RISC-V Processor in iVerilog

Introduction to Processor Architecture

Varun Shastry, M.P Samartha, Siddarth Gottumukkula
(2023112005) (2023102038) (2023102040)

March 8, 2025

Contents

1 Introduction

This project was undertaken as part of the *Introduction to Processor Architecture* course in the Spring 2025 semester. We have designed a basic implementation of a RISC-V 32I instruction set processor in iVerilog, supporting a limited instruction set, viz. `ld`, `sd`, `add`, `addi`, `sub`, `and`, `or`, and `beq`. Our initial approach involved implementing a sequential datapath, which we later structured into five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID) and register file reading, Execute and address calculation (EX), Memory Access (MEM), and Writeback (WB).

This project was collaboratively done by **M.P. Samartha** (GitHub), **Varun Shastry** (GitHub), and **Siddarth Gottumukkula** (GitHub). We explain each datapath unit in the sequential implementation before explaining the pipeline architecture.

2 Building the Sequential Datapath

The sequential architecture works so that an instruction will **not** be executed until the previous one is **completely** retired. Thus, the total time to execute the instruction, if we were to say break the datapath into five stages, would be $T\_T = T\_IF + T\_ID + T\_EX + T\_MEM + T\_WB$, and thus clock we can provide to it must be of a frequency $< 1/$ T_T for the processor to function correctly. The image of the implemented sequential datapath unit is provided below.

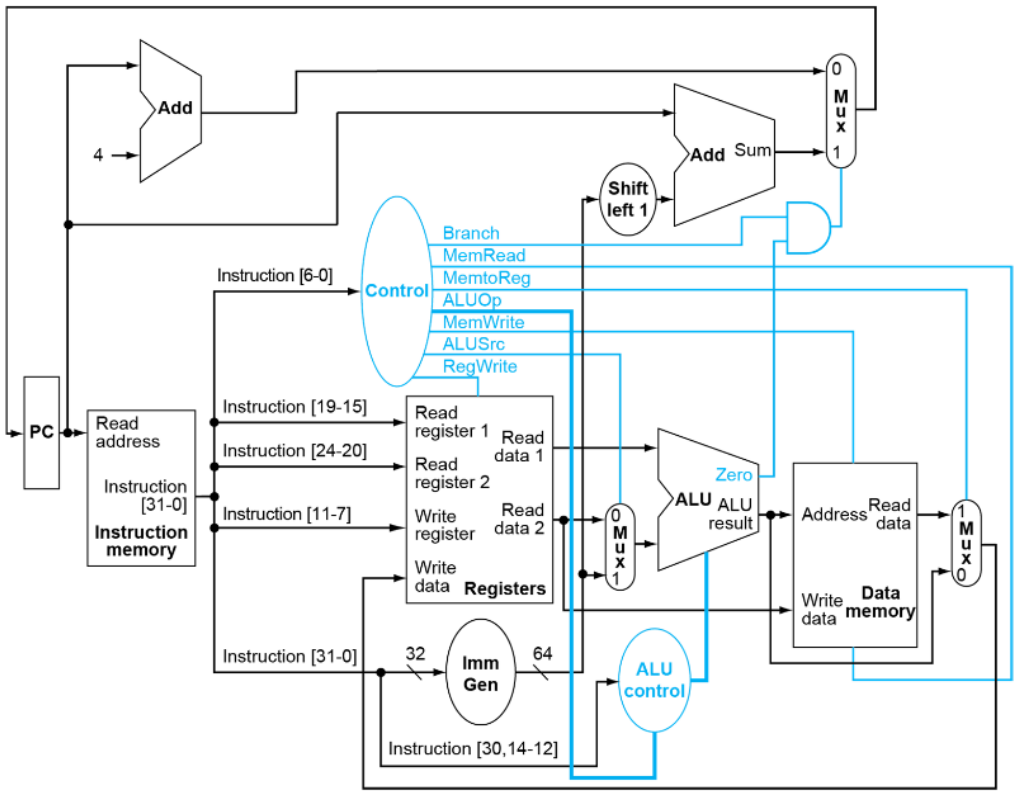


Figure 2.1: Complete Datapath of the Sequential processor

a) Program Counter (PC)

The Program Counter (PC) is a 64-bit register that stores the address of the current instruction. The PC is incremented by 4 (for 32-bit instructions) after every instruction unless a branch occurs, in which case it is updated according to the branch target address. The PC is updated with the value of `pc_in` every clock cycle, eliminating the need for an exclusive write signal. If the reset signal is asserted, the PC is set to 0.

1. **Inputs:** `clk`, `reset`, `pc_in`
2. **Output:** `pc_out`

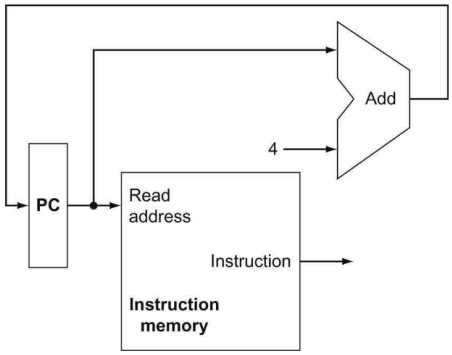


Figure 2.2: PC updation image

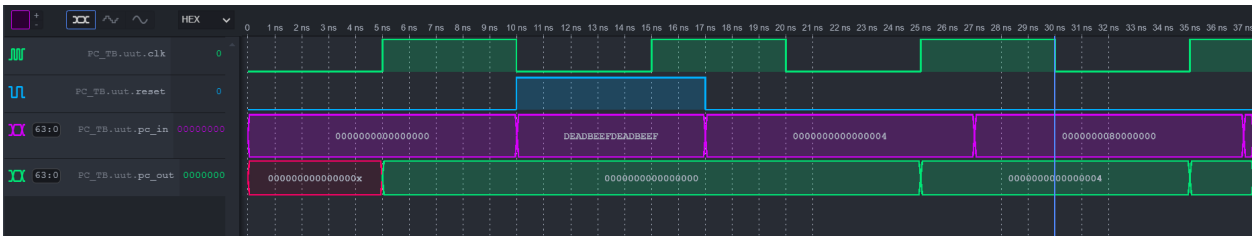


Figure 2.3: GTK-Wave Output for PC Testbench

b) Register File

```
Test Case 1: 0 hardwired to 0
Inputs:
  read_reg1 =  0
  read_reg2 =  0
  write_reg =  0
  write_data = ffffffffffffffff
  reg_write_en = 1
Status: PASS
  read_data1 = 0000000000000000 (Expected: 0000000000000000)
  read_data2 = 0000000000000000 (Expected: 0000000000000000)

Test Case 2:    Write to reg1
Inputs:
  read_reg1 =  1
  read_reg2 =  0
  write_reg =  1
  write_data = deadbeefdeadbeef
  reg_write_en = 1
Status: PASS
  read_data1 = deadbeefdeadbeef (Expected: deadbeefdeadbeef)
  read_data2 = 0000000000000000 (Expected: 0000000000000000)

Test Case 3:  reg2, read reg1
Inputs:
  read_reg1 =  1
  read_reg2 =  2
  write_reg =  2
  write_data = cafebabecafebabe
  reg_write_en = 1
Status: PASS
  read_data1 = deadbeefdeadbeef (Expected: deadbeefdeadbeef)
  read_data2 = cafebabecafebabe (Expected: cafebabecafebabe)

Test Case 4:   Write disabled
Inputs:
  read_reg1 =  1
  read_reg2 =  2
  write_reg =  1
  write_data = 1111111111111111
  reg_write_en = 0
Status: PASS
  read_data1 = deadbeefdeadbeef (Expected: deadbeefdeadbeef)
  read_data2 = cafebabecafebabe (Expected: cafebabecafebabe)
```

```
Test Case 5: ould be ignored)
Inputs:
  read_reg1 =  0
  read_reg2 =  1
  write_reg =  0
  write_data = ffffffffffffffff
  reg_write_en = 1
Status: PASS
  read_data1 = 0000000000000000 (Expected: 0000000000000000)
  read_data2 = deadbeefdeadbeef (Expected: deadbeefdeadbeef)

Test Case 6:     Write to x31
Inputs:
  read_reg1 = 31
  read_reg2 =  1
  write_reg = 31
  write_data = 1234567890abcdef
  reg_write_en = 1
Status: PASS
  read_data1 = 1234567890abcdef (Expected: 1234567890abcdef)
  read_data2 = deadbeefdeadbeef (Expected: deadbeefdeadbeef)

Test Case 7: of same register
Inputs:
  read_reg1 =  2
  read_reg2 =  2
  write_reg =  3
  write_data = aaaaaaaaaaaaaaaa
  reg_write_en = 1
Status: PASS
  read_data1 = cafebabecafebabe (Expected: cafebabecafebabe)
  read_data2 = cafebabecafebabe (Expected: cafebabecafebabe)

=== Test Summary ===
Total Tests: 7
Passed: 7
Failed: 0
==================

register_file_tb.v:148: $finish called at 150000 (1ps)
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
```

Figure 2.4: Command Window output for Resgister file testbench

1. **Inputs:** `clk`, `reset`, `read_reg1`, `read_reg2`, `write_reg`, `write_data`, `reg_write_en`

2. **Outputs:** `read_data1`, `read_data2`

The register file consists of 32 registers, which are hard-coded with value 0 stored in them, in an `initial` block. The register `x0` is always grounded to 0. We slice the 32-bit instruction into two 5-bit fields representing register addresses `read_reg1` and `read_reg2` (if both of them exist). These addresses are used to read data from the register file, outputting them as `read_data1` and `read_data2` (`read_data1` is directly hardwired to ALU Input 1, whereas `read_data2` is goes into a 2x1 mux to be selected between itself and the *imm* value). Reading **occurs continuously**, while writing only takes place when `reg_write_en` is set to 1. The `write_data` signal is a 64-bit line that carries the data to be written into the register specified by `write_reg`.

