# CprE 381: Computer Organization and Assembly-Level Programming

## Project Part 2 Report

Team Members: ___Luke Auderer_____

___Yin Choong_____

_____

Project Teams Group #:___ Project Group A_02_____

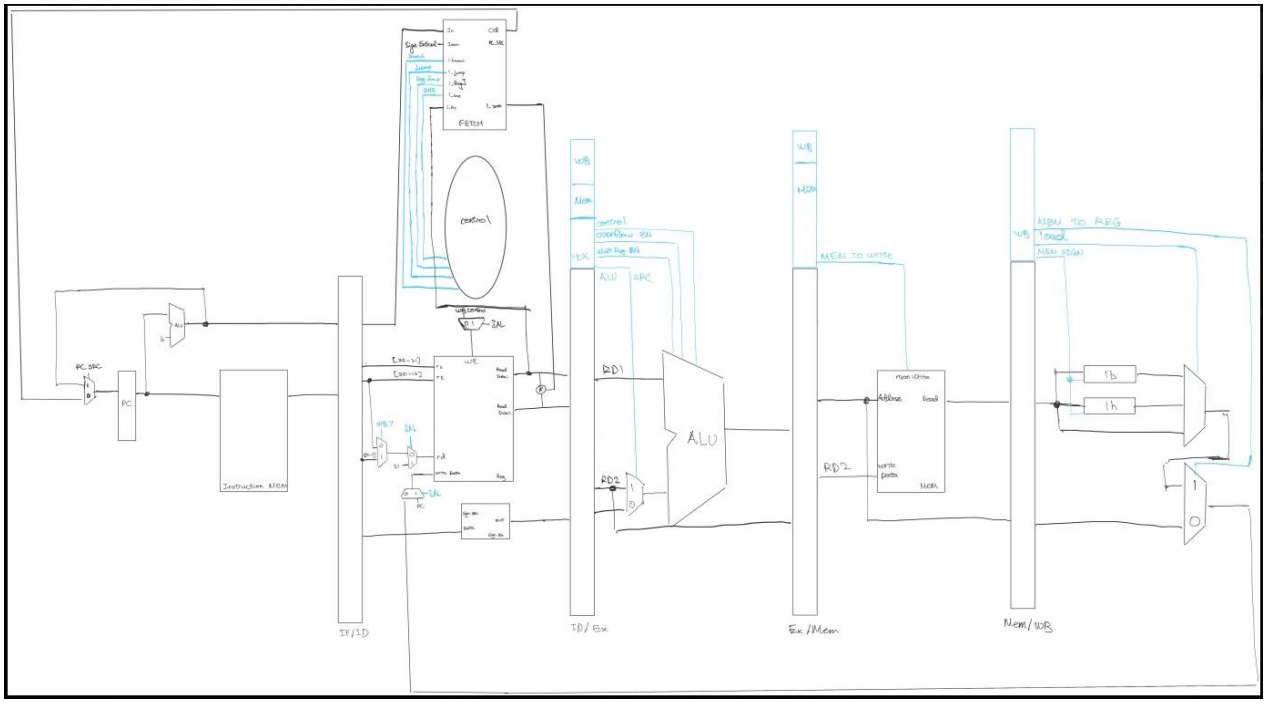
***Refer to the highlighted language in the project 1 instruction for the context of the following questions***.

[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

| Stage | Datapath Values | Control Signals |
|---|---|---|
| IF | i_pc Current Program Counter (PC)<br>i_instruction 26-bit instruction field for jump address<br>i_imm Immediate value for branch offset<br>i_rs Register value used for register jump (e.g., jr)<br>s_output Final computed PC value (goes to o_pc)<br>s_append Jump address after processing<br>s_adderbranch PC + branch offset<br>s_branchmux, s_branchmuxbne, s_branchMaster Intermediate mux outputs<br>s_xor Used to check if PC changes (helps generate pcSrc) | i_branch Indicates a conditional branch (e.g., beq)<br>i_zero Result of ALU zero flag (used for beq/bne decision)<br>i_jump Indicates an unconditional jump<br>i_RegJump Indicates a register-based jump (e.g., jr)<br>i_BNE Used to select between BEQ (i_zero) and BNE (!i_zero) behavior<br>s_and, s_andbne, s_inv Intermediate logic control signals<br>o_pcSrc Final control output to indicate PC has changed (for hazard handling etc.) |
| ID | Read Register 1 (Rs)<br>Read Register 2 (Rt)<br>Sign-extended Immediate<br>Instruction[15-11] (Rd) – used later for destination register selection<br>Instruction[20-16] (Rt) – also used for destination register selection<br>Instruction[25-21] (Rs) – register source 1<br>Instruction[15-0] – immediate value to be sign-extended<br>Jump Address (if applicable) – from instruction bits [25-0] | RegDst – decides between Rt and Rd as the destination register<br>ALUSrc – selects between register and immediate for ALU second operand<br>MemtoReg – decides if memory or ALU result is written to register file<br>RegWrite – enables writing to the register file<br>MemRead – enables reading from memory<br>MemWrite – enables writing to memory<br>Branch – used to determine branch decision<br>Jump – used to determine jump control<br>ALUOp – used by ALU control to generate exact ALU operation |

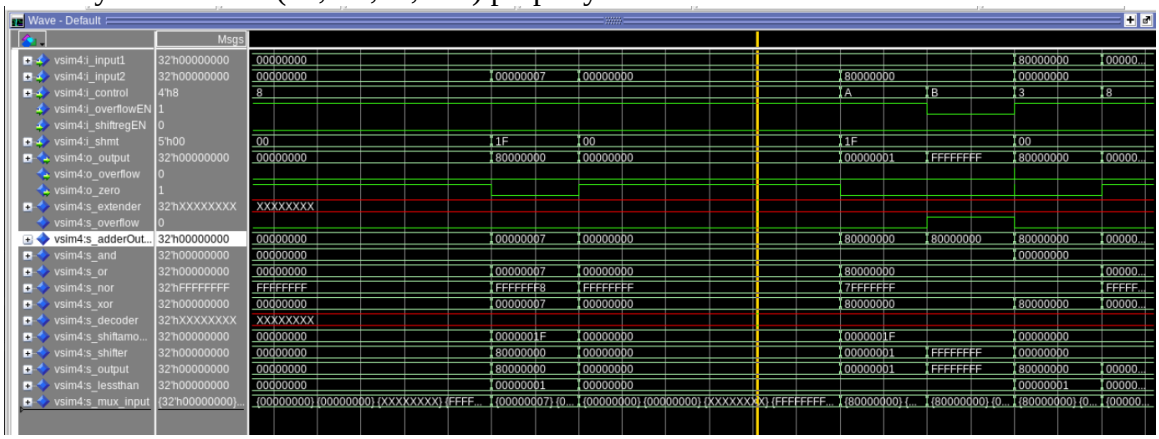
| | | |
|---|---|---|
| EX | i_input1 First operand for the ALU (from register file or shifter)<br>i_input2 Second operand for the ALU (from register file or immediate)<br>i_shmt Shift amount for shift operations<br>s_adderOutput Output from adder/subtractor (used for add, sub, SLT, etc.)<br>s_and Result of bitwise AND between i_input1 and i_input2<br>s_or Result of bitwise OR<br>s_xor Result of bitwise XOR<br>s_nor Result of bitwise NOR<br>s_shifter Output of shifter component<br>s_lessthan Result of SLT comparison (1 or 0)<br>s_output Final ALU result (based on control mux selection)<br>o_zero Zero flag output<br>o_overflow Overflow signal output | i_control(3 downto 0) Main ALU control signal to choose the operation (from ALU control)<br>i_overflowEN Enables overflow detection logic<br>i_shiftregEN Selects between using shift amount (i_shmt) or value in i_input1 |
| MEM | addr: The memory address that is generated from the EX stage, which is used to read or write to memory.<br><br>data: The data to be written into memory, coming from the EX stage or the register file. | we (Write Enable): This signal controls whether data is written to memory. It is active when the instruction is a store operation (e.g., sw).<br><br>q (Output from Memory): This is the data read from memory, which is passed to the WB stage for further processing (used in load instructions). |
| WB | i_d (Input Data): This is the 32-bit input data (from the MEM stage or other parts of the processor) that will be written into the registers.<br><br>o_D1 and o_D2: These are the output signals representing the data read from the registers rs and rt, respectively, after the write-back operation | i_we (Write Enable): This signal controls whether the data will be written into the registers. It is used to enable writing to the registers in the WB stage.<br><br>i_rst (Reset): This signal resets the registers, ensuring that their contents are cleared when necessary (usually at reset).<br><br>i_clock: The clock signal for synchronization during the write-back process |

[1.b.ii] high-level schematic drawing of the interconnection between components.



