Project Part 1 Report (CPRE 381)

CprE 381: Computer Organization and Assembly-Level Programming

Team Members:

- Conner Ohnesorge
- Levi Wenck

Project Teams Group #:TermProj1_2_02

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)]

> 🔗 **Task**
>
> 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.

👤 **Answer**

Instruction	Opcode (Binary)	Funct (Binary)	jr [jr mux]	jal [jal mux]	MemtoReg	MemWrite (we_mem)	RegWrite (we reg)	RegDst	Branch (PCSrc)	SignExt [determines sign extension on 16 to 32 bit extender]	j [jump]
I-TYPE											
addi	"001000"	"------"	0	0	0 [addi does NOT read from memory]	O [addi does NOT write to memory]	1 [addi writes back to a register]	0 [addi uses rt as destination register rather than rd]	0 [use $PC+4$ no branch]	1 [addi sign extended]	0 no jump
addiu	"001001"	"------"	0	0	0 [addiu does NOT read from memory]	0 [addiu does NOT write to memory]	1 [addiu writes back to a register]	0 [addiu uses rt as destination register rather than rd]	0 [use $PC+4$ no branch]	1 [addi sign extended]	0 no jump
andi	"001100"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rt as destination register rather than rd]	0 [use $PC+4$ no branch]	0 [zero extended]	0 no jump
xori	"001110"	"-----"	0	0	0 [does NOT read from memory]	O [does NOT write to memory]	1 [writes back to a register]	0 [uses rt as destination register rather than rd]	0 [use $PC+4$ no branch]	0 [zero extended]	0 no jump
ori	"001101"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rt as destination register rather than $rd\backslash]$	0 [use $PC+4$ no branch]	o [zero extended]	0 no jump
slti	"001010"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rt as destination register rather than rd]	0 [use $PC+4$ no branch]	1 [sign extended]	0 no jump
lui	"001111"	"-----"	0	0	O[does NOT read from memory]	O [does NOT write to memory]	1 [writes back to a register]	0 [uses rt as destination register rather than rd]	0 [use $PC+4$ no branch]	1 [does not matter but choose one because more sign extensions]	0 no jump
beq	"000100"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does NOT write to a register]	0 [does not matter]	1 [use branch if needed]	1 does not matter but choose one because more sign extensions]	0 no jump
bne	"000101"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does NOT write to a register]	0 [does not matter]	1 [use branch if needed]	1 [does not matter but choose one because more sign extensions]	0 no jump
lw	"100011"	"------"	0	0	1 [reads from memory]	0 [does NOT write to memory]	1 [writes back to a register]	as destination register rat	0 [use $PC+4$ no branch]	ded	0 no jump
sw	"101011"	"-----"	0	0	O [does not matter]	1 [writes to memory]	0 [does NOT write to a register]	0 [does not matter]	0 [use $PC+4$ no branch]	1 [sign extended]	0 no jump
lb	"100000"	"-----"	0	0	1 [reads from memory]	0 [does NOT write to memory]	1 [writes back to a register]	as destination register rat	0 [use $PC+4$ no branch]	ded	0 no jump
R-TYPE											
add	000000	100000	0	0	0 [add does NOT read from memory]	O [add does NOT write to memory]	1 [add writes back to a register]	1 [add uses rd as destination]	0 [use $PC+4$ no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]
addu	000000	100001	0	0	0 [addu does NOT read from memory]	O [addu does NOT write to memory]	1 [addu writes back to a register]	1 [addu uses rd as destination]	0 [use $PC+4$ no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]
and	000000	100100	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use $PC+4$ no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]

Instruction	Opcode (Binary)	Funct (Binary)	jr [jr mux]	jal [jal mux]	MemtoReg	MemWrite (we_mem)	RegWrite (we reg)	RegDst	Branch (PCSrc)	SignExt [determines sign extension on 16 to 32 bit extender]	j [jump]
nor	000000	100111	0	0	o[does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use $PC+4$ no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]
xor	000000	100110	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use $PC+4$ no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]
or	000000	100101	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use $PC+4$ no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]
sit	000000	101010	0	0	O [does NOT read from memory]	o [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use $PC+4$ no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]
sll	000000	000000	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 (uses rd as destination]	0 [use $PC+4$ no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]
srl	000000	000010	0	0	O [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 (uses rd as destination]	0 [use $PC+4$ no branch]	O [don't think it matters? zero extended]	0 [no jump]
sra	000000	000011	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use $PC+4$ no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]
sub	000000	100010	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 (uses rd as destination]	0 [use $PC+4$ no branch]	1 (does not matter but choose one because more sign extensions]	0 [no jump]
subu	000000	100011	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use $PC+4$ no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]
jr	"000000"	"001000"	1	0	O [does not matter]	0 [does NOT write to memory]	0 [does not matter]	0 [does not matter]	0 [does not matter]	1 [does not matter but choose one because more sign extensions]	0 [no jump]
J-TYPE											
j	"000010"	"-----"	0	0	0 [does not matter]	0 [does not matter]	0 [does not matter]	0 [does not matter]	0 [does not matter]	1 [does not matter but choose one because more sign extensions]	1 [jump]
jal	"000011"	"-----"	0	1	O[does NOT read from memory]	O[does NOT write to memory]	1	0 [does not matter]	0 [does not matter]	1 /does not matter but choose one because more sign extensions]	1 [jump]

[Part 2 (a.ii)]

> 🗂 **Task**
>
> Implement the control logic module using whatever method and coding style you prefer.
>
> Create a test-bench to test this module individually and show that your output matches the expected control signals from problem 1(a).

The output of the test-bench will be shown, discussed, and explained in the below the code block containing the test-bench:

