Question 1:

A 16-byte cache has 4-byte blocks, has 2 sets, and is 2-way set-associative. The cache initially is empty (all valid bits are off: indicated by a blank box in the table below). The cache receives requests in the sequence listed below. For each address in the sequence (a) split it into the tag, index, and offset; (b) categorize the access as a *hit*, a *compulsory miss*, a *conflict miss*, or a *capacity miss* (You can abbreviate hit=H, Compulsory=O, Conflict=F, Capacity=P); (c) show the new contents of the cache after the access—write the tags for each way, and note which way is LRU. The first one is done for you:

Address	Split Address				Set 0			Set 1		
	Tag	Index	Offset	Result	Way 0	Way 1	LRU Way	Way 0	Way 1	LRU Way
FF	1F	1	3	O			0	1F		1
22										
42										
31										
43										
30										
23										
FC										
08										
10										
08										
33										

Question 2:

You are designing the memory hierarchy for a processor which has a memory access latency of 120 ns.

1. The Level 3 cache can be accessed in 30 ns on a hit. What hit rate does it need to acheive an average access latency of 60ns?

2. Assuming the Level 3 cache acheives the hit rate in part 1 (thus its average access latency is 50ns), and that the Level 2 cache has a 80% hit rate, what hit latency does the L2 cache need in order to acheive an average access time of 25 ns?

3. You have two choices for the Level 1 data cache design:

   Option A 64KB, 8-way set associative, 2.5 ns hit latency, 95% hit rate

   Option B 32KB, 4-way set associative, 1 ns hit latency, 90% hit rate

   Assuming the 60ns L3 and 25ns L2 average access times above, which option would you pick? Why?

4. Some processors support the ability to dynamically increase their clock frequency (and voltage so that the logic runs faster) while they run. Suppose that the above data assumes the processor is running at 2.0GHz, but the processor can increase its frequency up to 4GHz. If the processor were to spend most of its time running this application at 4.0GHz, would your answer to part (3) change? Why or why not?

Question 3:

For this question, you will be implementing a simple cache in VHDL. Your cache will be direct-mapped (1-way), have 127 sets, and 64 byte blocks. Your cache will implement a writeback policy for stores. You will have a tag array (which will include a valid bit and a dirty bit), and a data array. We have provided a cache tester module which will help you test and debug your cache on the FPGA (explained later). However, this is not a substitute for simulating your cache and looking at waveforms.

Initialization. Before *rst* is asserted (high), the state of your cache is undefined. When *rst* is asserted, you should initialize your cache by clearing out the tag array (mark all blocks invalid), and performing any other initialization you want. When your initialization routine is complete, your cache should assert *ready* (high). Once *ready* goes high, the test framework will begin sending requests to you. You do not need to process multiple requests concurrently—once a request is inititiated, it will be completed before the test framework starts a new request.

Load Requests. We recommend starting with only loads and fully debugging them before moving on to stores. The cache tester will send a request to your cache by asserting *req_vld* and providing the request address on *addr* in the same cycle. During this cycle, if *req_st* is low, the request is a load. Your cache should process the request and determine if the request is a hit or a miss. Your cache may take any reasonable number of cycles to process the request (after 1024 cycles, the tester will determine your cache is stuck and signal an error). All load requests are for 8 bytes at a time, and are always 8-byte aligned. This is why *addr* only goes down to 3—the other bits are implicitly zeros.

Note that the request data will **only** be provided by the tester in the cycle when *req_vld* is asserted. After this cycle, the request inputs contain undefined values, so you must store them in registers in your cache.

Load Hits. If the request is a cache hit, your cache should assert *resp_hit*, while keeping *resp_miss* low. The value loaded should be placed on *data_out (64 downto 0)*. The data must be sent out in the same cycle that *resp_hit* is signaled.

