University of California - Merced

Single Cycle MIPS CPU

Team Members:

Samuel Carlos

Professor: Hyeran Jeon

Computer Architecture 140

May 6, 2020

Code Structure:

The code structure for the first part was based on the project details. Functions I used were Fetch(), Decode(), ControlUnit(), Execute(), Mem(), and Writeback(), all coming from the project details.

First Part:

I used iostream for input and output, fstream to open files, string for string variables, bitset for the binary machine code, and sstream and iomanip were used for hex and decimal conversions.

Global values initialized include control signals, program counter, jump target, total clock cycles, branch target, alu zero, registerfile[], were all used for the first implementation. Other control signals were added for the second implementation that I will discuss further down. Memory array and register file array are also populated with values used in the project description prior to running program.

Main function asks for the user input of a filename filled with machine code separated line by line. Using the file, we iterate through the lines to find the max address we will use to ensure we stay within the lines of machine code in a while loop. In the while loop that checks for line boundaries in the file, we begin the single cycle processor with Fetch(). Main will also print out the total clock cycles after the while loop.

Fetch() uses the filename and program counter(pc). The program counter finds the line we need and calls the Decode() function. Fetch() also increases the pc count by 4 for the next cycle.

Decode() uses the machine code binary string from Fetch(), and begins extracting data from the string. Using substrings, we can snip out parts of the machine code that correlate to the instruction format. Starting with the opcode, we use the first 6 bits in the machine code (31 – 26), and we find the instruction type. After finding the instruction type, we can begin extracting the bit depending on the instruction formats such as R-type having opcode, rs, rt, rd, shamt, and funct. We call the ControlUnit, using opcode and funct we can set all the control signals for every type of instruction we use and return the alu-op code for Execute(). We call Execute() within Decode and send over the alu_op code and the instruction format information such as rs, address, and immediate values.

ControlUnit() is called from Decode() and returns the alu-op code. Within ControlUnit() is also the Arithmetic Logic Control. Using the opcode, we can find the operations used and set the control signals that will be used for the operations. The ALU Control is what will generate the alu-op code used in execute. The ALU Control will use the funct and determine the type of R-Instruction operation and will set the alu-op to the needs of the operation, such as the adding for add. It will need the opcode for other functions as well, such as beq needing subtract, and lw needing addition. Again, alu-op will be sent to execute to determine the arithmetic needed along with the global control signals being set depending on the operation and funct value.

Execute() will do the calculations. Using the alu-op code from ControlUnit() and the values from decode, we have everything we need to begin computation. The control signals we be the basis to our logic code that will help determine what we need to do with the values. For example, if regDst set to true, this means we will use rd as the register where data will be written to. If regDst is set to false, then rt will be the register where data will be written to. Load word and store word are special where they will access the memory. Control signal memRead means that load word is likely being used and will read the data from memory and store it into a register using control signal memToReg. After computations, we again use control signals to determine what needs to be sent to Writeback() which will let us know what was changed as well change the values within register.

Mem() will be called within execute for sw and lw logic paths. Mem() can return a value from memory which will be sent as a result to Execute(). Store word and load word send the address of a register or a value of a register depending on what they need to Mem(). Mem() will change it's own memory through sw, and return a result for load word.

Writeback() is called from Execute() after the results, address and the like have been calculated. The results will be sent to Writeback() where again if the control signal allows it, it will change the value of registers. Writeback() will let us know what was changed. Finally the total clock cycle is incremented to keep a count of the amount of lines that were fetched().

Second Part:

Three new global variables were created for jal and jr: jalRegDst, jalMemToReg, and jrJump.

Main() and Fetch() were unchanged from the first part. Decode was changed slightly to account for jal, and jr, but still sends the address of these functions.

ControlUnit was changed to account for the new global variables. For jr, the return address for regiser write back needed to be $ra, so this was set with the jrJump control signal true.

Execute() was changed to calculate the next_pc to be sent to $ra, and change the current pc to the address in the jal machine instruction. Results are still sent to Writeback()

Mem() was left unchanged as the jr and jal did not need access to memory, only the pc.

Writeback() was changed to account for jalMemToReg where register ra was updated with the pc address. Jr was also an instruction type so I set a logic so that the return address was rs instead of rd using the jrJump control signal.

Code Compile:

```
g++ main.cpp -o ./main
./main
```

Execution Result Screen Shots:

Sample_part1.txt -

```
Enter the program file name to run:

sample_part1.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
```

Sample_part2.txt -

```
Enter the program file name to run:

sample_part2.txt

total_clock_cycles 1 :
$ra is modified to 0x4
pc is modified to 0x8

total_clock_cycles 2 :
$t0 is modified to 0x7
pc is modified to 0xc

total_clock_cycles 3 :
$v0 is modified to 0x3
pc is modified to 0x10

total_clock_cycles 4 :
pc is modified to 0x4

total_clock_cycles 5 :
pc is modified to 0x14

total_clock_cycles 6 :
memory 0x20 is modified to 0x3
pc is modified to 0x18

program terminated:
total execution time is 6 cycles
```