Basics of Computing – Chapter 2
Data Manipulation

Cory L. Strope

Computer Architecture

Machine Languages

Program Execution

Arithmetic/Logic Instructions

Device Communication
- Data/Communication

Computer Architecture

Machine Languages

Program Execution

Arithmetic/Logic Instructions

Device Communication
    Data/Communication

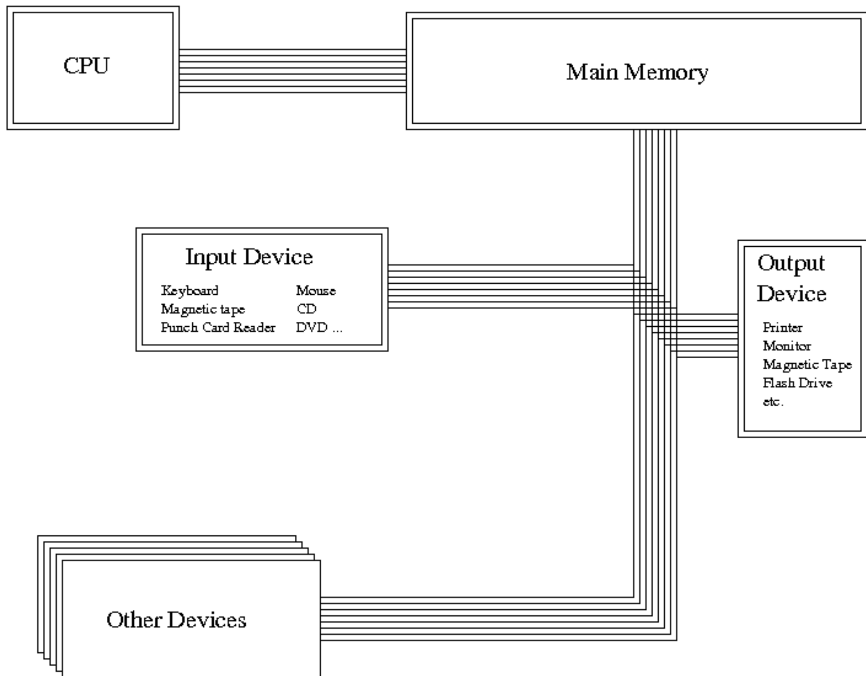
The basic components that define an electronic digital computer are:

- Central Processing Unit (CPU)
- Memory
- Input device
- Output device

Unnecessary components:

- Video card
- Modem
- Mass storage (hard drive)

An example of a basic computer is a Calculator.


CPU
Main Memory
Input Device
Keyboard
Mouse
Magnetic tape
CD
Punch Card Reader
DVD ...
Output Device
Printer
Monitor
Magnetic Tape
Flash Drive
etc.
Other Devices

Central Processing Unit (CPU)

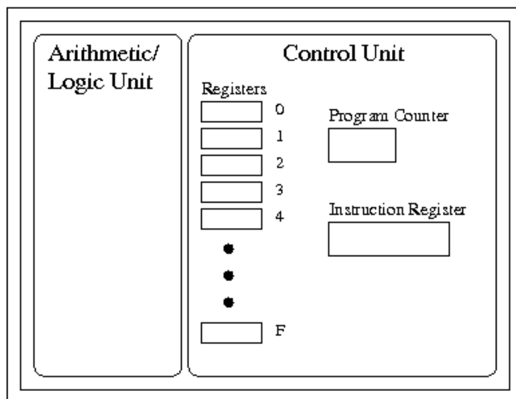
The CPU is essentially the “brain” of the computer, consisting of two parts:

- Arithmetic/logic unit (ALU)
- Control unit

The CPU also has **registers**, which are memory for the immediate future instructions.

- General purpose
- Special purpose
  - Program counter
  - Instruction register

Central Processing Unit (CPU)



Main Memory

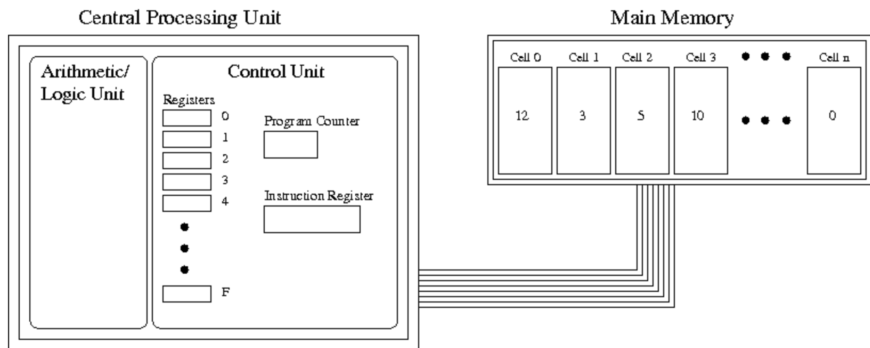
Main Memory:

- A sequence of addressed cells
- Stores information that will be needed in the near future
- Not to be confused with mass storage, which holds data that will likely not be needed in the immediate future.

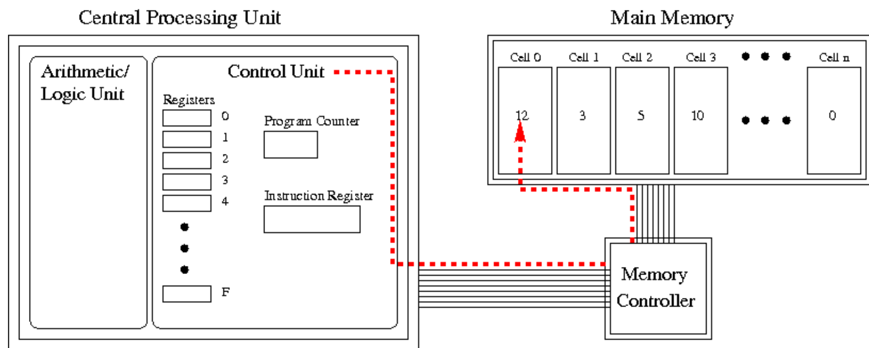
There are two types of information that can be used:

- Data – Information that is manipulated
- Instructions – a list of operations for the CPU to perform

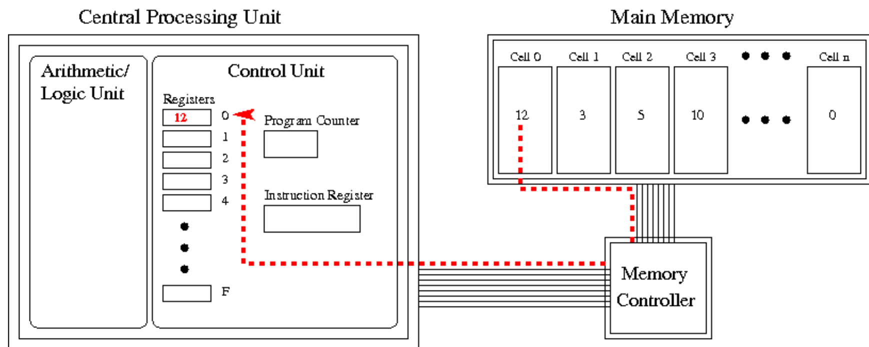
Setup



Step 1: Control Unit Fetches Number from Address Cell 0



Step 2: Memory Returns 12 to First Register (Register 0).



Step 3: Repeat Process to get Second Number.

Central Processing Unit

Arithmetic/
Logic Unit

Control Unit

Registers

12 0
10 1
2
3
4
F

Program Counter

Instruction Register

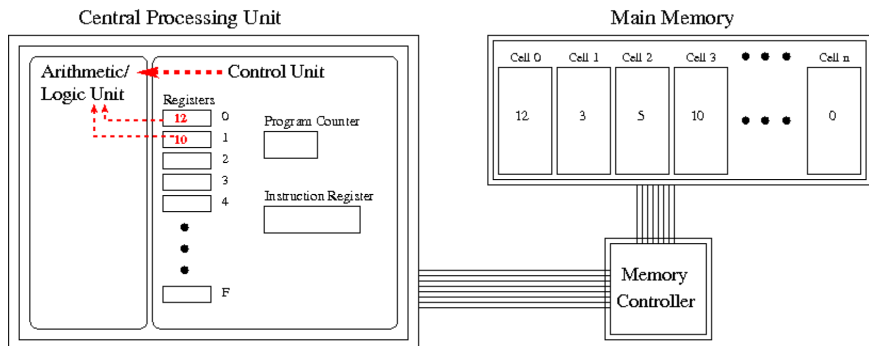
Main Memory

Cell 0 Cell 1 Cell 2 Cell 3 Cell n

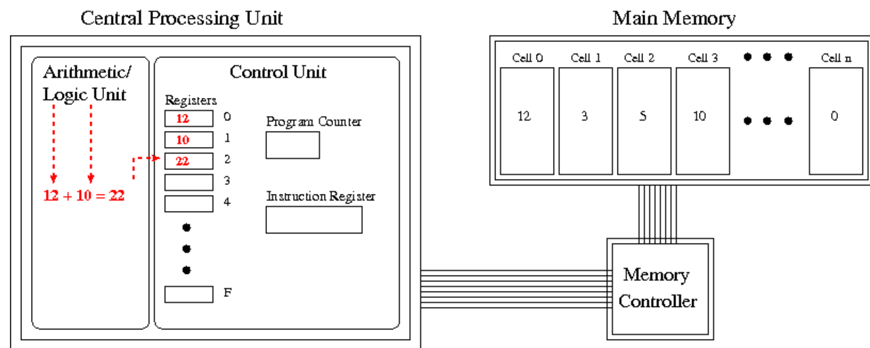
12 3 5 10 0

Memory
Controller

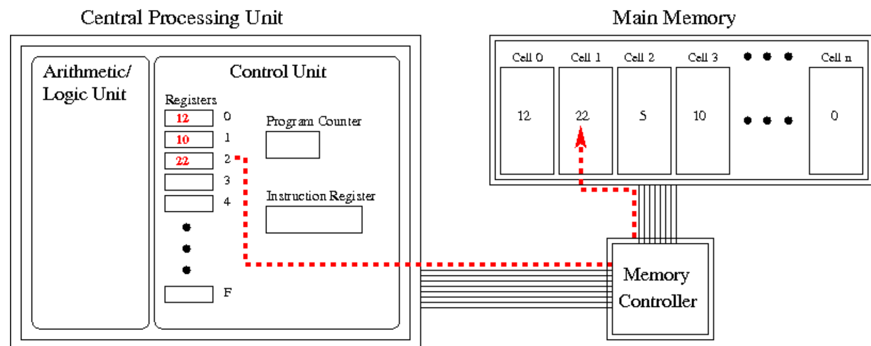
Step 4: Control Unit Activates Addition Circuit.



Step 5: ALU Receives Registers 0 & 1 as Input, Writes Result in Register 2.



Step 6: Store Result in Register 2 to Memory.



Computer Architecture

Machine Languages

Program Execution

Arithmetic/Logic Instructions

Device Communication
Data/Communication

Overview

1. Introduction to Computer Languages
2. Machine Language Design Issues
3. An Example of Machine Language

Language

A Definition

From American Heritage Dictionary (Dictionary.com):

> *Communication of thoughts and feelings through a system of arbitrary signals, such as voice sounds, gestures, or written symbols.*

Or, specifically for computer science:

> *A system of symbols and rules used for communication with or between computers.*

Symbols $\in \{0, 1\}$. Everything is encoded as a 0 or a 1, even rules.

