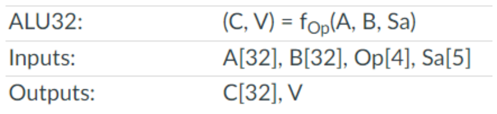
COMP2020 ALU Design

Vu Hong Phuc

October 19, 2021

1 ALU Overview

In this document is design principles for a RISC-V 32-bit ALU with 4-bit instruction. This 32-bit ALU has 4 inputs and 2 outputs with operator depends on the value of Op input. C[32] is the result from the operation and V is the overflow detection for add and subtract operators.

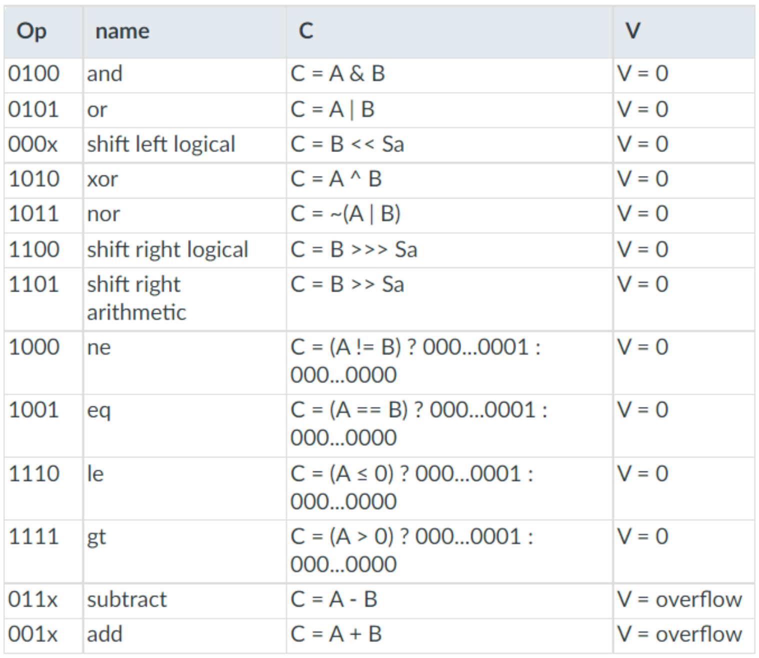


ALU32:	(C, V) = f_{Op}(A, B, Sa)
Inputs:	A[32], B[32], Op[4], Sa[5]
Outputs:	C[32], V

Figure 1: ALU input/output

There are total 13 operations with 4-bit opcode and they are divided into 4 main operation groups:

- Arithmetic Operator: subtract and add.
- Comparison Operator: ne, eq, le, and gt.
- Logical Operator: and, or, xor, and nor.
- Shift Operator: shift left logical, shift right logical, and shift right arithmetic.



Op	name	C	V
0100	and	C = A & B	V = 0
0101	or	C = A | B	V = 0
000x	shift left logical	C = B << Sa	V = 0
1010	xor	C = A ^ B	V = 0
1011	nor	C = ~(A | B)	V = 0
1100	shift right logical	C = B >>> Sa	V = 0
1101	shift right arithmetic	C = B >> Sa	V = 0
1000	ne	C = (A != B) ? 000...0001 : 000...0000	V = 0
1001	eq	C = (A == B) ? 000...0001 : 000...0000	V = 0
1110	le	C = (A ≤ 0) ? 000...0001 : 000...0000	V = 0
1111	gt	C = (A > 0) ? 000...0001 : 000...0000	V = 0
011x	subtract	C = A - B	V = overflow
001x	add	C = A + B	V = overflow

Figure 2: ALU details

We define bit numbering for opcode, starting at zero for the least significant bit (LSB) called Op0, and ending at 3 for the most significant bit (MSB) called Op3. In the table 1, the first row with 2 bits has the form of Op3-Op2 and the first column with 2 bits has the form of Op1-Op0. From the table, we can deduce which operator subcircuit in figure 3 will be activated.

	00	01	10	11
00	S	L	C	S
01	S	L	C	S
10	A	A	L	C
11	A	A	L	C

A	Arithmetic
C	Comparison
L	Logical
S	Shift

Table 1: Opcode table with 4 main operator groups

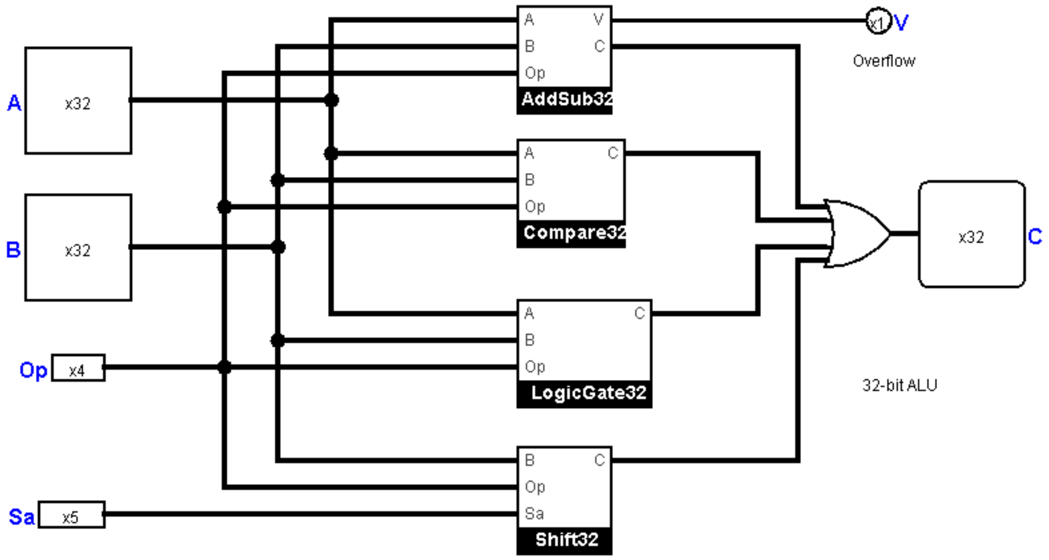


Figure 3: 32-bit ALU

2 Arithmetic Operator - AddSub32

The AddSub32 subcircuit consists of 2 operators: add and subtract. From table 1, this subcircuit would be active if and only if (Op3 == 0) and (Op1 == 1). Otherwise the outputs C[32] and V will be 0. Figure 4 demonstrates the implementation of this subcircuit with V denoting the overflow detection.

