CprE 381 – Computer Organization and Assembly Level Programming

Project Part 2

[Note: Developing two pipelined processors is the second part of the term project, and similar to the single-cycle processor, will involve substantial design, implementation, integration, test, and synthesis tasks. You have four weeks to complete the assignment and I have broken the design into two parts that you will evaluate individually. In the first part, you will design a pipeline that will always be working on five instructions—one in each stage. It will have no hazard (control or data) detection, no stalling, and no forwarding logic. In order to avoid such hazards, software must carefully schedule instructions (i.e., re-order instructions or insert NOPs). In the second part, you will add hazard detection, stalling, and forwarding logic such that any program that runs on your single-cycle processor can run unmodified on your pipelined processor. WARNING: the hardware implementation of hazard detection, stalling, and forwarding can be tricky and time-consuming to debug. I suggest that you complete the design and implementation of the software-scheduled pipeline as soon as possible.
***Disclaimer: Due to the complexity of the assignment, this document is subject to minor change between now and the due date – I will post any updates to Canvas so please continue to check Canvas regularly**.]*

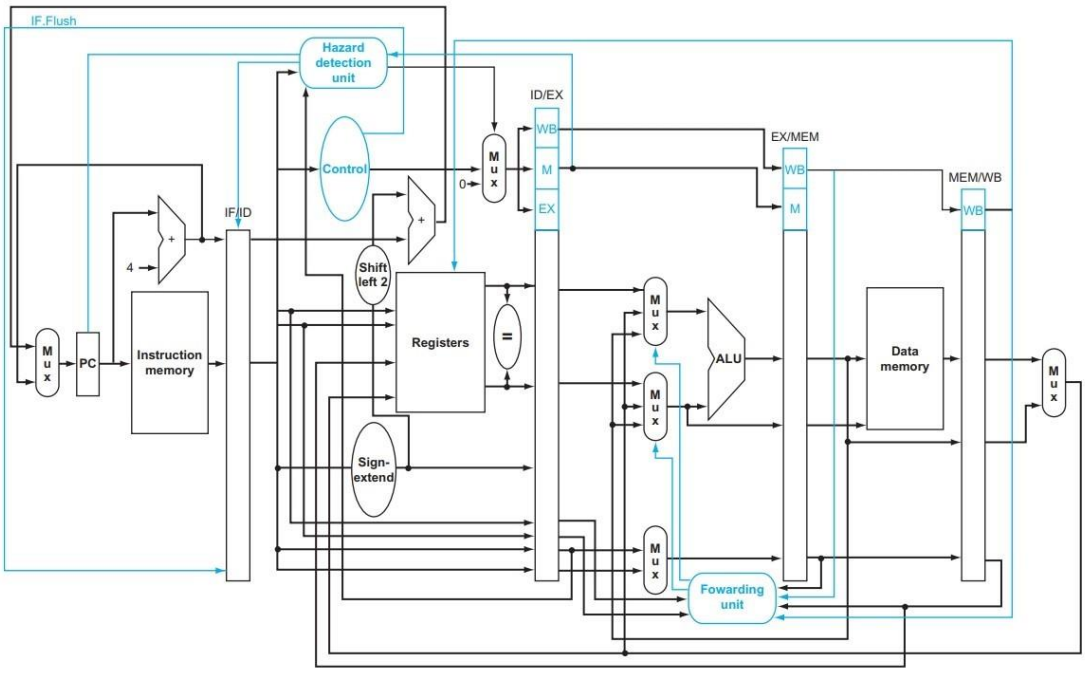

FIGURE 4.65 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.57 and the multiplexor controls from Figure 4.51.

0. **Planning and Teaming.** You will be continuing to work in the teams you completed the first part of your project with. Having completed a full design is a great opportunity to review how your team is functioning and how to improve on your past performance.

a. *Identify worked well for your team during the previous portion of the project and specifically agree to build on these approaches going forward.*

b. *Identify reasons your team did not perform as well as it could have on the previous portion of the project and brainstorm approaches to mitigating these going forward. Agree on a specific set of mitigations going forward.*

c. *Identify which team members will be responsible for which components of the project. Specifically identify the deadlines for each deliverable. Note that this part of the project should be easier to determine deadlines given your past experience with the project.*

d. *Complete and submit the team contract that reflects to above planning and teaming.*

