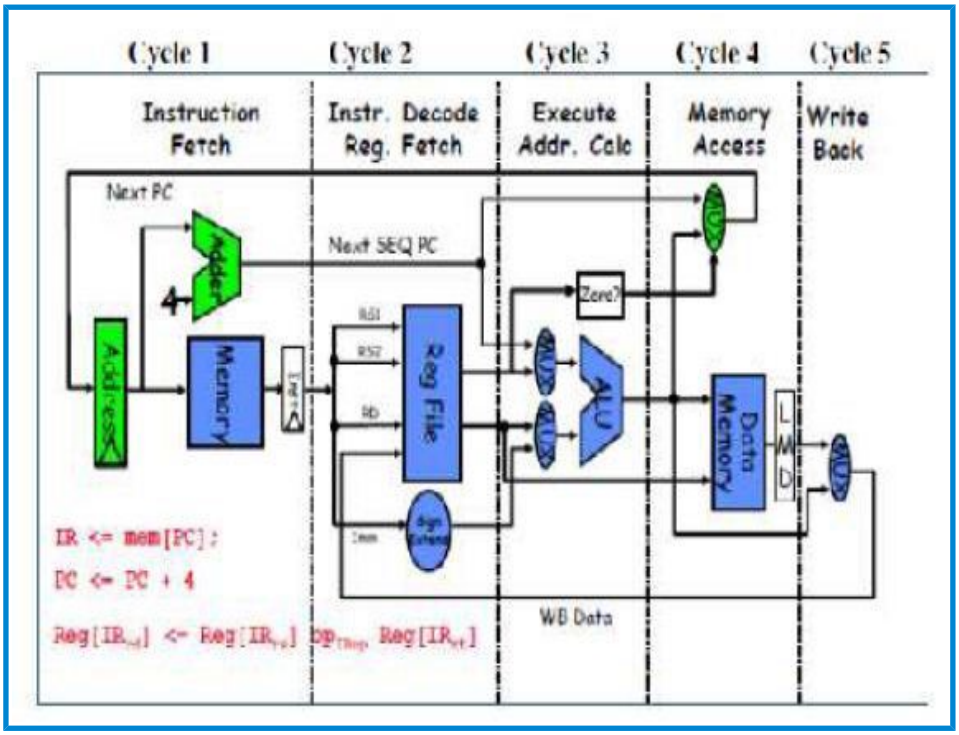
EENG 467

Project 1 Manual



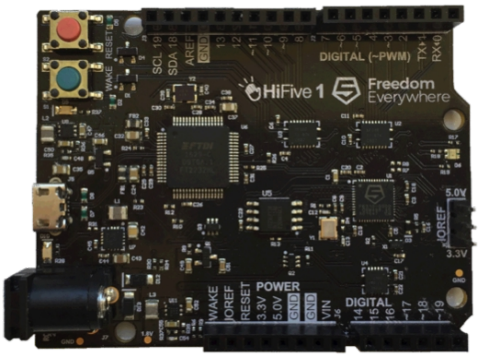
Part B: Processor Pipeline Implementation

Fall 2017

Prof. Jakub Szefer

1. Project Goals

The goal of the project is to implement in Verilog a simple in-order pipelined processor similar to the E31 RISC-V Core used on the SiFive's HiFive1[1] development board, including split L1 instruction and data caches.



The HiFive1 is an Arduino-Compatible development kit featuring the Freedom E310, the industry's first commercially available RISC-V SoC. More information is available at https://www.sifive.com/

1.1 Cache and Processor Pipeline Project Parts

The first part of the project is to implement and simulate split instruction and data caches. A testbench will be used to provide input addresses (and data if it is a write) to the cache, which then needs to interact with a memory module to store or fetch data if needed. Cache configuration and details are in Section 2.

The second part of the project is to implement a 5-stage pipeline which can execute the 32-bit RV32I ISA instructions. Details are in Section 3.

1.2 Evaluation

Two parts of the project will be evaluated together at the end of the project. For each part, testbenches will be used to test functionality of the corresponding part. For part A, testbenches will be used to provide inputs to the caches and observe the cache behavior. For part B, a test program (stored in the instruction memory) will be used in the simulations and the resulting contents of the data memory will be analyzed.

Projects should compile and be able to run testbenches (provided and student designed). The performance (in terms of cycles used for data access or instruction execution) will be used to evaluate each design. For part A, cache behavior needs to be correct and as few cycles as possible taken to complete different memory request. For part B, executed instructions need to yield correct instructions, and CPI will be used to rank the designs. If not all required RV32I instructions are implemented, number of correctly implemented instructions will be considered.

