Altera's Max+plus II Tutorial

Written by Kris Schindler

To accompany *Digital Principles and Design* (by Donald D. Givone)

8/30/02

About Max+plus II

Altera's Max+plus II is a powerful simulation package used in the digital design industry. It allows an engineer to design, prototype, test, and debug a circuit prior to implementation. This is very important, since it allows circuits to be implemented faster and cheaper. This tutorial provides an overview of Max+plus II and includes several design examples which have been worked through in detail.

Installation

Installing Max+plus II

To install the version of Max+plus II that comes with *Digital Principles and Design*, simply place the CD in your CD-ROM drive and select Start → Run. A dialog box will appear. Select the file *mp2_101se.exe* on the CD. As the application runs, it will step you through the installation procedure. Additional information regarding the installation procedure are outlined at http://www.alteral.com/education/univ/univ-student_install.html.

Obtaining a license

A license file is required to use Max+plus II. To obtain the license, go to the following website: http://www.altera.com/support/licensing/lic-university.html. Select MAX+PLUS II Student Edition software and click Continue. Enter your hard disk volume serial number and click Continue. Instructions on how to obtain this number are given on the web page. Fill in the fields on the form given on web page and click Continue. This completes the procedure. The license file will be sent to you via e-mail. The file is called license.dat

Implementing a Circuit from a Function or a Truth Table

When implementing a circuit, one must first determine the characteristic equations of the circuit so that it can be directly translated into a gate level description of the circuit. Minimization is recommended. Several minimization methods, such as Karnaugh maps, Quine McCluskey method, and Petrick's method are outlined in *Digital Principles and Design,* by Donald D. Givone. The details of determining these equations are not outlined in this tutorial, since these topics are covered in depth in the textbook.

Using Altera

To illustrate how to use Max+Plus II, a 2-bit priority encoder with an enable input (E) and a valid bit output (V) will be designed. The truth table for the encoder is shown in table 1.

D3	D2	D1	D0	E	A1	A0	V
1	X	X	X	1	1	1	1
0	1	X	X	1	1	0	1
0	0	1	X	1	0	1	1
0	0	0	1	1	0	0	1
X	X	X	X	0	0	0	0

Table 1 – Priority encoder truth table.

The equations characterizing each of the three outputs are shown in figure 1.

$$A_1 = ED_3 + ED_2$$
$$A_0 = ED_3 + ED_2'D_1$$
$$V = ED_3 + ED_2 + ED_1 + ED_0$$

Figure 1 – Equations for outputs A1, A0, and V.

Creating a Project

We start by creating a project. This is accomplished by selecting File → Project → Name, entering the working directory, and name of the project. A suitable name for our project is *encoder*. After this is done, the design can be started.

Schematic Capture

Circuits can be entered into Max+plus II using schematic capture. To implement the priority encoder, select Max+plus II → Graphic Editor. The blank graphic editor window that appears is where the design is entered. Components are incorporated into the design by selecting them from a list of devices. The encoder we are designing is composed of primitive components. To select from these components, right click in the graphic editor window. A menu will appear. Select Enter Symbol. Another dialog box will appear. Select the library which contains default primitive components (c:\maxplus2\max2lib\prim), by double clicking on the *prim* library. The devices listed under Symbol Files contain the components that can be selected. The gates in the list are named by the type of gate, followed by the fan-in of the gate. We will start with the output A1. Select the two input AND gate by double clicking on *and2*. Repeat the process for another two input AND gate, and a two input OR gate (*or2*). Gates can be moved by dragging the gate. Arrange the gates as shown in figure 2.

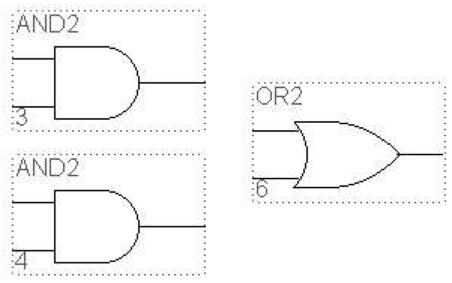


Figure 2 – Gates required for the priority encoder.