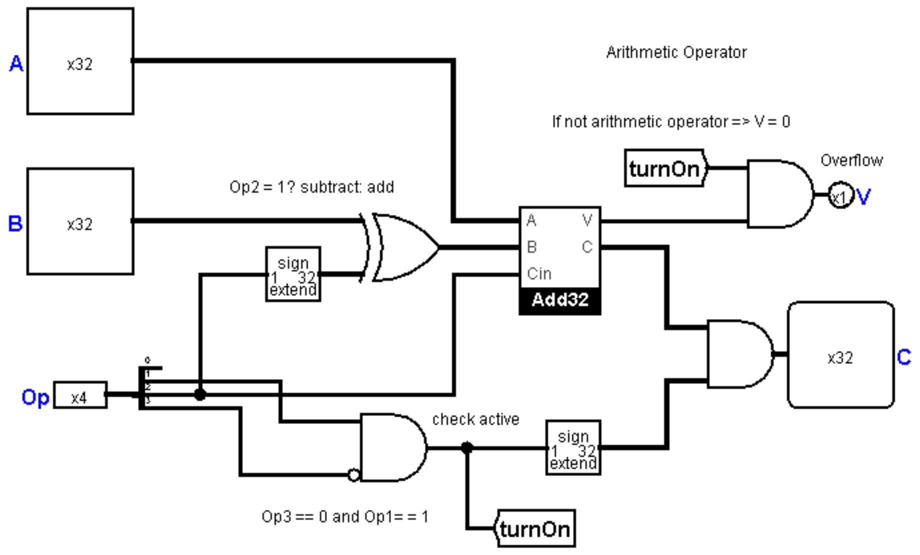


Figure 4: AddSub32 Subcircuit

The operation is decided to be add or subtract depends on the Op2 as in table 2. B[32] will be inverted with Cin is 1 if it is subtract operation before being the inputs for Add32 subcircuit.

Op2	Operator	Cin	A	B
0	add	0	A	B
1	subtract	1	A	∼B

Table 2: Arithmetic Operator Truth Table

2.1 32-bit Adder

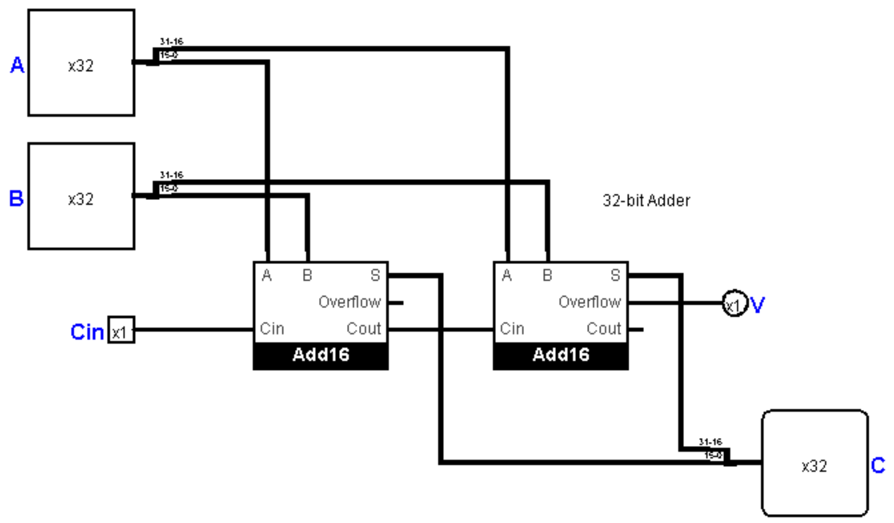
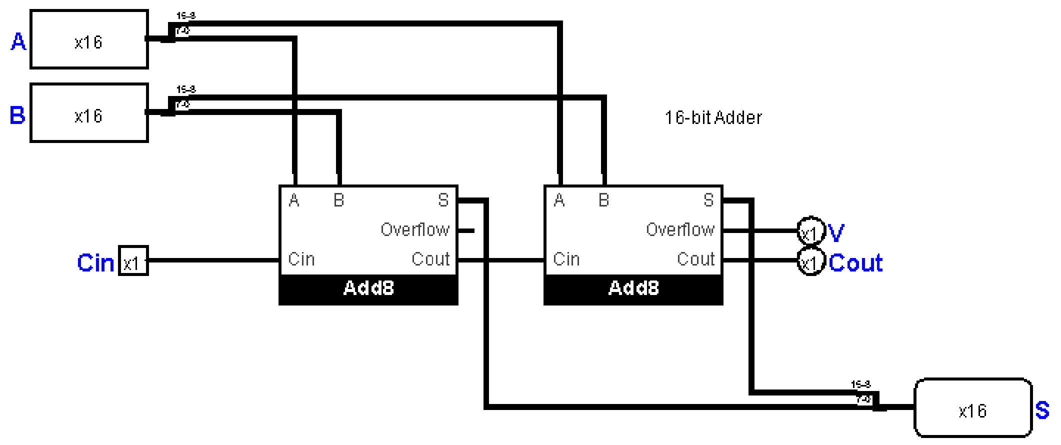