c) Instruction Memory

```
Test Case 1:
Address = 0x0000000000000000
Expected Instruction = 0x00500113
Actual Instruction   = 0x00500113
Status: PASS

Test Case 2:
Address = 0x0000000000000004
Expected Instruction = 0x00a00193
Actual Instruction   = 0x00a00193
Status: PASS

Test Case 3:
Address = 0x0000000000000008
Expected Instruction = 0x003100b3
Actual Instruction   = 0x003100b3
Status: PASS

Test Case 4:
Address = 0x000000000000000c
Expected Instruction = 0x40310133
Actual Instruction   = 0x40310133
Status: PASS

Test Case 5:
Address = 0x0000000000000010
Expected Instruction = 0x0021a233
Actual Instruction   = 0x0021a233
Status: PASS

Test Case 6:
Address = 0x0000000000000014
Expected Instruction = 0x0041f2b3
Actual Instruction   = 0x0041f2b3
Status: PASS

Test Case 7:
Address = 0x0000000000000018
Expected Instruction = 0x00416333
Actual Instruction   = 0x00416333
Status: PASS

Test Case 8:
Address = 0x000000000000001c
Expected Instruction = 0x004143b3
Actual Instruction   = 0x004143b3
Status: PASS

Test Case 9:
Address = 0x0000000000000020
Expected Instruction = 0x00002437
Actual Instruction   = 0x00002437
Status: PASS

Test Case 10:
Address = 0x0000000000000024
Expected Instruction = 0x06042223
Actual Instruction   = 0x06042223
Status: PASS

Test Case 11:
Address = 0x0000000000000028
Expected Instruction = 0x06442483
Actual Instruction   = 0x06442483
Status: PASS

Test Case 12:
Address = 0x000000000000002c
Expected Instruction = 0x00119493
Actual Instruction   = 0x00119493
Status: PASS

Test Case 13:
Address = 0x0000000000000030
Expected Instruction = 0x00115513
Actual Instruction   = 0x00115513
Status: PASS

Test Case 14:
Address = 0x0000000000000034
Expected Instruction = 0x40115593
Actual Instruction   = 0x40115593
Status: PASS

Test Case 15:
Address = 0x0000000000000038
Expected Instruction = 0x00208663
Actual Instruction   = 0x00208663
Status: PASS

=== Test Summary ===
Total Tests: 15
Passed: 15
Failed: 0
==================
```

Figure 2.5: Command Window output for Instruction memory testbench

1. **Inputs:** `clk`, `reset`, `addr`

2. **Output:** `instr`

The Instruction Memory must only provide read access because the datapath **does not write** instructions (we assume that the instructions are loaded already). Since the instruction memory only reads, we treat it as combinational logic, i.e. the output at any time reflects the contents of the location specified by the address input pointed to by the PC, and thus no read control signal is needed.

The instruction memory essentially extracts 32-bit instructions corresponding to the address provided by the Program Counter (PC). Reading occurs continuously from a text file (whose path is specified in the instruction memory module) containing the instructions. The instructions are stored in memory using **Big-Endian** instead of the little-endian format. However, the code can be modified to do the latter too.

d) Control Unit

1. **Input:** `opcode`
2. **Outputs:** `Branch`, `MemRead`, `MemtoReg`, `ALUOp`, `MemWrite`, `ALUSrc`, `RegWrite`

This module reads the `opcode` and generates the necessary control signals for the rest of the processor.

- `Branch` is set to 1 if the instruction is a branch instruction.
- `MemRead` is set to 1 if the instruction is a load instruction for reading the data memory.
- `MemtoReg` is set to 1 if the instruction is a load instruction for sending the data from the data memory to the registers in the write-back stage. If it is 0 (for an R-type instruction), the data to be written back is taken from the ALU output.

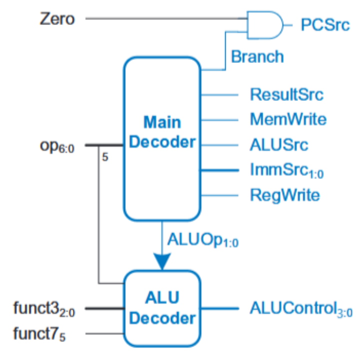
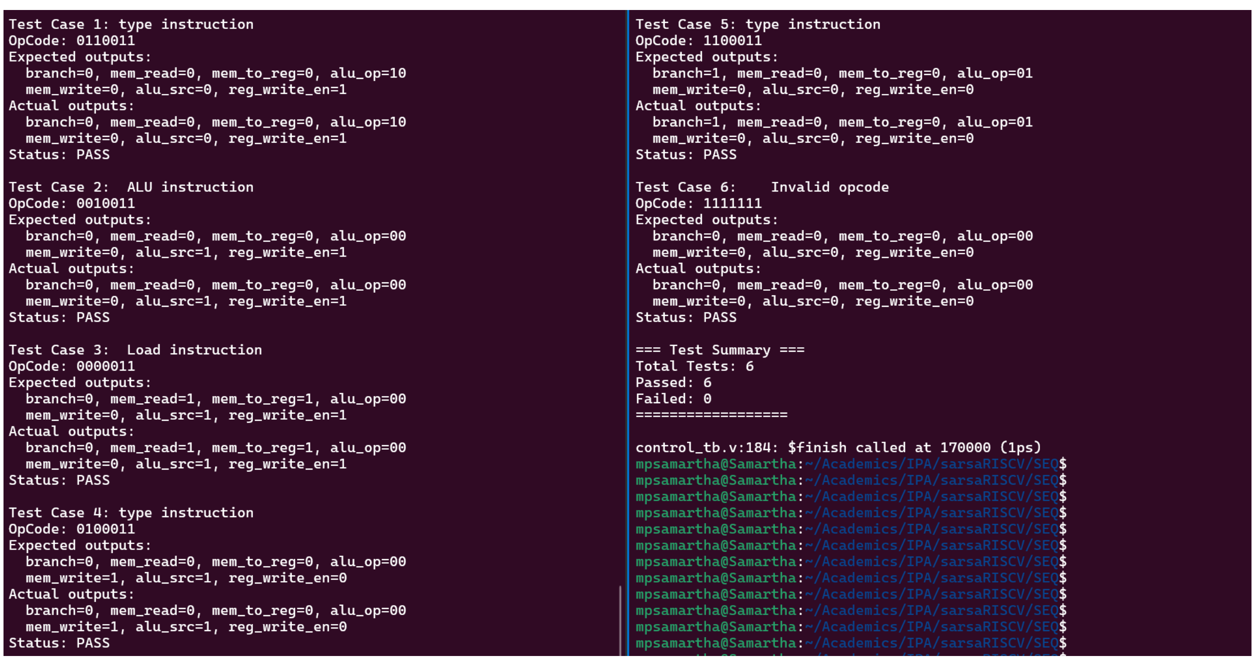


Figure 2.6: Control Signals Diagram



```
Test Case 1: type instruction
OpCode: 0110011
Expected outputs:
  branch=0, mem_read=0, mem_to_reg=0, alu_op=10
  mem_write=0, alu_src=0, reg_write_en=1
Actual outputs:
  branch=0, mem_read=0, mem_to_reg=0, alu_op=10
  mem_write=0, alu_src=0, reg_write_en=1
Status: PASS

Test Case 2:  ALU instruction
OpCode: 0010011
Expected outputs:
  branch=0, mem_read=0, mem_to_reg=0, alu_op=00
  mem_write=0, alu_src=1, reg_write_en=1
Actual outputs:
  branch=0, mem_read=0, mem_to_reg=0, alu_op=00
  mem_write=0, alu_src=1, reg_write_en=1
Status: PASS

Test Case 3:  Load instruction
OpCode: 0000011
Expected outputs:
  branch=0, mem_read=1, mem_to_reg=1, alu_op=00
  mem_write=0, alu_src=1, reg_write_en=1
Actual outputs:
  branch=0, mem_read=1, mem_to_reg=1, alu_op=00
  mem_write=0, alu_src=1, reg_write_en=1
Status: PASS

Test Case 4: type instruction
OpCode: 0100011
Expected outputs:
  branch=0, mem_read=0, mem_to_reg=0, alu_op=00
  mem_write=1, alu_src=1, reg_write_en=0
Actual outputs:
  branch=0, mem_read=0, mem_to_reg=0, alu_op=00
  mem_write=1, alu_src=1, reg_write_en=0
Status: PASS

Test Case 5: type instruction
OpCode: 1100011
Expected outputs:
  branch=1, mem_read=0, mem_to_reg=0, alu_op=01
  mem_write=0, alu_src=0, reg_write_en=0
Actual outputs:
  branch=1, mem_read=0, mem_to_reg=0, alu_op=01
  mem_write=0, alu_src=0, reg_write_en=0
Status: PASS

Test Case 6:    Invalid opcode
OpCode: 1111111
Expected outputs:
  branch=0, mem_read=0, mem_to_reg=0, alu_op=00
  mem_write=0, alu_src=0, reg_write_en=0
Actual outputs:
  branch=0, mem_read=0, mem_to_reg=0, alu_op=00
  mem_write=0, alu_src=0, reg_write_en=0
Status: PASS

=== Test Summary ===
Total Tests: 6
Passed: 6
Failed: 0
==================

control_tb.v:184: $finish called at 170000 (1ps)
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$
```

Figure 2.7: Command Window Output for Control Testbench

- `ALUOp` is set based on the instruction type:
  - 10 for R-type instructions.
  - 00 for I-type instructions.
  - 00 for S-type instructions.
  - 01 for branch instructions.
- `MemWrite` is set to 1 if the instruction is a store (sd) instruction for writing into the data memory.
- `ALUSrc` is set to 1 if the ALU's second input is taken from the immediate block. Otherwise, it takes the input from `read_data2`
- `RegWrite` is set to 1 if the instruction requires a write-back, like in the case of a load or R-type instruction.
- We also have an indirect control signal, the `PC_Src`, which chooses between PC + 4 (sequential execution) and PC + offset (for a branch), determined by the zero-flag (`z_flag`) from the ALU and the `branch` signals.

