Individual Project: Building a 32b ALU

ECE 361: Computer Architecture I

Manuel Blanco Valentin[1]

[1]Northwestern University, manuelvalentin2028@u.northwestern.edu

Disclaimer[1]: All symbols and figures have been made by the author using Adobe Illustrator tool.

Disclaimer[2]: Notice that the use of the second form of the plural in this work (we/us/our) is purely decorative due to the cultural background of the author and his tendency to nosism. **No one else but the author presented in the top of this document contributed to this project or the implementation of any of its parts.**

Contents

Section 1. *Project definition*

The objective of this project is to design a 32-bit ALU which will be later integrated into a single-cycle processor. It must be able to handle a small portion of the MIPS instruction set. Specifically, it must support the following operations (with their respective code, where *s* stands for source register, or *rs*, *t* for target register, or *rt*, *d* for destination or deposit register, or *rd*, *h* for the shift amount and *x* represents non-care bits).

(a) Arithmetic

i) add (with overflow): 0000 00ss ssst tttt dddd d000 0010 0000

ii) sub (subtraction with overflow): 0000 00ss ssst tttt dddd d000 0010 0010

(b) Logical

i) and (logical and): 0000 00ss ssst tttt dddd d000 0010 0100

ii) xor (exclusive logical or): 0000 00ss ssst tttt dddd dxxx xx10 0110

iii) or (logical or): 0000 00ss ssst tttt dddd d000 0010 0101

iv) sll (shift left logical): 0000 00ss ssst tttt dddd dhhh hh00 0000

v) srl (shift right logical): 0000 00xx xxxt tttt dddd dhhh hh00 0010

(c) Conditional

i) slt (set on less than): 0000 00ss ssst tttt dddd d000 0010 1010

ii) sltu (set on less than unsigned): 0000 00ss ssst tttt dddd d000 0010 1011

The instructions require certain provided components to be used (*eecs361lib_alu*). These components and their functionality can be found in Table 1.

Filename	Functionality
and_gate.v	a & b (bitwise AND, for single bit a and b)
and_gate_32.v	a & b (bitwise AND, for 32 bit a and b)
and_gate_n.v	a & b (bitwise AND, for n-bit a and b, where n is user-defined)
mux.v	2to1 MUX implemented with 2 bits of selection and two single-bit inputs
mux_n.v	2to1 MUX implemented with 2 bits of selection and two n-bit inputs, where n is user-defined
mux_32.v	2to1 MUX implemented with 2 bits of selection and two 32bit inputs
nand_gate.v	~ (a & b) bitwise NAND, for single bit a and b
nand_gate_n.v	~ (a & b) bitwise NAND, for n-bit a and b, where n is user defined
nand_gate_32.v	~ (a & b) bitwise NAND, for 32bit a and b
nor_gate.v	~ (a | b) bitwise NOR, for single bit a and b
nor_gate_32.v	~ (a | b) bitwise NOR, for 32 bit a and b
nor_gate_n.v	~ (a | b) bitwise NOR, for n-bit a and b, where n is used defined
not_gate.v	~ a bitwise NOT, for single bit a
not_gate_32.v	~ a bitwise NOT, for 32 bit a
not_gate_n.v	~ a bitwise NOT, for n-bit a, where n is user defined
or_gate_32.v	(a | b) bitwise OR, for 32 bit a and b
or_gate_n.v	(a | b) bitwise OR, for n-bit a and b, where n is used defined
or_gate.v	(a | b) bitwise OR, for single bit a and b
xnor_gate.v	bitwise XNOR for single bit a and b
xnor_gate_32.v	bitwise XNOR for 32 bit a and b
xnor_gate_n.v	bitwise XNOR for n-bit a and b, where n is used defined
xor_gate.v	(a ^ b) bitwise XOR, for single bit a and b
xor_gate_32.v	(a ^ b) bitwise XOR, for 32 bit a and b
xor_gate_n.v	(a ^ b) bitwise XOR, for n-bit a and b, where n is used defined

Table 1. Summary of the files included in the basic libraries used in project from which all the rest of the components of the ALU are built. The main functionality of each one of the modules implemented on each file is also find in this table.

The ALU that we have to design, which can be seen in Figure 1, must have the following IO signals:

1. inputs:
   (a) Operand A (32b)
   (b) Operand B (32b)
   (c) Control Signal (at least 4b) (In this project 11b were used)
2. outputs:
   (a) Result (32b)
   (b) Carryout (1b)
   (c) Overflow (1b)
   (d) Zero (1b)

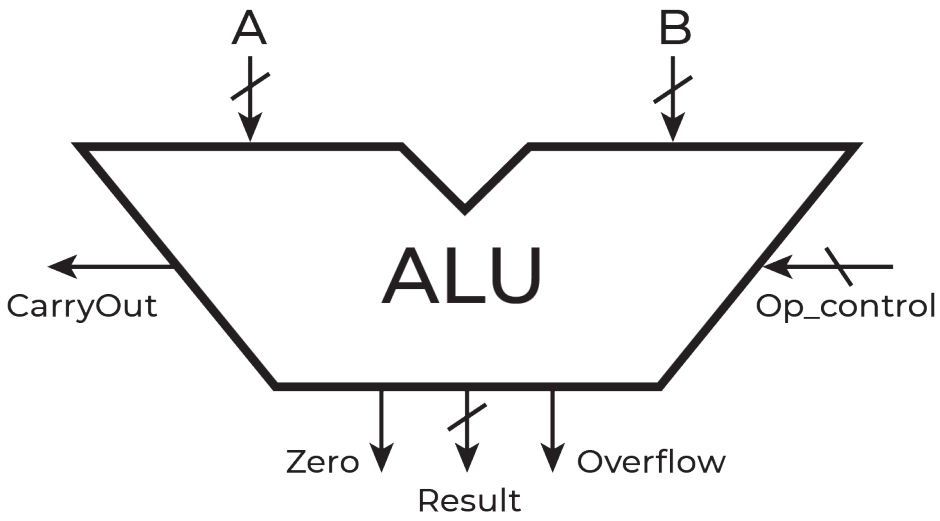

Figure 1. Diagram of the 32-bit arithmetic unit (ALU) designed in this project. Three buses input the unit: two 32-bit operators (A and B) and one 11-bit control bus (Op_control). On the other hand, four different signals output the unit: the 32-bit result bus and three other single-bit result flags (carryout, zero and overflow).

Section 2. *Introduction to our design*

2.1 Building blocks of our ALU

The first thing we can notice when looking at the different operations we have to implement is that they can be categorized in different groups, depending on the type of functions they perform and the types of results they yield. Each one of these groups are expected to have some sort of self-similarity in the components that form their blocks, so that they, as a group, can be implemented similarly or even using some of the same actual components.

In this project we have divided our ALU into four different blocks, each of which will be implemented separately before being used and combined into the final design of the ALU. Each block will perform a certain specific group of operations (and no operation will be performed by more than one group). These groups are:

1. **Logic Unit**: *and*, *xor*, *or* (and also *nor*, although was not required by the project description).
2. **Comparison Unit**: *slt*, *srl*.
3. **Arithmetic Unit**: *add*, *sub*.
4. **Shift Unit**: *sll*, *srl*.

The implementation of each one of these blocks and the submodules that were required for each one of them are introduced in the following sections of this document.

2.2 Control signal specification

One of the most important things in our design is the definition of the control signals used to control the ALU operation. These signals specify the different operation to be performed at each moment, as well as some extra parameters to be considered when computing the operation itself. All the different operations implemented in the ALU use the same control bus: $Op\_control$.

As introduced before, $Op\_control$ is a 11-bit bus. The different bits of this bus and their use is shown in Figure 2. The meaning and functionality of each control signal is detailed in Table 2.

Control signal	Bit span	Functionality
shamt (5b)	[10 : 6]	Amount of shift to apply to a certain 32-bit operator. Ranges from 0 ("00000") to 31 ("11111").
comparison_sel (1b)	[3]	Selection bit used to select the source of the output result of the ALU. In this case, when *comparison_sel* is 0, the result of the arithmetic unit is chosen, otherwise the result of the comparison block is selected.
comb_adder_sel (1b)	[2]	Selection bit used to select the source of the output result of the ALU. In this case, when *comb_adder_sel* is 1, the result of the logic unit is connected to the output of the ALU, otherwise the result of the arithmetic/comparison blocks is selected (depending on *comparison_sel*).
arithm_sel (1b)	[5]	Selection bit used to select the source of the output result of the ALU. In this case, when *arithm_sel* is 0, the result of the shift unit is connected to the output of the ALU, otherwise the result of the rest of the blocks is selected (depending on *comb_adder_sel*).
unsigned_ctl (1b)	[0]	This signal is used inside the comparison unit to specify whether the operators should be considered signed or unsigned. If 0, the operators are considered to be signed, while they are considered to be unsigned if the value of this flag is 1.
lu_ctl (2b)	[1 : 0]	This bus is used in the logic unit to select from the different results performed simultaneously on the operators. To be more precise, 00 is used to select the *and* operator; 01 to select the *or*; 10 to select the *xor*; and 11 to select the *nor*.
b_inv (1b)	[1]	This flag is used in the arithmetic unit to specify whether the operator B should be negated or not. As it will be explained in the following sections, this is useful when performing subtraction (and comparison, thus this flag is also used indirectly by the comparison unit). Specifically, if *b_inv* is 0, the arithmetic unit performs the addition of operators $A + B$; while if it is 1, it performs the subtraction $A - B$.
shift_to_right (1b)	[1]	This signal is used in the shift unit to specify whether the shift *shamt* executed is expected to be applied to the right or the left. Specifically, if 0 a left shift is performed, otherwise a right shift is performed.

Table 2. Detail of the different signals used to control the operation of the ALU, their bit span and their functionality.

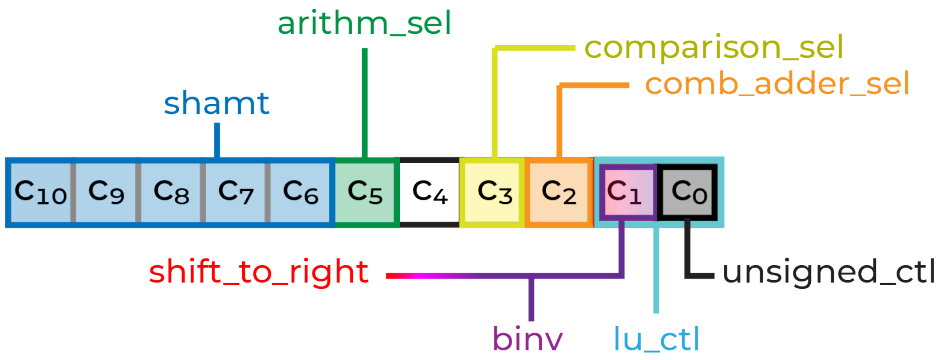

Figure 2. Detail of the different signals used to control the operation of our ALU. Each color represents a specific signal. Notice that bits can be shared by more than one signal (i.e., *b_inv* and *shift_to_right* share bit 1).

Section 3. *Designing the 32-bit logic unit*

3.1 Characterizing the logic operations

As specified in Section 2.1, there are four different operations that our logic unit (henceforth **LU**) is required to perform: *and*, *xor*, *or* and *nor*.

Each one of these operations can be implemented simply using logical gates. The library provided for the project already has the implementation required for all these functions:

1. The *and* operation will be implemented using the library module <*and_gate_32.v*>. This operation has a code 000000*ssssstttttddddd*00000100100.
2. The *xor* operation will be implemented using the library module <*xor_gate_32.v*>. This operation has a code 000000*ssssstttttddddd xxxxx*100110.
3. The *or* operation will be implemented using the library module <*or_gate_32.v*>. This operation has a code 000000*ssssstttttddddd*00000100101.
4. The *nor* operation will be implemented using the library module <*nor_gate_32.v*>. This operation has a code 000000*ssssstttttddddd*00000100111.

3.2 Design of the 4-to-1 MUX to control the block result

As the operations shown in the previous section require four different circuits, they will execute their operations simultaneously, which means that we will need to choose the right result to be output outside of the **LU** from among the result of all four gates.

The easiest way to implement a circuit to choose from different signals is to use a MUX [2]. In this case we need to choose from 4 operations, so we will need a 2-bit bus signal to control the MUX, thus we need to implement a 4-to-1 32-bit MUX.

If we look at the operation signal defined in Section 2.2, we can see that we had already defined the signal *lu_ctl* to select the result of the logic unit. One can notice that assigning bits $[1:0]$ to *lu_ctl* was no coincidence at all, as these two bits match precisely with the combination that we need to introduce to the MUX.

As we will need 2-to-1 MUXES in other parts of this project, and as 2-to-1 MUXES are so useful in many situations, we will implement our 4-to-1 MUX using 2-to-1 MUXES. The code for the implementation of our 2-to-1 MUX is shown in Code 1, and can be found in the submitted library as the file <*mux2to1_32.v*>. On the other hand, the code for the implementation of our 4-to-1 MUX is shown in Code 2, and can be found in the submitted library as the file <*mux4to1_32.v*>

Disclaimer[3]: The reason why the MUX given in the project library (<*mux_32.v*>) was not used here is simply because the selection signal of this module is a 32bit bus and as we only need 2 bits to control a 4-to-1 MUX it was considered more convenient to implement a different device with the appropriate selection signal size, instead of using the original device implemented in the file <*mux_32.v*>.

Code 1: Verilog code for the designed 2-to-1 32-bit MUX (<*mux2to1_32.v*>)

```
1   module mux2to1_32 (sel, src0, src1, z);
2     input sel;
3     input [31:0] src0;
4     input [31:0] src1;
5     output reg [31:0] z;
6
7     always @(sel or src0 or src1)
8        begin
9          if (sel == 1'b0) z <= src0;
10         else z <= src1;
11       end
12  endmodule
```

Code 2: Verilog code for the designed 4-to-1 32-bit MUX (*<mux4to1_32.v>*)

```
1   module mux4to1_32(sel, src0, src1, src2, src3, z);
2
3       input [1:0] sel;
4       input [31:0] src0;
5       input [31:0] src1;
6       input [31:0] src2;
7       input [31:0] src3;
8
9       output wire [31:0] z;
10
11      // Internal wires
12      wire [31:0] z_m0; //Contains either src0 or src1
13      wire [31:0] z_m1; //Contains either src2 or src3
14
15      // We can create a 4to1 mux using 3 pieces of 2to1 muxes
16      mux2to1_32 mux0 (.sel(sel[0]), .src0(src0), .src1(src1), .z(z_m0));
17      mux2to1_32 mux1 (.sel(sel[0]), .src0(src2), .src1(src3), .z(z_m1));
18      mux2to1_32 mux2 (.sel(sel[1]), .src0(z_m0), .src1(z_m1), .z(z));
19
20  endmodule
```

3.3 Implementation of the 32-bit LU

Figure 3 shows the diagram of the circuit to be implemented for the designed **LU**. As it can be seen, our **LU** consists of four different logic gates, each one of them performing a different logical operation on 32-bit operators *A* and *B*. The results of each gate go directly to the inputs of a 4-to-1 MUX (see Code 2 above). The signal *lu_ctl* controls the result to be selected. Signal *z* drives the final result (chosen via the *lu_ctl* signal) outside of the block.

The code that implements the logic unit shown in Figure 3 can be found in Code 3.

Code 3: Verilog code for the designed 32-bit Logic Unit (*<logic_unit_32.v>*)

```
1   module logic_unit_32(A, B, lu_ctl, z);
2       input [31:0] A;
3       input [31:0] B;
4       input [1:0] lu_ctl;
5       output wire [31:0] z;
6
7       // Create internal wiring for and, or, xor, nor results.
8       wire [31:0] and_z;
9       wire [31:0] or_z;
10      wire [31:0] xor_z;
11      wire [31:0] nor_z;
12
13      // AND (A & B) using and_gate_32.v
14      and_gate_32 and_g1(.x(A), .y(B), .z(and_z));
15
16      // OR (A | B) using or_gate_32.v
17      or_gate_32 or_g2(.x(A), .y(B), .z(or_z));
18
19      // XOR (A ^ B) using xor_gate_32.v
20      xor_gate_32 xor_g3(.x(A), .y(B), .z(xor_z));
21
22      // NOR ~(A | B) using nor_gate_32.v
23      nor_gate_32 nor_g4(.x(A), .y(B), .z(nor_z));
```

```
24
25          // Connect a 4to1 mux to select from AND/OR/XOR/NOR operations
26          mux4to1_32 logic_unit_mux(.sel(lu_ctl), .src0(and_z), .src1(or_z),
27                                    .src2(xor_z), .src3(nor_z), .z(z));
28
29      endmodule
```

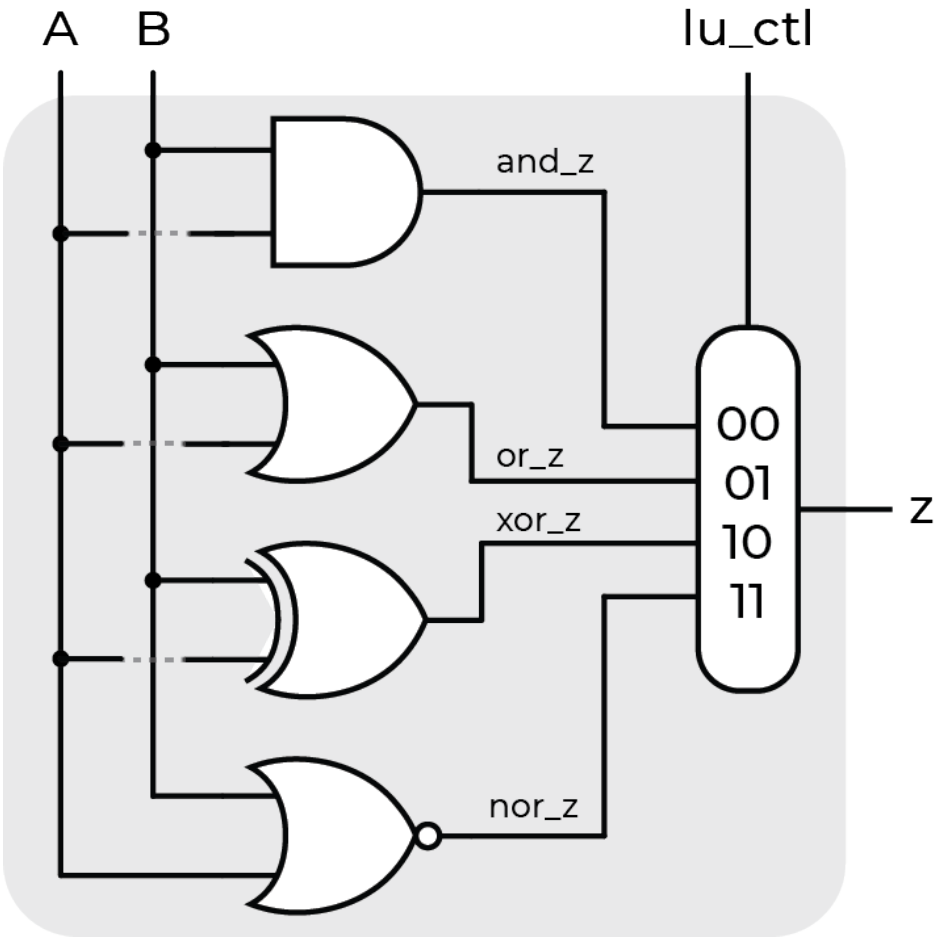

Figure 3. Diagram of the internal circuit of the Logic Unit (**LU**) implemented in this work.

Section 4. *Designing the 32-bit arithmetic unit*

4.1 Characterizing the arithmetic operations

As specified in Section 2.1, there are two different operations that our arithmetic unit (henceforth **AU**) is required to perform: Addition *add* and subtraction *sub*.

Let us start with addition first. If we consider two bits a and b, adding them together can only yield 3 different results:

(a) $a = 0$, $b = 0$: The result will be 0.

(b) Either $a = 1$, $b = 0$ or $a = 0$, $b = 1$: The result will be 1.

(c) $a = 1$, $b = 1$: The result will be 0.

Looking at the previous combination it is possible to realize that these results do not capture the true nature of the mathematical definition of addition. While the first two cases (a) and (b) yield complete results, the third one (c) seems to be missing the effect of bit overflow. When two bits are 1 and they are added, their value must "overflow" to the next significant bit, indicating the need to "carry" the extra bit of the result of the addition.

In order to perform this, an extra output is required while adding two binary numbers: the carry. The carry bit will be 0 for cases (a) and (b) mentioned before, while 1 for case (c). On the other hand, each significant bit to be added can also need to add the value of the carry bit computed in the previous significant bit position. This means that the result of the addition not only depends on the value of the operators a or b, but also on the value of the carry of the previous significant bit c_{in}. We can put it all together and show that the result z of the binary addition of two bits a and b, considering the carry bits can be specified by the truth-table shown in Table 3.

a	b	c_{in}	z	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. Truth Table showing the result z of performing full binary addition on two input bits a and b, taking into account input c_{in} and output c_{out} carry bits.

From the previous truth table it is possible to derive the combinatorial logic which performs these operations. The output z can be computed as (1), while the output carry signal c_{out} can be computed as (2),

$$z = a \oplus b \oplus c_{in} \tag{1}$$

$$c_{out} = (a \oplus b) \cdot c_{in} + a \cdot b \tag{2}$$

The combinatorial circuit that performs addition of two single bit operators A and B while taking into account the carry signal bits, as derived by the previous equations, is known as a **full adder** (henceforth **FA**). The circuit of a conventional **FA** is shown in Figure 4.

Although the circuit shown in Figure 4 is fully functional, there is a way it can be improved. Instead of using and extra AND and OR gates to implement the carry out signal, it is possible for us to use a 2-to-1 MUX (1b) for such purpose, as shown in Figure 5. By doing so we will achieve a much faster circuit (as MUX are faster than logic gates, see [4]).

Once the addition operation has been defined we can take a look at the implementation of the subtraction. We know that subtracting two operators A and B ($A - B$) is the same as adding A to the negated version of B ($A + (-B)$). When dealing with binary numbers, it is possible to obtain the inverse of a certain operator B by applying what's called as the two's-complement

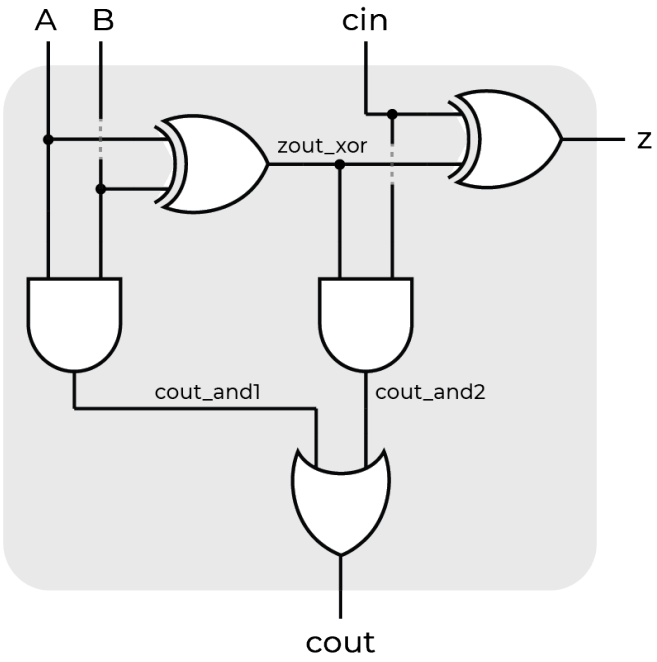

Figure 4. Diagram of the circuit for a conventional 32-bit full-adder.

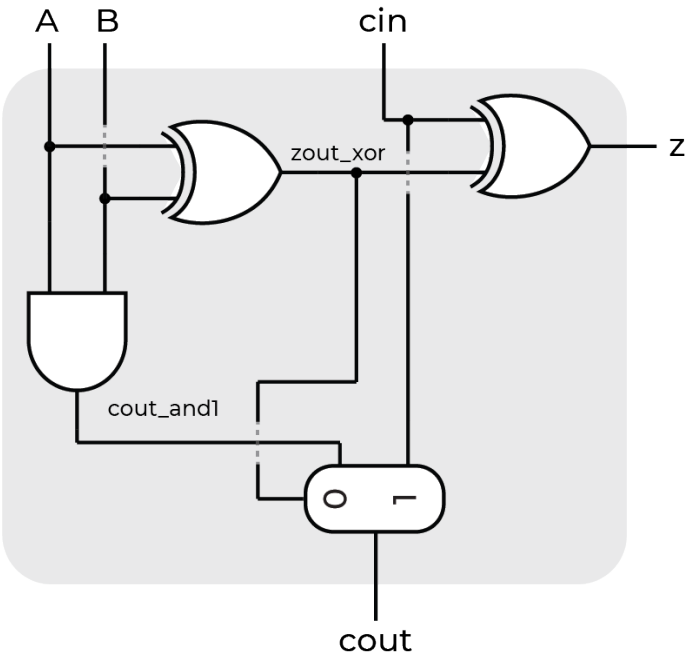

Figure 5. Diagram of the 32-bit full-fast-adder circuit implemented in this work.

[3], which consists in inverting each bit of the original operator B and adding one to its LSB. Thus, the relation between B and $-B$ can be written as (3), where $\sim$ represents the negation operator.

$$-B = (\sim B) + 1 \tag{3}$$

Let us now look back at the circuit we had designed to perform the addition operation, shown in Figure 5. Notice that it is possible to perform a subtraction, using this same circuit, by simply inverting B and setting c_{in} to 1. By applying these simple steps we reach the conclusion that both addition and subtraction can be performed by the full-fast-adder presented in Figure 5.

Selecting between both operations now simply consists on deciding whether to invert B or not and whether to set c_{in} to 0 or 1. The control code for each one of these operations is shown below:

1. This operation has a code 000000*ssssstttttddddd*00000100000.
2. This operation has a code 00000*ssssstttttddddd*00000100010.

If we take a look at the operation codes for these two functions, as described in Section 2.2, we realize that they only diverge on a single bit (#1), which is exactly the bit b_{inv} chosen to control the behavior of the arithmetic unit.

b_inv signal is expected to control if the block should perform addition or subtraction. When b_{inv} is low (0), the block will perform addition by non-inverting the B bits, and setting c_{in} to 0. On the other hand, when b_{inv} is high (1), the block will perform subtraction by inverting the B bits and setting c_{in} to 1.

Notice that in any case, the value of c_{in} of the first adder is always exactly the value of b_{inv}, which means that b_{inv} can be connected directly to c_{in} of the first adder block. On the other hand, we can force the bits of B to be inverted when b_{inv} is active by using *xor* gates.

The code that implements the full-fast-adder block for single-bit operators is shown in Code 4.

Code 4: Verilog code for the designed single-bit fast full adder (<*full_fast_adder.v*>)

```
1     module full_fast_adder(a, b, cin, z, cout);
2        // Define single-bit inputs
3        input a; // Operator A (1b)
```

```
4         input b; // Operator B (1b)
5         input cin; // Carry-in (1b)
6         // Define single-bit outputs
7         output wire z; // Result (1b)
8         output wire cout; // Carry-out (1b)
9
10        // Define internal wires
11        wire p; // Propagate carry out (for carry look-ahead)
12        wire g; // Generate carry out (for carry look-ahead)
13
14        // Define result
15        xor_gate z_g1 (.x(a), .y(b), .z(p));
16        xor_gate z_g2 (.x(p), .y(cin), .z(z));
17
18        // Define carry out
19        and_gate cout_g1 (.x(a), .y(b), .z(g));
20
21        // Use MUX
22        mux mux0 (.sel(p), .src0(g), .src1(cin), .z(cout));
23
24    endmodule
```

4.2 Implementation of the 32-bit AU

Taking into consideration what we discussed in the previous section, it is possible to setup the configuration for a 32-bit full-fast-adder by stacking several of the 1-bit full-fast-adder shown in Figure 5. In this configuration, the carry input c_{in} of each block is connected to the carry output c_{out} of the previous block, while the operator inputs of each block A_i, B_i are connected to each of the bits of the operators bus.

As discussed before, the carry in c_{in} of the first block is connected to b_{inv}, while the B inputs are negated by using *xor* gates connected to b_{inv}.

Nonetheless, our circuit is missing one important thing, which is the ability to detect **overflow**. Overflow represents the condition for which the result of the arithmetic operation performed by addition/subtraction cannot be represented by the number of bits used to represent the operators A and B used for the calculation. In example, trying to add the unsigned words 11 and 01 will result in overflow, as in order to represent the result it would be necessary to use (at least) 3-bit long words, instead of 2-bit ones. In summary, the overflow flag arises when the sign of the result of the operation is different from the combined signs of the input operators (such as adding two positive numbers and obtaining a negative result).

Detecting overflow is important to ensure fully functionality of our ALU. By arising overflow flags it is possible to anticipate and avoid any further errors that these results could cause. The expression that allows us to detect overflow from the last full-adder block is (4), where X_{31} represents the bit 31 of operator X.

$$overflow = A_{31} \cdot B_{31} \cdot \overline{Z_{31}} + \overline{A_{31}} \cdot \overline{B_{31}} \cdot Z_{31} \tag{4}$$

The previous expression can be implemented using three AND gates, one XOR, one inverter and one OR gate. The complete design for our arithmetic unit can be seen in Figure 6, while the code that implements its functionality can be found in Code 5. Notice that this code uses the definition of MUX found in the original library file *<mux.v>*.

Code 5: Verilog code for the designed 32-bit Arithmetic unit (*<arithmetic_unit_32.v>*)

```
1     module arithmetic_unit_32(A, B, b_inv, z, cout, overflow);
2
3         // Define inputs
4         input [31:0] A; // 32-bit operator A
5         input [31:0] B; // 32-bit operator B
6         input b_inv; // 1-bit control b_inv (add/sub). If true, invert B
7
```

```
8         // Define outputs
9         output wire [31:0] z; // output 32-bit result of add/sub
10        output wire cout; // 1-bit output carry
11        output wire overflow; // 1-bit output overflow flag
12
13        // b_inv basically controls whether the operation we have to perform is add
14        // (b_inv == 0) or subtraction (b_inv == 1)
15        // If we need to perform a subtraction then we have to calculate the
16        // 2-complement. Remember that (A-B) = A + (-B) = A + (!B + 1)
17        // Thus, if we have to subtract B from A, we need to invert B and add one.
18        // The trick here is to notice that we can perform that by using XOR gates.
19        // Remember that (x NOR y) = (xy' + x'y), thus if
20        // we set x-> B_i and y-> b_inv, if b_inv == 0, then we just input B_i to the
21        // adder, while if b_inv is 1, we invert B_i, which
22        // is exactly what we want. At the same time, that "1" we have to add in the
23        // case of the subtraction can be added if we set
24        // the carry_in of the first adder in the chain to b_inv. If b_inv is zero
25        // (perform addition), then this carry will be zero.
26        // But if we are performing a subtraction (b_inv == 1), then we will invert all
27        // B inputs (by the XOR gates) and adding one
28        // (by setting the carry_in of the first adder to b_inv, which in this case is
29        // one).
30
31        // Internal xor outputs
32        wire b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15;
33        wire b16, b17, b18, b19, b20, b21, b22, b23, b24, b25, b26, b27, b28, b29, b30
34        wire b31;
35
36        // Internal carries between adders
37        wire cout0, cout1, cout3, cout4, cout5, cout6, cout7, cout8, cout9, cout10;
38        wire cout11, cout12, cout13, cout14, cout15;
39        wire cout16, cout17, cout18, cout19, cout20, cout21, cout22, cout23, cout24;
40        wire cout25, cout26, cout27, cout28, cout29, cout30;
41
42        // Internal wires for overflow decection
43        wire and_of1;
44        wire nor_of1;
45        wire not_of1;
46        wire and_of2;
47        wire and_of3;
48
49        // CHAIN OF ADDERS
50
51        // First adder has b_inv as carry_in input
52        xor_gate xor_b0 (.x(B[0]), .y(b_inv), .z(b0));
53        full_fast_adder FA_0 (.a(A[0]), .b(b0), .cin(b_inv), .z(z[0]), .cout(cout0));
54
55        // Now we can just go on until adder 31
56
57        // ADDER 1
58        xor_gate xor_b1 (.x(B[1]), .y(b_inv), .z(b1));
59        full_fast_adder FA_1 (.a(A[1]), .b(b1), .cin(cout0), .z(z[1]), .cout(cout1));
60        // ADDER 2
61        xor_gate xor_b2 (.x(B[2]), .y(b_inv), .z(b2));
62        full_fast_adder FA_2 (.a(A[2]), .b(b2), .cin(cout1), .z(z[2]), .cout(cout2));
63        // ADDER 3
64        xor_gate xor_b3 (.x(B[3]), .y(b_inv), .z(b3));
65        full_fast_adder FA_3 (.a(A[3]), .b(b3), .cin(cout2), .z(z[3]), .cout(cout3));
```

```
66        // ADDER 4
67        xor_gate xor_b4 (.x(B[4]), .y(b_inv), .z(b4));
68        full_fast_adder FA_4 (.a(A[4]), .b(b4), .cin(cout3), .z(z[4]), .cout(cout4));
69        // ADDER 5
70        xor_gate xor_b5 (.x(B[5]), .y(b_inv), .z(b5));
71        full_fast_adder FA_5 (.a(A[5]), .b(b5), .cin(cout4), .z(z[5]), .cout(cout5));
72        // ADDER 6
73        xor_gate xor_b6 (.x(B[6]), .y(b_inv), .z(b6));
74        full_fast_adder FA_6 (.a(A[6]), .b(b6), .cin(cout5), .z(z[6]), .cout(cout6));
75        // ADDER 7
76        xor_gate xor_b7 (.x(B[7]), .y(b_inv), .z(b7));
77        full_fast_adder FA_7 (.a(A[7]), .b(b7), .cin(cout6), .z(z[7]), .cout(cout7));
78        // ADDER 8
79        xor_gate xor_b8 (.x(B[8]), .y(b_inv), .z(b8));
80        full_fast_adder FA_8 (.a(A[8]), .b(b8), .cin(cout7), .z(z[8]), .cout(cout8));
81        // ADDER 9
82        xor_gate xor_b9 (.x(B[9]), .y(b_inv), .z(b9));
83        full_fast_adder FA_9 (.a(A[9]), .b(b9), .cin(cout8), .z(z[9]), .cout(cout9));
84        // ADDER 10
85        xor_gate xor_b10 (.x(B[10]), .y(b_inv), .z(b10));
86        full_fast_adder FA_10 (.a(A[10]), .b(b10), .cin(cout9), .z(z[10]),
87           .cout(cout10));
88        // ADDER 11
89        xor_gate xor_b11 (.x(B[11]), .y(b_inv), .z(b11));
90        full_fast_adder FA_11 (.a(A[11]), .b(b11), .cin(cout10), .z(z[11]),
91           .cout(cout11));
92        // ADDER 12
93        xor_gate xor_b12 (.x(B[12]), .y(b_inv), .z(b12));
94        full_fast_adder FA_12 (.a(A[12]), .b(b12), .cin(cout11), .z(z[12]),
95           .cout(cout12));
96        // ADDER 13
97        xor_gate xor_b13 (.x(B[13]), .y(b_inv), .z(b13));
98        full_fast_adder FA_13 (.a(A[13]), .b(b13), .cin(cout12), .z(z[13]),
99           .cout(cout13));
100       // ADDER 14
101       xor_gate xor_b14 (.x(B[14]), .y(b_inv), .z(b14));
102       full_fast_adder FA_14 (.a(A[14]), .b(b14), .cin(cout13), .z(z[14]),
103          .cout(cout14));
104       // ADDER 15
105       xor_gate xor_b15 (.x(B[15]), .y(b_inv), .z(b15));
106       full_fast_adder FA_15 (.a(A[15]), .b(b15), .cin(cout14), .z(z[15]),
107          .cout(cout15));
108       // ADDER 16
109       xor_gate xor_b16 (.x(B[16]), .y(b_inv), .z(b16));
110       full_fast_adder FA_16 (.a(A[16]), .b(b16), .cin(cout15), .z(z[16]),
111          .cout(cout16));
112       // ADDER 17
113       xor_gate xor_b17 (.x(B[17]), .y(b_inv), .z(b17));
114       full_fast_adder FA_17 (.a(A[17]), .b(b17), .cin(cout16), .z(z[17]),
115          .cout(cout17));
116       // ADDER 18
117       xor_gate xor_b18 (.x(B[18]), .y(b_inv), .z(b18));
118       full_fast_adder FA_18 (.a(A[18]), .b(b18), .cin(cout17), .z(z[18]),
119          .cout(cout18));
120       // ADDER 19
121       xor_gate xor_b19 (.x(B[19]), .y(b_inv), .z(b19));
122       full_fast_adder FA_19 (.a(A[19]), .b(b19), .cin(cout18), .z(z[19]),
123          .cout(cout19));
```

```
124         // ADDER 20
125         xor_gate xor_b20 (.x(B[20]), .y(b_inv), .z(b20));
126         full_fast_adder FA_20 (.a(A[20]), .b(b20), .cin(cout19), .z(z[20]),
127            .cout(cout20));
128         // ADDER 21
129         xor_gate xor_b21 (.x(B[21]), .y(b_inv), .z(b21));
130         full_fast_adder FA_21 (.a(A[21]), .b(b21), .cin(cout20), .z(z[21]),
131            .cout(cout21));
132         // ADDER 22
133         xor_gate xor_b22 (.x(B[22]), .y(b_inv), .z(b22));
134         full_fast_adder FA_22 (.a(A[22]), .b(b22), .cin(cout21), .z(z[22]),
135            .cout(cout22));
136         // ADDER 23
137         xor_gate xor_b23 (.x(B[23]), .y(b_inv), .z(b23));
138         full_fast_adder FA_23 (.a(A[23]), .b(b23), .cin(cout22), .z(z[23]),
139            .cout(cout23));
140         // ADDER 24
141         xor_gate xor_b24 (.x(B[24]), .y(b_inv), .z(b24));
142         full_fast_adder FA_24 (.a(A[24]), .b(b24), .cin(cout23), .z(z[24]),
143            .cout(cout24));
144         // ADDER 25
145         xor_gate xor_b25 (.x(B[25]), .y(b_inv), .z(b25));
146         full_fast_adder FA_25 (.a(A[25]), .b(b25), .cin(cout24), .z(z[25]),
147            .cout(cout25));
148         // ADDER 26
149         xor_gate xor_b26 (.x(B[26]), .y(b_inv), .z(b26));
150         full_fast_adder FA_26 (.a(A[26]), .b(b26), .cin(cout25), .z(z[26]),
151            .cout(cout26));
152         // ADDER 27
153         xor_gate xor_b27 (.x(B[27]), .y(b_inv), .z(b27));
154         full_fast_adder FA_27 (.a(A[27]), .b(b27), .cin(cout26), .z(z[27]),
155            .cout(cout27));
156         // ADDER 28
157         xor_gate xor_b28 (.x(B[28]), .y(b_inv), .z(b28));
158         full_fast_adder FA_28 (.a(A[28]), .b(b28), .cin(cout27), .z(z[28]),
159            .cout(cout28));
160         // ADDER 29
161         xor_gate xor_b29 (.x(B[29]), .y(b_inv), .z(b29));
162         full_fast_adder FA_29 (.a(A[29]), .b(b29), .cin(cout28), .z(z[29]),
163            .cout(cout29));
164         // ADDER 30
165         xor_gate xor_b30 (.x(B[30]), .y(b_inv), .z(b30));
166         full_fast_adder FA_30 (.a(A[30]), .b(b30), .cin(cout29), .z(z[30]),
167            .cout(cout30));
168         // ADDER 31
169         xor_gate xor_b31 (.x(B[31]), .y(b_inv), .z(b31));
170         full_fast_adder FA_31 (.a(A[31]), .b(b31), .cin(cout30), .z(z[31]),
171            .cout(cout));
172
173         // Now we must calculate the overflow. Overflow arises when the sign of the
174         // result of the operation (add/sub) is not
175         // the same as the signs of the original operators.
176         and_gate and_g1 (.x(b31), .y(A[31]), .z(and_of1));
177         nor_gate nor_g1 (.x(b31), .y(A[31]), .z(nor_of1));
178         not_gate not_g1 (.x(z[31]), .z(not_of1));
179
180         and_gate and_g2 (.x(nor_of1), .y(z[31]), .z(and_of2));
181         and_gate and_g3 (.x(and_of1), .y(not_of1), .z(and_of3));
```

```
182
183         or_gate or_g1 (.x(and_of2), .y(and_of3), .z(overflow));
184
185     endmodule
```

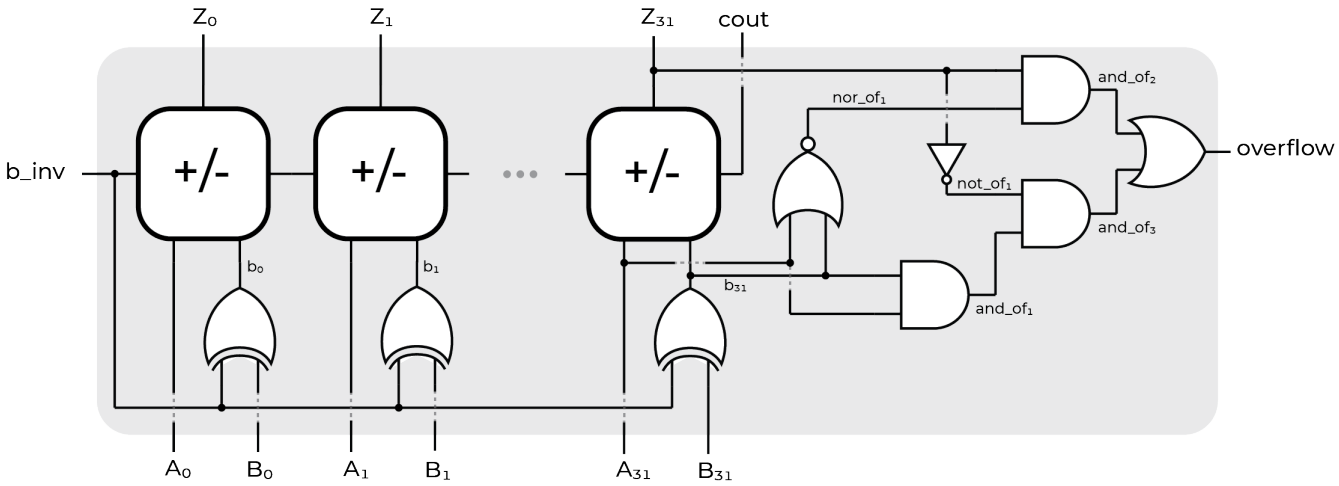

Figure 6. Diagram of the schematic used for the 32-bit arithmetic unit implemented in this work.

