range U+D800 through U+DFFF is skipped (They'll be needed for control codes later on))

- **Code planes:** The code points are divided into 17 code planes, with the exception of code plane 0, each code plane contains  $2^{16}$  code points

Unicode planes don't replace each other - they stack together to form one big unified code space from U+0000 to U+10FFFF. So when you're using characters from Plane 1 (like emojis), you can still freely use Plane 0 characters at the same time.

- **U+0000 - U+FFFF:** 4 hex digits, Plane 0 (BMP)
- **U+10000 - U+10FFFF:** 5 or 6 hex digits, Planes 1–16 (Supplementary Planes)

- **Code plane 0 (BMP):** Code plane 0 is the *basic multilingual plane* (BMP), it contains  $2^{16} - 8 * 2^8$  code points. They range from U+0000 to U+FFFF with the exception of U+D800 through U+DFFF. So, the BMP contains 63488 code points

Since even Chinese and Japanese need only 5000 to 10000 characters each for most purposes, the BMP is sufficient for most uses

The BMP (Unicode code plane 0) is shown below

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

Han is an attempt to use common characters for Chinese and Japanese where they overlap

- **Other planes:** Characters in the remaining code planes are used for various supplemental purposes. There are an additional 50,000 Chinese and other East Asian characters that are rarely used. They are used for poetry, historical writings, variant characters, etc. Some code points are used for hieroglyphics and other historical alphabets, some are saved for user expansion, many are unassigned

- **Relationship of Unicode to earlier standards:** The first 256 characters (U+0000 through U+00FF) match the ISO-8859-1 standard, this means that the lowest 128 characters (U+0000 through U+007F) match the ASCII standard

This is a statement about concepts, not implementation, it means that the printed character for each of the first 256 Unicode code points matches the printed character with the same hex value in ISO-8859-1

It does not mean that the hex implementation of each code point matches the hex value in ISO-8859-1

### 6.3.8 UTF

- **Intro:** Unicode is a content standard, it is not an implementation standard. Unicode defines what characters mean, their unique code points, and relationships between them. When Unicode "defines what characters mean," it's not talking about shapes (like fonts) or bytes (like encodings). It's talking about abstract identity and purpose; what a symbol represents in human language or writing.

For example,

Character	Unicode Code Point	Unicode Name	Meaning
A	U+0041	LATIN CAPITAL LETTER A	The capital letter "A" used in the Latin alphabet

So when we say Unicode defines what characters mean, we mean that Unicode gives every written symbol, letter, digit, accent, emoji, or special mark a unique identity and semantic purpose that is universally agreed upon.

It tells computers "This code point stands for this conceptual symbol in human writing." Unicode does not say how the letter looks or what bytes represent it.

Aspect	Example	Defined by
Meaning / identity	"U+0041 is the Latin capital A"	Unicode (content standard)
Shape / font	"A" looks serifed in Times New Roman, blocky in Courier	Font / rendering system
Bytes in memory	41 (UTF-8), 00 41 (UTF-16)	Encoding standard

So,

- Unicode identifies every possible text symbol as a unique abstract entity.
- It records its name, intended use, and relationships to other characters.
- It leaves the visuals and storage format to other systems.

Essentially, Unicode just specifies the content, It does not tell your computer how to store, transmit, or render those symbols in memory.

An implementation standard defines how to physically encode or represent data - e.g., the bytes used in memory or on disk.

UTF-8, UTF-16, and UTF-32 are implementation standards (also called encoding forms or encoding schemes). They define how a Unicode code point like U+1F642 becomes a sequence of bytes such as F0 9F 99 82 in UTF-8.

- Unicode says "there is a character U+1F642"
- UTF-8 says "to represent U+1F642 in bytes, use F0 9F 99 82."

That's why you can have multiple encodings for the same text, but they all refer to the same Unicode characters.

- **Implementation standards:** There are three implementation standards for different purposes, called encodings

- UTF-8, UTF-16 and UTF-32

Named after the number of bits they are based on, i.e., they are 1-byte, 2-byte and 4-byte schemes

UTF-16 and UTF-32 have big endian and little endian variants (written UTF-16 BE, etc.), If neither version is specified, big endian is the default. UTF-8 considers one byte at a time, so it doesn't need BE and LE variants

- **UTF-8:** UTF-8 is based on a single-byte model. Code points are represented by a single byte starting with 0 or a series of bytes, each of which starts with 1. No ambiguity: just look at first bit

If there is a reading error, you can just back up to the beginning of the byte string for that character (different from using escape char)

<b>Code point</b>	<b>byte 1</b>	<b>byte 2</b>	<b>byte 3</b>	<b>byte 4</b>
0000 - 007F	0xxxxxxx			
0080 - 07FF	110xxxxx	10xxxxxx		
0800 - D7FF, E000 - FFFF	1110xxxx	10xxxxxx	10xxxxxx	
10000 - 10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Code points U+0000 through U+007F are represented as single bytes starting with a 0 bit, this means that they look just like ASCII. In other words, no extra space is needed to convert existing ASCII files or strings to Unicode

The remaining code points are encoded with 2, 3, or 4 bytes, each starting with a 1 bit. That means there is no confusion, since the first 128 code points started with a 0 bit

Write the code point out in binary, then use the bits to fill in the x's in the chart

For example,  $U + 0041$  falls in the  $0000 - 007F$  range, so we will use one byte. 0041 in binary is 0000 0000 0100 0001. We have 7 x's, so we use the rightmost 7 bits. i.e 100 0001. Thus,  $U + 0041$  has one byte encoding  $01000001 = 0x41$ .

The first 128 code points uses one byte in the encoding, and the hex encoding is precisely the two rightmost hex digits in the code point.

$U + 00A9$  falls in the two byte  $0080 - 07FF$  range, so we use two bytes. 00A9 in binary is 0000 0000 1010 1001. According to the table above, we need to fill in 11 x's. We use the right most 6 for the second byte, and the next 5 for the first byte. The encoding is therefore

$$11000010\ 10101001 = 0xC2\ 0xA9.$$

$U+20AC$  falls into the 3 byte category. In binary, we have  $20AC = 0010\ 0000\ 1010\ 1100$ . We have 16 x's to fill, so we use all 16 bits. The rightmost 6 for the third byte, the next 6 for the second byte, and the remaining 4 for the first byte. The encoding is therefore

$$11100010\ 10000010\ 10101100 = E2\ 82\ AC.$$

We use the last four byte category to encode  $U + 1F600$  as  $F0\ 9F\ 98\ 80$

- **Ranges**

- **0000-007F:** code points 0-127 (ascii)
- **0080-07FF:** Code points 128 to 2047. These are the Non-ASCII Latin characters and many others.
- **0800-D7FF, E000-FFFF:** Code points 2048 to 55295, and code points 57344 to 65535. These are the Most common world scripts (Greek, Cyrillic, Arabic, etc.).
- **10000-10FFFF:** Code points 65535 to 1114111. This is the Supplementary planes - emoji, historic scripts, etc.

- **Code points 0-127, and 128-256:** Code points 0-127 are the one byte code points, they have the same hex encoding as ascii.

Code points 128-256 have the same hex value as extended ascii ISO-8859-1 in the code point representation, but not in the encoding. In fact, they can't, since they are all 2-byte strings and all of ISO-8859-1, like every other extended ASCII code page, consists of single bytes

- **Better UTF-8 Explanation:** UTF-8 is a variable-length encoding:

- It uses 1 to 4 bytes per character.
- Each byte has a specific bit prefix that tells you whether it's a single-byte character or part of a multibyte sequence.
- Because of these prefixes, there's no ambiguity when reading a byte stream.

Type	Prefix	Meaning
Single-byte (ASCII)	0xxxxxxx	Only 1 byte. Represents code points U+0000 to U+007F (standard ASCII).
Start of 2-byte sequence	110xxxxx	First byte of a 2-byte sequence.
Start of 3-byte sequence	1110xxxx	First byte of a 3-byte sequence.
Start of 4-byte sequence	11110xxx	First byte of a 4-byte sequence.
Continuation bytes	10xxxxxx	These never occur on their own; they follow a start byte.

Start bytes are easy to identify (0, 110, 1110, 11110), Continuation bytes always start with 10, a UTF-8 decoder can resynchronize after a read error by scanning until it finds a valid start byte (not beginning with 10), this is one reason UTF-8 is robust and preferred for network protocols and files.

- **Usefulness of UTF-8:**

- Code points in the BMP never need > 3 bytes
- Most alphabetic characters (as opposed to Asian characters) never need > 2 bytes
- But you do have to deal with a variable length encoding
- Hopefully a library in your chosen language will handle it for you

- **Uses of UTF-8:**

- As of 2015, over 85% of web pages were encoded in
- UTF-8. (Over 50% already by 2010)
- Recommended email encoding
- Used internally by Linux

We can check the internal Unicode implementation on Linux with

```
1 echo $LANG
```

An example output might be `en_US.UTF-8`. That UTF-8 tells you that your terminal and standard library routines (like `printf`) interpret text as UTF-8.

We can also run

```
1 locale charmap
```

Which will output UTF-8. That shows the active encoding used by your libc and shell environment.

- **UTF-16:** Each code point is encoded as 2 or 4 bytes, each code point in the BMP only needs 2 bytes, that means that for code points in the BMP, UTF-16 can be considered a fixed-length encoding

Note that the left half of each 2-byte unit in a 4-byte entry is one of the missing code points (D800–DFFF) – they were dropped to avoid ambiguity

- 2 bytes (16 bits) for characters in the Basic Multilingual Plane (BMP) - code points U+0000 through U+FFFF except a small gap.
- 4 bytes (two 16-bit units) for characters outside the BMP - i.e. U+10000 through U+10FFFF.

Each 16-bit unit is called a code unit.

<b>Code point</b>	<b>First two bytes</b>	<b>Second two bytes</b>
0000 - D7FF, E000 - FFFF	xxxx xxxx xxxx xxxx	
10000 - 10FFFF (subtract 10000 first)	1101 10xx xxxx xxxx	1101 11xx xxxx xxxx

We see, in UTF-16, all the code points from U+0000 through U+FFFF, excluding the surrogate gap U+D800–U+DFFF, are in a perfect one-to-one (bijective) correspondence with their 16-bit encoded form.

**Note:** UTF-16 is used internally by Windows and Java

- **The missing range (surrogates):** UTF-16 originally planned to fit everything in 16 bits (65,536 values), but it ran out of space. So for code points above U+FFFF, we need a way to represent them using two 16-bit words.

However, that creates a danger: How can a program tell whether a 16-bit value is a single code point or just half of a 4-byte pair? That's where the "missing code points" come in.

UTF-16 permanently reserved the 2,048 code points from U+D800 to U+DFFF and declared: "These values will never correspond to real characters."

At that time, no symbols had ever been assigned there, it was deliberately left blank. The gap has always been empty.

These are the surrogates:

- **D800–DBFF:** high surrogates
- **DC00–DFFF:** low surrogates

If they weren't dropped, a sequence of two code units could look like either:

- two real BMP characters, or
- one big code point's 4-byte encoding - impossible to tell apart.

By banning those 2,048 code points from being actual characters, UTF-16 can safely use them as markers for surrogate pairs.

For code points above U+FFFF, UTF-16 uses this scheme:

1. Subtract 0x10000 (65536) from the code point (so you now have a 20-bit value).
2. Split the 20 bits into two 10-bit halves:
  - high ten bits → "high surrogate"
  - low ten bits → "low surrogate"
3. Add offsets:
  - high surrogate = 0xD800 + high ten bits
  - low surrogate = 0xDC00 + low ten bits

Result: a 4-byte UTF-16 sequence that starts with a high surrogate (1101 10xx xxxx xxxx) and ends with a low surrogate (1101 11xx xxxx xxxx).

The goal of that subtraction is to normalize the range U+10000–U+10FFFF into a compact 0–0xFFFF space (20 bits = 1,048,576 values). That 0–0xFFFF range will then be represented by two 16-bit surrogate code units.

For example, consider  $U + 1F604$ ,

1. **Subtract 0x10000:**  $0x1F604 - 0x10000 = 0xF604 \rightarrow \text{binary } 1111\ 0110\ 0000\ 0100$
2. Split into 10-bit halves:
  - high 10 = 1111011000 = 0x3D8
  - low 10 = 00000100 = 0x004
3. Compute surrogates:
  - high = 0xD800 + 0x3D8 = 0xD3D8
  - low = 0xDC00 + 0x004 = 0xDC04
4. UTF-16 encoding = D83D DC04 (hex).

Those start with bits 110110 and 110111, matching the table.

When a UTF-16 decoder reads text, it looks at the next 16-bit code unit (2 bytes):

- If it's not in D800–DFFF, that value itself is the code point (within the BMP).
- If it is in D800–DBFF, it's a high surrogate, so the decoder knows: "I'm reading the first half of a 4-byte pair - expect a low surrogate next." It reverses the process to get the original
- **UTF-8 vs UTF-16:** One of the UTF standards may be more convenient than another for a given application. For example, suppose you are writing a guidebook to China in English, with an occasional Chinese name, recall that all the common characters in both English and Chinese are in the BMP
  - UTF-8 will be the most space-saving format, since it will only require one byte for each ASCII character and two bytes for each Chinese character
  - UTF-16 will require two bytes per character. In this case UTF-16 can be processed as a fixed length format, which makes the programming easier, but requires almost twice as much space
- **UTF-32:** Every code point is encoded the same way, simplest programming but the highest space usage. Fixed-length encoding with no programming required to retrieve the code points

If you are only using the BMP, you can get the same advantages from UTF-16. Although space is less of an issue every year, both in memory and on disk

Code point	4-byte unit
0000 - D7FF, E000 - 10FFFF	0000 0000 000x xxxx xxxx xxxx xxxx xxxx

UTF-32 still "skips" the surrogate gap U+D800–U+DFFF, even though it could represent every 32-bit value. Those 2,048 code points are still illegal in UTF-32 (and UTF-8, too). "The surrogate code points U+D800–U+DFFF are reserved for UTF-16 encoding and are not valid Unicode scalar values." So even though UTF-32 could numerically store 0xD800, it's not a valid character and must be treated as an error.

Otherwise, if UTF-32 allowed D800-DFFF to mean characters, then:

- Converting between UTF-16 ↔ UTF-32 would lose information or produce invalid data.
- Unicode's lossless round-trip guarantee would break.
- **Byte serialization:** Byte serialization = the order bytes are processed in = endian-ness, irrelevant for UTF-8

For UTF-16 and UTF-32, need a way to know whether file is big endian or little endian, If files are internal to your company, you might have a company standard. File might be identified by MIME descriptor (identifier of an email attachment). Mozilla and other programs try to guess from byte usage

If there is no other way or you just want to be clear, use a byte order mark (BOM)

- **Byte order mark (BOM) for UTF-16:** Normally people do want to be clear and use a BOM, if BOM is used, it must be at the beginning of the file
  - 0xfffe indicates big endian
  - 0xffff indicates little endian

If neither is present and there is no other way to know, big endian is the default

- **Byte order mark for UTF-32:** Same rules as above
  - 0x0000feff for big endian
  - 0xffffe0000 for little endian
- **Microsoft's non-standard use of a "BOM" for UTF-8:** Some Microsoft products (and others) add what they call a BOM at the front of UTF-8 files, It isn't really a BOM, just an indicator of UTF-8, since there is no order inside a single byte

The BOM is 0xeeebbf = UTF-8 rep. of U+FEFF

When Excel exports CSV files, it adds a BOM at the beginning, so does Blackboard when exporting grades, Google Docs adds a BOM when converting to plain text

- **Why this non-standard usage is problematic:** It means that a UTF-8 file of only Ascii characters doesn't match its Ascii equivalent any more, Although the Unicode Consortium doesn't forbid this, they recommend against it, because they want UTF-8 for low order characters to line up with Ascii
- **How to cope with Microsoft's non-standard usage:** You have to remove the extra BOM if the receiving program doesn't know to ignore it. Many programs that expect plain text treat the BOM as 3 extra characters

Shell scripts that expect "#!" at the beginning find these other characters at the beginning instead

- **Microsoft's non-standard use of the BOM for UTF-16 and UTF-32:** For UTF-16 and UTF-32, since Windows is little endian by default, some Windows applications assume that the default for files without a BOM is little endian. You could technically say this is OK if you're never going to use a Windows-created file outside of Windows software, but in fact other programs don't know your file was created by Windows, so it's a bad idea

### 6.3.9 Floating point concepts

- **Intro:** The signed magnitude, one's complement, and two's complement representation are used to represent integers.

We need a representation for fractional values in both scientific and business applications. Floating point representation solves this problem

- **Fixed-point:** We could arbitrarily decide that the binary point would be at a particular place in a 32-bit binary number, rather than at the end as for integers. That would be a “**fixed-point**” representation.
- **Rounding in binary:** Exactly like decimal rounding:
  1. Look at the bit after the last available bit.
  2. If that bit is 0, leave the last kept bit unchanged.
  3. If that bit is 1, add 1 to the last kept bit.

So, for  $0b.0001$ , if we only have three bits available, we round to  $0b.001$ .

- **Fixed-point mathematically:** If you choose to represent a number using  $n$  total bits, you may distribute those bits between
  - the integer portion (before the binary point)
  - the fractional portion (after the binary point)

The split is arbitrary because the binary point location is just a convention. For example,

- 4 bits for integer, 4 bits for fraction:

$$xxxx.yyyy.$$

- 6 bits integer, 2 bits fraction:

$$xxxxxx.yy.$$

- 1 bit integer, 7 bits fraction:

$$x.yyyyyy.$$

Any split is allowed. The binary point is a conceptual separator.

**Note:** For a  $n$ -bit fractional part, the maximum fractional part is

$$\begin{aligned} \frac{1}{2^1} + \frac{1}{2^2} + \dots + \frac{1}{2^n} &= \sum_{k=1}^n \frac{1}{2} \left(\frac{1}{2}\right)^k = \frac{\frac{1}{2}(1 - (\frac{1}{2})^n)}{1 - \frac{1}{2}} \\ &= 2 \left( \frac{1}{2} - \frac{1}{2^{n+1}} \right) = 1 - \frac{1}{2^n}. \end{aligned}$$

Further, the fraction part is written as

$$b_{-1}b_{-2}b_{-3} \cdot b_{-n}.$$

With decimal value

$$b_{-1}2^{-1} + b_{-2}2^{-2} + b_{-3}2^{-3} + \dots + b_{-n}2^{-n}.$$

Let's factor out a  $2^{-n}$ ,

$$2^{-n} (b_{-1}2^{n-1} + b_{-2}2^{n-2} + b_{-3}2^{n-3} + \dots + b_{n-1}2^1 + b_n).$$

Notice that the part in the parenthesis is precisely the integer value of a binary number, lets call it  $k \in \mathbb{Z}$ . Thus, the fractional part in binary fixed-point is precisely

$$\frac{k}{2^n} \quad k, n \in \mathbb{Z}.$$

So, only fractions whose denominator is a power of two can be represented exactly.

