ECE 429 Final Project

Fall 2016

(Report Due Date: 12/02 (Friday) at 10:00 AM in US-Central time)

***Note: Please submit the source codes and the report to Blackboard**

"Case Study for 32-bit Pipelined CPU design with New ALU Architecture"

Project Policy: This final project will be done **individually**.
Copying source codes or report will call for ECE disciplinary action strictly .

1. Introduction

This handout describes the final project for ECE 429. The objective of this project is to understand a 32-bit Pipelined Central Processing Unit (CPU). As the name of the design declares, the word length of the data used in the circuits is **32 bits**. Furthermore, since this circuit is pipelined, more than one instruction can be executed simultaneously. The operation of the circuit is synchronized by an externally set clock signal. Also the instruction signals for addressing the memory file, selecting the Arithmetic Logic Unit (ALU) operands and specifying the operation of the ALU are also external. The correct synchronization of those signals with the critical data path delay of the circuit that will determine the minimum operating period is one of the objectives of the project.

2. Circuit Description

An overview of the primary building blocks and signals of the CPU is shown in Fig. 1. As shown in Fig. 1, the primary building blocks are the memory file and the ALU. The external clock signal synchronizes the capture and release of data within the memory file block. The circuit is pipelined and each instruction is explained in two clock cycles. **In the first clock cycle**, the two decoders are used to decode the external address selection signals used for specifying the contents of the memory file that should be read at the memory ports in each clock cycle. Additionally, multiplexer blocks are used to select the operands for the ALU. **In the next clock cycle**, the ALU executes the specified operation. The ALU results can be read from the outside of the CPU through a tri-state buffer, based upon the value of the externally specified OEN (output enable) signal. Finally the ALU results can be written back in the memory file in the word specified by the *Address B*.

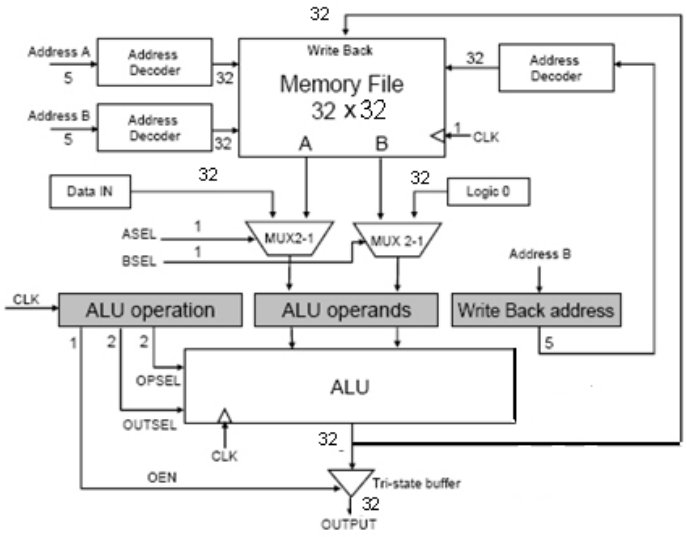

Figure 1. Overview of the primary blocks and signals

3. Memory File

The memory file of this design stores **32 32-bit words**. There are two read ports in the memory file and one write port. The words to be read in each clock cycle are specified by the external 5-bit words *address A* and *Address B*. The internal configuration of the memory file is illustrated below in Fig. 2.
As illustrated in Fig. 2, the primary storing element within the memory file is a D-register. The output of each D-register is connected to the two output ports of the memory file through tri-state buffers. The tri-state buffers are enabled by the decoded address (signals A and B) and the contents of the D-registers appear at the output ports A and B respectively.

Furthermore, the value of address B specifies the word address in the memory file where the results of the ALU computation are stored in the second cycle of instruction execution. The writing of the ALU results within the memory file is synchronized by the clock signal.

