Lab 1: Data Cache

Assigned: Thu., 20.09; Due: **Fri., 05.10** (Midnight)

Instructor: Onur Mutlu
TAs: Mohammed Alser, Can Firtina, Hasan Hassan, Jeremie Kim, Juan Gomez Luna, Geraldo Francisco de Oliveira, Minesh Patel, Ivan Puddu, Giray Yaglikci

1. Introduction

In this lab, you will extend the provided pipelined MIPS machine, which is specified in Section 2, by adding a data cache. In this pipelined MIPS machine, we assume that accessing (i.e., reading from and writing to) the memory takes *several cycles*. Thus, this will result in pipeline stalls at the memory stage. As the main goal of this lab, you will add a data cache to reduce the number of stalls due to memory accesses. Section 3 provides the specification of the data cache.

This lab is to be done individually. You are allowed to consult with the TAs, but the lab should be completely your own individual work.

2. Specification of the Pipelined MIPS Machine

2.1. Architecture

Instruction Set. The machine supports all of the following MIPS instructions, *excluding* those related to control-flow, as shown in the following table.

ADD	ADDU	ADDI	ADDIU	AND	ANDI	~~BEQ~~
~~BGEZ~~	~~BGEZAL~~	~~BGTZ~~	~~BLEZ~~	~~BLTZ~~	~~BLTZAL~~	~~BNE~~
DIV	DIVU	~~J~~	~~JAL~~	~~JALR~~	~~JR~~	LB
LBU	LH	LHU	LUI	LW	MFHI	MFLO
MTHI	MTLO	MULT	MULTU	NOR	OR	ORI
SB	SH	SLL	SLLV	SLT	SLTI	SLTIU
SLTU	SRA	SRAV	SRL	SRLV	SUB	SUBU
SW	SYSCALL	XOR	XORI			

System Call Instruction. Terminates the program.

Exceptions and Interrupts. No support.

2.2. Microarchitecture

A pipelined microarchitecture divides the "work" required to execute an instruction across multiple cycles. Each cycle corresponds to a *stage* within a pipeline. The major advantage of a pipelined microarchitecture is that it can execute multiple instructions in parallel: multiple instructions can be in the pipeline at the same time, albeit at different stages. If you need to brush up on pipelining and the associated dependence handling issues, please refer to the following lecture videos from Design of Digital Circuits (Spring 2018):

- Lecture 14: Pipelining (`https://youtu.be/f522l7Q-t7g`).
- Lecture 15: Pipelining Issues (`https://youtu.be/7XXgZIbBL1s`).

Pipeline Stages. The provided datapath implements the following *five-stage pipeline*:

Stage	Specification
1. IF	Instruction fetch
2. ID	Instruction decode and register file read
3. EX	Execution or memory address calculation
4. MEM	Memory access
5. WB	Writeback to register file

Handling Data Dependences. The provided five-stage pipeline implements *forwarding* to handle data dependencies. When a data dependence is detected, an earlier instruction is allowed to send data directly to a later instruction even *before* the data has been written back into the register file. Data is forwarded into the end of the decode stage. The implementation still stalls, but only when stalling cannot be prevented by forwarding data. Please, see the aforementioned lecture videos, to brush up on data dependence handling methods.

3. Your Task: Additions to the Pipelined MIPS Machine

In this lab, your task is to implement a data cache, as described in the rest of this section.

Data Cache. Your goal is to implement a **data** cache.[1] The data cache is accessed whenever a load or store instruction is in the memory stage. The specification of the data cache you will implement is listed below.

Organization. The data cache is **direct mapped** and has a **parameterized** capacity with a **32-bit block size**. The cache is empty to begin with, and it has the default capacity of 8KB.

Hit and Miss Timing. When a load instruction hits in the data cache in the memory stage, the data is retrieved within the same cycle. When a store instruction hits in the data cache, the data is written at the end of the cycle. On the other hand, when either a load or a store misses in the data cache, the block must be read from main memory and installed into the appropriate cache set. The top module that links the MIPS processor and memory in **testbench.v** reflects the multi-cycle access latency of memory. The memory takes 4 cycles to read or write the data.

Table 1 shows the timing of a load instruction that misses in the cache. The core accesses the cache in cycle 1 with `cache_addr`, which results in a cache miss. Within the same cycle, the cache issues a memory request with the load address of 0x04000000 through `mem_addr`, which is an address port from the `mips_core` module to the `mips_mem` module. In cycle 5, the data is returned and installed into the cache. After the data has returned to the core, the load instruction can fetch it in the next cycle from the cache through `cache_data_out`.[2]

Cycles	1	2	3	4	5	6
`cache_addr`	0x04000000	0x04000000	0x04000000	0x04000000	0x04000000	0x04000000
`mem_addr`	0x04000000	x	x	x	x	x
`cache_hit`	0	0	0	0	0	1
`mem_data_out`	x	x	x	x	0xdeadbeef	1
`cache_data_out`	x	x	x	x	x	0xdeadbeef

Table 1. Timing of a load that misses in the cache.