Regarding the  $m$ -bit integer part, the maximum is

$$1 + 2 + 4 + \dots + 2^{m-1} = \sum_{k=0}^{m-1} 2^k = \frac{1 - 2^{(m-1)+1}}{1 - 2} = -(1 - 2^m) = 2^m - 1.$$

- **Fixed-point implementations:** A fixed-point format defines:

- a fixed number of integer bits
- a fixed number of fractional bits
- and this split does not change

For example, a common format is  $Qm.n$ . This is  $m$  bits integer,  $n$  bits fraction.  $Q8.8$  format would be

`0biiiiiii.ffffffffff`

with

- **Largest integer value (all integer bits on):**  $2^8 - 1$
- **Largest fractional value (all fractional bits on):**  $1 - 2^{-8}$
- **Smallest fraction value (only last fractional bit on):**  $2^{-8}$

- **Problems with fixed-point:** Fixed point has several major flaws:

1. Because fractional powers of 10 (e.g., .1, .01, etc.) can't be exactly expressed by binary fractions, we will have roundoff errors when we try to do calculations.

For example, suppose we have only three bits after the binary point, then  $1/10 = 0b.001$ , and  $5 \times 1/10 = 0b.101$ . But, the correct answer is  $1/2 = 0b.100$

Converting  $1/10$  to binary yields  $0b.00001100110011\dots$ , if we only use three bits, we truncate to  $0b.001 = 1/8 = 0.125$ . If we multiply this approximation by 5, we get  $5 \cdot 1/8 = 5/8 = 0b.101$ .

2. We lose significance when subtracting two numbers of similar magnitude

The same problem can happen with multiplication (overflow) and division (underflow).

3. We can't express very large or very small numbers

Wouldn't it be better if we could move the binary point as necessary, giving us the equivalent of scientific notation?

- **Floating point intro:** Floating-point allows the scale of the number to change. Instead of committing to a fixed location for the radix point, the representation moves it to suit the magnitude. This means

- It can represent extremely large and extremely small numbers using the same number of bits.
- Precision is not uniform; it depends on magnitude.
- Range is much wider than fixed-point.

Floating point dedicates part of its representation to scaling the number instead of making all bits contribute directly to the magnitude. Because the scale of the number can change, the representable values are not limited to what you can write directly in  $n$  bits. The format can “zoom out” to represent very large numbers and “zoom in” to represent very small ones.

Fixed-point cannot change scale. Its range is tied directly to the number of bits and where the point is fixed.

So the larger range of floating point ultimately exists because the format includes a mechanism that allows the numeric value to grow by changing scale, rather than by using more magnitude bits.

When some of the available bits are used to control scale rather than directly represent digits of the number itself, fewer bits remain for the level of detail of the number. That means the spacing between representable values grows as the numbers get larger.

So floating point has:

- High relative precision for small numbers.
- Lower relative precision for large numbers.

Floating point mirrors the same idea used in scientific notation, just adapted for binary instead of decimal.

When we write very large or very small numbers, we do not write all the zeros. We write:

$$(\text{significant-digits}) \times (\text{base}^{\text{scale}}).$$

For example,

$$6.03 \times 10^{23}.$$

**Note:** To ensure only one representation per number, we always write exactly one digit before the radix point. Otherwise the first value above could have alternate representations like  $12.5 \times 10^{-2}$  and  $125 \times 10^{-3}$ .

- **Scientific notation:** Numbers written in scientific notation have three components:

1. Sign
2. Mantissa
3. True exponent

For example, consider again  $+1.25 \times 10^{-1}$ . In this case,

1. **Sign:** +
2. **Mantissa:** 1.25
3. **True exponent:** -1

- **Floating point:** Computer representation of a floating point number consists of three fixed-size fields:
  1. **Sign:** The one-bit sign field is the sign of the entire value.
  2. **Biased exponent:** The size of the exponent field determines the range of values that can be represented
  3. **Significand:** The size of the significand determines the precision of the representation, i.e., how many binary (or decimal) digits can be represented

The fields usually occur in this order, but it isn't necessary.

- **IEEE-754:**
  - **IEEE-754 single precision:** The IEEE-754 single precision floating point standard uses an 8-bit exponent and a 23-bit significand, so the total number of bits is 32
  - **IEEE-754 double precision:** The IEEE-754 double precision standard uses an 11-bit exponent and a 52-bit significand, so the total number of bits is 64.
- **Significand intro:** The significand of a floating point number is always preceded by an implied binary point. Thus the significand always contains a fractional binary value, which is an encoding of the mantissa
- **Biased exponent intro:** The exponent indicates, in an encoded fashion, the power of 2 to which the significand is raised.
- **Issues to resolve:** There are three issues that we need to resolve:
  1. Just as with fixed point, we want to make sure our representations are unique, i.e., each number has only one floating point representation.
  2. There is no sign in the exponent field, so how do we represent negative exponents?
  3. Should zero have a plus or a minus sign
- **Synonymous forms (normalization):** To resolve the problem of synonymous forms, we will establish a rule that the significand has only one bit before the binary point and that bit must be a 1. We adjust the exponent to make this true

For example, if the mantissa is 0b.001011, we convert it to a significand of 0b1.011 and subtract 3 from the exponent.

If the mantissa is 0b1011.00, we convert it to a significand of 0b1.011 and add 3 to the exponent.

Now, since the first bit is always 1, there is no need to represent it. In the IEEE-754 standard, it is implied.

That gives us one extra bit of precision, i.e., we now have 24 bits we can use to represent the fractional part of the significand rather than 23.

For example, if the significand is all 0's, that represents a mantissa of 1.0.

This process is called **normalization**, and the result is called a **normalized floating point number**

- **Biased exponents and the bias:** To provide for negative exponents, we use a biased exponent

We subtract the bias from the value stored in the exponent to determine the true value

In the IEEE standard, if the exponent has  $n$  bits, the bias is  $2^{n-1} - 1$ .

- **Excess-127:** The IEEE-754 single precision floating point standard has an 8-bit exponent, so the bias is 127. This is called **excess-127** representation.

Since an 8-bit number can range from 0 to 255, the true value of the exponent could theoretically range from -127 to 128. But all 0's and all 1's are saved for special purposes. Therefore, the exponent can range from -126 to 127.

- **IEEE-754 single precision:** A single-precision floating-point number uses 32 bits divided into three fields:

1. **Sign:** 1 bit
2. **Biased exponent:** 8 bits
3. **Fraction / Mantissa (F):** 23 bits

The layout is

1. **Bit 31:** Sign
2. **Bits 30–23:** Exponent
3. **Bits 22–0:** Fraction

A normalized finite number in IEEE-754 single precision has the form:

$$(-1)^S \times (1.F) \times 2^{E-127}.$$

Where

- $S \in \{0, 1\}$
- $E \in \{1, \dots, 254\}$  (range for normalized values)
- $F$  is a 23-bit binary fraction, contributing the digits after the binary point
- The constant 127 is the bias for the exponent field
- The leading 1. (called the hidden or implicit leading bit) is used only for normalized values

Notice that the **mantissa** is the fractional part, whereas the **significand** is  $1.F$ , with value  $2^0 + F = 1 + F$ . Recall that this leading one is not stored, it is implied.

- **Deriving the bias:** Consider the unaltered range for an  $N$ -bit exponent:

$$0 \text{ to } 2^N - 1.$$

This range has  $2^N - 1 - 0 + 1 = 2^N$  numbers. We wish to shift this range so that half is non-positive, and half is positive. Half of our  $2^N$  numbers (0 to  $2^N - 1$ ) is  $2^N/2 = 2^{N-1}$ .

If half of our new range is positive ( $> 0$ ), then the greatest number in this new range is  $2^{N-1}$ , since  $2^{N-1} - 1 + 1 = 2^{N-1}$ .

To find our bias that shifts the range, we just need to compute the number that takes us to  $2^{N-1}$  from  $2^N - 1$ . That number is precisely

$$2^N - 1 - 2^{N-1} = 2^N - \frac{2^N}{2} - 1 = 2^N \left(1 - \frac{1}{2}\right) - 1 = 2^N \left(\frac{1}{2}\right) - 1 = 2^{N-1} - 1.$$

Thus, the shifter (bias) is  $2^{N-1} - 1$  for an  $N$ -bit exponent.

- **Normalized values:** Normalized values are the regular floating-point numbers in IEEE-754 that use the implicit leading 1 in the significand. They make up almost the entire range of representable finite values.

A floating-point number is normalized when its exponent field satisfies

$$1 \leq E \leq 2^N - 2.$$

For IEEE-754 single precision, the exponent is 8-bits, so the range is

$$1 \leq E \leq 254.$$

In this case, IEEE-754 interprets the value as

$$(-1)^S \times (1.F) \times 2^{E-127}.$$

The key distinguishing feature is the implicit leading 1 in the significand

$$1.F = 1.f_1 f_2 \dots f_{23}.$$

So, normalized numbers have 24 bits of precision instead of 23.

Normalization ensures

1. The significand is always between 1 and 2.
2. There is a unique representation for each nonzero value (no multiple leading zeros).
3. Maximum precision is maintained.

**Note:** The bias rule only applies for normalized numbers

- **Range of normalized values (IEEE-754 single precision):** Consider IEEE-754 single precision. The smallest positive normal number is when  $S = 0$ ,  $E = 1$ , and all 23 mantissa bits are zero. So, the biased exponent is  $1 - 127 = -126$ , and the value is given by

$$(-1)^0 \times 1.0_2 \times 2^{-126} = \frac{1}{2^{126}} \approx 1.17549435 \times 10^{-38}.$$

The largest positive normal number is when  $S = 0$ ,  $E = 254$ , and all mantissa bits are one. In this case, the biased exponent is  $254 - 127 = 127$ , and the significant is

$$0b1.1111\ 1111\ 1111\ 1111\ 1111\ 111,$$

with value

$$1 + \sum_{k=1}^{23} \frac{1}{2} \left(\frac{1}{2}\right)^{k-1} = 1 + 1 - \frac{1}{2^{23}}.$$

Thus, the value is given by

$$(-1)^0 \times \left(2 - \frac{1}{2^{23}}\right) \times 2^{127} = 2^{128} - 2^{104} \approx 3.402823466 \times 10^{38}.$$

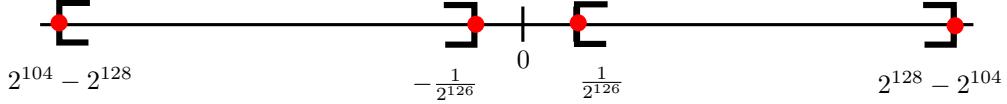
The smallest negative normal value is when  $E = 254$ ,  $S = 1$ , and all mantissa bits are one. It is the largest positive normal value multiplied by negative one, the value is

$$2^{104} - 2^{128} \approx -3.402823466 \times 10^{38}.$$

The largest negative normal value is when  $E = 1$ ,  $S = 1$ , and all mantissa bits are zero, it is

$$(-1)^1 \times 1.0_2 \times 2^{-126} = -\frac{1}{2^{126}} \approx -1.17549435 \times 10^{-38}.$$

So, the range of normalized values in the single precision system is



With

$$\begin{aligned} -2^{104} - 2^{128} &= -3.402823466 \times 10^{38} \\ -\frac{1}{2^{126}} &= -1.17549435 \times 10^{-38} \\ \frac{1}{2^{126}} &= 1.17549435 \times 10^{-38} \\ 2^{128} - 2^{104} &= 3.402823466 \times 10^{38} \end{aligned}$$

In fact, one can show quite easily that in general, the ends of the normalized range are given by

$$\pm \left( 2^{2^{n-1}} - 2^{2^{n-1}-1-|F|} \right),$$

and the inner bounds are given by

$$\pm \frac{1}{2^{2-2^{n-1}}}.$$

Where  $n$  is the number of bits in the exponent, and  $|F|$  is the number of bits in the mantissa.

- **Subnormal (denormalized):** Subnormal (or denormalized) numbers are a special class of floating-point values that allow IEEE-754 to represent numbers very close to zero with gradual loss of precision, instead of abruptly underflowing to zero.

A number is subnormal when its exponent field is all zeros ( $E = 0$ ), and its fractional field is **not** zero ( $F \neq 0$ )

For subnormals, IEEE-754 does not use the implicit leading 1, the significand begins with 0.

For subnormals, the exponent is always fixed at  $1 - \text{bias} = 1 - (2^{n-1} - 1) = 2 - 2^{n-1}$ . So if the exponent bits are all zero, the exponent is  $2 - 2^{n-1}$ , where  $n$  is the number of exponent bits. For example, in IEEE-754 single precision, if all 8 of the exponent bits are zero, the exponent is  $-126$ .

The form is therefore

$$(-1)^S \times (0.F)_2 \times 2^{2-2^{n-1}}.$$

Which is

$$(-1)^S \times (0.F)_2 \times 2^{-126}$$

for IEEE-754 single precision. Precision is reduced because the leading 1 is absent.

- **Why subnormals exist:** Subnormals exist to provide a gradual transition from the smallest normalized number to zero. Without them, the format exhibits abrupt underflow.

The smallest positive normalized number in single precision is  $2^{-126}$ . Without subnormals, any result whose magnitude is smaller than  $2^{-126}$  becomes exactly zero. This creates a discontinuity: you jump from  $2^{-126}$  directly to 0 with nothing in between.

- **Range of subnormals (IEEE-754 single precision):** The smallest positive subnormal is when  $S = 0$ ,  $f_i = 0$  except for  $f_{|F|} = 1$ , and all exponent bits zero. The value is therefore

$$(-1)^0 \times (0.0000\dots 1)_2 \times 2^{-126} = \frac{1}{2^{23}} \times \frac{1}{2^{126}} = \frac{1}{2^{149}} \approx 1.40129846 \times 10^{-45}.$$

And the largest negative subnormal is therefore

$$-\frac{1}{2^{149}} \approx -1.40129846 \times 10^{-45}.$$

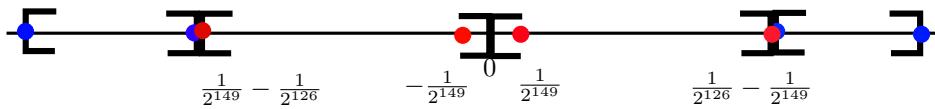
The largest positive subnormal is when  $S = 0$ ,  $f_i = 1$  for all  $i \in \{1, \dots, 23\}$ , and all exponent bits zero. The value is

$$\begin{aligned} (-1)^0 \times \sum_{k=1}^{23} \frac{1}{2} \left(\frac{1}{2}\right)^{k-1} \times 2^{-126} &= 1 - \frac{1}{2^{23}} \times 2^{-126} = \frac{1}{2^{126}} - \frac{1}{2^{149}} \\ &\approx 1.17549421 \times 10^{-38}. \end{aligned}$$

And the smallest negative subnormal is therefore

$$\frac{1}{2^{149}} - \frac{1}{2^{126}} \approx -1.17549421 \times 10^{-38}.$$

So, the range of subnormal values in the IEEE-754 single precision system is



With

$$\begin{aligned}
& - \frac{1}{2^{149}} - \frac{1}{2^{126}} \approx -1.17549421 \times 10^{-38} \\
& - - \frac{1}{2^{149}} \approx -1.40129846 \times 10^{-45} \\
& - \frac{1}{2^{149}} \approx 1.40129846 \times 10^{-45} \\
& - \frac{1}{2^{126}} - \frac{1}{2^{149}} \approx 1.17549421 \times 10^{-38}
\end{aligned}$$

Note that these lines are the floating point real lines, not the actual real lines. On the actual real line, there would be gaps between zero and the subnormals, and between the subnormals and the normals. Also, the intervals would not be continuous, as there is a gap between each representable floating point number.

One can easily show that the outer bounds of the subnormals are given by

$$\pm \left( \frac{1}{2^{2-2^{n-1}}} \right) - \frac{1}{2^{2-2^{n-1}+|F|}},$$

and the inner bounds are given by

$$\pm \frac{1}{2^{2-2^{n-1}+|F|}}.$$

- **The distance between subnormals:** The smallest positive subnormal is  $2^{-149}$ , the second smallest subnormal is  $2/2^{-149}$ , the third smallest is  $3/2^{-149}$ . In fact, the sequence is

$$x_k = k \cdot 2^{-149}, \quad k = 1, 2, \dots, 2^{23} - 1.$$

- **Representable numbers and the gaps:** A floating-point format can represent only finitely many real numbers. These values lie at specific discrete points on the real number line. Between any two representable numbers, there are infinitely many unrepresentable real numbers.

Thus, the floating-point real line looks like:

$$\cdots x_{k-1} < x_k < x_{k+1} \cdots$$

where each  $x_k$  is a representable value, and the interv

$$(x_k, x_{k+1})$$

contain an infinite number of real values that cannot be exactly stored. This is why gaps exist.

Single precision has three categories near zero:

- **Zero:** 0
- **Subnormals**

$$x_k = k \cdot 2^{-149}, \quad k = 1, 2, \dots, 2^{23} - 1.$$

- **Normal numbers:**

$$x_{minnormal} = 2^{-126}.$$

These form a sequence

$$0 < 2^{-149} < 2 \cdot 2^{-149} < \cdots < (2^{23} - 1)2^{-149} < 2^{-126}.$$

This is the complete list of representable positive values between 0 and the smallest normal.

The smallest positive representable number is:

$$x_1 = 2^{-149}.$$

Between zero and  $2^{-149}$ , there exist infinitely many real numbers that cannot be represented. Thus, the gap is

$$(0, 2^{-149}).$$

This is the uniform spacing of all subnormals.

In the context of normalized values, there is not uniform spacing between the representable values. In fact, the spacing grows as the values get very large or very small.

When a real number lies in a gap between two representable floating-point numbers, the computer cannot store it exactly. Floating-point numbers form a set like

$$\cdots < x_{k-1} < x_k < x_{k+1} \cdots .$$

All values between  $x_{k-1}$  and  $x_k$ , and all values between  $x_k$  and  $x_{k+1}$  are not representable. So if you try to store a real number

$$y \in (x_k, x_{k+1}),$$

the system must choose either  $x_k$  or  $x_{k+1}$ , depending on rounding rules.

Consider numbers near 1.0 in single precision. The spacing is

$$2^{-23} \approx 1.19 \times 10^{-7}.$$

The representable numbers are

$$1.000000000000000000000000000000, \quad 1.00000011920928955078125.$$

If we try to store 1.00000005, it lies between them, so it is rounded to

$$1.000000000000000000000000000000.$$

- **IEEE-754 rounding modes:** IEEE-754 defines four rounding modes:

1. **Round to nearest (default) (ties to even) (banker's rounding)**
2. Round toward  $+\infty$
3. Round toward  $-\infty$
4. Round toward 0

Unless you change it, the rounding mode is round to nearest, ties to even.

- **Round to nearest :** When you store a real number  $y$ , the computer finds

$$\begin{aligned} x_k &= \text{largest representable value} \leq y \\ x_{k+1} &= \text{smallest representable value} \geq y \end{aligned}$$

and selects

- $x_k$  if  $y < (x_k + x_{k+1})/2$
- $x_{k+1}$  if  $y > (x_k + x_{k+1})/2$
- The **even one** if  $y$  is exactly halfway. Uses the **binary significand rule**, also called the **even rule**.

This is why floating-point sometimes “rounds strangely”: it is trying to ensure statistical fairness.

The binary significand rule is the rule that decides how to round a number that lies exactly halfway between two representable floating-point numbers in IEEE-754's default rounding mode. When the real value is exactly halfway between two representable floating-point numbers:

IEEE-754 chooses the one whose significand (the binary digits after the implicit leading 1) has an even least significant bit (LSB).

Since a floating-point significand represents a binary integer if you ignore the binary point, we can look to the LSB of the integer. If the LSB is zero, the integer is even. If the LSB is one, the integer is odd. IEEE-754 chooses the representable number with an even LSB.