Programming Language
Instructions

Languages provide a framework for us to construct useful statements

- These statements are called **instructions**.

http://ascii.ws/

Instructions:

1. Enter a URL of an image (.gif / .jpg / .png) and click SET
2. Choose your Settings
3. Click RENDER

For example, instructions encoded in the symbols 0 and 1 in machine language

1000010111010100011010101101010100

In English:

Add 7 and 12

Machine Language

The language of computers: **Machine Language**

> *Instructions that a computer can execute directly. Machine language statements are written in a binary code, and each statement corresponds to one machine action. Programs are efficient to run, but the absence of words in commands makes for difficult programming.*

We often represent binary strings in hexadecimal.

There is no uniform standard for machine languages.

- Macintosh vs. PC

Assembly Language

> *but the absence of words in commands makes for difficult programming...*

Because it is difficult to understand machine language, **assembly language** was created to make it easier for programmers.

Machine language: 0011010111111110 (35FE)

vs. Assembly: ST R5, price (ST = store)

Computers do not understand assembly language.

- A converter (called a **compiler**) reads the assembly language code and translates the instructions to machine language.

High-Level Languages

Assembly languages also have problems:

- Machine dependent
- Difficult to understand.

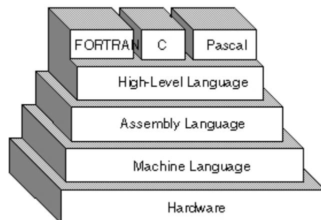
High-Level Languages are the current standard for programming:

> *A programming language that [...] may be more abstract, easier to use, or more portable across platforms. i.e. A problem-oriented language requiring little knowledge of the computer on which it will be run.*

Some examples: C++, Ada, Java, Ruby, PERL, Pascal, Basic, C#, COBOL, PHP, HTML, CGI, etc.

Languages

Comparison



C source code:

```
char name[40];
printf("Please enter your name\n");
scanf("%s", name);
printf("Hello %s", name);
```

Assembly Code

```
push      offset string "Please enter your name\n"
(41364Ch)
call      dword ptr [__imp__printf (415194h)]
add       esp,4
lea       eax,[name]
push      eax
push      offset string "%s" (413648h)
call      dword ptr [__imp__scanf (41519Ch)]
add       esp,8
lea       eax,[name]
push      eax
push      offset string "Hello %s" (41363Ch)
call      dword ptr [__imp__printf (415194h)]
add       esp,8
```

Machine Code:

```
68 4C 36 41 00 FF 15 94 51 41 00 83 C4 04 8D 45 D8
50 68 48 36 41 00 FF 15 9C 51 41 00 83 C4 08 8D 45
D8 50 68 3C 36 41 00 FF 15 94 51 41 00 83 C4 08
```

Programs

A **program** is a collection of instructions that do a required task.

- A program that adds two numbers and stores the result in a specific location in memory.
- Computer games.

There are two paradigms for making programs:

1. Build hardware to do a specific task.
2. Store programs that do the specific task.

Programs
Built-in Program Approach

In the **Built-in** approach, the steps of the program are built into (the program is stored in) the control unit as part of the machine.

Ms. Pacman, current WR score: 933580

- Makes the design of the machine simple
- Faster execution time

However,

- Machine is inflexible.

To gain flexibility, we need to make the program outside of the hardware.

Programs
Stored Program Concept

The **stored-program concept** came from the realization that the program instructions can be stored in Main memory, just as data is.

- A computer program can be changed by changing the contents of the computer's memory instead of reprogramming the control unit.

The CPU deals with each instruction one after another, and keeps track of which instruction will be executed next.

Machine Languages

A Closer Look

In order to make computers work for us, we need:

1. A set of instructions, and
2. What the computer should do for each instruction.

To apply this to the stored program concept, CPUs are designed to recognize instructions encoded as bit patterns.

- This collection of instructions along with the encoding system is called the **machine language**.
- An instruction expressed in this language is called a machine-level instruction (**machine instruction**).

Machine Language

Machine Instructions

Machine Language: Specifies the collection of instructions along with how they are encoded in binary.

- The machine language we will consider can be found in Appendix C.

Machine Instruction: An instruction expressed in the machine language.

Example: The instruction below:

0001010101101100

or (156C) means: "Load register 5 with the bit pattern found in memory location 6c."

Machine Language

0001010101101100 (156C)

A Machine instruction (such as the one above) needs to contain:

1. The instruction rule (Load)
2. The register number of the CPU
3. The location of the number in the memory.

Machine Language
Instruction Components

Machine language instructions are divided into two parts:

1. Op-code
2. Operand field.

$$\underbrace{0001}_{op-code}\ \underbrace{010101101100}_{operand\ \ field}$$

- The op-code specifies the instruction type (Add, Load, etc.)
- The operand field is divided into subparts specifying memory locations and register numbers.

Overview

What makes a "good" machine language?

A good machine language should be:

1. **Complete:** The machine language contains all of the instructions a program needs to perform its required task,
2. **Orthogonal:** The machine language does not contain multiple instructions that accomplish the same thing.

Design Issues

Three main design issues affect the stucture of any machine language:

1. **Number of Instructions:** How many instructions can the processor understand?
2. **Processor Architecture:** How many registers are there in the CPU?
3. **Memory Structure:** How many cells in main memory?

Each of these issues affect the op-code. How?

Design Issues

The design issues may conflict with each other...

- Many instructions:
  - ⟶ Computer can do more things with fewer instructions.
  - but Longer op-code,
    Instructions are longer,
    More complex circuitry is needed (slow operation)
- Fewer instructions:
  - ⟶ Simple circuitry (fast operation)
  - and Short instructions,
  - but Need more instructions to do the same task.

Design Issues
Number of Instructions

Suppose we want to build an artificial machine that adds and multiplies. We can:

1. Implement two digital circuits, one for addition and one for multiplication, or
2. Implement addition, and convert the multiplication process into addition.

$$3 \times 4 = 3 + 3 + 3 + 3$$

Trade-off: Simplicity, speed & redundancy.

