Computer Organization

HW1ALU design

111550168 陳奕

1. Architecture Diagrams

2. Questions

(1) Overflow = (operation == 2'b11)? 0 :
(operation[1]&~operation[0])&((~ai&~bi&sum)|(ai&bi&~sum))

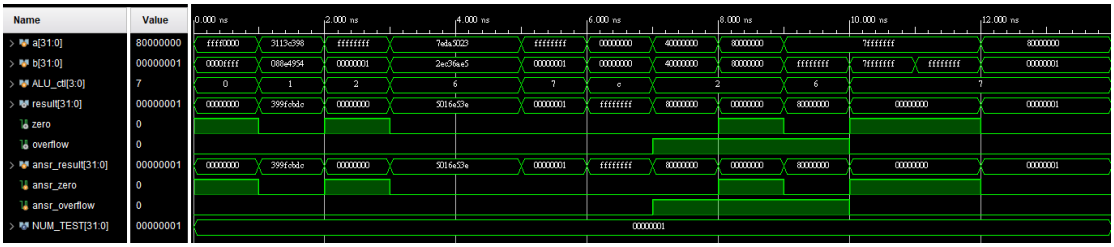
(2) Since each 1-bit ALU operation only allows two bits, by observing the ALU control value and operations, I have decided to assign ALU_ctl[2] and ALU_ctl[3] to every 1-bit ALU as the operation bits; ALU_ctl[1] and ALU_ctl[2] as a_invert and b_invert. This decision is based on the following observations:
For NOR operation: NOR = ~(a + b) = ~a • ~b. The values of a_invert and b_invert for NOR are both 1. The operation bits for NOR are 00. By performing multiplication operation, we can achieve ~a • ~b.
For SUB operation: SUB = a - b = a + (~b) + 1. The values of a_invert and b_invert for SUB are 0 and 1 respectively. The operation bits for SUB are 10. It performs addition operation, and by combining the invert values and operation bits, it computes the sum of A and the two's complement of B, which achieves a-b.

(3) If incorrect signals are assigned to the operation codes, the SUB operation will not be achieved through addition, and the NOR operation will not be achieved through AND. Consequently, the ALU will perform incorrect operations. This will lead to inaccurate output results.

(4) False.
In a MIPS datapath, the register file is typically accessed in separate stages for reading and writing, which ensures that read and write operations do not conflict with each other. Read stage is typically occurs early in the clock cycle, while write stage is occurs later in the clock cycle.

3. Experimental result

(1) Waveform screen shot of testbench(including more testcases)



(2) Other cases:

```
0xffff0000 0x0000ffff AND 0x00000000 1 0  // each line contains 1 case: {a b OP result [zero overflow]} separated by space
0x3113c398 0x088e4954  OR 0x399fcbdc 0 0  // numbers `a`, `b`, `result` can be hex(0x), binary(0b) or decimal
        -1          1 ADD          0 1 0  // `OP` must be either AND, OR, ADD, SUB, SLT or NOR
0x7eda5023 0x2ec36ae5 SUB 0x5016e53e 0 0  // `zero` and `overflow` must be either 0 or 1
0x7eda5023 0x2ec36ae5 SUB 0x5016e53e      // `zero` and `overflow` can be null (only test `result`)
        -1          1 SLT          1 0 0
0x00000000 0x00000000 NOR 0xffffffff 0 0  // try to add your own case!
0x40000000 0x40000000 ADD 0x80000000 0 1  //overflow
0x80000000 0x80000000 ADD 0x00000000 1 1  //overflow and zero detection
0x7fffffff 0xffffffff SUB 0x80000000 0 1  //overflow
0x7fffffff 0x7fffffff SLT 0 1 0           //SLT =
0x7fffffff 0xffffffff SLT 0 1 0           //SLT >
0x80000000 0x00000001 SLT 1 0 0           //SLT <
```

4. Problems

   I spent a lot of time linking wires to the registers. The most challenging part is deciding what signal the operation bit should be. I also spent a lot of time dealing with the OS. Since I am using a Macbook, it is a big challenge to install Vivado on my Mac. I successfully ran Vivado on my Mac by installing a virtual machine. However, errors occurred when I simulated my modules, and the simulation was interrupted frequently.

   Moreover, one test case will fail when I rerun it. It is weird and complicated to discover the problem since it may occur in the virtual machine or Windows in the arm version or the version of Vivado. Eventually, I borrowed a Windows laptop to finish this homework.

5. Feedback

   I found out that designing a CPU is different from writing a high-level program; it is not abstract since I need to monitor every wire and clearly understand how to assign wires and registers. I know how to design an alu by this homework.