**ECE 485 – Final Project - Stage 1 5-cycle Multi-cycle Architecture**

**Overview**

For the ECE 485 final project, you will be implementing a limited version of the RISC-V 5-cycle pipeline architecture discussed in Appendix C of your Hennessey and Patterson textbook. You will evolve this design in 4 stages:

**Stage 1**: 5-cycle Multicycle Architecture: convert the single cycle RISC-V architecture to a 5-cycle multicycle architecture (IF-ID-EX-MEM-WB) that is NOT pipelined. Each instruction will take 5 cycles to complete before the next instruction can begin. See Figure C.18

**Stage 2**: 5-cycle Pipelined Architecture: Add intermediate registers (IF\_ID, ID\_EX, EX\_MEM, MEM\_WB) to the Stage 1 architecture, so it is now pipelined. However, this stage does NOT have hazard detection and does NOT stall to prevent hazards. You will need to add NOPs to the code to ensure proper execution. See Figure C.19

**Stage 3**: 5-cycle Pipelined Architecture with Stall mitigation: convert the stage 2 architecture to a pipelined architecture that has hazard detection and stalls appropriately to ensure proper execution, but does NOT have forwarding.

**Stage 4**: 5-cycle Pipelined Architecture with Forwarding: convert the stage 3 architecture to a pipelined architecture that has hazard detection and stalls appropriately, plus has forwarding and branch decision moved to the ID stage, similar to Figure C.25.

The only RISC-V instructions implemented will be ADD, ADDI, SUBI, LW, SW, NOP, J, and BNE. You will also implement a custom instruction called *Load\_Addr*, which is similar to LA, but only loads the address 0x10000000. Details about these instructions are in a later section.

The memory map for this architecture assumes the program memory starts at address 0x00000000, and the data memory (for the Array), starts at address 0x10000000

**Program to Execute**

The primary program to be executed for these 4 stages is from the RISC-V exercise:

addi    x5, x0, 9       # x5 = number of values in array

        load\_addr x6, array     # x6 = address of array 0X10000000

        lw      x7, 0(x6)       # x7 = initial value

loop:   addi    x6, x6, 4

        lw      x10, 0(x6)

        add     x7, x10, x7

        subi    x5, x5, 1

        bne     x5, x0, loop

done:   j       done            # infinite loop (breakpoint)

# initialize data in the array [assumed to start at 0x10000000]

array: .word 0x5, 0x4, 0x10, 0x3, 0x12, 0x1, 0x7, 0x4, 0x8, 0x2

**Block Diagram**

Stage 1 design is based on Figure C.18 from your textbook. A draft of the design is given to you in RISC-V stage 1.pptx:

A diagram of a multi cycle diagram

AI-generated content may be incorrect.

**GitHub**

You are also given starter-code at the GitHub repo <https://github.com/GeorgeYork/ece485_final_cadet_stage1>

which includes:

|  |  |
| --- | --- |
| riscv\_multicycle.vhdl: | the top level |
| pc\_live.vhdl | updates new program counter [do not change] |
| instr\_mem.vhdl | instruction memory. Holds assembly program to execute |
| reg\_file.vhdl | The 32 32-bit Registers [do not change] |
| control\_unit.vhdl | Given the instruction OPCODE, outputs the control signals *reg\_write, mem\_read, mem\_write, alu\_src, branch, load\_addr, jump* [do not change] |
| immediate\_generator.vhdl | Given the instruction, pulls out and sign extends (to 32-bits) the immediate value for each type of instruction. [do not change] |
| alu\_control.vhdl | Given funct7 and funct3 for an instruction, outputs the alu\_op (alu operation) to perform |
| alu.vhdl | Performs the arithmetic logic unit operation (ADD, SUB, etc) on two operands. |
| data\_mem.vhdl | The data memory. Holds the array of values to be summed, starting at address 0x10000000 [do not change] |
| tb\_riscv\_multicycle.vhdl | Test bench to test overall architecture [do not need to change] |
| tb\_riscv\_multicycle\_behav.wcfg | Waveform configuration file, set up to have the required signals in the simulation plot for the main program test |
| RISC-V stage 1.pptx | Draft block diagram of stage 1 |

Fork and Clone and open the project ece485\_final\_cadet\_stage1.

