CprE 381: Computer Organization and Assembly-Level Programming

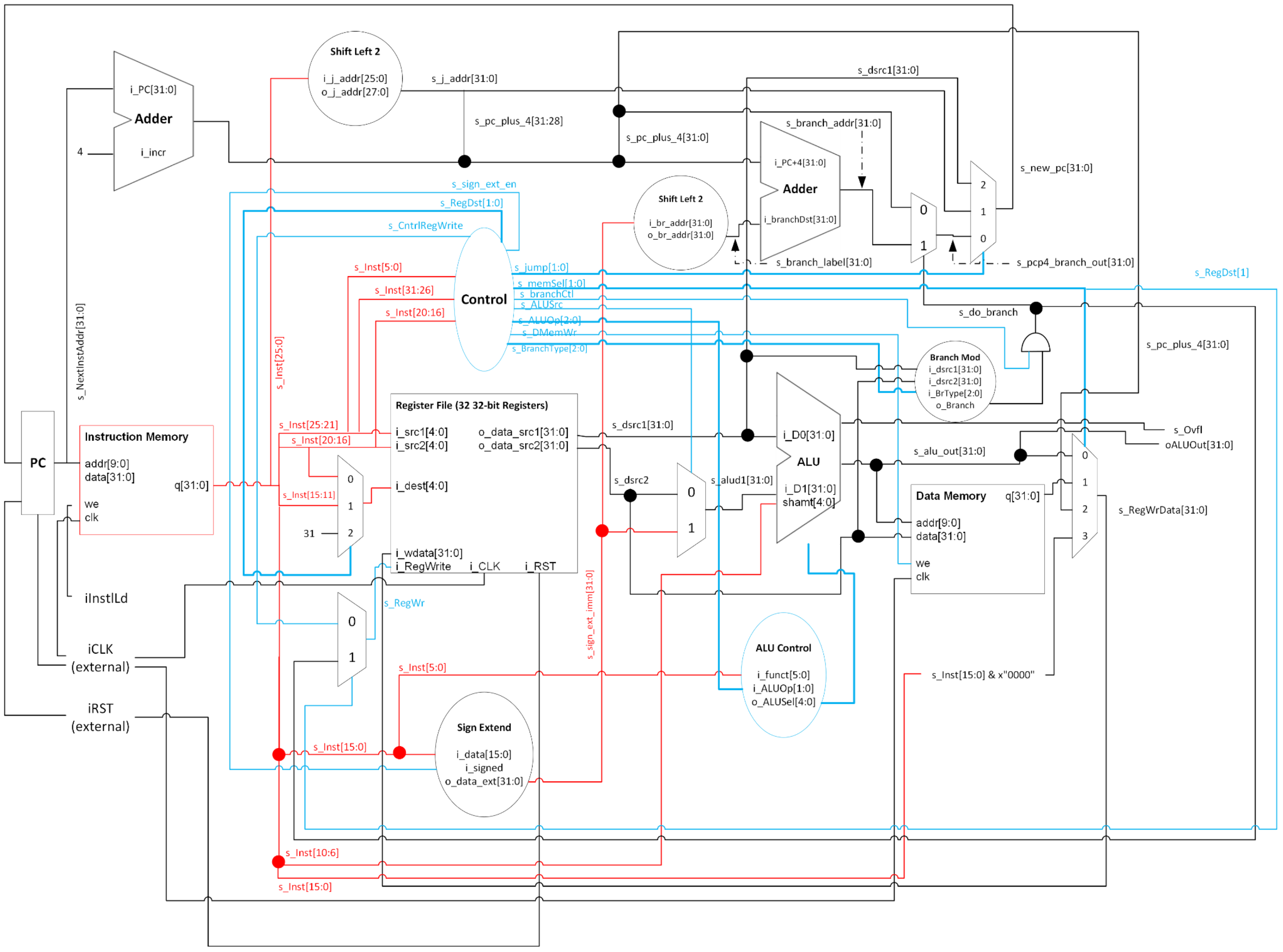
Project Part 1 Report

Team Members: Anthony Manschula
Henry Shires

Project Teams Group #: TermProj_3_02

Please see this folder for full-resolution waveform images.

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

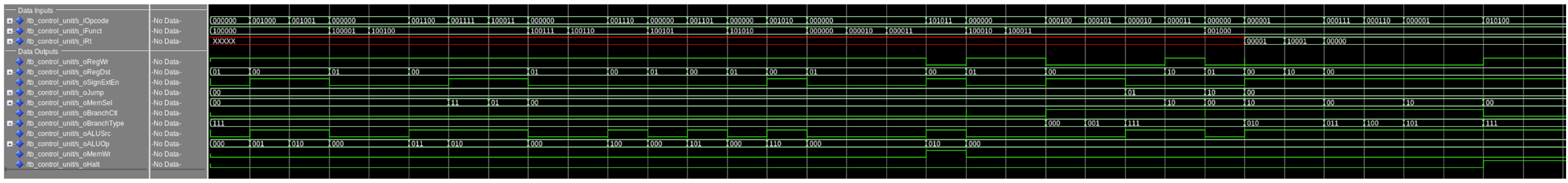


Our MIPS Single-Cycle Processor design

[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 result should be an *N*M* table where each row corresponds to the output of the control logic module for a given instruction.

Spreadsheet: [Processor Control Signals - TermProj_3_02]()

[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).

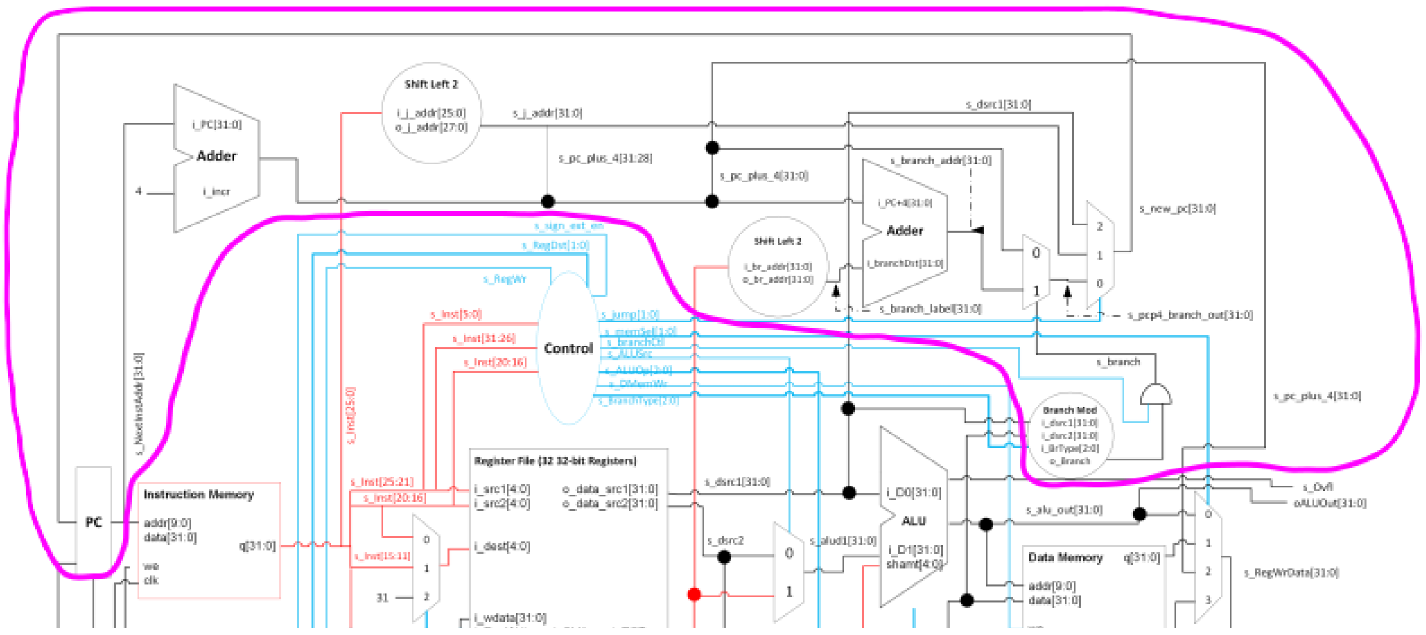

Data Inputs
/tb_control_unit/s_iOpcode
/tb_control_unit/s_iFunct
/tb_control_unit/s_iRt
Data Outputs
/tb_control_unit/s_oRegWr
/tb_control_unit/s_oRegDst
/tb_control_unit/s_oSignExtEn
/tb_control_unit/s_oJump
/tb_control_unit/s_oMemSel
/tb_control_unit/s_oBranchCtl
/tb_control_unit/s_oBranchType
/tb_control_unit/s_oALUSrc
/tb_control_unit/s_oALUOp
/tb_control_unit/s_oMemWr
/tb_control_unit/s_oHalt
-No Data-

As programmed in the test bench, the QuestaSim waveform outputs all opcodes in the order listed in our control signals spreadsheet. The spreadsheet can then be easily read from top to bottom and compared with the waveform from left to right. Following this process, each control signal output matches its intended values for our control unit implementation. Note: The `s_iRt` signal is undefined until it is set in the test bench by the `bgez` instruction. Then its values appear as intended for the remaining branch instructions that require it.

[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.

The instruction fetch logic must support several cases: Sequential execution (standard non-control flow instructions), conditionally setting the PC to an instruction address given by an immediate (branching) and occasionally setting $ra to the next instruction address in the case of *b*al*, and unconditionally setting the PC to the value of an immediate (jump) and setting $ra to the next instruction address in the case of a *jal*.

