Student name: Ahmed Elliethy

Lab3

Part 1 (ALU)

Abstract:

In this Lab, we implement a 16-bit ALU with register file. The proposed design get an input instruction with specified format, then extract the required values from the register file, and finally, execute at the ALU. We also simulate the delay in both the register file and ALU.

Design:

The proposed instruction is 29 bits wide. It has 2 types:

A. **R-type instruction**, its format is as follows

Op code (5) 28-24	Rd (4) Destination reg. 23-20	Rs(4) First operand reg. 19-16	Rt (4) Second operand reg. 15-12	Ext(12) Unused 11-0

B. **I-type instruction**, its format is as follows

Op code (5) 28-24	Rd (4) Destination reg. 23-20	Rs(4) First operand reg. 19-16	Immediate (16) 15-0

We determine from the last bit of the op-code, if the instruction takes all operands from the register file (R-type) or it will take one of its operand as immediate value (I-type).

The op-code for different instructions shown in the following table:

Op-code	Instruction
00000	AND (R-type)
00001	AND (I-type)
00010	OR (R-type)
00011	OR (I-type)
00100	ADD (R-type)
00101	ADD (I-type)
01100	SUB (R-type)
01101	SUB (I-type)
01110	SLT (R-type)
01111	SLT (I-type)
11000	NOR (R-type)
11001	NOR (I-type)

Also, we simulate the different delays of the function units. So, for registers, it takes 50 ps for reading and 50ps for writing, and for ALU it takes 200ps to do any calculation.

Please note that the output of each instruction is available after 300ps of placing the instruction.

Detect Overflow:

Overflow occurs when the magnitude of a number exceeds the range allowed by the size of the bit field (16-bits). This may happen when do a sum of two identically-signed numbers, but there will never be an overflow when two numbers of opposite signs are added together.

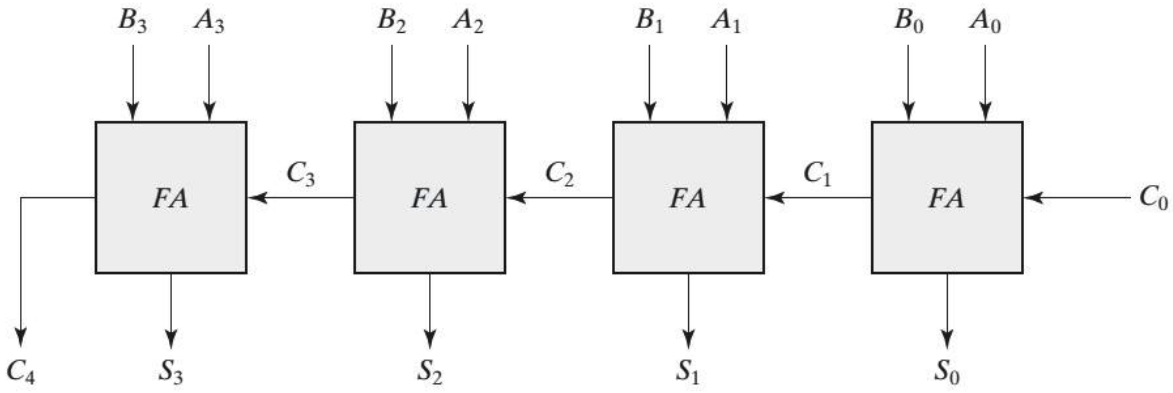
So, we detect the overflow only for add (when same sign numbers) and for subtract (when different sign numbers).

detecting overflow in these cases is done by comparing the last bit of the result with the last bits of the operands. If they are the same, there is no overflow, and if they are different then an overflow occurs.

Part 2 (CLA)

Carry Propagation:

The addition of two binary numbers in parallel implies that all the bits of the operands are available for computation at the same time. As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals. The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders, since each bit of the sum output depends on the value of the input and the previous bit summation carry.



By, providing the all carries at the same time, the adder can output the result very fast, without waiting for each bit carry to propagate to the next bit. This is called Carry Look Ahead (CLA) adder.

