Computer Systems

Kasun Gunawardana

E-mail: kgg

University of Colombo School of Computing

Inner Workings of the CPU

Computer Architecture

- The term refers to the design of a computer system
    - Hardware components and their organization
    - The way these components interact with software
- It includes various aspects of a computer system
    - CPU
    - Memory
    - I/O devices
    - Buses

Notable Computer Architectures

- Harvard architecture (1930s) –
  - Separate memory spaces for instructions and data
- Atanasoff-Berry Computer (ABC) –
  - late 1930s and early 1940s by John Atanasoff and Clifford Berry
  - used a special binary system.
- Colossus –
  - used by the British during **World War II** to break German codes
  - used a combination of vacuum tubes, switches, and other components to perform its calculations.
- ENIAC (mid-1940s) –
  - used vacuum tubes to perform calculations
  - required a large amount of space and power to operate
  - needed manual programming (by setting switches and plugging and unplugging cables)

EDVAC – A Computer with Stored Programs

- Programming was difficult with earlier computers.
- Scientists explored mechanisms to represent the program in a stored form with data.
- Then, the processing unit could get its instruction by reading it from the stored program.
- In this way a program can be altered and set easily
- This idea is was proposed by John Von Neumann in 1945, in a paper titled "First Draft of a Report on the EDVAC"

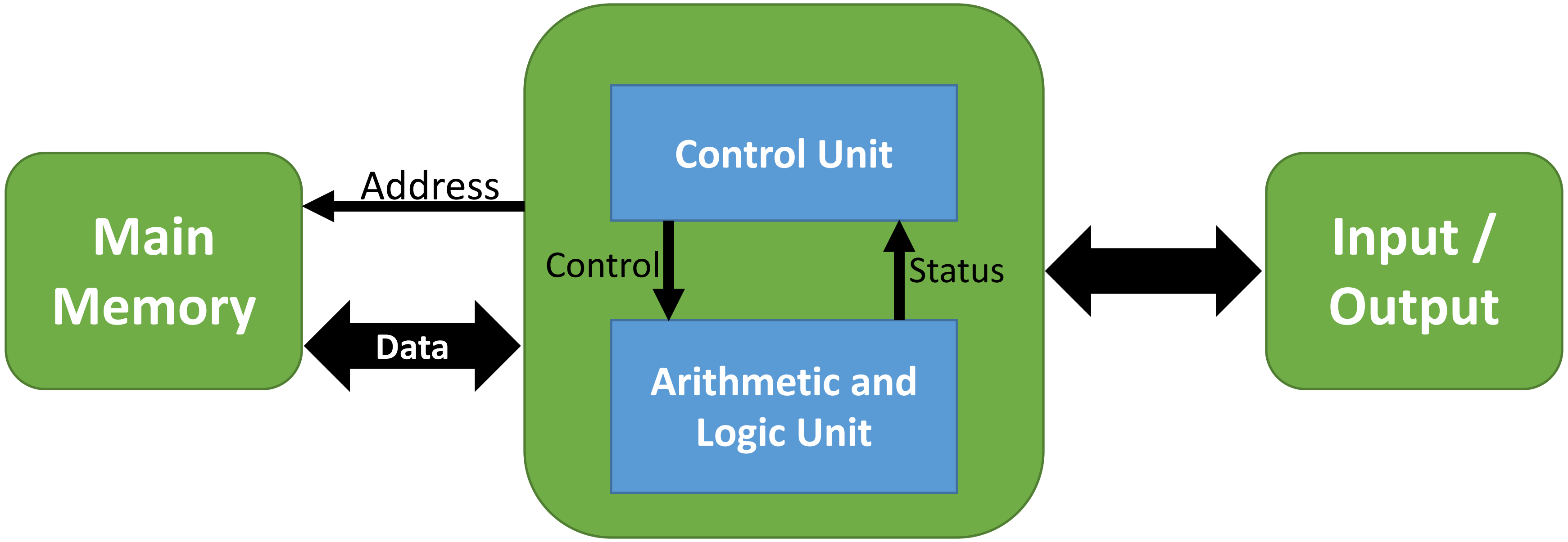
Von Neumann Architecture

1. The CPU is the brain of the computer that performs arithmetic and logic operations, and controls the flow of data between memory and other input/output devices.
   - Arithmetic and Logic Unit (ALU) and Control Unit (CU)
2. The computer's memory stores both data and instructions in a single address space. This allows the computer to execute instructions and access data in a unified way.
3. The Von Neumann architecture follows a fetch-execute cycle, where the CPU fetches an instruction from memory, decodes it, executes it, and then stores the result back in memory.

Von Neumann Architecture (Cont.)

4. The ISA is a set of instructions that the CPU can understand and execute. The Von Neumann architecture uses a single ISA that operates on both data and instructions stored in memory.
5. The stored program concept allows the computer to store both data and instructions in memory, which means that programs can be written and modified without physically changing the computer's hardware.
6. The I/O system allows the computer to communicate with external devices, such as keyboards, displays, and disk drives.