Clean Misses. If the request is a cache miss, your cache should assert *resp_miss* while keeping *resp_hit* low. For a clean miss, *resp_dirty* should remain low (dirty misses are described later). Once the cache has signaled the misse, it should wait for the cache tester to send in the fill data (also known as a "reload"). The availabiltiy of reload (fill) data is indicated by the signal *rld_vld* (which stands for "reload valid"). Whenever *rld_vld* is asserted, *data_in* has 16 bytes of reload data. Because reload data is sent to your cache 16 bytes at a time, 4 "beats" of data are required (meaning the entire reload process takes 4 cycles). The reload data will always be sent in ascending order of address—so the first beat will have bytes 0–15, the second will have bytes 16–31, the third will have bytes 32–47, and the fourth will have bytes 48–63 within the block.

Once the data is reloaded, your cache should complete the request. For loads, this means that it needs to signal a hit (*resp_hit*=1, *resp_miss*=0), correct data on *data_out*).

Implement, Test, and Debug the Above. We recommend you read through this entire document before starting (especially the section on debugging tools), but that you implement the above before proceeding to stores and dirty misses.

Store Requests. If *req_st* is high when *req_vld* is asserted, then the request is a store, not a load. As with loads, all store requests are for aligned 8-byte quantities. The data to store is provided on *data_in (63 downto 0)* in the cycle that *req_vld* is asserted. As with loads, all of the request inputs are undefined after the cycle where *req_vld* is asserted.

Store Hits. If the store request is a hit, your cache should update the data in the block to reflect the newly stored data. It should also set the dirty bit in the tag array so that you know that any future eviction of that block requires the data to be written back to the next level. Once the cache is properly updated, it should assert *resp_hit*.

Store Misses. Store misses undergo a very similar procedure to load misses: they signal *resp_miss*, possibly cast-out the dirty block (see below), wait for the completion of the reload, process the request (updating the newly reloaded block), then signal a hit by asserting *resp_hit*.

Dirty Misses. Implementing stores introduces the possibility of a dirty miss (for both store and load misses). For any miss, if the block being evicted is dirty, your cache should assert *resp_dirty* at the same time that it asserts *resp_miss*. When a dirty miss occurs, the test framework will wait for the cache to writeback (also called "castout") the dirty block before providing the reload data. Just like a reload requires 4 beats to fill data into the cache, a castout requires 4 beats to get data out of the cache. When your cache needs to perform a castout, it should cast the data out in the same order that data is reloaded (bytes 0–15 first, etc). On each of the 4 cycles where the cache castsout data, it should assert *co_vld*. Whenever *co_vld* is asserted, *co_addr* must have the address being castout, and *data_out* must have the data being cast out. Once all 4 beats of the castout have happened, the test framework will send reload data, and the rest of the miss handling proceeds in the same way as for a clean miss.

Testing and Debugging Tools.

We have done our best to provide you with a rich set of support infrastructure to assist you in testing and debugging your cache. The cache tester reads a ROM which contains a list of requests to send to your cache, as well as their expected outcomes. It also contains the reload data, and (for dirty misses) the expected castout data.

You can change which of the three traces the tester runs by using the Quartus Mega-function editor to change the hex file for the testrom. You should start with `no_stores.hex`, which has no stores. Once it works, you should try `all_stores.hex` which has only stores (and dirty misses), and `mixed.hex` which has a mixture of loads and stores.

Leds The green, red, and 7-segment LEDs will show you the state of the cache tester whenever it stops—either after succesfully running your cache through its list of commands, or after encountering an error. If all of the green LEDS (lower right of the board) come on (and none of the red ones), then the run was successful. If one or more of the red ones comes on, it indicates the following error(s):

0 Your cache indicated a hit, but the request should have been a miss

1 Your cache indicated a miss, but the request should have been a hit

2 Your cache indicated a the miss was dirty, but it should have been clean.

3 Your cache indicated a the miss was clean, but it should have been dirty.

4–6 The cast out address was incorrect. 4 indicates the tag portion was wrong, 5 indicates the index portion was wrong. 6 indicates the offset portion was wrong

7 A load produced the wrong data

8 Your cache either (a) asserted both *resp_hit* and *resp_miss* at the same time, or (b) asserted at least one of these at an un-expected time (e.g., no request sent, during reload processing, etc).