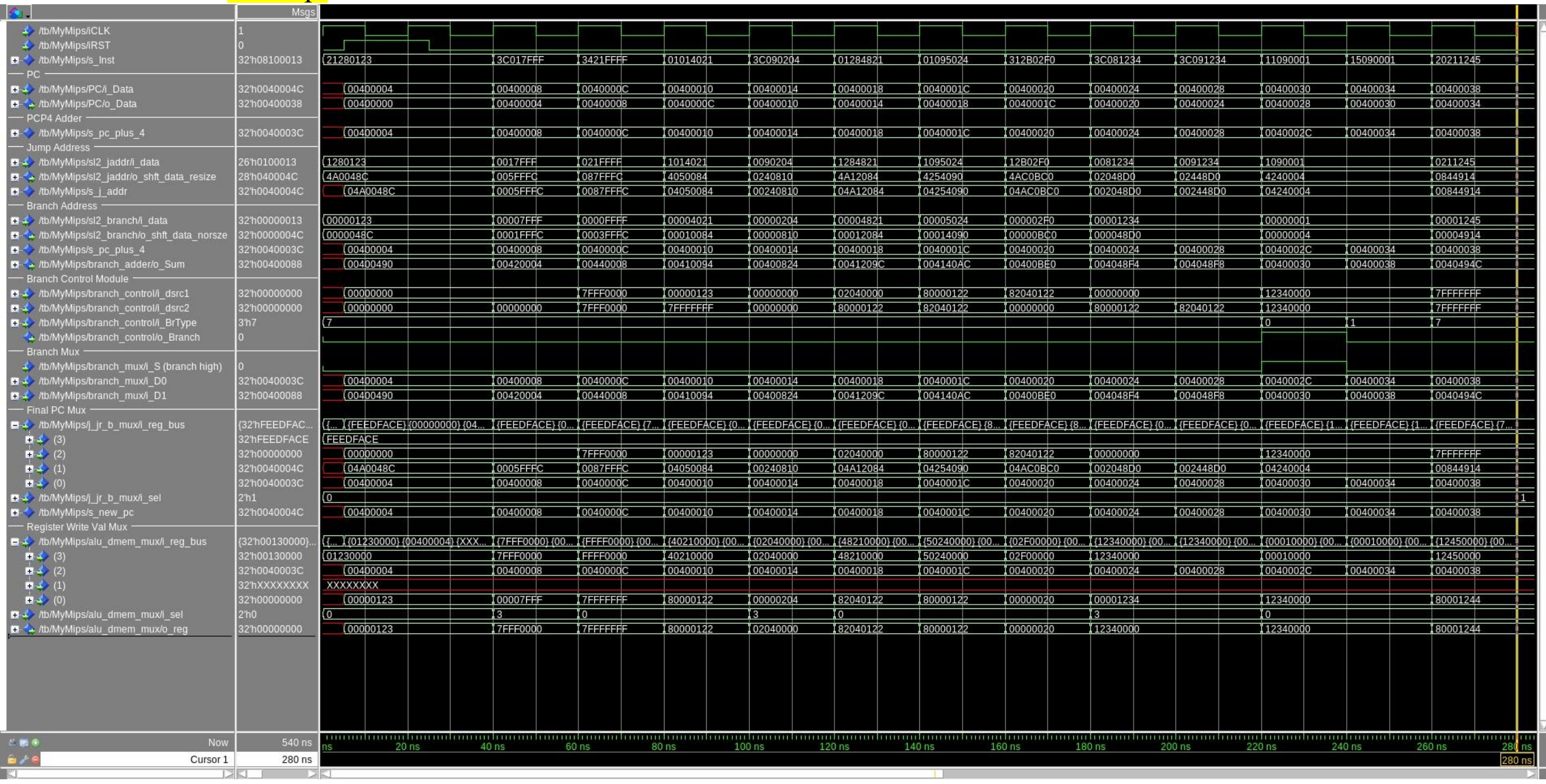
[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What are additional control signals needed?



Additional control signals we elected to include in our design:

- `s_jump`: Controls whether the PC takes PC+4 (or branch address), a jump address specified by an immediate, or an address value specified in a register from `jr`.
- We also designed a standalone branch control module, which takes the `s_BranchType` signal generated from the control unit and determines whether a branch should be taken based on input data and the type of control flow being performed.

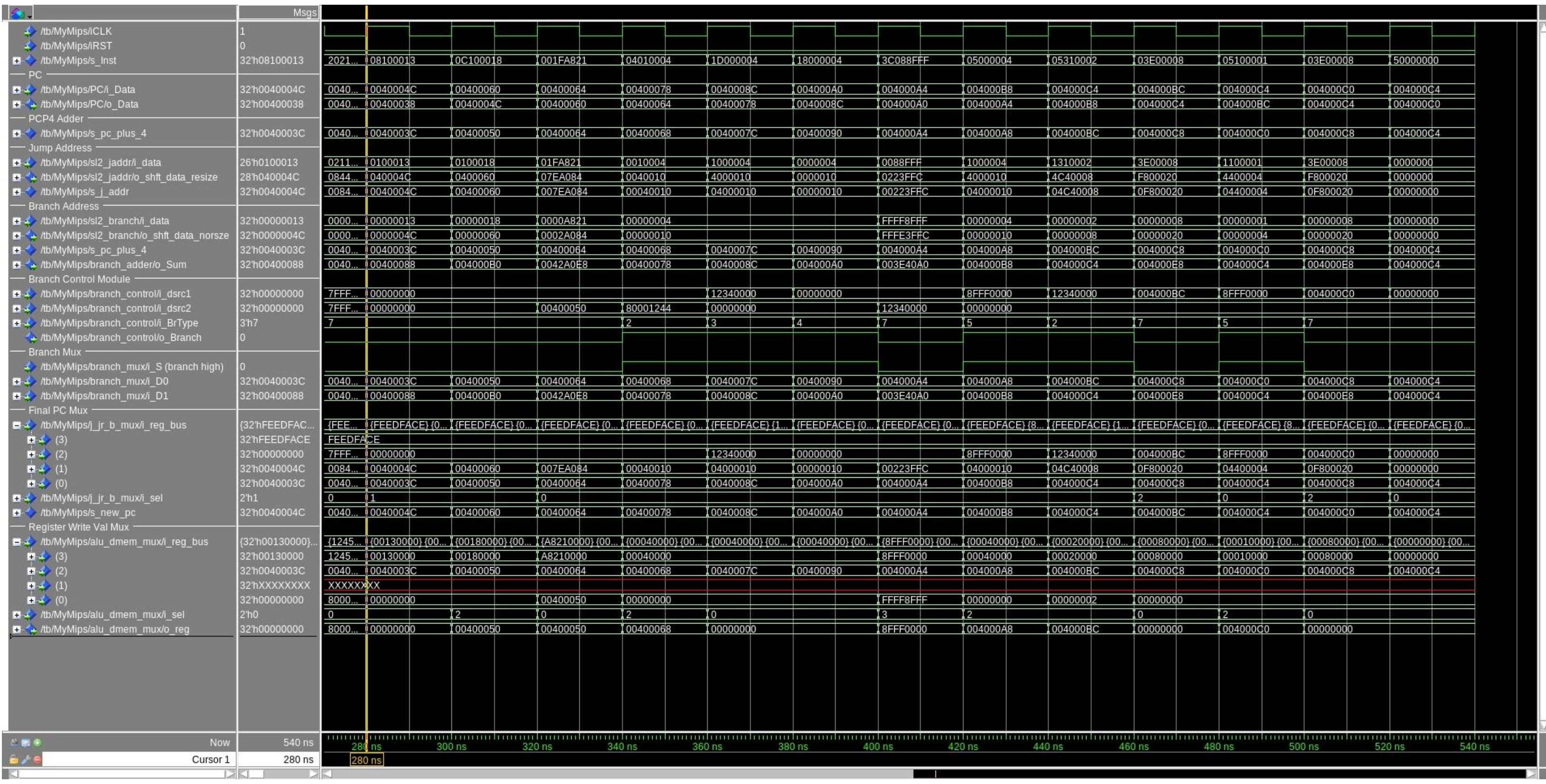
[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use QuestaSim to test your design thoroughly to ensure it works as expected. Describe how the execution of the control flow possibilities corresponds to the QuestaSim waveforms in your writeup.



```
                                                Msgs
/tb/MyMips/iCLK                                 1
/tb/MyMips/iRST                                 0
/tb/MyMips/s_Inst                               32'h08100013
PC
/tb/MyMips/PC/i_Data                            32'h0040004C
/tb/MyMips/PC/o_Data                            32'h00400038
PCP4 Adder
/tb/MyMips/s_pc_plus_4                          32'h0040003C
Jump Address
/tb/MyMips/sl2_jaddr/i_data                     26'h0100013
/tb/MyMips/sl2_jaddr/o_shft_data_resize         28'h040004C
/tb/MyMips/s_j_addr                             32'h0040004C
Branch Address
/tb/MyMips/sl2_branch/i_data                    32'h00000013
/tb/MyMips/sl2_branch/o_shft_data_norsze        32'h0000004C
/tb/MyMips/s_pc_plus_4                          32'h0040003C
/tb/MyMips/branch_adder/o_Sum                   32'h00400088
Branch Control Module
/tb/MyMips/branch_control/i_dsrc1               32'h00000000
/tb/MyMips/branch_control/i_dsrc2               32'h00000000
/tb/MyMips/branch_control/i_BrType              3'h7
/tb/MyMips/branch_control/o_Branch              0
Branch Mux
/tb/MyMips/branch_mux/i_S (branch high)         0
/tb/MyMips/branch_mux/i_D0                      32'h0040003C
/tb/MyMips/branch_mux/i_D1                      32'h00400088
Final PC Mux
/tb/MyMips/j_jr_b_mux/i_reg_bus                 {32'hFEEDFAC...
  (3)                                           32'hFEEDFACE
  (2)                                           32'h00000000
  (1)                                           32'h0040004C
  (0)                                           32'h0040003C
/tb/MyMips/j_jr_b_mux/i_sel                     2'h1
/tb/MyMips/s_new_pc                             32'h0040004C
Register Write Val Mux
/tb/MyMips/alu_dmem_mux/i_reg_bus               {32'h00130000}...
  (3)                                           32'h00130000
  (2)                                           32'h0040003C
  (1)                                           32'hXXXXXXXX
  (0)                                           32'h00000000
/tb/MyMips/alu_dmem_mux/i_sel                   2'h0
/tb/MyMips/alu_dmem_mux/o_reg                   32'h00000000

Now 540 ns    Cursor 1 280 ns

s_Inst:        21280123 3C017FFF 3421FFFF 01014021 3C090204 01284821 01095024 312B02F0 3C081234 3C091234 11090001 15090001 20211245
PC/i_Data:     00400004 00400008 0040000C 00400010 00400014 00400018 0040001C 00400020 00400024 00400028 00400030 00400034 00400038
PC/o_Data:     00400000 00400004 00400008 0040000C 00400010 00400014 00400018 0040001C 00400020 00400024 00400028 00400030 00400034
s_pc_plus_4:   00400004 00400008 0040000C 00400010 00400014 00400018 0040001C 00400020 00400024 00400028 0040002C 00400034 00400038
jaddr/i_data:  1280123 0017FFF 021FFFF 1014021 0090204 1284821 1095024 12B02F0 0081234 0091234 1090001 0211245
jaddr/o_shft:  4A0048C 005FFFC 087FFFC 4050084 0240810 4A12084 4254090 4AC0BC0 02048D0 02448D0 4240004 0844914
s_j_addr:      04A0048C 0005FFFC 0087FFFC 04050084 00240810 04A12084 04254090 04AC0BC0 002048D0 002448D0 04240004 00844914
branch/i_data: 00000123 00007FFF 0000FFFF 00004021 00000204 00004821 00005024 000002F0 00001234 00000001 00001245
branch/o_shft: 0000048C 0001FFFC 0003FFFC 00010084 00000810 00012084 00014090 00000BC0 000048D0 00000004 00004914
s_pc_plus_4:   00400004 00400008 0040000C 00400010 00400014 00400018 0040001C 00400020 00400024 00400028 0040002C 00400034 00400038
o_Sum:         00400490 00420004 00440008 00410094 00400824 0041209C 004140AC 00400BE0 004048F4 004048F8 00400030 00400038 0040494C
i_dsrc1:       00000000 7FFF0000 00000123 00000000 02040000 80000122 82040122 00000000 12340000 7FFFFFFF
i_dsrc2:       00000000 00000000 7FFF0000 7FFFFFFF 00000000 80000122 82040122 00000000 80000122 82040122 12340000 7FFFFFFF
i_BrType:      7 0 1 7
i_D0:          00400004 00400008 0040000C 00400010 00400014 00400018 0040001C 00400020 00400024 00400028 0040002C 00400034 00400038
i_D1:          00400490 00420004 00440008 00410094 00400824 0041209C 004140AC 00400BE0 004048F4 004048F8 00400030 00400038 0040494C
(3):           FEEDFACE
(2):           00000000 7FFF0000 00000123 00000000 02040000 80000122 82040122 00000000 12340000 7FFFFFFF
(1):           04A0048C 0005FFFC 0087FFFC 04050084 00240810 04A12084 04254090 04AC0BC0 002048D0 002448D0 04240004 00844914
(0):           00400004 00400008 0040000C 00400010 00400014 00400018 0040001C 00400020 00400024 00400028 00400030 00400034 00400038
i_sel:         0 1
s_new_pc:      00400004 00400008 0040000C 00400010 00400014 00400018 0040001C 00400020 00400024 00400028 00400030 00400034 00400038
(3):           01230000 7FFF0000 FFFF0000 40210000 02040000 48210000 50240000 02F00000 12340000 00010000 12450000
(2):           00400004 00400008 0040000C 00400010 00400014 00400018 0040001C 00400020 00400024 00400028 0040002C 00400034 00400038
(1):           XXXXXXXX
(0):           00000123 00007FFF 7FFFFFFF 80000122 00000204 82040122 80000122 00000020 00001234 12340000 80001244
i_sel:         0 3 0 3 0 3 0
o_reg:         00000123 7FFF0000 7FFFFFFF 80000122 02040000 82040122 80000122 00000020 12340000 12340000 80001244
```



```
                                                Msgs
/tb/MyMips/iCLK                                 1
/tb/MyMips/iRST                                 0
/tb/MyMips/s_Inst                               32'h08100013
PC
/tb/MyMips/PC/i_Data                            32'h0040004C
/tb/MyMips/PC/o_Data                            32'h00400038
PCP4 Adder
/tb/MyMips/s_pc_plus_4                          32'h0040003C
Jump Address
/tb/MyMips/sl2_jaddr/i_data                     26'h0100013
/tb/MyMips/sl2_jaddr/o_shft_data_resize         28'h040004C
/tb/MyMips/s_j_addr                             32'h0040004C
Branch Address
/tb/MyMips/sl2_branch/i_data                    32'h00000013
/tb/MyMips/sl2_branch/o_shft_data_norsze        32'h0000004C
/tb/MyMips/s_pc_plus_4                          32'h0040003C
/tb/MyMips/branch_adder/o_Sum                   32'h00400088
Branch Control Module
/tb/MyMips/branch_control/i_dsrc1               32'h00000000
/tb/MyMips/branch_control/i_dsrc2               32'h00000000
/tb/MyMips/branch_control/i_BrType              3'h7
/tb/MyMips/branch_control/o_Branch              0
Branch Mux
/tb/MyMips/branch_mux/i_S (branch high)         0
/tb/MyMips/branch_mux/i_D0                      32'h0040003C
/tb/MyMips/branch_mux/i_D1                      32'h00400088
Final PC Mux
/tb/MyMips/j_jr_b_mux/i_reg_bus                 {32'hFEEDFAC...
  (3)                                           32'hFEEDFACE
  (2)                                           32'h00000000
  (1)                                           32'h0040004C
  (0)                                           32'h0040003C
/tb/MyMips/j_jr_b_mux/i_sel                     2'h1
/tb/MyMips/s_new_pc                             32'h0040004C
Register Write Val Mux
/tb/MyMips/alu_dmem_mux/i_reg_bus               {32'h00130000}...
  (3)                                           32'h00130000
  (2)                                           32'h0040003C
  (1)                                           32'hXXXXXXXX
  (0)                                           32'h00000000
/tb/MyMips/alu_dmem_mux/i_sel                   2'h0
/tb/MyMips/alu_dmem_mux/o_reg                   32'h00000000

Now 540 ns    Cursor 1 280 ns

s_Inst:        2021... 08100013 0C100018 001FA821 04010004 1D000004 18000004 3C088FFF 05000004 05310002 03E00008 05100001 03E00008 50000000
PC/i_Data:     0040... 0040004C 00400060 00400064 00400078 0040008C 004000A0 004000A4 004000B8 004000C4 004000BC 004000C4 004000C0 004000C4
PC/o_Data:     0040... 00400038 0040004C 00400060 00400064 00400078 0040008C 004000A0 004000A4 004000B8 004000C4 004000BC 004000C4 004000C0
s_pc_plus_4:   0040... 0040003C 00400050 00400064 00400068 0040007C 00400090 004000A4 004000A8 004000BC 004000C8 004000C0 004000C8 004000C4
jaddr/i_data:  0211... 0100013 0100018 01FA821 0010004 1000004 0000004 0088FFF 1000004 1310002 3E00008 1100001 3E00008 0000000
jaddr/o_shft:  0844... 040004C 0400060 07EA084 0400010 4000010 0000010 0223FFC 4000010 4C40008 F800020 4400004 F800020 0000000
s_j_addr:      0084... 0040004C 00400060 007EA084 00040010 04000010 00000010 00223FFC 04000010 04C40008 0F800020 04400004 0F800020 00000000
branch/i_data: 0000... 00000013 00000018 0000A821 00000004 FFFF8FFF 00000004 00000002 00000008 00000001 00000008 00000000
branch/o_shft: 0000... 0000004C 00000060 0002A084 00000010 FFFE3FFC 00000010 00000008 00000020 00000004 00000020 00000000
s_pc_plus_4:   0040... 0040003C 00400050 00400064 00400068 0040007C 00400090 004000A4 004000A8 004000BC 004000C8 004000C0 004000C8 004000C4
o_Sum:         0040... 00400088 004000B0 0042A0E8 00400078 0040008C 004000A0 003E40A0 004000B8 004000C4 004000E8 004000C4 004000E8 004000C4
i_dsrc1:       7FFF... 00000000 12340000 00000000 8FFF0000 12340000 004000BC 8FFF0000 004000C0 00000000
i_dsrc2:       7FFF... 00000000 00400050 80001244 00000000 12340000 00000000
i_BrType:      7 2 3 4 7 5 2 7 5 7
i_D0:          0040... 0040003C 00400050 00400064 00400068 0040007C 00400090 004000A4 004000A8 004000BC 004000C8 004000C0 004000C8 004000C4
i_D1:          0040... 00400088 004000B0 0042A0E8 00400078 0040008C 004000A0 003E40A0 004000B8 004000C4 004000E8 004000C4 004000E8 004000C4
(3):           FEEDFACE
(2):           7FFF... 00000000 12340000 00000000 8FFF0000 12340000 004000BC 8FFF0000 004000C0 00000000
(1):           0084... 0040004C 00400060 007EA084 00040010 04000010 00000010 00223FFC 04000010 04C40008 0F800020 04400004 0F800020 00000000
(0):           0040... 0040003C 00400050 00400064 00400078 0040008C 004000A0 004000A4 004000B8 004000C4 004000C8 004000C4 004000C8 004000C4
i_sel:         0 1 0 2 0 2 0
s_new_pc:      0040... 0040004C 00400060 00400064 00400078 0040008C 004000A0 004000A4 004000B8 004000C4 004000BC 004000C4 004000C0 004000C4
(3):           1245... 00130000 00180000 A8210000 00040000 8FFF0000 00040000 00020000 00080000 00010000 00080000 00000000
(2):           0040... 0040003C 00400050 00400064 00400068 0040007C 00400090 004000A4 004000A8 004000BC 004000C8 004000C0 004000C8 004000C4
(1):           XXXXXXXX
(0):           8000... 00000000 00400050 00000000 FFFF8FFF 00000000 00000002 00000000
i_sel:         0 2 0 2 0 3 2 0 2 0
o_reg:         8000... 00000000 00400050 00400050 00400068 00000000 8FFF0000 004000A8 004000BC 00000000 004000C0 00000000
```

