ECE 429, Spring 2014
Final Project: Design and Synthesis of Central Processing Units

Project:	*5 weeks 03/31 – 05/02*
Oral Evaluation:	*05/01*
Progress Report Due:	*10:00am 04/14 (Mon.) Chicago time*
Final Report Due:	*10:00am 05/05 (Mon.) Chicago time*

1 Design Objective

This project introduces VLSI design concepts including combinational and sequential circuit design, standard cell based design flow, and design validation and verification through construction of an 32-bit Central Processing Unit (CPU) in Verilog, to be synthesized using commercial EDA tools from Synopsys and Cadence Design Systems. The intent is to show, first, how to construct a nontrivial digital system that can perform any computation through combinational and sequential circuit design techniques; second, how to make design trade-offs, e.g. performance, cost, and design time, through architectural exploration; and third, how the EDA tools transform the design implementation from higher abstraction levels, e.g. Verilog, to lower abstraction levels, e.g. layout, through cell-based design flow, what the differences are among the implementations and how to verify their correctness at the different abstraction levels.

In this project, a reference 8-bit CPU is provided as the starting point available from Blackboard. After understand the functionality of the reference 8-bit CPU, you are required to extend it to a 32-bit CPU and to explore different architectures in order to make trade-offs among performance and cost. Finally, as a bonus problem, you are required to explore the possibility of adding a new functionality within a short design time budget. All the designs should be synthesized through the cell-based flow. A pre-defined performance constraint must be met after the synthesis and the correctness of the design at different abstraction levels should be verified.

This project should be done individually. Discussions are encouraged. However, all the writings, results, and screen shots should be by yourself. COPY without proper CITATION, and extensive COPY from other materials including but not limited to project instructions and textbooks, will be treated as PLAGIARISM and called for DISCIPLINARY ACTION.

$\boxed{\textbf{NEVER share your writings/screenshots with others}}$.

2 Understand the Design of an 8-bit CPU

2.1 Overview

The CPU design will start from the understanding of a reference 8-bit CPU as shown in Fig. 1. It is organized hierarchically to reduce the complexity. Each components will be discussed in detail in the following subsections.

This CPU, though not having the full functionality as the multicore processors in your PC, has many of the key features. It is driven by two external signal – a clock signal and a power signal. When the power is on, it follows a *program* consisting of *machine code*s, or *instruction*s, written by you and executes one instruction every clock cycle. It can perform various arithmetic and logic computations and store the results into a memory. It can jump to different locations in the program depending on the value of the memory so your program can branch and loop. It is ready to be extended once you understand its behavior.

The CPU is implemented as a synchronous circuit with edge-triggered flip-flops as the sequential elements. For the CPU to function correctly, it is very important to synchronize all the signals properly. You will be asked to synthesis the CPU and to investigate the critical path delay.

2.2 Arithmetic Logic Unit (ALU)

The *arithmetic logic unit (ALU)* is the component that performs computation in a processor. As the name of the design suggests, the ALU takes two 8-bit data `a` and `b` as input and computes the 8-bit result `s` as the output. The computation executed is specified by the 3-bit input `op`, which is listed in the following table.

`op[2:0]`	Output	Comments	Examples
000	`s=a<<b[2:0]`	shift-left-logical	`48=3<<4`
001	`s=a>>b[2:0]`	shift-right-logical	`15=255>>4`
010	`s=a+b`	addition	`5=3+2`
011	`s=a-b`	subtraction	`5=8-3`
100	`s=a&b`	bit-wise and	`2=3&2`
101	`s=~(a^b)`	bit-wise xnor	`254=~(3^2)`

Note that since `a` only contains 8 bits, both the shift-left-logical and shift-right-logical operations use only the least 3 bits from `b`.

2.3 Memory

The memory is the component that stores values in a processor. There are 16 *word*s, named `@0`, `@1`, ..., `@15`, in our memory implementation. As the name of the design suggests, each word can store a value of 8 bits. The value of each word is stored in 8 flip-flops so they can be updated utilizing additional combinational logics.

