Project Proposal and Plan Ahead for ECE552, by Arman Ramiz

Homeworks Ahead:	HW Deadlines	Stages	Deadlines
HW2	23-Feb	Project plan	23-Feb
HW3	9-Mar	Design Review	4-Mar
HW4	6-Apr	Demo 1	23-Mar
HW5	15-Apr	Demo 2	13-Apr
HW6	29-Apr	Cache FSM turnin	15-Apr
		Cache Demo	4-May

Schematics	Complete and verify design by 4th of march Will use software to draw.
Unpipelined Design of WISC-SP13	Complete basic implementation by 10 March Advanced implementation, finalizing modules by 18 March. Verification and tests from script by 20 March
Pipelined Design	Complete basic implementation by 30 March Advanced implementation, hazard check by 7 Apr Complete instruction_timeline.pdf by 10 Apr Final checks before 13 Apr
Insertion of Two-way Cache	Will start alongside pipelined design. Expect completion by 15 Apr Draw detailed FSM
Cache Demo	Everything ready and set before 31 Apr Final checks until 4 May

Designing a microprocessor: WISC-SP13

Project Plan: -Create a feasible work schedule for this project
-Identify key milestones
-Define project design at a high level
- Detailed test plan, include descriptions of components, modules and tests.

Project Transition:

1. Build a single cycle non-pipelined processor with idealized memory
2. Pipeline the processor into distinct stages
3. Transition to a banked memory module. Can't respond to requests in a single cycle.
4. Implement cache that will improve memory performance.

INITIAL STEPS:

1. Download project file, examine the modules. Grasp general concept.
2. Learn to simulate design. Read verification and simulation page…
3. Sign up for design review.

Stage 1: Design Review:

Hand draw, or drawn using a graphic program (Openoffice or possibly KiCad), **schematic** of unpipelined WISC-SP13 implementation, label each module, bus and signal. Hierarchical design. One-to-one mapping of modules.
Check the textbook pipeline diagram, adapt it to WISC-SP13 ISA specifications.

Stage 2: Demo#1- Unpipelined Design:

Date: Expect to finish this before 23. March. Will start right after verifying design and schematics.

Design single-cycle, non-pipelined WISC-SP13 processor. Use single-cycle perfect memory. Use two separate memories, one for fetch instructions (for instruction memory) and one to Read- Write data (memory for data).
To meet the need for two separate memories (fetch and data), instantiate the memory module twice. Instruction Memory module and Data memory. Use same module definition for easier instantiation.

Modularize the design! Putting each of the 5 MIPS stages into separate sub-high-level module. Proc.v
Run **wsrun.p1** script to check that the processor works as desired. May create custom tests as necessary. Strive for “SUCCESS” messages. Save to following log files.

1. Simple tests: simple.summary.log
2. Complex tests: complex.summary.log
3. Random tests for demo1
    1. rand_simple: rand_simple.summary.log
    2. rand_complex: rand_complex.summary.log
    3. rand_ctrl: rand_ctrl.summary.log
    4. rand_mem: rand_mem.summary.log

Stage 3: Demo#2- Pipelined Design:
Date: Expect to start this by 30th, march. In accordance with my midterm schedule(ece340 midterm…)

Proceeded from the unpipelined design. Account for pipeline hazards like stalling. Use single-cycle perfect memory model. Write custom tests.

Tests:

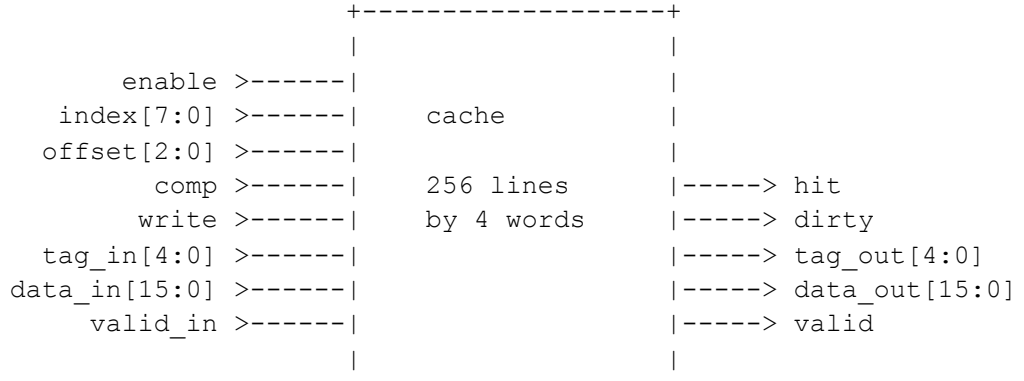
1. Simple tests: simple.summary.log
2. Complex tests for demo1: complex_demo1.summary.log
3. Random tests for demo1
    1. rand_simple: rand_simple.summary.log
    2. rand_complex: rand_complex.summary.log
    3. rand_ctrl: rand_ctrl.summary.log
    4. rand_mem: rand_mem.summary.log
4. Complex tests for demo2: complex_demo2.summary.log
5. Results for your own tests: mytests.summary.log

