# Lecture 1

**Part 1**

**An Introduction to Computer Architecture**

---

## Reading Assignment

- Microcontrollers and Microcomputers: Chapter 1, Appendix A, Chapter 2
- Software and Hardware Engineering: Chapter 1

*Or*

- Software and Hardware Engineering (new version): Chapter 1, Appendix A, Chapter 2 (2.1 & 2.2)

---

## Basic Computer System

<table>
<thead>
<tr>
<th>CPU</th>
<th>ALU</th>
<th>Control logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Bus</td>
<td>Address Bus</td>
<td>Control Bus</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>Program Data</th>
<th>Serial Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interf</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

## von Neumann Architecture

- A computer design model that uses a processing unit and a single separate storage structure to hold both instructions and data

- Named after mathematician and early computer scientist John von Neumann

---

## Early Computers

- **ENIAC** (Penn, 1946)
- **AVIDAC** (Argonne, 1953)
- **SAGE** (Air Force, 1958)

---

## Microprocessor

- CPU on a single IC
- 1971: Intel 4004 - The first microprocessor on the market
If Automobile Industry follows Moore’s Law...

- “If the automobile industry advanced as rapidly as the semiconductor industry, a Rolls Royce would get 1/2 million miles per gallon and it would be cheaper to throw it away than to park it.”

Gordon Moore, Intel Corporation

Dr. Tao Li

### Moore’s Law

- Graph showing the exponential increase in the number of transistors over time, illustrating Moore’s Law.

Dr. Tao Li

### µP: Performance and Power

- Chart comparing performance and power trends.

Dr. Tao Li

### µP Trend: Multi-core

- Comparison of current (Today) and future (Tomorrow) multi-core processor technologies.

Dr. Tao Li

### Microcontroller & Embedded System

- Microprocessor with extra on-chip features such as memory, I/O controller, A/D, etc. for embedded applications.

- What is an embedded system:
  - “A computer that doesn’t look like a computer”
  - Interacts directly with world (not human)
  - Primitive or no user interface
  - Part of other products

Dr. Tao Li

### Dominant Species

- Chart showing the dominance of embedded systems over desktop computers.

Dr. Tao Li
EEL 4744C: Microprocessor Applications

Lecture 1

Part 2 (self-study part)

Binary Codes (Textbook Appendix A)

Dr. Tao Li

Summary Points

• Binary Codes for Numerical Information
  – Unsigned-binary code: represents only positive information
  – Signed/magnitude binary code
  – One’s-complement code
  – Two’s-complement code and arithmetic
  – Binary coded decimal

• Binary Codes for Non-Numerical Information
  – ASCII 7-bit codes for alphanumeric characters (Table A-5)

Designing a Simple Computer

• Define the set of operations:
  – Assume 4 operations, 2 bits are needed to provide a unique op code for each operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>00</td>
</tr>
<tr>
<td>SUB</td>
<td>01</td>
</tr>
<tr>
<td>IN</td>
<td>10</td>
</tr>
<tr>
<td>OUT</td>
<td>11</td>
</tr>
</tbody>
</table>

Hardware for ADD, SUB

• ADD, SUB require 2 (8-bit) operands:
  – Operands are held in registers (arrays of flip-flops)

• What if we also want AND, OR?

• Arithmetic and logic unit (ALU):
  – Registers are used to hold operands
  – A Register is used as both source and destination register
  – B Register

Processing flow:
  – A Register and B Register hold the operands
  – The operands flow from A and B into ALU
  – ALU performs the ADD or SUB
  – The answer flows from ALU back into A
  – A Register is also called an Accumulator, because it accumulates answers
  – B Register is a general-purpose register

More details later:
  – The Registers need a clock signal
  – ALU must handle the carry signal produced by adder
Hardware for I/O

- **Input:**
  - 8 switches, or more choices, i.e. an A/D converter?

- **Output:**
  - 8 LED’s, or more choices, i.e. an D/A converter?

