**ECE 485 – Final Project - Stage 4 5-cycle Pipelined Architecture – Stall Detection**

**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. [except the BNE decision is in ID not EX]

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**

A draft of the design is given to you in RISC-V Stage 4.pptx:

A diagram of a 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_stage4>

which includes:

|  |  |
| --- | --- |
| riscv\_pipeline.vhdl: | the top level |
| pc\_live.vhdl | updates new program counter [do not change] |
| pipeline\_registers.vhdl | Temporary registers between each state (labeled with prefix *IF\_ID, ID\_EX, EX\_MEM, MEM\_WB*) to move an instruction from state to state. |
| hazard\_detection.vhdl | Monitors the data dependencies and branch decision delays to stall the processor |
| forwarding\_unit.vhdl | The forwarding unit has the MUX control signals to forward the inputs to the ALU results from previous instructions as needed. Note: we are not forwarding from data memory loads for this simplified RISC-V architecture. |
| 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\_pipeline.vhdl | Test bench to test overall architecture [do not need to change] |
| tb\_riscv\_pipeline\_behav.wcfg | Waveform configuration file, set up to have the required signals in the simulation plot for the main program test |
| RISC-V Stage 4.pptx | Draft block diagram of Stage 4 |

Fork and Clone and open the project ece485\_final\_cadet\_stage4.

**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 40, complete these tasks for Stage 4. For Stage 4, you will modify your 5-stage pipeline to include forwarding (from ALU results, not Data Memory loads) and moves branch decisions to the ID state.

1. Read through the given modules to understand the code, fix/complete the code, and update/complete the block diagram RISC-V Stage 4.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. You will initially use your instr\_mem.vhdl from stage 1.
   2. Update pipeline\_registers.vhdl to move key registers/signals from state to state [this may be the same as stage 3].
   3. Update forwarding\_unit.vhdl to control the MUX input A to the ALU
   4. Update hazard\_detection\_unit.vhdl to create the *start\_stall* signal when a data or control hazard is detected. Note an additional output called double\_stall has been added. You might find this is needed to handle the double stall that occurs as our BNE instruction needs the x5 value from the SUBI instruction (1st stall), and needs to also execute the branch stall (2nd stall)
   5. Update riscv\_pipeline.vhdl, removing syntax errors and ensuring correct operation of Stage 4. [many of these updates can be copied from riscv\_pipeline.vhdl from stage 3]
2. **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, pc, NPC, next\_PC, if\_id\_npc, id\_ex\_npc, ex\_mem\_npc, instr, opcode, if\_id\_rs1, reg1\_data, if\_id\_rs2, reg2\_data, mem\_wb\_rd, wb\_data, if\_id\_imm, alu\_input\_a, alu\_input\_b, alu\_result, mem\_data, mem\_read, mem\_write, alu\_src, id\_ex\_alu\_src, branch, ex\_mem\_branch, jump, ex\_mem\_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. Re-Order Instructions: modify instru\_mem.vhdl, reordering the instructions to remove any remaining stalls due to data dependencies (as you did in the RISC-V Exercise)
3. **Test Re-Ordered Program**: Run simulation in vivado, testing the re-ordered program works, including necessary signals, which as a minimum should include:

*Clk, reset, clock\_counter, pc, NPC, next\_PC, if\_id\_npc, id\_ex\_npc, ex\_mem\_npc, instr, opcode, if\_id\_rs1, reg1\_data, if\_id\_rs2, reg2\_data, mem\_wb\_rd, wb\_data, if\_id\_imm, alu\_input\_a, alu\_input\_b, alu\_result, mem\_data, mem\_read, mem\_write, alu\_src, id\_ex\_alu\_src, branch, ex\_mem\_branch, jump, ex\_mem\_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 about the re-ordered version:
   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 [instr\_mem.vhdl should be the version with the NOPs inserted]

**Policy if Stage 3 not working**

If your stage 3 code is not working, you can either attempt to get it working before implementing Stage 4, or for a cost of 10 points of your Stage 4 grade, your instructor can give you a working version of stage 2 to start from.

**Grading – Stage 4**

| **Item** | **Points** | **Out of** |
| --- | --- | --- |
| Updated block diagram |  | 10 |
| Updated forwarding\_unit.vhdl |  | 15 |
| Updated riscv\_pipeline.vhdl |  | 15 |
| Update hazard\_detection\_unit.vhdl |  | 10 |
| Test Main Program |  | 15 |
| Questions in Gradescope on Main Program |  | 10 |
| Re-ordered instr \_mem.vhdl |  | 5 |
| Test Re-Ordered Program |  | 10 |
| Questions in Gradescope on Re-Ordered Program |  | 10 |
| Late Penalties (-25 points per day late)  Instructor Provide Stage 1 code (-10 points) |  |  |
| **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 4 pipeline 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.