The following table lists the action of each control signal generated by the `Control` block.

Signal name	**Effect when de-asserted**	**Effect when asserted**
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Table 1: Control Signals and Their Effects
Source: Computer Organization and Design, RISC-V Edition

e) Immediate Generation

1. **Input:** `instruction`
2. **Output:** 64-bit sign-extended `immediate` value

This unit extracts the immediate value from the instruction and sign-extends it to 64 bits. The extracted immediate value depends on the instruction type, following the conventions outlined in the *RISC-V Card*.

```
Test Case 1:
Instruction: 0000 0000 1100 0001 0000 0000 1001 0011
Opcode: 0010011
Expected Immediate: 000000000000000c
Actual Immediate: 000000000000000c
Status: PASS

Test Case 2:
Instruction: 1111 1111 1100 0001 0000 0000 1001 0011
Opcode: 0010011
Expected Immediate: fffffffffffffffc
Actual Immediate: fffffffffffffffc
Status: PASS

Test Case 3:
Instruction: 0000 0001 0000 0001 0010 0000 1000 0011
Opcode: 0000011
Expected Immediate: 0000000000000010
Actual Immediate: 0000000000000010
Status: PASS

Test Case 4:
Instruction: 1111 1111 0000 0001 0010 0000 1000 0011
Opcode: 0000011
Expected Immediate: fffffffffffffff0
Actual Immediate: fffffffffffffff0
Status: PASS

Test Case 5:
Instruction: 0000 0010 0001 0001 0010 1010 0010 0011
Opcode: 0100011
Expected Immediate: 0000000000000034
Actual Immediate: 0000000000000034
Status: PASS

Test Case 6:
Instruction: 1111 1110 0001 0001 0010 1110 0010 0011
Opcode: 0100011
Expected Immediate: fffffffffffffffc
Actual Immediate: fffffffffffffffc
Status: PASS

Test Case 7:
Instruction: 0000 0000 0010 0000 1000 1000 0110 0011
Opcode: 1100011
Expected Immediate: 0000000000000010
Actual Immediate: 0000000000000010
Status: PASS

Test Case 8:
Instruction: 1111 1110 0010 0000 1000 1000 1110 0011
Opcode: 1100011
Expected Immediate: fffffffffffffff0
Actual Immediate: fffffffffffffff0
Status: PASS

Test Case 9:
Instruction: 0111 1111 1111 0000 0000 0000 0001 0011
Opcode: 0010011
Expected Immediate: 00000000000007ff
Actual Immediate: 00000000000007ff
Status: PASS

Test Case 10:
Instruction: 1000 0000 0000 0000 0000 0000 0001 0011
Opcode: 0010011
Expected Immediate: fffffffffffff800
Actual Immediate: fffffffffffff800
Status: PASS

Test Case 11:
Instruction: 0000 0000 0000 0000 0000 0000 0111 1111
Opcode: 1111111
Expected Immediate: 0000000000000000
Actual Immediate: 0000000000000000
Status: PASS

=== Test Summary ===
Total Tests: 11
Passed: 11
Failed: 0
==================
```

Figure 2.8: Command Window Output for Imm_Gen Testbench

1. **I-type Instructions (`ld, addi`)**
   - Bits used: imm[11 : 0] (bits 31-20)
   - The immediate value is directly extracted from bits 31-20.
   - It is sign-extended to 64 bits for execution.
2. **S-type Instructions (`sd`)**
   - Bits used: imm[11 : 5] (bits 31-25) and imm[4 : 0] (bits 11-7)
   - The immediate value is split into two parts:
     - Upper part: imm[11 : 5] from bits 31-25
     - Lower part: imm[4 : 0] from bits 11-7
   - The immediate is formed by concatenating these parts and sign-extending it to 64 bits.
3. **B-type Instructions (`beq`)**
   - Bits used: imm[12] (bit 31), imm[10 : 5] (bits 30-25), imm[4 : 1] (bits 11-8), imm[11] (bit 7)
   - The immediate value is formed as follows:

- imm[12] is the most significant bit.
- imm[11] is taken from bit 7.
- imm[10 : 5] comes from bits 30-25.
- imm[4 : 1] comes from bits 11-8.
- imm[0] is 0.

- The immediate is sign-extended and shifted left by 1 to make it word-aligned.

This ensures proper addressing and offset calculations in branch and memory access instructions.

f) ALU Control

1. **Inputs:** 2-bit `ALUOp`, `instruction[30,14-12]`
2. **Output:** 4-bit `ALUControl`

The `ALUControl` unit determines the ALU operation based on the 2-bit `ALUOp` signal and select bits (`instruction[30,14-12]`). While `ALUOp` sets the operation for most instruction types, R-type instructions require these select bits to differentiate operations like *add* and *sub*. The 4-bit `ALUControl` specifies the exact ALU instruction, allowing different formats (e.g., `add`, `ld`) to share the same ALU operation when needed.

```
mpsamartha@Samartha:~/Academics/IPA/sarsaRISCV/SEQ$ vvp a.out
VCD info: dumpfile alu_control_tb.vcd opened for output.
Expected: 0010, Obtained: 0010
Expected: 0110, Obtained: 0110
Expected: 0010, Obtained: 0010
Expected: 0110, Obtained: 0110
Expected: 0000, Obtained: 0000
Expected: 0001, Obtained: 0001
Total Passed Cases: 6 out of 6
alu_control_tb.v:53: $finish called at 60000 (1ps)
```

Figure 2.9: Command Window output for ALU Control Testbench

g) Arithmetic Logic Unit (ALU)

```
Test 1: Addition
a = 0000000000000005
b = 0000000000000003
result = 0000000000000008
Flags: C=z, V=z, N=z, Z=0

Test 2: Subtraction
a = 0000000000000008
b = 0000000000000003
result = 0000000000000005
Flags: C=z, V=z, N=z, Z=0

Test 3: AND
a = ffff0000ffff0000
b = ffffffff00000000
result = ffff000000000000
Flags: C=z, V=z, N=z, Z=0

Test 4: OR
a = ffff0000ffff0000
b = 00000000ffff0000
result = ffff0000ffff0000
Flags: C=z, V=z, N=z, Z=0

Test 5: XOR
a = ffffffffffffffff
b = ffffffffffffffff
result = 0000000000000000
Flags: C=z, V=z, N=z, Z=1

Test 6: SLT
a = ffffffffffffffff
b = 0000000000000000
result = 0000000000000001
Flags: C=z, V=z, N=z, Z=0

Test 7: SRA
a = 8000000000000000
b = 0000000000000001
result = c000000000000000
Flags: C=z, V=z, N=z, Z=0

Test 8: SLL
a = 0000000000000001
b = 0000000000000001
result = 0000000000000002
Flags: C=z, V=z, N=z, Z=0

Test 9: Addition with Overflow
a = 7fffffffffffffff
b = 0000000000000001
result = 8000000000000000
Flags: C=z, V=z, N=z, Z=0

Test 10: Addition of Two Large Negative Numbers
a = 8000000000000000
b = 8000000000000000
result = 0000000000000000
Flags: C=z, V=z, N=z, Z=1

Test 11: Subtraction with Underflow
a = 8000000000000000
b = 0000000000000001
result = 7fffffffffffffff
Flags: C=z, V=z, N=z, Z=0

Test 12: SLTU with Equal Values
a = ffffffffffffffff
b = ffffffffffffffff
result = 0000000000000000
Flags: C=z, V=z, N=z, Z=1

Test 13: SRA with Maximum Shift
a = 8000000000000000
b = 000000000000003f
result = ffffffffffffffff
Flags: C=z, V=z, N=z, Z=0

Test 14: SLL with Overflow
a = 4000000000000000
b = 0000000000000001
result = 8000000000000000
Flags: C=z, V=z, N=z, Z=0

Test 15: AND with Alternating Bits
a = aaaaaaaaaaaaaaaa
b = 5555555555555555
result = 0000000000000000
Flags: C=z, V=z, N=z, Z=1

Test 16: SLT with Extreme Values
a = 8000000000000000
b = 7fffffffffffffff
result = 0000000000000001
Flags: C=z, V=z, N=z, Z=0
```

Figure 2.10: Command Window Output for ALU Testbench

1. **Inputs:** `input1`, `input2`, `control_signal`
2. **Outputs:** `result`, `zero_flag`

The ALU takes two input values and executes the operation specified by the control signal from the ALU Control unit. The result output is stored in the `result`, and if the result is zero, the `zero_flag` is set, which is used for branch decisions.

h) Data Memory

1. **Inputs:** `clk`, `reset`, `address`, `Write_data`, `MemRead`, `MemWrite`
2. **Output:** `read_data`

The data memory block interacts with the processor to load and store values. Initially, we set the memory to store only 0s.

- If `MemWrite` is asserted, the value in `Write_data` is written to memory at the specified `address`.
- If `MemRead` is asserted, data from the given `address` is read and stored in `read_data`.