1. **Software-Scheduled Pipeline.**

   a. **Control Signals.** Come up with a global list of the datapath values and control signals that are required during each pipeline stage. Consider the following:

      i. The control unit does not need to be changed significantly (if at all) from the version that you created for your single-cycle design since we are supporting the same instructions.

      ii. You can implement the branch condition logic in the ID stage (as illustrated in the INCOMPLETE schematic above). Note that this could cause your critical path to be different, in particular, if you choose to use a negative-edge triggered register file. Do not worry about the performance now, but remember this decision in the next part of your project when you analyze the relative performance of your designs.

      iii. **Test framework specific: Your FINAL halt signal (s_Halt) should not become active until the `halt` instruction reaches the writeback stage, otherwise your processor may appear to miss the execution of the instructions immediately before the `halt`.**

   b. **Datapath.** From a datapath perspective, what distinguishes the software-scheduled pipeline 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 in the entity declaration the names of the individual control and datapath signals to be stored.

      i. Given your list from part 1.a above, implement each of the pipeline registers using whatever style of VHDL you prefer.

      ii. Insert these registers into your single-cycle design to create a software-scheduled pipeline (i.e., a pipeline without interlocking pipeline stages). You do not need to worry about stalling since the pipeline relies on software to insert NOPs or reorder operations to avoid control and data hazards. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components.

iii. Do NOT update your register file to bypass or forward.

iv. Save the final VHDL implementation of your software-scheduled pipeline so you can compare it with your other designs – you will need to submit this separately in the src_sw directory. In addition, if you find issues with the components (e.g., adder), please continue to update these files.

c. **Testing.** You must now test your software-scheduled pipeline design. To do this you will need to develop a series of assembly programs where instructions are carefully scheduled to avoid control and data hazards. Note that the provided automated testing framework will work for this processor as well (go ahead and think through how the 5-stage pipeline will produce the same sequence of memory and register writes – this is part of what makes MIPS easy to pipeline).

i. Write a relatively small/simple program that uses all instructions supported by your pipeline and avoids all control and data hazards. Test that your processor correctly runs this program and include an annotated waveform in your writeup and provide a short discussion of result correctness. Include the .wlf file demonstrating this program working in your code submission.

ii. Modify your bubblesort program from the previous project submission such that it avoids all control and data hazards (i.e., such that they will run correctly on your software-scheduled pipeline. **Do not just add four or five NOPs in between every instruction – this will result in performance worse than the single-cycle design!** Instead, take your time, rescheduling instructions or adding only those NOPs absolutely necessary for correctness. Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did not have to use the maximum number of NOPs. Include the .wlf file demonstrating this program working in your code submission.

d. **Synthesis.** As we have repeatedly discussed in lecture, performance (runtime for our purposes) depends on several factors: # of instructions that will be dynamically executed by an application, # of cycles each instruction takes to execute on average, and the length of the cycle. Up to this point you know the first two factors for our applications. 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 the previous project portion, 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).

2. **Hardware-Scheduled Pipeline.**

a. **Pipeline Register Update.**

i. 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.

ii. Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.

iii. Update your pipeline register implementations to include stalling and flushing/squashing. 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. Most importantly, 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 an instructor before proceeding.]*

b. **Data Hazard Avoidance**. Determining data dependencies that exist in the pipeline is a first step in creating data forwarding and hazard detection logic. We are implementing a 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).]*

i. Of the MIPS instructions you must support, list which instructions produce values, and what signals (i.e., bus names) in the pipeline these correspond to. *[Any instruction that writes to the register file or data memory produces a value. Note that a produced value may be present on multiple signals as its instruction propagates through the pipeline – list all signals!]*

ii. 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`). Make sure to consider that a value could be consumed at multiple signal locations. The store instruction presents a tricky edge-case.]*

iii. Given this *N* x *M* list of producing signals (*N*) and consuming signals (*M*), come up with a generalized list of potential data dependencies. From this generalized list, select those dependencies that can be forwarded (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls. *[Remember, stalling can always be used for correctness, but will increase the effective CPI of the pipeline.]*

iv. Update your global list of the datapath values and control signals that are required during each pipeline stage to include the datapath values (e.g., Rt register address) that need to be stored to ensure proper operation of the pipeline with hazard detection / forwarding.

