DRAFT Lab 1
Processor Operation and Address Decoding

This exercise provides an opportunity to review the operation of a five-step RISC-style processor and how that operation relates to the concept of address decoding, *which is fundamental for microprocessor interfacing and embedded systems*. For this exercise, address decoding is explored in the context of read-only memory (ROM) and writable random-access memory (RAM) within a system based on the same VHDL processor implementation that was used in ELEC 274. Simulation will be performed to observe the processor/memory interaction along with decoding to enable memory elements.

A potentially effective way of learning about microprocessor interfacing and embedded systems is to select a realistic and nontrivial application, derive specifications for it, construct hardware on a board with one or more chips, develop the full embedded software, perform testing and verification, pursue any redesign and reimplementation, and then finalize the design and implementation in a manner suitable for full-scale production. This approach is, however, extremely time-consuming and expensive.

Fortunately, for the hardware, FPGA technology provides an opportunity to pursue more rapid prototyping that still reflects the same philosophy of functionality-based partitioning as that for a board-level design. The difference, however, is that all components are integrated within the reconfigurable logic of a single FPGA chip. No tedious and error-prone physical wiring is necessary, and no costly printed-circuit board fabrication is involved when a ready-to-use FPGA board is available. Computer-aided design tools can process logic specifications in text files to generate a hardware configuration for an FPGA chip. (Good software tools to generate executable code can also enhance the development of complex embedded software for an FPGA-based environment.)

FPGA technology can be used to provide learning experiences at a low level of abstraction for topics related to hardware interfacing. These experiences also provide preparation for advanced high-level usage of FPGA chips and for more sophisticated computer-aided design tools that support embedded hardware/software development.

Synthesizing and testing a complete (albeit simplified) processor-based system within an FPGA is an appropriate way to obtain a good grasp of address decoding. This exercise uses small memory sizes and direct VHDL descriptions of ‘memory’ devices (rather than vendor-provided intellectual property) to maintain a clear focus on the relevant logic that is used to activate a specific device when a read/write command and an address within the range assigned to that device are provided from the processor.

Part 1: Review VHDL Description of the Memories, the Processor, and the System

To support this exercise, a package is available on the course Webpage with VHDL files that describe a ROM memory, a RAM memory, a five-step processor, and a top-level system. Download the package, and extract contents to an appropriate folder on your personal computer and/or USB drive (e.g., `...\Documents\ELEC371\LABS\LAB1`). You can also use your Queen's FEAS network storage, *but when using the CAD tool to process the VHDL files, use of a USB drive is recommended for faster local file access*.

In Altera Quartus II, create a project called `system` in the folder where you extracted the package above, and add all of the provided VHDL files to the project in preparation for the later step of compilation/synthesis. (The ROM file will synthesize, but does not contain valid code for meaningful simulation results. The in-lab activity will require the preparation of a specified program to generate code for simulation.)

The five-step processor within the system for this exercise supports four instructions: load word, store word, add immediate, and unconditional branch. The hardware description is therefore simplified, but these four instructions are sufficient to write small test programs for exploring concepts related to microprocessor interfacing.

At the system level, however, there are two important aspects to the organization of the VHDL description for this exercise. *First, there are distinct VHDL entities to encompass a ROM memory for holding code and a RAM memory for holding data.* This clear separation certainly mimics the use of discrete chips in board-level designs, but it also reflects a similar modular approach with distinct entities described in languages such as VHDL for synthesis by computer-aided tools and integration within single-chip designs.

The second notable feature for the system used in this exercise is that, with distinct entities for ROM and RAM, the essential concept of address decoding is reflected directly in similarly distinct logic that dictates which memory entity should be active. That logic uses when…else… syntax, but represents AND gates for pattern matching.

Open the given files and study the VHDL descriptions provided in the package for this exercise. Any text editor is fine, but if you use the Quartus text editor (File/Open from the menubar), you can benefit from the syntax coloring it does for more readable code.