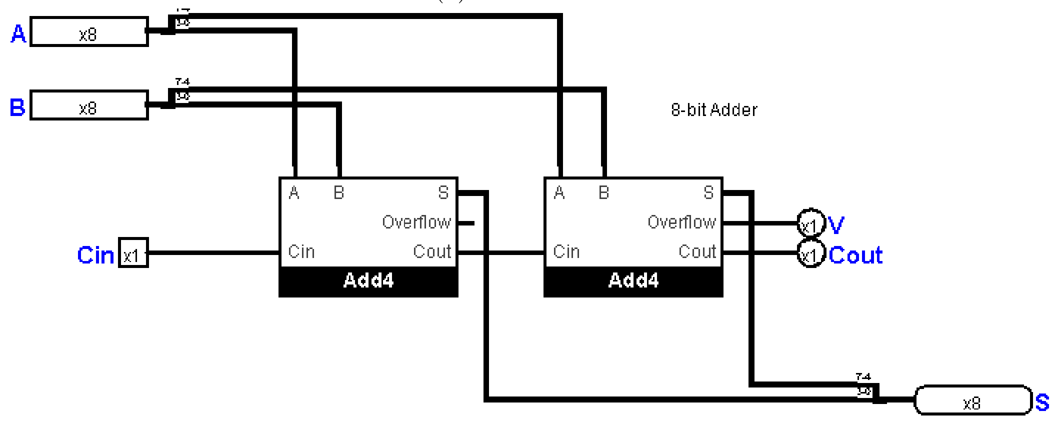
Figure 5: 32-bit Adder

This 32-bit Adder is formed with 2 16-bit Adder and performs the addition of 2 32-bit inputs A and B, returns S[32] for sum and V for overflow detection.

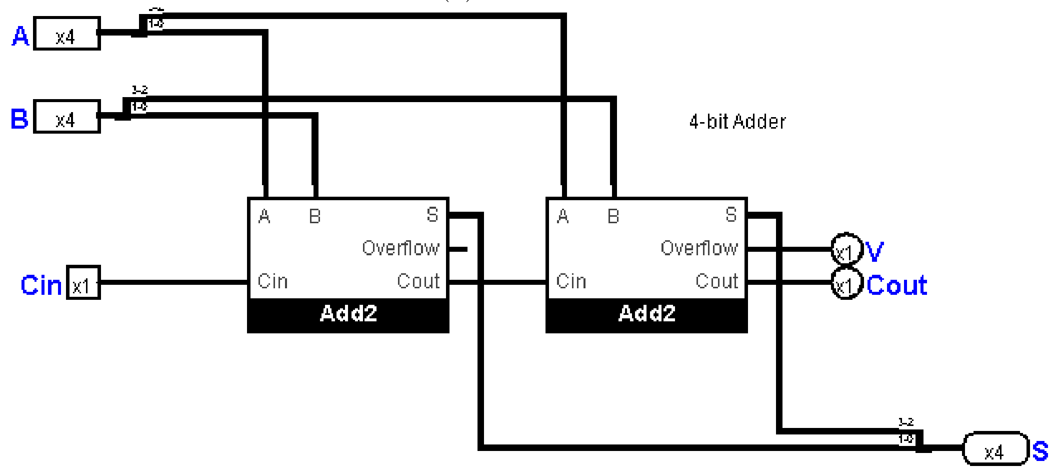
Similarly, Figure 6 shows 16-bit Adder is formed with 2 8-bit Adder, and so on.



(a) 16-bit Adder



(b) 8-bit Adder



(c) 4-bit Adder

Figure 6: Bit Adder

2.2 2-bit Adder

Combining 2 1-bit full Adders results in a 2-bit Adder with the output V for overflow detection (Cin msb != Cout msb).

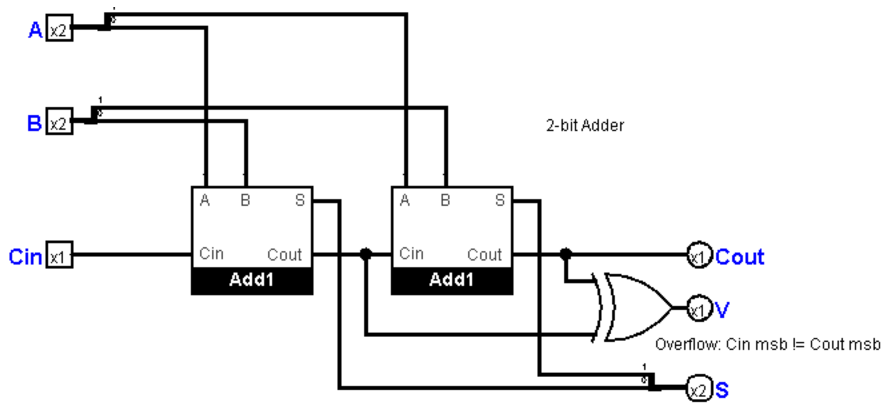


Figure 7: 2-bit Adder

2.3 1-bit Adder

The 1-bit full adder is to perform the addition of two 1-bit inputs with a carry bit Cin. This circuit is necessary for the construction of more advanced full adders as in above examples.

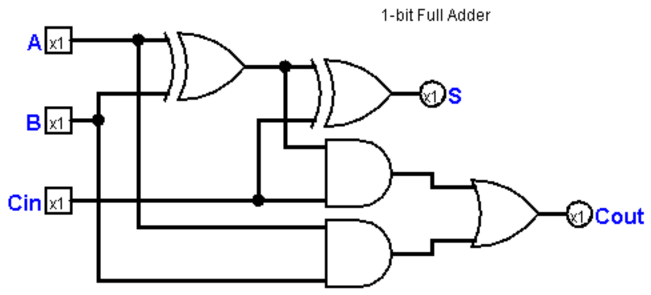


Figure 8: 1-bit Full Adder

In figure 8, A and B are the 1-bit inputs, while C_{in} is the carry-in bit. S is the output bit, while C_{out} is the carry-out bit. Figure 8 is the optimal circuit for the following truth table:

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 3: 1-bit Full Adder Truth Table