> 👤 **Answer**
>
> ```vhdl
> -- author(s): Conner Ohnesorge & Levi Wenck
> -- Department of Electrical and Computer Engineering
> -- Iowa State University
> ----------------------------------------------------------------------
> -- tb_control_unit.vhd
> ----------------------------------------------------------------------
> -- DESCRIPTION: This file contains the testbench for the control_unit
> -- this testbench mocks a signal coming in from IMEM
>
> -- and checks that the control unit will successfully convert/output the
> -- correct control values.
> --
> -- for our purposes the control unit basically acts like a database where
> -- you can either (XOR) search by opcode, or funct key for control values
> ----------------------------------------------------------------------
>
> library IEEE;
> use IEEE.std_logic_1164.all;
> use IEEE.std_logic_textio.all; -- For logic types I/O
> use IEEE.numeric_std.all;     -- For to_usnigned
> library std;
> use std.env.all;    -- For hierarchical/external signals
> use std.textio.all; -- For basic I/O
>
> entity tb_control_unit is
>   generic
>     (gCLK_HPER : time := 50 ns);
> end tb_control_unit;
>
> architecture behavior of tb_control_unit is
>
>   -- Calculate the clock period as twice the half-period
> ```

```vhdl
      constant cCLK_PER : time := gCLK_HPER * 2;
      component control_unit
        port (
          i_opcode    : in std_logic_vector(5 downto 0);  --opcode value, indicates a J or I type instruction (usually)
          i_funct     : in std_logic_vector(5 downto 0);  --funct value, indicates an R type instruction
          o_Ctrl_Unit : out std_logic_vector(14 downto 0) --output/control signals fetched correlating to function
        );
      end component;

      -- Temporary signals to connect to the dff component.
      signal s_CLK : std_logic := '0';

      signal s_opcode    : std_logic_vector(5 downto 0) := (others => '0');
      signal s_funct     : std_logic_vector(5 downto 0) := (others => '0');
      signal s_Ctrl_Unit : std_logic_vector(14 downto 0);
      signal expected_out : std_logic_vector(14 downto 0) := (others => '0');
  begin

      --constant running background clock
      P_CLK : process
      begin
        s_CLK <= '0';
        wait for gCLK_HPER;
        s_CLK <= '1';
        wait for gCLK_HPER/2;
      end process;

      DUT : control_unit
      port map(
        i_opcode    => s_opcode,
        i_funct     => s_funct,
        o_Ctrl_Unit => s_Ctrl_Unit
      );
      -- Testbench process
      P_TB : process
      begin
        -- test add
        s_opcode     <= "000000";
        s_funct      <= "100000";
        expected_out <= "000111000110100";
        wait for cCLK_PER/2;

        -- test addu
        s_opcode     <= "000000";
        s_funct      <= "100001";
        expected_out <= "000000000110100";
        wait for cCLK_PER/2;

        -- test and
        s_opcode     <= "000000";
        s_funct      <= "100100";
        expected_out <= "000001000110100";
        wait for cCLK_PER/2;

        -- test nor
        s_opcode     <= "000000";
        s_funct      <= "100111";
        expected_out <= "000010100110100";
        wait for cCLK_PER/2;

        -- test xor
        s_opcode     <= "000000";
        s_funct      <= "100110";
        expected_out <= "000010000110100";
        wait for cCLK_PER/2;

        -- test or
        s_opcode     <= "000000";
        s_funct      <= "100101";
        expected_out <= "000001100110100";
        wait for cCLK_PER/2;

        -- test slt
        s_opcode     <= "000000";
        s_funct      <= "101010";
        expected_out <= "000011100110100";
        wait for cCLK_PER/2;

        -- test sll
        s_opcode     <= "000000";
        s_funct      <= "000000";
        expected_out <= "000100100110100";
        wait for cCLK_PER/2;

        -- test srl
        s_opcode     <= "000000";
        s_funct      <= "000010";
        expected_out <= "000100000110000";
        wait for cCLK_PER/2;

        -- test sra
        s_opcode     <= "000000";
        s_funct      <= "000011";
        expected_out <= "000101000110100";
```

```vhdl
      wait for cCLK_PER/2;

      -- test sub
      s_opcode      <= "000000";
      s_funct       <= "100010";
      expected_out <= "000111100110100";
      wait for cCLK_PER/2;

      -- test subu
      s_opcode      <= "000000";
      s_funct       <= "100011";
      expected_out <= "000000100110100";
      wait for cCLK_PER/2;

      -- test addi
      s_opcode      <= "001000";
      s_funct       <= "000000";
      expected_out <= "001111000100100";
      wait for cCLK_PER/2;

      -- test addiu
      s_opcode      <= "001001";
      expected_out <= "001000000100100";
      wait for cCLK_PER/2;

      -- test andi
      s_opcode      <= "001100";
      expected_out <= "001001000100000";
      wait for cCLK_PER/2;

      -- test xori
      s_opcode      <= "001110";
      expected_out <= "001010000100000";
      wait for cCLK_PER/2;

      -- test ori
      s_opcode      <= "001101";
      expected_out <= "001001100100000";
      wait for cCLK_PER/2;

      -- test slti
      s_opcode      <= "001010";
      expected_out <= "001011100100100";
      wait for cCLK_PER/2;

      -- test lui
      s_opcode      <= "001111";
      expected_out <= "001011000100100";
      wait for cCLK_PER/2;

      -- test beq
      s_opcode      <= "000100";
      expected_out <= "000101100001100";
      wait for cCLK_PER/2;

      -- test bne
      s_opcode      <= "000101";
      expected_out <= "000110000001100";
      wait for cCLK_PER/2;

      -- test lw
      s_opcode      <= "100011";
      expected_out <= "001000010100100";
      wait for cCLK_PER/2;

      -- test sw
      s_opcode      <= "101011";
      expected_out <= "001000001000100";
      wait for cCLK_PER/2;

      -- test j
      s_opcode      <= "000010";
      expected_out <= "000000000000110";
      wait for cCLK_PER/2;

      -- test jal
      s_opcode      <= "000011";
      expected_out <= "010000000100110";
      wait for cCLK_PER/2;

      -- test jr
      s_opcode      <= "000000";
      s_funct       <= "001000";
      expected_out <= "100000000000110";
      wait for cCLK_PER/2;

      -- test halt
      s_opcode      <= "010100";
      s_funct       <= "000000";
      expected_out <= "000000000000001";
      wait for cCLK_PER/2;
      wait;
   end process;
```

```
end behavior;
```

[Part 2 (b.i)]

📑 **Task**

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.

👤 **Answer**

i. Control Flow Possibilities

The instruction fetch logic must accommodate several control flow possibilities, including:

1. Sequential Execution: The default case where the Program Counter (PC) is simply incremented to point to the next instruction in memory.
2. Branches (Conditional and Unconditional):

- Conditional Branches: The PC is updated based on a condition. If the condition is true, the PC is set to a specific address; otherwise, it proceeds to the next sequential instruction.
- Unconditional Jumps: The PC is directly set to a specific address, regardless of any conditions.

3. Function Calls and Returns:

- Calls: Similar to unconditional jumps, but the address of the next instruction (return address) is saved on a stack or in a register.
- Returns: The PC is set to the return address, popping it from the stack or reading from a register, to continue execution from the point after the function call.

4. Interrupts and Exceptions: Special cases where the PC might be set to a handler routine's address based on external events or errors during execution.

More specifically, the control flow possibilities are as follows:

`bne` : must take a Branch not equal input to an AND gate "&'ed" with not Zero, this must activate the branch address or keep the normally incremented PCAddress.

`beq` : must take a Branch is Equal input to an AND gate with ZERO, to then choose for a multiplexer whether to branch or stay on same PC Address.

`Jal` : Jump and link must set register 31 to the PC Address incremented 4, by using a multiplexer on the write address port of the register file and porting the PC Address + 4 into a multiplexer to choose to take the output from the mem or ALU or the PCAddress and activates the PC Address when Jal is active to 1.

`j` : Jump runs to a multiplexer to choose between the given jump address extended by 2 0s and taking the first bits of the PC Address, or to keep the normal PC Address.

`jr` : Jump Register jumps to the saved register address in RS usually register 31, Jr signal is used to choose between the Jr Address input, or the normal PC + 4 Address.
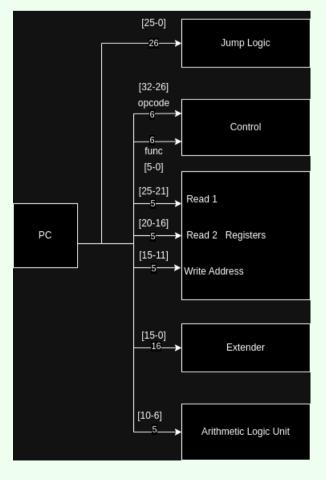
[Part 2 (b.ii)]

📑 **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.ii)**

The following diagram shows our general logic for fetching instructions:

The additional control signals needed are the following:

- `Branch` : This signal is used to determine whether the PC should be updated based on a branch condition.
- `Jump` : This signal is used to determine whether the PC should be updated based on a jump instruction.

- **Jump Register** : This signal is used to determine whether the PC should be updated based on a jump register instruction.
- **Jump and Link** : This signal is used to determine whether the PC should be updated based on a jump and link instruction.
- **PCSrc** : This signal is used to select the source of the new PC value (e.g., branch target, jump target, etc.).

[Part 2 (b.iii)]

🗂 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 report.

👤 Part 2 (b.iii)

Extracting some of the code straight from our `MIPS_Processor.vhd` file to show the implementation of the instruction fetch logic:

Line(s): ~ 55→56 Here we declare the signals that will be used to store the different parts of the instruction that we will need to decode and use for control signals; more specifically, the read source, read destination, and read target registers, the shift amount, and the immediate value.

```
--rs(instructions [25-21]), rt(instructions [20-16]),    rd (instructions [15-11])
signal s_shamt, s_lui_shamt, s_alu_shamt : std_logic_vector(4 downto 0);
```

Line(s): ~ 58 Here we declare a signal for the instruction bits [15-0] that will be used for the immediate value.

```
signal s_imm16    : std_logic_vector(15 downto 0);    -- instruction bits [15-0]
```

Line(s): ~ 63→64 Here we declare the signals that house the opcode and function code of the instruction.

```
signal s_opCode   : std_logic_vector(5 downto 0); --instruction bits[31-26]
signal s_funcCode : std_logic_vector(5 downto 0); --instruction bits[5-0]
```

Line(s) ~ 78 Here we declare the signal that will be used to carry the program counter value.

```
signal s_PCPlusFour    : std_logic_vector(N - 1 downto 0); -- pc + 4
```