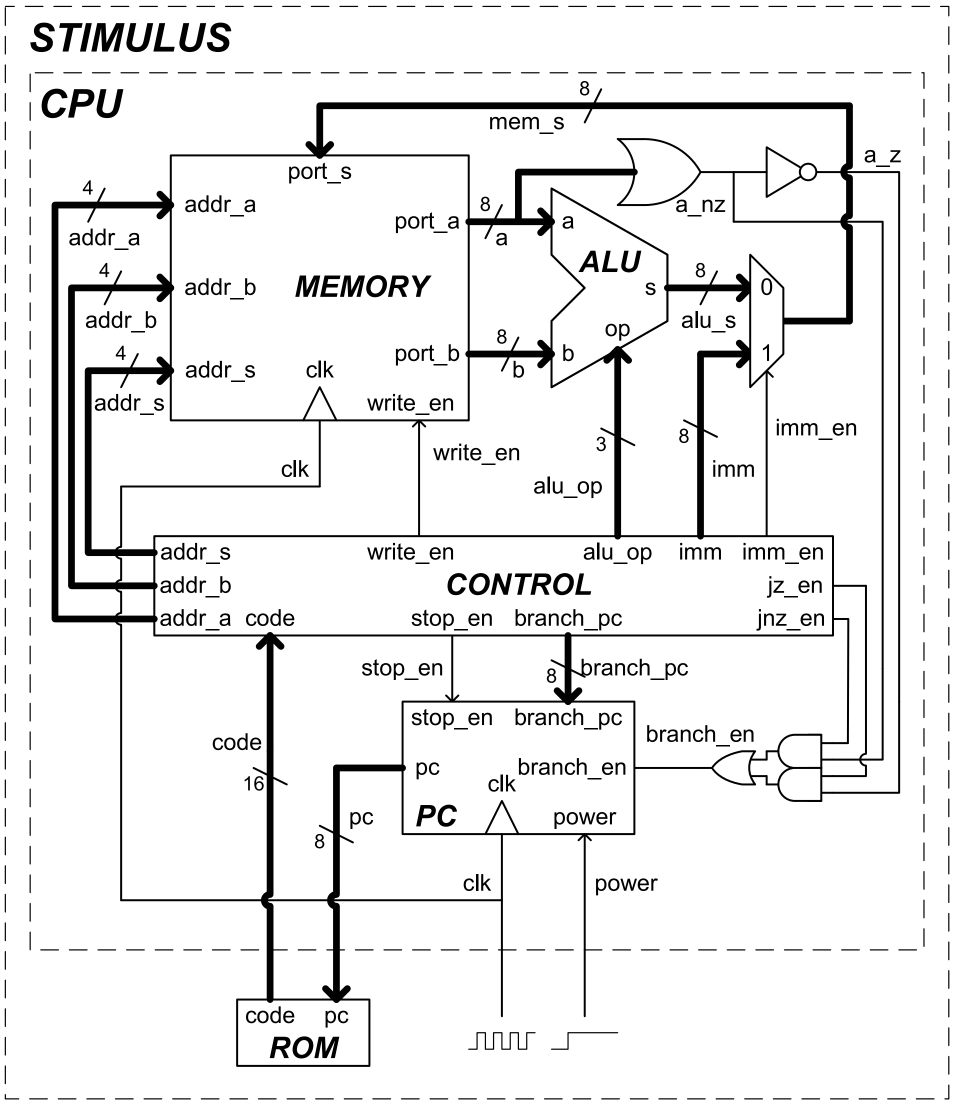

Figure 1: An 8-bit CPU with Test Bench

To refer to a specific word, a 4-bit *address* is used. The value of each word can be read out through two 8-bit outputs `port_a` and `port_b` With two 4-bit inputs `addr_a` and `addr_b` specifying the addresses of the words available on the two outputs respectively, i.e. if `addr_a=5` and `addr_b=9`, then `port_a=@5` and `port_b=@9`. Generally speaking, we need two 8-bit 16-to-1 mux's for such functionality. We implement such mux's by combining 4-to-16 decoders and transmission gates, i.e. tristates buffers.