Von Neumann Architecture



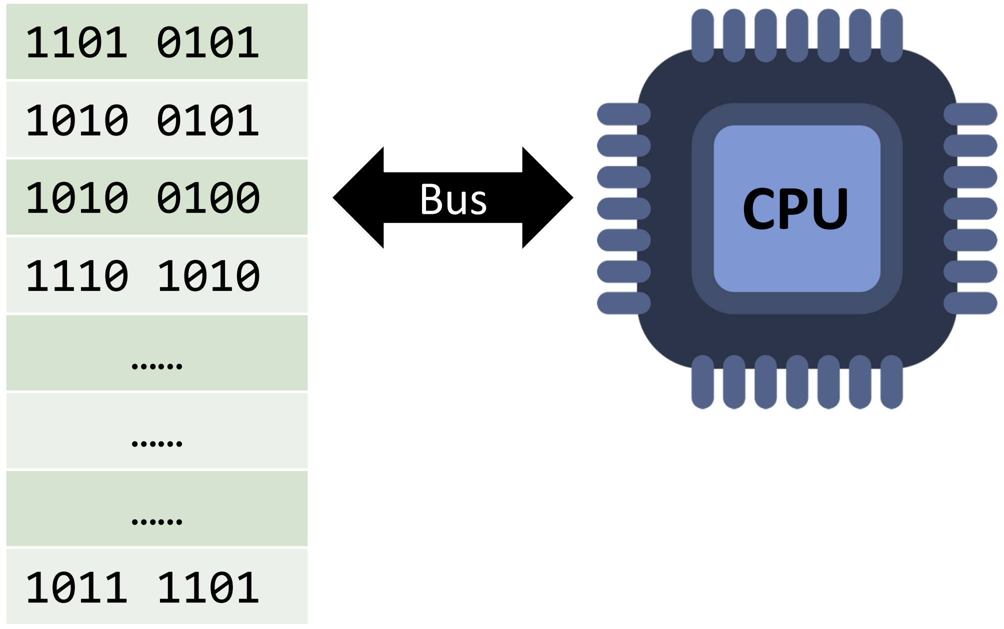
Components of Von Neumann Machine

- CPU
- Memory
- Input/ Output
- Bus
- Registers
- Instructions Set

Stored Program Concept

- A computer program is a series of instructions that tells the computer what to do.
- The program is stored in the same memory as the data that it operates on.
- The computer's processor reads the instructions from memory, interprets them, and executes them one at a time.
- This process is repeated until the program has completed its task.
- This concept allows a computer to be programmed to perform a wide range of tasks, simply by changing the program that is stored in memory.
- This makes a computer much more flexible and versatile than earlier computing machines, which had to be physically rewired to perform different tasks.
- This is the foundation of modern computer architecture, and it has enabled the development of a wide range of computer applications that have transformed many areas of modern life.

Von Neumann Machine



- **The program is stored in the same memory as the data that it operates on.**
- **CPU reads the instructions from memory, interprets them, and executes them one at a time.**
- **This process is repeated until the program has completed its task.**

Registers

- A register is a piece of hardware that can store binary information.
- Located on the CPU
- Faster memory access
- Registers can store wide variety of data
  - Instruction
  - Data
  - Address

Key Registers

MAR	Memory Address Register	Holds the memory location of data that needs to be accessed
MDR	Memory Data Register	Holds data that is being transferred to or from memory
AC	Accumulator	Where intermediate arithmetic and logic results are stored
PC	Program Counter	Contains the address of the next instruction to be executed
CIR	Current Instruction Register	Contains the current instruction during processing

MAR – Memory Address Register

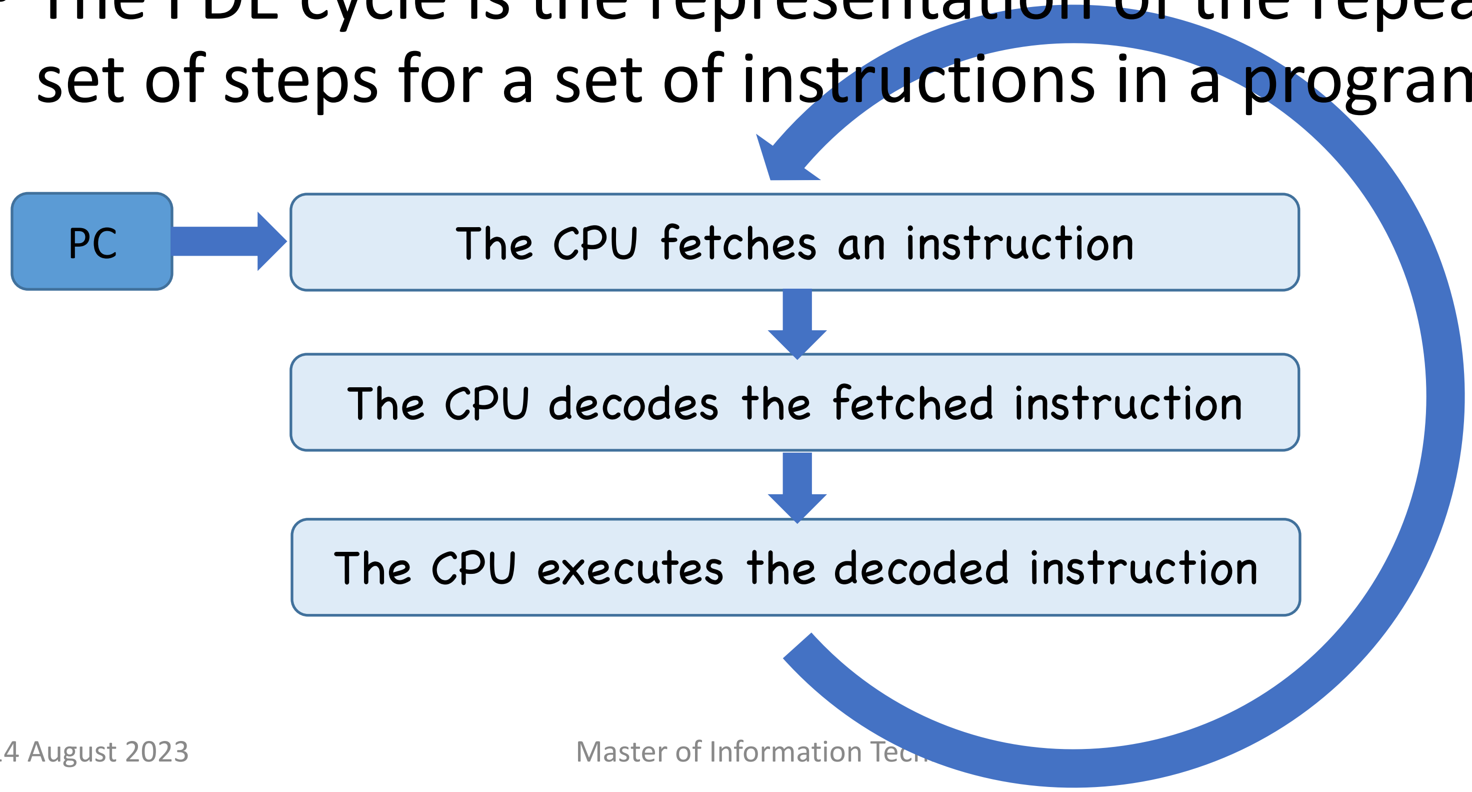
- Holds the memory address of the data or instruction that the CPU needs to access from memory.
- When a program is executed,
  - The CPU starts fetching the next instruction from memory using the PC
  - The address of the instruction is then loaded into the MAR
  - The IR in the CPU receives the instruction which is pointed by the address in the MAR
  - The CU decodes and identifies the operation encoded in the instruction that resides in the IR
  - The relevant circuitry is activated to execute the operation