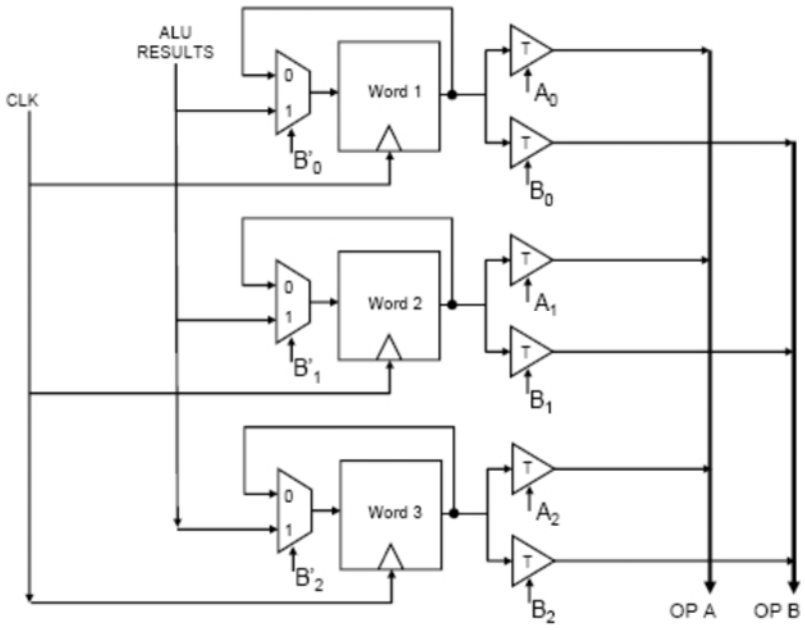

Figure 2. Memory file configuration

4. Arithmetic Logic Unit (ALU)

The ALU of the circuit has two operands A and B and an implement the following eight functions:
- A * B : multiplication
- A + B : addition
- A - B : subtraction
- B - A : subtraction
- A or B : logic OR function
- A and B : logic AND function
- A xor B : logic XOR function
- A xnor B : logic XNOR function

As illustrated in Fig. 1 the operand of the CPU can be selected among the following:

- Operand A: Operand A can be selected between the *read port A* of the memory file and the externally defined *data in*. The selection is done by the external signal *ASEL*.
- Operand B: Operand B can be selected between the read *port B* of the memory file and the logic zero value. The selection is done by the external signal *BSEL*.

Internally the ALU has three primary operation blocks: **the multiplier**, **the adder** and **the logic function block**. These blocks are illustrated in Fig 3. The multiplier can be implemented as 32-by-32 array-based multiplier. The multiplier executes the multiplication function. Notice, however, that the result of the multiplication operation is 64 bits. Therefore, in order to store the multiplication result back to memory file we need two clock cycles. Therefore the multiplication instruction is executed in **3 clock cycles**. This is made possible by pipelining the multiplier unit in order to produce the 16 least significant bits (LSB) of the result in once clock cycle and the following 16 most significant bits (MSB) in the next cycle. Notice however, that the instruction immediately following a multiplication operation should select the MSB of the multiplier at the output of the ALU. It should also specify the storage address of the MSB bits in the memory file.

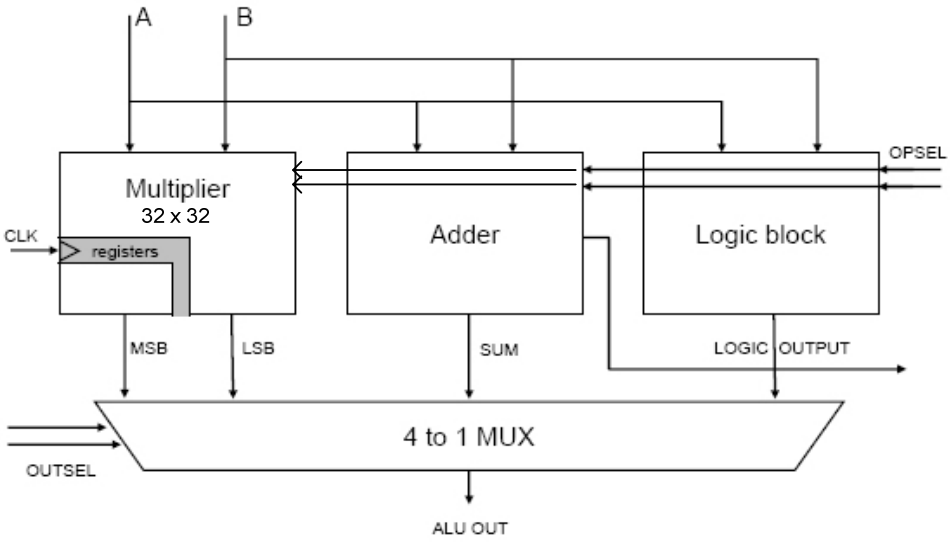

Figure 3. Block diagram of the ALU circuit

The adder circuit within the ALU is a 32-bit adder/subtractor circuit. It executes the addition and subtraction operations. The selection of the operation is done by the two externally defined operation select signals *OPSEL*. The same signals are used for specifying the operation executed within the logic function block of the ALU. The final output of the ALU is specified by the output select *OUTSEL* signals that control the final 4-to-1 multiplexer within the ALU.

Finally, the ALU creates a control output signal that can be used externally of the CPU:

- *Adder overflow* - the signal is 1 if there is an adder overflow.

5. Synchronization

To better understand the circuit synchronization sequence described below, please refer to Fig 1. The operation of the CPU is synchronized by the external clock signal.

