CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: __________Ethan Roepke___________

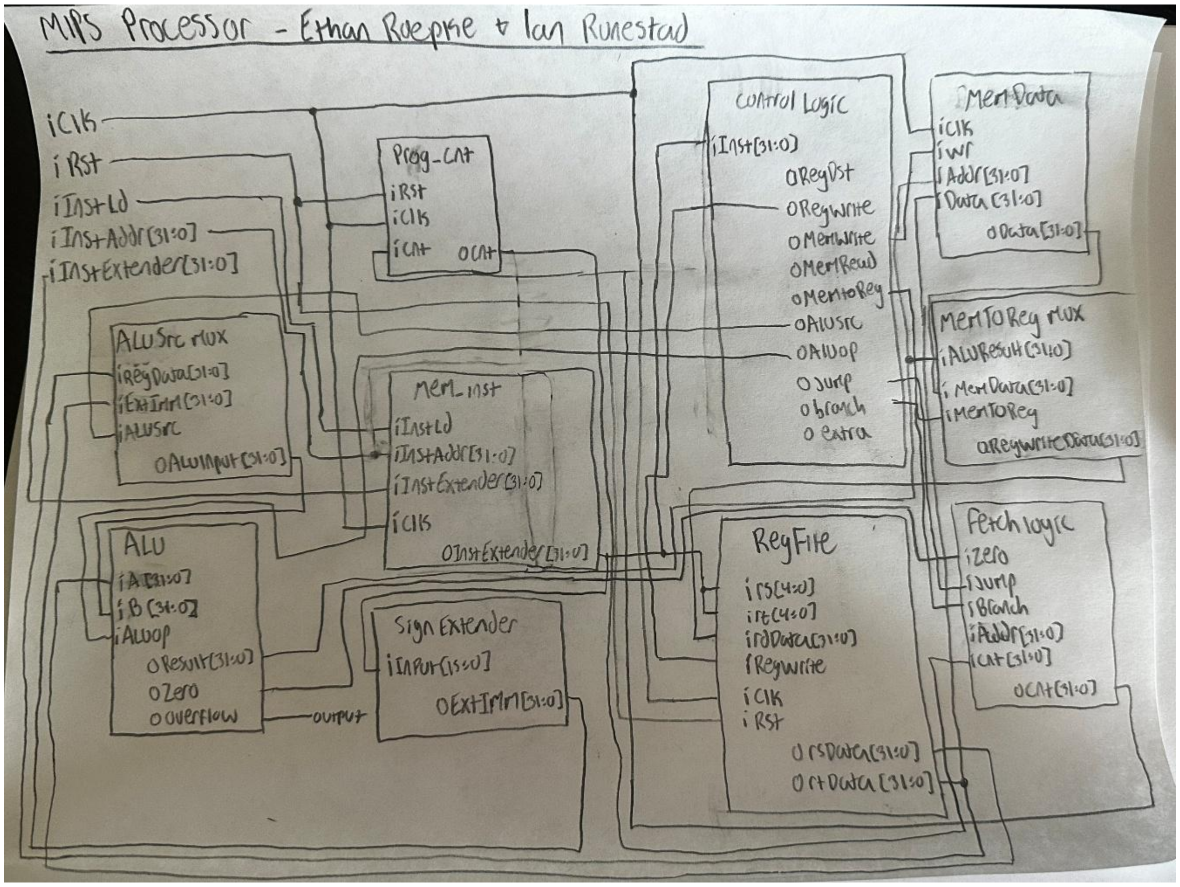
__________Ian Runestad_________________

____

Project Teams Group #:______ Group A_10 ____

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

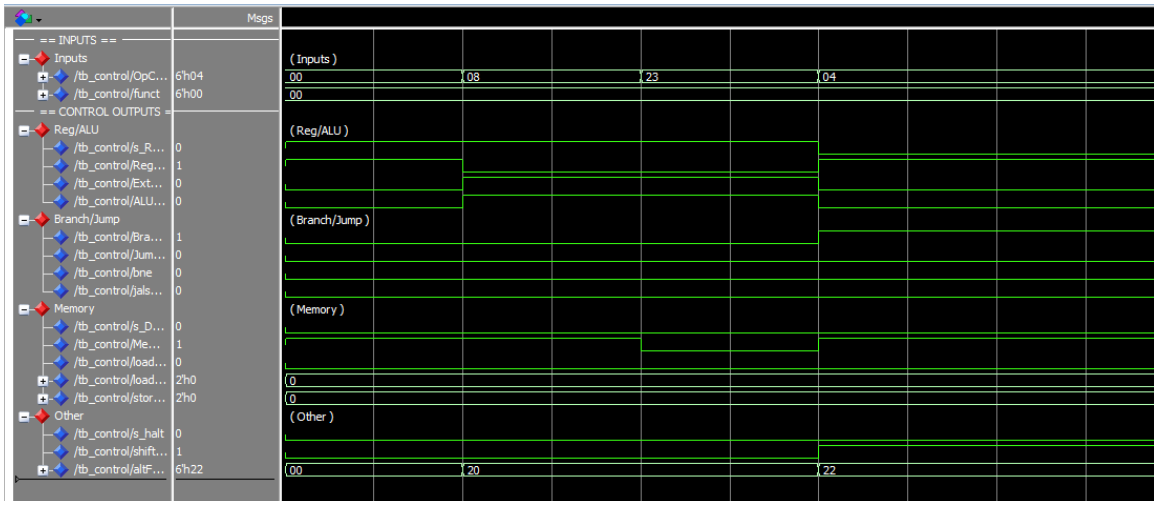
[Part 1 (d)] Include your final MIPS processor schematic in your lab report.



[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your datapath implementation. The end result should be an *N***M* table where each row corresponds to the output of the control logic module for a given instruction.