Design Issues

Number of Instructions

This gives rise to two main CPU design philosophies:

1. **CISC (Complex Instruction Set Computers)**:
   The CPU has the ability to execute a large number of complex instructions, even though many of them are technically redundant.
   - Ex: Pentium and Athlon series of processors (by Intel and AMD, respectively).
2. **RISC (Reduced Instruction Set Computer)**:
   The CPU executes a minimal set of machine instructions.
   - Ex: PowerPC series of processors (by Apple Computer, IBM, Motorola).

Design Issues

Machine Instruction Types

A typical machine language should contain basic instructions that the machine needs to process and store data.

$$\text{Instruction Types} = \begin{cases} \text{Data Transfer} = \begin{cases} \text{Memory} \\ \text{I/O Devices} \end{cases} \\ \\ \text{Arithmetic/Logic} = \begin{cases} \text{Add, etc.} \\ \text{AND, OR, etc.} \end{cases} \\ \\ \text{Control} = \begin{cases} \text{JUMP} \\ \text{BRANCH} \end{cases} \end{cases}$$

Virtual Computer

And a Warning...

From this point on, we will describe and make references to a "virtual computer".

This "computer" consists of three pieces:

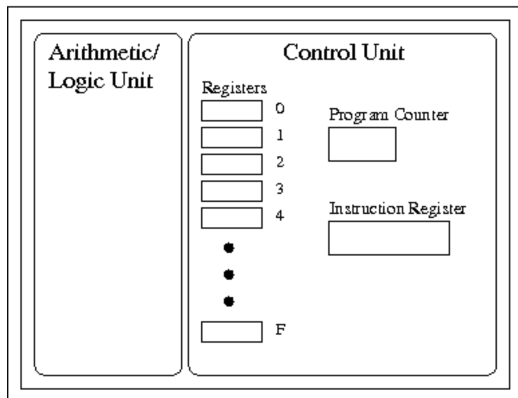
1. Processor
2. Memory
3. I/O Devices

Note:

- This "computer" will serve only as a learning aid.
- Not all computers adhere to the same specifications!

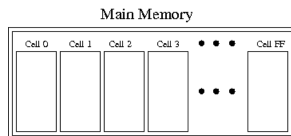
Virtual Computer

Processor Architecture



- **Instruction Register (IR):** The register that holds the instruction that the CPU is *currently executing*.
- **Program Counter (PC):** The register that keeps track of the next instruction that the CPU should execute.

Virtual Computer
Memory Structure



- Each cell in memory can hold 8 bits.
- Each cell has an address (written in hexadecimal for simplicity).
- Main memory contains 256 cells.

Virtual Computer

Input/Output Devices

For our virtual computer:

- We assume we can view the main memory.
- We place instructions in the main memory and view the results in either memory or in the registers.

In essence, we are the I/O devices.

An Illustrative Machine Language

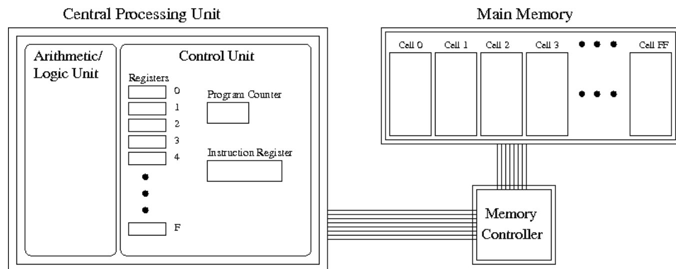
Using the *artificial* machine language given in Appendix C, we have:

1. **Number of Instruction:** There are only 12 instructions. Op-code length? ☐
2. **Processor Architecture:** The CPU has 16 registers, requiring a register address of ☐ bits.
3. **Memory Structure:** The main memory has 256 cells. The length of the address of each cell is ☐ bits.

The final length of each instruction is ☐ bits.

An Illustrative Machine Language

The instruction register can hold 16 bits, each memory cell and general purpose register can hold 8 bits.



An Illustrative Machine Language

Instruction Components

We can now define an instruction:

Total Instruction Length: 16 bits (Same length as IR)

- **Op-code Length:** 4 bits
- **Register Address Length:** 4 bits (16 registers)
- **Memory Address Length:** 8 bits (256 memory cells)

Each of the above instruction portions has a length divisible by 4 $\longrightarrow$ Hexadecimal notation.

Thus, for the 16 bit instruction 0001011000111111:

$$\underbrace{0001}_{1}\underbrace{0110}_{6}\underbrace{0011}_{3}\underbrace{1111}_{F}$$

An Illustrative Machine Language

Order of Operations

During execution, the computer goes through the following cycle:

1. Fetch – Request the instruction from main memory and place it in the instruction register (IR).
2. Decode – The control unit decodes the instruction and sets up the data paths in the processor to execute the instruction.
3. Execute – Execute the instruction.
4. Repeat.

Creatively, this is named the **Fetch, Decode and Execute cycle**.

Appendix C Instructions

Instruction Breakdown:

- Four data transfer instructions:
  - Load, Load Immediate, Store, Move
- Six arithmetic/logic instructions:
  - Add integer, Add FP, OR, AND, XOR, Rotate
- One control instruction
  - JUMP
- The instruction HALT

Register Transfer Notation (RTN)

Showing Machine Instructions Actions

Register Transfer Notation is a method to show the actions taken by each step in the program.

- **R[R]:** Denotes a Register
- **M[XY]:** Denotes a Memory Cell
- <**left**> ← <**right**>**:** Set the Register or Memory location on <left> to the value of the expression on <right>.
- **PC:** Denotes the Program Counter
- **IR:** Denotes the Instruction Register

For the following machine instructions, the letters R, S, and T will be used to stand for register numbers, and X and Y will be used to denote memory locations and numbers.

LOAD

- Op-code: 1
- Format: 1RXY
  - LOAD the register R with the bit pattern found in the memory cell whose address is XY.
- RTN: R[R] $\leftarrow$ M[XY]
- Example: 14A3 (R[4] $\leftarrow$ M[A3])
  - LOAD R[4] with the contents of M[A3].