An instruction to be executed by the CPU is determined by the external control signals. An example of an instruction word is illustrated in Fig. 4. After the clock switches high the instruction is applied (fetched) to the signals that control the CPU operation. Since each instruction is executed in two steps, some of these control signals need to be stored at the internal registers of the CPU. On the first step of the instruction, these signals will specify the contents of the memory file that will be read from the read ports *A* and *B*. Also they will specify the operands of the ALU. In the second cycle of the operation the control signals will determine the operation to be executed internally the ALU, and the value of the ALU output.

Figure 4. Instruction word contents

The results of the ALU will be available of the circuit if the *OEN* signal is set, and it will also be written back in the memory file. The address in memory where the ALU result is written is specified by the address B value. The data will be written that word at the next positive edge of the clock signal.

The control signals are set after a positive edge of the clock signal and should not change before the next positive edge. The period of the clock signal is determined by the longest path of data within the circuit.

Case Study-1

32-bit CPU design with Different Adders - Carry Ripple Adder, Carry Lookahead Adder, Carry Skip Adder, and Carry Select Adder (Source codes are provided)

6. Introduction: Case Study-1

We provide the soruce verilog code and test bench for the cpu design with **Carry Ripple Adder (cpu_CRA.v), Carry Lookahead Adder (cpu_CLA.v), Carry Skip Adder (cpu_CSA.v), Carry Select Adder (cpu_CSeA.v), and testbench verilog (tb_cpu.v)** so that you can do the logical synthesis and physical synthesis by using IIT-ECE429 ASIC flow. You can follow the standard cell based flow to synthesize and layout the CPU design. Please refer to the tutorial IV for detailed information which we already conducted in Lab. 9.

6.1 RTL Simulation

We must ensure that there is no bug in the design before synthesis. We verify the correctness by running testing testbench. The whole cpu design in Verilog is provided, “cpu_xxx.v”, and a testbench for verifying the CPU, “tb_cpu.v”, where we test functionality for store, read, addition, and subtraction.

The testbench (tb_cpu.v) we provided tests the following instruction set:

```
[0]             STORE       -10
[1]             STORE           10
[2]             STORE           30
[20] STORE              50
[0][1]          ADD
[1][2]          ADD
[0][20]         SUBTRACT
[1]             READ
[2]             READ
[20] READ
```

6.2 Logic Synthesis and Post-Synthesis Simulation

For logic synthesis with Synopsis DC, please set the initial desired clock frequency 30 MHZ (You can figure out max frequency later in your report. What is the maximum frequency after P&R?) Once finished, you should check the report files cell.rep and timing.rep for the area and timing of the design. Moreover, you will obtain the mapped circuit in cpu.vh.

You should simulate it with the Verilog models of the standard cells, i.e. gscl45nm.v, and compare the result with the RTL simulation. The command is:

verilog gscl45nm.v tb_cpu.v cpu.vh

6.3 Place & Route and Post-P&R Simulation

Now, we are ready to run place and route using Cadence SOC Encounter. It wil take some time for the tool to finish the automatic layout generation.

Once finished, you should check the final timing report in timing.rep.5.final in order to verify if all the circuit timings are met. Moreover, you will obtain the circuit netlist in final.v, which contains necessary buffers and inverters to overcome the interconnect delays in the signal propagation network and the clock distribution network. You should simulate it with the Verilog models of the standard cells, i.e. gscl45nm.v, and compare the result with the RTL simulation and the post-synthesis simulation. The command is:

verilog gscl45nm.v tb_cpu.v final.v (with four adders design)

6.4 Explanation about the Different Adders used in Case Study-1

VLSI adders are critically important in digital designs since they are untilized in ALUs, memory addressing, cryptography, and floating-point units. Since adders are often responsible for setting the minimum clock cycle time in a processor, they can be critical to any improvements seen at the VLSI level. In this project, we will examine four different adder architectures in 32-bit CPU design. We start from Ripple Carry Adder, which provides one of the simplest types of carry-propagate adder designs, and then Carry Lookahead Adder, which is improved by having the carries precomputed ahead of time. And, we move to Carry Skip Adder and Carry Select Adder, both of which are attempt to obtain some of the improvements by trying to limit the number of gates it has at the expense of some delay.

The four adder architectures that will be implemented in this project are listed below:

- **Carry Ripple Adder (Source code is provided, cpu_CRA.v)**
- **Carry Lookahead Adder (Source code is provided, cpu_CLA.v)**
- **Carry Skip Adder (Source code is provided, cpu_CSA.v)**
- **Carry Select Adder (Source code is provided, cpu_CSeA.v)**

Here are the brief description about the adder architectures.

