Chapter 4

The Von Neumann Model

- Basic components
- Instruction processing
- Memory
Building a computer

- To get a task done by a computer, we need two things:
  - a computer program that specifies what must be done to complete the task
  - hardware that is capable of completing those tasks as specified

- Low-level design of computer components is “digital design”
  - CEG 360 will cover much of this in detail.
  - Chapters 3 and 4 of the text will give you enough background to move forward to the next level of abstraction.

- The basic computer model was proposed by John von Neumann in 1946
The von Neumann Model - Illustration

- Memory: holds both data and instructions
- Processing Unit: carries out the instructions
- Control Unit: sequences and interprets instructions
- Input: external information into the memory
- Output: produces results for the user
Central Processing Unit (CPU)
The von Neumann Model - CPU

- Processing Unit - Does the actual work!
  - Can consist of many units, each specializing in one complex function.
  - At a minimum, has Arithmetic & Logic Unit (ALU) and temporary storage (registers).
  - The number of bits a basic Processing Unit operation can handle is called the WORD SIZE of the machine.
- ALU
  - Performs basic operations: add, subtract, and, not, etc.
  - Generally operates on whole words of data.
  - Some can also operate on subsets of words (eg. single bits or bytes).
- Registers:
  - Fast “on-board” storage for a small number of words.
  - Invaluable for intermediate data storage while processing.
  - Close to the ALU (much faster access than RAM).
  - General Purpose Registers (GPRs) are available to the programmer.
A Register file: Fast, easy memory
CPU Overview
Notations

- **Sets of Bits**
  - \( A[3:0] \) denotes a set of 4 bits: \( A_3, A_2, A_1, A_0 \)
  - The content of an \( n \)-bit register \( R_0 \) is referred to as \( R_0[n-1:0] \)
    - \( R_0_{n-1} \) is the most significant bit (MSB), or leftmost bit
    - \( R_0_0 \) is the least significant bit (LSB), or rightmost bit
    - Given \( R_0[31:0] \), \( R_0[7:4] \) refers to the four bits from \( R_4 \) to \( R_7 \)

- **Bit Assignment**
  - \( R_3[5:0] ← R_2[13:8] \)
    - Means that bits 5 to 0 of register \( R_3 \) get assigned the values of bits 13 to 8 of register \( R_2 \).

- **Contents**
  - \([R1]\) means “contents of \( R_1 \)”
  - \( \text{Mem}[\text{loc}] \) means “content of memory location \text{loc}”
Memory

- Memory
  - MAR
  - MDR
- Processing Unit
  - ALU
  - TEMP
- Control Unit
  - PC
  - IR
- Input
  - (keyboard)
- Output
  - (monitor)
RAM Memory: A difference in scale

- Memory is a large number of addressable fixed-sized locations
  - Each location has an *address* and *contents*
    - Address: bit pattern that uniquely identifies a memory location
    - Contents/Data: bit pattern stored at a given address.
    - Analogy: p.o. boxes have fixed numbers, but changing contents.
  - A large number of locations
  - Each uniquely identifiable (its address)
  - Each having the ability to store a value of a certain size (its addressability)

- Address Space
  - n bits allow the addressing of $2^n$ memory locations.
  - Example: 24 bits can address $2^{24} = 16,777,216$ locations (i.e. 16M locations)... but what SIZE is each location?
Memory Addressability

• **Word**: the basic unit of data used by a device (generally the processing unit)

• Ideally, one word is fetched with a single memory access
  – The **addressability** of Memory is the number of bits of memory referenced by a given address (generally in bytes)
  – Processor Word size for most contemporary computers is 32 bits
  – The memory may be byte OR word **addressable**.

• Normally, a whole word is written and read at a time:
  – If the computer is **word** addressable, this is simply a single address location.
  – If the computer is **byte** addressable, and uses a multi-byte word, then the word address is conventionally either that of its most significant byte (big endian machines) or of its least significant byte (little endian machines).
 Byte Ordering Example

- How should bytes within multi-byte word be ordered in memory?
- **Big Endian**
  - Least significant byte has highest address
- **Little Endian**
  - Least significant byte has lowest address
- **Example**
  - Variable `x` has 4-byte representation `0x01234567`
  - Address given by `&x` is `0x100`

### Big Endian

<table>
<thead>
<tr>
<th></th>
<th>0x0100</th>
<th>0x0101</th>
<th>0x0102</th>
<th>0x0103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

### Little Endian

<table>
<thead>
<tr>
<th></th>
<th>0x0100</th>
<th>0x0101</th>
<th>0x0102</th>
<th>0x0103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
What is stored in memory?

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1000</td>
<td>x0030</td>
</tr>
<tr>
<td>x1001</td>
<td>x0300</td>
</tr>
<tr>
<td>x1002</td>
<td>xC2FE</td>
</tr>
<tr>
<td>x1003</td>
<td>x1F00</td>
</tr>
<tr>
<td>x1004</td>
<td>x1000</td>
</tr>
<tr>
<td>x1005</td>
<td>x3F3F</td>
</tr>
</tbody>
</table>

Is this value...
- A positive integer?
- A negative integer?
- A float?
- An address?
- Data?
- Part of a program (a machine instruction)?

How can we tell?
The von Neumann Model - Memory

- We will access memory via a standard interface
  - The Memory Address Register sets up the decoder circuitry in the memory
  - The Memory Data Register hold contents to/from memory
  - The Write enable signal indicates which direction information should flow.

