Computer System and Network Laboratory Module

Muhammad Husni Santriaji, S.Si., M.T., M.S., Ph.D.

August 27, 2024

Contents

Chapter 1

Introduction to Computer Architecture and Assembly Language

1.1 Objectives

1. The student will understand the concepts of registers, the Arithmetic Logic Unit (ALU), and the program counter.
2. The student will understand how the CPU operates with assembly language.
3. The student will be able to read and write simple assembly language programs.

1.2 Computer Architecture

A computer system primarily consists of four main components: the Central Processing Unit (CPU), memory, interconnections, and input/output (I/O) devices. Each of these components plays a crucial role in the functioning of a computer system.

1.2.1 Central Processing Unit (CPU)

The Central Processing Unit (CPU) is often referred to as the brain of the computer. It performs arithmetic and logic operations and controls the execution of instructions. The CPU consists of several key parts:

- **Registers:** Small, fast storage locations within the CPU used to hold data and instructions temporarily.
- **Arithmetic Logic Unit (ALU):** Responsible for performing arithmetic and logical operations.
- **Control Unit (CU):** Directs the operation of the processor by fetching instructions from memory, decoding them, and executing them.
- **Program Counter (PC):** A register that keeps track of the address of the next instruction to be executed.

1.2.2 Memory

Memory in a computer system is used to store data and instructions. It can be categorized into two main types:

- **Primary Memory (RAM):** Random Access Memory (RAM) is the main memory used by the CPU to store data and instructions that are currently in use. It is volatile, meaning that it loses its contents when the power is turned off.
- **Secondary Memory:** This includes storage devices such as hard drives, SSDs, and optical disks. Secondary memory is non-volatile and is used for long-term data storage.

1.2.3 Interconnections

Interconnections are the pathways that allow different components of the computer to communicate with each other. They include:

- **System Bus:** A collection of wires or traces on the motherboard that transfers data, address, and control signals between the CPU, memory, and I/O devices.
- **Data Bus:** Carries the actual data being transferred.
- **Address Bus:** Carries the address to and from which data is being transferred.
- **Control Bus:** Carries control signals to manage the operations of the CPU and other components.

1.2.4 Input/Output (I/O)

Input/Output devices allow the computer to interact with the external environment. They are divided into:

- **Input Devices:** Devices such as keyboards, mice, and scanners that provide data to the computer.
- **Output Devices:** Devices such as monitors, printers, and speakers that receive data from the computer and present it to the user.

Each component is essential for the overall functioning of a computer system, and their efficient interaction determines the performance and capability of the computer.

1.3 Assembly Language

Assembly language is a low-level programming language that is closely related to the architecture of a computer's CPU. It provides a way to write instructions that directly correspond to the machine code executed by the CPU, making it a crucial tool for understanding and controlling the underlying hardware.

1.3.1 Overview of Assembly Language

Assembly language consists of a set of symbolic instructions that the CPU can execute. These instructions are a more human-readable form of the binary code used by the machine. Each assembly language instruction typically corresponds to a single machine language instruction, making it a low-level language that interacts closely with the CPU's architecture.

1.3.2 Components of Assembly Language in Terms of CPU Architecture

- **Registers:** In assembly language, registers are used to hold data temporarily during the execution of instructions. Instructions often operate directly on registers, allowing for quick access to frequently used data. For example, an instruction might specify that data should be moved from one register to another or that an arithmetic operation should be performed using the values in two registers.

- **Arithmetic Logic Unit (ALU):** The ALU performs arithmetic and logical operations in response to assembly language instructions. Instructions such as addition, subtraction, and logical comparisons are executed by the ALU. Assembly language provides commands that directly manipulate the ALU to perform these operations.
- **Program Counter (PC):** The Program Counter is a special register that keeps track of the address of the next instruction to be executed. In assembly language, control flow instructions such as jumps and branches modify the value of the PC to alter the sequence of execution. For example, a 'jump' instruction changes the PC to a different memory address, allowing the program to execute a different set of instructions based on certain conditions.
- **Memory Addressing:** Assembly language allows programmers to specify memory addresses where data is stored or where instructions are located. Instructions can include memory addresses to read from or write to specific locations in memory. Addressing modes, such as immediate, direct, and indirect addressing, determine how these memory addresses are specified and accessed.

