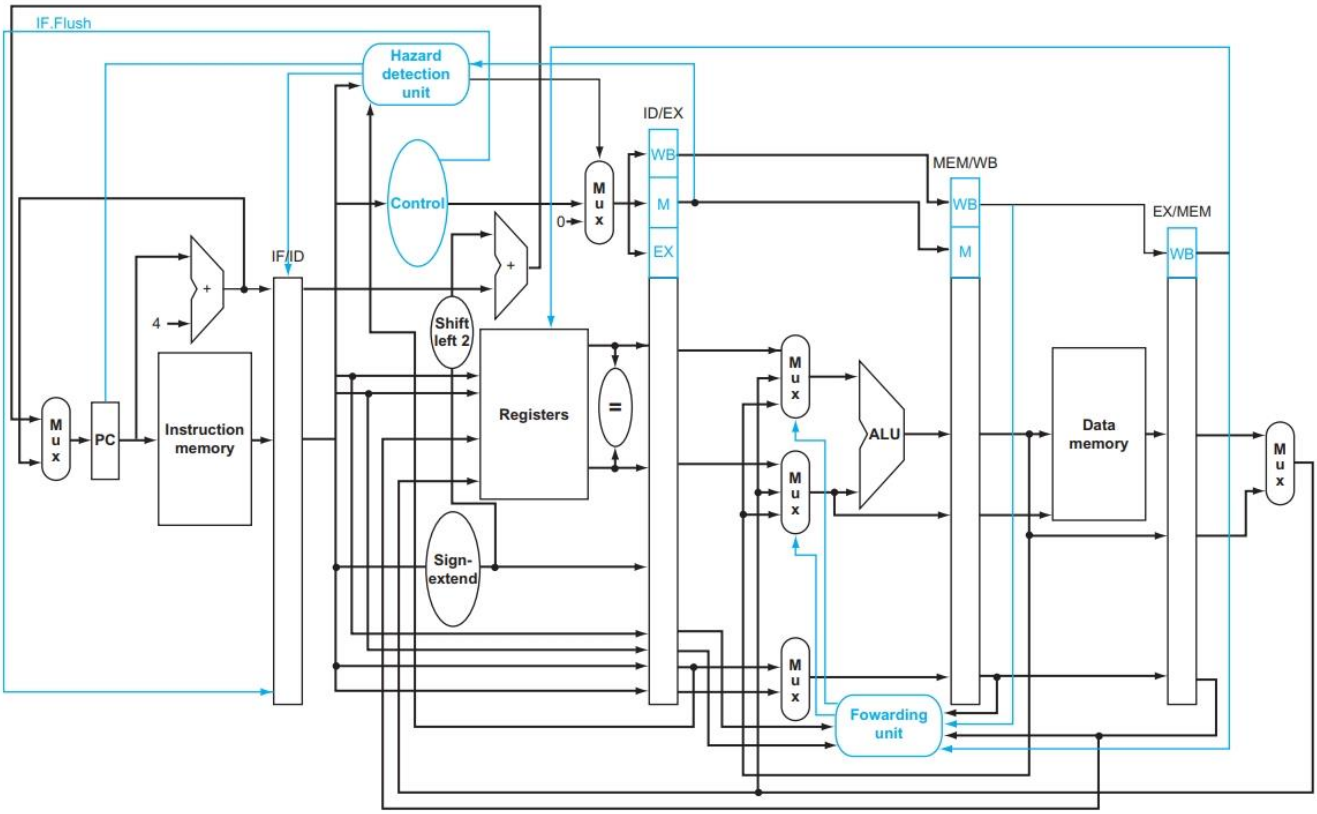
CprE 381 – Computer Organization and Assembly Level Programming

Project Part C

[Note from Joe: this pipelined processor is the final lab of the semester, and similar to the single-cycle processor, will involve substantial design, implementation, integration, and test tasks. You again have four weeks to complete the assignment and I have broken the project into two phases of roughly equal length. In ***Phase I****, the goal is to generate a pipeline that will require software to carefully schedule instructions to avoid data and control hazards.* ***Phase II*** *requires you to design hazard detection, stalling, and forwarding logic. As you may have seen in completing Project Part B, it is very common for an integrated design to exhibit several problems that were not visible during the individual component testing. Although some aspects can be worked on concurrently, for that reason, it is still recommended that you finish Phase I as early as possible, so that the final 2+ weeks can be spent on whole-CPU testing for Phase II. As we've reached the end of the semester, I cannot provide any extension for this lab, and consequently you will be graded based on what you can demo and submit by the due date.]*



0) Prelab. Assuming a software-scheduled pipeline (i.e., no hazard or forwarding logic), come up with a global list of the datapath values and control signals that are required during each pipeline stage. Consider the following:

- The control unit does not need to be changed significantly (if at all) from the version that you created for Project Part B.
- You should implement the branch condition logic in the ID stage (as illustrated in the incomplete schematic above). You can assume that the branch delay slot will be filled with a useful instruction or a NOP that is always executed.
- Beyond the generated control signals, keep in mind what datapath values (e.g. Rt register address, ALU output) that needs to be stored to ensure proper operation of a pipeline.
- Specific to the CprE 381 testing framework, your halt signal should not become active until the `syscall` instruction reaches the writeback stage, otherwise your processor will appear to miss the execution of the instructions immediately before the `syscall`.

Phase I: Software-Scheduled Pipeline

1) From a datapath perspective, what distinguishes this processor from the single-cycle version is the presence of **pipelined registers**, which are used to store intermediate control and data values after every stage. Although these registers can be implemented using the generic N-bit register, it is recommended that you create individual IF/ID, ID/EX, EX/MEM, and MEM/WB registers that include as ports on the entity declaration the names of the individual control and datapath signals to be stored.