Instruction	Op5	Op4	Op3	Op2	Op1	Op0	Fun5	Fun4
add	0	0	0	0	0	0	1	0
sub	0	0	0	0	0	0	1	0
and	0	0	0	0	0	0	1	0
or	0	0	0	0	0	0	1	0
xor	0	0	0	0	0	0	1	0
slt	0	0	0	0	0	0	1	0
sll	0	0	0	0	0	0	0	0
srl	0	0	0	0	0	0	0	0
sra	0	0	0	0	0	0	0	0
lw	1	0	0	0	1	1	0	0
lh	1	0	0	1	0	1	0	0
lb	1	0	0	1	0	0	0	0
lhu	1	0	1	0	0	1	0	0
sw	1	0	1	0	1	1	0	0
beq	0	0	0	1	0	0	0	0
bne	0	0	0	1	0	1	0	0
addi	0	0	1	0	0	0	0	0
j	0	0	0	0	1	0	0	0
jal	0	0	0	0	1	1	0	0
halt	1	0	1	0	0	0	0	0

[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually and show that your output matches the expected control signals from problem 1(a).


```
                                   Msgs
== INPUTS ==
Inputs
  /tb_control/OpC...               6'h04
  /tb_control/funct                6'h00
== CONTROL OUTPUTS ==
Reg/ALU
  /tb_control/s_R...               0
  /tb_control/Reg...               1
  /tb_control/Ext...               0
  /tb_control/ALU...               0
Branch/Jump
  /tb_control/Bra...               1
  /tb_control/Jum...               0
  /tb_control/bne                  0
  /tb_control/jals...              0
Memory
  /tb_control/s_D...               0
  /tb_control/Me...                1
  /tb_control/load...              0
  /tb_control/load...              2'h0
  /tb_control/stor...              2'h0
Other
  /tb_control/s_halt               0
  /tb_control/shift...             1
  /tb_control/altF...              6'h22
```

- Normal Sequential Flow:
  - PC is updated as PC = PC + 4

    - Used by most instructions like add, sub, lw, sw
- Unconditional Jump:
    - Updates PC with jump target: PC = {PC[31:28], target, 00}
    - Jump address is embedded in the instruction
- Jump and Link:
    - Same PC update as j: PC = {PC[31:28], target, 00}
    - Also saves return address (PC + 4) to register $ra
- Jump Register:
    - PC is set to the value in a register: PC = $rs
    - Requires reading a register for the next PC (resolved in Execute stage)
- Branch if Equal:
    - Conditional jump: if $rs == $rt, then PC = PC + 4 + (offset << 2)
    - Target is relative to current PC
- Branch if Not Equal:
    - Conditional jump: if $rs != $rt, then PC = PC + 4 + (offset << 2)
    - Also uses a relative offset

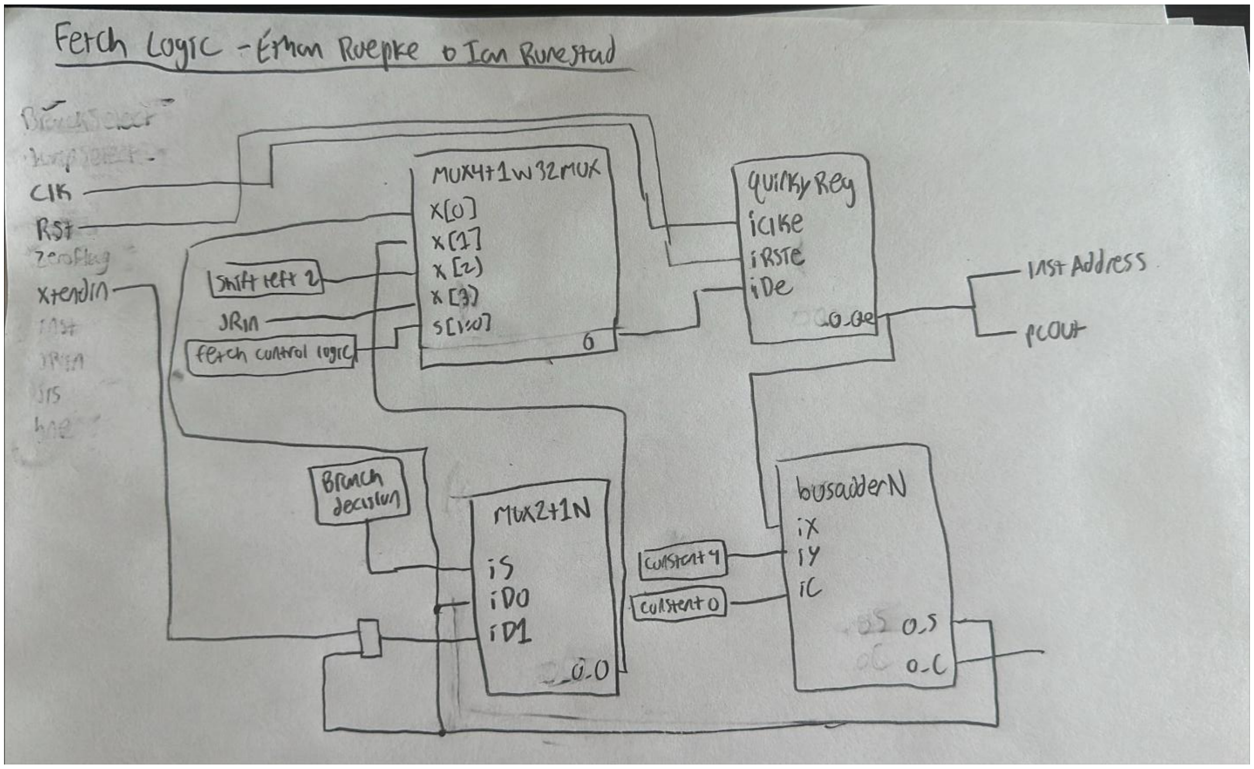
[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

- Sequential Execution:
    - Default case when no jump, branch, or register jump is active.
    - Fetch logic increments PC by 4 to fetch the next instruction.
- Unconditional Jump:
    - When JumpSelect = 1, fetch logic replaces PC with jump target address from instruction.
    - Target address is computed as {PC[31:28], inst[25:0], 00}.
- Branch Instructions:

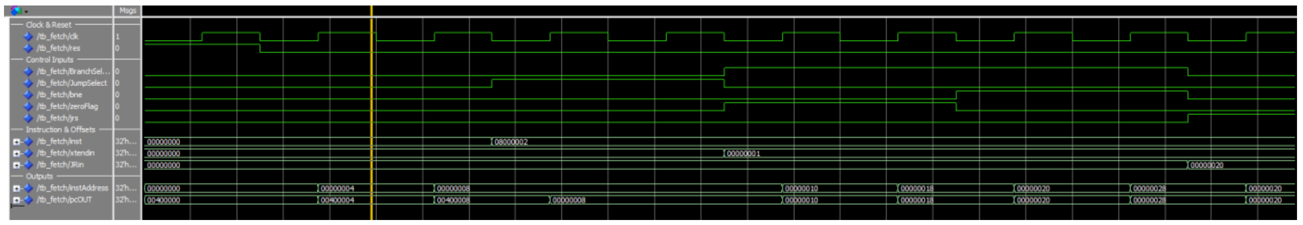
- When BranchSelect = 1, fetch logic adds sign-extended and shifted xtendin (offset) to PC + 4.
- Condition (zeroFlag or bne logic) determines if branch is taken.
- If branch taken → PC = PC + 4 + (offset << 2).
- If not taken → PC = PC + 4.

- Jump Register:
    - When jrs = 1, fetch logic loads PC directly from register value JRin.
    - Overrides all other PC updates when active.

[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?



[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use QuestaSim to test your design thoroughly to make sure it is working as expected. Describe how the execution of the control flow possibilities corresponds to the QuestaSim waveforms in your writeup.



- Initial Reset (res = '1')
    - PC (instAddress / pcOUT) is held at 0.
    - No instruction fetch happens.
    - Confirms correct reset behavior.
- Sequential Execution (default case)
    - When JumpSelect, BranchSelect, and jrs are all '0', the PC increments by 4 each clock.
    - Waveform shows PC = 0 → 4 → 8 → 12, etc.
    - Confirms PC + 4 progression.
- Jump Instruction (JumpSelect = '1')
    - Jump target from inst(25:0) is used to compute jump address
    - PC jumps to 0x08, visible as a discontinuity in waveform.
    - Confirms correct jump operation.
- Branch Equal (BranchSelect = '1', zeroFlag = '1')
    - Branch is taken; offset (xtendin) is added to PC+4.
    - PC jump visible as skip ahead in waveform.
    - Confirms BEQ behavior.
- Branch Not Equal (BranchSelect = '1', bne = '1', zeroFlag = '0')
    - Branch taken when zero flag is 0.
    - Offset added as in BEQ.

- PC jump by 8 shows in waveform.
- Confirms BNE behavior.

• Jump Register (jrs = '1')

- PC is updated directly from JRin.
- PC = 0x00000020 shows as abrupt jump in waveform.
- Confirms JR support and direct control of PC.

[Part 2 (c.i.1)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does MIPS not have a `sla` instruction?

In MIPS, the logical shift right and arithmetic shift righ are two types of shift operations that differ in how they handle the sign bit during shifting. The SRL operation shifts the bits to the right while filling the leftmost bits with zeros regardless of the sign of the number. On the other hand, the SRA operation shifts the bits to the right as well, but it preserves the sign of the number by filling the leftmost bits with the sign bit. This makes the SRA suitable for signed values, ensuring that the result maintains the correct sign when performing the shift.

MIPS does not include a shift left arithmetic instruction because left shifts do not require special handling of the sign bit. When performing a shift left, all bits are moved to the left, and the rightmost bits are filled with zeros. This operation is identical for both signed and unsigned values, so there is no need for a separate arithmetic left shift instruction. The shift left logical operation, which is the equivalent of the SLA is enough for both signed and unsigned numbers.

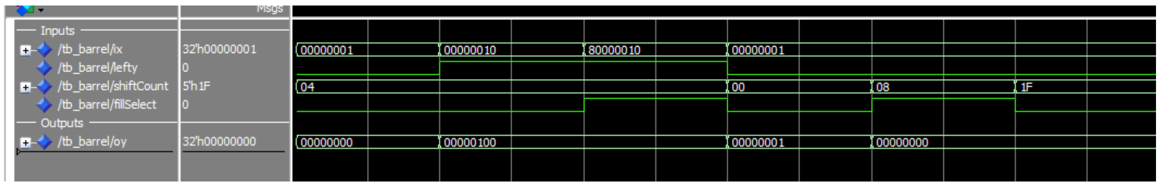
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

The VHDL code implements logical shifts by setting lefty to 0 and fillSelect to 0, filling vacated positions with 0s. Arithmetic shifts are implemented by setting fillSelect to 1, causing the vacated positions to be filled with the sign bit, preserving the number's sign. Both shifts use multiplexers to select the appropriate data path based on the shift amount and direction.

[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

To enhance the right barrel shifter to support left shifting, the lefty control signal can be used to determine the shift direction. When lefty is set to 0, the data will shift left; when lefty is set to 1, the data will shift right. The multiplexers would need to be adjusted to correctly handle left shifts, ensuring the vacated positions are filled with 0s for logical shifts or sign extended bits for arithmetic shifts. This allows the shifter to handle both left and right shifts based on the lefty signal.

[Part 2 (c.i.4)] Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your writeup.



Test 1: Logical Shift Left (LSL)

- Description: Shift ix = 0x00000001 left by 4 positions.
- Expected Result: oy = 0x00000010

Test 2: Logical Shift Right (LSR)

- Description: Shift ix = 0x00000010 right by 4 positions.
- Expected Result: oy = 0x00000001

Test 3: Arithmetic Shift Right (ASR)

- Description: Shift ix = 0x80000010 right by 4 positions (arithmetic shift).
- Expected Result: oy = 0xF0000001

Test 4: No Shift (Shift Count = 0)

- Description: No shift when shiftCount = 0.
- Expected Result: oy = 0x00000001

Test 5: Arithmetic Shift Left (ASL)

- Description: Shift ix = 0x00000001 left by 8 positions.
- Expected Result: oy = 0x01000000

Test 6: Maximal Shift (Shift Count = 31)

- Description: Shift ix = 0x00000001 by 31 positions.
- Expected Result: oy = 0x0000000

[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

For the design of the ALU, we focused on creating a simple and efficient module capable of performing basic arithmetic and logical operations. One key design decision involved how to handle the Overflow and Carryout flags. These flags are essential for detecting errors in arithmetic operations, especially for signed numbers and shift operations. We used sign extension to detect overflow in addition and subtraction. We carefully managed the Carryout signal for addition and subtraction. The design utilized a multiplexer to select operations and a case statement to handle various ALU operations. Resources that we used includes, VHDL libraries and course provided ALU design examples.