It takes more effort to write the memory, i.e. to update the words. Since sometimes no word should be updated, we use an input `write_en` to indicate if any writing should happen. Moreover, to simplify the circuit we only allow one word, specified by the 4-bit input `addr_s`, to be updated per clock cycle. The new value is specified through the 8-bit input `port_s`. To synchronize the operations of CPU, writing to the words should only happen when the positive edge of the clock comes. Thus the inputs `write_en`, `addr_s`, and `port_s` should be ready before the positive edge of the clock.

When a word should not be updated, its value should not be lost. Therefore we need a feedback loop from the output of the flip-flops to their inputs.

2.4 Program Counter

As we mentioned before, our CPU follows a *program* consisting of *instruction*s. In each clock cycle, one instruction is executed. The instructions are numbered from 0 and at most 256 instructions are allowed in our design. An 8-bit *program counter*, or simply `pc`, is used to address the one that should be executed in the current clock cycle. To synchronize the operations of CPU, the `pc` is stored in 8 flip-flops and is updated when the positive edge of the clock comes to indicate the next instruction to be executed.

During the normal operation of the CPU, we increment `pc` by 1 every clock cycle such that the next instruction will be executed in the next clock cycle. However, it is possible that at some point, the program may require to branch into another executing path starting from an instruction other than the next one. To address such requirement, two inputs `branch_en` and `branch_pc` are introduced. The input `branch_en` specifies that the branch is necessary and the 8-bit input `branch_pc` indicates the address of the next instruction.

To specify when the executing starts and stops, we introduce two inputs `power` and `stop_en`. If `stop_en` is 1 when the positive edge of the clock comes, the `pc` will remain unchanged thereafter; otherwise, it will either become `pc+1` or `branch_pc` depending on `branch_en`. If `power` is 0 when the positive edge of the clock comes, the `pc` will point to the first instruction; otherwise, it will take a value from `pc`, `pc+1`, or `branch_pc` depending on `stop_en` and `branch_en`.

2.5 Instruction Set

Now, we are ready to study the instruction set of our CPU and to write our own programs. An *assembly language* is developed for your to understand the instructions.

Usually a program called *assembler* will convert each instruction of the program writing in the assembly language into a *machine code* that can be "understand" by the hardware. However, for you to understand the design of the CPU better, you need to do the conversion by yourself in the test bench.

Although the data path of the CPU is 8 bits wide, each instruction in machine code consists of 16 bits. The CPU supports 10 instructions divided into 3 groups.

The first group includes 6 ALU operations as listed in the following table.

ASM Code	Machine Code	Comments
`sll @s,@a,@b`	`0000 addr_s[3:0] addr_a[3:0] addr_b[3:0]`	`@s=@a<<@b[2:0]`
`srl @s,@a,@b`	`0001 addr_s[3:0] addr_a[3:0] addr_b[3:0]`	`@s=@a>>@b[2:0]`
`add @s,@a,@b`	`0010 addr_s[3:0] addr_a[3:0] addr_b[3:0]`	`@s=@a+@b`
`sub @s,@a,@b`	`0011 addr_s[3:0] addr_a[3:0] addr_b[3:0]`	`@s=@a-@b`
`and @s,@a,@b`	`0100 addr_s[3:0] addr_a[3:0] addr_b[3:0]`	`@s=@a&@b`
`xnor @s,@a,@b`	`0101 addr_s[3:0] addr_a[3:0] addr_b[3:0]`	`@s=~(@a^@b)`

Every instruction in this group reads two values `@a` and `@b` from the memory and stores the result back to the memory at `@s`.

The second group includes only one instruction in the following table.

ASM Code	Machine Code	Comments
`loadi @s,imm`	`1010 addr_s[3:0] imm[7:0]`	`@s=imm`

This instruction stores an 8-bit immediate value, as specified by the least 8 bits in the instruction, into the memory at `@s`. It provides a method to write your own data into the CPU.

The last group includes 3 branch instructions in the following table.