- **Design options for input:**
  - Use 2 input operations, IN1 (switches) and IN2 (A/D)
  - Include the operand with the operation => instruction

- **Instruction = operation + operand**
  - What is to be done, and what is to be operated on

Hardware for I/O (2)

- Assume 4 input and 4 output devices:
  - **Operand code** is needed to specify one of 4 I/O devices
  - Append this operand code to the op code
  - We now have 8-bit instruction codes
  - Input device number is encoded with “ii”
  - Output device number is encoded with “oo”

- **The MOV operations:**
  - Input into accumulator A, output from accumulator A
  - Need to move number from A to B
  - Define MOV to transfer data from A to B
  - MOV copies data from source (A) to destination (B)
  - Need to add 1 more bit to the Op Code

Hardware for I/O (3)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand</th>
<th>Instruction code = Op Code + Operand Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>(ions)</td>
<td>001----</td>
</tr>
<tr>
<td>SUB</td>
<td>(ions)</td>
<td>011----</td>
</tr>
<tr>
<td>IN</td>
<td>Device #</td>
<td>101--</td>
</tr>
<tr>
<td>OUT</td>
<td>Device #</td>
<td>111--</td>
</tr>
<tr>
<td>MOV</td>
<td>(ion)</td>
<td>010----</td>
</tr>
</tbody>
</table>

The current instruction set

Operands of I/O instructions

No need to specify operands

Assembly Code Sequence

Hardware for I/O (4)

Assembly Code Sequence

Computer Memory

- Read a number from the switches and multiply it by 3, then display the result on the LED’s:
  - Assume switches are Device 1 and LED’s are Device 2

- **What is the assembly code?**
  - IN 1
  - MOV
  - ADD
  - OUT 2

- **Contents of the memory before execution?**
  - Assembling the program => encoding the assembly code into 1’s and 0’s
Example Program (2)

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>1010 0001</td>
</tr>
<tr>
<td>1:</td>
<td>0100 0000</td>
</tr>
<tr>
<td>2:</td>
<td>0010 0000</td>
</tr>
<tr>
<td>3:</td>
<td>0010 0000</td>
</tr>
<tr>
<td>4:</td>
<td>1110 0010</td>
</tr>
</tbody>
</table>

Instruction Register & Decoder

- When an instruction is fetched from the computer memory, it is transferred to an instruction register and decoder.
- The instruction register holds the instruction, the decoder then decodes the instruction:
  - The decoder is a combinational logic circuit.
  - It takes the op code bits from the instruction and generates logic signals that are asserted for each operation.
  - These logic signals go to appropriate places in the computer.
- Basically, the instruction decoder decodes the operation code and asserts proper control signals.

Instruction Register & Decoder (2)

Program instructions transfer from memory into an instruction register and decoder. The decoder asserts a control signal appropriate for each defined operation of the computer.

Instruction Decoder Circuit: An Example

Instruction Register & Decoder (2)

Control Signals from Decoder

When it is asserted by the decoder, IN activates the clock to latch the data from Device #1 into A Register. When OUT is asserted, the data in A Register is transferred to the LED’s.

PC & Memory Address Register

To run a program, we need a Program Counter (PC) and a Memory Address Register to “step through” the program.
Instruction Execution Cycle

- The sequence of steps the processor goes through to do the complete instruction and to get ready for the next, until the program is finished or stopped.
- Specific steps (simplified):
  - IF: instruction fetched
  - ID: instruction decoded, control signals asserted
  - RT/EX: register transfer and/or execution
  - PC: increment program counter
- These steps are repeated for each instruction execution cycle until the program is done, or until something is done to make the program stop.

Sequential State Machine

- The instruction execution cycle is partitioned into 5 different states, each state is an element of time long enough for an event to occur:
  - State 1: fetch instruction from memory
  - State 2: decode the instruction
  - State 3: register transfer op, e.g. IN, OUT, MOV
  - State 4: ALU op, e.g. ADD, SUB
  - State 5: PC=PC+1, MAR<=PC
- Note that after State 4, we enter State 3 to transfer the ALU result (ADD/SUB) into the A Register.
- Time taken by each state should be equal!!

