CSE 140 Project Report

Team *ID* : Amy Chan
Date: May 11th

1. Single-cycle RISC-V CPU

1.1 Overview

This is a single-cycle RISC-V CPU program written in Python that executes RISC-V instructions with a single clock cycle per instruction. It uses the RISCV- pipeline stages of Fetch, Decode, Execute, Memory, and Writeback. The program reads the machine code from the text file and processes each instruction, updating the registers and memory based on the instruction.

1.2 Code Structure

The code is structured to use multiple functions with some helper functions to get machine code from a text file and decode the instruction to various fields to conduct the instruction. When the program is run, it asks the user for the program file name and kicks off the program using run_cpu function with the filename which calls other functions to decode the instructions and execute them.

1.2.1 Functions

Fetch(Instruction)

The argument instruction is a list of machine instructions. It uses the global variables pc and next_pc and gets the next instruction from memory based on the index what is calculated by dividing pc by 4. If the index is larger than the amount of instructions in the list, it returns None (which tells the program it's done). It also updates next_pc to pc+4. The instruction at the index calculated in the instruction list is returned to be decoded. This function is used in the run_cpu function to get the current instruction.

FindOp(code)

This function returns the instruction type based on the machine code. It takes the machine code instruction and gets the opcode from the machine code using the index. Based on the opcode, it returns the instruction type as a string. It is called in the Decode function to find the instruction type.

FindOperation(fields, itype, opcode, mcode)

This function takes in the fields, itype, opcode, mcode. Based on the itype and fields, it finds the operation. For R type instructions, it uses the funct3 and funct7 fields to find the operation. For I type instructions, there's different categories based on the opcode (load instructions, immediate operations, and jalr). S and SB type instructions use the funct3 field to find the instruction. The U type instruction is determined based on the opcode and there is only the jal operation in the UJ type instruction. This function is called in the Decode function, which takes fields from findI function , itype from findOp function, and the opcode from the instruction.

FindI(code, itype)

This function takes in the machine code and itype (which is found by the findOp function) to determine the fields based on the itype and return the fields as a dictionary with the field as the key and the field value as the value. Funct3, funct7, rs1, rs2, rd, and the immediate are determined based on the itype, which are then found through indexing the machine code. The immediate is calculated depending on the instruction type by calling the sign_extend function which returns the sign extended intermediate. This immediate calculation is crucial for calculating the branch target and later the jump target for the JAL instruction. The function returns the fields and is called in the Decode function.

Sign_extend(imm, bits)

This function does sign extension which takes in the immediate to be sign extended and the original bit width of the immediate value. The function converts the input to an interger if it's a binary string. It checks if the most significant bit is 1, which if it is 1. it'll subtract 2^bits to convert the unsigned integer to a signed integer. Afterwards it returns the sign extended integer. This function is used in the FindI function to find the immediate for the I, S, SB, and UJ instruction types.

Decode(instr)

The instr argument is 32 bit machine code. This function determines the insturction type, This determines the instruction type through calling the findOp function with instr, extracts the fields (rd, rs1, rs2, imm, funct3, funct7) using the findI function with arguments instr and itype, and identifies the operation using the findOperation function. Inside the Decode function, rd, rs1, rs2 is determined if these fields are in the fields, else it returns None or 0. The rs1, rs2, rd, rd register string (e.g. x10), immediate, and operation are returned in a dictionary

Execute(decoded)

This function does the operation that's specified in the decoded instruction. It performs ALU operations, updates the pc (also updates pc for branch and jump instructions), calculates the memory address, and sets alu_zero for branch instructions. It uses the global variables pc, alu_zero (to see if alu is zero for branches), branch_target (the target address for branch), and next_pc(the default next instruction address which is pc + 4). Program counter is updated based on the control signals. The ALU operations are done using variables a and b, with a being determined from the register and be either the immediate or register value. The ALU results are returned which are used in the WriteBack and Mem functions.

Mem(decoded, alu_result)

This takes in the decoded instruction and the alu results and handles the memory operations (load and store) used in memory operations. The alu_result is the memory address found in the Execute function. For lw, it takes the address and sees if the address is in a valid memory range and returns the 32 bit word from memory using integer division. For the sw, it uses the memory address and rs2 register value and writes the value to the memory if the memory address is valid.