The next step is to connect the gates together. Wires are implemented by left clicking at the starting point of the wire, and dragging the wire to the endpoint. Note that Max+plus II will automatically create a 90 degree bend in the wire as needed. To illustrate this, place the cursor over the output of one of the AND gates. The cursor will change to the wire tool (+). Left click on the output, and create a wire that bends and ends inline with the input of the OR gate, as shown in Figure 3. Create another wire to complete the connection to the OR gate, as shown in figure 4.

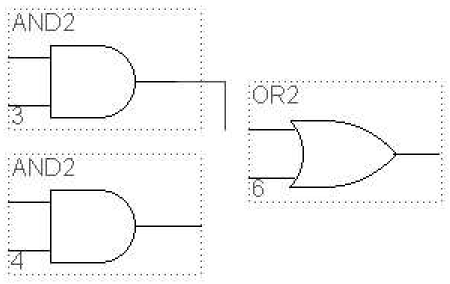


Figure 3 – Wire from output of the AND gate.

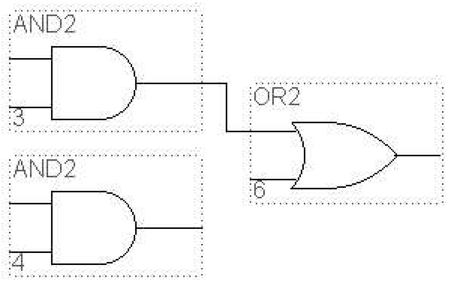


Figure 4 – AND and OR gates connected together.

Repeat the wiring procedure to connect the output of the second AND gate to the input of the OR gate. The result is shown in figure 5.

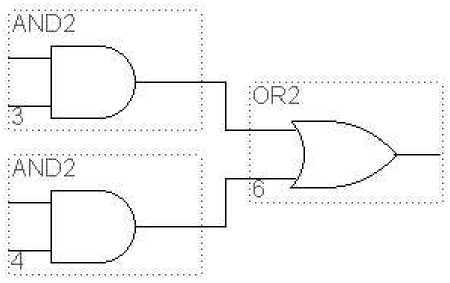


Figure 5 – AND and OR gates wired together

Now that the gates have been placed and wired, we need to create input and output terminals. Using the same procedure we used for the gates, select three input terminals and one output terminal from the list of primitive components. The component names are *input* and *output*. Place the terminals as shown in figure 6.

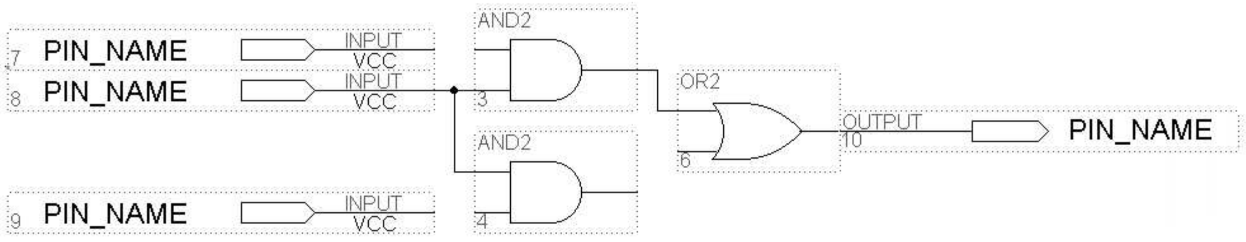


Figure 6 – Placement and wiring of input/output terminals.

The next step is to wire the input and output terminals to the gates. We'll start with the enable input (E). The second input terminal will be used. Wire the terminal to the upper AND gate. Then wire the second AND gate to the wire you created in the previous step. Notice the connection dot that is created. This connection dot can be removed or created by placing the cursor over the intersection of two wires, right clicking, and selecting Toggle Connection Dot.