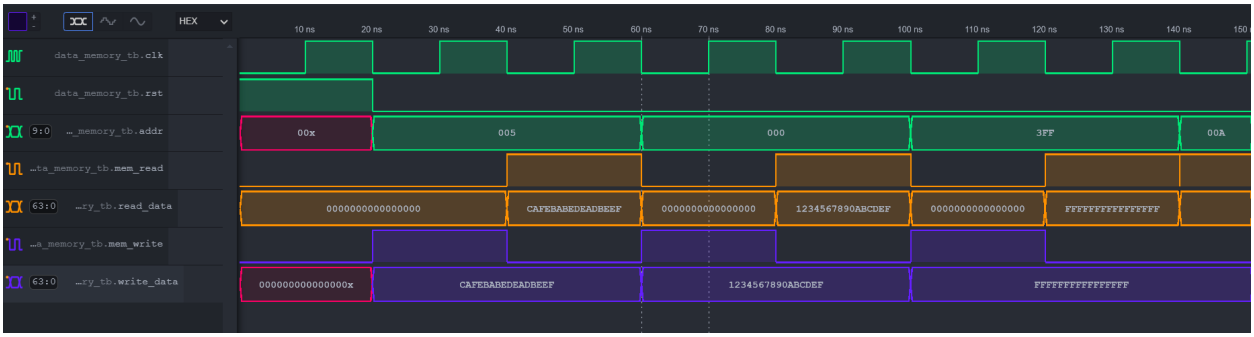


Figure 2.11: GTK-Wave Output for Data Memory Testbench

i) Multiplexers (MUXes)

Apart from the main datapath units, MUXes are placed in three locations:

1. **ALU Second Input Selection**: Selects between the immediate value (from Immediate Generation block) or a register value (from Register File), controlled by `ALUSrc`.
2. **WriteBack Stage Selection**: Selects between the ALU result (for R-type instructions) or the Data Memory output (for load instructions), controlled by `MemtoReg`.
3. **Branch Target Address Selection**: Selects between `PC + 4` (sequential execution) or the branch target address (from Immediate Generation block), controlled by the logical AND of `Branch` and `zero_flag`, which is the `PC_Src`.

j) Adder Block

The adder blocks are implemented using a carry look-ahead adder and serve two primary purposes:

- Calculating the next sequential program counter (`PC + 4`).
- Computing the branch target address when a branch instruction is executed.

3 The Sequential Processor

Combining these datapath units gives us the final design of the sequential processor, as shown in 2.1. This implementation supports the following RISC-V instructions: `addi`, `and`, `sub`, `add`, `or`, `ld`, `sd`, and `beq`. To test and validate the functionality of the final Sequential Processor, we run the following Testing Codes. These are carefully designed so as to check for edge-cases and other such conditions under which our processor could be susceptible to fail. NOTE that our sequential processor runs for an extra clock cycle to detect the end of the program.

4 Testing Codes

The test files contain various RISC-V assembly instructions to verify functionality. Each file is structured into two parts:

- `Name_exp.txt`: Contains the human-readable assembly instructions with comments.
- `Name_Code.txt`: Contains the byte-addressed hexadecimal machine code.

The machine code is automatically generated using the Python script `riscv_instruction_encoder.py`, and the respective file path is specified in `instruction_memory.v`.

a) Test Cases

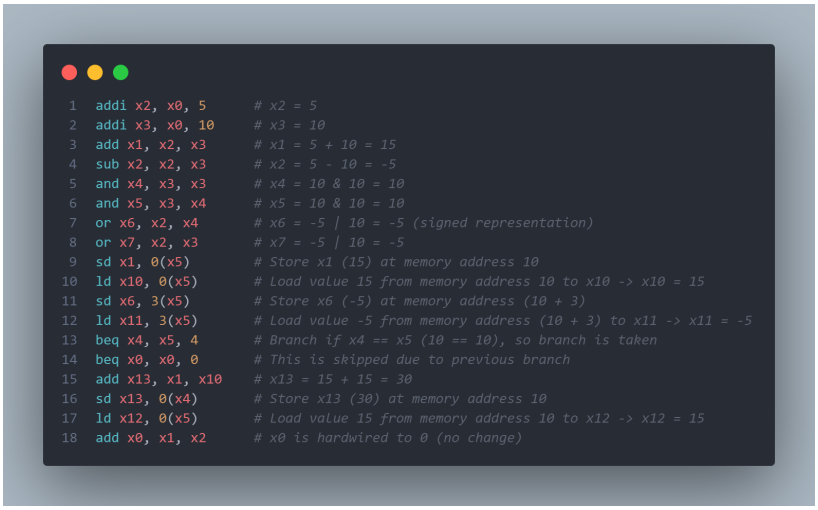
1. **Test_Basic_exp.txt**: Verifies arithmetic, logical, store (`sd`), load (`ld`), and branch (`beq`) instructions (18 instructions). Includes an edge case where a value is written to register `x0`.
2. **Test_SumN_exp.txt**: Computes the sum of the first N natural numbers (9 instructions).
3. **Test_Vector_Add.txt**: Implements vector addition (34 instructions).
4. **Test_Fibonacci_exp.txt**: Generates the first 10 Fibonacci numbers (28 instructions).
5. **Test_LinearSearch_exp.txt**: Performs linear search on an array and stores the zero-based index of the element found (9 instructions).
6. **Test_Overflow_exp.txt**: Brute forces an overflow case by handling large numbers by repeated addition. The sum of positive numbers eventually becomes a negative number, which is an overflow.
7. **Test_FaultInstruction_exp.txt**: Similar to **Test_Basic_exp.txt** but includes a faulty instruction outside of the support (19 instructions). The output remains unaffected by the fault.

5 Results of Testing Codes

a) Test_Basic_Code

Register File Contents:

```
x0:   0000000000000000
x1:   000000000000000f
x2:   fffffffffffffffb
x3:   000000000000000a
x4:   000000000000000a
x5:   000000000000000a
x6:   fffffffffffffffb
x7:   fffffffffffffffb
x8:   0000000000000000
x9:   0000000000000000
x10:  000000000000000f
x11:  fffffffffffffffb
x12:  000000000000001e
x13:  000000000000001e
```



```
addi x2, x0, 5      # x2 = 5
addi x3, x0, 10     # x3 = 10
add x1, x2, x3      # x1 = 5 + 10 = 15
sub x2, x2, x3      # x2 = 5 - 10 = -5
and x4, x3, x3      # x4 = 10 & 10 = 10
and x5, x3, x4      # x5 = 10 & 10 = 10
or x6, x2, x4       # x6 = -5 | 10 = -5 (signed representation)
or x7, x2, x3       # x7 = -5 | 10 = -5
sd x1, 0(x5)        # Store x1 (15) at memory address 10
ld x10, 0(x5)       # Load value 15 from memory address 10 to x10 -> x10 = 15
sd x6, 3(x5)        # Store x6 (-5) at memory address (10 + 3)
ld x11, 3(x5)       # Load value -5 from memory address (10 + 3) to x11 -> x11 = -5
beq x4, x5, 4       # Branch if x4 == x5 (10 == 10), so branch is taken
beq x0, x0, 0       # This is skipped due to previous branch
add x13, x1, x10    # x13 = 15 + 15 = 30
sd x13, 0(x4)       # Store x13 (30) at memory address 10
ld x12, 0(x5)       # Load value 15 from memory address 10 to x12 -> x12 = 15
add x0, x1, x2      # x0 is hardwired to 0 (no change)
```

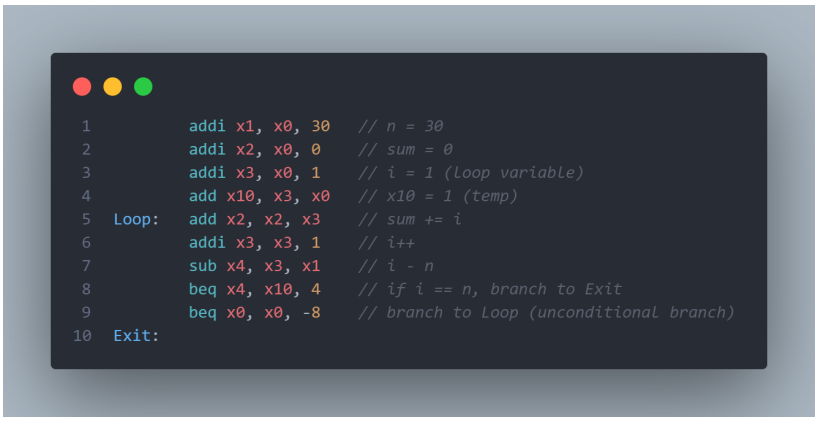
Figure 5.1: Assembly Instructions for `Test_Basic_Code`

As explained in the comments above, we have performed some basic Arithmetic (addi/add/-sub/or/and), Memory (ld/sd) and Branch (beq) instructions for verifying the functionality of the processor. The Register values on the left are 64-bit Hex numbers. They match the expected results and the test is successful.

b) Test_Sum_Numbers

Register File Contents:

```
Register File Contents:
x0:   0000000000000000
x1:   000000000000001e
x2:   00000000000001d1
x3:   000000000000001f
x4:   0000000000000001
x5:   0000000000000000
x6:   0000000000000000
x7:   0000000000000000
x8:   0000000000000000
x9:   0000000000000000
x10:  0000000000000001
```



```
        addi x1, x0, 30   // n = 30
        addi x2, x0, 0    // sum = 0
        addi x3, x0, 1    // i = 1 (Loop variable)
        add x10, x3, x0   // x10 = 1 (temp)
Loop:   add x2, x2, x3    // sum += i
        addi x3, x3, 1    // i++
        sub x4, x3, x1    // i - n
        beq x4, x10, 4    // if i == n, branch to Exit
        beq x0, x0, -8    // branch to Loop (unconditional branch)
Exit:
```