Write Policies. The data cache uses the **write-back** policy. No data is returned from the cache on a store hit. On a store miss, the cache first reads the block from memory into the cache, and after the block is in the cache, it then performs the store operation into the block.

[1] We are not asking you to implement an instruction cache. You do not need to modify anything in the IF stage.

[2] The ports, `cache_addr` and `cache_data_out`, are not provided in the starter code. They are used in the handout for demonstration purposes.

Handling Dirty Block Evictions. When a dirty block is being evicted due to a cache conflict miss, it needs to be written back to memory. In this lab, we will simply write back the dirty data first, before we allocate a new block in the set. Table 2 shows the timing of a load instruction (with address 0x04000000) conflicting with a dirty block in the cache, thus evicting it. Assuming the dirty block's address is 0x04008000, the cache will start writing back the dirty data to the memory in the first cycle of the cache-miss operation. Note that `we` is asserted along with the `mem_data_in` in the first cycle. Your design will need to hold the `mem_addr`, which is the address of the dirty block being written back to memory, until the *fifth* cycle due to the four-cycle delay associated with the propagation of data to the memory. After the data is written back to memory in the fifth cycle, a memory read operation will start, in order to fetch the new block into the cache. The sixth cycle is exactly the same as the first cycle in Table 1.

Cycles	1	2	3	4	5	6
`cache_addr`	0x04000000	0x04000000	0x04000000	0x04000000	0x04000000	0x04000000
`mem_addr`	0x04008000	0x04008000	0x04008000	0x04008000	0x04008000	**0x04000000**
`cache_hit`	0	0	0	0	0	0
`we`	**'b1111**	0	0	0	0	0
`mem_data_in`	0xcafecafe	x	x	x	x	x
`cache_data_out`	x	x	x	x	x	x

Table 2. Timing of a dirty block eviction from the cache.

Interface. We provide the interface between the memory and the core. Note that we do *not* provide a cache interface, so you have the freedom of designing your own interface between the cache and the core.

Tests. To test the correctness of your core after adding the data cache, you are provided with four programs (`memtest0`, `memtest1`, `arithmetic`, and `addiu`). Each of these includes the assembly code (`*.s`) and a hexadecimal file for user text (`*.text.dat`). You will have to include the name of this file in `mips_mem.v`, line 222.

You are also encouraged to write your own programs. To this end, you are provided with the SPIM assembler and a Python script (`asm2hex`) that will allow you to generate machine code to load it into your MIPS machine as a user program, using this command:[3] `./asm2hex [INPUT_MIPS_PROGRAM]`, where `INPUT_MIPS_PROGRAM` is your MIPS assembly file (`*.s`). Upon running the command, the output file (`*.text.dat`) will be generated in the same directory as the input file.

We will test your MIPS machine with our own test cases.

4. Lab Resources

4.1. Source Code

The source code of the pipelined MIPS machine is provided in the Moodle of the Computer Architecture course (`https://moodle-app2.let.ethz.ch/mod/resource/view.php?id=273287`). `inputs` contains the four programs mentioned in the previous section. `assembler` contains SPIM and the script (`asm2hex`). In `src` you will find the Verilog code for the pipelined MIPS machine. The list of files in `src` is listed below. Please, treat all source files confidentially and do not share with anyone.

- `testbench.v`: The testbench.
- `mips_core.sv`: The MIPS standalone processor module.
- `mips_core_if.v`: IF stage.
- `mips_core_id.v`: ID stage.

[3] To execute this command, the SPIM assembler and `asm2hex` should be in the same folder.

- `mips_core_ex.v`: EX stage.
- `mips_core_mem.v`: MEM stage.
- `mips_core_wb.v`: WB stage.
- `mips_decode.v`: Decodes MIPS instructions.
- `mips_mem.v`: Dual-ported virtual word-addressed memory with 5 segments.
- `memory_controller.v`: The memory controller.
- `mips_alu.v`: ALU unit.
- `multiply_coprocessor.v`: The multiplier module.
- `Div32.v`: The divider module.
- `mips_control.v`: Control unit.
- `branch_unit.v`: Branch unit. Not used in this lab.
- `mips_register_file.v`: The register file.
- `reg_write_queue.v`: Register write queue.
- `regfile_3port.v`: An array of 3-ported registers. (A "file" is an array of registers.)
- `dff.v`: Multi-bit D flip-flop.
- `exception_unit.v`: The exception unit. Not used in this lab.
- `mips_parse.v`: Instruction parse.
- `mips_debug.v`: Verilog debugging.
- `internal_defines.vh`: Internal signal constants.
- `mips_defines.vh`: Numerical parameters of the MIPS processor
- `regfile_init.vh`: Register file initialization.
- `syscall_unit.v`: The system call unit: exercised only when the `syscall` instruction is invoked to terminate a program.