The next step is to label the input terminal with the appropriate pin name. Label the input terminal just wired by double clicking on PIN_NAME, and then replacing the selected text with E. Notice that the output terminal in figure 6 was placed such that the port (connection to the component) was placed at the output of the OR gate. Hence, the connection was created automatically, without the need for explicitly creating a wire.

Continue to apply the techniques introduced in this section to construct the rest of the circuit, as shown in figure 7. Then save your design, by selecting File → Save As, and entering a name for your design. A good name would be *encoder*.

After the design is complete, compile the design, by selecting File → Project → Save & Compile. Any errors in the design will be displayed. Correct the errors and recompile. Once you have a design that compiles, you are now ready to simulate the circuit.

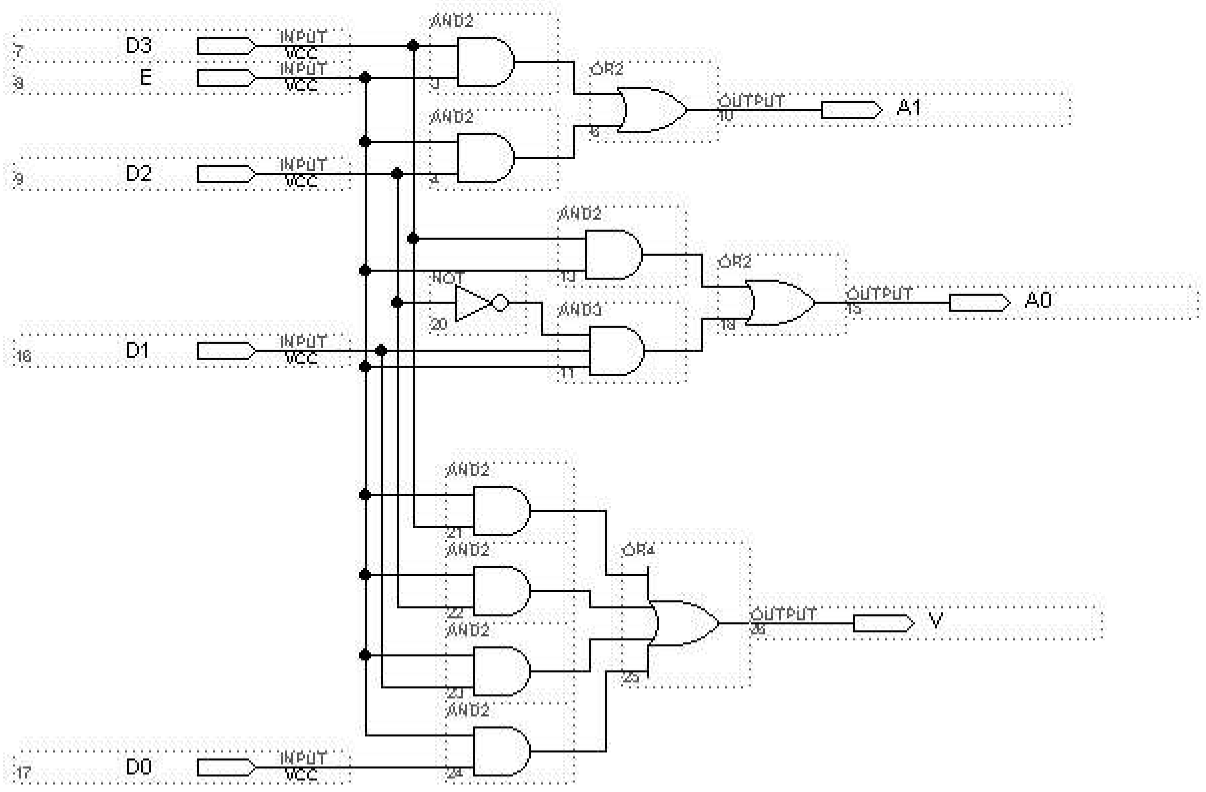


Figure 7- Priority encoder.

