EE 577a Final Project
SP 2017 Nazarian **Score:___/100**

Design of a General Purpose Microprocessor using Software and Hardware Components

Assigned: Friday, March 10th, 2017
Due: Phase I: Friday, March 31st, 2017 at 11:59pm
Phase II: Wednesday, April 19th, 2017 at 11:59pm
Phase III: Monday, May 1st, 2017 at 11:59pm

No late submission for Phase III. Late submissions will be accepted for Phase I and II only during the first two days after their deadline with a maximum of 15% penalty per day. For each day, submissions between 12 and 1am: 2%, 1 and 2am: 4%, 2 and 3am: 8% and after 3am: 15%. There is no late submission possible for Phase III. Project demo time will be announced later.

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 any discussions should be limited to your team members, your TAs and the instructors. Refer to the first lecture for the AI policies of USC and this course and in case of any questions or doubts contact the instructor.
- You need to read this document and watch related discussion videos and also monitor the lectures carefully, especially pay attention to the project requirements, before starting the implementation.
- Phase I includes schematic as well as Perl/Python scripting, Phase II is the optimization for Power, Delay and Area, and Phase III is the layout.
- You are welcome to use any of 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.

I. Introduction

The goals is to design an efficient general purpose Multi-Cycle CPU that supports simple Instructions such as Add, Bitwise operation, Store Word, and Load Word. A simple structure of CPU contains Decoding Logic, Register File, Execution Units, Memory, and other surrounding circuitries. If you have never taken any computer architecture course, you are encouraged to look at some basics of CPU design using online resources such as:
http://en.wikipedia.org/wiki/Classic_RISC_pipeline

Here in our final project, it is your opportunity to try to implement a simple general purpose CPU with the knowledge you gained in EE577A, including your datapath and SRAM designs as well as your (Python/Perl) scripting experience of the EE577A labs, and aim at optimizing for Area, Delay, and Power.

Note that the CPU designed in this final project may not be exactly the same as the ones you may have seen your computer architecture courses. The details of implementation are however made flexible, so that you have some level of freedom in designing your work. This also means 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 will hopefully give you some ideas on how to proceed with your design.

Design evaluation for this project is based on Area×Delay×Power product. This means you need to practice optimizing your design to have desirable power, delay and area. Therefore one main task of EE577A is optimization for delay and power, in addition to functionality correctness you should implement at least one of your circuit blocks using dynamic logic. Also you need to take at least one power optimization measure you would learn from 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 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 be 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, let's assume $0 is reserved only for output check purpose, and don't put any data into it during normal operation. (further explained in "Pipelining" specification)

Supported Arithmetic Operation

- **LOAD/LOADI/STORE/STOREI**.
- **16bit AND/ANDI /OR/ORI**. We use little endian system, i.e., "ANDI $1 $2 #7C4DH" means performing 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 little endian system, e.g., "ADDI $1 $2 #1A2BH" means adding two signed 16-bit numbers: one is the word stored in register $2 (assume it's 0000H), and the other is 1A2BH, thus the result 1A2BH is stored back to register $1.
- **5bit signed MUL/MULI.** We use little endian system, e.g., "MULI $1 $2 #1FH" means multiplying two signed 5-bit numbers: one is the data stored in the lower 5-bit in register $2 (assume it's 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 to the 10-bit number to make it 16-bit.
- **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 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 "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 LOAD $0 command to get data from 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 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 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 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 don't need to consider address dependency.

Error Report

- For STOREI, burst length can only be 1 or 2 or 4. If it's another value, Perl/Python should ignore this command and show an error information on the screen:

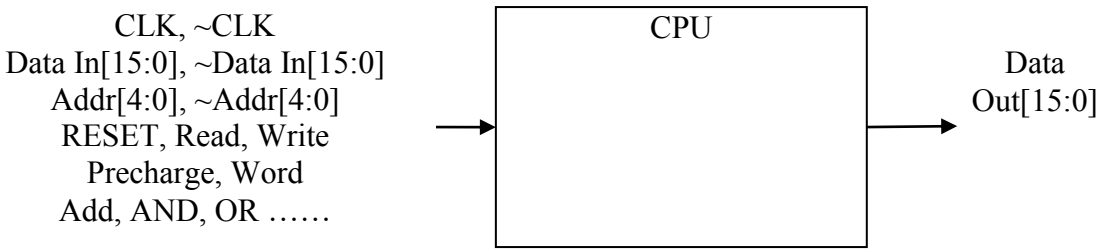
"Error000: Command … has invalid burst length."