Writeback(decoded, alu_result, mem_data)

This function takes in the decoded dictionary, alu_result, and mem_data. It checks if the destination register is valid and if the control signal Regwrite is there, if both conditions are true, it writes mem_data into the destination register if MemtoReg control signal is enabled (used for instructions like lw). Otherwise with the two conditions fulfilled and no MemtoReg, it writes the alu_result to the destination register (used for instructions like add).

ControlUnit(opcode, op_name)

This function takes in the opcode and the operation name (like "add") sets the control signals based on the operation which are used in other functions by enabling or disabling operations like memory writes. It resets the control signals to the default values and based on the instruction; it updates the control signals dictionary to enable different signals depending on the instruction.

Run_cpu(filename)

This takes in the file name and removes white spaces, storing each line into a list. It uses the global variables pc (initialized to 0) and total_clock_cycles (initialized to 1).

In the loop while pc divided by 4 is within the bound of the instruction list and the cycle count doesn't exceed max cycles, the function calls the Fetch, Decode, ControlUnit, Execute, Mem, and WriteBack functions to stimulate a CPU cycle. It prints modifications to a register or memory when modified and prints the pc after each cycle (adding to the total_clock_cycles).

1.2.2 Variables

pc is program counter which tracks the memory address of the current instruction executed. It is intialized as 0 and updated in the Execute() function for branchs and jumps otherwise it'll increase by 4 by default. Next_pc holds the address of the next instruction which is by default pc + 4, this is set in the Fetch() function but is overwritten in the Execute() function if branch or jump is taken.

Rf is a list with 32 registers from x0-x31 which is initialized to be 0. This is used in the WriteBack() function when the RegWrite control signal is 1 (for instructions like add). D_mem is a list that represents the data memory with 32 words and is used in the Mem() function for instructions such as lw. The values of the rf and d_mem for part 1 and part 2 are preset, which can be used through uncommenting the sections related to the parts.

Control_signals which dictates CPU behavior is a dictionary of control signals that's set by the ControlUnit() function. It includes RegWrite, Branch, MemRead, MemWrite, MemtoReg, ALUSrc, JAL, and JALR. This is used in the Execute(), Mem(), and WriteBack() functions to do different operations based on the control signals.

Total_clock_cycles counts the number of CPU cycles executed and is increased by 1 after each full instruction cycle.

alu_zero is used as a flag to check if the alu result is zero for branches, which is set to true if rs1 and rs2 are equal to each other in beq. Otherwise, this flag is set at zero.

branch_target is used to store the branch target address and is calculated in the Execute() function with pc + imm and is used to update the pc if alu_zero is true.

1.3 Execution Results

When the program is run, the program prompts for the text file's name for it to look over, which is inputted. The program executes each instruction and prints register and memory updates. This is the program output for sample part 1 text file with the part 1 register and data set.

```
Enter the program file name to run:
samplepart1.txt
total_clock_cycles 1 :
x3 is modified to 0x10
pc is modified to 0x4
total_clock_cycles 2 :
x5 is modified to 0x1B
pc is modified to 0x8
total_clock_cycles 3 :
pc is modified to 0xC
total_clock_cycles 4 :
x5 is modified to 0x2B
pc is modified to 0x10
total_clock_cycles 5 :
x5 is modified to 0x2F
pc is modified to 0x14
total_clock_cycles 6 :
memory 0x70 is modified to 0x2F
pc is modified to 0x18
```

1.4 Challenges and Limitations

Managing to get the memory updates and pc to update correctly was a challenge, especially for the branch instructions. This may be due to the pc being calculated incorrectly and issues in implementing branch instructions to update the pc with the branch target.

Limitations include lack of dependencies handling (if add x1, x2, x3 and then the instruction lw x04, 0(x1) would not work) and there is a hard coded cycle limit of 10 which is used to prevent infinite loops. The max_cycles variable can be updated to be higher to account for this limitation.

2. Extended RISC-V CPU

2.1 Overview

The findOperation, findI, Execute and Control Unit functions were modified to handle jal and jalr. New control signals for JAL and JALR are added to the control_signals dictionary.

2.2 Baseline Code Structure

The decoding functions were updated to decode the JAL and JALR instructions, with the findI instruction crucial for calculating the immediate offset for JAL and JALR. The Execute() function is updated to handle jump addresses and return addresses. The ControlUnit() function is updated to add control signals for JAL and JALR.