Figure 5.2: Assembly Instructions for `Test_Sum_Numbers`

The above code calculates the sum of first N natural numbers. Here we run the code for **N = 30**. The expected result is thus 465. The final sum is stored in `x2`. Here we see that we get `1D1`, this actually translates to 465 in Decimal. Hence the test is successful.

c) Test_Vector_Add

```
 Register File Contents:
x0:   0000000000000000
x1:   0000000000000000
x2:   0000000000000000
x3:   0000000000000005
x4:   0000000000000005
x5:   000000000000000a
x6:   0000000000000014
x7:   0000000000000028
x8:   0000000000000000
x9:   0000000000000000
x10:  0000000000000003
x11:  0000000000000009
x12:  0000000000000000
x13:  0000000000000000
x14:  0000000000000000
x15:  0000000000000008
x16:  000000000000000A
x17:  000000000000000C
x18:  0000000000000000
x19:  0000000000000000
x20:  000000000000000e
x21:  0000000000000018
x22:  0000000000000000
x23:  0000000000000000
x24:  0000000000000000
x25:  000000000000002c
```

```
1         addi x4, x0, 5      // x4 = 5
2         add x5, x4, x4      // x5 = 10
3         add x6, x5, x5      // x6 = 20
4         add x7, x6, x6      // x7 = 40
5         addi x10, x10, 1    // x10 = 1
6         addi x11, x0, 7     // x11 = 7
7         sd x10, 0(x5)       // store 1 at mem[10]
8         sd x11, 0(x6)       // store 7 at mem[20]
9         add x11, x11, x10   // x11 = 8
10        add x10, x10, x10   // x10 = 2
11        sd x10, 1(x5)       // store 2 at mem[11]
12        sd x11, 1(x6)       // store 8 at mem[21]
13        addi x10, x0, 3     // x10 = 3
14        addi x11, x0, 9     // x11 = 9
15        sd x10, 2(x5)       // store 3 at mem[12]
16        sd x11, 2(x6)       // store 9 at mem[22]
17  Loop: add x20, x5, x3     // calculate address to be accessed of A
18        add x21, x6, x3     // calculate address to be accessed of B
19        ld x22, 0(x20)      // load A[i]
20        ld x23, 0(x21)      // load B[i]
21        add x24, x22, x23   // compute sum
22        add x25, x7, x3     // calculate address to be accessed of C
23        sd x24, 0(x25)      // store the sum
24        addi x3, x3, 1      // increment loop var
25        beq x4, x4, 4       // jump to Exit if loop var == 5
26        beq x0, x0, -18     // jump to Loop (unconditional)
27  Exit: ld x15, 0(x7)       // load C[0]
28        ld x16, 1(x7)       // load C[1]
29        ld x17, 2(x7)       // load C[2]
```

Figure 5.3: Assembly Instructions for `Test_Vector_Add`

The above code adds two vectors A and B, which are of length 5 stored in the Memory Location `x5 = 10` and `x6 = 20` respectively. First, we store `A = [1, 2, 3, 0, 0]` and `B = [7, 8, 9, 0, 0]` at their memory location. Then, we run a loop to iterate through the vectors in location. In each loop, we load the values, perform the sum, and then store back the value at the memory location base address specified by `x7 = 40` with the offset given by the loop variable. Note that the `Data Memory` is an array of size 1024 with each array of width 64 bits (8 bytes), hence we only need to increment the address by 1 for accessing the next location. Thus, we skip the multiplication by 8 instruction, which is performed when Data Memory is Byte addressed.

We have two branch (`beq`) conditions, one for the exit and the other for the Uncoditional Branching. Once the operation is complete, to view the output, we load the Sum stored in the Memory into Registers, `x15, x16, x17`. The expected result is `C = [8, 10, 12, 0, 0]`. As seen in the register file above, we see that the output is indeed right. Their Hexadecimal values correspond to the expected result. Hence, the test is successful.

d) Test_Fibonacci_Sequence

Register File Contents:

```
Register File Contents:
x0:   0000000000000000
x1:   000000000000000a
x2:   0000000000000000
x3:   0000000000000015
x4:   0000000000000022
x5:   000000000000000a
x6:   0000000000000022
x7:   0000000000000000
x8:   0000000000000000
x9:   0000000000000000
x10:  0000000000000000
x11:  0000000000000001
x12:  0000000000000001
x13:  0000000000000002
x14:  0000000000000003
x15:  0000000000000005
x16:  0000000000000008
x17:  000000000000000d
x18:  0000000000000015
x19:  0000000000000022
```

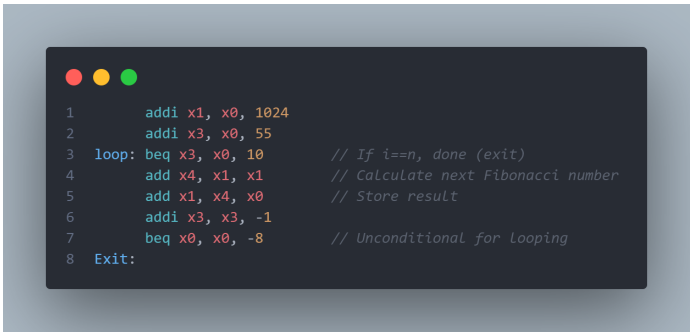
```
1           addi x1, x0, 10       // Calculate 8 Fibonacci numbers
2           addi x2, x0, 0        // Address for storing results
3           addi x3, x0, 0        // Fib(0) = 0
4           addi x4, x0, 1        // Fib(1) = 1
5           sd x3, 0(x2)          // Store Fib(0)
6           addi x2, x2, 1        // Increment address (changed from 8)
7           sd x4, 0(x2)          // Store Fib(1)
8           addi x2, x2, 1        // Increment address (changed from 8)
9           addi x5, x0, 2        // i = 2
10 fib_loop: beq x5, x1, 16       // If i==n, done
11          add x6, x3, x4          // Calculate next Fibonacci number
12          sd x6, 0(x2)          // Store result
13          addi x2, x2, 1        // Increment address (changed from 8)
14          addi x5, x5, 1        // i++
15          add x3, x0, x4        // a = b
16          add x4, x0, x6        // b = result
17          beq x0, x0, -14       // Loop back
18 fib_done: add x2, x0, x0       //iterating address for loading
19          ld x10, 0(x2)         //put the series onto the registers (x10-x19)
20          ld x11, 1(x2)
21          ld x12, 2(x2)
22          ld x13, 3(x2)
23          ld x14, 4(x2)
24          ld x15, 5(x2)
25          ld x16, 6(x2)
26          ld x17, 7(x2)
27          ld x18, 8(x2)
28          ld x19, 9(x2)
```

Figure 5.4: Assembly Instructions for `Test_Fibonacci_Sequence`

The above code is for generating the first 10 number of the Fibonacci Sequence. We initially store the first 2 elements of the sequence viz. 0 and 1 in the registers `x3, x4`. Then we store the next 8 elements of the sequence in Memory in a loop. At the end of the iteration, we check for the Exit condition, then with an unconditional branch we loop back and perform the next iteration. Once 8 iterations, are done, we move to `fib_done`, then load the values onto the registers to view them on the terminal window. As seen in the output of the register files (`x10-x19`) above (64 bit Hexadecimal Numbers), we clearly see that the sequence is generated as expected. Hence, the test is successful.

e) Test_Overflow_Code

Now we look at one brute-force case where overflow will occur in the processor. Since the registers are of 64-bits wide, to cause the overflow, we would require very large numbers. Currently the I-type instruction only support 12-bit immediate value, thus to reach large values, we repeatedly, add 2^{11} to itself. This is essentially multiplication by 2 and in around 52 such iterations, the value would reach 2^{63}. This is actually an Overflow case and is interesting to analyse, because addition of positive numbers has lead to a negative number. Our ALU has a built-in Overflow flag which can be asserted to indicate. Although currently not implemented, it can be used in wrapper to detect such faults during arithmetic.



```
1        addi x1, x0, 1024
2        addi x3, x0, 55
3   loop: beq x3, x0, 10        // If i==n, done (exit)
4        add x4, x1, x1         // Calculate next Fibonacci number
5        add x1, x4, x0         // Store result
6        addi x3, x3, -1
7        beq x0, x0, -8         // Unconditional for Looping
8   Exit:
```

Figure 5.5: Assembly Instructions for Overflow case

```
R-type instruction

Cycle 269:
PC: 000000000000000c
Instruction: 00108233
ALU Output: 0000000000000000
Register Write Enable: 1
Memory Write Enable: 0
Branch: 0

x4: 8000000000000000

==============================

R-type instruction

Cycle 270:
PC: 0000000000000010
Instruction: 000200b3
ALU Output: 0000000000000000
Register Write Enable: 1
Memory Write Enable: 0
Branch: 0

x4: 0000000000000000

==============================
```

Figure 5.6: Command Window output for Overflow case

Above on the left is the assembly instructions which repeatedly multiplies $2^1 1$ with 2 by adding the register value to itself and storing in it again. On the write is the command window output where we see that the register `x4` has acquired a negative value, according to the signed notation in Clock cycle - 269. Following that, in the next clock cycle, this 1 is shifted again to get a 0. Hence such large numbers arithmetic has to be carefully handled.

f) Test_Fault_Instruction

This is a simple test to check if the processor is affected by faulty instructions. Currently the processor doesn't support the `slli`. We just appended this instruction to the `Test_Basic_Code` to check for errors. The processor indeed worked as it was supposed to and executed the earlier instructions without any problems and skipped this faulty instruction as it did not recognize it.