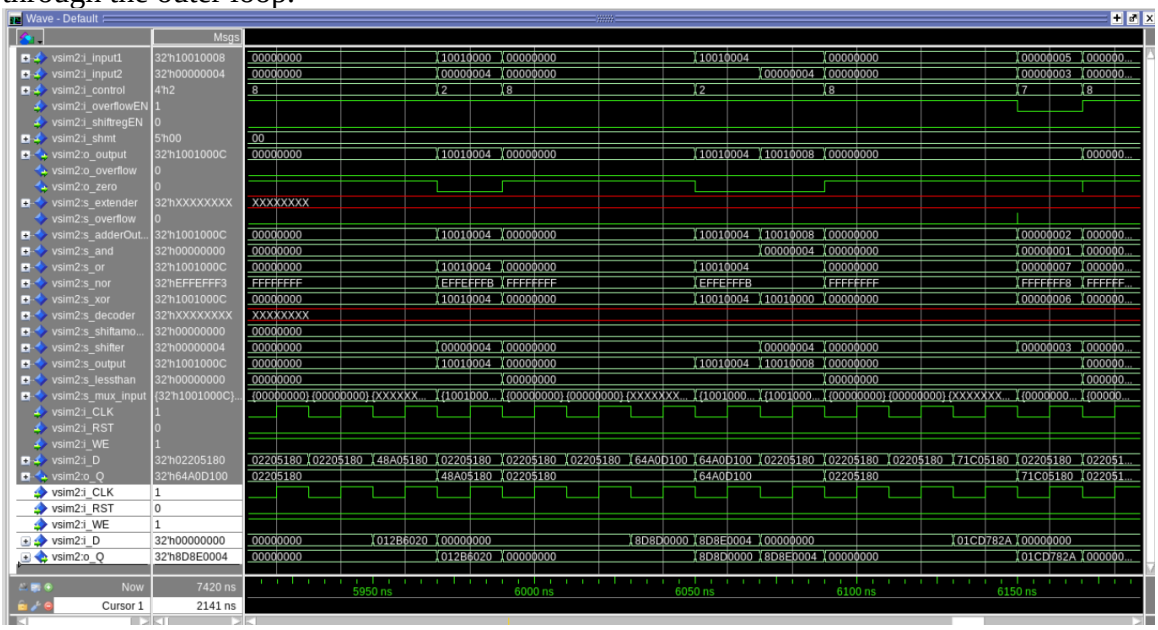

[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.

This is the simple test running. It is correct because we can clearly trace the execution of our program and notice the values changing correctly. Everything appears to be working, and we are confident in the result. Registers like $t0, $t1, and $t2 hold expected values from addi, add, and addiu, and the results of logical operations (and, or, xor, etc.) match bitwise expectations. Branching behavior is correct because beq is not taken and bne is taken. The jump to the function via jal and return with jr behaves as expected, and memory instructions (lw, sw, lb, etc.) properly access and store the correct values.




[1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did not have to use the maximum number of NOPs.

This is two iterations of bubble sort:
It is correct because we can clearly trace the execution of our program and notice the values changing correctly. Everything appears to be working, and we are confident in the result. The waveform for Proj2_bubblesort.s confirms that the bubble sort algorithm executes correctly, with the initial unsorted array [7, 3, 9, 1, 5] being sorted step-by-step into [1, 3, 5, 7, 9]. I can see each pairwise comparison and conditional swap being performed across nested loop iterations, driven by the values of $t2, $t5, and $t6, and the correct use of slt and beq to control flow. Memory addresses accessed by lw and updated by sw reflect proper element shifting, and the loop counters decrement as expected through the outer loop.




1. **The first data flow example where I did not have to use the maximum number of nops:**
   Originally the code was this:
   nop
   nop
   nop
   lasw $t0, size
   nop
   nop
   nop
   lw $t0,0($t0)
   nop
   nop
   nop

   lasw $t1, array  # Load base address of array
   addi $t0, $t0, -1  # N-1 iterations

but I changed it to this to avoid using so many nops. This works because I was able to insert an independent instruction between the two instructions which had a data hazard, and this instruction took the place of a nop. Because it is an independent instruction (doesn't have a RAW dependency with the above instruction), there is no issue.

```
nop
nop
nop
lasw $t0, size
nop
lasw $t1, array         # Load base address of array
nop
lw $t0,0($t0)
nop
nop
nop
addi $t0, $t0, -1     # N-1 iterations
```

This saved a nop.

2. **The second data flow example where I did not have to use the max number of nops:**

Originally the code was this:

```
sll $t3, $t2, 2       # $t3 = i * 4
nop
nop
nop
add $t4, $t1, $t3     # $t4 = base + offset
```

But I changed it to this. This works because I was able to insert an independent instruction between the two instructions which had a data hazard, and this instruction took the place of a nop. Because it is an independent instruction (doesn't have a RAW dependency with the above instruction), there is no issue.

```
sll $t3, $t2, 2
nop
lw $t5, 0($t4)
nop
add $t4, $t1, $t3
```

This saved a nop.

3. **A control flow example where I did not have to use the max number of nops is this:**

Originally the code was this:

```
beq $t7, $zero, skip_swap # Skip if already sorted
```

nop
nop

The two nops were needed because this could potentially be a control hazard. This is because the processor doesn't know whether the branch is taken until the EX (execute) stage. Meanwhile, the instructions after the beq have already started moving through the pipeline. This can cause the CPU to fetch the wrong instructions if the branch is taken or not taken, unless it waits or predicts the branch, so that's why I added the two nops.

However I was able to do this:

```
beq $t7, $zero, skip_swap
addi $t2, $t2, 1      # Move i++ here
nop
```

because this addi is an instruction that is always executed regardless of the control flow, so it's not an issue having it here. The good thing about this is it saves me a nop.

[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

The maximum frequency it can run at is 51.32 MHz

The critical path goes through the main register file because we can see that component is taking the longest in the output file that the tool flow provides us. Because this component is taking the longest, we know this is the component that our critical path goes through. The critical path as a whole goes though main register, to our xor comparator unit, and then to the fetch module, and then to the PC_SRC select mux, and then to PC module.

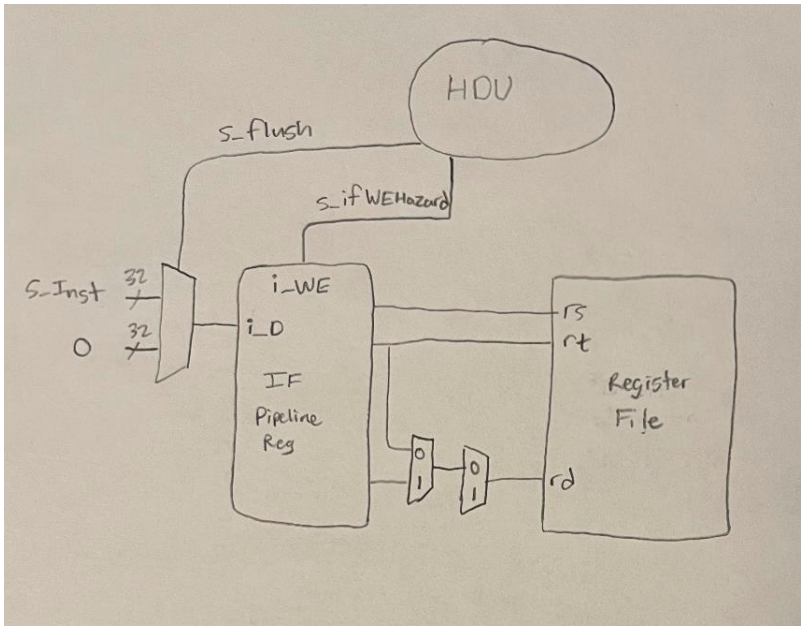
```
#
# CprE 381 toolflow Timing dump
#

FMax: 51.32mhz Clk Constraint: 20.00ns Slack: 0.52ns

The path is given below

 ====================================================================
 From Node    : mem:IMem|altsyncram:ram_rtl_0|altsyncram_g8d1:auto_generated|ram_block1a0~portb_address_r
 To Node      : PC_Module:PC|N_Reg:g_pc|neg_dffg:\NBit_DFF:1:dffi|s_Q
 Launch Clock : iCLK (INVERTED)
 Latch Clock  : iCLK (INVERTED)
 Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ==================================
     10.000     10.000           launch edge time
     13.458      3.458  F        clock network delay
     13.721      0.263     uTco  mem:IMem|altsyncram:ram_rtl_0|altsyncram_g8d1:auto_generated|
ram_block1a0~portb_address_reg0
     16.611      2.890 FR  CELL  IMem|ram_rtl_0|auto_generated|ram_block1a0|portbdataout[4]
     17.250      0.639 RR    IC  IMem|ram~54|datad
     17.405      0.155 RR  CELL  IMem|ram~54|combout
     18.778      1.373 RR    IC  MainRegister|g_mux1|Mux20~0|dataa
     19.215      0.437 RF  CELL  MainRegister|g_mux1|Mux20~0|combout
     19.945      0.730 FF    IC  MainRegister|g_mux1|Mux20~1|datad
     20.095      0.150 FR  CELL  MainRegister|g_mux1|Mux20~1|combout
     21.123      1.028 RR    IC  MainRegister|g_mux1|Mux20~2|datad
     21.278      0.155 RR  CELL  MainRegister|g_mux1|Mux20~2|combout
     21.515      0.237 RR    IC  MainRegister|g_mux1|Mux20~3|dataa
     21.912      0.397 RR  CELL  MainRegister|g_mux1|Mux20~3|combout
     22.947      1.035 RR    IC  MainRegister|g_mux1|Mux20~19|datab
     23.349      0.402 RR  CELL  MainRegister|g_mux1|Mux20~19|combout
     23.607      0.258 RR    IC  g_zeroflag|Equal0~6|datab
     24.039      0.432 RF  CELL  g_zeroflag|Equal0~6|combout
     25.019      0.980 FF    IC  g_zeroflag|Equal0~9|dataa
     25.372      0.353 FF  CELL  g_zeroflag|Equal0~9|combout
     25.651      0.279 FF    IC  g_zeroflag|Equal0~20|dataa
     26.004      0.353 FF  CELL  g_zeroflag|Equal0~20|combout
     26.254      0.250 FF    IC  Fetch|g_RegJump|\G_NBit_MUX:4:MUXI|o_O~2|datac
     26.535      0.281 FF  CELL  Fetch|g_RegJump|\G_NBit_MUX:4:MUXI|o_O~2|combout
     27.496      0.961 FF    IC  Fetch|g_RegJump|\G_NBit_MUX:12:MUXI|o_O~0|datac
     27.777      0.281 FF  CELL  Fetch|g_RegJump|\G_NBit_MUX:12:MUXI|o_O~0|combout
     28.014      0.237 FF    IC  Fetch|g_RegJump|\G_NBit_MUX:12:MUXI|o_O~1|datac
     28.295      0.281 FF  CELL  Fetch|g_RegJump|\G_NBit_MUX:12:MUXI|o_O~1|combout
```

[2.a.ii] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.