ASM Code	Machine Code	Comments
`jz @a,bpc`	`1000 bpc[7:4] addr_a[3:0] bpc[3:0]`	jump to `bpc` if `@a==0`
`jnz @a,bpc`	`1001 bpc[7:4] addr_a[3:0] bpc[3:0]`	jump to `bpc` if `@a!=0`
`stop`	`1111 xxxx xxxx xxxx`	stop the program

All these three instructions may change the progress of the program by updating the `pc` as indicated in the previous section.

As an example, let us consider a program that performs multiplication by multiple times of additions as follows.

```
0: loadi @1,30
1: loadi @2,5
2: loadi @3,0
3: loadi @4,1
4: jz @2,8
5: add @3,@3,@1
```

```
6: sub @2,@2,@4
7: jnz @2,5
8: stop
```

This program computes `@1*@2` and stores the result into `@3`. The corresponding program in machine code is as follows.

```
0: 1010 0001 0001 1110
1: 1010 0010 0000 0101
2: 1010 0011 0000 0000
3: 1010 0100 0000 0001
4: 1000 0000 0010 1000
5: 0010 0011 0011 0001
6: 0011 0010 0010 0100
7: 1001 0000 0010 0101
8: 1111 1111 1111 1111
```

2.6 Program Storage

As almost all the modern processors, we store our program outside the CPU. A module is provided in the test bench as the storage of the program. Given the program counter `pc` as the input, the module outputs `code` as the machine code of the program at the position referred by `pc`. We implement this module to read a file `code.hex` which stores the program externally in hexadecimal format. For the program in the previous section, the file looks like:

```
a11e
a205
a300
a401
8028
2331
3224
9025
ffff
```

You can edit this file with any text editor, e.g. notepad, to include your own program.

2.7 Control Unit

Given the instruction `code` in machine code, the *control* unit generate all the necessary signals to control the operation of the ALU, the memory, and the program counter `pc`. To be more specific, for the ALU instructions, it determines `alu_op`, `addr_s`, `addr_a`, and `addr_b`, sets `write_en` to 1, and sets `imm_en` to 0 such that the ALU output will be stored. For `loadi`, it determines `addr_s` and `imm`, sets both `write_en` and `imm_en` to 1 such that `imm` instead of the ALU output will be stored. For `jz` and `jnz`, it sets `jz_en` and `jnz_en` to 1 respectively, and generates the branching address `branch_pc`. For `stop`, it sets the `stop_en` signal.

2.8 CPU and Test Bench

Combining all the above components together with a few logics, we obtain our CPU design as in the file `cpu8.v`. It takes `clk` and `power` as the input control signals. It also outputs the current program counter `pc` and expects to read the current instruction `code` in machine code from the outside. A few more signals are outputted for you to understand the design, including the controlling signals `write_en`, `imm_en`, and `branch_en`, the address of the next instruction `next_pc`, and the data to be stored into the memory `mem_s`.

We construct a test bench in the file `cpu8_test.v`. It provides the clock and the power signals `clk` and `power` to drive the CPU and includes our program for the CPU to execute. The progress of the execution is reported periodically. The simulation is stopped one cycle after the first `stop` instruction.

3 Automatic Synthesis of the 8-bit CPU

3.1 Functional Validation

We must ensure that there is no bug in our CPU design before synthesis. We validate the functionality of the CPU by running testing programs. A testing program is provided as follows and you should validate that the output is as expected.

```
 0: loadi @0, 0 ;    @0 = 0;
 1: loadi @1, 1;     @1 = 1;
 2: loadi @2, 49;    @2 = 49;
 3: loadi @3, 5;     @3 = 5;
 4: loadi @6, 0;     @6 = 0;
 5: add @4, @2, @0;  @4 = @2;
 6: add @5, @3, @0;  @5 = @3;
 7: and @7, @4, @1;  do{ @7 = @4&1;
 8: jz @7, 10;           if (@7 != 0)
 9: add @6, @6, @5;         @6 += @5;
10: srl @4, @4, @1;      @4 /= 2;
11: sll @5, @5, @1;      @5 *= 2;
12: jnz @4, 7;       }while (@4 != 0);
13: add @5, @5, @0;  @5 = @5; check @5 == 64
14: add @6, @6, @0;  @6 = @6; check @6 == 245
15: add @4, @3, @0;  @4 = @3;
```

```
16: add @5, @2, @0;  @5 = @2;
17: and @7, @4, @1;  do{ @7 = @4&1;
18: jz @7, 20;           if (@7 != 0)
19: sub @6, @6, @5;          @6 -= @5;
20: srl @4, @4, @1;      @4 /= 2;
21: sll @5, @5, @1;      @5 *= 2;
22: jnz @4, 17;      }while (@4 != 0);
23: add @5, @5, @0;  @5 = @5; check @5 == 136
24: add @6, @6, @0;  @6 = @6; check @6 == 0
25: stop
```