3 Comparison Operator - Compare32

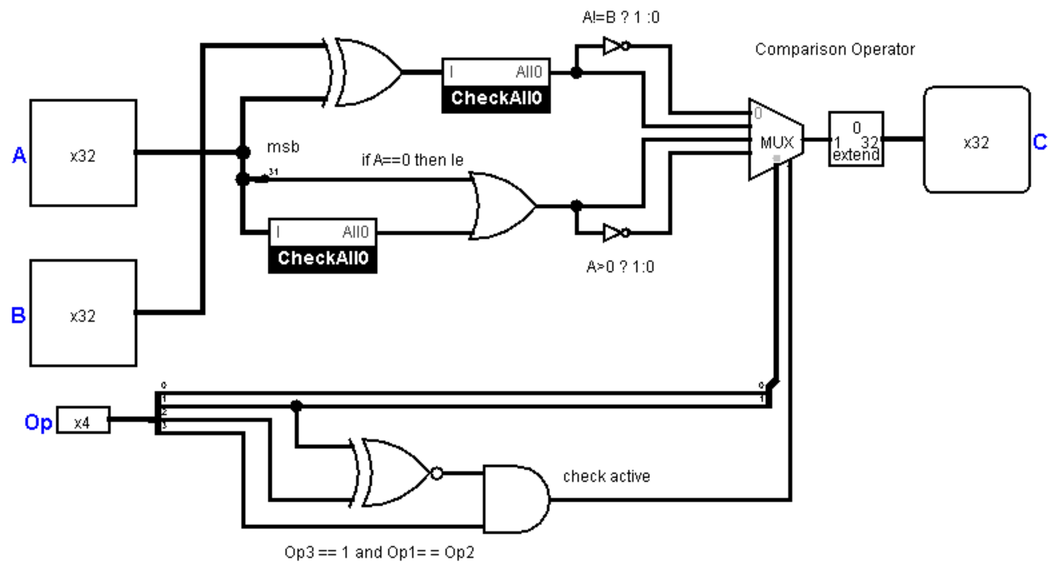


Figure 9: Compare32 Subcircuit

The Compare32 subcircuit consists of 4 operators which are ne, eq, le, and gt. From table 1, this subcircuit would be active if and only if (Op3 == 1) and (Op1 == Op2). Otherwise the outputs C[32] will be 0. Figure 9 demonstrates the implementation of this subcircuit.

The operation will be decided based on Op0 and Op1 as in table 4.

Op1	Op0	Operator	
0	0	ne	A != B
0	1	eq	A == B
1	0	le	$A \leq 0$
1	1	gt	$A > 0$

Table 4: Value Comparison

In the evaluation of ne and eq operators, XOR gate is used, followed by a CheckAll0 subcircuit to check if all bits of A and B are equal. The CheckAll0 subcircuit return 1 if the input equals 0.

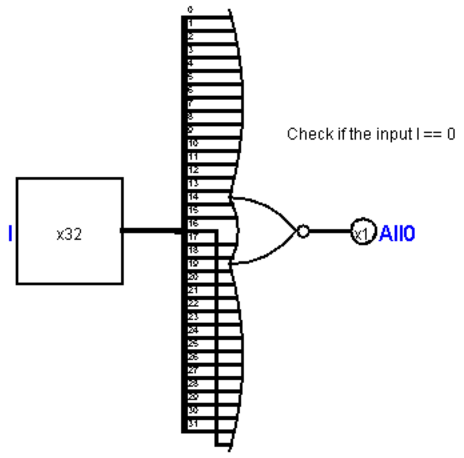


Figure 10: CheckAll0 Subcircuit

CheckAll0	A==B	A!=B
0	1	0
1	0	1

Table 5: Equality Truth Table

In terms of operation le and gt, MSB is the sign of input A[32]: leading 1's for negative and 0's for positive. Therefore, only MSB of A will be taken into consideration along with CheckAll0 subcircuit for the case A is zero. From the truth table 6, OR gate will be used for le and gt operators.

MSB	CheckAll0	A $\leq$ 0	A > 0
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Table 6: Compare to 0 Truth Table

4 Logical Operator - LogicGate32

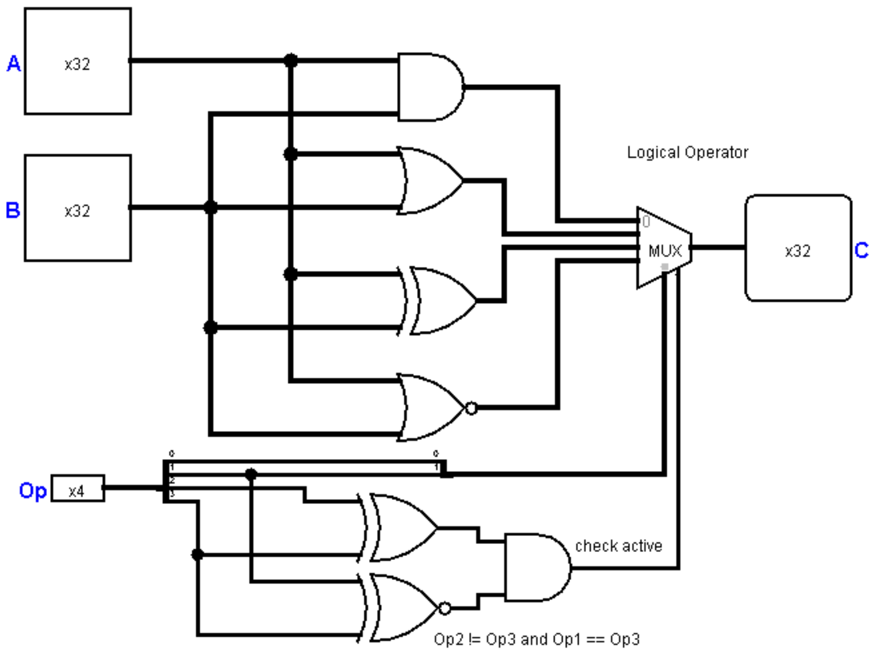


Figure 11: LogicGate32 Subcircuit

The LogicGate32 subcircuit consists of 4 operators which are and, or, xor, and nor. From table 1, this subcircuit would be active if and only if (Op2 != Op3) and (Op1 == Op3). Otherwise the outputs C[32] will be 0. Figure 11 demonstrates the implementation of this subcircuit.