Note how the memory entity descriptions reflect the ROM size of 256 bytes (64 words) and the RAM size of 8 bytes (2 words). Note how in *system.vhd*, a "chip-select" (i.e., enable) signal is defined for each memory element. The select logic for the ROM is for a starting address of 0. The select logic for the RAM is for an address of 0x1000. Note also how the VHDL descriptions for the memory generate all-zero outputs when those memorys are not active, and how in *system.vhd* the two memory outputs are ORed together for the input to the processor (used within an FPGA – no tri-state logic).

Part 2: Prepare and Integrate Machine-Language Instruction Sequence for ROM

Like any processor, the one in the VHDL description for this exercise must be fed machine instructions in appropriate binary format. You will use the assembler/linker for the Nios II to generate the machine representation of a program. From that machine representation, you will use the `objdump` utility program to obtain a hexadecimal representation of the machine instructions, and you will copy/paste that content into the storage array definition in the VHDL file that represents the ROM component.

For initial preparation before your scheduled session, a sample program is provided (same one used with the VHDL processor in ELEC 274). But for in-lab activity, another program will be specified. Any code for this exercise must be limited to the four available instructions that the simple processor implements. The code is positioned at location 0 (start of ROM) and limited to 256 bytes. Rather than use .word or .skip directives to define data or reserve storage in memory, use the .equ directive to symbolically specify a data address in RAM (one word at 0x1000, and another at 0x1004). The reason for not using other data directives is to limit the contents of the final executable image to instructions only (no data).

The sample program below shows all the directives to use and serves as a template for the program to be implemented for in-lab activity.

```
          .set noat /* prevents msgs about using reg r1 */

          .equ DATA_LOC, 0x1000    /* first RAM location */

          .text
          .global      _start

          .org 0x0000    /* first location in ROM */

_start:   ldw  r1, DATA_LOC(r0)  /* read data from RAM */
          addi r1, r1, 1         /* increment reg value */
          stw  r1, DATA_LOC(r0)  /* put new value in RAM */
          br   _start            /* repeat */

          .end
```

Create a file **lab1.s** containing the above code. The only registers that are used are r0 and r1 because those are the only two registers that the processor supports.

Use the Windows Start Menu search feature to open the Altera Nios II cygwin command shell to use Nios II code-generation tools. For convenience, increase the shell font size for easier reading of the on-screen display within the command shell.

Assuming that the directions for using a USB drive are following, in the Nios II shell, use '`cd /cygdrive/<USB_drive_letter>/...`' to navigate to your working directory with the **lab1.s** file that you created. Use the following three commands in the given sequence to assemble the source file, process it with the linker to finalize the executable image, and then generate an output object file dump for the machine instruction sequence in hexadecimal representation.

```
nios2-elf-as  -o  lab1.o  lab1.s
nios2-elf-ld  -o  lab1.elf  -Ttext  0x0000  lab1.o
nios2-elf-objdump  -D  lab1.elf
```

The `-o` options specify output file names. The `-Ttext` option for the linker specifies the desired starting address for `.text` segment of the entire program in the final executable image. Even though `.org` may have been used in the assembly-language source file, there is still a measure of control that is provided at the link stage for specifying the position of the entire executable image of the text segment in memory.

Should you have any syntax errors reported by the assembler for your test program, edit the source file to correct them and assemble the file again.

Because you are using the command shell, you have the convenience of using the *up/down arrow keys to scroll through the command history*, so you do not need to retype the full text of any of the commands given above if you must repeat any of them.

When you have successfully created the output from the linker, examine the output from the `objdump` utility program (*note that register at is actually register r1*). There are three columns of interest: the address of each instruction, the hexadecimal representation of each instruction word, and the disassembled human-readable representation of the instruction word annotated with symbolic information where appropriate (such as the association of the symbol `_start` with the target of the branch instruction).

The hexadecimal representation of each instruction is the most relevant aspect. Each instruction must be positioned in the appropriate location in the storage array that defines the contents of the ROM for the system implementation.