Line(s): ~ 239→253 Here we slice the instruction into the different parts that we will need to decode and use for control signals.

```
instructionSlice : process (s_Inst) -- snip the Instruction data into smaller parts
begin
    s_imm16(15 downto 0)          <= s_Inst(15 downto 0);  -- bits[15-0] into Sign Extender
    s_funcCode(5 downto 0)        <= s_Inst(5 downto 0);   -- bits[5-0] into ALU Control
    s_shamt(4 downto 0)           <= s_Inst(10 downto 6);  -- bits[1--6] into ALU (for Barrel Shifter)
    s_regD(4 downto 0)            <= s_Inst(15 downto 11); -- bits[11-15] into RegDstMux bits[4-0]
    s_RegInReadData2(4 downto 0)  <= s_Inst(20 downto 16); -- bits[16-20] into RegDstMux and Register (bits[4-0])
    s_RegInReadData1(4 downto 0)  <= s_Inst(25 downto 21); -- bits[25-21] into Register (bits[4-0])
    s_opCode(5 downto 0)          <= s_Inst(31 downto 26); -- bits[26-31] into Control Brick (bits[5-0])

    s_jumpAddress(0)              <= '0';
    s_jumpAddress(1)              <= '0';                  -- Set first two bits to zero
    s_jumpAddress(27 downto 2) <= s_Inst(25 downto 0); -- Instruction bits[25-0] into bits[27-2] of jumpAddr
end process;
```

Line(s): ~ 257→263 Here we map the control signals to the control unit as required by the control unit.

```
control : control_unit -- grabs the fields from the instruction after decoding that translate to control signals
port
map(
  i_opcode    => s_opCode,   -- in std_logic_vector(5 downto 0);
  i_funct     => s_funcCode, -- in std_logic_vector(5 downto 0);
  o_Ctrl_Unit => s_Ctrl      -- out std_logic_vector(14 downto 0)); (all the control signals needed lumped into 1 vector)
);
```

Line(s): ~ 265→287 Here we map the control signals outputted from the control brick to the different parts of the processor that need them.

```
controlSlice : process (s_Ctrl) -- action of cutting up the lumped up control signals into other wires
begin
    --Control Signals
    s_abnormal          <= s_Ctrl(20); -- selects between either loading a word or a half/byte from memory
    s_HorBExt           <= s_Ctrl(19);
    s_HorB              <= s_Ctrl(18);
    s_BranchNE          <= s_Ctrl(17);
    s_shamt_s           <= s_Ctrl(16);
    s_lui               <= s_Ctrl(15);
    s_jr                <= s_Ctrl(14);
    s_jal               <= s_Ctrl(13);
    s_ALUSrc            <= s_Ctrl(12);
    s_ALUOp(3 downto 0) <= s_Ctrl(11 downto 8); --opcode for the mainALU
    s_MemtoReg          <= s_Ctrl(7);
    s_DMemWr            <= s_Ctrl(6);
    s_RegWr             <= s_Ctrl(5);
    s_RegDst            <= s_Ctrl(4);
    s_Branch            <= s_Ctrl(3);
    s_SignExt           <= s_Ctrl(2);
    s_jump              <= s_Ctrl(1);
```

```
        s_Halt <= s_Ctrl(0);
end process;
```

[Part 2 (c.i.1)]

⎙ Part 2 (c.i.1)

Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does MIPS not have a `sla` instruction?

⚇ Part 2 (c.i.1)

Logical shifts (`srl`) and arithmetic shifts (`sra`) are two types of bit shifting operations.

The primary difference lies in how they handle the introduction of new bits during the shift. In a logical shift, zeroes are inserted into the vacated bit positions. On the other hand, an arithmetic shift copies the sign bit (the most significant bit for signed numbers) into the vacated positions, preserving the number's sign.

MIPS architecture does not include a specific "shift left arithmetic" (`sla`) instruction. This is because the sign bit's significance is primarily at the most significant end of the number. Shifting left would involve filling in the least significant bit, but for arithmetic purposes, the action of simply adding zeros on the right (which happens in both logical and arithmetic left shifts) is sufficient.

Incorporating a `sla` operation that specifically modifies the sign in a manner different from normal left shifts would not align with how numbers are typically managed in binary arithmetic, potentially leading to unintended alterations in the value of the data being shifted.

[Part 2 (c.i.2)]

⎙ Part 2 (c.i.2)

In your report, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

⚇ Part 2 (c.i.2)

The VHDL code for the arithmetic and logical shifting operations is implemented using a barrel shifter. The barrel shifter is a combinational circuit that can perform both logical and arithmetic shifts by shifting the bits in the desired direction. The barrel shifter consists of multiple stages, each of which shifts the bits by a specific amount. By combining the outputs of these stages, the barrel shifter can perform both logical and arithmetic shifts. The control signals determine the direction and amount of the shift, allowing the barrel shifter to perform the desired operation.
We use a multiplexer to select between direction of shift, and a mux to select between the logical and arithmetic shift. The barrel shifter is designed to shift the bits in the appropriate direction based on the control signals, effectively performing the desired operation.
We called the multiplexer that selects between the logical and arithmetic shift the "mux0_t" mux, and the mux that selects between the direction of the shift the "mux_unflip".

[Part 2 (c.i.3)]

⎙ Part 2 (c.i.3)

In your report, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

⚇ Part 2 (c.i.3)

The right barrel shifter can be enhanced to support left shifting operations by simply reversing the bits before the shift operation. This can be achieved by changing the direction of the shift operation, effectively shifting the bits in the opposite direction. By modifying the control signals to indicate a left shift instead of a right shift, the barrel shifter can be repurposed to perform left shifts.
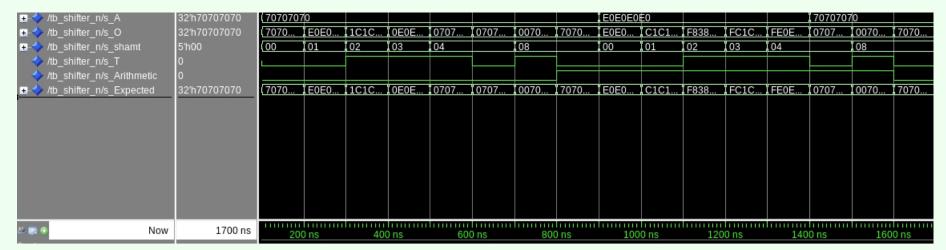This concept is shown on lines ~170-180 of `shifter_N.vhd` or below where we have the mux that selects the direction of the shift depending on the control signal `i_T`:

```
--------------------------------------------------------------------------
-- Level UNFLIP: this level muxes between unreversing or reversing output
--------------------------------------------------------------------------
mux_unflip : mux2t1_N
port
map(
    i_D0 => s_mux_unflip,              -- normal signal (sll)
    i_D1 => bit_reverse(s_mux_unflip), -- unflip signal (sra/srl)
    i_S  => i_T,                       -- right shift(0) or left(1)
    o_0  => o_0                        -- output/End of circuit
);
```

[Part 2 (c.i.4)]

⎙ Part 2 (c.i.4)

Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your report.

⚇ Part 2 (c.i.4)

In the image shown above, the barrel shifter is used to perform both logical and arithmetic shifts. The control signals determine the direction and amount of the shift, allowing the barrel shifter to perform the desired operation. The waveforms show the input data, control signals, and output data for both logical and arithmetic shifts.

The logical shift operation shifts the bits to the right, filling the vacated positions with zeroes. The arithmetic shift operation shifts the bits to the right, preserving the sign bit by copying it into the vacated positions. The waveforms demonstrate how the barrel shifter performs both logical and arithmetic shifts based on the control signals.

[Part 2 (c.ii.1)]

📑 Part 2 (c.ii.1)

In your report, 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.

👤 Part 2 (c.ii.1)

After completing the exercises in zybooks, we frequently found ourselves referring back to the material to ensure we were implementing the correct operations.
We also used the provided MIPS instruction set green-sheet to verify the correct implementation of each instruction.
Additionally, we used the QuestaSim waveforms to verify the correct execution of the operations and to identify any issues or errors in the implementation.
Finally, we used the MARS simulator X-Ray tool to compare the results of our implementation with the expected results to ensure accuracy.

As for design decisions that we had to make, we had to determine the best course of action during the implementation of the fetch control logic design as we both had written two solutions to the problem. We had to decide which solution was the most efficient and effective for our processor design.

