# Description of My Design Implementation

- ECE 550 Project Checkpoint2
- Name: Rui Cao
- netID: rc384

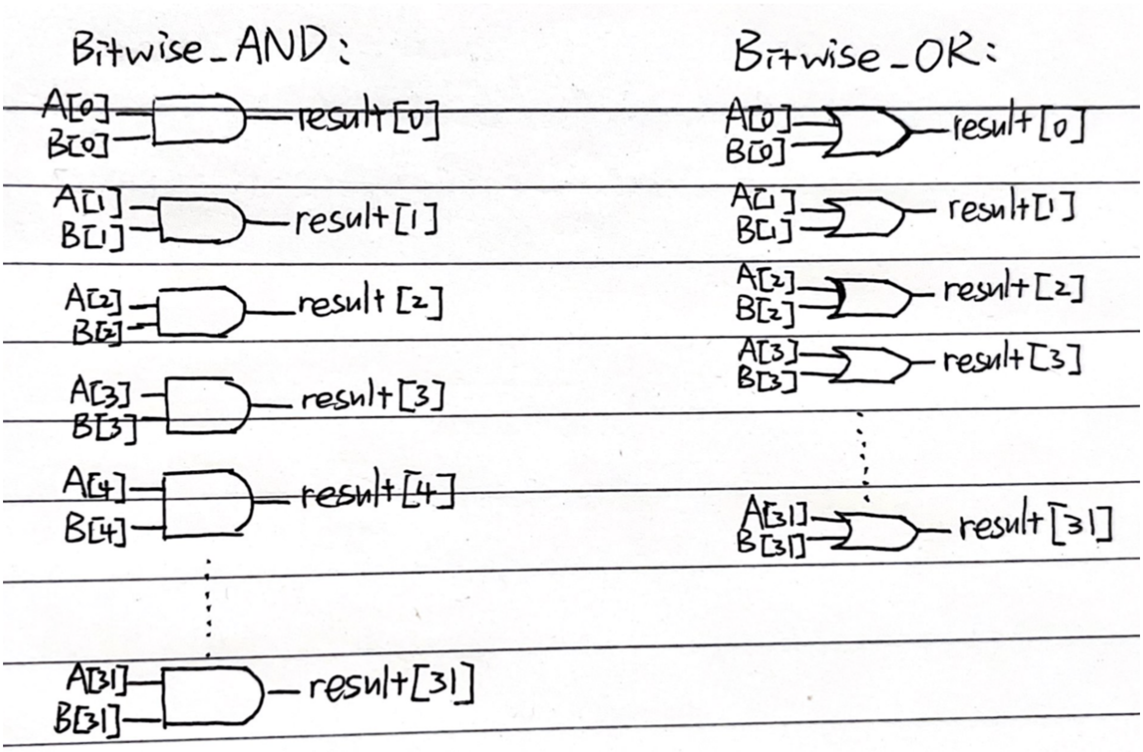
## 1. Overall Design

- Bitwise AND (data_operandA & data_operandB): bitwise_and.v
- Bitwise OR (data_operandA | data_operandB): bitwise_or.v
- Logical left-shift on data_operandA: sll.v
- Arithmetic right-shift on data_operandA: sra.v
- isNotEqual based on the result of subtract operation (from previous checkpoint): isNotEqual.v
    - Asserts true iff data_operandA and data_operandB are not equal
- isLessThan based on the result of subtract operation: isLessThan.v
    - Asserts true iff data_operandA is strictly less than data_operandB

## 2. Detailed Description

### 2.1. Bitwise AND:

Run the AND operation on the corresponding bits of the two operands of A and B. The left part of the figure below shows the circuit of the bitwise AND operation. Perform AND operation on each corresponding bit of A and B, and get result of each bit position. For example, A[0] & B[0] = result[0].




### 2.2. Bitwise OR:

Run the OR operation on the corresponding bits of the two operands of A and B. The right part of the figure above shows the circuit of the bitwise OR operation. Perform OR operation on each corresponding bit of A and B, and get result of each bit position. For example, A[0] | B[0] = result[0].

