1 Pipelined SPIM VHDL Model: Overview

This document contains the description of a VHDL model of the pipelined SPIM datapath from *Computer Organization and Design: The Hardware/Software Interface 4th Edition.* The description focuses on the basic components, their operation, and how to make modifications. Note that the VHDL model described here may differ in minor ways from the text.

Further, note that this model is rather verbose in the explicit instantiation of signals and components at a lower level of detail that found in the behavioral Verilog models you might find on the web. However, the number of actual operational lines of code is comparable. It is the explicit use of pipeline registers, the exposure of all component signals in each pipeline register, and the interconnection of all of these modules/signals in the top level *spim_pipe.vhd,* that takes up space. This was deliberate to enable explicit manipulation of low-level features of design in assignments so as to reinforce the understanding gained from the text.

2 Installing and Testing the Model

To gain an understanding of how the model is organized and to ensure that you have a correct, functioning model, execute the following steps using the model supplied in class. The following steps are predicated on the use of the ModelSim tool set and it assumed that you have been through the basic ModelSim tutorial and therefore familiar with the basic functionality.

- Create a project directory called myproj. Copy the *PS_SPIM.zip* file into this directory and extract the files. Check the README file and make sure that you have all of the VHDL files. The top-level file is *spim_pipe.vhd*. This file contains the VHDL code with declarations of all of the pipelined datapath components, declarations of all of the signals used to connect these components, and component instantiation statements that instantiate & interconnect the components. .
- You must add all of the files to the project. You can do this from Project →Add To Project to add an existing source file. In the dialog box you can select the option to copy this file into the project directory.
- Compile all of the modules with Compile → Compile All
- Click on the library tab in the browser window (this is the left window in the ModelSim GUI). Select the file spim_pipe under the option work (you can expand the selection work) for simulation. Right click and select simulate. This will bring up the sim tab in the ModelSim browser window. In this tab select (right click) Add To → Wave → All Items in Region. The selected signals are those you selected from the adjacent window that exposes all of the signals. This will bring up the waveform viewer. I suggest selecting the individual multi-bit signals and set their viewing options to hexadecimal rather that binary for clarity (right click Radix→ hexadecimal). Select at least signals that include

the PC, instruction, and contents of the registers being read so as to be able to determine what is being executed in the pipe.

- Simulate for 800 ns. You should now see the execution trace. This trace is meaningful only if you know what you are looking at. A few observations below.
    - A few things to note. In this pipelined model, the mux in the last stage has been removed and the signals have been run directly into the decode module. This means the choice of operand to store (whether from memory or the ALU output) must be made inside of the decode module. Hence the MemtoReg control signal must also be provided to the decode module. As for signal naming conventions, all the signals used to connect the modules are prefixed with if_, id_, ex_, mem_ or wb_ depending in the stage in which the signal is used (note: if a signal is generated in WB and used in IF it will be prefaced with wb_).
    - All of the signals driven out of a pipeline stage are explicitly connected to a pipeline register. The naming convention should be able to make it easy to follow the (admittedly verbose) interconnections in top-level spim_pipe.vhd module. For example at line 263 (or thereabouts) where the instruction fetch module is instantiated, the signal PC_Out from IF is connected to the signal if_PC4. This signal in turn is an input to the pipeline register pipe_reg1 (line 270). Note that if_PC4 is connected to pipeline register input if_PC4. If you now examine the contents of *pipe_reg1.vhd*, you will find the input signal if_PC4 is connected to the pipeline output register signal id_PC4. This is as shown in the pipelined datapath figure for the IF/ID register. In a similar manner all signals coming out of a pipeline stage's combinational logic can be identified easily and manipulated.
    - On the trace, the top set of signals are all named such that each signal name is preceded by "out_". Each of these signals has been pulled from one pipeline stage. So that by looking at the top 16 signals you have some information from every pipeline stage. The signal name further defines the stage and signal. For example, the signal out_ex_rt is the contents of the rt signal in the execute stage. This corresponds to the contents of the rt register for the instruction that is currently in the execute stage. If you need more detail the remaining signals correspond to all of the signals exposed between all stages of the pipeline. The signal name will identify the pipeline stage.
- The trace is the execution of the program contained in instruction memory located module *ps_fetch,vhd*.
- The cycle time of datapath is 100 ns (check the module *ps_clock.vhd* that generates the clock and reset pulse).
- The datapath generates an 80 ns reset pulse on startup. Thus the first rising edge of the clock will see reset high and can be used to initialize signals and registers.

3 Model Overview

The following short descriptions of each module are intended to help quickly come up to speed on the specific implementation. All quantities including the PC, Branch Address, and data memory are 32 bits. However, only some bits of the PC are used to address instruction memory - a function of the size of instruction memory.

3.1 spim_pipe.vhd

This is the top level model that instantiates all of the individual modules, connects them to each other, and connects some subset of local signals to the Modelsim interface. Note the naming convention described above. The entity interface simply picks few signals to export to Modelsim. This is useful when you only want to see a few important signals and do not want all of the local signals cluttering the trace.