1.3.3 Example of Assembly Language Instructions

To illustrate how assembly language interacts with CPU architecture, consider the following example:

```
MOV AX, 5       ; Load the value 5 into register AX
ADD AX, 10      ; Add the value 10 to the value in register AX
MOV [1000], AX ; Store the result from register AX into memory location 1000
```

In this example: - 'MOV' is an instruction that moves data between registers and memory. - 'AX' is a register used to hold data. - The 'ADD' instruction performs an arithmetic operation using the ALU. - '[1000]' represents a memory address where the result is stored.

Through these instructions, assembly language allows precise control over the CPU's operations, making it a powerful tool for low-level programming and system design.

1.4 Important Assembly Language Operands

1.4.1 MOV

`MOV` is used to move data from one location to another. This operand transfers the value from the source to the destination.

Example:

```
MOV AX, 5
```

This instruction moves the value 5 into the register AX.

1.4.2 ADD

`ADD` adds two operands and stores the result in the destination operand. It performs arithmetic addition.

Example:

```
ADD AX, BX
```

This instruction adds the value in register BX to the value in register AX and stores the result in AX.

1.4.3 SUB

`SUB` subtracts the second operand from the first and stores the result in the destination operand.

Example:

```
SUB AX, 3
```

This instruction subtracts the value 3 from the value in register AX and stores the result in AX.

1.4.4 MUL

`MUL` multiplies the contents of the accumulator (usually AX) by the specified operand and stores the result in the accumulator.

Example:

```
MUL BX
```

This instruction multiplies the value in register AX by the value in register BX and stores the result in AX.

1.4.5 DIV

`DIV` divides the contents of the accumulator by the specified operand and stores the result in the accumulator.

Example:

```
DIV CX
```

This instruction divides the value in register AX by the value in register CX, with the quotient stored in AX and the remainder in DX.

1.4.6 JMP

`JMP` unconditionally jumps to the specified address, altering the flow of execution.

Example:

```
JMP 2000H
```

This instruction sets the instruction pointer to address 2000H, causing the CPU to jump to that location in the code.

1.4.7 CMP

`CMP` compares two operands and sets the flags based on the result of the comparison.

Example:

```
CMP AX, BX
```

This instruction compares the value in register AX with the value in register BX and updates the CPU flags accordingly.

1.4.8 JE

`JE` (Jump if Equal) jumps to the specified address if the zero flag is set, indicating that the previous comparison was equal.

Example:

```
JE 3000H
```

This instruction jumps to address 3000H if the zero flag is set.

1.4.9 CALL

`CALL` calls a procedure at the specified address and saves the return address on the stack.

Example:

```
CALL 4000H
```

This instruction calls a procedure located at address 4000H and pushes the return address onto the stack.

1.4.10 RET

`RET` returns from a procedure, popping the return address from the stack and continuing execution at that address.

Example:

```
RET
```

This instruction returns to the address saved on the stack from a procedure call.

1.4.11 PUSH

`PUSH` pushes data onto the stack, which is used for temporary storage of values.

Example:

```
PUSH AX
```

This instruction pushes the value in register AX onto the stack.

1.4.12 POP

`POP` pops data from the stack into a register or memory location.

Example:

```
POP BX
```

This instruction pops the top value from the stack into register BX.

1.5 Understanding the Relation of CPU and Assembly Language

In the process of software development, the code written by developers in high-level programming languages must be translated into a form that can be executed by the CPU. This process involves several stages of compilation, ultimately producing a binary executable.

1.5.1 Compilation Process

1. **High-Level Code**: The source code is written in a high-level programming language such as C, C++, or Java. This code is human-readable and includes abstractions that are not directly executable by the CPU.

2. **Compilation**: The high-level code is passed through a compiler, which translates it into intermediate representations. During this stage, the compiler may generate assembly language code or intermediate code, depending on the compiler's design and target architecture.

3. **Assembly**: If the compiler generates assembly language code, this code is then processed by an assembler. The assembler translates the assembly instructions into machine code, which consists of binary instructions that the CPU can execute directly. In this step, assembly instructions are converted into binary opcodes that represent low-level operations.

4. **Linking**: After assembly, the resulting object code is combined with other object files and libraries by a linker. The linker resolves references between different parts of the program and generates the final executable binary.

