

Project: Part #1 for ECE 152

Register File and Finite State Machine

Must be submitted electronically by 10:00AM on Weds, Jan 23

The project for this course is the design and implementation of the Duke152/16, a MIPS-like architecture that has been scaled down a bit to make it feasible for a class project. I have specified the architecture, and you will incrementally build a computer that implements this architecture. The complete specification of the entire architecture will be available soon at: <http://www.ee.duke.edu/~sorin/ece152/project/duke152-arch.pdf>

To become reacquainted with digital design and the digital design tools, we're going to start off by implementing a register file and a finite state machine.

VERY IMPORTANT DESIGN RULES: For all portions of this semester-long project (unless otherwise specified), you may NOT use any of the “mega-functions” provided by Quartus II—you must implement all components from the most basic structural building blocks, using either block diagrams or structural VHDL. You may use FOR/GENERATE loops, but you may not use behavioral constructs like PROCESS blocks, CASE statements, and loops that are not FOR/GENERATE. If you have any questions about these design rules, please post a question on the course's Google group, and the TAs or I will provide clarification. I will not accept “But I thought that was allowed” as an excuse.

Part 1a: Register File

The register file is the collection of registers that the processor uses in its computations. The Duke152/16 has eight 16-bit general-purpose registers, \$r0-\$r7, where \$r0 is *always* equal to zero and \$r6 is *always* the base address of the static data segment (which equals 0x4000). At any given time, we can be reading two registers (identified by `ctrl_readRegA[2:0]` and `ctrl_readRegB[2:0]`). Reading is not clocked. On any rising edge of the clock, we can write one register (identified by `ctrl_writeReg[2:0]`) if `ctrl_writeEnable` is true. Figure 1 is a Quartus screenshot of the register file, and **it shows the signal names that you all MUST use in your designs** (to facilitate testing and grading). In general, all control signals in the Duke152/16 will begin with `ctrl_`, in order to

improve clarity. The signal `ctrl_reset` resets all registers (sets their values to all-zero), which you'll want to be able to do when you boot up your Duke152/16.

To implement the register file, you will use the Quartus II software. You should create one .bdf file that has exactly the same format as the diagram in Figure 1. This high-level .bdf file is likely to refer to other lower-level .bdf files (e.g., `onebitregister.bdf`, etc.). The high-level .bdf is what you will then use later in the semester when you need a register file for your computer.

After implementing your register file, you should test it thoroughly to demonstrate that it works correctly. I have provided one test for you at <http://www.ee.duke.edu/~sorin/ece152/project/regfiletest.vwf>. In addition, I will grade this assignment by running additional tests (not provided), so don't assume you can ignore bugs that don't manifest themselves on only the one test that I have provided.

Part 1b: Finite State Machine

You will design a fairly simple finite state machine (FSM). **For this portion of the project, you MAY use behavioral VHDL (with CASE statements) to specify the FSM. You may also use behavioral VHDL to specify arithmetic computations (e.g., addition), rather than implementing the arithmetic units (e.g., adder) yourself.**

The FSM has the following 1-bit inputs: `clock`, `reset`, `isLoadInstruction`, `isStoreInstruction`, `isAddInstruction`, `isMultiplyInstruction`, `regSource1`, `regSource2`, `regDestination`. It has the following outputs: `PC` (16-bits), `readInstructionMemory` (1-bit), `readRegs` (1-bit),

