Project Extra Credit Report - Pipeline

Hieu Nguyen

Spring 2024

1 Loading ROM(s) with the Microcode

In my setup, I've organized different ROMs into separate tables, each labeled for clarity. These tables use bits to represent instructions, and I convert these bits into HEX code and transfer this code into the ROMs to load them up.

The EX ROM sits in one part of the circuit, while the MEM ROM is spread out over MEM, WB, and Forwarder subcircuits. Similarly, the WrREG ROM is implemented in the WB subcircuit, and the HALT ROM within IF. The two DF ROMs are located within the Decider component of the Forwarder subcircuit. Opcode inputs are used to specify the relevant instructions for each ROM, with associated labels facilitating the handling of output values for each instruction within its respective subcircuit.

The HALT ROM enables program halting if the instruction is a halt command

The EX ROM manages instruction execution and variation handling.

The MEM ROM helps with the memory management by distinguishing between LW, SW, or no memory operations.

The WrREG ROM signals whether an instruction writes to a register.

The DF ROM facilitates data forwarding operations.

2 Pipeline Implementation

2.1 Design of Each Stage

1. **IF Stage**:

   In my IF stage, I use the three main components: the PC register, the IMEM RAM, and the HALT ROM. The process involves outputting PC + 1 under normal conditions, provided there are no branches, stalls, or halts. However, if a branch occurs, the output switches to the branched PC. In the case of a stall or halt, the output remains solely the PC value. Additionally, the IF stage outputs the instruction associated with the current PC, referred to as IR. It also takes inputs such as BranchPC, Branch, and LwStall into consideration.

   After the data is processed, it proceeds to the FBUF, which includes registers for PC and IR. Under normal circumstances without stalls, the PC and IR data are written into their corresponding registers. However, if a stall occurs, the writing process is halted. Similarly, if a branch is detected, both registers receive a value of 0, indicating a NOOP instruction. This buffer ensures that the PC and IR data are correctly stored in their designated registers.

2. **ID/RR Stage**:

   Moving on to the ID/RR stage, it involves retrieving the PC and IR from the FBUF. Within this stage, I use the DPRF that is in this stage. Additionally, I have an Instruction-Splitter that splits the IR into opcode, Rx, Ry, OFF20, and Rz components. Based on the opcode, a MUX determines whether each instruction requires Rx, Ry, Rz, or none as either source 1 (scr1) or source 2 (scr2). These sources are then fed into the DPRF, producing RdData1 and RdData2 as outputs. This stage outputs PC, IR, RdData1, and RdData2, while also receiving WrDst, WrData, and WrEn signals from the WB stage for potential register writes, which will be addressed later.

   Then, the data moves to the DBUF, where it incorporates PC and IR from the FBUF, along with RdData1 and RdData2 from the data forward mux, and stores them into their respective registers. Again, if a Branch or LwStall condition arises, the registers are filled with zero or NOOP values instead. Finally, the values within these registers are outputted for further processing.

3. **EX Stage**:

   In the EX stage, data received from the DBUF is processed. Using an Instruction-Splitter, I extract the opcode and Offset20 from the IR, with Offset20 being directed into the B MUX. Through my A and B MUX, there are decisions made regarding which values should be designated as A and B for arithmetic operations within the ALU. The EX ROM signals the selection process by controlling the specific data and operations each instruction should choose for ALU execution.

   Moreover, the BranchPC is computed by adding the PC and Offset20, or RdData2 in the case of the JALR instruction. The Branch signal is determined based on whether the output from the ALU is less than 0 (note: despite appearing as greater in my circuitsim, adjustments were made for consistency), and the ChkCmp flag is enabled, or if the instruction is either Br or JALR. Subsequently, the output of the EX stage includes IR, SwData (representing RdData2), the ALU's output (referred to as OUT), BranchPC, and Branch. BranchPC and Branch are directed back to the IF stage, while the remaining outputs are channeled into the EBUF.

   Within the EBUF, IR, SwData, and Out are respectively stored in their designated registers, with the output reflecting the contents of these registers.

4. **MEM Stage**:

   The MEM stage actively processes each of the incoming outputs, simultaneously incorporating the MEM ROM. By extracting the opcode from the instructions, we can decide whether the instruction pertains to a Load (Lw) or Store (Sw) operation using the ROM. In the case of an Lw instruction, memory is actively accessed from the location specified by the Out value, resulting in the retrieval of (Data out) Dout. Conversely, for an Sw instruction, SwData is actively stored at the memory address indicated by Out. Consequently, the outputs of this stage encompass IR, Dout, and Out.