In the Altera Quartus II software (with the project that you created for this exercise already open) edit the **rom256.vhd** file that you had studied in Part 1.

As given to you, the ROM has the storage array with one arbitrary word defined at address 0x0. The remaining 63 words at addresses 0x4, 0x8, etc., are all zero.

Now use the output of the `objdump` utility program to replace the initial ROM contents in the first few locations of the storage array with the hexadecimal representation of the instructions of the test program. You can type the information directly. Alternatively, to help avoid mistakes,you copy/paste as follows. In the Nios II command shell, right mouse-click over the icon in the top-left corner, select *Edit → Mark*, highlight the desired text, then press the Enter key to copy. Go to the desired position in the destination text file, then paste. For each ROM data item, insert a VHDL '– –' comment after the data to provide a description of the address and the human-readable representation of that instruction word.

Ensure that you have properly inserted the machine instructions in the ROM storage array *starting at location 0*, and that the remaining elements are still specified as zero.

Save the modified VHDL file.

Part 3: Synthesize FPGA Implementation of System

With the ROM description now containing valid instructions, synthesize the system with Altera Quartus II to ensure that there are no errors and that the result of synthesis could be implemented to operate in the hardware of an actual FPGA chip.

Click on the triangular 'Play' icon to initiate the synthesis in Quartus (or in the main menubar, select *Processing → Start Compilation*). Should any errors be reported, open the relevant VHDL files, make appropriate corrections, and attempt the synthesis again.

Successful synthesis should result in the Flow Summary report in Quartus indicating that several hundred logic elements were needed to implement the complete system. Because you did not make any explicit pin assignments, Quartus made its own choices for the pins to use for the ports on the top-level system entity. Consequently, this initial synthesis *cannot* be used as is for the Cyclone III chip in the DE0 board because it does not match the pin assignments specific to that board. But a successful synthesis means that on any custom board with a Cyclone III chip, appropriate pin assignments can be made for that board, and the implementation should operate correctly in hardware.

Part 4: Simulate Operation of System and Observe Output Waveforms

With the system successfully synthesized to confirm that it could be implemented in hardware, its operation can now be simulated functionally in order to observe the low-level behavior of the logic signals. The simulation can be performed by first creating an input waveform diagram that contains the clock and reset inputs, and all of the relevant outputs to observe the system behavior. Then, the functional simulator can be invoked from the menubar of the waveform editor.

In Quartus II, use *File → New ...* and chose *University Program VWF* under the heading of Verification/Debugging Files. The waveform editor should appear. Before adding any waveforms, use *File → Save As ...* with a meaningful name such as **system_test.vwf**. Because the **system** project should still be open as the current project in Quartus II, this waveform file will automatically be associated with the project when it is saved.

Select *Edit → Set End Time...* and change the end time to **4** *microseconds*.

For convenience, you can maximize the waveform editor window. Right-mouse-click in the white area under the “Name” heading. In the popup menu that appears, select the *Insert Node or Bus ...* option. A new dialog box should open. In that dialog box, press *Node Finder ...* button. Another dialog box should then appear for pin selection. Press the *List* button to show the available pins and multi-bit pin *groups*. Now, in the order that is given below, add the individual pins and *groups* of pins to the input waveform file. *One at a time*, click on the desired pin or pin group in the left side of the dialog box, then press on the ‘>’ button to have the pin or pin group appear in the right side of the dialog box. If you inadvertently skip over a pin or pin group, remove selections in the right side with the ‘<’ button until you are back again to the listed position of the desired pin or pin group, then continue again with selections from the left side of the dialog box.

```
clk
reset_n
ifetch_out
mem_addr_out  (the group, i.e., not the individual bits within the group)
mem_read
mem_write
rom_active
```

`ram_active`
`data_from_procr` (*the* *group*)
`data_to_procr` (*the* *group*)

When you have selected all of the signals above in the correct order, press *OK* in the Node Finder. *But don't close the Insert Node or Bus dialog box yet.*

The Insert Node or Bus dialog box should now show "**Multiple Items**" with a default radix of *Binary*. Change the radix to *Hexadecimal* so that multi-bit output waveforms that are displayed after the simulation is performed can be interpreted more easily. Then press *OK* on the Insert Node or Bus dialog box to close it.