We remark that no two adjacent floats have the same LSB in the significand. This is due to the way binary counting works.

- **ULP:** ULP stands for Unit in the Last Place. It is the spacing between two adjacent representable floating-point numbers at a given magnitude.

$$\text{ULP}(x) = \text{distance between the two nearest floats around } x.$$

ULP is the fundamental measure of floating-point resolution. Floating-point numbers are not evenly spaced on the real line.

- **Near 0 (subnormals), spacing is constant:**  $2^{-149}$
- **Near 1, spacing is:**  $2^{-23}$
- **Near 8, spacing is:**  $2^{-20}$
- **Near  $2^{127}$ , spacing is:**  $2^{104}$

These are all ULP sizes at different scales. ULP is the change in last bit of the significand. In IEEE-754 single precision, the significand has 23 explicitly stored bits, but with the hidden bit (implicit one), precision is effectively 24 bits.

Incrementing the least significant bit of the significand changes the value by

$$\text{ULP} = 2^{E-23}.$$

Thus, the spacing depends on the unbiased exponent ( $E_S - 127$ ). Consider a fixed unbiased exponent  $E = E_S - 127$ , and a float encoded with this exponent,

$$(1.F) \times 2^E.$$

The smallest change in the significand is  $2^{-23}$ . Let  $x = 1.F \times 2^E$ , and  $x_{\text{next}} = (1.F + 2^{-23}) \times 2^E$ . The difference is

$$\begin{aligned} \text{ULP} &= x_{\text{next}} - x = \left(1.F + \frac{1}{2^{23}}\right) \times 2^E - 1.F \times 2^E \\ &= 1.F \times 2^E + \frac{2^E}{2^{23}} - 1.F \times 2^E = \frac{2^E}{2^{23}} = 2^{E-23} = 2^{E_S-127-23} = 2^{E_S-150}. \end{aligned}$$

For example, floats near 1.0 have  $E = 0$ , where  $E_S = 127$ , so

$$\text{ULP}(1.0) = 2^{127-127-23} = 2^{-23} \approx 1.192092896 \times 10^{-7}.$$

- **Binade:** a binade is a set of numbers in a binary floating-point format that all have the same sign and exponent. In other words, a binade is the interval  $[2^e, 2^{e+1})$  or  $(-2^{e+1}, -2^e]$  for some integer value  $e$ , that is, the set of real numbers or floating-point numbers  $x$  of the same sign such that  $2^e \leq |x| < 2^{e+1}$

In other words, A **binade** is the set of all **normalized** floating-point numbers with the same exponent field value (i.e., same unbiased exponent). For a given exponent  $e$ , all numbers of the form

$$m \cdot 2^e, \quad 1 \leq m < 2$$

belong to one binade. Since there are 254 exponents in IEEE-754 single precision that yield normalized numbers, there are 254 binades.

Every binade has spacing  $2^{e-(p-1)}$ , where  $p$  is the precision.

- **Round to nearest with bits:** When performing floating-point operations (or converting from a real number to IEEE format), the true result often has more precision than the format allows (e.g., more than 23 bits in single precision). So, IEEE-754 computes extra bits beyond the stored mantissa:

- **Guard bit (G):** 1 extra bit
- **Round bit (R):** 1 more bit
- **Sticky bit (S):** 1 bit representing all remaining discarded bits

Suppose your mantissa is limited to  $n$  bits, but a calculation produces:

$$[n \text{ stored bits}][G][R][S].$$

Where

- **Stored fraction (n bits):** These are the bits that will appear in the IEEE float.
- **Guard bit (G):** The first bit beyond the stored LSB.
- **Round bit (R):** The next bit after the guard bit.
- **Sticky bit (S):** This bit is:
  - \* 1 if any bits beyond R were 1
  - \* 0 if all remaining bits were 0

So the sticky bit is an OR-accumulator of all remaining bits.

The guard bit is  $2^{-24}$ , with magnitude  $2^{-24} \cdot 2^E = 2^{E-24}$ . In terms of the ULP,

$$2^{E-24} = \frac{1}{2} \cdot 2^{E-23} = \frac{1}{2} \text{ULP}.$$

So, the gaurd bit indicates whether the result is at least  $\frac{1}{2}$ ULP past the stored float.  $G = 1$  indicates that the stored float is at least halfway between two representable floats.

Similarly, the round bit R has magnitude  $2^{-25} \cdot 2^E = 2^{E-25} = \frac{1}{4}$ ULP. Thus, R combined with G tells us if we are strictly more than halfway past the midpoint. R combined with the sticky bit S helps distinguish

- exactly half
- slightly more than half
- much more than half

The sticky bit S is the or of all bits past R, the remaining discarded bits.

To see if we need to round down, all we need is G

- **Round down (truncate):** If  $G = 0$ , we are less than halfway between representable floats
- If  $G = 1$  we are at least halfway between representable floats, we look to R and S
  - **Ties to even, choose the one whose mantissa has an even least significant bit (LSB):** If  $G = 1$ , and  $R = S = 0$ , then we are exactly halfway between representable floats
  - **Round up:** If  $G = 1$ ,  $R = 0$ , but  $S = 1$ , we are slightly more than halfway between representable floats
  - **Round up:** If  $G = 1$ ,  $R = 1$ , and  $S = 0$  or  $S = 1$ , we are way past halfway between representable floats
- **Rounding examples:** Suppose we have  $0b.0010101$ , and we want to round to three binary places. So  $G = 0$ ,  $R = 1$ ,  $S = 1$ . Since  $G = 0$ , we round down (truncate). So,

$$0b.0010101 = 0b.001.$$

Next, suppose that we have  $0b.0011101$ . In this case,  $G = 1$ ,  $R = 1$ ,  $S = 1$ , so we round up. With three bits, this number is between

$$(0b.001, 0b.010).$$

So, we round up to  $0b.010$ . Thus,

$$0b.0011101 = 0b.010.$$

Next, consider  $0b.0011000$ . In this case,  $G = 1$ ,  $R = S = 0$ . So, we are exactly halfway between representable values, halfway between

$$(0b.001, 0b.010).$$

We choose the one that is even (lsb=0). So, we choose  $0b.010$ , and

$$0b.0011000 = 0b.010.$$

Finally, consider  $0b.0101000$ . In this case,  $G = 1$ , and  $R = S = 0$ . So, we are exactly halfway between

$$(0b.010, 0b.011).$$

Since  $0b.010$  is the even one,

$$0b.0101000 = 0b.010.$$

- **Roundoff error:** Roundoff error (or rounding error) is exactly what happens when a real number cannot be represented in floating-point, so it is rounded to the nearest representable value.

When a real number  $x$  is rounded to a floating-point number  $\hat{x}$

$$\hat{x} = \text{round}(x),$$

the difference  $\hat{x} - x$  is the roundoff error.

- **Representing zero:** Zero is represented when both fields are all zeros:

- **Sign bit:** 0 or 1
- **Exponent field:** 00000000
- **Fraction field:** 00000000000000000000000000000000

So the bit patterns are:

- +0:

$$0\ 00000000\ 00000000000000000000000000000000.$$

- -0:

$$1\ 00000000\ 00000000000000000000000000000000.$$

Thus, the representation is

$$(-1)^S \times 0.$$

- **Real to encoding:** Consider  $x \in \mathbb{R}$ , to convert this number to its IEEE-754 single precision encoding we need to determine the sign, mantissa, and exponent.

First, we determine the sign bit.

$$S = \begin{cases} 0 & \text{if } x \geq 0 \\ 1 & \text{if } x < 0 \end{cases}.$$

Next, we convert the number to binary (normalized), we need to express  $x$  as

$$|x| = (1.F)_2 \times 2^E.$$

Where the exponent  $E = E_{\text{stored}} - 127$ . Note that  $E_{\text{stored}} = E + 127$ , the exponent we store is an unsigned 8-bit integer.

We convert the number to binary, and move the binary point so that the number is between 1 and 2.

$$1 \leq (1.F) < 2.$$

This gives us  $|F|$  binary digits after the leading 1, and  $E$ , our unbiased exponent. The single precision bias uses bias = 127, so

$$E_{\text{stored}} = E + 127.$$

This yields an 8-bit exponent. The fraction field is just the bits of  $F$

$$F = f_1 f_2 f_3 \dots f_{23}.$$

If  $F$  has fewer than 23 bits, pad with zeros. If more than 23 bits, round according to IEEE-754 rounding mode (usually round-to-nearest-even).

Then, we assemble the final 32-bit pattern

$$x = S\ E_{\text{stored}}\ F.$$

This is the bitstring you store.

The algorithm is

1. Determine sign bit
2. Convert real number to normalized binary scientific notation  $1.F \times 2^E$
3. Bias the exponent  $E + 127$ , this is the value we store as the exponent.
4. Store the 23 fraction bits, round if needed. Remember that the leading one is implicit, not stored.
5. Assemble the final 32-bit pattern

For example, consider  $x = 0.15625$ . First, we determine that the sign bit is 0, and convert to binary.

$$0.15625 = 0b.00101.$$

We normalize by moving the binary point three places to the right, so we have  $0b1.01 \times 2^{-3}$ . Thus,  $F = 01$ , and  $E = -3$ . Since  $E = -3$ ,  $E_{\text{stored}} = -3 + 127 = 124 = 0b0111 1100$ .

So  $S = 0$ ,  $E_s = 01111100$ , and  $F = 01000000000000000000000000000000$ . Notice the number of bits in each field is precisely the amount required for 32 bit single precision floats, we pad with zeros to make sure of this. With all of our bits acquired, our encoded float is

$$00111110001000000000000000000000000000000.$$

- **Real to encoding with rounding:** Consider  $x = 0.1$ , which cannot be represented exactly with binary, converting to binary gives

$$000\ 1100\ 1100\ 1100\ 1100\ 1100....$$

After converting to binary scientific notation with normalization, we have

$$1.100\ 1100\ 1100\ 1100\ 1100... \times 2^{-4}.$$

So, we have  $E = -4$ ,  $E_S = -4 + 127 = 123 = 0111 1011$ . From  $1.F$ , the mantissa is

$$F \approx 100\ 1100\ 1100\ 1100\ 1100....$$

But, we need 23 bits, so we continue the pattern to get the next 4 bits. So,

$$100\ 1100\ 1100\ 1100\ 1100\ 1100....$$

Since the pattern continues, we round. In the default rounding mode, *round to nearest (ties to even)* the next bits are 11 (nonzero), so we round the 23rd bit (0) to 1. Thus, there are 23 mantissa bits are

$$F = 100\ 1100\ 1100\ 1100\ 1100\ 1101.$$

Thus, the final encoding is

$$0100\ 1100\ 1100\ 1100\ 1100\ 1101\ 0111\ 1011.$$

**Note:** Using  $(-1)^S \times (1.F)_2 \times 2^{E-127}$ , this encoding has value

$$0.1000000015.$$

The representable floats that 0.1 falls between are when  $E = -4$ ,  $F_k = 100\ 1100\ 1100\ 1100\ 1100$ , and  $F_{k+1} = 100\ 1100\ 1100\ 1100\ 1100\ 1101$ . The values are

$$(x_k, x_{k+1}) = (.099999994, 0.10000000149011612).$$

Since the midpoint is

$$\frac{x_k + x_{k+1}}{2} = 0.09999999776482582,$$

and  $0.1 > 0.09999999776482582$ , 0.1 was rounded to  $x_{k+1} = 0.10000000149011612 \approx 0.1000000015$

- **Precision:** When we say single precision (FP32) has 24 bits of significand precision, that number tells you how close two representable floating-point numbers can be, and therefore how many decimal digits you can reliably distinguish. Precision means the smallest relative difference the format can detect. And that is determined directly by how many bits are in the significand.

A normalized binary significand is IEEE-754 single precision is

$$1.f_1f_2f_3\dots f_{23}.$$

This is 24 bits. This represents a binary number between

$$1.000\dots 000 \quad \text{and} \quad 1.111\dots 111.$$

The smallest step between two distinct representable significands is  $2^{-23}$ , because the LSB of the significand has weight  $2^{-23}$ .

Normalized floats are of the form

$$(1.F)2^E.$$

And the spacing between representable numbers for a given exponent is given by the ULP

$$\text{ULP} = 2^{E-23} = 2^{E_S-150}.$$

The ULP describes absolute spacing, but absolute spacing grows as numbers get bigger. So absolute spacing does not tell you how much precision you really have. To understand precision, you must compare the spacing to the value itself.

$$\frac{\text{ULP}(x)}{x} = \frac{2^{E-23}}{(1.F)2^E} == \frac{1}{(1.F)2^E}.$$

But,  $1.F$  is always between 1.0 and 2.0, so

$$\frac{1}{2^{23}} \leq \frac{\text{ULP}(x)}{x} \leq \frac{1}{2^{24}}.$$

So, relative precision is always about  $10^{-7}$ . Relative precision is nearly constant for all normalized floats.

Thus, 24 bits in the significand gives a relative error bounded above by  $2^{-24}$ , which is roughly  $10^{-7}$ . So, we have  $\sim 7$  digits of accuracy.

We can also ask how many decimal digits do we need if we have 24 bits to match the precision. So,

$$10^D = 2^{24} \implies \log_{10}(2^{24}) = D \implies D = 24 \log_{10}(2) = 7.22471989594.$$

In double precision we have 54 significand bits, so

$$10^D = 2^{54} \implies D = 54 \log_{10}(2) = 16.2556197659.$$

So, about 16 digits of accuracy in double precision, and 7 digits of accuracy in single precision.

It guarantees only that a float can distinguish numbers that differ in their first 7 digits. Whether a decimal number is representable exactly depends on its binary expansion

If two real numbers differ within their first 7 digits, a float can tell them apart. If they differ only in the 8th digit, a float will round them to the same value.

- **Which numbers can be represented:** Consider the values  $(1.F) \times 2^E$ .  $1.F$  is

$$1 + \left( \frac{1}{2^i} + \frac{1}{2^j} + \dots + \frac{1}{2^k} \right).$$

Where  $i < j < \dots < k$ , we can factor out a  $\frac{1}{2^k}$ ,

$$1 + \frac{1}{2^k} \left( \frac{2^k}{2^i} + \frac{2^k}{2^j} + \dots + 1 \right) = 1 + \frac{1}{2^k} (2^{k-i} + 2^{k-j} + \dots + 1).$$

Notice that  $2^{k-i}, 2^{k-j} > 0$ , so  $2^{k-i} + 2^{k-j} + \dots + 1 \in \mathbb{Z}$ , call this integer  $\ell$ . We have

$$1 + \frac{\ell}{2^k} = \frac{2^k + \ell}{2^k}.$$

Notice that  $2^k + \ell$  is still an integer, call this value  $s$ . So,

$$1.F = \frac{s}{2^k} \quad s \in \mathbb{Z}.$$

Now, we multiply by  $2^E$ . If  $E \geq 0$ ,  $2^E \geq 1$ , so

$$(1.F) \times 2^E = \frac{s \times 2^E}{2^k}.$$

Notice that  $s \times 2^E \in \mathbb{Z}$ , call this value  $m$ . So,

$$(1.F) \times 2^E = \frac{m}{2^k} \quad m \in \mathbb{Z}.$$

Now, if  $E < 0$ , then  $2^E < 0$ , so

$$1.F \times 2^E = \frac{s}{2^k \cdot 2^E} = \frac{s}{2^{k+E}}.$$

In either case, our value is  $\frac{m}{2^k}$ , where  $m$  is some integer. That is, only values that can be expressed as an integer divided by a power of two can be represented exactly using floating point. Furthermore, the number of digits must be less than or equal to 7 in single precision, and less than or equal to 16 in double precision.

- **Infinity and NaN:** We have one more exponent to cover, when  $E_S = 255 = 0b11111111$ , so  $E = 255 - 127 = 128$ . This case is reserved to define two special values, either  $\pm\infty$ , or NaN (not a number).

- $\pm\infty$ : Significand equals zero
- **NaN**: Significand different from zero

**Note:** The sign bit of a NaN has no significance, it can be ignored.

- **NaN:** “not a number” is a special value that some computers use to indicate an error

For example, if a compiler initializes memory to NaN (instead of 0 or not initializing at all), then it can tell when you are using a variable that hasn’t been assigned a value yet.

You can do calculations with infinity and NaN. This can prevent your program from terminating abnormally. Of course, you will still have to debug it once you see infinity or NaN in the output.

- **Double precision bias:** The IEEE double precision standard has an 11-bit exponent, which means its bias is 1023. This is called excess-1023 representation. Everything works the same in double precision, except for minor differences in some numbers.
- **Range of values:** Consider IEEE-754 single precision. The largest real number is when  $S = 0$ ,  $E_S = 254$ , and all  $F$  bits on. The value is

$$L = \left(1 + \sum_{k=1}^{23} \frac{1}{2} \left(\frac{1}{2}\right)^{k-1}\right) 2^{254-127} = \left(1 + 1 - \frac{1}{2^{23}}\right) 2^{127} = \left(2 - \frac{1}{2^{23}}\right) 2^{127} = 2^{128} - 2^{104}.$$

Whereas the smallest number is therefore

$$S = -L = -(2^{128} - 2^{104}) = 2^{104} - 2^{128}.$$

Thus, the range is

$$L - S = 2^{128} - 2^{104} - (2^{104} - 2^{128}) = 2^{128} - 2 \cdot 2^{104} + 2^{128} = 2^{129} - 2^{105}.$$

- **Counting the number of representable values:** Consider the single precision standard. For normalized numbers, the biased exponent  $E \in \{1, 254\}$ . For each exponent, there are  $2 \cdot 2^{23} = 2^{24}$  bit patterns for the sign and significand. Thus, there are

$$2^{24} \cdot 254$$

normalized numbers. When the exponent is zero, the number is subnormal. Excluding when the significand is all zero, we have

$$2(2^{23} - 1) = 2^{24} - 2$$

Subnormals. Regarding the representable real numbers (zeros + subnormals + normalized), there are

$$2^{24} \cdot 254 + 2^{24} - 2 + 2 = 2^{24}(254 + 1) = 2^{24}(255) = 510 \cdot 2^{23}.$$

When the exponent is 255 (all ones), there are two infinities. If we include the infinites to the count of the real numbers, we have

$$510 \cdot 2^{23} + 2.$$

If we include NaN,

$$510 \cdot 2^{23} + 3.$$

Finally, there are  $2^{32}$  bit patterns.

- **Overflow and underflow:** Floating-point numbers have finite range and finite precision. Because of this, some values are too large or too small to represent.

- **Overflow:** Overflow occurs when the magnitude of the true mathematical result exceeds the largest representable finite floating-point number. In IEEE-754 single precision, the largest finite value is

$$2^{128} - 2^{104}.$$

If a computation produces a result larger than this (in magnitude), the format cannot encode it. When this happens, the result is replaced by  $\pm\infty$ , depending on the sign of the result. Then, an overflow flag is raised.

- **Underflow:** Underflow is more subtle than overflow because IEEE-754 allows two regimes near zero.
  - (a) **Tiny result:** A result is tiny if its magnitude is smaller than the smallest normal floating-point number. For single precision, the smallest normal value is

$$2^{-126}.$$

A value smaller in magnitude than this is subnormal territory.

- (b) **Underflow event:** Underflow occurs when a tiny result cannot be represented exactly in the subnormal range after rounding. If the tiny result is representable as a subnormal, no underflow event. If rounding forces the tiny result to zero, underflow event.

So underflow is not “entering the subnormal region”; it is failing to represent the tiny result accurately even as a subnormal.