Program Execution Time

- Sequential state machine is operated by a clock.
- This allows different instructions to take different amounts of time to finish.
- Execution time = Total # states x time per state.
- If basic clock frequency is 1 MHz, then time per state is 1 µs = 1 sec/1000000.
- If total # states = 21, then total program execution time will be 21 µs.

Sequence Controller

- Combination of the sequential state machine and the instruction decoder:
  - Generates control signals at the correct time for operation.
  - Example (Slide 29): signal is generated in State 3 for IN.
  - Allows different instructions to be executed in different amounts of time; controls computer’s sequential operation.

Sequence Controller (2)

- Instruction Type:
  - Assert signals at corresponding clock cycle.
I/O Synchronization & Wait State

- How do we make the computer “wait for” human input, e.g. entering data from outside?
- For example, your program is designed to input 2 numbers from the 8-bit switches and add them up
- How do you make the computer wait for you when you input those 2 numbers?
- We need a “Wait” state, where the processor will “spin its wheels” until a “ready” signal is asserted, allowing synchronization with slow input devices

For IN, the SSM enters the Wait state and stays there until an external control signal, i.e. READY, is asserted by the user.

I/O Synchronization & Wait State (2)

- MOV dd, ss (dd) ←(ss)
- Moves data from (ss) source register to (dd) destination register
- Register codes: A=00, B=01, C=10, D=11
- Example: MOV B, A will be assembled as the following binary code: 010-0100

I/O Synchronization & Wait State (3)

- How about the OUT instruction? Should it have a Wait state also?
- The READY signal for OUT can be asserted when the output device has received the data

Memory Reference Instructions

- MVI dd, 65 (dd) ← 65 (next memory location)
- 2-byte instruction that moves data from memory location right after the op code, i.e. the data is 65\textsubscript{10} in this case, to the destination register (dd)
- Register codes: A=00, B=01, C=10, D=11
- Example: MVI A, 65 will be assembled as below:
  First byte: 110---00 Op + Operand Code
  Second byte: 01000001 Data(= 65\textsubscript{10})

A Program Example

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
<th>Operation</th>
<th>Operand</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>1010 0001</td>
<td>IN</td>
<td>1</td>
<td>(A) ← (Register)</td>
</tr>
<tr>
<td>1:</td>
<td>0100 0100</td>
<td>MOV</td>
<td>B, A</td>
<td>(B) ← (A)</td>
</tr>
<tr>
<td>2:</td>
<td>1010 0001</td>
<td>IN</td>
<td>1</td>
<td>Get the 2nd number</td>
</tr>
<tr>
<td>3:</td>
<td>0010 0010</td>
<td>ADD</td>
<td>A, B</td>
<td>(A) ← (A + B)</td>
</tr>
<tr>
<td>4:</td>
<td>1100 0100</td>
<td>MVI</td>
<td>C, 65</td>
<td>(C) ← (location 5)</td>
</tr>
<tr>
<td>5:</td>
<td>0100 0001</td>
<td></td>
<td></td>
<td>The data is 65\textsubscript{10}</td>
</tr>
<tr>
<td>6:</td>
<td>0100 0100</td>
<td>ADD</td>
<td>A, C</td>
<td>(A) ← (A + C)</td>
</tr>
<tr>
<td>7:</td>
<td>1110 0100</td>
<td>OUT</td>
<td>2</td>
<td>(BEEP) = (A)</td>
</tr>
</tbody>
</table>

What does this program do?
Control Instructions

- **HALT**
  - Stops the clock pulses going into the sequence controller

- **JMP or BRA**
  - Transfers the PC from one place of the program to another
  - Operand is the memory location from which the computer must fetch its next instruction
  - This is a 3-byte instruction

- Example: “unconditional branch”: BRA 15
  - First byte: Op Code
  - Second byte: (branch address)
  - Third byte: (branch address)

Control Instructions (2)

- **Conditional branch**
  - Condition code (RC) register contains bits which are set or reset when an ALU operation is performed
  - A RC register can hold flip-flops for carry, zero, negative, two’s complement overflow, odd/even parity
  - The RC register bits are connected to the sequence controller
  - We can then design branch-if-carry, branch-if-overflow, etc.

Putting them Together