4.2 Largest and Smallest Floating Point Numbers

Because any fixed length encoding of length k is restricted to 2^k codewords, therefore, in any floating point encoding using code wards of length k, only 2^k values can be represented exactly. This constrains the largest and smallest values that can be represented. For example, given m = 8, and n= 3:

- The largest positive finite value is determined as follows:
  - The largest possible exponent of a normalized number has codeword 1110. Since the bias is 7, the value represented by this codeword is $1110 - 0111 = 14 - 7 = 7$.
  - The largest possible fraction has codeword 111. It represents the significant 1.111.
  - Concatenating codewords for sign, exponent, and fraction we obtain 01110111 and it represents 1.111×2^7.
  - $1.111 \times 2^7 = 1111 \times 2^4 = 15 \times 16 = 240$
- The smallest positive normalized value is obtained in as similar manner:
  - The smallest possible normalized exponent is 0001. It represents $0001 - 0111 = 1 - 7 = -6$
  - The smallest possible fraction is 000. It represents 1.000.
  - Concatenating the codewords for sign, exponent, and fraction we obtain 00001000 and it represents 1.000×2^{-6}.
  - $1 \times 2^{-6} = \frac{1}{64}$

  However an even smaller value can be represented if denormalized floating point is used:
  - The exponent is 0000. By convention, it represents $1 - bias = 1 - 7 = -6$
  - The smallest possible fraction is 001. By convention, it represents 0.001.
  - Concatenating the codewords, we obtain 00000001 and it represents 0.001×2^{-6}.
  - $0.001 \times 2^{-6} = 2^{-9} = \frac{1}{512}$

5 COMPUTER DESIGN

A "digital system" is a circuit that processes discrete signals representing logic values. These values are commonly represented symbolically by the values '0' and '1'. A sequence of such symbols is called a binary sequence. IN digital systems, binary sequences are stored in hardware devices called "registers" and transferred between devices on groups of wires called "buses." A single wire is called a "signal line." The size of a register or bus is the given by the length of binary sequence that is can store or transmit.

A "computer" is a digital system that executes instructions stored in an external memory as binary sequences. The design of a computer therefore begins with the formulation of a set of instructions that must then be encoded into binary.

There are many decisions to be made that influence how a CPU is eventually implemented to execute the instructions:

- What instructions should be included;
- How should the instructions be represented symbolically and as machine instructions;
- What hardware model should be adopted: von Neumann vs non-von Neuman, RISC vs CISC, etc;
- Performance requirements;
- Cost considerations.

The collection of instructions, formats, memory model, and related design decisions defines an *instruction set architecture*. In the following sections these issues are described in more detail.

In general, the choice of instructions and the design of instruction formats is guided by the goal of efficient execution. To observe the consequences of different choices and designs, it is important to choose a formal computer model for the eventual architecture of the computer system. The most popular model is the "von Neumann" or "Princeton" model. The principal characteristic of this model is that both instructions and data share the same external memory.

In designing an instruction set architecture for a von Neuman model, the organization and role of external memory includes the following:

1. Both instructions and Data are stored in the memory:
2. The memory can be viewed as a 1-dimensional array of binary sequences.
3. The memory is defined by two numbers: 2^m and n:
    - 2^m specifies the number of locations in a memory;
    - Each location is specifed by a number called an "address";
    - Addresses are numbered from 0 to $2^m - 1$.
    - Each array location can store one binary sequence of length n;
    - The contents of an array location is called a "word";
4. Each Instruction is defined by a binary sequence.
5. Each data value is defined by a binary sequence.
6. Each location is identified by a binary sequence (its address).
7. Instructions and data may be stored in one or more locations of memory

5.1 Machine Instructions

- A "machine instruction" is a sequence of elements that identify the task to perform and the operands to use.
- Each element is a pre-defined, fixed-length binary sequence.

- The binary sequence that defines a machine instruction is the concatenation of the binary sequences that define the elements.