- Sequential execution
    - The first 10 cycles are executing simple add, or, and, and lui instructions, as evidenced by s_Inst. It can be seen that the PC output data increments by 0x4 after the execution completes.
- Branching
    - The 11th cycle fetches a branch instruction. We can see that, although PC + 4 calculates to 0x0040002C, the input of the PC register is 0x00400030, which is the address of the branch. Moving down, we can ignore jump address since we are not jumping. We can see that the PC-relative branch address is calculated to be 0x4 (we are only branching over one instruction), and it is added PC+4 in order to obtain the final branch address. The branch control module compares the two inputs based on the i_BrType signal (0 =, 1 !=. 2 >= 0, 3 >0, 4 <= 0, 5 < 0) generated by the control module. In this case we are testing equality since BrType is 0, and since i_dsrc1 and i_dsrc2 are equal, the branch output is asserted. This same process applies for any of the other branch instructions. The output of the branch module controls the select line for the branch mux. Since it is 1, we choose to take the value of the branch address (calculated in section “Branch address”) instead of PC+4. We then move on to the final step, which is choosing the data to send back to the PC. The select line of this multiplexer also comes from the control unit, and in this case, a select value of 2b00 will result in the data from the branch mux (in the case of a branch this will be the branch address) being sent to the PC input via s_new_pc. There is also another case, which are the *b*al* instructions. These must set $31 to the value of PC+4 when the branch is taken. If we jump to instruction 22 in the program (a BGEZAL), we can see that in addition to the aforementioned steps, we also set the select signal for the register write value mux to 2b10, which allows PC+4 to pass through to the write data port of the register file. A select signal also sets the destination register to a hardcoded decimal 31 (not shown).
- Jumping
    - The 14th instruction is a jump (at 280ns). We see that the input of the PC is 0x0040004C, which indicates that the jump address was properly calculated. Indeed, in the “Jump address” section, we can see that the input data from the instruction is properly shifted left two bits and resized. Skipping down to the “Final PC Mux” section, the calculated branch address is seen on the input bus, and the select signal coming from the control module is 1, which indicates we should send the jump address to the PC input. In the 15th cycle (300ns), we perform a *jal*. All of the relevant information just mentioned still applies, and we can see that we end up with a proper jump address calculated and sent to the PC. However, since we are linking, we also need to write PC+4 to $31. We can see that the value of the register write value mux select signal is 2, which indicates we should pass PC+4 to the reg write port. The last jump case is *jr*, which is straightforward. The 25th instruction (500ns) is performing a *jr $31* instruction to return from a *jal.* We simply set the select signal for

the Final PC Mux to 2b10, and this passes through the value of data source 1 from the register file.

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

Logical shifts do not account for the value of the bit at either extreme of the signal (i.e., bit 0 or bit 31) when shifting. They simply shift bits left or right and insert 0s at the opposite end of the direction of the shift. Arithmetic right shift preserves the sign bit's value by shifting the sign bit's value. MIPS does not have a `sla` instruction because the LSB doesn't represent much useful information in the context of an arithmetic shift.