... is the error command with 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 afore-mentioned assembly-like instructions should be given to instruction decode stage of 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 will be implemented in software using Perl/Python, hence you need to program a Perl/Python script to convert given benchmarks (set of instructions) to Cadence input vector files, which feed the primary inputs in the above graph.

b) Block Diagram

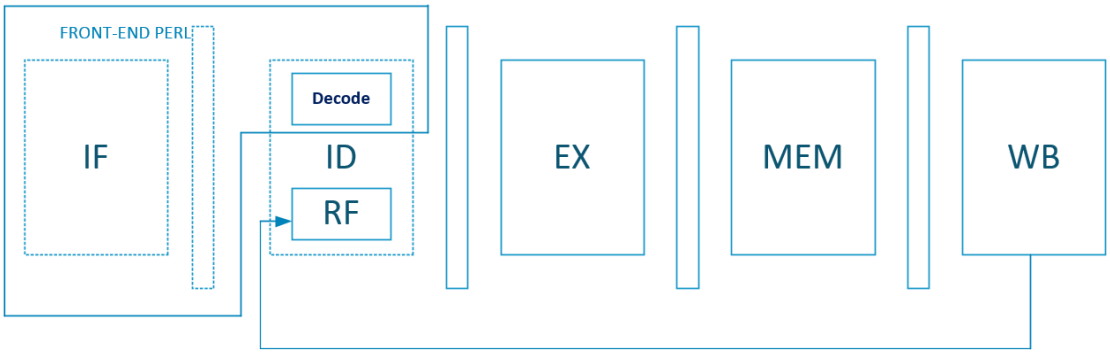


Figure. Block diagram of the general purpose microprocessor (this is ONLY a sample and you DO NOT necessarily implement your final project in this way).

There are five stages in the pipeline structure of this project, namely, IF, ID, EX, MEM and WB. Our 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 as hardware in cadence.

Discussion about the block diagram – Please discuss these questions within your team before starting the project:

1. In ID stage, what are the inputs and outputs of the RF? Should the RF be SRAM-based design or registers?
2. In 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 EX stage, do you need to add some buffers to drive the multiplier?
4. In EX stage, if adder or multiplier is not used, can we apply some power-saving techniques?
5. In EX stage, can we do Out-Of-Order (OoO) operation? What if there are data dependencies?

6. In MEM stage, is the SRAM the bottleneck? Will it help if we allow 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 depends on your schematic?

III. Dynamic Logic

Dynamic Logic is supposed to be used as part of your design. You are not restricted in implementing any specific part in dynamic logic, meaning you are allowed to select any part of any block to be done in dynamic logic. You are also allowed to use any of the circuit families in pre-577A slide in addition to the domino logic discussed in our class. Also follow the lecture suggestion on how to apply logical effort for delay optimization of the dynamic parts as well.

IV. Power Optimization

You are highly encouraged to apply as many of the power optimization techniques that are discussed during 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. You are encouraged to further employ clock-gating to other modules if possible. The effect of power optimization must be demonstrated in the report, such as 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. Look at the fact that power and timing optimizations come with area costs. Therefore check with any technique (e.g., power gating) would be helpful enough considering the technology we are using in our project and also the Area×Delay×Power product.

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. Choose the DFF structure that is the best fit for your final project and state the reasons. Explain your effort on this part in the report.

VI. 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 another 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.

VII. 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 use store buffer in the MEM stage! We will randomly choose one command during demo 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 10ps.

Parasitic capacitance must be extracted.

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 first instruction to obtaining the result of 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:

- 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.

Phase 2:

- The PDF report shows final version of the schematic of your data-path after optimization, dynamic logic block, power optimization. Perl/Python scripts and corresponding vector files. Explain all the optimization measure applied in the design.

Phase 3:

- The 2nd part of the PDF report shows final version of the schematic of your data-path, Perl/Python scripts and corresponding vector files.
- The 3nd part of the PDF report shows the Data-path layout and the evaluation metric.

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

75% for functionally correct designs, 5% for dynamic logic, 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 each module works individually.
For the 10% performance: 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.
Top designs will get extra credit based on the performance.
Top 5 design groups 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.