9 Your cache asserted *co_vld* at an un-expected time.

10–13 Your cache provided incorrect cast out data. Each of these LEDs indicates a different 4 bytes (10=bytes 0–3, 11=bytes 4–7, 12=bytes 8–11, 13= bytes 12–15).

14–16 Your cache seemed to get stuck (took longer than 1024 cycles) while the tester was waiting for a response from it. What the tester was waiting for is encoded as follows:

- **001** A hit
- **010** A miss
- **011** Hit after a reload (to complete a request that was originally a miss)
- **100** Castout bytes 0–15
- **101** Castout bytes 16–31
- **110** Castout bytes 32–47
- **111** Castout bytes 48–63, initialization (ready signal), or any other scenario not listed here.

17 An internal error occured, such a the trace missing reload or castout data.

7-segment LEDs When the tester stops (either on success or failure), it will show you some information on the 7-segment LEDs. What exactly this information is depends on the settings of the right-most four switches on the board. The following explains this based on the switches where 0 indicates the switch is in the down position and 1 indicates it is in the up position:

0000 The right-most four numbers show the hex encoding of the test ROM entry number that the tester was on (this may be off by 1 if the tester had prepared to read the next entry before signaling the error). You can use this for two purposes: (1) to see if you are making progress from one bug fix to the next (getting futher) and (2) you can look in the hex files to see what is supposed to be happening. The next digit displays the tester's internal state number before the error. The next digit shows wiether or not it expected a hit (lowest bit), whether or not it was doing a store (next bit), and whether or not it expected the miss to be dirty (next bit). The remaining two digits contain whatever debug data your cache has sent up for display.

0001 The last address your cache cast out.

1001 The expected cast out address.

0100 Bits 31 downto 0 of your cache's *data_out.*

1100 The expected value for bits 31 downto 0 of your cache's *data_out.*

0101 Bits 63 downto 32 of your cache's *data_out.*

1101 The expected value for bits 63 downto 32 of your cache's *data_out.*

0110 Bits 95 downto 64 of your cache's *data_out.*

1110 The expected value for bits 95 downto 64 of your cache's *data_out* (for cast outs only).

0111 Bits 127 downto 96 of your cache's *data_out.*

1111 The expected value for bits 127 downto 96 of your cache's *data_out* (for cast outs only).

Note that the encoding scheme is designed so that flipping the fourth switch toggles between "what you did" and "what the tester expected."

Your own debug data. As described above, when the four switches which control debug output are all in the low (0) position, you can display 8 bits of debug data from your cache on the left-most 2 7-segment LEDs (as a 2-digit hex number). Your cache should send whatever data you want it to send out for debugging on *debug_info*. Your cache also receives a 4-bit input *debug_sel* which lets you choose what input you want to display based on switches 8–11. Note that even when the tester has stopped, the clock is still running, so you should not have *debug_info* setup in a way that it changes every cycle when requests are not coming in (else it may be unreadable).

Single-stepping. The left-most switch on the board sets the tester into single-step mode. During this mode, the clock only advances one cycle per push of Key 3 (except during initialization). This allows you to see what is happening in the system on a cycle-by-cycle basis. You can also change the debug output selectors while the clock is stopped, and the output will display properly on the LEDs, since there is only combinatorial logic to select which output is displayed (no flip-flops).

Note that to reset your processor while in single-step mode, you will need to press Key 3 once while holding the reset button (Key 0), so that the clock can advance while reset is active. This is required because the reset signal is latched in a DFF before being sent to the rest of the system to avoid weird conditions where it transitions late in a clock cycle. Without advancing the clock, the DFF which latches the reset signal never captures it.