[2.a.iii] Create a testbench that instantiates all four of the registers in a single design. Show that values that are stored in the initial IF/ID register are available as expected four cycles later, and that new values can be inserted into the pipeline every single cycle. Most importantly, this testbench should also test that each pipeline register can be individually stalled or flushed.
Done, verified in TA office hours.

[2.b.i] list which instructions produce values, and what signals (i.e., bus names) in the pipeline these correspond to.

R-Type Instructions (Opcode = 000000)
These produce values (write to a register):
add – writes result to rd
addu – writes result to rd
sub – writes result to rd
subu – writes result to rd
and – writes result to rd
or – writes result to rd
xor – writes result to rd
nor – writes result to rd
slt – writes result to rd
sll – writes result to rd
sllv – writes result to rd
srl – writes result to rd
srlv – writes result to rd
sra – writes result to rd
srav – writes result to rd

These R type instructions which produce values, correspond to these signals in the pipeline

1. **IF Stage**:
    - s_Inst
    - s_IFInst

2. **ID Stage**:
    - s_IFInst
    - s_rs
    - s_rt
    - s_extended (unused for add)
3. **EX Stage**:
    - s_forwardA
    - s_forwardB
    - s_IDControl(25 downto 22)
    - s_ALUDATA
    - s_EXalu
4. **MEM Stage**:
    - s_EXalu
    - s_WBalu
5. **WB Stage**:
    - s_WBalu
    - s_RegWrAddr
    - s_RegWrData

I-Type Instructions
These produce values (write to a register or memory):
addi – writes result to rt
addiu – writes result to rt
andi – writes result to rt
ori – writes result to rt
xori – writes result to rt
lui – writes result to rt
slti – writes result to rt
lw – loads from memory into rt (register gets value)
lb, lh, lbu, lhu – all load to rt (register gets value)
sw – stores register value to memory (produces value on memory write path)

These I type instructions which produce values, correspond to these signals in the pipeline

**IF Stage:**
- s_Inst
- s_IFInst

**ID Stage:**
- s_IFInst
- s_rs
- s_extended

**EX Stage:**
- s_forwardA
- s_forwardB
- s_IDControl(25 downto 22)
- s_ALUDATA
- s_EXalu

**MEM Stage:**
- s_EXalu
- s_WBalu

**WB Stage:**

- s_WBalu
- s_RegWrAddr
- s_RegWrData

J-Type Instructions
jal – writes return address to $ra ($31), so it produces a value.

The jal instruction corresponds to these signals in the pipeline

**IF Stage:**

- s_Inst
- s_IFInst
- s_IFPC

**ID Stage:**

- s_IFInst
- s_JumpLink
- s_RegJump

**EX Stage:**

- s_JBSrc
- s_EXPC

**MEM Stage:**

- s_EXPC
- s_WBPC

**WB Stage:**

- s_WBPC
- s_RegWrAddr
- s_RegWrData

[2.b.ii] List which of these same instructions consume values, and what signals in the pipeline these correspond to.

R-Type (Opcode = 000000)

| Instruction | Consumes | Notes |
|---|---|---|
| add | $rs, $rt | Adds two registers |
| addu | $rs, $rt | Unsigned add |
| sub | $rs, $rt | Subtracts two registers |
| subu | $rs, $rt | Unsigned subtract |
| and | $rs, $rt | Bitwise AND |
| or | $rs, $rt | Bitwise OR |
| xor | $rs, $rt | Bitwise XOR |
| nor | $rs, $rt | Bitwise NOR |
| slt | $rs, $rt | Set on less than |
| sll | $rt, shamt | Shift uses rt, not rs |
| sllv | $rt, $rs | Shift amount comes from rs |
| srl | $rt, shamt | Logical shift right |
| srlv | $rt, $rs | Variable shift right |

sra $rt, shamt Arithmetic shift
srav $rt, $rs Arithmetic shift variable

For the R type instructions above, these are the signals they correspond to:

**IF Stage:**

- s_Inst
- s_IFInst

**ID Stage:**

- s_IFInst
- s_rs
- s_rt
- s_forwardA2
- s_forwardB2

**EX Stage:**

- s_forwardA
- s_forwardB
- s_IDControl(25 downto 22)
- s_ALUDATA
- s_EXalu

**MEM Stage:**

- s_EXalu
- s_WBalu

**WB Stage:**

- s_WBalu
- s_RegWrAddr
- s_RegWrData

Note: jr is a special case, it still consumes $rs.
jr $rs Jump to register address

For the jr instruction:
IF Stage: s_Inst, s_IFInst
ID Stage: s_IFInst, s_rs, s_forwardA2
EX Stage: s_forwardA, s_IDControl(25 downto 22)
Control Signals: s_RegJump, s_pcSrc

I-Type

| Instruction | Consumes | Notes |
|---|---|---|
| addi | $rs, imm | Uses one register and an immediate |
| addiu | $rs, imm | Same as above, unsigned |
| andi | $rs, imm | Bitwise AND with immediate |
| ori | $rs, imm | Bitwise OR |
| xori | $rs, imm | Bitwise XOR |
| lui | — | Only consumes the immediate |
| slti | $rs, imm | Compare register with imm |

| | | |
|---|---|---|
| lw | $rs, imm | Address = $rs + offset |
| lb | $rs, imm | Load byte |
| lh | $rs, imm | Load halfword |
| lbu | $rs, imm | Load byte unsigned |
| lhu | $rs, imm | Load halfword unsigned |
| sw | $rs, $rt, imm | Address = $rs + offset, data from $rt |
| beq | $rs, $rt | Compares two registers |
| bne | $rs, $rt | Same |

For the I type instruction above these are the signals they correspond to:

**IF Stage:**

- s_Inst
- s_IFInst

**ID Stage:**

- s_IFInst
- s_rs
- s_extended
- s_forwardA2

**EX Stage:**

- s_forwardA
- s_forwardB
- s_IDControl(25 downto 22)
- s_ALUDATA
- s_EXalu

**MEM Stage:**

- s_EXalu
- s_WBalu

**WB Stage:**

- s_WBalu
- s_RegWrAddr
- s_RegWrData

Special Case: sw. Consumes two values:

$rs → for the memory address

$rt → for the data to write

For the sw these are the signals:

**IF Stage:**

- s_Inst
- s_IFInst

**ID Stage:**

- s_IFInst
- s_rs
- s_rt

- s_extended
- s_forwardA2

**EX Stage:**

- s_forwardA
- s_forwardB
- s_IDControl(25 downto 22)
- s_ALUDATA
- s_EXalu
- s_EXrt

**MEM Stage:**

- s_EXalu (memory address)
- s_EXrt (value to be stored)
- s_DMemAddr
- s_DMemData

**WB Stage:**

- Not Applicable

[2.b.iii] generalized list of potential data dependencies. From this generalized list, select those dependencies that can be forwarded (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls.

| Dependency Type | Producing Signal(s) | Consuming Signal(s) | Typical Example |
|---|---|---|---|
| ALU-ALU | s_EXalu, s_WBalu | s_forwardA, s_forwardB | add → sub |
| Load-ALU | s_WBalu (from lw) | s_forwardA, s_forwardB | lw → add |
| Load-Store | s_WBalu | s_EXalu (address), s_EXrt | lw → sw |
| ALU-Branch | s_EXalu | s_forwardA2, s_forwardB2 | add → beq |
| Load-Branch | s_WBalu | s_forwardA2, s_forwardB2 | lw → beq |
| jal-Any | s_WBPC | s_forwardA, s_forwardB, or others | jal → jr |
| ALU-ShiftVar | s_EXalu | s_forwardA (when using rs) | add → sllv/srlv/srav |
| sw Dependency | s_EXalu, s_EXrt | s_DMemAddr, s_DMemData | add → sw (address), lw → sw (data) |

| Dependency Type | Forward From → To | Notes |
|---|---|---|
| EX → EX | s_EXalu or s_EXPC → | Common for most ALU-to-ALU R-type |

| Dependency Type | Forward From → To | Notes |
|---|---|---|
| | s_forwardA, s_forwardB | and I-type instruction pairs. |
| MEM → EX | s_WBalu or s_WBPC → s_forwardA, s_forwardB | Slightly later producer, but still usable for next instruction's EX stage. |
| EX → MEM | s_EXrt → s_DMemData | For sw, forwarding $rt from previous instruction |
| MEM → MEM | s_EXrt from MEM stage → another sw in MEM stage | Less common, but can occur. |
| WB → EX | s_RegWrData → s_forwardA, s_forwardB | Only works when pipeline has 3+ stage delay and back-to-back instructions aren't dependent. |

Stall-Required Dependencies
These occur when the value is not yet available when needed in the EX stage — i.e., a load-use hazard or too tight of a timing window.

| Dependency Type | Reason | Example |
|---|---|---|
| Load (MEM) → EX | Load value is not ready until end of MEM stage, but EX needs it now | lw $t1, 0($t0) → add $t2, $t1, $t3 |
| jal → jr | Return address not written until WB, but jr uses it in ID/EX | jal foo → jr $ra |
| Load → Store (sw) | lw result is in MEM/WB, but sw needs it in EX for s_EXrt | lw $t0, 0($s1) → sw $t0, 4($s2) |

