Marmara University

Faculty of Engineering

CSE 3015 - Digital Logic Design

Term Project Report

Emre Gürkan 150121823

Table of Contents

Introduction

In this project, as requested, I built a fully functional, twenty-bit basic general purpose processor. The processor is roughly consists of five parts:

- Instruction Memory:
  - Program Counter
  - Electrically Programmable Erasable Read-Only Memory
  - Instruction Register
- Data Memory:
  - Memory Address Register
  - Static Random-Access Memory
  - Memory Data Register
- Register File
- Arithmetic Logic Unit
- Control Unit

Instruction Set Architecture

The opcode is four-bit wide. The instruction set architecture addresses thirteen different instructions. The last three addresses are reserved. Although there are thirteen instructions, the processor can handle eighteen instructions. The arithmetic instructions (from 0x1 to 0x5) can be unwrapped to ten instructions with the extra five instructions where the immediate values are handled.

The processor decides whether an arithmetic operation is in the immediate mode or not by checking if the seventh bit (from MSB) is high or not. I choose this trade-off, sacrificing an immediate bit, in order to reserve three extra instructions that can be used in future.

There are some instructions such as 0x8 (unconditional jump) which can seem to support sixteen bit addresses but the address width of our processor is really ten. So the processor will only interpret the ten significant bits.

I have to mention that I have failed to implement a stack. So the stack instructions (0x9 and 0xa) are basically NOOp.

Assembler

The assembler is written in Java programming language using Java SE 19. In the project folder “out” is where the precompiled binary files are located. You may use them accordingly if you do not wish to compile it yourself.

“How to compile ?”

In order to compile the assembler, change the current working directory to “src” folder in the project. Then run the command:

```
“$ javac Main.java -d ../out”
```

“How to run ?”

In order to run the assembler, change the current working directory to “out” folder in the project. Then run the command:

```
“$ java Main $ASSEMBLY_FILE”
```

The $ASSEMBLY_FILE is the variable that you should provide. There are some sample assembly files located in the “samples” folder in the project. You may assemble them accordingly:

```
“$ java Main ../../samples/fib.s”
```

You will get the output “Done.” if it successfully assembles it. If there is a warning then most probably you provided a stack instruction which as I mentioned, not supported.

Otherwise it will output the error message which indicates the cause of the problem.

The assembler will generate the “a.hex” file in the “out” folder in order to use it in Logisim. Use this file by loading it to the instruction memory.

Logisim Design

I built almost every component by myself even if I used the built-in version (such as D Flip-Flops). In order to keep this report short I will only get into the details of the control unit since the other components are really very common and their design is very simple.

The control unit consists of eleven finite state machines with a total of twenty-four signals in order to keep things debuggable. Every finite state machine has four different states with a four bit input of the opcode. In the first state 0x0, the input bits are treated as "don't care" bits since this state is the "fetch" state. You can check the truth table of the signals in the resources section (beware it is hand-written).

The control unit works with the inverted clock signal since signals are generated in the time between clock pulses as expectedly. Since there are four states, every instruction takes four clock cycles to execute.

It should be mentioned that I used the method of microprogramming in the first iteration since the design was very unstable and it is the easiest method for debugging. I changed the design all over several times. I added new components such as memory data registers and some others. It took some real amount of work to get this processor running. Once the processor was stable I designed the control unit as a finite state machine.

Thank you.

Resources

Control Unit Truth Table