CS552 Spring 2008 Design Contest

Prof. Karu Sankaralingam

Problem statement

Design a load-store queue, which provides the following functionality:

- Detects ordering violations between load and store instructions in an out-of-order processor
- Forwards values between load and store instructions when issued out of order

Logistics

- This contest contains 3 parts.
- Each part containts the same weight.
- I believe all 3 parts can be completed in the given time, but we'll see.
- Implement your verilog for each part in a separate directory: part1, part2, and part3
- At 12:13pm, handin 3 separate tar-balls of each directory called: dc-part1.tgz, dc-part2.tgz, and dc-part3.tgz. Assignment name for handin is dc.
- When you finish a problem, run the regress.sh script, make sure there are no failures, and raise your hand.

Part 1

Description

Consider the abstract pipeline shown below:

```
Fetch ......   Execute/AddressComputer+MemoryAccess..... Writeback
```

Every load and store instruction goes through three phases as it passes through this pipeline. Every load and store is fetched in program order, can execute out-of-order and is again written back in program order. This is different from the type of processor organization that you have been designing where instructions execute in program order. The job of the load-store queue is to ensure that when loads and stores are executed out-of-order bad things don't happen due to ordering violations. Such a structure is required because the data addresses accessed by the load and store instructions are not available until the execute stage.

Interface

The high-level module interface is below:

```
module lsq(/*AUTOARG*/
   // Outputs
   exec_dataOut, exec_LSQHit, exec_PCOut, exec_addrOut,
   exec_validOut, exec_stall, flush_valid, flush_mask,
   // Inputs
   fetch_PC, fetch_itype, fetch_valid, exec_PC, exec_addr,
   exec_value, exec_valid, wb_PC, wb_valid, clk, rst
   );

   output [15:0] exec_dataOut;
   output        exec_LSQHit;
   output [15:0] exec_PCOut;
   output [15:0] exec_addrOut;
   output        exec_validOut;

   output        exec_stall;
   output        flush_valid;
   output [31:0] flush_mask;

   input [15:0] fetch_PC;
   input        fetch_itype;
   input        fetch_valid;

   input [15:0] exec_PC;
   input [15:0] exec_addr;
   input [15:0] exec_value;
   input        exec_valid;

   input [15:0] wb_PC;
   input        wb_valid;

   input clk;
   input rst;


endmodule // lsq
```

Functionality

You must design a load-store queue which contains 32 entries. Each entry records the information for one instruction in the program and keeps track of its progress through the pipeline. Each entry contains the following fields. See Figure 1.

- Status information:
    1. valid - 1 bit
    2. type - 1 bit, (ld = 0, st = 1)
    3. fetch - 1 bit, load or store has been fetched

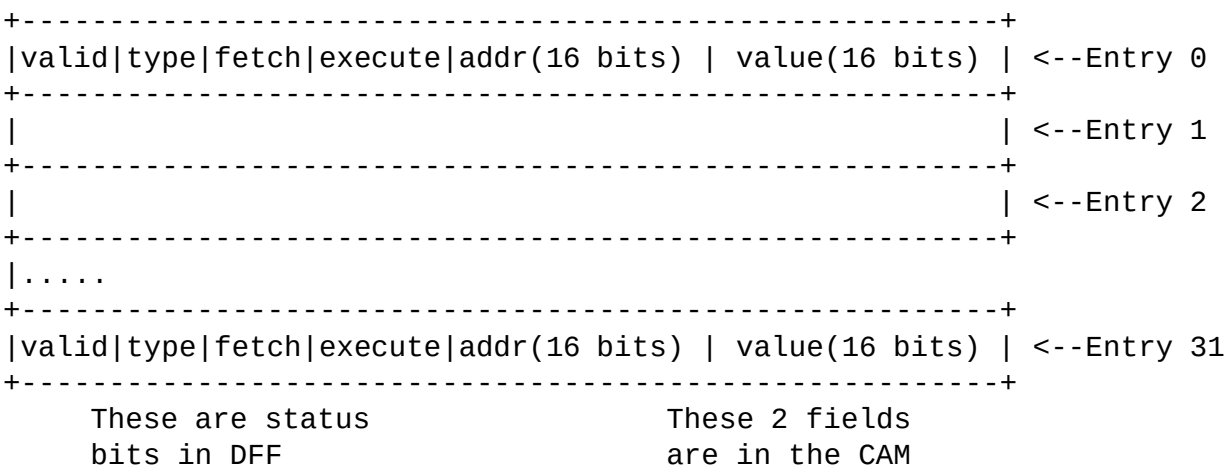

Figure 1: LSQ high level organization

4. execute - 1 bit, load or store has been executed

- Actual data:

1. addr - 16 bits, address for a load or store instruction
2. value - 16 bits, value stored for a store instruction

You must create flip-flops using the provided dff.v to maintain the status bits. You will use the provided basic building block of the CAM to store the address and value information. See the CAM description below.