Consider the circuit of the full adder shown in figure above. If we define two new binary variables:

$$gi = ai \bullet bi$$

$$pi = ai + bi$$

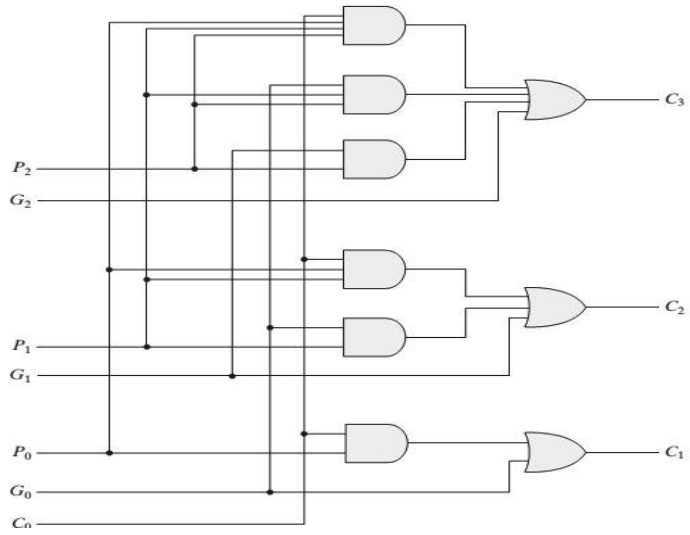
The output carry can be expressed as,

$$ci+1 = gi + pi \bullet ci$$

So,

$$c1 = g0 + p0 \bullet c0$$
$$c2 = g1 + p1 \bullet g0 + p1 \bullet p0 \bullet c0$$
$$c3 = g2 + p2 \bullet g1 + p2 \bullet p1 \bullet g0 + p2 \bullet p1 \bullet p0 \bullet c0$$
$$c4 = g3 + p3 \bullet g2 + p3 \bullet p2 \bullet g1 + p3 \bullet p2 \bullet p1 \bullet g0 + p3 \bullet p2 \bullet p1 \bullet p0 \bullet c0$$

As we see from the right figure, as the number of bits increases, the complexity and number of gates increase very fast.



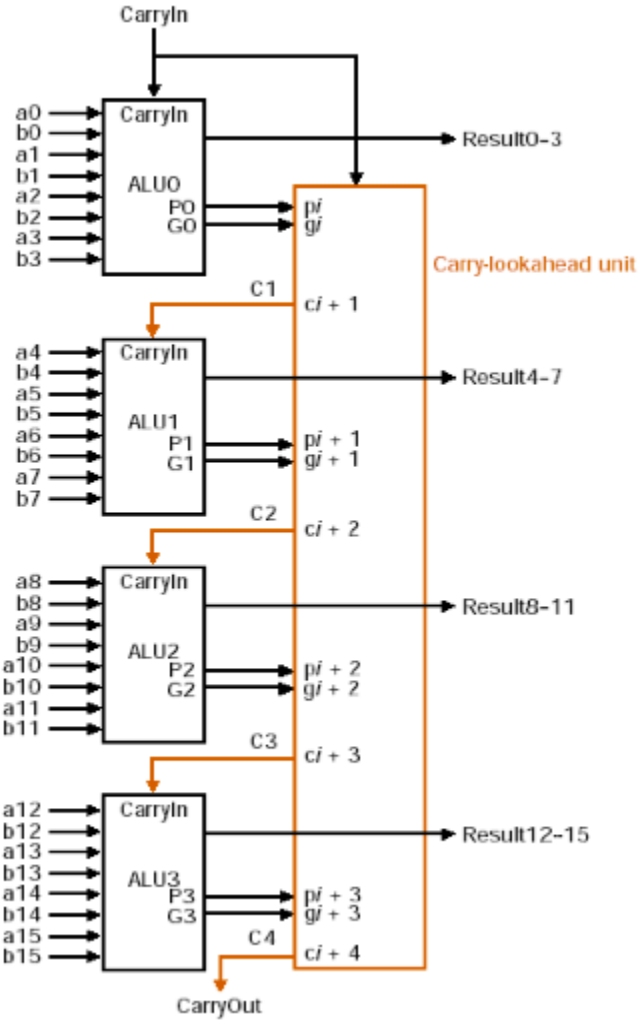
To avoid this complexity, we split the input 16 bits to 4 groups, each group has 4 bits. So, our design of the 16-bit CLA adder is done by combining four 4-bit CLAs with an additional unit called Carry-look ahead unit which computes all carries of groups at once