Section 5. *Designing the 32-bit comparison unit*

5.1 Characterizing the comparison operations

As specified in Section 2.1, there are two different operations that our comparison unit (henceforth **CU**) is required to perform: *slt*, *sltu*. The codes and an explanation on the functionality of each one of these operations can be found in:

1. The *slt* operation compares two **signed** operators A and B and returns 1 if $A < B$, or 0 otherwise. This operation has a code 000000*sssssttttt**ddddd*00000101010.

2. The *sltu* operation compares two **unsigned** operators A and B and returns 1 if $A < B$, or 0 otherwise. This operation has a code 000000*sssssttttt**ddddd*00000101011.

As it can be seen, both these operations require the use of the arithmetic unit to perform the subtraction of operators, so that the inequality between their values can be checked.

Assuming the subtraction $sub_Z = A - B$ has already been performed, and that our comparison unit has access to the operators original values A, B, as well as to the result sub_Z and the *carryout* of the subtraction, validate whether the two unsigned operators A and B are smaller or greater one from each other is as simple as checking whether the *carryout* signal is active or not. In common binary subtraction, the *carryout* signal is high (1) when the result is greater than 0 ($(A - B) > 0$), thus the signal *sltu* can be obtained by simply inverting the *carryout* signal.

On the other hand, the *slt* signal is a bit more complex to obtain. Checking whether two signed binary operators are greater or smaller one from each other requires to consider their signs before and after the operation. If operator A is negative ($A_{31} = 1$) and B is positive ($B_{31} = 0$), we immediately can ensure that the result will be negative, and thus that $slt = 1$. However, if the original signs of the operators were equal and the result is negative, we know that the condition $A < B$ is also true.

Therefore, the comparison unit is a purely combinatorial block which can be implemented using only logic gates as the ones provided by the project libraries. Both the results obtained by each operation is calculated simultaneously, which means that we can use a MUX to select between them and connect them to the output of the block, as we did in the blocks presented in previous sections. In any case, the diagram of the 32-bit comparison unit *CU* implemented in this work can be seen in Figure 7.

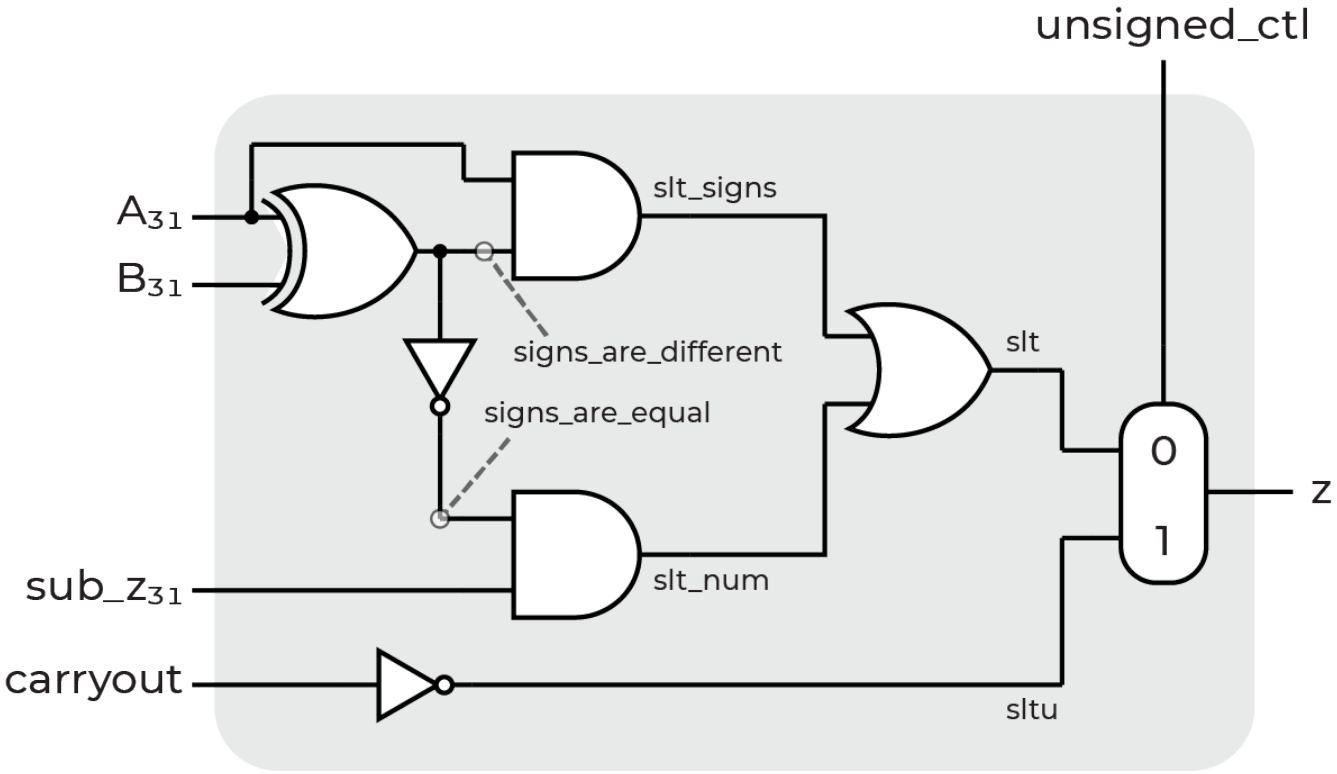

Figure 7. Diagram of the 32-bit comparison unit designed and implemented in this work for the 32-bit ALU here presented.

5.2 Implementation of the 32-bit CU

The code that implements the logic unit shown in Figure 3 can be found in Code 3. Notice that this code uses the definition of the 32-bit not logical gate found in the original library file *<not_gate_32.v>*.

Code 6: Verilog code for the designed 32-bit Comparison Unit (*<comparison_unit_32.v>*)

```
module comparison_unit_32(A, B, sub_z, carryout, unsigned_ctl, z);

    // Define inputs
    input [31:0] A;
    input [31:0] B;
    input [31:0] sub_z;
    input carryout;
    input unsigned_ctl;

    // Define outputs
    output wire [31:0] z;

    // Define internal signals
    wire [31:0] less;
    wire [31:0] less_u;
    wire signs_are_different;
    wire signs_are_equal;
    wire slt_signs;
    wire slt_num;
    wire slt;
    wire sltu;

    // Assign zeros to less
    assign less[31:1] = 31'b0;
    assign less_u[31:1] = 31'b0;

    // If the signs are the different, we simply have to check whether A is positive
    // or negative.
    // If A[31] == 0, that means A is positive, and thus B is negative, which means
    // that Less <= 0.
    // If A[31] == 1, that means A is negative, and thus B is positive, which means
    // that Less <= 1.
    // In any case we realize that if the signs are different, Less <= A[31].
    // Checking whether the signs are different can be implemented using a XOR gate.
    xor_gate xor_g1 (.x(A[31]), .y(B[31]), .z(signs_are_different));
    // signs_are_different is 0 if A == B, else 1
    and_gate and_g1 (.x(A[31]), .y(signs_are_different), .z(slt_signs));
    // slt_signs is 1 when A != B and A[31] == 1

    // Now, if the signs are equal, we have to check whether the sign of the result
    // is negative or positive. If the
    // sign of the result is negative (Res[31] == 1), it means Less <= 1.
    // On the other hand, if the sign of the result
    // is positive (Res[31] == 0), it means Less <= 0. So basically, Less <= Res[31].
    not_gate not_g1 (.x(signs_are_different), .z(signs_are_equal));
    and_gate and_g2 (.x(sub_z[31]), .y(signs_are_equal), .z(slt_num));

    // Finally connect everything
    or_gate or_g2 (.x(slt_signs), .y(slt_num), .z(slt));

    // Assign last bit to less
    assign less[0] = slt;
```

```
53
54          // Checking whether unsigned A < B is easier. If A and B are unsigned,
55          // then there is no representation
56          // of negative numbers (they cannot be negative). This means that the result of
57          // (A-B) can only be
58          // positive. If the carryout of (A-B) is 1, it means the result is indeed
59          // positive (carryout high)
60          // in subtraction is the normal value. However if carryout is 0, this means the
61          // result of the
62          // subtraction overflowed, which can only happen if and only if B > A. From this
63          // we conclude that
64          // less_u <= not(carryout)
65          not_gate not_g3 (.x(carryout), .z(sltu));
66          assign less_u[0] = sltu;
67
68          // Now wire-up a mux to decide whether we output the SLT or the SLTU result
69          mux2to1_32 mux0 (.sel(unsigned_ctl), .src0(less), .src1(less_u), .z(z));
70
71      endmodule
```

Section 6. *Designing the 32-bit shift unit*

6.1 Characterizing the shifting operations

As specified in Section 2.1, there are two different operations that our shift unit (henceforth **SU**) is required to perform: *sll*, *srl*. The codes and an explanation on the functionality of each one of these operations can be found in:

1. The *sll* operation shifts the 32-bit operator A (found in register s) to the left. It is a logical shift, which means that any LSB in the new shifted word will be filled with zeros. This operation has a code 000000*ssssstttttddddd*hhhhh000000. The amount of positions the word should be shifted is given by the 5b word determined by the h symbols in the previous code.

2. The *srl* operation shifts the 32-bit operator A (found in register s) to the right. It is a logical shift, which means that any MSB in the new shifted word will be filled with zeros. This operation has a code 000000*xxxxxttttt*ddddd*hhhhh*000010. The amount of positions the word should be shifted is given by the 5b word determined by the h symbols in the previous code.

Shifting a vector can be implemented by the use of concatenated MUXs. Let us imagine for a moment that we want to shift a certain vector A with n bits of length, one single position to the left. Then, the value of each position of the output shifted vector Z will correspond to the value of the input vector at one position shifted to the left. In other words, $Z[i+1] <= A[i]$ for all i in $[0..n]$. In this particular case, it is noticeable that choosing between a shift of 0 or 1 could be easily implemented by using 2-to-1 MUXES for each one of the bits of the vector. A shift of 0 would connect the output of each MUX $Z[i]$ to the original values $A[i]$, while a shift of 1 would make the output of the MUXES $Z[i]$ to be connected to the original values shifted by one $A[i-1]$.

Now, if we wanted to shift the original vector A two positions to the left, we could implement a specific circuit with MUXES dedicated to this unique case, however one can imagine the total cost that implementing all combinations for 32-bit input vectors would have. Instead, we could use the previously implemented sequence of 1-shift MUXES and stack another set of MUXES at its output. Now, each of these new row of MUXES would have its second input connected to the MUX two positions below itself (instead of one, as did the MUXES in the previous configuration).

By applying this idea we are able to achieve a total of 2^r shifting combinations to the original vector A, where r is the number of rows of MUXES stacked one after another. In our case, stacking 5 rows is enough to be able to shift the operators in the range $[0..31]$.

One last trick can be applied to this methodology to perform right shift instead of left shift, by using the same exact circuit introduced before. If we introduce one set of additional stack of MUXES before the first shifting row, and after the last shifting row, we will be able to flip the input operator A right before performing the shift, as well as to flip the shifted result back at the end.

Flipping our operators allow us to perform the right shift by using the same circuit used for the left shift. This behavior can be controlled by the $shift_to_right$ flag introduced to the **SU**, as explained in Section 2.2.

6.2 Implementation of the 32-bit SU

The code that implements the shift unit shown in Figure 8 can be found in Code 7. Notice that this code uses the definition of the 32-bit not logical gate found in the original library file *<not_gate_32.v>*.

Code 7: Verilog code for the designed 32-bit Shift Unit (*<shift_unit_32.v>*)

```
1    module shift_unit_32(A, shamt, shift_to_right, z);
2
3       // Define inputs
4       input [31:0] A;
5       // shamt amount ranges from 0 ("00000") to 31 ("11111")
6       input [4:0] shamt;
7       // Indicates the direction of the shamt. If 1, shamt to right, else to left.
8       input shift_to_right;
9
10      // Define outputs
11      output [31:0] z;
12
```

```
13        // Internal signals in case we need to shift to right
14        wire x31, x30, x29, x28, x27, x26, x25, x24, x23, x22, x21, x20, x19, x18, x17;
15        wire x16, x15, x14, x13, x12, x11, x10, x9, x8, x7, x6, x5, x4, x3, x2, x1, x0;
16        wire y31, y30, y29, y28, y27, y26, y25, y24, y23, y22, y21, y20, y19, y18, y17;
17        wire y16, y15, y14, y13, y12, y11, y10, y9, y8, y7, y6, y5, y4, y3, y2, y1, y0;
18
19        // Internal signals for first row of muxes outputs
20        wire r31, r30, r29, r28, r27, r26, r25, r24, r23, r22, r21, r20, r19, r18, r17;
21        wire r16, r15, r14, r13, r12, r11, r10, r9, r8, r7, r6, r5, r4, r3, r2, r1, r0;
22
23        // Internal signals for second row of muxes outputs
24        wire s31, s30, s29, s28, s27, s26, s25, s24, s23, s22, s21, s20, s19, s18, s17;
25        wire s16, s15, s14, s13, s12, s11, s10, s9, s8, s7, s6, s5, s4, s3, s2, s1, s0;
26
27        // Internal signals for third row of muxes outputs
28        wire t31, t30, t29, t28, t27, t26, t25, t24, t23, t22, t21, t20, t19, t18, t17;
29        wire t16, t15, t14, t13, t12, t11, t10, t9, t8, t7, t6, t5, t4, t3, t2, t1, t0;
30
31        // Internal signals for fourth row of muxes outputs
32        wire u31, u30, u29, u28, u27, u26, u25, u24, u23, u22, u21, u20, u19, u18, u17;
33        wire u16, u15, u14, u13, u12, u11, u10, u9, u8, u7, u6, u5, u4, u3, u2, u1, u0;
34
35        // Internal signals for fifth row of muxes outputs
36        wire v31, v30, v29, v28, v27, v26, v25, v24, v23, v22, v21, v20, v19, v18, v17;
37        wire v16, v15, v14, v13, v12, v11, v10, v9, v8, v7, v6, v5, v4, v3, v2, v1, v0;
38
39        // Assign LSB depending on type of shamt:
40        // Logical shift --> LSB <= 0
41        // Arithmetic shift --> LSB <= A[0]
42        // Rotation shift --> LSB <= depends on A
43        assign LSB = 1'b0;
44
45        // In case shift_to_right, make sure we flip inputs
46        mux mux_x31 (.sel(shift_to_right), .src0(A[31]), .src1(A[0]), .z(x31));
47        mux mux_x30 (.sel(shift_to_right), .src0(A[30]), .src1(A[1]), .z(x30));
48        mux mux_x29 (.sel(shift_to_right), .src0(A[29]), .src1(A[2]), .z(x29));
49        mux mux_x28 (.sel(shift_to_right), .src0(A[28]), .src1(A[3]), .z(x28));
50        mux mux_x27 (.sel(shift_to_right), .src0(A[27]), .src1(A[4]), .z(x27));
51        mux mux_x26 (.sel(shift_to_right), .src0(A[26]), .src1(A[5]), .z(x26));
52        mux mux_x25 (.sel(shift_to_right), .src0(A[25]), .src1(A[6]), .z(x25));
53        mux mux_x24 (.sel(shift_to_right), .src0(A[24]), .src1(A[7]), .z(x24));
54        mux mux_x23 (.sel(shift_to_right), .src0(A[23]), .src1(A[8]), .z(x23));
55        mux mux_x22 (.sel(shift_to_right), .src0(A[22]), .src1(A[9]), .z(x22));
56        mux mux_x21 (.sel(shift_to_right), .src0(A[21]), .src1(A[10]), .z(x21));
57        mux mux_x20 (.sel(shift_to_right), .src0(A[20]), .src1(A[11]), .z(x20));
58        mux mux_x19 (.sel(shift_to_right), .src0(A[19]), .src1(A[12]), .z(x19));
59        mux mux_x18 (.sel(shift_to_right), .src0(A[18]), .src1(A[13]), .z(x18));
60        mux mux_x17 (.sel(shift_to_right), .src0(A[17]), .src1(A[14]), .z(x17));
61        mux mux_x16 (.sel(shift_to_right), .src0(A[16]), .src1(A[15]), .z(x16));
62        mux mux_x15 (.sel(shift_to_right), .src0(A[15]), .src1(A[16]), .z(x15));
63        mux mux_x14 (.sel(shift_to_right), .src0(A[14]), .src1(A[17]), .z(x14));
64        mux mux_x13 (.sel(shift_to_right), .src0(A[13]), .src1(A[18]), .z(x13));
65        mux mux_x12 (.sel(shift_to_right), .src0(A[12]), .src1(A[19]), .z(x12));
66        mux mux_x11 (.sel(shift_to_right), .src0(A[11]), .src1(A[20]), .z(x11));
67        mux mux_x10 (.sel(shift_to_right), .src0(A[10]), .src1(A[21]), .z(x10));
68        mux mux_x9 (.sel(shift_to_right), .src0(A[9]), .src1(A[22]), .z(x9));
69        mux mux_x8 (.sel(shift_to_right), .src0(A[8]), .src1(A[23]), .z(x8));
70        mux mux_x7 (.sel(shift_to_right), .src0(A[7]), .src1(A[24]), .z(x7));
```

```
71        mux mux_x6 (.sel(shift_to_right), .src0(A[6]), .src1(A[25]), .z(x6));
72        mux mux_x5 (.sel(shift_to_right), .src0(A[5]), .src1(A[26]), .z(x5));
73        mux mux_x4 (.sel(shift_to_right), .src0(A[4]), .src1(A[27]), .z(x4));
74        mux mux_x3 (.sel(shift_to_right), .src0(A[3]), .src1(A[28]), .z(x3));
75        mux mux_x2 (.sel(shift_to_right), .src0(A[2]), .src1(A[29]), .z(x2));
76        mux mux_x1 (.sel(shift_to_right), .src0(A[1]), .src1(A[30]), .z(x1));
77        mux mux_x0 (.sel(shift_to_right), .src0(A[0]), .src1(A[31]), .z(x0));
78
79        // Build the first row of muxes (shamt 0 or 1)
80        mux mux_s0_b31 (.sel(shamt[0]), .src0(x31), .src1(x30), .z(r31));
81        mux mux_s0_b30 (.sel(shamt[0]), .src0(x30), .src1(x29), .z(r30));
82        mux mux_s0_b29 (.sel(shamt[0]), .src0(x29), .src1(x28), .z(r29));
83        mux mux_s0_b28 (.sel(shamt[0]), .src0(x28), .src1(x27), .z(r28));
84        mux mux_s0_b27 (.sel(shamt[0]), .src0(x27), .src1(x26), .z(r27));
85        mux mux_s0_b26 (.sel(shamt[0]), .src0(x26), .src1(x25), .z(r26));
86        mux mux_s0_b25 (.sel(shamt[0]), .src0(x25), .src1(x24), .z(r25));
87        mux mux_s0_b24 (.sel(shamt[0]), .src0(x24), .src1(x23), .z(r24));
88        mux mux_s0_b23 (.sel(shamt[0]), .src0(x23), .src1(x22), .z(r23));
89        mux mux_s0_b22 (.sel(shamt[0]), .src0(x22), .src1(x21), .z(r22));
90        mux mux_s0_b21 (.sel(shamt[0]), .src0(x21), .src1(x20), .z(r21));
91        mux mux_s0_b20 (.sel(shamt[0]), .src0(x20), .src1(x19), .z(r20));
92        mux mux_s0_b19 (.sel(shamt[0]), .src0(x19), .src1(x18), .z(r19));
93        mux mux_s0_b18 (.sel(shamt[0]), .src0(x18), .src1(x17), .z(r18));
94        mux mux_s0_b17 (.sel(shamt[0]), .src0(x17), .src1(x16), .z(r17));
95        mux mux_s0_b16 (.sel(shamt[0]), .src0(x16), .src1(x15), .z(r16));
96        mux mux_s0_b15 (.sel(shamt[0]), .src0(x15), .src1(x14), .z(r15));
97        mux mux_s0_b14 (.sel(shamt[0]), .src0(x14), .src1(x13), .z(r14));
98        mux mux_s0_b13 (.sel(shamt[0]), .src0(x13), .src1(x12), .z(r13));
99        mux mux_s0_b12 (.sel(shamt[0]), .src0(x12), .src1(x11), .z(r12));
100       mux mux_s0_b11 (.sel(shamt[0]), .src0(x11), .src1(x10), .z(r11));
101       mux mux_s0_b10 (.sel(shamt[0]), .src0(x10), .src1(x9), .z(r10));
102       mux mux_s0_b9 (.sel(shamt[0]), .src0(x9), .src1(x8), .z(r9));
103       mux mux_s0_b8 (.sel(shamt[0]), .src0(x8), .src1(x7), .z(r8));
104       mux mux_s0_b7 (.sel(shamt[0]), .src0(x7), .src1(x6), .z(r7));
105       mux mux_s0_b6 (.sel(shamt[0]), .src0(x6), .src1(x5), .z(r6));
106       mux mux_s0_b5 (.sel(shamt[0]), .src0(x5), .src1(x4), .z(r5));
107       mux mux_s0_b4 (.sel(shamt[0]), .src0(x4), .src1(x3), .z(r4));
108       mux mux_s0_b3 (.sel(shamt[0]), .src0(x3), .src1(x2), .z(r3));
109       mux mux_s0_b2 (.sel(shamt[0]), .src0(x2), .src1(x1), .z(r2));
110       mux mux_s0_b1 (.sel(shamt[0]), .src0(x1), .src1(x0), .z(r1));
111       mux mux_s0_b0 (.sel(shamt[0]), .src0(x0), .src1(LSB), .z(r0));
112
113       // Build the second row of muxes (shamt 0 or 2)
114       mux mux_s1_b31 (.sel(shamt[1]), .src0(r31), .src1(r29), .z(s31));
115       mux mux_s1_b30 (.sel(shamt[1]), .src0(r30), .src1(r28), .z(s30));
116       mux mux_s1_b29 (.sel(shamt[1]), .src0(r29), .src1(r27), .z(s29));
117       mux mux_s1_b28 (.sel(shamt[1]), .src0(r28), .src1(r26), .z(s28));
118       mux mux_s1_b27 (.sel(shamt[1]), .src0(r27), .src1(r25), .z(s27));
119       mux mux_s1_b26 (.sel(shamt[1]), .src0(r26), .src1(r24), .z(s26));
120       mux mux_s1_b25 (.sel(shamt[1]), .src0(r25), .src1(r23), .z(s25));
121       mux mux_s1_b24 (.sel(shamt[1]), .src0(r24), .src1(r22), .z(s24));
122       mux mux_s1_b23 (.sel(shamt[1]), .src0(r23), .src1(r21), .z(s23));
123       mux mux_s1_b22 (.sel(shamt[1]), .src0(r22), .src1(r20), .z(s22));
124       mux mux_s1_b21 (.sel(shamt[1]), .src0(r21), .src1(r19), .z(s21));
125       mux mux_s1_b20 (.sel(shamt[1]), .src0(r20), .src1(r18), .z(s20));
126       mux mux_s1_b19 (.sel(shamt[1]), .src0(r19), .src1(r17), .z(s19));
127       mux mux_s1_b18 (.sel(shamt[1]), .src0(r18), .src1(r16), .z(s18));
128       mux mux_s1_b17 (.sel(shamt[1]), .src0(r17), .src1(r15), .z(s17));
```

```
129         mux mux_s1_b16 (.sel(shamt[1]), .src0(r16), .src1(r14), .z(s16));
130         mux mux_s1_b15 (.sel(shamt[1]), .src0(r15), .src1(r13), .z(s15));
131         mux mux_s1_b14 (.sel(shamt[1]), .src0(r14), .src1(r12), .z(s14));
132         mux mux_s1_b13 (.sel(shamt[1]), .src0(r13), .src1(r11), .z(s13));
133         mux mux_s1_b12 (.sel(shamt[1]), .src0(r12), .src1(r10), .z(s12));
134         mux mux_s1_b11 (.sel(shamt[1]), .src0(r11), .src1(r9), .z(s11));
135         mux mux_s1_b10 (.sel(shamt[1]), .src0(r10), .src1(r8), .z(s10));
136         mux mux_s1_b9 (.sel(shamt[1]), .src0(r9), .src1(r7), .z(s9));
137         mux mux_s1_b8 (.sel(shamt[1]), .src0(r8), .src1(r6), .z(s8));
138         mux mux_s1_b7 (.sel(shamt[1]), .src0(r7), .src1(r5), .z(s7));
139         mux mux_s1_b6 (.sel(shamt[1]), .src0(r6), .src1(r4), .z(s6));
140         mux mux_s1_b5 (.sel(shamt[1]), .src0(r5), .src1(r3), .z(s5));
141         mux mux_s1_b4 (.sel(shamt[1]), .src0(r4), .src1(r2), .z(s4));
142         mux mux_s1_b3 (.sel(shamt[1]), .src0(r3), .src1(r1), .z(s3));
143         mux mux_s1_b2 (.sel(shamt[1]), .src0(r2), .src1(r0), .z(s2));
144         mux mux_s1_b1 (.sel(shamt[1]), .src0(r1), .src1(LSB), .z(s1));
145         mux mux_s1_b0 (.sel(shamt[1]), .src0(r0), .src1(LSB), .z(s0));
146
147         // Build the third row of muxes (shamt 0 or 4)
148         mux mux_s2_b31 (.sel(shamt[2]), .src0(s31), .src1(s27), .z(t31));
149         mux mux_s2_b30 (.sel(shamt[2]), .src0(s30), .src1(s26), .z(t30));
150         mux mux_s2_b29 (.sel(shamt[2]), .src0(s29), .src1(s25), .z(t29));
151         mux mux_s2_b28 (.sel(shamt[2]), .src0(s28), .src1(s24), .z(t28));
152         mux mux_s2_b27 (.sel(shamt[2]), .src0(s27), .src1(s23), .z(t27));
153         mux mux_s2_b26 (.sel(shamt[2]), .src0(s26), .src1(s22), .z(t26));
154         mux mux_s2_b25 (.sel(shamt[2]), .src0(s25), .src1(s21), .z(t25));
155         mux mux_s2_b24 (.sel(shamt[2]), .src0(s24), .src1(s20), .z(t24));
156         mux mux_s2_b23 (.sel(shamt[2]), .src0(s23), .src1(s19), .z(t23));
157         mux mux_s2_b22 (.sel(shamt[2]), .src0(s22), .src1(s18), .z(t22));
158         mux mux_s2_b21 (.sel(shamt[2]), .src0(s21), .src1(s17), .z(t21));
159         mux mux_s2_b20 (.sel(shamt[2]), .src0(s20), .src1(s16), .z(t20));
160         mux mux_s2_b19 (.sel(shamt[2]), .src0(s19), .src1(s15), .z(t19));
161         mux mux_s2_b18 (.sel(shamt[2]), .src0(s18), .src1(s14), .z(t18));
162         mux mux_s2_b17 (.sel(shamt[2]), .src0(s17), .src1(s13), .z(t17));
163         mux mux_s2_b16 (.sel(shamt[2]), .src0(s16), .src1(s12), .z(t16));
164         mux mux_s2_b15 (.sel(shamt[2]), .src0(s15), .src1(s11), .z(t15));
165         mux mux_s2_b14 (.sel(shamt[2]), .src0(s14), .src1(s10), .z(t14));
166         mux mux_s2_b13 (.sel(shamt[2]), .src0(s13), .src1(s9), .z(t13));
167         mux mux_s2_b12 (.sel(shamt[2]), .src0(s12), .src1(s8), .z(t12));
168         mux mux_s2_b11 (.sel(shamt[2]), .src0(s11), .src1(s7), .z(t11));
169         mux mux_s2_b10 (.sel(shamt[2]), .src0(s10), .src1(s6), .z(t10));
170         mux mux_s2_b9 (.sel(shamt[2]), .src0(s9), .src1(s5), .z(t9));
171         mux mux_s2_b8 (.sel(shamt[2]), .src0(s8), .src1(s4), .z(t8));
172         mux mux_s2_b7 (.sel(shamt[2]), .src0(s7), .src1(s3), .z(t7));
173         mux mux_s2_b6 (.sel(shamt[2]), .src0(s6), .src1(s2), .z(t6));
174         mux mux_s2_b5 (.sel(shamt[2]), .src0(s5), .src1(s1), .z(t5));
175         mux mux_s2_b4 (.sel(shamt[2]), .src0(s4), .src1(s0), .z(t4));
176         mux mux_s2_b3 (.sel(shamt[2]), .src0(s3), .src1(LSB), .z(t3));
177         mux mux_s2_b2 (.sel(shamt[2]), .src0(s2), .src1(LSB), .z(t2));
178         mux mux_s2_b1 (.sel(shamt[2]), .src0(s1), .src1(LSB), .z(t1));
179         mux mux_s2_b0 (.sel(shamt[2]), .src0(s0), .src1(LSB), .z(t0));
180
181         // Build the fourth row of muxes (shamt 0 or 8)
182         mux mux_s3_b31 (.sel(shamt[3]), .src0(t31), .src1(t23), .z(u31));
183         mux mux_s3_b30 (.sel(shamt[3]), .src0(t30), .src1(t22), .z(u30));
184         mux mux_s3_b29 (.sel(shamt[3]), .src0(t29), .src1(t21), .z(u29));
185         mux mux_s3_b28 (.sel(shamt[3]), .src0(t28), .src1(t20), .z(u28));
186         mux mux_s3_b27 (.sel(shamt[3]), .src0(t27), .src1(t19), .z(u27));
```

```
187         mux mux_s3_b26 (.sel(shamt[3]), .src0(t26), .src1(t18), .z(u26));
188         mux mux_s3_b25 (.sel(shamt[3]), .src0(t25), .src1(t17), .z(u25));
189         mux mux_s3_b24 (.sel(shamt[3]), .src0(t24), .src1(t16), .z(u24));
190         mux mux_s3_b23 (.sel(shamt[3]), .src0(t23), .src1(t15), .z(u23));
191         mux mux_s3_b22 (.sel(shamt[3]), .src0(t22), .src1(t14), .z(u22));
192         mux mux_s3_b21 (.sel(shamt[3]), .src0(t21), .src1(t13), .z(u21));
193         mux mux_s3_b20 (.sel(shamt[3]), .src0(t20), .src1(t12), .z(u20));
194         mux mux_s3_b19 (.sel(shamt[3]), .src0(t19), .src1(t11), .z(u19));
195         mux mux_s3_b18 (.sel(shamt[3]), .src0(t18), .src1(t10), .z(u18));
196         mux mux_s3_b17 (.sel(shamt[3]), .src0(t17), .src1(t9), .z(u17));
197         mux mux_s3_b16 (.sel(shamt[3]), .src0(t16), .src1(t8), .z(u16));
198         mux mux_s3_b15 (.sel(shamt[3]), .src0(t15), .src1(t7), .z(u15));
199         mux mux_s3_b14 (.sel(shamt[3]), .src0(t14), .src1(t6), .z(u14));
200         mux mux_s3_b13 (.sel(shamt[3]), .src0(t13), .src1(t5), .z(u13));
201         mux mux_s3_b12 (.sel(shamt[3]), .src0(t12), .src1(t4), .z(u12));
202         mux mux_s3_b11 (.sel(shamt[3]), .src0(t11), .src1(t3), .z(u11));
203         mux mux_s3_b10 (.sel(shamt[3]), .src0(t10), .src1(t2), .z(u10));
204         mux mux_s3_b9 (.sel(shamt[3]), .src0(t9), .src1(t1), .z(u9));
205         mux mux_s3_b8 (.sel(shamt[3]), .src0(t8), .src1(t0), .z(u8));
206         mux mux_s3_b7 (.sel(shamt[3]), .src0(t7), .src1(LSB), .z(u7));
207         mux mux_s3_b6 (.sel(shamt[3]), .src0(t6), .src1(LSB), .z(u6));
208         mux mux_s3_b5 (.sel(shamt[3]), .src0(t5), .src1(LSB), .z(u5));
209         mux mux_s3_b4 (.sel(shamt[3]), .src0(t4), .src1(LSB), .z(u4));
210         mux mux_s3_b3 (.sel(shamt[3]), .src0(t3), .src1(LSB), .z(u3));
211         mux mux_s3_b2 (.sel(shamt[3]), .src0(t2), .src1(LSB), .z(u2));
212         mux mux_s3_b1 (.sel(shamt[3]), .src0(t1), .src1(LSB), .z(u1));
213         mux mux_s3_b0 (.sel(shamt[3]), .src0(t0), .src1(LSB), .z(u0));
214
215         // Build the fifth and last row of muxes (shamt 0 or 16)
216         mux mux_s4_b31 (.sel(shamt[4]), .src0(u31), .src1(u15), .z(v31));
217         mux mux_s4_b30 (.sel(shamt[4]), .src0(u30), .src1(u14), .z(v30));
218         mux mux_s4_b29 (.sel(shamt[4]), .src0(u29), .src1(u13), .z(v29));
219         mux mux_s4_b28 (.sel(shamt[4]), .src0(u28), .src1(u12), .z(v28));
220         mux mux_s4_b27 (.sel(shamt[4]), .src0(u27), .src1(u11), .z(v27));
221         mux mux_s4_b26 (.sel(shamt[4]), .src0(u26), .src1(u10), .z(v26));
222         mux mux_s4_b25 (.sel(shamt[4]), .src0(u25), .src1(u9), .z(v25));
223         mux mux_s4_b24 (.sel(shamt[4]), .src0(u24), .src1(u8), .z(v24));
224         mux mux_s4_b23 (.sel(shamt[4]), .src0(u23), .src1(u7), .z(v23));
225         mux mux_s4_b22 (.sel(shamt[4]), .src0(u22), .src1(u6), .z(v22));
226         mux mux_s4_b21 (.sel(shamt[4]), .src0(u21), .src1(u5), .z(v21));
227         mux mux_s4_b20 (.sel(shamt[4]), .src0(u20), .src1(u4), .z(v20));
228         mux mux_s4_b19 (.sel(shamt[4]), .src0(u19), .src1(u3), .z(v19));
229         mux mux_s4_b18 (.sel(shamt[4]), .src0(u18), .src1(u2), .z(v18));
230         mux mux_s4_b17 (.sel(shamt[4]), .src0(u17), .src1(u1), .z(v17));
231         mux mux_s4_b16 (.sel(shamt[4]), .src0(u16), .src1(u0), .z(v16));
232         mux mux_s4_b15 (.sel(shamt[4]), .src0(u15), .src1(LSB), .z(v15));
233         mux mux_s4_b14 (.sel(shamt[4]), .src0(u14), .src1(LSB), .z(v14));
234         mux mux_s4_b13 (.sel(shamt[4]), .src0(u13), .src1(LSB), .z(v13));
235         mux mux_s4_b12 (.sel(shamt[4]), .src0(u12), .src1(LSB), .z(v12));
236         mux mux_s4_b11 (.sel(shamt[4]), .src0(u11), .src1(LSB), .z(v11));
237         mux mux_s4_b10 (.sel(shamt[4]), .src0(u10), .src1(LSB), .z(v10));
238         mux mux_s4_b9 (.sel(shamt[4]), .src0(u9), .src1(LSB), .z(v9));
239         mux mux_s4_b8 (.sel(shamt[4]), .src0(u8), .src1(LSB), .z(v8));
240         mux mux_s4_b7 (.sel(shamt[4]), .src0(u7), .src1(LSB), .z(v7));
241         mux mux_s4_b6 (.sel(shamt[4]), .src0(u6), .src1(LSB), .z(v6));
242         mux mux_s4_b5 (.sel(shamt[4]), .src0(u5), .src1(LSB), .z(v5));
243         mux mux_s4_b4 (.sel(shamt[4]), .src0(u4), .src1(LSB), .z(v4));
244         mux mux_s4_b3 (.sel(shamt[4]), .src0(u3), .src1(LSB), .z(v3));
```

```
245         mux mux_s4_b2 (.sel(shamt[4]), .src0(u2), .src1(LSB), .z(v2));
246         mux mux_s4_b1 (.sel(shamt[4]), .src0(u1), .src1(LSB), .z(v1));
247         mux mux_s4_b0 (.sel(shamt[4]), .src0(u0), .src1(LSB), .z(v0));
248
249         // In case shift_to_right is high, flip values back to normal.
250         // In any case, assign values to output.
251         mux mux_y31 (.sel(shift_to_right), .src0(v31), .src1(v0), .z(z[31]));
252         mux mux_y30 (.sel(shift_to_right), .src0(v30), .src1(v1), .z(z[30]));
253         mux mux_y29 (.sel(shift_to_right), .src0(v29), .src1(v2), .z(z[29]));
254         mux mux_y28 (.sel(shift_to_right), .src0(v28), .src1(v3), .z(z[28]));
255         mux mux_y27 (.sel(shift_to_right), .src0(v27), .src1(v4), .z(z[27]));
256         mux mux_y26 (.sel(shift_to_right), .src0(v26), .src1(v5), .z(z[26]));
257         mux mux_y25 (.sel(shift_to_right), .src0(v25), .src1(v6), .z(z[25]));
258         mux mux_y24 (.sel(shift_to_right), .src0(v24), .src1(v7), .z(z[24]));
259         mux mux_y23 (.sel(shift_to_right), .src0(v23), .src1(v8), .z(z[23]));
260         mux mux_y22 (.sel(shift_to_right), .src0(v22), .src1(v9), .z(z[22]));
261         mux mux_y21 (.sel(shift_to_right), .src0(v21), .src1(v10), .z(z[21]));
262         mux mux_y20 (.sel(shift_to_right), .src0(v20), .src1(v11), .z(z[20]));
263         mux mux_y19 (.sel(shift_to_right), .src0(v19), .src1(v12), .z(z[19]));
264         mux mux_y18 (.sel(shift_to_right), .src0(v18), .src1(v13), .z(z[18]));
265         mux mux_y17 (.sel(shift_to_right), .src0(v17), .src1(v14), .z(z[17]));
266         mux mux_y16 (.sel(shift_to_right), .src0(v16), .src1(v15), .z(z[16]));
267         mux mux_y15 (.sel(shift_to_right), .src0(v15), .src1(v16), .z(z[15]));
268         mux mux_y14 (.sel(shift_to_right), .src0(v14), .src1(v17), .z(z[14]));
269         mux mux_y13 (.sel(shift_to_right), .src0(v13), .src1(v18), .z(z[13]));
270         mux mux_y12 (.sel(shift_to_right), .src0(v12), .src1(v19), .z(z[12]));
271         mux mux_y11 (.sel(shift_to_right), .src0(v11), .src1(v20), .z(z[11]));
272         mux mux_y10 (.sel(shift_to_right), .src0(v10), .src1(v21), .z(z[10]));
273         mux mux_y9 (.sel(shift_to_right), .src0(v9), .src1(v22), .z(z[9]));
274         mux mux_y8 (.sel(shift_to_right), .src0(v8), .src1(v23), .z(z[8]));
275         mux mux_y7 (.sel(shift_to_right), .src0(v7), .src1(v24), .z(z[7]));
276         mux mux_y6 (.sel(shift_to_right), .src0(v6), .src1(v25), .z(z[6]));
277         mux mux_y5 (.sel(shift_to_right), .src0(v5), .src1(v26), .z(z[5]));
278         mux mux_y4 (.sel(shift_to_right), .src0(v4), .src1(v27), .z(z[4]));
279         mux mux_y3 (.sel(shift_to_right), .src0(v3), .src1(v28), .z(z[3]));
280         mux mux_y2 (.sel(shift_to_right), .src0(v2), .src1(v29), .z(z[2]));
281         mux mux_y1 (.sel(shift_to_right), .src0(v1), .src1(v30), .z(z[1]));
282         mux mux_y0 (.sel(shift_to_right), .src0(v0), .src1(v31), .z(z[0]));
283
284     endmodule
```