For the convenience of human programmers, each machine instruction has a symbolic representation called its "Assembly Language Instruction." Each element of an assembly language instruction is a symbol associated with a binary sequence of the corresponding machine instruction.

Regardless of what instructions define an instruction set, all instructions must provide the same basic elements:

- A way of identifying what instruction or family of instructions is to be executed. This element, called the "opcode", is a binary sequence that determines how the rest of the instruction is to be interpreted.
- Operands that specify what, where, or how values that are required to execute the instruction specified by the opcode are to be obtained. The operands may specify locations in internal memory (registers), locations in external memory, and literal values.

  Operands may be explicit or implicit. Explicit operands are those that must be provided by the programmer as part of the instruction. Implicit operands are those that are implied by the opcode. That is, the instruction is implemented in a fashion such that the implicit operands are predetermined and therefore defined at the time of execution.

In designing an instruction set both the syntax and semantics of each instruction must be defined:

The **syntax of an instruction** specifies the structure of an instruction; that is, what each instruction looks like:

Syntax of an instruction in assembly language : Defines how the programmer is to represent the instruction.

- **Mnemonic**: A symbolic label for the opcode. It identifies what task is to be performed.
- **Explicit Operands**: identify which operands the programmer provides.

Syntax of a Machine Instruction : Defines the "physical format" for the binary sequence that represents the machine language instruction:

- **Opcode field**: Identifies which bits define the task to be performed.
- **Operand field(s)**: Identifies which bits define which operands.
- **Unused field(s)**: Identifies which bits are not used.

The **Instruction Semantics** define the purpose of each element; that is, how that element is to be interpreted and what the sequence of elements are to do. Specifically:

Semantics of an assembly language instruction :

- Describes the intent of the mnemonic; that is, what operation function or task is to be performed.
- Describes the role played by the operands; that is, how each operand is to be interpreted.

Semantics of a machine instruction : Describes the correspondence between the binary sequences representing the elements of a machine instruction and their assembly language representations. It is usually given by a "physical instruction format". An example is provided on page 19.

5.2 The Instruction Execution Algorithm

The behavior of all von Neumann CPU's is described by an algorithm called the "Instruction Execution Algorithm", also known as the "Fetch/Execute Cycle". This algorithm is defined as follows:

1. Identify the location of the first instruction.
2. Retrieve a binary sequence to be interpreted as an instruction from memory.
3. Decode the binary sequence to determine the instruction and operands
4. Execute the instruction using the operands given. This may require retrieving some of the operands, also expressed as binary sequences, from memory.
5. identify the location of the next instruction.
6. Go to step 2.

5.3 Instruction Set Design

The design of an instruction set is a precursor to the design of a CPU. Instruction set design requires decisions to be made about what instructions will actually be included. Instruction sets may be large or small:

RISC Philosophy : A small number of general purpose instructions will define the instruction set:

- A small number of instructions means a small opcode and shortens the length of a (machine) instruction.
- A small number of instructions leads to simpler CPU architectures.
- Simpler CPU architectures are easier and more economical to fabricate.

CISC Philosophy : A large number of instructions allows the inclusion of special purpose instructions:

- Performance gains can be obtained by customizing the CPU to accommodate complex special purpose instructions,.
- Variations on instructions provide the programmer with more choice in the implementation of an algorithm.
- Less need for subprograms to be defined as such functions can be implemented as instructions.

5.4 Computer Memory

In addition to deciding on the set of instructions that the CPU will implement, it is also important to develop an abstract model of what memory will be provided for the CPU to access and how it will be organized.

Computer memory is organized into two systems:

External Memory : is the "main" memory where the program and its data are initially stored. External memory, as the name implies, resides outside the CPU. This memory is typically implemented with SRAM's and DRAM's. However, from a programmer's perspective, it can viewed as a 1-dimensional array of locations.

