Cache Design

For the last part of the project, we will design a cache controller. Considering the amount of time left we will only require you to design a cache controller, so you will not be integrating the cache with your processor implementation.

The cache is an important component of modern computers. It helps us speed up the memory operations. As discussed in the lectures there can be many different types of cache designs like direct mapping, 2-way set associative, fully associative, etc. For the last part of your project, you will be designing a direct-mapped cache. We have provided the cache datapath (`cache.v`) and memory module (`four_bank_mem.v`) so effectively all you need to do is design is a cache controller (or to be specific memory system controller)

Note: For this part of the project you only need to edit `mem_system.v`. You should not change any other files

Also do note that **Discussion 10** covers this topic in good detail. We have discussed details regarding designing a cache controller FSM as well as the components you will use in this assignment. It is a very good starting point and you are highly encouraged to take a look at the slides (Link) or view the video (Link) on canvas. You will find the slides under Files -> " Discussion Section" and video can be found under "Kaltura Gallery" on canvas.

As usual, all the Verilog coding rules, naming conventions, etc apply to this project also. If you have forgotten them please visit the hw1.pdf (or Canvas-> Files-> Verilog Tools, Scripts & Guides).

Provided Components

For this assignment following files are provided to you:

File	Description
cache.v	the basic cache data structure module outlined below
clkrst.v	standard clock reset module
final_memory.v	memory banks used in four_bank_mem
four_bank_mem.v	four banked memory
loadfile_*.img	memory image files
mem.addr	sample address trace used by perfbench

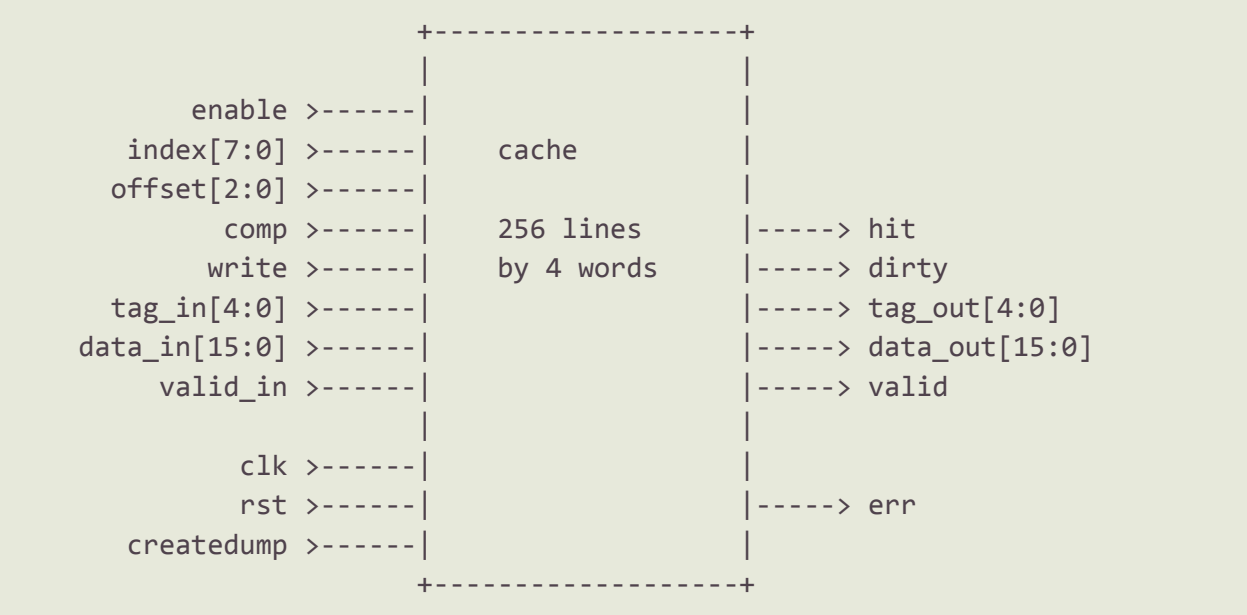
memc.v	used by the cache to store data
mem_system_hier.v	instantiates the mem_system and a clock for the testbench
mem_system_perfbench.v	testbench that uses supplied memory access traces
mem_system_randbench.v	testbench that uses random memory access patterns
mem_system_ref.v	reference memory design used by testbench for comparison
mem_system.v	the memory system: the cache and main memory are instantiated here and this is where you will need to make your changes
memv.v	used by the cache to store valid bits

