Labs Using TinyMIPS

Each of the following projects can be implemented as a modification to the TinyMIPS basic machine. Before attempting any of these projects, you will need to be familiar with the TinyMIPS machine model (see the Models page on the CD), and install the GateSim software under which it runs (see the Software page on the CD).

Barrel Shifter

Implement a full 0–31-bit shifter in either direction, so that it executes in one cycle. Adjust the instruction decode so that it knows about your instructions (`sll`, `sllv`, `srl`, `srlv`, `sra` and `srav`).

Look-ahead Carry

Design and implement a look-ahead carry part that you can use to reduce the present 64-gate-delay carry propagation to something closer to seven gate delays, so that add/subtract/compare operations take about the same time as logic operations. Adjust the control logic as necessary to achieve this timing.

Pipeline

Design and implement a pipeline execution unit, so that a new instruction starts every clock cycle.

State Machine Sequencer

Redesign the current instruction sequencer to use a finite-state machine (FSM), so that instructions that need to take longer do not slow down those that can go faster. Verify that all programs still work correctly.

Hardware Multiply

The current TinyMIPS circuit implementation has no multiply hardware. Your enhancement would implement a 32-bit signed multiply instruction (`mult`/`multu`) that starts when the instruction is issued and then runs asynchronously until it finishes, leaving the product in the Hi and Lo registers.

Hardware Divide

The current TinyMIPS circuit implementation has no divide hardware. Your enhancement would implement a 32-bit signed divide instruction (`div`/`divu`) that starts when the instruction is issued and then runs asynchronously until it finishes, leaving the quotient and remainder in the Hi and Lo registers.

Floating-Point Add

Your task is to implement FP addition and subtraction (`add.s` and `sub.s`) and the two conversions (`cvt.s.w` and `cvt.w.s`, which use much of the same logic).

Floating-Point Multiply

Your task is to implement a FP multiply instruction (`mul.s`).

Data Cache

Build an associative memory cache that retains recently accessed RAM data with its addresses and eliminates redundant reads from memory. Be sure to write data stores through the cache to main memory.

Instruction Cache

Build a cache memory that reads ahead the next few words of ROM memory, so that the next instruction is probably already in the cache when needed. Make sure you keep several words already executed, in case of small loops. For extra credit, decode unconditional jumps to continue fetching from the jumped address, essentially removing them from the instruction stream.

Byte- and Half-word Loads and Stores

The current TinyMIPS only loads and stores aligned full data words. Partial words and unaligned words require some extra logic. Implement it.

Parallel Processors

Implement a dual-CPU shared memory version of TinyMIPS. Note that you will need to resolve memory contention and provide for the two CPUs to start at different memory addresses.