### 2.3. Logical Left Shift:

- Notes: for unsigned 32-bit integer
- data_operandA << ctrl_shiftamt (binary number): Shift the bits of data_operandA to the left by ctrl_shiftamt (base 10) positions, bringing in 0s at right, excess bits “fall off”. For example, 0001 << 10. the bits of 0001 are shifted to the left by 2 (10 is converted to base 10), so the result is 0100.
- When ctrl_shiftamt is a 5-bit binary number, we can do like this (Shift the bits of operand A 5 times):
    - Step 1: Shift the bits of data_operandA to the left by ctrl_shiftamt[0]*2^0 positions, and get result_after_shiftamt0 which is the result after the first shift;
    - Step 2: Shift the bits of result_after_shiftamt0 to the left by ctrl_shiftamt[1]*2^1 positions, and get result_after_shiftamt1 which is the result after the second shift;
    - Step 3: Shift the bits of result_after_shiftamt1 to the left by ctrl_shiftamt[2]*2^2 positions, and get result_after_shiftamt2 which is the result after the third shift;
    - Step 4: Shift the bits of result_after_shiftamt2 to the left by ctrl_shiftamt[3]*2^3 positions, and get result_after_shiftamt3 which is the result after the fourth shift;
    - Step 5: Shift the bits of result_after_shiftamt3 to the left by ctrl_shiftamt[4]*2^4 positions, and get the final result which is the result after the fifth shift.
- The five steps above can be translated as:
    - Step 1: if ctrl_shiftamt[0]=1, left shift 2^0 bit; if ctrl_shiftamt[0]=0, no changes;
    - Step 2: if ctrl_shiftamt[1]=1, left shift 2^1 bits; if ctrl_shiftamt[1]=0, no changes;
    - Step 3: if ctrl_shiftamt[2]=1, left shift 2^2 bits; if ctrl_shiftamt[2]=0, no changes;
    - Step 4: if ctrl_shiftamt[3]=1, left shift 2^3 bits; if ctrl_shiftamt[3]=0, no changes;
    - Step 5: if ctrl_shiftamt[4]=1, left shift 2^4 bits; if ctrl_shiftamt[4]=0, no changes.

### 2.4. Arithmetic (or signed) Right Shift:

- Notes: for signed 32-bit integer
- data_operandA >> ctrl_shiftamt (binary number): Shift the bits of data_operandA to the right by ctrl_shiftamt (base 10) positions, bringing in sign bit at left, excess bits “fall off”. For example, 0101 >> 10. the bits of 0101 are shifted to the right by 2 (10 is converted to base 10), so the result is 0001.
- Different from logical left shift, arithmetic right shift has two situations: (1) operand A is positive, so the most significant bit (MSB) of A is 0. (2) operand A is negative, so the MSB of A is 1. In different cases, the bits added on the left are different, which means bringing in sign bit at left.
- For different situations, perform the same steps below:

- Step 1: if ctrl_shiftamt[0]=1, right shift 2^0 bit and brining in sign bit (0 or 1) at left; if ctrl_shiftamt[0]=0, no changes;
- Step 2: if ctrl_shiftamt[1]=1, right shift 2^1 bits and brining in sign bit (0 or 1) at left; if ctrl_shiftamt[1]=0, no changes;
- Step 3: if ctrl_shiftamt[2]=1, right shift 2^2 bits and brining in sign bit (0 or 1) at left; if ctrl_shiftamt[2]=0, no changes;
- Step 4: if ctrl_shiftamt[3]=1, right shift 2^3 bits and brining in sign bit (0 or 1) at left; if ctrl_shiftamt[3]=0, no changes;
- Step 5: if ctrl_shiftamt[4]=1, right shift 2^4 bits and brining in sign bit (0 or 1) at left; if ctrl_shiftamt[4]=0, no changes.

### 2.5. isNotEqual:

- We will use the result of subtract operation (A-B=result) to decide whether data_operandA and data_operandB are not equal.
- First, according to our class PPT, we know that if overflow occurs, there are two situations: (1) operand A is negative and B is positive; (1) operand A is positive and B is negative. Therefore, when overflow occurs (overflow = 1), A and B are not equal.
- Second, if there is no overflow (overflow = 0) and A - B = result = 0, we can conclude that A = B. Therefore, I use OR gate to decide whether the 32 bits of result and the 1 bit of overflow are all 0s.

### 2.6. isLessThan:

- We will use the result of subtract operation (A-B=result) to decide whether data_operandA is strictly less than data_operandB.
- First, according to our class PPT, we know that if overflow occurs in subtraction operation, there are two situations: (1) operand A is negative and B is positive; (1) operand A is positive and B is negative. Therefore, when overflow occurs (overflow = 1), if the MSB (sign bit) of A is 1, A is positive, so A > B; if the MSB (sign bit) of A is 0, A is negative, so A < B.
- Second, if there is no overflow (overflow = 0), if A - B = result < 0, A < B; if A - B = result >= 0, A >= B.