Instruction Processing - Overview

1. The CPU fetches the next instruction from memory. The instruction is read from the program memory and loaded into the CPU's instruction register.
2. The CPU decodes the instruction. The instruction is interpreted by the CPU, and the necessary operands are fetched from memory and stored in the CPU's registers.
3. The CPU executes the instruction. The CPU performs the operation specified by the instruction, such as adding two numbers or comparing two values.
4. The CPU updates the program counter. After the instruction is executed, the program counter is incremented to point to the next instruction in memory.
5. The CPU repeats the process. The CPU fetches the next instruction, decodes it, executes it, and updates the program counter, repeating the process until the program is complete.

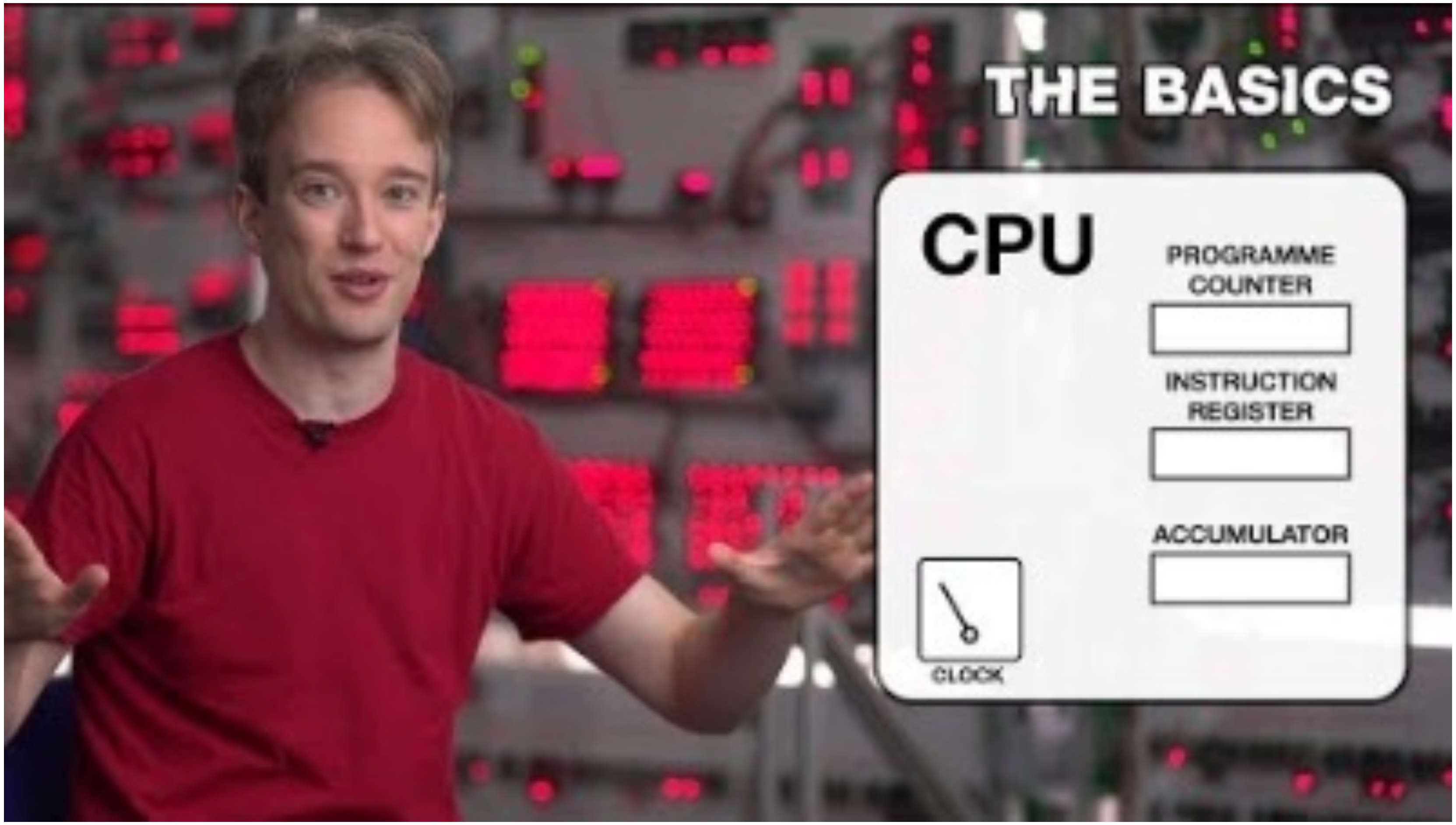
The Fetch-Decode-Execute Cycle

- All general purpose computers follow the same basic machine cycle (i.e. Fetch-Decode-Execute).
- The FDE cycle is the representation of the repeated set of steps for a set of instructions in a program.



PC
The CPU fetches an instruction
The CPU decodes the fetched instruction
The CPU executes the decoded instruction

The Fetch-Execute Cycle: What's Your Computer Actually Doing?