The input waveform display should now show the pins and pin groups that were selected. You will shortly modify the default waveforms to provide a proper input for observing the simulation operation of the system.

Before proceeding further, use *File* → *Save* to save what you have done so far.

Now you will make the appropriate changes to the input waveforms to provide the stimulus for the system to execute the test program that is included in the ROM.

Under the "Name" heading, left-click on '`clk`' to highlight the *entire* waveform. Now left-click on the Overwrite Clock icon (the one with the clock graphic) in the row of icons at the top of the waveform editor. The Overwrite Clock icon is between the 'C' icon and the '?' icon. A dialog box should appear with default period of 10 ns. Change the period to 20 ns so that you will ultimately see enough of the system behavior in the simulation output. Then press *OK*. The clock waveform should now show many clock cycles.

To make the next waveform-editing step easier, zoom in by pressing ctrl-space at least once (or use *View* → *Zoom In*).

Now left-click on '`reset_n`' to highlight the *entire* waveform. Click on the '1' icon in the row of icons at the top to make the *entire* waveform a high logic input. Now click once anywhere else in the waveform grid to *un*select the highlighted portion of the reset waveform.

Ensure that the left side of the waveform grid is at time 0. Use the horizontal scroll bar at the bottom if necessary to reposition the display to show time 0 on the left.

Having zoomed in somewhat, it should now be possible to select just the first 10 ns of the reset waveform. Click and hold the left button near the time 0 point of the reset waveform, then move the cursor to the right while holding the left button until you have covered approximately 10 ns. Release the button. If more than the desired amount of time has been selected, or if other waveforms were also selected inadvertently, simply click in empty white space on the grid to unhighlight your selection and try again.

Once the first 10 ns of the reset waveform have been selected, click on the '0' icon in the row of icons at the top to cause a short low pulse at the beginning of the reset waveform. The remainder of the waveform should still be at a high logic level. This low pulse will ensure that the system is properly reset at the start of the simulation.

The changes are now complete, so use *File → Save* to write the updated waveforms to the VWF file.

To perform the simulation, use *Simulation → Run Functional Simulation* from the menubar of the input waveform editor window. A temporary shell window will appear while the software does the various tasks to prepare and execute the simulation. A new output waveform window (which is read-only) should ultimately appear.

Maximize the output waveform display for convenience. Use ctrl-space or ctrl-shift-space to zoom in or out as desired so that you can read the hexadecimal values shown on the waveforms.

Use `ifetch_out` and `mem_read` to identify the start of each instruction and the memory read for that step. Use other assertions of `mem_read` to identify each load instruction. Use assertions of `mem_write` to identify each store instruction. Observe the values on the `data_to_procr` or `data_from_procr` waveforms for each instruction fetch or data read/write. If you try the sample program given above, you should know the correct behavior that is expected, and you should be able to confirm that the desired behavior is achieved. If not, determine why (likely an error in the code inserted in the ROM) and correct the problem to resynthesize and simulate again.

Closely examine the assertion of `rom_active` *and* `ram_active`, *relating their assertion to the particular step during the execution of each instruction, and the value of* `mem_addr_out` *in that step*. Zoom in as necessary to see the full hexadecimal address in each cycle. Make certain you understand what is happening in each cycle and for each instruction as it concerns the desired behavior of the program.

In the simulation output waveform display, scroll horizontally and adjust the zoom in/out as necessary so that you can read all hexadecimal values conveniently.

At the left side of the window, if the name of each signal is not fully visible, left-click and hold on the vertical line to the right of “Name” and drag it further to the right as needed.