- **Machine epsilon:** Machine epsilon is the distance between 1 and the next larger representable floating-point number. Formally,

$$\varepsilon_{\text{mach}} = \min\{\varepsilon > 0 : \text{fl}(1 + \varepsilon) > 1\}.$$

It is therefore a spacing property of the floating-point format itself, not a property of rounding.

The machine epsilon is given by

$$\varepsilon = 2^{-(p-1)},$$

where  $p$  is the significand precision. In IEEE-754 single precision, the significand precision is 24 bits, so

$$\varepsilon_{\text{mach}} = 2^{-(24-1)} = 2^{-23} \approx 1.19 \times 10^{-7}.$$

This is the gap between 1.0 and the next representable float above 1.0.

The machine epsilon tells us

- How finely the real line can be resolved around the value 1.
- The ULP (unit in the last place) size at 1.

- **Unit roundoff:** Unit roundoff (sometimes denoted  $u$ ) is the maximum relative error introduced by rounding a real number to the nearest floating-point number. Formally,

$$u = \frac{1}{2}\varepsilon_{\text{mach}} = 2^{-p}.$$

For rounding-to-nearest, the IEEE-754 default, this is the correct bound. For single precision,

$$u = 2^{-24} \approx 5.96 \times 10^{-8}.$$

- **ULP is proportional to magnitude:** Consider a floating point number

$$z = m \cdot 2^e, \quad 1 \leq m < 2.$$

Recall that

$$\text{ulp}(z) = 2^{e-p}.$$

If we look at the ratio  $\text{ulp}(z)/|z|$ , we see

$$\frac{\text{ulp}(z)}{z} = \frac{2^{e-p}}{m2^e} = \frac{2^e \cdot 2^{-p}}{m2^e} = \frac{2^{-p}}{m}.$$

But,  $2^{-p}$  is the unit roundoff  $u$ , so

$$\frac{\text{ulp}(z)}{|z|} = \frac{u}{m}.$$

Thus,

$$\text{ulp}(z) = \frac{u}{m} |z|, \quad 1 \leq m < 2.$$

This is the relationship between the spacing of normalized floats around  $z$ , and the magnitude of  $z$ . However, because  $1 \leq m < 2$ , we have

$$\frac{1}{2} < \frac{1}{m} \leq 1.$$

So,

$$\frac{1}{2}u|z| < \text{ulp}(z) \leq u|z|.$$

$\text{ulp}(z)$  is the absolute error, with the above fact, we can say that the relative error is bounded by

$$\frac{1}{2}u < \frac{\text{ulp}(z)}{|z|} \leq u,$$

and has value

$$\frac{\text{ulp}(z)}{|z|} = \frac{1}{m} \cdot u.$$

Informally, we can say

$$\text{ulp}(z) \approx u|z|.$$

- **Floating point arithmetic:** For  $x, y \in \mathbb{R}$ ,

$$\text{fl}(x \text{ op } y) = (x \text{ op } y)(1 + \delta), \quad |\delta| \leq u.$$

Where  $u$  is the unit roundoff.

Consider  $z \in \mathbb{R}$ ,

$$\text{fl}(z) = z + \varepsilon, \quad |\varepsilon| \leq \frac{1}{2} \text{ulp}(z) \approx u |z|.$$

We can convert this into relative error by dividing by  $z$ , so

$$\frac{\text{fl}(z)}{z} = 1 + \frac{\varepsilon}{z} \quad \left| \frac{\varepsilon}{z} \right| \leq u.$$

Let  $\delta := \frac{\varepsilon}{z}$ , so  $|\delta| \leq u$ . Then,

$$\frac{\text{fl}(z)}{z} = 1 + \delta, \quad |\delta| \leq u.$$

Thus,

$$\text{fl}(z) = z(1 + \delta), \quad |\delta| \leq u.$$

In any expression involving real numbers with arithmetic operators, IEEE will first perform that operation, and then round the result to the nearest representable float. So,  $z = x \circ y$ , where  $\circ \in \{+, -, *, /, \dots\}$ . This is why we say that

$$\text{fl}(x \circ y) = (x \circ y)(1 + \delta), \quad |\delta| \leq u.$$

## 6.4 Chapter 4: Overview of computer architecture

### 6.4.1 The Von Neumann Model

- **Introduction:** On the ENIAC, all programming was done at the digital logic level. Programming the computer involved moving plugs and wires. A different hardware configuration was needed to solve every unique problem type.

Configuring the ENIAC to solve a simple problem required many days of labor by skilled technicians

Inventors of the ENIAC, John Mauchley and J. Presper Eckert, conceived of a computer that could store instructions in memory. The invention of this idea has since been ascribed to a mathematician, John von Neumann, who was a contemporary of Mauchley and Eckert.

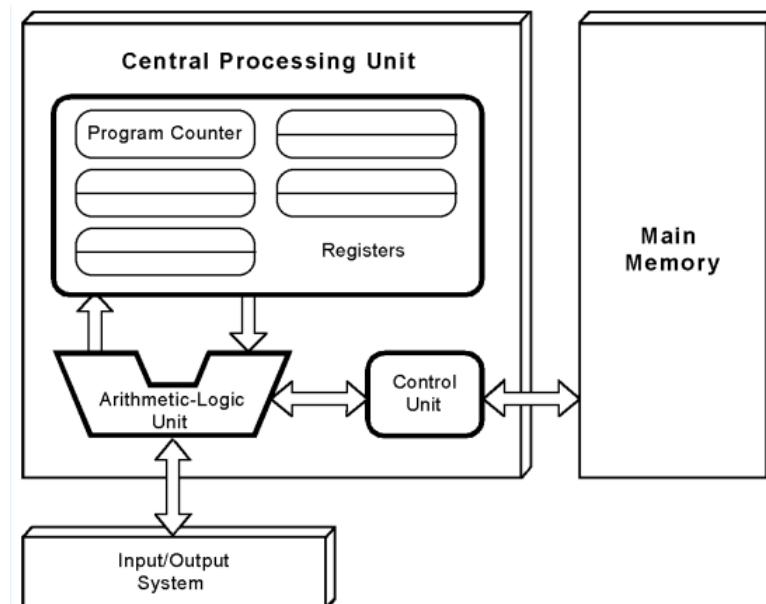
Stored-program computers have become known as von Neumann architecture systems

- **Characteristics of a stored program computer:** Today's stored-program computers have the following characteristics:

- Three hardware systems:
  1. A central processing unit (CPU)
  2. A main memory system
  3. An I/O system
- The capacity to carry out sequential instruction processing.
- A single data path between the CPU and main memory.

**Note:** This single path is known as the von Neumann bottleneck.

- **The von Neumann model:** This is a general depiction of a von Neumann system:



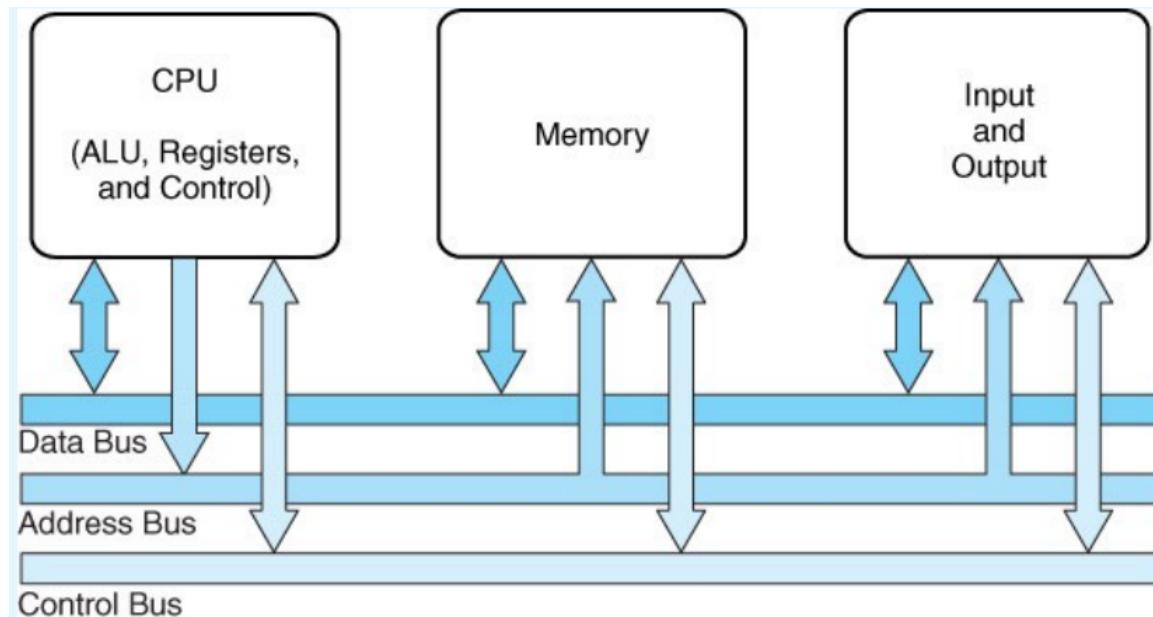
These computers employ a fetch-decode-execute cycle to run programs

The control unit fetches the next instruction from memory using the program counter to determine where the instruction is located. The instruction is decoded into a language that the ALU can understand.

Data operands required to execute the instruction are fetched from memory and placed into registers in the CPU.

The ALU executes the instruction and places results in registers or memory.

- **Buses:** Buses allow data to flow between components.



As we see, there are three buses

1. Data bus
2. Address bus
3. Control bus

- **Modern improvements:** Conventional stored-program computers have undergone many incremental improvements over the years. These improvements include adding specialized buses, floating-point units, and cache memories, to name only a few.

But, enormous improvements in computational power require departure from the classic von Neumann architecture. Adding processors is one approach.

In the late 1960s, high-performance computer systems were equipped with dual processors to increase computational throughput. In the 1970s supercomputer systems were introduced with 32 processors.

Supercomputers with 1,000 processors were built in the 1980s. In 1999, IBM announced its Blue Gene system containing over 1 million processors.

#### 6.4.2 CPU basics and the Bus

- **CPU basics:** The computer's CPU fetches, decodes, and executes program instructions
- **Principal parts of the CPU:** The two principal parts of the CPU are the datapath and the control unit
  1. The datapath consists of an arithmetic-logic unit (ALU) and storage units (registers) that are connected by a data bus that is also connected to main memory.
  2. Various CPU components perform sequenced operations according to signals provided by its control unit.
- **CPU basics (2):** Registers hold data that can be readily accessed by the CPU. They can be implemented using D flip-flops

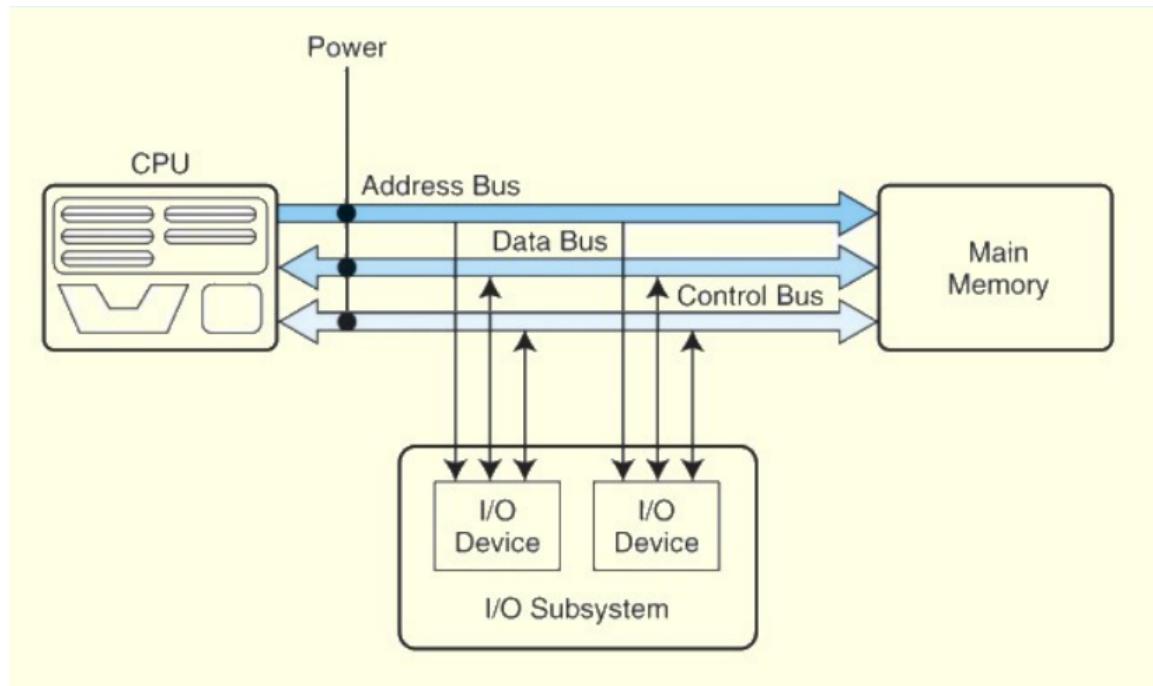
The arithmetic-logic unit (ALU) carries out logical and arithmetic operations as directed by the control unit.

The control unit determines which actions to carry out according to the values in a program counter register and a status register

- **The Bus:** The CPU shares data with other system components by way of a data bus. A bus is a set of wires that simultaneously convey a single bit along each line.
- **Two types of buses:** Two types of buses are commonly found in computer systems: point-to-point, and multipoint buses
- **Data, control, and address lines:** Buses consist of data lines, control lines, and address lines.
  - Data lines convey bits from one device to another
  - Control lines determine the direction of data flow, and when each device can access the bus.
  - Address lines determine the location of the source or destination of the data.
- **Address bus:** The address bus is used by the CPU to send addresses to the main memory for lookup
- **Data bus:** Memory uses the data bus to send back data from the desired address. The CPU also uses the data bus to send data to memory for updating the value at an address.

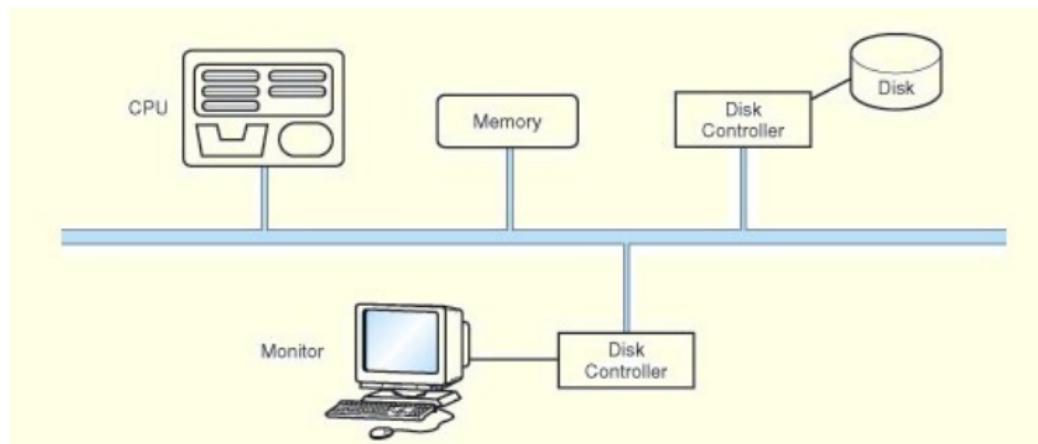
**Note:** The CPU has a similar relationship to the I/O subsystem. All devices need the control bus.

- **The bus (figure):**



- **Multipoint bus:** A multipoint bus is shown below.

Because a multipoint bus is a shared resource, access to it is controlled through protocols, which are built into the hardware.



- **Bus Arbitration:** What happens when more than one device wants to use the bus at the same time? *Arbitration* refers to the hardware algorithm (protocol) for resolving conflicts.
  - **Daisy chain:** Permissions are passed from the highest priority device to the lowest.
  - **Centralized parallel:** Each device is directly connected to an arbitration circuit.

- **Distributed using self-detection:** Devices decide which gets the bus among themselves.
- **Distributed using collision-detection:** Any device can try to use the bus. If its data collides with the data of another device, it tries again.

### 6.4.3 Marie

- **Intro:** We can now bring together many of the ideas that we have discussed to this point using a very simple model computer

Our model computer, the Machine Architecture that is Really Intuitive and Easy, MARIE, was designed to illustrate basic architecture concepts.

While this system is too simple to do anything useful in the real world, understanding its functions will make it easier to understand more complex system architectures.

- **Characteristics:** The MARIE architecture has the following characteristics:

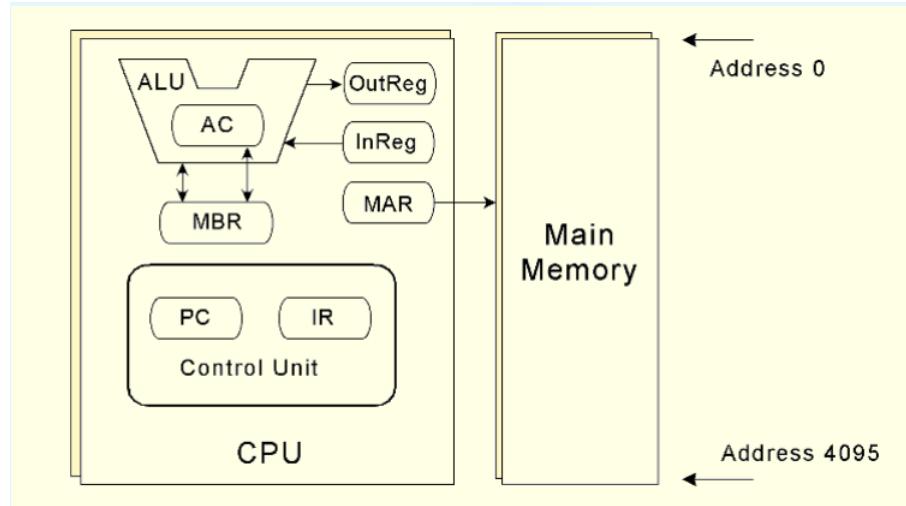
1. Binary, two's complement data representation.
2. Stored program, fixed word length data and instructions.
3. 4K words of word-addressable main memory.
4. 16-bit data words.
5. 16-bit instructions, 4 for the opcode and 12 for the address.
6. 12-bit addresses
7. A 16-bit arithmetic logic unit (ALU).
8. Seven registers for control and data movement.

**Note:** There are no general purpose registers in the Marie model. There are seven special (named) registers

- **Seven registers:**

1. **Accumulator, AC:** a 16-bit register that holds a conditional operator (e.g., "less than") or one operand of a two-operand instruction.
2. **Memory buffer register, MBR:** a 16-bit register that holds the data after its retrieval from, or before its placement in memory.
3. **Instruction register, IR:** a 16-bit register which holds an instruction immediately preceding its execution.
4. **Memory address register, MAR:** a 12-bit register that holds the memory address of an instruction or the operand of an instruction.
5. **Program counter, PC:** a 12-bit register that holds the address of the next program instruction to be executed
6. **Input register, InREG:** an 8-bit register that holds data read from an input device.
7. **Output register, OutREG:** an 8-bit register, that holds data that is ready for the output device.

- **Marie architecture graphic**



The registers are interconnected, and connected with main memory through a common data bus.

Each device on the bus is identified by a unique number that is set on the control lines whenever that device is required to carry out an operation.