In terms of shifting operations, we had to decide how to implement the different shifting operations using the barrel shifter and how to modify the control signals to support both logical and arithmetic shifts. Discussing with the TA's and other students in the class helped us make informed decisions on the most clean and efficient way to implement the operations: mirroring the shift direction and using a mux to select between the logical and arithmetic shift. These design decisions were crucial in ensuring the correct implementation of the operations and the overall functionality of the processor.
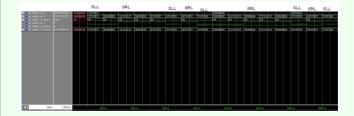
[Part 2 (c.ii.2)]

📑 Part 2 (c.ii.2)

Use QuestaSim to test your 32-bit ALU thoroughly to make sure it is working as expected. Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your report.

👤 Part 2 (c.ii.2)

Barrel Shifter
The annotated test bench wave form and the test bench itself shows accurate arithmetic and logical shifts in the right and left directions.

Test bench of the barrel shifter in its entirety:

```vhdl
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY tb_shifter_N IS
        GENERIC (
                gCLK_HPER : TIME := 50 ns;
                N : INTEGER := 32);
END tb_shifter_N;
ARCHITECTURE behavior OF tb_shifter_N IS
        -- Calculate the clock period as twice the half-period
        CONSTANT cCLK_HPER : TIME := gCLK_HPER * 2;
        COMPONENT shifter_N IS
                GENERIC (N : INTEGER := 32);
                PORT (
                        i_A : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0); --input data to be shifted
                        i_shamt : IN STD_LOGIC_VECTOR(4 DOWNTO 0); --enough to shift 32 bits to the right
                        i_T : IN STD_LOGIC; --shifting right or left (0 = right | 1 = left)
                        i_Arithmetic : IN STD_LOGIC; --logical or arithmetic shift
                        o_O : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0)); --new shifted output
        END COMPONENT;
        SIGNAL s_A, s_O : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
        SIGNAL s_shamt : STD_LOGIC_VECTOR(4 DOWNTO 0);
```

```vhdl
        SIGNAL s_T, s_Arithmetic : STD_LOGIC; -- shift type (left or right) & shift lpgical
        SIGNAL s_Expected : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
    BEGIN
        shifter_1 : shifter_N
        PORT MAP(
                i_A => s_A,
                i_shamt => s_shamt,
                i_T => s_T,
                i_Arithmetic => s_Arithmetic,
                o_O => s_O);
        P_test : PROCESS
        BEGIN
                --logical shift tests firsts
                s_Arithmetic <= '0';
                WAIT FOR cCLK_HPER;
                s_T <= '0'; --sll
                s_A <= x"70707070";
                s_Expected <= x"70707070";
                s_shamt <= "00000";
                WAIT FOR cCLK_HPER;
                s_T <= '0'; --sll
                s_A <= x"70707070";
                  s_Expected <= x"E0E0E0E0";
                s_shamt <= "00001";
                WAIT FOR cCLK_HPER;
                s_T <= '1'; --srl
                s_A <= x"70707070";
                  s_Expected <= x"1C1C1C1C";
                s_shamt <= "00010";
                WAIT FOR cCLK_HPER;
                s_T <= '1'; --srl
                s_A <= x"70707070";
                  s_Expected <= x"0E0E0E0E";
                s_shamt <= "00011";
                WAIT FOR cCLK_HPER;
                s_T <= '1'; --srl
                s_A <= x"70707070"; -- shift by 4 should mean it becomes 07070707
                  s_Expected <= x"07070707";
                s_shamt <= "00100";
                WAIT FOR cCLK_HPER;
                s_T <= '0'; --sll
                s_A <= x"70707070"; -- shift by 4 should mean it becomes 07070700
                  s_Expected <= x"07070700";
                s_shamt <= "00100";
                WAIT FOR cCLK_HPER;
                s_T <= '1'; --srl
                s_A <= x"70707070"; -- shift by 8 should mean it becomes 00707070
                  s_Expected <= x"00707070";
                s_shamt <= "01000";
                WAIT FOR cCLK_HPER;
                s_T <= '0'; --sll
                s_A <= x"70707070"; -- shift by 8 should mean it becomes 70707000
                  s_Expected <= x"70707000";
                s_shamt <= "01000";
                --arithmetic shift tests second (should start seeing sign extension)
                s_Arithmetic <= '1';
                WAIT FOR cCLK_HPER;
                s_T <= '0'; --sll
                s_A <= x"E0E0E0E0";
                  s_Expected <= x"E0E0E0E0";
                s_shamt <= "00000";
                WAIT FOR cCLK_HPER;
                s_T <= '0'; --sll
                s_A <= x"E0E0E0E0";
                  s_Expected <= x"C1C1C1C0";
                s_shamt <= "00001";
                WAIT FOR cCLK_HPER;
                s_T <= '1'; --srl
                s_A <= x"E0E0E0E0";
                  s_Expected <= x"F8383838";
                s_shamt <= "00010";
                WAIT FOR cCLK_HPER;
                s_T <= '1'; --srl
                s_A <= x"E0E0E0E0";
                  s_Expected <= x"FC1C1C1C";
                s_shamt <= "00011";
                WAIT FOR cCLK_HPER;
                s_T <= '1'; --srl
                s_A <= x"E0E0E0E0"; -- shift by 4 should mean it becomes F7070707
                  s_Expected <= x"FE0E0E0E";
                s_shamt <= "00100";
                WAIT FOR cCLK_HPER;
                s_T <= '0'; --sll
                s_A <= x"70707070"; -- shift by 4 should mean it becomes 07070700
                  s_Expected <= x"07070700";
                s_shamt <= "00100";
                WAIT FOR cCLK_HPER;
                s_T <= '1'; --srl
                s_A <= x"70707070"; -- shift by 8 should mean it becomes 00707070
                  s_Expected <= x"00707070";
                s_shamt <= "01000";
                WAIT FOR cCLK_HPER;
                s_T <= '0'; --sll
                s_A <= x"70707070"; -- shift by 8 should mean it becomes 70707000
```

```
                     s_Expected <= x"70707000";
                 s_shamt <= "01000";
         END PROCESS;
     END behavior;
```

The annotated ALU test bench wave form shows correct outputs for the ALU Component and annotates the operation with the corresponding instruction used to execute the given operation.

Here is our test bench:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_textio.ALL; -- For logic types I/O
LIBRARY std;
USE std.env.ALL; -- For hierarchical/external signals
USE std.textio.ALL; -- For basic I/O
-- The entity for the ALU test bench
ENTITY tb_alu IS
        GENERIC (gCLK_HPER : TIME := 10 ns); -- Generic for half of the clock cycle period
END tb_alu;
-- The architecture for the ALU test bench
ARCHITECTURE arch OF tb_alu IS
        --define the total clock period time
        CONSTANT cCLK_PER : TIME := gCLK_HPER * 2;
        COMPONENT alu IS
                PORT (
                        CLK : IN STD_LOGIC;
                        i_Data1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                        i_Data2 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                        i_shamt : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
                        i_aluOp : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
                        o_Zero : OUT STD_LOGIC;
                        o_F : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
        END COMPONENT;
        -- Create signals for all of the inputs and outputs of the file that you are testing
        SIGNAL iCLK, reset : STD_LOGIC := '0';
        SIGNAL s_Data1 : STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL s_Data2 : STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL s_shamt  : STD_LOGIC_VECTOR(4 DOWNTO 0);
        SIGNAL s_ALUOp  : STD_LOGIC_VECTOR(3 DOWNTO 0);
        SIGNAL s_Zero : STD_LOGIC;
        SIGNAL s_ALURslt : STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL s_Expected: STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN
        DUT0 : alu
        PORT MAP(
                CLK => iCLK,
                i_Data1 => s_Data1,
                i_Data2 => s_Data2,
                i_shamt => s_shamt,
                i_aluOp => s_ALUOp,
                o_Zero => s_Zero,
                o_F => s_ALURslt);
        --This first process is to setup the clock for the test bench
        P_CLK : PROCESS
        BEGIN
                iCLK <= '1'; -- clock starts at 1
                WAIT FOR gCLK_HPER; -- after half a cycle
                iCLK <= '0'; -- clock becomes a 0 (negative edge)
                WAIT FOR gCLK_HPER; -- after half a cycle, process begins evaluation again
        END PROCESS;
        P_RST : PROCESS
        BEGIN
                reset <= '0';
                WAIT FOR gCLK_HPER/2;
                reset <= '1';
                WAIT FOR gCLK_HPER * 2;
                reset <= '0';
                WAIT;
        END PROCESS;
        -- Assign inputs for each test case.
        P_TEST_CASES : PROCESS
        BEGIN
                WAIT FOR gCLK_HPER/2;
              -- add, addi, addiu, addu, and, andi, lui, lw, nor, xor, xori, or,
              -- ori, slt, slti, sll, srl, sra, sw, sub, subu, beq, bne, j, jal,
              -- jr, lb, lh, lbu, lhu, sllv, srlv, srav
              -- add, sub, and, or, nor, xor, slt,
                --Test case 1: or operation
                s_Data1 <= x"0000a000";
                s_Data2 <= x"000000a0";
                s_Expected <= x"0000a0a0";
                s_shamt <= "00000"; --should be able to make this anything I want during non shift operations
                s_ALUOp <= "1001";
                WAIT FOR gCLK_HPER * 2;
                WAIT FOR gCLK_HPER * 2;
                --Test case 2: nor operation
                s_Data1 <= x"0000F00F";
```