https://www.youtube.com/watch?v=Z5JC9Ve1sfI

Fetch Phase

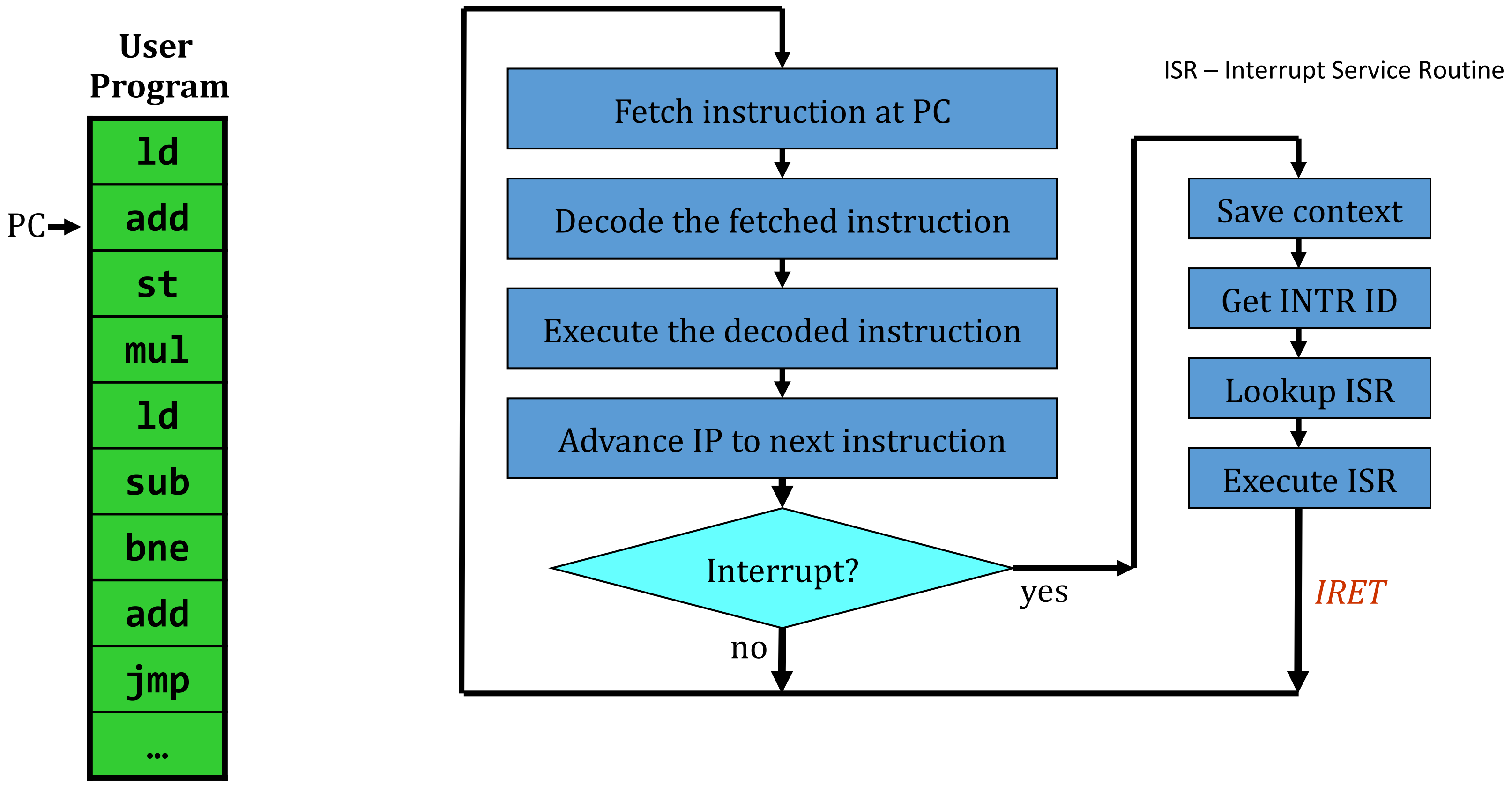
- Copy the contents of the PC to the MAR: MAR ← PC
- Fetch the instruction found at the address in the MAR and place it in the IR: IR ← M[MAR]
    - Increment PC by 1: PC ← PC+1

Note: PC is incremented by 1 if the memory system is word addressable. If the memory is byte addressable, and the word size is 2bytes, then the PC should be incremented by 2. If the memory is byte addressable with 32 bits words, then PC must be incremented by 4.

Decode and Execute Phases

- Control circuitry interprets the opcode portion of the IR

CPU Fetch-Decode-Execute Cycle



Instruction Set Architecture (ISA)

- The instruction set architecture of a machine specifies the instructions that the computer can perform and the format for each instruction.

Levels of Modern Computing Systems

- User
- High Level Language: C++, Java, etc.
- Assembly Code: Assembly Language
- ISA: Machine Language
- Hardware: Digital Logics

Example: Machine Architecture

- 256 byte Main Memory (`0x00 – 0xFF`)
  - Byte addressable memory
- 16 General Purpose Registers (`0x0 – 0xF`)
- 16 Bit Instruction
  - 4 bits for Operation Code (OpCode)
  - 12 bits for Operand (or other)
- 8 Bit Integer Format (2's Complement)
- 8 Bit Floating Point Format
  - 1 Sign Bit
  - 3 Exponent Bits
  - 4 Bit Mantissa
- 16 Instructions (`0x0 – 0xF`)

00	0001 0001
01	0011 0000
02	0001 0010
03	0100 0000
04	0011 0001
05	0100 0000
...	...
...	...
...	...
...	...
...	...
...	...
...	...
FF	0100 0000

Types of Machine Instructions

- **Data Transfer**
  - transfer data between registers and memory cells
- **Arithmetic/Logic Operations**
  - perform addition, AND, OR, XOR and etc.
- **Control Operations**
  - control the execution of the program

Data Transfer Instructions

`L    R,  A`	LOAD the register R with the content of memory cell A
`LI   R,  I`	LOAD the register R with I (I is called an immediate number)
`ST   R,  A`	STORE the content of the register R to the memory cell whose address is A
`LR   R1, R2`	LOAD the register R1 with the content of the register R2

