Logic Design Lab 2: ALU Report

110062219

April 25, 2022

1 Design

We are to carry out a 4-bit carry-lookahead adder, a 16-bit adder and a 16-bit Arithemtic & Logic Unit in the Lab 2.

1.1 4-bit Carry-Lookahead Adder

I used the **dataflow description** to implement my CLA.

First I declared wire arrays g, p, c of size 4 and assign $g = A \wedge b, p = A \oplus B$. Then the carries c_i are $g_{i-1} \vee p_{i-1} \wedge c_{i-1}$ respectively. The expressions of c_i are recursively expanded in terms of only g_j, p_j, Cin. Finally, we have the output $S = p \oplus c, Cout = g_3 \vee p_3 \wedge c_3$.

1.2 16-bit Adder

Though the course materials shows that we could have a 16-bit adder by four 4-bit CLAs in hierarchical manner, the spec of out CLA does not include the group generate and the group propagate.

As a consequence, I serialized four 4-bit CLAs just as what a ripple-carry adder is done.

1.3 16-bit Arithemtic & Logic Unit

Here comes the most interesting and surprising part. I implemented a very basic ALU which lies in every CPU!

Since we have 16 different operations, I used a `always-case` block to perform as a multiplexer and declared the outputs to be registers. In case of generating unexpected latches, The value of each output is set even if it is don't-care.

1.3.1 Addition & Subtraction

First I declared two wire arrays s_a, s_s of size 16 to store the result of *adder* and *suber* and two wires $cout_a, cout_s$ to store the carry-out of *adder* and *suber*.

Then I declared two 16-bit `adder` $adder(A, B, Cin, s_a, cout_a)$ and $suber(A, -B, 0, s_s, cout_s)$.

Finally in the cases of addition and subtraction, Y is assigned to s_a, s_s and $Cout$ to $cout_a, cout_s$ respectively. As for $Overflow$, it is

$$A_{15} \wedge B_{15} \wedge \neg Y_{15} \vee \neg A_{15} \wedge \neg B_{15} \wedge Y_{15}$$

for addition and

$$\neg A_{15} \wedge B_{15} \wedge Y_{15} \vee A_{15} \wedge \neg B_{15} \wedge \neg Y_{15}$$

for subtraction.

1.3.2 `Find-First-Set`

For the last operation to find the first set bit from the higher side, I used a `casex` block to brute-force enumerate all 17 cases.

I was confused what should I output when there is no set bit, i.e., the input is 0. After some search on the Internet, I realized that `FFS` usually return 0 whereas `count-leading-zeros` (`__builtin_clz()`, called by `std::__lg()`) sometimes return the size of the integer type.

1.3.3 Comparator

I used a `if-else` branch.

1.3.4 Decoder

I used this expression $Y = 1 << A_{3:0}$.

1.3.5 Other Operations

For logical and arithmetic, left and right shift; bitwise and, or, exclusive-or, not operations, I used `verilog` operators directy.

2 Problems

It's definitely not an easy lab for a beginner of hardware like me. I encountered several problems.

Some of them is so stupid. For instance, I made a typo that mistook $c_1 = g_0 \vee p_0 \wedge c_0$ for $c_1 = g_0 \vee p_0 \wedge g_0$ in my `CLA`, which cost my plenty of time not until I wrote a test bench did I find it.

2.1 Signed and Unsigned Numbers

Another problem occurred when comparing two integers. I found that my `ALU` had trouble comparing negative numbers. After declaring input with `signed`, it worked properly.

Later, I saw another approach is to use `$signed(`x`)` of a variable x in the test bench provided by TAs.

2.2 Wire Connected Multiple Times

In the beginning, some result of test bench showed that my $Cout$s were `x`. I was really frustrated then. I checked the $Cout$s of both `CLA` and `adder` various times.

Subsequently, I was skeptical about my `ALU` and I found that I had connected wire $Cout$ to both *adder* and *suber*! So I declared $Cout$ to be a register and another two wires $cout_a, cout_a$ connected to the two adders.

2.3 Carry-in of Subtraction

There were only three **WA** test cases then, which were all subtractions with $Cin = 1$.

I found that the odd-even parity was weird. For instance, the correct answer of an even number subtracts another even number with $Cin = 1$ is also even, rather than odd as I expected. Until reading our spec cautiously, I found that Cin is don't-care!

So I got **AC** finally!

3 Questions and Reflection

Although I've finished this lab, I still have some questions:

1. In my `CLA`, I expanded c_i in terms of only g_j, p_j, Cin. For instance, $c_2 = g_1 \vee p_1 \wedge c_1 = g_1 \vee p_1 \wedge g_0 \vee p_1 \wedge p_0 \wedge Cin$. What if I just assign

$c_2 = g_1 \vee p_1 \wedge c_1$? Would `ncverilog` expand it automatically, or would it decline to a ripple-carry adder?

2. I couldn't `make compile`. `make` said that "dv: Command not found", but in my current shell, `which dv` showed that "aliased to design_vision".

3. The professor told us parameterized module in class. If we want to implement out CLA and adder in that way, how could we do that? Can we use a loop to declare something inside a module?

By the way, it's really nice for you to provide many files for us. I modified the directory structure to put my source file in `src/`, test benches in `tests/` and our targets in `build/`. I tried to modify `Makefile` to let it more like a Makefile but in vain, though.

I 'm interested in `ALU.tcl` the most. I heard of `Tcl` when I learnt the GUI framework Tk used by Python. It's so cool to control the GUI by command line, but I'm wondering why there is not a command to synthesis.

After all, despite the difficulty of this lab, I found sense of achievement and satisfaction after synthesis the module and pass the test bench successfully.