CprE 381: Computer Organization and Assembly-Level Programming

Project Part Extra Report

Team Members: Jake Hafele

Thomas Gaul

Introduction/Goals

We started this project with the goal of running our hardware scheduled pipelined processor on an FPGA. We used the DE2-115 board to run our processor. The minimum goal we had was to be able to run any of the programs we tested out our design within the toolflow on the FPGA and verify their success with the I/O.

Tools Used

For this project, we checked out a DE2-115 FPGA development board from the ETG to configure our processor. This board was useful since it had 8 seven segment decoders, meaning we could display a full 32 bit output at the same time on the board. The board also included a large amount of green LEDs, red LEDs, buttons, and switches to interface with the processor. The button was useful to act as a debounced clock, and the switches were nice to have to MUX between different 32 bit outputs from our processor. The DE2 board uses a power supply and USB cable to power and reconfigure the board respectively.

Since we used the DE2-115 development board, we used Quartus to synthesize our project. By creating a top level module Quartus_synth.v, we were able to use the inputs and outputs of this module to assign the pins using the pin wrapper tool in Quartus.

To generate our Intel Hex files for the project, we used the provided MARS jar with the given and designed assembly instructions we used for each of the three lab projects. We were able to generate memory hex files for programs such as bubble sort, fibonacci, and grendel to verify functionality.

Finally, we were able to use the provided toolflow to generate a trace output and clock cycle count for each simulated program. The trace output was useful since it stated what the ALU output would be for every clock cycle, and the cycle count was useful to determine if our program was taking the right branch conditions throughout the program.

Designing the FPGA wrapper + Bugs

The first thing that we did for this project was make a new branch in git to not potentially ruin our earlier projects. We then made a blank Quartus project, and created a new top level module named *Quartus_synth.v*. We started by importing all of the VHDL files for our processor, alongside a counter, clock divider, button debounce, and seven segment decoder module. The inputs to the *Quartus_synth.v* module would include different switches, buttons and the 50 MHz clock on the DE2 board. The outputs included all 8 of the seven segment decoders, and multiple LEDs to indicate certain control bits or flags for the processor. By using the Pin Planner, under Assignments/Pin Planner, we were able to assign each of these inputs/outputs to the corresponding pins on the DE2 development board.

We decided to use multiple clocks with our board due to being in different states of verification to test different functionality of our processor. We debounced a button, KEY0, to act as a manual clock at our own speed, to verify our ALU output or data memory output. We also instantiated two clock dividers, titled *g_clock_divider_slow* and *g_clock_divider_fast* to create two other clock sources at 5 Hz and 25 MHz respectively. We then used two switches, SW1 and SW0, to multiplex between each of the three signals, so they could be changed for our needs at any given moment.

Another button, KEY1, was used to act as an asynchronous reset signal for our processor and I/O modules. One issue we came across was that we did not realize that the buttons, when left open, would output a 1 signal to our module. This meant that the reset was always driving high for every module inside of our wrapper. After inverting both the reset and clock buttons, we could safely assume both inputs would be driven as 1.

A feature we added to our design was a clock counter and a clock latch on the halt signal. The clock counter was just a basic counter with its reset hooked up to the reset of the rest of the processor and its clock whichever clock is driving the processor. The latch on halt kept the clock counter from incrementing past the end of the program. The latch on the clock affected the clock signal going to the processor saving the final state of the processor. This clock counter and latch allowed us to verify the number of clock cycles in testing.

Another change we made for debugging was a multiplexer to select the data output of the 7-segment display. With it having 8 7-segment displays we could show all 8 hexadecimal values of a 32-bit number. This ability to see different lines inside the processor helped with testing and extended upon more in the verification section of the report.