These 4 logical operators will be decided based on Op0 and Op1 as in table 7.

Op1	Op0	Operator	
0	0	and	A & B
0	1	or	A $\mid B$
1	0	xor	A $\oplus B$
1	1	nor	$\neg(A \mid B)$

Table 7: Logical Operation

5 Shift Operator - Shift32

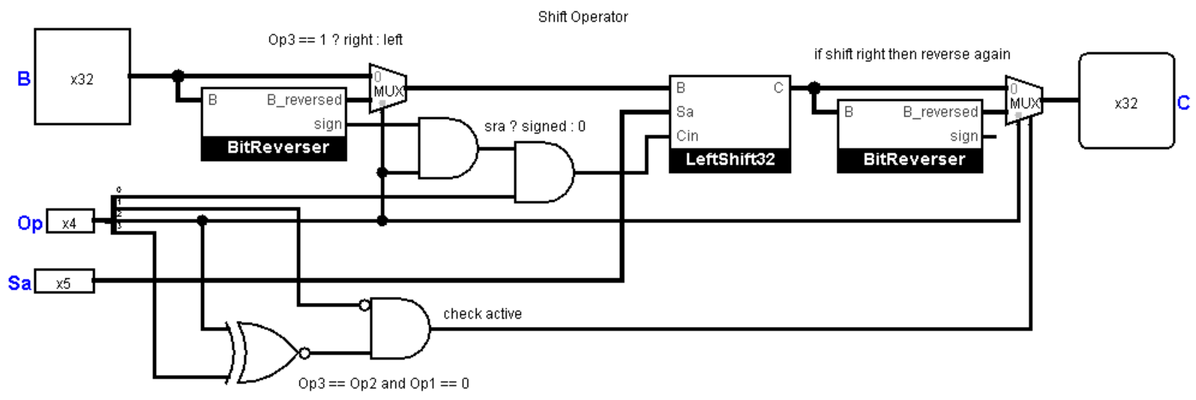


Figure 12: Shift32 Subcircuit

The Shift32 subcircuit consists of 3 operators which are shift left logical (SLL), shift right logical (SRL), and shift right arithmetic (SRA). From table 1, this subcircuit would be active if and only if (Op3 == Op2) and (Op1 == 0). Otherwise the outputs C[32] will be 0. Figure 12 demonstrates the implementation of this subcircuit.

5.1 LeftShift32 Subcircuit

The LeftShift32 subcircuit return output C[32] = (B << Sa) | carrybits with carrybits is Sa copies of Cin. Sa[5] is used to control the shift in 5 stages: shift 16, 8, 4, 2, 1 bits so that this subcircuit can shift any desired amount.

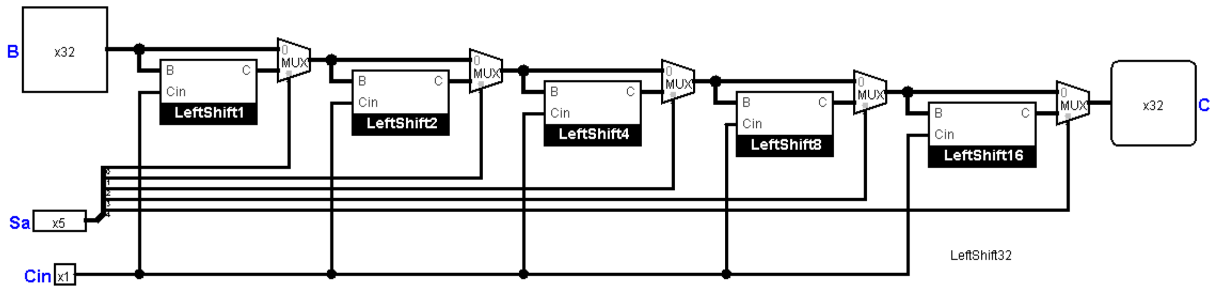
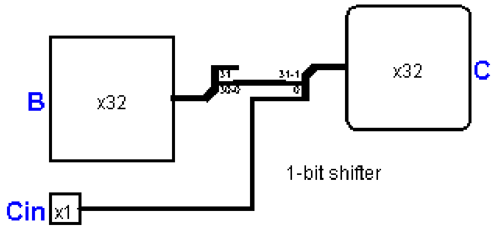
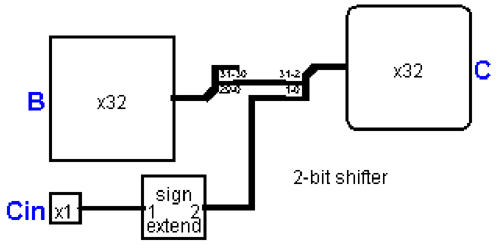


Figure 13: LeftShift32 Subcircuit

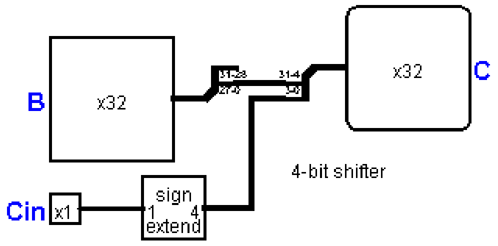
Figure 14 shows the design of all shiftleft subcircuits used in LeftShift32 subcircuit.