E.g.- Data Transfer Instructions

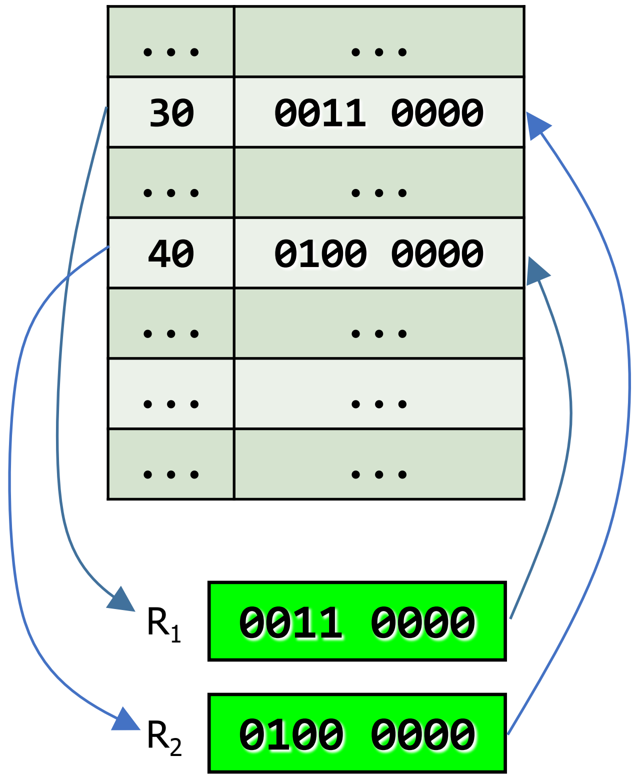
Swap the content of two memory cells 0x**30** and 0x**40**

L 1 , 30 /*Load R_1 with the content in memory cell 30 */

L 2 , 40 /* Load R_2 with the content in memory cell 40 */

ST 1 , 40 /* Store R_1 to 40 */

ST 2 , 30 /* Store R_2 to 30 */

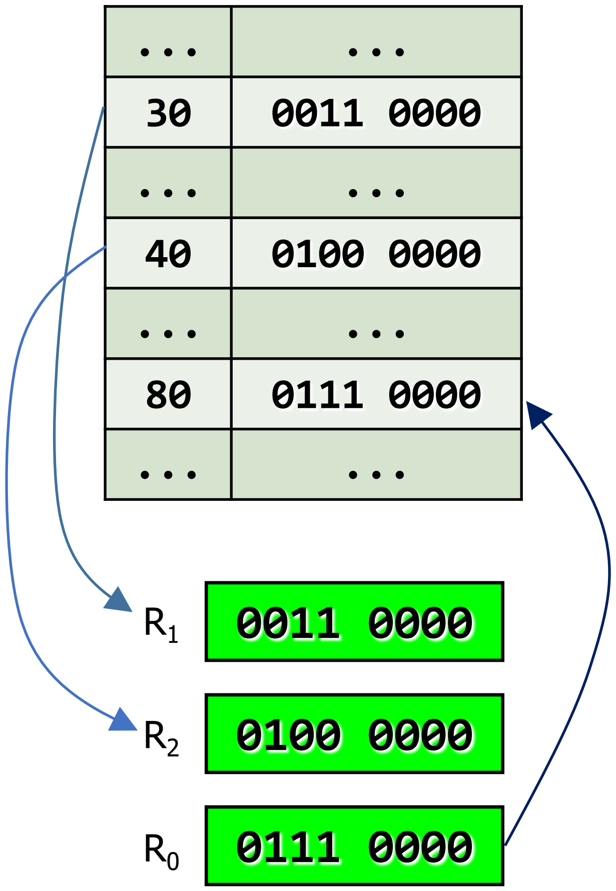


Arithmetic/ Logic Instructions

`ADD R0, R1, R2`	**ADD** the numbers in R1 and R2 representing in 2’s complement and place the result in R0
`AFP R0, R1, R2`	**ADD** the numbers in R1 and R2 representing in floating-point and place the result in R0

Arithmetic/ Logic Instructions

```
LOAD   R1 , 30

LOAD   R2 , 40

ADD    R0 , R1 , R2

STORE  R0 , 80
```



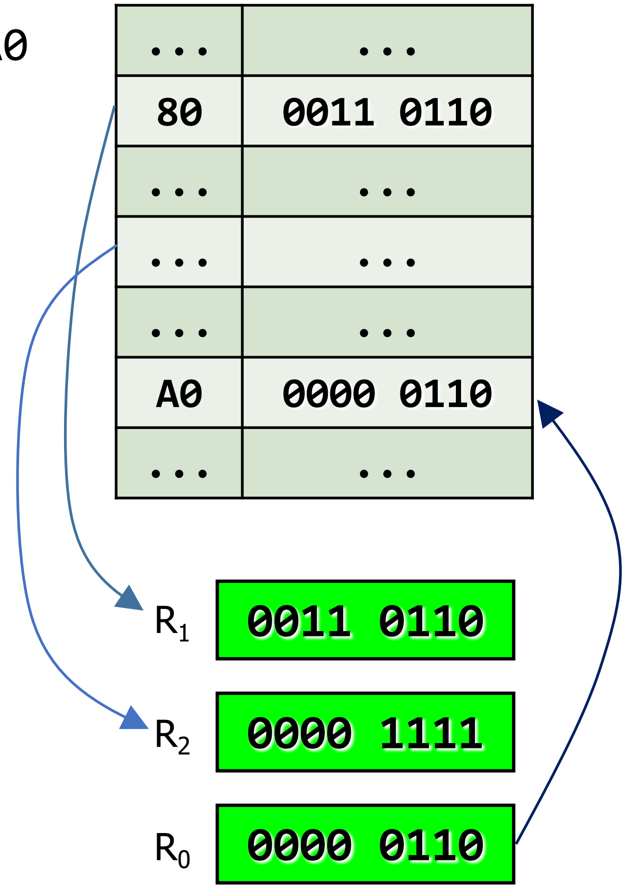
Arithmetic/ Logic Instructions

`OR  R0, R1, R2`	**OR** the bit patterns in R1 and R2 and place the result in R0
`AND R0, R1, R2`	**AND** the bit patterns in R1 and R2 and place the result in R0
`XOR R0, R1, R2`	**XOR** the bit patterns in R1 and R2 and place the result in R0