```vhdl
        s_Data2 <= x"000000FF";
        s_Expected <= x"FFFF0F00";
        s_shamt <= "11111";
        s_ALUOp <= "1000";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 3: and operation
        s_Data1 <= x"0000a00F";
        s_Data2 <= x"000000aF";
        s_Expected <= x"0000000F";
        s_shamt <= "11111";
        s_ALUOp <= "0101";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 4: xor operation
        s_Data1 <= x"0000a00F";
        s_Data2 <= x"000000aF";
        s_Expected <= x"0000a0a0";
        s_shamt <= "01111";
        s_ALUOp <= "1010";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        -- Test case 4.1: xor operation of two negative numbers
        s_Data1 <= x"FFFFFFFF";
        s_Data2 <= x"FFFFFFFF";
        s_Expected <= x"00000000";
        s_shamt <= "01111";
        s_ALUOp <= "1010";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        -- Test case 4.2: xor operation of two zero numbers
        s_Data1 <= x"00000000";
        s_Data2 <= x"00000000";
        s_Expected <= x"00000000";
        s_shamt <= "01111";
        s_ALUOp <= "1010";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 5 :shift left logical (sll) of data2 by 5
        s_Data1 <= x"0000a000";
        s_Data2 <= x"000000a0";
        s_Expected <= x"FFFFFFFF";
        s_shamt <= "00101";
        s_ALUOp <= "0110";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 5.1 :shift left logical (sll) of data2 by 0
        s_Data1 <= x"0000a000";
        s_Data2 <= x"000000a0";
        s_Expected <= x"FFFFFFFF";
        s_shamt <= "00000";
        s_ALUOp <= "0110";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 5.2 :shift left logical (sll) of data2 by 1
        s_Data1 <= x"0000a000";
        s_Data2 <= x"000000a0";
        s_Expected <= x"FFFFFFFF";
        s_shamt <= "00001";
        s_ALUOp <= "0110";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 6 :shift right logical (srl) of data2 by 1
        s_Data1 <= x"0000a000";
        s_Data2 <= x"000000a0";
        s_Expected <= x"00000050";
        s_shamt <= "00001";
        s_ALUOp <= "1110";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 6.1 :shift right arithmetic (srl) of data2 by 1
        s_Data1 <= x"0000a000";
        s_Data2 <= x"FFFFFFFF";
        s_Expected <= x"FFFFFFFF";
        s_shamt <= "00001";
        s_ALUOp <= "1110";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 6.5 :shift right arithmetic (sra) of data2 by 1 (negative)
        s_Data1 <= x"0000a000";
        s_Data2 <= x"FFFFFFFF";
        s_Expected <= x"00000000";
        s_shamt <= "00001";
        s_ALUOp <= "1111";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 6.5 :shift right arithmetic (sra) of data2 by 2 (positive)
        s_Data1 <= x"0000a000";
        s_Data2 <= x"0000000F";
        s_Expected <= x"00000000";
        s_shamt <= "00010";
        s_ALUOp <= "1111";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
```

```vhdl
            --Test case 7: lw operation (either unsigned add or sll by 0)
            s_Data1 <= x"0000a000";
            s_Data2 <= x"000000a0";
            s_Expected <= x"0000a0a0";
            s_shamt <= "00000";
            s_ALUOp <= "0000";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 14: slt operation (if 1 is less than 2)
            s_Data1 <= x"00000000";
            s_Data2 <= x"80000000";
            s_Expected <= x"00000001"; -- will not be set because 1 is larger
            s_shamt <= "00000";
            s_ALUOp <= "0111"; --alu will do subtraction (LSB) and then choose the SLT status output
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 14: slt operation (if 1 is less than 2)
            s_Data1 <= x"000000a0";
            s_Data2 <= x"0000a000";
            s_Expected <= x"00000001"; -- will be set because 1 is smaller
            s_shamt <= "00000";
            s_ALUOp <= "0111";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 14: sltu operation (if 1 is less than 2)
            s_Data1 <= x"0000a000";
            s_Data2 <= x"000000a0";
            s_Expected <= x"00000000"; -- will not be set because 1 is larger
            s_shamt <= "00000";
            s_ALUOp <= "1101"; --alu will do subtraction (LSB) and then choose the SLT status output
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 14: sltu operation (if 1 is less than 2)
            s_Data1 <= x"000000a0";
            s_Data2 <= x"FFFFFFFF";
            s_Expected <= x"00000001"; -- will be set because 2 is larger
            s_shamt <= "00000";
            s_ALUOp <= "1101";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 8: lui instruction (takes the data2 and puts it into the upper 16 bits)
            s_Data1 <= x"12345678";
            s_Data2 <= x"00069420";
            s_Expected <= x"94200000";
            s_shamt <= "10000";
            s_ALUOp <= "0110";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 8: add the two values from data1 and data2 (add)
            s_Data1 <= x"0000a000";
            s_Data2 <= x"000000a0";
            s_Expected <= x"0000a0a0";
            s_shamt <= "00000";
            s_ALUOp <= "0010";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 9: add the two values from data1 and data2 (add) (negative number)
            s_Data1 <= x"00000002";
            s_Data2 <= x"FFFFFFFF";
            s_Expected <= x"00000001";
            s_shamt <= "00000";
            s_ALUOp <= "0010";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 9: add the two values from data1 and data2 (add) (2 negative numbers)
            s_Data1 <= x"FFFFFFFF";
            s_Data2 <= x"FFFFFFFF";
            s_Expected <= x"FFFFFFFE";
            s_shamt <= "00000";
            s_ALUOp <= "0010";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 10: add the two values from data1 and data2 (addu)
            s_Data1 <= x"FFFFFFFF";
            s_Data2 <= x"80000000";
            s_Expected <= x"7FFFFFFF";
            s_shamt <= "00000";
            s_ALUOp <= "0000";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 11: add the two values from data1 and data2 (addu) ("negative" number)
            s_Data1 <= x"00000008";
            s_Data2 <= x"80000000";
            s_Expected <= x"80000008";
            s_shamt <= "00000";
            s_ALUOp <= "0000";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 11.5: add the two values from data1 and data2 (addu) ("negative" number)
            s_Data1 <= x"00000006";
            s_Data2 <= x"90000000";
            s_Expected <= x"90000006";
            s_shamt <= "00000";
            s_ALUOp <= "0000";
```