(a) Depending on control and data dependencies, each pipeline register may need to be *stalled*, in order to prevent writing of new values, or *flushed*, to remove stored values entirely. Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.

(b) Implement the pipeline registers using whatever style of VHDL you prefer, and create a testbench that instantiates all four of the registers in a single design. Show that values that are stored in the initial IF/ID register are available as expected four cycles later, and that new values can be inserted into the pipeline every single cycle. This testbench should also test that each pipeline register can be individually stalled or flushed. *[This is a crucial part of the project. Verify with your TA before proceeding.]*

2) Insert these registers into your single-cycle design to create a **MIPS S**oftware-Scheduled Pipelined Processor**. For now, you do not need to worry about controlling the values of the stall and flush signals (set them to '0'), since this version of the pipeline relies on software to insert NOPs or reorder operations to avoid control and data hazards. In your writeup, provide your schematic for this part, describe what challenges (if any) you faced in implementing this module.

3) Test your processor to ensure that it can still fully implement all of the instructions from Project Part B. As before, continue to use the automated testing framework. In your writeup, show the ModelSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly:

(a) Write a relatively small program that uses all instructions supported by your pipeline and avoids all control and data hazards. Include this file in your submission as `Proj-C_test1.s`.

(b) Modify your previous Bubblesort implementation such that it avoids all control and data hazards. Do not just add four+ NOPs in between every instruction – this will result in performance as bad as the single-cycle design! Instead, take your time and add only those NOPs as absolutely necessary for correctness. Include this file in your submission as `Proj-C_test2.s`.

4) As in the previous assignment, once you have completed your software-scheduled pipeline you will **synthesize** it to the DE2 board's FPGA to determine the maximum cycle time. Using the same synthesis procedure as in Project Part B, report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

Phase II: Hardware-Scheduled Pipeline

5) Determining **data dependencies** is a first step in creating data forwarding and hazard detection logic. We are implementing a much larger set of instructions then what is portrayed in P&H chapter 4, and consequently there are several more potential sources of dependencies. In general, the RAW dependencies we are worried about exist whenever an instruction that *produces* a value is followed by an instruction that *consumes* that value. In order to simplify this analysis, it is recommended that you create the following lists. *[These steps can be done independently from the prelab and part 1).]*

(a) Of the MIPS instructions supported for Project Part B, list which instructions produce values, and what signals in the pipeline these correspond to. *[Any instruction that writes to the register file or data memory produces a value.]*

(b) List which of these same instructions consume values, and what signals in the pipeline these correspond to. *[Instructions can both produce and consume values (e.g. `add $1,$2,$3`).]*

(c) Given this *N* x *M* list of producing signals and consuming signals, come up with a generalized list of potential data dependencies. From this generalized list, select those dependencies that will require forwarding (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls.

6) You should now be ready to write a more generalized series of **data forwarding** and **hazard detection logic** equations based on the result from part 5). These should be of the format of those found in P&H 4.7, but there will be several more types of dependencies to consider. Given a set of equations, it will be straightforward to implement and test your forwarding and detection units in VHDL. It is recommended that you do not integrate them into the rest of the datapath until you are confident that the software scheduled pipeline executes properly.

7) At this point, the major components should be in place for you to be able to implement the **MIPS Hardware-scheduled pipelined Processor** using only structural VHDL. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components. Some general hints as how to proceed:

- Forwarding logic will require additional muxes in front of the functional units that consume data. There is no harm with initially skipping a few and adding them (or

widening existing muxes) as you begin to test different instructions, but it is important to label them appropriately to keep the design readable.

- While P&H implies that writeback data hazards are preventable by writing to the register file on the negative edge of the cycle, a more hardware-amenable strategy would be to implement forwarding in the writeback stage. To do this, you will have to modify the register file in order for reads to get the new value for the register that is being written to. One simple way is for registers to have the new value be "forwarded" or "bypassed" around the actual register when it is being written to.
- Do not implement branch or load delay slots. Instead, simply stall the IF or EX stage when necessary. The MARS implementation in the CprE 381 processor testing fraemwork is configured to not include delay slot instructions.
- Similar to Project Part B (and many groups did not implement this properly), your processor will need to be "reset", in the sense that the pipelined registers are cleared and the PC is set to some predetermined initialization address. Implementing this as a single control signal from a VHDL testbench will be much easier than implementing a series of "reset" instructions.

8) Testing your processor will require more effort than what you have done for past labs, as the interaction between instructions now potentially affects every single component. In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly. *[You should use MARS simulator to test that your test applications work properly in simulation, then use the processor toolflow to confirm that the results are consistent with what you are seeing from your VHDL implementation.]*

(a) Verify that your test applications from Project Part B work on this processor without requiring further modification.

(b) Create an application that attempts to exhaustively test the forwarding and detection logic in your pipeline. Include this file in your submission as `Proj-C_test3.s`.

9) You will be expected to **demo** your pipelined implementation to the TAs by the project due date. Each member of the project group will be required to be present for the demo, which will take place during regular lab hours or office hours During this time, you will describe the various components of your design and how they work together, you will show simulation of the test applications from Part 8), and you will also run an additional test application that will only be provided to you during the demo.

10) As in Part 4), once you have completed your hardware-scheduled pipeline you will **synthesize** it to the DE2 board's FPGA to determine the maximum cycle time. Using the same synthesis procedure as in Project Part B, report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematic. In your writeup, briefly discuss your critical path results. What components would you focus on to improve the frequency?