2.2.1 Functions

FindOperation(fields, itype, opcode, mcode)

The determination of the jalr and jal operation is added based on the instruction type and opcode.

FindI(code, itype)

For the jal command (UJ type instruction), the offset is calculated through finding the 4 parts of the immediate in the machine code (split from the immediate) and reconstructing the immediate offset by adding up the 4 parts in the appropriate order with the result sign extended to 32 bits. This is then shifted left by 1 to convert to byte offset. For jalr, it uses a sign extended immediate using the bits from 0 to 11. This immediate and the register destination are stored in the fields dictionary.

Execute(decoded)

The function was updated to calculate the return address for JAL using the next_pc and the pc is updated with pc + the immediate, the return address is returned to be saved in rd. For JALR, the return address is determined with next_pc and it calculates pc with rs1 plus the immediate with the least significant bit forced to be zero.

ControlUnit()

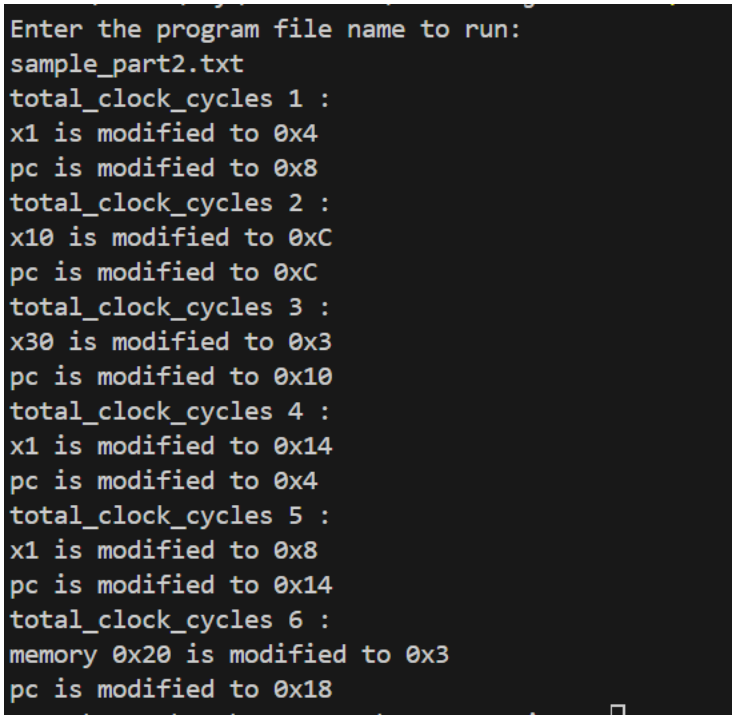
Jal and Jalr control signals are added which are used to enable register writeback and pc updates.

2.2.2 Variables

Pc which is the program counter to track the current instruction address and next_pc which is the default next instruction (pc+4) which is modified when there's a jump.
Branch_target is used to store the jump target for branches.
Alu_zero used for branch decisions.
Control signals which is a dictionary holding the values for the control signals.

2.4 Execution Results

The program prompts the user to input a text file name.



2.5 Challenges and Limitations

Calculating the immediate for jal and jalr was the main challenge, as incorrect calculations of the immediate led to the pc not updating correctly (such as to 1000). Reconstructing the immediate offset for jal also needed to be deliberate to consider for the 4 different parts. Once the immediate offset calculation was coded correctly, the pc issue was resolved.

The limitation is a lack of check for invalid jump targets and dealing for dependencies, which should be implemented to check for valid jump targets.

3. Pipelined RISC-V CPU (optional – only for extra credit)

3.1 Overview

This program uses sections of the code from the Single Cycle CPU and has updated code to implement the five stage pipeline (IF, ID, EXE, MEM, WB). New implementations include pipeline registers, hazard detection, and more.

3.2 Baseline Code Structure

// Explain the detailed code structure that shows the functions and variables with how you implemented and how those are interacting with the other functions or variables.
// Especially in pipelined CPU, explain how you handled NOPs and Flushes (e.g., implementing dependency checking logic and so on)
There are four pipeline register classes which are used to pass information between stages. IF_ID is used to pass information between Fetch and Decode, ID_EX is used to pass between

Decode and Execution, EX_Mem is between Execute and Memory, Mem_WB is between Memory and Writeback.