6 Pipeline Implementation

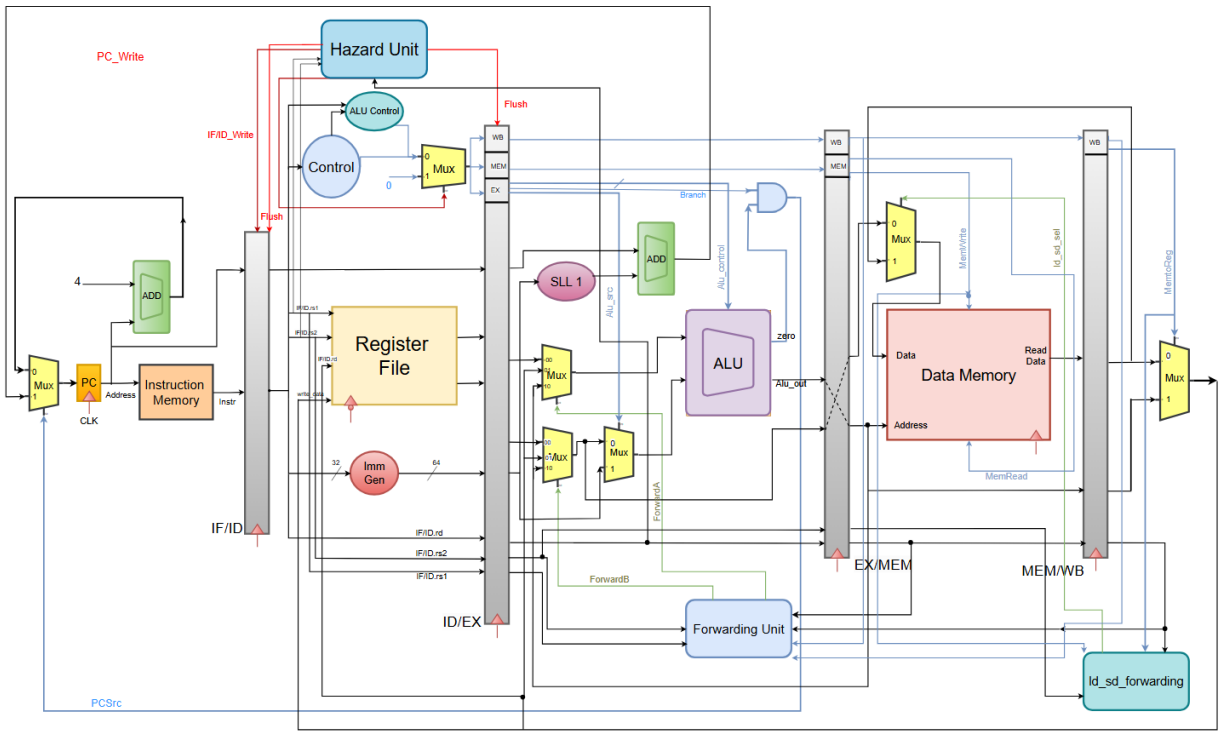


Figure 6.1: Pipelined Processor with forwarding and hazard detection units.

Introduction

We discussed the sequential design in which you do not start the execution of the next instruction until the previous one has completed. But what if we were to break down the full chain into 5 clock cycles such that the clock frequency is now, $< 1/\max(\text{T\_IF, T\_ID, T\_EX, T\_MEM, T\_WB})$ and the design was pipelined (an implementation technique in which multiple instructions are overlapped in execution).

A pipeline processor generally increases throughput at the cost of latency, which is a good tradeoff. It is an extension of the sequential processor. We divide the sequential processor into five stages:

- **Instruction Fetch (IF)**
- **Instruction Decode (ID)**
- **Execute (EX)**
- **Memory (MEM)**
- **Write Back (WB)**

We add registers (called **pipeline registers**) with appropriate inputs and outputs between these stages and connect them accordingly. Pipeline registers are added to store intermediate values between the stages. These registers are synchronous and update at the rising edge of the clock,

except for the ID/EX register file. Unless explicitly stated otherwise, all registers have a clock and reset as inputs. Since the pipeline is an extension of the sequential as mentioned above, the working of the datapath units here is the same as it was in the sequential datapath. We also have a `flush` signal in the case of a control hazard, in which case we pass a `nop` for a misprediction in the branch and continue in the pipeline. Similar to the sequential implementation, the pipeline also takes an extra clock cycle.

a) IF/ID Pipeline Register

1. **Inputs:** 32-bit `instr`, 64-bit `pc_out`
2. **Outputs:** 32-bit `instr_IF_ID`, 64-bit `IF_ID_pc_out`

The instruction from the instruction memory and the program counter value are stored in the IF/ID register file. The instruction is then passed to the ID stage, while the PC value is used to fetch the next instruction.

b) ID/EX Register File

1. **Inputs:**
   64-bit `IF_ID_pc_out`, `read_data1`, `read_data2`, `imm`
   5-bit `rs1`, `rs2`, `rd`
   1-bit `reg_write_en_out_mux`, `mem_read_out_mux`, `mem_to_reg_out_mux`, `mem_write_out_mux`, `alu_src_out_mux`, `reg_write_out_mux`, `branch_out_mux`
   4-bit `op_out_mux`
2. **Outputs:** Similar names as the input, prefixed with `ID_EX`

This is an important pipeline register because it houses the control signals generated by the control unit. The inputs with the suffix, `_out_mux` are the ones coming out of the `control_mux` which is used to decide between the typical control signals coming out of the control, and 0s (to be passed as control signals for `nops`). Apart from the control signals, we also have the `read_data1/read_data2` being passed from the register file. (NOTE: the register file is written on the **negative edge** of the clock cycle, to allow simultaneous writes and reads, in the same cycle). `rs1, rs2` are stored for forward checks, and `rd` too, alongside for it's need to point to the register it needs to write to in the WB stage.

Note that instead of `ALUOp` being passed to the ID/EX pipe-reg, we pass the output of the ALU control (which is present in the ID stage). The ID/EX register thus stores values of registers, immediate values, opcodes, and control signals for execution.

c) EX/MEM Pipeline Register

1. **Inputs:**
   1-bit `mem_to_reg_ID_EX`, `reg_write_en_ID_EX`, `mem_read_ID_EX`, `mem_write_ID_EX`
   64-bit `alu_out`, `alu_in_2`
   5-bit `rs2_ID_EX`, `rd_ID_EX`
2. **Outputs:** Similar names suffixed with `EX_MEM`

In this pipeline register, the results of the EX stage are stored, which include the ALU results (`alu_out`), `read_data2` for the case of a `sd` instruction. It also has the same `rd_ID_EX` for the write-back register address from the previous pipeline register. The EX/MEM register thus stores ALU results, control signals, and memory access data, passing them to the MEM stage.

d) MEM/WB Pipeline Register

1. **Inputs:** 1-bit `mem_to_reg_EX_MEM`, `reg_write_en_EX_MEM`
   64-bit `data`, `alu_out_EX_MEM`
   5-bit `rd_EX_MEM`

2. **Outputs:** Similar names prefixed with `MEM_WB`

The MEM/WB register file stores the final ALU result and memory data to be selected and written back to the register file. It also houses the control signals and the destination register `rd_EX_MEM` required in the WB stage.

e) Forwarding Unit

Some data hazards can be solved by forwarding (also called bypassing) a result from the MEM or WB stage to a dependent instruction in the EX stage. This does require additional hardware, viz., adding multiplexers in front of the ALU to select its operands from the register file or the Memory or Writeback stage. We have also added the functionality of checking for load-store data hazards, wherein we need to forward from the MEM/WB stage to the MEM stage, as given in the examples below. This hazard alone needs a MUX at the memory access stage itself to choose from the forwarded data or the typical data coming from the `EX/MEM` pipeline register. For this purpose of selecting the operands, we have the forwarding unit, checking for data hazards and forwarding data appropriately. The cases where we require forwarding are listed below.

1. Data that is used in an operation is changed in the previous instruction. In this case, we have to forward the value from the EX/MEM register file to the EX stage:

   ```
   add x1, x2, x3
   sub x4, x1, x5
   ```

   The scenario also holds for the second instruction replaced by, `sd x1, 0(x4)`, **OR** `sd x4, 0(x1)`, **OR** `ld x4, 0(x1)`, **OR** `beq x1, x4, 0x4`.

2. Data that is used in an operation is changed in the previous two instructions. In this case, we have to forward the value from the MEM/WB register file to the EX stage:

   ```
   add x1, x2, x3
   add x4, x5, x6
   sub x6, x1, x7
   ```

3. We need forwarding from the MEM/WB stage to the MEM stage if we have a load instruction followed by a store instruction:

   ```
   ld x1, 0(x2)
   sd x1, 0(x3)
   ```

If a *double data hazard* occurs, EX/MEM pipe register values take priority over the MEM/WB pipe register because it houses the more recent value!

However, there are a few data hazards where we need to stall the pipeline as well, along with forwarding. This is the load-use data hazard.

Load-Use Data Hazard

Forwarding is sufficient to solve RAW (read-after-write) data hazards when the result is computed in the EX stage of an instruction because its result can then be forwarded to the EX stage of the next instruction. Unfortunately, the `ld` instruction does not finish reading data until the end of the MEM stage, so its result cannot be forwarded to the EX stage of the next instruction (we thus say that the `ld` instruction has a **two-cycle latency** because a dependent instruction cannot use its result until two cycles later).

So, when a load instruction is followed by an instruction that uses the loaded value, we need to stall the pipeline by inserting a bubble in the pipeline.

- Example 1:

```
ld  x1, 0(x2)
add x3, x1, x4
```

- Example 2:

```
ld  x1, 0(x2)
beq x1, x3, 0x4
```

- Example 3:

```
ld  x1, 0(x2)
ld  x3, 0(x1)
```

- Example 4:

```
ld  x1, 0(x2)
sd  x4, 0(x1)
```

The inputs and outputs of the forwarding unit are explained below.