Separate connections are also provided between the accumulator and the memory buffer register, and the ALU and the accumulator and memory buffer register.

This permits data transfer between these devices without use of the main data bus.

- **Bus addresses:**

The bus address refers to the address lines on the address bus - the part of the bus system that carries the memory address from the CPU to main memory.

MARIE uses a single shared 16-bit bus for both data and addresses (time-multiplexed). That means:

- The same 16 wires carry either addresses or data at different times.
- Control signals decide what's currently being sent.

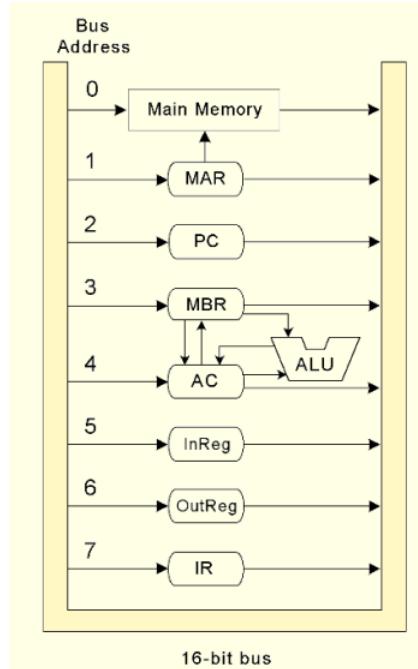
Each "Bus Address" (0–7) identifies which device (register or memory) is connected to that bus line.

When one component needs to send or receive data:

- The Control Unit activates that device's bus address line.
- That component either places data on the bus (output) or reads from it (input).

So "Bus Address 3" doesn't mean memory address 3 - it means "the bus line connected to the MBR."

- **Marie data path:**



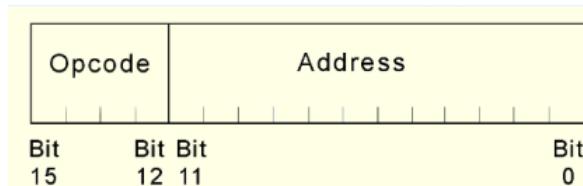
All the bits of the control bus enter and leave each component. The bus address on the diagram shows which bit configuration needs to be represented (e.g., 1 = 001, 2 = 010, etc.) in order for that component to know that it is being spoken to.

- **Marie instruction set (ISA):** A computer's instruction set architecture (ISA) specifies the format of its instructions and the primitive operations that the machine can perform.

The ISA is an interface between a computer's hardware and its software. Some ISAs include hundreds of different instructions for processing data and controlling program execution.

The MARIE ISA has only thirteen instructions

- **Instruction format:**



- **Fundamental instructions:**

Instruction Number			
Binary	Hex	Instruction	Meaning
0001	1	Load X	Load contents of address X into AC.
0010	2	Store X	Store the contents of AC at address X.
0011	3	Add X	Add the contents of address X to AC.
0100	4	Subt X	Subtract the contents of address X from AC.
0101	5	Input	Input a value from the keyboard into AC.
0110	6	Output	Output the value in AC to the display.
0111	7	Halt	Terminate program.
1000	8	Skipcond	Skip next instruction on condition.
1001	9	Jump X	Load the value of X into PC.

- **Labels in Marie:** In MARIE assembly, a label is used to mark a memory address so you can refer to it by name instead of by number. You define a label like this:

```
o  LOOP,    LOAD    X
```

The comma tells the assembler: "This is a label that identifies this memory address." The comma itself does not occupy memory and is not executed - it's purely syntactic.

We have three types of labels

1. Pure
2. Data
3. Subroutine

- **Defining storage in Marie:** In MARIE, you define storage (memory locations for data) at the end of your program using assembler directives like DEC or HEX

```
o  LABEL,  DEC  value
```

- **Marie's characteristics:** Marie has
  - One accumulator (AC)
  - No stack
  - No registers other than AC, MAR, MBR, IR, PC
  - No call frame / local scope
- **Pure labels:** A pure label is simply a name that marks a memory address but does not define data

A pure label is just a symbolic name used to refer to a specific location in your program, such as a jump target.

```
o  START,  LOAD    X
```

START, is a pure label - it marks the address of the first instruction (LOAD X).

- **Marie RTL:** Each of our instructions actually consists of a sequence of smaller instructions called **microoperations**.

The exact sequence of microoperations that are carried out by an instruction can be specified using register transfer language (RTL)

In the MARIE RTL, we use the notation  $M[X]$  to indicate the actual data value stored in memory location  $X$ , and  $\leftarrow$  to indicate the transfer of bytes to a register or memory location

- **Skipcond:** Skipcond allows us to skip the next instruction bases on the value of the accumulator. Recall that if a Marie instruction requires an operand, the rightmost 12 bits are the operand. In Skipcond, if we allowed say an address as an operand, then there would be no room for the condition. Thus, we do not have an address as an operand, instead we decide to use bits 10 and 11 as the condition. If the rightmost 12 bits of the skipcond operand are

$$b_{11}b_{10}b_9b_8 \dots b_1b_0$$

then we evaluate skipcond as follows

- If  $b_{11}b_{10} = 00$ , skip the next instruction if the  $AC$  is negative. (**skipcond 000**)
- If  $b_{11}b_{10} = 01$ , skip the next instruction if the  $AC$  is zero. (**skipcond 400**)
- If  $b_{11}b_{10} = 10$ , skip the next instruction if the  $AC$  is positive. (**skipcond 800**)

So, if we wanted to skip the next instruction if the  $AC$  is zero, our instruction would be

1000 0100 0000 0000.

Which, is

8400

in Hex, or just **skipcond 400**.

- **Looping with branch and skipcond:** Suppose we want to repeat a block of code five times, the general structure is

0	LOAD	five
1	LOOP	...
2	.	.
3	.	.
4	.	.
5	SUBT	one
6	SKIPCOND	400
7	JUMP	LOOP
8	JUMP	END
9	END	...
10	.	.
11	.	.
12	.	.

Provided that five and one are defined in memory.

- **If else statement:** Suppose we want to write an if ( $x$  AND  $y$ ) statement, we could do

```
0      LOAD      x
1      SKIPCOND 400      / skip next if x == 0
2      JUMP      CHECK_Y / x != 0, check y
3      JUMP      ELSE     / x == 0, skip entire if
4
5      CHECK_Y, LOAD      y
6      SKIPCOND 400      / skip next if y == 0
7      JUMP      DO_IF    / both x and y nonzero
8      JUMP      ELSE     / y == 0, skip if body
9
10     DO_IF,   ...        / body of the if(x and y)
11     .
12     .
13     .
14
15     JUMP      FI
16     ELSE,   ...
17     .
18     .
19     .
20
21     FI
```

- **If ( $x + y$ ) else:**

```
0      LOAD      x
1      SKIPCOND 400
2      JUMP      DO_IF
3
4      CHECK_Y, LOAD      y
5      SKIPCOND 400
6      JUMP      DO_IF
7      JUMP      ELSE
8
9      DO_IF,   ...
10     .
11     .
12     .
13
14     JUMP      FI
15     ELSE,   ...
16     .
17     .
18     .
19
20     FI
```

- **If elseif, else:** Suppose we have some value  $x$ , a mystery value, and we want to write if ( $x == 0$ ) else if ( $x < 0$ ) else

```

0  if (x == 0) {
1    ...
2  } else if (x < 0) {
3    ...
4  } else {
5    ...
6 }

```

In Marie, we could write

```

0      LOAD      x
1      SKIPCOND 400
2      JUMP      ELSEIF
3  DO_IF    ...
4  .
5  .
6  .
7      JUMP      FI
8  ELSEIF  SKIPCOND 000
9      JUMP      ELSE
10 DO_ELIF ...
11 .
12 .
13 .
14      JUMP      FI
15 ELSE    ...
16 .
17 .
18 .
19 FI      ...

```

- **RTL for LOAD:** The RTL for the LOAD instruction is:

$$\begin{aligned} \text{MAR} &\leftarrow X \\ \text{MBR} &\leftarrow M[\text{MAR}] \\ \text{AC} &\leftarrow \text{MBR}. \end{aligned}$$

- **RTL for STORE  $X$ :** The RTL for the STORE instruction is

$$\begin{aligned} \text{MAR} &\leftarrow X \\ \text{MBR} &\leftarrow \text{AC} \\ M[\text{MAR}] &\leftarrow \text{MBR}. \end{aligned}$$

- **RTL for ADD  $X$ :** Similarly, the RTL for the ADD instruction is

$$\begin{aligned} \text{MAR} &\leftarrow X \\ \text{MBR} &\leftarrow M[\text{MAR}] \\ \text{AC} &\leftarrow \text{AC} + \text{MBR}. \end{aligned}$$

- **RTL for SUBT X:**

```

MAR  $\leftarrow X$ 
MBR  $\leftarrow M[\text{MAR}]$ 
AC  $\leftarrow AC - \text{MBR}.$ 

```

- **RTL for INPUT:**

```
AC  $\leftarrow \text{inREG}.$ 
```

- **RTL for OUTPUT:**

```
outREG  $\leftarrow AC.$ 
```

- **RTL for HALT:** No operations needed, no RTL

- **RTL for SKIPCOND:**

```

0  if IR[11 - 10] = 00 then
1    if AC < 0 then
2      PC  $\leftarrow PC + 1$ 
3    endif
4
5  else if IR[11 - 10] = 01
6    if AC = 0 then
7      PC  $\leftarrow PC + 1$ 
8    endif
9
10 else if IR[11 - 10] = 10 then
11   if AC > 0 then
12     PC  $\leftarrow PC + 1$ 
13   endif
14 endif

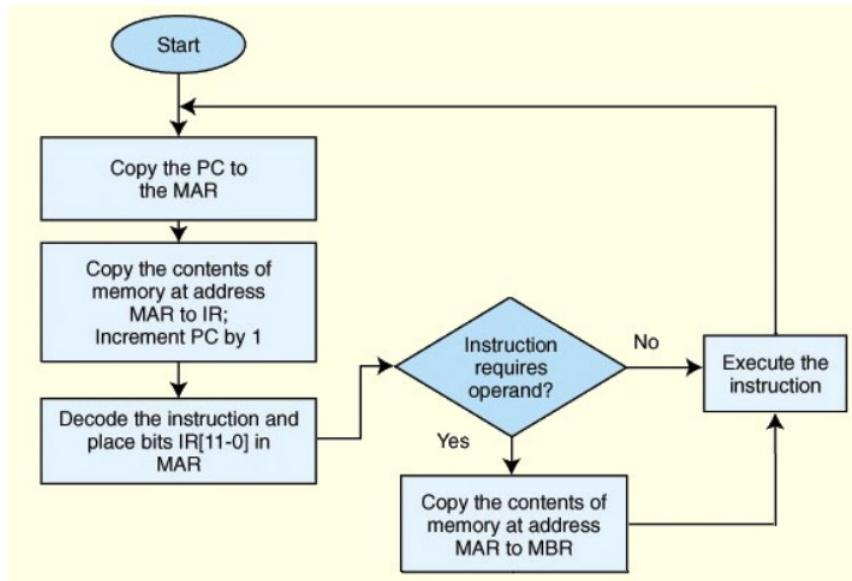
```

- **RTL for JUMP X:**

```
PC  $\leftarrow \text{IR}[11 - 0].$ 
```

- **Instruction Processing:** The fetch-decode-execute cycle is the series of steps that a computer carries out when it runs a program.

1. We first have to fetch an instruction from memory, and place it into the IR.
2. Once in the IR, it is decoded to determine what needs to be done next.
3. If a memory value (operand) is involved in the operation, it is retrieved and placed into the MBR
4. With everything in place, the instruction is executed.



**Note:** The fetch-decode-execute cycle is sometimes called the fetch-decode-fetch-execute cycle.

- **Example program:** Consider the simple MARIE program given below. We show a set of mnemonic instructions stored at addresses 100 - 106 (hex):

Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Subt 106	0100000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	111111111101001	FFE9
106	0000	0000000000000000	0000

Let's look at what happens inside the computer when our program runs. This is the LOAD 104 instruction.

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		100	---	---	---	---
Fetch	$\text{MAR} \leftarrow \text{PC}$	100	---	100	---	---
	$\text{IR} \leftarrow M[\text{MAR}]$	100	1104	100	---	---
	$\text{PC} \leftarrow \text{PC} + 1$	101	1104	100	---	---
Decode	$\text{MAR} \leftarrow \text{IR}[11-0]$	101	1104	104	---	---
	(Decode IR[15-12])	101	1104	104	---	---
Get operand	$\text{MBR} \leftarrow M[\text{MAR}]$	101	1104	104	0023	---
Execute	$\text{AC} \leftarrow \text{MBR}$	101	1104	104	0023	0023

The line in parentheses means that the RTN "branches" according to which instruction was decoded. That line needs no RTN because it uses combinational logic. That instruction will be executed in the execute step.

Our second instruction is ADD 105.

The fetch, decode and "get operand" (i.e., fetch the data) steps are the same. The only thing that is different is the execute, since the instruction is 'add' instead of 'load'.

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	$\text{MAR} \leftarrow \text{PC}$	101	1104	101	0023	0023
	$\text{IR} \leftarrow M[\text{MAR}]$	101	3105	101	0023	0023
	$\text{PC} \leftarrow \text{PC} + 1$	102	3105	101	0023	0023
Decode	$\text{MAR} \leftarrow \text{IR}[11-0]$	102	3105	105	0023	0023
	(Decode IR[15-12])	102	3105	105	0023	0023
Get operand	$\text{MBR} \leftarrow M[\text{MAR}]$	102	3105	105	FFE9	0023
Execute	$\text{AC} \leftarrow \text{AC} + \text{MBR}$	102	3105	105	FFE9	000C

The AC now has the value of  $\text{AC} + \text{MBR}$ . Remember that MARIE has 2's complement addition.

- **Register transfer notation (RTL):** RTN = register transfer notation, a synonym for RTL.
- **Indirect addressing:** So far, all of the MARIE instructions that we have discussed use direct addressing. This means that the address of the operand is explicitly stated in the instruction.

In indirect addressing, the address of the address of the operand is given in the instruction. In other words, the instruction gives the address of a memory location that contains the address of the operand.

Indirect addresses are needed to point at different values in a table so you can use a loop to process them, i.e., to get values from an address that changes during the run.

- **Indirect instructions:** MARIE contains two indirect instructions:
  - ADDI  $x$  (op code B): Look at address  $x$ , lookup the address stored there, and add that value to the AC.
  - JUMPI  $x$  (op code C): Branch to the address in memory cell  $x$ . Can be used to return from a subroutine implemented with JNS (see next slide for JNS).
- **Additional instructions:** MARIE also contains two other useful instructions that we have not described before:
  - JNS  $x$  (jump and store) (op code 0): Store return address (i.e., next sequential address) at  $x$ , then jump to  $x + 1$ .
  - CLEAR (op code A) sets the contents of the accumulator to zero.
- **JNS  $x$  (jump and store):** JNS  $x$  does the following:
  1. Stores the return address (i.e., the address of the next sequential instruction) at address  $x$ .
  2. Jumps to the address specified at address  $x + 1$
- **RTL for ADDI  $x$ :** The ADDI instruction specifies the address of the address of the operand. The following RTL shows us what is happening at the register level:

```

MAR  $\leftarrow x$ 
MBR  $\leftarrow M[\text{MAR}]$ 
MAR  $\leftarrow \text{MBR}$ 
MBR  $\leftarrow M[\text{MAR}]$ 
AC  $\leftarrow \text{AC} + \text{MBR}.$ 

```

- **RTL for JNS  $x$**

```

MBR  $\leftarrow \text{PC}$ 
MAR  $\leftarrow x$ 
M[MAR]  $\leftarrow \text{MBR}$ 
MBR  $\leftarrow x$ 
AC  $\leftarrow 1$ 
AC  $\leftarrow \text{AC} + \text{MBR}$ 
PC  $\leftarrow \text{AC}.$ 

```

- **RTL for CLEAR:** All it does is set the contents of the accumulator to all zeroes

$$\text{AC} \leftarrow 0.$$

- **Subroutines:** The jump-and-store instruction, JNS, gives us limited subroutine functionality.

We jump to a subroutine with JNS  $x$ , where the subroutine begins at address  $x + 1$ . That is, one word after the label used in JNS ( $x$ )

After the subroutine is finished, JUMPI  $x$  can be used to return to the original address, since JUMPI jumps to the address stored at  $x$ .

## Notes

- MARIE does not support parameter passing directly. You always work through global memory (shared variables in main memory).
- JNS cannot handle recursive calls because JNS  $x$  always uses the same address, namely  $x$ , to store the return address. In other words, the return address is stored in the first word of the subroutine. (Of course, that is not good programming practice today.)

Subroutine labels need to be defined as

- o NAME, HEX 0

Strictly because of the fact that JNS NAME jumps to the location NAME+1.

- **The control unit:** A computer's control unit ensures that each instruction is executed in sequence, making sure that data flows to the correct components as each instruction is executed..

There are two general ways in which a control unit can be implemented: hardwired control and microprogrammed control.

1. In a hardwired machine, digital logic components are used to select the "from" and "to" components and ensure that data flows to them at the correct time.
  2. A microprogrammed machine contains a small program in a piece of read-only memory in a component called the microcontroller. The microprogram contains the RTL for each instruction. In order to execute an instruction, the microcontroller interprets the corresponding piece of this program.
- **Marie is a hardwired machine:** The operation of MARIE's control unit is defined by the RTL for each instruction..

Each line of RTL is implemented with digital logic components that cause the appropriate data to flow from the component on the right-hand side of the RTL line to the left-hand side..

For example, the RTL for the ADD instruction is:

$$\begin{aligned} \text{MAR} &\leftarrow x \\ \text{MBR} &\leftarrow M[\text{MAR}] \\ \text{AC} &\leftarrow \text{AC} + \text{MBR}. \end{aligned}$$

We will soon see how each line of RTL is implemented.

- **Logical component:** When we call something a logical component, we mean it's a functional part of the CPU - defined by what it does, not necessarily by where it physically sits.
  - The Arithmetic Logic Unit (ALU) is the logical component that performs arithmetic and logic.
  - The Control Unit (CU) is the logical component that issues control signals and manages sequencing.

- The Registers (like MAR, PC, AC, IR) are logical components that store data or addresses.
- **More on Marie's control unit (CU):** In MARIE, the Control Unit (CU) is part of the CPU, and its main role is to generate and coordinate the control signals that tell the rest of the system what to do during each phase of the fetch-decode-execute cycle.

It's a logical component (a section of the CPU) responsible for directing the operation of the processor. It interprets the instruction currently in the Instruction Register (IR) and activates the correct control signals in the right sequence. These control signals go to components like the ALU, registers (AC, MAR, MBR, PC, IR), and the bus, ensuring data moves and operations occur correctly.

