**Project Report**

Team: Eric Ponce

CSE 140

**Delegation of Tasks:**

I decided to challenge myself and take on the project by myself. I didn’t want any assistance while working on the project to see if I could get it done on my own. That being said, the initial portion of the project was pretty straightforward.

**Code Breakdown (Initial 10 instructions):**

Global variables:

pc = 0 # This is the program counter which keeps track of the current instruction

next\_pc = 0 # This is the next program counter which tracks the next instruction

opcode = '0' # Stores the opcode of an instruction

rd = '0' # Stores the destination register of an instruction

rs1 = '0' # Stores the first register of an instruction

rs2 = '0' # Stores the second register of an instruction

funct3 = '0' # Stores the funct3 of an instruction

funct7 = '0' # Stores the funct7 of an instruction

imm = '0' # Stores the immediate value of an instruction

sign\_extended\_imm = '0' # Stores the sign extended immediate of a function

jal\_imm = 0 # Stores the immediate value for jal and jalr instructions

register1\_val = 0 # Stores the value that the first register holds

register2\_val = 0 # Stores the value that the second register holds

RegWrite = 0 # Stores the RegWrite control unit value

Branch = 0 # Stores the Branch control unit value

ALUSrc = 0 # Stores the ALUSrc control unit value

ALUOp = 0 # Stores the ALUOp control unit value

MemWrite = 0 # Stores the MemWrite control unit value

MemtoReg = 0 # Stores the MemtoReg control unit value

MemRead = 0 # Stores the MemRead control unit value

alu\_ctrl = '0' # Stores the aluctrl value

rf = ['0'] \* 32 # Declares an array of size 32 with each entry initialized to 0 by default

rf[1] = '0x20' # Stores 0x20 in position 1 of the register file array

rf[2] = '0x5' # Stores 0x5 in position 2 of the register file array

rf[10] = '0x70' # Stores 0x70 in position 10 of the register file array

rf[11] = '0x4' # Stores 0x4 in position 11 of the register file array

rf[8] = '0x20' # sample\_test2

rf[10] = '0x5'

rf[11] = '0x2'

rf[12] = '0xa'

rf[13] = '0xf'

alu\_zero = 0 # Stores the value of alu zero

d\_mem = ['0'] \* 32 # Declares an array of size 32 with each entry initialized to 0 by default

d\_mem[28] = '0x5' # Address: 0x70 # Storing 0x5 in position 28 of the data memory array

d\_mem[29] = '0x10' # Address: 0x74 # Storing 0x10 in position 29 of the data memory array

branch\_target = 0 # Stores the value of the branch target

total\_clock\_cycles = 0 # Stores the value for the number of clock cycles

new\_address = '0' # Stores the value of the new address generated when adding the immediate offset to the target address for load and store word

i = 0 # Keeps tracks of instruction line from text file

**Fetch() function**

This function starts out by asking the user what text file we wish to read from. The user response is stored and used to open the user specified file. The file is read and the instructions in the file are stored in an array. One instruction is read at a time until there are no instructions left. In this function, we keep track of the ‘pc’ and ‘next\_pc’ values which are updated before we call the Decode() function. If at any point the ‘branch\_target’ value gets updated to be greater than 0, the function knows that we are branching. The variable ‘i’ is updated by adding the array position of the current instruction to the ‘branch\_target’. We reset ‘branch\_’target’ back to zero and restart the loop which reading the instructions, this time starting at the new position of ‘i’. Once there are no more instructions to be read, the function prints “Program terminated” and the total clock cycle count.

**Decode() function**

This function is called by the fetch function every time an instruction is read. The purpose of this function is to read the 32 bit binary and convert it to assembly language. Depending on the instruction type, we obtain the opcode, registers, immediate values, and funct3/7 values from the function. Once the instruction is decoded and the program knows what registers to deal with, the values those registers contain are stored in the variables ‘register1\_val’ and ‘register2\_val’. This function is also responsible for generating the sign extended immediate values. Once an instruction is fully decoded the program calls the ControlUnit and Execute functions.

