CSE 140 Project Report

Team 28 : Andrew Nguyen and Vincent Zhang
Date : 05/05/2001

1. Single-cycle MIPS CPU

a. Overview

The code is all written in the file 'singleCycle.cpp' and contains all of the main functions and variables required to read all of the binary instructions listed in the 'sample_binary.txt' file and run them through a single-cycle MIPS CPU.

b. Code Structure

i. Functions

Primary :

The primary functions that were used to run the code were Fetch, Decode, Execute, Mem, and writeBack.

These functions obtain and read the binary instructions from the sample_binary.txt file, then decode the instructions opcode, rs,rt,rd,imm, and funct depending on what type of function that they may be. While decoding the variables, it also converts their values from binary into an integer. After this process is completed, the execute function is called and runs the decoded instruction. Once completed, the mem function is used to store or load the values that were created through the execute function**.** After the memory portion, the writeback portion of the code is reached where the completed value is stored in the designated register, finishing the cycle and the next instruction is fetched. This cycle continues until all instructions listed in sample_binary.txt is completed.

Inside Fetch:

This function allows for the CPU to access the next instruction through the **PCForward** variable and calls the decode function using the current instruction.

Inside Decode:

In the beginning of the function, it uses the first 6 bits and finds the opcode of the instruction. Using this information, it is able to decide what type the instruction may be, from I, J , or R type. After that, it separates the instruction to its core parts: **rs, rt, rd, immediate, funct**. Following this, the code uses the opcode again to find the ALUop through the control unit function, which is used alongside the information from the decode to run Execute.

Inside ControlUnit:

Using the opcode provided by the decode function, this function tells what operation that the ALU , indicated by **ALUop,** should run through the CPU.

Inside Execute:

This function uses the **ALUop** from the control unit as well as **rt, rs , imm**. Once the operation is run, the total_clock_cycles is incremented by 1 and the values of each variable is changed according to the operation completed.

Inside Mem:

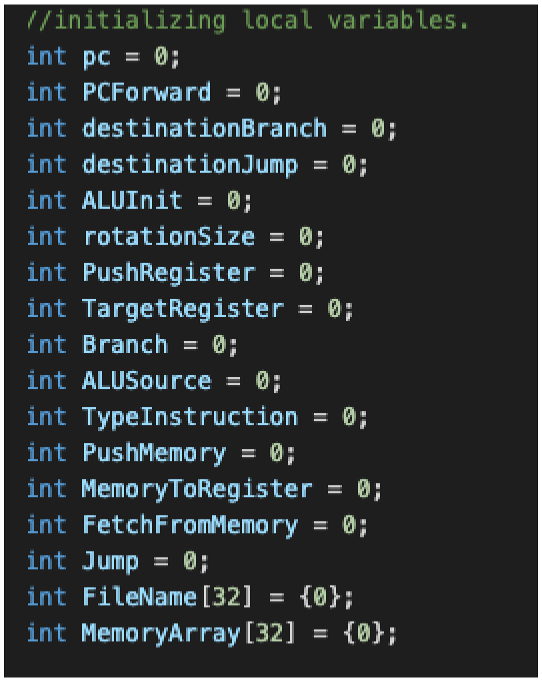
Within this function, tells the CPU to either **lw** or **sw**. When **lw** is called, it moves from this stage to the writeback stage with the target register and its value, which is stored in our memoryArray. On the other hand, when sw is called, the value is stored inside the **memoryArray** at the targeted location and the **pc** is moved to the next instruction.

Inside Writeback:

In this function, we display the value of the modified register when the CPU is told to **lw**.

---

ii. Variables



```
//initializing local variables.
int pc = 0;
int PCForward = 0;
int destinationBranch = 0;
int destinationJump = 0;
int ALUInit = 0;
int rotationSize = 0;
int PushRegister = 0;
int TargetRegister = 0;
int Branch = 0;
int ALUSource = 0;
int TypeInstruction = 0;
int PushMemory = 0;
int MemoryToRegister = 0;
int FetchFromMemory = 0;
int Jump = 0;
int FileName[32] = {0};
int MemoryArray[32] = {0};
```

In the image to the left displays the critical variables that allow our CPU to complete binary instructions that are given in the text files.