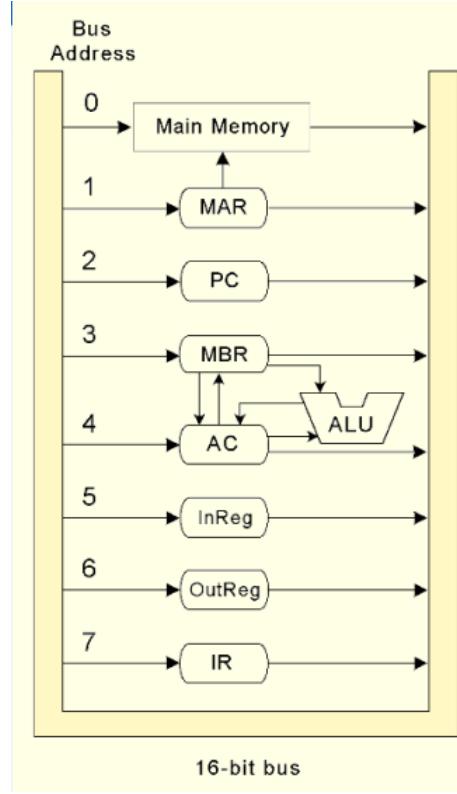
It's not just a "storage area" that contains control signals - control signals are generated dynamically by the CU's logic circuits, not stored like data. It doesn't hold data like registers or memory do. Instead, it's composed of decoding and timing logic that responds to the current instruction and clock cycle

When we say the control unit generates control signals, we mean it produces specific voltage patterns on wires (control lines) inside the CPU. Each instruction (like LOAD or ADD) has a unique combination of these voltage signals that tell different hardware components what to do that clock cycle.

Each control line is literally a wire connecting the control unit to some component (registers, ALU, memory interface, etc.).

- **Bus address signals:** Each of MARIE's registers has a unique address along the datapath, the addresses take the form of signals issued by the control unit. MARIE needs 3 lines to identify addresses 0 - 7 on the bus.

Many machines would use 8 signals, one of which would be high at any given time.

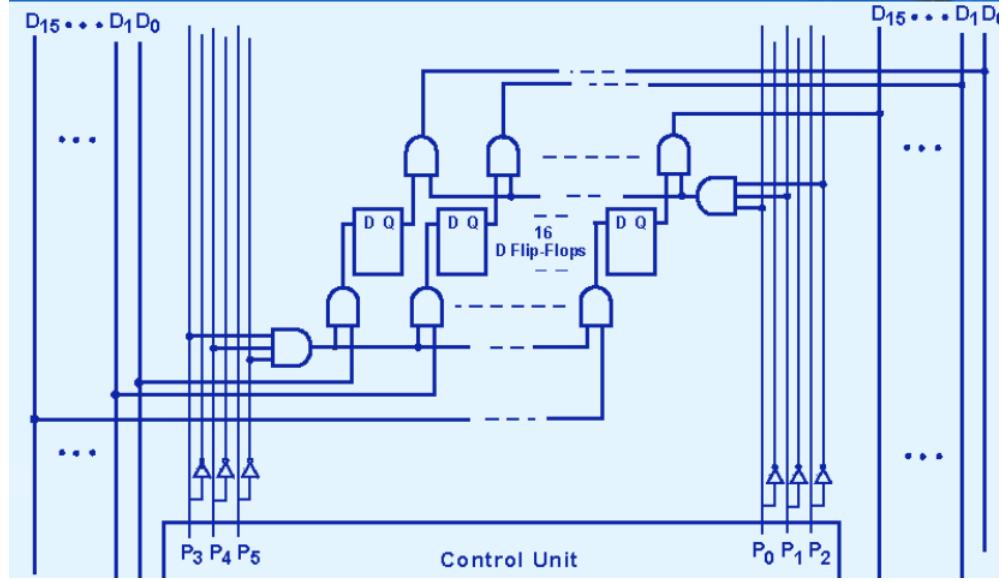


The MARIE CPU contains two sets of three signals.  $P_2$ ,  $P_1$ , and  $P_0$  determine which component data will be read from.  $P_5$ ,  $P_4$ , and  $P_3$  determine which component data will be written to

For example, the MBR has address 3. Since the MBR is a 12-bit register, we can build it as a set of 12 D flipflops. We add gates implementing  $P_3$  AND  $P_4$  AND NOT  $P_5$  before the input to these flipflops so that data will only flow into the MBR (i.e., writing to the MBR) when address 3 is activated.

Similarly, we add gates implementing  $P_0$  AND  $P_1$  AND NOT  $P_2$  after the output of these flipflops so that data will only flow out of the MBR (i.e., reading from the MBR) to the bus when those signals are activated.

- **The MBR:**



- **First line of ADD RTL:** Let's look again at the first line of the RTL for ADD:

$$MAR \leftarrow x.$$

After an ADD instruction is fetched, the address field,  $x$ , is in the rightmost 12 bits of the IR. The IR has a datapath address of 7. According to this line of RTL,  $x$  must be copied to the MAR. The MAR has a datapath address of 1. Thus, we need to raise signals  $P_2$ ,  $P_1$ , and  $P_0$  to read from the IR, and signal  $P_3$  to write to the MAR.

When the high order bits of the IR contain the opcode for ADD, these signals must be set high to implement the first line of the RTL. In succeeding clock ticks, the remaining lines of the RTL for ADD will be implemented.

- **Timing signals:** Most instructions need multiple clock ticks, one for each line of RTL. A binary counter is used to identify which line of RTL is being executed.

MARIE has a maximum of 3 lines of RTL for an instruction (7 if you include the advanced instructions). Therefore 4 timing signals (8 including the advanced instructions) are needed, one to indicate each line of RTL and one to reset the counter for the next instruction. These timing signals are named

$$T_0, T_1, T_2, \dots, T_7.$$

The  $C_r$  signal is used to reset the counter.

Theoretically, we could use combinations of two timing signals to obtain 4 clock times, or combine 3 timing signals to get 8 clock times, but we won't do it that way

- **ALU signals:** Looking at the RTL chart, we can see that the ALU has only three operations: add, subtract, and clear.

We have four ALU signals,  $A_0$  through  $A_3$ , available to indicate these operations. We will use

$$A_0 = \text{add}, \quad A_1 = \text{subtract}, \quad A_2 = \text{clear}.$$

We could use  $A_3 = \text{no operation}$ , but we won't. Another design alternative would be to use combinations of two signals to generate four values. When we want to signal an ALU instruction, we raise the corresponding signal.

**Note:** These signals don't need to match the ALU control values

- **Marie's control signals:** The entire set of MARIE's control signals contains
  - Register controls  $P_0$  through  $P_5$
  - ALU controls  $A_0$  through  $A_3$
  - Timing signals  $T_0$  through  $T_7$  and counter reset  $C_r$

All of the RTL lines needed for MARIE's instructions can be implemented by ensuring that the correct register and/or ALU control signals are raised during the correct clock ticks for a specific instruction.

Here is the full signal sequence for ADD. It shows which control signals must be set high at each clock tick

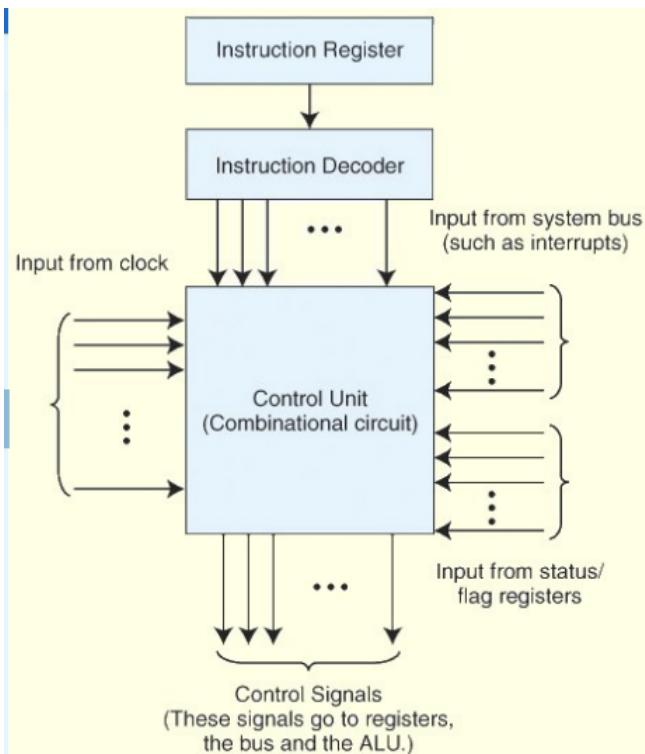
$$\begin{aligned}
 P_2 P_1 P_0 P_3 T_0 : & \text{ MAR} \leftarrow x \\
 P_4 P_3 T_1 : & \text{ MBR} \leftarrow M[\text{MAR}] \\
 A_0 P_1 P_0 P_5 T_2 : & \text{ AC} \leftarrow \text{AC} + \text{MBR} \\
 C_r T_3 : & [\text{Resetcounter}].
 \end{aligned}$$

- **Line 1:** input is address 7 (low order bits of the IR) and output is address 1 (MAR).
- **Line 2:** input is address 0 (main memory) and output is address 3 (MBR).
- **Line 3:** input is address 3 (MBR), output is address 4 (AC), and the operation happening inside the ALU (i.e., what happens to the AC) is addition.
- **Timing diagram:** This diagram is called a timing diagram ( $P_4$  is not shown). You can see which signals, including the timing signals, are high at each clock tick. This instruction has 4 clock ticks,  $C_0 - C_3$ .

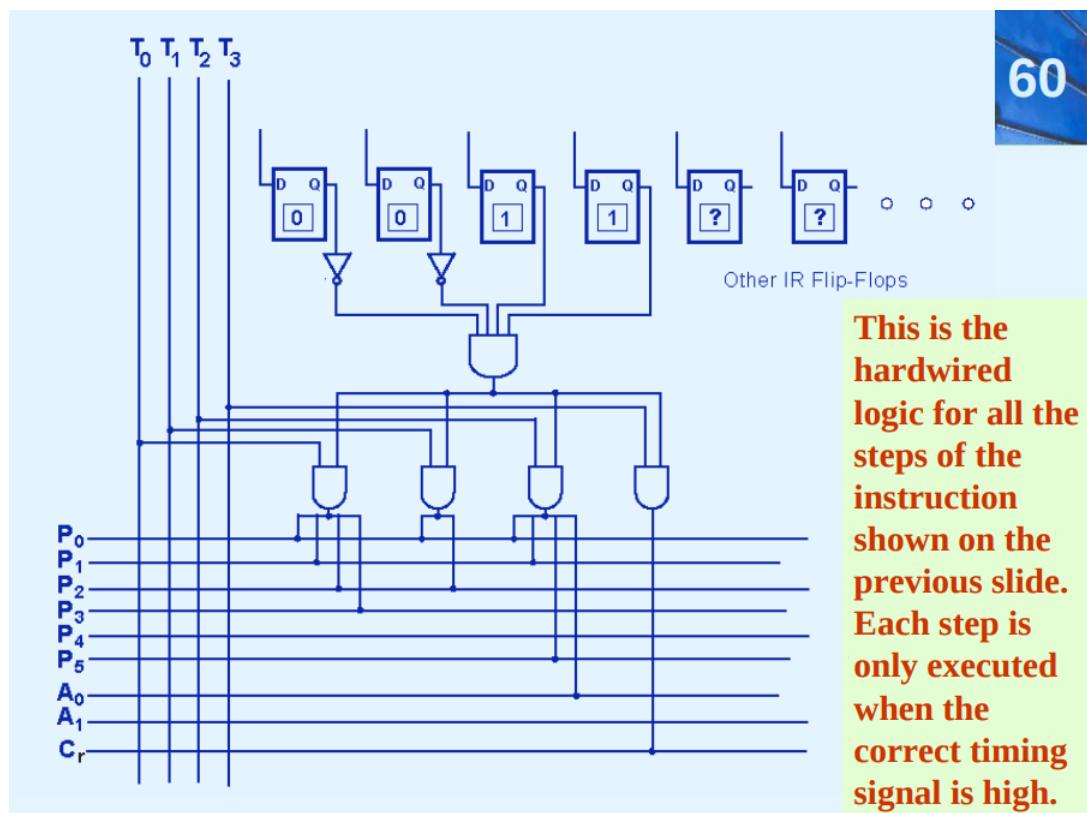
$$\begin{aligned}
 P_0 P_1 P_2 P_3 T_0 & \text{ are high at } C_0 \\
 P_0 P_2 T_1 & \text{ are high at } C_1 \\
 A_0 P_0 P_1 P_5 T_2 & \text{ are high at } C_2 \\
 C_r T_3 & \text{ are high at } C_3.
 \end{aligned}$$

- **More on the control unit:** The RTL and the control signals are the same whether the machine is hardwired or microprogrammed.

In hardwired control, the bit pattern of an instruction feeds directly into the combinational logic of the control unit.



- Control unit circuit (only shows the ADD instruction portion)



- **ROM:**

- A non-volatile memory chip - it keeps its contents even when power is off.
- Stores fixed data or instructions that rarely (or never) change.
- The CPU can read from it but normally can't write to it.
- Used to hold essential low-level code like a bootloader or microprogram for the control unit.

Think of ROM as permanent storage for instructions the hardware needs to start or operate.

- **Firmware:**

- The software stored in ROM (or Flash) that controls the hardware.
- It's the middle layer between hardware and higher-level software (OS, apps).
- Examples: BIOS/UEFI in PCs, router firmware, or the microcode inside a CPU.

- **PROM (Programmable read only memory):**

- Stands for Programmable ROM.
- It's manufactured blank, and you can program it once using a special device called a PROM programmer (it literally burns tiny fuses inside).
- After that, it's permanent - can only be read, not erased or re-written.

PROM = one-time programmable ROM..

- **Intro to microprogramming:** Now we switch to looking at microprogramming. In microprogrammed control, execution of microcode instructions produces control signal changes. The microprogram converts each microcode instruction into control signals.

The microprogram is stored in firmware, which is a small piece of read-only memory also called the control store. One microcode instruction is retrieved during each clock cycle.

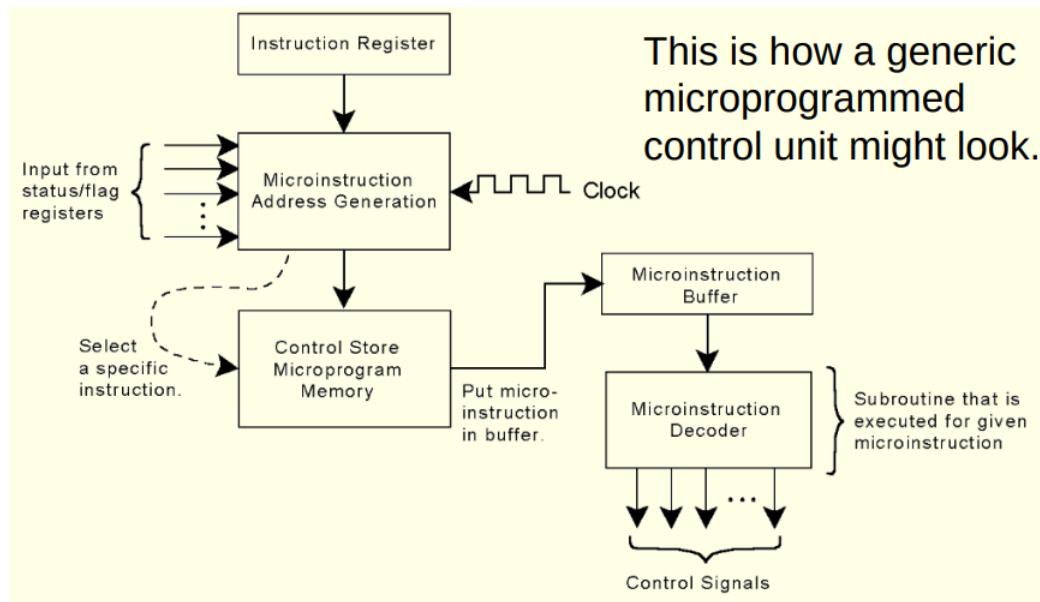
in a microprogrammed CPU, the microprogram replaces (or substitutes for) the hard-wired control circuit. However, it doesn't remove the control unit entirely - it changes its implementation.

So instead of a big network of AND/OR gates and flip-flops generating control signals directly, you have:

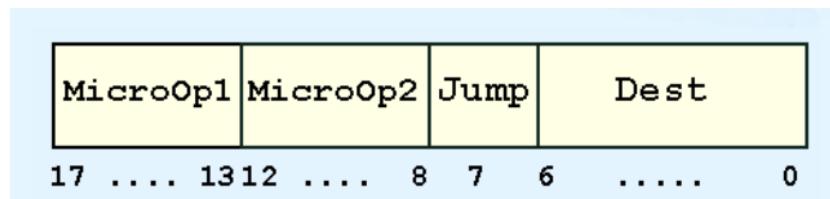
- A small, special-purpose memory (the control store) containing microinstructions.
- A microsequencer that fetches, decodes, and executes those microinstructions - just like the CPU itself fetches and executes regular instructions.

Hardwired Control	Microprogrammed Control
Uses combinational logic (gates, decoders, timing signals) to generate control signals directly.	Uses microinstructions stored in memory to specify which control signals go high each cycle.
Faster, but less flexible - to change instruction behavior, you must redesign the logic.	Slower, but more flexible - you can change or extend the instruction set by modifying the microcode.
Example: MARIE's control unit circuit with AND/OR gates.	Example: A microprogram stored in ROM that defines each instruction's control steps

- Generic microprogrammed control unit:



- **Microinstruction format:** If MARIE were microprogrammed, the microinstruction format might look like this:



We see that a microinstruction is 18 bits long.

- **Opcodes:** 5 bits
- **Jump bit:** 1 bit
- **Dest:** 7 bits

Thus, addresses are 7 bits.

MicroOp1 and MicroOp2 contain binary codes for each instruction. Jump is a single bit indicating that the value in the Dest field is an address that should be placed in the microsequencer, indicating which microprogram instruction should be executed next

- Marie micro opcodes:** The table below contains MARIE's micro-opcodes along with their corresponding RTL:

MicroOp Code	Microoperation	MicroOp Code	Microoperation
00000	NOP	01100	MBR $\leftarrow$ M[MAR]
00001	AC $\leftarrow$ 0	01101	OutREG $\leftarrow$ AC
00010	AC $\leftarrow$ AC - MBR	01110	PC $\leftarrow$ IR[11-0]
00011	AC $\leftarrow$ AC + MBR	01111	PC $\leftarrow$ MBR
00100	AC $\leftarrow$ InREG	10000	PC $\leftarrow$ PC + 1
00101	IR $\leftarrow$ M[MAR]	10001	If AC = 00
00110	M[MAR] $\leftarrow$ MBR	10010	If AC > 0
00111	MAR $\leftarrow$ IR[11-0]	10011	If AC < 0
01000	MAR $\leftarrow$ MBR	10100	If IR[11-10] = 00
01001	MAR $\leftarrow$ PC	10101	If IR[11-10] = 01
01010	MAR $\leftarrow$ X	10110	If IR[11-10] = 10
01011	MBR $\leftarrow$ AC	10111	If IR[15-12] = MicroOp2[4-1]
11000	MBR $\leftarrow$ PC	11001	MBR $\leftarrow$ X
11010	AC $\leftarrow$ 1	11011	PC $\leftarrow$ AC
11100	AC $\leftarrow$ MBR		

- First lines of Marie's microprogram:

Address	Microop 1	Microop 2	Jump	Dest
0000000	MAR $\leftarrow$ PC	NOP	0	0000000
0000001	IR $\leftarrow$ M[MAR]	NOP	0	0000000
0000010	PC $\leftarrow$ PC + 1	NOP	0	0000000
0000011	MAR $\leftarrow$ IR[11-0]	NOP	0	0000000
0000100	If IR[15-12] = MicroOp2[4-1]	00000	1	0100000
0000101	If IR[15-12] = MicroOp2[4-1]	00010	1	0100111
0000110	If IR[15-12] = MicroOp2[4-1]	00100	1	0101010
0000111	If IR[15-12] = MicroOp2[4-1]	00110	1	0101100
0001000	If IR[15-12] = MicroOp2[4-1]	01000	1	0101111
...	...	...	...	...