The individual modules once instantiated are given local names. For example the entity Ifetch in *ps_fetch.vhd* is instantiated here as IFE. Check the component instantiation statements for the names of each instantiated component.

3.2 ps_fetch.vhd

Note that the code is organized into a clock sensitive portion and a combinational component (as are the other modules). These statements generally track the combinational and sequential components of the design as shown in the figure in your text. Be careful when constructing (modifying) modules to avoid feedback loops through the design.

3.3 ps_decode.vhd

The decode module is relatively straightforward – extracting fields from the instruction, reading registers, propagating write addresses etc. The MemToReg multiplexor is implemented in this module rather than in a separate write-back module. **Note that this module does not forward write results to read operations in the same clock cycle.**

3.4 ps_execute.vhd

The execute module is relatively straightforward implementing the functionality shown in the figure text. Note that the ALU control logic is implemented in this module. The branch address is constructed in this module and the RegDst multiplxor is implemented here.

3.5 ps_control.vhd

Straightforward implementation of pipeline control.

3.6 ps_clock.vhd

Rather than use a stimulator provided by the simulation tool or rely on a test bench, the model uses a simple clock generator module that generates a 10 MHZ clock and an 80ns reset pulse on start-up. The clock period and reset pulse is easily changed although a little thought will convince you that the actual value is irrelevant.

4 Executing the Model

4.1 Loading Programs

Instruction memory is modeled in the `IF` unit. which is described in *ps_fetch.vhd*. Therefore if you wanted to change the program being executed by this datapath, follow these steps.

- Assemble your program manually or use the SPIM assembler.
- Edit *ps_fetch.vhd* to initialize the contents of the memory words to the encoded values of your assembled program. Extend the size of the array as much as you need. However, remember that if you make the array larger you may need to modify code that uses only a few bits of the PC to address the instruction memory array. For example with 8 words of instruction memory, the model uses bits `(4 downto 2)` of the program counter to index this array to prevent array out of bounds errors during simulation. However, errors due the program counter wrapping around may still occur. Update code to reflect the assembled program size.
- Re-compile *ps_fetch.vhd* and execute the simulation as described earlier.

4.2 Extending the Datapath

Add new instructions to the datapath via following these steps.

- Determine the operations to be performed to execute the instruction.
- Decide which module(s) will implement the desired functionality.
- Modify the computations in each affected module. **Test each module separately!** This can be done by simply clocking the inputs and checking the values of the outputs using the waveform viewer.
- If you have added new signals to any module's interface, some other module will be using that value (or providing that value). Therefore you need to add new signals between modules. You can do so as described in Section 4.3.
- Load a new test program into *ps_fetch.vhd.*
- Re-Compile, load and execute the model as described above.

4.3 Adding New Signals to Datapath

Let us suppose that you have modified the module *ps_execute.vhd* to create a jump address that you now wish to transmit back to the `IF` module. You need the following sequence of steps.

- Add a new port to *ps_execute.vhd* to be of type `out` with the name of this new signal.
- Add new port to *ps_fetch.vhd* corresponding to the new jump address. This port must be of type `in` and clearly have the same number of bits.

- Modify the corresponding component declarations in *spim_pipe.vhd* to include this new port in the component declarations for IF and EX modules.
- Declare a new signal, say ex_jump_address *spim_pipe.vhd.* This is the signal that we will use to connect the new port of *ps_execute.vhd* to the new port in *ps_fetch.vhd.* It should have the same number of bits.
- Add a single line to the component instantiation statement for EXE.

  ```
  jump_address => ex_jump_address
  ```

  where jump_address is the name of the new port in *ps_execute.vhd.* A similar statement appears in the instantiation statement for the IF component. Note that for some other additions it may not be quite so simple in that sometimes signals may have to be propagated through the pipeline registers and thus modifications may have to made in several pipeline stages.

4.4 Some Helpful Pointers

If signals appear as undefined, i.e., XX, this may be because

- o They are not correctly connected. Check *pim_pipe.vhd* to ensure that all signals are correctly connected or even connected at all. If they are not connected at all they will appear as XX on the trace.
- o Some un-initialized signal is being propagated through the design. Operations on an uninitialized can result in XX appearing on a signal at the output of the module.
- o Multiple sources simultaneously driving the signal to different values.
- o The occurrence and display of XXX is simulator specific so be careful in interpreting what it means.

- If memory addresses are out of range, this will not cause an error. The address calculation will simply wrap around the array. Therefore the unexpected execution of an instruction at some unexpected location or the storage of data at some unexpected memory address may be because of address out of range.
- If the ALUSrcB multiplexor is set incorrectly (for example the value is U or X) then the input to the ALU is selected as 0XBBBBBBBB.
- If the ALU receives an illegal value of an ALU opcode (remember the 3-bit opcode for the ALU) then the output is set to 0xFFFFFFFF. This might occur if for example the Function field of the instruction had incorrect, e.g., U or X, values.
- The register file is initialized to specific values that can be helpful in debugging. Check the values in *ps_decode.vhd*.
- The future value of the PC (to be updated on the next clock) will be set to 0xCCCCCCCC if there is no valid value for the PCSource multiplexor.