- **Carry Ripple Adder**

An n-bit CRA is formed by concatenating n FAs in cascade, with the cary output from one full adder connected to the carry input of the next full adder. The carries are connected in a cahin through the full adders as shown in Figure 5. (See ‘module fa32’ in the cpu_CRA.v)

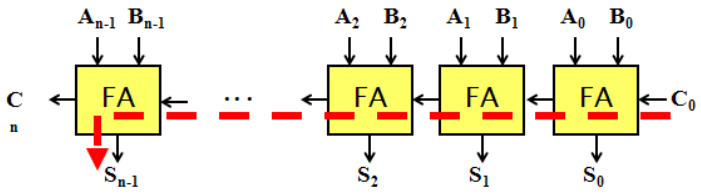

Figure 5 Carry Ripple Adder

- **Carry Lookahead Adder**

The Carry Lookahead circuits are special logic circuit that can dramatically reduce the time to perform addition at the price of more complex hardware. The carry Lookahead design can be obtained by a transformation of the ripple carry design in

which the carry logic over fixed groups of bits of the adder is reduced to two-level logic. An example is shown for 4-bit adder group in Figure 6. We can extend the 4-bit design to 32 bit (See 'module cla32' in the cpu_CLA.v).

A = 1011
B = 0110
A + B = 10001

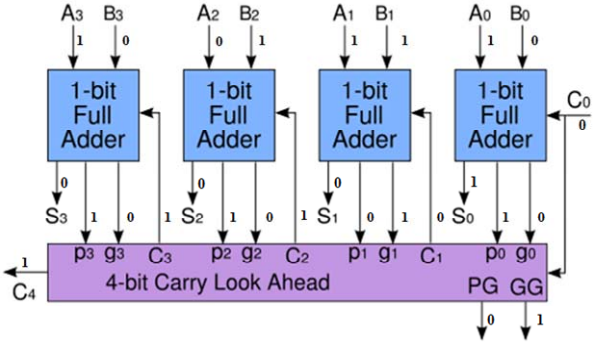

Figure 6 4-bit Carry Lookahead Adder

where

$$C_1 = g_0 + p_0 C_0$$

$$C_2 = g_1 + p_1(g_0 + p_0 C_0) = g_1 + p_1 g_0 + p_1 p_0 C_0$$

$$C_3 = g_2 + p_2(g_1 + p_1(g_0 + p_0 C_0)) = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0$$

*Note: $p_i = A_i + B_i$ (or $A_i \oplus B_i$) and $g_i = A_i \bullet B_i$

- **Carry Skip Adder**

In the Carry Skip Adder, the operands are divided into blocked of r bits blocks. Within each block, a ripple carry adder can be utilized to produce the sum bits and a carryout bit for the block. Carry Skip Adder is based on observation that carry process can skip stage for which $x_i \neq y_i$ (that is $p_i = x_i \oplus y_i = 1$). Each group generates "Group Carry-Propagate"=1 if all p_i=1 in each group. An example for 4-bit Carry Skip Adder strcuture is given in Figure 7, where $C_{o,3} = C_{i,0}$ when "Block Propagate" is equal to 1, i.e. p_i=1 for all four bits.

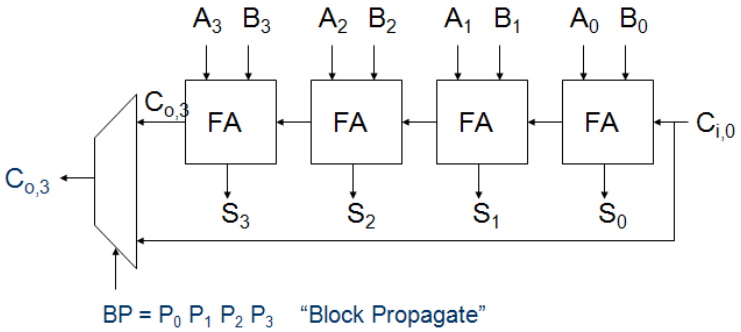

Figure 7 4-bit Carry Skip Adder

- **Carry Select Adder**

The Carry Select Adder divides the operands to be added into r bit blocks similar to Carry Skip Adder. For each block, two r-bit Ripple Carry Adders operates in parallel to form 2 sets of sum bits and carry out signals. Each Ripple Carry Adder has two sets of hard-coded carry-in signals. One Ripple Carry Adder has a carry-in of 0, whereas, the other has a carry-in of 1. An example of 4-bit Carry Select Adder is shown in Figure 8.