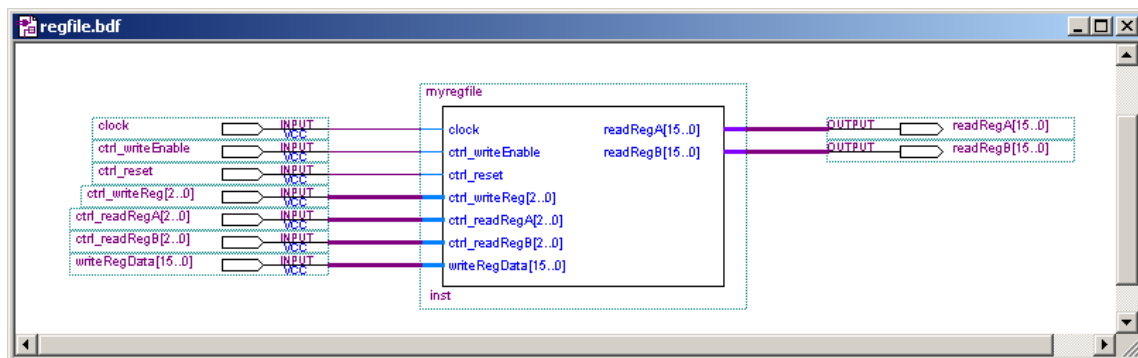


FIGURE 1. Quartus II screenshot of Duke152 register file

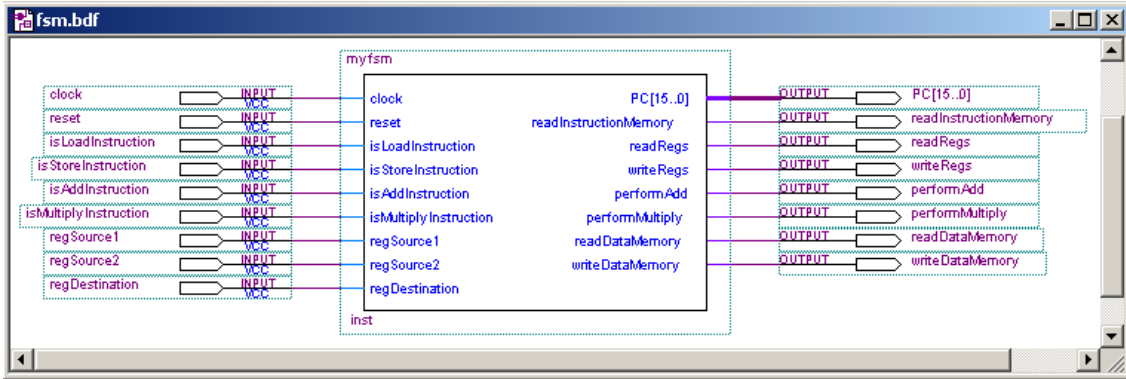


FIGURE 2. Quartus II screenshot of FSM

writeRegs (1-bit), performAdd (1-bit), performMultiply (1-bit), readDataMemory (1-bit), writeDataMemory (1-bit).

When the reset input signal is high, the FSM should go into its initial state and PC should be set to zero. After reset, the FSM will continuously repeat a 5-cycle pattern.

On the first rising edge of the clock (cycle 1), readInstructionMemory must be set high. PC is unchanged. All other outputs should be set low.

On cycle 2 (the second rising edge of the clock), readRegs must be set high. PC should equal PC+1. All other outputs should be set low.

On cycle 3, if isAddInstruction or isLoadInstruction or isStoreInstruction is high, then set performAdd high (else set it low). If isMultiplyInstruction is high, then set performMultiply high (else set it low). PC is unchanged. All other outputs should be set low.

On cycle 4, if isLoadInstruction is high, then set readDataMemory high (else set it low). If isStoreInstruction is high, then set writeDataMemory high (else set it low). PC is unchanged. All other outputs should be set low.

On cycle 5, if isStoreInstruction is high, then set writeRegs low (else set it high). PC is unchanged. All other outputs should be set low.

After cycle 5, the loop repeats. For example, cycle 6 is the same as cycle 1. Cycle 7 is the same as cycle 2. Etc.

You should create one .bdf file that has exactly the same format as the diagram in Figure 2.

This high-level .bdf file may refer to other lower-level .bdf files.

Submitting This Assignment

To submit this assignment, you'll create a packaged and compressed file of your working directory. I'll assume your working directory is called `~myname/ece152/project1/`. Make sure all of your files are in this directory and that your high-level `.bdf` files are named `reg-file.bdf` and `fsm.bdf`. Names of lower-level components are unrestricted. To package and compress a directory on a Unix machine, do the following from the parent directory (`~myname/ece152/`):

```
tar cvf project1.tar project1           // creates project1.tar, a package file
gzip project1.tar                       // compresses project1.tar into project1.tar.gz
```

Now you should have a file called `project1.tar.gz`. Direct your web browser to <https://www.ee.duke.edu/sorin-upload/>. Upload your file called `project1.tar.gz`.

For those of you using Windows, you have two options. Either copy your directory over to your Unix account on ECE (using SSH) and follow the directions above, or download and install Windows software for packaging and compressing. One free software option is called 7-zip (<http://www.7-zip.org/>). The software is somewhat confusing, but it can be made to do what you need (which is create a `.tar.gz` file).