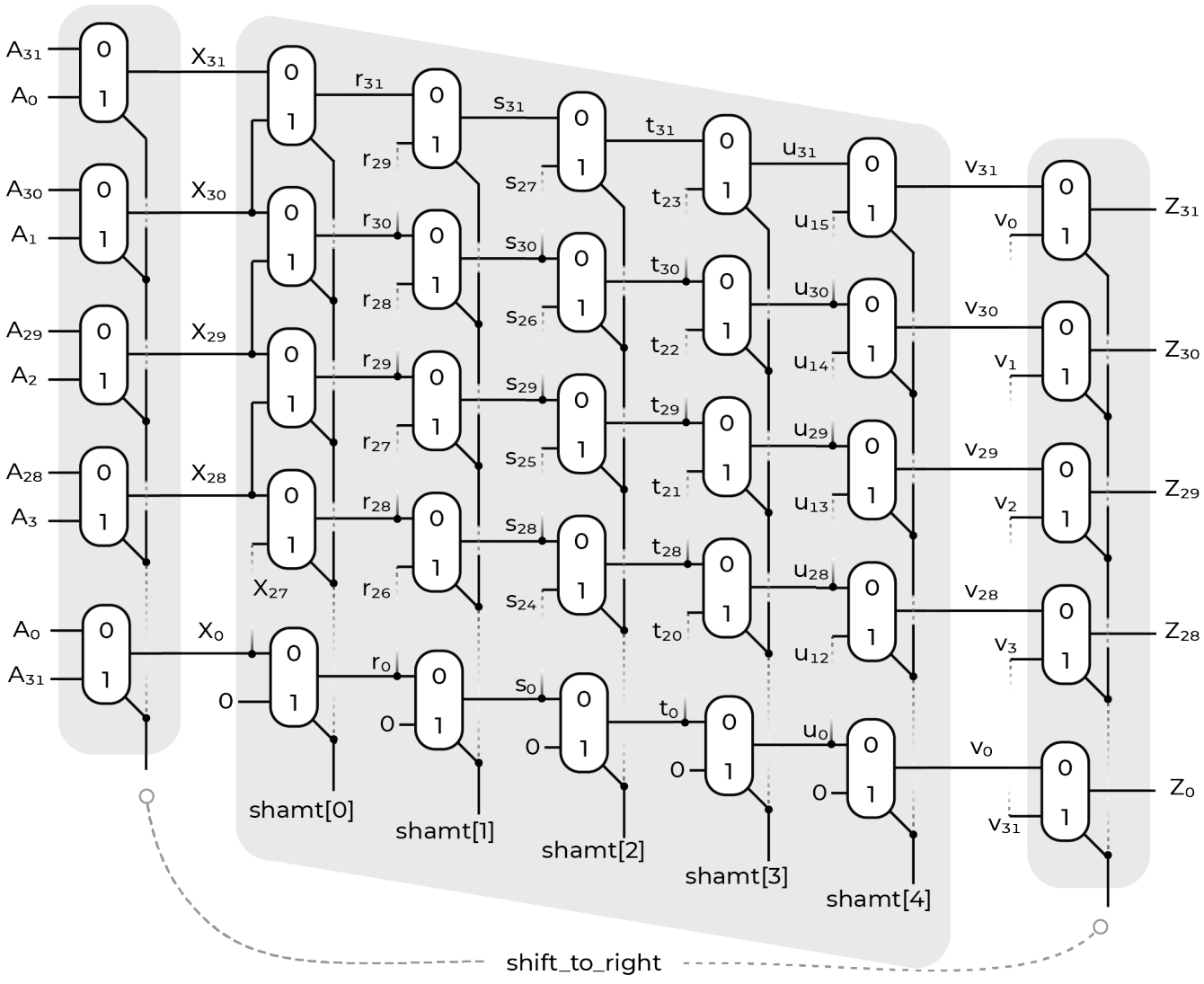

Figure 8. Diagram of the circuit used for the 32-bit shift unit implemented for the 32-bit ALU designed in this work.

Section 7. *Putting the 32-bit ALU together*

7.1 Connection of the blocks

Once all the internal blocks of the ALU have been implemented, as shown in the previous sections, all that is left to do is to connect them and put them together so the final unit is fully functional. As all these blocks are purely combinatorial, the results of all the blocks will be available simultaneously for each input combination of operators and op_code. This means that, as we had to when implementing the inner blocks, we will have to use MUXES to select from the results of each of the different units.

There is one last functionality which has not been implemented yet, which is the zero detection. Detecting whether the final result of the operation is null (all bits are zero) can help speed up further processes to be applied to the result in other parts of the processor. This detection can be, nonetheless, very easily implemented by using a single 32 input NOR gate. The verilog code for this component can be seen in Code 8.

Code 8: Verilog code for the designed 32-bit NOR gate (*<nor_gate_32.v>*)

```
1   module nor_gate_32to1(x, z);
2
3       // Define inputs
4       input [31:0] x;
5
6       // Define outputs
7       output wire z;
8
9       // Define output
10      assign z = ~(|x);
11
12  endmodule
```

7.2 Implementation of the 32-bit ALU

Figure 9 shows all the blocks showed in the previous sections are connected inside the ALU. The ALU itself has three inputs, as expected: two 32-bit operators A and B and one 11-bit operation control bus $op_control$. From the $op_control$ bus all the flags and signals required to control the operation of the ALU are extracted and introduced into the different singular blocks. Notice how the result of the arithmetic unit AU is used also for the comparison unit CU (see Section 5.1).

Some bits of the $op_control$ act as MUX selectors so that the final 32-bit result of the ALU Z is connected to the output of the right block: AU, CU, LU or SU. The *carryout* and $overflow$ signals calculated by the arithmetic unit AU are also sent outside of the ALU. Finally, the *Zero* signal is also computed to check whether the result of the ALU is null. This 1-bit *Zero* signal is also passed to the output of the ALU, for later use by the rest of the processor.

The code that implements the whole ALU shown in Figure 9 can be found in Code 9.

Code 9: Verilog code for the designed 32-bit ALU (*<ALU_32.v>*)

```
1   module ALU_32(A, B, Z, op_ctl, overflow, zero, carryout);
2
3       // Define our inputs
4       input [31:0] A;   // 32b Operator A
5       input [31:0] B;   // 32b Operator B
6       input [10:0] op_ctl;  // 11b control bus
7
8       // Define our outputs
9       output wire [31:0] Z;  // 32b Result (A op B)
10      output overflow; // 1b overflow flag, THIS IS ACTIVE WHEN
11      // 1b zero flag, THIS IS ACTIVE WHEN the result of the operation applied to
12      // A/B is zero.
13      output zero;
14      // 1b carryout flag, THIS IS ACTIVE WHEN the result of the
```

```
15      // operation applied to A/B requires a carry out (for further operations)
16      output carryout;
17
18      // Split op_control into ctl and shamt
19      // 6b Control line, used to select the operation to be applied to A and B
20      wire [5:0] ctl;
21      // 5b Control line indicating the amount of shift to be applied (if op is shift)
22      wire [4:0] shamt;
23      assign ctl = op_ctl[5:0];
24      assign shamt = op_ctl[10:6];
25
26      // Define internal wires for control units
27
28      // Selector of the operator of the logic unit (00: AND, 01: OR, 10: XOR, 11: NOR)
29      wire [1:0] lu_ctl;
30      // Flag to specify whether operator B should be negated or not.
31      // When this is 0, AU performs addition. When this is 1, AU performs subtraction.
32      wire b_inv;
33      // zero if we have to use the logic unit, one if we have to use the
34      // logic/comparison unit.
35      wire comb_adder_sel;
36      // zero if we have to select the logic_unit result, one if we have to select
37      / the comparison unit result.
38      wire comparison_sel;
39      // zero indicates shift to left, one indicates shift to right.
40      wire shift_to_right;
41      // zero indicates use shift_unit, one indicates use adder/comparison/logical units
42      wire arithm_sel;
43      // zero indicates perform comparison using signed integers, one indicates
44      // unsigned comparison
45      wire unsigned_ctl;
46
47      // Define internal outputs from blocks
48      wire [31:0] LU_z;
49      wire [31:0] AU_z;
50      wire [31:0] CU_z;
51      wire [31:0] SU_z;
52      // Holder for the output of the mux between the output of the arithmetic
53      // unit AU_z and the output of the comparison block CU_z
54      wire [31:0] AC_z;
55      // Holder for the output of the mux that chooses between
56      // arithmetic/comparison and logic units.
57      wire [31:0] AC_L_z;
58
59      // Assign controls for units
60      assign lu_ctl = ctl[1:0];
61      assign b_inv = ctl[1];
62      assign comb_adder_sel = ctl[2];
63      assign comparison_sel = ctl[3];
64      assign shift_to_right = ctl[1];
65      assign arithm_sel = ctl[5];
66      assign unsigned_ctl = ctl[0];
67
68      // The first and easiest thing to do is to set the Zero output. The zero output
69      // is high when the result is zero.
70      //assign zero = (Z == 32'b0);
71      nor_gate_32to1 ZNOR_1 (.x(Z), .z(zero));
72
```

```
73        // Wire-up the LOGIC UNIT
74        logic_unit_32 LU32_1 (.A(A), .B(B), .lu_ctl(lu_ctl), .z(LU_z));
75
76        // Wire-up the ARITHMETIC UNIT
77        arithmetic_unit_32 AU32_1 (.A(A), .B(B), .b_inv(b_inv), .z(AU_z),
78           .cout(carryout), .overflow(overflow));
79
80        // Wire-up the comparison block
81        comparison_unit_32 CU32_1 (.A(A), .B(B), .sub_z(AU_z),
82           .carryout(carryout), .unsigned_ctl(unsigned_ctl), .z(CU_z));
83
84        // Wire-up the shifter block
85        shift_unit_32 SU32_1 (.A(A), .shamt(shamt), .shift_to_right(shift_to_right),
86           .z(SU_z));
87
88        // Connect the MUX to select between AU_z or CU_z depending on
89        mux2to1_32 mux_0 (.sel(comparison_sel), .src0(AU_z), .src1(CU_z), .z(AC_z));
90
91        // Connect the MUX to select between LU and AU results using ctl[2]
92        mux2to1_32 mux_1 (.sel(comb_adder_sel), .src0(AC_z), .src1(LU_z), .z(AC_L_z));
93
94        // Connect the MUX to select between ((Arithmetic/Comparison)/Logic)/Shift units
95        // and assign to output of ALU
96        mux2to1_32 mux_2 (.sel(arithm_sel), .src0(SU_z), .src1(AC_L_z), .z(Z));
97
98    endmodule
```

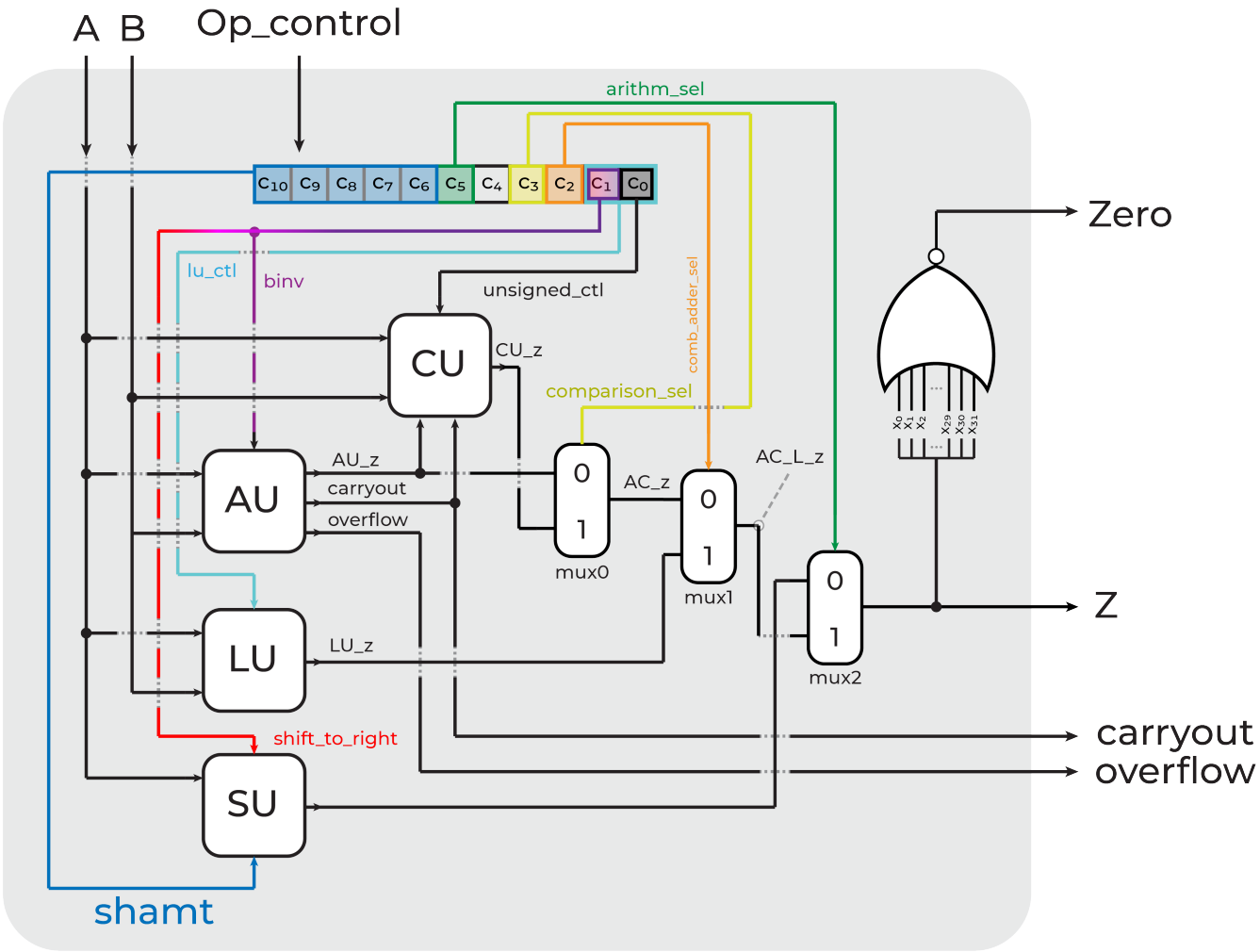

Figure 9. Diagram of the internal circuit that composes the 32-bit ALU designed in this work. The ALU is composed by four main blocks which implement the different groups of operations performed: the arithmetic unit *AU*, which performs the arithmetic operations *add* and *sub*; the logic unit *LU*, which performs the logic operations *and*, *or*, *xor* and *nor*; the comparison unit *CU*, which performs the comparison operations *slt* and *sltu*; and the shift unit *SU*, which performs the logical shifting operations *sll* and *srl*.

Section 8. *Synthesis of the 32-bit ALU*

8.1 Synthesis environment

It is worth commenting that the tool used for synthesis was **Cadence GENUS**, which was running in an environment with the following characteristics:

Host: coolr1.ece.northwestern.edu
$(\text{x86}_64w/Linux2.6.32 - 431.el6.x86_64)$ $(4cores * 4cpus * 1physicalcpu * Intel(R)Xeon(R)CPUE5 - 2609v2@2.50GHz10240KB)$ $(32979720KB)PID : 7701$
$OS : CentOSrelease6.10(Final)$

8.2 Automation of the synthesis script

In order to automate both the synthesis and simulation processes, a Makefile script was created. This makefile allow us to perform synthesis by running a single line of code in the command-line. The makefile created for this task can be seen in Code 10.

Code 10: Makefile created for synthesis and simulation automation (<*Makefile*>)

```
1     TOP_MODULE = ALU_tb
2
3     WORK_LIB = work
4
5     XRUN = xrun
6     XGEN = genus
7     COMPILER = iverilog #xmverilog
8
9     PROJ_DIR = /home/mvalentin/ECE361_ALU_32
10    BASE_DIR = $(PROJ_DIR)/src
11    LIB_DIR = $(BASE_DIR)/eecs361lib_alu_Verilog/lib
12    GENUS_TCL = $(PROJ_DIR)/genus_synthesis.tcl
13
14    VPATH :=
15    VPATH += $(BASE_DIR)
16    VPATH += $(LIB_DIR)
17
18    V_SOURCES :=
19    V_SOURCES += $(BASE_DIR)/ALU_tb.v
20    V_SOURCES += $(BASE_DIR)/logic_unit_32.v
21    V_SOURCES += $(BASE_DIR)/full_fast_adder.v
22    V_SOURCES += $(BASE_DIR)/comparison_unit_32.v
23    V_SOURCES += $(BASE_DIR)/arithmetic_unit_32.v
24    V_SOURCES += $(BASE_DIR)/shift_unit_32.v
25    V_SOURCES += $(BASE_DIR)/mux4to1_32.v
26    V_SOURCES += $(BASE_DIR)/mux2to1_32.v
27    V_SOURCES += $(BASE_DIR)/nor_gate_32to1.v
28
29    V_SOURCES += $(LIB_DIR)/and_gate_32.v
30    V_SOURCES += $(LIB_DIR)/or_gate_32.v
31    V_SOURCES += $(LIB_DIR)/nor_gate_32.v
32    V_SOURCES += $(LIB_DIR)/not_gate_32.v
33    V_SOURCES += $(LIB_DIR)/xor_gate_32.v
34    V_SOURCES += $(LIB_DIR)/xor_gate.v
35    V_SOURCES += $(LIB_DIR)/and_gate.v
36    V_SOURCES += $(LIB_DIR)/nor_gate.v
37    V_SOURCES += $(LIB_DIR)/or_gate.v
```

```
38      V_SOURCES += $(LIB_DIR)/not_gate.v
39      V_SOURCES += $(LIB_DIR)/mux_32.v
40      V_SOURCES += $(LIB_DIR)/mux.v
41
42      V_HLS := $(V_SOURCES)
43      V_HLS += $(BASE_DIR)/ALU_32.v
44
45      V_SYN := $(V_SOURCES)
46      V_SYN += $(PROJ_DIR)/Synthesis/ALU_32_syn.v
47      V_SYN += $(BASE_DIR)/cad/NangateOpenCellLibrary.v
48
49
50      CAD_LIB = /cad/library/TSMC/TSMC65_LP/Base_PDK/digital/Front_End/verilog/tcbn65lp_200a/tcbn65lp
51
52
53      .SUFFIXES: .v .sv
54
55      setup: $(WORK_LIB)
56
57
58      run-synthesis:
59         $(XGEN) -files $(GENUS_TCL) $^
60      .PHONY: run-synthesis
61
62
63      run-post-hls: $(V_HLS)
64         $(XRUN) -access rwc -gui $^
65      .PHONY: run-post-hls
66
67      run-post-syn: $(V_SYN)
68         $(XRUN) -access rwc -gui $^
69      .PHONY: run-post-syn
70
71      clean:
72         rm -rf xcelium.d xrun* *.vcd waves.shm *.diag *.err *.log* *.cmd* fv*
73      .PHONY: clean
```

Apart from the Makefile, a *.tcl* file has been created to automate the synthesis procedure. This file basically loads the verilog modules implemented in this project (as well as those given in the project libraries) and elaborates the design. The code used in this *.tcl* file can be seen in Code 11.

Code 11: Tcl file created for synthesis automation with genus (<*genus_synthesis.tcl*>)

```
1       read_hdl ../src/ALU_32.v
2       read_hdl ../src/arithmetic_unit_32.v
3       read_hdl ../src/comparison_unit_32.v
4       read_hdl ../src/full_adder.v
5       read_hdl ../src/full_fast_adder.v
6       read_hdl ../src/logic_unit_32.v
7       read_hdl ../src/mux2to1_32.v
8       read_hdl ../src/mux4to1_32.v
9       read_hdl ../src/nor_gate_32to1.v
10      read_hdl ../src/shift_unit_32.v
11      read_hdl ../src/eecs361lib_alu_Verilog/lib/and_gate_32.v
12      read_hdl ../src/eecs361lib_alu_Verilog/lib/and_gate.v
13      read_hdl ../src/eecs361lib_alu_Verilog/lib/mux_32.v
14      read_hdl ../src/eecs361lib_alu_Verilog/lib/mux.v
```

```
15    read_hdl ../src/eecs361lib_alu_Verilog/lib/nand_gate_32.v
16    read_hdl ../src/eecs361lib_alu_Verilog/lib/nand_gate.v
17    read_hdl ../src/eecs361lib_alu_Verilog/lib/nor_gate_32.v
18    read_hdl ../src/eecs361lib_alu_Verilog/lib/nor_gate.v
19    read_hdl ../src/eecs361lib_alu_Verilog/lib/not_gate_32.v
20    read_hdl ../src/eecs361lib_alu_Verilog/lib/not_gate.v
21    read_hdl ../src/eecs361lib_alu_Verilog/lib/or_gate_32.v
22    read_hdl ../src/eecs361lib_alu_Verilog/lib/or_gate.v
23    read_hdl ../src/eecs361lib_alu_Verilog/lib/xnor_gate_32.v
24    read_hdl ../src/eecs361lib_alu_Verilog/lib/xnor_gate.v
25    read_hdl ../src/eecs361lib_alu_Verilog/lib/xor_gate_32.v
26    read_hdl ../src/eecs361lib_alu_Verilog/lib/xor_gate.v
27    set_db library ../src/cad/NangateOpenCellLibrary_typical.lib
28    set_db lef_library ../src/cad/NangateOpenCellLibrary.lef
29    elaborate
30    current_design ALU_32
31    read_sdc ../src/ALU_32.sdc
32    syn_generic
33    syn_map
34    syn_opt
35    report_timing > ALU_32_timing.rpt
36    report_area > ALU_32_area.rpt
37    write_hdl > ALU_32_syn.v
38    quit
```

Notice that Code 11 uses the time constraints defined in file <*ALU_32.sdc*>, whose content can be seen in Code 12.

Code 12: Time constraints used for synthesis specified in file (<*alu_32.sdc*>)

```
1   #create_clock -name clk -period 1.0 -waveform { 0 0.5 } [get_ports clk]
2
3    # ------------------------- Input constraints ---------------------------------
4
5    #set_input_delay -clock clk -max 0.2 [get_ports {din start rstb wr_ctrl_test_crtl}]
6    #set_input_delay -clock clk -min -0.2 [get_ports {din start rstb wr_ctrl_test_crtl}]
7
8    # ------------------------- Output constraints --------------------------------
9
10   #set_output_delay -clock clk -max 0.2 [get_ports {addr_out*}]
11   #set_output_delay -clock clk -min -0.2 [get_ports {addr_out*}]
12
13
14   set_max_delay 1.0 -from [all_inputs] -to [all_outputs]
15
16   # Assume 50fF load capacitances everywhere:
17   set_load 0.050 [all_outputs]
18   # Set 10fF maximum capacitance on all inputs
19   set_max_capacitance 0.010 [all_inputs]
20
21   # set clock uncertainty of the system clock (skew and jitter)
22   #set_clock_uncertainty -setup 0.03 [get_clocks clk*]
23   #set_clock_uncertainty -hold 0.06 [get_clocks clk*]
24
25
26   # set maximum transition at output ports
27   set_max_transition 0.07 [current_design]
28
```

```
29      # set_attr use_scan_seqs_for_non_dft false
```

8.3 Directory organization

The organization of the directory used for this project (implementation, synthesis & validation) is showed in Figure 10.

8.4 Synthesis results

Having created the automation scripts, we can synthesize our project by executing the following code in the command line from inside the synthesis folder:

```
$ make -f ../Makefile clean && make -f ../Makefile run-synthesis
```

The previous command generates several synthesis files. Three of them are specially important for us:

1. *<alu_32_area.rpt>*: Report of the total area of the synthesized design.
2. *<alu_32_timing.rpt>*: Report of the timing analysis performed on the synthesized design.
3. *<alu_32_syn.v>*: Netlist of the synthesized design containing its optimal digital implementation.

Due to its length, the final netlist, which is contained in the file *<ALU_32_syn.v>*, can be seen in Code 16, at the end of this document.

The content of file *<alu_32_area.rpt>* can be seen in Code 13. By looking at this file we can notice that the total area is 2449.816. If we consider that this area is given in μm^2, we can find that the value in mm^2 is $0.002449816 mm^2$.

Code 13: Area report of the synthesized design generated by genus (*<alu_32_area.rpt>*)

```
1   ====================================================================
2     Generated by: Genus(TM) Synthesis Solution 19.10-p002_1
3     Generated on: Oct 21 2020 05:23:00 am
4     Module: ALU_32
5     Operating conditions: typical
6     Interconnect mode: global
7     Area mode: physical library
8   ====================================================================
9
10  Instance Module Cell Count Cell Area Net Area Total Area
11  ---------------------------------------------------------------------
12  ALU_32 887 1099.112 1348.310 2447.422
```

The content of file *<alu_32_timing.rpt>* can be seen in Code 14. According to this report the longest data path is $999 ps$, while the slack is $1 ps$.

Code 14: Timing report of the synthesized design generated by genus (*<alu_32_timing.rpt>*)

```
1   ====================================================================
2     Generated by: Genus(TM) Synthesis Solution 19.10-p002_1
3     Generated on: Oct 21 2020 05:23:00 am
4     Module: ALU_32
5     Operating conditions: typical
6     Interconnect mode: global
7     Area mode: physical library
8   ====================================================================
9
10
```

batgirl@eecs.northwestern.edu:/home/mbv6231/ECE361_ALU_32

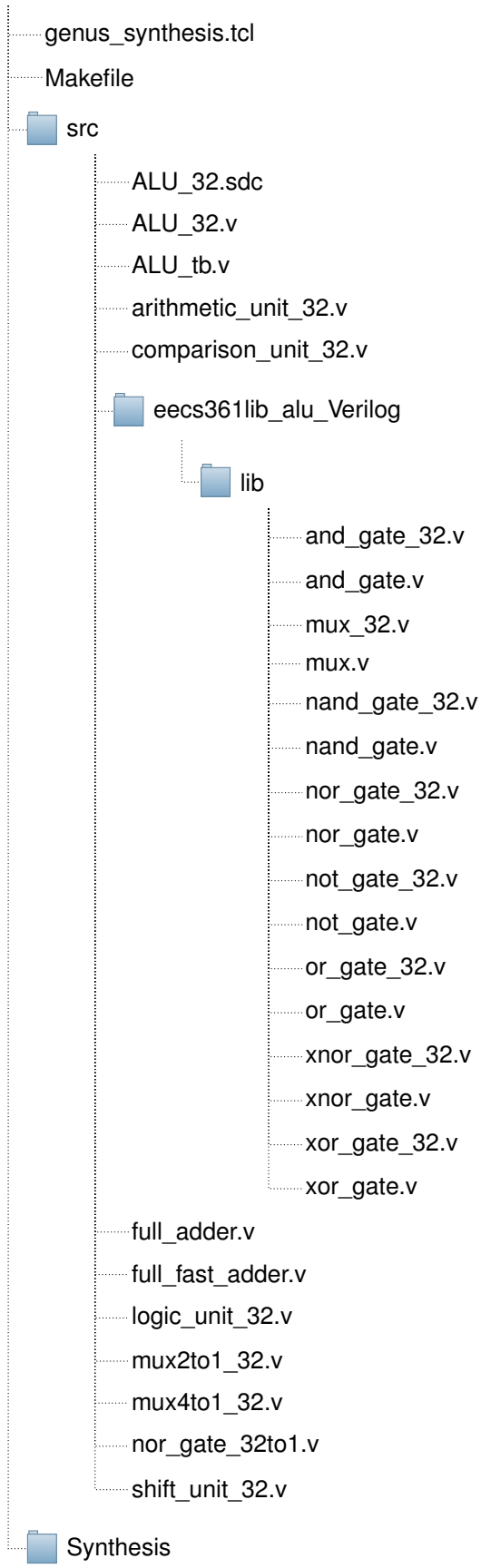

Figure 10. Organization of the directory and files used in this project.

```
Path 1: MET (1 ps) Path Delay Check
    Startpoint: (F) op_ctl[6]
     Endpoint: (R) zero

               Capture Launch
     Path Delay:+ 1000 -
     Drv Adjust:+ 0 0
       Arrival:= 1000

  Required Time:= 1000
     Data Path:- 999
        Slack:= 1

Exceptions/Constraints:
 max_delay 1000 ALU_32.sdc_line_16

#-------------------------------------------------------------------------------------------------
# Timing Point Flags Arc Edge Cell Fanout Load Trans Delay Arrival Instance
# (fF) (ps) (ps) (ps) Location
#-------------------------------------------------------------------------------------------------
 op_ctl[6] - - F (arrival) 2 3.6 0 0 0 (-,-)
 drc_bufs23344/Z - A->Z F BUF_X2 3 6.6 7 24 24 (-,-)
 g22889/ZN - A->ZN R INV_X2 4 12.6 17 23 48 (-,-)
 g22885__4547/ZN - A2->ZN R AND2_X2 29 54.0 65 95 142 (-,-)
 g22860/ZN - A->ZN F INV_X1 2 3.4 16 15 158 (-,-)
 g22777__9906/ZN - A1->ZN R NAND2_X1 1 2.2 11 19 177 (-,-)
 g22618__1309/Z - S->Z F MUX2_X1 4 6.6 15 69 247 (-,-)
 g22511__2391/ZN - B1->ZN R AOI22_X1 1 2.0 26 45 292 (-,-)
 g22411__2250/ZN - A2->ZN F NAND2_X1 2 3.6 13 22 314 (-,-)
 g22283__2250/ZN - B1->ZN R AOI221_X1 2 3.9 57 89 402 (-,-)
 g22263__2900/ZN - B1->ZN F OAI21_X1 2 1.5 17 20 422 (-,-)
 g22219__5703/ZN - A3->ZN F OR4_X1 1 1.2 17 112 534 (-,-)
 g22205__2683/ZN - A3->ZN F OR4_X1 1 1.9 18 115 649 (-,-)
 g22192__2703/ZN - A4->ZN R NOR4_X1 1 2.0 45 85 734 (-,-)
 g22171__4547/ZN - A4->ZN F NAND4_X1 1 1.7 23 34 768 (-,-)
 g22155__2703/ZN - A2->ZN R NOR4_X1 1 1.9 44 71 839 (-,-)
 g22139__1474/ZN - A2->ZN F NAND4_X1 1 1.6 23 36 876 (-,-)
 g22122__5266/ZN - A1->ZN R NOR4_X1 1 1.8 43 53 929 (-,-)
 g22098__2900/ZN - A1->ZN F NAND4_X1 1 1.6 21 34 962 (-,-)
 g22084__8780/ZN - A1->ZN R NOR4_X1 1 0.4 30 37 999 (-,-)
 zero <<< - R (port) - - - 0 999 (-,-)
#-------------------------------------------------------------------------------------------------
```

Section 9. *Testing the design*

Testing the design requires the creation of a testbench for both the implemented HLS and the netlist synthesized by genus. The first one will be used as a baseline for the result we expect to obtain without considering any time constraints, while the second one will serve as proof that the actual synthesized RTL functions as expected.

Both these validation steps will use verification tool **Cadence XCELIUM** and will be called using the Makefile introduced before in Code 10. Also, both of them use the exact same testbench, which is shown in Code 15.

Before moving to the actual verification analysis, let us perform a brief description of how the testbench code is organized and implemented, so we will understand better the results we will obtain.

The first lines of the testbench simply define the timescale and timeprecision that will be used in the simulation. This is required by xcelium. After this, the testbench module (henceforth ALU_32_tb) is instantiated so that all the components that form the ALU, as well as the stimuli can be implemented inside it.

The first signal to be created inside ALU_32_tb is the clock. Notice that despite our implementation of the ALU is purely combinatorial, we are forced to create a clock signal to be able to emulate the ALU behavior while feeding values and testing all operations and functionalities.

A total of ten different operations have to be tested, in the end: *add*, *sub*, *and*, *xor*, *or*, *nor*, *sll*, *srl*, *slt* and *sltu*. Each one of them has a specific code to be fed to the ALU, so we might as well define these codes as static parameters which will be used later. For each one of these operations, we will feed a total number of MAX_COUNT random operators (A and B) so that the results have some level of reliability. Two counters will help us iterate over all 10 functions and over all MAX_COUNT number of samples per operation.

Each one of the operations that we will be testing will obviously generate a 32-bit result word. Instead of manually checking whether the value of this result is right or not, we can simply pre-compute the expected result prior to every operation, and then compare whether the expected and actual results match or not. A register will also help us keep track of how many times these two results do not match, so that at the end we find out which functions (if any) did not function properly, and how many mismatching errors occurred.

Some other registers are required to store the values of the operators A, B, the result of the ALU Z, the $op_control$ bus, as well as the overflow, carry and zero flags that come out of the ALU.

At this point we can instantiate the actual ALU device (take a look at Section 7.2) and connect all the signals to the module I/O.

We are now able to describe the actual behavior of the test we want to implement. First, the *initial* block will help us create the dump file where the waves will be stored, as well as to initialize some of the registers we created before. Secondly, we can implement the clock, which will be driven accordingly to the cycle specified by the variable $HALF_CLK_PERIOD$.

A block activated at rising edge of the clock allow us to perform the actual test of the ALU. First of all, the control bus is initialized to zeros. Then, the operators A and B are set to completely random 32-bit values. Then depending on the value of the $op_counter$ (which iterates over all different 10 operations to be tested), we set the 6 LSB of the control bus to one of the control codes for each one of the operations that the ALU can perform. Notice that at this step we also compute the expected value for each operation, and that this expected value is computed using verilog built-in functions that have absolutely nothing to do with our implementation of the ALU presented here (which means that the computation of both values are independent).

Before checking whether the expected and actual results of the operation under test match, we wait for a couple of *ns* just to make sure we do not cause a timing violation. After this, and in case the result obtained from the ALU and the expected one do not match, we increase the error counter and print the error to xcelium console.

Finally, we increase the counter of samples until all samples have been tested for a particular operation, and then the operation counter is increased so that all operations are tested. Once all operations and samples have been tested, the *finish_simulation* flag is set to 1, thus activation the last *always* block defined in the testbench, which will print a summary of the results obtained in the testbench to the xcelium console right before stopping the simulation.