Dependency checking logic is used to implement stall detection for load use hazards in the Fetch() function. When the conditions are met, stall and NOP are implemented.

3.2.1 Functions

FindOp(), findI(), findOperation(), ControlUnit(), and sign_extend()

The functionality of these functions remain the same as the single cycle cpu program and work identically to them. The find functions are called in the Decode() function with the sign_extend called in the findI() function.

Fetch(instruction)

This function takes in an instruction and initializes the stall variable, which is used to check for stalls. If there is a data hazard (where Ex/Mem is reading from memory and there's a destination register which may be used by teh current If/ID instruction), the stall flag is set to be True, which then inserts NOP and marks the instruction is invalid, stalls the pipeline by returning without updating the pc.

If there's no stall, it finds the instruction index by dividing the pc by 4 and checks if the index is within the instruction list, if it is, it then fetches the instruction and stores it in the IF/ID register. It then sets the IF/ID pc value and affirms that the instruction is valid. It handles branch and jump instructions by setting the pc to the branch target for branch instructions, set pc to the calculated jump target from alu for JAL and JALR. Otherwise, it'll do the default operation of incrementing pc by 4. This function is called in the run_cpu() function.

Decode()

The decode function uses the IF/ID and ID/EX registers alongside the register file variables. It checks if IF/ID has an invalid instruction and sets ID/EX as invalid if it is. It gets the instruction from IF/ID and calls the findOp, findI, and findOperation functions to find the instruction type, fields, and operation. The register values are obtained from fields which are then written to the ID/EX register with the values from the register file. The ID/EX register is also passed the IF/ID pc, immediate, operation, and the validation flag. Control signals are made using the ControlUnit() function which are copied to the ID/EX register. This function is called in the run_cpu function.

Execute()

The function does the EX stage by taking the input from the ID/EX register, conducting ALU operations, writing the results of the ALU operations into EX/MEM, updating alu_zero, branch_target, and pc if necessary. If the control signal for branch target is detected, it calculates the branch target using the pc and immediate from ID/EXE. This function is called in the run_cpu function.

Memory()

The function uses EX/Mem, Mem/WB, and d_mem variables. It checks if EX/Mem instruction is valid which if not, it sends the invalid to the next stage. The main functionality is to handle load and store operations with the data memory, passes the results from the EX/Mem, and set up the Mem/WB pipeline register. It is called in the run_cpu function.

Writeback()

The function uses Mem/WB and the register file variables. If checks the validness of Mem/WB and if it's valid continues to write back the results of Mem/WB to the register file if there's a destination register and the RegWrite control signal, writing the memory result if MemtoReg flag is there or ALU results if the flag for MemtoReg is false.

Run_cpu(filename)

The function uses pc and total_clock_cycles global variables and initializes them to 0. It opens the text file and saves each line of machine code into a list and then initializes the pipeline. The function then runs a loop to execute the instructions until the program ends or the max cycle variable is met. The pipeline functions are called in reverse order and prints the results once the pipeline is filled.

3.2.2 Variables

The pc and next_pc variables are used for the program counter.
Branch_target is used for the branch instructions.
Alu_zero is used for branch instructions.
Total_clock_cycles are used for reporting results and for the run_cpu loop.
Rf and d_mem are lists used for the register file and data memory.

3.3 Execution Results

// Show how to run your program and a sample output screenshot
// If you implemented data forwarding, please add one more sample output with data forwarding that may have fewer cycles.

```
samplepart1.txt
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:
pc is modified to 0x10
total_clock_cycles 5:
pc is modified to 0x14
total_clock_cycles 6:
pc is modified to 0x18
program terminated:
total execution time is 6 cycles
```

3.4 Challenges and Limitations

// If you encountered any challenges while implementing the code, discuss here
// If you think your program has any limitations (e.g., some part is not working properly), explain here with potential reasons.
Challenges include implementing handling data hazards and handling instructions being passed between pipeline stages. The program is not functioning properly, which some parts are functioning such as the pipelining, however the branch instructions may have errors due to no branch prediction and having all branches being predicted to be not taken. Hazard detection is also limited to certain dependencies, with more unaccounted. There is also no data forwarding, which will affect performance and have more stalls.