EE 577a VLSI System Design Spring 2020

Score:___/100

Final Project
Pierluigi Nuzzo
TA: Yinghua Hu, Subhajit Dutta Chowdhury

Design of a General-Purpose Microprocessor Using Software and Hardware Components

Assigned: Monday, March 16th, 2020
Due: Phase I: Friday, April 3rd, 2020 at 11:59pm
Phase II: Friday, April 17th, 2020 at 11:59pm
Phase III: Friday, May 8th, 2020 at 11:59pm
No late submissions are allowed.

Notes:

- The final project will be based on teamwork of up to 3 students. Team members should be chosen before Phase I is started and kept through all the three phases.
- Collaboration across teams is NOT allowed. This means that any discussions should be limited to your team members, your TA and the instructor.
- You need to read this document, watch the related discussion videos, and monitor the lectures carefully; pay attention to the project requirements before starting the implementation.
- Phase I includes schematic-level design as well as Perl/Python scripting, Phase II includes the optimization for power, delay, and area, and Phase III includes the layout.
- You are welcome to use any components from your previous labs (only yours or your teammates'). However, feel free to redesign any parts, e.g., using full-custom techniques.
- Your report must be one PDF file including all the documents. Other file formats will not be graded.
- Each team only needs to submit one report per phase.
- Each report should include a clear statement of the responsibility and contribution of each team member.
- There will be a project demo right before or soon after the due date of the report for Phase 3. All team members should be present and ready to answer questions by the TAs.
- If it becomes clear that not all members of the team contributed roughly the same amount to the project individual grades may differ.

I. Introduction

The goal is to design an efficient general-purpose CPU that supports simple instructions such as Add, Bitwise operations, Store Word, and Load Word. A simple structure of the CPU contains Decoding Logic, Register File, Execution Units, Memory, and other surrounding circuitries. You are encouraged to look at some basics of CPU design using online resources such as http://en.wikipedia.org/wiki/Classic_RISC_pipeline.

In our final project, you have the opportunity to try and implement a simple CPU with the knowledge you gained in EE577A, including your Datapath and SRAM designs as well as your (Python or Perl) scripting experience from the EE577A labs, and aim at optimizing for area, delay, and power.

Note that the CPU designed in this project may not be exactly the same as the ones you may have seen in your computer architecture courses. The details of the implementation are made flexible, so that you have some level of freedom in the design. This also means that different teams may come up with different designs, but all are supposed to execute the same set of instructions. Section II (b) presents a sample structure that gives you some ideas on how to proceed.

In addition to providing correct functionality, you need to take at least one power optimization measure that you learn in class. Please check the requirements in Section III and IV. Phase II is dedicated to optimization. Poor designs will be subject to point deduction, so please be prepared to spare some time on optimization given the time limit.

II. Implementation of a Pipelined CPU

a) Instruction Fetching and Decoding (Front-end Perl/Python Scripting)

Your design is supposed to execute the following 17 instructions:

Instruction format	Description
STOREI {bl} xxH #xxxx {#xxxx}	Store xxxx into word xxH
STORE xxH $R	Store data from register $R into word xxH
LOADI $R #xxxx	Load xxxx into register $R
LOAD $R xxH	Load word xxH into register $R
AND $x $y $z	Bitwise AND value in register y with value in register z, save the result in register x
ANDI $x $y #xxxx	Bitwise AND value in register y with value xxxx, save the result in register x
OR $x $y $z	Bitwise OR value in register y with value in register z, save the result in register x
ORI $x $y #xxxx	Bitwise OR value in register y with value xxxx, save the result in register x
NOP	No operation
ADD $x $y $z	ADD value in register y with value in register z, save the result in register x
ADDI $x $y #xxxx	ADD value in register y with value xxxx, save the result in register x
MUL $x $y $z	MUL value in register y (lower 5 bits) with value in register z (lower 5 bits), save the result in register x
MULI $x $y #xx	MUL value in register y (lower 5 bits) with value xx, save the result in register x
MIN $x $y $z	Save the minimum value in register y and value in register z to register x
MINI $x $y #xxxx	Save the minimum value in register y and value xxxx to register x
SFL $x $y #xxxx	Left shift the value in register y by xxxx bits and store the result in register x
SFR $x $y #xxxx	Right shift the value in register y by xxxx bits and store the result in register x

Specifications:

Address

- The address can either be binary or hexadecimal, e.g., xxxxxB or xxH.