5. **Executable Binary**: The final product is a binary file that contains machine code instructions. This binary code is a sequence of binary digits (0s and 1s) that represent the operations to be performed by the CPU.

1.5.2 Assembly Language and Binary Instructions

Even though modern compilers and assemblers often streamline the process, the essence of executing a program on a CPU involves binary instructions. These instructions, whether generated directly from high-level code or via assembly language, are understood by the CPU.

Each binary instruction corresponds to a specific operation that the CPU can perform, such as arithmetic operations, data movement, or control flow changes. Assembly language serves as a more human-readable intermediary,

allowing programmers to write code in a form that is closer to the machine's language.

In some cases, compilers may generate machine code directly from high-level code, bypassing the explicit assembly stage. However, the final machine code remains composed of binary instructions that ultimately correspond to operations specified by assembly language.

In summary, the process of compilation translates high-level code into binary instructions that the CPU can execute. This process may include an intermediate assembly language stage, but the end result is always a binary format that the CPU can interpret and execute.

1.6 Assignment

This assignment aims to familiarize you with the process of compiling C++ code and examining the resulting assembly code. You will also practice writing C++ code based on the provided assembly code. You will need a laptop or PC with Ubuntu and superuser access to install the necessary tools.

1.6.1 GitHub (20 points)

- Sign up for GitHub.
- Create a public repository named `yourname-SKJ-Lab`.
- Create a `README.md` file in your repository that includes your name and NIM.
- Create a folder named `Assignment1` in the repository.
- Put all of the code related to this assignment in the `Assignment1` folder.
- Include the link to your GitHub project in the PDF file that you will submit to Google Classroom.

1.6.2 First Task: C++ Code to Assembly (50 points)

1. **Write a Simple C++ Program (10 points)**

   Write a C++ program that adds two integers.

2. **Compile the Code (10 points)**

   Use the 'g++' compiler to compile the C++ code. Open a terminal and run the following command:

```
g++ -o add_numbers add_numbers.cpp
```

This command compiles the code and creates an executable file.

3. **Disassemble the Code (10 points)**

   Disassemble the compiled executable to view the generated assembly code. Use the ‘objdump‘ command as follows:

   ```
   objdump -d add_numbers
   ```

   This will display the assembly code corresponding to the compiled binary.

4. **Write a Makefile (10 points)**

   Create a ‘Makefile‘ that includes the following targets:

   - ‘all‘: Compiles the code.
   - ‘dump‘: Disassembles the compiled code.
   - ‘clean‘: Removes all output files created by ‘make all‘ and ‘make dump‘, but not the source code.
   - ‘run‘: Runs the compiled executable.

   Example ‘Makefile‘ (may not run, you should write it yourself):

   ```
   all: add_numbers

   add_numbers: add_numbers.cpp
       g++ -o add_numbers add_numbers.cpp

   dump: add_numbers
       objdump -d add_numbers > add_numbers.asm

   clean:
       rm -f add_numbers add_numbers.asm

   run: add_numbers
       ./add_numbers
   ```

5. **Document Your Work (10 points)**

   Document your work in a PDF file. Include screenshots or outputs showing the results of the ‘make‘ commands. Submit this PDF file to Google Classroom.

1.6.3 Second Task: Assembly to C++ (30 points)

1. **Analyze the Provided Assembly Code (10 points)**

   Consider the following assembly code (for illustration purposes; it may not compile directly):

```
section .data
    num1 dw 5
    num2 dw 10
    result dw 0

section .text
    global _start

_start:
    mov ax, [num1]
    imul ax, [num2]
    mov [result], ax

    ; Exit the program
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

   Explain and describe each line of the code.

2. **Write the Equivalent C++ Code (10 points)**

   Based on the provided assembly code, write a C++ program that performs the same functionality. The C++ program should produce the same result as the assembly code.

3. **Write a Makefile (10 points)**

   Create a ‘Makefile‘ similar to the one used in the first task, with targets for ‘all‘, ‘clean‘, ‘dump‘, and ‘run‘.

1.6.4 Submission

Submit a single PDF file to Google Classroom. The PDF should include:

1. Your name.
2. The GitHub link to your project.
3. Documentation of your assignment, including:
    - Screenshots or outputs showing the results of the `make` commands.