LOAD Immediate

- Op-code: 2
- Format: 2RXY
  - LOAD the register R with the bit pattern XY.
- RTN: R[R] $\leftarrow$ XY
- Example: 24A3 (R[4] $\leftarrow$ A3)
  - LOAD R[4] with the hexadecimal value A3 (10100011).

STORE

- Op-code: 3
- Format: 3RXY
  - STORE the bit pattern found in register R in the memory cell whose address is XY.
- RTN: M[XY] ← R[R]
- Example: 3C15 (M[15] ← R[C])
  - STORE the bit pattern in R[C] in M[15].

MOVE

- Op-code: 4
- Format: 40RS
  - MOVE the bit pattern found in register R to register S.
- RTN: R[R] $\leftarrow$ R[S]
- Example: 405E (R[5] $\leftarrow$ R[E])
  - MOVE the bit pattern in R[5] to R[E].
- Always has a 0 in the second hex character.
- Name is misleading. COPY would be more accurate.

ADD INTEGER

- Op-code: 5
- Format: 5RST
  - ADD the bit patterns in registers S and T as though they were two's complement representations and place the result in register R.
- RTN: $R[R] \leftarrow R[S] +_2 R[T]$
- Example: 5423 ($R[4] \leftarrow R[S] +_2 R[T]$)
  - ADD the two's complement bit patterns in R[2] and R[3] and place the result in R[4].

ADD FP

- Op-code: 6
- Format: 6RST
  - ADD the bit patterns in registers S and T as though they were floating point representations and place the result in register R.
- RTN: $R[R] \leftarrow R[S] +_{FP} R[T]$
- Example: 64AE ($R[4] \leftarrow R[A] +_{FP} R[E]$)
  - ADD the FP bit patterns in R[A] and R[E] and place the result in R[4].

OR

- Op-code: 7
- Format: 7RST
  - OR the bit patterns in registers S and T and place the result in register R.
- RTN: R[R] ← R[S] OR R[T]
- Example: 7C26 (R[C] ← R[2] OR R[6])
  - OR the bit patterns in R[2] and R[6] and place the result in R[C].

AND

- Op-code: 8
- Format: 8RST
  - AND the bit patterns in registers S and T and place the result in register R.
- RTN: R[R] $\leftarrow$ R[S] AND R[T]
- Example: 8534 (R[5] $\leftarrow$ R[3] AND R[4])
  - AND the bit patterns in R[3] and R[4] and place the result in R[5].

XOR

- Op-code: 9
- Format: 9RST
  - XOR the bit patterns in registers S and T and place the result in register R.
- RTN: R[R] ← R[S] XOR R[T]
- Example: 9CAB (R[C] ← R[A] XOR R[B])
  - XOR the bit patterns in R[A] and R[B] and place the result in R[C].

ROTATE

- Op-code: A
- Format: AR0X
  - ROTATE the bit pattern in register R one bit to the right X times. Each time, take the least significant bit and place it as the most significant bit.
- RTN: ROTATE(R[R], X)
- Example: A301 (ROTATE(R[3], 1))
  - ROTATE the bit pattern in R[3] one bit to the right 1 time.

$$01110001 \longrightarrow 10111000$$

- The third hex character will always be 0.

JUMP

- Op-code: B
- Format: BRXY
  - If the bit patterns in registers R and 0 are the same, then JUMP to the instruction at memory location XY. Otherwise, continue with the normal sequence of instructions.
- RTN: If branch is true, PC = XY. If not, Nothing.
- Example: B212
  - If the bit patterns in R[2] and R[0] are the same, then JUMP to the instruction at M[12]. Otherwise, do nothing.
- The JUMP is accomplished by placing the bit pattern XY in the Program Counter (PC).

HALT

- Op-code: C
- Format: C000
  - Stop execution.
- RTN: HALT

Virtual Computer
Programming

We will now build machine language programs. But, before we do so:

1. The CPU can access only the bits that are stored inside its registers.
2. Registers in the CPU hold data temporarily. Thus, to "store" the result of adding two numbers, the CPU should STORE this result in main memory.
3. Every program we write should end with the HALT instruction (C000), so the CPU knows we are done.

Virtual Computer

Writing a Program

Write a program in the machine language from Appendix C that adds the two's complement numbers located at memory addresses 6C and 6D in memory. Store the result in memory cell 6E.

Step 1: LOAD the first number from the memory cell 6C to one of the registers in the CPU, call it register 2.

Step 2: LOAD the second number from the memory cell 6D to another register in the CPU (R3).

Step 3: ADD the contents of the two registers and place the result in a third register (R0).

Step 4: STORE back the result from the third register into the memory cell 6E.

Step 5: HALT.

Virtual Computer

Writing a Program

Step 1:	LOAD the first number from the memory cell 6C to one of the registers, call it register 2	⟶ 126C
Step 2:	LOAD the second number from the memory cell 6D to another register in the CPU, (R3).	⟶ 136D
Step 3:	ADD the contents of the two registers and place the result in a third register, (R0).	⟶ 5023
Step 4:	STORE the result from the third register into the memory cell 6E	⟶ 306E
Step 5:	HALT	⟶ C000

Virtual Computer

Next Steps:

We have just written a small program in the machine language from Appendix C. Now, we will examine how the CPU *executes* the program we wrote.

Computer Architecture

Machine Languages

Program Execution

Arithmetic/Logic Instructions

Device Communication
    Data/Communication

The Machine Cycle

Machine Cycle: A three-step process that the control unit in the CPU performs every time it executes an instruction:

1. Read the instruction from memory (Fetch).
2. Translate the instruction (Decode).
3. Perform the instruction (Execute).

Note: This is done with the **stored-program** concept. We will assume that we know the address of the cell that has the first instruction of the program. Later, we will learn that the Operating System actually does this.