Trace format. You may wish to look at the HEX files to see what requests are happening in the system. This section provides a brief description of the format, in case you wish to do so.

Each line has a 130-bit hexidecimal number. The first hex digit (which corresponds to 2 bits) tells what kind of entry it is. 0=end, 1=make request, 2=castout, 3=reload.

The entries which start with 0 are simplest: they are all 0s and indicate that the test is complete.

The entries which start with 1 (requests) are the most complex:

63 downto 0 The data (either expected value for a load, or data to store).

95 downto 67 The address to store or load

106 1 if this is a store request, 0 if this is a load request.

107 1 if this request is expected to miss, 0 if a hit is expected

108 1 if this is a dirty miss, 0 if this is a clean miss (ignored if a hit is expected).

127 downto 109 The expected tag portion of the address for a castout (this is ignored if the request is not a dirty miss). Note that only the tag is stored since the remaining portions can be calcuated (index from the requested address and offset from which beat of the castout is expected next).

Entries which start with 2 (cast out) or 3 (reload) simply contain 16-bytes of data in the remaining 128-bits of the entry. This data is either the data sent into the cache (reloads) or the expected data (castouts).

As a side note: a useful consequence of this is that the switch settings to show expected castout data (11XX) can also be used to show the entry read from the ROM, since the expected castout data is the entire ROM entry. This could be useful in situations such as single-stepping the FPGA and wanting to see what data will be sent in for a reload (which is also the entire ROM entry).

You can also create ROMs of your own to test specific situations if you want.

Other.

Clock Frequency. The default clock frequency for your cache to run at is 75 MHz. This should be a reasonably easy frequency target to make (mine could do 80MHz without any work tuning it to go faster), however, you should work on getting your processor to work first, then work fast. You might want to set your frequency to something very slow (5 MHz) until it works, then set the frequency back. You can change your frequency by using the Quartus MegaFunction Wizard and editing the PLL's settings.

Once your cache works, you may wish to optimize it for speed (or may have difficulty getting to 75MHz). You can target the following frequencies for the indicates gain (or loss) of (extra) points:

25 MHz minus 5 points.

50 MHz minus 3 points.

75 MHz default: no extra points lost or gained

100 MHz +5 extra credit, as long as your cache's Thit is 3 or fewer cycles

125 MHz +10 extra credit, as long as your cache's This is 3 or fewer cycles

IMPORTANT NOTE: You **MUST** pass the timing analyzer at the frequency you use. If you do not make timing but your cache "looks like" it is working, that is because the error checking logic does not fit into the clock cycle and the error signals aren't making it to the DFFs in time to get registered.

Hit latency. While real caches need to have fast hit latencies, your cache does not need to. However, if you choose to optimize it for speed, you may receive extra credit by having a fast Thit. Inorder to receive any extra credit points for Thit, your design must be clocked at *at least* 75 MHz (that is, you may not loose points for slow frequency and gain them back for fast Thit). You can gain extra credit with the following hit latencies:

1 cycle If you are able to assert *resp_hit* on the cycle **immediately** following *req_vld* then you will receive 7 points of extra credit. Of course, for your cache to work correctly, you must provide the correct data in that cycle as well.

2 cycles If you are able to assert *resp_hit* two cycles after *req_vld* then you will receive 3 points of extra credit. Of course, for your cache to work correctly, you must provide the correct data in that cycle as well.

Note that in the best case, you can receive a score of 117 (out of 100) on this homework if your cache is clocked at 125MHz with Thit = 1. This is likely *quite* difficult.

Restrictions. The following restrictions are in place on how you may implement this homework:

- You may not use `process` or `variable`.
- You may not use any Quartus Megafunctions, except for those which we have provided (RAMS for the tags and data)

- You may modify the code that we have provided in order to optimize it for speed for extra credit. However, if you do so, you should re-test your code (at the standard clock frequency) with an un-modified version of the tester. Your code must still work with the regular tester at the end.
- You may **not** change the interface between the cache and the tester.