CO224 Computer Architecture - 2020
Department of Computer Engineering

# Lab 6 - Building a Memory Hierarchy

In this lab you will be adding a memory sub-system for your single-cycle CPU. Some systems store both instructions and data in the same memory device, while other systems use separate memory devices for instructions and data. For our system, we will use separate memory devices. This lab will be completed in three parts.

## Part 1 - Data Memory [25 marks]

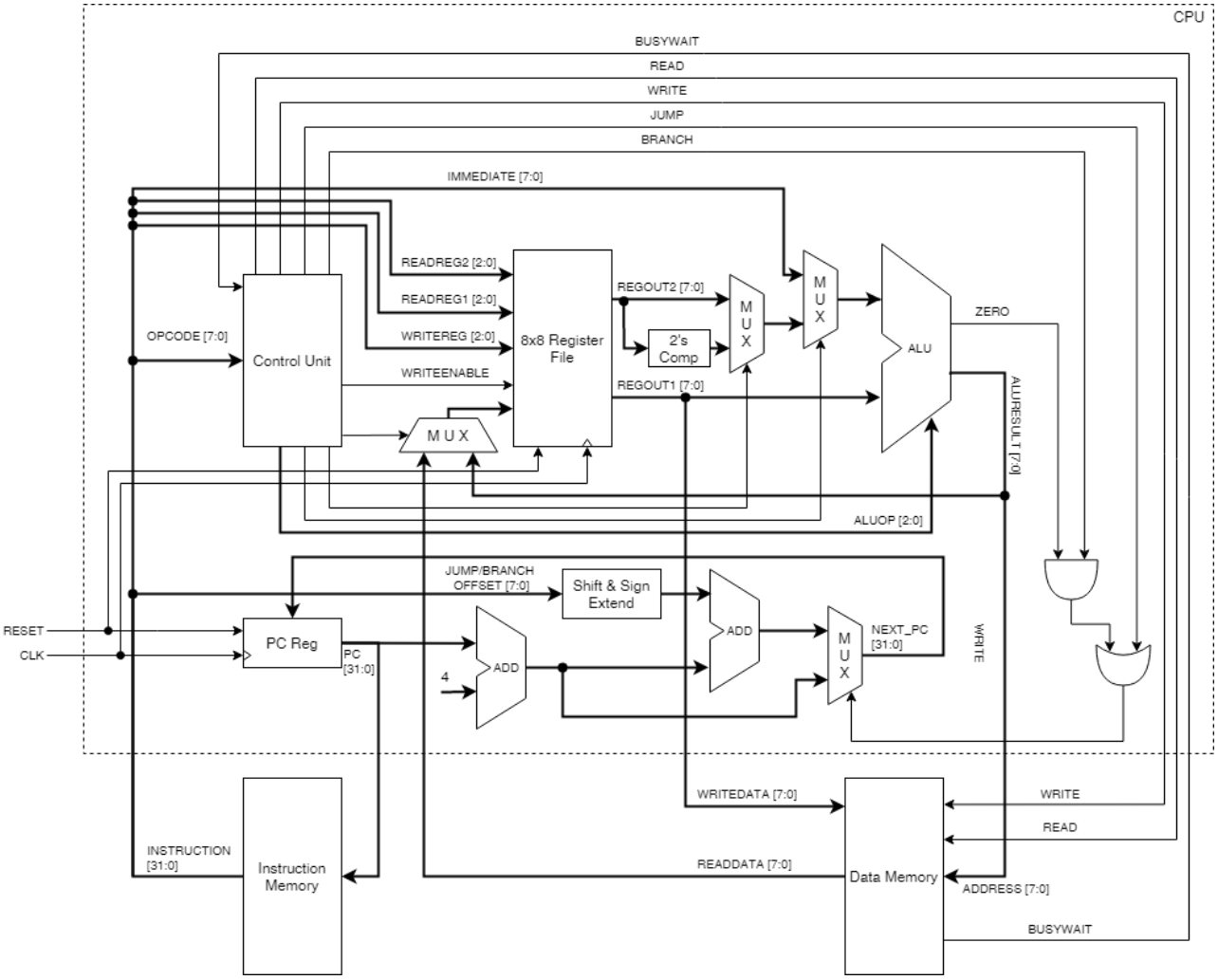



**Figure 1: CPU with Data Memory**

The diagram in Figure 1 shows a 256 Byte Data Memory module being connected to the CPU that you built in the previous lab. A sample memory module is given to you which uses 256 8-bit registers to store data. The memory module uses the following signals to interface with the processor:

`ADDRESS`: Location in memory being accessed (read/write). ALU provides the value for this signal.