The Machine Cycle

Fetch	Decode	Execute
Retrieve the instruction from memory (as indicated in the PC) and increment the PC (i.e. point it to the next instruction in the program).	Decode the bit pattern in the instruction register.	Perform the action requested by the instruction in the instruction register.

Repeat until reaching the HALT instruction.

The Machine Cycle

Comments about the Program Counter

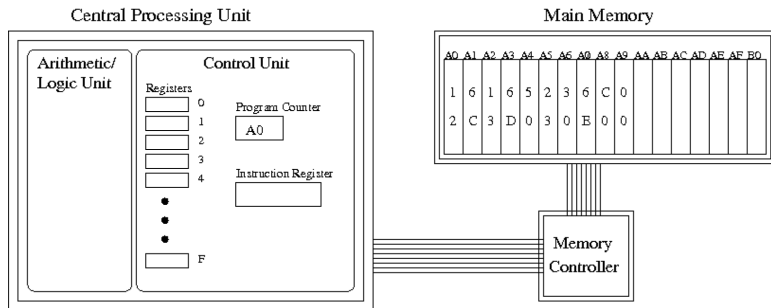
- The CPU increments the program counter *at the same time* it reads the current instruction.
- When the CPU starts a new cycle, it finds the address of the next instruction (already in the PC).
- The architecture of the machine defines the amount by which the PC is incremented.
- In our machine, each instruction will be stored in **two cells** in the memory. Why?

The Machine Cycle

Comments about the Program Counter

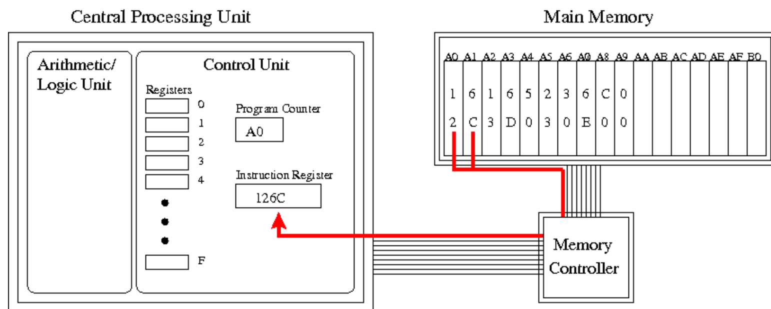
- The CPU increments the program counter *at the same time* it reads the current instruction.
- When the CPU starts a new cycle, it finds the address of the next instruction (already in the PC).
- The architecture of the machine defines the amount by which the PC is incremented.
- In our machine, each instruction will be stored in **two cells** in the memory. Why?
- Each instruction is **16 bits** long, while each cell in the memory can only hold 8 bits. Thus, to get to the next instruction, skip 2 steps.

Load the Program to Memory



Fetch

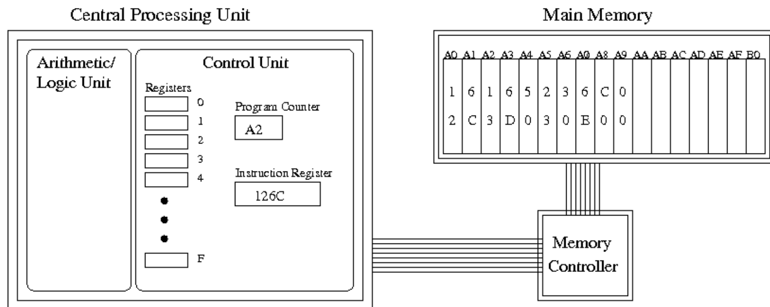
Step 1, Load IR



Registers are 16 bits, cells are 8 bits. Load IR with 2 memory cells.

Fetch

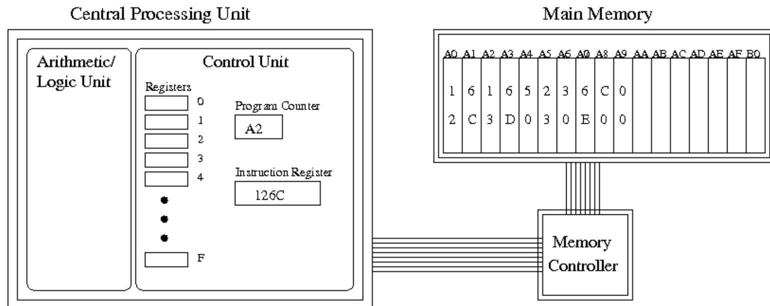
Step 2, Increment PC



Increment PC by 2 cells.