In assembly language programming, memory locations can be labelled with symbolic addresses. Programmers can then reference these addresses to store or retrieve values from external memory. This is analogous to the declaration of variables used in high-level language programming.

Internal Memory : is the smaller amount of memory that is internal to the CPU. This memory is usually implemented with individual registers and may also include a register file. A register file is an actual array of registers with multiple output ports.

The locations of internal memory within a CPU usually have fixed pre-defined labels to which the assembly language programmer can refer when a value is to be retrieved or stored in internal memory.

5.5 Instruction Types

The goal in designing an instruction set is to formulate a set of instructions that defines are "general purpose". That is, they are sufficient to implement any algorithm. It has been shown that a general purpose instruction set requires the following types of instructions:

Data Transfer (including I/O) : These instructions permit the programmer to move data between registers, memory locations, and input or output devices

Computational : These instructions make it possible to perform any arithmetic or logical function. To insure that this is the case, the set of arithmetic and logical instructions must be "functionally complete." This means that , using only the instructions given, it is possible preform any arithmetic calculation (i.e., addition, subtraction, multiplication, and division), and any logical calculation (i.e., conjunction, disjunction, complement)

Testing : It must be possible to compare two quantities and save the result. Instruction that perform this task fall into this category.

Branching ; Most interesting programs require that it be possible to make a choice as to the next instruction to execute. This includes the ability to perform iteration. Branching instructions make it possible to perform these tasks.

5.6 The von Neumann Bottleneck

One of the principal limitations in the performance of a von Neumann CPU is that external memory access is the slowest aspect of the instruction execution algorithm. Therefore a key instruction design goal is to minimize the number of external memory accesses; that is minimize the effect of the von Neumann "bottleneck".

Among the Factors affecting the bottleneck are:

1. Number of instructions: this affects the length of an instruction. More instructions means longer opcodes.
2. Number of explicit operands per instruction: Each explicit operand adds to the length of the machine instruction.
3. Number of operands that require memory access: Each such operand will result in an additional access to external memory during the execution phase of the fetch/execute cycle.

During one iteration of the instruction execution algorithm there are two occasions when external memory may be accessed:

1. **During the Fetch Phase**: Since every instruction must be retrieved from external memory, that memory must be accessed at least once during this phase, for each instruction.
2. **During the Execute Phase**: Any instruction whose operand references external memory will result in at least one external memory access when the instruction is executed. Note that not all instructions will need to make an external memory access during this phase.

6 INSTRUCTION SET DESIGN

Every assembly language instruction requires the design of a machine instruction format. The following principles for instruction design simplify the eventual CPU design:

- Use only a few different instruction formats,
- Fields in different formats with the same purpose should be in the same positions within each format,

The design of Instruction formats requires the following steps:

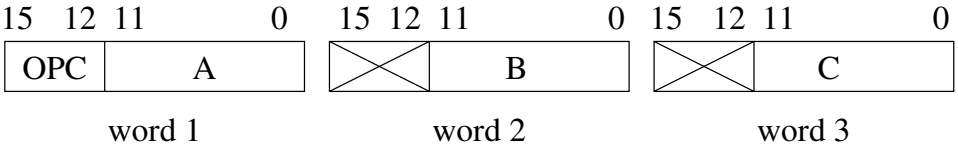
1. Assign opcodes to all instructions,
2. Determine the size of each operand to be encoded.
3. Group instructions into sets having similar operand requirements.
4. Determine the "logical format" for the instruction group.
5. Determine the "physical format" of each instruction group.