Verification

Max+plus II provides a powerful, user-friendly interface for verification of a design. We will now verify the functionality of the priority encoder. Select Max+plus II → Waveform Editor. Once the waveform editor has appeared, right click, and select Enter Nodes from SNF. Click on List, and click on the right arrow (⇒) to select all the input and output nodes. The nodes will be copied from the available nodes and groups list to the selected nodes and groups list. At this point, the input pattern to be used in the simulation will be constructed. Once this step has been completed, the circuit will be simulated to analyze the outputs and verify the correct functionality.

To construct the input pattern, first set the grid size by selecting Options → Grid Size and setting the grid size to 20 ns. Select the inputs D3 through D0 by holding down the shift key and clicking on D3, D2, D1, and D0. With the cursor over the highlighted area, right click and select Enter Group. The group name will default to D[3..0]. This will allow these inputs to be utilized as a 4-bit bus. Select the Radix as Hex, and then click on OK.

To better view the waveforms, select View → Time Range, set the range from 0 ns to 640 ns, and click OK. This is the range required for our exhaustive simulation.

If the group D[3..0] is not still selected, left click on the group. Over the highlighted area, left click on the count button (C) located along the left hand side of the window. This will allow us to exhaustively test the circuit without having to set each of the inputs D3, D2, D1, and D0 individually. The starting value should be 0, the *increment by* field should be 1, the *multiplied by* value should be 1, and the *count type* should be set to binary. Verify that all of these values are set appropriately and click on OK. If the group is still not selected, select it. Right click on the highlighted area, and select ungroup. This will display D3, D2, D1, and D0 independently.

The next input that needs to be set is the enable input (E). Select the Enable waveform between 0 ns and 320 ns. Click on the logic 1 button (1) on the left hand side of the window to set the selected waveform to a logic 1. The input waveform has now been created. The waveforms are shown in figure 8.

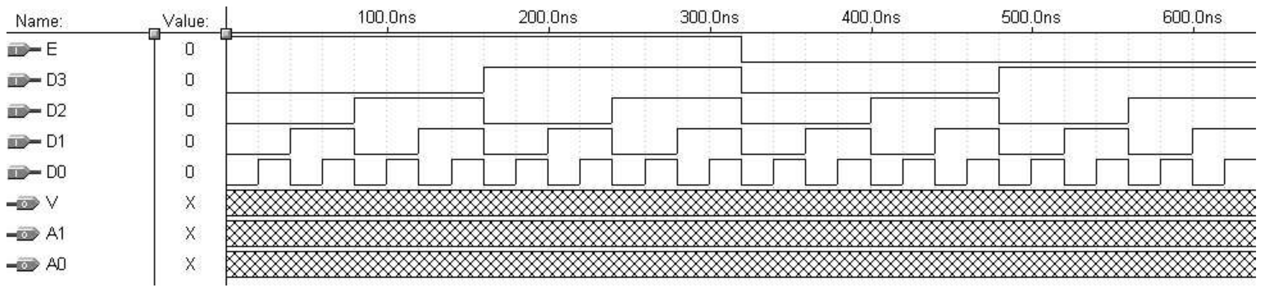


Figure 8 – Priority encoder waveforms prior to simulation.

To run the simulation, select File → Project → Save, Compile, & Simulate . A dialog box will appear, allowing the simulation file to be saved. Click on OK. The circuit is then simulated. Afterwards, a dialog box will appear indicating the simulation has completed. Click on OK. In the Simulator: Timing Simulation window, click on Open SCF. The resulting simulation is shown in figure 9.

