CPU Analysis

Introduction

The CPU created for this assignment, is a 5-stage pipelined processor, that supports arithmetic, logical, memory and branching operations. This report will focus on several key subcomponents of the processor, discussing the motivations behind their implementations and the decisions taken during the development process. The report will also focus on tradeoffs of this implementation, compared with other possible alternatives.

ALU

The ALU used in this processor is the same that was used for the 1st checkpoint. The ALU is able to perform addition, subtraction, logical and, logical or, arithmetic right shift and logical left shift. The ALU is also able to compare the two inputs and determine whether they are not equal to each other or if one is less than the other. Unlike the standard design specifications, this ALU features two separate adder modules, one for addition and one for subtraction. This allows the isLessThan signal to return accurate result without the need for a subtraction operation to occur. The notEqual signal, does not rely on the subtractor module, instead using 32 2-input xnor gates, and using a AND gate to combine their inputs to check for equality. This allows for faster computation of inequality, because it only takes 2 gate delays to compare the two inputs. If, however a comparator was built to use the result of subtraction and compare it to 0, then each of the 32 bit outputs of the output of the subtractor, along with overflow bit, would need to be channeled into an OR gate, adding an additional gate delay on top of the gate delay of the 2 level carry lookahead adder used for subtraction.

Register File

The register file used in the processor is the same as the one used for the 2nd checkpoint. The register file contains 32 32-bit registers, 2 read ports and 1 write port, each with its own decoder and . Each register is made up of 32 D flip-flops. The the output of each register is connected to two tri-state buffer, one for read_registerA, one for read_registerB. The read decoder outputs are passed to the tri state buffers to select the appropriate output registers. This approach is much faster than the 32 to 1 MUX, which proved to be too slow for the 50 Mhz clock rate due to

consecutive gate delays. The write decoder outputs are used to set write enabled signals on registers.

MultDiv module

The Multiplier used in the CPU was taken from the provided behavioral components. The original multdiv built for the checkpoint 3 featured a modified booth multiplier, which was able to complete the computation in 16 clock cycles, and a naïve divider able to complete division in 32 clock cycles. Because of a bug in multdiv module, and the lack of time to complete the CPU an behavioral component provided by the instructors was chosen instead. When multdiv finishes the operation or an error occurs an entire pipeline is stopped (write enable is set to false for PC and all latches), and the output of the multiplier is written to the register file. The Stalling, Pipeline and Bypassing Sections contain more details about how the read after write and write after write hazards are managed for the multiplier.

Pipeline Latches

The pipeline features 5 latches. A F/D latch stores the instruction retrieved from imem and the program counter value. A D/X latch stores the output of the two read registers, as well as the program counter and instruction values outputted by the F/D latch. The M/X latch stores the output of the ALU, the value of registerA coming from the D/X latch (this value is written to dmem in SW), as well as the value of the instruction coming from the D/X latch. The M/W latch stores the output of dmem as well as the output of the ALU coming from D/X latch, and the value of the instruction that also comes from the D/X Latch. Each of the 4 latches described above is followed by its own decoder module that parses out the control signals from the intruction. The 5^{th} latch that is not part of the traditional 5-stage pipeline is a special MULTDIV latch, that stores the instruction that caused multiplication or division to take place, as well as the status of whether the multiplication is currently ongoing. One tradeoff that had to be made when constructing the latches is whether or not to pass the entire instruction to each latch, and have appropriate control signal parsed out at each stage of the pipeline, or to parse out all the control signals needed for the 5 stages at the very beginning and pass those into the latches instead. While it is true that the chosen approach causes some redundancy as the same control signals are often parsed out at multiple stages of the pipeline (for instance the values of rd, rs, rt and the opcode of the instruction), this approach also has its benefits. The benefits of this approach is

that it allows for faster development and easier debugging as each stage in the pipeline becomes an entirely independent entity that can be developed, tested and improved on separately, without the risk of causing an error at a different stage of the pipeline. Additionally, storing every possible control signal needed would increase the hardware cost as more flipflops would be needed, so while that approach would lower the hardware cost of the control logic, it would increase hardware cost of storage. If time is not the factor in the development process, the two approaches are quite comparable, and the most efficient approach is probably some combination of the two, however as time is a limited resource in this project, and storing instructions in latches leads to faster development, that approach seems better suited for the situation.

Bypassing Logic

The bypassing logic of the cpu is largely contained in the rwHazardController module. The logic was implemented iteratively, first to handle just ALU instruction, then LW and SW, then multiplication, and finally jumps and branches. The rwHazardController module would take instructions in DX, XM and MW stages of the pipeline as inputs. By looking at the opcodes it would figure out whether XM or MW instructions were writing to a register. It would use the opcode of DX instruction to figure out which registers it reads from. It would also compare the 5-bit register fields within the DX instruction to the destinations of XM and MW. Finallly if there was a match between DX and ether XM or MW and that command wrote to rd, on overwrite output signal would be set to true. The module also handled one case in which MW value would need to be bypassed into XM, that being if XM is a store word instruction, and MW writing to the rd of the XM. In that case a similar comparison was performed. The reason for multiplication output not being bypassed is that any read after write of write after write conflicts with multiplication were handled in stall logic.

Stalling

The stall logic was handed by the stallController module, which took instructions in the FD,DX and Multiplicaiton latches as its inputs. If the DX instruction was a load word instruction, and its destination was read by FD instruction then a 1 cycle stall would be necessary, as it takes 1 clock cycle to bring the value out of memory. If there was a match between the target of the multiplication and the source or destination of the FD command, and the multiplication was ongoing, a stall until the completion of the multiplication or division operation was necessary.

There was no need to check the state of the DX register, because the clock of the multdiv register is inverted, so when dx decoder identifies multiplication, it writes it to multdiv latch, and multdiv module, and that leaves another ½ of a clock cycle to stall the instruction in fd latch if necessary.

Branch Prediction

The CPU features a simple branch predictor. Because branches are used by programmers to write loops, when a branch instruction comes in, the probability of it being taken after its resolved is higher than the probability of it not being taken. With this fact in mind a simple branch predictor that always assumes the branch needs to be taken can be implemented. Such predictor also allows for all jump instructions except for jump register to be resolved in decode stage. Following the F/D latch is a F/D Decoder module that checks if the instruction is a branch or a jump (except for JR), and if so takes it, overwriting the program counter input. When the instruction gets to the DX stage, and into the D/X decoder, it identifies if the instruction is a branch and if so feeds the appropriate inputs to the ALU. The isNotEqualTo and isLessThan output of the ALU are then used to determine whether the branch needed to be taken, and if not, the PC value is overwritten back to the PC value in the DX latch, and the FD instruction is flushed. This results in only 1 instruction needing to be flushed, which 1less than the approach talked about in class. If time permitted, an additional modification would have been done to the branch predictor, where a separate latch would have been added to store the value of the instruction and PC coming from imem after FD, and then that value would have been plugged back into F/D instead of a noop, making the predictor even more efficient.

Known Issues

Currently there are no known issues with the processor. It is possible that the bypassing and stalling logic still misses a few edge cases especially around multiplication, and $30 and $31 registers, however the tests conducted during development did not find any errors. The tests folder in the git repository contains the majority of the test suite used.