***The Pipeline Explanation***

1. *Introduction*

Overall, the implementation of the pipelined processor is fairly uniform—the first stage performs fetches, the next performs decoding and register reading, the one after that performs arithmetic, then the next to last manipulates memory, and the final writes back to registers. While simple in principle, the actual implementation leaves a multitude of unanswered questions that stir up a great multitude of responses. For my pipelined processor implementation, the most notable high-level design choice that I made was the use of a ROM for each stage. While in practice, ROMs tend to be slow to read and would be somewhat of a hindrance in terms of performance, for the purposes of creating a toy simulation, a ROM provides an ideal playground as it allows for flexible changes to microcode as while one can change a ROM, it is much more difficult to understand a group of combinational logic to determine control signals, let alone figure out how to change it to obtain a specific desired outcome. In addition to this, the choice to use multiple ROMs means that one does not need to think about a “next stage”—if one were to use a single ROM to output control signals, one would need to consider that each instruction takes five cycles, so one would need to consider the ways that these five different stages can interact, while when using multiple ROMs to control each stage, one only needs to think about what must be accomplished in each stage for a given instruction. In addition, by using many ROMs, indexing into the ROM becomes trivial—one must only use the opcode of an instruction instead of thinking about what comes “after” some stage—by using multiple ROMs, this is handled completely by the pipelined nature of the processor—it makes for an intuitive division of labor, or a well-tuned assembly line in a factory. One minor but important deviation from what might be considered a “standard” pipeline implementation in the CS2200 class may be my choice to put the logic for selecting between a source register’s output or an immediate value into the ALU in the EX stage instead of the Decode/RR stage. While I will elaborate on this later in the challenges I faced, I found it more convenient to make this choice in EX since it turned out that because of previous choices I’d made, there were scenarios in which I needed both the immediate value and a source register and not one or the other, so it became necessary to pass this information through the DBUF. With these changes that I’ve chosen to make in my pipeline implementation, I shall now explain the choices I made for each stage in a finer grain.

1. *Fetch*

The general structure of a pipelined processor detailed in the provided pdf given to students left many implementation details of the Fetch stage open to a variety of solutions—in my case, I needed to find a way to stop PC from counting when there was a stall in the pipeline. To resolve this problem, I inserted a mux that will choose between either the incremented PC or PC depending on whether or not there is a stall, and then this will feed into the mux that chooses between a branch address or PC+1—this way, branches can still flush out stalled instructions. In addition to this, I also added some additional functionality to my FBUF in that it will not latch in a new value when there is a stall—this would be rather catastrophic for the pipeline as latching a new value in during a stall would result in the older instruction being deleted and the younger instruction being duplicated—note that we are talking about the instructions being processed in IF and Decode/RR. Aside from these two choices, little deviated from “standard” implementation in the Fetch stage of the pipeline.

1. *Decode/RR*

In the Decode/RR stage, the problem of register forwarding was easily the most challenging feature to implement in the entire project. While this idea was discussed in class at some level, implementation details were relatively scant. After much thought regarding this challenge, I borrowed the idea from the TAs of using priority encoders to “choose” a stage to forward from, and to verify that a stage is eligible to forward by using subcircuits that would compare what destination register is being written to in some stage against the registers that an instruction in the Decode/RR stage. While this is useful, it is not enough to consider what some instruction’s destination register is—it is imperative that one also considers the operation being performed, as this will provide crucial information in determining whether some destination register number is real—the instruction will actually write to a register—or if it is some garbage data that happens to coincide with a register number. With these in mind, one must also consider the possibility of data hazards—most notably the RAW hazard. While a proper implementation of data forwarding will ensure that most RAWs are never a problem in that they neither threaten the correctness of an execution nor produce bubbles, the scenario in which a register reads directly after a LW instruction will, using the pipeline implementation taught in class, inevitably produce a bubble since there is one stage of separation between the Decode/RR stage and the MEM stage. To check that there will be no such hazard, the circuits that choose forwarded values based on priority assigned by priority encoders will also check that if the forwarded value originated from the EX stage, that its opcode is not LW. If it is, then a stall will be caused. In addition to this most major consideration was also the problem of implementing halting. This was ultimately implemented using a combination of microcode signaling HALT and hardware that will in effect continue to duplicate the HALT instruction and not let any other instructions pass into the DBUF. Furthermore, some muxes were put into place that were controlled by microcode that determined which registers in an instruction would be used as a source and which would be used as a destination—considering how intuitive this approach is though, this is likely a “standard” approach to determining sources and destinations.