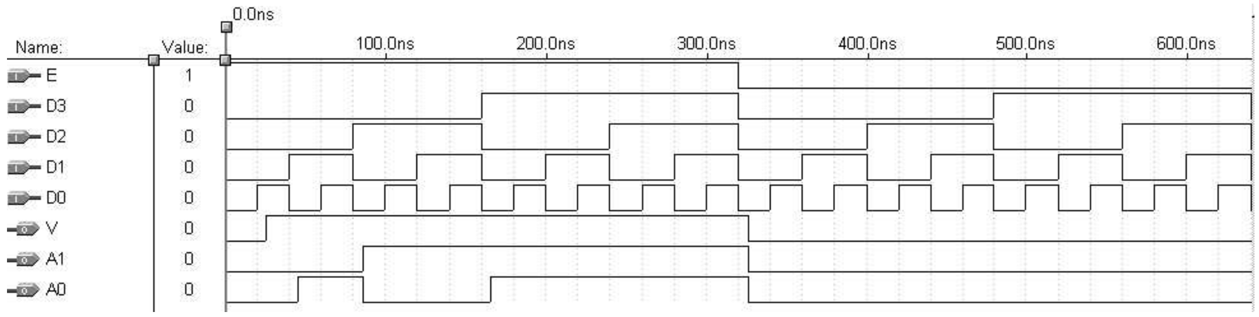


Figure 9 – Priority encoder waveforms after simulation.

Functional Simulation

The simulation performed in the previous section is a timing simulation. Delays associated with each of the gates are part of the library. To perform a functional simulation where the delays are not used, proceed as follows. Close the Waveform Editor and Simulator windows. Click on the Compiler window so it is active. If the Compiler window is not open, select File → Project → Save and Compiler. Select Processing → Functional SNF Extractor. The next step is to run the simulation by selecting File → Project → Save, Compile, & Simulate. Open the SCF to display the results of the functional simulation. The waveform for the encoder is shown in figure 10.

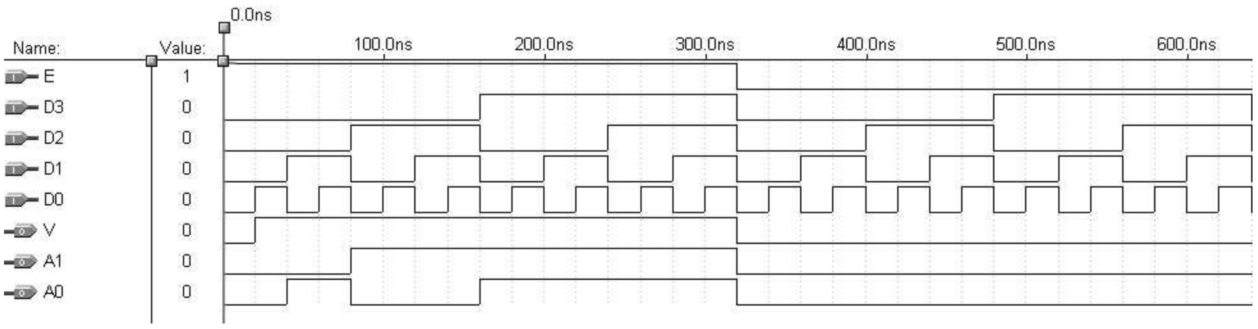
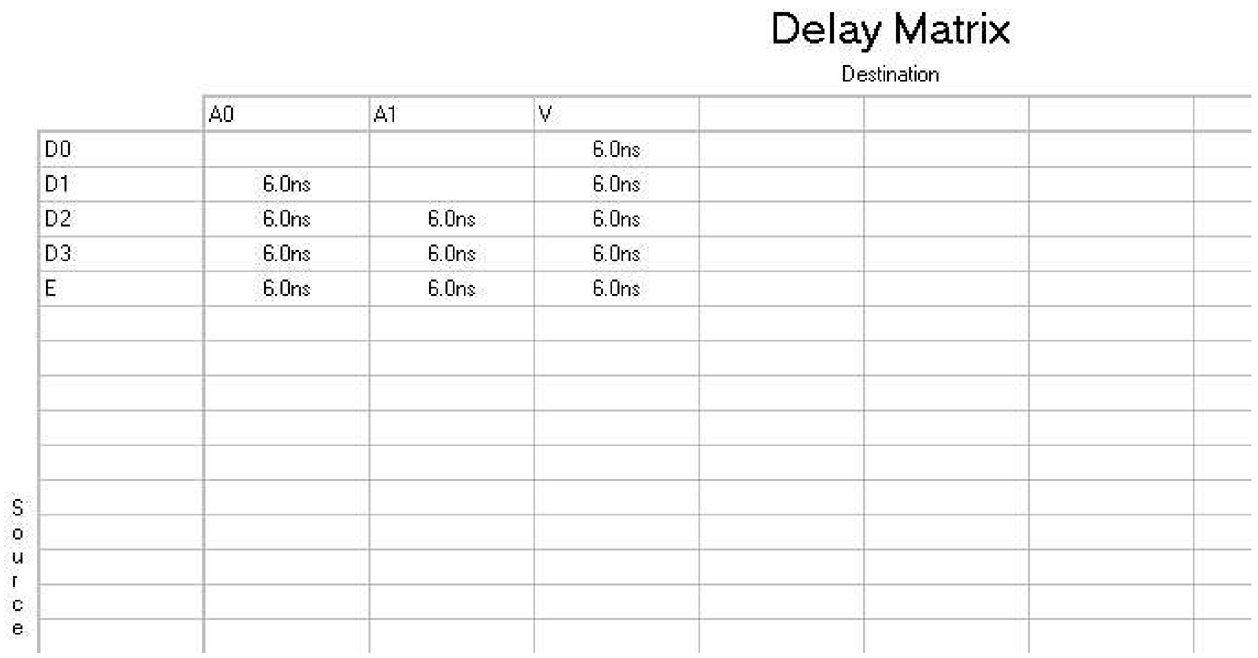