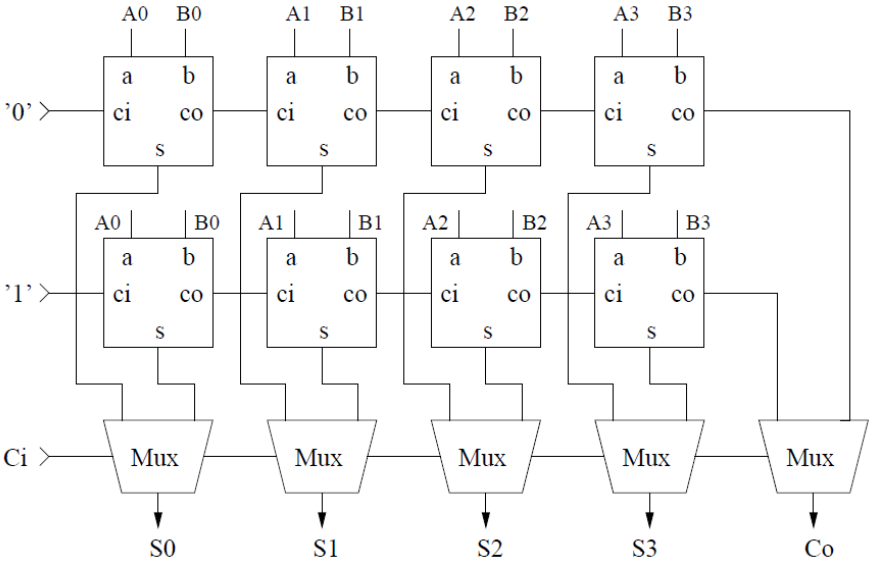

Figure 8 4-bit Carry Select Adder

The upper half is implented by two independent 4-bit adders, one whose carry-in is hardwired to 0, another whose carry-in is hardwired to 1. In parallel, these compute two alternative sums. The carry-out from the previous 4-bit adder block controls multiplexers that select between the two alternative sums. Following the same methodology, the two alternative carry-outs are selected by carry-out from the previous block controling a multiplexer that selects the appropriate carry-out for the next block. A structure of 16-bit Carry Select Adder blocks is shown in Figure 9.

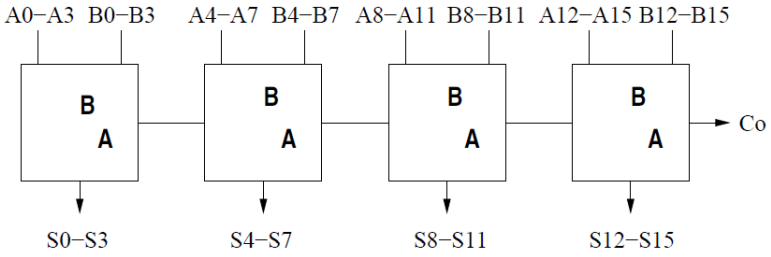

Figure 9 16-bit Carry Select Adder

6.5 Report Submission for Case Study-1

For each cpu design with differrent adders:

1. Generate the display screenshot or the text output of the RTL simulation and the screenshot from simvision with provided test bench (tb_cpu.v).
2. Synthesize the design and summarize cell.rep and timing.rep.
3. Provide the display screenshot or the text output of the post-synthesis simulation and the screenshot from simvision.
4. Summarize timing.rep.5.final. What is the maximum clock frequency this circuit can run.
5. Provide the display screenshot or the text output of the post-P&R simulation and the screenshot from simvision.
6. Generate a new test bech file (tb_test.v) for the following instruction set.

```
[0]     STORE  0000_0005
[1]     STORE  AAAA_AAAA
[2]     STORE  5555_5555
[3]     STORE  0000_000A
[4]     STORE  0000_0001
[5]     STORE  FFFF_FFFF
[6]     STORE  0000_00C8
[7]     STORE  0000_012C
```

[8]	STORE	0000_0001
[9]	STORE	AAAA_AAAB
[10]	STORE	5555_5555
[2][0]	ADD	
[1][2]	ADD	
[6][7]	ADD	
[0][3]	ADD	
[2][4]	SUB	
[2][8]	ADD	
[2][0]	SUB	
[9][10]	ADD	
[7]	READ	
[3]	READ	
[1]	READ	

7. Provide the display screenshot and the text output of the RTL simulation and the screenshot from simvision for each cpu desgin (cpu_CRA.v, cpu_CLA.v, cpu_CSA.v, cpu_CSeA.v) with the new generated test bench (tb_test.v).

8. Fill out the following performance comparison table after synthesis and analyze the results (explain the reasons of your comparison results).

		CRA	CLA	CSA	CSeA
Calculate the Path Delay for Each Operation (Post-Synthesis Gate-Level Delay)	5555_5555 + 0000_0005				
	AAAA_AAAA + 5555_5555				
	0000_00C8 + 0000_012C				
	5555_555A + 0000_000A				
	FFFF_FFFF - 0000_0001				
	FFFF_FFFF + 0000_0001				
	FFFF_FFFF – 5555_555A				
	AAAA_AAAB + 5555_5555				

Case Study-2
32-bit CPU design with New ALU Architecture
(Source codes are provided)

7. Introduction: Case Study-2: Comparator Design in the ALU for the 32-bit CPU

In this project, we will add a 32-bit comparator block into the ALU design.