Code 15: Testbench used to test the ALU designed and implemented in this work (<*ALU_tb.v*>)

```
1     //timeunit 1ns;
2     //timeprecision 1fs;
3     `timescale 1 ns / 1 fs
4
5     // Instantiate testbench module
6     module ALU_32_tb;
7
```

```
8       // Create clock registers and init parameters (NOTICE CLOCK IS JUST NECESSARY
9       // FOR THE TESTBENCH, ALU IS PURELY COMBINATIONAL)
10      reg clk = 0;
11      `define HALF_CLK_PERIOD 12.5
12
13      // Create registers for simulation states
14      reg finish_simulation;
15
16      // ALU_CTL CODES
17      parameter ALU_CTL_ADD = 6'b100000;
18      parameter ALU_CTL_SUB = 6'b100010;
19
20      parameter ALU_CTL_AND = 6'b100100;
21      parameter ALU_CTL_XOR = 6'b100110;
22      parameter ALU_CTL_OR = 6'b100101;
23      parameter ALU_CTL_NOR = 6'b100111;
24
25      parameter ALU_CTL_SLL = 6'b000000;
26      parameter ALU_CTL_SRL = 6'b000010;
27      parameter ALU_CTL_SLT = 6'b101010;
28      parameter ALU_CTL_SLTU = 6'b101011;
29
30      // Create counter register to keep track of counts
31      parameter MAX_COUNT = 20;
32      reg [31:0] counter;
33      reg [31:0] op_counter;
34
35      // Create register to hold the expected value (calculated in testbench,
36      // outside DUT),
37      // as well as bookeeping variables to keep track of mismatches between expect vs
38      // actual results
39      reg [31:0] expected_result;
40      reg [31:0] num_errors;
41
42      // Create ALU signal registers
43      reg [31:0] ALU_OP_A; //ALU operator A
44      reg [31:0] ALU_OP_B; //ALU operator B
45      wire [31:0] ALU_OP_OUT; //ALU operator OUT
46      reg [10:0] ALU_CTL; //ALU control line
47      wire ALU_OVERFLOW;
48      wire ALU_ZERO;
49      wire ALU_CARRYOUT; //ALU overflow, carryout and zero flags
50
51      // Instantiate DUT (Device under test)
52      ALU_32 DUT (
53            .A(ALU_OP_A),
54            .B(ALU_OP_B),
55            .op_ctl(ALU_CTL),
56            .Z(ALU_OP_OUT),
57            .overflow(ALU_OVERFLOW),
58            .zero(ALU_ZERO),
59            .carryout(ALU_CARRYOUT));
60
61      // Setup trace file
62      initial begin
63         $dumpfile ("dut.vcd");
64         $dumpvars;
65
```

```
66          // Initialize variables
67          counter = 0;
68          op_counter = 0;
69          finish_simulation = 1'b0;
70
71          // Initialize operators
72          ALU_OP_A = 32'b0;
73          ALU_OP_B = 32'b0;
74
75          // Initialize number of counted errors
76          num_errors = 0;
77
78       end
79
80       // Generate testbench clock
81       always begin
82          #`HALF_CLK_PERIOD clk = ~clk;
83       end
84
85
86       // Increase operators and operation counters
87       always @(posedge clk) begin
88
89             // Perform test only if simulation not finished
90             if (finish_simulation == 1'b0) begin
91
92                   // Make sure to initialize shamt to zero
93                   ALU_CTL[10:6] = 5'b0;
94
95                   // Now Get random values for operators
96                   ALU_OP_A = $random;
97                   ALU_OP_B = $random;
98
99                   // Run different operation depending on counter
100                  case (op_counter)
101                        0:  begin
102                                    ALU_CTL[5:0] = ALU_CTL_AND;
103                                    expected_result = (ALU_OP_A & ALU_OP_B);
104                              end
105                        1:  begin
106                                    ALU_CTL[5:0] = ALU_CTL_OR;
107                                    expected_result = (ALU_OP_A | ALU_OP_B);
108                              end
109                        2:  begin
110                                    ALU_CTL[5:0] = ALU_CTL_XOR;
111                                    expected_result = (ALU_OP_A ^ ALU_OP_B);
112                              end
113                        3:  begin
114                                    ALU_CTL[5:0] = ALU_CTL_NOR;
115                                    expected_result = ~(ALU_OP_A | ALU_OP_B);
116                              end
117                        4:  begin
118                                    ALU_CTL[5:0] = ALU_CTL_ADD;
119                                    expected_result = (ALU_OP_A + ALU_OP_B);
120                              end
121                        5:  begin
122                                    ALU_CTL[5:0] = ALU_CTL_SUB;
123                                    expected_result = (ALU_OP_A - ALU_OP_B);
```

```
124                               end
125                         6:  begin
126                                     ALU_CTL[5:0] = ALU_CTL_SLT;
127                                     expected_result = 32'b0;
128                                     if ($signed(ALU_OP_A) < $signed(ALU_OP_B))
129                                        expected_result[0] = 1'b1;
130                               end
131                         7:  begin
132                                     ALU_CTL[5:0] = ALU_CTL_SLTU;
133                                     expected_result = 32'b0;
134                                     if (ALU_OP_A < ALU_OP_B) expected_result[0] = 1'b1;
135                               end
136                         8: begin
137                                     ALU_CTL[5:0] = ALU_CTL_SLL;
138                                     ALU_CTL[10:6] = $random;
139                                     expected_result = (ALU_OP_A << ALU_CTL[10:6]);
140                               end
141                         9: begin
142                                     ALU_CTL[5:0] = ALU_CTL_SRL;
143                                     ALU_CTL[10:6] = $random;
144                                     expected_result = (ALU_OP_A >> ALU_CTL[10:6]);
145                               end
146                   endcase
147
148                   // Wait certain amount of time
149                   #5;
150
151                   // Check if actual value is equal to value returned by the DUT
152                   if (ALU_OP_OUT != expected_result) begin
153                         $display("[%04d] - ERROR: Expected value %b but received %b",
154                            $time, expected_result, ALU_OP_OUT);
155                         // Increase error counter
156                         num_errors = num_errors + 1;
157                   end
158
159                   // Increase counter or stop simulation
160                   if (counter >= MAX_COUNT) begin
161                         if (op_counter >= 9) begin
162                               // Finish test
163                               finish_simulation = 1'b1;
164                         end else begin
165                               // Reset counter and increase op_counter
166                               op_counter = op_counter + 1'b1;
167                         end
168                         counter = 0;
169                   end else begin
170                         // Increase counter
171                         counter = counter + 1;
172                   end
173
174             end
175
176       end
177
178
179       // Stop testbench after 100 counts after finish_simulation signal goes high
180       always @(posedge clk) begin
181             if (finish_simulation == 1'b1) begin
```

```
182                     // Display error info before leaving
183                     if (num_errors == 0) begin
184                           $display("[%04d] - VALIDATION: PASSED with no errors.
185                              ALU WORKING!", $time);
186                     end else begin
187                           $display("[%04d] - VALIDATION: FAILED with %d errors.
188                              ALU __NOT__ WORKING!", \$time, num_errors);
189                     end
190
191                     $finish;
192               end
193         end
194
195     endmodule
```

9.1 Testbenching the HLS design (pre-synthesis)

In order to avoid conflict of files, we have created a folder called **run-post-hls** inside the verification folder, where the results of this validation step will be stored. From inside that directory, we can invoke the pre-synthesis verification step by running the following command in the prompt:

```
$ make -f ../../Makefile clean && make -f ../../Makefile run-post-hls
```

An overview of the whole simulation can be seen in Figure 11. The verification window was divided into 4 panels/groups to aid visualization (left). At the top, some static parameters are shown. Then some of the control signals such as the clock and the counters. Right below this group, the operators that go into the ALU are displayed. Finally, at the bottom, the results of the ALU, as well as the expected results (and the counter of errors) appear.

The first most important thing that should catch the eye in this simulation is that the total number of errors is 0 for the whole simulation. This means that both the value expected for each sample and operation, as well as the actual result computed by the ALU are exactly the same for all tested cases. Apart from that, it is possible to check that most of the values vary significantly through the whole process (take a look at the evolution of the overflow or the carryout signals). Notice that some of these signals might not be completely essential at some points of the simulation (for instance, if we are performing AND operation, the carryout value will lack meaning, even though the ALU will compute its value and output it).

On the other hand, it is hard to extract any more details from this Figure, due to its resolution. Let us take a closer look at certain steps of the process, so we can analyze better the validation itself.

Figures 12 to 21 show a close-up of random samples applied to the ALU to perform the operations *AND*, *OR*, *NOR*, *XOR*, *ADD*, *SUB*, *SLT*, *SLTU*, *SLL* and *SRL*, respectively. Manual computation of the operators *A* and *B* for each one of these operations show that, indeed, the values computed by the designed ALU are as expected.

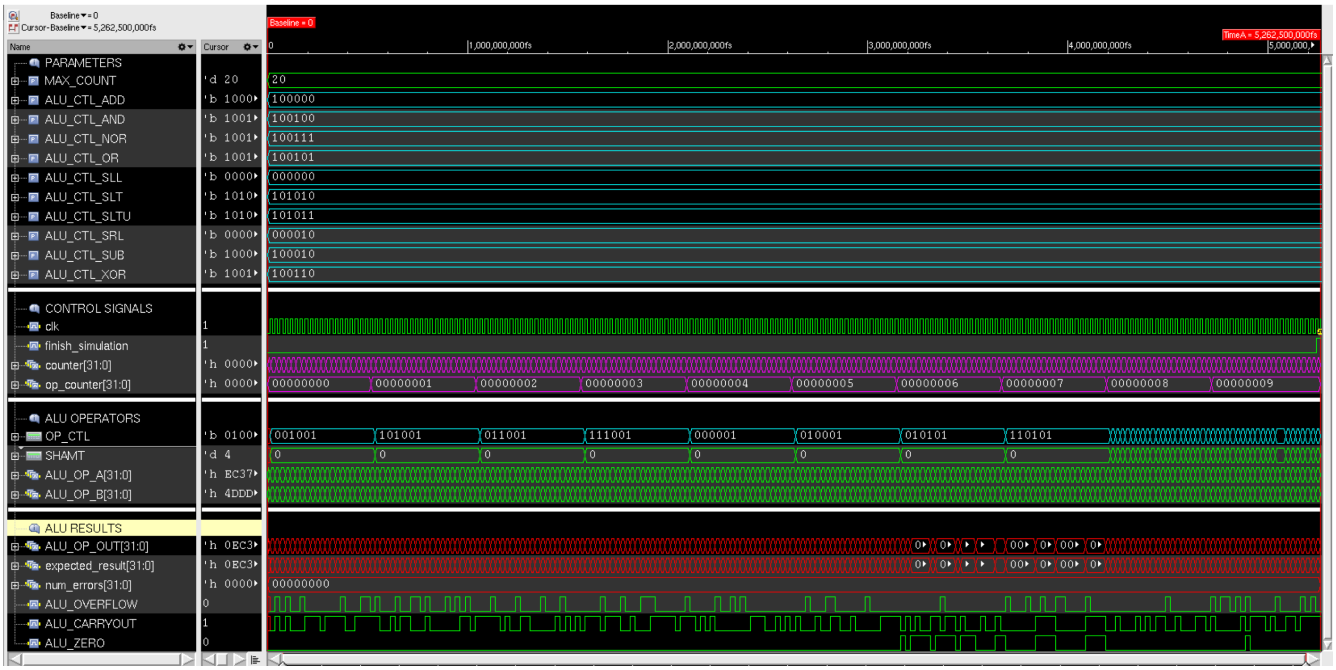

Figure 11. Overview of the whole simulation for the testbench specified in Code 15, which takes approximately 5000 *ns* to complete.

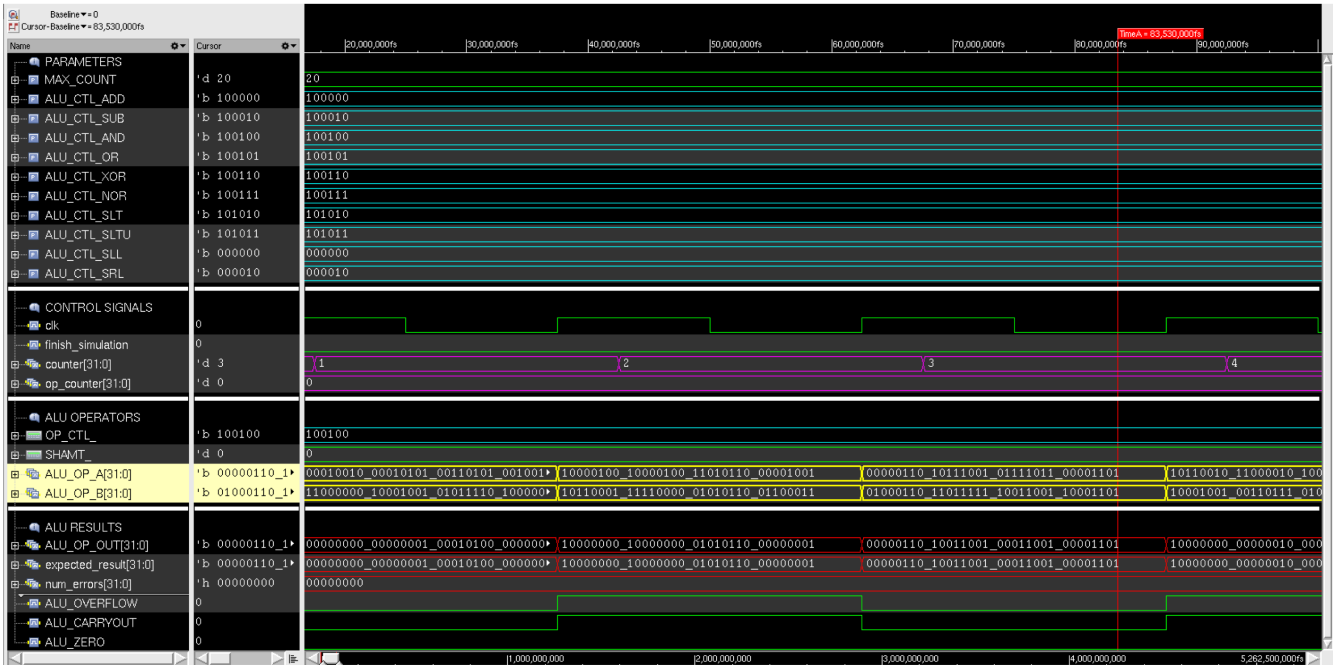

Figure 12. Close-up capture of the validation step for the *AND* operation.

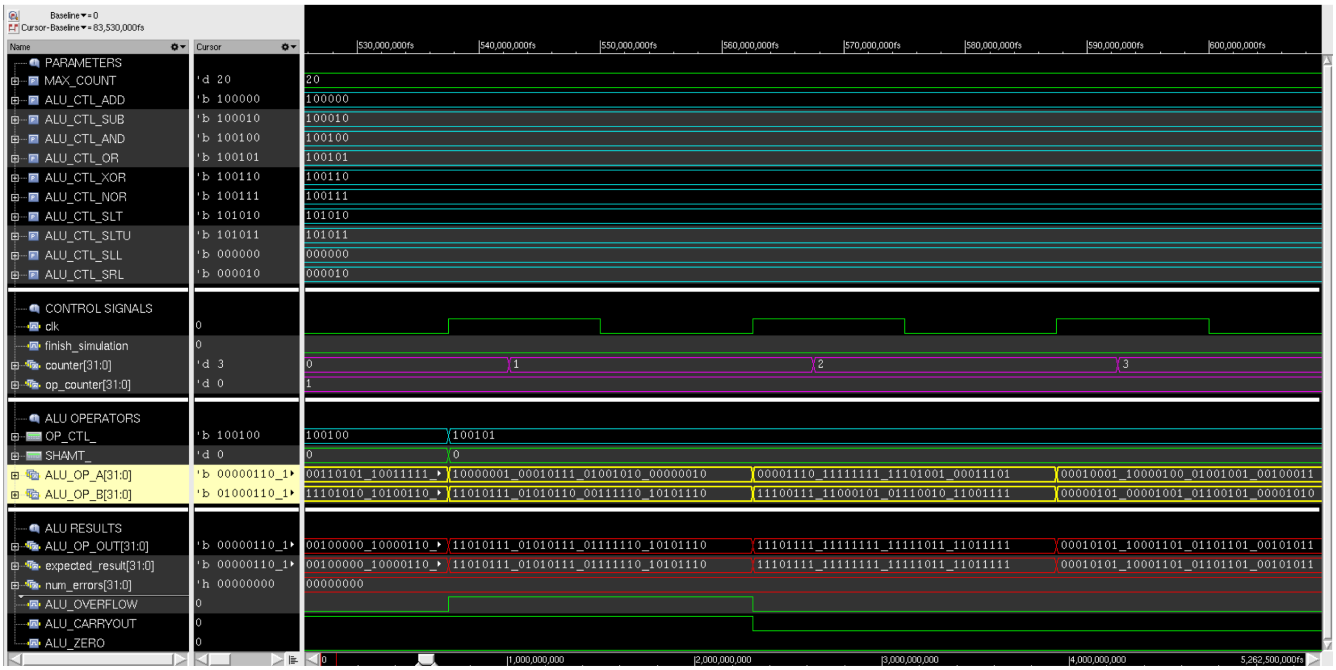

Figure 13. Close-up capture of the validation step for the *OR* operation.

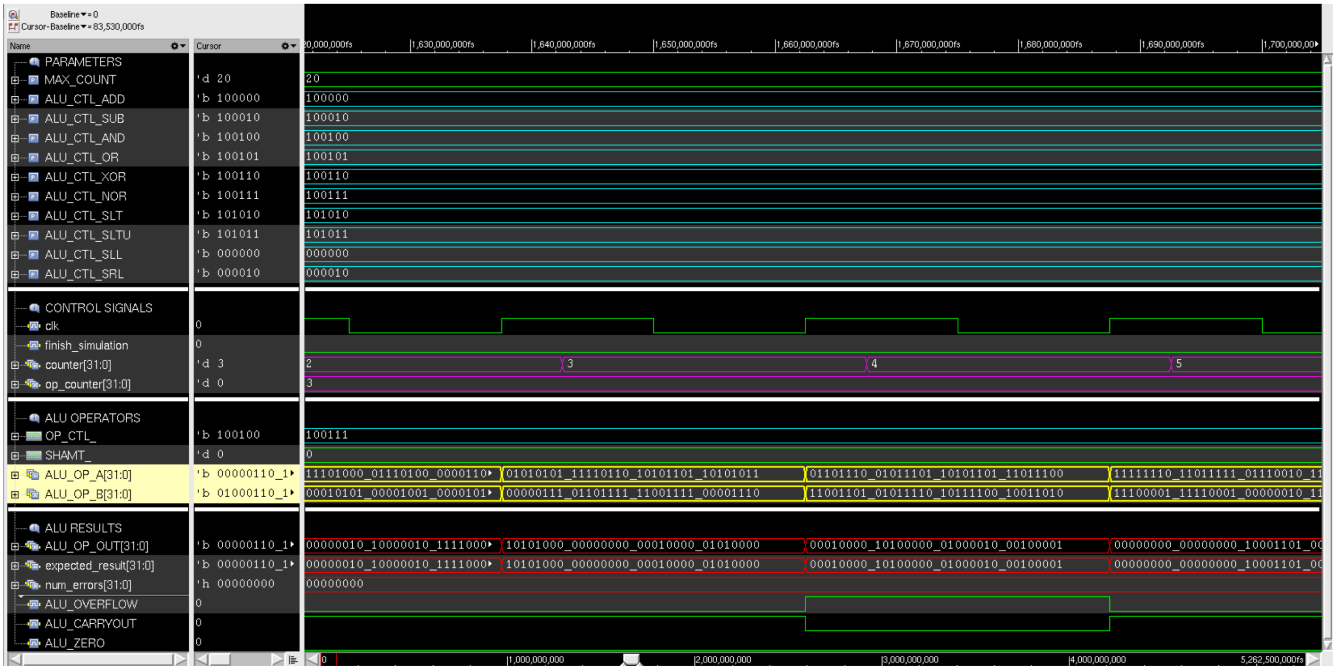

Figure 14. Close-up capture of the validation step for the *NOR* operation.

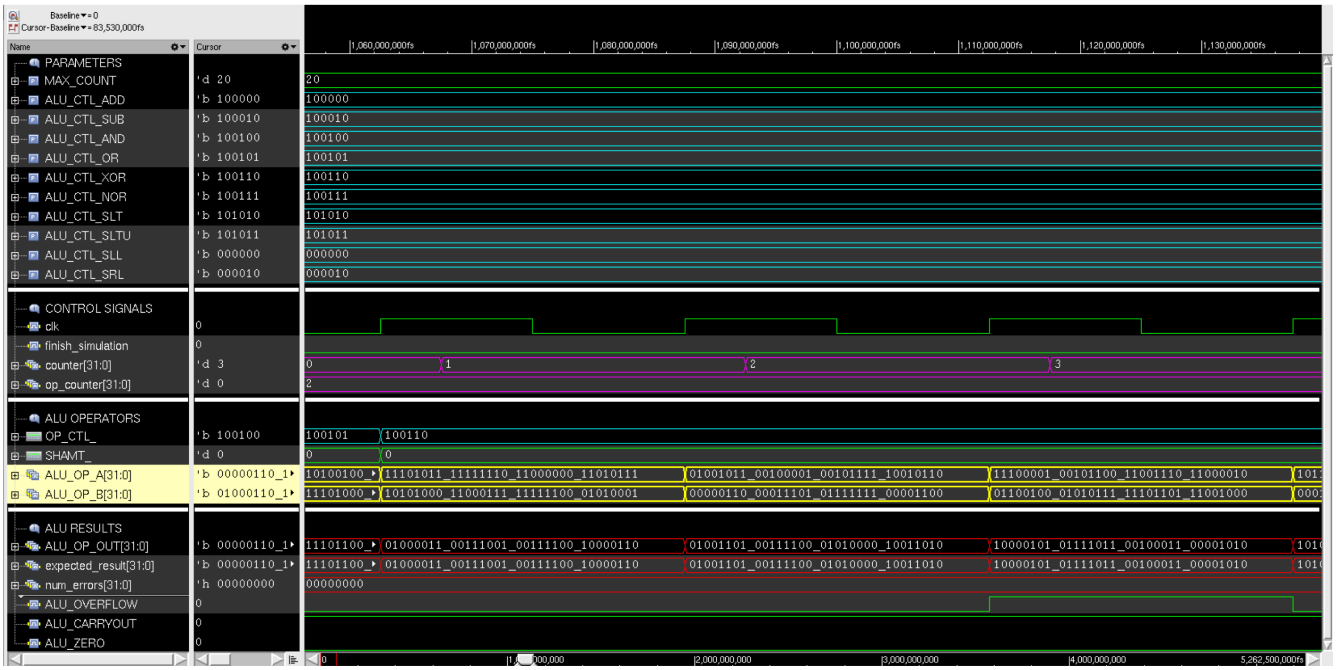

Figure 15. Close-up capture of the validation step for the *XOR* operation.

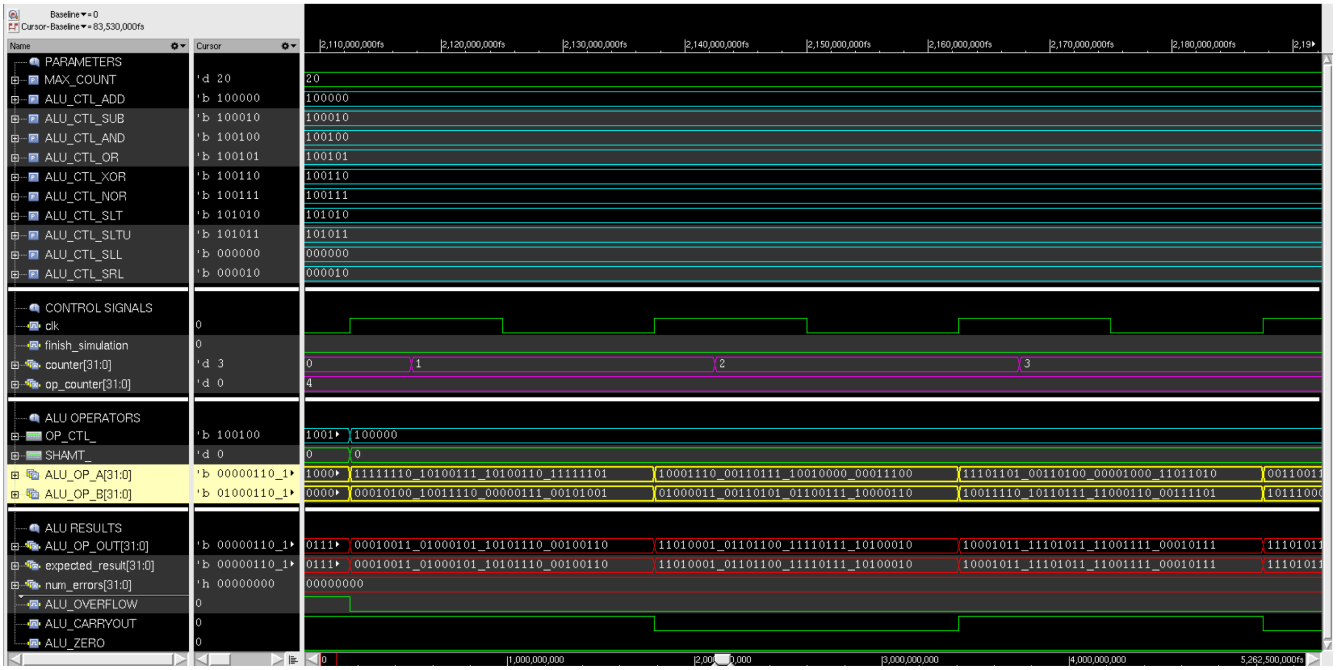

Figure 16. Close-up capture of the validation step for the *ADD* operation.

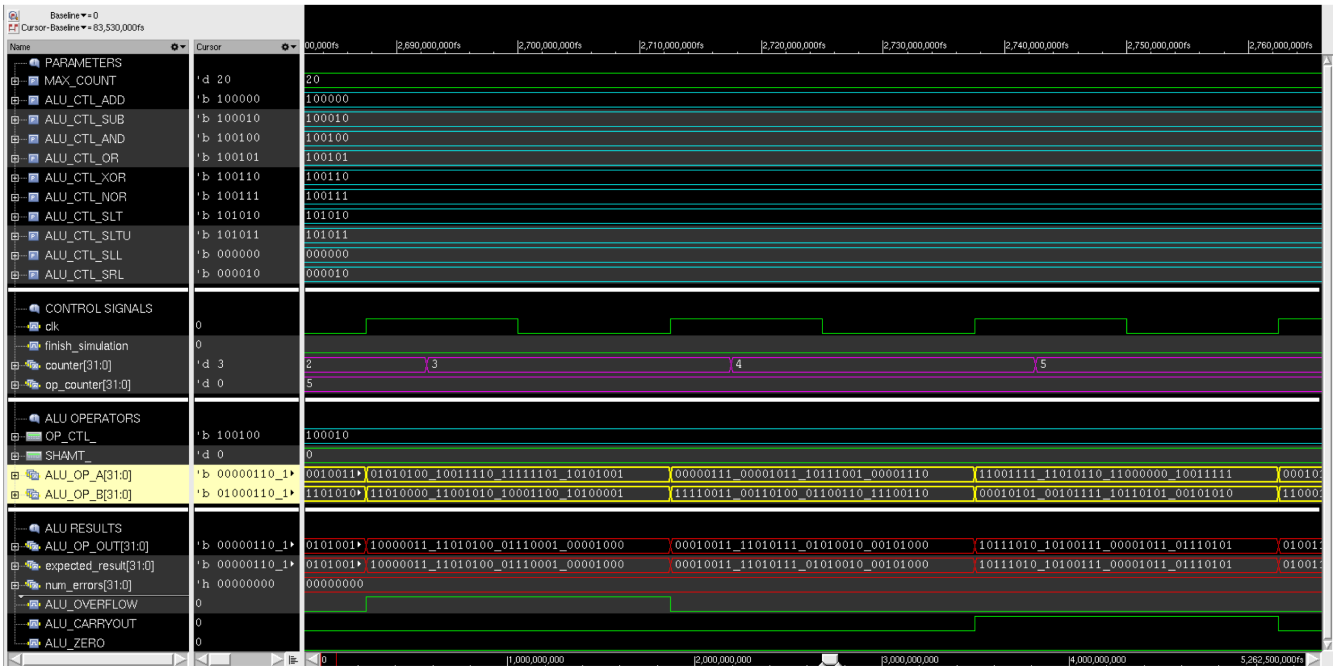

Figure 17. Close-up capture of the validation step for the *SUB* operation.

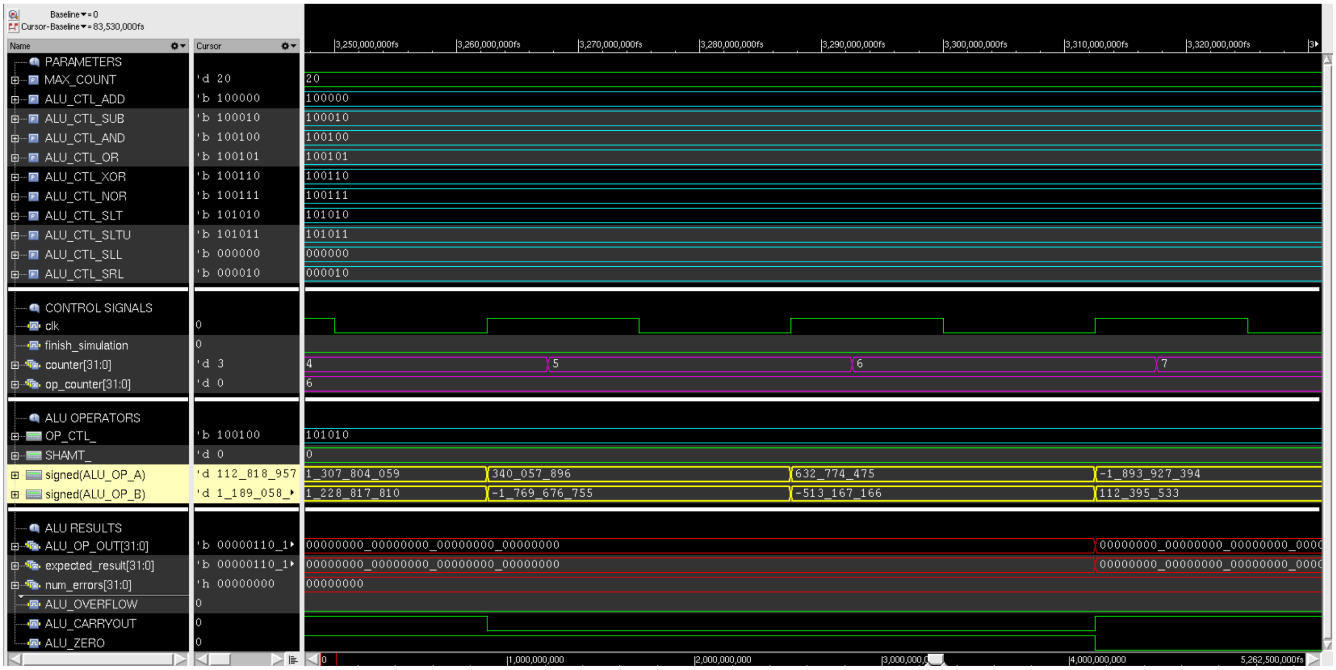

Figure 18. Close-up capture of the validation step for the *SLT* operation.

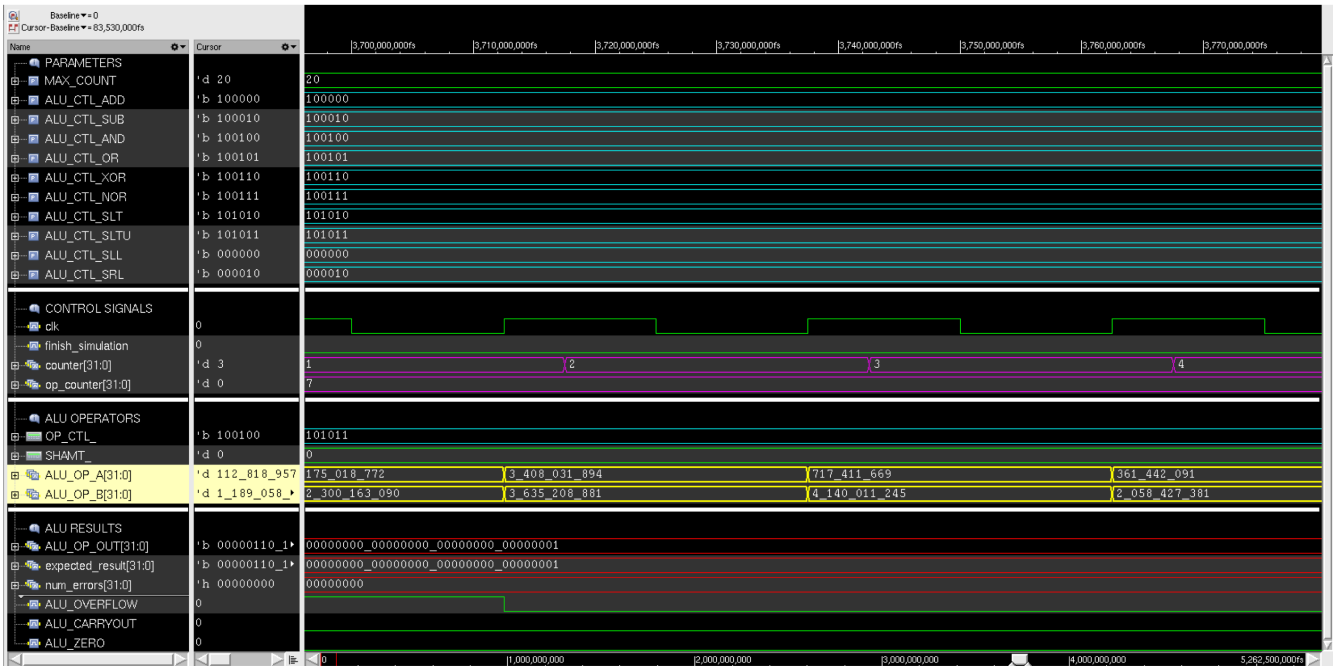

Figure 19. Close-up capture of the validation step for the *SLTU* operation.

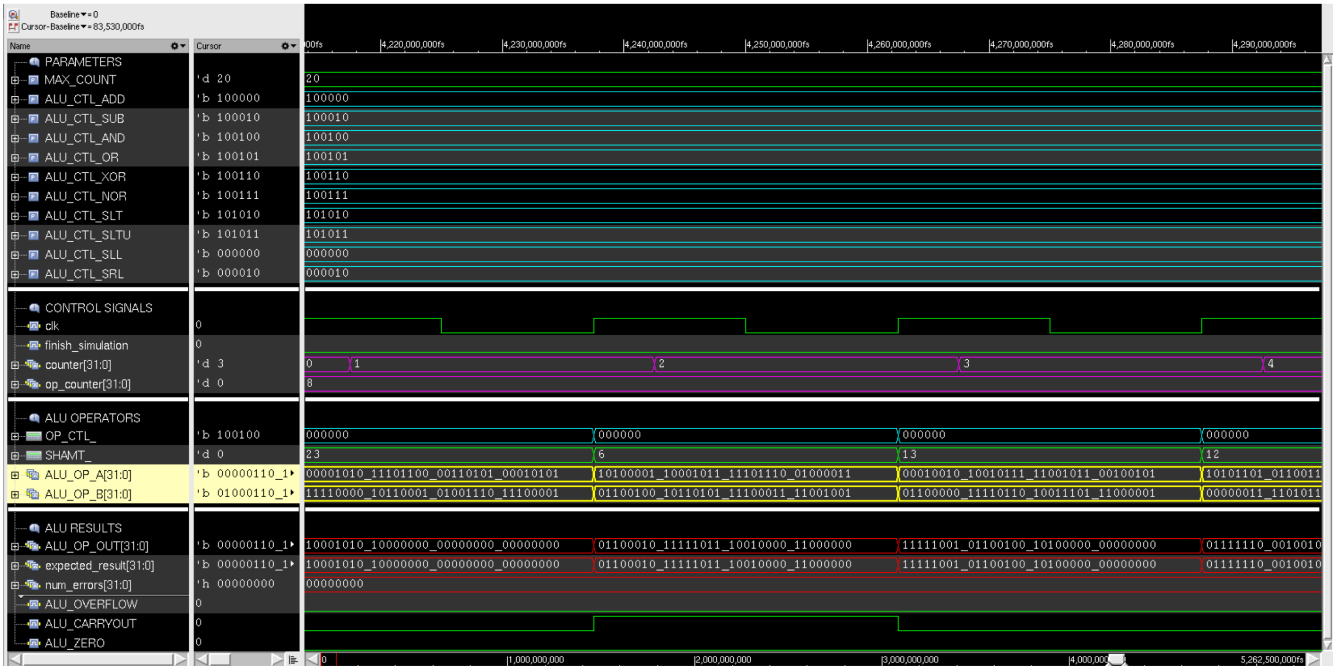

Figure 20. Close-up capture of the validation step for the *SLL* operation.

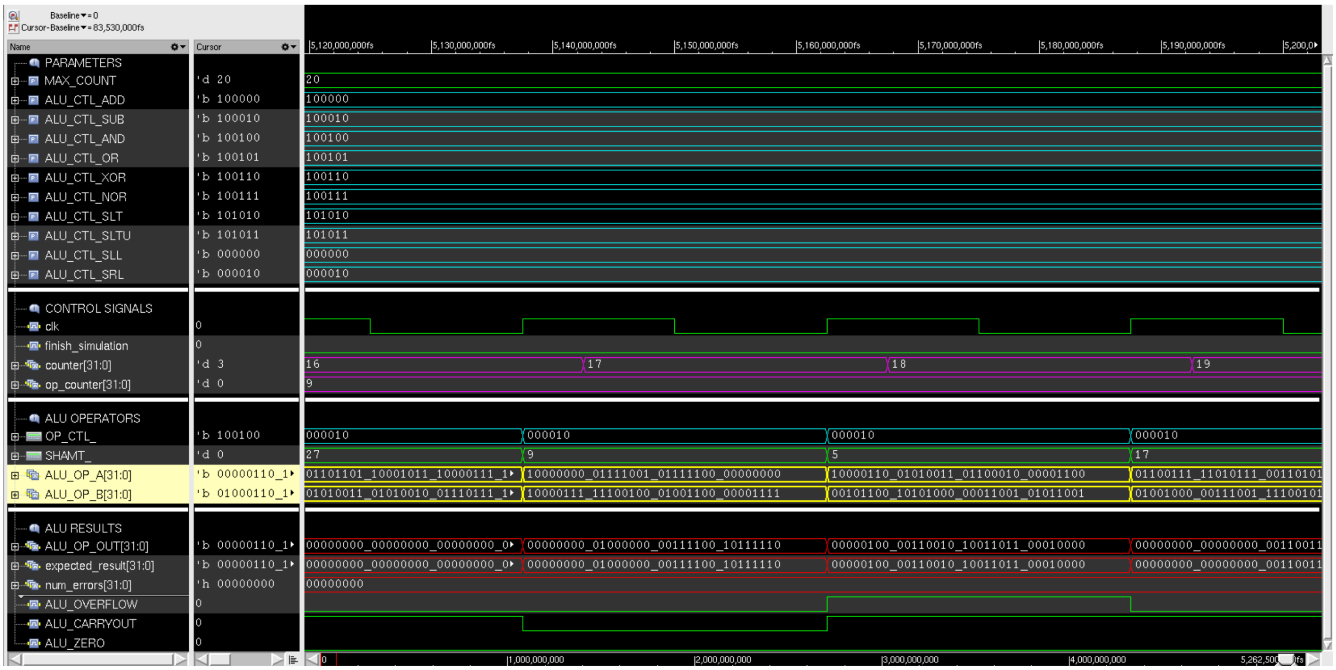

Figure 21. Close-up capture of the validation step for the *SRL* operation.

