RISC-V RV32IM Processor Coursework

Personal Statement of Contributions

Darryl Boulton

Overview

ALU

Relevant Commits:

- ALU, ALU_decoder and

The ALU in this project implements all the logical and arithmetic instructions consistent with the RV32I instruction set architecture. However, I decided to include multiply instructions, implementing the RV32M instructions in essence making a RV32IM CPU.

To account for the extra eight instructions, the ALU_SRC signal from the control unit had to be extended to five bits:

alu_src (hex)	Operation	Description
0x00	ADD	Add (signed/unsigned addition)
0x01	SUB	Subtract (signed/unsigned subtraction)
0x02	SLL	Shift left logical
0x03	SLT	Set less than (signed)
0x04	SLTU	Set less than (unsigned)
0x05	XOR	Bitwise XOR
0x06	SRL	Shift right logical
0x07	SRA	Shift right arithmetic (signed)
0x08	OR	Bitwise OR
0x09	AND	Bitwise AND

0x0A	MUL	Multiply lower 32 bits (signed)
0x0B	MULH	Multiply upper 32 bits (signed × signed)
0x0C	MULHSU	Multiply upper 32 bits (signed × unsigned)
0x0D	MULHU	Multiply upper 32 bits (unsigned × unsigned)
0x0E	DIV	Division (signed)
0x0F	DIVU	Division (unsigned)
0x10	REM	Remainder (signed)
0x11	REMU	Remainder (unsigned)
Else (default)	NOP/0	Outputs zero

A case statement was used on alu_src to determine the correct operation.

ALU Decoder

Relevant Commits:

- ALU, ALU_decoder and
- Added muxes for forwarding and alu input selection. Made minor changes to CU, combining alu_op wires. Improved consistency of parameter names.

With new multiply instructions, this necessitates that the ALU Decoder is able to distinguish between these instructions and create the appropriate signal. This proved slightly more challenging as the RV32M instructions had the same func3 as those in RV32I.

Thus, the func7 signal from Lab 4 was extended to two bits:

```
  input   logic [1:0]             func7_5_0,
```

This signal includes the last bit, bit zero of func7 which uniquely differentiates RV32M instructions. This signal was used to determine a temp variable 'is_multiplication'.

```
 logic is_multiplication;
 assign is_multiplication = (func7_5_0 == 2'b01);
```

This signal was then used to differentiate between ADD, SUB and MUL instructions as seen in code below:

```
if (func7_5_0 == 2'b00) begin                          // ADD
```

```
    alu_src = 5'b00000;
end else if (func7_5_0 == 2'b1x) begin                // SUB
    alu_src = 5'b00001;
end else if (is_multiplication) begin                 // MUL
    alu_src = 5'b01010;
```

This same variable was used to differentiate MULH, MULHSU, MULHU and DIV from SLL, SLT, SLTU and XOR respectively, as well as REM and REMU from OR and AND. See alu_decoder.sv for full code.

To deal with immediate values, a separate case statement was required (since I-type instructions do not have a func7).

Pipelining Branch

Relevant Commits:

- Pipeline registers with no control added yet
- Combined everyting into a top level file. Sub-top files were made for sake of organisation.

To create a pipelined CPU, four new pipeline registers were created:

1. IF_ID: This separates the Fetch stage from the Decode stage. It is the smallest of the four registers as it does need to take any control signal inputs or previous stage outputs. This register can be stalled and flushed.
2. ID_EX: This separates the Decode stage from the Execute stage. This register can only be flushed.
3. EX_MEM: This separates the Execute stage from the Memory stage.
4. MEM_WB: This separates the Memory stage from the Write-Back stage.

Here is the top-level diagram illustrating how the registers and hazard unit are integrated along with the rest of the CPU:

In the top-level sheet, the Fetch, Decode and Execute stages were combined into their own individual files.