The function of a 32-bit comparator in Verilog is shown in Table 1. Suppose we have two 32-bit inputs (we assume them to be unsigned in this project) A and B. Since the result of comparing them can be A > B , A = B and A < B. So two bits are needed to represent the comparison result (two outputs f1 and f0). Note that when f1 = 1, it means two integers are equal. Otherwise, f0 is used to determine the relation of A and B.

	f1	f0
A > B	0	1
A < B	0	0
A = B	1	0

In this project, you are going to design the 32-bit comparator in a structural way. First of all, we will explain the structure by using 4-bit comparator. Then we will give the structure view of the 32-bit comparator, and you are supposed to finish the Verilog coding according to the structure.

7.1 The structure of 4-bit comparator

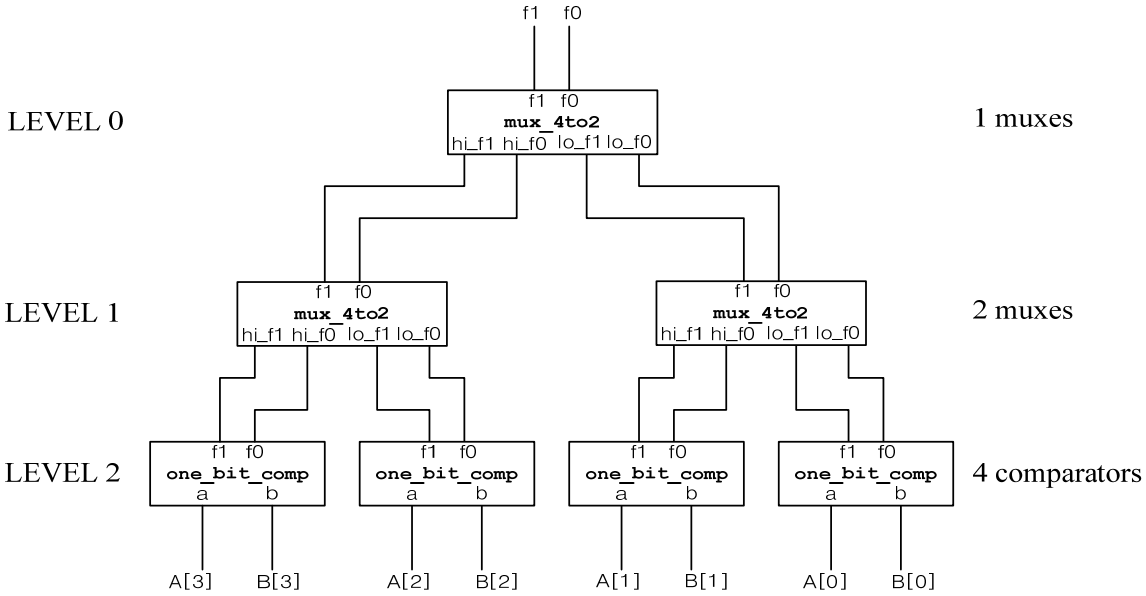

Fig. 10 Structure of 4-bit comparator

The structure of 4-bit comparator is shown in Fig. 10. It is designed in a tree structure. At the bottom level (Level 2), there are 4 one bit comparators. Each of them is used to compare the corresponding bit in A and B. The meaning of the output f1 and f0 are the same as the meaning in Fig. 10 (f1f0=10 means a=b, f1f0=00 means a<b, f1f0=01 means a>b).

Notice that the final comparison result depends on the comparison result of the most significant bit which has determined the relation of the two integers. Take the 4 bit comparator shown in Fig. 10 for example. If the results from MSB A[3] and B[3] has shown that A[3] > B[3] or A[3] < B[3] (in other words, f1 = 0), then it means A > B or A < B. On the other hand, if A[3] = B[3] (f1 = 1), then we have to refer to the comparison result of next significant bit A[2] and B[2]. If A[2] and B[2] are equal, we have to compare A[1] and B[1], and so on. If all the 4 bits are equal (f1 from all the four one bit comparators are all 1s), then A = B. In fact, rather than comparing bits from MSB to LSB sequentially, we can do the comparison in parallel in order to save time, as we can see from Fig. 10. Remember that the left part of f1 and f0 results always have higher priority than the right part of the f1 and f0. To be more specific, for the component of mux_4to2 in Fig. 10, if hi_f1 = 0 which means the relation of A and B has already been determined, then its output f1 and f0 should be consistent with hi_f1 and hi_f0. Otherwise, f1 and f0 should be consistent with lo_f1 and lo_f0.