The variables **pc** and **PCFoward** are counters that are incremented to allow for the CPU to know which instruction that they are currently on and their next instruction.

destinationBranch are used to forward

*Note : **ALUInit** and **destinationJump** is used in the pipelined version of the code*

rotationSize is a counter of the amount of cycles that the CPU has gone.

The **TargetRegister**, **ALUSource, MemoryToRegister, PushRegister, FetchFromMemory, PushMemory, Branch, Jump, TypeInstruction,** and **ALUop** are used to tell the CPU what operation to run either logical or arithmetic.

Lastly, the two arrays **FileName** and **MemoryArray** are used to store the values of registers and data memory. In the beginning, the main function initializes them to values specified by the project outline. Which are:

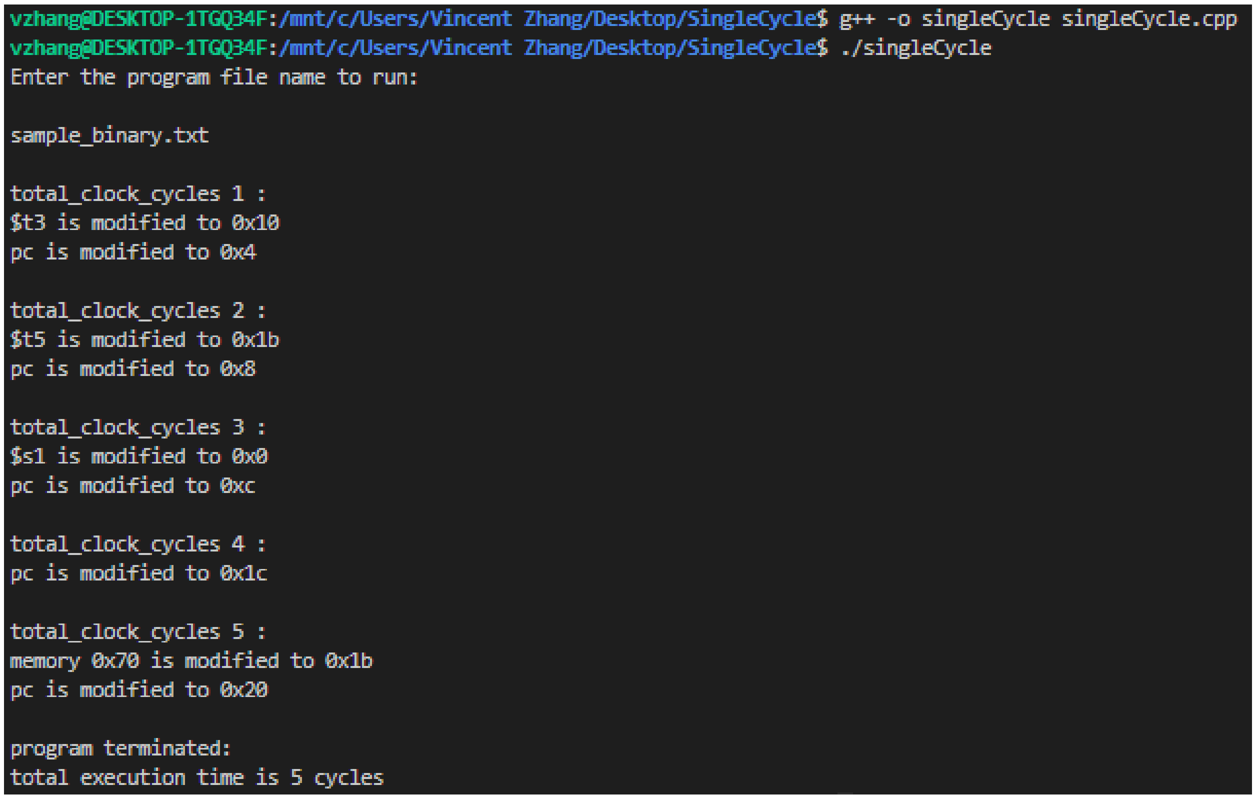
```
   FileName[16] = 112;
   FileName[9] = 32;
   FileName[10] = 5;
   MemoryArray[29] = 16;
   MemoryArray[28] = 5;
```

Note:
The values are hex converted into integers.