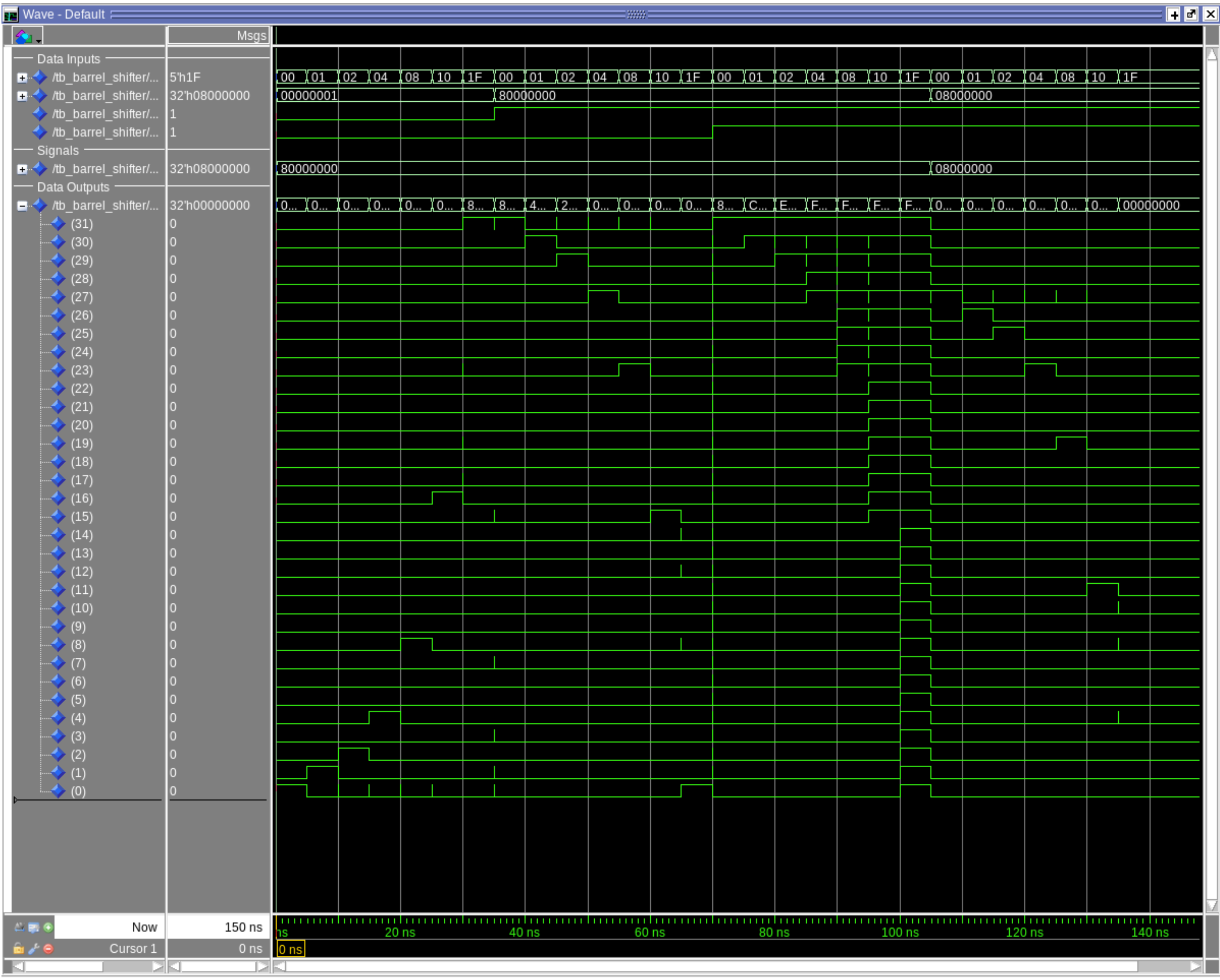
[Part 2 (c.i.2)] In your write-up, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

In logical shifting, bits left over after shifting are always set to 0. For the left shift, this means the bits to the right of the shifted bits are set to 0, and vice versa for the right shift. In arithmetic shifting, instead of replacing the leftover bits with 0s, they are replaced with the sign of the original binary number, in this case, using 2s complement, the most significant bit's value (MSB). Our VHDL implementation uses 2:1 multiplexers to manipulate each bit of the 32-bit input vector `i_shsrc[31:0]`. For each bit of the 5-bit shift amount vector `i_shamt[4:0]`, a row of 32 2-1 MUXs with the corresponding shift amount bit as a selection for all is used to shift the bits X digits left or right. Suppose X is greater than the remaining bits available to shift. In that case, the MUX will use a 0 or 1 to assert on the shifted bit instead, with a 0 for logical operations and a 1 for arithmetic operations. The signal propagates through the X = 16-bit row, 8-bit row, 4-bit row, 2-bit row, and 1-bit row, a total of five rows, each matching a shift amount bit (1 or a 0). The outputted vector with the correct shift applied should appear at the end of the multiplexer array.

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

Since the right barrel shift output bits directly manipulate `i_shsrc` in order, `i_shsrc` can be split up into bits 0-31 and inputted into the first row of 2-1 MUXs from left to right, shifting right based on the value of `i_shamt`. With left shifting, the current vector direction will result in an incorrect shift. While seemingly complicated, the right barrel shift can be easily modified to reverse the vector bits using an additional set of MUXs, where bit 31 of the original input is now located at bit 0 and vice versa, continuing until the entire vector has been flipped. This row of MUXs utilizes a separate select `i_shdir` where 0 corresponds to the left shift and 1 to the right shift. Each bit is either left alone or reversed with its opposite position (0 and 31, 1 and 30, etc.) based on the select value. Then the exact same right barrel shifter can be used with the reversed input for a left shift. The reverse vector process is repeated to undo the reversal of bits at the end of the barrel shifter to ensure the original input with the correct shift is outputted.

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



In our QuestaSim test bench, four common cases are tested from left to right. From 0-35, ns is `sll` with *`i_shsrc`* = 0x00000001, 35-70 ns `srl` with *`i_shsrc`* = 0x80000000, 70-105ns `sra` with *`i_shsrc`* = 0x80000000, and 105-140 ns `sra` with *`i_shsrc`* = 0x08000000. These specific input values help visualize the shift that occurs one after another. For cases 3 and 4, case 3 is run with the MSB of the input = 1 and case 4 with the MSB = 0 to ensure the arithmetic operation follows the sign of the vector. In all four cases, the '1' value on the input is seen shifted 1, 2, 4, 8, 16, and 31 bits as intended. Case

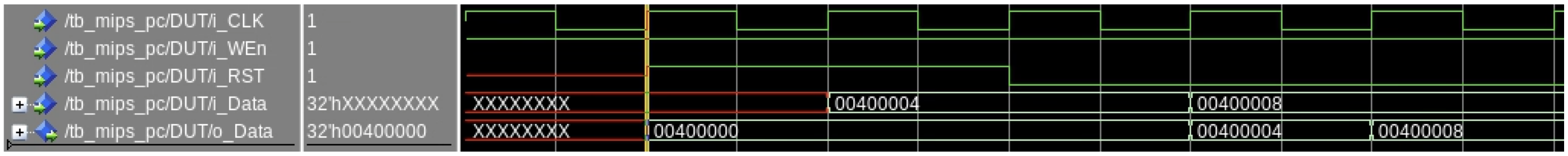
1 shifts it from LSB to MSB (left), case 2 from MSB to LSB (right), case 3 from MSB to LSB while tracing the sign of 1, and case 4 from MSB to LSB while tracing the sign of 0.

[Part 2 (c.ii.1)] In your write-up, 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.

- MIPS PC
    - Design: We needed a component that was similar in behavior to a register but with a MIPS-compliant reset value. We followed the n-bit register design closely and included a reset value of 0x00400000.
- Shift Left 2
    - We needed a shift left 2 unit that could either take the form of an n-bit and n+2-bit input/output or an n-bit input/output. The choice was made to use a combination of dataflow and behavioral VHDL in order to create something that was reusable for both the jump address shift, as well as the branch address shift.
- Sign extender
    - We wanted an approach that was straightforward, so we opted to use a select statement in combination with some dataflow-style VHDL to map the input signal to the lower part of the output, and fill the upper part of the output with either 0's or the sign bit, depending the value of the sign extension enable control.
- ALU Control
    - The ALU control module needs to tell the ALU which arithmetic operation to perform based on the *funct* field of the instruction, as well as the ALUOp control output. The ALUSel output is selected for an R-type instruction when ALUOp is '000' and a matching *funct* is found on the input. The value of ALUOp may also be the same for some non-R-type instructions, such as `add` and `addi`.
- Branch Module
    - We quickly realized that the "zero" output of the ALU would not have been sufficient to support the required branch instructions and that trying to shoe-horn support into the ALU would have been messy and error-prone. We instead opted to add an entirely new module that outputs a "do branch" signal based on the two register outputs and a new control signal called BranchType, which is generated by the control and is unique to each type of branch instruction.

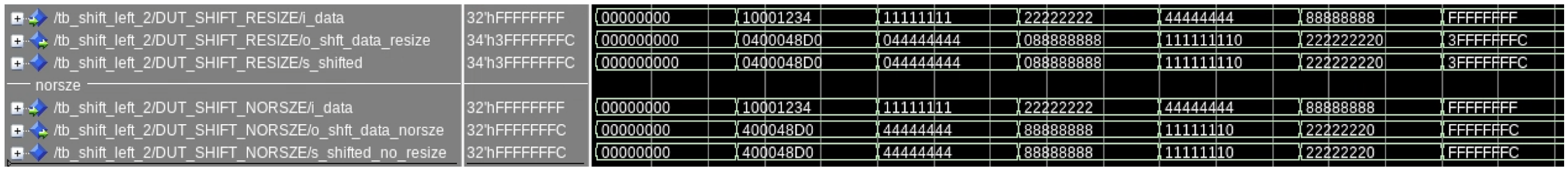
[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

MIPS PC


/tb_mips_pc/DUT/i_CLK 1
/tb_mips_pc/DUT/i_WEn 1
/tb_mips_pc/DUT/i_RST 1
/tb_mips_pc/DUT/i_Data 32'hXXXXXXXX
/tb_mips_pc/DUT/o_Data 32'h00400000
XXXXXXXX 00400004 00400008
XXXXXXXX 00400000 00400004 00400008

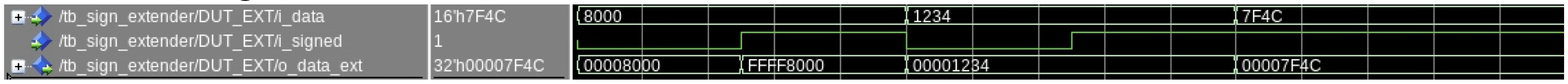
- At its core, the MIPS PC is just a register with a custom reset value, and we've already validated the register design in previous labs. Therefore, we tested the reset functionality by making sure reset assigned the correct value, and that setting the input while reset is asserted did not affect the output in the first two clock cycles. From there, the register operates as expected with the reset signal not asserted.

