Processor Defense: Technical Report

Duke Community Standard

By submitting this LaTeX document, I affirm that

1. I understand that each `git` commit I create in the processor repository is a submission.
2. I affirm that each submission complies with the Duke Community Standard and the guidelines set forth for this assignment.
3. I further acknowledge that any content not included in this commit under the version control system cannot be considered as a part of my submission.
4. Finally, I understand that a submission is considered submitted when it has been received by the server.

Introduction

This technical report discusses the implementation details of a pipeline processor built in ECE 350. The processor uses multiple subcomponents created earlier in the semester and provides safe computations under data hazards and control hazards. All instructions in the ISA are supported excluding mult/div due to time constraints of the project.

Subcomponents

Regfile

The regfile was set to be negative-edge triggered to save one cycle for pipelining and data hazard protection. The regfile includes 32 32-bit registers.

The register file uses the 32-bit bus out of the 5-to-32 decoder and creates 32 AND gates with the ctrlWriteEnable bit. The output of each AND gate goes into the writeEnable input of each register. Thus, only one register at a time will have a its writeEnable pin high (since the decoder outputs a bus of 31 zeroes and 1 one).

The register file also creates 32 32-bit registers. The 0th register has its writeEnable input set to zero always, since we must not write to the 0th register. The data of each register is sorted into a 32x32 matrix. Then, 64 muxes are created (32 for readRegA, 32 for readRegB). Each ith mux takes in 32 bits, where each of the bits corresponds to the ith bit of a register. Thus, each mux picks one bit from the selected register. Then all of these bits are bundled into a bus to display as readRegA or readRegB.

With all of these modules, the register file can write to one register at a time (on positive clock edge), read from two registers at a time, cannot write to register0, enable/disable write enabling, and reset any register regardless of the clock's state.

ALU

The ALU, consisted of a 32-bit CLA, 32-bit SLL, 32-bit SRA, 32-bit AND module, and a 32-bit OR module. Tristate buffers were used to output the correct data using the ALU opcode.
My 32-bit CLA was implemented using four blocks of 8-bit CLAs. In essence, this is a second-level carry lookahead. This module computes the carry-out of each 8-bit CLA block C7, C15, C23, and C31. This is computed using AND/OR gates and propagate and generate functions. Four 8-bit CLA blocks are then created to calculate the sum using these carry-out bits.
My implementation of the 32-bit SLL as a barrel shifter involves a hierarchy of fixed logical left shifters. First, the 32-bit input along with the 4th bit (MSB) of the ctrlshiftamt is passed to a 16-bit fixed SLL as the enable bit. If the 4th bit is 1, then the input is logical left shifted by 16-bits.

Otherwise, the input remains the same. The output of the 16-bit fixed SLL is passed into the 8-bit fixed SLL along with the 3rd bit of the ctrlshiftamt. Similarly to the previous step, if the 3rd bit is 1, the input is shifted by 8 bits. Otherwise, it remains the same. This process continues for the 4, 2, and 1-bit fixed SLLs. Using each bit of the ctrlshiftamt, the input is logical left shifted the correct number of bits.

Mult/Div

Unfortunately, the mult/div subcomponents were not fully implemented into the processor due to time constraints of the project itself and complexity with respect to the pipelining of the processor. The Multiplier shift register holds the product bits and shifts two bits to the right every clock cycle. The extra bit stored the implied '0' booth LSB. The counter keeps track of the current clock cycle and was implemented with a 5-bit adder and a 5-bit register that increments by one each clock cycle. The 32-bit adder adds or subtracts the multiplicand from the upper 32 product bits (MSBs). The tristate buffers are used to input the correct bits into the upper 32 bits of the product register. More specifically, four 32-bit adders are used in parallel to output all possible adder output (add/subtract multiplicand or add/subtract shifted multiplicand). Only one of these results will enter the product register. Muxes were used to determine whether to send the multiplier bits or the current lower product bits into the lower bits of product register.
The shift register (RQB) was used to store the remainder of the dividend (upper 32 bits) and the quotient (lower 32 bits) and shifted by one bit each clock cycle. The counter was used to keep track of the current clock cycle (similar to the multiplier's counter). The adder was used to subtract the divisor from the remainder. Muxes were used to determine whether to send the dividend or the current lower 32 bits of the RQB into the lower 32 bits of the RQB. The two's complement module was used to negate the operands/quotient to support negative division.

Memory elements

D flip flops were used within registers. An SR latch was used mult/div to perform the respective operation based on the specified multiplier/divider control bits. The latch sets the output of the MultDiv to the output of either the Multiplier or Divider, depending on which control bit (ctrlMULT or ctrlDIV) was last switched on.

1 Full Processor

Design implementation details of each stage (F, D, X, M, W)

The implementation of the pipelined processor was based off lecture slides.

- The Fetch stage retrieves the current instruction and PC address bits from the PC latch and Instruction Memory module.
- The Decode stage stores the current instruction and PC address bits from the F latch. The Decode stage also controls the outputs of the read ports of the regfile.
- The Execute stage performs ALU operations and branches/jumps. The Execute stage receives current instruction and PC bits from the D latch. It also controls the input address to the PC.
- The Memory stage stores and loads bits to/from the Data Memory module.
- The Write stage controls the destination port of the regfile and the write enable.

Handling hazards

The processor can handle data hazards (read after writes) by using stalls and no ops. It can also handle control hazards (branches/jumps) by flushing the fetch and decode latches when a branch is taken or a jump is performed.

How fast the processor can be clocked

The processor can be clocked at roughly 50 MHz.