- Memory Read
  - Read: the contents of the specified address will be written to the Memory Data Register.
  - If: Write enable is not applied in any given clock tick
  - Then: \( \text{MDR} \leftarrow M[(\text{MAR})] \)

- Memory Write
  - Write: the value to be stored is first written to the Memory Data Register, then the Write Enable is asserted, and the contents of the MDR are written to the specified address.
  - If: a Write enable is applied is any given clock tick
  - Then: \( M[(\text{MAR})] \leftarrow \text{MDR} \)
What sequence of control signals would need to be executed to add the value in R3 to the value in memory at address x30FF and place the result in R5?

*You may assume that register R6 already contains the value x30FF*
Control Unit

Input
(keyboard)

Memory
MAR
MDR

Processing Unit
ALU
TEMP

Control Unit
PC
IR

Output
(monitor)
The von Neumann Model – Control Unit and I/O

• Control Unit

  – The control unit coordinates all actions needed to execute the instruction

    • It fetches & decodes each instruction, and sets up the appropriate inputs for the Memory, Processing, and I/O units as required.

    • It communicates with memory via the Program Counter (PC) and Instruction Register (IR)

  – PC (aka Instruction Pointer)

    • Holds the address of the next instruction to be fetched.

  – IR

    • Holds the instruction currently being executed.
    • This can be a single word, or multiple words.

  – PSC (processor status register)

    • Holds information about the status of the running program.

• Input & Output

  – Generically known as peripherals - they are external to the CPU.

  – This means we will have to develop mechanisms for autonomous devices to communicate with each other - more on this later.
The LC-3: A von Neumann Machine
Instruction Cycle - overview

• The Control Unit orchestrates the complete execution of each instruction:
  – At its heart is a Finite State Machine that sets up the state of the logic circuits according to each instruction.
  – This process is governed by the system clock - the FSM goes through one transition (“machine cycle”) for each tick of the clock.

• Six phases of the complete Instruction Cycle (and repeat...)
  – **Fetch**: load IR with instruction from memory
  – **Decode**: determine action to take (set up inputs for ALU, RAM, etc.
  – **Evaluate address**: compute memory address of operands, if any
  – **Fetch operands**: read operands from memory or registers
  – **Execute**: carry out instruction
  – **Store results**: write result to destination (register or memory)
One clock cycle is often (though not always) the time it takes for the machine to perform a simple instruction (such as a register-to-register add).

Usually one clock cycle corresponds to a complete instruction cycle.

The clock cycle is broken down into sub-cycles for each phase.

The control unit executes the appropriate signals for each phase:

```plaintext
if (instruction is add && sub-cycle is 4) {
    SR1 ← source register number
    etc...
}
```
Instruction Cycle - steps 1 & 2

- **Fetch**
  - The first step is to read an instruction from memory.
  - This actually takes several smaller steps, or “micro-instructions”:
    - MAR ← (PC) ; use the value in PC to access memory
    - PC ← (PC) + 1 ; increment the value of PC
    - MDR ← Mem[MAR] ; read memory location to MDR
    - IR ← (MDR) ; copy (MDR) to IRDecode
  - Steps 1, 2 & 4 take a single machine cycle each, but step 3 (memory access) can take many machine cycles.

- **Decode**
  - The opcode is input to a decoder, which sets up the ensuing sequence of events according the instruction.
Instruction Cycle - steps 3 & 4

- **Evaluate Address**
  - Computes the address of the operand (if any), or of the memory location to be accessed: e.g. the location from which to obtain a value in a LOAD instruction.
    - This is known as the **Effective Address** (EA).
    - Register to Register instructions don’t have an evaluate address phase.

- **Fetch Operands**
  - Obtains the source operand(s), if required for execution.
  - Operands can come from Registers or RAM, or be embedded in the instruction itself.
    - The Effective Address (EA) determined in the previous step may be used to obtain an operand from memory.
Instruction Cycle - steps 5 & 6

● Execute
  – Now that everything is in place, the instruction is executed.
    ● e.g. if the opcode was ADD, the two source operands are added by the ALU.
    ● If the opcode was a control instruction, a value is written to the PC
    ● Data Movement instructions don’t have an execute phase

● Store Result
  – If there is a result from the operation it is written to memory (using the EA),
    or to a register.

● Then REPEAT
  – The control unit just keeps repeating this whole process: Recall that the PC
    was incremented in the first step (FETCH), so the instruction retrieved will be
    the next in the program as stored in memory - unless the instruction just
    executed changed the contents of the PC.
Problems

- 4.5, 4.7, 4.8, 4.10
Types of Instruction

- **Operate Instructions**
  - process data (addition, logical operations, etc.)

- **Data Movement Instructions**
  - move data between memory locations and registers.

- **Control Instructions**
  - change the sequence of execution of instructions in the stored program.
    - The default is sequential execution: the PC is incremented by 1 at the start of every Fetch, in preparation for the next one.
    - Control instructions set the PC to a new value during the Execute phase, so the next instruction comes from a different place in the program.
    - This allows us to build control structures such as loops and branches.
Machine Words

• Machine Has “Word Size”
  – Nominal size of integer-valued data
    • Including addresses
  – Most current machines are 32 bits (4 bytes)
    • Limits addresses to 4GB
    • Becoming too small for memory-intensive applications
  – High-end systems are 64 bits (8 bytes)
    • Potentially address \( \approx 1.8 \times 10^{19} \) bytes
  – Machines support multiple data formats
    • Fractions or multiples of word size
    • Always integral number of bytes

Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
</tr>
</tbody>
</table>

Or any other pointer