```vhdl
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 12: subtract value 2 from value 1 (sub)
            s_Data1 <= x"0000a000";
            s_Data2 <= x"000000a0";
            s_Expected <= x"00009F60";
            s_shamt <= "00000";
            s_ALUOp <= "0011";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 13: subtract value 2 from value 1 (sub) (negative result)
            s_Data1 <= x"000000a0";
            s_Data2 <= x"0000a000";
            s_Expected <= x"FFFF60A0";
            s_shamt <= "00000";
            s_ALUOp <= "0001";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 14: subtract value 2 from value 1 (sub) (negative number)
            s_Data1 <= x"0000a000";
            s_Data2 <= x"FFFFFFFF";
            s_Expected <= x"0000a001";
            s_shamt <= "00000";
            s_ALUOp <= "0001";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 12: subtract value 2 from value 1 (subu)
            s_Data1 <= x"00000000";
            s_Data2 <= x"00000002";
            s_Expected <= x"FFFFFFFE";
            s_shamt <= "00000";
            s_ALUOp <= "0001"; -- binary subtraction is not a thing
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 14: subtract value 2 from value 1 (subu) (negative number)
            s_Data1 <= x"FFFFFFFF";
            s_Data2 <= x"FFFFFFF0";
            s_Expected <= x"0000000F";
            s_shamt <= "00000";
            s_ALUOp <= "0001";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 14: subtract value 2 from value 1 (subu) (negative number)
            s_Data1 <= x"000000a0";
            s_Data2 <= x"FFFFFFFF";
            s_Expected <= x"000000a1";
            s_shamt <= "00000";
            s_ALUOp <= "0001";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 14: subtract value 2 from value 1 (subu) (negative number)
            s_Data1 <= x"0000a000";
            s_Data2 <= x"000000a0";
            s_Expected <= x"00009F60";
            s_shamt <= "00000";
            s_ALUOp <= "0001";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 14: subtract value 2 from value 1 (subu) (negative number)
            s_Data1 <= x"00000000";
            s_Data2 <= x"00000000";
            s_Expected <= x"00000000";
            s_shamt <= "00000";
            s_ALUOp <= "1101";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 18: add value 2 from to 1 (addi) (negative number)
            s_Data1 <= x"00000000";
            s_Data2 <= x"FFFFFFF6";
            s_Expected <= x"00000000";
            s_shamt <= "00000";
            s_ALUOp <= "0010";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 18.1: add value 2 from to 1 (addi) (negative number)
            s_Data1 <= x"FFFFFFFF";
            s_Data2 <= x"00000006";
            s_Expected <= x"FFFFFFF5";
            s_shamt <= "00000";
            s_ALUOp <= "0010";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
        END PROCESS;
    END arch;
```
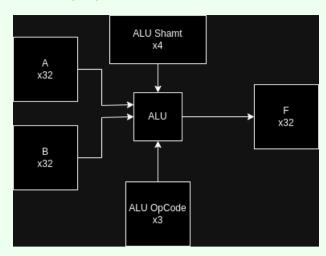
[Part 2 (c.iii)]

Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions:

- How is Overflow calculated?

- How is Zero calculated?
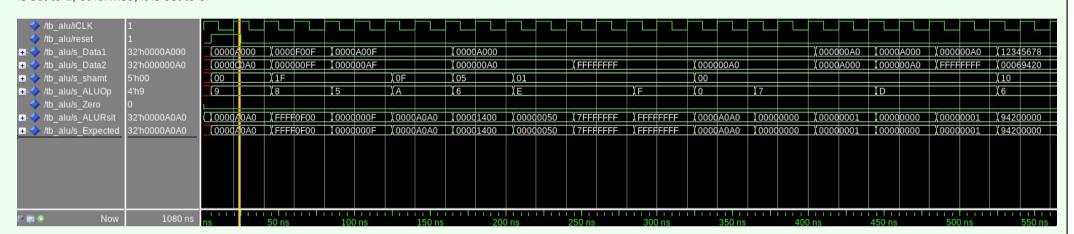- How is `slt` implemented?

Overflow (`o_Overflow` of the `alu`) is calculated by examining the `o_overflow` output of the `adder_subtractor` unit. This signal is high when the result of an addition or subtraction operation overflows the 32-bit signed integer range. Overflow occurs when the sign of the result does not match the sign of the operands, indicating that the result is too large or too small to be represented in the given number of bits.

Zero (`o_Zero` of the `alu`) is calculated by looping through the output `o_y` of the `adder_subtractor` unit and checking if all the bits are zero. If all the bits are zero, the Zero signal is set high, indicating that the result of the operation is zero.

The `slt` operation is implemented by using our added `s_o_slt` signal in the ALU. The `slt` operation compares two signed integers and sets the output to 1 if the first operand is less than the second operand. The `slt` signal is generated by comparing the most significant bits of the two operands. If the first operand is less than the second operand, the `slt` signal is set to 1; otherwise, it is set to 0.

The wave diagram above shows the ALU testbench waveforms. The ALU performs various arithmetic and logical operations, including addition, subtraction, AND, OR, XOR, NOR, and shift operations. The waveforms demonstrate the inputs, outputs, and control signals for each operation. The ALU correctly calculates the results of the operations, including overflow and zero detection, as shown in the waveforms.

[Part 2 (c.v)]

📄 **Part 2 (c.v)**

Within the context on your ALU test bench, describe how the execution of the different operations corresponds to the QuestaSim waveforms in your report.

👤 **Part 3**

The annotated ALU test bench wave form shown below demonstrates correct outputs for the ALU component
and annotates the operation with the corresponding instruction used to execute the given operation.

Here is our test bench in its entirety.

```vhdl
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_textio.ALL; -- For logic types I/O
LIBRARY std;
USE std.env.ALL; -- For hierarchical/external signals
USE std.textio.ALL; -- For basic I/O
-- The entity for the ALU test bench
ENTITY tb_alu IS
        GENERIC (gCLK_HPER : TIME := 10 ns); -- Generic for half of the clock cycle period
END tb_alu;
-- The architecture for the ALU test bench
ARCHITECTURE arch OF tb_alu IS
        --define the total clock period time
        CONSTANT cCLK_PER : TIME := gCLK_HPER * 2;
        COMPONENT alu IS
                PORT (
                        CLK : IN STD_LOGIC;
                        i_Data1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                        i_Data2 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                        i_shamt : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
                        i_aluOp : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
                        o_Zero : OUT STD_LOGIC;
```

```vhdl
                          o_F : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
        END COMPONENT;
        -- Create signals for all of the inputs and outputs of the file that you are testing
        SIGNAL iCLK, reset : STD_LOGIC := '0';
        SIGNAL s_Data1 : STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL s_Data2 : STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL s_shamt  : STD_LOGIC_VECTOR(4 DOWNTO 0);
        SIGNAL s_ALUOp  : STD_LOGIC_VECTOR(3 DOWNTO 0);
        SIGNAL s_Zero : STD_LOGIC;
        SIGNAL s_ALURslt : STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL s_Expected: STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN
        DUT0 : alu
        PORT MAP(
                CLK => iCLK,
                i_Data1 => s_Data1,
                i_Data2 => s_Data2,
                i_shamt => s_shamt,
                i_aluOp => s_ALUOp,
                o_Zero => s_Zero,
                o_F => s_ALURslt);
        --This first process is to setup the clock for the test bench
        P_CLK : PROCESS
        BEGIN
                iCLK <= '1'; -- clock starts at 1
                WAIT FOR gCLK_HPER; -- after half a cycle
                iCLK <= '0'; -- clock becomes a 0 (negative edge)
                WAIT FOR gCLK_HPER; -- after half a cycle, process begins evaluation again
        END PROCESS;
        P_RST : PROCESS
        BEGIN
                reset <= '0';
                WAIT FOR gCLK_HPER/2;
                reset <= '1';
                WAIT FOR gCLK_HPER * 2;
                reset <= '0';
                WAIT;
        END PROCESS;
        -- Assign inputs for each test case.
        P_TEST_CASES : PROCESS
        BEGIN
                WAIT FOR gCLK_HPER/2;
            -- add, addi, addiu, addu, and, andi, lui, lw, nor, xor, xori, or,
            -- ori, slt, slti, sll, srl, sra, sw, sub, subu, beq, bne, j, jal,
            -- jr, lb, lh, lbu, lhu, sllv, srlv, srav
            -- add, sub, and, or, nor, xor, slt,
                --Test case 1: or operation
                s_Data1 <= x"0000a000";
                s_Data2 <= x"000000a0";
                s_Expected <= x"0000a0a0";
                s_shamt <= "00000"; --should be able to make this anything I want during non shift operations
                s_ALUOp <= "1001";
                WAIT FOR gCLK_HPER * 2;
                WAIT FOR gCLK_HPER * 2;
                --Test case 2: nor operation
                s_Data1 <= x"0000F00F";
                s_Data2 <= x"000000FF";
                s_Expected <= x"FFFF0F00";
                s_shamt <= "11111";
                s_ALUOp <= "1000";
                WAIT FOR gCLK_HPER * 2;
                WAIT FOR gCLK_HPER * 2;
                --Test case 3: and operation
                s_Data1 <= x"0000a00F";
                s_Data2 <= x"000000aF";
                s_Expected <= x"0000000F";
                s_shamt <= "11111";
                s_ALUOp <= "0101";
                WAIT FOR gCLK_HPER * 2;
                WAIT FOR gCLK_HPER * 2;
                --Test case 4: xor operation
                s_Data1 <= x"0000a00F";
                s_Data2 <= x"000000aF";
                s_Expected <= x"0000a0a0";
                s_shamt <= "01111";
                s_ALUOp <= "1010";
                WAIT FOR gCLK_HPER * 2;
                WAIT FOR gCLK_HPER * 2;
                -- Test case 4.1: xor operation of two negative numbers
                s_Data1 <= x"FFFFFFFF";
                s_Data2 <= x"FFFFFFFF";
                s_Expected <= x"00000000";
                s_shamt <= "01111";
                s_ALUOp <= "1010";
                WAIT FOR gCLK_HPER * 2;
                WAIT FOR gCLK_HPER * 2;
                -- Test case 4.2: xor operation of two zero numbers
                s_Data1 <= x"00000000";
                s_Data2 <= x"00000000";
                s_Expected <= x"00000000";
                s_shamt <= "01111";
                s_ALUOp <= "1010";
                WAIT FOR gCLK_HPER * 2;
                WAIT FOR gCLK_HPER * 2;
```

