Gold Variable-Width Processor
Project Part 2
Assigned: March 7, 2016
Due: March 30, 2016 (11:50pm)

Objective:
The goal is for students to design and verify the 64-bit variable data-width Gold processor using a synthesizable RTL style of Verilog and verify through appropriate testbenches. The emphasis is on correct functionality, not timing for this part of the project. You should not be using delays in the design itself, as we will attempt to synthesize the code in later stages of the project. To establish a point of consistency across all projects, simulate at a clock cycle of 4ns (250MHz). For further verification of your processor's functionality, you will also write a test bench to test the processor architecture. Except for the instruction and data memory all of your design should be fully synthesizable to be prepared for Part 4 of the project.

Team Policies:
Everyone should continue to work in their 2-student teams for the project unless already granted permission to work individually on the project this semester.

Verilog Setup:
Use the Verilog environment that you used for Project Part-1 to complete this project phase. Specifically, use the NC-Sim environment that has been recommended in the class. Codes compiling with other tools but not with the NC-Sim environment will be penalized severely in a similar manner to a non-compiling code.

What to do?
1. Download the Gold Variable-Width Processor Instruction Set Architecture (ISA) Manual from the course website.
2. Download the provided tar-file which provides a framework for simulations that uses a makefile to compile and test your design. The provided tar-file provides Instruction/Data memory code and the top-level test-bench design. You are required to strictly follow the port-names as specified in the top-level design; failure of which will be severely penalized in a similar manner to a non-compiling code. You are required to submit your designs in the provided directory structure, as the TAs/mentor will use makefile to compile and test your code. You are free to use your own framework for designing and testing sub-modules of your processor code.
3. Write RTL Verilog code for a variable-width 4-stage pipelined processor that executes all the instructions in the Gold Variable-Width Processor Instruction Set Manual. (Since Load/Store instructions use only immediate address specifiers, memory operations can be performed in the same stage as ALU functions; therefore, a 4-stage pipeline results). The pipeline stages are then –
   a. Instruction Fetch (IF)
   b. Instruction Decode and Register Fetch (ID)
   c. Execution or Memory Access (EX/MEM)
   d. Write Back (WB)

A general high-level diagram is shown in Figure 1. The top-level module has only clock and reset inputs and memory ports for the instruction and data memories. You may need to add control multiplexors or other building blocks in addition to these. All pipeline stage registers are assumed to be clocked on the rising edge of the clock. You may also wish to connect the buses slightly differently. For instance, it is possible to feed certain bits of the instructions directly to ports of the register file without going through the Instruction Decode block. Furthermore, few of the control connections are shown such as data-path control, data memory control, etc. A short description of the elements shown in Figure 1 follows:-

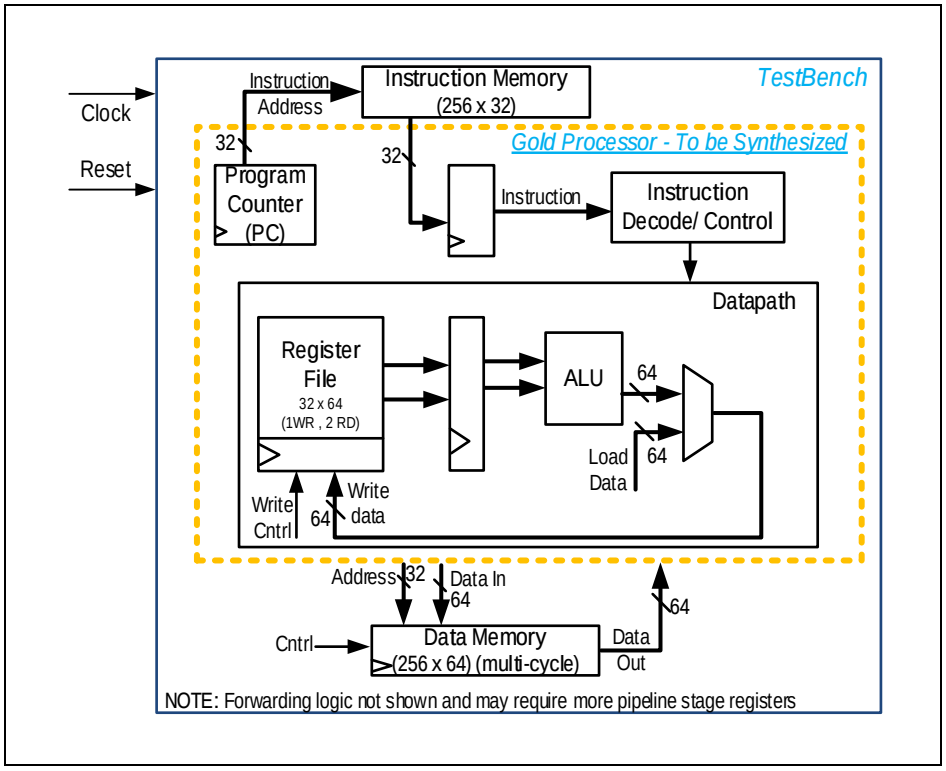

Figure 1: A high-level view of Gold Variable-Width Processor with Instruction/Data memory simulation models

a. The system assumes a SYNCHRONOUS active high RESET. At reset, you should clear all the contents of stage-registers.

b. The Program Counter (PC) unit should implement a 32-bit PC which increments by 4 every clock (since instructions are 4-bytes long) and has a value of 32`h0000_0000 at reset. The PC should be positive-edge triggered with a synchronous reset.