[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

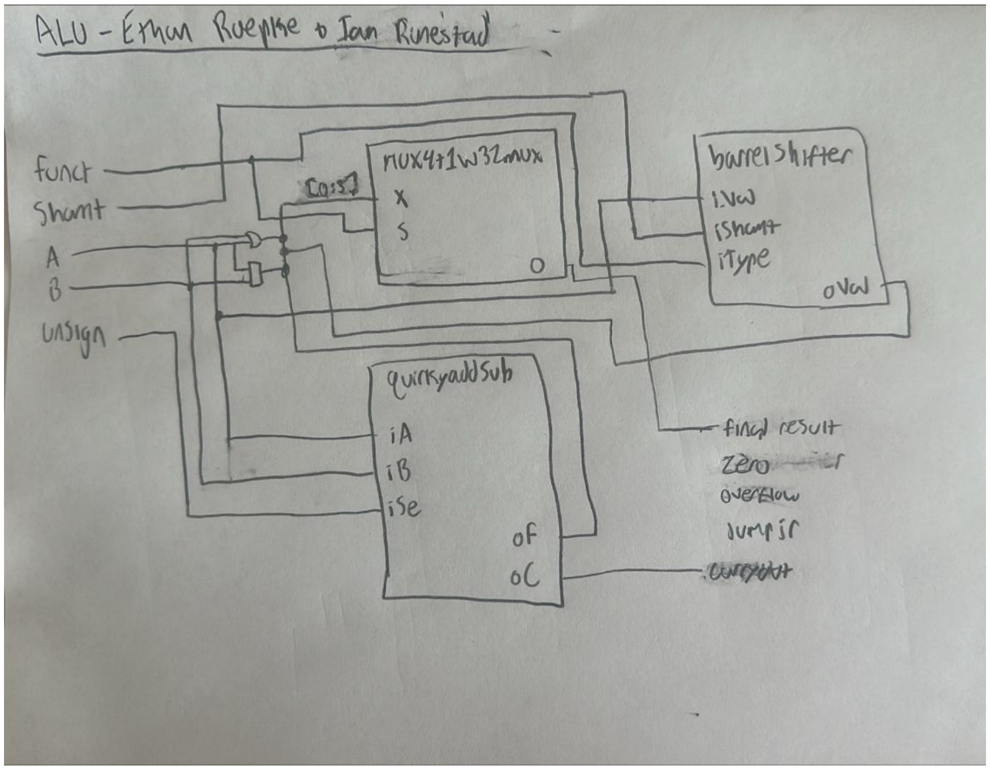
Still need to do, uinsure what to do here

[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is `slt` implemented?

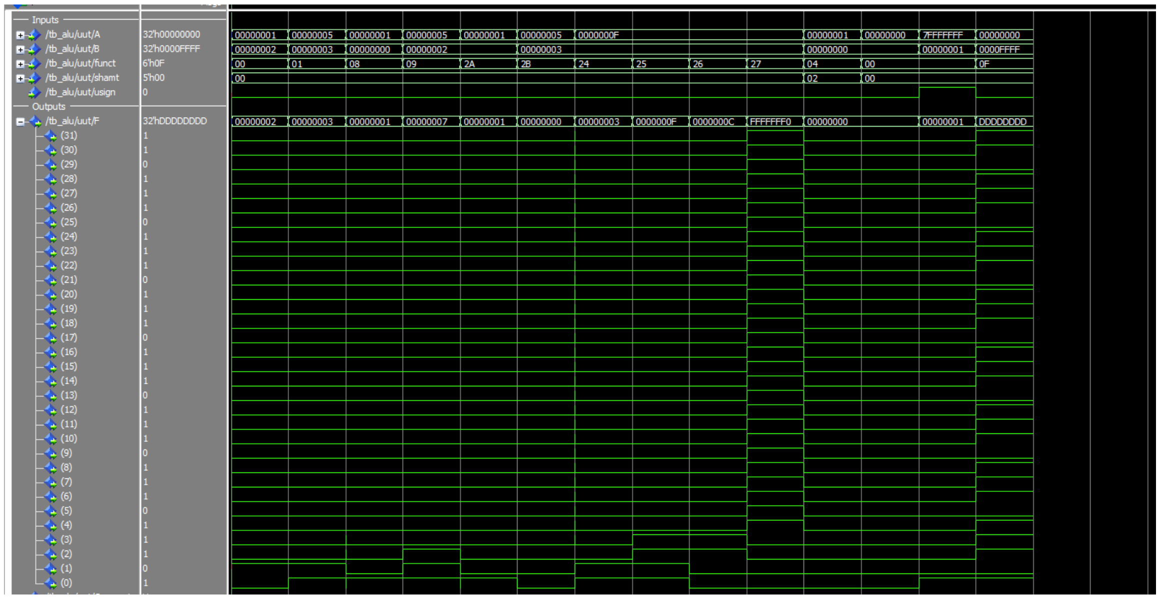
Overflow is calculated during signed addition and subtraction by comparing the sign bits of the operands and the result. If the operands have the same sign but the result's sign is different, overflow occurs. This indicates that the result cannot be represented correctly in signed form.

Zero is calculated by checking if the ALU result is all zeros. If the entire 32-bit result is zero, the Zero flag is set to '1'; otherwise, it is '0'. This simple comparison is used to detect when an operation yields a zero value.

SLT is implemented by subtracting B from A and checking the sign bit of the result. If the result is negative (sign bit is '1'), SLT outputs '1'; otherwise, it outputs '0'. This result is expanded into a 32-bit word with only the least significant bit set.