This testing program first computes the product of two numbers stored in `@2` and `@3` and stores the result in `@6`. It then subtracts the product from `@6` so eventually it returns to 0. Note that since the machine code is not given, you should first translate the program into the machine code and store it in `code.hex` before simulating the program.

Once you have validated the program, you should change it to compute the product of the last two digits of your CWID and 2 for all the following simulations.

3.2 Design Synthesis

We follow the standard cell based design flow (Tutorial IV) to synthesize the CPU.

For logic synthesis with Synopsys Design Compiler, please set the desired clock frequency to be 250Mhz. Synopsys Design Compiler will automatically pass the specification to Cadence Encounter for place & route.

3.3 Progress Report

You are required to submit a progress report after finishing the above tasks. The general requirement of the ECE lab/project reports with a template can be found on the course website. Follow the template to structure your progress report. It is up to you to use a single or double column format but your progress report should be no more than 6 pages. Use common sense to choose proper fonts, line spacing, and paper margins. Report the measurements of chip area, critical path delay, and power consumption at various stages of the design flow. Screenshots of the simulation results using your own CWID at various design stages, the chip layout, and the equivalence checking report should be attached as the Appendix that does not count toward the 6-page limit.

The following questions should be answered in the Interpretation section.

1. What functionality should a processor provide? How do the various components work together to provide the desired functionality?

2. What metrics are used to evaluate the design? How are they measured in the design flow? What differences have you observed for the measurements at different design stages?

3. What methods are used to validate the functional correctness of the design? Compare their advantages and disadvantages at various design stages.

4 Architectural Exploration of a 32-bit CPU

4.1 Overview

Advances in Electronic Design Automation (EDA) tools makes it possible to automatically synthesis the chip layout from the high-level system description in Verilog. Such automated tools allow the designers to explore many different implementations to evaluate their advantages and disadvantages during a short time period. You are required to first extend the width of the data path of the reference 8-bit CPU to 32 bits and to explore different architectures for the adder in the ALU in order to make design trade-offs.

There are three candidate adder architectures: the carry-skip adder, the carry-lookahead adder, and the carry-select adder. You can also choose other adder architectures, e.g. the ones on the textbook. Note that a faster adder in theory will not always be the faster one in practice because it usually requires more area and thus will incur interconnect delay.

4.2 Design and Synthesis of the 32-bit CPU

You are required to extend the 8-bit CPU in the first part of the final project into a 32-bit one in this project. As the name of the design suggests, the width of the data path should be extended to 32 bits, though we keep the instructions and the number of the memory words unchanged for simplicity.

You can start the modification from the ALU by extending the adder into a 32-bit carry-ripple adder. Then the circuits for addition and subtraction can be updated. For the shift-left-logical and shift-right-logical operations, since each operand is 32-bit now, both of them should use the least 5 bits from `b` instead of the least 3 bits. The bit-wise logical operations and the multiplexers should be updated accordingly to support 32-bit operations. The ALU operations are summarized as follows.