`WRITEDATA`: Data value to be stored in the memory, at the location pointed to by `ADDRESS`. This value is supplied by the Register File, then used by the data memory on a positive clock edge.

`READDATA`: Data value read from the memory, at the location pointed to by `ADDRESS`. This value is sent by the memory on a positive clock edge to the Register File for storing.

`READ`: Control signal to request the memory to perform a read operation on the provided `ADDRESS`. This signal is supplied by the CPU control unit, and cleared when memory de-asserts the `BUSYWAIT` signal.

`WRITE`: Control signal to request the memory to perform a write operation on the provided `ADDRESS`. This signal is supplied by the CPU control unit, and cleared when memory de-asserts the `BUSYWAIT` signal.

`BUSYWAIT`: Memory asserts this signal when CPU sets `READ/WRITE` control signals, and keeps it asserted while the operation is in progress. CPU control unit should stall the processor and hold the `ADDRESS` and `READ/WRITE` control signals stable while this wait signal is asserted. Memory de-asserts `BUSYWAIT` when a reading or writing operation is concluded. The next instruction should not be fetched by CPU until `BUSYWAIT` is de-asserted by the memory.

Study the given memory module and see how to connect it to your CPU. Note that a latency of 5 CPU clock cycles (#40 time units) is artificially added to the read and write operations inside the memory module in order to simulate it with realistic timing.

You need to implement hardware support for four new instructions (`lwd, lwi, swd` and `swi`) in your CPU, to access the new data memory. The new instructions will follow a similar encoding format as previous instructions.

| OP-CODE (bits 31-24) | RD (bits 23-16) | RT (bits 15-8) | RS/IMM (bits 7-0) |
|---|---|---|---|

In the new instructions, memory accessing can be done using two different addressing modes: register direct addressing; or immediate addressing. See examples below:

`lwd 4 2` : Read memory at address given in register 2 (`RS`)and store result in register 4 (`RD`). Ignore `bits 15-8`

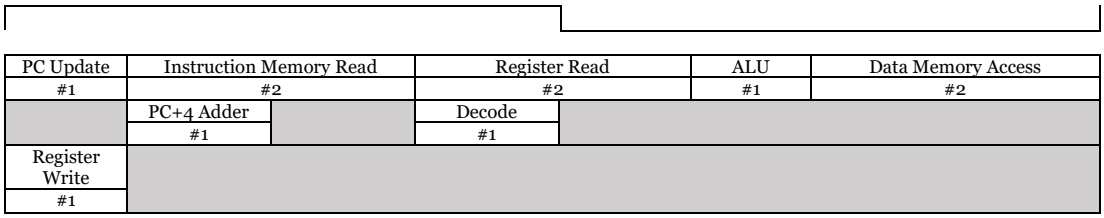
`lwi 4 0x1F` : Read memory at address `0x1F` (`IMM`) and store result it in register 4 (`RD`). Ignore `bits 15-8`

`swd 2 3` : write value from register 2 (`RT`) to the memory at address given in register 3 (`RS`). Ignore `bits 23-16`

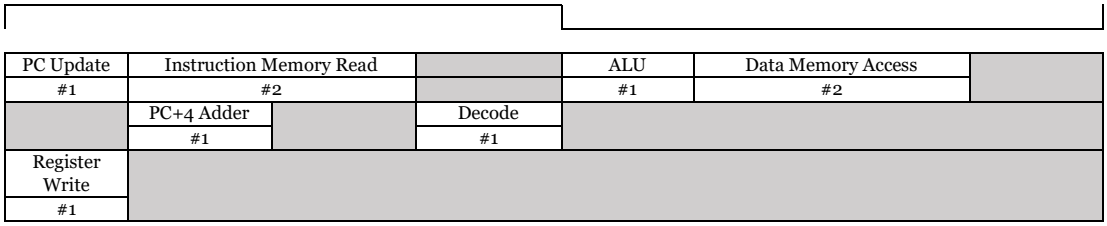
`swi 2 0x8C` : write value from register 2 (`RT`) to the memory at address `0x8C` (`IMM`). Ignore `bits 23-16`

Datapath timing for the new memory access instructions are given below. Note that Data Memory Access time here is based on ideal caches. Your system doesn't have caches in part 1 and accessing memory directly will incur much higher delays, consequently stalling the CPU for several clock cycles.

