1 Multi-Cycle SPIM VHDL Model: Overview

This document contains the description of a VHDL model of the multi-cycle SPIM from *Computer Organization and Design: The Hardware/Software Interface*. 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.

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 *MS_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 *MS_MIPS.vhd*. This file lists all of the components and describes how all of the components are connected.
- You must add all of the files to the project. You can do this from File →Add To Project to add an existing source file. In the dialog box select the option to copy this file into the project directory.
- Compile all of the modules.
- Click on the library tab in the browser window (this is the left window in the ModelSim GUI). Select the file mips 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) mips → Add → Add To Wave. 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).
- Simulate for 1800 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.
    - All local signals in the top level module, *MS_MIPS.VHD* begin with an "s_", r example s_instruction is a signal that connects the IFE and ID modules (see Section 3.1 to know what these mean) and carries the hexadecimal value of the encoding of the currently executing instruction. Signals with the "_out" suffix, are simply copies of some of the local signals that are pushed out through the entity interface of the *MS_MIPS.VHD*model. These are there primarily for legacy and other simulation tool related issues.
    - s_PC_Out is the current value of the program counter.
    - s_micropc is the address of the microinstruction being currently executed (from *MS_CONTROL.VHD*)
    - The signal names largely follow those from the figure in your text and class vugraphs the prefixes and suffixes as described above.

  - read_data_1, read_data_2: These are the contents of the registers rs and rt and is provided by ID. This means these were the values read from the register file when the instruction was in ID.

- The trace is the execution of the program contained in instruction memory located module *MS_IFETCH.VHD*.
- The cycle time of datapath is 100 ns (check the module *MS_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 model are intended to help quickly come up to speed on the specific implementation. Note that unlike the single cycle SPIM VHDL model, all quantities including the PC, Branch Address, and data memory are 32 bits. However, only some bits of the PC are used as a function of the size of instruction memory.

3.1 MIPS.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 interface. Note the naming convention – the prefix "s_" on a signal denotes a local signal that interconnects two modules. The suffix "_out" on a signal denotes a local signal that is connected to the entity interface. The entity interface simple picks few signals to export. This is useful when you only want to see a few important signals and do not want all of the local signals to cluttering the trace.

The individual modules once instantiated are given local names. For example the entity Ifetch in *MS_IFETCH.VHD* is instantiated here as IFE. Similarly the entity Idecode in *MS_IDECODE.VHD* is instantiated here as ID. The remaining modules are instantiated as EXE, CTL and MC. Check the component instantiation statements.

3.2 MS_IFETCH.VHD

In the multi-cycle datapath model, instructions and data share the same memory. However, data memory and instruction memory are implemented as two physically distinct arrays. Consequently the control signal IorD is used in a manner different from the text. Two local signals, Local_IR and Local_data, read from the instruction and data memories respectively. These local signals are set to default values or the contents of their respective memories depending on the value of IorD. Depending on the execution state, at the next clock edge, one of the values may be driven out of the module. 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 MS_IDECODE.VHD

The decode module is relatively straightforward – extracting fields from the instruction, constructing read and write (using the RegDst control signal) addresses, and constructing the sign extended offset. Read and write operations take place at a clock edge.

3.4 MS_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.

3.5 MS_CONTROL.VHD

The control module provides a ROM based implementation of the controller. A 20 bit microinstruction is used. The ROM is implemented as an array of 20 bit words initialized to the microprogram described in the text. The dispatch tables are small enough they are implemented as simple conditional statements. The combinational part of the controller implements the dispatch tables and microinstruction update. The clock edge controls the output of the microsequencers multiplexor that updates the address of the next microinstruction. The microinstruction format is apparent from the text.

3.6 MS_CLOCK.VHD

Rather than use a stimulator provided by the simulation tool or rely on a testbench, the model uses a simple clock generator module that generates a 10 MHZ clock and a 8ns 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 *MS_IFETCH.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 *MS_IFETCH.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 *MS_IFETCH.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 *MS_IFETCH.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 *MS_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 *MS_EXECUTE.VHD* to be of type `out` with the name of this new signal.
- Add new port to *MS_IFETCH.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 *MS_MIPS.VHD* to include this new port in the component declarations for `Ifetch` and `Execute`.
- Declare a new signal, say `s_jump_address` *MS_MIPS.VHD.* This is the signal that we will use to connect the new port of *MS_EXECUTE.VHD* to the new port in *MS_IFETCH.VHD.* It should have the same number of bits.
- Add a single line to the component instantiation statement for `EXE`.

  `jump_address => s_jump_address`

  where `jump_address` is the name of the new port in `MS_EXECUTE.VHD` A similar statement appears in the instantiation statement for `IF` component.

4.4 Some Helpful Pointers

- If signals appear as undefined, i.e., `XX`, this may be because
  - They are not correctly connected. Check *MS_MIPS.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.
  - 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.
  - Multiple sources simultaneously driving the signal to different values.

  - 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 microcode sequencer erroneously receives a negative address, the microinstruction will receive the value 0x12340. Note the microinstruction is 20 bits and this value will cause specific behavior by the datapath which you can discern by knowing the purpose and function of each bit in the microinstruction (see MS_Control.vhd). This "should" only happen during initialization and be corrected by reset.
- If the ALUSrcA or ALUSrcB multiplexors are set incorrectly (for example their values are U or X) then the inputs to the ALU are selected as 0xAAAAAAAA and 0XBBBBBBBB respectively.
- 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 such that register i has the value i.
- 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.