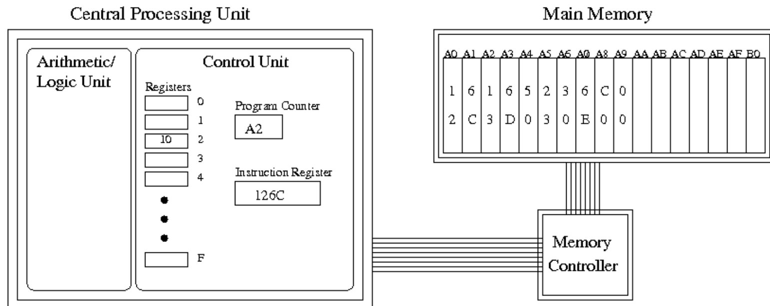
Decode

Control Unit Analyzes Instruction in IR for Command



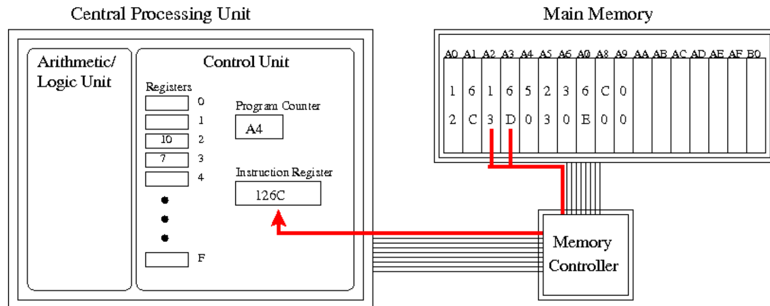
Execution

Machine LOADs Register 2 with Contents of Memory Cell at Address 6C



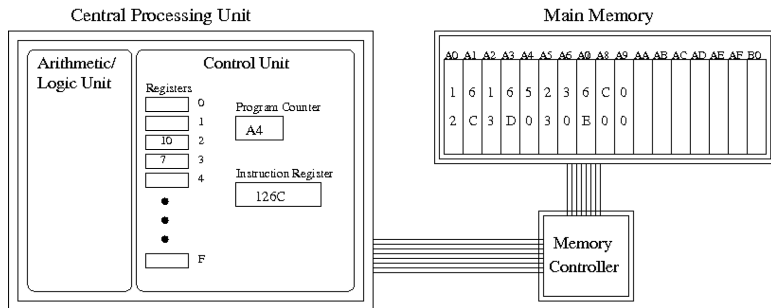
Repeat Cycle: Fetch

Step 1, Load IR



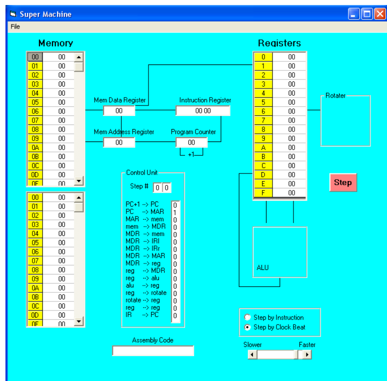
Repeat Cycle: Fetch

Step 1, Increment PC



Etc., etc., etc.

The Super Machine

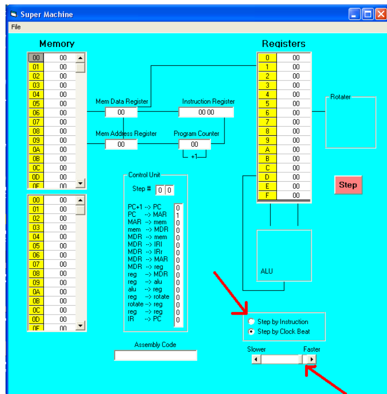


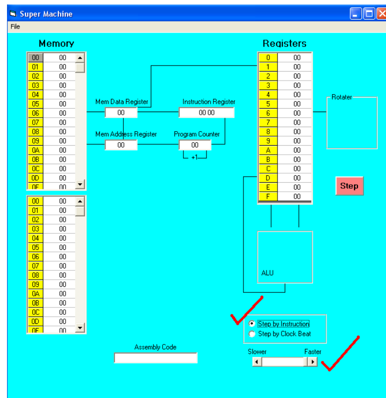
Comments about the Super Machine:

- Recognizes the machine language from Appendix C
- Can Save and Load programs (Useful for the homework).
- "Assembly Code" is **not equivalent** to RTN!

For our purposes, we would like the Super Machine execute a full instruction at a time.

The Super Machine





The Super Machine
HINTS

Some hints about programming for the Super Machine:

- START EARLY!!! Leaving a programming assignment until the last day is the surest path to getting no credit.
- Get an idea of the program. For example, 3×4 is the same as $3 + 3 + 3 + 3$. Seems easy? You need 4 registers to perform this...
- Work problems on paper first! The super machine is not intuitive, just as machine language is not intuitive. You will solve problems much faster by doing them on paper first.
- You have 16 registers, which gives you 16 variables you can play with. Use them.

The Super Machine
Examples

In the book, all of the problems will be good examples to watch.

- For example, problem 17, p. 112 (Course Documents, Chapter Notes, Chapter 2, problem17.smo)
- 3×4: Same place.

“Running” a Program

Table view

Since we are the I/O devices, we need a method to represent the operations that the virtual computer is running. We do this by keeping track of the elements below in a table:

1. The Program Counter,
2. The Instruction Register, and
3. The Action taken (i.e., RTN)

Each of the above elements is one column in the table.

"Running" a Program

Table view

For example, in the previous program:

PC	IR	RTN
A0		Program Begins.
A2	126C	$R[2] \leftarrow M[6C]$
A4	136D	$R[3] \leftarrow M[6D]$
A6	5023	$R[0] \leftarrow R[2] +_2 R[3]$
A8	306B	$M[6B] \leftarrow R[0]$
AA	C000	HALT

"Running" a Program

Odds and Ends

In the previous program, note:

- Each row in the table is written **after** the Fetch-Decode-Execute cycle has occurred.
- The first entry shows the beginning point of the program (i.e., where the PC begins). At this point, the IR is empty.
- The program ends with the HALT statement.

Computer Architecture

Machine Languages

Program Execution

Arithmetic/Logic Instructions

Device Communication
Data/Communication

Machine Instruction Types

Arithmetic/Logic Instructions

$$\text{Instruction Types} = \begin{cases} \text{Data Transfer} = \begin{cases} \text{Memory} \\ \text{I/O Devices} \end{cases} \\ \textbf{Arithmetic/Logic} = \begin{cases} \textbf{Add, etc.} \\ \textbf{AND, OR, etc.} \end{cases} \\ \text{Control} = \begin{cases} \text{JUMP} \\ \text{BRANCH} \end{cases} \end{cases}$$

Machine Instruction Types

Arithmetic/Logic Instructions

Arith/Log Ops =
- Logic =
  - AND
  - OR
  - XOR
- **Rotation/Shift** =
  - **Rotate bits left and right**
  - **Shift bits left and right**
- Arithmetic =
  - Add two′s complement
  - Add floating point format

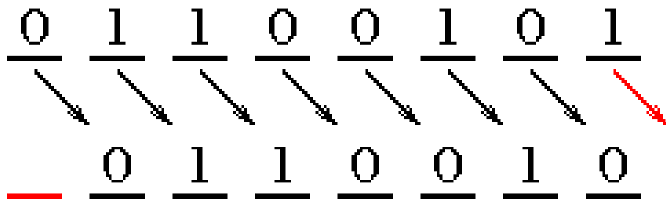
All instructions are applied on registers only!!