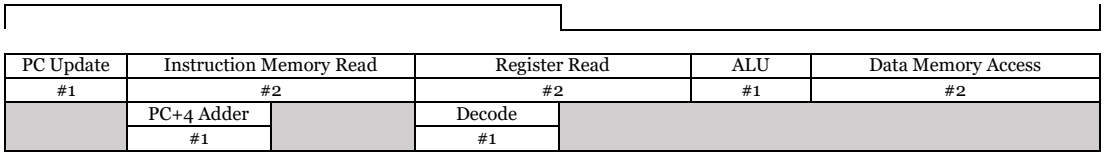
### lwd (register direct addressing):



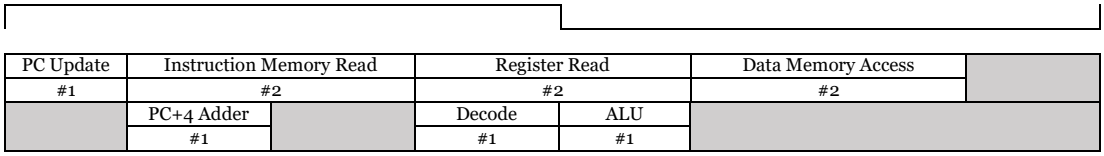

### lwi (immediate addressing):




### swd (register direct addressing):




### swi (immediate addressing):




1. Connect the given *data_memory* module to your *cpu* as specified, via a testbench. Implement the new instructions, and modify the *cpu* accordingly. Make sure to stall the processor when the `BUSYWAIT` signal is asserted by the Memory. Include **a lot of comments**.

2. Test your processor and memory with several software programs containing the new instructions (in addition to the instructions from Lab 5). Program can be hardcoded inside the testbench, or loaded from a file. Store/load data values in the data memory via your program.

3. Submit a compressed file ***groupXX_lab6_part1.zip*** containing your Verilog files with all the modules in your design, testbench, and screenshots of timing diagrams clearly showing signals related to memory accesses.
   **Note that any form of plagiarism will result in zero marks for the entire lab.**

