ECE/CS 552: INTRODUCTION TO COMPUTER ARCHITECTURE

Project Description – Stage 2

First Stage Project Demo: **April 17th (30% of grade)**
Final Stage Project Demo: **May 6th & 8th (70% of grade)**

The project should be completed in a group of two or three students.

1) Review of the First Stage Project

Before you start the second stage of the project, you should have a working design pipelined design that implements the following instructions.

Table 1: Table of opcodes

Function	Opcode
ADD	0000
ADDZ	0001
SUB	0010
AND	0011
NOR	0100
SLL	0101
SRL	0110
SRA	0111
LW	1000
SW	1001
LHB	1010
LLB	1011
B	1100
JAL	1101
JR	1110
HLT	1111

To see the detailed information of the ISA, please refer to the first stage project description. You cannot proceed if you do not have a working 5-stage pipe design for these 16 instructions.

2) Memory System

In this stage of the project we add both an instruction cache (Icache) and a data cache (Dcache) that remain single cycle access, but share a single main memory which has a latency of 4 cycles for read and write. This shared main memory will be a 2nd level store, that is accessed upon misses from the Icache or Dcache. Verilog models for the shared memory (unified_mem.v) and for the basic caches (cache.v) will be provided. Both caches are direct-mapped, and have a block size of 4 words (8-bytes) and a data array size of 256 words (organized as 64 4-word lines). Therefore, The address[1:0] is the block offset, address[7:2] is the index and address[15:8] are tag bits.

The cache model provided (to be used for both the Icache and Dcache) has a hit output, and a dirty output. The cache memory includes a valid bit for each line that is cleared on reset (because no entries in the cache are valid at first). The hit output is set on a cache read or write if the tag bits match the address, and the valid bit is set. The dirty output is used for implementing write back. If the cache line has been modified (dirty bit in cache line set) and the cache line is valid the dirty output will be high. The dirty output can be ignored when implementing the Icache control because we should never be writing to the Icache (we do not support self modifiying code).

Figure 1 shows a possible memory structure for use with the WISC_S14. Depending on how you choose to partition your design this may not work well for you, and you may have to partition another way.