Shift Left 2


/tb_shift_left_2/DUT_SHIFT_RESIZE/i_data 32'hFFFFFFFF
/tb_shift_left_2/DUT_SHIFT_RESIZE/o_shft_data_resize 34'h3FFFFFFFC
/tb_shift_left_2/DUT_SHIFT_RESIZE/s_shifted 34'h3FFFFFFFC
norsze
/tb_shift_left_2/DUT_SHIFT_NORSZE/i_data 32'hFFFFFFFF
/tb_shift_left_2/DUT_SHIFT_NORSZE/o_shft_data_norsze 32'hFFFFFFFC
/tb_shift_left_2/DUT_SHIFT_NORSZE/s_shifted_no_resize 32'hFFFFFFFC
00000000 10001234 11111111 22222222 44444444 88888888 FFFFFFFF
000000000 0400048D0 044444444 088888888 111111110 222222220 3FFFFFFFC
000000000 0400048D0 044444444 088888888 111111110 222222220 3FFFFFFFC
00000000 10001234 11111111 22222222 44444444 88888888 FFFFFFFF
00000000 400048D0 44444444 88888888 11111110 22222220 FFFFFFFC
00000000 400048D0 44444444 88888888 11111110 22222220 FFFFFFFC

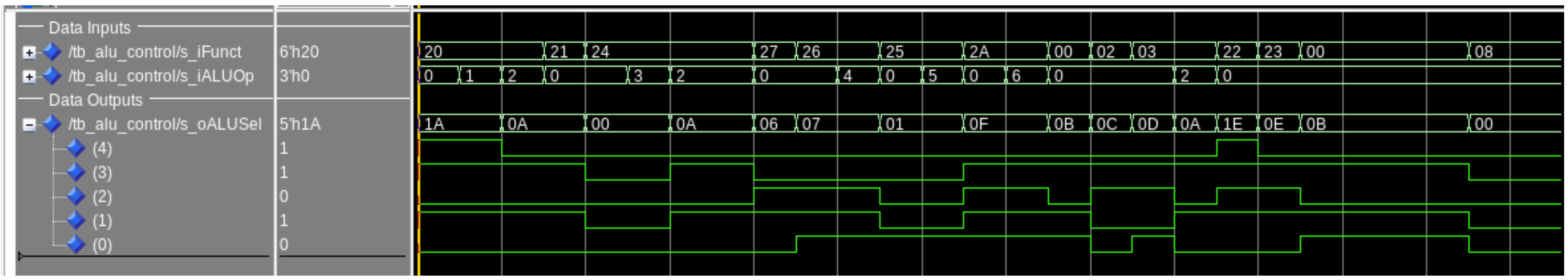
- The shift left 2 module shifts the input left by two bits, and optionally increases the output width by two bits as well. Width increase is important because it is used in the jump address calculation part of the datapath. As it can be seen in the waveform, the input bits are shifted left two positions and the vacated positions are filled in with zeroes. Additionally, it can be seen that the output is resized correctly in the case of the top signal, and that it works properly for all 0's and all 1's input.

Sign Extender


/tb_sign_extender/DUT_EXT/i_data 16'h7F4C
/tb_sign_extender/DUT_EXT/i_signed 1
/tb_sign_extender/DUT_EXT/o_data_ext 32'h00007F4C
8000 1234 7F4C
00008000 FFFF8000 00001234 00007F4C

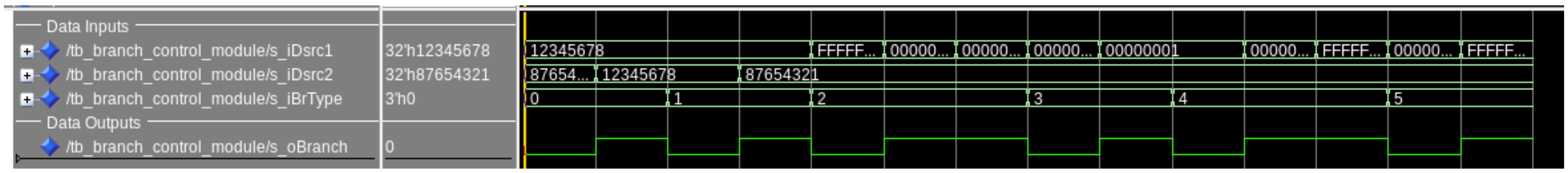
- The sign extender simply maps the 16-bit input signal into the lower 16 bits of the output, and then concatenates either 0's or the value of input[15] onto output[31:16] when i_signed is 0 or 1, respectively. Due to the nature of this implementation, it will either work, or it will not work. There won't be some input values that it works for and some that it doesn't, so extensive testing would be redundant. It can be seen that for s_signed = 0, 0's are concatenated regardless of the value of input[15]. When s_signed = 1, the value of bit[15] is concatenated, as seen in the outputs for input 0x8000, as well as 0x1234 and 0x7F4C.

ALU Control


Data Inputs
/tb_alu_control/s_iFunct 6'h20
/tb_alu_control/s_iALUOp 3'h0
Data Outputs
/tb_alu_control/s_oALUSel 5'h1A
(4) 1
(3) 1
(2) 0
(1) 1
(0) 0
20 21 24 27 26 25 2A 00 02 03 22 23 00 08
0 1 2 0 3 2 0 4 0 5 0 6 0 2 0
1A 0A 00 0A 06 07 01 0F 0B 0C 0D 0A 1E 0E 0B 00

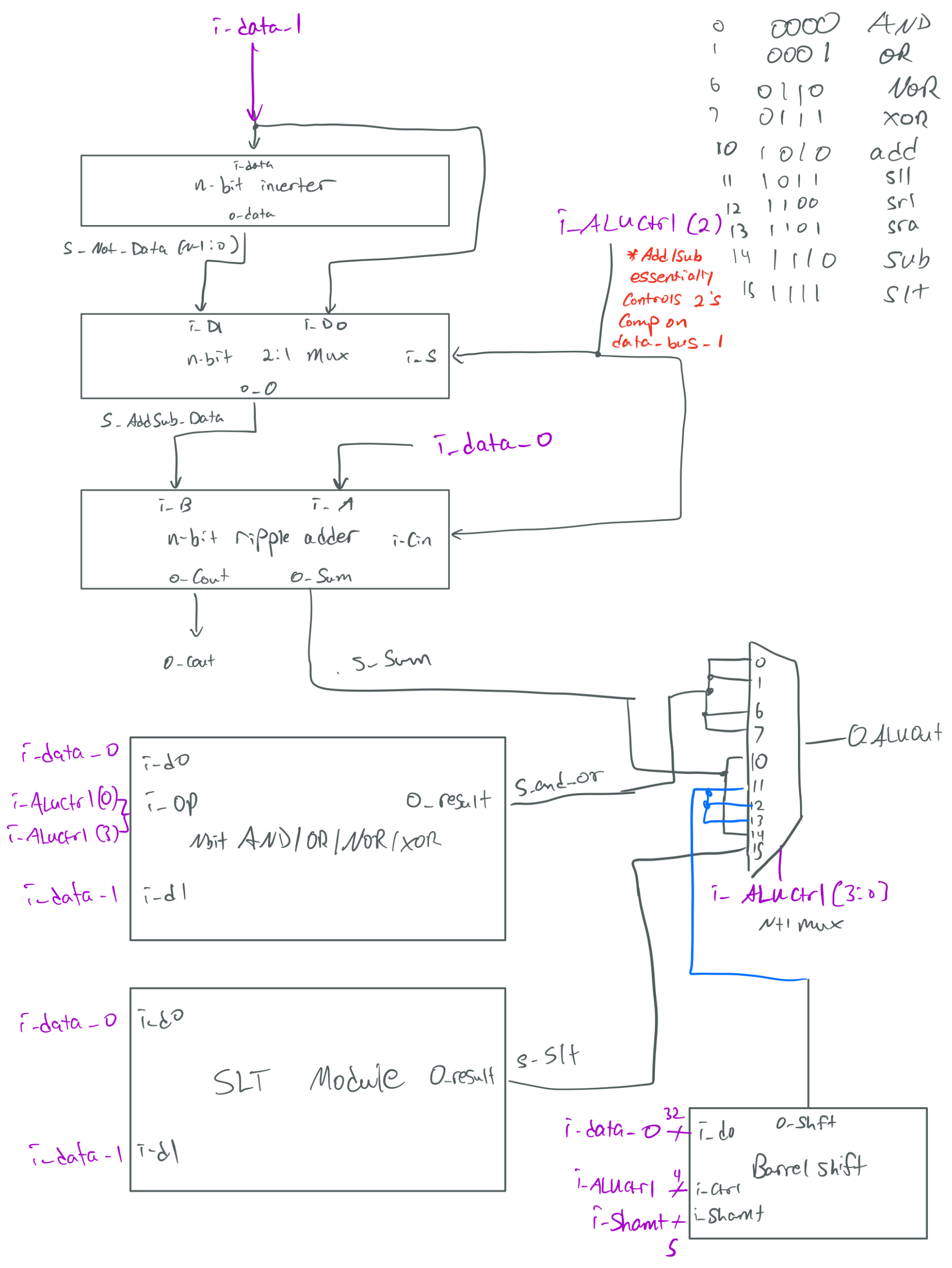
- The ALU Control module outputs ALUSel to the ALU in order to tell it which arithmetic operation to perform. Some ALU operations are selected based purely on the *funct* field and *ALUOp == b000 (meaning an R-type),* whereas some ALU operations are selected with a funct and zero ALUOp or a non-zero ALUOp. This is for ALU operations that might have multiple associated instructions, such as addi and add (at the end of the day they are both adding, although one is I-Type and the other is R-Type). We can see that, for ALU operations with both a non-zero ALUOp as well as a zero ALUOp and a funct, we test both cases to ensure we get the correct ALUSel from the spreadsheet. Otherwise, we just compare the ALUSel output to the expected ALUOp on the spreadsheet.

Branch Module



- We test each branch type input with data that will cause the branch output to assert and data that will not cause the branch output to assert. We start with branch equal. The data is not the same, so it doesn't branch, but we see the output go high when it is. For a branch not equal (BrType = 1), it is the reverse of the previous condition. We test branch on greater than or equal to 0 (BrType = 2) for a negative, 0, and a positive, all giving the expected results. We follow that with a branch greater than 0 (BrType = 3) using a 0 input and a positive, also giving what we expect to see. We repeat the greater-than-equal test data, but instead, we choose a branch less than equal to 0 for the branch type (4), and the expected output is seen. The last branch type is less than 0 (5), and with 0 data, we expect to not branch, and with negative data, we do branch, which is what happens in reality.

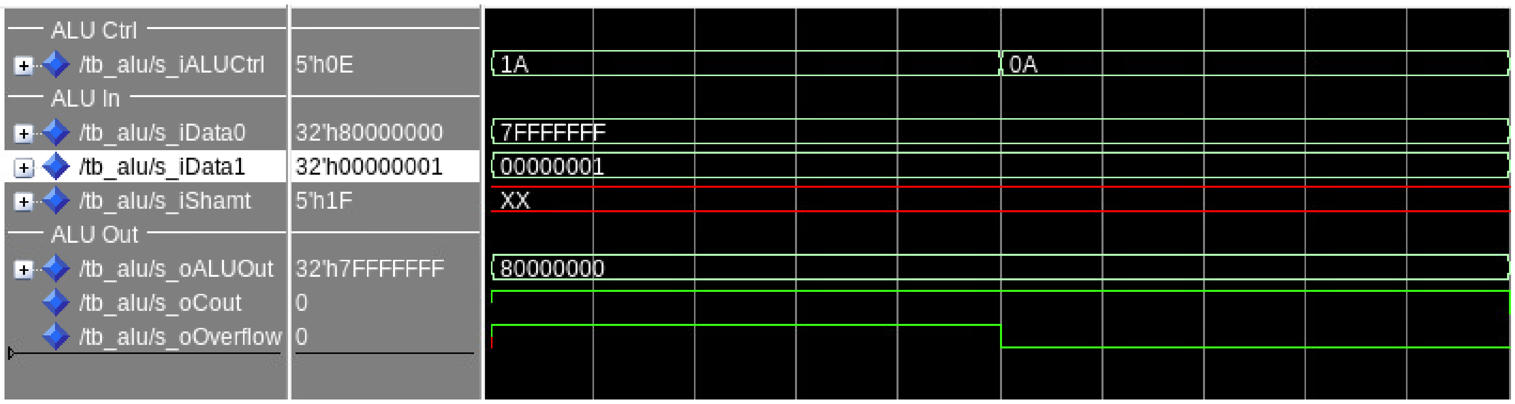
[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is `slt` implemented?