These results would suggest that the designed ALU, at least at HLS level (prior to synthesis), passed the test, and it is not only fully functional, but mainly works as expected.

9.2 Testbenching the netlist (post-synthesis)

In order to avoid conflict of files, we have created a folder called **run-post-syn** inside the verification folder, where the results of this validation step will be stored. From inside that directory, we can invoke the post-synthesis verification step by running the following command in the prompt:

```
$ make -f ../../Makefile clean && make -f ../../Makefile run-post-syn
```

An overview of the whole simulation can be seen in Figure **??**. The verification window was divided into the same 4 panels as introduced in the previous section.

As happened in the previous section, it is hard to extract any more details from this Figure, due to its resolution. Let us take a closer look at certain steps of the process, so we can analyze better the validation itself.

Figures 23 to 32 show a close-up of random samples applied to the ALU to perform the operations *AND*, *OR*, *NOR*, *XOR*, *ADD*, *SUB*, *SLT*, *SLTU*, *SLL* and *SRL*, respectively. Manual computation of the operators *A* and *B* for each one of these operations show that, indeed, the values computed by the designed ALU are as expected.

Two things should strike our eye in this case: the first, which can be checked in the overview Figure, is that, as occured with the post-syn case, the validation is complete with no errors; the second one is that there appears to be some instabilities in the values of the output registers of the ALU after each clock period. The effect of the second point is very noticeable, almost in all of the figures for the post-syn close-ups. These ripples might indicate a timing violation. One solution (which has not been investigated in this work) could be to change the clock frequency or to improve the design itself.

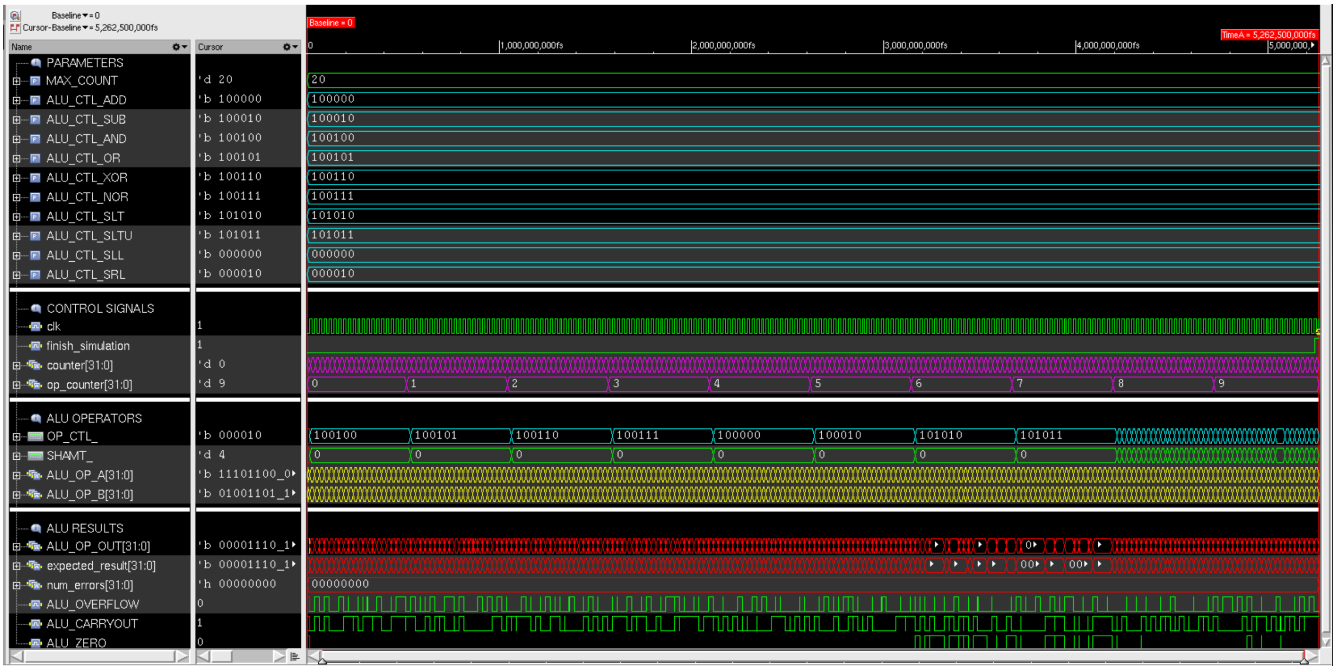

Figure 22. Overview of the whole simulation for the testbench specified in Code 15, which takes approximately 5000 *ns* to complete.

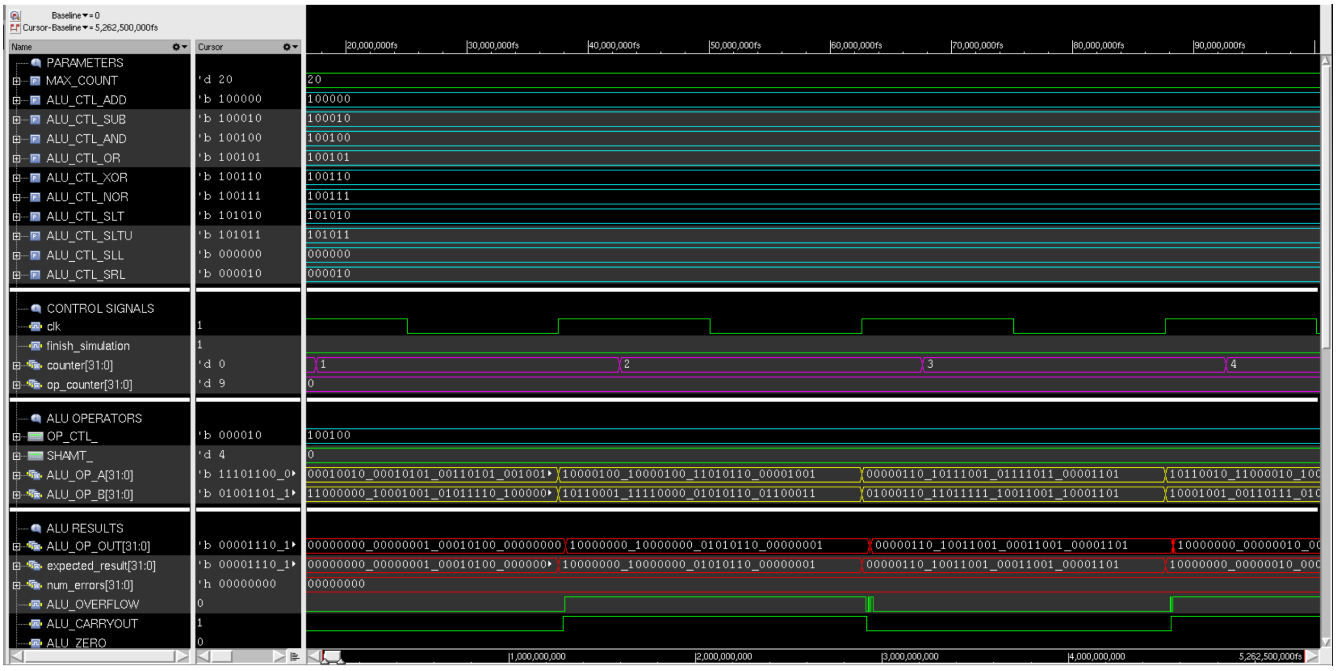

Figure 23. Close-up capture of the validation step for the *AND* operation.

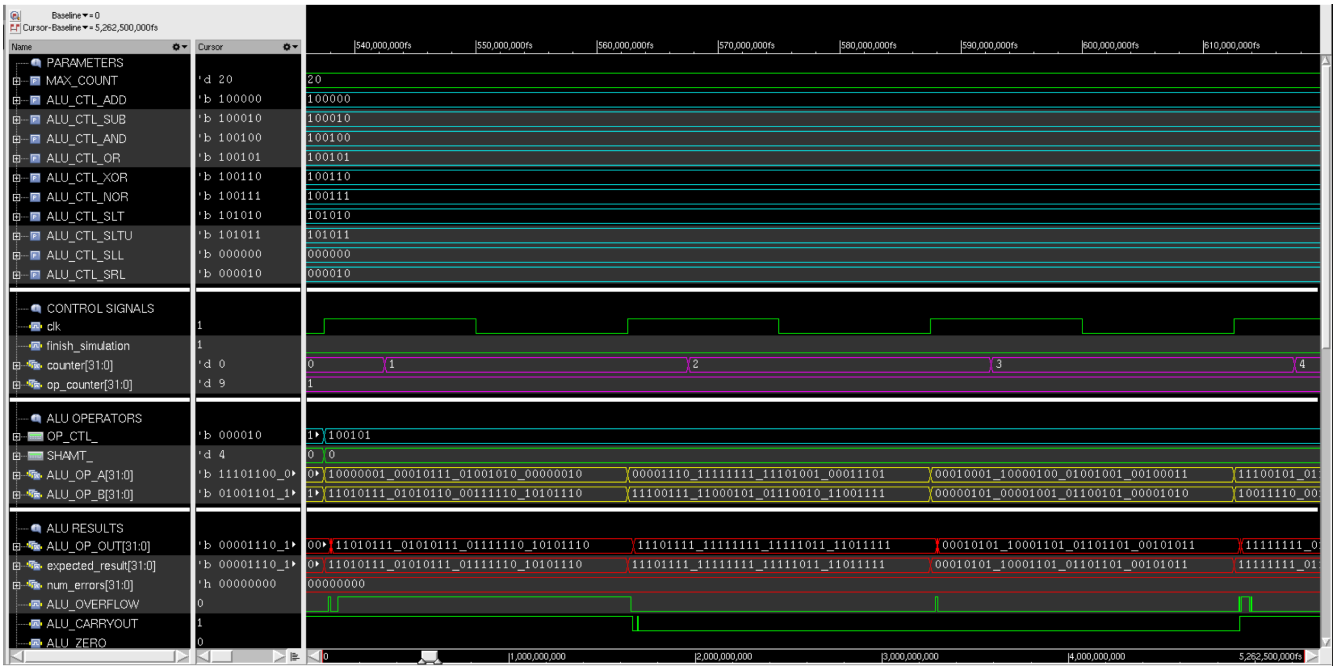

Figure 24. Close-up capture of the validation step for the *OR* operation.

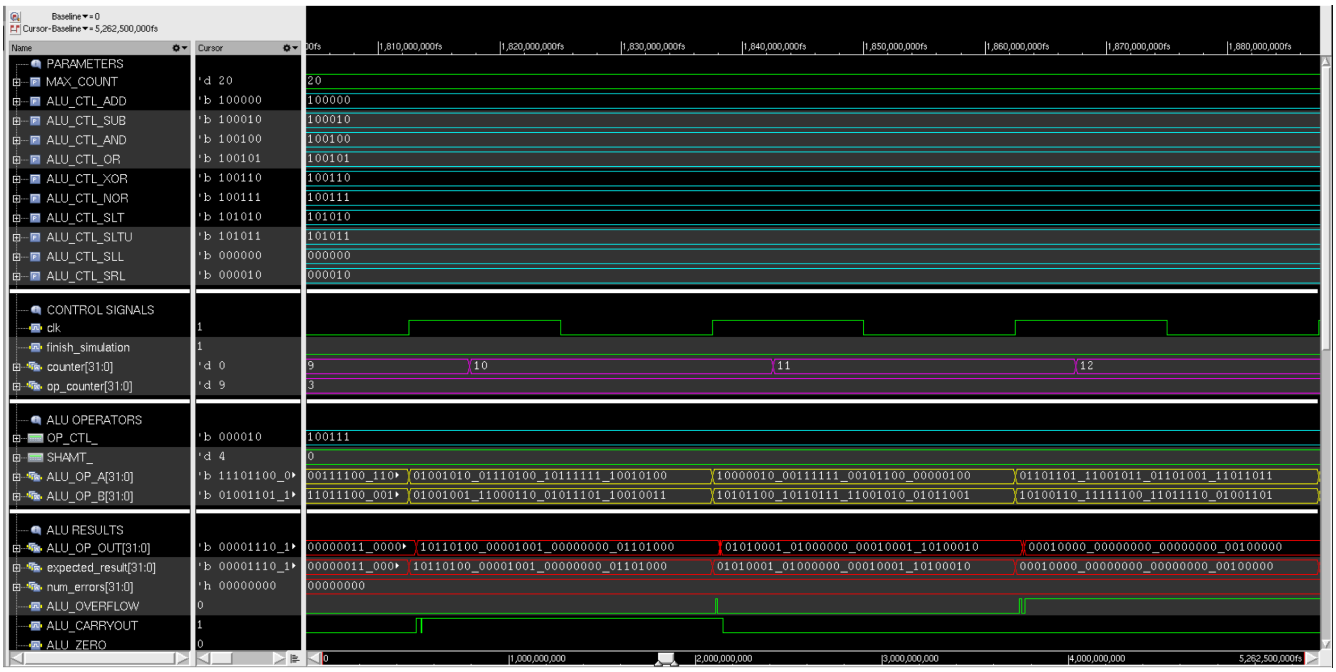

Figure 25. Close-up capture of the validation step for the *NOR* operation.

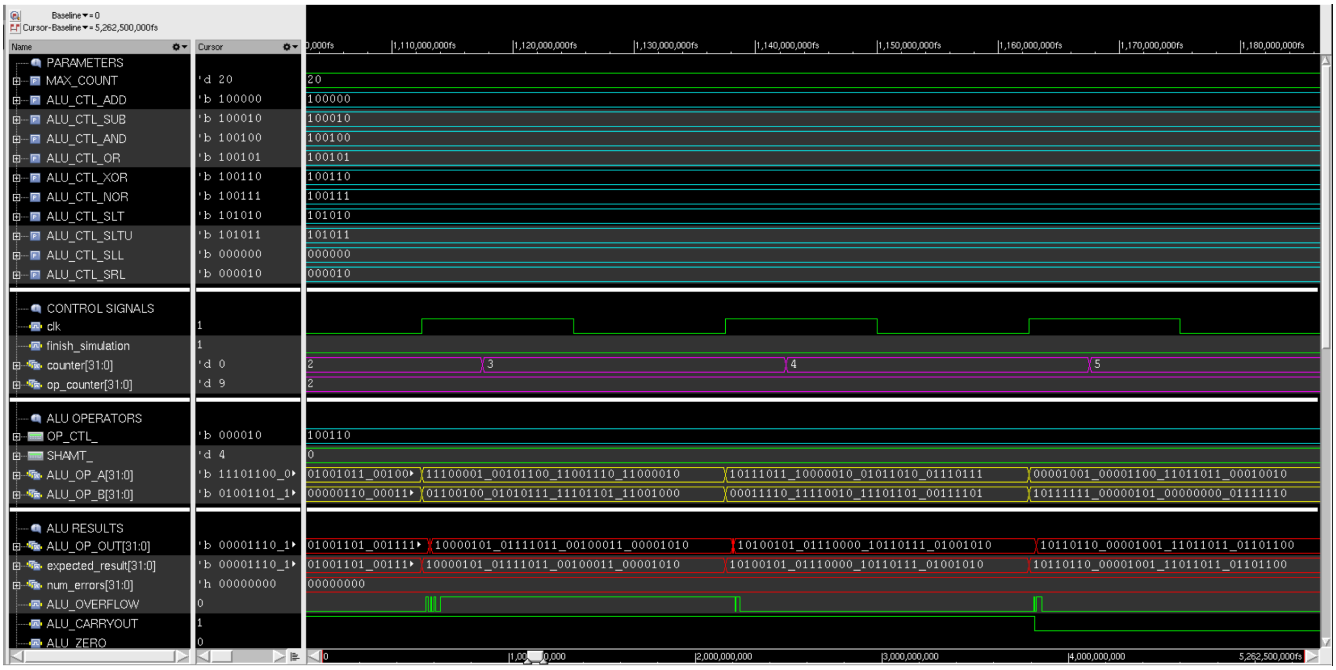

Figure 26. Close-up capture of the validation step for the *XOR* operation.

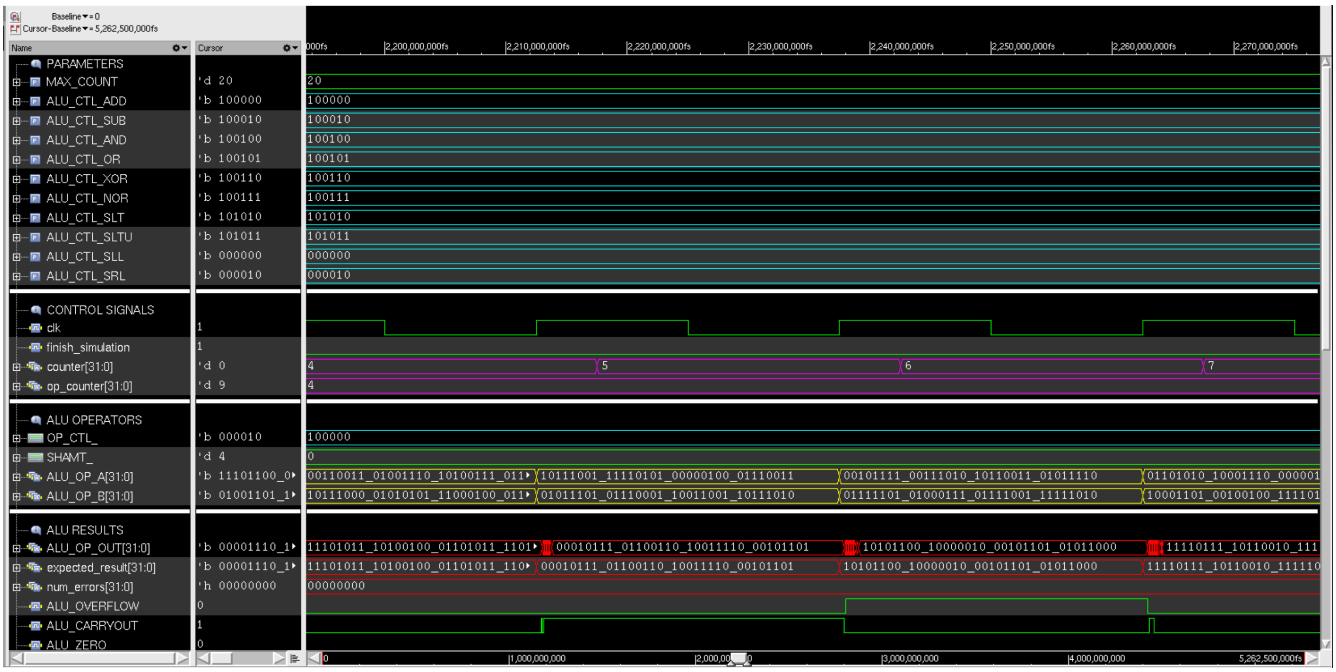

Figure 27. Close-up capture of the validation step for the *ADD* operation.

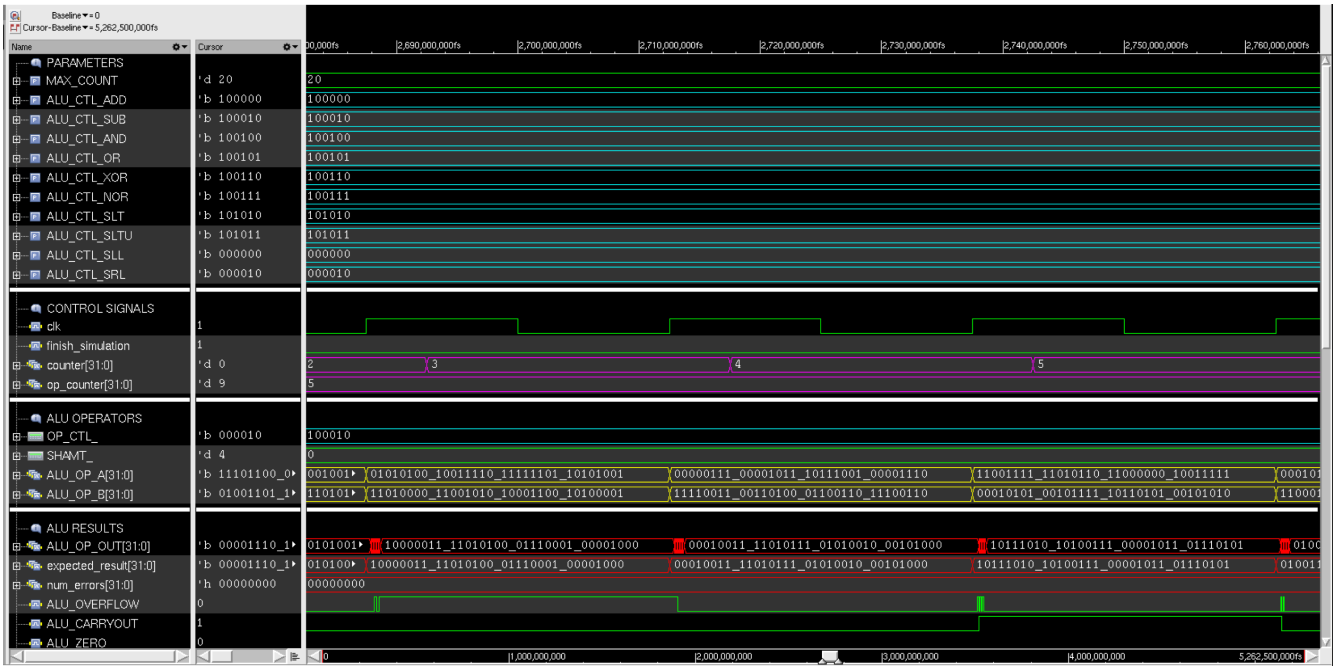

Figure 28. Close-up capture of the validation step for the *SUB* operation.

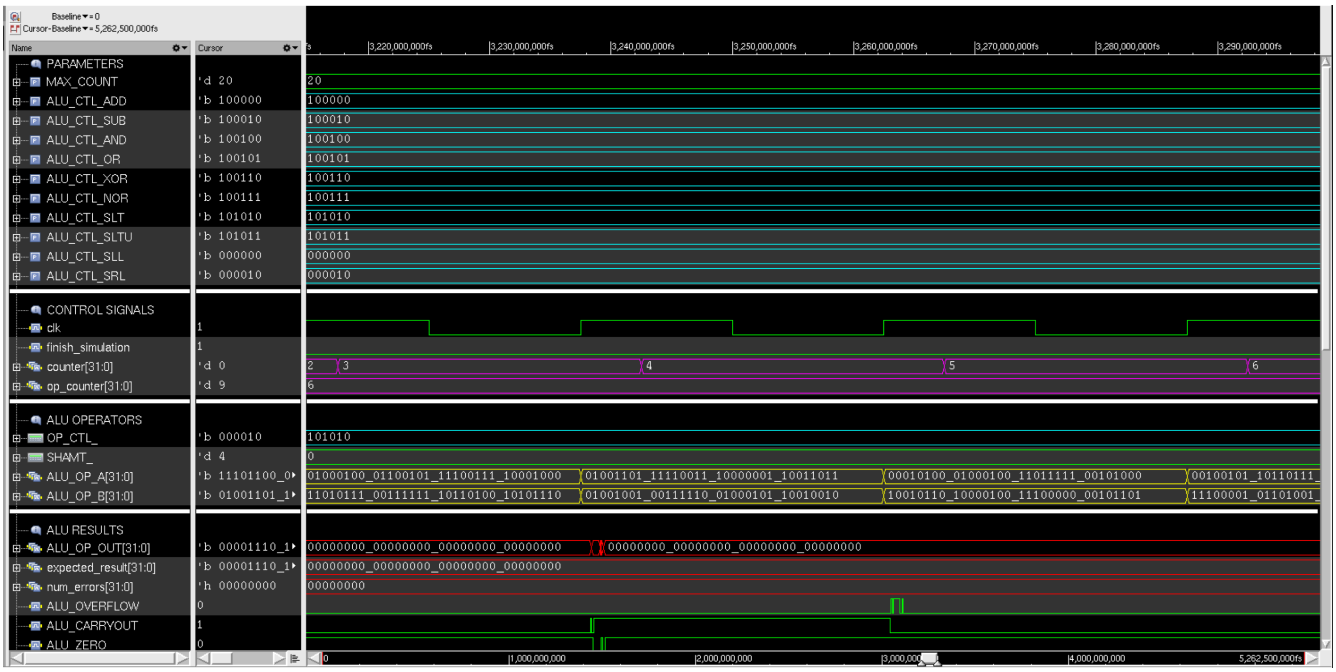

Figure 29. Close-up capture of the validation step for the *SLT* operation.

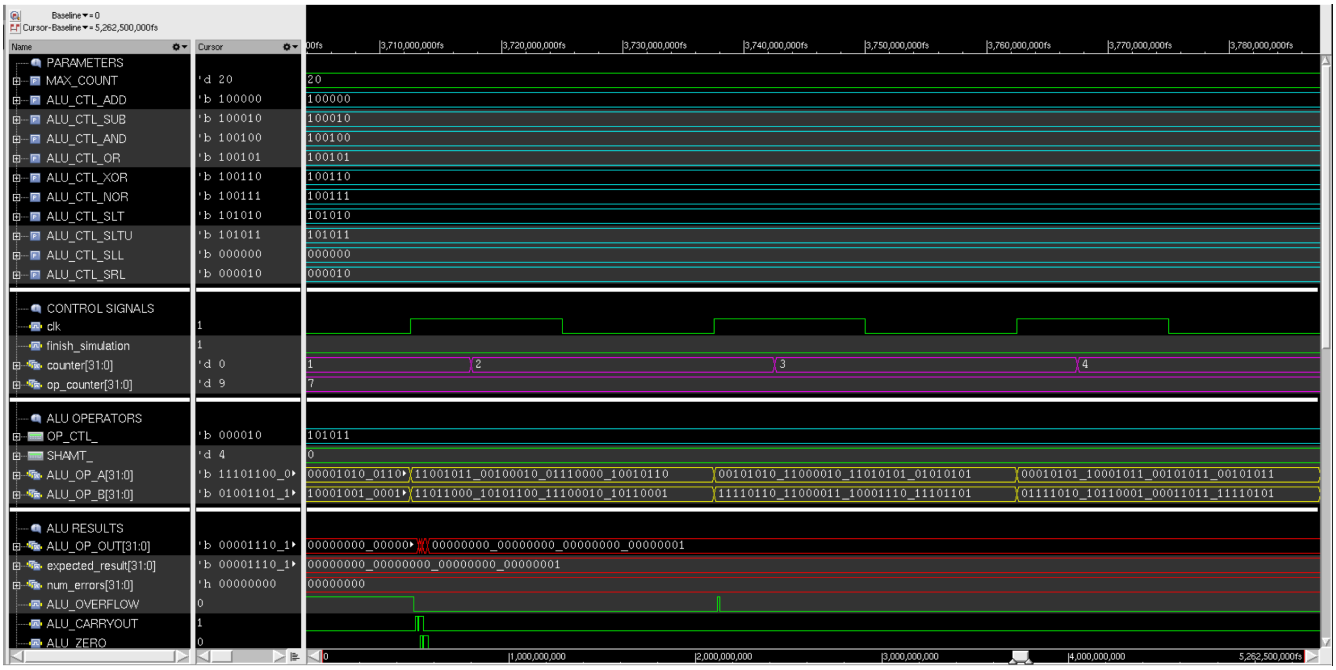

Figure 30. Close-up capture of the validation step for the *SLTU* operation.

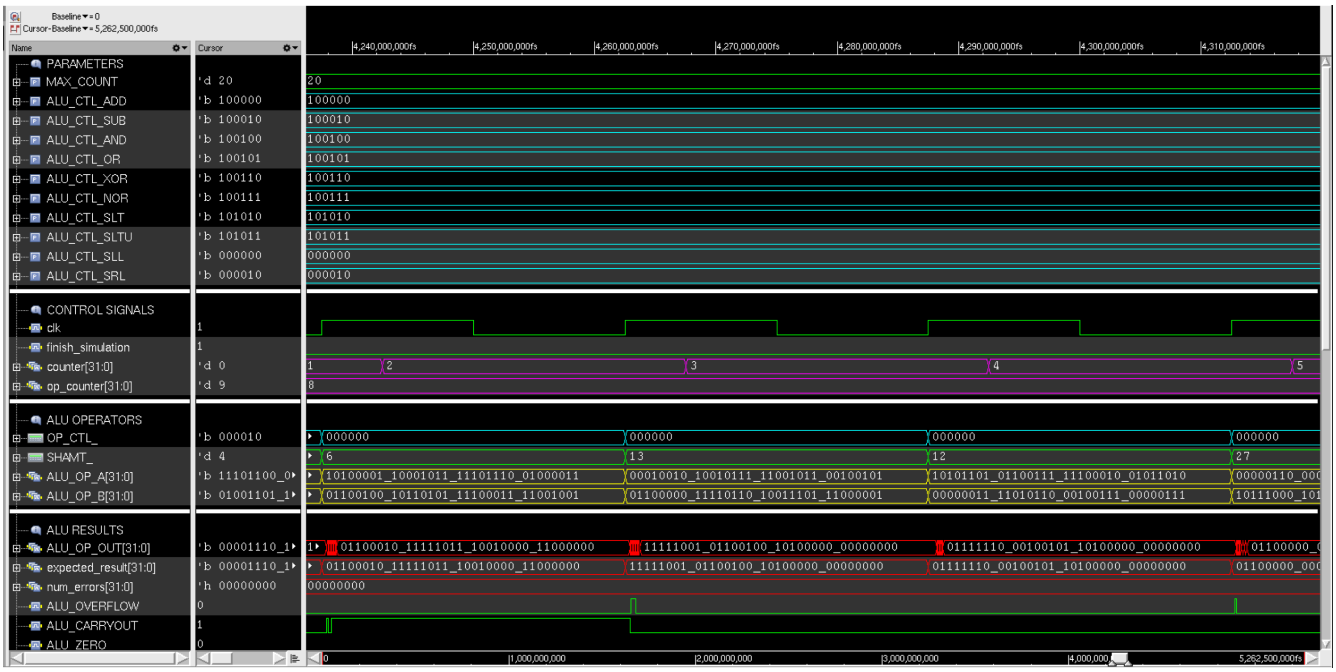

Figure 31. Close-up capture of the validation step for the *SLL* operation.

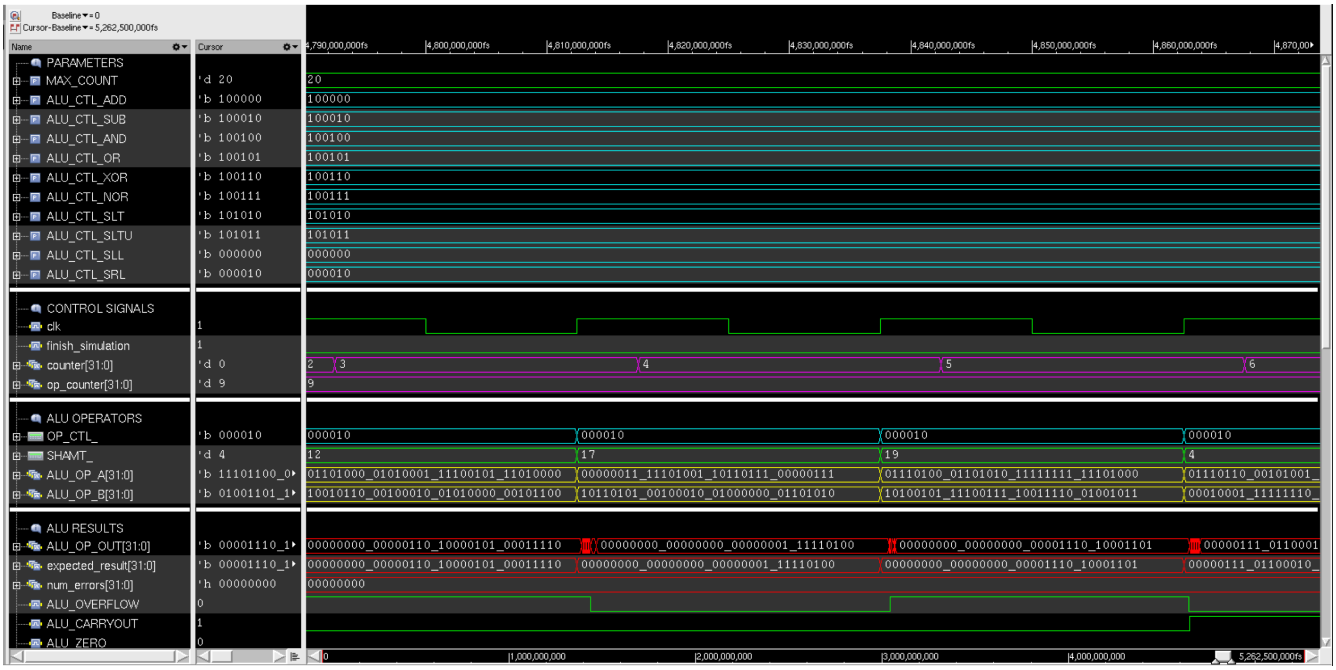

Figure 32. Close-up capture of the validation step for the *SRL* operation.

These results would suggest that the designed ALU, also for the synthesized netlist, passed the test, and it is not only fully functional, but mainly works as expected, despite some minor instabilities (ripples) found at clock transitions.

Section 10. *Conclusions*

In this work we have shown step-by-step how to implement a 32-bit arithmetic-logic unit (ALU), as well as all their blocks and components, by simply using purely combinatorial circuits (logic gates). The design has been implemented using verilog and synthesized using *CadenceGENUS*.

Both the original design and the synthesized netlist have been verified using *Cadencexcelium* tool with the same testbench, and in both cases our design passed the test.

We have shown that our 32-bit ALU is able to process 32-bit operators asynchronously and perform *AND*, *OR*, *XOR*, *NOR*, *ADD*, *SUB*, *SLT*, *SLTU*, *SLL*, *SRL*, returning the values expected for each tested case.

We tested each operation with 20 different totally random operators A and B, and no errors appeared in any of both the tested cases.

Finally, our design showed some level of instability and ripples in the output values of the ALU right after each clock cycle, which could cause some time violations (although no actual mismatch errors were noticed in our test). This could be probably solved by a further investigation and improvement of the design, and/or by tuning the clock cycle of the testbench, for instance.

In any case, we can conclude that the 32-bit ALU presented in this work is fully functional and operates as expected.