The most important change in this project was how we updated the *mem.vhd* file. To load the generated hex files from MARS, we needed to separate the instruction memory and data memory modules into two separate components, so that they could both load a different hex file for their memory contents. We started by copying the *mem.vhd* file and created the *dmem.vhd* and *imem.vhd* files in the /TopLevel/ src folder. To initialize the contents of the hex files, we included the following two attribute lines of VHDL to each of our memory files, with both of the respective hex files *imem.hex* and *dmem.hex*. In Quartus, we included these hex files under /Quartus_synth/ and included them in our Quartus file so they could be read when the design was compiled and synthesized. This change also required us to update the mem module in the *MIPS_Processor.vhd* file, so that the new modules were included and referenced.

```
architecture rtl of dmem is

    -- Build a 2-D array type for the RAM
    subtype word_t is std_logic_vector((DATA_WIDTH - 1) downto 0);
    type memory_t is array(2 ** ADDR_WIDTH - 1 downto 0) of word_t;

    -- Declare the RAM signal and specify a default value.  Quartus Prime
    -- will load the provided memory initialization file (.mif).
    signal ram : memory_t;
    attribute ram_init_file : string;
    attribute ram_init_file of ram : signal is "dmem.hex";


begin
```

We had a few issues occur with the memory modules. We began by debugging programs with just the instruction memory, since it should have never been written over. We were able to run programs successfully after finding the previously mentioned issue with the reset signal always being driven as 1. To ensure that no instructions would be written over, we set the input iInstLd to 0 when we instantiated the *MIPS_Processor.vhd* module. When running Grendel, we noticed that there were more than 1024 entries of memory, since the memory was byte addressable. So, we had to add 1 bit of the address width for the instruction memory module, leading to modifying the *MIPS_Types.vhd* file and *MIPS_Processor.vhd* file to update the address generic and index input. With this, the instruction memory was functioning properly. We were having issues with loading contents from the data memory module, and learned that the DE2 board had to be turned on, off, and reconfigured for the initial data memory hex file to be properly reloaded. So, when testing a program with data memory loaded in, it is required to turn power on/off and reconfigure the board first. It is also required to set the correct clock setting before reconfiguring, to avoid metastability issues when flipping the noisy switches.

Files Added/Altered

Changes to files:

- dmem.vhd
    - Made own module from mem.vhd
    - Include dmem hex file for megafunction (Lines 34, 35)
- imem.vhd
    - Made own module from mem.vhd
    - Include imem hex file for megafunction (Lines 34, 35)
- MIPS_types.vhd
    - Created separate address bit widths for imem and dmem
    - IMEM uses 11 bits to house grendel properly
    - DMEM uses same 10 bits, could also be reduced for faster synth time

- MIPS_Processor
    - Updated ADDR_WIDTH constants from the MIPS_types.vhd file
    - Changed imem and dmem indexing to start from bit 0 instead of bit 2
    - Added more signals to be output from the processor module for seven segment decoder verification

Added I/O files:

- Debouncer.bdf
    - Debounce circuit from 281 for the buttons
- Clock_divider_1024.bdf
    - Clock divider used for above Debouncer circuit
- Counter.v
    - Counter for Mod N clock divider
- Clock_divider.v
    - Clock divider for 5 Hz and 25 MHz clock
- Seven_seg_decoder.v
    - Seven segment coder to see 32 bit outputs from processor

Top Level synth file:

- Quartus_synth.v
    - Puts together processor and I/O
    - Inputs/Outputs of this module used for pin mappings

Process to flash program

We always worked with our FPGA on the Coover 2050 Linux computers, using a DE2-115 FPGA development board.

The following steps were followed to reconfigure the FPGA with a new instruction memory and data memory:

1. Output the instruction memory and data memory as an Intel Hex File memory format from MARS
2. Copy the contents of the instruction memory and data memory from MARS to the Quartus hex files in the existing project
    a. Copy generated instruction memory to Quartus_synth/imem.hex
    b. Copy generated data memory to Quartus_synth/dmem.hex
3. In Quartus_synth/dmem.hex, delete all memory entries with 0x00000000
    a. The extra lines need to be deleted so that the first memory contents are not overwritten with 1024 byte addressable entries
    b. Ensure the last line of the hex file, “:00000001FF”, is still included as the last line
4. Open the Quartus_synth project in Quartus Prime
5. Select “Compile Design” to compile the full project with the new memory files

    a. The compile should take around 4 minutes with a DE2 Board on the Coover 2050 Linux computers
6. Power on the DE2 Board and plug in J9, the Blaster, into a USB port of the computer
7. Select “Program Device”, then “Hardware Setup” in the Programmer window
    a. Select the connected USB port to the Blaster to reconfigure the FPGA
8. Select the desired switch configuration for your clock setting BEFORE reconfiguring the FPGA
    a. Data memory does not get reset, so either the fast clock, slow clock, or debounce button clock NEED to be selected before sending the bitstream file
    b. Switching from the button to the fast clock will lead to noise and can affect the program output.
    c. Switch configuration: {SW[1], SW[0]}
        i. 00: Debounced button KEY0
        ii. 01: 5 Hz clock
        iii. 10: 25 MHz clock
9. Select Start in the Programmer window to reconfigure the FPGA with the compiled code
    a. If the FPGA is flashed correctly, all of the LED’s will become dim for around 5 seconds, then the program will be loaded
10. Verify the program by reading multiple 32 bit signals on the seven segment displays that have been propagated up from the MIPS processor
    a. Switch configuration: {SW[4], SW[3], SW[2]}
        i. 000: IF Instruction
        ii. 001: EX ALU output
        iii. 010: EX Register Write Address
        iv. 011: WB Data Memory Output
        v. 100: Clock Counter
        vi. 101: WB Register Write Data
11. Either celebrate because grendel works or cry

Verifying Results

To verify that the FPGA was running the MIPS programs on our processor properly we used the outputs and compared them to the ms.trace file for each program which is generated by the toolflow. Specifically we looked at the ALU output at the execute state, the data memory output at the writeback state, the instruction at the instruction fetch state and the clock counters. Each of these we could select to view on the LCD display. Each of these data options we used to verify different portions of the program.

Instruction at the instruction fetch stage: We used this one to verify the correct program got flashed onto the processor and that the processor was using the program counter correctly to step through each instruction in the program

ALU output at the execute stage: We checked this to check a number of things. First to make sure the basic functions of the program worked correctly such as adding, subtracting, anding, and oring. We also used it when debugging and testing the Data Memory to ensure that we were indexing to the correct location in memory.

Data Memory Output in the writeback state: This location we used to debug issues with the data memory by seeing what was loaded front eh data memory to be saved back into the register file.

Clock Counter: We used the clock counter for rough verification that the program worked correctly. To avoid hitting the clock button and checking instructions or outputs for the 3000 cycles of grendel we ran it on a fast clock and made sure it ran the correct number of cycles. This is not a perfect check but ensures it branched properly every time and most likely is correct.

Additionally we had a handful of LED's mapped to assist in the Debugging

- LEDR0 Clock select 0: used to verify the buttons are set correctly for selecting clock
- LEDR1 Clock select 1: used to verify the buttons are set correctly for selecting clock
- LEDR2 Clock Mux: Shows the output of the clock for which one is running the project
- LEDG0 RegWr: The enable for the writing to registers used to verify information is saved
- LEDG1 DMEM_mem_WE: The The enable for the writing to memory used to verify is storing
- LEDG2 halt: Latches on halt so the we can see when the program finishes running
- LEDG3 overflow: Shows if a given instruction is a overflow to ensure it is properly running

Conclusion

Overall running our processor on the FPGA went pretty smoothly. The vast majority of the debugging came from learning the program and the pains of I/O. The only design change we had to make to our processor was expanding the Imem to handle grendel. We accomplished the goal of running any of our programs on the FPGA. Other things we were planning on looking into was outputting the assembly line on the LCD screen or outputting data via UART to the computer. Unfortunately we did not have the time to accomplish this before the end of the semester but would be within reach with this working processor. This was an interesting project getting to see this processor we worked on all semester run physically on a board.