   Then, these processed outputs are channeled into the MBUF and stored within their respective registers, with the values within these registers constituting the final output.

5. **WB Stage**:

   Moving on to the WB stage, we will retrieve the IR, DataOut, and Out for processing. Revisiting the MEM

ROM once more, we actively ascertain whether the instruction corresponds to a Lw. If affirmative, we actively designate Dout as our Out value. Otherwise, Out retains its original value. Additionally, we use a WrREG ROM to actively determine whether the instruction involves writing into a register, and this information is actively outputted.

Furthermore, we actively dissect the instruction to obtain its Rx, which is then actively set as dstReg and outputted. Subsequently, dstReg, Out, and WrReg are all actively directed into the ID/RR stage for the purpose of writing into the registers housed within the DPRF.

2.2 Instruction and Data Memory

Instructions in IF usually do not write memory to themselves, so it is actually optimal to implement a separate memory to write instruction memory to. That way, we will have instruction memory will be wired to IF and data memory will be wired to the MEM, allowing MEM reading or writing data memory with the correct data from WB that is passed from Data MEM instead of stalling IF.

The separation of instruction memory and data memory allows for concurrent access to these two types of memory, which is particularly advantageous in a pipelined architecture. While the IF stage reads instructions from the instruction memory, the MEM stage can simultaneously access the data memory for read or write operations. This can maximize the use of the memory bandwidth and reduces wait times that would otherwise occur if both types of accesses were competing for the same memory resource.

For IF and Data Hazard, we can also make sure that data operations don't interfere with the fetching of instructions. The implementation helps keep the pipeline running smoothly without unnecessary pauses. Moreover, the separation helps in reducing data hazards in which EX stage could be affected by the unavailability of the needed data. Since the IF process is isolated from data reads and writes, the risk of pipeline stalls caused by dependencies between successive instructions is reduced.

2.3 Data Forwarding Mechanism

Here's my approach to managing data forwarding: I gather instructions from ID/RR, EX, MEM, and WB and feed them into my Data Forwarder. Then, I compare the instruction in ID/RR with those in EX, MEM, and WB using three data deciders to determine if there should be a data forward from that instruction. Additionally, I check for a possible LwStall for the instructions in ID/RR and EX.

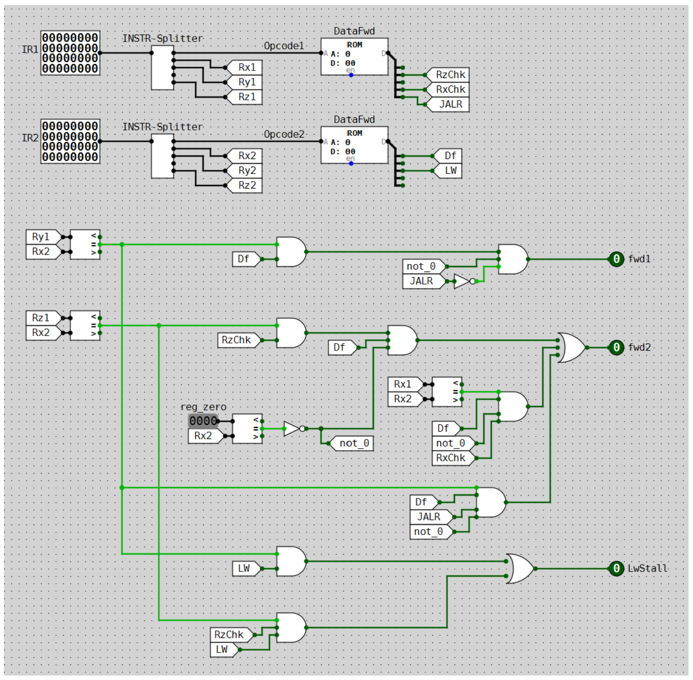

Figure 1: Data Decider

In the Decider, I've implemented 2 DF ROMs to examine important values for each instruction. For the ID/RR instruction or IR1, I input its opcode and check if there's a read for Rz or Rx, and whether it's a JALR instruction, as that requires special handling. For the second IR, or IR2, I check if there's a potential data forwarding (df) hazard and whether it's a Lw instruction. The Decider outputs three values: DF1, DF2, and LwStall, particularly if IR2 is from EX. DF1 is triggered if Ry of IR1 equals Rx of IR2 and there's a potential hazard, while DF2 is activated under various conditions including Rz1 equals Rx2 and other hazard potentials. Additionally, I check for LwStall, triggered if Ry1 equals Rx2 and IR2 is either Lw or if Rz1 equals Rx2 with associated conditions.