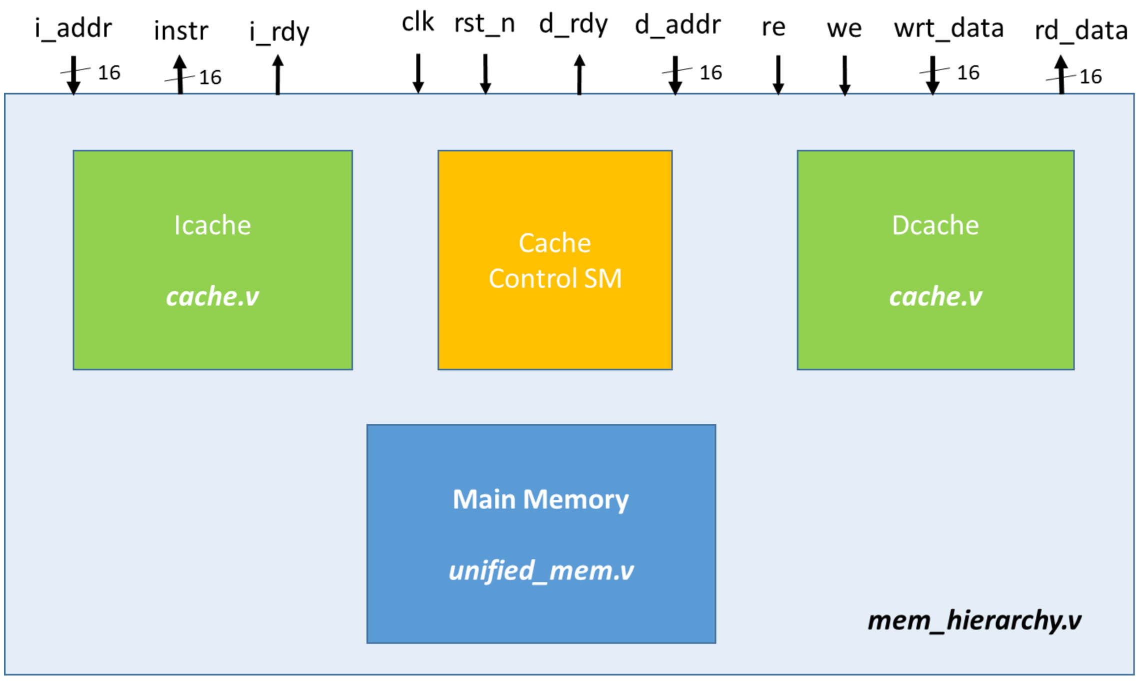

Figure 1: System Memory

From figure 1 we can see that I-cache and D-cache share one main memory. There will be times in which both the Icache and Dcache will be experiencing a miss simultaneously. They will both have to access the main memory fulfill their access. In such instances the Dcache miss should be serviced first. Since the main memory only has one read port only one miss can be serviced at a time, so there is only need for a single cache controller. This cache controller, however, is not a simple thing. There is also a lot more going on in Figure 1 than shown. There are, of course, many interconnections forming the address to the unified memory, the write data to the Dcache, the control signals to the Dcache, … Some of these interconnections are intentionally left vague for the student to figure out.

You can assume that the I-cache is never written by the processor. Writes to the I-cache only happen on fetch misses. In this case your controller will fill a cache line in the Icache by reading the data from the main memory. The address space is shared, however, be careful to ensure the address space you use for data is much higher in the address space than that used for instructions. **Don't inadvertently make your life miserable by writing self modifying code**.

Having to service writes makes controlling the Dcache considerably more complex than controlling the Icache. You will be implementing **a write back** policy for the Dcache. When there is a write instruction (SW) in your pipeline, your cache needs to first check if the data is in the cache. If it is a hit, the new data word will be placed in the proper position in the 4-word cache line, and the cache line will be written. This will all happen in a single cycle because the cache model read on clock high (we then have the hit signal in first half of clock), and writes on clock low. This is actually a bit unrealistic of a memory model, however, our alternative was to have you implement a write buffer, and this project is complex enough already. Remember when a cache write happens from the CPU you must set he dirty bit to indicate that line is now different than is corresponding location in main memory. If it is a write miss, the cache controller will have to check if the current line in the cache is dirty. If it is dirty, that line will have to be written back to main memory. Then a read access will be performed to main memory to get the line that the word to be written resides in. The word to be written will then be placed in the proper position in the 4-word line read from the main memory. This new line will be written to the Dcache and the dirty bit will be set. At the end of these operations the d_rdy line is asserted.

Verilog modules are provided (cache.v for both Icache and Dcache, and unified_mem.v for the main memory). The interface of these memories is described below:

a) Cache Interface

Signal:	Dir:	Description:
clk	in	clock
rst_n	in	Active low reset
addr[13:0]	in	Address. Lowest 2-bits not used (4-words/line)
wr_data[63:0]	in	Data to be written to cache line
wdirty	in	Hold high while writing to mark a line as dirty
we	in	Write enable to write a line to the cache
re	in	Read enable
rd_data[63:0]	out	Cache line read out
tag_out[7:0]	out	Tag bits for the cache line. Need this when evicting line from cache
hit	out	Hit means the tag bits matched and the line was valid
dirty	out	Cache line has been modified. Need to write back.

b) Cache Timing

The caches are single cycle access. Address and ***re/we*** should be presented and valid in the first half of the clock. Data is read on clock high (so Hit and dirty bits are available in first half of cycle) and written on clock low. Since the read and writes happen in opposite phases of the clock it is possible to assert ***re*** & ***we*** in the same clock cycle, however, address cannot change (have to read and write same cache line). Basically a read modify write operation.

c) Unified Memory Interface

Signal:	Dir:	Description:
clk	in	Clock
rst_n	in	Active low reset
addr[13:0]	in	Address. Lowest 2-bits not used (4-word access)
re	in	Read enable
we	in	Write enable
wdata[63:0]	in	Data to write to 4-word line
rd_data[63:0]	out	64-bit line read
rdy	out	Indicates operation (read/ write) is complete this cycle

d) Unified Memory Timing:

The main memory has a timing of 4-cycles for a read or write. The read and write both occur on the clock high phase of the 4th clock cycle. ***Rdy*** is also asserted on the clock high phase of the 4th cycle. Both ***re*** & ***we*** and ***addr*** should be asserted prior to the rising edge of the first clock cycle of access. See the timing diagram below for clarification.