Section 11. *Appendix A: Synthesized Netlist*

Code 16: Netlist synthesized by genus from the implemented design (*<ALU_32_syn.v>*)

```
1
2     // Generated by Cadence Genus(TM) Synthesis Solution 19.10-p002_1
3     // Generated on: Oct 21 2020 05:23:00 CDT (Oct 21 2020 10:23:00 UTC)
4
5     // Verification Directory fv/ALU_32
6
7     module ALU_32(A, B, Z, op_ctl, overflow, zero, carryout);
8       input [31:0] A, B;
9       input [10:0] op_ctl;
10      output [31:0] Z;
11      output overflow, zero, carryout;
12      wire [31:0] A, B;
13      wire [10:0] op_ctl;
14      wire [31:0] Z;
15      wire overflow, zero, carryout;
16      wire n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7;
17      wire n_8, n_9, n_10, n_11, n_12, n_13, n_14, n_15;
18      wire n_16, n_17, n_18, n_19, n_20, n_21, n_22, n_23;
19      wire n_24, n_25, n_26, n_27, n_28, n_29, n_30, n_31;
20      wire n_32, n_33, n_34, n_35, n_36, n_37, n_38, n_39;
21      wire n_40, n_41, n_42, n_43, n_44, n_45, n_46, n_47;
22      wire n_48, n_49, n_50, n_51, n_52, n_53, n_54, n_55;
23      wire n_56, n_57, n_58, n_59, n_60, n_61, n_62, n_63;
24      wire n_64, n_65, n_66, n_67, n_68, n_69, n_70, n_71;
25      wire n_72, n_73, n_74, n_75, n_76, n_78, n_79, n_80;
26      wire n_81, n_82, n_83, n_84, n_85, n_86, n_87, n_88;
27      wire n_89, n_90, n_91, n_92, n_93, n_94, n_95, n_96;
28      wire n_97, n_98, n_99, n_100, n_101, n_102, n_103, n_104;
29      wire n_105, n_106, n_107, n_108, n_109, n_110, n_111, n_112;
30      wire n_113, n_114, n_115, n_116, n_117, n_118, n_119, n_120;
31      wire n_121, n_122, n_123, n_124, n_125, n_126, n_127, n_128;
32      wire n_129, n_130, n_131, n_132, n_133, n_134, n_135, n_136;
33      wire n_137, n_138, n_139, n_140, n_141, n_142, n_143, n_144;
34      wire n_145, n_146, n_147, n_148, n_149, n_150, n_151, n_152;
35      wire n_153, n_154, n_155, n_156, n_157, n_158, n_159, n_160;
36      wire n_161, n_162, n_163, n_164, n_165, n_166, n_167, n_168;
37      wire n_169, n_170, n_172, n_173, n_174, n_175, n_176, n_177;
38      wire n_178, n_179, n_180, n_181, n_182, n_183, n_184, n_185;
39      wire n_186, n_187, n_188, n_189, n_190, n_191, n_192, n_193;
40      wire n_194, n_195, n_196, n_197, n_198, n_199, n_200, n_201;
41      wire n_202, n_203, n_204, n_205, n_206, n_207, n_208, n_209;
42      wire n_210, n_211, n_212, n_213, n_214, n_215, n_216, n_217;
43      wire n_218, n_219, n_220, n_221, n_222, n_223, n_224, n_225;
44      wire n_226, n_227, n_228, n_229, n_230, n_231, n_232, n_233;
45      wire n_234, n_235, n_236, n_237, n_238, n_239, n_240, n_241;
46      wire n_242, n_243, n_244, n_245, n_246, n_247, n_248, n_249;
47      wire n_250, n_251, n_252, n_253, n_254, n_255, n_256, n_257;
48      wire n_258, n_259, n_260, n_261, n_262, n_263, n_264, n_265;
49      wire n_266, n_267, n_268, n_269, n_270, n_271, n_272, n_273;
50      wire n_274, n_275, n_276, n_277, n_278, n_279, n_280, n_281;
51      wire n_282, n_283, n_284, n_285, n_286, n_287, n_288, n_289;
52      wire n_290, n_291, n_292, n_293, n_294, n_295, n_296, n_297;
53      wire n_298, n_299, n_300, n_301, n_302, n_303, n_304, n_305;
54      wire n_306, n_307, n_308, n_309, n_311, n_312, n_313, n_314;
55      wire n_315, n_316, n_317, n_318, n_319, n_320, n_321, n_322;
```

```
56      wire n_323, n_324, n_325, n_326, n_327, n_328, n_329, n_331;
57      wire n_332, n_333, n_334, n_335, n_336, n_337, n_338, n_339;
58      wire n_340, n_341, n_342, n_343, n_344, n_345, n_346, n_347;
59      wire n_348, n_349, n_350, n_351, n_352, n_353, n_354, n_355;
60      wire n_357, n_358, n_359, n_360, n_361, n_362, n_363, n_364;
61      wire n_366, n_367, n_368, n_369, n_370, n_371, n_372, n_373;
62      wire n_374, n_375, n_379, n_380, n_381, n_382, n_383, n_384;
63      wire n_385, n_386, n_387, n_388, n_389, n_390, n_391, n_392;
64      wire n_393, n_395, n_396, n_397, n_398, n_399, n_400, n_401;
65      wire n_402, n_405, n_406, n_407, n_408, n_409, n_410, n_411;
66      wire n_412, n_413, n_414, n_415, n_416, n_417, n_418, n_419;
67      wire n_420, n_421, n_422, n_423, n_424, n_425, n_426, n_427;
68      wire n_428, n_429, n_430, n_431, n_432, n_433, n_434, n_435;
69      wire n_436, n_437, n_438, n_439, n_440, n_441, n_442, n_443;
70      wire n_444, n_445, n_446, n_447, n_449, n_450, n_451, n_452;
71      wire n_453, n_454, n_455, n_456, n_457, n_458, n_459, n_460;
72      wire n_461, n_462, n_463, n_464, n_465, n_466, n_467, n_468;
73      wire n_469, n_470, n_471, n_472, n_473, n_474, n_475, n_476;
74      wire n_477, n_478, n_479, n_480, n_481, n_482, n_483, n_484;
75      wire n_485, n_486, n_487, n_488, n_489, n_490, n_491, n_492;
76      wire n_493, n_494, n_495, n_497, n_498, n_499, n_500, n_501;
77      wire n_502, n_503, n_504, n_505, n_506, n_507, n_508, n_510;
78      wire n_511, n_512, n_513, n_514, n_515, n_516, n_517, n_518;
79      wire n_519, n_520, n_521, n_522, n_523, n_524, n_525, n_526;
80      wire n_528, n_529, n_530, n_531, n_532, n_533, n_534, n_535;
81      wire n_536, n_537, n_538, n_539, n_540, n_541, n_542, n_543;
82      wire n_544, n_545, n_546, n_547, n_548, n_549, n_550, n_551;
83      wire n_552, n_553, n_554, n_555, n_556, n_557, n_558, n_559;
84      wire n_560, n_561, n_562, n_563, n_564, n_565, n_566, n_567;
85      wire n_568, n_569, n_570, n_571, n_572, n_573, n_574, n_575;
86      wire n_576, n_577, n_578, n_579, n_580, n_581, n_582, n_583;
87      wire n_584, n_585, n_586, n_587, n_588, n_589, n_590, n_591;
88      wire n_592, n_593, n_594, n_595, n_596, n_597, n_598, n_599;
89      wire n_600, n_601, n_602, n_603, n_604, n_605, n_606, n_607;
90      wire n_608, n_609, n_610, n_611, n_612, n_613, n_614, n_615;
91      wire n_616, n_617, n_618, n_619, n_620, n_621, n_622, n_623;
92      wire n_624, n_625, n_626, n_627, n_628, n_629, n_630, n_631;
93      wire n_632, n_633, n_634, n_635, n_636, n_637, n_638, n_639;
94      wire n_640, n_642, n_643, n_644, n_645, n_646, n_647, n_648;
95      wire n_649, n_650, n_651, n_652, n_653, n_654, n_655, n_656;
96      wire n_657, n_658, n_659, n_660, n_661, n_662, n_663, n_664;
97      wire n_665, n_666, n_667, n_668, n_669, n_670, n_671, n_672;
98      wire n_673, n_674, n_675, n_676, n_677, n_678, n_679, n_680;
99      wire n_681, n_682, n_683, n_684, n_685, n_690, n_691, n_692;
100     wire n_693, n_694, n_695, n_696, n_697, n_698, n_699, n_700;
101     wire n_701, n_702, n_703, n_707, n_708, n_709, n_710, n_711;
102     wire n_712, n_713, n_714, n_715, n_716, n_717, n_718, n_719;
103     wire n_720, n_722, n_723, n_724, n_725, n_726, n_730, n_731;
104     wire n_733, n_734, n_736, n_737, n_738, n_739, n_741, n_742;
105     wire n_743, n_744, n_745, n_747, n_748, n_749, n_751, n_752;
106     wire n_753, n_754, n_755, n_756, n_757, n_758, n_759, n_760;
107     wire n_761, n_762, n_763, n_765, n_766, n_768, n_769, n_770;
108     wire n_774, n_775, n_777, n_778, n_779, n_781, n_782, n_784;
109     wire n_785, n_786, n_788, n_789, n_790, n_791, n_792, n_793;
110     wire n_794, n_795, n_796, n_797, n_798, n_799, n_800, n_801;
111     wire n_802, n_804, n_805, n_806, n_807, n_808, n_809, n_810;
112     wire n_812, n_813, n_815, n_819, n_820, n_822, n_823, n_824;
113     wire n_825, n_826, n_827, n_828, n_830, n_833, n_835, n_837;
```

```
114       wire n_838, n_842, n_843, n_845, n_846, n_848, n_849, n_916;
115       wire n_917, n_918, n_919, n_962, n_992, n_997, n_1005, n_1010;
116       wire n_1015, n_1020, n_1025, n_1030, n_1035, n_1040, n_1045, n_1050;
117       wire n_1055, n_1060, n_1065, n_1070, n_1075, n_1080, n_1085, n_1090;
118       wire n_1095, n_1100, n_1105, n_1110, n_1115, n_1120, n_1125, n_1130;
119       wire n_1138, n_1143, n_1151, n_1177, n_1184, n_1197, n_1210, n_1223;
120       wire n_1233, n_1241, n_1251, n_1256, n_1261, n_1266, n_1271, n_1276;
121       wire n_1281, n_1286, n_1291, n_1296, n_1301, n_1306, n_1311, n_1316;
122       wire n_1321, n_1326, n_1331, n_1336, n_1341, n_1346, n_1351, n_1356;
123       wire n_1361, n_1366, n_1371, n_1376, n_1381, n_1386, n_1391, n_1396;
124       wire n_1401, n_1406, n_1411, n_1416, n_1419, n_1424, n_1456, n_1861;
125       wire n_1934, n_2750, n_3287, n_3288, n_3289, n_3290, n_3291, n_3292;
126       wire n_3293, n_3294, n_3295, n_3296, n_3297;
127       NOR4_X1 g22084__8780(.A1 (n_837), .A2 (Z[0]), .A3 (n_846), .A4
128           (n_845), .ZN (zero));
129       NAND2_X1 g22085__4296(.A1 (n_849), .A2 (n_683), .ZN (Z[0]));
130       NAND4_X1 g22086__3772(.A1 (n_848), .A2 (n_13), .A3 (n_1251), .A4
131           (n_1416), .ZN (n_849));
132       OAI22_X1 g22089__1474(.A1 (n_824), .A2 (n_1411), .B1 (carryout), .B2
133           (n_175), .ZN (n_848));
134       NAND2_X1 g22088__4547(.A1 (n_828), .A2 (n_842), .ZN (n_846));
135       NAND2_X1 g22087__9682(.A1 (n_830), .A2 (n_843), .ZN (n_845));
136       INV_X1 g22097(.A (n_843), .ZN (Z[17]));
137       INV_X1 g22094(.A (n_838), .ZN (carryout));
138       AOI221_X1 g22099__2683(.A (n_605), .B1 (n_770), .B2 (n_815), .C1
139           (n_3287), .C2 (n_825), .ZN (n_843));
140       NOR3_X1 g22095__1309(.A1 (Z[31]), .A2 (Z[20]), .A3 (Z[29]), .ZN
141           (n_842));
142       AOI21_X1 g22096__6877(.A (n_833), .B1 (n_823), .B2 (n_696), .ZN
143           (n_838));
144       NAND4_X1 g22098__2900(.A1 (n_819), .A2 (n_835), .A3 (n_810), .A4
145           (n_820), .ZN (n_837));
146       INV_X1 g22102(.A (n_827), .ZN (Z[31]));
147       INV_X1 g22110(.A (n_835), .ZN (Z[30]));
148       MUX2_X1 g22101__2391(.A (n_833), .B (n_137), .S (n_826), .Z
149           (overflow));
150       INV_X1 g22090(.A (n_830), .ZN (Z[21]));
151       INV_X1 g22092(.A (n_828), .ZN (Z[19]));
152       AOI221_X1 g22105__7118(.A (n_439), .B1 (n_640), .B2 (n_822), .C1
153           (n_826), .C2 (n_825), .ZN (n_827));
154       AOI21_X1 g22100__8757(.A (n_7), .B1 (n_826), .B2 (n_10), .ZN (n_824));
155       OAI221_X1 g22111__1786(.A (n_681), .B1 (n_813), .B2 (n_684), .C1
156           (n_663), .C2 (n_806), .ZN (n_823));
157       AOI21_X1 g22113__5953(.A (n_626), .B1 (n_807), .B2 (n_825), .ZN
158           (n_835));
159       AOI221_X1 g22093__5703(.A (n_596), .B1 (n_737), .B2 (n_822), .C1
160           (n_808), .C2 (n_825), .ZN (n_828));
161       AOI221_X1 g22091__7114(.A (n_617), .B1 (n_718), .B2 (n_822), .C1
162           (n_809), .C2 (n_825), .ZN (n_830));
163       INV_X1 g22116(.A (n_820), .ZN (Z[16]));
164       NOR4_X1 g22122__5266(.A1 (n_802), .A2 (Z[28]), .A3 (Z[27]), .A4
165           (Z[15]), .ZN (n_819));
166       INV_X1 g22106(.A (n_812), .ZN (Z[20]));
167       AOI221_X1 g22118__2250(.A (n_631), .B1 (n_784), .B2 (n_815), .C1
168           (n_804), .C2 (n_825), .ZN (n_820));
169       MUX2_X1 g22119__6083(.A (n_697), .B (n_685), .S (n_805), .Z (n_826));
170       AOI221_X1 g22108__2703(.A (n_561), .B1 (n_722), .B2 (n_822), .C1
171           (n_3288), .C2 (n_825), .ZN (n_812));
```

```
172       INV_X1 g22107(.A (n_810), .ZN (Z[18]));
173       XNOR2_X1 g22103__5795(.A (n_801), .B (n_292), .ZN (n_809));
174       XNOR2_X1 g22104__7344(.A (n_799), .B (n_348), .ZN (n_808));
175       OAI21_X1 g22117__1840(.A (n_633), .B1 (n_798), .B2 (n_791), .ZN
176           (Z[29]));
177       XNOR2_X1 g22123__5019(.A (n_794), .B (n_806), .ZN (n_807));
178       NOR2_X1 g22121__1857(.A1 (n_805), .A2 (n_556), .ZN (n_813));
179       AOI221_X1 g22109__9906(.A (n_584), .B1 (n_751), .B2 (n_822), .C1
180           (n_797), .C2 (n_825), .ZN (n_810));
181       OAI21_X1 g22129__8780(.A (n_632), .B1 (n_792), .B2 (n_788), .ZN
182           (Z[28]));
183       HA_X1 g22124__4296(.A (n_793), .B (n_555), .CO (n_805), .S (n_804));
184       NAND4_X1 g22139__1474(.A1 (n_786), .A2 (n_774), .A3 (n_766), .A4
185           (n_775), .ZN (n_802));
186       OAI21_X1 g22114__4547(.A (n_336), .B1 (n_782), .B2 (n_800), .ZN
187           (n_801));
188       OAI21_X1 g22115__9682(.A (n_347), .B1 (n_796), .B2 (n_795), .ZN
189           (n_799));
190       OAI21_X1 g22125__2683(.A (n_825), .B1 (n_790), .B2 (n_789), .ZN
191           (n_798));
192       XOR2_X1 g22127__1309(.A (n_796), .B (n_795), .Z (n_797));
193       MUX2_X1 g22130__6877(.A (n_664), .B (n_662), .S (n_793), .Z (n_794));
194       XNOR2_X1 g22140__2900(.A (n_3289), .B (n_672), .ZN (n_792));
195       AND2_X1 g22128__2391(.A1 (n_790), .A2 (n_789), .ZN (n_791));
196       INV_X1 g22131(.A (n_785), .ZN (Z[15]));
197       OAI221_X1 g22133__7675(.A (n_494), .B1 (n_778), .B2 (n_788), .C1
198           (n_589), .C2 (n_753), .ZN (Z[27]));
199       INV_X1 g22143(.A (n_786), .ZN (Z[26]));
200       AOI221_X1 g22132__7118(.A (n_635), .B1 (n_784), .B2 (n_769), .C1
201           (n_777), .C2 (n_825), .ZN (n_785));
202       NAND2_X1 g22138__8757(.A1 (n_779), .A2 (n_499), .ZN (n_793));
203       AOI221_X1 g22144__1786(.A (n_535), .B1 (n_676), .B2 (n_822), .C1
204           (n_765), .C2 (n_825), .ZN (n_786));
205       MUX2_X1 g22135__5953(.A (n_674), .B (n_661), .S (n_781), .Z (n_790));
206       AOI21_X1 g22136__5703(.A (n_469), .B1 (n_781), .B2 (n_299), .ZN
207           (n_796));
208       AOI211_X1 g22137__7114(.A (n_510), .B (n_567), .C1 (n_781), .C2
209           (n_562), .ZN (n_782));
210       INV_X1 g22141(.A (n_781), .ZN (n_779));
211       XNOR2_X1 g22142__2250(.A (n_763), .B (n_646), .ZN (n_778));
212       XNOR2_X1 g22145__6083(.A (n_768), .B (n_458), .ZN (n_777));
213       INV_X1 g22147(.A (n_775), .ZN (Z[14]));
214       NOR4_X1 g22155__2703(.A1 (Z[24]), .A2 (n_756), .A3 (Z[23]), .A4
215           (Z[13]), .ZN (n_774));
216       AOI221_X1 g22148__5795(.A (n_609), .B1 (n_770), .B2 (n_769), .C1
217           (n_762), .C2 (n_825), .ZN (n_775));
218       AND2_X2 g22149__7344(.A1 (n_768), .A2 (n_417), .ZN (n_781));
219       INV_X1 g22150(.A (n_766), .ZN (Z[25]));
220       XNOR2_X1 g22156__1840(.A (n_761), .B (n_666), .ZN (n_765));
221       AOI221_X1 g22151__5019(.A (n_520), .B1 (n_691), .B2 (n_822), .C1
222           (n_757), .C2 (n_825), .ZN (n_766));
223       MUX2_X1 g22152__1857(.A (n_668), .B (n_647), .S (n_758), .Z (n_763));
224       OAI21_X1 g22159__9906(.A (n_622), .B1 (n_754), .B2 (n_788), .ZN
225           (Z[24]));
226       NAND2_X2 g22154__8780(.A1 (n_759), .A2 (n_360), .ZN (n_768));
227       XNOR2_X1 g22160__4296(.A (n_760), .B (n_755), .ZN (n_762));
228       MUX2_X1 g22161__3772(.A (n_657), .B (n_667), .S (n_760), .Z (n_761));
229       INV_X1 g22157(.A (n_758), .ZN (n_759));
```

```
230       XNOR2_X1 g22158__1474(.A (n_747), .B (n_655), .ZN (n_757));
231       NAND4_X1 g22171__4547(.A1 (n_749), .A2 (n_739), .A3 (n_731), .A4
232           (n_730), .ZN (n_756));
233       INV_X1 g22162(.A (n_752), .ZN (Z[13]));
234       AND2_X2 g22164__9682(.A1 (n_748), .A2 (n_755), .ZN (n_758));
235       XNOR2_X1 g22169__2683(.A (n_3290), .B (n_658), .ZN (n_754));
236       OAI221_X1 g22165__1309(.A (n_601), .B1 (n_743), .B2 (n_788), .C1
237           (n_579), .C2 (n_753), .ZN (Z[23]));
238       AOI221_X1 g22163__6877(.A (n_600), .B1 (n_751), .B2 (n_736), .C1
239           (n_742), .C2 (n_825), .ZN (n_752));
240       INV_X1 g22173(.A (n_749), .ZN (Z[22]));
241       INV_X1 g22167(.A (n_748), .ZN (n_760));
242       MUX2_X1 g22166__2900(.A (n_660), .B (n_656), .S (n_744), .Z (n_747));
243       NAND2_X1 g22168__2391(.A1 (n_745), .A2 (n_354), .ZN (n_748));
244       AOI221_X1 g22174__7675(.A (n_577), .B1 (n_711), .B2 (n_822), .C1
245           (n_738), .C2 (n_825), .ZN (n_749));
246       INV_X1 g22170(.A (n_744), .ZN (n_745));
247       XNOR2_X1 g22172__8757(.A (n_734), .B (n_669), .ZN (n_743));
248       XNOR2_X1 g22175__1786(.A (n_741), .B (n_351), .ZN (n_742));
249       AND2_X2 g22177__5953(.A1 (n_741), .A2 (n_287), .ZN (n_744));
250       INV_X1 g22178(.A (n_739), .ZN (Z[12]));
251       XNOR2_X1 g22183__5703(.A (n_733), .B (n_643), .ZN (n_738));
252       AOI221_X1 g22179__7114(.A (n_591), .B1 (n_737), .B2 (n_736), .C1
253           (n_724), .C2 (n_825), .ZN (n_739));
254       MUX2_X1 g22180__5266(.A (n_645), .B (n_670), .S (n_725), .Z (n_734));
255       NAND2_X2 g22182__2250(.A1 (n_726), .A2 (n_521), .ZN (n_741));
256       OAI21_X1 g22185__6083(.A (n_916), .B1 (n_723), .B2 (n_644), .ZN
257           (n_733));
258       INV_X1 g22188(.A (n_731), .ZN (Z[11]));
259       NOR4_X1 g22192__2703(.A1 (Z[10]), .A2 (Z[9]), .A3 (Z[8]), .A4
260           (n_707), .ZN (n_730));
261       INV_X1 g22184(.A (n_725), .ZN (n_726));
262       XNOR2_X1 g22186__5795(.A (n_723), .B (n_353), .ZN (n_724));
263       AOI221_X1 g22190__7344(.A (n_583), .B1 (n_722), .B2 (n_736), .C1
264           (n_720), .C2 (n_825), .ZN (n_731));
265       AND2_X2 g22187__1840(.A1 (n_723), .A2 (n_185), .ZN (n_725));
266       INV_X1 g22194(.A (n_719), .ZN (Z[10]));
267       XOR2_X1 g22193__1857(.A (n_717), .B (n_716), .Z (n_720));
268       AOI221_X1 g22195__9906(.A (n_616), .B1 (n_718), .B2 (n_736), .C1
269           (n_715), .C2 (n_825), .ZN (n_719));
270       MUX2_X2 g22191__8780(.A (n_132), .B (n_717), .S (n_716), .Z (n_723));
271       INV_X1 g22198(.A (n_712), .ZN (Z[9]));
272       OAI21_X2 g22196__4296(.A (n_218), .B1 (n_714), .B2 (n_713), .ZN
273           (n_717));
274       XOR2_X1 g22197__3772(.A (n_714), .B (n_713), .Z (n_715));
275       AOI221_X1 g22199__1474(.A (n_576), .B1 (n_711), .B2 (n_736), .C1
276           (n_710), .C2 (n_825), .ZN (n_712));
277       XOR2_X1 g22201__4547(.A (n_709), .B (n_708), .Z (n_710));
278       AOI21_X4 g22200__9682(.A (n_189), .B1 (n_709), .B2 (n_708), .ZN
279           (n_714));
280       OR4_X1 g22205__2683(.A1 (Z[7]), .A2 (Z[6]), .A3 (n_690), .A4 (Z[5]),
281           .ZN (n_707));
282       INV_X1 g22202(.A (n_703), .ZN (Z[8]));
283       AOI221_X1 g22203__1309(.A (n_574), .B1 (n_580), .B2 (n_736), .C1
284           (n_702), .C2 (n_825), .ZN (n_703));
285       OAI21_X4 g22204__6877(.A (n_699), .B1 (n_701), .B2 (n_124), .ZN
286           (n_709));
287       INV_X1 g22208(.A (n_700), .ZN (Z[7]));
```

```
288       XOR2_X1 g22206__2900(.A (n_698), .B (n_701), .Z (n_702));
289       AOI221_X1 g22209__2391(.A (n_565), .B1 (n_621), .B2 (n_736), .C1
290           (n_695), .C2 (n_825), .ZN (n_700));
291       NAND2_X2 g22207__7675(.A1 (n_698), .A2 (n_701), .ZN (n_699));
292       INV_X1 g22212(.A (n_692), .ZN (Z[6]));
293       XNOR2_X1 g22227__7118(.A (n_682), .B (n_696), .ZN (n_697));
294       OAI21_X4 g22210__8757(.A (n_223), .B1 (n_694), .B2 (n_693), .ZN
295           (n_698));
296       XOR2_X1 g22211__1786(.A (n_694), .B (n_693), .Z (n_695));
297       AOI221_X1 g22213__5953(.A (n_518), .B1 (n_680), .B2 (n_825), .C1
298           (n_691), .C2 (n_736), .ZN (n_692));
299       OR4_X1 g22219__5703(.A1 (Z[4]), .A2 (Z[3]), .A3 (Z[1]), .A4 (Z[2]),
300           .ZN (n_690));
301       INV_X1 g22216(.A (n_677), .ZN (Z[5]));
302       XNOR2_X1 g22229__7114(.A (n_684), .B (n_696), .ZN (n_685));
303       INV_X1 g22230(.A (n_675), .ZN (n_683));
304       AND2_X1 g22236__5266(.A1 (n_665), .A2 (n_681), .ZN (n_682));
305       XOR2_X1 g22215__2250(.A (n_679), .B (n_678), .Z (n_680));
306       AOI21_X4 g22214__6083(.A (n_186), .B1 (n_679), .B2 (n_678), .ZN
307           (n_694));
308       AOI221_X1 g22217__2703(.A (n_537), .B1 (n_652), .B2 (n_825), .C1
309           (n_676), .C2 (n_736), .ZN (n_677));
310       OAI221_X1 g22231__5795(.A (n_364), .B1 (n_919), .B2 (n_1251), .C1
311           (n_634), .C2 (n_363), .ZN (n_675));
312       MUX2_X1 g22234__7344(.A (n_673), .B (n_350), .S (n_672), .Z (n_674));
313       MUX2_X1 g22232__1840(.A (n_670), .B (n_341), .S (n_669), .Z (n_671));
314       MUX2_X1 g22235__5019(.A (n_179), .B (n_667), .S (n_666), .Z (n_668));
315       NAND2_X1 g22239__1857(.A1 (n_664), .A2 (n_637), .ZN (n_665));
316       INV_X1 g22241(.A (n_654), .ZN (n_663));
317       OAI21_X1 g22237__9906(.A (n_653), .B1 (n_661), .B2 (n_789), .ZN
318           (n_662));
319       INV_X1 g22221(.A (n_651), .ZN (Z[4]));
320       AND2_X1 g22240__8780(.A1 (n_638), .A2 (n_681), .ZN (n_684));
321       AOI22_X1 g22242__4296(.A1 (n_659), .A2 (n_312), .B1 (n_658), .B2
322           (n_194), .ZN (n_660));
323       OAI22_X1 g22243__3772(.A1 (n_656), .A2 (n_655), .B1 (n_636), .B2
324           (n_349), .ZN (n_657));
325       OAI221_X1 g22244__1474(.A (n_653), .B1 (n_620), .B2 (n_642), .C1
326           (n_430), .C2 (n_789), .ZN (n_654));
327       XOR2_X1 g22220__4547(.A (n_650), .B (n_649), .Z (n_652));
328       AOI221_X1 g22222__9682(.A (n_480), .B1 (n_627), .B2 (n_825), .C1
329           (n_588), .C2 (n_736), .ZN (n_651));
330       OAI21_X4 g22218__2683(.A (n_209), .B1 (n_650), .B2 (n_649), .ZN
331           (n_679));
332       MUX2_X1 g22251__1309(.A (n_647), .B (n_181), .S (n_646), .Z (n_648));
333       OAI22_X1 g22245__6877(.A1 (n_644), .A2 (n_335), .B1 (n_643), .B2
334           (n_183), .ZN (n_645));
335       AOI21_X1 g22252__2900(.A (n_210), .B1 (n_619), .B2 (n_182), .ZN
336           (n_673));
337       OAI21_X1 g22254__2391(.A (n_539), .B1 (n_623), .B2 (n_642), .ZN
338           (n_664));
339       OAI21_X1 g22233__7118(.A (n_548), .B1 (n_603), .B2 (n_639), .ZN
340           (Z[2]));
341       OAI21_X1 g22224__8757(.A (n_629), .B1 (n_614), .B2 (n_639), .ZN
342           (Z[3]));
343       OAI21_X1 g22249__1786(.A (n_637), .B1 (n_607), .B2 (n_538), .ZN
344           (n_638));
345       AOI21_X1 g22253__5953(.A (n_585), .B1 (n_606), .B2 (n_566), .ZN
```

```
346         (n_670));
347       AOI21_X1 g22250__5703(.A (n_426), .B1 (n_612), .B2 (n_636), .ZN
348         (n_667));
349       OAI221_X1 g22291__7114(.A (n_271), .B1 (n_630), .B2 (n_639), .C1
350         (n_634), .C2 (n_269), .ZN (n_635));
351       AOI221_X1 g22255__5266(.A (n_294), .B1 (n_547), .B2 (n_815), .C1
352         (n_602), .C2 (n_822), .ZN (n_633));
353       NOR2_X1 g22257__2250(.A1 (n_608), .A2 (n_553), .ZN (n_659));
354       AOI221_X1 g22268__6083(.A (n_311), .B1 (n_628), .B2 (n_815), .C1
355         (n_613), .C2 (n_822), .ZN (n_632));
356       OAI221_X1 g22278__2703(.A (n_243), .B1 (n_630), .B2 (n_753), .C1
357         (n_634), .C2 (n_242), .ZN (n_631));
358       AOI21_X1 g22261__5795(.A (n_461), .B1 (n_598), .B2 (n_272), .ZN
359         (n_661));
360       OAI211_X1 g22262__7344(.A (n_571), .B (n_281), .C1 (n_586), .C2
361         (n_570), .ZN (n_656));
362       AOI221_X1 g22226__1840(.A (n_295), .B1 (n_628), .B2 (n_769), .C1
363         (n_560), .C2 (n_825), .ZN (n_629));
364       XOR2_X1 g22225__5019(.A (n_625), .B (n_624), .Z (n_627));
365       OAI221_X1 g22256__1857(.A (n_400), .B1 (n_610), .B2 (n_753), .C1
366         (n_634), .C2 (n_256), .ZN (n_626));
367       OAI21_X1 g22258__9906(.A (n_433), .B1 (n_563), .B2 (n_432), .ZN
368         (n_644));
369       AOI21_X4 g22223__8780(.A (n_187), .B1 (n_625), .B2 (n_624), .ZN
370         (n_650));
371       AOI21_X1 g22271__4296(.A (n_492), .B1 (n_582), .B2 (n_358), .ZN
372         (n_623));
373       AOI221_X1 g22265__3772(.A (n_296), .B1 (n_550), .B2 (n_161), .C1
374         (n_621), .C2 (n_822), .ZN (n_622));
375       AOI21_X1 g22260__1474(.A (n_618), .B1 (n_611), .B2 (n_279), .ZN
376         (n_620));
377       OAI211_X1 g22272__4547(.A (n_493), .B (n_479), .C1 (n_568), .C2
378         (n_359), .ZN (n_619));
379       AOI21_X1 g22275__9682(.A (n_618), .B1 (n_572), .B2 (n_546), .ZN
380         (n_647));
381       OAI221_X1 g22300__2683(.A (n_253), .B1 (n_615), .B2 (n_595), .C1
382         (n_634), .C2 (n_252), .ZN (n_617));
383       OAI221_X1 g22303__1309(.A (n_265), .B1 (n_615), .B2 (n_599), .C1
384         (n_634), .C2 (n_264), .ZN (n_616));
385       INV_X1 g22308(.A (n_613), .ZN (n_614));
386       OR2_X1 g22266__6877(.A1 (n_611), .A2 (n_504), .ZN (n_612));
387       OAI21_X1 g22263__2900(.A (n_505), .B1 (n_610), .B2 (n_639), .ZN
388         (Z[1]));
389       OAI221_X1 g22316__2391(.A (n_490), .B1 (n_615), .B2 (n_593), .C1
390         (n_530), .C2 (n_592), .ZN (n_676));
391       OAI221_X1 g22287__7675(.A (n_239), .B1 (n_604), .B2 (n_639), .C1
392         (n_634), .C2 (n_238), .ZN (n_609));
393       AOI21_X1 g22269__7118(.A (n_669), .B1 (n_581), .B2 (n_503), .ZN
394         (n_608));
395       AOI21_X1 g22267__8757(.A (n_642), .B1 (n_597), .B2 (n_525), .ZN
396         (n_607));
397       OAI21_X1 g22273__1786(.A (n_543), .B1 (n_557), .B2 (n_523), .ZN
398         (n_606));
399       OAI221_X1 g22282__5953(.A (n_245), .B1 (n_604), .B2 (n_753), .C1
400         (n_634), .C2 (n_244), .ZN (n_605));
401       INV_X1 g22284(.A (n_602), .ZN (n_603));
402       INV_X1 g22270(.A (n_587), .ZN (n_640));
403       INV_X1 g22295(.A (n_578), .ZN (n_601));
```

```
404       OAI221_X1 g22288__5703(.A (n_237), .B1 (n_594), .B2 (n_599), .C1
405           (n_634), .C2 (n_236), .ZN (n_600));
406       OAI21_X1 g22290__7114(.A (n_597), .B1 (n_524), .B2 (n_542), .ZN
407           (n_598));
408       OAI221_X1 g22301__5266(.A (n_249), .B1 (n_590), .B2 (n_595), .C1
409           (n_634), .C2 (n_248), .ZN (n_596));
410       OAI221_X1 g22292__2250(.A (n_438), .B1 (n_594), .B2 (n_593), .C1
411           (n_512), .C2 (n_592), .ZN (n_602));
412       OAI221_X1 g22302__6083(.A (n_235), .B1 (n_590), .B2 (n_599), .C1
413           (n_634), .C2 (n_234), .ZN (n_591));
414       INV_X1 g22285(.A (n_588), .ZN (n_589));
415       AOI221_X1 g22274__2703(.A (n_437), .B1 (n_549), .B2 (n_558), .C1
416           (n_784), .C2 (n_1138), .ZN (n_587));
417       AOI21_X1 g22280__5795(.A (n_585), .B1 (n_544), .B2 (n_552), .ZN
418           (n_586));
419       OAI221_X1 g22281__7344(.A (n_247), .B1 (n_594), .B2 (n_595), .C1
420           (n_634), .C2 (n_246), .ZN (n_584));
421       OAI221_X1 g22315__1840(.A (n_424), .B1 (n_590), .B2 (n_593), .C1
422           (n_529), .C2 (n_592), .ZN (n_613));
423       OAI221_X1 g22289__5019(.A (n_255), .B1 (n_569), .B2 (n_599), .C1
424           (n_634), .C2 (n_254), .ZN (n_583));
425       INV_X1 g22309(.A (n_581), .ZN (n_582));
426       INV_X1 g22307(.A (n_579), .ZN (n_580));
427       OAI221_X1 g22298__1857(.A (n_259), .B1 (n_573), .B2 (n_595), .C1
428           (n_634), .C2 (n_258), .ZN (n_578));
429       OAI221_X1 g22299__9906(.A (n_205), .B1 (n_575), .B2 (n_595), .C1
430           (n_634), .C2 (n_204), .ZN (n_577));
431       OAI221_X1 g22304__8780(.A (n_241), .B1 (n_575), .B2 (n_599), .C1
432           (n_634), .C2 (n_240), .ZN (n_576));
433       OAI221_X1 g22305__4296(.A (n_203), .B1 (n_573), .B2 (n_599), .C1
434           (n_634), .C2 (n_202), .ZN (n_574));
435       INV_X1 g22306(.A (n_554), .ZN (n_572));
436       OAI21_X1 g22294__3772(.A (n_571), .B1 (n_540), .B2 (n_570), .ZN
437           (n_611));
438       OAI221_X1 g22293__1474(.A (n_3291), .B1 (n_569), .B2 (n_593), .C1
439           (n_3293), .C2 (n_592), .ZN (n_588));
440       AOI21_X1 g22310__4547(.A (n_541), .B1 (n_567), .B2 (n_566), .ZN
441           (n_568));
442       OAI221_X1 g22311__9682(.A (n_217), .B1 (n_491), .B2 (n_536), .C1
443           (n_634), .C2 (n_216), .ZN (n_565));
444       AOI21_X1 g22279__2683(.A (n_343), .B1 (n_551), .B2 (n_526), .ZN
445           (n_564));
446       AOI21_X1 g22277__1309(.A (n_545), .B1 (n_522), .B2 (n_562), .ZN
447           (n_563));
448       OAI221_X1 g22276__6877(.A (n_251), .B1 (n_569), .B2 (n_595), .C1
449           (n_634), .C2 (n_250), .ZN (n_561));
450       XNOR2_X1 g22246__2900(.A (n_516), .B (n_559), .ZN (n_560));
451       OAI221_X1 g22317__2391(.A (n_468), .B1 (n_573), .B2 (n_593), .C1
452           (n_513), .C2 (n_592), .ZN (n_621));
453       OAI21_X2 g22228__7675(.A (n_532), .B1 (n_559), .B2 (n_126), .ZN
454           (n_625));
455       AOI221_X1 g22341__7118(.A (n_514), .B1 (n_302), .B2 (n_1138), .C1
456           (n_485), .C2 (n_558), .ZN (n_630));