1. **Inputs:** 5-bit `ID_EX_rs1`, `ID_EX_rs2`, `EX_MEM_rd`, `MEM_WB_rd`
   1-bit `EX_MEM_reg_write_en`, `MEM_WB_reg_write_en`
2. **Outputs:** 2-bit `ForwardA` , `ForwardB`

The inputs to the ALU are selected by this forwarding unit. The forwarding logic is given below:

```
EX hazard:
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and
    (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
        ForwardA = 10
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and
    (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
        ForwardB = 10
```

The EX hazard check essentially checks if the `rd` in EX_MEM is not the same as the source `rs1/rs2` in ID_EX while ensuring that we are writing. Also, we need to make sure that we don't write into a register which isn't x0!

```
MEM hazard:
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0) and
    not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and
    (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) and
    (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
        ForwardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0) and
    not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and
    (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) and
    (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
        ForwardB = 01
```

The MEM hazard check also similarly checks if the `rd` in MEM_WB is not the same as the source `rs1/rs2` in EX_MEM while ensuring that we are writing. Also, we need to make sure that we don't write into a register which isn't x0! In addition to this, to handle double data hazards, we also make sure that the forwarding condition from EX_MEM is not true because that forwarding has a higher priority.

The load-store data hazard is handled by another forwarding unit with the inputs and outputs mentioned below.

Load-Store Forwarding Unit

1. **Inputs:** 5-bit `ld_rd`, 64-bit `sd_rs2_data`
   1-bit `ld_sd_mem_to_reg`, `ld_sd_mem_write`

2. **Outputs:** 1-bit `ld_sd_sel`

```
if (ld_sd_mem_to_reg &&
    (ld_rd == sd_rs2_data) &&
    (ld_rd != 5'b0) && ld_sd_mem_write)
        ld_sd_sel_reg = 1'b1;        // Forward from MEM/WB
```

The select line is for a multiplexer that selects between the data from the MEM/WB register file and the data from the data memory, which is read in the MEM stage. This completes dependency checks for cases that can be solved by forwarding.

f) Hazard Detection Unit

As mentioned above, certain scenarios require pipeline stalls, which hold up operations until the data is available. The hazard detection unit is essential to handle control hazards. It checks for control hazards and stalls/flushes the pipeline if required.

The cases where a stall is required are often with forwarding. These cases are listed in the forwarding section. We will discuss the case of a flush here in case of a branch. **We assume that the branch is not taken when we fetch the instruction**. If the prediction turns out to be wrong, i.e., the branch is taken, we have to flush the pipeline. In this case, we have to change the PC value to the branch target address and insert `nops` in the pipeline. This is done by flushing

the IF/ID and ID/EX register files since they will already have the two wrong instructions fetched after the branch instruction.

The inputs and outputs of this unit are specified below.

1. **Inputs:** 5-bit `IF_ID_rs1`, `IF_ID_rs2`,`ID_EX_rd`
   1-bit `ID_EX_mem_read`, `ld_sd_mem_write`, `ld_sd_mem_read`
2. **Outputs:** 1-bit `pc_write`, `IF_ID_write`, `control_mux_sel`

The types of cases which lead to stalls are listed below.

1. **Load-Use Hazard**: A load instruction followed by an instruction that uses the loaded value.

```
ld  x1, 0(x2)
add x3, x1, x4
```

2. **Control Hazard**: A branch instruction followed by an instruction modifying the PC.

```
beq x1, x2, 0x4
add x3, x4, x5
```

It is very important to hold the state of the registers and memories in such cases. The bubble is introduced by zeroing out the EX stage control signals during an ID stage stall so that the bubble performs no action and changes **no architectural state**. Pipeline bubbles are inserted to handle these hazards efficiently.

Explaining the hazard unit logic for a **STALL**, we have the pseudo-code below.

```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
     (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
    pc_write = 0;
    IF_ID_write = 0;
    control_mux_select = 1;
    flush = 0
```

Note that we don't flush for a stall, so `flush` remains 0 while setting all control signals to 0 for the rest of the pipeline from ID/EX onwards. We also do not update `pc` and `IF/ID`, so the execution can be performed as is after the stalling is done. Thus, in the pseudo-code, we check if we are reading from the memory (in the case of a `ld`) and then check if the next instruction after it uses the register in which the data is loaded from in the previous instruction.

Now, this does execute instructions correctly, but we can optimise it for the special cases mentioned below.

1. **Load followed by a load**: There is no `rs2` for a load, but it corresponds to the lower 5 bits of the immediate in the I-type instruction corresponding to `ld`, so there could be an accidental match. If `rs1` had to match the `rd` of the previous load, then it's a stall regardless.
2. **Load followed by a store**: `rs2` of the store is the same as `rd` of the load, so this can be forwarded. If `rs1` is the same as `rd` of the load, then a stall is needed anyway.

These cases are currently being stalled by the logic above, but we can forward the data from the MEM/WB to the memory access stage, avoiding stalling of the processor. Thus, we want the hazard detection unit to ignore such cases so the forwarding unit can handle them. Hence, the updated stall condition is:

```
if (ID_EX_mem_read &&
    ((ID_EX_rd == IF_ID_rs1) || (ID_EX_rd == IF_ID_rs2 &&
    !(ld_sd_mem_read || ld_sd_mem_write))) &&
    (ID_EX_rd != 5'b0))
    // update write, select and flush signals as above.
```

The signals `ld_sd_mem_read, ld_sd_mem_write` correspond to the second instruction, i.e. the one after the load. For the case of a store following the load, the `ld_sd_mem_write` will be asserted, thus failing the entire stall condition. For the case of the load followed by a load, if the lower 5 bits of the second load do correspond with the `rd` of the load before it, we suppress this asserted stall condition by checking it with the logical NOT of the `ld_sd_mem_write`, which would evaluate to 0.

We thus efficiently handle data hazards by letting the processor forward when it can. For the cases that it cannot, we have the flush, explained below.

Now, coming to the case of a **FLUSH**, which is also handled by the hazard-detection unit, we know that we need to branch once the `pc_src` corresponding to the branch instruction in the EX stage is asserted. Thus, the logic is simple enough. If the `pc_src` is asserted, we assert flush, which sets all data in the IF/ID and ID/EX pipeline registers to 0.

After integrating the blocks with the pipeline registers, we get the **final pipelined processor**, as shown in the Figure 6.1 The instructions supported by this five-stage pipelined processor are `addi, and, sub, add, or, ld, sd, beq`. NOTE that we have to add four extra dummy instructions for successful completion.

7 Results

a) Forwarding

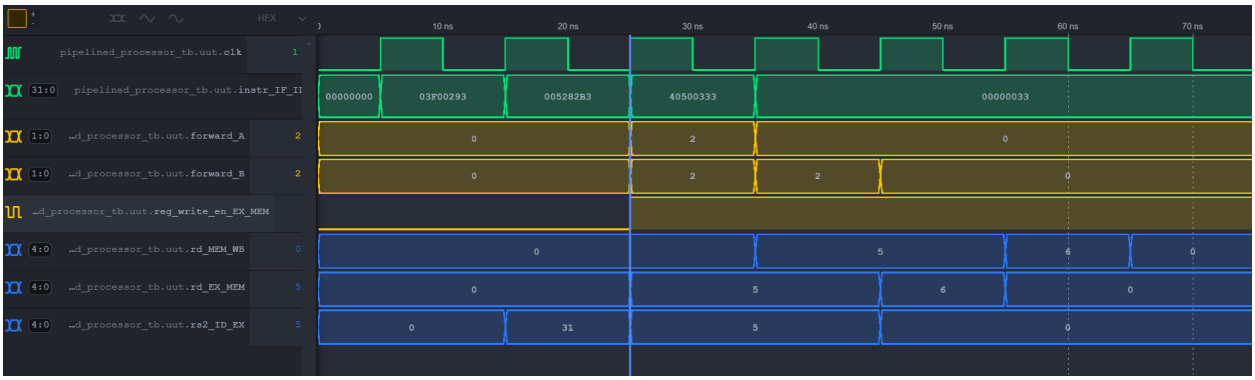


Figure 7.1: GTK-Wave Output for Forwarding

The value of `rd` in the EX/MEM stage matches `rs2` in the ID/EX stage. We also see that the `reg_write_en_EX_MEM` is asserted. This satisfies the conditions for a forward, and we see that the lines `forward_A` and `forward_B` are set to appropriate values and the values are forwarded.

b) Stall

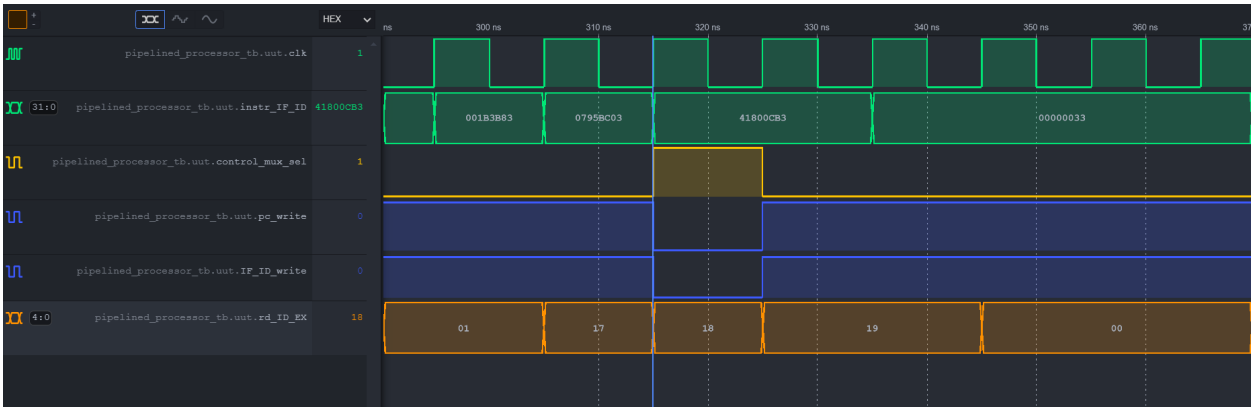


Figure 7.2: GTK-Wave Output for Stall

We observe in this case that the `rd_ID_EX` value matches the `rs2` value in the IF/ID file(not plotted). After checking the other conditions, we determine this is a stall. The control is set to 0, and `pc_write` and `IF_ID_write` are de-asserted. We can see that the instruction is stalled for another cycle.