1. *EX*

The EX stage turned out to be perhaps the messiest of the stages implemented due to a number of questionable decisions made to support a handful of edge cases. A quick scan of the EX stage will allow one to notice that there were signals made specifically for the LEA instruction and the JALR instruction, and some further inspection will reveal hardware solutions to determine if branching should occur for a BGT or BEQ instruction. While one might find these solutions questionable and a jarring break from the microcode being used to control signals, first, the Decode/RR and Fetch utilize some signals that are generated by hardware. Secondly, to hint at some possible optimizations one could make, hardware runs significantly faster than the long and tedious process of reading control ROMs. Aside from these pieces of hardware made to handle branching and the like, the implementation of the EX stage adheres closely to the provided example image in the pdf aside from the aforementioned movement of a particular mux. The reason for its movement is that in cases of BGT and BEQ, one will need both a Source Register 1, a Source Register 2, and an offset. Interpreting the image in the pdf provided with the project would lead one to believe that one would require logic to choose one or the other while in the Decode/RR stage, or to somehow give both to the DBUF for use in these instructions, however, one elegant solution is to simply defer this decision to the EX stage and to provide all three values to the DBUF rather than choosing only two—this considerably simplifies the logic required for BGT and BEQ. We also see the first appearance of the hardware discussed earlier which will output a 1 or a 0 for SR1 and SR2 for the instruction Decode/RR depending on whether or not the destination register of this instruction matches SR1 or SR2—this proves to be crucial for successfully implementing forwarding, an essential feature for an efficient pipeline. One may also notice that these values have tri-state buffers that are controlled by some sort of signal—these signals will only allow for values to be outputted if the instruction in question is capable of writing—this solves the problem of desired source register values coinciding with a number of bits in an irrelevant piece of an instruction that could be interpreted as a destination register in some cases—this logic ensures that destination registers are only interpreted as such and that bits that are not destination register bits will not be misinterpreted this way.

1. *MEM and WB*

Considering how straightforward MEM and WB are, I thought that their implementations could be aptly combined into a single section explaining at a high level how each work. The MEM stage has microcode that controls which instructions can write into data memory and which ones can read into data memory. It also has some bits that will determine whether a value read from memory should be passed on to the MBUF or if it should be the output of the ALU or if it should be the incremented PC. This allows for considerable logic simplification in the WB stage. As for the WB stage, there is only a signal to control which instructions may write to the register file, which is of course controlled by microcode.

1. *Potential for Optimization*

With these implementation details in mind, I must first confess that there is much room for optimizations and for making the pipeline in general run closer to a “perfect” pipeline’s performance without stalls. One such option to accomplish such would be to implement hardware such that it is possible to perform memory reads and execution simultaneously. In practice, this may be impractical without caching due to the cost associated with memory accesses, but in a toy simulation, this could result in a reduction in the number of clock cycles needed to execute some program. In addition to this idea, smarter branch prediction would be helpful in reducing the number of flushes and therefore the number of wasted clock cycles. With these two optimizations in mind, one could hope to build a considerably faster pipeline.

1. *Instructions for Your Convenience*

To load the microcode into the appropriate ROMs, load the microcode for *X* stage into its corresponding ROM. The microcode will say which stage it corresponds to, and the ROMs can all be found either at the bottom of the circuit or to the right of the pipeline in the case of WB. The values that one wants to copy are the hex values in the yellow column to the right. Note that you may have to scroll down a couple dozen rows before finding the microcode—I apologize for my lack of skill in Microsoft Excel. As for loading instructions and data, simply copy the appropriate *.hex* data into both the instruction and data RAMs—labeled IMEM and DMEM respectively. You may expect to find these in the Fetch and MEM stages of the pipeline respectively.

A green background with black dots

Description automatically generated

*A snippet of the microcode for the Decode/RR microcode in the excel spreadsheet*

1. *Stats*

Of course, the best way to back up the performance of my pipelined processor is to provide statistics of its performance versus the performance of the vanilla LC2200 I implemented in project 1. Running the same code, I required 7,693 cycles, while with my pipeline implementation, I only needed 1,737 cycles, meaning a 442.89% speedup over the vanilla LC2200 using my pipelined implementation, demonstrating the utility of pipelined architectures.