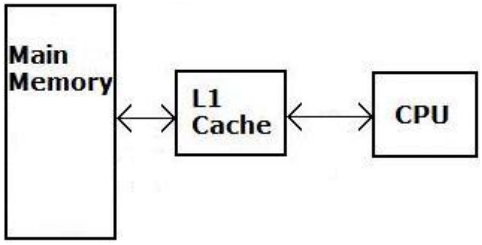
1 Image: http://ieeexplore.ieee.org/document/7097722/

2. Part A: Caches

Two caches should be implemented:

- 16 KB Instruction Cache, 2-way set associative, 1 word block size, 32-bit words
- 32 KB Data Cache, 4-way set associative, 2 word block size, 32-bit words

All caches should be inclusive, write-back and write-allocate caches. Assume each cache interacts with only the CPU and Memory. Also, each cache interacts with different memory (split instruction and data memories) so there is no need to consider how to multiplex one memory between two caches. Also, assume memory data width matches the block size (data and instruction memories will have different data widths).



2.1 Data Cache

The data cache is a 32 KB Data Cache, 4-way set associative, 2 word block size, with 32-bit words. When the CPU makes a request, it provides the byte address of the data to be read or written, 32-bit data (if it's a write) and a rden or wren signal depending if it's a read or write. The next clock cycle, the cache either responds by setting hit_miss signal to 1 (meaning it's a hit) and data is available from q port. If it's a miss, hit_miss is 0 as long as the cache is handling the miss, once data is available, it changes to 1. Thus the best the CPU can do is to have 2 CPI for loads or stores: first cycle you provide the address (and data) and next cycle you get a hit. If it's a miss, CPI will be more than 2.

On a miss, the cache has to find a victim data to evict to the memory, and communicate with the memory via the m* ports (e.g. mdout is data going from cache to memory). The memory responds in one cycle in the sample code! First cycle you provide address (and data) and next cycle you can read the data from memory or the data has been written. Each read or write is 1 cycle and there is no acknowledgement signal.

All data widths related to the cache and CPU are 32 bits. For memory, it depends on the cache block size, for data cache, the data memory width is 64 bits.

Note that data memory addresses are word address. Each word is 64 bits.

2.2 Instruction Cache

The instruction cache is 16 KB Instruction Cache, 2-way set associative, 1 word block size, with 32-bit words. Operation is the same as the data cache, only difference is the data width for data to/from memory which is also 32 bits (as instruction cache block size is 32 bits).

2.3 Simulation and Testing

To test the design, you need to write two separate testbenches, one for each cache. Sample template for the testbenches is provided. In the testbench, you will simulate requests from the CPU (to the cache). Based on each input the cache will either return a hit or will miss and require some communication with the memory.

3. Part B: Processor Pipeline

For part B you should implement a 5-stage processor pipeline which is able to execute basic RV32I instructions. The instruction fetch stage should read instructions directly from instruction memory (see sample testbench file), and the memory stage should read/write directly to the data memory (see sample testbench file again). Thus, there are no caches, and all data is available in the next clock cycle. I.e. give address before rising clock edge, and data becomes available after the clock edge.

The specific 32-bit instructions to implement are listed below (see the “greencard” in the textbook for instruction formats, etc.):

load word (lw)
store word (sw)
add (add)
add immediate (addi)
subtract (sub)
set < (slt)
set < immediate (slti)
branch equal (beq)
branch less than (blt)
branch greater than or equal (bge)

You should also implement pipeline stalling when a hazard is detected (i.e. don’t fetch more instructions if current instruction needs data that is not yet written back to the register file). You don’t have to implement data forwarding.

4. Report Format + Code Submission

Your code should be uploaded to git as with homework assignments. In addition, write 2 pages, single spaced, report for each part A and B (total of 4 pages). In the reports for part A and B, provide sections on:

- Overview of how you design works, especially timing (e.g. how many cycles it takes for cache hit, cache miss, etc.) and functionality (how do you deal with stalling pipeline, etc.)
- How you tested the design, provide any custom test benches you have used
- Any bugs or error that you uncovered and how you fixed them
- If you have worked with anybody (in case there is very similar code among multiple people, it is okay to work with others, but each project is individual)
- Make sure to put your name and date on the report
- submit the reports as report_part_A.pdf and report_part_B.pdf

5. Updated Timeline & Logistics

The two parts of the project will be graded together, but will be implemented separately (i.e. caches will not be used for the processor pipeline, processor pipeline will assume you can read and write directly to the two memories – this will reduce complexity and not require integrating the two parts).

Updated timeline:

Dec. 18 (in 2 weeks) cache implementation, pipeline implementation, and final report due

Dec. 19 (in 2 weeks + 1 day) final exam