`op[2:0]`	Output	Comments	Examples
000	`s=a<<b[4:0]`	shift-left-logical	`65536=1<<16`
001	`s=a>>b[4:0]`	shift-right-logical	`15=255>>4`
010	`s=a+b`	addition	`5=3+2`
011	`s=a-b`	subtraction	`5=8-3`
100	`s=a&b`	bit-wise and	`2=3&2`
101	`s=~(a^b)`	bit-wise xnor	`254=~(3^2)`

For the memory, there are still 16 *words*, named `@0`, `@1`, . . ., `@15`, though each word is 32 bits wide now instead of 8 bits wide. The addresses of the words remain 4 bits wide. You will need more flip-flops and tristates buffers to support the storage and the query.

Since we keep the width of the instructions unchanged as 16 bits and the width of the program counter `pc` unchanged as 8 bits, it is not necessary to modify the logics concerning `pc`. However, since each memory word is 32 bits now, the semantic of the `loadi` instruction is changed as follows,

Assemble	Machine Code	Comments
`loadi @s,imm[7:0]`	`1010 addr_s[3:0] imm[7:0]`	`@s=imm` where `imm[31:24]=0`

i.e. the highest 24 bits of `imm` is assumed to be 0. Because of such change, the control unit should be modified accordingly.

Note that we can still load a 32 bits value into any memory word by combination of the instructions `loadi`, `sll`, and `add`. For example, if we need to store `0x12345678` (the `0x` stands for hexadecimal numbers) into `@1`, the following instructions can be used.

```
loadi @2,8
loadi @1,0x12
sll @1,@1,@2
loadi @3,0x34
add @1,@1,@3
sll @1,@1,@2
loadi @3,0x56
add @1,@1,@3
sll @1,@1,@2
loadi @3,0x78
add @1,@1,@3
```

You should follow the same steps for the reference 8-bit CPU to synthesize and simulate the 32-bit CPU. A clock frequency of 125Mhz should be used.

4.3 Architectural Exploration of Adder

For the 32-bit CPU constructed from the previous subsection, the carry-ripple adder may limit its performance since the carry has to be propagated through 32 stages. You are required to implement an adder with a different architecture other than the carry-ripple one.

There are a few choices as detailed in Chapter 11.2 of the textbook. You can choose one adder architecture from the carry-skip adder, the carry-lookahead adder, and the carry-select adder, or any other fast adder architecture. Note that you need to decide the parameter for that adder architecture, e.g. for the carry-skip adder, you can choose among the one with 8 4-bit groups or the one 4 8-bit groups. Finally, you should implement the adder in *structural Verilog* and replace the carry-ripple adder with your adder in the 32-bit CPU design.

To verify the correctness of your implementation and to compare your implementation in terms of the performance and the area overhead to the 32-bit CPU in the previous subsection, you need to synthesize and verify your design, as for the 32-bit CPU in the previous subsection.

4.4 Results Collection and Analysis

The results similar to that of the 8-bit CPU design should be collected and analyzed for your 32-bit CPU designs including the one with the carry-ripple adder and the one with the improved adder.

Moreover, since we can implement 32-bit multiplication through the program now, you are required to change the program to compute the product of 4111 and the last six digits of your CWID. The program should be executed on the 32-bit CPU design with the improved adder and the simulation results should be collected and analyzed.

Finally, since the two 32-bit CPU designs, one with the carry-ripple adder and the other with the improved adder should be functionally the same, you should establish such correctness measure through equivalence checking.

5 Bonus Design Problem

In practice, VLSI designs are driven by the need of the market. The economic of the market requires a short time-to-market window from the design specification to design implementation and a reduced design cost to improve the profit margin. To meet such tight budgets in time and cost, designers usually resort to design reuse, i.e., to use a previously designed component in the new design, possibly with some modifications to meet the new design specification.

After our 32-bit processor is built and just before it is released to the market, some customers ask if it is possible to execute integer division on our processor as they need the functionality as soon as possible. To be more specific, given two positive integers a and b, you are required to compute the quotient q and the remainder r, which are non-negative integers such that $a = b * q + r$ and $r < b$. For example, if $a = 12$ and $b = 5$, then $q = 2$ and $r = 2$.

Now, as a designer, you will be award an extra 20 points for the final project to solve this problem within the original deadline. If the problem cannot be solved before the deadline, we may lose many of our valuable customers.

