Processor Checkpoint for ECE 350

Yi Chen (yc311)

Note: This document can still change as I continue working on the processor. Therefore, the implementations described here are up to date as of 4/3/2021.

Currently only failing tests that involve exceptions because I have not implemented exceptions yet.

Completed Components

General Structure

- This is a 5 stage pipelined MIPS processor, with integer multiplication and division capabilities.
- The processor uses CLA adders and Booth's algorithm for fast arithmetic operations.
- The processor has hardware interlocking to prevent data hazards.
- Between each pipeline stage, there is a register that latches information from the previous stage. These latches are built as specialized registers. There is one extra latch for the `multDiv` module.
- The modules `stallControl` and `bypassControl` are for stalling and various types of bypasses.

Regfile

- My regfile is made slightly differently from a conventional register file. Each individual register are specialized registers with 2 output enable inputs and 2 data outputs.
- This allows each register to be connected to both outputs of the regfile, and moves the tri-state buffers to the inside of the individual registers rather in the regfile module. Personally, I think this simplifies the wiring.

Pipeline

- The processor has 5 pipeline stages and uses special registers for each of the stages.
- The each pipeline stage register has separate inputs and outputs depending on what data it holds. For example, the `XM` register has 32 bit inputs and outputs `O`, `B`, and `IR`, for the ALU output, data read from register B, and the instruction.
- There is a pipeline special register `PW` that is used by the `multDiv` module. This skips the M stage and directly connects to the `MW` stage.

Bypass

- The processor has a separate `bypassControl` module for the bypass logic for MX, WX, and WM bypassing.
- Internally, `bypassControl` determines which bypass is necessary, and outputs control signals for each of the multiplexers at the bypassed locations.

Stall

- The processor has a separate `stallControl` module for deciding whether or not to stall.

- The `stallControl` module checks for remaining data hazard cases that cannot be fixed by bypass, using logic given on the slides.
- The module also checks whether there is an active multdiv operation, and stalls for multdiv data hazard conditions (described below).
- Stalls will write `nop` to the `IR` input of `DX`, and will also disable the write enable for `PC` and `FD` latches.

Branch

- Since the processor continues to load instructions after encountering a branch instruction, it "assumes" branches are not taken.
- Branching conditions are detected in the X stage after the ALU `LT` and `NE` results are obtained.
- Branch uses a separate adder to calculated the offset PC, and controls multiplexers at the input of PC so that the new PC is immediately received. It also writes `nop` to all pipeline latches before it to flush already loaded instructions.

Jump

- Jump instructions are detected in the X stage so that instructions such as `jal` and `jr` can read and write to the correct registers.
- They work similar to branching instructions and adds an additional layer of ternary logic to regfile inputs and outputs.

Arithmic

- The processor uses a normal ALU for simple arithmetic, bypassing and stalling when necessary.
- The processor has a `signExtender` module to process signed immediates for I-type instructions.
- For multiplication and division operations, the processor uses the `multDiv` module. There is a special `PW` pipeline stage for the multdiv module that connects directly to MW. This latch is only write enabled when a multdiv instruction is in progress.
- The multdiv module uses the modified Booth algorithm for multiplication, meaning that it only needs 16 clock cycles to complete rather than 32. However, division still takes 32 cycles.
- While multDiv is in progress, operations that do not affect the `$rd` of the `mul` or `div` instruction are executed normally. If a instruction tries to read or write to that register, the processor will stall until the `multDiv` result is ready.
- Output of `multDiv` always bypasses the memory stage directly to the writeback stage.
- The processor will not stall during a multiplication or division unless an instruction attempts to read or write to the `$rd` of the multdiv operation before it is complete, or if it is followed by another multdiv operation.

Incomplete Components

Exception Related Features

- The exception related components are conceptually straightforward to implement, I was out of commission after a vaccine shot and was busy for the week after that, therefore I have yet to complete it.
- Determining exception codes:

    - Exceptions will have a separate module called `exceptionControl`. This module will be responsible for determining which exception occurred and outputting a correct exception code.
    - The module will take as input the ALUop at `DX.IR`.
    - The module will output a 32 bit number.
    - Internally, the module implements a logic circuit that correlates the ALUop to the output as follows: 00000 = 1, 00001 = 3, 00100 = 4, 00111 = 5, else = 2. The else case allows the `addi` case, which does not have an ALUop, to be accounted for.
- Writing exception to the correct register:
    - This involves two changes to the input to the `XM` latch: the data result from the ALU, and the `$rd`.
    - To implement this, I will add a ternary operator to the ALU output, so that if the ALU's `overflow` is flagged, the output from the ALU into `IR.O` gets replaced by the output from `exceptionControl`.
    - The second ternary operator is placed on the input to `XM.IR`. This will replace bits `[26:22]` in the instruction with `11110`, which corresponds to `$30`.
    - When the newly injected instruction and ALU result reaches the W stage, the exception will be written correctly to `$rstatus`.
- `setx`:
    - This instruction can be simplified by knowing that the upper `[31:27]` bits of `$rstatus` is never used.
    - Therefore, we place a new ternary operator on the input to `XM.O` so that if the opcode is `10101`, we replace `XM.O` with the entire `DX.IR`, which will make bits `[26:0]` of `$rstatus` contain the correct exception value given in the instruction.
    - We also add a condition to the ternary operator at `XM.IR` input so that it replaces bits `[26:22]` in the instruction with `11110`, which corresponds to `$30`, when the opcode is `10101`. (Since exception codes only go up to 5, it is fine to have bits `[26:22]` of the instruction written to `$rstatus` be changed)
- `bex`:
    - This instruction will work similar to `jr`.
    - At the D stage, a new layer of ternary operator is added to `ctrl_readRegA`, so that if the opcode at `FD.IR` is `10110`, `ctrl_readRegA` will be replaced by `11110`.
    - At the X stage, new ternary operator will be adde to assert `w_jump` if the opcode at `DX.IR` is `10110`.
    - The already existing mechanism for jump instructions in the X stage means that the assertion of `w_jump` will cause the `w_jumpAddress` to be set to bits `[26:0]` of `DX.IR`.
    - As a result, PC will be immediately changed to bits `[26:0]` of the `bex` instruction and the previous pipeline stages will be flushed.

Conclusion

I was able to complete most of the important aspects of the processor, and the remaining exception related instructions are not difficult to implement. I simply need to wait to implement these until I catch up with my work for my other classes, since I fell behind on my other classes this week from working on the processor.

The time I spent for the processor averges at approximately 5.4 hours per day between Mar 20 to Mar 29. There was one day where I did not work on the processor because I was out of commission from my COVID vaccine. I did not know that it was a legitimate reason to submit an incapacitation form. Most of my time was spent on debugging, because I initially used print statements to debug rather than GTKwave.

I had some difficulties implementing multDiv because my module does not behave exactly as expected: my `multDiv` module did not remember the two data operands passed to it, therefore I was unable to allow other instructions to run while multdiv is running. I made edits to my `multDiv` module to allow that. Another issue was that my `multdivReady` output asserted for longer than one clock cycle. I was unable to figure out how to make it only assert for one, but I added additional logic in my CPU to account for that.

Overall I believe that I have a solid understanding of the workings of a pipelined CPU, and I could definitely have made my coding style neater given more time. I could also have made my CPU code more readable by breaking more components into separate modules.