[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.


Inputs
/tb_alu/uut/A 32'h00000000
/tb_alu/uut/B 32'h0000FFFF
/tb_alu/uut/funct 6'h0F
/tb_alu/uut/shamt 5'h00
/tb_alu/uut/usign 0
Outputs
/tb_alu/uut/F 32'hDDDDDDDD
A: 00000001 00000005 00000001 00000005 00000001 00000005 0000000F 00000001 00000000 7FFFFFFF 00000000
B: 00000002 00000003 00000000 00000002 00000003 00000000 00000001 0000FFFF
funct: 00 01 08 09 2A 2B 24 25 26 27 04 00 0F
shamt: 00 02 00
F: 00000002 00000003 00000001 00000007 00000001 00000000 00000003 0000000F 0000000C FFFFFFF0 00000000 00000001 DDDDDDDD
(31) 1
(30) 1
(29) 0
(28) 1
(27) 1
(26) 1
(25) 0
(24) 1
(23) 1
(22) 1
(21) 0
(20) 1
(19) 1
(18) 1
(17) 0
(16) 1
(15) 1
(14) 1
(13) 0
(12) 1
(11) 1
(10) 1
(9) 0
(8) 1
(7) 1
(6) 1
(5) 0
(4) 1
(3) 1
(2) 1
(1) 0
(0) 1
Tests 1-8 focuses on verifying the ALU's ability to perform addition and subtraction operations with both signed and immediate values, ensuring correct results and proper handling of overflow and carryout flags. Tests 9 and 10 evaluate the Set Less Than operation, ensuring the ALU correctly sets the result to 1 when A is less than B or the immediate value, and 0 otherwise. Tests 11 through 18 test the ALU's ability to perform bitwise operations such as AND, OR, XOR, and NOR, with immediate values to verify the

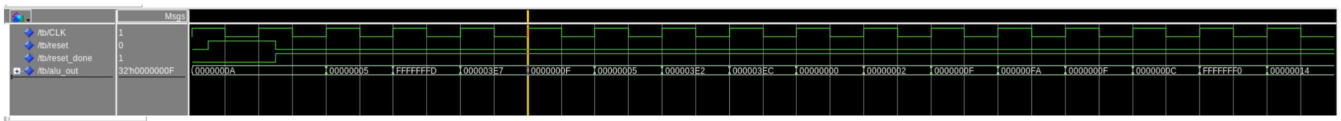
correctness of the bitwise results. Tests 19-24 focuses on shift operations, checking the ALU's ability to shift operands left or right by a specified amount, using the shift amount. Test 25 verifies the Zero flag, ensuring it is correctly set when the ALU operation results in a zero value. Test 26 evaluates the Overflow flag, particularly for unsigned arithmetic, to ensure that overflow is detected correctly when large values are involved. Finally, Test 27 validates the LUI operation, ensuring that the ALU correctly loads an immediate value into the upper 16 bits of the result.

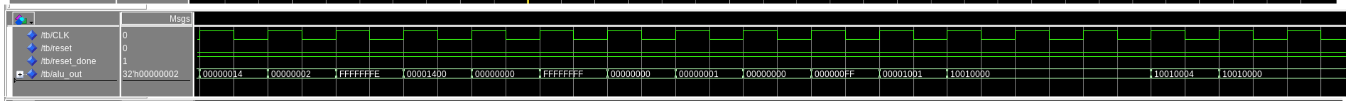
[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

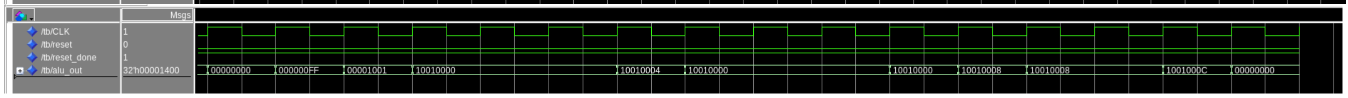
The plan includes tests for arithmetic operations, logical operations, and shift operations which are fundamental to any processor. We have each operation being tested for correctness and special conditions like overflow, carryout, and zero flags. These are explicitly checked to ensure the ALU handles them properly. The addition of tests for flag handling ensures that the ALU behaves as expected under all conditions. The waveforms for these tests clearly demonstrate the ALU's correct operation with flags set as expected. This thorough testing of both normal and edge cases ensures that the ALU performs all its functions correctly and handles corner cases like overflow and zero values accurately.

[Part 3] 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.

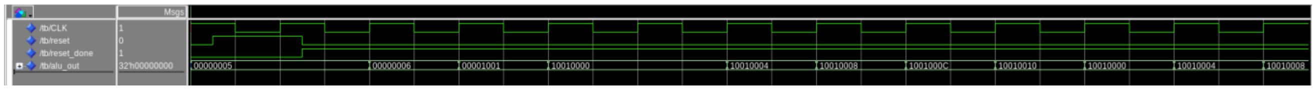
[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Name this file Proj1_base_test.s.

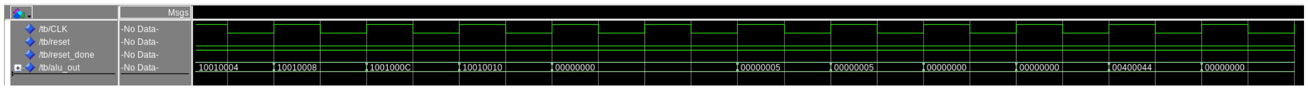




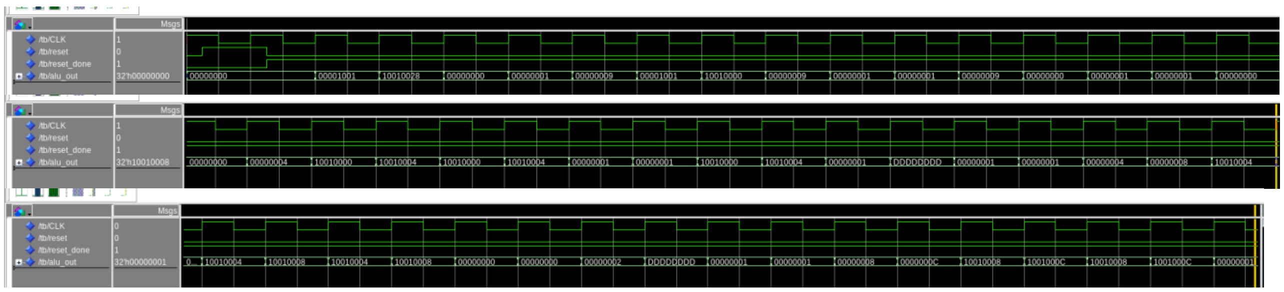


[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.