Figure 10 – Functional simulation of priority encoder

Timing

Max+plus II provides a timing analysis tool which analyzes the delays in a circuit. To run this analysis, go back to the Compiler window and make sure that the Functional SNF Extractor is not checked. If it is, click on it to deselect it. Simulate the circuit and open the Waveform Editor. To view the timing analysis, select Utilities → Analyze Timing. The delay matrix window which appears analyzes the timing delays between each input and each output. The matrix for the priority encoder is shown in figure 11.



Delay Matrix

Destination

Source	A0	A1	V
D0			6.0ns
D1	6.0ns		6.0ns
D2	6.0ns	6.0ns	6.0ns
D3	6.0ns	6.0ns	6.0ns
E	6.0ns	6.0ns	6.0ns

Figure 11 – Timing analysis of priority encoder.

Modular Design

Max+plus II supports modular design. To illustrate this, let us consider the priority encoder. Go back to the graphic editor showing the schematic of the priority encoder. To create a symbol of the priority encoder that can be utilized in a modular design, select File → Create Default Symbol. The symbol which is created can be selected for use in other designs in the same way that gates, input terminals, and output terminals were selected for use in the encoder. The default symbol is shown in figure 12. Notice that the location of the pins are similar to that of the schematic. Modifications can be made to the symbol (such as the rearrangement of pins) by selecting File → Edit Symbol. Pins and labels can be selected and dragged to new locations or otherwise modified as desired. Figure 13 shows a modified symbol. Notice that the pins were rearranged so that D3 through D0 are adjacent.

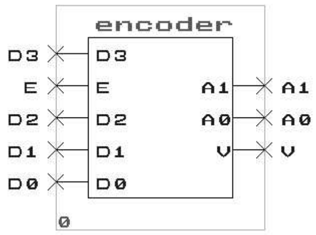


Figure 12 – Default symbol for the priority encoder.

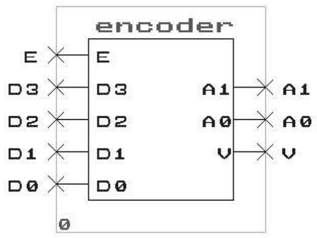


Figure 13 – Edited symbol for the priority encoder.