[2.b.iv] global list of the datapath values and control signals that are required during each pipeline stage

**IF Stage**

- **Datapath Values**
  - i_pc - Current Program Counter (PC) value (Input)
  - s_output - Final computed next PC value (goes to o_pc) (Output)
  - s_adderbranch - PC + branch offset (Internal Calculation)
  - s_append - Jump address after processing (Internal Calculation)
  - i_instruction - The fetched instruction (Output, stored in IF/ID register)
  - PC_plus_4 - The value of PC + 4 (Output, stored in IF/ID register)
- **Control Signals**
  - i_branch - Indicates a conditional branch (e.g., beq) (Input from instruction decode)
  - i_zero - Result of ALU zero flag (used for beq/bne decision) (Input from EX/MEM)
  - i_jump - Indicates an unconditional jump (Input from instruction decode)

- i_RegJump - Indicates a register-based jump (e.g., jr) (Input from instruction decode)
- i_BNE - Used to select between BEQ (i_zero) and BNE (!i_zero) behavior (Input from instruction decode)
- s_branchmux, s_branchmuxbne, s_branchMaster - Intermediate mux control outputs (Internal Logic)
- s_and, s_andbne, s_inv - Intermediate logic control signals (Internal Logic)
- s_xor - Used to check if PC changes (helps generate pcSrc) (Internal Logic)
- o_pcSrc - Final control output to indicate PC has changed (for hazard handling etc.) (Output)

**ID Stage**

- **Datapath Values**
  - i_instruction - Instruction from IF/ID register (Input)
  - i_PC_plus_4 - PC + 4 from IF/ID register (Input)
  - Read Register 1 Value (Value of Rs) (Output, stored in ID/EX register)
  - Read Register 2 Value (Value of Rt) (Output, stored in ID/EX register)
  - Sign-extended Immediate Value (Output, stored in ID/EX register)
  - Instruction bits [15-11] (Potential Rd address) (Output, stored in ID/EX register)
  - Instruction bits [20-16] (Potential Rt address) (Output, stored in ID/EX register)
  - Instruction bits [25-21] (Rs address) (Output, stored in ID/EX register)
  - Jump Target Address (Calculated from instruction bits [25-0] and PC+4) (Output, relevant for jump control)
  - PC_plus_4 - Value of PC + 4 (Passed through to ID/EX register)
- **Control Signals**
  - Decoded control signals based on opcode and funct fields (Output, stored in ID/EX register):
    - RegDst
    - ALUSrc
    - MemtoReg
    - RegWrite
    - MemRead
    - MemWrite
    - Branch
    - Jump
    - RegJump (if not handled purely in IF PC control)
    - BNE (if not handled purely in IF PC control)
    - ALUOp

**EX Stage**

- **Datapath Values**
  - i_Read_Data_1 - Value of Rs from ID/EX register (Input)
  - i_Read_Data_2 - Value of Rt from ID/EX register (Input)

    - i_Sign_Extended_Immediate - Sign-extended immediate from ID/EX register (Input)
    - i_Instruction_15_11 - Rd address from ID/EX register (Input)
    - i_Instruction_20_16 - Rt address from ID/EX register (Input)
    - i_Instruction_25_21 - Rs address from ID/EX register (Input)
    - i_PC_plus_4 - PC + 4 from ID/EX register (Input)
    - i_shmt - Shift amount for shift operations (Input from instruction bits [10-6])
    - ALU First Operand (i_input1) (Input, selected by mux based on instruction type/forwarding)
    - ALU Second Operand (i_input2) (Input, selected by ALUSrc and forwarding)
    - ALU Result (s_output) (Output, stored in EX/MEM register)
    - s_adderOutput, s_and, s_or, s_xor, s_nor, s_shifter, s_lessthan - Internal ALU/shifter results (Internal Calculation)
    - o_zero - Zero flag output from ALU (Output, stored in EX/MEM register)
    - o_overflow - Overflow signal output from ALU (Output)
    - Read_Data_2 - Value of Rt (needed for store instructions) (Output, stored in EX/MEM register)
    - Destination Register Address (Selected Rd or Rt based on RegDst) (Output, stored in EX/MEM register)
    - Branch Target Address (Calculated from PC+4 and Sign-extended Immediate) (Output, compared with Zero flag for branches)
- **Control Signals**
    - Input Control Signals from ID/EX register: RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, Jump, RegJump, BNE, ALUOp
    - i_control(3 downto 0) - Main ALU control signal (Input from ALU control unit)
    - i_overflowEN - Enables overflow detection logic (Input from ALU control unit)
    - i_shiftregEN - Selects between using shift amount (i_shmt) or register value for variable shifts (Input from ALU control unit)
    - Output Control Signals (relevant for MEM and WB, stored in EX/MEM register):
        - MemRead
        - MemWrite
        - MemtoReg
        - RegWrite
        - Branch (passed for potential hazard logic in MEM)
        - Zero (passed for branch decision)

**MEM Stage**

- **Datapath Values**
    - i_ALU_Result - ALU result (memory address or data) from EX/MEM register (Input)

    - i_Read_Data_2 - Value of Rt (data to be written for stores) from EX/MEM register (Input)
    - i_Destination_Register_Address - Destination register address from EX/MEM register (Input)
    - addr - The memory address generated from the EX stage (Input/Internal)
    - data - The data to be written into memory (Input/Internal)
    - Data Read from Memory (q) (Output, stored in MEM/WB register if MemRead is active)
    - ALU_Result - ALU result (passed through for non-memory operations) (Output, stored in MEM/WB register)
    - Destination_Register_Address - Destination register address (Passed through to MEM/WB register)
- **Control Signals**
    - Input Control Signals from EX/MEM register: MemRead, MemWrite, MemtoReg, RegWrite, Branch, Zero
    - we (Write Enable) - Controls memory write (Output to data memory)
    - Output Control Signals (relevant for WB, stored in MEM/WB register):
        - MemtoReg
        - RegWrite

**WB Stage**

- **Datapath Values**
    - i_Memory_Data - Data read from memory from MEM/WB register (Input)
    - i_ALU_Result - ALU result from MEM/WB register (Input)
    - i_Destination_Register_Address - Destination register address from MEM/WB register (Input)
    - i_d (Input Data) - This is the final data to be written to the register file (selected between i_Memory_Data and i_ALU_Result based on MemtoReg) (Input to Register File)
    - o_D1 and o_D2 - Data read from registers (Outputs from Register File, not directly pipeline values stored for this instruction)
- **Control Signals**
    - Input Control Signals from MEM/WB register: MemtoReg, RegWrite
    - i_we (Write Enable) - Controls writing to the register file (Input to Register File)
    - i_rst (Reset) - Resets the registers (Input to Register File)
    - i_clock - Clock signal (Input)

[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

jr (Jump Register) - R-Type
beq (Branch if Equal) - I-Type
bne (Branch if Not Equal) - I-Type
jal (Jump and Link) - J-Type

jr: The target address is taken from a register, which is read during the Instruction Decode (ID) stage. The decision and the new PC value are determined in this stage.
beq: The comparison of registers and the calculation of the branch target address (PC + offset) occur in the EX stage. The decision to take the branch is made there, leading to a potential non-sequential PC update.
bne: the comparison and target address calculation happen in the EX stage, and the branch decision is made there.
jal: The jump target address is calculated based on the immediate field and the current PC in the ID stage. The return address (PC + 4) is also saved in the $ra register in this stage. The non-sequential PC update is determined in ID

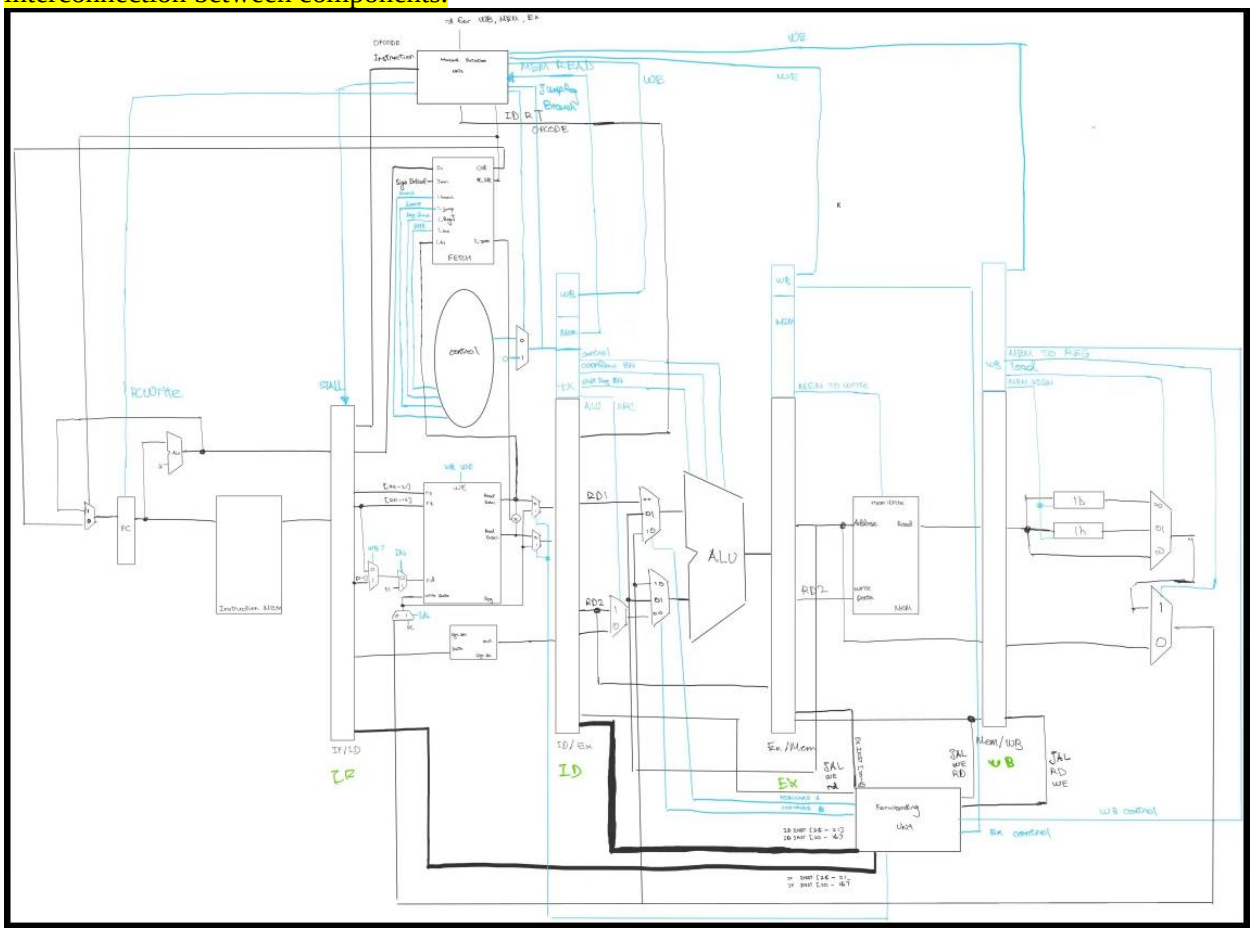
[2.c.ii] For these instructions, list which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each of these instructions is in.