[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.



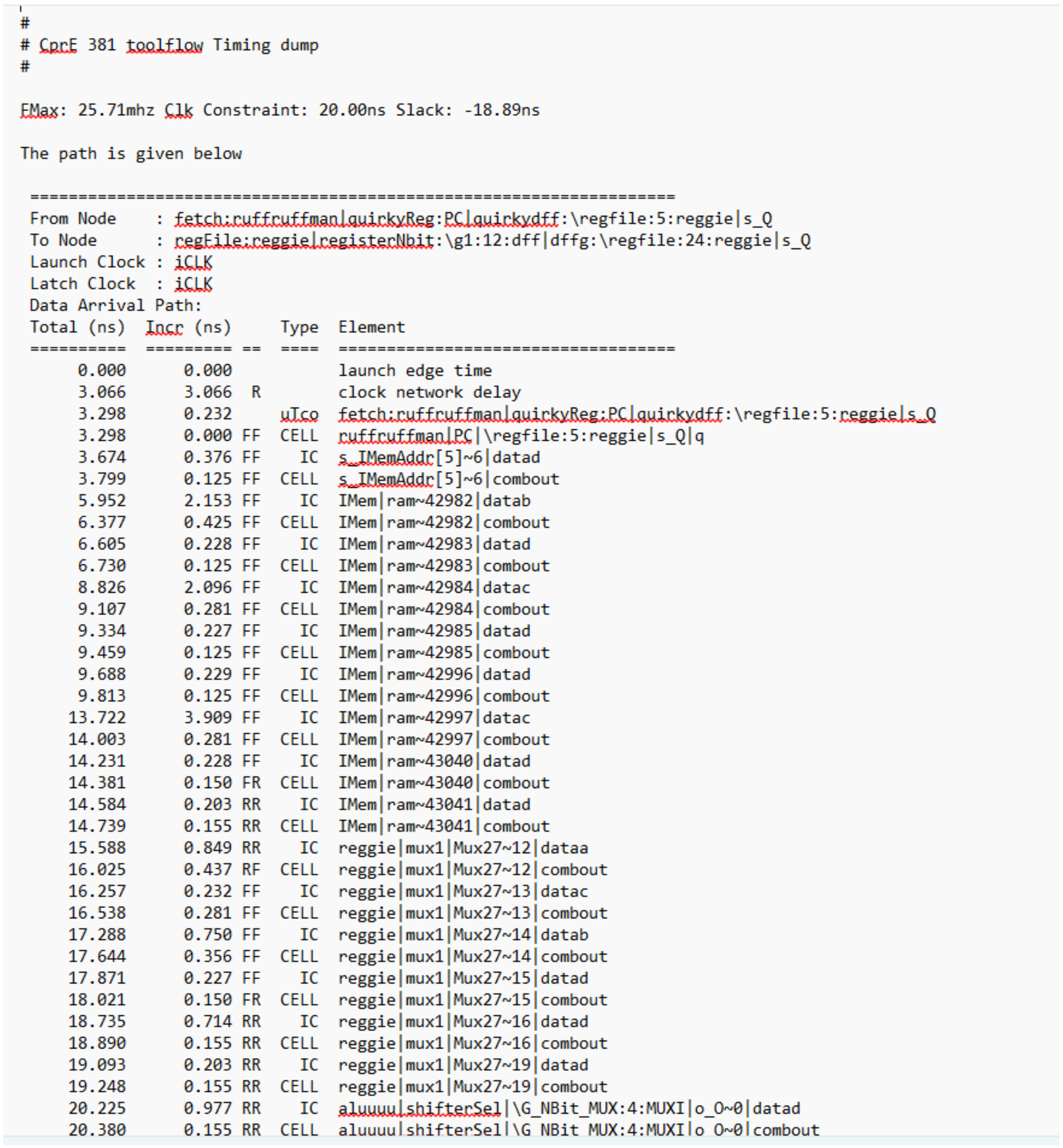
[Part 4] Report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematic from part 1. What components would you focus on to improve the frequency?

Critical max frequency: 25.71mhz

Critical path:

PC (Program Counter) → Instruction Memory (IMem) → various multiplexers and ALU shifter → Data Memory (DMem) → final mux and logic → Register File (write input).

One critical area to address is the clock network delay, which contributes significantly to the overall timing. Optimizing the clock distribution network can help reduce this delay and improve the overall speed. Additionally, reducing delays in the memory accesses, particularly in components like the Instruction Memory and Data Memory will directly impact the critical path. This can be achieved by optimizing the read/write paths and ensuring faster memory access times. Another area of improvement is the multiplexers and registers, as they add incremental delays throughout the path. Streamlining the design of these elements can reduce their contribution to the overall path delay. Finally, minimizing the interconnect delay between these components, particularly in areas where the signal travels through several stages of interconnects, would further reduce the total critical path time and allow the processor to run at a higher frequency.