457       OAI221_X1 g22313__8757(.A (n_452), .B1 (n_575), .B2 (n_593), .C1
458           (n_472), .C2 (n_592), .ZN (n_691));
459       AOI211_X1 g22297__1786(.A (n_556), .B (n_434), .C1 (n_501), .C2
460           (n_555), .ZN (n_557));
461       AOI211_X1 g22312__5953(.A (n_553), .B (n_585), .C1 (n_552), .C2
```

```
462         (n_551), .ZN (n_554));
463       AOI222_X1 g22314__5703(.A1 (n_533), .A2 (n_558), .B1 (n_550), .B2
464         (n_531), .C1 (n_549), .C2 (n_474), .ZN (n_579));
465       AOI221_X1 g22238__7114(.A (n_293), .B1 (n_547), .B2 (n_769), .C1
466         (n_495), .C2 (n_825), .ZN (n_548));
467       AOI221_X1 g22318__5266(.A (n_460), .B1 (n_546), .B2 (n_362), .C1
468         (n_502), .C2 (n_301), .ZN (n_597));
469       AOI221_X1 g22283__2250(.A (n_477), .B1 (n_475), .B2 (n_558), .C1
470         (n_770), .C2 (n_1138), .ZN (n_610));
471       AOI21_X1 g22319__6083(.A (n_585), .B1 (n_545), .B2 (n_566), .ZN
472         (n_581));
473       OAI21_X1 g22328__2703(.A (n_543), .B1 (n_471), .B2 (n_542), .ZN
474         (n_544));
475       INV_X1 g22322(.A (n_540), .ZN (n_541));
476       INV_X1 g22321(.A (n_538), .ZN (n_539));
477       OAI221_X1 g22329__5795(.A (n_213), .B1 (n_534), .B2 (n_536), .C1
478         (n_634), .C2 (n_212), .ZN (n_537));
479       OAI221_X1 g22331__7344(.A (n_225), .B1 (n_534), .B2 (n_519), .C1
480         (n_634), .C2 (n_224), .ZN (n_535));
481       MUX2_X1 g22320__1840(.A (n_533), .B (n_550), .S (n_992), .Z (n_784));
482       NAND2_X1 g22247__5019(.A1 (n_515), .A2 (n_559), .ZN (n_532));
483       AOI221_X1 g22337__1857(.A (n_473), .B1 (n_486), .B2 (n_531), .C1
484         (n_487), .C2 (n_558), .ZN (n_604));
485       OAI222_X1 g22338__9906(.A1 (n_405), .A2 (n_593), .B1 (n_446), .B2
486         (n_592), .C1 (n_530), .C2 (n_528), .ZN (n_751));
487       OAI222_X1 g22339__8780(.A1 (n_529), .A2 (n_528), .B1 (n_507), .B2
488         (n_592), .C1 (n_508), .C2 (n_511), .ZN (n_722));
489       OAI221_X1 g22335__4296(.A (n_481), .B1 (n_3293), .B2 (n_528), .C1
490         (n_422), .C2 (n_592), .ZN (n_737));
491       OAI221_X1 g22364__3772(.A (n_497), .B1 (n_447), .B2 (n_992), .C1
492         (n_425), .C2 (n_506), .ZN (n_615));
493       OAI211_X1 g22323__1474(.A (n_454), .B (n_456), .C1 (n_500), .C2
494         (n_355), .ZN (n_526));
495       OR2_X1 g22324__4547(.A1 (n_524), .A2 (n_523), .ZN (n_525));
496       OAI21_X1 g22325__9682(.A (n_483), .B1 (n_288), .B2 (n_521), .ZN
497         (n_522));
498       OAI221_X1 g22326__2683(.A (n_263), .B1 (n_517), .B2 (n_519), .C1
499         (n_634), .C2 (n_262), .ZN (n_520));
500       OAI221_X1 g22330__1309(.A (n_215), .B1 (n_517), .B2 (n_536), .C1
501         (n_634), .C2 (n_214), .ZN (n_518));
502       OAI21_X1 g22332__6877(.A (n_653), .B1 (n_462), .B2 (n_789), .ZN
503         (n_538));
504       INV_X1 g22259(.A (n_515), .ZN (n_516));
505       NOR2_X1 g22373__2900(.A1 (n_513), .A2 (n_528), .ZN (n_514));
506       OAI222_X1 g22340__2391(.A1 (n_512), .A2 (n_528), .B1 (n_534), .B2
507         (n_511), .C1 (n_3296), .C2 (n_592), .ZN (n_718));
508       OAI221_X1 g22336__7675(.A (n_476), .B1 (n_517), .B2 (n_511), .C1
509         (n_3294), .C2 (n_592), .ZN (n_711));
510       AOI21_X1 g22333__7118(.A (n_585), .B1 (n_510), .B2 (n_566), .ZN
511         (n_540));
512       MUX2_X1 g22342__8757(.A (n_3296), .B (n_534), .S (n_992), .Z (n_594));
513       MUX2_X1 g22365__1786(.A (n_508), .B (n_507), .S (n_506), .Z (n_590));
514       AOI21_X1 g22296__5953(.A (n_401), .B1 (n_440), .B2 (n_825), .ZN
515         (n_505));
516       AOI211_X1 g22334__5703(.A (n_570), .B (n_498), .C1 (n_457), .C2
517         (n_418), .ZN (n_504));
518       NAND3_X1 g22327__7114(.A1 (n_482), .A2 (n_566), .A3 (n_562), .ZN
519         (n_503));
```

```
520       INV_X1 g22347(.A (n_524), .ZN (n_502));
521       NAND2_X1 g22348__5266(.A1 (n_500), .A2 (n_499), .ZN (n_501));
522       OR2_X1 g22352__2250(.A1 (n_510), .A2 (n_399), .ZN (n_545));
523       AOI21_X1 g22350__6083(.A (n_498), .B1 (n_470), .B2 (n_282), .ZN
524           (n_552));
525       OAI211_X1 g22345__2703(.A (n_442), .B (n_497), .C1 (n_421), .C2
526           (n_992), .ZN (n_569));
527       OAI22_X1 g22344__5795(.A1 (n_3294), .A2 (n_992), .B1 (n_517), .B2
528           (n_506), .ZN (n_770));
529       XOR2_X1 g22286__7344(.A (n_489), .B (n_488), .Z (n_495));
530       INV_X1 g22356(.A (n_484), .ZN (n_494));
531       INV_X1 g22357(.A (n_492), .ZN (n_493));
532       INV_X1 g22358(.A (n_550), .ZN (n_491));
533       AOI221_X1 g22370__1840(.A (n_420), .B1 (n_423), .B2 (n_466), .C1
534           (n_467), .C2 (n_465), .ZN (n_490));
535       OR2_X1 g22353__5019(.A1 (n_510), .A2 (n_431), .ZN (n_551));
536       OAI21_X2 g22264__1857(.A (n_208), .B1 (n_489), .B2 (n_488), .ZN
537           (n_515));
538       OAI22_X1 g22362__9906(.A1 (n_487), .A2 (n_992), .B1 (n_486), .B2
539           (n_506), .ZN (n_575));
540       OAI221_X1 g22363__8780(.A (n_497), .B1 (n_485), .B2 (n_992), .C1
541           (n_339), .C2 (n_506), .ZN (n_573));
542       OAI221_X1 g22359__4296(.A (n_227), .B1 (n_508), .B2 (n_519), .C1
543           (n_634), .C2 (n_226), .ZN (n_484));
544       INV_X1 g22346(.A (n_482), .ZN (n_483));
545       NAND2_X1 g22349__3772(.A1 (n_628), .A2 (n_1138), .ZN (n_481));
546       OAI221_X1 g22360__1474(.A (n_207), .B1 (n_508), .B2 (n_536), .C1
547           (n_634), .C2 (n_206), .ZN (n_480));
548       NAND2_X1 g22351__4547(.A1 (n_478), .A2 (n_479), .ZN (n_618));
549       OAI21_X1 g22361__9682(.A (n_478), .B1 (n_571), .B2 (n_342), .ZN
550           (n_492));
551       NAND3_X1 g22355__2683(.A1 (n_463), .A2 (n_546), .A3 (n_643), .ZN
552           (n_524));
553       OAI222_X1 g22375__1309(.A1 (n_419), .A2 (n_155), .B1 (n_333), .B2
554           (n_435), .C1 (n_174), .C2 (n_436), .ZN (n_477));
555       NAND2_X1 g22371__6877(.A1 (n_475), .A2 (n_474), .ZN (n_476));
556       NOR2_X1 g22372__2900(.A1 (n_472), .A2 (n_528), .ZN (n_473));
557       AND2_X1 g22374__2391(.A1 (n_470), .A2 (n_459), .ZN (n_471));
558       OAI21_X1 g22383__7675(.A (n_455), .B1 (n_453), .B2 (n_344), .ZN
559           (n_469));
560       AOI221_X1 g22417__7118(.A (n_402), .B1 (n_443), .B2 (n_449), .C1
561           (n_444), .C2 (n_450), .ZN (n_513));
562       INV_X1 g22420(.A (n_451), .ZN (n_533));
563       OAI222_X1 g22366__8757(.A1 (n_441), .A2 (n_338), .B1 (n_410), .B2
564           (n_411), .C1 (n_396), .C2 (n_409), .ZN (n_550));
565       AOI221_X1 g22397__1786(.A (n_384), .B1 (n_467), .B2 (n_466), .C1
566           (n_445), .C2 (n_465), .ZN (n_468));
567       INV_X1 g22367(.A (n_463), .ZN (n_464));
568       INV_X1 g22382(.A (n_461), .ZN (n_462));
569       INV_X1 g22369(.A (n_478), .ZN (n_460));
570       OR3_X1 g22391__5953(.A1 (n_352), .A2 (n_459), .A3 (n_458), .ZN
571           (n_500));
572       OAI21_X1 g22354__5703(.A (n_499), .B1 (n_361), .B2 (n_458), .ZN
573           (n_482));
574       INV_X1 g22368(.A (n_457), .ZN (n_567));
575       OAI21_X1 g22393__7114(.A (n_456), .B1 (n_398), .B2 (n_455), .ZN
576           (n_510));
577       OAI211_X1 g22381__5266(.A (n_357), .B (n_395), .C1 (n_386), .C2
```

```
578         (n_346), .ZN (n_534));
579     AND2_X1 g22402__2250(.A1 (n_453), .A2 (n_455), .ZN (n_454));
580     AOI221_X1 g22406__6083(.A (n_385), .B1 (n_408), .B2 (n_466), .C1
581         (n_415), .C2 (n_465), .ZN (n_452));
582     AOI221_X1 g22436__2703(.A (n_393), .B1 (n_366), .B2 (n_450), .C1
583         (n_406), .C2 (n_449), .ZN (n_451));
584     INV_X1 g22422(.A (n_446), .ZN (n_447));
585     AOI221_X1 g22410__7344(.A (n_383), .B1 (n_445), .B2 (n_450), .C1
586         (n_444), .C2 (n_449), .ZN (n_530));
587     AOI221_X1 g22433__1840(.A (n_369), .B1 (n_389), .B2 (n_450), .C1
588         (n_388), .C2 (n_449), .ZN (n_507));
589     INV_X1 g22423(.A (n_416), .ZN (n_549));
590     AOI221_X1 g22409__1857(.A (n_381), .B1 (n_443), .B2 (n_407), .C1
591         (n_444), .C2 (n_414), .ZN (n_529));
592     NAND2_X1 g22376__9906(.A1 (n_441), .A2 (n_992), .ZN (n_442));
593     XOR2_X1 g22388__8780(.A (n_429), .B (n_428), .Z (n_440));
594     OAI221_X1 g22387__4296(.A (n_277), .B1 (n_634), .B2 (n_275), .C1
595         (n_375), .C2 (n_595), .ZN (n_439));
596     AOI221_X1 g22385__3772(.A (n_305), .B1 (n_379), .B2 (n_466), .C1
597         (n_413), .C2 (n_465), .ZN (n_438));
598     OAI221_X1 g22384__1474(.A (n_380), .B1 (n_273), .B2 (n_436), .C1
599         (n_303), .C2 (n_435), .ZN (n_437));
600     OAI21_X1 g22390__4547(.A (n_427), .B1 (n_434), .B2 (n_326), .ZN
601         (n_542));
602     NAND2_X1 g22377__9682(.A1 (n_433), .A2 (n_432), .ZN (n_463));
603     NAND2_X1 g22378__2683(.A1 (n_431), .A2 (n_412), .ZN (n_457));
604     OAI21_X1 g22389__1309(.A (n_430), .B1 (n_479), .B2 (n_370), .ZN
605         (n_461));
606     AOI21_X2 g22343__6877(.A (n_188), .B1 (n_429), .B2 (n_428), .ZN
607         (n_489));
608     OAI21_X1 g22392__2900(.A (n_427), .B1 (n_434), .B2 (n_286), .ZN
609         (n_523));
610     NOR2_X1 g22380__2391(.A1 (n_441), .A2 (n_8), .ZN (n_628));
611     NAND2_X1 g22379__7675(.A1 (n_426), .A2 (n_666), .ZN (n_478));
612     AOI21_X1 g22394__7118(.A (n_382), .B1 (n_425), .B2 (n_1005), .ZN
613         (n_517));
614     AOI221_X1 g22428__8757(.A (n_334), .B1 (n_345), .B2 (n_466), .C1
615         (n_423), .C2 (n_465), .ZN (n_424));
616     INV_X1 g22421(.A (n_421), .ZN (n_422));
617     NOR3_X1 g22407__1786(.A1 (n_419), .A2 (n_528), .A3 (n_1005), .ZN
618         (n_420));
619     NAND3_X1 g22430__5953(.A1 (n_562), .A2 (n_372), .A3 (n_417), .ZN
620         (n_418));
621     AOI221_X1 g22443__5703(.A (n_316), .B1 (n_415), .B2 (n_414), .C1
622         (n_413), .C2 (n_450), .ZN (n_416));
623     NOR2_X1 g22415__7114(.A1 (n_434), .A2 (n_412), .ZN (n_470));
624     NAND2_X1 g22411__2250(.A1 (n_307), .A2 (n_374), .ZN (n_475));
625     OAI221_X1 g22412__6083(.A (n_387), .B1 (n_391), .B2 (n_411), .C1
626         (n_410), .C2 (n_409), .ZN (n_487));
627     AOI221_X1 g22434__2703(.A (n_317), .B1 (n_408), .B2 (n_450), .C1
628         (n_415), .C2 (n_449), .ZN (n_512));
629     AOI221_X1 g22442__7344(.A (n_309), .B1 (n_327), .B2 (n_414), .C1
630         (n_340), .C2 (n_407), .ZN (n_446));
631     AOI221_X1 g22413__1840(.A (n_314), .B1 (n_406), .B2 (n_414), .C1
632         (n_331), .C2 (n_407), .ZN (n_472));
633     INV_X1 g22396(.A (n_405), .ZN (n_547));
634     INV_X1 g22494(.A (n_390), .ZN (n_402));
635     OAI221_X1 g22386__5019(.A (n_229), .B1 (n_634), .B2 (n_228), .C1
```

```
636         (n_3297), .C2 (n_178), .ZN (n_401));
637     AND2_X1 g22398__1857(.A1 (n_257), .A2 (n_320), .ZN (n_400));
638     NOR3_X1 g22408__9906(.A1 (n_398), .A2 (n_371), .A3 (n_397), .ZN
639         (n_399));
640     OAI221_X1 g22438__8780(.A (n_297), .B1 (n_410), .B2 (n_392), .C1
641         (n_396), .C2 (n_395), .ZN (n_421));
642     NAND2_X1 g22419__4296(.A1 (n_425), .A2 (n_284), .ZN (n_405));
643     AND2_X1 g22416__3772(.A1 (n_306), .A2 (n_543), .ZN (n_456));
644     INV_X1 g22450(.A (n_412), .ZN (n_453));
645     OAI221_X1 g22414__1474(.A (n_325), .B1 (n_368), .B2 (n_411), .C1
646         (n_291), .C2 (n_395), .ZN (n_485));
647     OAI22_X1 g22523__4547(.A1 (n_199), .A2 (n_392), .B1 (n_391), .B2
648         (n_395), .ZN (n_393));
649     AOI22_X1 g22519__9682(.A1 (n_389), .A2 (n_414), .B1 (n_388), .B2
650         (n_407), .ZN (n_390));
651     AOI22_X1 g22518__2683(.A1 (n_200), .A2 (n_414), .B1 (n_386), .B2
652         (n_407), .ZN (n_387));
653     INV_X1 g22491(.A (n_324), .ZN (n_385));
654     INV_X1 g22492(.A (n_322), .ZN (n_384));
655     INV_X1 g22493(.A (n_319), .ZN (n_383));
656     INV_X1 g22496(.A (n_328), .ZN (n_382));
657     INV_X1 g22497(.A (n_329), .ZN (n_381));
658     AOI22_X1 g22513__1309(.A1 (n_198), .A2 (n_466), .B1 (n_379), .B2
659         (n_465), .ZN (n_380));
660     AOI22_X1 g22511__2391(.A1 (n_467), .A2 (n_449), .B1 (n_444), .B2
661         (n_407), .ZN (n_374));
662     OAI21_X1 g22512__7675(.A (n_290), .B1 (n_304), .B2 (n_436), .ZN
663         (n_373));
664     NOR2_X1 g22482__7118(.A1 (n_372), .A2 (n_755), .ZN (n_459));
665     NAND2_X1 g22489__8757(.A1 (n_499), .A2 (n_371), .ZN (n_412));
666     OR2_X1 g22485__1786(.A1 (n_370), .A2 (n_789), .ZN (n_642));
667     NOR2_X1 g22487__5953(.A1 (n_398), .A2 (n_298), .ZN (n_562));
668     OAI22_X1 g22524__5703(.A1 (n_368), .A2 (n_392), .B1 (n_201), .B2
669         (n_395), .ZN (n_369));
670     AOI22_X1 g22525__7114(.A1 (n_366), .A2 (n_414), .B1 (n_406), .B2
671         (n_407), .ZN (n_367));
672     AOI222_X1 g22460__2250(.A1 (n_177), .A2 (n_28), .B1 (n_276), .B2
673         (n_363), .C1 (n_268), .C2 (n_27), .ZN (n_364));
674     NAND2_X1 g22458__6083(.A1 (n_337), .A2 (n_278), .ZN (n_362));
675     AND2_X1 g22403__2703(.A1 (n_274), .A2 (n_360), .ZN (n_361));
676     INV_X1 g22449(.A (n_358), .ZN (n_359));
677     AOI22_X1 g22431__5795(.A1 (n_396), .A2 (n_450), .B1 (n_3297), .B2
678         (n_1005), .ZN (n_357));
679     AND4_X1 g22427__7344(.A1 (n_354), .A2 (n_360), .A3 (n_353), .A4
680         (n_184), .ZN (n_355));
681     AND3_X1 g22429__1840(.A1 (n_354), .A2 (n_360), .A3 (n_351), .ZN
682         (n_352));
683     MUX2_X1 g22437__5019(.A (n_211), .B (n_350), .S (n_672), .Z (n_430));
684     MUX2_X1 g22440__1857(.A (n_280), .B (n_349), .S (n_655), .Z (n_426));
685     AOI21_X1 g22441__9906(.A (n_348), .B1 (n_347), .B2 (n_795), .ZN
686         (n_427));
687     AOI22_X1 g22537__8780(.A1 (n_197), .A2 (n_346), .B1 (n_345), .B2
688         (n_1143), .ZN (n_419));
689     NOR2_X1 g22418__4296(.A1 (n_344), .A2 (n_398), .ZN (n_431));
690     INV_X1 g22424(.A (n_343), .ZN (n_433));
691     MUX2_X1 g22447__3772(.A (n_3297), .B (n_195), .S (n_346), .Z (n_441));
692     AOI211_X1 g22446__1474(.A (n_658), .B (n_342), .C1 (n_669), .C2
693         (n_341), .ZN (n_546));
```

```
694       OAI221_X1 g22448__4547(.A (n_289), .B1 (n_340), .B2 (n_411), .C1
695           (n_339), .C2 (n_338), .ZN (n_508));
696       OAI221_X2 g22395__9682(.A (n_337), .B1 (n_283), .B2 (n_336), .C1
697           (n_300), .C2 (n_335), .ZN (n_585));
698       OAI22_X1 g22516__2683(.A1 (n_333), .A2 (n_436), .B1 (n_196), .B2
699           (n_435), .ZN (n_334));
700       AOI22_X1 g22531__1309(.A1 (n_406), .A2 (n_450), .B1 (n_331), .B2
701           (n_449), .ZN (n_332));
702       AOI22_X1 g22529__2900(.A1 (n_467), .A2 (n_450), .B1 (n_445), .B2
703           (n_449), .ZN (n_329));
704       AOI22_X1 g22528__2391(.A1 (n_327), .A2 (n_450), .B1 (n_340), .B2
705           (n_449), .ZN (n_328));
706       INV_X1 g22501(.A (n_344), .ZN (n_326));
707       AOI22_X1 g22510__7675(.A1 (n_340), .A2 (n_414), .B1 (n_327), .B2
708           (n_449), .ZN (n_325));
709       AOI22_X1 g22514__7118(.A1 (n_379), .A2 (n_321), .B1 (n_413), .B2
710           (n_323), .ZN (n_324));
711       AOI22_X1 g22515__8757(.A1 (n_345), .A2 (n_321), .B1 (n_423), .B2
712           (n_323), .ZN (n_322));
713       OR3_X1 g22466__1786(.A1 (n_3297), .A2 (n_436), .A3 (n_639), .ZN
714           (n_320));
715       AOI22_X1 g22517__5953(.A1 (n_443), .A2 (n_414), .B1 (n_389), .B2
716           (n_407), .ZN (n_319));
717       OAI22_X1 g22520__5703(.A1 (n_308), .A2 (n_392), .B1 (n_368), .B2
718           (n_395), .ZN (n_318));
719       OAI22_X1 g22522__7114(.A1 (n_315), .A2 (n_392), .B1 (n_313), .B2
720           (n_395), .ZN (n_317));
721       OAI22_X1 g22526__5266(.A1 (n_191), .A2 (n_409), .B1 (n_315), .B2
722           (n_395), .ZN (n_316));
723       OAI22_X1 g22527__2250(.A1 (n_315), .A2 (n_411), .B1 (n_313), .B2
724           (n_409), .ZN (n_314));
725       NOR2_X1 g22484__6083(.A1 (n_3297), .A2 (n_411), .ZN (n_486));
726       NOR2_X1 g22483__2703(.A1 (n_570), .A2 (n_342), .ZN (n_358));
727       NAND2_X1 g22486__5795(.A1 (n_553), .A2 (n_312), .ZN (n_571));
728       NAND2_X1 g22488__7344(.A1 (n_455), .A2 (n_347), .ZN (n_434));
729       OAI21_X1 g22399__1840(.A (n_233), .B1 (n_634), .B2 (n_232), .ZN
730           (n_311));
731       OAI22_X1 g22534__1857(.A1 (n_308), .A2 (n_411), .B1 (n_368), .B2
732           (n_409), .ZN (n_309));
733       AOI22_X1 g22536__9906(.A1 (n_423), .A2 (n_450), .B1 (n_445), .B2
734           (n_414), .ZN (n_307));
735       OR2_X1 g22456__8780(.A1 (n_347), .A2 (n_348), .ZN (n_306));
736       OAI22_X1 g22432__4296(.A1 (n_304), .A2 (n_435), .B1 (n_303), .B2
737           (n_436), .ZN (n_305));
738       INV_X1 g22541(.A (n_375), .ZN (n_302));
739       NAND3_X1 g22426__3772(.A1 (n_543), .A2 (n_336), .A3 (n_300), .ZN
740           (n_301));
741       INV_X1 g22565(.A (n_298), .ZN (n_299));
742       AOI22_X1 g22532__1474(.A1 (n_331), .A2 (n_450), .B1 (n_193), .B2
743           (n_449), .ZN (n_297));
744       OAI21_X1 g22400__4547(.A (n_261), .B1 (n_634), .B2 (n_260), .ZN
745           (n_296));
746       OAI21_X1 g22401__9682(.A (n_222), .B1 (n_634), .B2 (n_221), .ZN
747           (n_295));
748       OAI21_X1 g22404__2683(.A (n_267), .B1 (n_634), .B2 (n_266), .ZN
749           (n_294));
750       OAI21_X1 g22405__1309(.A (n_231), .B1 (n_634), .B2 (n_230), .ZN
751           (n_293));
```

```
752       INV_X1 g22451(.A (n_566), .ZN (n_498));
753       OAI21_X1 g22445__6877(.A (n_300), .B1 (n_336), .B2 (n_292), .ZN
754           (n_343));
755       INV_X1 g22542(.A (n_371), .ZN (n_556));
756       OAI21_X2 g22538__2900(.A (n_220), .B1 (n_219), .B2 (n_80), .ZN
757           (n_429));
758       AOI22_X1 g22540__2391(.A1 (n_291), .A2 (n_346), .B1 (n_180), .B2
759           (n_1143), .ZN (n_425));
760       NAND2_X1 g22545__7675(.A1 (n_413), .A2 (n_466), .ZN (n_290));
761       NAND2_X1 g22547__7118(.A1 (n_291), .A2 (n_449), .ZN (n_289));
762       NAND3_X1 g22509__8757(.A1 (n_417), .A2 (n_755), .A3 (n_287), .ZN
763           (n_288));
764       INV_X1 g22543(.A (n_360), .ZN (n_372));
765       OAI21_X1 g22539__1786(.A (n_286), .B1 (n_555), .B2 (n_285), .ZN
766           (n_344));
767       NAND2_X1 g22562__5953(.A1 (n_176), .A2 (n_285), .ZN (n_371));
768       NAND2_X1 g22555__5703(.A1 (n_339), .A2 (n_284), .ZN (n_375));
769       NAND2_X1 g22556__7114(.A1 (n_806), .A2 (n_160), .ZN (n_681));
770       NOR2_X1 g22490__5266(.A1 (n_283), .A2 (n_800), .ZN (n_566));
771       AND2_X1 g22461__2250(.A1 (n_543), .A2 (n_458), .ZN (n_282));
772       INV_X1 g22566(.A (n_280), .ZN (n_281));
773       INV_X1 g22569(.A (n_342), .ZN (n_279));
774       INV_X1 g22570(.A (n_553), .ZN (n_278));
775       AOI221_X1 g22577__6083(.A (n_162), .B1 (n_276), .B2 (n_275), .C1
776           (n_270), .C2 (n_6), .ZN (n_277));
777       OR2_X1 g22455__2703(.A1 (n_354), .A2 (n_190), .ZN (n_274));
778       AOI221_X1 g22602__5795(.A (n_164), .B1 (n_173), .B2 (n_1151), .C1
779           (A[0]), .C2 (n_172), .ZN (n_273));
780       NAND2_X1 g22579__7344(.A1 (n_555), .A2 (n_286), .ZN (n_298));
781       INV_X1 g22568(.A (n_272), .ZN (n_370));
782       AOI222_X1 g22478__1840(.A1 (n_270), .A2 (n_16), .B1 (n_276), .B2
783           (n_269), .C1 (n_268), .C2 (n_15), .ZN (n_271));
784       AOI222_X1 g22464__5019(.A1 (n_270), .A2 (n_46), .B1 (n_276), .B2
785           (n_266), .C1 (n_268), .C2 (n_45), .ZN (n_267));
786       AOI222_X1 g22465__1857(.A1 (n_270), .A2 (n_68), .B1 (n_276), .B2
787           (n_264), .C1 (n_268), .C2 (n_67), .ZN (n_265));
788       AOI222_X1 g22467__9906(.A1 (n_270), .A2 (n_38), .B1 (n_276), .B2
789           (n_262), .C1 (n_268), .C2 (n_37), .ZN (n_263));
790       AOI222_X1 g22468__8780(.A1 (n_270), .A2 (n_36), .B1 (n_276), .B2
791           (n_260), .C1 (n_268), .C2 (n_35), .ZN (n_261));
792       AOI222_X1 g22469__4296(.A1 (n_276), .A2 (n_258), .B1 (n_268), .B2
793           (n_33), .C1 (n_270), .C2 (n_34), .ZN (n_259));
794       AOI222_X1 g22470__3772(.A1 (n_270), .A2 (n_48), .B1 (n_276), .B2
795           (n_256), .C1 (n_268), .C2 (n_47), .ZN (n_257));
796       AOI222_X1 g22471__1474(.A1 (n_276), .A2 (n_254), .B1 (n_268), .B2
797           (n_57), .C1 (n_270), .C2 (n_58), .ZN (n_255));
798       AOI222_X1 g22472__4547(.A1 (n_270), .A2 (n_30), .B1 (n_276), .B2
799           (n_252), .C1 (n_268), .C2 (n_29), .ZN (n_253));
800       AOI222_X1 g22473__9682(.A1 (n_270), .A2 (n_26), .B1 (n_276), .B2
801           (n_250), .C1 (n_268), .C2 (n_25), .ZN (n_251));
802       AOI222_X1 g22474__2683(.A1 (n_270), .A2 (n_24), .B1 (n_276), .B2
803           (n_248), .C1 (n_268), .C2 (n_23), .ZN (n_249));
804       AOI222_X1 g22475__1309(.A1 (n_270), .A2 (n_20), .B1 (n_276), .B2
805           (n_246), .C1 (n_268), .C2 (n_19), .ZN (n_247));
806       AOI222_X1 g22476__6877(.A1 (n_270), .A2 (n_18), .B1 (n_276), .B2
807           (n_244), .C1 (n_268), .C2 (n_17), .ZN (n_245));
808       AOI222_X1 g22477__2900(.A1 (n_270), .A2 (n_76), .B1 (n_276), .B2
809           (n_242), .C1 (n_268), .C2 (n_75), .ZN (n_243));
```

```
  AOI222_X1 g22463__2391(.A1 (n_270), .A2 (n_66), .B1 (n_276), .B2
      (n_240), .C1 (n_268), .C2 (n_65), .ZN (n_241));
  AOI222_X1 g22479__7675(.A1 (n_276), .A2 (n_238), .B1 (n_268), .B2
      (n_69), .C1 (n_270), .C2 (n_70), .ZN (n_239));
  AOI222_X1 g22480__7118(.A1 (n_270), .A2 (n_74), .B1 (n_276), .B2
      (n_236), .C1 (n_268), .C2 (n_73), .ZN (n_237));
  AOI222_X1 g22481__8757(.A1 (n_270), .A2 (n_72), .B1 (n_276), .B2
      (n_234), .C1 (n_268), .C2 (n_71), .ZN (n_235));
  AOI222_X1 g22503__1786(.A1 (n_270), .A2 (n_44), .B1 (n_276), .B2
      (n_232), .C1 (n_268), .C2 (n_43), .ZN (n_233));
  AOI222_X1 g22504__5953(.A1 (n_270), .A2 (n_52), .B1 (n_276), .B2
      (n_230), .C1 (n_268), .C2 (n_51), .ZN (n_231));
  AOI222_X1 g22505__5703(.A1 (n_270), .A2 (n_50), .B1 (n_276), .B2
      (n_228), .C1 (n_268), .C2 (n_49), .ZN (n_229));
  AOI222_X1 g22507__7114(.A1 (n_270), .A2 (n_42), .B1 (n_276), .B2
      (n_226), .C1 (n_268), .C2 (n_41), .ZN (n_227));
  AOI222_X1 g22508__5266(.A1 (n_270), .A2 (n_40), .B1 (n_276), .B2
      (n_224), .C1 (n_268), .C2 (n_39), .ZN (n_225));
  NAND2_X1 g22549__2250(.A1 (n_693), .A2 (n_166), .ZN (n_223));
  AOI222_X1 g22521__6083(.A1 (n_270), .A2 (n_54), .B1 (n_276), .B2
      (n_221), .C1 (n_268), .C2 (n_53), .ZN (n_222));
  NAND2_X1 g22546__2703(.A1 (n_219), .A2 (n_962), .ZN (n_220));
  NAND2_X1 g22548__5795(.A1 (n_713), .A2 (n_165), .ZN (n_218));
  INV_X1 g22596(.A (n_308), .ZN (n_388));
  NAND2_X1 g22558__7344(.A1 (n_397), .A2 (n_159), .ZN (n_455));
  AOI222_X1 g22459__1840(.A1 (n_270), .A2 (n_62), .B1 (n_276), .B2
      (n_216), .C1 (n_268), .C2 (n_61), .ZN (n_217));
  AOI222_X1 g22457__5019(.A1 (n_270), .A2 (n_60), .B1 (n_276), .B2
      (n_214), .C1 (n_268), .C2 (n_59), .ZN (n_215));
  AOI222_X1 g22454__1857(.A1 (n_270), .A2 (n_32), .B1 (n_276), .B2
      (n_212), .C1 (n_268), .C2 (n_31), .ZN (n_213));
  INV_X1 g22567(.A (n_210), .ZN (n_211));
  NAND2_X1 g22572__9906(.A1 (n_649), .A2 (n_168), .ZN (n_209));
  NAND2_X1 g22574__8780(.A1 (n_488), .A2 (n_167), .ZN (n_208));
  AOI222_X1 g22453__4296(.A1 (n_276), .A2 (n_206), .B1 (n_268), .B2
      (n_55), .C1 (n_270), .C2 (n_56), .ZN (n_207));
  AOI222_X1 g22452__3772(.A1 (n_276), .A2 (n_204), .B1 (n_268), .B2
      (n_21), .C1 (n_270), .C2 (n_22), .ZN (n_205));
  AOI222_X1 g22462__1474(.A1 (n_270), .A2 (n_64), .B1 (n_276), .B2
      (n_202), .C1 (n_268), .C2 (n_63), .ZN (n_203));
  INV_X1 g22601(.A (n_327), .ZN (n_201));
  INV_X1 g22600(.A (n_396), .ZN (n_200));
  INV_X1 g22599(.A (n_331), .ZN (n_199));
  INV_X1 g22592(.A (n_304), .ZN (n_198));
  INV_X1 g22591(.A (n_196), .ZN (n_197));
  INV_X1 g22593(.A (n_195), .ZN (n_386));
  NOR2_X1 g22580__4547(.A1 (n_312), .A2 (n_194), .ZN (n_280));
  INV_X1 g22597(.A (n_313), .ZN (n_366));
  NOR2_X1 g22584__9682(.A1 (n_672), .A2 (n_646), .ZN (n_272));
  INV_X1 g22598(.A (n_193), .ZN (n_391));
  NAND2_X1 g22583__2683(.A1 (n_789), .A2 (n_156), .ZN (n_653));
  INV_X1 g22594(.A (n_192), .ZN (n_415));
  INV_X1 g22595(.A (n_191), .ZN (n_408));
  MUX2_X1 g22618__1309(.A (n_1197), .B (n_1223), .S (n_98), .Z (n_444));
  NAND2_X1 g22586__6877(.A1 (n_666), .A2 (n_636), .ZN (n_342));
  NAND2_X1 g22560__2900(.A1 (n_795), .A2 (n_158), .ZN (n_347));
  NAND2_X1 g22588__2391(.A1 (n_458), .A2 (n_157), .ZN (n_499));
  NOR2_X1 g22587__7675(.A1 (n_170), .A2 (n_341), .ZN (n_553));
```

```
868       OR2_X1 g22559__7118(.A1 (n_348), .A2 (n_795), .ZN (n_398));
869       NAND2_X1 g22564__8757(.A1 (n_190), .A2 (n_119), .ZN (n_360));
870       HA_X1 g22544__5953(.A (n_136), .B (n_1151), .CO (n_833), .S (n_696));
871       NOR2_X1 g22550__5703(.A1 (n_708), .A2 (n_127), .ZN (n_189));
872       NOR2_X1 g22576__7114(.A1 (n_428), .A2 (n_125), .ZN (n_188));
873       NOR2_X1 g22575__5266(.A1 (n_624), .A2 (n_128), .ZN (n_187));
874       NOR2_X1 g22571__2250(.A1 (n_678), .A2 (n_129), .ZN (n_186));
875       OR2_X1 g22551__6083(.A1 (n_185), .A2 (n_184), .ZN (n_521));
876       AND2_X1 g22610__2703(.A1 (n_144), .A2 (n_148), .ZN (n_195));
877       OR2_X1 g22554__5795(.A1 (n_335), .A2 (n_292), .ZN (n_283));
878       NAND2_X1 g22553__7344(.A1 (n_335), .A2 (n_183), .ZN (n_337));
879       OR2_X1 g22552__1840(.A1 (n_800), .A2 (n_292), .ZN (n_432));
880       NOR2_X1 g22581__5019(.A1 (n_182), .A2 (n_181), .ZN (n_210));
881       AND2_X1 g22604__1857(.A1 (n_114), .A2 (n_92), .ZN (n_196));
882       AND2_X1 g22605__9906(.A1 (n_108), .A2 (n_154), .ZN (n_333));
883       AND2_X1 g22612__8780(.A1 (n_106), .A2 (n_93), .ZN (n_192));
884       NOR2_X1 g22609__4296(.A1 (n_180), .A2 (n_1143), .ZN (n_339));
885       NAND2_X1 g22608__3772(.A1 (n_111), .A2 (n_89), .ZN (n_345));
886       OR2_X1 g22582__1474(.A1 (n_666), .A2 (n_179), .ZN (n_479));
887       NAND2_X1 g22557__4547(.A1 (n_292), .A2 (n_121), .ZN (n_300));
888       AND2_X1 g22606__9682(.A1 (n_143), .A2 (n_133), .ZN (n_291));
889       AND2_X1 g22607__2683(.A1 (n_113), .A2 (n_102), .ZN (n_304));
890       OR2_X1 g22589__1309(.A1 (n_658), .A2 (n_669), .ZN (n_570));
891       NAND2_X1 g22585__6877(.A1 (n_800), .A2 (n_120), .ZN (n_336));
892       AND2_X2 g22611__2900(.A1 (n_109), .A2 (n_90), .ZN (n_368));
893       NAND2_X1 g22561__2391(.A1 (n_351), .A2 (n_123), .ZN (n_354));
894       NAND2_X1 g22613__7675(.A1 (n_146), .A2 (n_88), .ZN (n_445));
895       NAND2_X1 g22563__7118(.A1 (n_348), .A2 (n_122), .ZN (n_543));
896       NAND2_X1 g22712__8757(.A1 (n_321), .A2 (n_822), .ZN (n_178));
897       NAND2_X1 g22711__1786(.A1 (n_81), .A2 (n_788), .ZN (n_177));
898       INV_X1 g22639(.A (n_555), .ZN (n_176));
899       NAND2_X1 g22627__5953(.A1 (n_142), .A2 (n_84), .ZN (n_193));
900       AND2_X1 g22614__5703(.A1 (n_83), .A2 (n_87), .ZN (n_191));
901       AND2_X1 g22624__7114(.A1 (n_139), .A2 (n_99), .ZN (n_313));
902       INV_X1 g22634(.A (n_637), .ZN (n_806));
903       INV_X1 g22635(.A (n_397), .ZN (n_286));
904       INV_X1 g22641(.A (n_636), .ZN (n_655));
905       INV_X1 g22643(.A (n_458), .ZN (n_417));
906       AND2_X1 g22615__5266(.A1 (n_135), .A2 (n_134), .ZN (n_308));
907       OAI221_X1 g22620__2250(.A (n_91), .B1 (n_97), .B2 (n_118), .C1
908           (n_163), .C2 (n_4), .ZN (n_413));
909       AND2_X1 g22628__6083(.A1 (n_150), .A2 (n_141), .ZN (n_410));
910       NAND2_X1 g22629__2703(.A1 (n_147), .A2 (n_85), .ZN (n_331));
911       AND2_X1 g22630__5795(.A1 (n_138), .A2 (n_100), .ZN (n_396));
912       NAND2_X1 g22631__7344(.A1 (n_115), .A2 (n_107), .ZN (n_327));
913       NAND2_X1 g22622__1840(.A1 (n_104), .A2 (n_117), .ZN (n_379));
914       NAND2_X1 g22623__5019(.A1 (n_105), .A2 (n_94), .ZN (n_443));
915       AND2_X1 g22619__1857(.A1 (n_149), .A2 (n_116), .ZN (n_315));
916       NAND2_X1 g22621__9906(.A1 (n_145), .A2 (n_95), .ZN (n_389));
917       NAND2_X1 g22625__8780(.A1 (n_140), .A2 (n_86), .ZN (n_406));
918       NAND2_X1 g22617__4296(.A1 (n_103), .A2 (n_82), .ZN (n_423));
919       NAND2_X1 g22616__3772(.A1 (n_152), .A2 (n_101), .ZN (n_467));
920       NAND2_X1 g22626__1474(.A1 (n_110), .A2 (n_112), .ZN (n_340));