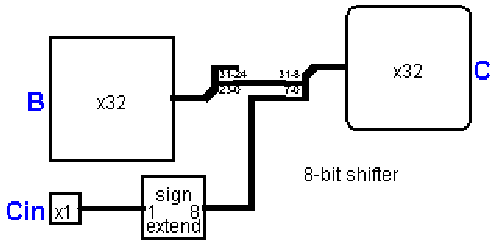
(a) LeftShift1



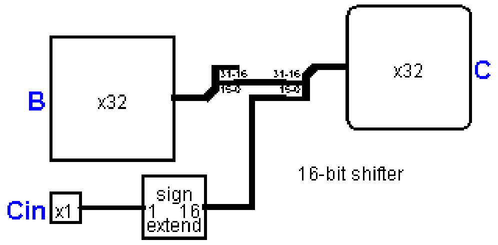
(b) LeftShift2



(c) LeftShift4



(d) LeftShift8



(e) LeftShift16

Figure 14: LeftShift Subcircuits

5.2 BitReverser Subcircuit

The BitReverser subcircuit reverses the input B[32] and it is used to execute right shift operator given the leftshift subcircuit. In addition, the output sign is the sign of input B will be used for SRA operator.

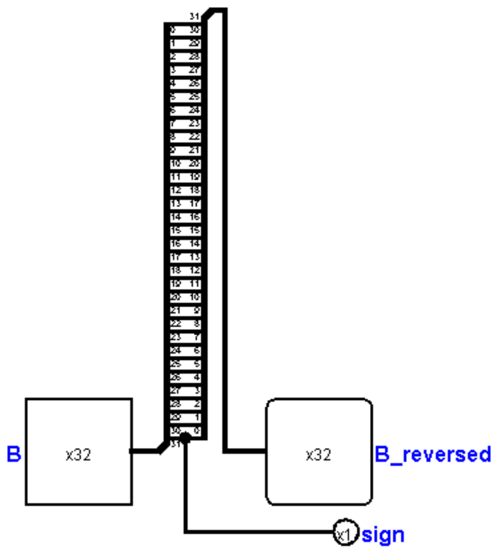


Figure 15: BitReverser Subcircuit

5.3 Shift32 Evaluation

Each shift operator's state depends on Op2 and Op0 as in table 8. From the table, the first MUX will decide whether the shift operator is left or right based on Op2. If shift right then the input B of LeftShift32 will be reversed.

Op2	Op0	SLL	SRL	SRA	Cin	B
0	0	1	0	0	0	B
0	1	1	0	0	0	B
1	0	0	1	0	0	B_reversed
1	1	0	0	1	B_sign	B_reversed

Table 8: Shift Truth Table

Regarding SRL and SRA, the difference would be the leftmost bit as the arithmetic shift preserves the sign of the input. Therefore, if it is right shift, get the sign of the input (first AND gate of Op2 and the sign of input B) then it is arithmetic if Op0==1, input Cin of LeftShift32 will take the sign of the input B (the second AND gate of the first AND gate and Op0) as in figure 12.

After getting the result from LeftShift32 subcircuit, if the operator is right shift then reverse the output again.

6 Evaluation

Overall, for each opcode input Op, only one subcircuit in ALU returns the right value and the other subcircuits return 0. Therefore, an OR gate is used to get the output C[32]. However, it can be seen from table 1 that this opcode might be not optimal as it is impossible to differentiate one subcircuit from another in ALU with 1 bit in opcode.

7 ALU with Carry-lookahead Adder

In ALU32_CLA, 32-bit carry-lookahead adder is used instead of 32-bit ripple-carry adder (CLA) for faster and effective design. The problem with ripple-carry adder is the calculation of Ci, taking linear time to the number of bits of the input. With carry-lookahead adder, the propagation delay is reduced as the carry ouputs depends on the initial carry bit.

