CprE 381 – Computer Organization and Assembly Level Programming, Spring 2018

Project Part 3

[Note: this pipelined processor is the third part of the term project, and similar to the single-cycle processor, will involve substantial design, implementation, integration, test, and synthesis tasks. You once again have four weeks to complete the assignment and I have broken the project into two parts. The first part, which is two-weeks long, is to generate a pipeline with 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. The second part, which makes up the remaining two weeks, requires you to generate hazard detection, stalling, and forwarding logic. WARNING: the hardware implementation of hazard detection, stalling, and forwarding can be tricky and time-consuming. I suggest that you complete part a as soon as possible so you can begin working on part b.
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.]

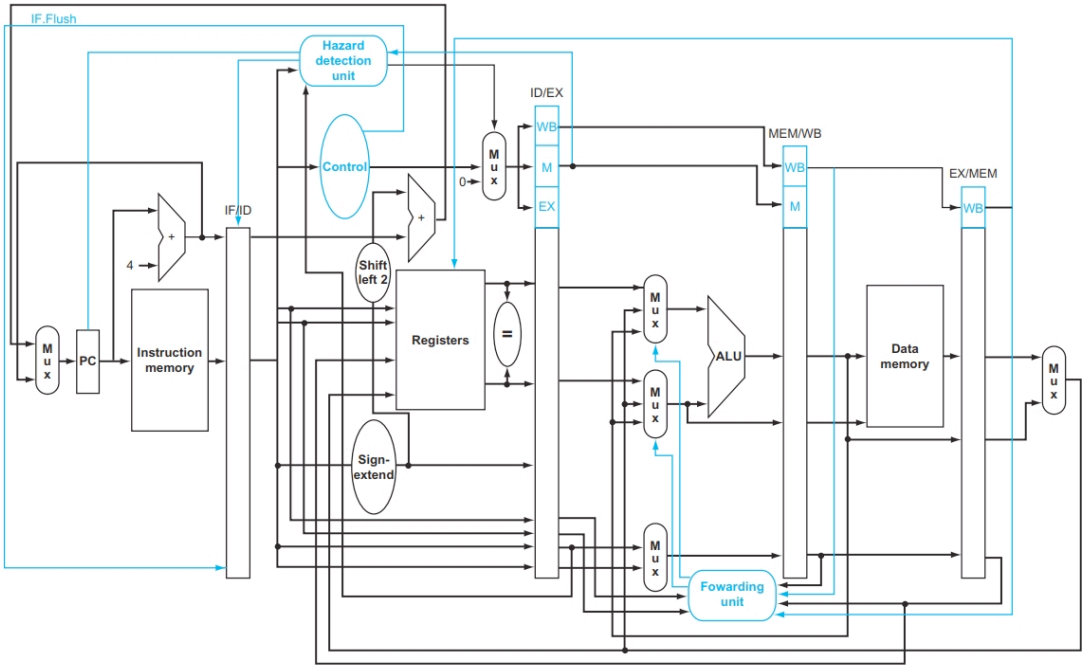

FIGURE 4.65 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.57 and the multiplexor controls from Figure 4.51.

0) Prelab. Review pipeline lecture notes. Reflect on how long and how much effort Part 2 took your team. Ensure you have a "can-do" attitude.

Part (a): A Software-Scheduled Pipeline

1) 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 2.
- 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.

2) 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 on the entity declaration the names of the individual control and datapath signals to be stored.

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

3) You must now **test** your software-scheduled pipeline design. 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. Then modify your merge-sort and bubble sort programs from part 2 such that they avoid all control and data hazards (i.e., such that they will run correctly on your software-scheduled pipeline. 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. Include the .wlf file demonstrating this program working in your code submission. You have two options to test your design:

(a) Write your own testbench and toolflow similar to what you used in Project Part 2. This may require a lot of manual checking of waveforms and will likely be more challenging to debug.

(b) Use the test framework from the course website (no implied warranty). This will take an assembly file, compile it, simulate it on your processor and compare each state update to that of a MARS simulation. Once you've set this up it will provide an easy, automated check of functionality and in the case of errors will provide a specific cycle at which to begin debugging.

4) 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 Project Part 2, 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). Additionally, include a graph of the slack distribution from Quartus.

Part (b): A Hardware-Scheduled Pipeline

5) 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, ALU output) that needs to be stored to ensure proper operation of the pipeline with hazard detection / forwarding.

6) 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.

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

(b) Update your pipeline register implementations to include stalling and flushing. 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 your TA before proceeding.]*

7) Determining **data dependencies** 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).]*

(a) Of the MIPS instructions supported for Project Part 2, 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.]*

(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 (*N*) and consuming signals (*M*), 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.

8) 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 should be of the format of those found in P&H 4.7, 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.

9) 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:

- 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.
- 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.
- Do not implement branch or load delay slots. Instead, simply stall the IF or EX stage when necessary. If you are using your own testing infrastructure, you may need to modify your SPIM or MARS toolflow to ensure your hex files do not include delay slot instructions.
- Similar to Project Part 2 (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.

10) Testing your processor will require more effort than what you have done for past labs, as multiple instructions interact directly in the pipeline in every cycle. 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 the SPIM or 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.]*

(a) Verify that your three test applications from Project Part 2 work on this processor without being modified.

(b) Create an application that tries to exhaustively test the forwarding and hazard detection capabilities of your pipeline. You should be able to use your HW8 solutions as a starting point.

11) 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. During this time, you will describe the various components of your design and how they work together, you will show simulation of the three test applications from Part (b.10), and you will also run a fourth test application that will only be provided to you during the demo. *[Your processors and toolflow are not expected to be perfect!]*

12) 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 Project Part 2, 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). Additionally, include a graph of the slack distribution from Quartus.

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