Part 2 - Data Cache [50 marks]

Now that your CPU can access data in memory, your next task is to implement a simple data cache. The goal of using a cache is to reduce the time spent on accessing memory for most accesses based on locality (make the common case fast!). The data cache will act as an intermediary between CPU and data memory (see figure below). For the sake of simplicity, your cache module should use the same signals as the memory module in part 1 when connecting to the CPU.

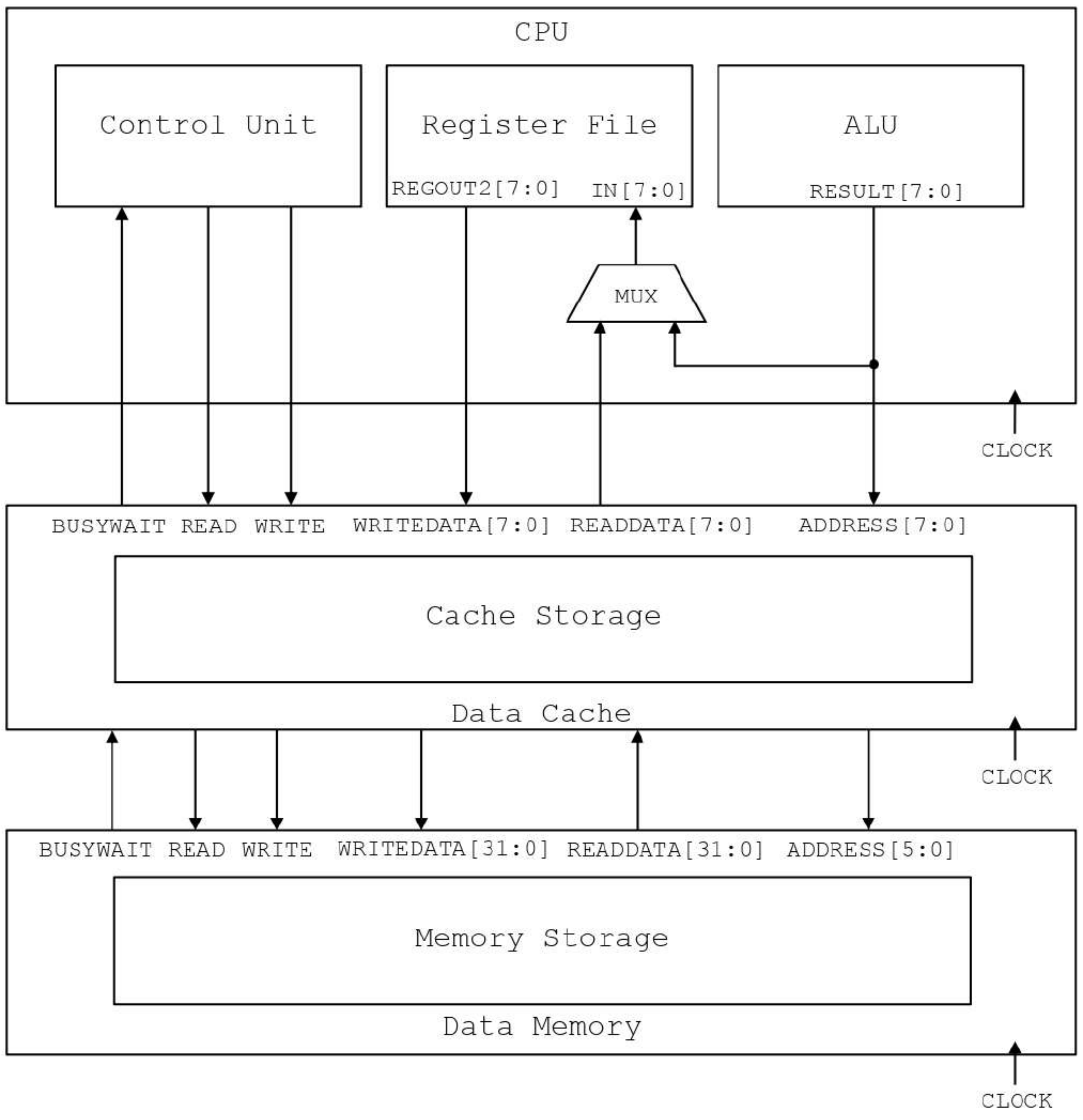

Figure 2: CPU with Data Cache and Data Memory

Following are the parameters to be used when designing your data cache:

- Data ***word-size*** is 1 Byte (8 bits), as defined by the ISA.
- ***Cache size*** is 32 Bytes.
- Use a ***block-size*** of 4 Bytes (∴ cache can hold eight data blocks).
- Use ***direct-mapped*** block placement.
- You need to store the corresponding ***address tag*** along with every data block.
- You need to store a ***valid bit*** along with every data block. At the beginning, all cache entries are empty, therefore invalid.
- You need to store a ***dirty bit*** along with every data block. This should be used to indicate data blocks which are "inconsistent" with the memory. When a block receives a write access, its dirty bit should be set.
- Use ***write-back*** policy for writes. When a block is evicted from the cache AND its dirty bit is set, that block should be written to the memory. An eviction can happen in the event of a cache miss.

In our single-cycle CPU, the address for memory access is made available through the ALU while the `READ/WRITE` control signals are generated early by the control unit (similar to single-cycle MIPS). Therefore, your cache controller should assert its `BUSYWAIT` signal when a `READ/WRITE` control signal is detected, to notify the CPU.