**ControlUnit() function**

The control unit function is responsible for generating control unit signals and alu control values for each instruction. The variables being updated are ‘RegWrite’, ‘Branch’, ‘ALUSrc’, ‘ALUOp’, ‘MemWrite’, ‘MemtoReg’, ‘MemRead’, and ‘alu\_ctrl’. The program will use some of these values later on for other functions, so it is important that the values be changed accurately.

**Execute() function**

This function is responsible for handling the arithmetic needed for the specified instruction. Based on the ‘alu\_ctrl’ value generated by the ControlUnit() function, the program can decide what arithmetic operation needs to take place (add, sub, or, and). For load and store word instruction, the new address is generated by adding the immediate offset to the target address. This new address tells the program where to load a value from or where to store a value. The Mem() function is called after the new address has been determined. For branching if equal, if the register 1 value minus the register 2 value is equal to 0, the program is told to branch. The ‘alu\_ zero’ value changes to 1, indicating a branch, the ‘branch\_target’ is generated, and the total clock cycles and the value of pc are printed. If the register 1 value minus the register 2 value is not 0, this program knows this is not the branch is not taken. ‘alu\_zero’ is set to 0 and the total clock cycles and pc value are printed. The rest of the instructions are different forms of addition, subtraction, and, and or which are stored in a variable called ‘value’. The value is passed as an argument to the Writeback() function.

**Mem() function**

This function is only used for load and store word instructions. The program checks the current value of ‘RegWrite’ to determine which instruction is currently being dealt with. If ‘RegWrite’ is equal to 1, it is a load word instruction and if it is equal to 0, it is a store word instruction. For load word instructions, the new address calculated in the Execute() function is used to access the data memory of that specified address. The value that address contains is stored in a variable called value. That value is passed as an argument to the Writeback() function. For store word instructions, the value of register 2 is stored in a variable called value and passed as an argument to the Writeback() function.

**Writeback() function**

This function is responsible for storing the computation result back to a destination register. For store word instructions, the value that was passed as an argument is stored at the specified new address in data memory that was calculated in the Execute() function. The total clock cycles, memory location that was modified, and pc value are printed. For every other instruction that calls the Writeback() function, the value that was passed as an argument is stored in the destination register. The total clock cycles, value that the destination register was changed to, and pc value are printed.

**Jal and Jalr Implementation:**

In order to achieve functionality for these two instructions, I had to alter my original code slightly. The fetch() function is still reading the instructions from the sample text file. I added a new condition to check if a branch target is ever not equal to 0. Then another check takes place to see if the opcode for the current instruction belongs to either jal or jalr. This tells my program that we are jumping to a desired location. The decode() takes in one instruction at a time from the fetch() function. This is a 32-bit binary instruction from the sample text file. The decode function() breaks down the instruction and tells the program whether is it a jal or jalr instruction. The specific components of either function are also provided. If the instruction is jalr, the rs1 value is obtained and ready to be passed to the execute() function. For each function, the immediate is being store in a global variable called jal\_imm. Once decoded, the ControlUnit() function is called to generate the control signals and alu\_ctrl value. After these values are generated, the Execute() function is called. For the jal instruction, the branch\_target is calculated by dividing the jal\_imm by 4 and adding i after. i is the value of the current instruction we are reading. If the first instruction is being read, i would be 0. The writeback() function is called to store the pc value in the ra register and the jump is completed shortly after. For the jalr instruction, the branch target is calculated by adding the rs1 value to the jal\_imm and dividing the sum by 4. The writeback() function is called to store the next\_pc value in the ra register and the jump is performed shortly after.

sample\_part1.txt

A screenshot of a computer program

Description automatically generated

sample\_part2.txt

A screenshot of a computer

Description automatically generated