4.2. Software Tools

For this course, we use the software Vivado for programming and simulation. The computers in rooms HG E26.1 and 26.3 are already installed with the necessary software. If you wish to use your own computer, you can refer to the following instructions: `https://goo.gl/VUJ34J`

5. Getting Started & Tips

5.1. Getting Started

1. We recommend that you review the material on pipelining from the pipelining and dependence handling lecture videos for Design of Digital Circuits (Spring 2018):
   - Lecture 14: Pipelining (`https://youtu.be/f522l7Q-t7g`).
   - Lecture 15: Pipelining Issues (`https://youtu.be/7XXgZIbBLls`).
2. **Please do not distribute the provided MIPS pipeline and program files. These are for exclusive individual use of each student of the Computer Architecture course. Distribution and sharing violates the copyright of the software provided to you.**
3. Create a new project in Vivado, and include all the provided files in it.

5.2. Tips

- **Read this handout in detail.**
- **If needed, please ask questions to the TAs using the online Q&A forum in Moodle.** (`https://moodle-app2.let.ethz.ch/mod/forum/view.php?id=273285`.)
- When you encounter a technical problem, please first read the logs/reports generated by the software tools. A search on the web can usually solve many of tool related debugging issues, and error messages.
- Your Verilog code will require many wires. Please adopt a consistent scheme for naming them. **You will review your code with us. So please make sure your code is readable by a third person.**
- The system call instruction should terminate the program only after all other preceding instructions have completed execution.
- Make sure your Verilog implementation is **synthesizable**.

6. Submission

Use the corresponding assignment in Moodle (`https://moodle-app2.let.ethz.ch/mod/assign/view.php?id=273292`). You should submit all the files needed to compile and simulate your code in a single tarball (with the name `lab1_`*YourSurname*`_`*YourName*`.tar.gz`). Please, include comments to explain what you have done, in the Verilog files.

7. Extra Credit

We will offer up to 100% additional credit for this lab for exploring different design aspects of the cache. Please, make sure you attempt optimizations only after you get your cache design (Section 3) functionally correct and testing it well.

1. **Critical path:** We will hold a performance competition. Among all implementations that are correct, the "top"[4] students that have the lowest critical path will receive up to 25% additional credit *for this lab.*
2. **Four-way set-associate cache and replacement policy:** You will first need to design and implement a four-way set-associate cache. On top of your cache, you will explore various cache replacement and/or insertion policies. The *cache replacement policy* specifies which cache block in a set is replaced when a new block is inserted into the cache. The *cache insertion policy* specifies where in the list of blocks the new block is placed. You can, for example, implement a replacement policy that evicts (replaces) the least-recently-used block, and an insertion policy that places new blocks at the most-recently-used position. However, other replacement and insertion policies have been studied, and some have been shown to achieve significantly better performance (fewer cache misses) for certain access patterns [1, 2]. You should experiment with a variety of test programs and optimize the cache replacement/insertion policy. Among all implementations that are correct, the top students that have the fastest execution time for an undisclosed set of test inputs will receive up to 40% additional credit for this lab. We will likely give significant extra credit to the working implementations as well.
3. **Any other cache improvements you would like to explore:** We covered a large number of cache optimizations in Lecture 2. You are free to implement any optimization you would like, e.g., a victim cache [3] or hash-based indexing [4], with the goal of improving the performance of you basic cache design. Since this part is open-ended, the instructor reserves the amount of

[4] The instructor reserves all rights for the precise definition of the word "top".

extra credit that can be obtained but 35% extra credit is possible, depending on the difficulty of the optimization and the goodness of the resulting design and implementation.

Please write a clear and detailed-enough report (`report.pdf`) that describes 1) how you optimize the critical path, 2) your findings on cache replacement/insertion policies, and 3) your findings on the other cache improvements you have implemented. Your report does not need to be more than four pages. Please also submit the version of your simulator that implements the extra credit parts you have completed.

References

[1] M.K. Qureshi et al. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[2] V. Seshadri et al. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *PACT*, 2012.

[3] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA*, 1990.

[4] O. Mutlu. Computer Architecture. Lecture 3: Cache management and memory parallelism, Fall 2017.