Arithmetic/ Logic Instructions

Mask the first 4 bits of the binary string in memory A0

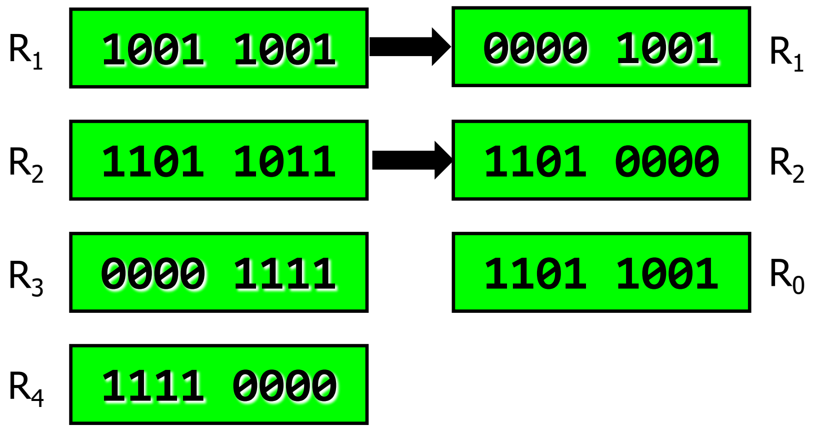
```
L     1,   80

LI    2,   0F

AND   0,   1, 2

ST    0,   A0
```



E.g. - Arithmetic/Logic Instructions

```
L     1, A0
L     2, A1
LI    3, 0F
LI    4, F0
AND   1, 1, 3
AND   2, 2, 4
OR    0, 1, 2
ST    0, E0
```

...	...
A0	1001 1001
A1	1101 1011
...	...
...	...
E0	1101 1001
...	...



Control Instructions

`JMP  R, A`	**JUMP** the instruction located in the memory cell A if the bit pattern in R is equal to the one in R0 (If equal reset the Program Counter to the Memory Address)
`HALT`	**HALT** the execution

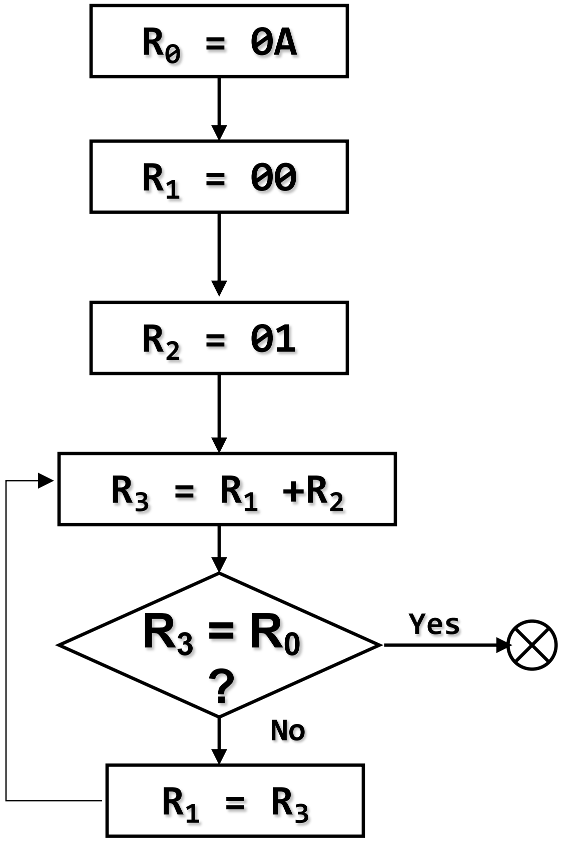
E.g. – Control Instructions

```
30  LI     0, 0A
32  LI     1, 00
34  LI     2, 01
36  ADD    3, 1, 2
38  JMP    3, 3E
3A  LR     1, 3
3C  JMP    0, 36
3E  HALT
```

R_0	0000 1010
R_1	0000 0000
R_2	0000 0001
R_3	0000 0001

Note - LR – Load Register, R_1, R_3

Copy content of R3 to R1



A Case Study

- Let’s assume an architecture,
  - Word addressable memory (word size = 16 bits)
  - Instructions are 16 bits long
    - 4 bits for OpCode
    - 4 bits are used for register identifier
  - Addresses are 8 bits long

A Case Study

- Let’s assume an architecture,
  - Word addressable memory (word size = 16 bits)
  - Instructions are 16 bits long
    - 4 bits for Opcode
    - 4 bits are used for register identifier
  - Addresses are 8 bits long

```
L     2, A1
LI    0, 0A
ADD   3, 1, 2
AND   1, 1, 3
JMP   3, 3E
LR    1, 3
ST    0, 3E
JMP   0, 36
HALT
```

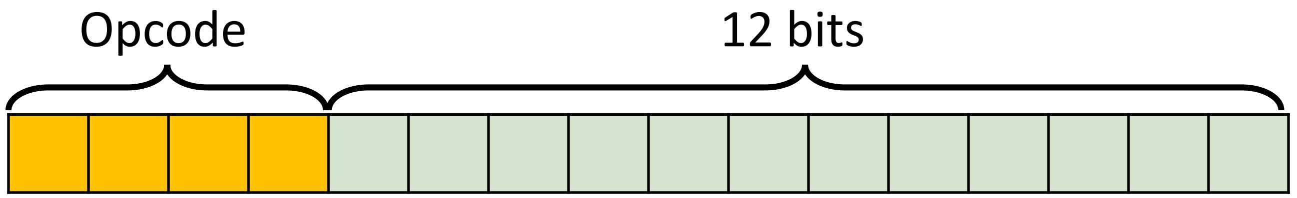
So how are we going to ensure all these instructions fit into 16 bits?

Quick Questions..

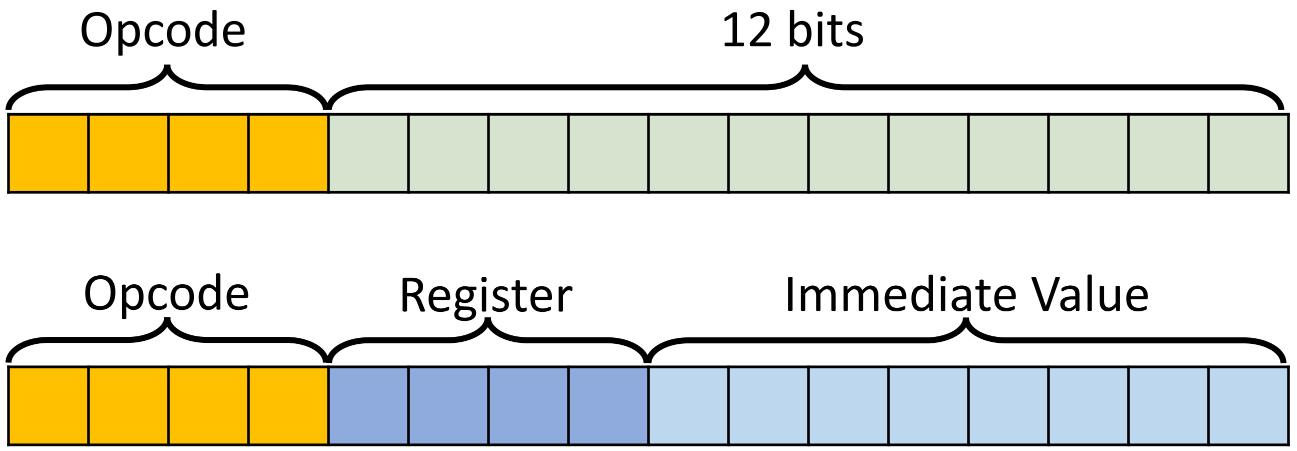
- What is the **maximum** number of **registers** this machine can have?