From Fig. 10, we can see that the number of mux_4to2 is 3 which is equal to 4 – 1 and the level of the tree is 3 which is equal to $\log_2(4) + 1$. More generally, if two N-bit (N is the power of 2) unsigned integers are compared, then the tree comparator will be $(\log_2(N) + 1)$ levels, and it will consists of N -1 mux_4to2 and N one_bit_comp.

7.2 The structure view of the 32-bit comparator

The structure of the 32-bit comparator is shown in Fig. 11. You are supposed to finish the Verilog coding of this structure and include it in the ALU design. There should be three modules in your Verilog code: one_bit_comp, mux_4to2, and tree_comp. The definition part of each module is included in file **cpu_comp.v**, and they are listed in Fig. 12. You should finish the code in order to complete your new cpu design.

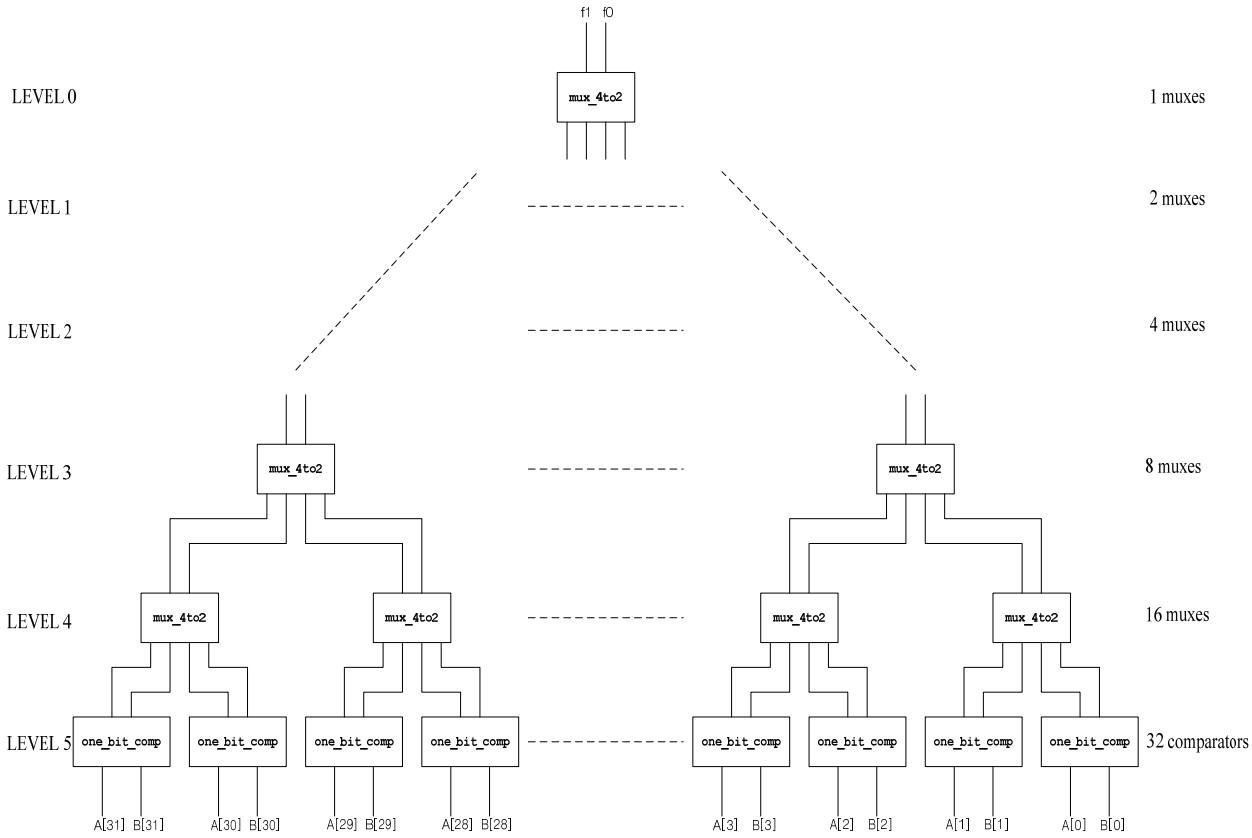

Fig. 11 Structure of 32-bit comparator

```
//one bit comparator
module one_bit_comp(a, b, f1, f0);
input a, b;
output f1, f0;

//if (a > b) then {f1, f0} = 2'b01
//if (a < b) then {f1, f0} = 2'b00
//if (a == b) then {f1, f0} = 2'b10
//write your code here

endmodule

//mux to select the f1 f0 outputs
module mux_4to2(hi_f1, hi_f0, lo_f1, lo_f0, f1, f0);

input hi_f1, hi_f0, lo_f1, lo_f0;
output f1, f0;

//use hi_f1 to select the correct outputs
//write your code here

endmodule

//32-bit tree comparator
module tree_comp(A, B, f1, f0);
input [31 : 0] A, B;
output f1, f0;

wire [31 : 0] f1_L5, f0_L5;
wire [15 : 0] f1_L4, f0_L4;
wire [7 : 0] f1_L3, f0_L3;
wire [3 : 0] f1_L2, f0_L2;
wire [1 : 0] f1_L1, f0_L1;

//write your code here

//Level 5: 32 one_bit_comp go here

//Level 4: 16 mux_4to2 go here

//Level 3: 8 mux_4to2 go here

//Level 2: 4 mux_4to2 go here

//Level 1: 2 mux_4to2 go here

//Level 0: 1 mux_4to2 goes here

endmodule
```