Our CPU expects the memory sub-system to respond in under #2 time units (as `lwd` and `swd` instructions only have #2 time units for memory access, while `lwi` and `swi` instructions have up to #3 time units until the end of the clock cycle). Therefore the cache must resolve hits within that time, in order to let the CPU continue without stalling in the event of a hit. However, cache misses will consume a longer time and the CPU needs to be stalled in such events. The `BUSYWAIT` signal should be held asserted until either the hit is resolved or miss is handled accordingly.

The CPU accesses a single word at a time (word-size = 1 Byte) using an 8-bit memory address. Your cache should split the address into *Tag*, *Index* and *Offset* sections appropriately. Finding the correct cache entry and extracting stored data block, tag, valid and dirty bits should be done based on the *Index*. Include an artificial indexing latency of #1 time unit when extracting these stored values. Then perform *Tag* comparison and validation to determine whether the access is a hit or a miss. Include an artificial latency of #0.9 time units for the tag comparison. These operations should be carried out asynchronously.

Read-hit:

Cache should select the requested data word from the block based on the *Offset*, and send the data word to the CPU asynchronously. Include an artificial latency of #1 time unit in the data word selection. Note that this data word selection latency can overlap with tag comparison, while sending the data to the CPU should be done afterwards based on the hit status. Which means our cache can detect the hit status #1.9 time units after `ADDRESS` is received. Therefore, in the event of a hit, cache controller can de-assert the `BUSYWAIT` signal in order to prevent the CPU from stalling.

Write-hit:

Cache should write the data provided by the CPU to the correct word within the block, based on the *Offset*. Include an artificial latency of #1 time unit in this writing operation. Note that this writing latency *cannot* overlap with tag comparison, as it must depend on the hit status (because we use the write-back policy). The corresponding valid and dirty bits must also be updated at the same time. Since #1.9 time units have already elapsed to detect hit status before writing can be done, cache controller can write the data at the positive edge of the clock (at the start of the next clock cycle). As there is no need to stall the CPU on a write-hit, cache controller can de-assert the `BUSYWAIT` signal once the hit is detected, just like in a read-hit.

Read-miss:

If the existing block is not dirty, the missing data block should be fetched from memory. For this, cache controller should assert the memory `READ` control signal as soon as the miss is detected. Reading the missing block from memory can then start at the positive clock edge. Note that you are given a new memory module for part 2, which deals with blocks of 4-Byte data instead of individual Bytes. A 6-bit block-address should be used when the cache is accessing the memory. Reading a block will take a long time (4×5 = 20 cycles), so the cache controller must wait until the memory de-asserts its `BUSYWAIT` signal while holding the relevant signals stable.

If the existing block is dirty, that block must be written back to the memory before fetching the missing block. For this, cache controller should assert the memory `WRITE` control signal as soon as the miss is detected. Writing back the existing block to memory can then start at the next positive clock edge. Writing back a block will also take a long time (4×5 = 20 cycles), so the cache controller must wait until the memory de-asserts its `BUSYWAIT` signal while holding the relevant signals stable.

On the positive clock edge that the write-back completes, the cache should assert the `READ` control signal. Fetching the missing data block from the memory can then start at the next positive clock edge. Which means there is a 1 cycle gap between write-back and fetch events.

After fetching the missing data block from the memory, the cache should write the fetched data block into the indexed cache entry and update the tag, valid and dirty bits accordingly. Include an artificial latency of #1 time unit in this writing operation. Then the original read access can be served by the asynchronous circuitry, where the status of the hit signal will change after further #1.9 time units, and consequently de-assert the `BUSYWAIT` signal of the cache while sending the requested data word to the CPU.

Therefore, the total data miss penalty add up to 42 CPU cycles if the existing block was dirty, or 21 CPU cycles otherwise.

Write-miss:

A write-back should be performed based on the dirty bit of the existing block, then the missing data block should be fetched from memory and written to the indexed cache entry (similar to a read-miss).

Then the original write access can be served using the asynchronous circuitry, where the status of the hit signal will update after further #1.9 time units, and consequently de-assert the `BUSYWAIT` signal of the cache. The data word sent by the CPU should then be written to the indexed cache entry at the start of the next clock cycle.

The miss penalty should be the same as that for a read-miss.

Note that you may design and implement a finite-state-machine for the part of the cache controller that handles cache misses from the following positive clock edge.

In order to properly simulate fractional delays such as #0.9, you will need to set the timescale of the simulation accordingly. To do that, you can use the `` `timescale <time_unit>/<time_precision> `` syntax in Verilog (you can read more about it from here: https://www.chipverify.com/verilog/verilog-timescale). It's recommended to use a timescale of 1ns/100ps for your implementation. You can include the line to set the timescale (e.g. `` `timescale  1ns/100ps ``) at the beginning of your Verilog file. Also, make sure you include the same timescale in all the Verilog files so that the same timescale is used by all the modules in your design.

1. Implement the data cache module as specified and connect to the CPU via a testbench. Include **a lot of comments**.

2. Test your system thoroughly with several software programs containing data access instructions. Programs can be hardcoded inside the testbench or loaded from a file.

3. Compare your system's performance with the cache-less one from part 1, using test programs, and write a brief report.

4. Submit a compressed file ***groupXX_lab6_part2.zip*** containing your Verilog files with all the modules in your design, testbench, comparison report and screenshots of timing diagrams clearly showing signals related to cache and CPU control.

Note that any form of plagiarism will result in zero marks for the entire lab.