Bypassing logic

Due to time constraints and priority over correctness of computations, bypassing was not implemented. Bypassing was attempted near the end of the project timeline, but due to fear of producing incorrect data, bypassing was left out of the processor.

Efficiency details

The processor takes priority of correctness over efficiency, so the processor safely stalls instructions with no ops in order to avoid data hazards. Since no bypassing was implemented the processor's efficiency is at least much higher than a non-pipelined processor.

2 Instruction Implementation

R instructions were implemented using the ALU and register read ports and performing operations within the ALU.

I instructions were more complex. $addi$, bne, blt required modification of the ALU opcode since I instructions do not have an ALU opcode within the ISA. Because value comparisons are made within the ALU, multiplexers were used to set the ALU opcode.

JI instructions were computed in the execution stage, which controls the input to the PC latch. For jal, the ctrlwriteReg bit was set to $r31$ due to instruction requirements.

JII instructions were implemented very similarly to JI instructions. The only difference was that rd was the only register read from the regfile. The data from the register was multiplexed within the Execute stage in order to set the PC address equal to the data within this register, since Execute stage controls the input to the PC (i.e. it controls the current Instruction Memory address).

- add: This instruction was implemented by reading data from both of the two read ports of the regfile in the Decode stage and computing the addition within the ALU component in the Execute stage. The result was written to the specified destination register in the Write stage.
- addi: This instruction was implemented by reading data from only one of the read ports of the regfile. The immediate was extracted from the instruction bits and fed into the ALU using a multiplexer. The ALU opcode was set to 00000 (for add) for this instruction. The result was writted to the specified destination register.
- sub, and, or, sll, sra: These instructions were implemented exactly the same as addi. The ALU opcode was set automatically from the ISA.
- mul, div: These instructions were not able to be implemented in time for the deadline. However, with more time, the processor can be modified to produce a 32-cycle stall for multiplication or division.
- sw: This instruction was implemented by computing the data memory address in the ALU (rs + N). The write enable for data memory was set high. The data in rd was set as the input of the data memory.
- lw: This instruction was implemented by computing the data memory address in the ALU. Unlike sw, the write enable was set low. The output of the data memory was sent to the Write stage to write to the respective register.
- j: This instruction was implemented by extracting the target bits from instruction and setting the PC input equal to the target concatenated with the upper 5 bits of the PC. This

concatenation was done in the Execute stage since this stage controls the input address to the PC.

- bne: This instruction was implemented by reading from the two specified registers and feeding them into the ALU. The ALU opcode was set to be subtraction in order to compute the isNotEqual bit. If the register were not equal, the Execute stage simply sets the PC = PC + 1.
- jal: This instruction was implemented setting the PC input equal to the target (similar to j). The only difference is that $r31$ is specified as the ctrlWriteReg and the PC + 1 bits are sent through the Memory stage to the Write stage.
- jr: Similar to 'j', this instruction sets the PC input equal to a value. The value for jr is simply the data read from a register. Thus, the data is sent from Decode to Execute to PC.
- blt: Similar to 'bne', this instruction reads from the two specified registers and feeds their data into the ALU. The ALU opcode was also set to subtraction in order to compute the isLessThan bit. If this bit were high, then adder result, PC + 1 + N, would be sent to the PC. Otherwise, PC + 1 is sent.
- bex: This instruction reads from a single register and sends its data to the ALU. The other operand of the ALU is a bus of 32 bits. The ALU opcode is set to subtraction to compute the isNotEqual bit. If this bit is high, then the target from the instruction gets sent to the PC input.
- setx: This instruction simply writes data to a specific register, $r30$. The target was sent from Execute through Memory to Write. The ctrlWriteReg control signal was set to 30 in order to write to rstatus (a.k.a. $r30$).

3 Process

Why did you choose your implementation?

This implementation was chosen due to simplicity of design and thoroughness of details given in lecture slides. Given the assurance that the designs discussed in class were viable for this project, a vast majority of the process implementation was, thus, inspired from lecture.

How did you test your implementation?

A Python script was used to generate test benches and convert MIPS to machine code. These test benches compare any register values after all instructions have been completed. Internal wires were also monitored for further debugging.

What errors exist in your processor?

The processor doesn't support mult/div due to time constraints and extreme clock cycle management. This error can be fixed given more time to implement the component.
Another error is detecting data hazards for register $rstatus$. Data hazards are checked for instructions bex and $setx$ but not for others due to time constraints. However, data hazards are checked for register $r31$. This error can be easily fixed given more time to complete the project. Because the data hazard involving $r30$ can arise in the majority of the instructions, more time is needed to program the logic in verilog.

Challenges - what were the main challenges you faced and how did you overcome them?

The main challenge faced was debugging the pipelined processor. Much time was devoted to building python script that could create testbenches by converting MIPS to machine code.
Extreme time management was needed due to the very short timeline of the project. Prioritizing tasks helped with this so that the processor is built systematically and efficiently.

Main learning points - what did you learn by making this processor?

I learned about the several possible data hazards that can occur in a pipelined processor. I also learned briefly why negative edge triggering a regfile can help save one cycle per instruction and for data hazards (inserting no ops).

Conclusion

Although not very much time was given to complete the project, the implementation of the processor is satisfactory in the sense that most instructions are computed correctly. Correctness is the top priority, so protecting from data hazards and control hazards was key to computing correct results. Efficiency was also in mind when building the processor in the sense that stalls were used only when needed. Bypassing would the next goal of the project if the project deadline were to be extended. Overall, many lessons were learned regarding technical knowledge of processor, time management, and importance of debugging efficiently by developing tests.