Overflow occurs when the input signals have the same sign, and the output has the opposite sign when performing addition. When performing subtraction, overflow occurs if the signs of the inputs are different and the output has the same sign as the second data source. The overflow signal is only asserted when an instruction deals with data represented in a signed form. We do not calculate a zero signal since it was originally used for branching, but since we have a standalone branch module, it is redundant. SLT is implemented via a simple behavioral module that compares the two (signed) inputs and asserts an output if the first input is less than the second.

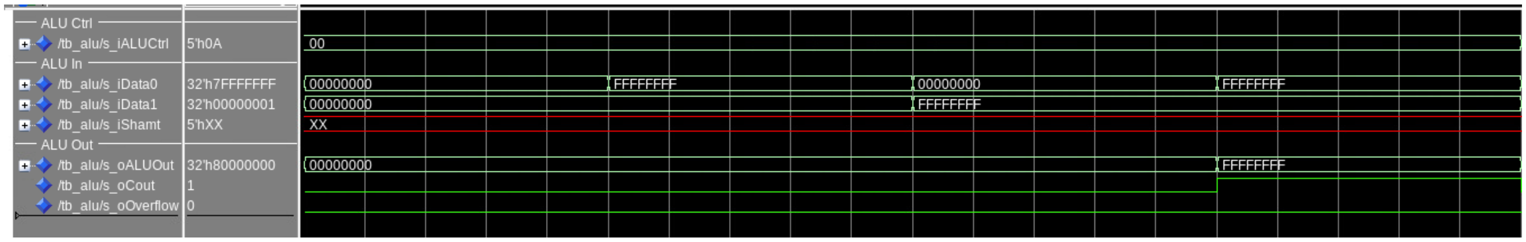
[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

In our test bench, our ALU is operating as expected. Addition operations are either incrementing the number or wrapping it around if it reaches a result requiring more than 32 bits to represent. This is also the case for subtraction, but with decrementing the number. AND and OR also behave as expected, following the respective truth tables outputs.



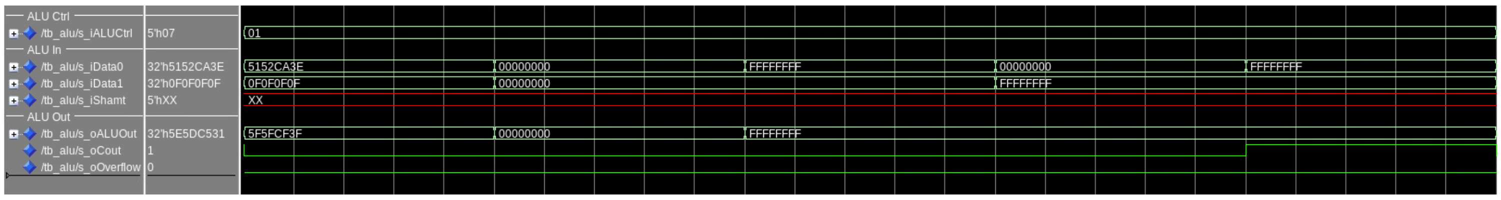
ALU Ctrl
/tb_alu/s_iALUCtrl 5'h0E 1A 0A
ALU In
/tb_alu/s_iData0 32'h80000000 7FFFFFFF
/tb_alu/s_iData1 32'h00000001 00000001
/tb_alu/s_iShamt 5'h1F XX
ALU Out
/tb_alu/s_oALUOut 32'h7FFFFFFF 80000000
/tb_alu/s_oCout 0
/tb_alu/s_oOverflow 0

Add (signed with overflow then unsigned)



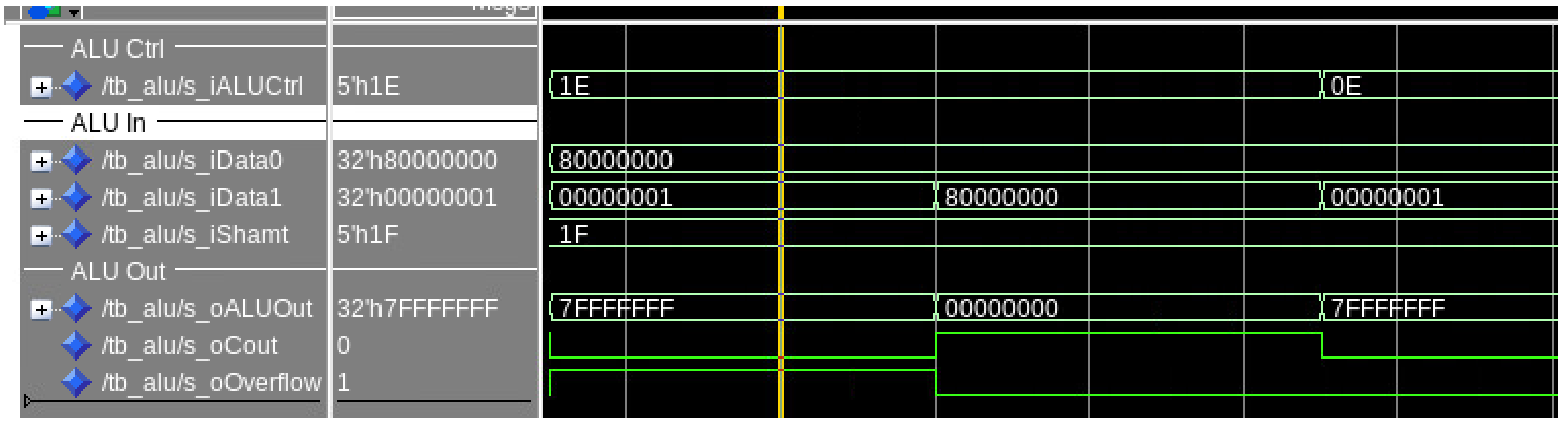
ALU Ctrl
/tb_alu/s_iALUCtrl 5'h0A 00
ALU In
/tb_alu/s_iData0 32'h7FFFFFFF 00000000 FFFFFFFF 00000000 FFFFFFFF
/tb_alu/s_iData1 32'h00000001 00000000 FFFFFFFF
/tb_alu/s_iShamt 5'hXX XX
ALU Out
/tb_alu/s_oALUOut 32'h80000000 00000000 FFFFFFFF
/tb_alu/s_oCout 1
/tb_alu/s_oOverflow 0

AND common cases from the truth table



ALU Ctrl
/tb_alu/s_iALUCtrl 5'h07 01
ALU In
/tb_alu/s_iData0 32'h5152CA3E 5152CA3E 00000000 FFFFFFFF 00000000 FFFFFFFF
/tb_alu/s_iData1 32'h0F0F0F0F 0F0F0F0F 00000000 FFFFFFFF
/tb_alu/s_iShamt 5'hXX XX
ALU Out
/tb_alu/s_oALUOut 32'h5E5DC531 5F5FCF3F 00000000 FFFFFFFF
/tb_alu/s_oCout 1
/tb_alu/s_oOverflow 0

OR common cases from the truth table + edge case



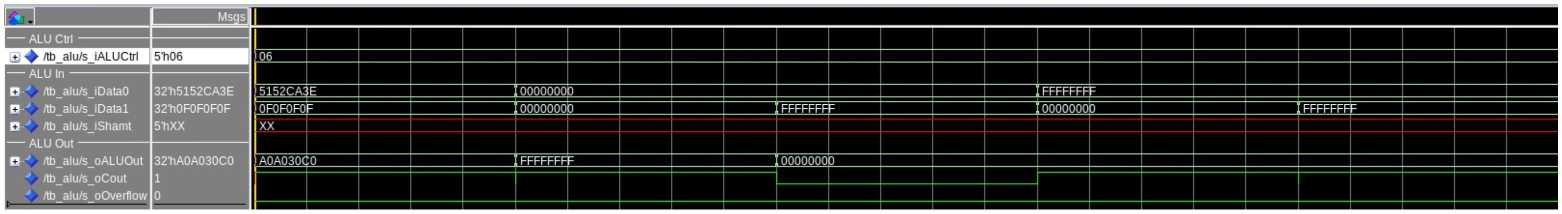
```
ALU Ctrl
/tb_alu/s_iALUCtrl    5'h1E          1E                                 0E
ALU In
/tb_alu/s_iData0      32'h80000000   80000000
/tb_alu/s_iData1      32'h00000001   00000001      80000000             00000001
/tb_alu/s_iShamt      5'h1F          1F
ALU Out
/tb_alu/s_oALUOut     32'h7FFFFFFF   7FFFFFFF      00000000             7FFFFFFF
/tb_alu/s_oCout       0
/tb_alu/s_oOverflow   1
```

Sub (signed with overflow then unsigned)

[Part 2 (c.viii)] Justify why your test plan is comprehensive. Include waveforms that demonstrate your test program's functioning.

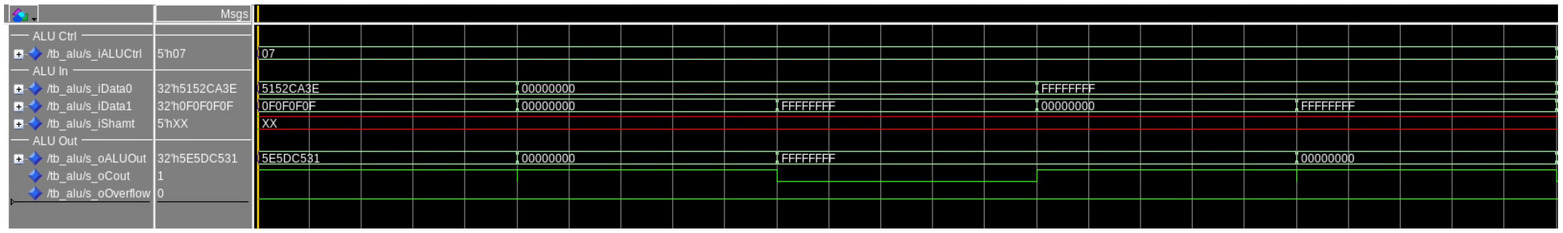
To ensure our testbench is comprehensive, we've included multiple common and edge cases for each instruction where applicable. For the addition operation, we've included a normal, signed case, a normal unsigned case, and an overflow signed case. This is the same for the subtraction operation. For AND, we incorporated a bit-addressable example and all 0s or all 1s in the vector and changed when they were used for four cases. We replicated the alternation of using 0x00000000 and 0xFFFFFFFF for AND and other logical operations (NOR, XOR, and OR). These cases matched the general idea of their truth tables but at a byte level, while our edge case addressed individual bits for each. For set less than operations, we included changes in the sign of each input, alternating between two positives, one positive and one negative (and vice versa), and both negatives. Finally, for sll, srl, and sra, each case has a single 1 on the input and shifts by the next bit (0, 1, 2, 4, 8, 16, and 31) during each succeeding case. This produces the same output from the barrel shifter test bench to ensure an easy match between the two outputs.