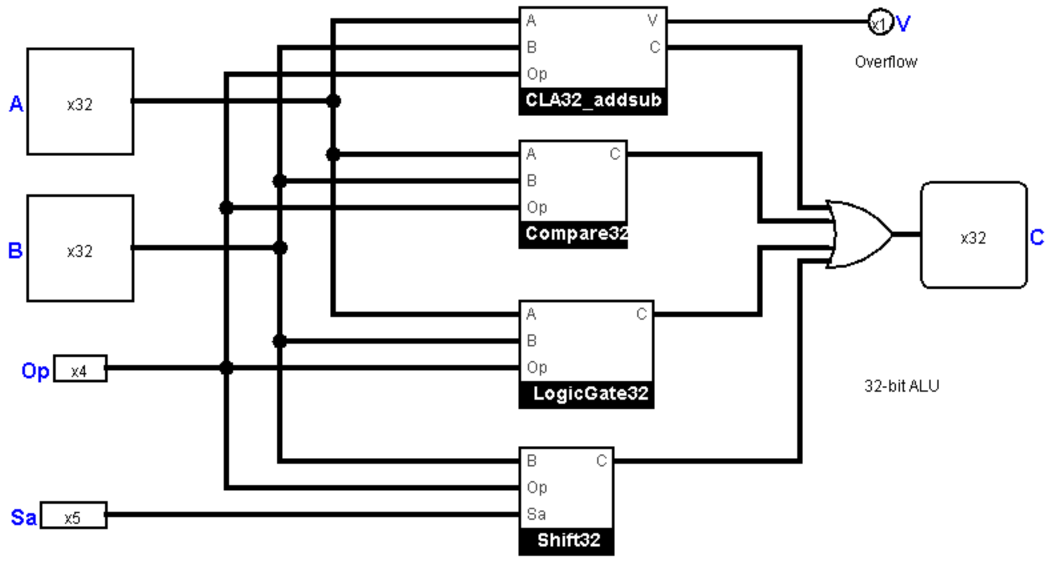


Figure 16: 32-bit ALU with CLA

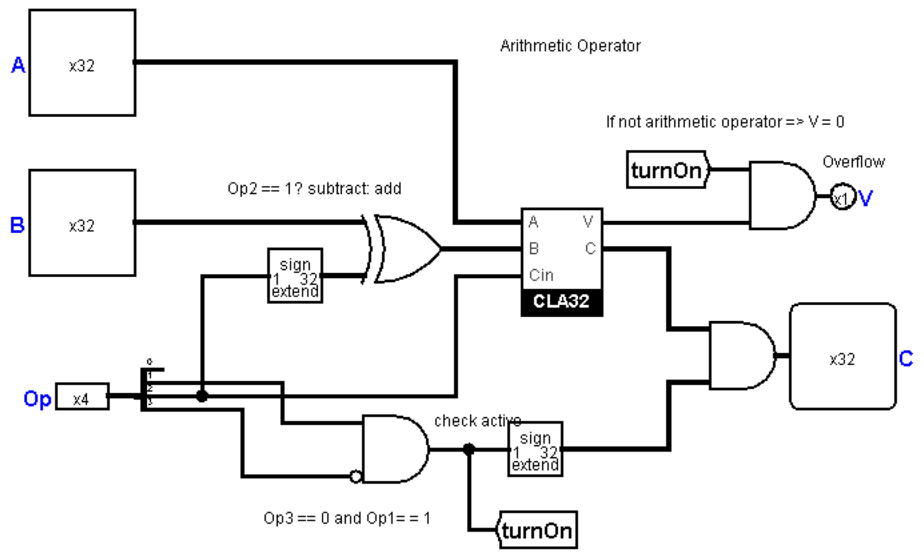


Figure 17: CLA32_addsub Subcircuit

In figure 18 the 32-bit CLA subcircuit is implemented by cascading 2 16-bit CLA. In figure 19 16-bit CLA is implemented by cascading 4 4-bit CLA.

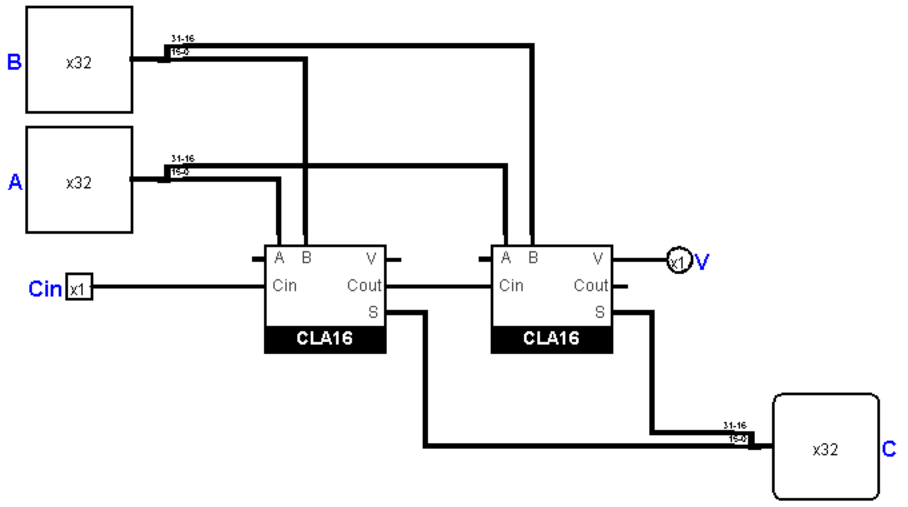


Figure 18: CLA32 Subcircuit

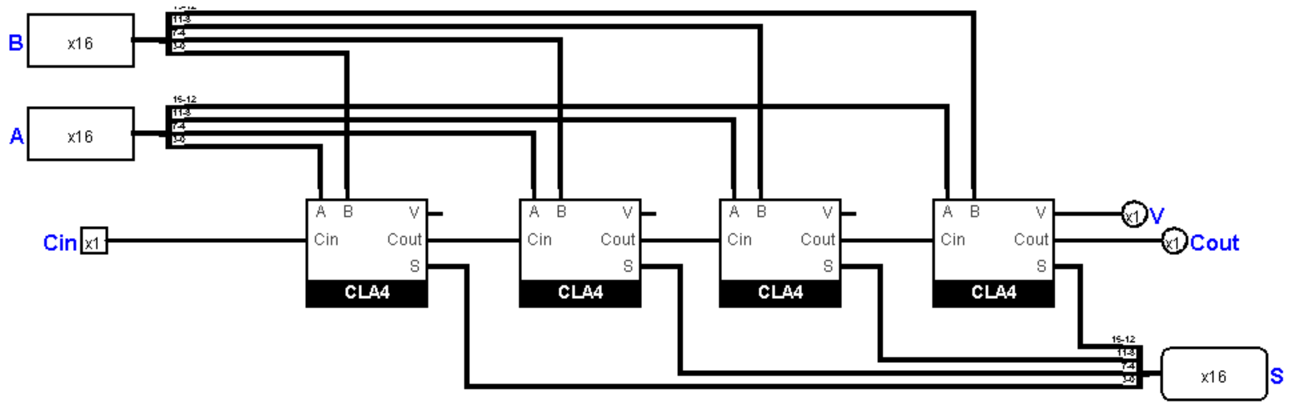


Figure 19: CLA16 Subcircuit

Carry-lookahead Adder uses generating and propagating carries.