v. You should now be ready to write a more generalized series of data forwarding and hazard detection logic equations based on the result from the previous part. These can be of the format of those found in P&H 4.7 (or of the similar dataflow constructs you are familiar with), but there will be several more types of dependencies to consider. At this point you should implement and test your forwarding and detection units in VHDL, but I recommend that you do not integrate them into the rest of the datapath until you are confident that non-data dependent code executes properly. *[Note that hazard detection logic is impacted by the forwarding paths present. If a*

forwarding path exists to mitigate a specific data hazard, a stall is no longer required.]

c. **Control Hazard Avoidance.**

   i. Of the MIPS instructions you must support, list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs. *[Note that for the INCOMPLETE pipeline figure shown above, there is only one stage where control hazards are detected and action on. However, other figures from the book and your own design choices may result in instructions such as beq needing hazard avoidance in multiple stages.]*

   ii. For these instructions, list which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each of these instructions is in. I encourage you to start with no prediction and no delay slots and only add at a later point if you are interested.

d. **Hardware-Scheduled Pipeline.** At this point, the major components should be in place for you to be able to implement the hardware-scheduled pipeline 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:

   i. Start by implementing your hazard detection logic assuming no forwarding hardware. This will get you to a completely functioning hardware-scheduled pipeline faster. You can then incrementally improve the performance by adding forwarding and, for larger groups, control prediction.

   ii. 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. Each time you add a new forwarding path, remove it from the hazard detection logic.

   iii. 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 "bypass" around the actual register when it is being written to.

   iv. Do **NOT** implement branch or load delay slots. Instead, simply stall the IF or EX stage when necessary.

   v. Similar to previous project parts, 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.

e. **Testing.** Testing your processor will require more thoughtful effort than past project parts, as multiple instructions interact directly in the pipeline in every cycle. In your writeup, show the QuestaSim 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, convert the assembly to MIPS machine code, and confirm that the results are consistent with what you are seeing from your VHDL implementation.]*

i. Create a set of assembly programs that exhaustively tests the data forwarding and hazard detection capabilities of your pipeline. Minimally you should create one assembly program for each of your hazard detection and forwarding cases. Then you should create a set of test programs that activates combinations of your data hazard detection and forwarding cases that can occur simultaneously within your pipeline. Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

ii. Create a set of assembly programs that exhaustively tests control hazard avoidance. Minimally include one test program per control flow instruction. Then you should create a set of test programs that activates combinations of these instructions in the pipeline. Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

iii. Verify that your three test applications you created *for your **single-cycle** processor* work on this processor without being modified.

f. **Synthesis.** As we have repeatedly discussed in lecture, performance (runtime for our purposes) depends on several factors: # of instructions that will be dynamically executed by an application, # of cycles each instruction takes to execute on average, and the length of the cycle. Up to this point you know the first two factors for our applications. Now you will synthesize our design to the DE2 board's FPGA to determine the maximum cycle time. Using the same synthesis procedure as in the previous project portion, report the maximum frequency your hardware-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

SUBMISSION:

- Complete the lab report (using template *Proj2_report.doc* on Canvas). Generate a PDF of the report.
- Use the toolflow to generate your submission by using the "submit" command of the shell script ("381_tf.sh" – consult the toolflow manual, part 2.4, for more information) for both your software and hardware scheduled parts. The generated submission will follow the structure given in *Proj1.pdf* – simply add sw or hw suffixes as shown for each submission generation. The final submission structure will be as follows:
    1. *submit.zip*
        a. src_sw files folder containing all of your VHDL *design* files for the software-scheduled implementation.
        b. test_sw files folder containing all of your VHDL *testbench* files for the software-scheduled implementation.
        c. mips_sw files folder containing your test assembly programs for the software-scheduled implementation.
        d. src_hw files folder containing all of your VHDL *design* files for the hardware-scheduled implementation.

e. test_hw files folder containing all of your VHDL *testbench* files for the hardware-scheduled implementation.
f. mips_hw files folder containing your test assembly programs for the hardware-scheduled implementation.
g. *Proj2_report.pdf*

2. *Proj2_report.pdf*

- Submit the ZIP and report PDF files on Canvas under the "Project Part 2: Pipelined MIPS Processor" assignment in a single submission. **Note: Even though the report is included in the ZIP file, you must also submit it as a PDF separately from the ZIP on Canvas in the same submission as shown above in the submission structure.**

Credit: Parts of this project description were originally created by Dr. Joe Zambreno.