Fig. 12: Codes to be finished

7.3 The new ALU design

After adding the 32-bit comparator to the ALU design, the new ALU will look like Fig. 13. Note that we should extend the two bit output {f1, f0} to 32-bit result. Moreover, the original 4 to 1 MUX in the ALU should be changed to a 5 to 1 MUX, and its select signal OUTSEL should be 3 bits now. The ALU design has already been modified so that you can focus on the comparator design.

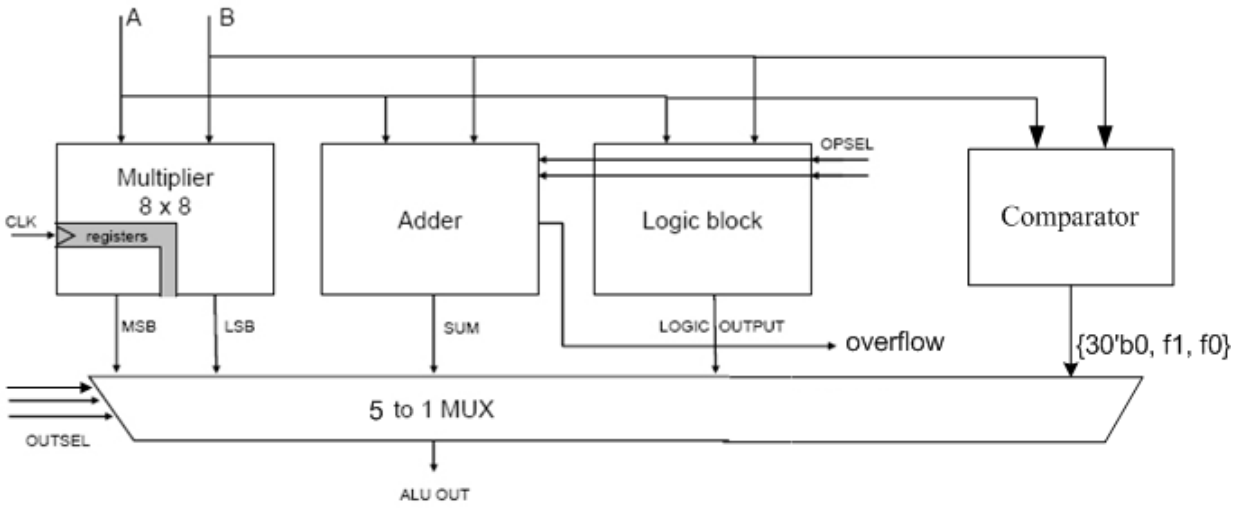

Fig. 13 Block diagram of the new ALU circuit

7.4 Report Submission for Case Study-2

For the cpu design with CLA adder:

1. Generate a new test bech file (tb_test_comp.v) for the following instruction set.

[0]	STORE	0000_0005
[1]	STORE	AAAA_AAAA
[2]	STORE	5555_5555
[3]	STORE	0000_000A
[4]	STORE	0000_0001
[5]	STORE	FFFF_FFFF
[6]	STORE	0000_00C8
[7]	STORE	0000_012C
[8]	STORE	0000_0001
[9]	STORE	AAAA_AAAB
[10]	STORE	5555_5555
[2][0]	ADD	
[1][2]	ADD	
[6][7]	ADD	
[0][3]	ADD	
[2][4]	SUB	
[2][8]	ADD	
[2][0]	SUB	
[9][10]	ADD	
[8]	READ	
[10]	READ	
[4]	READ	
[2]	READ	
[7]	READ	
[3]	READ	
[8][10]	CMP	
[4][2]	CMP	
[3][7]	CMP	
[10]	READ	
[2]	READ	
[7]	READ	

2. Provide the display screenshot or the text output of the RTL simulation and the screenshot from simvision with the test bench (tb_test_comp.v).
3. Synthesize the design and summarize cell.rep and timing.rep.
4. Provide the display screenshot or the text output of the post-synthesis simulation and the screenshot from simvision.
5. Summarize timing.rep.5.final. What is the maximum clock frequency this circuit can run.
6. Provide the display screenshot or the text output of the post-P&R simulation and the screenshot from simvision.

Good luck!