

446 EXP2 – Report

1.2.1: Datapath Design

Question 1:

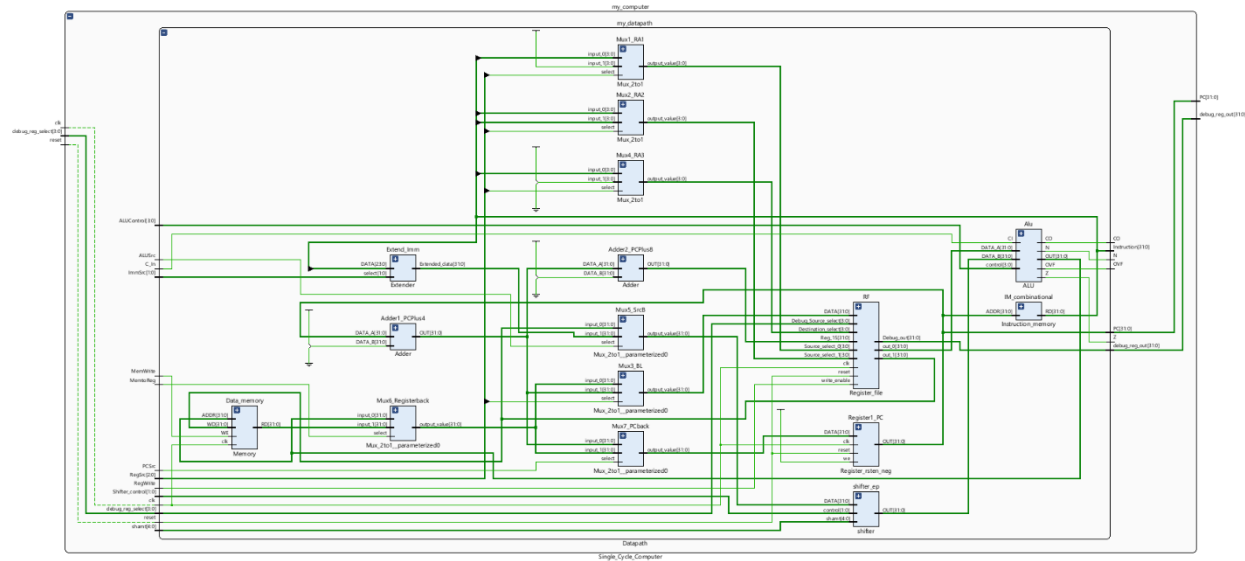


Figure 1: Datapath's RTL Schematic

Question 2:

• Register Shifted Immediate Operations:

In my design I added a dedicated shifter module. After selecting the second operand using a 2-to-1 mux (Mux5_SrcB) that chooses between the register value (RD2) and the immediate value (after extension), the output is passed through the shifter. The shifter is controlled by two signals:

- *Shifter_control* determines whether the operand is shifted by a register value or by an immediate rotate amount.
- *shamt* holds the shift amount.

This arrangement lets the datapath support instructions where the second operand must be shifted (as in many data-processing instructions) without adding extra logic outside the provided modules.

• Both MOV Operations:

For MOV instructions, my ALU is designed with a pass-through path for the second operand. In the datapath, the mux that selects the ALU's second operand (via ALUSrc) effectively chooses between a register value and an immediate value. When a MOV instruction is executed, the control unit sets ALUControl so that the ALU passes its second

input directly to the output. This design supports both forms of MOV—one that moves a register value (when ALUSrc is low) and one that moves a rotated immediate (when ALUSrc is high and the shifter applies the proper rotation).

- **BL and BX Instructions:**

Supporting BL (Branch with Link) and BX (Branch and Exchange) required additional datapath modifications. Two extra multiplexers were integrated:

- *Mux3_BL* selects between the normal ALU result and PCPlus4, which is needed for BL instructions (so that the return address is stored).

- *Mux4_RA3* selects the destination register between the one specified in the instruction and register 14 (which is used to hold the link address).

For BX instructions, the datapath leverages the ALU's pass-through functionality so that the branch target (held in a register) is correctly routed to the PC. These extra muxes ensure that the link address is saved when needed and that the correct register is used in branch exchange operations.