Out of all these files, you only need to edit the mem_system.v file.

The following 2 sub-sections provide details regarding the working of cache.v and four_bank_mem.v modules. You will use these files in your mem_system.v to design your memory system.

Cache Interface and Organization

The cache module is already designed and available for your use. You will use this cache datapath to design your memory system. The following diagram shows the external interface of the cache module:

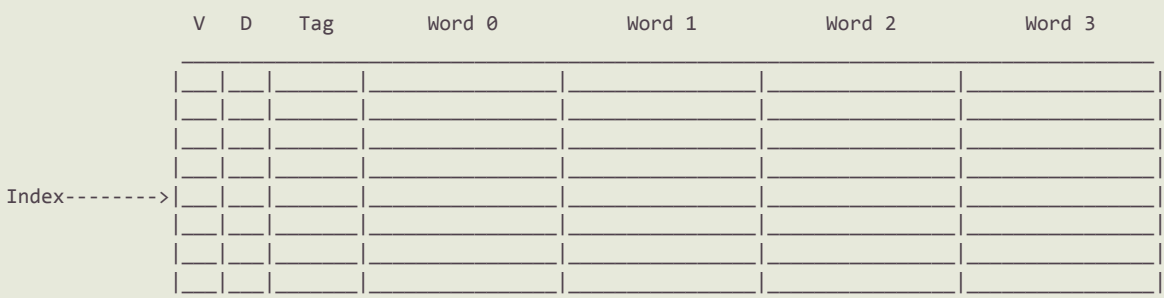


Details regarding each signal can be found in the table below:

Signal	In/Out	Width	Description
enable	In	1	Enable cache. Active high. If low, "write" and "comp" have no effect, and all outputs are zero.
index	In	8	The address bits used to index into the cache memory.
offset	In	3	offset[2:1] selects which word to access in the cache line. The least significant bit should be 0 for word alignment. If the least significant bit is 1, it is an error condition.
comp	In	1	Compare. When "comp"=1, the cache will compare tag_in to the tag of the selected line and indicate if a hit has occurred; the data portion of the cache is read or written but writes are suppressed if there is a miss. When "comp"=0, no compare is done and the Tag and Data portions of the cache will both be read or written.
write	In	1	Write signal. If high at the rising edge of the clock, a write is performed to the data selected by "index" and "offset", and (if "comp"=0) to the tag selected by "index".
tag_in	In	5	When "comp"=1, this field is compared against stored tags to see if a hit occurred; when "comp"=0 and "write"=1 this field is written into the tag portion of the array.
data_in	In	16	On a write, the data that is to be written to the location specified by the "index" and "offset" inputs.
valid_in	In	1	On a write when "comp"=0, the data that is to be written to valid bit at the location specified by the "index" input.
clk	In	1	Clock signal; rising edge active.
rst	In	1	Reset signal. When "rst"=1 on the rising edge of the clock, all lines are marked invalid. (The rest of the cache state is not initialized and may contain X's.)
createdump	In	1	Write contents of entire cache to memory file. Active on rising edge.
hit	Out	1	Goes high during a compare if the tag at the location specified by the "index" lines matches the "tag_in" lines.
dirty	Out	1	When this bit is read, it indicates whether this cache line has been written to. It is valid on a read cycle, and also on a compare-write cycle when hit is false. On a write with "comp"=1, the cache sets the dirty bit to 1. On a write with "comp"=0, the dirty bit is reset to 0.

tag_out	Out	5	When "write"=0, the tag selected by "index" appears on this output. (This value is needed during a writeback.)
data_out	Out	16	When "write"=0, the data selected by "index" and "offset" appears on this output.
valid	Out	1	During a read, this output indicates the state of the valid bit in the selected cache line.