c) Flush

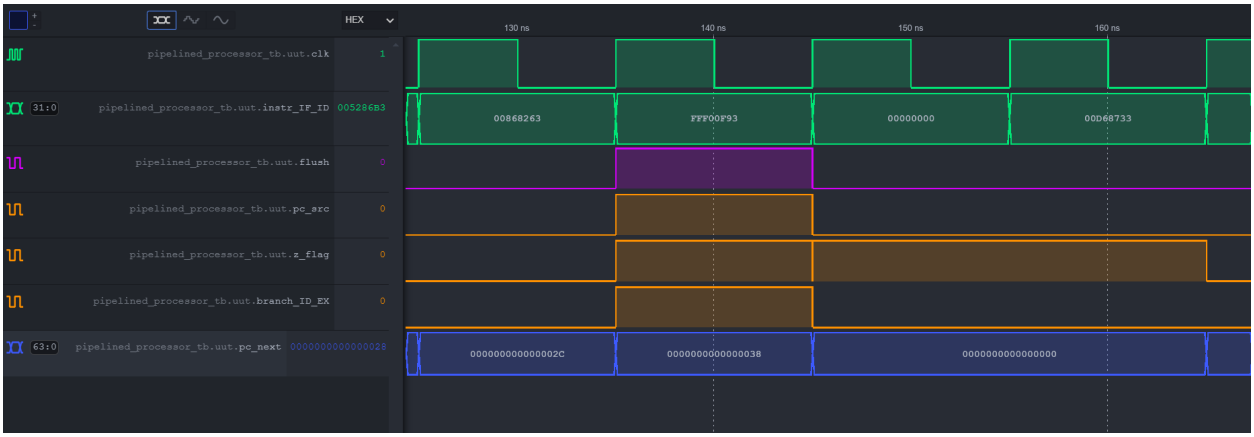


Figure 7.3: GTK-Wave Output for Flush

If a branch is true, our prediction that the branch is not taken is false, and we have to flush the pipeline. We see that when the `z_flag` is high and the branch condition is asserted, the flush lines are set. The flush is synchronous. We see that the `instr_IF_ID` register value is being set to 0 at the next clock edge. The `pc_next` changes to the new value, and the execution proceeds normally. **The penalty for a flush is 2 clock cycles**.

d) Stall Followed by Flush

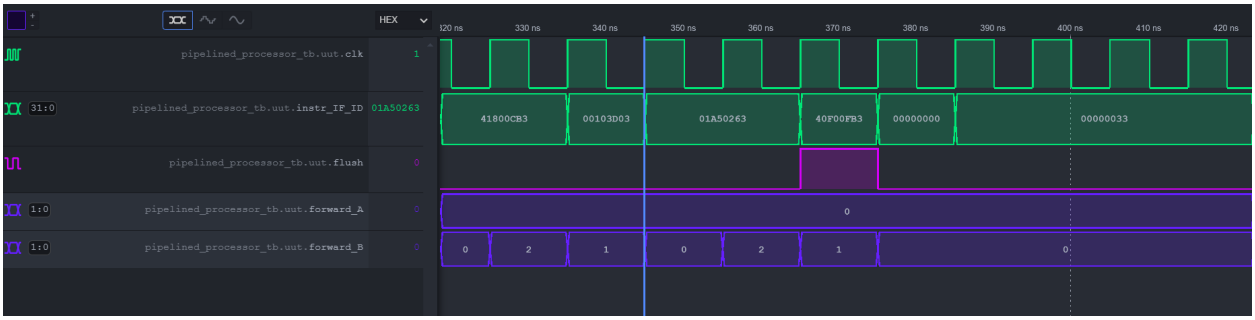


Figure 7.4: GTK-Wave Output of a Stall followed by a flush

This is just a test case where we test for a flush right after a stall condition. We see that the stall happens from the `instr_IF_ID` value, which is followed by an asserted flush line. The value of the `instr_IF_ID` register after the flush clock cycle is rightly set to 0.

To compare the number of clock cycles taken by the sequential processor and the pipelined processor, we run the same code and compare the results.

```
=== Final Processor State ===
Total Cycles: 46

Register File Contents:
x0: 0000000000000000
x1: ffffffffffffff04
x2: 0000000000000011
x3: 0000000000000011
x4: ffffffffffffff04
x5: 000000000000007e
x6: ffffffffffffff82
x7: ffffffffffffff82
x8: 00000000000000fc
x9: ffffffffffffff82
x10: 0000000000000011
x11: 0000000000000011
x12: ffffffffffffff82
x13: 00000000000000fc
x14: 00000000000001f8
x15: 000000000000007e
x16: 0000000000000276
x17: 0000000000000276
x18: 0000000000000762
x19: 0000000000000762
x20: ffffffffffffff82
x21: ffffffffffffff82
x22: 0000000000000011
x23: ffffffffffffff82
x24: ffffffffffffff82
x25: 000000000000007e
x26: 0000000000000011
x27: 0000000000000011
x28: ffffffffffffff82
x29: 0000000000000012
x30: ffffffffffffff82
x31: 0000000000000000
```

```
=== Final Processor State ===
Total Cycles: 58

Register File Contents:
x0: 0000000000000000
x1: ffffffffffffff04
x2: 0000000000000011
x3: 0000000000000011
x4: ffffffffffffff04
x5: 000000000000007e
x6: ffffffffffffff82
x7: ffffffffffffff82
x8: 00000000000000fc
x9: ffffffffffffff82
x10: 0000000000000011
x11: 0000000000000011
x12: ffffffffffffff82
x13: 00000000000000fc
x14: 00000000000001f8
x15: 000000000000007e
x16: 0000000000000276
x17: 0000000000000276
x18: 0000000000000762
x19: 0000000000000762
x20: ffffffffffffff82
x21: ffffffffffffff82
x22: 0000000000000011
x23: ffffffffffffff82
x24: ffffffffffffff82
x25: 000000000000007e
x26: 0000000000000011
x27: 0000000000000011
x28: ffffffffffffff82
x29: 0000000000000012
x30: ffffffffffffff82
x31: 0000000000000000
```

Figure 7.5: Comparison of sequential (left) and pipelined (right) clock cycles.

The code has 47 instructions. Note that both our processors take 1 extra clock cycle to end the execution (The Pipelined processor has 4 dummy instructions for correct operation). The code

causes 2 flushes and 4 stalls. This corresponds to 8 additional clock cycles. The pipelined processor also takes (number of stages - 1) = 4 clock cycles to produce the first output. Thus, the pipelined processor takes 12 additional clock cycles to run this program. However, it provides a significant advantage over the sequential design as the clock cycle time is nearly 5 times as fast (It would approach this for large instruction codes). We don't observe this advantage in verilog as there is no delay between operations in verilog!

8 Contributions and Acknowledgment

This project was a collaborative effort, with all three members actively involved in brainstorming ideas, implementing solutions, and debugging. While we've listed the components in the datapath based on primary contributions, every aspect was refined by collective discussion rather than individual effort.

The program counter (PC) and control unit were done by Siddarth. The instruction memory (file reading), register file, and immediate generation units were done by Varun. The ALU implementation was similar among the team members from the first assignment, with Varun's ALU module being instantiated in the processor. Samartha handled the ALU Control block and the Data Memory units. The final integration of the SEQ was done by both Siddarth and Varun. We of course, need to mention the AI usage in the project, which was done only for simple redundant work and automation (a Python script that encodes assembly instructions into hex code), so that we could save time in parts which are not relevant to the essence of the project. Test cases were brainstormed together, and we also tested some standard codes, like the sum of N numbers, linear search, Fibonacci sequence and vector addition, etc., as has been already discussed above.
For the pipeline, most of the datapath units have been directly taken from the sequential processor, with the additional pipeline registers done by Siddarth, the forwarding unit done by Varun, and the merging and integration of these units done by Samartha with help from the other team members. The hazard detection unit (which was a bonus), the load-store forwarding unit, `README.md` and this report were written by all three members.

However, in our opinion, pinpointing the work doesn't do justice to the work done by each member, and we would like to reiterate that the work was done collaboratively with equal contribution by the team members, as is supposed to be done in a group project, and every team member has complete knowledge of the entire codebase. It was not the coding which took the most effort, but the collective brainstorming and debugging.

We sincerely thank Prof. Deepak Gangadharan and the Teaching Assistants for their invaluable guidance. A special mention to our group TA, Aniruth Suresh, for his patience, clarity in resolving our doubts, and continuous support throughout the project.

References

[1] Sarah L Harris and David Money Harris, *Digital Design and Computer Architecture RISC-V Edition*, Morgan and Kaufmann, 2022.

[2] David A. Patterson and John L. Hennessy, *Computer Organization and Design The Hardware-Software Interface: RISC-V Edition*, Morgan and Kaufmann, 2018.

[3] The StackExchange community for solving doubts and explaining concepts beautifully. https://stackoverflow.com/a/69125543

[4] LupLab, RISC-V Instruction Decoder, https://luplab.gitlab.io/rvcodecjs