CAM

A CAM provides the capability to store values like a RAM. It can also be queried with a value and search its entries to find if any entry matches the address provided. The cam module's interface is shown below:

```
module cam (/*AUTOARG*/
   // Outputs
   data_out_addr_match, data_out_addr, data_out_value,
   // Inputs
   data_in_addr, data_in_value, index, cmd_add, cmd_remove,
   cmd_match, cmd_read, createdump, clk, rst
   );
   output [31:0] data_out_addr_match; // which of 32 address match
   output [15:0] data_out_addr;
   output [15:0] data_out_value;
   input [15:0]  data_in_addr;        // input data address
   input [15:0]  data_in_value;        // input data value
   input [4:0]   index;                // index to write to
```

```
Program order          Execution order          Write-back order
(fetch order)

PC=0: cycle 1: St a     cycle 10: st a          cycle 14: st a
PC=1: cycle 2: St b     cycle 11: st b          cycle 15: st b
PC=2: cycle 4: St c     cycle 12: ld a          cycle 26: st c
PC=3: cycle 6: ld a     cycle 20: ld c          cycle 27: ld a
PC=4: cycle 8: ld c     cycle 21: st c          cycle 28: ld c
PC=5: cycle 8: ld a     cycle 22: ld a
```

Figure 2: Example sequence. This is pseudo-code. The `a, b, c` indicate the generated address for the load and store instructions. Therefore `ld a` and `st a` access the same address.

```
   input         cmd_add;              // write
   input         cmd_remove;           // remove
   input         cmd_match;            // match
   input         cmd_read;             // read cam
   input         createdump;
   input         clk;
   input         rst;

endmodule
```

It can perform the following functions:

- cmd_add - add an (address, value) pair to the cam (you may pass in junk values for loads. Note the CAM itself doesn't distinguish between loads and stores, you must maintain type information in separate status bits). The pair is added at the location pointed to by index.
- cmd_remove - remove the (address, value) pair at location pointed to by index.
- cmd_match - find all entries that match the provided address (value is ignored).
- cmd_read - read the (address, value) pair at location pointed to by index.

For the first part of this problem, we will assume PC's are restricted to values between 0 and 31 and all programs are no longer than 31 instructions and use up PCs 0 through 31 only. For this problem we will assume programs have only load and store instructions! We will assume 8-bit instructions, and hence you can simply use the lower order bits to index the storage structures inside your LSQ. For the second part we will make this interesting by asking you to design for 128 instructions, but instantiating four copies of the CAM. Desiging for any number of instructions and building your storage structures as a circular buffer, is more interesting and is left as exercise at home :-)

The functionality you must provide is as follows. Since instructions can execute out-of order, the following cases arise.

1. **Fetch:** When every instruction (load or store) is fetched, indicated by fetch_PC and by fetch_valid being set, allocate an entry in the corresponding LSQ entry to valid.

2. **Execute (Load):** For every load, when it "executes" i.e. exec_valid is set, look for all prior stores earlier in program order that have executed and see if any of the address of those stores match the address of this load. Use the cmd_match functionality of the CAM. There may be multiple matches, determine **the last** store that matches the address of this load and the output of the LSQ for that cycle must be the value of the matching store, and set exec_LSQHit=1. Use the cmd_read command. If none of the prior addresses match, exec_LSQHit = 0.

   Consider the example sequence in Figure 2. In cycle 12, there is a load-hit and the dataOut must be the value read from PC=0. Cycle 22 however, is NOT a load hit because, in cycle 14, the store has been written back and hence deallocated from the LSQ. See write back bullet below. Cycle 20 is NOT a load hit because, the store has not yet reached the execute stage.

3. **Execute (Store):** For every store that "executes", find the store's address that matches the address of a load that has already executed. Use the cmd_match command. This can occur, because a load that is later in the program can execute before a store. In this case, find the first load that matches the address of this store and generate a flush mask which has 1 for all instructions that have been fetched after (and including) the load. Also set, exec_LSQHit which is what the testbench uses for checking.