```vhdl
                    --Test case 5 :shift left logical (sll) of data2 by 5
                    s_Data1 <= x"0000a000";
                    s_Data2 <= x"000000a0";
                    s_Expected <= x"FFFFFFFF";
                    s_shamt <= "00101";
                    s_ALUOp <= "0110";
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    --Test case 5.1 :shift left logical (sll) of data2 by 0
                    s_Data1 <= x"0000a000";
                    s_Data2 <= x"000000a0";
                    s_Expected <= x"FFFFFFFF";
                    s_shamt <= "00000";
                    s_ALUOp <= "0110";
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    --Test case 5.2 :shift left logical (sll) of data2 by 1
                    s_Data1 <= x"0000a000";
                    s_Data2 <= x"000000a0";
                    s_Expected <= x"FFFFFFFF";
                    s_shamt <= "00001";
                    s_ALUOp <= "0110";
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    --Test case 6 :shift right logical (srl) of data2 by 1
                    s_Data1 <= x"0000a000";
                    s_Data2 <= x"000000a0";
                    s_Expected <= x"00000050";
                    s_shamt <= "00001";
                    s_ALUOp <= "1110";
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    --Test case 6.1 :shift right arithmetic (srl) of data2 by 1
                    s_Data1 <= x"0000a000";
                    s_Data2 <= x"FFFFFFFF";
                    s_Expected <= x"FFFFFFFF";
                    s_shamt <= "00001";
                    s_ALUOp <= "1110";
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    --Test case 6.5 :shift right arithmetic (sra) of data2 by 1 (negative)
                    s_Data1 <= x"0000a000";
                    s_Data2 <= x"FFFFFFFF";
                    s_Expected <= x"00000000";
                    s_shamt <= "00001";
                    s_ALUOp <= "1111";
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    --Test case 6.5 :shift right arithmetic (sra) of data2 by 2 (positive)
                    s_Data1 <= x"0000a000";
                    s_Data2 <= x"0000000F";
                    s_Expected <= x"00000000";
                    s_shamt <= "00010";
                    s_ALUOp <= "1111";
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    --Test case 14: slt operation (if 1 is less than 2)
                    s_Data1 <= x"00000000";
                    s_Data2 <= x"80000000";
                    s_Expected <= x"00000001"; -- will not be set because 1 is larger
                    s_shamt <= "00000";
                    s_ALUOp <= "0111"; --alu will do subtraction (LSB) and then choose the SLT status output
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    --Test case 14: slt operation (if 1 is less than 2)
                    s_Data1 <= x"000000a0";
                    s_Data2 <= x"0000a000";
                    s_Expected <= x"00000001"; -- will be set because 1 is smaller
                    s_shamt <= "00000";
                    s_ALUOp <= "0111";
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    --Test case 14: sltu operation (if 1 is less than 2)
                    s_Data1 <= x"0000a000";
                    s_Data2 <= x"000000a0";
                    s_Expected <= x"00000000"; -- will not be set because 1 is larger
                    s_shamt <= "00000";
                    s_ALUOp <= "1101"; --alu will do subtraction (LSB) and then choose the SLT status output
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    --Test case 14: sltu operation (if 1 is less than 2)
                    s_Data1 <= x"000000a0";
                    s_Data2 <= x"FFFFFFFF";
                    s_Expected <= x"00000001"; -- will be set because 2 is larger
                    s_shamt <= "00000";
                    s_ALUOp <= "1101";
                    WAIT FOR gCLK_HPER * 2;
                    WAIT FOR gCLK_HPER * 2;
                    --Test case 8: lui instruction (takes the data2 and puts it into the upper 16 bits)
                    s_Data1 <= x"12345678";
                    s_Data2 <= x"00069420";
                    s_Expected <= x"94200000";
                    s_shamt <= "10000";
```

```vhdl
        s_ALUOp <= "0110";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 8: add the two values from data1 and data2 (add)
        s_Data1 <= x"0000a000";
        s_Data2 <= x"000000a0";
        s_Expected <= x"0000a0a0";
        s_shamt <= "00000";
        s_ALUOp <= "0010";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 9: add the two values from data1 and data2 (add) (negative number)
        s_Data1 <= x"00000002";
        s_Data2 <= x"FFFFFFFF";
        s_Expected <= x"00000001";
        s_shamt <= "00000";
        s_ALUOp <= "0010";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 9: add the two values from data1 and data2 (add) (2 negative numbers)
        s_Data1 <= x"FFFFFFFF";
        s_Data2 <= x"FFFFFFFF";
        s_Expected <= x"FFFFFFFE";
        s_shamt <= "00000";
        s_ALUOp <= "0010";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 10: add the two values from data1 and data2 (addu)
        s_Data1 <= x"FFFFFFFF";
        s_Data2 <= x"80000000";
        s_Expected <= x"7FFFFFFF";
        s_shamt <= "00000";
        s_ALUOp <= "0000";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 11: add the two values from data1 and data2 (addu) ("negative" number)
        s_Data1 <= x"00000008";
        s_Data2 <= x"80000000";
        s_Expected <= x"80000008";
        s_shamt <= "00000";
        s_ALUOp <= "0000";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 11.5: add the two values from data1 and data2 (addu) ("negative" number)
        s_Data1 <= x"00000006";
        s_Data2 <= x"90000000";
        s_Expected <= x"90000006";
        s_shamt <= "00000";
        s_ALUOp <= "0000";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 12: subtract value 2 from value 1 (sub)
        s_Data1 <= x"0000a000";
        s_Data2 <= x"000000a0";
        s_Expected <= x"00009F60";
        s_shamt <= "00000";
        s_ALUOp <= "0011";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 13: subtract value 2 from value 1 (sub) (negative result)
        s_Data1 <= x"000000a0";
        s_Data2 <= x"0000a000";
        s_Expected <= x"FFFF60A0";
        s_shamt <= "00000";
        s_ALUOp <= "0001";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 14: subtract value 2 from value 1 (sub) (negative number)
        s_Data1 <= x"0000a000";
        s_Data2 <= x"FFFFFFFF";
        s_Expected <= x"0000a001";
        s_shamt <= "00000";
        s_ALUOp <= "0001";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 12: subtract value 2 from value 1 (subu)
        s_Data1 <= x"00000000";
        s_Data2 <= x"00000002";
        s_Expected <= x"FFFFFFFE";
        s_shamt <= "00000";
        s_ALUOp <= "0001"; -- binary subtraction is not a thing
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 14: subtract value 2 from value 1 (subu) (negative number)
        s_Data1 <= x"FFFFFFFF";
        s_Data2 <= x"FFFFFFF0";
        s_Expected <= x"0000000F";
        s_shamt <= "00000";
        s_ALUOp <= "0001";
        WAIT FOR gCLK_HPER * 2;
        WAIT FOR gCLK_HPER * 2;
        --Test case 14: subtract value 2 from value 1 (subu) (negative number)
        s_Data1 <= x"000000a0";
        s_Data2 <= x"FFFFFFFF";
```