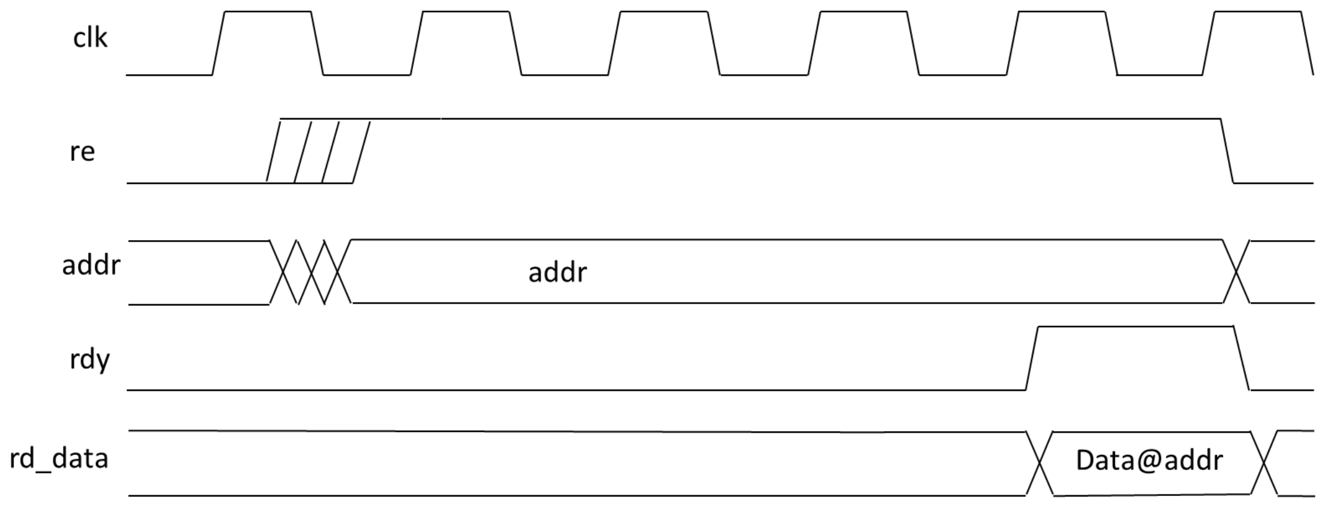

Figure 2: Timing Diagram of Main Memory

A write is similar to *Figure 2* above in that the write occurs during the clock high of the 4th clock cycle.

3) Project Grading

This phase of the project will be worth 70% of the project score, and will be graded largely based on quantitative metrics.

60% Quantitative
10% Code readability & commenting, and test suit comprehensiveness.

a) Quantitative Grade

The quantitative grade is based on the following metrics:

- % of tests passed from Eric & Kaushik's functional test suite
- Geometric mean of your IPC for our performance test suite
- Relative area of your synthesized netlist

$$QuantScore = \frac{\#FuncTestsPassed}{\#FuncTests} * 50 + \frac{IPC\_GeometricMean}{IPC\_refDesign} * 25 + \sqrt{\frac{SynthAreaRefDesign}{SynthArea}} * 25$$

Your design must meet the minimum synthesis frequency target of 800MHz. If your design does not synthesize it area will be taken as 120% of the largest class project that does synthesize.

4) Project Demonstration

Each group will be asked to demonstrate their project on **Tuesday or Thursday May 6th or 8th.** All team members should be present at the demo.

1) You must provide a tarball of all your verilog files like you did for Stage I
2) A code review will be performed. This review will look at code readability and commenting as well as at test suite comprehensiveness.