Overflow is asserting only on signed operations and asserting only when overflow occurs.

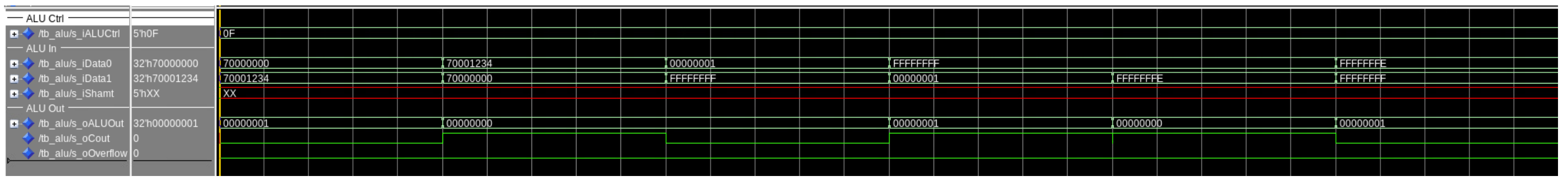


```
Msgs
ALU Ctrl
/tb_alu/s_iALUCtrl    5'h06          06
ALU In
/tb_alu/s_iData0      32'h5152CA3E   5152CA3E   00000000                FFFFFFFF
/tb_alu/s_iData1      32'h0F0F0F0F   0F0F0F0F   00000000    FFFFFFFF    00000000    FFFFFFFF
/tb_alu/s_iShamt      5'hXX          XX
ALU Out
/tb_alu/s_oALUOut     32'hA0A030C0   A0A030C0   FFFFFFFF    00000000
/tb_alu/s_oCout       1
/tb_alu/s_oOverflow   0
```

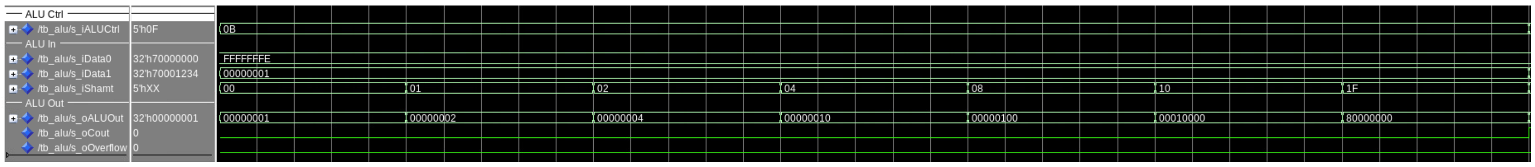
NOR


Msgs
ALU Ctrl
/tb_alu/s_iALUCtrl 5'h07
ALU In
/tb_alu/s_iData0 32'h5152CA3E
/tb_alu/s_iData1 32'h0F0F0F0F
/tb_alu/s_iShamt 5'hXX
ALU Out
/tb_alu/s_oALUOut 32'h5E5DC531
/tb_alu/s_oCout 1
/tb_alu/s_oOverflow 0
07
5152CA3E 00000000 FFFFFFFF
0F0F0F0F 00000000 FFFFFFFF 00000000 FFFFFFFF
XX
5E5DC531 00000000 FFFFFFFF 00000000

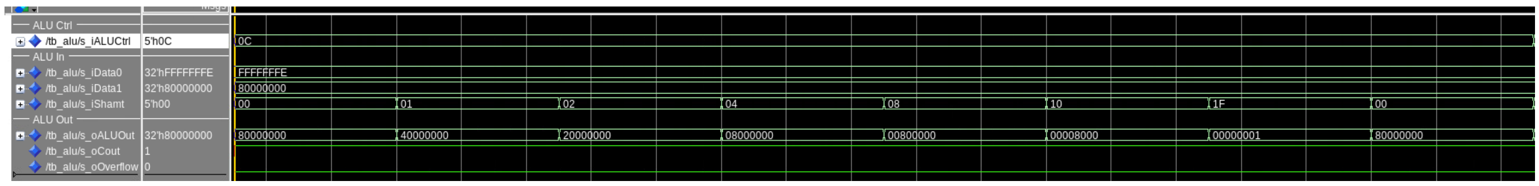
XOR


ALU Ctrl
/tb_alu/s_iALUCtrl 5'h0F
ALU In
/tb_alu/s_iData0 32'h70000000
/tb_alu/s_iData1 32'h70001234
/tb_alu/s_iShamt 5'hXX
ALU Out
/tb_alu/s_oALUOut 32'h00000001
/tb_alu/s_oCout 0
/tb_alu/s_oOverflow 0
0F
70000000 70001234 00000001 FFFFFFFF FFFFFFFE
70001234 70000000 FFFFFFFF 00000001 FFFFFFFE FFFFFFFF
XX
00000001 00000000 00000001 00000000 00000001

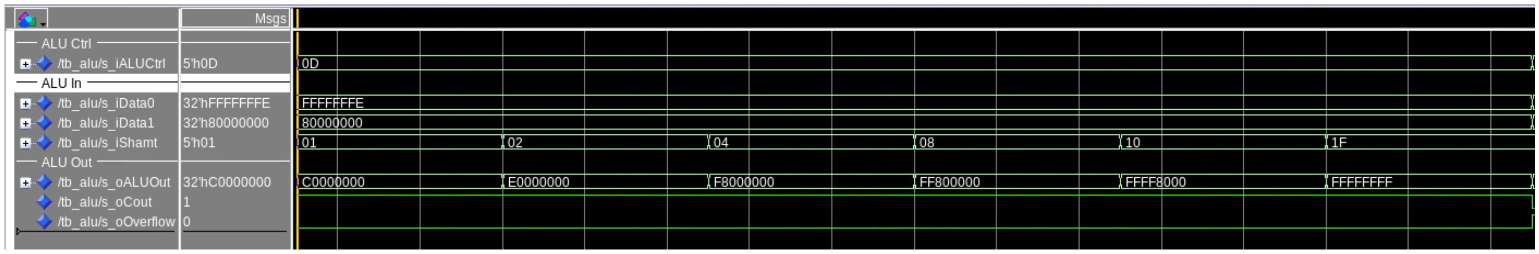
SLT



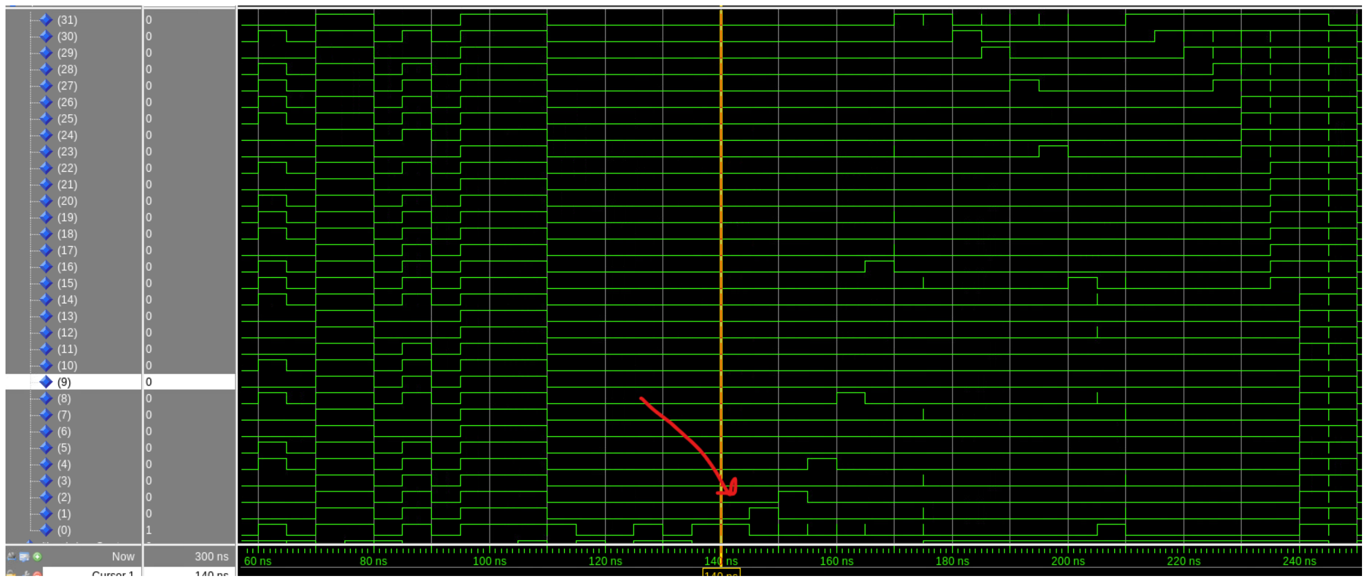
SLL



SRL



SRA

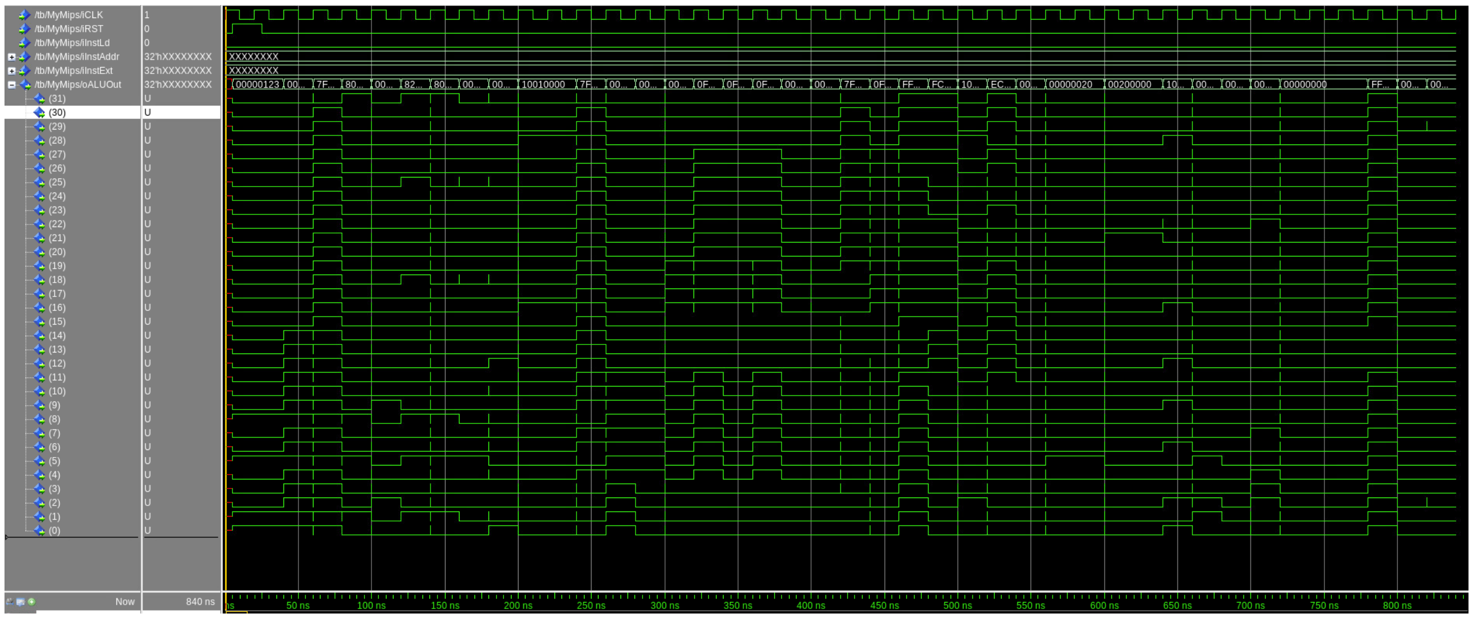


Reproduced Barrel Shifter Output (the arrow is the start of sll tests). The stack of waveforms matches the barrel shifter test bench earlier in this report.