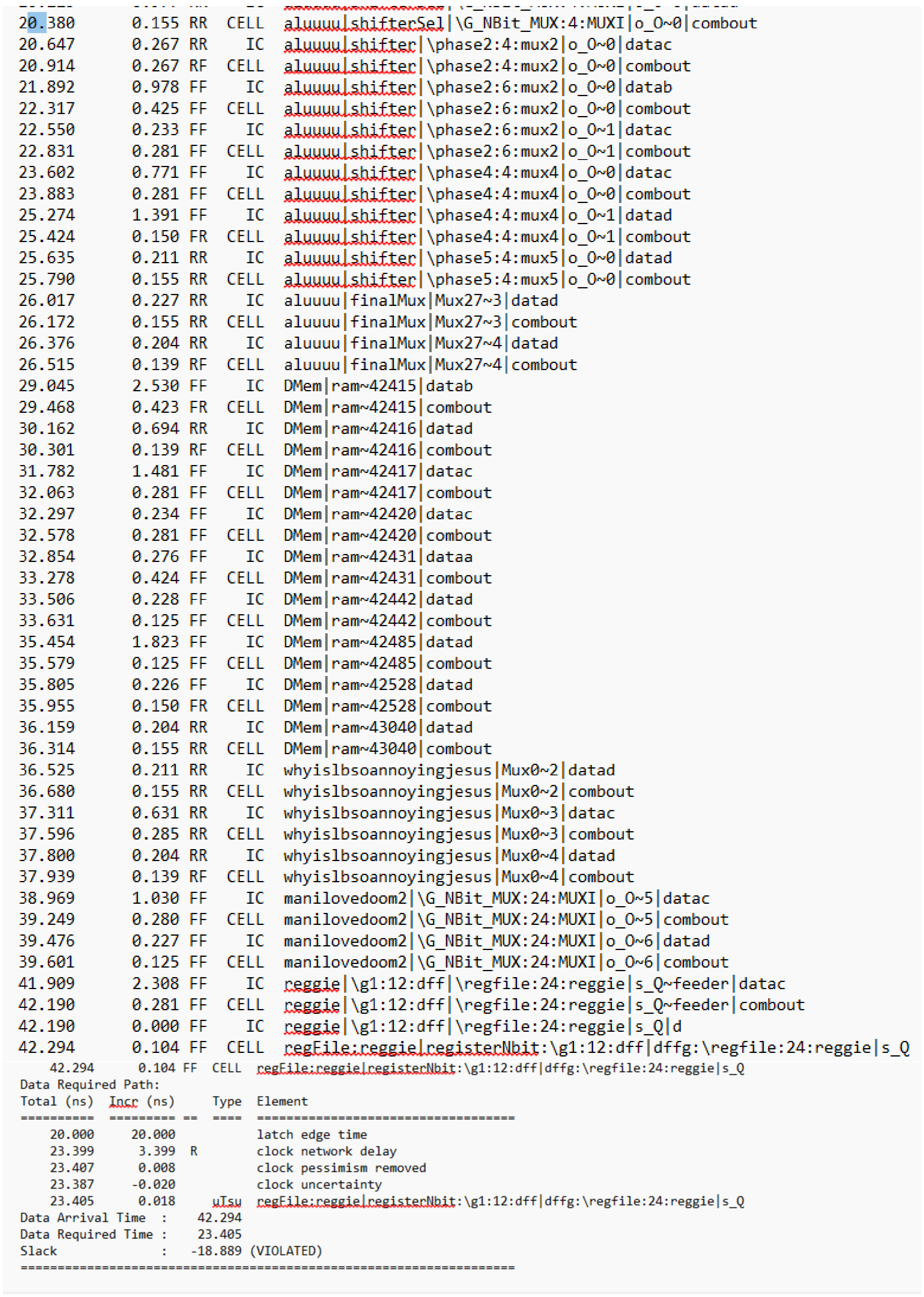


```
#
# CprE 381 toolflow Timing dump
#

FMax: 25.71mhz Clk Constraint: 20.00ns Slack: -18.89ns

The path is given below

 ===================================================================
 From Node    : fetch:ruffruffman|quirkyReg:PC|quirkydff:\regfile:5:reggie|s_Q
 To Node      : regFile:reggie|registerNbit:\g1:12:dff|dffg:\regfile:24:reggie|s_Q
 Launch Clock : iCLK
 Latch Clock  : iCLK
 Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ===================================
      0.000      0.000           launch edge time
      3.066      3.066  R        clock network delay
      3.298      0.232     uTco  fetch:ruffruffman|quirkyReg:PC|quirkydff:\regfile:5:reggie|s_Q
      3.298      0.000 FF  CELL  ruffruffman|PC|\regfile:5:reggie|s_Q|q
      3.674      0.376 FF    IC  s_IMemAddr[5]~6|datad
      3.799      0.125 FF  CELL  s_IMemAddr[5]~6|combout
      5.952      2.153 FF    IC  IMem|ram~42982|datab
      6.377      0.425 FF  CELL  IMem|ram~42982|combout
      6.605      0.228 FF    IC  IMem|ram~42983|datad
      6.730      0.125 FF  CELL  IMem|ram~42983|combout
      8.826      2.096 FF    IC  IMem|ram~42984|datac
      9.107      0.281 FF  CELL  IMem|ram~42984|combout
      9.334      0.227 FF    IC  IMem|ram~42985|datad
      9.459      0.125 FF  CELL  IMem|ram~42985|combout
      9.688      0.229 FF    IC  IMem|ram~42996|datad
      9.813      0.125 FF  CELL  IMem|ram~42996|combout
     13.722      3.909 FF    IC  IMem|ram~42997|datac
     14.003      0.281 FF  CELL  IMem|ram~42997|combout
     14.231      0.228 FF    IC  IMem|ram~43040|datad
     14.381      0.150 FR  CELL  IMem|ram~43040|combout
     14.584      0.203 RR    IC  IMem|ram~43041|datad
     14.739      0.155 RR  CELL  IMem|ram~43041|combout
     15.588      0.849 RR    IC  reggie|mux1|Mux27~12|dataa
     16.025      0.437 RF  CELL  reggie|mux1|Mux27~12|combout
     16.257      0.232 FF    IC  reggie|mux1|Mux27~13|datac
     16.538      0.281 FF  CELL  reggie|mux1|Mux27~13|combout
     17.288      0.750 FF    IC  reggie|mux1|Mux27~14|datab
     17.644      0.356 FF  CELL  reggie|mux1|Mux27~14|combout
     17.871      0.227 FF    IC  reggie|mux1|Mux27~15|datad
     18.021      0.150 FR  CELL  reggie|mux1|Mux27~15|combout
     18.735      0.714 RR    IC  reggie|mux1|Mux27~16|datad
     18.890      0.155 RR  CELL  reggie|mux1|Mux27~16|combout
     19.093      0.203 RR    IC  reggie|mux1|Mux27~19|datad
     19.248      0.155 RR  CELL  reggie|mux1|Mux27~19|combout
     20.225      0.977 RR    IC  aluuuu|shifterSel|\G_NBit_MUX:4:MUXI|o_O~0|datad
     20.380      0.155 RR  CELL  aluuuu|shifterSel|\G_NBit_MUX:4:MUXI|o_O~0|combout
```



```
20.380      0.155 RR  CELL  aluuuu|shifterSel|\G_NBit_MUX:4:MUXI|o_O~0|combout
20.647      0.267 RR    IC  aluuuu|shifter|\phase2:4:mux2|o_O~0|datac
20.914      0.267 RF  CELL  aluuuu|shifter|\phase2:4:mux2|o_O~0|combout
21.892      0.978 FF    IC  aluuuu|shifter|\phase2:6:mux2|o_O~0|datab
22.317      0.425 FF  CELL  aluuuu|shifter|\phase2:6:mux2|o_O~0|combout
22.550      0.233 FF    IC  aluuuu|shifter|\phase2:6:mux2|o_O~1|datac
22.831      0.281 FF  CELL  aluuuu|shifter|\phase2:6:mux2|o_O~1|combout
23.602      0.771 FF    IC  aluuuu|shifter|\phase4:4:mux4|o_O~0|datac