- The address can be in the range from 00H to 1FH (or 00000B – 11111B).
- Register file address can be simply assigned as integer number, since there are only 8 16-bits registers, e.g., $0, $1, …, $7.
- For the register files, assume $0 is reserved only for output checking purposes, and no data should be put into it during normal operation (this is further explained in the "Pipelining" specifications).

Supported Arithmetic Operation

- All arithmetic operations in this part should be signed operations.
- **LOAD/LOADI/STORE/STOREI**.
- **16bit AND/ANDI /OR/ORI**. We use the little endian system, e.g., "ANDI $1 $2 #7C4DH" refers to the bitwise AND function of two 16-bit numbers: one is the word stored in register $2 (assume the stored value is 0000H), and the other is 7C4DH; thus, the result 0000H is stored back to register $1.
- **16bit signed ADD/ADDI.** We use the little endian system, e.g., "ADDI $1 $2 #1A2BH" refers to adding two signed 16-bit numbers: one is the word stored in register $2 (assume this is 0000H), and the other is 1A2BH; thus, the result 1A2BH is stored back to register $1.
- **5bit signed MUL/MULI.** We use the little endian system, e.g., "MULI $1 $2 #1FH" refers to multiplying two signed 5-bit numbers: one is the data stored in the lower 5-bit in register $2 (assume this is 00H), and the other is 1FH; thus, the 10-bit result 000H is stored back to register $1. Notice that you need to add zeros or ones to the 10-bit number to make it 16-bit. Whether adding zeros or ones depends on the sign of the 10-bit number.
- **16bit signed MIN/MINI.** A comparator should be designed to compare two 16-bit signed numbers and the minimum value should be stored back to the destination register.
- **16bit SFL/SFR.** Left/right (arithmetic) shift the value in register y by #xxxx bits and store the result in register x. #xxxx is between 1 and 15.

Burst Operation

- Only one instruction STOREI supports burst operation. If bl is not mentioned, then bl=1. For bl=2, addresses are as follows: xxxx0B to xxxx1B, otherwise ignore this invalid command and the error should be reported (further explained in the "Error Report" specification).

Pipelining

- You are required to implement a 5-stage pipeline as described in the following parts.
- For simplicity, we ONLY consider the data dependencies in either register file or memory, thus you have to insert one or several NOP instruction(s) into your actual operation sequence, if necessary, to avoid dependency.
- To check correctness, we can use the LOAD $0 command to get data from the memory, and then examine the output data bus.
- Since initially there is no data in the RF/MEM, we should execute a few STOREI to save data into the memory, then execute a few LOAD to get data into RF, or execute a few LOADI to save data into RF.
- Control signals: to implement the pipelined CPU, you need to register the control signals along with data in the pipeline. These control signals include (but are not limited to) instruction type control, address bits, etc.
- Design optimization ideas: the clock cycle of the design is determined by the worst path in the whole pipeline. Therefore, if you find that one path P has significantly smaller delay than the worst path, you may size all the gates on P to minimum size to reduce area.

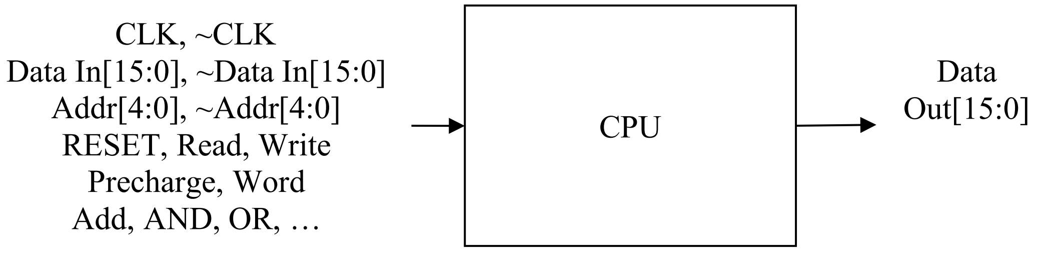
Out-of-Order Execution of the MUL/MULI Commands

- To improve your performance, you are **required** to design a separate pipelined multiplier and use out-of-order execution for MUL/MULI commands.
- You will also need to handle consecutive MUL/MULI commands in the test.
- You do not need to consider address dependency.