The Carry-look ahead unit accepts the group propagate (P) and group generate (G) from each of the four CLAs and generate C1, C2, C3, C4 at once. (P) and (G) have the following expressions:

$P = p_3.p_2.p_1.p_0$

$G = g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.g_0$

Then, each of the four 4-bit CLAs takes the carries output from the Carry-look ahead unit not from each previous adder. So, there is no wait to get the previous carry, as the Carry-look ahead unit generates all of them at once. This is shown in the right figure.



Our Verilog design has 4 modules:

1 - CLA_Group_Generation4, which generate P and G for any group.

2 - CLA_unit_16, which generates the 4 group carries C1, C2, C3, and C4 at once.

3 - CLA_adder_4, which is a 4-bit CLA adder. It takes two 4-bit inputs and output the 4-bit sum, and **there is no output carry** as we already generate all carries using the Carry-look ahead unit in module 2

4 - CLA_add_16, which is the main module. It implements the 16 bit CLA adder by wiring up the previous modules.

Part 3 (Code generation)

The required here is to generate instructions for our ALU in the part 1 to execute the following program

```
short s, i;
s=0;
for (i=0; i<10; i++) {
s = s + i;
}
```

Suppose that $R0 =0 and (s) corresponds to $R1 and (i) corresponds to $R2. Also, we will unroll the loop as we don't have jump instruction in our machine, so the c-code and their ALU instructions will be:

c-code	ALU instructions	Binary instruction (machine code)

`short s, i;`
`s=0;`
`i=0;`
	
	addi $R1, $R0, 0
	addi $R2, $R0, 0
		
		00101 0001 0000 0000000000000000
		00101 0010 0000 0000000000000000

`s = s + i;` `i = i + 1;`	**add $R1, $R1, $R2** **addi $R2, $R2, 1**	**00100 0001 0001 0010 000000000000** **00101 0010 0010 0000000000000001**
`s = s + i;` `i = i + 1;`	**add $R1, $R1, $R2** **addi $R2, $R2, 1**	**00100 0001 0001 0010 000000000000** **00101 0010 0010 0000000000000001**
`s = s + i;` `i = i + 1;`	**add $R1, $R1, $R2** **addi $R2, $R2, 1**	**00100 0001 0001 0010 000000000000** **00101 0010 0010 0000000000000001**
`s = s + i;` `i = i + 1;`	**add $R1, $R1, $R2** **addi $R2, $R2, 1**	**00100 0001 0001 0010 000000000000** **00101 0010 0010 0000000000000001**
`s = s + i;` `i = i + 1;`	**add $R1, $R1, $R2** **addi $R2, $R2, 1**	**00100 0001 0001 0010 000000000000** **00101 0010 0010 0000000000000001**
`s = s + i;` `i = i + 1;`	**add $R1, $R1, $R2** **addi $R2, $R2, 1**	**00100 0001 0001 0010 000000000000** **00101 0010 0010 0000000000000001**
`s = s + i;` `i = i + 1;`	**add $R1, $R1, $R2** **addi $R2, $R2, 1**	**00100 0001 0001 0010 000000000000** **00101 0010 0010 0000000000000001**
`s = s + i;` `i = i + 1;`	**add $R1, $R1, $R2** **addi $R2, $R2, 1**	**00100 0001 0001 0010 000000000000** **00101 0010 0010 0000000000000001**
`s = s + i;` `i = i + 1;`	**add $R1, $R1, $R2** **addi $R2, $R2, 1**	**00100 0001 0001 0010 000000000000** **00101 0010 0010 0000000000000001**
`s = s + i;` `i = i + 1;`	**add $R1, $R1, $R2** **addi $R2, $R2, 1**	**00100 0001 0001 0010 000000000000** **00101 0010 0010 0000000000000001**

The expected result after finish execution of this program is that **$R1 (s)** will be **45** and **$R2 (i)** will be **10**. This is verified using the test bench Verilog file. Please look at the last 2 values and make the simulation time more than 8ns.