Stage 4: Cache Demo- Two-way Set-associative Cache:

Overview: Final processor will use instruction and data caches. Design and verify a direct-mapped cache. Will follow up a two-way set-associative cache afterwards. Inside *cache_*/* only edit the *mem_system.v* file. For two-way set-associative, finish two standalone cache designs (*cache_direct and cache_assoc*) and pass necessary tracing tests.

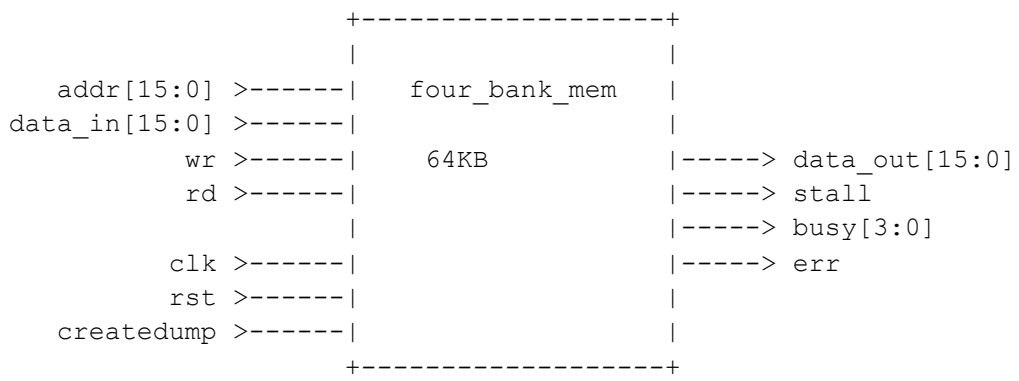
2.Cache Interface and Organization:

The external interface to the base cache module: Signals described on course website.



Cache module will contain 256 lines. V, valid bit; D, dirty bit; Tag, 5-bit tag and four 16-bit words: Word 0-3

Four-Banked memory module: Project memory system. Allows for memory storage in multiple banks. Byte-aligned, word addressable 16-bit wide 64K-byte memory.
Four -cycle, four-banked memory:
four_bank_mem.v : top level *final_memory.v* : single banks



*Signal descriptions found on course website

3.State Machine:
Mem_system module interface will define the state machine for the processor.

1. Determine how cache is arranged and functions
2. Draw the state machine for cache controller
3. Implement Moore machine (Recommended). More likely to prefer Meally...
4. Expect a large state machine…

- The inputs for FSM are:
  - The inputs to the mem_system module;
  - The outputs of cache c0 and main memory mem.
- The outputs for FSM:
  - The inputs to cache c0 and main memory mem;
  - The outputs of the mem_system module.

4. Direct-mapped Cache:

Implement a memory system of direct-mapped cache over four-banked main memory.
Inside *cache_direct/* directory.
When enable is high, control signals comp and write will push us to a case:

Compare Read(comp = 1, write = 0)
Compare Write (comp = 1, write =0)
Access Read (comp = 0, write =0)
Access Write(comp = 0, write =1)

Tests:

Perfbench testing: uses address trace files to describe a sequence of read and writes.
Randbench testing: Random bench; will be formed of random memory request. Requests from restricted addresses. Get the SUCCESS message.
Synthesis: Run synthesis on direct mapped cache, strive for no errors. Save/ turn in reports generated by synthesis. Synthesis results will be in: *cache_direct/synthesis* directory.

5.Two-way Set-associative Cache:

Implement this after fully completing direct-mapped cache.

Compare Read(comp = 1, write =0)
Compare Write (comp = 1, write =1)
Access Read(comp = 0, write =0)
Access Write(comp =0, write =1)

Tests:

Perfbench testing: uses address trace files to describe a sequence of read and writes. Add different address traces, differ from the direct mapped cache. Reflect differences.
Randbench testing: Random bench; will be formed of random memory request. Requests from restricted addresses. Get the SUCCESS message.
Synthesis: Synthesize the set-associative cache. Turn in reports generated by synthesis.

6. Instantiate Cache Modules.