In the Forwarder, I've set up a Priority Encoder where EX-Datafwd takes precedence, followed by MEM and then WB. These encoded outputs determine DF1 and DF2, alongside the LwStall output from the first Decider. I also verify if IR3 is a Lw, and if so, it's outputted.

Next, I use a data forward MUX that feeds into DBUF. RdData1 and RdData2 from ID/RR occupy the first two

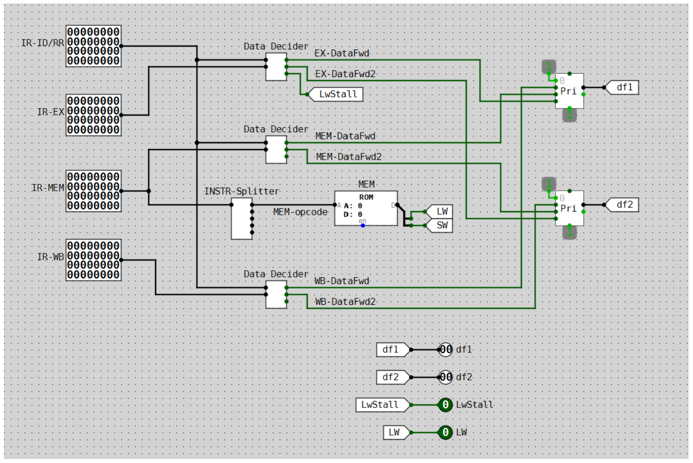

Figure 2: Data Forwarder

bits, followed by WB-out if DF is 1, MEM-out if it's 2, or EX-out if it's 3. If IR3 is a Load, MEM-out will reflect Dout from MEM; otherwise, it will be MEM-out from MEM.

2.4 Stall on Branches

In our pipelined processor, handling branch instructions is tricky because we don't immediately know the outcome of a branch.

To handle this, when a branch instruction comes up in the ID/RR stage, I check if it might change the usual sequence of executing instructions. But I can't be sure whether we'll actually need to jump to a new part of the program or just carry on as usual until we reach the end of the EX stage. That's when I can really see what should happen based on the branch conditions.

If it looks like we might take the branch, I have to stop any instructions that came after the branch from being executed, just in case they're not supposed to run. To manage this, I put in a stall by inserting a bubble (NOOP) right after the branch instruction. This bubble stays there until we're certain about whether we're taking the branch or not. This way, I can ensure that we don't run any instructions that shouldn't run, maintaining both the accuracy and efficiency of the processor.

3 Challenges during development

During the development, I found it difficult to implement the microcode for EX ROM and DF ROM control signals, especially with the control signals for SHIFT (SLL, SRL, ROL, and ROR). Additionally, there are some data forwarding logic that I struggled to understand if I only read from the project description, and I had to do further researches on external resources such as powerpoint files and even online videos explaining the data forwarding logic for hardware implementation. The pipeline wiring was also a challenge itself as there are simply so many wires needed for data forwarding.

4 Cycle Count and Pipelining Metrics

For cycle counting, I implemented a cycle counter that measures cycle by setting a register to count until the program halted. When halt instruction is executed, the HALT opcode will be fed into HALT ROM and stop the count by the time the program halted.

Due to some potential inaccuracy in implementing microcode into ROMs, the pipeline often ended up never branch from EX and hence did not succeed in displaying the mechanism of data forwarding. In other instances, the pipeline would continuously reset PC value and branch after hitting a certain cycle limit, which I assume to be caused by data fowarding or microcode.

5 Potential Areas of Improvement/ Further Optimization.

There might be some better implementions for the pipeline out there that I could take advantages of. There were two office hour sessions for this extra credit project that may have assisted me further in completing this project better, but I was not able to attend unfortunately. The microcode is shown to be faulty, even though there were multiple attempts in incorporating ALU's new instructions control signals into it. However, overall, I believe that the structure and reasoning of my pipeline should be sufficient enough for properly executing programs faster and more efficient than the first project's datapath, given the optimized microcode.