1. jr (Jump Register)
   - Decision/Target determined in: Instruction Decode (ID) stage.
   - When the jr instruction is in the ID stage, the instruction that was fetched immediately after it (and is now in the IF stage) is the wrong instruction because the jump is unconditional.
   - Stages to be Stalled (relative to jr in ID): None immediately at this point
   - Stages to be Squashed/Flushed (relative to jr in ID): The instruction in the IF stage. This instruction is discarded, and the fetch unit is redirected to the target address determined in the ID stage for the next cycle.
2. beq (Branch if Equal) and bne (Branch if Not Equal)
   - Decision determined in: EX
   - Control hazards for branches are handled by stalling until the branch outcome is known
   - When the branch instruction (beq or bne) reaches the ID stage, the pipeline detects a potential control hazard. To avoid fetching and executing incorrect instructions, the pipeline is stalled.
   - Stages to be Stalled (relative to branch in ID): The IF stage and the latch between IF and ID are stalled. This prevents new instructions from entering the pipeline behind the branch.
   - When the branch instruction reaches the EX stage, the condition is evaluated, and the decision to take the branch is made.
   - If the branch is not taken, the stall is released, and the instructions that were held in IF and at the ID/EX latch proceed through the pipeline. No squashing is needed.
   - If the branch is taken, the instructions that were held in the IF and ID stages (which are the instructions sequentially after the branch) are incorrect.
   - Stages to be Squashed/Flushed (relative to branch in EX, if taken): The instructions in the ID stage and the IF stage. These instructions are discarded, and the fetch unit is redirected to the branch target address determined in the EX stage for the next cycle.

3. jal (Jump and Link)
   - Decision/Target determined in: Instruction Decode (ID) stage.

- Similar to jr, the jal instruction unconditionally changes the PC. When jal is in the ID stage, the instruction in the Instruction Fetch (IF) stage is the sequentially fetched instruction immediately following the jal, which is incorrect.
- Stages to be Stalled (relative to jal in ID): None immediately at this point
- Stages to be Squashed/Flushed (relative to jal in ID): The instruction in the IF stage. This instruction is discarded, and the fetch unit is directed to the target address calculated in the ID stage for the next cycle.

[2.d] implement the hardware-scheduled pipeline using only structural VHDL. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components.

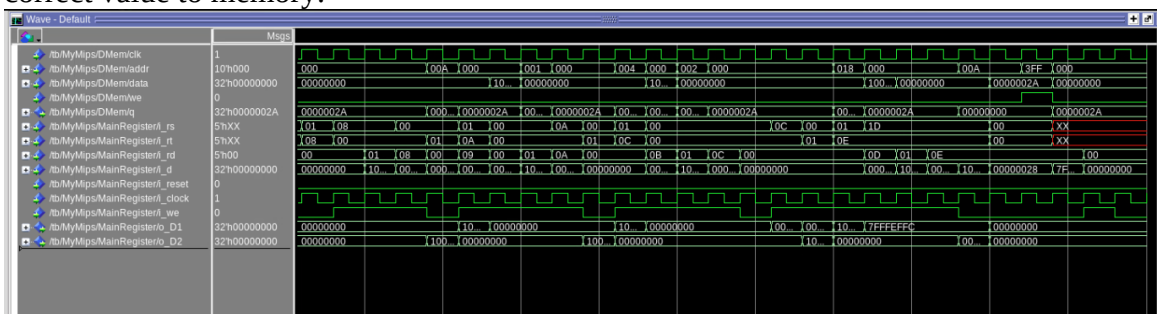



[2.e – i, ii, and iii] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

load_use.s

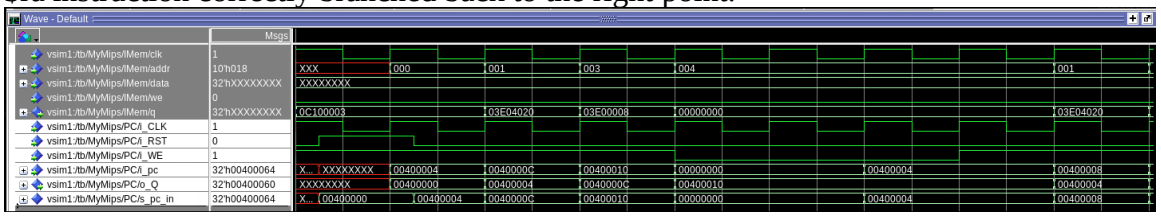
This waveform is correct because in Test 1, I observed a stall inserted between the lw and add instructions, which is expected due to a load-use hazard — the add depends on the result of the lw, which hasn't yet reached the MEM/WB stage. This confirms that the processor detects the hazard and stalls the pipeline. In Test 2, with one nop between the

lw and dependent add, the forwarding logic kicks in from the MEM/WB pipeline register, and no stall is needed — this is exactly what I expected to see. For Test 3, with a lot of instructions between the load and the dependent add, there is no hazard, and the pipeline goes without any stalls. in Test 4, where a lw is followed by a sw using the same register, I verified that the value loaded into $t6 was correctly handled so that the store writes the correct value to memory.




jal.s

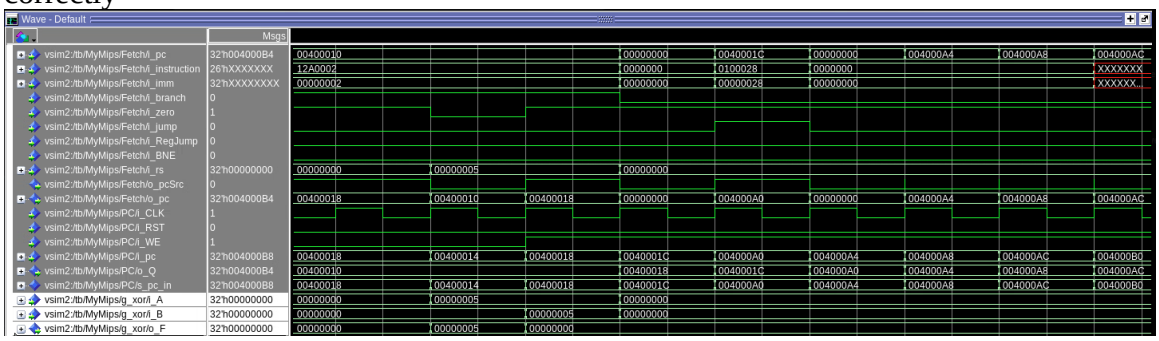
In the QuestaSim waveform for jal.s, the output clearly shows that the processor correctly handles return address forwarding from the jal instruction in various pipeline stages. In Test 1, the add $t0, $ra, $zero instruction immediately follows jal, and I confirmed from the waveform that $ra was correctly forwarded from the ID stage. This is essential because $ra isn't yet written back to the register file at this point. In Test 2, with a single nop delay between the jal and the add, the waveform showed that forwarding occurred from the MEM/WB stage instead — this behavior is expected and demonstrates the processor's support for multi-stage forwarding paths. In Test 3, after multiple nops between jal and the add, the waveform confirmed that no hazard occurred, and $ra was read directly from the register file without needing forwarding. Each return from the jr $ra instruction correctly branched back to the right point.