1.2.2: Controller Design

I have split controller into Decoder and Conditional Logic.

Question 1:

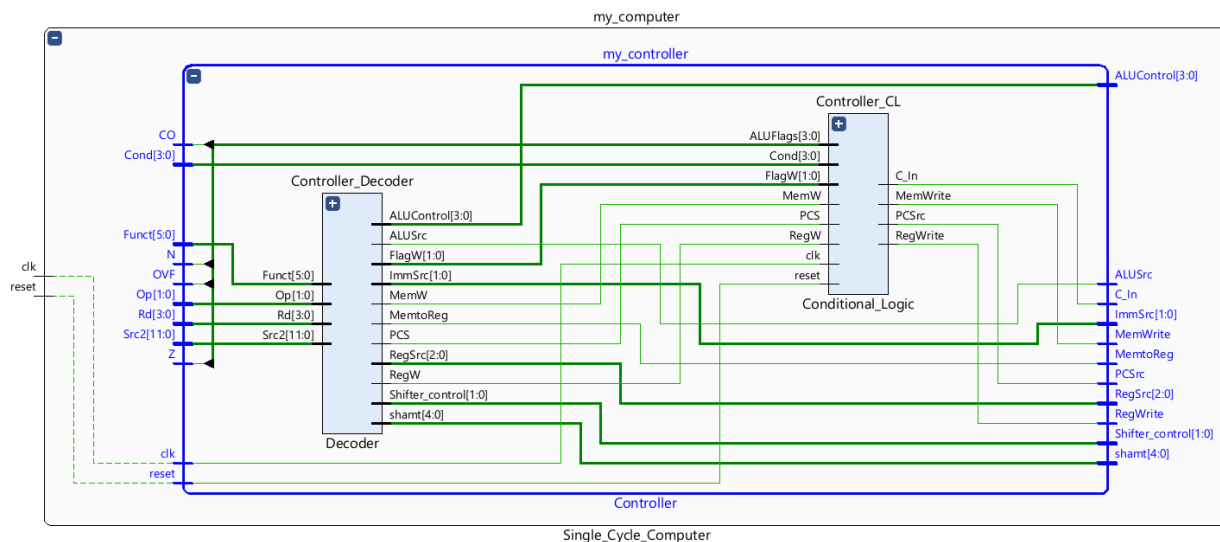


Figure 2: Controller's General RTL Schematic

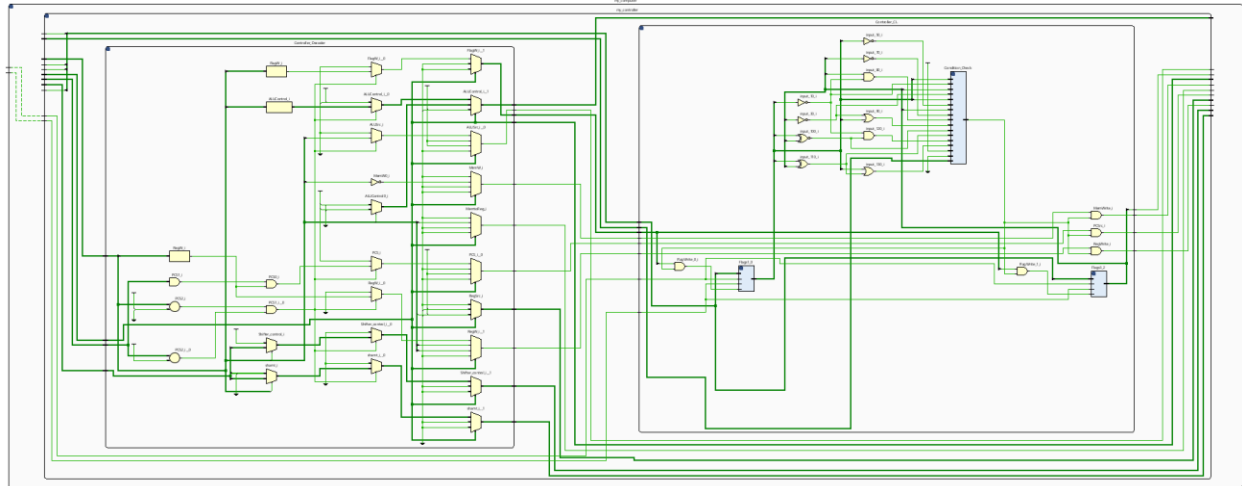


Figure 3: RTL Schematic of Decoder & Conditional Logic Extended

Question 2:

• **Register Shifted Immediate Operations:**

The Decoder module inspects the immediate bit (Funct[5]) to decide whether the second ALU operand comes directly from a register or from a rotated immediate. When the immediate bit is set, the Decoder routes the instruction's 12-bit Src2 field to a dedicated shifter. It generates both the shifter control signal (Shifter_control) and the shift amount (shamt) by selecting the appropriate bits from Src2 (using a "rot-immediate" format where the rotate amount is taken from bits [11:8] and extended with a zero). This enables the datapath to correctly compute operands that require shifting as part of the instruction.

• **Both MOV Operations:**

The MOV instruction appears in two forms: one that moves a register value (MOV register) and one that moves an immediate value (MOV immediate or rot-imm8). In my Decoder, both cases are identified by Funct[4:1] equal to 1101. The distinction is made via the immediate bit (Funct[5]).

– When Funct[5] is low, the design treats it as a register-to-register MOV, so the second operand is taken from the register file (after any required shifting determined by the Src2 field).

– When Funct[5] is high, the rotated immediate is used instead. In either case, the ALU is commanded (via ALUControl = 1101) to simply pass the second operand through to the result, effectively implementing MOV.

This single decoding scheme thus accommodates both MOV operations without additional extra logic.

• **BL (Branch with Link):**

For BL instructions, the Decoder generates control signals that perform two functions:

- It directs the PC update mux (via PCS and RegSrc) to choose the branch link address (typically PCPlus4) as the value to be stored.
- The RegSrc signal is manipulated (e.g., by concatenating a bit from Funct with a constant pattern) so that the link (return) address is written into register 14.

This way, BL instructions correctly update the program counter while preserving the return address.

• BX (Branch and Exchange):

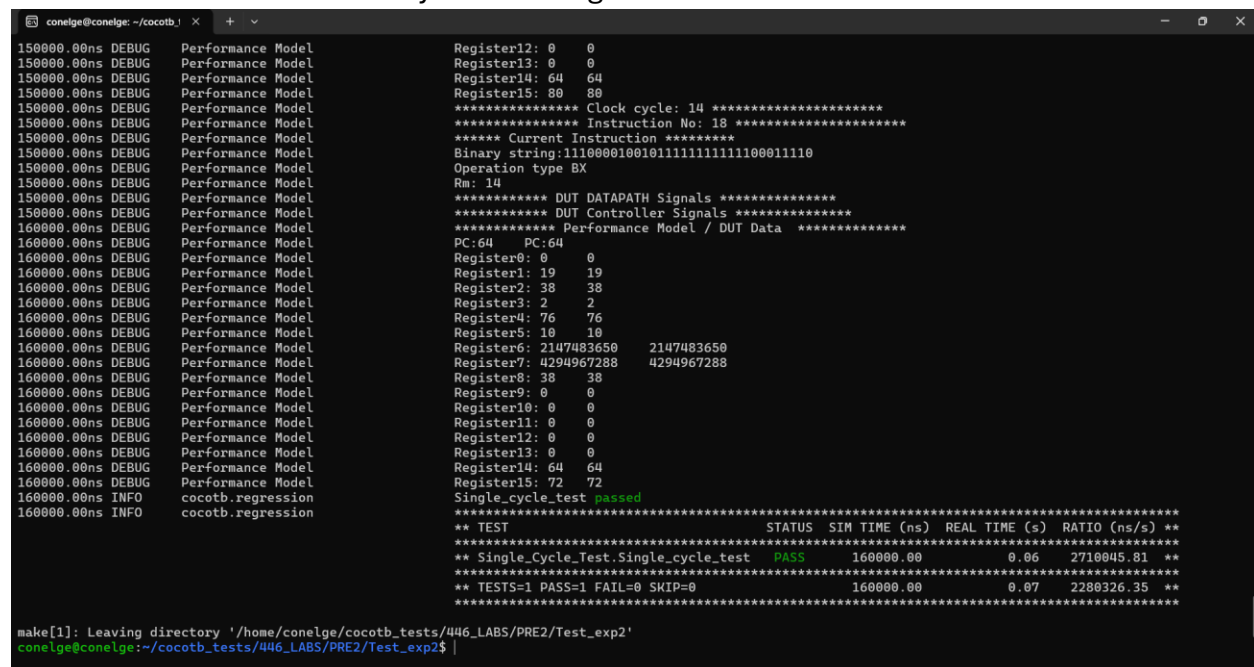
The Decoder is further modified to recognize BX instructions by checking if Funct equals 6'b010010 and if Rd equals 4'b1111. When these conditions are met, the control signals are set to:

- Update the PC directly from a register (by asserting PCS),
- Disable register writes (RegW = 0) and memory writes (MemW = 0), and
- Configure the ALU (ALUControl = 1101) to act as a pass-through so that the branch target value is correctly delivered.

This targeted control ensures that BX instructions fetch the new program counter from the designated register.

1.2.4: Testbench

It is important to note that, I have edited *Single_Cycle_Test.py* line 194 as **tb = TB(instruction_lines,dut, dut.PC, dut.my_datapath.RF)** since my register file has the instance of RF. I don't have any other change in testbench.



```
conelge@conelge: ~/cocotb$  
150000.00ns DEBUG Performance Model Register12: 0 0  
150000.00ns DEBUG Performance Model Register13: 0 0  
150000.00ns DEBUG Performance Model Register14: 64 64  
150000.00ns DEBUG Performance Model Register15: 80 80  
150000.00ns DEBUG Performance Model ***** Clock cycle: 14 *****  
150000.00ns DEBUG Performance Model ***** Instruction No: 18 *****  
150000.00ns DEBUG Performance Model ***** Current Instruction *****  
150000.00ns DEBUG Performance Model Binary string:11100001001011111111111100011110  
150000.00ns DEBUG Performance Model Operation type BX  
150000.00ns DEBUG Performance Model Rn: 14  
150000.00ns DEBUG Performance Model ***** DUT DATAPATH Signals *****  
150000.00ns DEBUG Performance Model ***** DUT Controller Signals *****  
150000.00ns DEBUG Performance Model ***** Performance Model / DUT Data *****  
160000.00ns DEBUG Performance Model PC:64 PC:64  
160000.00ns DEBUG Performance Model Register0: 0 0  
160000.00ns DEBUG Performance Model Register1: 19 19  
160000.00ns DEBUG Performance Model Register2: 38 38  
160000.00ns DEBUG Performance Model Register3: 2 2  
160000.00ns DEBUG Performance Model Register4: 76 76  
160000.00ns DEBUG Performance Model Register5: 10 10  
160000.00ns DEBUG Performance Model Register6: 2147483650 2147483650  
160000.00ns DEBUG Performance Model Register7: 4294967288 4294967288  
160000.00ns DEBUG Performance Model Register8: 38 38  
160000.00ns DEBUG Performance Model Register9: 0 0  
160000.00ns DEBUG Performance Model Register10: 0 0  
160000.00ns DEBUG Performance Model Register11: 0 0  
160000.00ns DEBUG Performance Model Register12: 0 0  
160000.00ns DEBUG Performance Model Register13: 0 0  
160000.00ns DEBUG Performance Model Register14: 64 64  
160000.00ns DEBUG Performance Model Register15: 72 72  
160000.00ns INFO cocotb.regression Single_cycle_test passed  
160000.00ns INFO cocotb.regression *****  
** TEST STATUS SIM TIME (ns) REAL TIME (s) RATIO (ns/s) **  
** Single_Cycle_Test.Single_cycle_test PASS 160000.00 0.06 2710045.81 **  
** TESTS=1 PASS=1 FAIL=0 SKIP=0 160000.00 0.07 2280326.35 **  
*****  
make[1]: Leaving directory '/home/conelge/cocotb_tests/446_LABS/PRE2/Test_exp2'  
conelge@conelge:~/cocotb_tests/446_LABS/PRE2/Test_exp2$
```

Figure 4: Test Bench Results