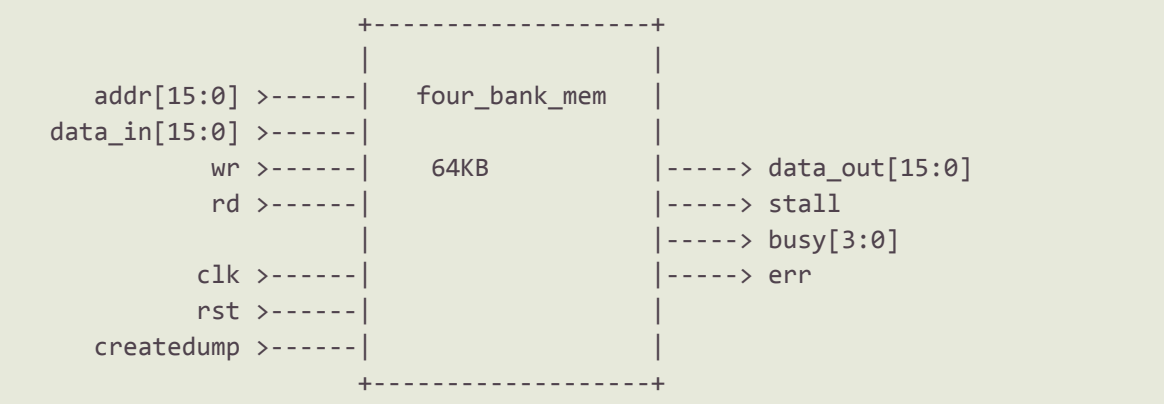
This cache module contains 256 lines. Each line contains one valid bit, one dirty bit, a 5-bit tag, and four 16-bit words:



Four Banked Memory Module:

The second important module provided to you is the four banked memory module. Four Banked Memory is a better representation of a modern memory system. It breaks the memory into multiple banks. The four-cycle, four-banked memory is broken into two Verilog modules, the top level `four_bank_mem.v` and single banks `final_memory.v`.

Following is the diagram of the four banked memory module:



It is important to understand the timing of the four banked memory module. Please go through the timing diagram, signal definitions, and additional details below to understand who the module is really working.
Timing:

```
    |            |            |            |            |            |
    | addr       | addr etc   | read data  |            | new addr   |
    | data_in    | OK to any  | available  |            | etc. is    |
    | wr, rd     |*diffferent*|            |            | OK to      |
    | enable     | bank       |            |            | *same*     |
    |            |            |            |            | bank       |
                  <----bank busy; any new request to--->
                       the *same* bank will stall
```

Signal description table:

Signal	In/Out	Width	Description
addr	In	16	Provides the address to perform an operation on.
data_in	In	16	Data to be used on a write.
wr	In	1	When wr="1", the data on DataIn will be written to Mem[Addr] **four cycles** after wr is asserted.
rd	In	1	When rd="1", the DataOut will show the value of Mem[Addr] **two cycles** after rd is asserted.
clk	In	1	Clock signal; rising edge active.
rst	In	1	Reset signal. When "rst"=1, the memory will load the data from the file "loadfile".
createdump	In	1	Write contents of memory to file. Each bank will be written to a different file, named dumpfile_[0-3]. Active on rising edge.
data_out	Out	16	Two cycles after rd="1", the data at Mem[Addr] will be shown here.
stall	Out	1	Is set to high when the operation requested at the input cannot be completed because the required bank is busy.
busy	Out	4	Shows the current status of each bank. High means the bank cannot be accessed.
err	Out	1	The error signal is raised on an unaligned access.