Examples/Problems

Arithmetic Logic Unit (ALU)

In this example, a 4-bit arithmetic logic unit (ALU) will be designed. The ALU operates on two 4-bit inputs (X, Y) as defined by a 2-bit opcode (OP). The opcode selects between addition, subtraction, arithmetic shift right, and logical shift left. The opcodes and their associated functions are summarized in table 2. The result is placed on the 4-bit output, F. There is also an overflow bit, which indicates if the result of the addition or subtraction has overflowed the 4-bit output. This overflow bit is reset to logic 0 for both shift operations. The ALU is shown in figure 14.

Function	**Opcode**	**Description**
Add	00	X + Y
Subtract	01	X – Y
Logical Shift Left	10	Logical left shift X *n* places, where *n* is represented by Y_1Y_0
Arithmetic Shift Right	11	Arithmetic right shift X *n* places, where *n* is represented by Y_1Y_0

Table 2 – Summary of ALU functions.

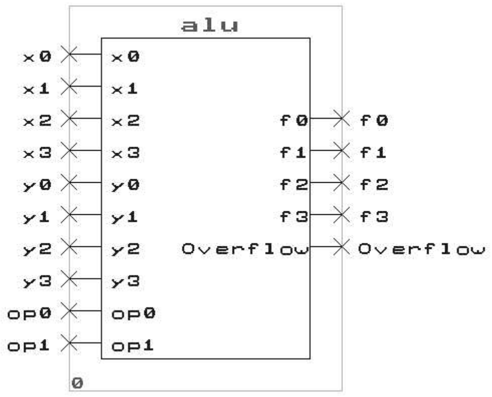


Figure 14 – ALU Symbol.

A top-down design approach will be used to implement this design. Functional units will be needed for addition/subtraction, logical left shift, and arithmetic right shift. In addition, a series of multiplexors will select the output from the appropriate functional unit, and route the data to F and Overflow, based on the opcode. A block diagram of the ALU is shown in figure 15.