- What is the **largest memory** this machine can have?

Instruction Layout

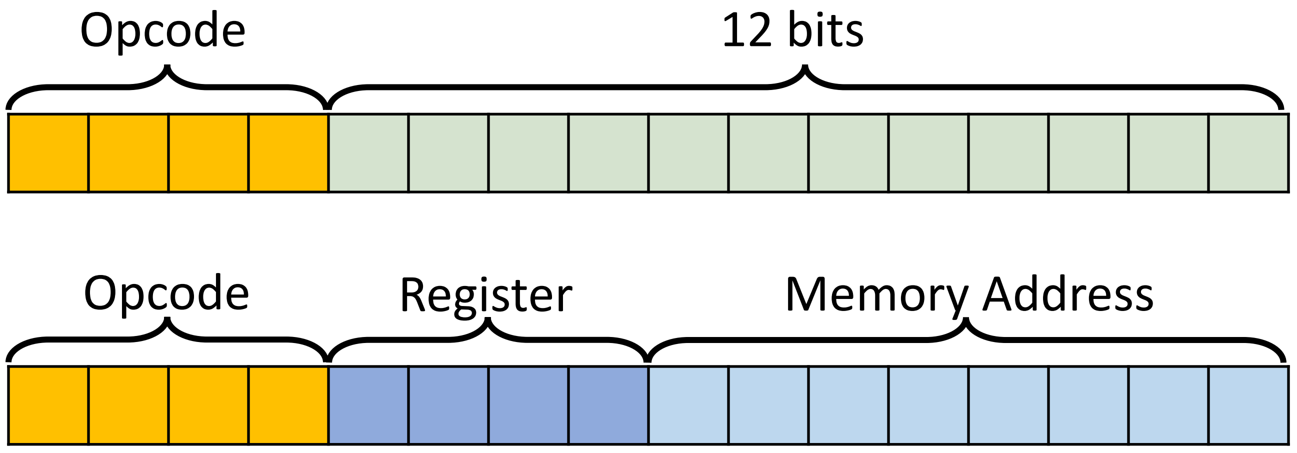


Instruction Layout: Type 1



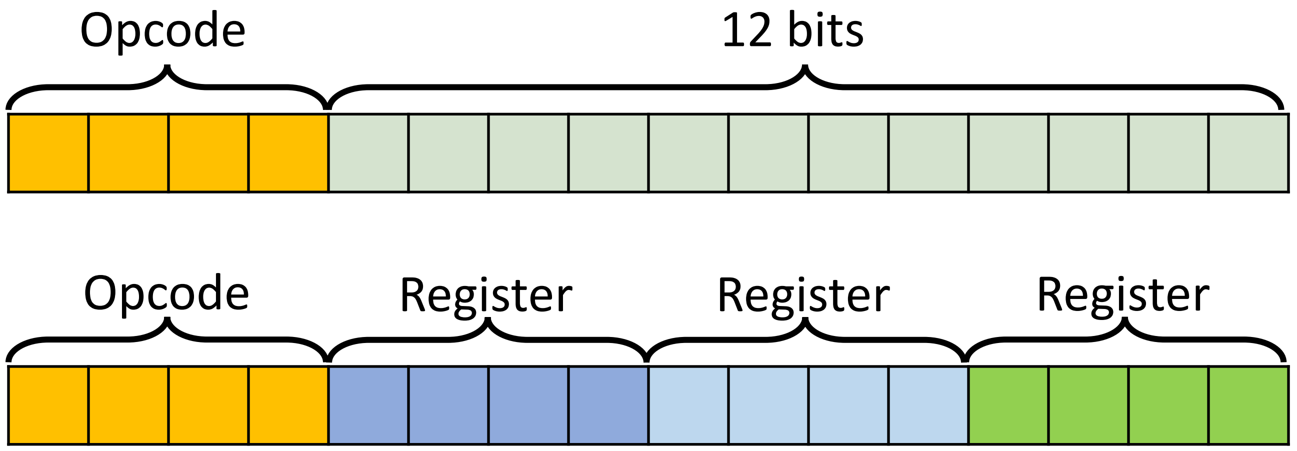
Opcode	Instruction	Meaning
0x2	`LI R, I`	Load Immediate
0xA	`RL R, I`	Rotate Left
0xB	`RR R, I`	Rotate Right
0xC	`SL R, I`	Shift Left
0xD	`SR R, I`	Shift Right