- Propagate Signal: $P_i = A_i \oplus B_i$.
- Generate Signal: $G_i = A_i B_i$
- Carry output $C_{i+1} = G_i + P_i C_i$
- Sum $S = P_i \oplus C_i$

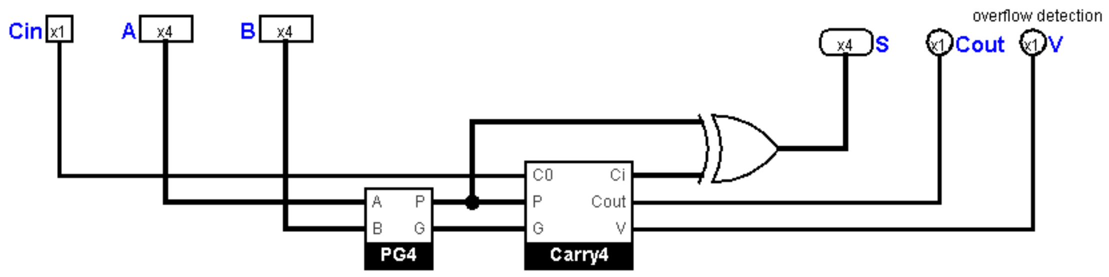


Figure 20: CLA4 Subcircuit

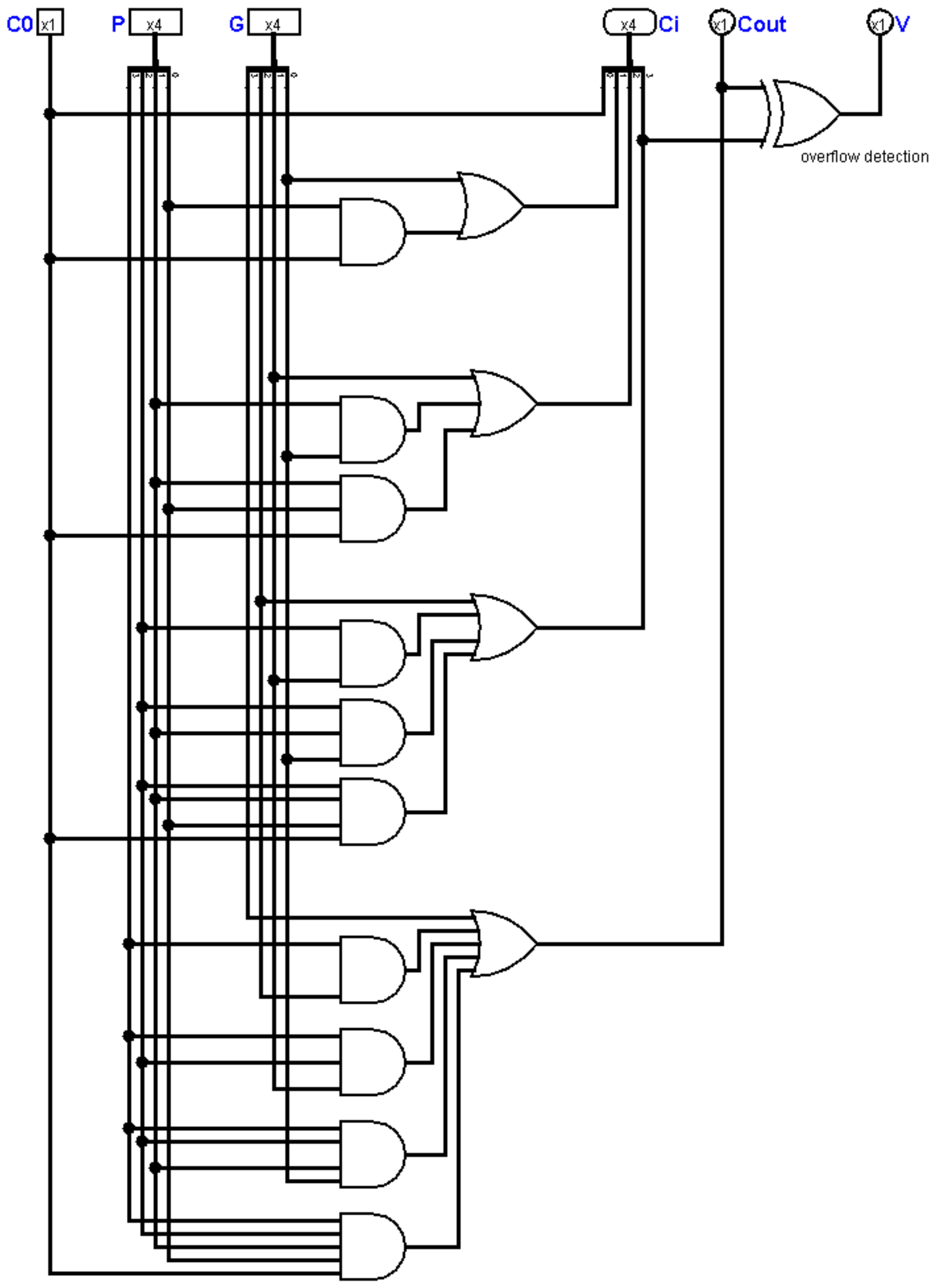


Figure 21: Carry4 Subcircuit

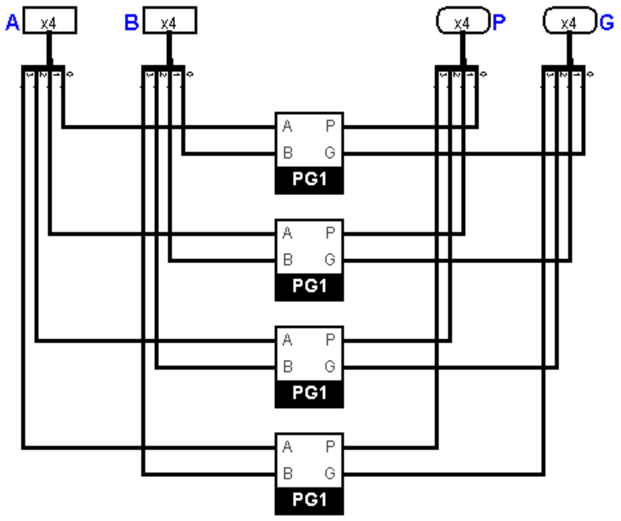


Figure 22: PG4 Subcircuit

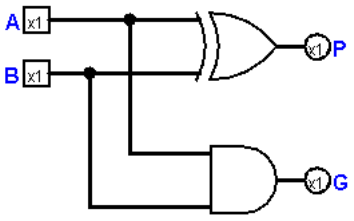


Figure 23: PG1 Subcircuit