beq_bne_forwarding.s

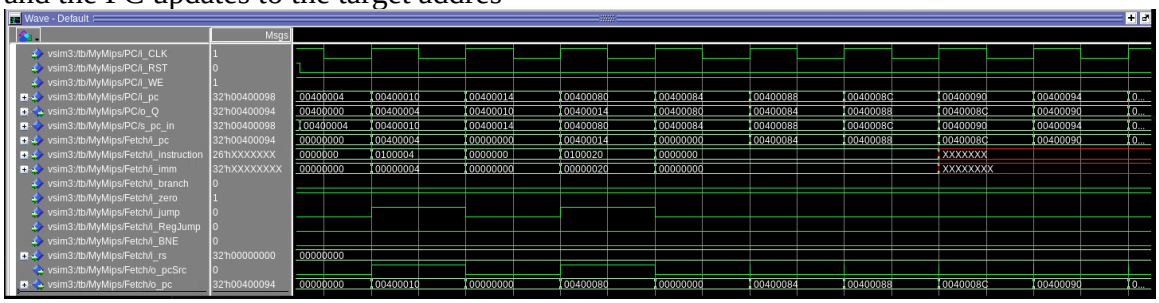
In the QuestaSim waveform for beq_bne_forwarding.s, I observed that each branch instruction was executed with the correct behavior based on the forwarding paths. For Tests 1 and 2, the processor successfully forwarded values to the beq instruction from the EX/MEM and MEM/WB stages, and the branches were taken as expected. Similarly, for Tests 4 and 5, bne was correctly resolved without branching when the forwarded values matched. Test 6 verified that bne correctly branched when the operands were different. In the waveform, you can see that when the comparison result is 0 (indicating equality for beq or inequality for bne), the branch was taken and the PC updated

correctly



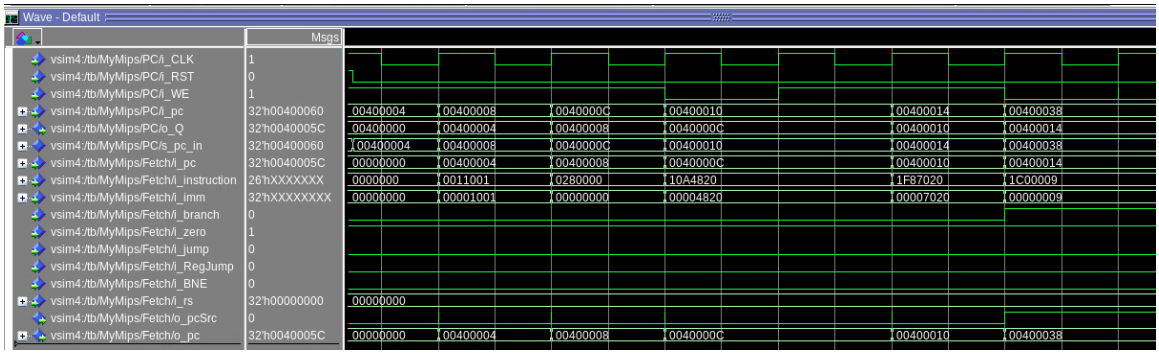

jr_forwarding.s

In the waveform for jr_forwarding.s, each jump register (jr) instruction correctly used the most recent value of the register being jumped to, demonstrating proper forwarding behavior. For Tests 1 and 4, I verified that the processor correctly forwarded the value from the EX/MEM stage to the jr, allowing an immediate jump. In Tests 2 and 5, the forwarding from the MEM/WB stage worked as expected, and in Test 6, I confirmed that $ra was correctly written during the ID stage and used immediately in the following jr. In the waveform, you can see that when the value is correctly forwarded, the branch occurs and the PC updates to the target addres



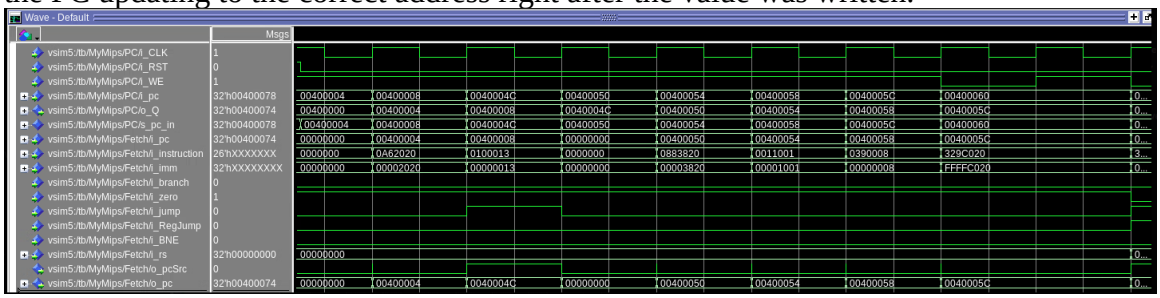

combo1.s

In the waveform for combo1.s, I observed correct handling of several types of data hazards. The load-use hazards after both lw instructions were managed properly, with the second instruction stalling for one cycle before using the loaded data. The I-type hazard with beq worked as expected—forwarding the result of the previous add to the comparison logic, and branching correctly based on the outcome. The J-type hazard with jal correctly saved the return address in $ra, and the jr that followed later successfully used the computed jump target from a forwarded register. In the waveform, I could see that when a branch or jump occurred (indicated by a PC update), it was always due to correctly forwarded or resolved register values.



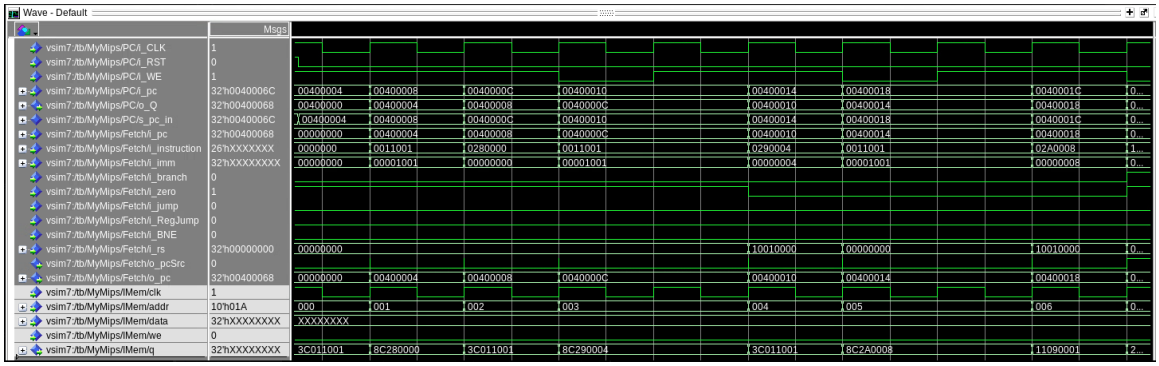

combo2.s

In the waveform for combo2.s, I verified that all hazards were correctly handled by the forwarding and stalling logic. The J-type hazard with jal worked properly—$a0 was forwarded into the subroutine, and I saw that $a3 received the correct value. For the load-use hazards, I observed a one-cycle stall after each lw, indicating that the pipeline correctly inserted bubbles to avoid using data before it's ready. The I-type hazard with bne correctly forwarded the new $s0 value to the branch comparison, and the R-type hazard involving jr after computing the jump target in $t7 also worked as expected, with the PC updating to the correct address right after the value was written.



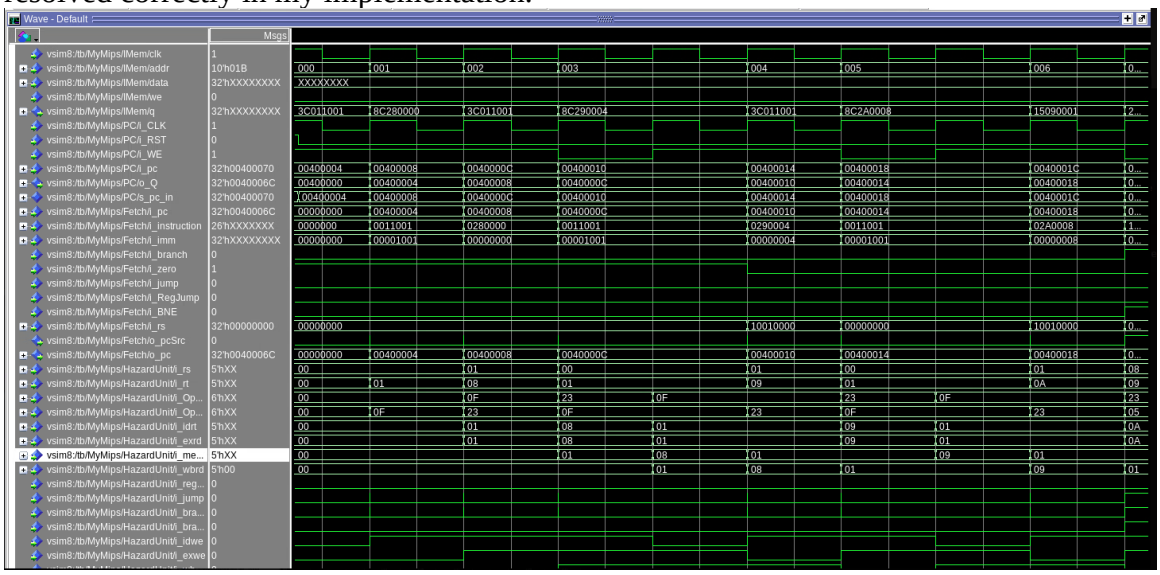

beq_tests.s

In the waveform for beq_tests.s, I observed that all branch instructions behaved as expected based on the register values. In Test 1, the branch was correctly taken since $t0 == $t1, and the instruction following the beq was skipped. In Test 2, the branch was not taken because $t0 != $t2, so the addi executed normally. The consecutive branches in Test 3 showed that the pipeline correctly handled back-to-back control hazards—one taken and one not taken. Finally, in Test 4, I saw the loop run exactly three times, with $t3 decrementing and the branch taken until it reached zero, at which point control exited the loop. Each of these behaviors confirms proper control hazard handling and correct PC updates.