Instruction Layout: Type 2



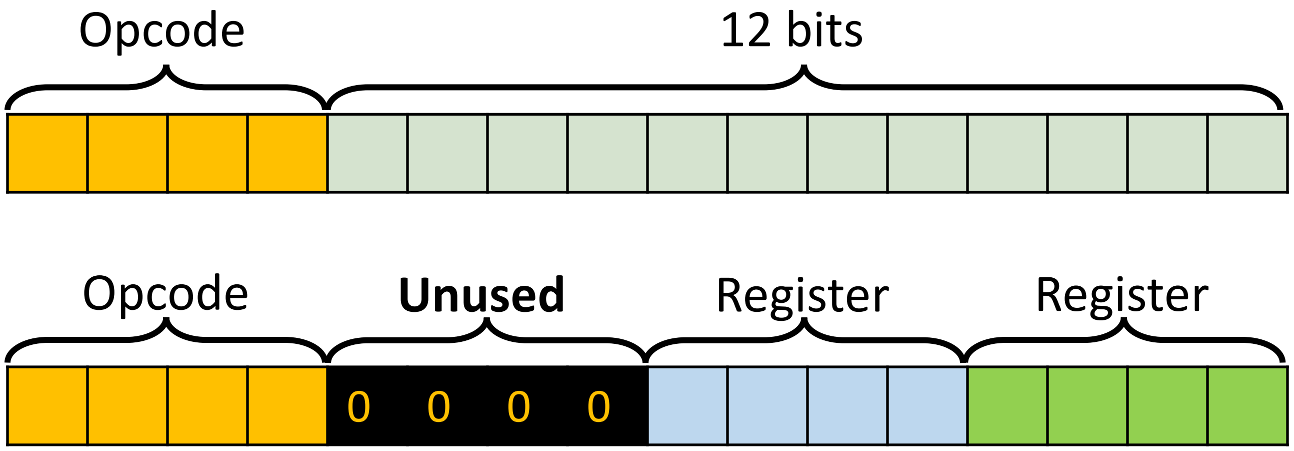
Opcode	Instruction	Meaning
`0x1`	`L R1, 0x11`	Load from a memory address
`0x3`	`ST R2, 0x12`	Store at a memory address
`0xE`	`JMP R3, 0x11`	Conditional jump

Instruction Layout: Type 3



Opcode	Instruction	Meaning
0x5	`ADD R1,R2,R3`	Addition (R1=R2+R3)
0x6	`AFP R1,R2,R3`	Floating point addition
0x7	`OR R1,R2,R3`	Logical OR
0x8	`AND R1,R2,R3`	Logical AND
0x9	`XOR R1,R2,R3`	Logical XOR

Instruction Layout: Type 4



Opcode	Instruction	Meaning
`0x4`	`LR R1, R2`	Load Register (R1=R2)

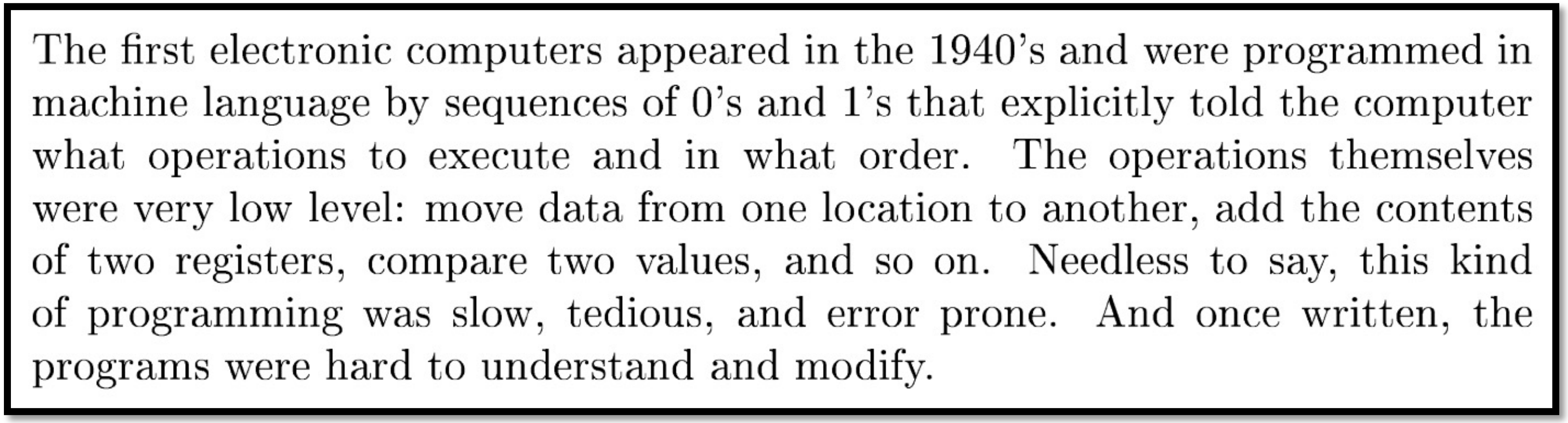
Full Instruction Set: Example Machine

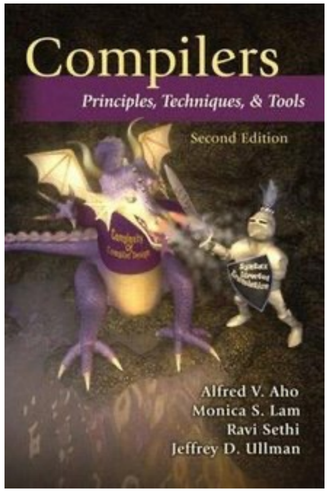
```
1. L     R, A                9. XOR   R0, R1, R2
2. LI    R, I                A. RL    R, I
3. ST    R, A                B. RR    R, I
4. LR    R1, R2              C. SL    R, I
5. ADD   R0, R1, R2          D. SR    R, I
6. AFP   R0, R1, R2          E. JMP   R, A
7. OR    R0, R1, R2          F. HALT
8. AND    R0, R1, R2         0. <Any additional Instruction>
                             MULT R0, R1,R2
```

Example

```
Assembler    Machine Code           Hexa
=========    ===================    ====
L    1, 30   0001 0001 0011 0000    1130
L    2, 40   0001 0010 0100 0000    1240
ST   1, 40   0011 0001 0100 0000    3140
ST   2, 30   0011 0010 0011 0000    3230
```

10	0001 0001
11	0011 0000
12	0001 0010
13	0100 0000
14	0011 0001
15	0100 0000
16	0011 0010
17	0011 0000
30	1001 1001
40	0110 1101


The first electronic computers appeared in the 1940's and were programmed in machine language by sequences of 0's and 1's that explicitly told the computer what operations to execute and in what order. The operations themselves were very low level: move data from one location to another, add the contents of two registers, compare two values, and so on. Needless to say, this kind of programming was slow, tedious, and error prone. And once written, the programs were hard to understand and modify.



Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
Compilers - Principles, Techniques, and Tools (2006)

Thank You..!