   Considering again the example in Figure 2. In cycle 21, `st c` executes, which is ahead of `ld c` in program order, and hence will trigger a flush for instructions with PC $>=$ 3. The flush_mask must be set appropriately.

4. **Writeback:** When a load or store reaches, writeback de-allocate it from the LSQ and mark its entry as invalid.

Remember, in a single cycle all three events (fetch, execute, and writeback) can occur. But never for the same instruction in the same cycle. This behavior is summarized in the table below.

In addition, looking ahead to the interface for Part 3, the interface of the LSQ is designed such that, its the outputs the address, and PC that were provided as inputs to it.

fetch_valid	Load/Store	Set valid bit in LSQ, set type, and set fetch=1
exec_valid	Load	Search LSQ, if hit, exec_dataOut = data from last matching store and exec_LSQHit=1. Set execute=1.
	Load	Search LSQ, if miss, exec_LSQHit = 0, dataOut= dont care. Set execute=1.
exec_valid	Store	Find if any later store matches, and if so flush that load and all instructions that succeed it, by setting flush_mask
wb_valid	Load/Store	Set valid bit to 0 for that instruction

Table 1: LSQ behavior. In addition whenever, exec_valid is set, the LSQ must set exec_validOut, exec_PCOut, exec_addrOut which are simply copies of exec_valid, exec_PC, and exec_addr.

Other notes

- For part 1 exec_stall is always 0.
- You are provided with the following modules:
  - cam.v - A content addressable memory or CAM, which can be associatively searched to find a value. See details below.

  - encoder32x5.v - 32 to 5 encoder
  - decoder5x32.v - 5 to 32 decoder

- This part is purely combinational logic in terms of what you need to design

Files

Files needed for this design are located in:
/u/k/a/karu/courses/cs552/spring2008/handouts/dc2008/verilog/release

Testbench

You are provided a random testbench lsq_rand_bench.v which you can use for testing. This testbench will be used for final pass/fail. To generate different random sequences, use the -seed flag to pass in different seed. For example:

```
wsrun.pl -seed 73 lsq_rand_bench *.v
wsrun.pl -seed 81 lsq_rand_bench *.v
wsrun.pl -seed 90 lsq_rand_bench *.v
```

This testbench checks every load and store when it is "executed" if it gets the right value (for stores forwarded from the LSQ as exec_dataOut) and if the LSQ correctly determines LSQ hits. It prints cycle-by-cycle when different events occur, and a print a line called "TRACE:" when an instruction executes.

You may also develop your own targeted testbenches for specific pieces.

A regression script (regress.sh) is also provided which will run your desing with hundred random seeds.

The testbench generates random fetch, execute, and writeback events with random addresses and values. You can control the number of unique addresses it uses by changing the "define N_UNIQ_ADDRESSES" in common.v (or common_128.v for next part). The fewer unique addresses, more load-store conflicts.

In addition to the random seed, this number of unique addresses is another parameter you can use to test different corner cases.

Disclaimer: There may be bugs in testbench which you must debug!

Part 2

Now we will extend the problem to a more general case. Use four copies of the CAM and replicate other storage structures to support 128 instructions. Use the lsq_rand_bench_128.v testbench and common_128.v in the part2 directory.

Use a separate directory for this part.

Part 3

Take your problem from part 1 and build a 2-state pipelined LSQ module. Fetch and writeback requests are still serviced in the same cycle that they arrive just like in part 1. However, execute requests can take one or two cycles depending on whether they cause a LSQ hit. Real-life LSQ and CAMs resemble this case more because it is easier to build a CAM that must support only read or only match in one cycle.

- If an instructions results in a LSQ miss i.e. no other instructions match that same address, then this can be detected in a single cycle from the cam_match operation and LSQHit=0 is returned in the cycle in which the request was provided, just like in part 1.
- However, on a LSQHit, the data that needs to be returned for a load requires a second CAM operation of reading the CAM. This operation is performed in the next cycle, during which the processor must be stalled. The LSQ should assert the exec_stall signal and return data and LSQHit the next cycle. Also remember that exec_PCOut, exec_addrOut, exec_validOut will also be delayed by one cycle.
- On a LSQHit for a store, the flush mask is calculated in the following cycle, and exec_stall is asserted for one cycle like in the above case, and LSQHit is retured in the second cycle.

Use cam_sync.v for this part. You can use the same testbench, lsq_rand_bench.v