The first four lines contain the fetch (first 3 lines) and decode (line 4) steps of the execution cycle. The remaining lines are the beginning of a jump table.

The jump table entries indicate that if the high order bits of the IR (i.e., the opcode) have a given value, the microprogram will jump to the address where the remaining microoperations for that opcode will be found.

For example, LOAD has opcode 1. Looking through the jump table, we can see that a LOAD instruction will cause the microcode processor to jump to 010 0111.

At that address, it will find the fetch operand step for that instruction (if required) followed by the execute step, i.e., the RTL in microcode format:

```
010 0111 MAR ← X NOP 0 000 0000  
010 1000 MBR ← M[MAR] NOP 0 000 0000  
010 1001 AC ← MBR NOP 1 000 0000.
```

The last step of the encoded RTL will have a 1 in the JUMP field with a DEST of 000 0000. That will cause the system to jump to 000 0000, i.e., the beginning of the fetch cycle for the next operation.

The chart above shows the microcode in RTL for convenience, but we could also show it in binary.

```
010 0111 01010 00000 0 000 0000  
010 1000 01100 00000 0 000 0000  
010 1001 11100 00000 1 000 0000.
```

- **Micropogramming:** A microprogrammed control unit works like a system-in-miniature. Microinstructions are fetched, decoded, and executed just like regular instructions. This extra level of interpretation makes micropogrammed control slower than hard-wired, but this is not important since computers are getting faster every day.

The advantage of microprogrammed control is that only the microprogram needs to be changed if the instruction set changes. In fact, all modern machines are microcoded. We study hardwired machines first just to make it easier to understand the microcode architecture.

## 6.5 Chapter 5: Instruction set architecture

### 6.5.1 Pipelining

- **Instruction-level pipelining:** Most CPUs today divide the fetch-decode-execute cycle into smaller steps. These smaller steps can often be executed in parallel to increase throughput, such parallel execution is called **instruction-level pipelining**

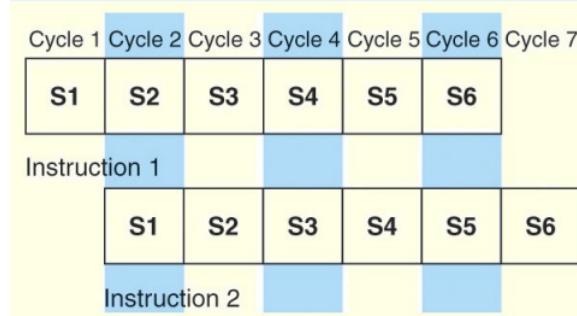
**Note:** Throughput is the amount of material or items passing through a system or process.

- **Example:** Suppose a fetch-decode-execute cycle were broken into the following smaller steps:

1. Fetch instruction.
2. Decode opcode.
3. Calculate effective address of operands
4. Fetch operands.
5. Execute instruction.
6. Store result.

Then a circuit could be built so that each stage is complete before the next one starts, creating a sixstage pipeline.  $s_1$  fetches the instruction,  $S_2$  decodes it,  $S_3$  determines the address of the operands,  $S_4$  fetches them,  $S_5$  executes the instruction, and  $S_6$  stores the result.

For every clock cycle, one small step is carried out, and the stages are overlapped.



With

- S1. Fetch instruction.
- S2. Decode opcode.
- S3. Calculate effective address of operands
- S4. Fetch operands.
- S5. Execute.
- S6. Store result.

- **No pipelining vs pipelining:**

- **No pipeline:** Suppose each of the stages below takes one clock cycle. If there is no pipelining, the processing time for an instruction is 6 cycles.

Suppose the machine speed is 10 ns/cycle with no pipeline

$$6 \frac{\text{cycles}}{\text{instruction}} \times 10 \frac{\text{ns}}{\text{cycle}} = 60 \frac{\text{ns}}{\text{instruction}}.$$

So,

$$\left( \frac{1}{60} \times 10^9 \right) \text{inst/sec.}$$

- **Full pipeline:** If pipelining is supported and the pipeline can be kept full, six instructions can be processed at once (in different stages), so the machine can run six times as fast.

In fact, one instruction will emerge from the pipeline every cycle. Suppose machine speed is 10 ns with full pipeline:

$$1 \text{cycle/instr} * 10 \text{ ns/cycle} = 10 \text{ ns/instr.}$$

So,

$$\left( \frac{1}{10} \times 10^9 \right) \text{inst/sec} = (100 \times 10^6) \text{ inst/sec.}$$

**Note:** We conventionally use multiples of 3 for the powers of 10, e.g., 106 mips = 106 million instr/sec

- **Steps of different lengths:** Suppose  $S_4$  takes 2 cycles and  $S_5$  takes 4 cycles. The total time for an instruction to pass through the pipeline is

$$1 + 1 + 1 + 2 + 4 + 1 = 10 \text{ cycles.}$$

The slowest step (i.e., the step with the maximum time) determines the speed of the pipeline, so one instruction emerges every 4 cycles. In other words, we get 4 cycles/instr instead of 10 cycles/instr with no pipeline.

At 10 ns/cycle with full pipeline,

$$4 \text{cycle/instr} \times 10 \text{ns/cycle} = 40 \text{ns/instr.}$$

So,

$$\left( \frac{1}{40} \times 10^9 \right) \text{inst/sec} = (25 \times 10^6) \text{ inst/sec.}$$

- **Multiple copies of a step:** Suppose we still have that  $S_5$  takes 4 cycles per instruction. We could speed up the process by adding a second copy of the circuit for that step. Then the slowest step would take 2 cycles/instr, since instructions can alternate going through the left-hand copy or the right-hand copy of S5.
- **two copies of a step, but duplication doesn't double the speed:** Remember that total time is determined by the slowest step, which is not necessarily the duplicated step.

Suppose S4 takes 3 cycles and S5 takes 4. Duplicating S5 means that S5 effectively takes 2 cycles/sec. But duplicating S5 won't double the instructions per second because S4 is now the slowest step. In this case, total time = 11 cycles/instr. The slowest step is 3 cycles/sec.

- **Pipeline stalling:** Stalling means that the pipeline is no longer full because one instruction took more cycles than expected
- **Pipeline flushing:** Flushing the pipeline means that no new instructions can enter the pipeline until the given instruction is finished.
- **Pipeline hazards:** These calculations takes a number of things for granted. First, we have to assume that the architecture supports fetching instructions and data in parallel. Second, we assume that the pipeline can be kept filled at all times. This is not always the case. Pipeline hazards arise that cause pipeline conflicts and stalls

An instruction pipeline may stall or be flushed for any of the following reasons

1. **Resource conflicts:** When an instruction needs a resource, such as a register, that a previous instruction still in the pipeline is still using
2. **Data dependencies:** When an instruction needs the result of a previous instruction still in the pipeline
3. **Conditional branching:** When a branch statement such as "if a then go to b else go to c" occurs in the pipeline, and b or c occur soon after this statement. Neither b nor c can enter the pipeline until the result of the branch is determined, because we do not know whether they will ever be executed at all.

When any of these pipeline hazards occurs, the later instruction cannot enter or continue along the pipeline until the conflict is resolved

Measures can be taken at the software level as well as at the hardware level to reduce the effects of these hazards, but they cannot be totally eliminated

When generating code one statement at a time, it is easy to get redundant instructions like STORE R1, X as the last step of one instruction followed by LOAD R1, X as the first step of the next. An optimizing compiler can remove these.

An optimizing compiler can also use different registers for operations we may want to execute in parallel

### 6.5.2 Instruction set architecture (ISA)

- **ISA intro:** The instruction set (ISA) of an architecture is the set of instructions that the CPU can handle. The ISA is usually defined by a manual such as the IBM mainframe Principles of Operation. To conform to principles of good hardware and software design, the ISA is usually highly organized. It is not just a random set of instructions.

A key fact about an ISA is the number of bits per instruction. This can be fixed or variable. A second key fact about an ISA is the basic operating model of the CPU. We will study three models

1. Stack-based
2. Accumulator-based
3. Register-based. Also called GPR (general purpose register).

Another key fact about an ISA is the set of instruction types that it contains. Instructions can be characterized by

- Number of operands per instruction
- Size of operands
- Type of operands, e.g., memory or register

- **IBM GPR design:** The IBM mainframe (a GPR design) includes the following types of instructions:

- **Two-byte instructions (1 or 2 addresses):**
  - \* **SI (storage-immediate):** one operand is in memory, the other is given in the instruction (i.e., as a constant)
  - \* **RR:** register to register
- **Four-byte instructions (2 or 3 addresses):**
  - \* **RS:** register to memory
  - \* **RX:** register to memory, with another register as an index register
- **Six-byte instructions (3 addresses):**
  - \* **SS (storage-to-storage):** multiple operands are memory addresses

- **Comparing ISAs:** Although there is no way to say that one ISA is "better" than another, some metrics that can be applied to an ISA include:

- Amount of memory required to implement a specific algorithm
- Instruction complexity
- Average instruction length (in bits)
- Total number of instructions in the instruction set

- **Designing an ISA:** Some factors to take into account in designing an ISA include

- Instruction length: short, long, or variable
- Number of operands needed
- Number of addressable registers
- Memory organization: byte- or word addressable
- Addressing modes. any or all of direct, indirect or indexed

ISA and hardware have to be designed together, the factors above are determined by basic features of the CPU

- **CPU operating models:** There are three main options

1. Stack architecture
2. Accumulator architecture
3. General purpose register (GPR) architecture

In choosing one over the other, the basic tradeoff is simplicity (and cost) of hardware design vs. ease of use

As machines become faster and software becomes more sophisticated, these differences become less important, the Assembler can make up for lack of options in the ISA, the compiler usually hides these differences from user

- **Stack architecture:** In a stack architecture, the stack is an implicit operand of every operation, only the top element (or elements) of the stack can be accessed

In an accumulator architecture, the accumulator is an implicit operand of every binary operation, the other operand is in memory, creating lots of bus traffic

In a general purpose register (GPR) architecture, registers can be used instead of memory

- Faster than accumulator architecture
- Efficient implementation for compilers
- Results in longer instructions because all operands need to be explicitly specified (no implicit use of stack or accumulator)

- **GPR systems:** GPR systems are very common today. There are three types

1. Memory-memory where two or three operands may be in memory
2. Register-memory where at least one operand of an instruction must be in a register
3. Load-store where no operands may be in memory except for the LOAD and STORE instructions

The number of operands in an instruction and the number of available registers both affect instruction length

- **IBM mainframe:** IBM mainframe is a prime example of a GPR system. Its design was a savvy business decision in two ways

1. **Upgrade path:** From the beginning, IBM planned to offer a series of machines with different architectures but the same ISA
2. **Features for multiple kinds of users:** Provided decimal for business users, floating point for scientific users, and binary integers for everyone

- **Infix notation:** Infix notation is the common way we write mathematical expressions, where the operator is placed between two operands.

- Operators appear in the middle (between operands).
- It often requires parentheses to clarify order of operations.
- It's the standard notation used in everyday math and programming.

For example,  $a + b$  or  $(a + b) \cdot (c + d)$ .

- **Postfix notation (reverse polish notation):** Reverse Polish notation is a way of writing expressions where the operator comes after the operands, with no need for parentheses. You write the operands first, then the operator. Operations are evaluated as soon as the operator appears, using the most recent operands.

- **Infix to postfix conversion:**

1. Put parentheses around each binary operation that doesn't already have them.  
Follow the normal order of operations
2. Starting at the innermost level, move each binary operator to follow its operands.
3. Repeat at the next level up of parentheses
4. Continue until all operators follow their operands.
5. Remove all the parentheses

For example,

$$\begin{aligned} a \cdot b + c/d &= ((a \cdot b) + (c/d)) \\ &\rightarrow ((ab) + (cd/)) \\ &\rightarrow ((ab)(cd/)+) \\ &= ab \cdot cd / + . \end{aligned}$$

The principal advantage of postfix notation is that parentheses are not needed

- **Stack machines:** Stack machines use zero- and one-operand instructions.

1. **LOAD and STORE** instructions require a single operand in memory to indicate where the data is being loaded from or stored to
2. The **POP**: operation involves only the top element of the stack
3. Binary instructions (e.g., **ADD, MULT**) use the top two items on the stack and place the result on the stack. In our examples, we will use the convention that the top of the stack is the rightmost operand.

In a stack ISA, the postfix expression

$$z = x \ y \times \ w \ u \times \ + .$$

Is implemented as

```
PUSH x
PUSH y
MULT
PUSH w
PUSH u
MULT
ADD
STORE z
POP.
```

When a binary operation is executed:

1. The two operands are popped off the stack.

2. (\*) The top one (the last one pushed) is on the right side of the operation

3. The operation is executed and the inputs are popped
4. The result is pushed onto the stack.

**Note:** STORE doesn't pop its input, so you need to.

- **Expressions in an Accumulator machine:** The infix expression

$$x \cdot y + w \cdot u$$

can be calculated like this

```
LOAD X
MULT Y
STORE TEMP
LOAD W
MULT U
STORE TEMP2
LOAD TEMP
ADD TEMP2
STORE Z.
```

the AC is the left-hand operand of a binary operator.

- **Expressions in a GPR machine:** We can calculate the same expression like

```
LOAD R1, X
MULT R1, Y
LOAD R2, W
MULT R2, U
ADD R1, R2
STORE Z, R1.
```

With a three-address machine, we could do

```
MULT R1, X, Y
MULT R2, W, U
ADD Z, R1, R2.
```

- **Architecture comparison:** Which of these architectures is faster depends on many factors, including:

- Memory access speed
- Cache size and speed. A cache is a small amount of fast memory where frequently used data can be stored.
- Cycle time
- Pipelining.

In other words, neither the fact that one machine has shorter instructions nor the fact that another requires fewer instructions to implement a given algorithm is a guarantee of greater speed

Again, regardless of hardware manufacturers' ads, cycle time alone is not a determining factor of speed

- A compute-bound program is one where the CPU is busier than I/O, e.g., meteorology.
- An I/O-bound program is one where I/O prevents full use of the CPU. Most of our programs are I/O bound, which means that disk access time is another factor determining speed.

## 6.6 Chapter 8: Software

### 6.6.1 Assemblers

- **Intro:** Mnemonic instructions, such as LOAD 104, are easier for humans to write and understand than binary instructions.

The assembler translates instructions that are comprehensible to humans into binary for the CPU to interpret

The assembler creates executable software compatible with the operating system and available hardware.

- **Machine code:** When an assembler translates an instruction like LOAD  $x$ , the assembler produces a binary sequence that contains:
  1. **Opcode:** the instruction identifier (e.g., “LOAD”)
  2. **Operands:** such as the address, register number, or immediate value
  3. **Possibly other fields:** depending on the CPU (flags, function bits, etc.)

That binary sequence is the machine code.

- **Object files:** Assemblers create an *object program file* from mnemonic source code in two passes.

An object file is a partially compiled file produced by a compiler before linking. It contains machine code, but the code is not yet fully ready to run because external references (functions, variables in other files, libraries, etc.) are still unresolved.

- **Forward / backward references:** Forward references happen when your code refers to something before it has been defined.

A **backward reference** is simply when your code refers to something that has already been defined earlier.

- **The two passes:**
  1. **During the first pass:** The assembler assembles as much of the program as it can, while it builds a symbol table that contains memory references for all symbols in the program
  2. **During the second pass:** The instructions are completed using the values from the symbol table.
- **The symbol table and the first/second pass:** The **symbol table** is basically the assembler’s or compiler’s name-tracker.

It’s a map which records every important name it sees—like labels, variables, and functions—and remembers information about them (such as their address or type).

- When the assembler sees a name (like a label), it writes it into the symbol table along with where it is in the program.
- If the program uses a name before it’s defined (a forward reference), the assembler notes it and later comes back to fill in the correct address once the name is finally defined.
- When linking multiple object files, the linker uses the symbol tables from each file to match up which names belong to which actual addresses.

The symbol table is how the compiler/assembler keeps track of what every name means and where it is.

Two passes are needed to handle forward references. Suppose your program contains `B x`, where  $x$  is an address that occurs after the branch statement.

How can the assembler know the absolute address of  $x$ , i.e., how can it know how many bytes will occur between the branch statement and the label  $x$ ?

During the first pass, when the assembler first encounters the label / symbol  $x$ , the assembler adds the label  $x$  to the symbol table. It puts zeros in the address field of the code being assembled for the branch statement, i.e., it leaves the address unresolved until it finds it. Still in the first pass, once  $x$  is found, it updates the symbol table to now map  $x$  to its address, since it now knows where  $x$  is defined.

References are resolved during the second pass. By the end of the first pass, the symbol table is fully built. As long as the symbols/labels we use in the program are defined, the symbol table will map that symbol / label to its address. On the second pass, it looks up all unresolved references and replaces the zeros (in the machine code) with the correct address.

- **The linker:** The linker is a tool that takes one or more object files (the partially-compiled outputs from the assembler/compiler) and combines them into a single final program, which is either an executable file, a library, or a module

When you compile a program with multiple files, each object file contains

- machine code
- missing addresses for external symbols (functions/variables in other files)
- a symbol table

The linker's job is to:

- Match up all the undefined symbols with their actual definitions.
- Fill in all missing addresses in the machine code.
- Lay out all the code and data in memory in a single, unified executable.

- **The two passes, example:** Consider the program

Address	Instruction
100	Load $x$
101	Add $y$
102	Store $z$
103	Halt
104 $x,$	DEC 35
105 $y,$	DEC -23
106 $z,$	HEX 0000

During the first pass, we know that LOAD will generate op code '1', but we don't know the address of  $x$  yet. For HALT, we can generate the entire instruction because it has no addresses to resolve.

When we get to  $x$ , we can see that its address will be 104. So, at the end of the first pass, we have the symbol table

Symbol	Address
$x$	<i>unresolved</i>
$y$	<i>unresolved</i>
$z$	<i>unresolved</i>

and partial machine code

Machine code
1 000
3 000
2 000
7 000

In the second pass, we fill in addresses for  $x$ ,  $y$ , and  $z$ .

Machine code
1 104
3 105
2 106
7 000

- **Output of assemblers (relocatable binary code):** The output of most assemblers is a stream of **relocatable binary code**. In relocatable code, operand addresses are relative to where the operating system chooses to load the program.

When relocatable code is loaded for execution, special registers provide the base address. Addresses specified within the program are interpreted as offsets from the base address.

**Note:** Absolute (nonrelocatable) code is most suitable for device and operating system control programming

- **Absolute code:** Absolute code is machine code that uses fixed, hard-coded memory addresses. Because the addresses are built directly into the code, the program:
  - must be placed at a specific location in memory
  - cannot be moved without breaking the program
  - has no relocation, no flexibility

This was common in early computers and simple embedded systems.

If code is absolute, then

- Jumps contain absolute addresses
- Loads/stores contain absolute addresses
- The code cannot be moved
- The code cannot be relocated
- The code cannot be linked with other modules
- The code must run at a specific fixed address

Absolute code cannot be linked because to link two (or more) object files together, the linker must:

- Combine their code sections into one big program
- Reposition code (since each file starts at address 0)
- Assign final addresses for all functions and variables
- Rewrite instructions so they point to the right final address

This process is called relocation. But if the code is absolute, the linker cannot change those addresses.