Error Report

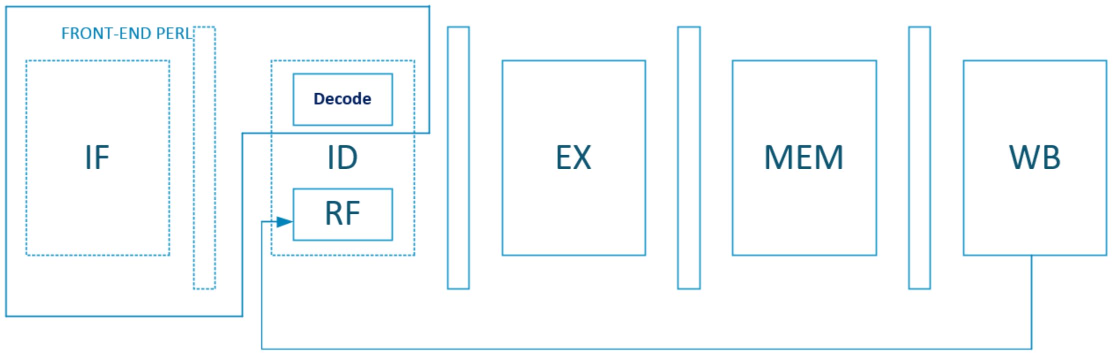
- For STOREI, burst length can only be 1 or 2 or 4. If it assumes another value, Perl/Python should ignore this command and show an error information on the screen:
  "Error000: Command ... has invalid burst length."
- If the starting address for bl=2 is not xxxx0B or the starting address for bl=4 is not xxx00B, then ignore this command. Report the error:
  "Error001: Command ... is not aligned properly."

The aforementioned assembly-like instructions should be given to the instruction decode stage of the CPU and the instruction decode stage is supposed to generate the binary code, i.e., inputs that would be needed to execute the instructions using your data-path elements such as the adder or multiplier. The following figure shows the block level view of your design and the required inputs and outputs.



To simplify the hardware design part of this project, Instruction Fetch and part of the Decode block will be implemented in software using Perl/Python, hence you need to program a Perl/Python script to convert the given benchmarks (set of instructions) to Cadence input vector files, which feed the primary inputs in the above graph.

b) Block Diagram



Block diagram of the general-purpose microprocessor. (This is ONLY a sample and you DO NOT have to necessarily implement your final project in this way.)

There are five stages in the pipeline structure of this project, namely, IF, ID, EX, MEM, and WB. The front-end Perl/Python script will function as IF and decode logic in the ID stage, whereas the rest of the stages and *Register File* (RF) will be implemented in hardware in Cadence.

Discussion– Please discuss these questions within your team before starting the project:
1. In the ID stage, what are the inputs and outputs of the RF? Should the RF be SRAM-based design or registers?
2. In the EX stage, compare the delay of adder and multiplier, which one is faster? Do we need to further divide the slower one into deeper pipelines? How many stages do we need if we decide to divide adder or multiplier?

3. In the EX stage, do you need to add some buffers to drive the multiplier?
4. In the EX stage, if the adder or multiplier is not used, can we apply some power-saving techniques?
5. In the EX stage, can we do Out-Of-Order (OoO) operation? What if there are data dependencies?
6. In the MEM stage, is the SRAM the bottleneck? Will it help if we allow for the MEM stage 2 cycles to operate? Is it necessary to keep the MEM stage or is it possible to combine the MEM stage with the last stage of the multiplier?
7. When should we insert bubbles? Does the bubble insertion depend on your schematic?

III. Dynamic Logic

Dynamic logic is supposed to be used as part of your design. You are not restricted to implement any specific part in dynamic logic; you are allowed to select any part of any block. You are also allowed to use any of the circuit families you learned about before this course, in addition to the domino logic discussed in class. Also, follow the lecture suggestions on how to apply logical effort for delay optimization of the dynamic parts as well. Using Dynamic logic is optional.