This is a byte-aligned, word-addressable 16-bit wide 64K-byte memory.

Requests may be presented every cycle. They will be directed to one of the four banks depending on the [2:1] bits of the address.

Two requests to the same bank which are closer than cycles N and N+4 will result in the second request not happening, and a "stall" output being generated.
Busy output reflects the current status of each individual bank.
Concurrent read and write are not allowed.
On reset, memory loads from file "loadfile_0.img", "loadfile_1.img", "loadfile_2.img", and "loadfile_3.img". Each file supplies every fourth word. (The latest version of the assembler generates these four files.)

Format of each file:

```
@0
<hex data 0>
<hex data 1>
...etc
```

If input create_dump is true on the rising clock, contents of memory will be dumped to file "dumpfile_0", "dumpfile_1", etc. Each file will be a dump from location 0 up through the highest location modified by a write in that bank.

Design the FSM:

So the first part that you will need to do is to design an FSM for your memory system. Discussion 10 covers in good detail how you can go about it so we would highly recommend that you take a look at it once (slides (Link) or the video (Link) are on canvas).

Please go ahead and take a look at `mem_system.v` code and see the module interface.

Notes on the mem_system module:

- The system should ignore new inputs when it is stalled (not an error). Timing will be an important part of this problem; Designs that stall longer than necessary will be docked points.
- The Done signal should be asserted for exactly one cycle. If the request can be satisfied in the same cycle that data should be presented, Done should be asserted in that same cycle.
- The CacheHit signal should indicate whether the request was a hit in the cache. Note that hit & ~valid should be treated as a miss.
- The Stall output should indicate whether the memory system is stalling, ignoring incoming requests.
- The memtype parameter decides whether this is an instruction or data cache which is used to generate the names for the dump files.

You will need to determine how your cache is arranged and functions before starting implementation. Draw out the state machine for your cache controller as this will be required. You may implement either a Mealy or Moore machine though a Moore machine is

recommended as it will likely be easier. Be forewarned that the resulting state machine will be relatively large so it is best to start early.

Some details for the FSM:

- The inputs to your control FSM are:
  - The inputs to the `mem_system` module;
  - The outputs of cache `c0` and main memory `mem`.
- The outputs of your control FSM should be:
  - The inputs to cache `c0` and main memory `mem`;
  - The outputs of the `mem_system` module.
- Your FSM should make it clear what policies are being used for writing data, evicting blocks, etc.
- Notice that we have exact knowledge that our four-banked main memory access takes four cycles for write and two cycles for read. Take this into account.
- We recommend drawing a Moore machine (output solely depends on the state, instead of on each edge). Please make a complete list of all cache control signal values ({`cache c0 inputs, main memory mem inputs, mem_system outputs`}) on every state.
- For an example please refer to discussion 10, but if you want an additional example you can take a look here: http://user.engineering.uiowa.edu/~hpca/lecturenotes/verilogcachelinesize2.pdf. (Note that this example uses a Mealy machine, has different inputs from our case, does not list its complete outputs, and does not make it clear which policies are being used.)

You will be **required to submit** a cache FSM diagram so make a clean pdf with your cache FSM named `cache_controller_fsm.pdf` and submit it along with your code.
(Also please go through the rest of the information in this document before making a cache controller FSM.)

Direct-Mapped Cache:

You will need to implement a direct-mapped cache over four mapped memory module. Please refer to the lecture notes to understand how direct-mapped cache works.