- **Relocatable code:** Relocatable code (also called position-independent code or maintainable code) does NOT contain final memory addresses. Instead, it has symbolic references that the linker or loader fills in later.

the machine code contains:

```
o jump SOME_LABEL    <- unresolved reference
```

and the linker later patches in the correct address. This makes the program:

- movable
- linkable with other object files
- loadable anywhere in memory

**Note:** If you write a literal address, the assembler encodes it literally, If you reference a symbol (like a label), the assembler inserts:

- An instruction with a placeholder
- a relocation entry saying: “The linker must fix this address later.”

Relocatable code means:

- symbols are not bound to final addresses yet
- the machine code contains placeholders for symbolic references
- the assembler creates relocation entries
- the linker or loader will fill in the real addresses later

**Note:** The assembler decides whether to generate relocation entries.

- **Relocation entry:** A relocation entry is a record that tells the linker: “At this specific location in the machine code, you must insert or adjust the address of this symbol.”

A relocation entry contains:

- location inside the object file where the placeholder lives
- type of relocation (absolute? relative? PC-relative?)
- size (how many bytes to rewrite)
- which symbol it refers to
- addends (if needed for offset calculations)

### 6.6.2 Binding

- **Binding:** The process of assigning physical addresses to program variables is called **binding**.

Binding can occur at compile time, load time, or run time. Compile time binding gives us absolute code.

- **Compile time binding (absolute code):** Compile time binding gives us absolute code. In absolute code, a program is always loaded into memory at the same location

Absolute code is only used today for programs such as the operating system and network handlers, which can't be moved around because who would do it? (Imagine the operating system moving itself around while it is running...)

Compile-time binding means all symbol addresses are fixed (decided) during compilation/assembly, not delayed until linking or loading.

If the compiler/assembler knows every address, it must:

- compute and insert final addresses into the machine code
- immediately “bind” symbols to their real numeric addresses

There is nothing left to fix later. If you choose to resolve every address at compile time, the resulting machine code must hard-code those addresses. And that is absolute code. There is no relocation stage, because the compiler has already “locked in” all the addresses.

- **The loader:** The loader is a program / subsystem whose job is to load an executable into memory and prepare it to run.

The loader is the part of the operating system that:

- Reads the executable file (ELF on Linux, PE on Windows, Mach-O on macOS)
- Creates a new process
- Allocates virtual memory for code, data, stack, heap
- Loads the program’s segments into memory
- Performs load-time relocation (if needed)
- Links shared libraries (dynamic linking)
- Sets up the initial stack and arguments
- Jumps to the program’s entry point (main/\_start)

Then your program begins executing.

- **Load time binding:** Load time binding assigns physical addresses as the program is loaded into memory.

With load time binding the program can be loaded into different places but it cannot be moved once it is loaded. That is not common today and is included here only for the sake of completeness.

- **Run time binding:** Run time binding requires a base register or equivalent to carry out the address mapping. That is the normal way of loading programs today

Runtime binding means the program does NOT know the address of a symbol until the moment that symbol is actually used while the program is running.

- **Link editors (linkers) and static linking:** On most systems, binary instructions must pass through a link editor (or linker) to create an executable module.

Link editors combine multiple binary object files into a single executable file, resolving references to all external symbols. An external symbol is a reference to data or code in another object module

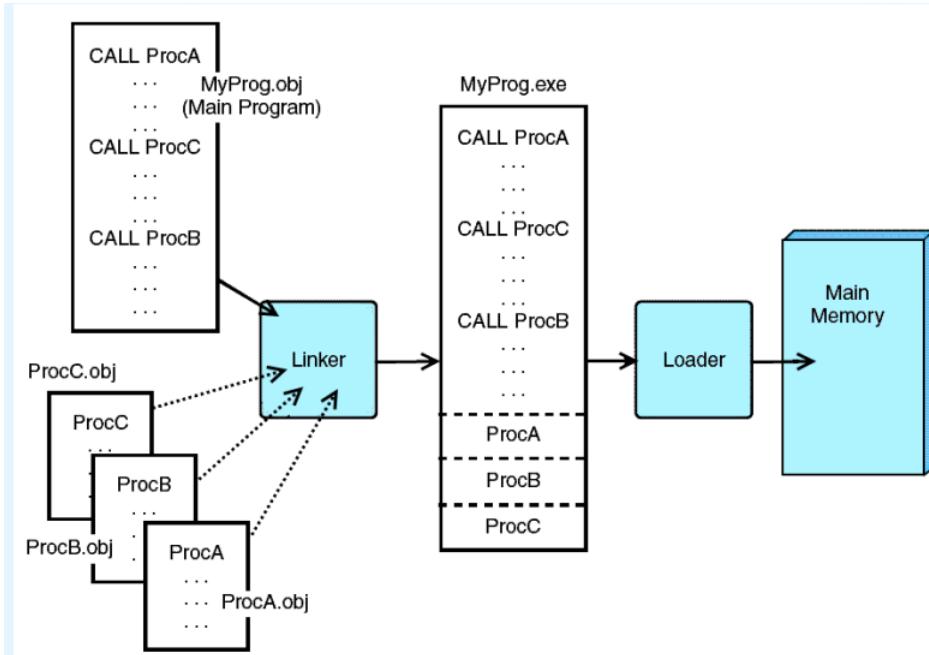
**Note:** This is called **static linking**.

- **Linkers two passes:** The link editor has two passes, just like the assembler, and for the same reason.

Suppose module A calls function B which is in a different object file. The first time the linker reads module A, it has not assigned an address to function B yet.

The first pass reads all the object modules and creates a symbol table containing the names of all external references.

The second pass reads all the object modules again, assigning addresses from the symbol table to all the external references. It also builds an executable module, which may have a different format from the object modules.



- **Dynamic linking:** A problem with static linking comes when someone wants to update a function in one of the modules. After updating your function, you need to find every executable it was linked into and relink it.

A way to solve this problem is dynamic linking. In dynamic linking, any module with a function that can be called is loaded into a dynamic link library (DLL).

As the program runs, when an external function is called, the system finds the correct module in the DLL and calls the function

The advantage of dynamic linking is that when you want to change a function, you only need to update it in its DLL. You do not need to find all the programs that call it

Another advantage of dynamic linking is that executables are smaller, since they don't contain copies of all the functions they call

A disadvantage of dynamic linking is the inverse of the disadvantage of static linking

Suppose your program still needs the old version of a function. If the DLL is updated with a new version that is incompatible or simply works differently, your program may no longer work. So you still need to be aware of any library updates

### 6.6.3 Compilers

- **Compilers:** A compiler converts a programming language that is comprehensible to humans into binary
- **Assemblers vs compilers:** In assembly language, there usually is a one-to-one correspondence between a mnemonic instruction and its machine code.

Compilers allow higher-level constructs such as  $X = Y + Z$

- **Higher level assemblers:** Some higher-level assemblers allow constructs such as IF, looping, or even pointer notation. This bridges the difference between a high-level assembler and a low-level language such as C

C is called a low-level language because, although it contains constructs such as IF, looping and pointers, there is a much more direct relationship between C code and the compiled binary than with higher-level languages

- **Macros:** In some assemblers, higher-level constructs are implemented as macros.

In a macro, the programmer writes the higher-level construct, and a pre-pass of the assembler converts the construct to a set of assembler language instructions before the regular assembly process.

Macros have become deprecated because they create hard-to-find bugs when mixed with compiled code

- **Compiler implementation:** Compilers bridge the gap between the higher level language and the machine's binary instructions.

A traditional approach to this translation is a six-phase process. The first three are analysis phases, the last three compiler phases are synthesis phases

1. **Lexical analysis** extracts tokens, e.g., reserved words and variables.
  2. **Syntax analysis (parsing)** checks statement construction.
  3. **Semantic analysis** checks data types and the validity of operators.
  4. **Intermediate code generation** creates an intermediate representation to facilitate optimization and translation.
  5. **Optimization** creates assembly code while taking into account architectural features that can make the code efficient.
  6. **Code generation** creates binary code from the optimized assembly code.
- **Intermediate code generation: Three address notation:** A traditional form of intermediate code is three-address notation.

In this form, each instruction has at most three operands, typically a binary operator and an assignment. For example,  $Z = X + Y$ .

- **Modularity:** Through the use of modularity, compilers can be written for various platforms by rewriting only the last two phases.

Thus front-ends for multiple languages can generate the same three-address notation. Then different code can be generated for different platforms.

- **More (or fewer) phases:** Compilers may have more or fewer phases. Combining the first two or even the first three phases is very common today.

These phases may be accomplished via the use of a compiler constructor such as YACC (“yet another compiler constructor”). A compiler constructor builds the phases of a compiler from a description of the language.

The description is often based on BNF (Backus-Naur form), a format for defining each type of statement similar to those used in elementary programming textbooks.

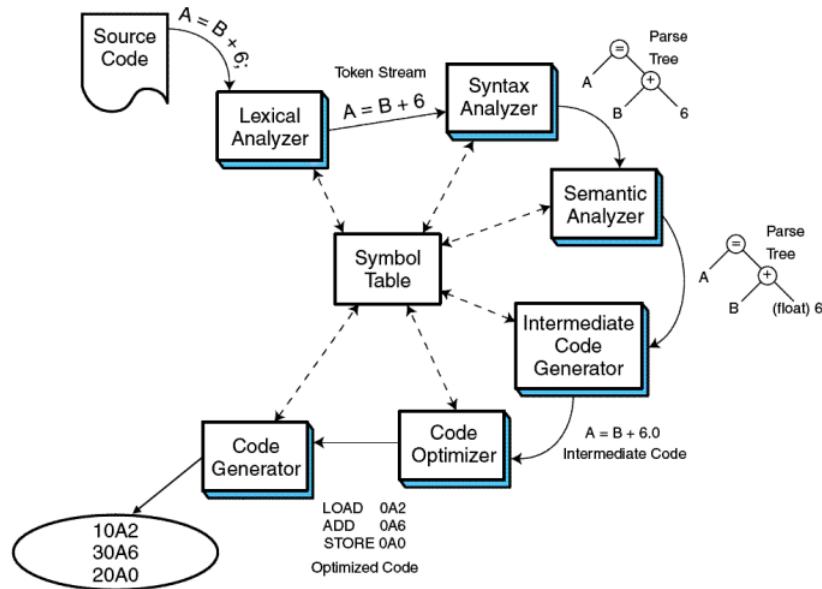
- **Optimization:**

- **Pinhole:** Pinhole optimizers use a sliding window to look at small numbers of consecutive assembler instructions. Redundant sequences such as “STORE X, LOAD X, STORE X” can be removed.
- **Global optimizers:** Global optimizers look at the bigger picture. Perhaps a constant “TAX\_RATE = .03” is defined but never used.

Even if it is used, perhaps it is not necessary to define storage for the variable. Simply replace the variable by the constant wherever it is used.

Optimizers need to take function calls into account.

- **Compiler phase figure**



- **Interpreters:** Interpreters produce executable code (not really) from source code in real time, one line at a time. This makes interpreted languages slower than compiled languages, but they are convenient for debugging because you can update one line at a time and keep running.

An interpreter:

- Reads bytecode/AST instructions,
- Decodes them one by one,

- Executes their behavior using pre-written handler routines,
- Never produces native CPU instructions.

A pre-written handler routine is a small piece of code inside an interpreter that implements the behavior of one specific bytecode or AST operation. It is the mechanism an interpreter uses to simulate program instructions instead of compiling them.

In other words, it simulates the program using its own execution loop.

Interpreters do not convert IR into machine code. Instead, each instruction is routed to one of these handlers.

So execution looks like this:

1. Fetch next bytecode instruction.
2. Look up its handler (often in a switch or dispatch table).
3. Jump to that handler code.
4. Handler performs the operation.
5. Return to the interpreter loop.

This dispatch overhead is why interpreters are slower.

It would be nice to have the speed of the compiler and the flexibility of the interpreter at the same time. That is possible now due to faster computers that can recompile modules faster and more sophisticated software designs that can handle on-the-fly software updates, Python is a good example.

- **Python:** Python demonstrates that a language can behave like an interpreter while still doing significant compilation, providing the best of both worlds.

When you run Python

```
1 python main.py
```

it seems like the interpreter is just executing your source code line by line, but Python actually compiles your code automatically.

Before execution, CPython (the official Python implementation) compiles your source (.py) into bytecode (.pyc), which lives in `__pycache__/`. This bytecode is

- Lower-level
- Faster to execute
- Platform-independent
- Not the original source code

This compilation happens on the fly, so you don't see it. So Python is not "purely interpreted", it's a hybrid.

The Python "interpreter" is a virtual machine, called the Python Virtual Machine (PVM). The PVM

- Reads bytecode instructions
- Executes them one at a time
- Just like a classic interpreter executes source lines

This hybrid design is why Python can maintain the interactive benefits of an interpreter while gaining speed from compilation.

Modern Python implementations even use extra optimizations. This includes

- JIT (Just-In-Time compilation) in PyPy
- C-accelerated modules (NumPy, CPython internals)
- aggressive caching (.pyc files)
- dynamic recompilation of hot code paths

These make execution much faster than early interpreters.

- **JITs (Just-In-Time Compilers):** A Just-In-Time (JIT) compiler is a component of a runtime system that compiles program code while the program is executing. Unlike a traditional ahead-of-time (AOT) compiler that produces machine code before execution, a JIT compiler generates optimized machine code on demand during runtime.

A JIT compiler generates native machine code in memory only, during execution.

That code:

- lives in RAM,
- is used by the CPU immediately,
- disappears when the program exits.

A JIT

- Compiles only the "hot" (frequently executed) sections of the program.
- Leaves cold code interpreted.

The program never becomes a complete native binary.

- **Compilers and interpreters:** Theoretically, both an assembler and a compiler produce binary machine code. An interpreter allows code to be executed directly, without compilation. In reality, life has become more complicated than that

Compilers have adopted some interpreted features. For example, binary code can be augmented with additional debugging information, allowing interactive debugging

Interpreters have adopted some features of compilers. Usually the interpreter produces some sort of intermediate representation (e.g., Java bytecode) to allow efficient execution. People have written compilers, e.g., JIT, for Java bytecode

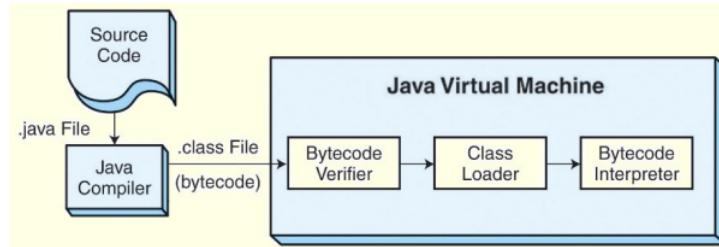
#### 6.6.4 Java case study

- **JVM:** Java programs (classes) execute within a virtual machine, the Java Virtual Machine (JVM). This allows the language to run on any platform for which a virtual machine environment has been written.

Therefore Java is partially compiled, partially interpreted. The output of the compilation process is an assembly-like intermediate code (bytecode) that is interpreted by the JVM.

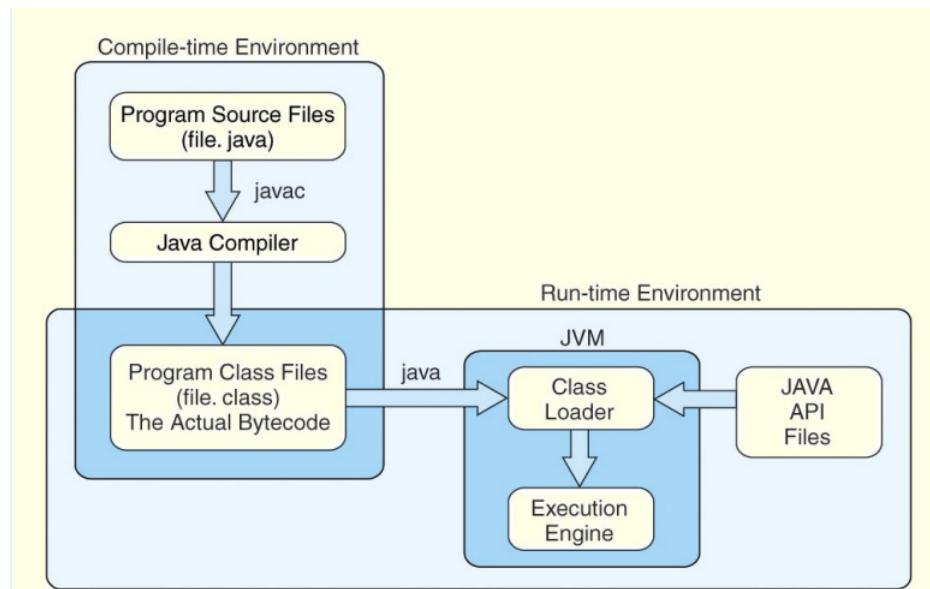
The JVM is an operating system in miniature. It loads programs, links them, starts execution threads, manages program resources, and deallocates resources when the programs terminate

Because the JVM runs on top of the operating system, its performance cannot match the performance of a traditional compiled language.



JVMs have been written for many processors, including MIPS and Intel. Like a real machine, the JVM bytecode has its own ISA. This ISA was designed to be compatible with the architecture of any machine on which one might want to run Java

Extra layers of software often cause a performance hit



At execution time, a Java Virtual Machine must be running on the host system. It loads and executes the bytecode class file

- While loading the class file, the JVM verifies the integrity of the bytecode.
- The loader then performs a number of run-time checks as it places the bytecode in memory.
- Finally, the loader invokes the bytecode interpreter.

- **The bytecode interpreter:**

1. Performs a link edit of the bytecode instructions by asking the loader to supply all referenced classes and system binaries, if they are not already loaded.
2. Creates and initializes the main stack frame and local variables.
3. Creates and starts execution thread(s).
4. Manages heap storage by deallocating unused storage while the threads are executing.
5. Deallocates resources of terminated threads.
6. Upon program termination, kills any remaining threads and terminates the JVM

Java bytecode is a stack-based language

- Most instructions are zero address instructions
- The JVM has four registers that provide access to five regions of main memory
- All references to memory are offsets from these registers.

- **Java performance:** Because the JVM loads and executes bytecode rather than executing a binary program, it can't match the performance of a compiled language. This is true even when software like Java's Just-In-Time (JIT) compiler is used.

However, Java performance has been steadily increasing over the years. For many applications, the difference is not important, since execution speed is not the limiting factor in an application. Human thinking and business processes are often the limiting factors in computer systems.

- **Advantages of interoperability:** An advantage of Java is that class files can be created and stored on one platform and executed on a different platform

This “write once, run-anywhere” paradigm is of enormous benefit for enterprises with disparate and geographically separate systems.

Java was designed for platform interoperability. Original slogan: “Write once, run anywhere”, took a while to achieve

Another advantage of the bytecode system is that other languages can be created that compile into Java bytecodes, Kotlin is one example.

That is much less work than building a full compiler.

- **Disadvantages of interoperability:** Interoperability has performance cost. Over the years people have started to build native mode compilers for Java (i.e., compiling bytecode instead of interpreting it) to improve performance

Interoperability can have a compatibility cost. You need to have correct version of Java Runtime Engine (JRE) on your machine to run a Java program. In practice, that is largely automatic today.

Java tries hard to keep new versions of the JRE backwards compatible so that they can run any Java program.