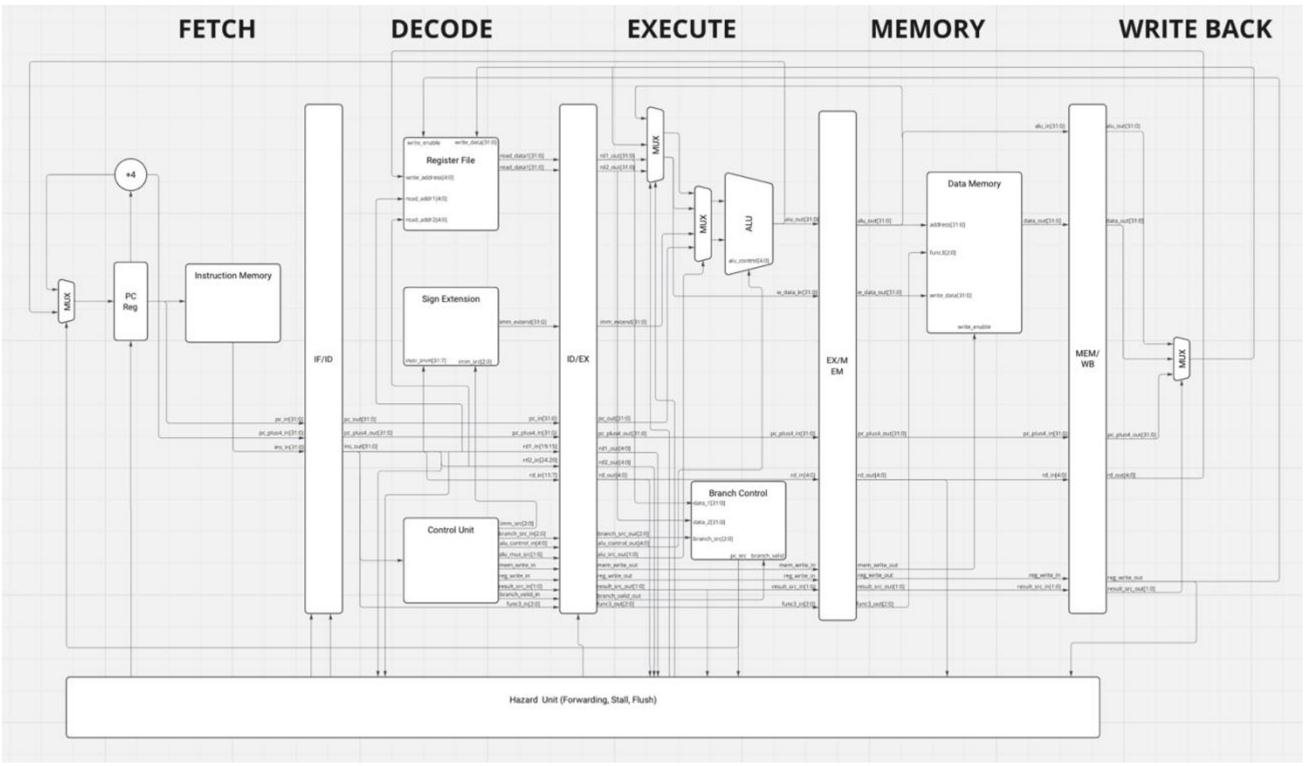

Figure 1: Top-level design

The next sections outline the implementation of the pipeline in more depth.

Hazard Unit

Relevant Commits:

- Modified pipeline registers to include control signals, and created harzard unit (forwarding only currently)
- Added stalling to pipeline for lw instructions
- Added corrected/up to date pipeline registers and hazard unit to pipelined branch

Software Interlocks - Forwarding

The hazard unit resolves software interlocks using forwarding. It takes as an input the registers used from the execute stage, the destination registers in the memory and write-back stages and the reg_write (write enable) bit from the memory stage. Two signals ‘forward_a’ and ‘forward_b’ are then generated according to this logic:

```
if (((rs1_e == rd_m) & reg_write_m) & (rs1_e != 0)) begin
    forward_a = 2'b10;
```

```
end else if (((rs1_e == rd_w) & reg_write_w) & (rs1_e !=
0)) begin
    forward_a = 2'b01;
end else begin
    forward_a = 2'b00;
end
```

The first and second conditions represent forwarding from memory access stage and write-back stage respectively. The last condition represents the default condition. This logic statement represents one of the inputs to the ALU. The other statement is identical bar 'rs2_e' and 'forward_b' being used instead.

The forward signals feed into two 4x1 multiplexers which determine the correct output signal. As a sidebar, the signals from these forwarding multiplexers are then fed into another set of ALU multiplexers (2x1). This is necessary to fully contextualise the ALU inputs and how they are generated. The signal 'alu_src' is generated by the control unit and is dependent on the instruction type.