```vhdl
            s_Expected <= x"000000a1";
            s_shamt <= "00000";
            s_ALUOp <= "0001";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 14: subtract value 2 from value 1 (subu) (negative number)
            s_Data1 <= x"0000a000";
            s_Data2 <= x"000000a0";
            s_Expected <= x"00009F60";
            s_shamt <= "00000";
            s_ALUOp <= "0001";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 14: subtract value 2 from value 1 (subu) (negative number)
            s_Data1 <= x"00000000";
            s_Data2 <= x"00000000";
            s_Expected <= x"00000000";
            s_shamt <= "00000";
            s_ALUOp <= "1101";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 18: add value 2 from to 1 (addi) (negative number)
            s_Data1 <= x"00000000";
            s_Data2 <= x"FFFFFFF6";
            s_Expected <= x"00000000";
            s_shamt <= "00000";
            s_ALUOp <= "0010";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
            --Test case 18.1: add value 2 from to 1 (addi) (negative number)
            s_Data1 <= x"FFFFFFFF";
            s_Data2 <= x"00000006";
            s_Expected <= x"FFFFFFF5";
            s_shamt <= "00000";
            s_ALUOp <= "0010";
            WAIT FOR gCLK_HPER * 2;
            WAIT FOR gCLK_HPER * 2;
        END PROCESS;
    END arch;
```

[Part 2 (c.viii)]

Part 2 (c.viii)

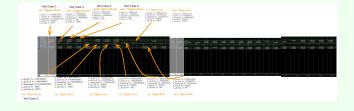Justify why your test plan is comprehensive.
Include waveforms that demonstrate your test programs functioning.

Part 2 (c.viii)

To justify why our test plan is comprehensive, we broke down our testing to consider the following factors:

- **Instruction Coverage**: We tested each instruction in isolation to ensure that it performs the correct operation and produces the expected results. This includes testing arithmetic, logical, control flow, and memory instructions.
- **Edge Cases**: We tested edge cases to ensure that the processor can handle unusual or unexpected inputs. This includes testing the minimum and maximum values for arithmetic operations, as well as testing boundary conditions for control flow instructions.
- **Control Signals**: We tested the control signals to ensure that they are generated correctly and that the processor executes the correct operations based on the control signals. This includes testing the control signals for arithmetic, logical, and control flow instructions.
- **Data Hazards**: We tested for data hazards to ensure that the processor can handle dependencies between instructions and that the correct data is read from and written to registers. This includes testing for read-after-write hazards and write-after-read hazards.
  Using QuestaSim, we were able to thoroughly test the processor design and verify that it performs the correct operations and produces the expected results.
  The test plan covers a wide range of scenarios and conditions to ensure that the processor is robust and reliable.
  By testing each instruction and control signal in isolation, we can identify and address any issues or errors in the design, ensuring that the processor functions correctly under all conditions.

The waveforms below demonstrate the test programs functioning correctly:

"ALU Test Bench Wave Form.png" could not be found.

"Pasted image 20240421103347.png" could not be found.

[Part 3]

Part 3

In your report, 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.

The following answer have all the corresponding tests required by this question and its various parts.

[Part 3 (a)] Create a Test App that uses arithmetic/logical instructions

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`.

```
  .data
arr: .word 0 : 10
  .text
  la $s0, arr
  addTests:
  addi $t0, $zero, 100  #$t0 = 100
  addiu $t1, $zero, -1  #$t1 = 4,294,967,295
  add $t2, $t0, $zero   #$t2 = $t0
  addu $t3, $t0, $t1
andTests:
  and $t1, $t3, $t0
  andi $t0, $t0, 66
  swTest:
  sw $t0, 0($s0)
loadTests1:
  lui $t0, 0x1010
  lw $t1, 0($s0)
orTests:
  nor $t2, $t0, $t1
  xor $t3, $t0, $t1
  xori $t4, $t0, 0x11001100
  or $t5, $t0, $t1
  ori $t6, $t0, 0x4444
shiftTests:
  slt $t1, $t4, $t5
  slti $t2, $t2, 55
  addi $t0, $zero, 0x00000001
  sll $t0, $t0, 4
  srl $t0, $t0, 4
  addi $t0, $zero, -3456
  sra $t0, $t0, 4
subTests:
  addi $t0, $zero, 10
  addi $t1, $zero, 8
  sub $t2, $t0, $t1
  sub $t3, $t1, $t0
  subu $t4, $t1, $t0
loadTests2:
  lb $t0, 0($s0)
  lh $t1, 4($s0)
  lbu $t2, 8($s0)
  lhu $t3, 12($s0)
shiftVariableTests:
  addi $t0, $zero, 0x00000001
  addi $t7, $zero, 4
  sllv $t1, $t0, $t7
  srlv $t2, $t0, $t7
  addi $t0, $zero, -3456
  srav $t3, $t0, $t7
halt
```

[Part 3 (b)]

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`.

```
  .data
msg: .asciiz "Final result: "

  .text
  .globl main

# Function declarations for clarity
# Each function calls the next, incrementing a value passed through registers
func1:
```

```
    addi $a0, $a0, 1      # Increment the argument
    jal func2               # Call func2
    jr $ra                  # Return to caller
func2:
    addi $a0, $a0, 1      # Increment the argument
    jal func3               # Call func3
    jr $ra                  # Return to caller
func3:
    addi $a0, $a0, 1      # Increment the argument
    jal func4               # Call func4
    jr $ra                  # Return to caller
func4:
    addi $a0, $a0, 1      # Increment the argument
    jal func5               # Call func5
    jr $ra                  # Return to caller
func5:
    addi $a0, $a0, 1      # Increment the argument
    jr $ra                  # Return to caller with final value in $a0
main:
    # Initial setup
    li $a0, 0              # Initialize argument to 0
    # Call the first function in the chain
    jal func1
    # After returning from the call chain, print the result
    move $a0, $v0         # Move the result to $a0 for printing
    li $v0, 4             # System call for print_string
    la $a0, msg           # Load address of msg into $a0
    syscall
    li $v0, 1             # System call for print_int
    move $a0, $v0         # Move final result into $a0
    syscall
    # Exit program
    li $v0, 10            # System call for exit
    syscall
```

[Part 3 (c)]

👤 **Part 3 (c)**

Here is our tested bubble sort algorithm in MIPS assembly.

```
#Some starting data in memory to test things
.data
arr: .word 0x10, 0x20, 0x18, 0x28, 0x8, 0x0, 0x10

.text
.globl main

main:
#Where A is an array of ints
#la $s0, arr #base Address of array A <-- Load the memory and maybe change this
lui $s0, 4097
addi $s1, $0, 6 #Len of array A
addi $t0, $0, 0
ori $s0, 0
#addi $s1, $s1, -1 #$s1 = A.len() - 1 Just put the right amount in at the start
#   $t0 = i
#   $t1 = j

Outer:
        slt $at, $t0, $s1
        beq $at, $0 Exit #branches when $t0 > $s1        exits when i>N
        addi $t7, $s0, 0 #temp holding the Base Addr of A. I will increase this in parallel with J.
        sub $t3, $s1, $t0 #Find $t3 = (n-i) (number of elements - outer iterator) index past which the array is sorted
        addi $t1, $0, 0 #j
        Inner:
                slt $at, $t1, $t3
                #Do Logic "Set-up"
                lw $t4, 0($t7) #A[j]
                lw $t5, 4($t7) #A[j+1]
                beq $at, $0 BackToOuter #branches when not set so; t1 > t0
                slt $at,  $t5, $t4 #A[j+1] < A[j] then swap them; Set if j=1 < j
                beq $at, $0, EndJ #goes back to the outer loop
                sw $t5, 0($t7)
                sw $t4, 4($t7)
                EndJ:
                addi $t1, $t1, 1 #j++
                addi $t7, $t7, 4 #go to next element in A
                j Inner
        BackToOuter:
        addi $t0, $t0, 1
        j Outer
Exit:
halt
```

[Part 4] Report the maximum frequency

↪ Part 4

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?

⇌ Part 5

Our processor can run at a maximum frequency of 20.04 MHz.
The critical path is the path from the MIPS PC register to the register file.
To improve the frequency, we would focus on optimizing the components along this critical path, such as the MIPS PC register, the instruction memory, and the register file. By optimizing these components, we can reduce the propagation delay and improve the overall performance of the processor.

Output of the timing tool-flow:

```
FMax: 20.04mhz Clk Constraint: 20.00ns Slack: -29.90ns
```