---

c. Execution Results

In order to run the code, you compile the file using 'g++ -o singleCycle singleCycle.cpp'. Once it is finished compiling, run the code using './singleCycle' and enter the 'sample_binary.txt' file. Below is the sample output :



```
vzhang@DESKTOP-1TGQ34F:/mnt/c/Users/Vincent Zhang/Desktop/SingleCycle$ g++ -o singleCycle singleCycle.cpp
vzhang@DESKTOP-1TGQ34F:/mnt/c/Users/Vincent Zhang/Desktop/SingleCycle$ ./singleCycle
Enter the program file name to run:

sample_binary.txt

total_clock_cycles 1 :
$t3 is modified to 0x10
pc is modified to 0x4

total_clock_cycles 2 :
$t5 is modified to 0x1b
pc is modified to 0x8

total_clock_cycles 3 :
$s1 is modified to 0x0
pc is modified to 0xc

total_clock_cycles 4 :
pc is modified to 0x1c

total_clock_cycles 5 :
memory 0x70 is modified to 0x1b
pc is modified to 0x20

program terminated:
total execution time is 5 cycles
```

d. Challenges and Limitations

Some challenges that rose up while implementing the code were understanding how the functions would interact with each other and how the memory should be stored. But once figured out, it was as simple as implementing it. Other than that, there were not many other limitations in our development of the single-cycle CPU.

2. Pipelined MIPS CPU

a. Overview

The code is all written in the file ‘pipeline.cpp’ and contains all of the main functions and variables required to read all of the binary instructions listed in the ‘sample_binary.txt’ file and run them through a pipelined MIPS CPU.

b. Code Structure

i. Functions

Primary :

The primary functions that were used to run the code were binDeci, toPush, consoleDisplay, binHex, binDec, decode, checkState,

binDeci:

This function allows for the binary code in the sample_binary.txt file to be converted into decimal values.

ConsoleDisplay:

This function allows for the terminal to display the correct format for the output after reading the contents of sample_binary.txt.

toPush:

This function allows for the memory to be modified and count the dmem location in the array.

binHex:

This function allows for the binary code in the sample_binary.txt file to be converted into hexidecimal values.

binDec:

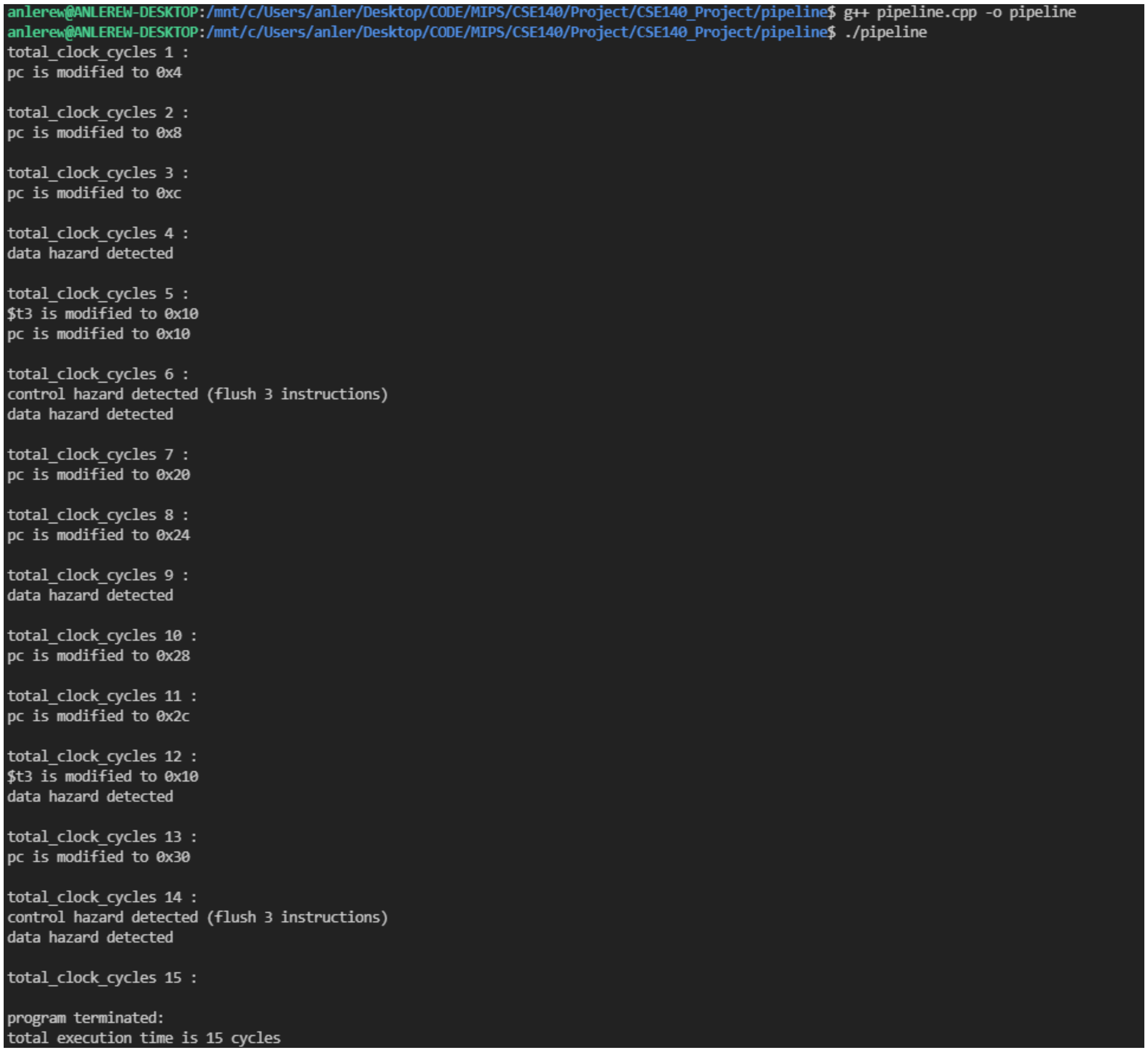
This function allows for the complete check if the output has successfully exported as a decimal value.

decode:

In the beginning of the function, it uses the first 6 bits and finds the opcode of the instruction. Using this information, it is able to decide what type the instruction may be, from I, J , or R type. After that, it separates the instruction to its core parts: **rs, rt, rd, immediate, funct**.

c. Execution Results

In order to run the code, you compile the file using ‘g++ -o pipeline pipeline.cpp’. Once it is finished compiling, run the code using ‘./pipeline’ and enter the ‘sample_binary.txt’ file. Below is the sample output :



```
anlerew@ANLEREW-DESKTOP:/mnt/c/Users/anler/Desktop/CODE/MIPS/CSE140/Project/CSE140_Project/pipeline$ g++ pipeline.cpp -o pipeline
anlerew@ANLEREW-DESKTOP:/mnt/c/Users/anler/Desktop/CODE/MIPS/CSE140/Project/CSE140_Project/pipeline$ ./pipeline
total_clock_cycles 1 :
pc is modified to 0x4

total_clock_cycles 2 :
pc is modified to 0x8

total_clock_cycles 3 :
pc is modified to 0xc

total_clock_cycles 4 :
data hazard detected

total_clock_cycles 5 :
$t3 is modified to 0x10
pc is modified to 0x10

total_clock_cycles 6 :
control hazard detected (flush 3 instructions)
data hazard detected

total_clock_cycles 7 :
pc is modified to 0x20

total_clock_cycles 8 :
pc is modified to 0x24

total_clock_cycles 9 :
data hazard detected

total_clock_cycles 10 :
pc is modified to 0x28

total_clock_cycles 11 :
pc is modified to 0x2c

total_clock_cycles 12 :
$t3 is modified to 0x10
data hazard detected

total_clock_cycles 13 :
pc is modified to 0x30

total_clock_cycles 14 :
control hazard detected (flush 3 instructions)
data hazard detected

total_clock_cycles 15 :

program terminated:
total execution time is 15 cycles
```

d. Challenges and Limitations

The pipeline CPU was quite the challenge as we had a very hard time to understand what was happening within our code. In the end we pulled through.