The logic circuit of the multiplexers is shown in the diagram below:

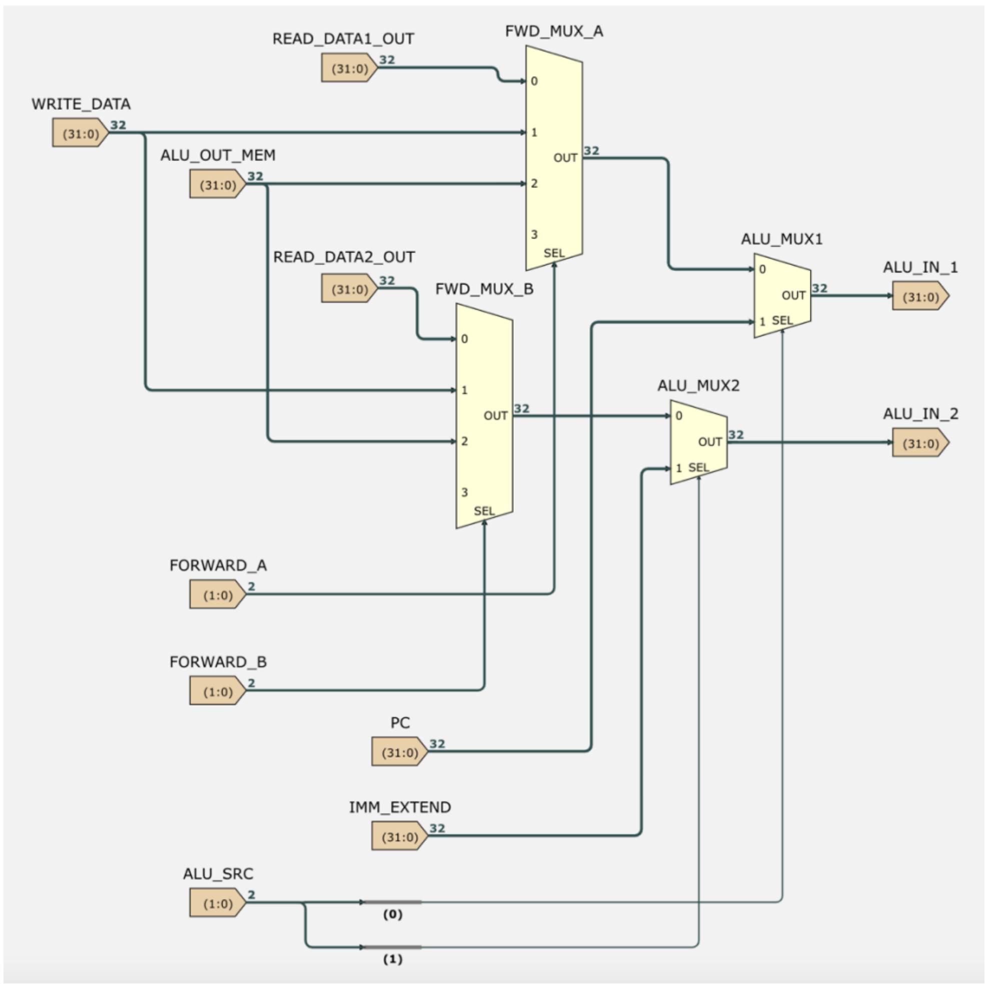

Figure 2: Forwarding and selection for ALU inputs``

Load Word Instructions & Branching – Stalling & Flushing

For load word instructions, the hazard unit generates three signals to stall and flush the correct registers:

1. stall_f: Stalls PC register. This signal is inverted then connected to the enable input on the register. When 'stall_f' is high, the register is no longer enabled, causing a stall.
2. stall_d: Stalls the IF_ID register. Input into register is also inverted.
3. flush_e: Flushes the ID_EX register by zeroing out all outbound control signals.

To implement this, a temporary variable was created 'lw_stall' which would be true in the event a lw_stall is meant to take place, see the following logic statement:

```
lw_stall = ((result_src_e == 2'b1x) && ((rs1_d == rd_e) || (rs2_d == rd_e)));
```

The input 'result_src_e' is generated by the control unit, with the most significant bit determining a 'lw' instruction.

All three signals are set to be equal to the value of 'lw_stall'.

In the event of a branch, a new signal 'flush_d' is required to flush the IF_ID register. 'flush_e' is still required making this a dual use signal.

Additional Comments

During the course of the project, it was decided that two pipelined-cpus would be made experimenting with different architectures. This particular implementation served as a basis for the full implementation with cache, and provided essential groundwork for its development.

While this implementation of the pipelined CPU demonstrates key concepts such as instruction fetch, decode, execute, memory access, and write-back stages and hazard mitigation, certain functional aspects remain incomplete or exhibit unexpected behaviour under specific conditions. These issues have been identified for future refinement.

Despite these issues, the project provides a solid foundation for further development and highlights the practical complexities of designing a fully functional pipelined CPU. The experience gained in debugging and testing has been invaluable in deepening my understanding of pipelining concepts, hazard mitigation strategies, and overall CPU design.