**Details of Instructions**

|  |  |
| --- | --- |
| ADD | R-Type Instruction  *add rd, rs1, rs2 # rd = rs1 + rs2*  rd: destination register (where the result is stored)  rs1: first source register  rs2: second source register  **Instruction Format**:  “<7-bit funct7> <5-bit rs2> <5-bit rs1> <3-bit funct3> <5-bit rd> <7-bit opcode>”  funct7= “0000000”  funct3= “000”  Opcode= “0110011” |
| ADDI | I-Type Instruction  *addi rd, rs1, imm # rd = rs1 + imm*  rd: destination register  rs1: source register  imm: immediate, sign extended to 32-bit  **Instruction Format**:  “<12-bit imm> <5-bit rs1> <3-bit funct3> <5-bit rd> <7-bit opcode>”  funct3= “000”  Opcode= “0010011” |
| LW | I-Type Instruction: Load Word from Memory  *lw rd, imm(rs1) # rd = memory[rs1 + imm]*  rd: destination register  rs1: register containing base address (or pointer)  imm: immediate, sign extended to 32-bit; offset to be added to pointer  **Instruction Format**:  “<12-bit imm> <5-bit rs1> <3-bit funct3> <5-bit rd> <7-bit opcode>”  funct3= “010”  Opcode= “0000011” |
| SW | Not implementing this year |
| BNE | B-Type Instruction:  *bne rs1, rs2, imm # PC = PC + 4 + imm, if rs1 is not equal to rs2*  imm: immediate, sign extended to 32-bit; offset to be added to PC + 4  **Instruction Format**:  “<imm[11]> <imm[9:4]> <5-bit rs2> <5-bit rs1> <3-bit funct3> <imm[3:1]>  <unused bit><imm[10]><7-bit opcode>”  funct3= “001”  Opcode= “1100011”.  So the final PC = PC + 4 + (sign extend)(imm[11:1])  Note: for our implementation, imm[11:1] points to a byte [not word or 16-bit]  Unused bit is due to shifting right, which drops least significant bit. |

Custom Instructions [replacing Psuedo-instructions]:

|  |  |
| --- | --- |
| J | In the RISC-V architecture, J (jump to label) is a pseudo-instruction, where  *j label*  is replaced with  *jal rd, imm # jump to PC = PC + 4 + (sign extend) imm*  You will add a custom J instruction to your implementation.  J-Type Instruction:  *j imm # PC = PC + 4 + imm*  imm: immediate, sign extended to 32-bit; offset to be added to PC + 4  **Instruction Format**:  “<imm[20]> <imm[10:1]> <imm[11]> <imm[19:12]><5-bit rd> <7-bit opcode>”  rd = not used “00000”  Opcode= “1101111”.  So the final PC = PC + 4 + (sign extend)(imm[20:1])  Note: for our implementation, imm[20:1] points to a byte [not word or 16-bit] |
| SUBI | In the RISC-V architecture, SUBI (subtract immediate) is a pseudo-instruction, where  *subi rd, rs1, imm # rd = rs1 - imm*  is replaced with  addi rd rs1, imm #. rd = rs1 + -imm  You will add a custom SUBI instruction to your implementation.  **Instruction Format**: “<20-bit don’t care> <5-bit rd> <7-bit opcode>”  rd = destination register  Opcode= “0010111” |
| Load\_Addr | In the RISC-V architecture, there is a LA pseudo-instruction, which loads an address in a register to be used as a pointer to a data array. Since we cannot have a load with a 32-bit immediate value, this pseudo-instruction is replace with two instructions to create the 32-bit value:  *auipc x5, offset[31:12]*  *addi  x5, x5, offset[11:0]*  Instead of doing this, we will create a custom instruction called load\_addr, which always loads the fixed value of our array 0x10000000.  **Instruction Format**: “<20-bit don’t care> <5-bit rd> <7-bit opcode>”  rd = destination register  Opcode= “0010111” |
| NOP: | In the RISC-V architecture, NOP is actually a pseudo-instruction, which is replaced by an instruction that does nothing, *addi x0, x0, 0*. In stage 2, you will implement a NOP instruction, but instead of using the ADDI, you will use the **32-bit instruction “0x00000000**”, and when detected, your system will set all the control signals to zero (reg\_write <= '0'; alu\_src <= '0'; mem\_read <= '0'; mem\_write <= '0'; branch <= '0'; jump <= '0'; load\_addr <= '0'; see control\_unit.vhdl |

**Tasks**

By Taps Lesson 21, complete these tasks for stage 1.

1. Read through the given modules to understand the code, fix/complete the code, and update/complete the block diagram RISC-V stage 1.pptx to match your design, labeling the correct signal names and connecting or labeling the red control signals. Turn the completed diagram in via gradescope and copy to your repo (share your repo with your instructor).
   1. Add support for the custom SUBI instruction. Should only require a modification to alu\_control.vhdl and alu.vhdl
   2. Remove syntax errors in instru\_mem.vhdl by commenting out array entries 1 through 8; so only entry 0 “addi x5, x0, 9” is included for initial testing
   3. Complete riscv\_multicycle.vhdl
2. **Initial Test**: Run simulation in vivado, testing the single instruction “addi x5, x0, 9”. The testbench tb\_riscv\_multicycle.vhdl is already created for you and should not need any modifications. Also, the simulation waveform configuration file, tb\_riscv\_multicycle\_behav.wcfg is also given, which should have all the necessary signals, which as a minimum should include:

*Clk, reset, clock\_counter, state, pc, NPC, next\_PC, instr, opcode, rs1, reg1\_data, rs2, reg2\_data, rd, wb\_rd, imm, alu\_input\_a, alu\_input\_b, alu\_result, alu\_src, reg\_write, reg\_write\_chip, display\_x5*

**Turn in the simulation plot in gradescope**, showing the x5 register is updated with the sum of 0 and 9.

1. **SUBI Test**: Modify instru\_mem.vhdl, keeping array entry 0 “*addi x5, x0, 9*”, and adding entry 1 “*subi x5, x5, 4*” and entry 2 “*subi x5, x5, -4*”. Run simulation in vivado, testing the SUBI instructions work, including necessary signals, which as a minimum should include:

*Clk, reset, clock\_counter, state, pc, NPC, next\_PC, instr, opcode, rs1, reg1\_data, rs2, reg2\_data, rd, wb\_rd, imm, alu\_input\_a, alu\_input\_b, alu\_result, alu\_src, reg\_write, reg\_write\_chip, display\_x5*

**Turn in your instru\_mem.vhdl code and the simulation plot in gradescope**.

1. **Main Program**: Modify instru\_mem.vhdl, to implement the 9 lines of the main program. This will require creating the 32-bit assembly code word for each instruction. [**Note**: the data array is already stored in data\_mem.vhdl.] [**Another Note**: unlike the single-cycle RISC-V architecture, the multi-cycle RISC-V architecture automatically computes NPC=PC+4 during IF stage, so the branch and jump relative address is relative to PC+4, not PC.]

**Turn in your instru\_mem.vhdl code and 9 assembly code entries in gradescope**

1. **Test Main Program**: Run simulation in vivado, testing the main program works, including necessary signals, which as a minimum should include:

*Clk, reset, clock\_counter, state, pc, NPC, next\_PC, instr, opcode, rs1, reg1\_data, rs2, reg2\_data, rd, wb\_rd, imm, alu\_input\_a, alu\_input\_b, alu\_result, mem\_data, mem\_read, mem\_write, alu\_src, branch, jump, load\_addr, reg\_write, reg\_write\_chip, display\_x5, display\_x6, display\_x7, display\_x10*

**Turn in your simulation plot in gradescope**.

1. Answer the following questions in gradescope:
   1. How many clock cycles to complete (end of first call to jump’s WB stage)?
   2. What was the final value of register 5?
   3. What was the final value of register 7?
   4. How do the above answers compare to your expectations from the RISC-V exercise?
2. Push all code to your github repo and share with your instructor

**Grading – Stage 1**

| **Item** | **Points** | **Out of** |
| --- | --- | --- |
| Updated block diagram |  | 10 |
| Updated alu\_control.vhdl and alu.vhdl |  | 10 |
| Updated riscv\_multicycle.vhdl |  | 15 |
| Initial Test: addi x5, x0, 9 |  | 10 |
| SUBI test |  | 10 |
| Assembled Main Program (9 lines) |  | 15 |
| Test Main Program |  | 20 |
| Questions in Gradescope |  | 10 |
| Late Penalties (-25 points per day late) |  |  |
| **Total** |  | **100** |

**GenAI and Collaboration Policy**

For this assignment, use of generative AI (e.g., ChatGPT) or search engines is “level 2: Use of GenAI for brainstorming or idea generation.” For example, you can prompt GenAI on how to implement basic hardware in VHDL, such as “how to you implement a 32-bit register in VHDL?” “How to you implement a multiplexer using the *when* statement?”, or ask basic questions about the RISC-V Stage 1 multicycle architecture, like “why is the ALU controller separate from the Control Unit in RISC-V?”; however, any coding must be your own work. Do not ask GenAI to produce specific code for this assignment, like “Given this VHDL code for the ALU, add the SUBI function.”

Any use of GenAI or internet searches must be clearly documented, including all prompts used. Failure to follow this policy will result in **NO** academic credit. One warning about GenAI and RISC-V: besides the common problem of GenAI hallucinations, there are different versions of RISC-V, different from this assignment, so GenAI could lead you astray.

The only collaboration allowed with classmates is for general questions, such has how to implement VHDL code, like “how to make a process statement to implement a register”, not for specific coding in this assignment, or for general concepts regarding the blocks in the block diagram. All coding must be your own work. Do not look at each other’s code. You are not allowed to have another student look at your code to help with debugging, unless given specific authorization from your instructor. All collaboration must be clearly documented; copying another person’s work, with or without documentation, will result in **NO** academic credit.