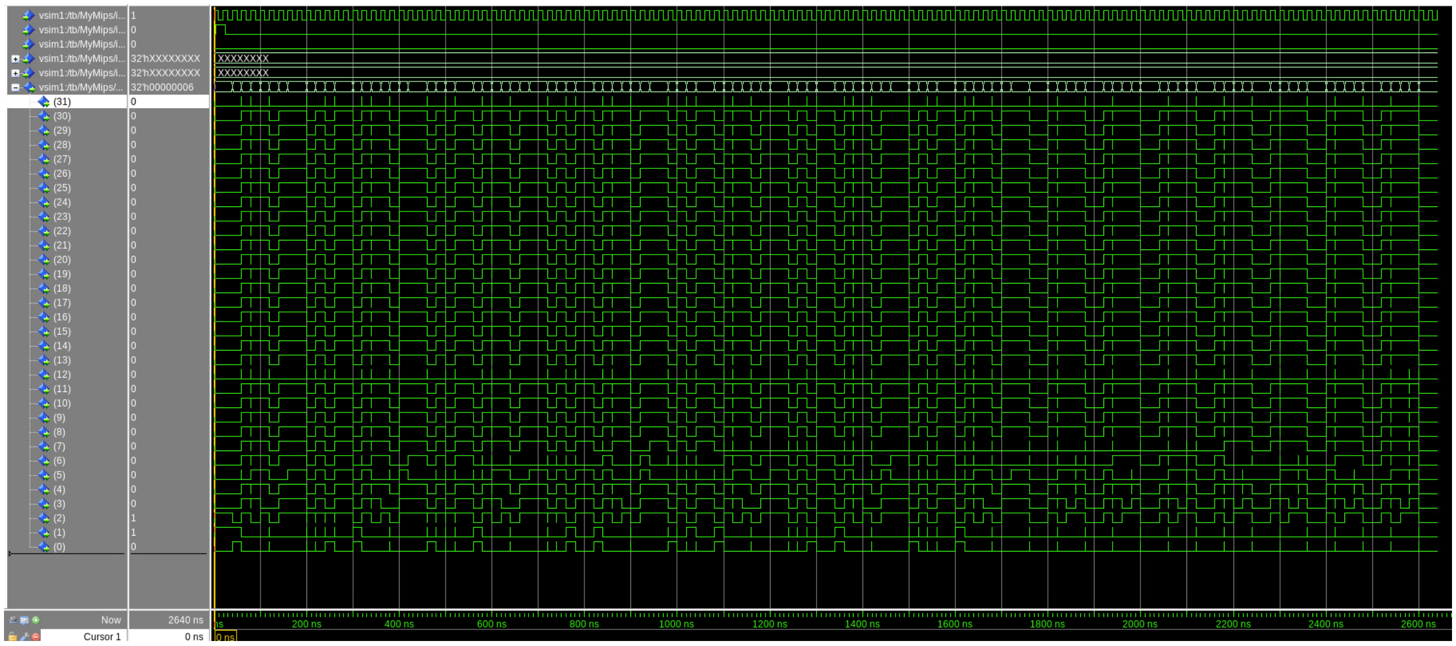
[Part 3] In your writeup, show the QuestaSim output for each of the following tests and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

In all three waveforms, with each instruction usually resulted in a register or memory update. Each waveform shows the output of the ALU, which should correspond with any register changes. For example, in the base_test program, register $8 is written $t1 + 0x123, which is populated on the waveform from the ALU.
Additionally, in cf_test, $4 is written 0 + 6, which populates on its waveform as 0x6, which is expected. Finally, the Bubblesort algorithm had a very large execution time and generated thousands of nanoseconds of signal propagations due to its recursive nature and the intensity of the sorting algorithm. After ~20,000 nanoseconds, Bubblesort completes and calls a halt, which sets the ALU output to 0 to exit the program seen on the waveform. Additionally, the address of the array data label can be seen computed after several previous instructions, which are used to reference the items to be sorted.

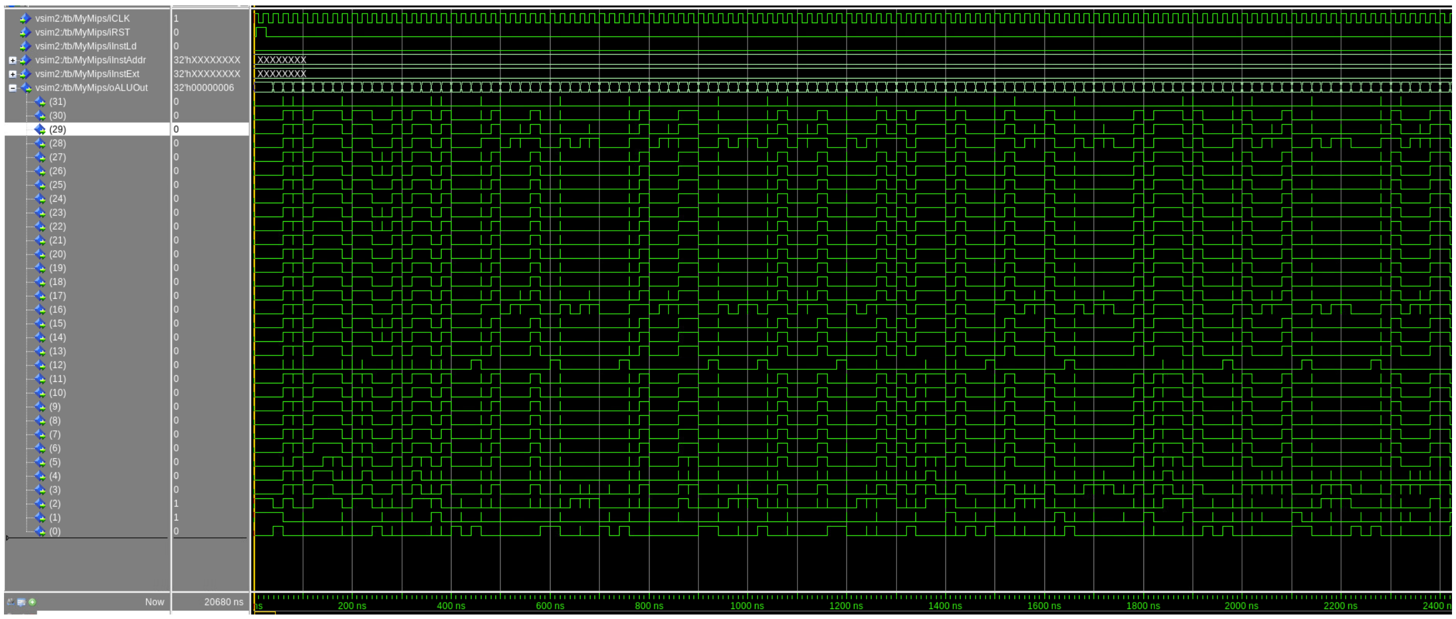
[Part 3 (a)] Create a test application that uses every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1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



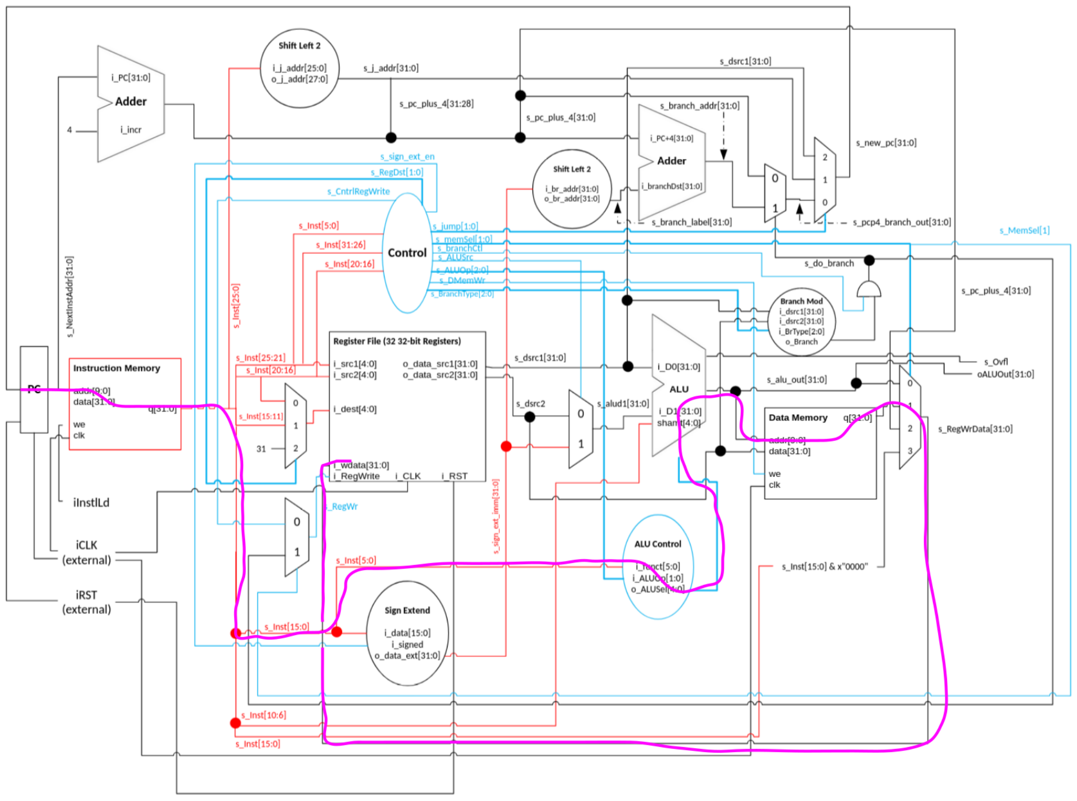
[Part 3 (b)] Create and test an application that uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.



[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?



- The max frequency is 22.49MHz.
- The critical path is PC > IMEM > ALU Control > ALU > DMEM > DMEM Multiplexer > Write to reg.
- It would probably make the most sense to optimize the ALU Control since there are 3.5ns of latency alone in the first step of the path through it. The barrel shifter also adds quite a bit of latency, so it might be interesting to see what a non-structural implementation could result in.