Signal Interactions:

Although there are a lot of signals for the cache, its operation is pretty simple. When "enable" is high, the two main control lines are "comp" and "write". Here are the four cases for the behavior of the direct-mapped cache:

Compare Read (comp = 1, write = 0)

This case is used when the processor executes a load instruction. The "tag_in", "index", and "offset" signals need to be valid. Either a hit or a miss will occur, as indicated by the "hit" output during the same cycle. If a hit occurs, "data_out" will contain the data, and "valid" will indicate if

the data is valid. If a miss occurs, the "valid" output will indicate whether the block occupying that line of the cache is valid. The "dirty" output indicates the state of the dirty bit in the cache line.

Compare Write (comp = 1, write = 1)

This case occurs when the processor executes a store instruction. The "data_in", "tag_in", "index", and "offset" lines need to be valid. Either a hit or a miss will occur as indicated by the "hit" output during the same cycle. If there is a miss, the cache state will not be modified. If there is a hit, the word will be written at the rising edge of the clock, and the dirty bit of the cache line will be written to "1". (The "dirty" output is not meaningful as this is a write cycle for that bit.) NOTE: On a hit, you also need to look at the "valid" output! If there is a hit, but the line is not valid, you should treat it as a miss.

On a miss, the "valid" output will indicate whether the block occupying that line of the cache is valid. The dirty bit will be read, and will indicate whether or not the block occupying that line is dirty. On the other hand, if "hit" is true while "write" and "comp" are true, "dirty" output is not meaningful and will remain zero (because the dirty bit of the cache was performing a write).

Access Read (comp = 0, write = 0)

This case occurs when you want to read the tag and the data out of the cache memory. You will need to do this when a cache line is victimized (i.e. you are evicting that cache line), to see if the cache line is dirty and to write it back to memory if necessary. With "comp"=0, the cache basically acts like a RAM. The "index" and "offset" inputs need to be valid to select what to read. The "data_out", "tag_out", "valid", and "dirty" outputs will be valid during the same cycle.

Access Write (comp = 0, write = 1)

This case occurs when you bring in data from memory and need to store it in the cache. The "index", "offset", "tag_in", "valid_in" and "data_in" signals need to be valid. On the rising edge of the clock, the values will be written into the specified cache line. Also, the dirty bit will be set to zero.

Testing:

For testing, we will use 2 types of testing. The first one will be to see the performance of the cache. This basically means the number of hits and misses your cache is giving for a given sequence of accesses. And the second testing will be for correctness where we will check whether your memory system is giving the correct data output or not.
For performance testing, we will use the `mem_system_perfbench.v` and for correctness, we will use the `mem_system_randbench.v`. More details for both of them is covered in the following sections:

Perfbench Testing:

The first testing that you will be doing is the Perfbench testing. The tests here are designed to check your cache performance. This includes testing your cache latency (amount of cycles spend by the cache) and the correct behavior of HITs and MISSes.

To do the testing you will write your own address traces (at least 5). These traces will try to test the cache for different types of behavior. Try to cover as many different types of behavior as possible.

An example address trace file (mem.addr) is provided. The format of the file is the following:

- Each line represents a new request
- Each line has 4 numbers separated by a space
- The numbers are: Wr Rd Addr Value

You can follow the following steps to do your testings:

1. Once you have designed you cache you can run your address trace using following command:

```
wsrun.pl -addr mem.addr mem_system_perfbench *.v
```

   Note: We have provided an example trace in the files start with that.

2. If it runs correctly you will see following output:

```
# Using trace file   mem.addr
# LOG: ReQNum    1 Cycle       12 ReqCycle        3 Wr Addr 0x015c Value
0x0018 ValueRef 0x0018 HIT 0
#
# LOG: ReqNum    2 Cycle       14 ReqCycle       12 Rd Addr 0x015c Value
0x0018 ValueRef 0x0018 HIT 1
#
# LOG: Done all Requests:          2 Replies:          2 Cycles:         14
Hits:          1
# Test status: SUCCESS
# Break at mem_system_perfbench.v line 200
# Stopped at mem_system_perfbench.v line 200
```

   Few important things to note here:

   - Test status: **SUCCESS does not mean you have passed the tests.** There will be some additional testing in next steps to verify the behavior.
   - The trace is for the given example which shows that the first request was a STORE request which was a cache MISS, and the second one was a LOAD request which was a cache HIT.
   - Please take note of this behavior, especially the order of HITS and MISSES. We will compare this behavior later.