Because of the tight time budget, you are expected to reuse the previous processor design. The first step is to decide if the required functionality is currently available or more hardware should be added. After that, a high-level prototype can be built to validate your idea before diving into implementation. Finally, once you finish the implementation, test benches should be provided to convince others that your design works. Note that you should not change the interface of the CPU, e.g. each instruction should remain 16 bits long, and your implementation should be built with primitive logic gates and structural Verilog.

You are encouraged to discuss your design with your colleagues. However, the implementation and the report should be done individually. Partial credit will be awarded if the proposed design can handle the example discussed above (i.e. $a = 12$, $b = 5$, $q = r = 2$).

6 Project Evaluation

In addition to the progress report, the final project will be evaluated from two more aspects – oral evaluation and final report evaluation. The final project will contribute to 16% of the course grade, plus the bonus which contributes an additional 2%.

Progress Report	Oral Evaluation	Final Report	Bonus Problem
40	40	80	20

6.1 Oral Evaluation

Oral evaluation will be performed in the last day of the lab and the lecture. Each student should prepare a 5 minutes interview with the instructor to explain the motivation, implementation, and evaluation of the CPU design. The following topics will be covered.

- System level
  - Components (memory, ALU, etc.) and their functionalities
  - Processor as a whole digital system
- Circuit level
  - Implementation of components as sequential and/or combinational circuits
  - Critical path and performance
- Synthesis flow
  - Stages in the synthesis flow and tools used
  - Input/output of each stage
- Functional validation and verification
  - Methods and tools to validate and verify the functionality of the designs at each design stage

For main campus students, the evaluation is during your designated lab time or the lecture time. Detailed schedule will be posted separately.

For online and remote students, you are recommended to visit the professor in person for oral evaluation; otherwise, a PowerPoint presentation should be prepared to cover the topics and is due at the same time of the final report.

6.2 Final Report

You are required to submit a stand-alone report to summarize the whole final project. This report should include the necessary results and analysis as required in the previous sections, and be readable without reference to other materials, including but not limited to the project instructions, the textbook, and the progress report.

The report should be limited to 15 pages at most and be legible when printing on letter-size papers. Please use common sense to format your report, e.g. report with small fonts/figures will not be graded. Your codes/screen shots/results can be attached as the appendix which will not count toward the 15-pages limit. However, you should discuss them within the 15-pages limit.

You can include your OWN writings from your progress report. All the writings, results, codes, and screen shots should be by yourself. COPY without proper CITATION, and extensive COPY from other materials including but not limited to project instructions and textbooks, will be treated as PLAGIARISM and called for DISCIPLINARY ACTION.

You should clearly separate your contribution from existing works, e.g. to separate your implementation from the implementation that is already given.

The following sections are recommended as an effective way to organize your final report.

- **Abstraction**
  In less than 100 words, briefly discuss your contributions in the project.
- **Introduction**
  Summarize the motivation of the project. Highlight the engineering and design challenges in this project and the methods to overcome these challenges.
- **Background**
  Give concise descriptions of the processor design and the adder architectures. Note that you should cite various references properly, e.g. the project instructions and the textbook.
- **32-bit CPU Implementation**
  Explain the approach you take to extend the 8-bit processor into a 32-bit one in plain English, possibly with pieces of Verilog code.
- **Architectural Exploration of Adders**
  Discuss the trade-offs among the adders and then motivate your choice of adder architecture. Explain your implementation in plain English, possibly with pieces of Verilog code.
- **Division Support**
  Discuss your solution to the bonus design problem if you decide to solve it.

- **Functional Validation and verification**
  Discuss the approaches taken to validate and verify your designs, e.g. RTL simulations with the test bench and the equivalence checking between the two 32-bit CPU designs. Justify your claims of correctness with simulation results or equivalence checking reports.

- **Synthesis Results**
  Discuss the synthesis flow including RTL synthesis, place & route, etc. Explain the differences of the designs at various design stages. Compare your designs in terms of performance and cost.

- **Conclusion and Future Work**
  Summarize your contributions and discuss possible future works.

- **Appendix**
  Verilog code/screen shots/results listing.

- **References**