IV. Power Optimization

You are highly encouraged to apply as many of the power optimization techniques that are discussed during the lectures. For example, clock-gating can be performed for the multiplier and the adder as the computations of these two modules are not required for every cycle. In addition, pipeline stages that are performing a NOP can be clock gated. You are encouraged to further employ clock-gating to other modules if possible. For example, you can consider data-gating at the inputs to combinational blocks that would otherwise have access switching. The effect of power optimization must be demonstrated in the report, for example, by showing the idle inputs/outputs of a clock-gated module during power saving cycles, and/or power measurement during the execution of a sequence of instructions for both cases of gating on and off, i.e., by measuring the power from simulation of the schematic or extracted views with and w/o this feature. Note the fact that power and timing optimizations come with area costs and these should be quantified. Therefore, check whether any technique (e.g., power gating) would be helpful enough considering the technology we are using in our project and the need to also minimize the area.

V. DFF Optimization

DFF arrays are a large part of this project. Please follow the lecture slides and try TGFFs, pulse triggered FFs, or other FF structures in Virtuoso. You can also use latches and leverage time borrowing. Choose the DFF/latch structure that is the best fit for your final project and state the reasons. Explain your effort on this part in the report.

VI. Clock Tree Design

Your design should include a clock tree. Assume a single clock source that can drive a 10x-size inverter. If you choose a multi-phase clock network, derive the other phases from the single clock source. Make sure you consider wire delay and clock skew optimization techniques in your design, starting from the schematic.

VII. Automated Result Verification (Back-end Perl/Python Scripting)

The execution results of an instruction sequence must be verified by using a script. This script will first pre-compute the expected values that should be stored in SRAM after the instruction sequence is executed. After circuit simulation by using Spectre, the script will then compare the pre-computed values with the values in SRAM.
One way to obtain the simulated result in SRAM is described as follows. Use a sequence of *load* instructions to read the SRAM after the execution of the *original* instruction sequence; the values read from SRAM can be then dumped to text-based formats. This script can but need not be separated from the instruction fetch/decode script.
You need to write a Perl/Python script to output the correct/golden values stored in the register files and memory when all the operations are executed correctly. Then, use this golden results to compare with the simulation.

VIII. Other Requirements:

a) Simulation requirement

You are not allowed to use Perl/Python to do any data calculation of the processor and then put the value right into the RF or SRAM. You CANNOT directly initiate the data in the MEM stage! We will randomly choose one command during the demo session and you should be able to identify the corresponding part of the waveform. Any violation will be treated as cheating.
Layout metal layers can be used up to Metal 6. All the input slews are 5ps.

b) Performance evaluation metric

Delay*Area*Power
Area definition: The smallest square size that contains your whole project layout.
Delay definition: The duration from the start of the first instruction to obtaining the result of the last instruction.
Power definition: The average total power, which is VDD multiplied by the average current for the duration of the applied test vectors during testing of your design.

c) Submitting files

Phase 1:
- The PDF report shows the Perl/Python scripting results, the schematic design of the microprocessor and correct functionality. You also need to submit your front-end and back-end Perl/Python script(s) along with your report.
- Simulation results of the schematic.

Phase 2:
- The PDF report shows the final version of the schematic of your data-path after optimization, power optimization. Perl/Python scripts and corresponding vector files. Explain all the optimization measures applied in the design.
- Simulation results after the optimization.

Phase 3:
- Submit the same materials required by Phase 2 and any new optimizations you made so far.
- The report should show the data-path layout and the evaluation metric.
- Screenshots of the DRC and LVS results.
- Simulation results of the layout.

d) Report requirement

Explain each part in detail. You should clearly state what parts are NOT working. Submitting any nonworking schematic or layout parts without informing us would be considered cheating.

e) Grading standard

80% for functionally correct designs, 5% for DFF optimization, 5% for power optimization measures, 10% for the performance (Area×Delay×Power) product.
In case your project is not fully functional, you may still receive partial credit for your partial work, however you are responsible to present your work and establish the necessary information that supports the correctness of each part. For instance, for a design with the multiplier as the only non-working part, the group is responsible to prove that the contents stored in SRAM are all correct except the ones that are executed or contaminated by the non-working multiplier. Partial (but very limited) credit will be given if the group can only prove that each module works individually.
For the 10% performance credit, any design that is better than average gets 10, any design that is worse than average but is better than 10 times average gets 5, any design that is worse than 10 times average gets 0. The top designs will get extra credit based on the performance.
In particular, the top 5 designs will receive up to 10% to 40% extra credit based on the performance. Performance credit and extra credit only consider the complete designs, including layout with correct function.