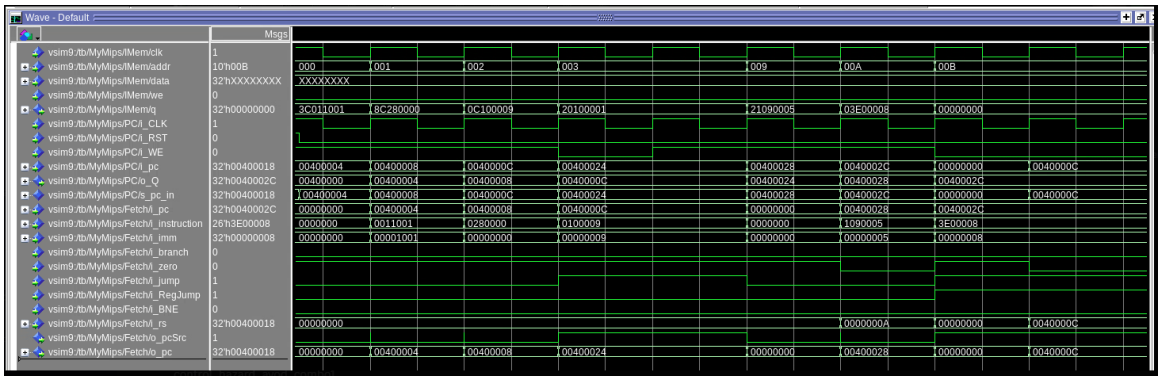

bne_tests.s

In the waveform for bne_tests.s, each branch behaved as expected according to the values in the registers. In Test 1, the branch was correctly taken since $t0 != $t1, and the instruction immediately following the bne was skipped. In Test 2, the branch was not taken because $t0 == $t2, allowing the addi to execute as intended. The consecutive bne branches in Test 3 demonstrated correct pipeline behavior, with one branch taken and the next not taken, and the value of $s2 being set properly. Finally, Test 4 showed the loop running exactly three times, confirming that the bne successfully controlled the loop with proper decrementing and exit behavior. This all confirms that control hazards were resolved correctly in my implementation.



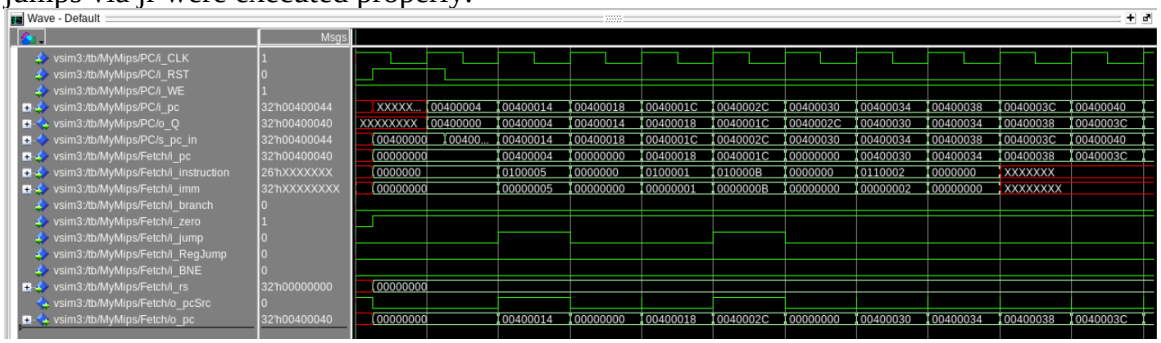

jal_tests.s

In the waveform for this test, everything behaved as expected. For Test 1, the jal func1 correctly saved the return address in $ra, and the delay slot instruction addi $s0, $zero, 1 executed before the jump occurred. The subroutine returned properly using jr $ra, and control resumed right after the jal, executing the nop. Test 2 also behaved correctly—jal func2 jumped to the second subroutine while executing the delay slot instruction that set $s1, and then returned cleanly to continue execution. This confirms that both jump-and-link behavior and delay slot handling are functioning correctly in my pipeline.



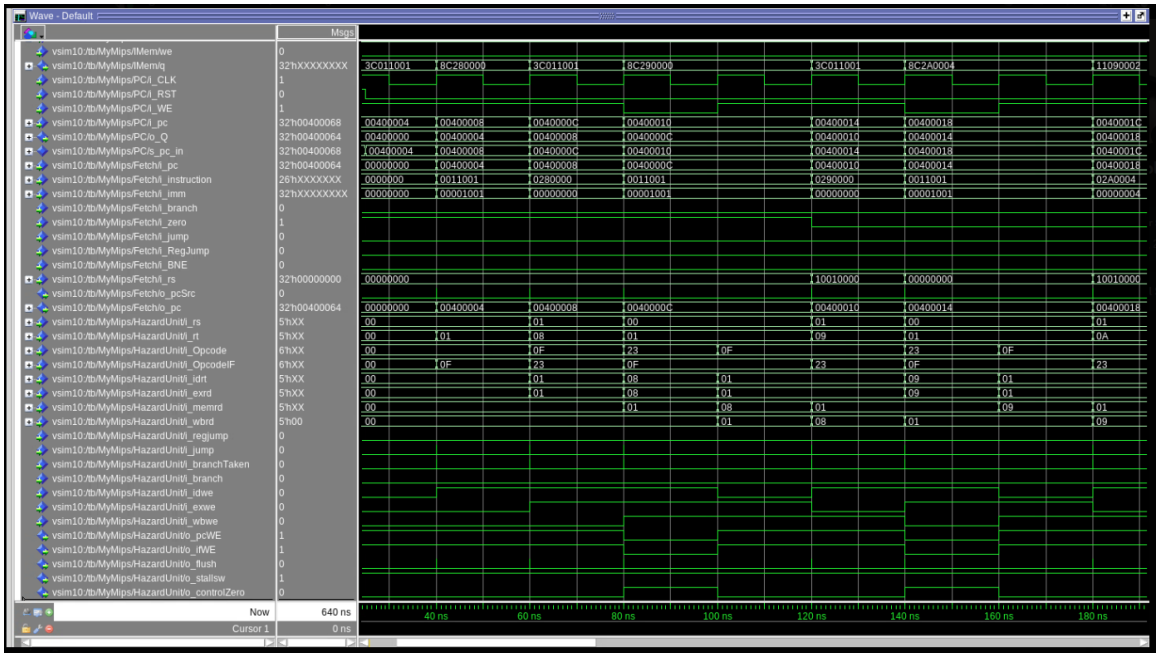

jr_tests.s

The waveform for jr_test.s shows that both jr instructions function correctly by jumping to the target addresses stored in $ra. After the first jal target_return, the return address is correctly saved and then moved to $t0, and the jr $t0 successfully redirects execution to target_return, where $s0 is set to 1 as expected. The same pattern occurs for the second test: jal second_target sets $ra, $t1 copies it, and jr $t1 correctly transfers control to second_target, where $s1 is set to 2. The correct values in $s0 and $s1 confirm that both jumps via jr were executed properly.



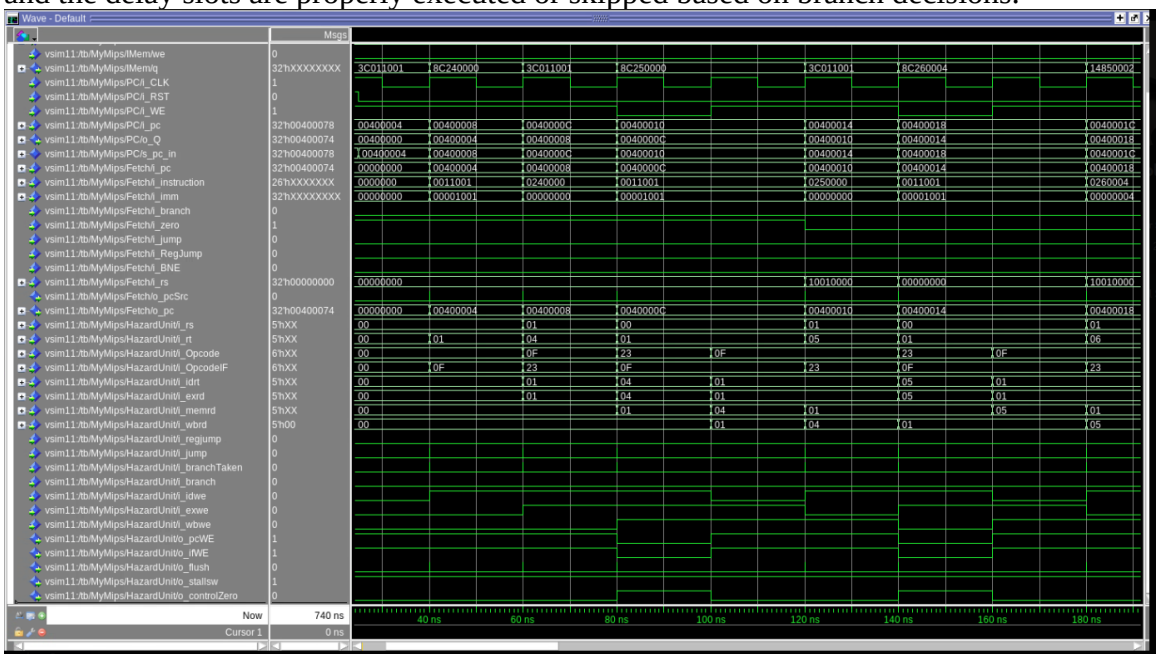

control_hazard_avoidance_combo1.s

The waveform output for this test sequence confirms correct handling of control hazards. In Test 1, the beq $t0, $t1 comparison evaluates true since both hold 42, so the branch is taken and the addi $s0, $zero, 1 instruction is correctly skipped. Similarly, in Test 2, bne $t0, $t2 evaluates true (42 ≠ 100), so the branch to label_bne_taken is taken, and the addi $s1 is skipped as expected. Finally, in Test 3, the jal my_subroutine correctly jumps to the subroutine, executes the delay slot, and then uses jr $ra to return, allowing the program to cleanly jump to end_program and halt. This shows the pipeline is correctly managing control flow changes and delay slots.




control_hazard_avoidance_combo2.s

The waveform output confirms the correct handling of control hazards and delay slots in this program. In Test 1, the bne $a0, $a1 evaluates to false (both are 10), so the branch is not taken and the addi $s3, $zero, 4 instruction is executed. Similarly, in Test 2, the beq $a0, $a2 is false (10 ≠ 99), so the branch is not taken, and addi $s4, $zero, 5 executes as expected. Test 3 shows the jal branch_subroutine instruction correctly jumping to the subroutine, and in Test 4, the beq $a1, $a0 is true (both are 10), so the branch is taken and the addi $s5 is skipped. This confirms that the pipeline correctly handles control hazards, and the delay slots are properly executed or skipped based on branch decisions.



