LSU EE 3755 **Homework 5** Solution **Due: 20 November 2012**

For this assignment read Chapter 4 of Patterson & Hennessy, Computer Organization 4th Edition, up to and including Section 4.4. These sections describe a MIPS implementation which is most similar to our functional simulation model,
`http://www.ece.lsu.edu/ee3755/2012f/mips_fs.v.html`*, however for this assignment compare the Section 4.1-4.4 implementation to our Multi-cycle Implementation 1,*
`http://www.ece.lsu.edu/ee3755/2012f/mips_hc.v.html`*.*

Problem 1: Consider how the instructions `beq` and `bne` are implemented in the two designs.

(*a*) How does the implementation described in Section 4.3 detect whether the `beq` and `bne` instructions are taken?

The ALU in the Section 4.3 implementation has the usual result output but also a 1-bit output, `Zero`, which indicates whether the result is equal to zero. To execute `beq` and `bne` instructions the ALU is set to perform a subtract operation, control logic examines the `Zero` output and considers the branch to be taken if `Zero` is `1` for `beq` or `0` for `bne`.

(*b*) How is that different than the method used by the Multicycle MIPS Implementation 1 presented in class?

The implementation described in class also uses the ALU, however our ALU lacks a `Zero` output. Instead, there is a set-equal ALU operation which sets the output to `1` if the two inputs are equal.

Problem 2: The Section 4.4 implementation illustrated in Figure 4.15 has a `RegWrite` signal.

(*a*) What does that signal do?

It commands the register file to write the data at input `Write data` in to the register number at input `Write register`.

(*b*) The multi-cycle implementation covered in class lacks such a signal. Explain how it gets by without it.

The class implementation does a write every cycle so it doesn't need a `Write data` signal. To avoid disturbing things when a register write is not needed (for example, when executing a branch or store instruction), the register number is set to zero. The Verilog code for our implementation does not write the register file if register zero is given as the destination register. (That's another way of saying that our register file is designed so that writes to register number zero have no effect.)

Problem 3: The implementation described in Section 4.4 is a *single-cycle* implementation and is most similar to our functional simulation model. The multi-cycle implementation that we are now (16 November 2012) covering has cost and performance advantages.

(Don't forget: The pipelined implementation, which will be covered in EE 4720, has even greater performance benefits. The multi-cycle implementation is both a pedagogical bridge to the pipelined implementation, but it also is a suitable implementation technique for instruction sets more complex [and not in a good way] than MIPS. Many mid 20th century computers used multi-cycle implementations, by the 80s pipelined implementations replaced them.)

(*a*) Consider the cost of the implementation illustrated in Figure 4.17 as compared to our multi-cycle implementation. Indicate the major units of hardware illustrated in 4.17 that are not needed in the multi-cycle implementation. (Those 4.17 units are not needed because a single unit in the multi-cycle implementation is used for different purposes over different clock cycles.)

The implementation illustrated in Figure 4.17 has two major units that the multi-cycle implementation lacks: an adder and a memory port.

The not-needed adder appears in the upper-right of the diagram, and is labeled as an ALU (even though it only performs addition). It is used to compute branch targets. The multi-cycle implementation uses the same ALU for both branch target computation and to perform instructions' arithmetic and logical operations.

The not-needed memory port appears in the lower-right side of the diagram, labeled `Data memory`. This memory port is used for load and store instructions (a second data port (on the lower left-hand side) is used for instructions). The class multi-cycle implementation uses the same data port both for fetching instructions and for loads and stores.

Note that the terms *Data memory* and *Instruction memory* used in the text can be considered misleading because in MIPS the same address space (set of addresses, or oversimplifying, set of memory locations) is used for both instructions and data. In real CPU implementations there are usually separate *memory ports* for instructions and data. Both ports can reach the same set of memory locations, those locations are reached by passing through multiple *cache* layers and crossing one or more busses and maybe networks. Memory ports are expensive however the cost is justified in real CPUs because one can fetch an instruction and perform a load or store operation simultaneously.