921       OR2_X2 g22735__4547(.A1 (n_96), .A2 (n_175), .ZN (n_634));
922       AOI222_X1 g22573__9682(.A1 (n_173), .A2 (n_1100), .B1 (n_169), .B2
923           (n_1456), .C1 (n_172), .C2 (n_997), .ZN (n_174));
924       INV_X1 g22638(.A (n_669), .ZN (n_170));
925       AOI222_X1 g22578__1309(.A1 (n_151), .A2 (n_1020), .B1 (n_169), .B2
```

```
926           (n_130), .C1 (n_153), .C2 (n_1130), .ZN (n_303));
927       XNOR2_X2 g22644__6877(.A (n_79), .B (n_11), .ZN (n_219));
928       INV_X1 g22637(.A (n_755), .ZN (n_190));
929       INV_X1 g22632(.A (n_185), .ZN (n_353));
930       INV_X1 g22640(.A (n_658), .ZN (n_312));
931       INV_X1 g22642(.A (n_335), .ZN (n_643));
932       XNOR2_X1 g22655__2900(.A (n_168), .B (n_1040), .ZN (n_649));
933       XNOR2_X1 g22646__2391(.A (n_167), .B (n_1184), .ZN (n_488));
934       XNOR2_X1 g22648__7675(.A (n_166), .B (n_1035), .ZN (n_693));
935       XNOR2_X1 g22651__7118(.A (n_165), .B (n_1025), .ZN (n_713));
936       INV_X1 g22636(.A (n_182), .ZN (n_646));
937       OAI22_X1 g22760__8757(.A1 (n_12), .A2 (n_1), .B1 (n_163), .B2 (n_3),
938           .ZN (n_164));
939       AND3_X1 g22713__1786(.A1 (n_268), .A2 (n_1151), .A3 (n_1241), .ZN
940           (n_162));
941       INV_X1 g22633(.A (n_351), .ZN (n_287));
942       INV_X1 g22766(.A (n_161), .ZN (n_519));
943       XOR2_X1 g22661__5953(.A (n_160), .B (n_1100), .Z (n_637));
944       XOR2_X1 g22672__5703(.A (n_349), .B (n_1110), .Z (n_636));
945       XNOR2_X1 g22663__7114(.A (n_159), .B (n_1085), .ZN (n_397));
946       XNOR2_X1 g22658__5266(.A (n_158), .B (n_1080), .ZN (n_795));
947       XOR2_X1 g22670__2250(.A (n_285), .B (n_1223), .Z (n_555));
948       INV_X1 g22769(.A (n_769), .ZN (n_599));
949       XNOR2_X1 g22674__6083(.A (n_157), .B (n_1197), .ZN (n_458));
950       XNOR2_X1 g22664__2703(.A (n_156), .B (n_1210), .ZN (n_789));
951       NOR2_X1 g22733__5795(.A1 (n_155), .A2 (n_1143), .ZN (n_466));
952       NOR2_X1 g22734__7344(.A1 (n_155), .A2 (n_346), .ZN (n_465));
953       AOI22_X1 g22754__1840(.A1 (n_153), .A2 (n_1105), .B1 (n_172), .B2
954           (n_1020), .ZN (n_154));
955       AOI22_X1 g22707__5019(.A1 (n_173), .A2 (n_1070), .B1 (n_151), .B2
956           (n_1095), .ZN (n_152));
957       AOI22_X1 g22717__1857(.A1 (n_173), .A2 (n_1035), .B1 (n_172), .B2
958           (n_1115), .ZN (n_150));
959       AOI22_X1 g22721__9906(.A1 (n_173), .A2 (n_1085), .B1 (n_151), .B2
960           (n_1197), .ZN (n_149));
961       AOI22_X1 g22722__8780(.A1 (n_153), .A2 (n_1184), .B1 (n_172), .B2
962           (n_1130), .ZN (n_148));
963       AOI22_X1 g22715__4296(.A1 (n_173), .A2 (n_1120), .B1 (n_172), .B2
964           (n_1070), .ZN (n_147));
965       AOI22_X1 g22706__3772(.A1 (n_173), .A2 (n_1080), .B1 (n_151), .B2
966           (n_1125), .ZN (n_146));
967       AOI22_X1 g22729__1474(.A1 (n_173), .A2 (n_1095), .B1 (n_151), .B2
968           (n_1070), .ZN (n_145));
969       AOI22_X1 g22720__4547(.A1 (n_173), .A2 (n_1020), .B1 (n_151), .B2
970           (n_1210), .ZN (n_144));
971       AOI22_X1 g22719__9682(.A1 (n_173), .A2 (n_1184), .B1 (n_151), .B2
972           (n_1100), .ZN (n_143));
973       AOI22_X1 g22718__2683(.A1 (n_173), .A2 (n_1015), .B1 (n_172), .B2
974           (n_1060), .ZN (n_142));
975       AOI22_X1 g22763__1309(.A1 (n_153), .A2 (n_1045), .B1 (n_151), .B2
976           (n_1110), .ZN (n_141));
977       AOI22_X1 g22716__6877(.A1 (n_173), .A2 (n_1090), .B1 (n_172), .B2
978           (n_1080), .ZN (n_140));
979       AOI22_X1 g22714__2900(.A1 (n_173), .A2 (n_1197), .B1 (n_172), .B2
980           (n_1223), .ZN (n_139));
981       AOI22_X1 g22710__2391(.A1 (n_173), .A2 (n_1040), .B1 (n_151), .B2
982           (n_1105), .ZN (n_138));
983       NOR2_X1 g22709__7675(.A1 (n_136), .A2 (n_1151), .ZN (n_137));
```

```
984       AOI22_X1 g22708__7118(.A1 (n_173), .A2 (n_1025), .B1 (n_151), .B2
985           (n_1060), .ZN (n_135));
986       AOI22_X1 g22747__8757(.A1 (n_153), .A2 (n_1015), .B1 (n_172), .B2
987           (n_1065), .ZN (n_134));
988       AOI22_X1 g22723__1786(.A1 (n_153), .A2 (n_997), .B1 (n_172), .B2
989           (n_1210), .ZN (n_133));
990       XOR2_X1 g22650__5953(.A (n_132), .B (n_1120), .Z (n_716));
991       NAND2_X1 g22732__5703(.A1 (n_131), .A2 (n_130), .ZN (n_180));
992       XNOR2_X1 g22654__7114(.A (n_129), .B (n_1045), .ZN (n_678));
993       XNOR2_X1 g22653__5266(.A (n_128), .B (n_1050), .ZN (n_624));
994       XNOR2_X1 g22652__2250(.A (n_127), .B (n_1015), .ZN (n_708));
995       XNOR2_X1 g22666__6083(.A (n_181), .B (n_1105), .ZN (n_182));
996       XNOR2_X1 g22649__2703(.A (n_126), .B (n_1020), .ZN (n_559));
997       XNOR2_X1 g22656__5795(.A (n_184), .B (n_1095), .ZN (n_185));
998       XNOR2_X2 g22647__7344(.A (n_125), .B (n_997), .ZN (n_428));
999       XNOR2_X2 g22645__1840(.A (n_124), .B (n_1030), .ZN (n_701));
1000      XNOR2_X1 g22659__5019(.A (n_123), .B (n_1090), .ZN (n_351));
1001      XOR2_X1 g22665__1857(.A (n_350), .B (n_1130), .Z (n_672));
1002      XNOR2_X1 g22668__9906(.A (n_179), .B (n_1055), .ZN (n_666));
1003      XNOR2_X1 g22660__8780(.A (n_122), .B (n_1075), .ZN (n_348));
1004      XOR2_X1 g22671__4296(.A (n_194), .B (n_1115), .Z (n_658));
1005      XNOR2_X1 g22673__3772(.A (n_183), .B (n_1060), .ZN (n_335));
1006      XNOR2_X1 g22657__1474(.A (n_121), .B (n_1065), .ZN (n_292));
1007      NOR2_X1 g22792__4547(.A1 (n_753), .A2 (n_1138), .ZN (n_769));
1008      XNOR2_X1 g22662__9682(.A (n_120), .B (n_1070), .ZN (n_800));
1009      XOR2_X1 g22667__2683(.A (n_119), .B (n_1125), .Z (n_755));
1010      XNOR2_X1 g22669__1309(.A (n_341), .B (n_118), .ZN (n_669));
1011      AOI22_X1 g22748__6877(.A1 (n_153), .A2 (n_1115), .B1 (n_172), .B2
1012          (n_1045), .ZN (n_117));
1013      AOI22_X1 g22724__2900(.A1 (n_153), .A2 (n_1223), .B1 (n_172), .B2
1014          (n_1125), .ZN (n_116));
1015      AOI22_X1 g22725__2391(.A1 (n_173), .A2 (n_1045), .B1 (n_151), .B2
1016          (n_1055), .ZN (n_115));
1017      AOI22_X1 g22726__7675(.A1 (n_173), .A2 (n_1055), .B1 (n_151), .B2
1018          (n_1045), .ZN (n_114));
1019      AOI22_X1 g22727__7118(.A1 (n_173), .A2 (n_1105), .B1 (n_151), .B2
1020          (n_1040), .ZN (n_113));
1021      AOI22_X1 g22744__8757(.A1 (n_153), .A2 (n_1020), .B1 (n_172), .B2
1022          (n_1105), .ZN (n_112));
1023      AOI22_X1 g22728__1786(.A1 (n_173), .A2 (n_1115), .B1 (n_151), .B2
1024          (n_1030), .ZN (n_111));
1025      AOI22_X1 g22730__5953(.A1 (n_173), .A2 (n_1050), .B1 (n_151), .B2
1026          (n_1130), .ZN (n_110));
1027      AOI22_X1 g22731__5703(.A1 (n_173), .A2 (n_1030), .B1 (n_151), .B2
1028          (n_1115), .ZN (n_109));
1029      AOI22_X1 g22737__7114(.A1 (n_173), .A2 (n_1130), .B1 (n_151), .B2
1030          (n_1050), .ZN (n_108));
1031      AOI22_X1 g22745__5266(.A1 (n_153), .A2 (n_1040), .B1 (n_172), .B2
1032          (n_1110), .ZN (n_107));
1033      AOI22_X1 g22738__2250(.A1 (n_173), .A2 (n_1075), .B1 (n_151), .B2
1034          (n_1090), .ZN (n_106));
1035      AOI22_X1 g22739__6083(.A1 (n_173), .A2 (n_1125), .B1 (n_151), .B2
1036          (n_1080), .ZN (n_105));
1037      AOI22_X1 g22740__2703(.A1 (n_173), .A2 (n_1110), .B1 (n_151), .B2
1038          (n_1035), .ZN (n_104));
1039      AOI22_X1 g22741__5795(.A1 (n_173), .A2 (n_1060), .B1 (n_151), .B2
1040          (n_1025), .ZN (n_103));
1041      AOI22_X1 g22742__7344(.A1 (n_153), .A2 (n_1055), .B1 (n_172), .B2
```

```
1042          (n_1050), .ZN (n_102));
1043        AOI22_X1 g22743__1840(.A1 (n_153), .A2 (n_1075), .B1 (n_172), .B2
1044          (n_1120), .ZN (n_101));
1045        AOI22_X1 g22746__5019(.A1 (n_153), .A2 (n_1050), .B1 (n_172), .B2
1046          (n_1055), .ZN (n_100));
1047        AOI22_X1 g22762__1857(.A1 (n_153), .A2 (n_1125), .B1 (n_151), .B2
1048          (n_1085), .ZN (n_99));
1049        NAND2_X1 g22777__9906(.A1 (n_97), .A2 (n_163), .ZN (n_98));
1050        INV_X1 g22773(.A (n_268), .ZN (n_96));
1051        AOI22_X1 g22749__8780(.A1 (n_153), .A2 (n_1120), .B1 (n_172), .B2
1052          (n_1075), .ZN (n_95));
1053        AOI22_X1 g22750__4296(.A1 (n_153), .A2 (n_1090), .B1 (n_172), .B2
1054          (n_1085), .ZN (n_94));
1055        AOI22_X1 g22752__3772(.A1 (n_153), .A2 (n_1080), .B1 (n_172), .B2
1056          (n_1095), .ZN (n_93));
1057        AOI22_X1 g22753__1474(.A1 (n_153), .A2 (n_1110), .B1 (n_172), .B2
1058          (n_1040), .ZN (n_92));
1059        AOI22_X1 g22755__4547(.A1 (n_153), .A2 (n_1060), .B1 (n_172), .B2
1060          (n_1030), .ZN (n_91));
1061        AOI22_X1 g22756__9682(.A1 (n_153), .A2 (n_1035), .B1 (n_172), .B2
1062          (n_1233), .ZN (n_90));
1063        AOI22_X1 g22757__2683(.A1 (n_153), .A2 (n_1233), .B1 (n_172), .B2
1064          (n_1035), .ZN (n_89));
1065        AOI22_X1 g22758__1309(.A1 (n_153), .A2 (n_1085), .B1 (n_172), .B2
1066          (n_1090), .ZN (n_88));
1067        AOI22_X1 g22759__6877(.A1 (n_153), .A2 (n_1070), .B1 (n_172), .B2
1068          (n_1025), .ZN (n_87));
1069        AOI22_X1 g22761__2900(.A1 (n_153), .A2 (n_1095), .B1 (n_151), .B2
1070          (n_1075), .ZN (n_86));
1071        AOI22_X1 g22764__2391(.A1 (n_153), .A2 (n_1025), .B1 (n_151), .B2
1072          (n_1065), .ZN (n_85));
1073        AOI22_X1 g22736__7675(.A1 (n_153), .A2 (n_1030), .B1 (n_151), .B2
1074          (n_1233), .ZN (n_84));
1075        AOI22_X1 g22765__7118(.A1 (n_173), .A2 (n_1065), .B1 (n_151), .B2
1076          (n_1120), .ZN (n_83));
1077        AOI22_X1 g22751__8757(.A1 (n_153), .A2 (n_1065), .B1 (n_172), .B2
1078          (n_1015), .ZN (n_82));
1079        INV_X1 g22772(.A (n_270), .ZN (n_81));
1080        NOR2_X1 g22786__1786(.A1 (n_528), .A2 (n_639), .ZN (n_161));
1081        INV_X1 g22767(.A (n_435), .ZN (n_323));
1082        INV_X1 g22768(.A (n_436), .ZN (n_321));
1083        INV_X1 g22770(.A (n_595), .ZN (n_815));
1084        INV_X1 g22798(.A (n_79), .ZN (n_80));
1085        NAND2_X1 g22790__5953(.A1 (n_449), .A2 (n_474), .ZN (n_435));
1086        INV_X1 g22771(.A (n_825), .ZN (n_788));
1087        AND3_X2 g22795__5703(.A1 (n_78), .A2 (n_962), .A3 (n_1411), .ZN
1088          (n_276));
1089        AND2_X2 g22797__7114(.A1 (n_78), .A2 (n_14), .ZN (n_268));
1090        NAND2_X1 g22783__2250(.A1 (n_474), .A2 (n_1005), .ZN (n_155));
1091        NAND2_X1 g22788__6083(.A1 (n_474), .A2 (n_822), .ZN (n_536));
1092        NAND2_X1 g22791__2703(.A1 (n_450), .A2 (n_474), .ZN (n_436));
1093        NAND2_X1 g22793__5795(.A1 (n_736), .A2 (n_593), .ZN (n_595));
1094        AND3_X2 g22796__7344(.A1 (n_78), .A2 (n_175), .A3 (n_962), .ZN
1095          (n_270));
1096        HA_X1 g22704__1840(.A (n_1311), .B (n_1223), .CO (n_75), .S (n_76));
1097        HA_X1 g22703__5019(.A (n_1351), .B (n_1090), .CO (n_73), .S (n_74));
1098        HA_X1 g22702__1857(.A (n_1331), .B (n_1095), .CO (n_71), .S (n_72));
1099        HA_X1 g22701__9906(.A (n_1376), .B (n_1125), .CO (n_69), .S (n_70));
```

```
1100      HA_X1 g22700__8780(.A (n_1271), .B (n_1025), .CO (n_67), .S (n_68));
1101      HA_X1 g22699__4296(.A (n_1386), .B (n_1015), .CO (n_65), .S (n_66));
1102      HA_X1 g22698__3772(.A (n_1381), .B (n_1030), .CO (n_63), .S (n_64));
1103      HA_X1 g22697__1474(.A (n_1276), .B (n_1035), .CO (n_61), .S (n_62));
1104      HA_X1 g22696__4547(.A (n_1346), .B (n_1045), .CO (n_59), .S (n_60));
1105      HA_X1 g22695__9682(.A (n_1356), .B (n_1120), .CO (n_57), .S (n_58));
1106      HA_X1 g22694__2683(.A (n_1401), .B (n_1050), .CO (n_55), .S (n_56));
1107      HA_X1 g22693__1309(.A (n_1366), .B (n_1020), .CO (n_53), .S (n_54));
1108      HA_X1 g22692__6877(.A (n_1286), .B (n_1184), .CO (n_51), .S (n_52));
1109      HA_X1 g22691__2900(.A (n_1256), .B (n_997), .CO (n_49), .S (n_50));
1110      HA_X1 g22690__2391(.A (n_1301), .B (n_1100), .CO (n_47), .S (n_48));
1111      HA_X1 g22689__7675(.A (n_1291), .B (n_1210), .CO (n_45), .S (n_46));
1112      HA_X1 g22688__7118(.A (n_1130), .B (n_1326), .CO (n_43), .S (n_44));
1113      HA_X1 g22687__8757(.A (n_1105), .B (n_1321), .CO (n_41), .S (n_42));
1114      HA_X1 g22686__1786(.A (n_1371), .B (n_1055), .CO (n_39), .S (n_40));
1115      HA_X1 g22685__5953(.A (n_1266), .B (n_1110), .CO (n_37), .S (n_38));
1116      HA_X1 g22684__5703(.A (n_1341), .B (n_1115), .CO (n_35), .S (n_36));
1117      HA_X1 g22683__7114(.A (n_1336), .B (n_1233), .CO (n_33), .S (n_34));
1118      HA_X1 g22682__5266(.A (n_1281), .B (n_1040), .CO (n_31), .S (n_32));
1119      HA_X1 g22681__2250(.A (n_1406), .B (n_1065), .CO (n_29), .S (n_30));
1120      HA_X1 g22680__6083(.A (n_1177), .B (n_1261), .CO (n_27), .S (n_28));
1121      HA_X1 g22679__2703(.A (n_1396), .B (n_1070), .CO (n_25), .S (n_26));
1122      HA_X1 g22678__5795(.A (n_1391), .B (n_1075), .CO (n_23), .S (n_24));
1123      HA_X1 g22677__7344(.A (n_1361), .B (n_1060), .CO (n_21), .S (n_22));
1124      HA_X1 g22676__1840(.A (n_1316), .B (n_1080), .CO (n_19), .S (n_20));
1125      HA_X1 g22675__5019(.A (n_1306), .B (n_1085), .CO (n_17), .S (n_18));
1126      HA_X1 g22705__1857(.A (n_1296), .B (n_1197), .CO (n_15), .S (n_16));
1127      XNOR2_X1 g22803__9906(.A (n_14), .B (n_1286), .ZN (n_167));
1128      XNOR2_X1 g22806__8780(.A (n_14), .B (n_1291), .ZN (n_156));
1129      INV_X1 g22860(.A (n_173), .ZN (n_97));
1130      XNOR2_X1 g22802__4296(.A (n_14), .B (n_1271), .ZN (n_165));
1131      XNOR2_X1 g22801__3772(.A (n_14), .B (n_1296), .ZN (n_157));
1132      XNOR2_X1 g22805__1474(.A (n_14), .B (n_1306), .ZN (n_159));
1133      XNOR2_X1 g22810__4547(.A (n_14), .B (n_1276), .ZN (n_166));
1134      XNOR2_X1 g22819__9682(.A (n_14), .B (n_1311), .ZN (n_285));
1135      XNOR2_X1 g22818__2683(.A (n_14), .B (n_1266), .ZN (n_349));
1136      INV_X1 g22829(.A (n_822), .ZN (n_753));
1137      INV_X1 g22830(.A (n_449), .ZN (n_409));
1138      AND3_X2 g22794__1309(.A1 (n_2), .A2 (n_13), .A3 (n_1251), .ZN
1139          (n_825));
1140      INV_X1 g22831(.A (n_153), .ZN (n_12));
1141      XNOR2_X1 g22811__6877(.A (n_14), .B (n_1281), .ZN (n_168));
1142      OAI22_X1 g22778__2900(.A1 (n_11), .A2 (n_962), .B1 (n_14), .B2 (n_9),
1143          .ZN (n_131));
1144      XNOR2_X1 g22799__2391(.A (n_14), .B (n_1301), .ZN (n_160));
1145      XNOR2_X1 g22812__7675(.A (n_14), .B (n_1316), .ZN (n_158));
1146      XNOR2_X1 g22817__7118(.A (n_14), .B (n_1241), .ZN (n_136));
1147      OAI22_X2 g22800__8757(.A1 (op_ctl[1]), .A2 (n_0), .B1 (n_14), .B2
1148          (B[0]), .ZN (n_79));
1149      INV_X1 g22824(.A (n_531), .ZN (n_511));
1150      INV_X1 g22828(.A (n_736), .ZN (n_639));
1151      INV_X1 g22857(.A (n_474), .ZN (n_528));
1152      INV_X1 g22858(.A (n_450), .ZN (n_411));
1153      INV_X1 g22826(.A (n_392), .ZN (n_414));
1154      INV_X1 g22825(.A (n_558), .ZN (n_592));
1155      NAND2_X1 g22833__1786(.A1 (n_9), .A2 (n_1241), .ZN (n_10));
1156      XNOR2_X1 g22808__5953(.A (n_1366), .B (n_962), .ZN (n_126));
1157      XNOR2_X1 g22807__5703(.A (n_1346), .B (n_962), .ZN (n_129));
```

```
1158        MUX2_X1 g22776__7114(.A (n_1210), .B (n_1184), .S (n_962), .Z
1159            (n_169));
1160        XNOR2_X1 g22809__5266(.A (n_1401), .B (n_962), .ZN (n_128));
1161        XOR2_X1 g22781__2250(.A (n_1351), .B (n_962), .Z (n_123));
1162        XOR2_X1 g22785__6083(.A (n_1376), .B (n_962), .Z (n_119));
1163        XOR2_X1 g22784__2703(.A (n_1396), .B (n_962), .Z (n_120));
1164        XOR2_X1 g22779__5795(.A (n_1356), .B (n_962), .Z (n_132));
1165        XNOR2_X1 g22787__7344(.A (n_1326), .B (n_962), .ZN (n_350));
1166        XOR2_X1 g22789__1840(.A (n_1361), .B (n_962), .Z (n_183));
1167        XNOR2_X1 g22816__5019(.A (n_1321), .B (n_962), .ZN (n_181));
1168        XNOR2_X1 g22820__1857(.A (n_1341), .B (n_962), .ZN (n_194));
1169        NOR2_X1 g22881__9906(.A1 (n_13), .A2 (n_5), .ZN (n_78));
1170        NOR2_X1 g22849__8780(.A1 (n_506), .A2 (n_1138), .ZN (n_558));
1171        NAND2_X1 g22850__4296(.A1 (n_346), .A2 (n_1005), .ZN (n_392));
1172        NOR2_X2 g22852__3772(.A1 (n_14), .A2 (n_1251), .ZN (n_736));
1173        AND2_X1 g22883__1474(.A1 (n_338), .A2 (n_346), .ZN (n_450));
1174        AND2_X2 g22885__4547(.A1 (n_14), .A2 (n_130), .ZN (n_173));
1175        INV_X1 g22823(.A (n_284), .ZN (n_8));
1176        NOR2_X1 g22844__9682(.A1 (n_9), .A2 (n_1241), .ZN (n_7));
1177        XOR2_X1 g22775__2683(.A (n_1151), .B (n_1241), .Z (n_6));
1178        XNOR2_X2 g22813__1309(.A (n_1861), .B (op_ctl[1]), .ZN (n_125));
1179        XNOR2_X1 g22804__6877(.A (n_1386), .B (n_962), .ZN (n_127));
1180        XOR2_X1 g22780__2900(.A (n_1406), .B (n_962), .Z (n_121));
1181        XOR2_X1 g22782__2391(.A (n_1391), .B (n_962), .Z (n_122));
1182        XNOR2_X1 g22814__7675(.A (n_1381), .B (n_962), .ZN (n_124));
1183        XNOR2_X1 g22822__7118(.A (n_1331), .B (n_962), .ZN (n_184));
1184        NOR2_X1 g22848__8757(.A1 (n_593), .A2 (n_992), .ZN (n_531));
1185        INV_X1 g22859(.A (n_151), .ZN (n_163));
1186        XNOR2_X1 g22821__1786(.A (n_1371), .B (n_962), .ZN (n_179));
1187        XNOR2_X1 g22815__5953(.A (n_1336), .B (n_962), .ZN (n_341));
1188        AND2_X1 g22882__5703(.A1 (n_593), .A2 (n_506), .ZN (n_474));
1189        INV_X1 g22827(.A (n_395), .ZN (n_407));
1190        AND2_X1 g22853__7114(.A1 (n_14), .A2 (n_5), .ZN (n_822));
1191        AND2_X1 g22854__5266(.A1 (n_338), .A2 (n_1143), .ZN (n_449));
1192        NOR2_X4 g22855__2250(.A1 (n_130), .A2 (n_962), .ZN (n_153));
1193        NOR2_X4 g22856__6083(.A1 (n_14), .A2 (n_1456), .ZN (n_172));
1194        NOR2_X1 g22875__2703(.A1 (n_1060), .A2 (n_1361), .ZN (n_204));
1195        NOR2_X1 g22878__5795(.A1 (n_1197), .A2 (n_1296), .ZN (n_269));
1196        NOR2_X1 g22832__7344(.A1 (n_1045), .A2 (n_1346), .ZN (n_214));
1197        NOR2_X1 g22864__1840(.A1 (n_1110), .A2 (n_1266), .ZN (n_262));
1198        NOR2_X1 g22838__5019(.A1 (n_1080), .A2 (n_1316), .ZN (n_246));
1199        NOR2_X1 g22846__1857(.A1 (n_1210), .A2 (n_1291), .ZN (n_266));
1200        NOR2_X1 g22865__9906(.A1 (n_1115), .A2 (n_1341), .ZN (n_260));
1201        NOR2_X1 g22874__8780(.A1 (n_1105), .A2 (n_1321), .ZN (n_226));
1202        NOR2_X1 g22868__4296(.A1 (n_1233), .A2 (n_1336), .ZN (n_258));
1203        NOR2_X1 g22840__3772(.A1 (n_1030), .A2 (n_1381), .ZN (n_202));
1204        NOR2_X1 g22877__1474(.A1 (n_1223), .A2 (n_1311), .ZN (n_242));
1205        NOR2_X1 g22862__4547(.A1 (n_1035), .A2 (n_1276), .ZN (n_216));
1206        NOR2_X1 g22843__9682(.A1 (n_1075), .A2 (n_1391), .ZN (n_248));
1207        NOR2_X1 g22839__2683(.A1 (n_1151), .A2 (n_1241), .ZN (n_275));
1208        NOR2_X1 g22873__1309(.A1 (n_1261), .A2 (n_1177), .ZN (n_363));
1209        NAND2_X1 g22880__6877(.A1 (n_992), .A2 (n_1005), .ZN (n_497));
1210        NOR2_X1 g22847__2900(.A1 (n_992), .A2 (n_1005), .ZN (n_284));
1211        NAND2_X1 g22851__2391(.A1 (n_1005), .A2 (n_1143), .ZN (n_395));
1212        NOR2_X1 g22876__7675(.A1 (n_1055), .A2 (n_1371), .ZN (n_224));
1213        NOR2_X1 g22879__7118(.A1 (n_1120), .A2 (n_1356), .ZN (n_254));
1214        NOR2_X1 g22842__8757(.A1 (n_1184), .A2 (n_1286), .ZN (n_230));
1215        NOR2_X1 g22869__1786(.A1 (n_1025), .A2 (n_1271), .ZN (n_264));
```

```
1216      NOR2_X1 g22872__5953(.A1 (n_1125), .A2 (n_1376), .ZN (n_238));
1217      NOR2_X1 g22866__5703(.A1 (n_1085), .A2 (n_1306), .ZN (n_244));
1218      NOR2_X1 g22835__7114(.A1 (n_1050), .A2 (n_1401), .ZN (n_206));
1219      NOR2_X1 g22834__5266(.A1 (n_1020), .A2 (n_1366), .ZN (n_221));
1220      NOR2_X1 g22863__2250(.A1 (n_1065), .A2 (n_1406), .ZN (n_252));
1221      NOR2_X1 g22871__6083(.A1 (n_1090), .A2 (n_1351), .ZN (n_236));
1222      NOR2_X1 g22841__2703(.A1 (n_1100), .A2 (n_1301), .ZN (n_256));
1223      NOR2_X1 g22845__5795(.A1 (n_997), .A2 (n_1256), .ZN (n_228));
1224      NOR2_X1 g22837__7344(.A1 (n_1130), .A2 (n_1326), .ZN (n_232));
1225      NOR2_X1 g22870__1840(.A1 (n_1015), .A2 (n_1386), .ZN (n_240));
1226      NOR2_X1 g22836__5019(.A1 (n_1070), .A2 (n_1396), .ZN (n_250));
1227      NOR2_X1 g22867__1857(.A1 (n_1040), .A2 (n_1281), .ZN (n_212));
1228      NOR2_X1 g22861__9906(.A1 (n_1095), .A2 (n_1331), .ZN (n_234));
1229      AND2_X2 g22884__8780(.A1 (n_1010), .A2 (n_1424), .ZN (n_151));
1230      INV_X1 g22887(.A (n_1419), .ZN (n_13));
1231      INV_X2 g22889(.A (n_1010), .ZN (n_130));
1232      INV_X1 g22896(.A (B[0]), .ZN (n_0));
1233      INV_X1 g22897(.A (n_1261), .ZN (n_11));
1234      INV_X16 g22902(.A (op_ctl[1]), .ZN (n_14));
1235      NOR2_X1 g2(.A1 (n_564), .A2 (n_464), .ZN (n_916));
1236      INV_X1 g4(.A (n_918), .ZN (n_919));
1237      MUX2_X1 g22903(.A (n_917), .B (n_640), .S (n_962), .Z (n_918));
1238      NOR2_X1 g3(.A1 (n_375), .A2 (n_1138), .ZN (n_917));
1239      INV_X4 fopt22958(.A (n_14), .ZN (n_962));
1240      INV_X1 drc_bufs23099(.A (n_1151), .ZN (n_9));
1241      CLKBUF_X1 fopt23118(.A (B[0]), .Z (n_1177));
1242      CLKBUF_X1 drc_buf_sp(.A (op_ctl[2]), .Z (n_1419));
1243      CLKBUF_X1 drc_buf_sp23313(.A (op_ctl[1]), .Z (n_1424));
1244      INV_X1 drc_bufs23331(.A (n_997), .ZN (n_3));
1245      BUF_X2 drc_bufs23344(.A (op_ctl[6]), .Z (n_1010));
1246      INV_X1 drc_bufs23366(.A (n_1015), .ZN (n_4));
1247      INV_X1 drc_bufs23542(.A (n_1100), .ZN (n_1));
1248      INV_X1 drc_bufs23627(.A (n_1143), .ZN (n_346));
1249      INV_X1 drc_bufs23638(.A (n_1138), .ZN (n_593));
1250      INV_X1 drc_bufs23649(.A (n_1233), .ZN (n_118));
1251      INV_X1 drc_bufs23676(.A (n_1251), .ZN (n_5));
1252      BUF_X2 drc_bufs23688(.A (B[1]), .Z (n_1861));
1253      CLKBUF_X2 fopt23709(.A (A[0]), .Z (n_1261));
1254      INV_X1 drc_bufs23749(.A (n_1934), .ZN (n_1286));
1255      INV_X1 drc_bufs23750(.A (B[2]), .ZN (n_1934));
1256      INV_X1 drc_bufs24036(.A (n_1411), .ZN (n_175));
1257      INV_X1 drc_bufs24047(.A (n_1416), .ZN (n_2));
1258      CLKBUF_X1 drc_buf_sp24051(.A (B[31]), .Z (n_1241));
1259      CLKBUF_X1 drc_buf_sp24054(.A (A[16]), .Z (n_1223));
1260      CLKBUF_X1 drc_buf_sp24057(.A (A[29]), .Z (n_1210));
1261      CLKBUF_X1 drc_buf_sp24060(.A (A[15]), .Z (n_1197));
1262      CLKBUF_X1 drc_buf_sp24063(.A (A[2]), .Z (n_1184));
1263      CLKBUF_X1 drc_buf_sp24066(.A (A[31]), .Z (n_1151));
1264      INV_X1 drc_bufs24106(.A (n_1005), .ZN (n_338));
1265      CLKBUF_X1 drc_buf_sp24175(.A (A[6]), .Z (n_1045));
1266      CLKBUF_X1 drc_buf_sp24178(.A (A[7]), .Z (n_1035));
1267      CLKBUF_X1 drc_buf_sp24181(.A (A[8]), .Z (n_1030));
1268      CLKBUF_X1 drc_buf_sp24184(.A (A[9]), .Z (n_1015));
1269      CLKBUF_X1 drc_buf_sp24187(.A (A[5]), .Z (n_1040));
1270      CLKBUF_X1 drc_buf_sp24190(.A (A[11]), .Z (n_1120));
1271      CLKBUF_X1 drc_buf_sp24193(.A (A[12]), .Z (n_1095));
1272      CLKBUF_X1 drc_buf_sp24196(.A (A[13]), .Z (n_1090));
1273      CLKBUF_X1 drc_buf_sp24199(.A (A[14]), .Z (n_1125));
```

```
1274      CLKBUF_X1 drc_buf_sp24202(.A (op_ctl[3]), .Z (n_1416));
1275      CLKBUF_X1 drc_buf_sp24205(.A (B[26]), .Z (n_1371));
1276      CLKBUF_X1 drc_buf_sp24208(.A (A[17]), .Z (n_1085));
1277      CLKBUF_X1 drc_buf_sp24211(.A (A[18]), .Z (n_1080));
1278      CLKBUF_X1 drc_buf_sp24214(.A (A[19]), .Z (n_1075));
1279      CLKBUF_X1 drc_buf_sp24217(.A (A[20]), .Z (n_1070));
1280      CLKBUF_X1 drc_buf_sp24220(.A (A[21]), .Z (n_1065));
1281      CLKBUF_X1 drc_buf_sp24223(.A (A[22]), .Z (n_1060));
1282      CLKBUF_X1 drc_buf_sp24226(.A (A[23]), .Z (n_1233));
1283      CLKBUF_X1 drc_buf_sp24229(.A (A[24]), .Z (n_1115));
1284      CLKBUF_X1 drc_buf_sp24232(.A (A[25]), .Z (n_1110));
1285      CLKBUF_X1 drc_buf_sp24235(.A (B[10]), .Z (n_1271));
1286      CLKBUF_X1 drc_buf_sp24238(.A (A[27]), .Z (n_1105));
1287      CLKBUF_X1 drc_buf_sp24241(.A (A[28]), .Z (n_1130));
1288      CLKBUF_X1 drc_buf_sp24244(.A (A[10]), .Z (n_1025));
1289      CLKBUF_X1 drc_buf_sp24247(.A (B[9]), .Z (n_1386));
1290      CLKBUF_X1 drc_buf_sp24250(.A (B[25]), .Z (n_1266));
1291      CLKBUF_X1 drc_buf_sp24253(.A (B[21]), .Z (n_1406));
1292      CLKBUF_X1 drc_buf_sp24256(.A (B[22]), .Z (n_1361));
1293      CLKBUF_X1 drc_buf_sp24265(.A (B[3]), .Z (n_1366));
1294      CLKBUF_X1 drc_buf_sp24268(.A (B[4]), .Z (n_1401));
1295      CLKBUF_X1 drc_buf_sp24271(.A (B[24]), .Z (n_1341));
1296      CLKBUF_X1 drc_buf_sp24274(.A (B[6]), .Z (n_1346));
1297      CLKBUF_X1 drc_buf_sp24277(.A (B[8]), .Z (n_1381));
1298      CLKBUF_X1 drc_buf_sp24280(.A (B[7]), .Z (n_1276));
1299      CLKBUF_X1 drc_buf_sp24283(.A (B[5]), .Z (n_1281));
1300      CLKBUF_X1 drc_buf_sp24286(.A (op_ctl[0]), .Z (n_1411));
1301      CLKBUF_X1 drc_buf_sp24289(.A (B[11]), .Z (n_1356));
1302      CLKBUF_X1 drc_buf_sp24292(.A (B[12]), .Z (n_1331));
1303      CLKBUF_X1 drc_buf_sp24295(.A (B[13]), .Z (n_1351));
1304      CLKBUF_X1 drc_buf_sp24298(.A (B[14]), .Z (n_1376));
1305      CLKBUF_X1 drc_buf_sp24301(.A (B[15]), .Z (n_1296));
1306      CLKBUF_X1 drc_buf_sp24304(.A (B[16]), .Z (n_1311));
1307      CLKBUF_X1 drc_buf_sp24307(.A (B[17]), .Z (n_1306));
1308      CLKBUF_X1 drc_buf_sp24310(.A (A[26]), .Z (n_1055));
1309      CLKBUF_X1 drc_buf_sp24313(.A (B[19]), .Z (n_1391));
1310      CLKBUF_X1 drc_buf_sp24316(.A (A[30]), .Z (n_1100));
1311      CLKBUF_X1 drc_buf_sp24319(.A (op_ctl[5]), .Z (n_1251));
1312      CLKBUF_X1 drc_buf_sp24322(.A (op_ctl[10]), .Z (n_1138));
1313      CLKBUF_X1 drc_buf_sp24325(.A (B[23]), .Z (n_1336));
1314      CLKBUF_X1 drc_buf_sp24328(.A (A[4]), .Z (n_1050));
1315      CLKBUF_X2 drc_buf_sp24331(.A (A[1]), .Z (n_997));
1316      CLKBUF_X1 drc_buf_sp24334(.A (B[20]), .Z (n_1396));
1317      CLKBUF_X1 drc_buf_sp24337(.A (B[27]), .Z (n_1321));
1318      CLKBUF_X1 drc_buf_sp24340(.A (B[28]), .Z (n_1326));
1319      CLKBUF_X1 drc_buf_sp24343(.A (B[29]), .Z (n_1291));
1320      CLKBUF_X1 drc_buf_sp24346(.A (B[18]), .Z (n_1316));
1321      CLKBUF_X1 drc_buf_sp24349(.A (A[3]), .Z (n_1020));
1322      CLKBUF_X1 drc_buf_sp24352(.A (B[30]), .Z (n_1301));
1323      BUF_X1 drc_buf_sp24355(.A (op_ctl[7]), .Z (n_1143));
1324      INV_X1 drc_bufs24470(.A (n_2750), .ZN (n_1456));
1325      INV_X1 drc_bufs24471(.A (op_ctl[6]), .ZN (n_2750));
1326      CLKBUF_X1 drc_buf_sp24474(.A (B[1]), .Z (n_1256));
1327      BUF_X1 drc_buf_sp24477(.A (op_ctl[8]), .Z (n_1005));
1328      XOR2_X1 g24876(.A (n_813), .B (n_397), .Z (n_3287));
1329      XOR2_X1 g24877(.A (n_782), .B (n_800), .Z (n_3288));
1330      MUX2_X1 g24878(.A (n_673), .B (n_648), .S (n_768), .Z (n_3289));
1331      MUX2_X1 g24879(.A (n_659), .B (n_671), .S (n_741), .Z (n_3290));
```

```
1332      AOI221_X1 g24880(.A (n_373), .B1 (n_379), .B2 (n_323), .C1 (n_408),
1333          .C2 (n_465), .ZN (n_3291));
1334      INV_X1 g24881(.A (n_3292), .ZN (n_3293));
1335      OAI221_X1 g24882(.A (n_367), .B1 (n_192), .B2 (n_411), .C1 (n_315),
1336          .C2 (n_409), .ZN (n_3292));
1337      AOI221_X1 g24883(.A (n_318), .B1 (n_443), .B2 (n_450), .C1 (n_389),
1338          .C2 (n_449), .ZN (n_3294));
1339      INV_X1 g24884(.A (n_3295), .ZN (n_3296));
1340      OAI221_X1 g24885(.A (n_332), .B1 (n_391), .B2 (n_392), .C1 (n_410),
1341          .C2 (n_395), .ZN (n_3295));
1342      AOI222_X4 g24886(.A1 (n_131), .A2 (n_1010), .B1 (n_172), .B2
1343          (n_1100), .C1 (n_173), .C2 (n_997), .ZN (n_3297));
1344      INV_X1 drc_bufs(.A (n_506), .ZN (n_992));
1345      INV_X1 drc_bufs24898(.A (op_ctl[9]), .ZN (n_506));
1346    endmodule
```

References

[1] Reuben Louis Goodstein. *Boolean algebra*. Courier Corporation, 2007.

[2] A Anand Kumar. *Fundamentals of digital circuits*. PHI Learning Pvt. Ltd., 2016.

[3] Olin L MacSorley. “High-speed arithmetic in binary computers”. In: *Proceedings of the IRE* 49.1 (1961), pp. 67–91.

[4] Hamed Naseri and Somayeh Timarchi. “Low-power and fast full adder by exploring new XOR and XNOR gates”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.8 (2018), pp. 1481–1493.