To illustrate, suppose that an assembly language instruction with syntax: "ADD A, B, C" has the following semantics:

$$M[A] \leftarrow M[B] + M[C]$$

Furthermore, suppose that this instruction is part of an set of 10 instructions and the CPU is interfaced to an external $2^{12} \times 16$ external memory. From this information, the design of the machine instruction is:

1. The opcode will be 4 bits. Assuming that 0001 is not assigned to another instruction, let 0001 be the opcode for ADD.
2. According to the semantics each explicit operand, A, B, and C are addresses. The number of locations of external memory is 2^{12}; therefore 12 bits will be required to specify the address of each of the three operands.
3. The "logical format" for the instruction is therefore 40 bits long (4 bits for the opcode plus 12 bits for each of the 3 operands)
4. Since the word size of external memory is 16 bits, three words of memory will be required to store all of the ADD instruction in memory. But three words of memory provides enough bits to store a binary sequence of length 48. So not all the bits of storage will be required. One possible way of distributing the 40 bit sequence among the 3 words is defined by the following proposed physical format:



6.1 Instruction Set Design Examples

The following examples illustrate how the the choice of instructions influences the style of programming as well as the performance when executed.

The goal is to write a sequence of assembler instructions to evaluate the following:

```
z = (p + q) * ( p - q)
```

EXAMPLE 1: A 3-operand machine design :

NOTE: An instruction set where the arithmetic instructions require the programmer to specify "n" explicit operands is referred to as an "n-operand machine."

Number of instructions: 10

External memory: $2^{12} \times 16$ bits

Internal (CPU) memory: 0

1. **Proposed Instruction Set (partial)**:

	SYNTAX		SEMANTICS
OP	INST	OPNDS	FUNCTION
1	ADD	A, B, C	M[A] ← M[B] + M[C]
2	SUB	A, B, C	M[A] ← M[B] − M[C]
3	MPY	A, B, C	M[A] & M[A+1] ← M[B]×M[C]

2. **Physical Formats for Each Instruction**:

Since the same physical parameters (instruction set size, external memory size) used for the example physical format provided previously are applicable to all instructions in this instruction set, the physical format obtained previously is applicable to the SUB and MPY instructions.

To illustrate the application of a physical format to the translation of an assembly language instruction into a machine language instruction, suppose we wish to encode the assembly language instruction "SUB xA00, x000, x001" into machine language (the notation xNNN denotes that NNN is a hexadecimal number):

1. The semantics of this instruction indicate that the following operation is to be performed:

$$M[A00] \leftarrow M[000] - M[001]$$

2. Assume the instruction is stored in memory beginning at location x01C. Applying the physical format to this instruction (unused bit positions are set to '0') the instruction would be stored in memory as follows:

ADDR	BINARY	HEX
01C	0010 1010 0000 0000	A00
01D	0000 0000 0000 0000	000
01E	0000 0000 0000 0001	001

3. **Review Design Consequences**: To assess the expressiveness of the instruction set as well as to examine the effects of the von Neumann bottleneck, it is useful to write and analyze some sample programs. For example, a program to evaluate the expression:

```
z = (p + q) * (p - q)
```

INST	OPERANDS	FUNCTION
ADD	TMP1, P, Q	M[TMP1] ← M[P] + M[Q]
SUB	TMP2, P , Q	M[TMP1] ← M[P] − M[Q]
MPY	Z, TMP1, TMP2	M[Z]&M[Z+1] ← M[TMP1] × M[TMP2]

4. **Quantify Performance**: The number of memory accesses can be used as a measure of performance because of the significant effect of memory accesses on execution time (i.e., the von Neumann Bottleneck). For each instruction, count the number of memory accesses required to retrieve the instruction and the number of memory accesses that occur when it is executed:

OPCODE	INST	OPERANDS	FETCH ACCESSES	EXECUTE ACCESSES
6	ADD	TMP1, P, Q	3	3
7	SUB	TMP2, P , Q	3	3
8	MPY	Z, TMP1, TMP2	3	4

Hence a total of 19 memory accesses will occur when the fetch/execute algorithm is applied to this program, stored in memory.

6.2 Improving Performance

The limiting factor in instruction set performance is the effect of the von Neumann bottleneck. Therefore, by reducing the number of memory accesses required during the execution of a program, the performance of a program can be expected to improve.