ALU

The basic idea of **rotation** and **shift** operations is to move the bit pattern in a given register either to the left or the right.

- A Rotations operation is also known as a circular shift.

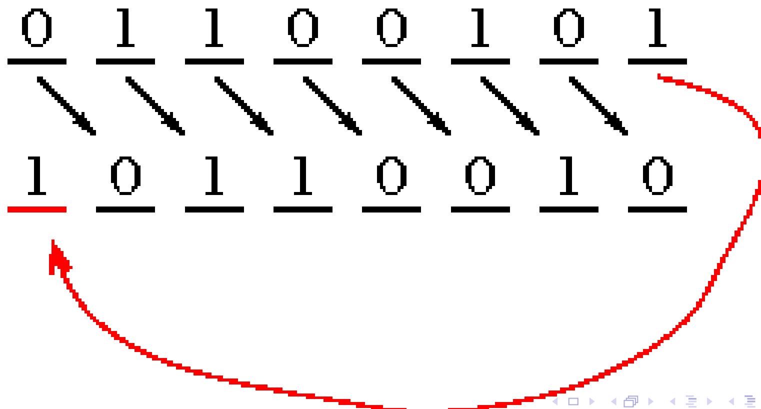
Idea:



ALU

Rotation

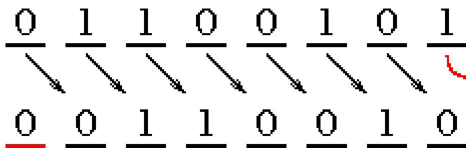
Rotation is when we feedback the floating bit to the hole in the shifted sequence.

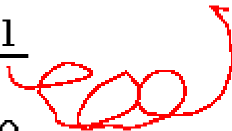


ALU
Shift

Another shift is the **logical shift**, in which we always fill the hole with a 0.

Arithmetic Shift: In cases where we care about the sign bit, we fill the hole with whatever the previous bit was.





Logic
Application

Recall that one of the steps in converting a floating-point format number into decimal notations is to extract the exponent of the 8-bit floating-point number:

01101111

How can *computers* extract these three bits?

Masking

Computers can isolate bits using a process called **masking**:

- Masking is a process that is used to manipulate data, involving two operands
  - Data
  - Mask – A bit pattern of 1s and 0s
- We have three logic operations that we can apply for masking: AND, OR, XOR

Masking

Extracting the Exponent

We want to extract only the 2nd, 3rd, and 4th bit.

- If a bit is a 1, we want to get a 1, if a bit is 0, we want to get a zero.

For any of the other bits:

- Regardless of the bit value, we want it to be a zero.

What logical operation does this sound like?

Using this logical operation, what should the mask be?

Masking

Extracting the Exponent

	01101111	⟵	Data
AND	01110000	⟵	Mask
	01100000	⟵	Result

Masking

Result of Logical Operations

Using a mask with each of the logical operations produces a unique result:

AND

- Duplicates part of a string while placing 0s in the non-duplicated part.

OR

- Duplicates part of a string while placing 1s in the nonduplicated part.

XOR

- Used to **complement** a bit string.

Examples

- Page 101, number 6:
  - What logical operation together with what mask can you use to change ASCII codes of lowercase letters to uppercase? What about uppercase to lowercase?
- Page 101, number 2,3:
  - Suppose you want to isolate the middle four bits of a byte by placing 0s in the other four bits without disturbing the middle four bits. What mask must you use together with what operation?
  - Suppose you want to complement the four middle bits of a byte while leaving the other four bits undisturbed. What mask must you use together with what operation?
- Other:
  - How can we multiply an 8-bit number by 2? By 16?
  - How can we divide an 8-bit number by 2? By 8?

Computer Architecture

Machine Languages

Program Execution

Arithmetic/Logic Instructions

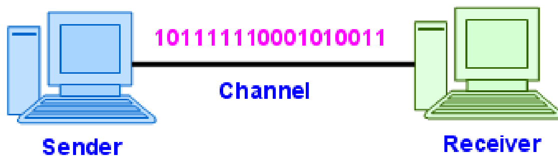
Device Communication
Data/Communication

Definition

Device Communication: The process of sending and receiving data between two entities.

This definition does not specify the transmission medium.

- Wireless (air),
- Wired (telephone wire),
- Optical (optical wires)



Communication Rates

A crucial property of any communication system is the rate by which the channel can transfer bits from the sender to the receiver.

- The rate at which bits are transferred to the receiver is measured in **bits per second (bps)**.
- Common units:
  - Kbps (1000 bps),
  - Mbps (1 million bps),
  - Gbps (1 billion bps)

The maximum rate that a communication channel has is often equated to the communication channels **bandwidth**.

Communication Channel Types

Parallel and Serial Communication

Communication in computers occur through **communication channels**. There are two methods of communication:

Parallel – Several bits are transferred at the same time, each on a separate line.

- Capable of transferring data rapidly (on the order of Mbps and higher).
- Requires a complex communication channel.
- Examples: The computers bus.
- In general, much of the communication between the computer and its peripheral devices is parallel.

Communication Channel Types

Parallel and Serial Communication

Serial – Only one bit at a time is transferred.

- Slower transmission.
- Possible to do using a simple data channel. Bits transferred one after another.
- Communication between computers is usually serial (telephone wires are serial communication systems).

Communication between Computers
Digital Computer vs. Analog Phone Line

Computers are connected together using the pre-existing technology of analog phone lines.

- The challenge, at the time, was making the digital computer able to transfer data over an analog system.
- The **modem** is the interface between your computer and phone line that takes the digital signal, and translates it to an analog signal.

Questions over Chapter 2?