[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

| Hazard Type | Test Description | Test File | Coverage Justification |
|---|---|---|---|
| R-type (jr) | Tests data forwarding from EX/MEM and MEM/WB to jr using $ra and $t registers | jr_forwarding.s | This test is needed because it covers forwarding paths from EX/MEM and MEM/WB, and no hazard case. These cases are handled in the processor so we want to test that they are actually working. |
| I-type (beq, bne) | Tests branch decision hazards with beq and bne using back-to-back register comparisons | beq_bne_forwarding.s | This test is needed because it covers conditional branches with register comparisons and delayed branching logic. These cases are handled in the processor so we want to test that they are actually working. |
| J-type (jal) | Tests return address ($ra) written by jal and used immediately, after 1 delay, or after multiple instructions | jal.s | This test is needed because it Ccovers return address hazards immediately after jal, with delay, and long delay. These cases are handled in the processor so we want to test that they are actually working. |
| Load-use hazard | Tests classic load-use hazard requiring stalling or forwarding to handle dependency | load_use.s | This is needed because it covers cases of immediate use, 1-instruction delay, and safe delay after load. These cases are handled in the processor so we want to test that they are actually working. |
| Combination of data hazard detection and forwarding cases | Activates combinations of different data hazard detection and forwarding cases that can occur simultaneously within the pipeline | combo1.s | combo1.s tests multiple data hazards, including load-use, R-type (jr), I-type (beq), and J-type (jal) hazards in a MIPS pipeline. The program checks if the hazard detection unit and forwarding mechanisms can handle simultaneous dependencies across different instruction types. Specifically, it verifies the system's ability to forward data and manage control flow hazards, such as correctly jumping based on the contents of a register. This program ensures that the pipeline can properly resolve hazards without |

| | | | |
|---|---|---|---|
| | | | unnecessary stalls or errors. |
| More combinations of data hazard detection and forwarding cases | Activates combinations of different data hazard detection and forwarding cases that can occur simultaneously within the pipeline | combo2.s | combo2.s extends the testing of data hazards by introducing multiple load-use hazards alongside R-type (jr), I-type (bne), and J-type (jal) hazards in the pipeline. It evaluates how effectively the processor handles consecutive load-use dependencies and whether data forwarding is applied correctly for both load and jump instructions. The program also stresses the control flow logic, verifying the proper handling of branching and function calls in the presence of multiple hazards. This combination of challenges ensures comprehensive testing of hazard detection, forwarding, and branch prediction mechanisms. |

[2.e.ii] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

| **Hazard Type** | **Test Description** | **Test File** | **Coverage Justification** |
|---|---|---|---|
| Control hazard beq | Tests branch-if-equal control hazards with both taken and not-taken paths, adjacent branches, and dependent arithmetic instructions before/after the branch. | beq_tests.s | Verifies pipeline behavior on conditional branches, branch decision latency, and ensures proper flushing or forwarding is applied when control flow depends on equality comparisons. |
| Control hazard bne | Evaluates branch-if-not-equal behavior under similar conditions as beq, including back-to-back branches and branches with register dependencies. | bne_tests.s | Confirms correct PC update and pipeline behavior under conditional inequality checks; helps test alternative branching condition and its hazard resolution in the pipeline. |
| Control hazard jal | Tests jump-and-link behavior for subroutine calls. Stores return address in $ra, executes instructions after the jump, and checks correct return behavior. | jal_tests.s | Ensures proper handling of function calls, correct saving of the return address, and delay slot behavior. Validates interaction between jumps and return addresses in control hazard scenarios. |

| | | | |
|---|---|---|---|
| Control hazard jr | Tests jump-register instructions used for returning from subroutines. Includes both static jumps to labels and jumps using $ra. Ensures the jump is executed correctly and does not cause infinite loops. | jr_tests.s | Verifies behavior of register-based jumps. Ensures that control returns properly from a jal call and that jump targets stored in registers are handled correctly. Confirms predictable PC updates and pipeline continuation. |
| Combo multiple types | Combines beq, bne, jr, and jal in a single pipeline-intensive scenario. Uses conditional branches followed by subroutine jumps and returns to create a diverse hazard pattern. | control_hazard_avoidance_combo1.s | Stresses the pipeline with overlapping control hazards. Verifies correct hazard detection and resolution when multiple types of control flow instructions interact. Tests branch prediction, PC updates, delay slots, and pipeline flush logic comprehensively. |
| Combo multiple types | Similar to combo1 but varies instruction order, register values, and target addresses. Ensures broad test coverage and that logic isn't overly specialized to specific values or instruction sequences. | control_hazard_avoidance_combo2.s | Increases robustness of testing by validating hazard handling logic in diverse instruction mixes. Ensures correctness and stability of the hazard avoidance mechanism under a different execution path and runtime behavior. |

[2.f] report the maximum frequency your hardware-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

The maximum frequency our processor can run at is 55.35 MHz.

The critical path goes through the main register file because we can see that component is taking the longest in the output file that the tool flow provides us. Because this component is taking the longest, we know this is the component that our critical path goes through. The critical path as a whole goes though main register, to our xor comparator unit, and then to the fetch module, and then to the PC_SRC select mux, and then to PC module.

```
FMax: 55.35mhz Clk Constraint: 20.00ns Slack: 1.93ns

The path is given below

 ====================================================================
 From Node    : N_Reg:IFRegInst|dffg:\NBit_DFF:21:dffi|s_Q
 To Node      : PC_Module:PC|N_Reg:g_pc|dffg:\NBit_DFF:5:dffi|s_Q
 Launch Clock : iCLK
 Latch Clock  : iCLK
 Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ===================================
      0.000      0.000           launch edge time
      3.024      3.024  R        clock network delay
      3.256      0.232     uTco  N_Reg:IFRegInst|dffg:\NBit_DFF:21:dffi|s_Q
      3.256      0.000 FF  CELL  IFRegInst|\NBit_DFF:21:dffi|s_Q|q
      4.609      1.353 FF    IC  MainRegister|g_mux1|Mux28~12|datab
      5.001      0.392 FR  CELL  MainRegister|g_mux1|Mux28~12|combout
      5.952      0.951 RR    IC  MainRegister|g_mux1|Mux28~13|datad
      6.107      0.155 RR  CELL  MainRegister|g_mux1|Mux28~13|combout
      6.816      0.709 RR    IC  MainRegister|g_mux1|Mux28~14|datab
      7.161      0.345 RR  CELL  MainRegister|g_mux1|Mux28~14|combout
      8.497      1.336 RR    IC  MainRegister|g_mux1|Mux28~15|datad
      8.636      0.139 RF  CELL  MainRegister|g_mux1|Mux28~15|combout
      8.905      0.269 FF    IC  MainRegister|g_mux1|Mux28~16|datab
      9.309      0.404 FF  CELL  MainRegister|g_mux1|Mux28~16|combout
      9.724      0.415 FF    IC  MainRegister|g_mux1|Mux28~19|dataa
     10.128      0.404 FF  CELL  MainRegister|g_mux1|Mux28~19|combout
     10.555      0.427 FF    IC  g_zeroflag|Equal0~1|datac
     10.836      0.281 FF  CELL  g_zeroflag|Equal0~1|combout
     11.607      0.771 FF    IC  g_zeroflag|Equal0~4|datab
     11.957      0.350 FF  CELL  g_zeroflag|Equal0~4|combout
     12.233      0.276 FF    IC  g_zeroflag|Equal0~20|dataa
     12.586      0.353 FF  CELL  g_zeroflag|Equal0~20|combout
     12.835      0.249 FF    IC  Fetch|g_RegJump|\G_NBit_MUX:27:MUXI|o_O~2|datad
     12.960      0.125 FF  CELL  Fetch|g_RegJump|\G_NBit_MUX:27:MUXI|o_O~2|combout
     14.158      1.198 FF    IC  Fetch|g_RegJump|\G_NBit_MUX:7:MUXI|o_O~0|dataa
     14.582      0.424 FF  CELL  Fetch|g_RegJump|\G_NBit_MUX:7:MUXI|o_O~0|combout
     14.809      0.227 FF    IC  Fetch|g_RegJump|\G_NBit_MUX:7:MUXI|o_O~1|datad
     14.934      0.125 FF  CELL  Fetch|g_RegJump|\G_NBit_MUX:7:MUXI|o_O~1|combout
     15.225      0.291 FF    IC  Fetch|g_zeroflag|Equal0~3|dataa
     15.649      0.424 FF  CELL  Fetch|g_zeroflag|Equal0~3|combout
     16.600      0.951 FF    IC  Fetch|g_zeroflag|Equal0~5|datad
```