23.883      0.281 FF  CELL  aluuuu|shifter|\phase4:4:mux4|o_O~0|combout
25.274      1.391 FF    IC  aluuuu|shifter|\phase4:4:mux4|o_O~1|datad
25.424      0.150 FR  CELL  aluuuu|shifter|\phase4:4:mux4|o_O~1|combout
25.635      0.211 RR    IC  aluuuu|shifter|\phase5:4:mux5|o_O~0|datad
25.790      0.155 RR  CELL  aluuuu|shifter|\phase5:4:mux5|o_O~0|combout
26.017      0.227 RR    IC  aluuuu|finalMux|Mux27~3|datad
26.172      0.155 RR  CELL  aluuuu|finalMux|Mux27~3|combout
26.376      0.204 RR    IC  aluuuu|finalMux|Mux27~4|datad
26.515      0.139 RF  CELL  aluuuu|finalMux|Mux27~4|combout
29.045      2.530 FF    IC  DMem|ram~42415|datab
29.468      0.423 FR  CELL  DMem|ram~42415|combout
30.162      0.694 RR    IC  DMem|ram~42416|datad
30.301      0.139 RF  CELL  DMem|ram~42416|combout
31.782      1.481 FF    IC  DMem|ram~42417|datac
32.063      0.281 FF  CELL  DMem|ram~42417|combout
32.297      0.234 FF    IC  DMem|ram~42420|datac
32.578      0.281 FF  CELL  DMem|ram~42420|combout
32.854      0.276 FF    IC  DMem|ram~42431|dataa
33.278      0.424 FF  CELL  DMem|ram~42431|combout
33.506      0.228 FF    IC  DMem|ram~42442|datad
33.631      0.125 FF  CELL  DMem|ram~42442|combout
35.454      1.823 FF    IC  DMem|ram~42485|datad
35.579      0.125 FF  CELL  DMem|ram~42485|combout
35.805      0.226 FF    IC  DMem|ram~42528|datad
35.955      0.150 FR  CELL  DMem|ram~42528|combout
36.159      0.204 RR    IC  DMem|ram~43040|datad
36.314      0.155 RR  CELL  DMem|ram~43040|combout
36.525      0.211 RR    IC  whyislbsoannoyingjesus|Mux0~2|datad
36.680      0.155 RR  CELL  whyislbsoannoyingjesus|Mux0~2|combout
37.311      0.631 RR    IC  whyislbsoannoyingjesus|Mux0~3|datac
37.596      0.285 RR  CELL  whyislbsoannoyingjesus|Mux0~3|combout
37.800      0.204 RR    IC  whyislbsoannoyingjesus|Mux0~4|datad
37.939      0.139 RF  CELL  whyislbsoannoyingjesus|Mux0~4|combout
38.969      1.030 FF    IC  manilovedoom2|\G_NBit_MUX:24:MUXI|o_O~5|datac
39.249      0.280 FF  CELL  manilovedoom2|\G_NBit_MUX:24:MUXI|o_O~5|combout
39.476      0.227 FF    IC  manilovedoom2|\G_NBit_MUX:24:MUXI|o_O~6|datad
39.601      0.125 FF  CELL  manilovedoom2|\G_NBit_MUX:24:MUXI|o_O~6|combout
41.909      2.308 FF    IC  reggie|\g1:12:dff|\regfile:24:reggie|s_Q~feeder|datac
42.190      0.281 FF  CELL  reggie|\g1:12:dff|\regfile:24:reggie|s_Q~feeder|combout
42.190      0.000 FF    IC  reggie|\g1:12:dff|\regfile:24:reggie|s_Q|d
42.294      0.104 FF  CELL  regFile:reggie|registerNbit:\g1:12:dff|dffg:\regfile:24:reggie|s_Q
    42.294      0.104 FF  CELL  regFile:reggie|registerNbit:\g1:12:dff|dffg:\regfile:24:reggie|s_Q
Data Required Path:
Total (ns)  Incr (ns)     Type  Element
==========  ========= ==  ====  ==================================
    20.000     20.000           latch edge time
    23.399      3.399  R        clock network delay
    23.407      0.008           clock pessimism removed
    23.387     -0.020           clock uncertainty
    23.405      0.018     uTsu  regFile:reggie|registerNbit:\g1:12:dff|dffg:\regfile:24:reggie|s_Q
Data Arrival Time  :    42.294
Data Required Time :    23.405
Slack              :   -18.889 (VIOLATED)
==================================================================
```