c. An ASYNCHRONOUS instruction memory which is 256 words by 32 bits/word should be sufficient for the testbench programs on the project, and it can be assumed to support READ operations ONLY. This component is part of your simulation environment used for testbench purposes only and is not required to be synthesized. This is provided to you and should not be modified.

d. The Instruction Decode unit should decode the fetched instruction and generate all necessary control signals for the correct execution of the fetched instruction.

e. The Register File is a 3-ported (2-READ port and 1-WRITE port) general purpose register file that implements 32 64-bit registers. The reading from register-file is ASYNCHRONOUS while writing to register file is SYNCHRONOUS. Each port consists

of 5-bit address (address specifier) and a 64-bit data for reading/writing the contents to the specified register. At reset, all register contents must be cleared. Register0 needs to be hard-wired to 64'h0000_0000_0000_0000 and is READ ONLY, i.e. Register0 location cannot be written to. The register file should support a selective write mechanism such that each byte of data could be written independently. Lastly, the write port should contain a write enable signal to control write-operation of the register file. The read ports do not have enable signals, so each read-port will read the register contents based on the associated 5-bit address presented (asynchronously).

f. The Data-path block performs all the operations specified in the Instruction Set Architecture (ISA) manual instructions.

g. A SYNCHRONOUS data memory with a 2 clock-cycle latency that is 256 words by 64-bits/word would be sufficient for the testbench programs on the project. Unlike the Instruction Memory, the Data Memory must support both read/write operations. This component is part of your simulation environment used for testbench purposes only and is not required to be synthesized. The data memory is provided to you and should not be modified.

h. You are required to implement two different branch instructions. The branch instruction requires you to read a register-file value and decide whether the branch is to be executed or not. As one can read the contents of an entry in the register-file in the Instruction Decode (ID) stage, you must compare the contents of the register-file location and decide whether the branch is to be taken in the same stage (ID-stage). In case the branch is taken, you would update PC with the immediate-address value which is the branch-target address. The new instruction would be fetched in the next clock-cycle. In this scenario, you would need to flush one instruction behind the branch instruction, or turn it into a bubble in other words, which was in its Instruction Fetch (IF) stage. We will not be executing any delayed-slot branches in this processor. Flushing an instruction means that the instruction should not update contents of any stage-registers like Instruction/Data Memory and register-files. In the other case, where the branch instruction is not taken, the next instruction which was in the IF stage must be decoded and executed like any other normal instruction.

i. You are allowed to use Synopsys DesignWare components in your design.

4. Verify your design using the provided testbench framework. The good thing about processor designs is that they can be verified by assembly programs. The provided testbench loads your program into your Instruction Memory from file "imem.fill" and initial data needed into the Data Memory from file "dmem.fill". You may set the cycle time of your design by modifying "CYCLE_TIME" defined at the top of your testbench. A sample test assembly program is provided. You should write more elaborate test benches to exercise every instruction in the instruction set at least once, as well as every possible setting of PPP and WW bits (refer to instruction manual). Make sure when testing different WW settings you use appropriate instructions and data settings to verify that operand width selection works properly. Your code should cause unique values to be written into registers to verify that each instruction execution had the desired effect. Do not modify the provided testbench other than modifying the CYCLE_TIME. To test various instructions, simply modify imem.fill and dmem.fill files.

5. Compile and test your processor. You may wish to perform pre-tests on subcomponent modules before integrating all the modules.

6. Synthesize your processor design to simply prove that your Verilog is synthesizable with no errors. The goal of this task is to prove that you have synthesizable code. So while the clock target is not critical, use a target clock of 4ns, but if you must relax the target to compile in synthesis, that's allowable. Be sure to submit the final script showing the clock period with your design. Submit your synthesis script (**gold_processor.tcl**), resulting netlist (**gold_processor.syn.v**) along with the output of the `check_design` command (**gold_processor.log**). Check the output of the `check_design` command to make sure there are NO LATCHES generated. If so, modify your RTL to remove any latches in the design.

What to Submit?

Submit the complete Verilog code in a directory structure specified for your processor design including the testbench used for verification. Please include adequate comments not only in your design but also in your testbench to justify that your testbench adequately verifies your design. In addition, include the program and data files that are loaded into the Instruction Memory and Data Memory for test purpose. Please comment your binary assembly code with the corresponding assembly mnemonics. You will also need to turn in results for an instructor's test bench that will be provided shortly before submission.

How to submit?

We will use electronic submission. For electronic submission, **follow the instructions given below strictly.** Use the directory provided with the assignment to organize your submission.
"tar" all the required files for the project using the "**make submit**" command and use the upload link in DEN for electronic submission. Use "**make submit**" command to create the tarfile, this allows grading team to use automated tools to grade assignment.

(1) Modify **AUTHORS** file by adding name of the team and team members. Write name of the team in the first line of the file, followed with team-member names on subsequent lines.
(2) Modify the **makefile** to declare name of your team, this will be used to create the submission tar-file.
(3) All of your verilog design files should be in **design/** directory. All test-bench files should be in **tb/** directory. All the synthesis script files should be kept in **script/** directory. The generated netlists (*.syn.v) and SDF files (*.sdf) should be kept in **netlist/** and **sdf/** directory respectively. All generated outputs (.out), logs, reports, pdfs and other documents should be kept in **reports/** directory.
(4) Modify README file to describe status of the project, also include any additional instructions (if any) required for simulation/testing your assignment.

If you need to re-upload your submission before the deadline, please contact a TA.
One submission per group. If the group accidentally submits twice, you must coordinate with the TA's.