3. Now to understand whether the behavior generated by the cache is correct or not, you will run a cache simulator using the following command:

```
cachesim 1 2048 8 mem.addr
```

This command is basically means the following:

```
cachesim <associativity> <size_bytes> <block_size_bytes> <trace_file>
# <associativity> = 1: Because Direct-Mapped Cache
# <size_bytes> = 2048: Because our cache is 256 lines * 4 word (16-bit) sized
# <block_size_bytes> = 8: Because 4 16-bit sized words per block.
```

4. This should output following for the example trace:

```
Store Miss for Address 348
Load Hit for Address 348
```

Compare this behavior with the output from step 2. This behavior should match else your cache behavior is not correct.

5. If all the above steps passed with the example address trace file. You need to create new trace files (at least 5) named mem-1.addr, mem-2.addr, and so on. Make those address traces to test different types of cache behavior. Now jump back to step 1 and repeat the same testing with the new traces.

You need to submit the trace files that you generated i.e. mem-*.addr files along with your final submission.

Note: During final grading we will have our own set of trace files that we will use to grade this section. So try to be through here in your testing.

Once you pass all the test cases here you can move forward with Randbench testing.

Randbench Testing:

Once you are confident that your design is working you should test it using the random testbench. The random bench does the following:

- full random: 1000 memory requests completely random
- small random: 1000 memory requests restricted to addresses 0 to 2046
- sequential addresses: 1000 memory requests restricted to address 0x08, 0x10, 0x18, 0x20, 0x28, 0x30, 0x38, 0x40
- two sets address: 1000 memory requests to test accessing two sets. Design to yield predominantly hits if run on a two-way set-associative cache.

Please note that the randbench will not be testing your performance or behavior. So make sure you pass the Perfbench first before moving here. Randbench will only test the correctness of data from the memory system.

You can run the randbench using following command:

```
wsrun.pl mem_system_randbench *.v
```

Once you have run the command you will see something like following at the end of a successful run:

```
# LOG: Done two_sets_addr Requests: 8001, Cycles: 76816 Hits: 0
# LOG: Done Requests: 8001 Replies: 8001 Cycles: 76817 Hits: 1764
# Test status: SUCCESS
```

Important things to note here:

- The test status SUCCESS shows that you have passed the tests.
- On the line just above test status you will see total requests processed and number of HITS. This hit count may be different but it should be a very small number or 0.
- Lastly, the two_set_addr trace is designed specifically for two-way set associative cache which we are not implementing so it will show 0 hits.

You have passed this test if you have `# Test status: SUCCESS` printed out and the hits count of line just above SUCCESS message is reasonably larger (i.e. its not 0 or very small number)

Grading:

Following is the distribution of grading for this assignment:

1. FSM - 25 points
    - Here we will judge your FSM based on correctness and efficient usage of cycles.
2. Perfbench Testing - 35 points
    - For Perfbench testing, you will have to do your own testing as detailed in the Perfbench section. You will have to submit the address trace (mem-*.addr) that you run for testing with your final submission.
    - We will have our own test cases to see whether the cache is having an appropriate hit rate or not. These test cases will not be provided to you so be sure to do thorough testing on your own.
3. Randbench Testing - 40 points
    - For Randbench testing, the testbench will provide a message saying whether the test passed or failed.

What to submit:

You need to submit a single tar file named `project-cache.tar`.You will submit it directly on the canvas. It will contain the following files:

```
- All *.v files
- cache_controller_fsm.pdf
- mem-*.addr files (used for Perfbench testing, at least 5)
- All *.vcheck files
```