## 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.

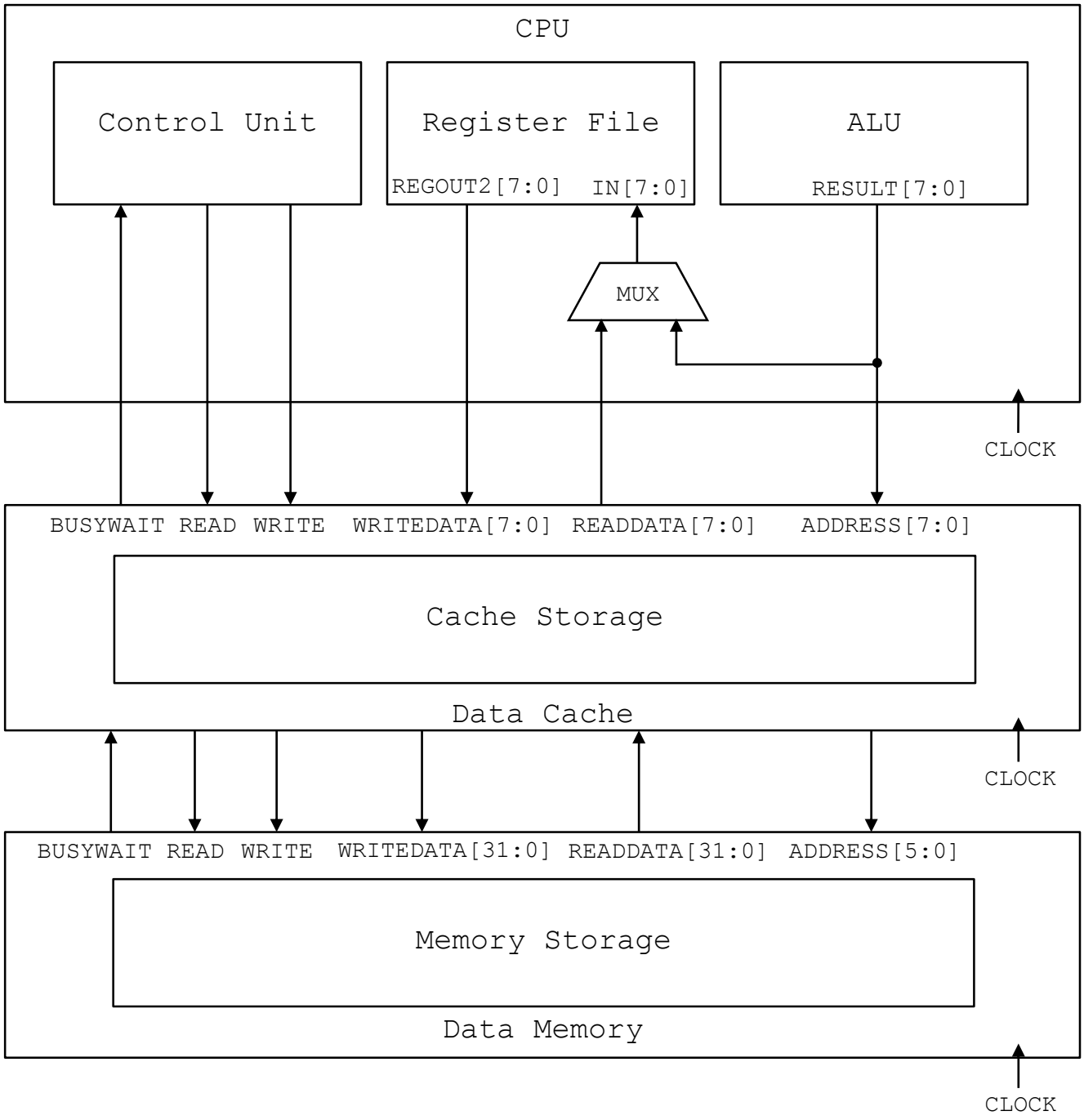



**Figure 2: CPU with Data Cache and Data Memory**

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.**

## Part 3 - Instruction Cache and Memory [25 marks]

Once you have the data cache and data memory working properly, use the same concepts to add an instruction cache and an instruction memory to your system. Note that you will not be using the hardcoded instruction array in your test bench (with #2 time units artificial reading latency) anymore.

Following are the parameters to be used when designing the instruction cache:

- Instruction ***word-size*** is 4Bytes (32 bits), as defined by the ISA.
- ***Cache size*** is 128Bytes
- Use a ***block-size*** of 16 Bytes (∴ cache can hold eight instruction blocks)
- Use ***direct-mapped*** block placement
- You need to store the corresponding ***address tag*** along with every instruction block
- You need to store a ***valid bit*** with every instruction block. At the beginning, all cache entries are empty, therefore invalid.
- Dirty bit and write policies are not needed, as the CPU does not write to the instruction memory. Evicted blocks can simply be discarded.

Your instruction words are 4 Bytes wide. The instruction memory should be able to hold 256 instruction words, making the total size of instruction memory 1024 Bytes. The CPU accesses a single instruction word at a time using a 10-bit word address (from the program counter) where the two least significant bits are zeros.

The instruction cache should split the address into *Tag*, *Index* and *Offset* sections appropriately. Finding the correct cache entry and extracting stored data block, tag and valid bits should be done based on the *Index*. Include an artificial indexing latency of #1 time unit when extracting the 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.

Read hits should be handled asynchronously similar to the data cache, with the only differences being the size of the address and word-size. Include an artificial latency of #1 time unit for selecting the requested instruction word from the block (which can happen in parallel to the tag comparison).

In the event of a miss, the CPU must be stalled using a `BUSYWAIT` signal. Cache controller should assert the memory `READ` control signal as soon as the miss is detected and start fetching the missing 16-Byte block from the instruction memory on the next positive clock edge.

Note that you are given a separate instruction memory module for part 3, which deals with blocks of 16 Bytes. A 6-bit block-address should be used when the cache is accessing the instruction memory. The fetching will take a long time (16×5 = 80 cycles), so the cache must wait (holding the relevant signals stable) until the memory de-asserts its `BUSYWAIT` signal to retrieve the instruction block.

On the positive clock edge this memory read completes, the cache controller should write the fetched block into the indexed cache entry and update the tag and valid bit accordingly. Include an artificial latency of #1 time unit in this writing operation. Then the original read access can be served by the asynchronous logic, 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 at the next positive clock edge while sending the requested instruction word to the CPU.

Therefore, the total instruction miss penalty adds up 81 CPU cycles.

1. Implement the instruction cache module as specified and connect to the CPU via the testbench, along with the instruction memory module. Include **a lot of comments**.

2. Test your system thoroughly with several software programs. Programs can be hardcoded inside the instruction memory module or loaded from a file.

3. Submit a compressed file ***groupXX_lab6_part3.zip*** containing your Verilog files with all the modules in your design, testbench and screenshots of timing diagrams clearly showing signals related to instruction cache and CPU control.

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

Have fun coding. May the force be with you!