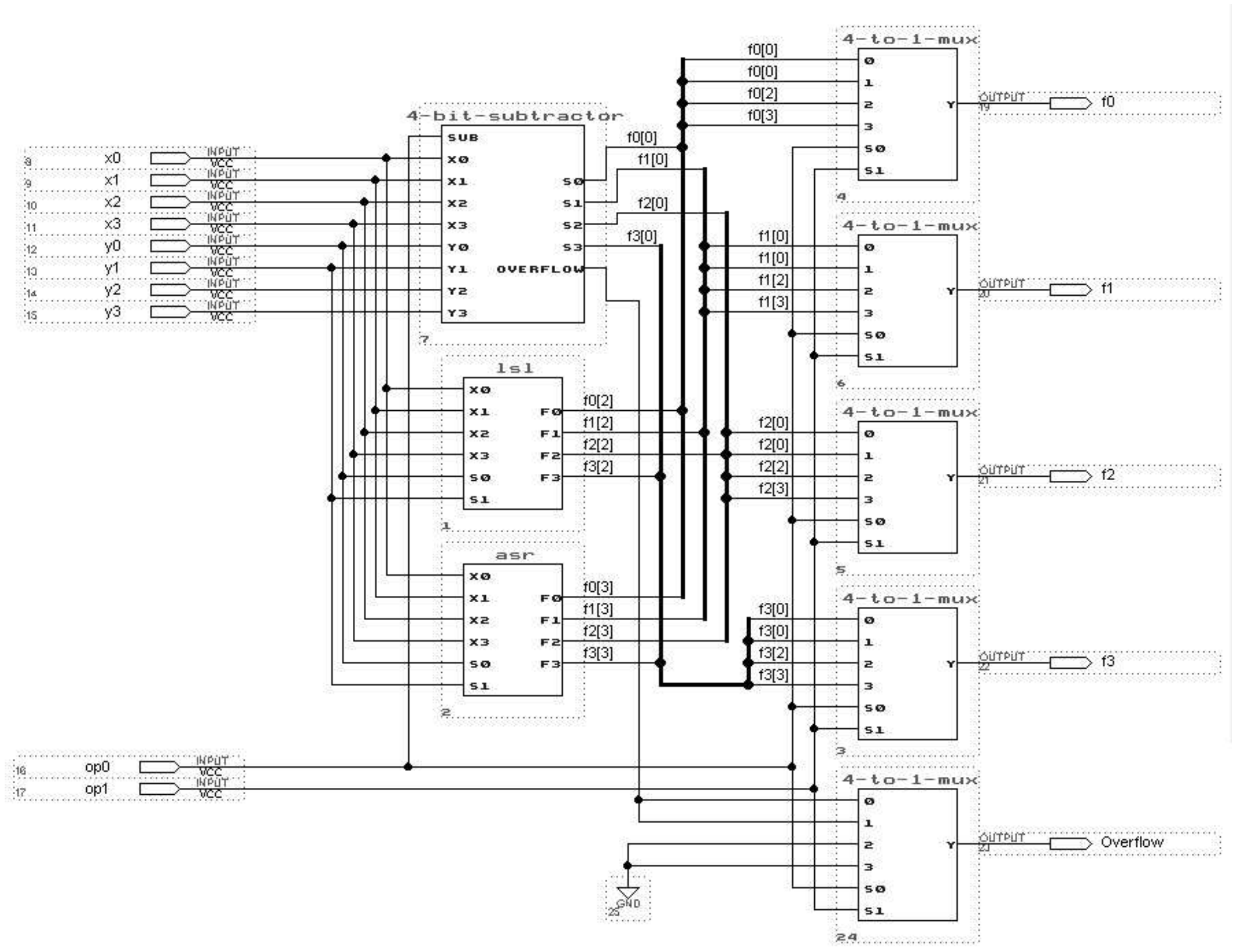


Figure 15 – Block diagram of the ALU.

We'll start with the design of the 4-to-1 multiplexor. A 4-to-1 multiplexor can be designed from three 2-to-1 multiplexors, as shown in figure 16. Hence, we can focus our efforts on designing and testing the 2-to-1 multiplexor, and then use that as a building block for the 4-to-1 multiplexor.

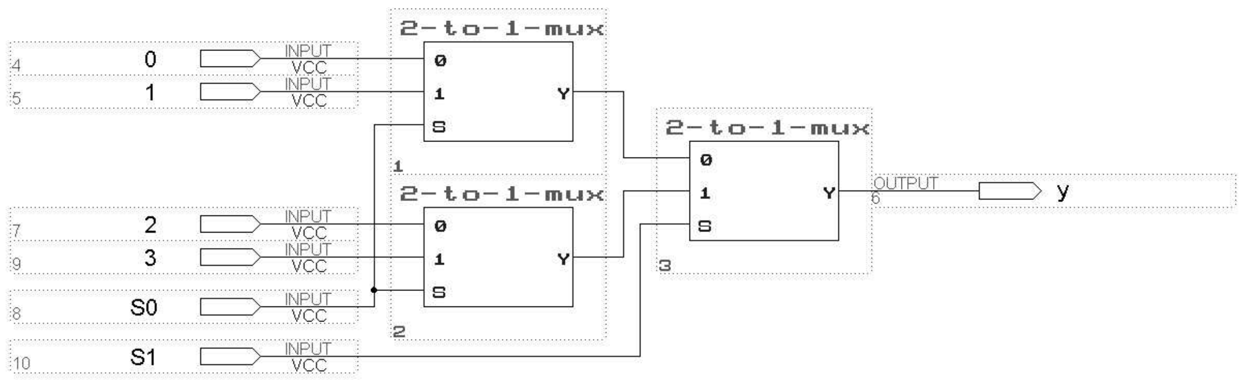


Figure 16 – 4-to-1 multiplexor.

A gate level diagram of the 2-to-1 multiplexor is shown in figure 17. The first step in the design of the ALU is to implement this circuit and thoroughly test it so that it can be instantiated where ever needed throughout the design. An exhaustive simulation verifying the circuit's functionality is shown in figure 18.

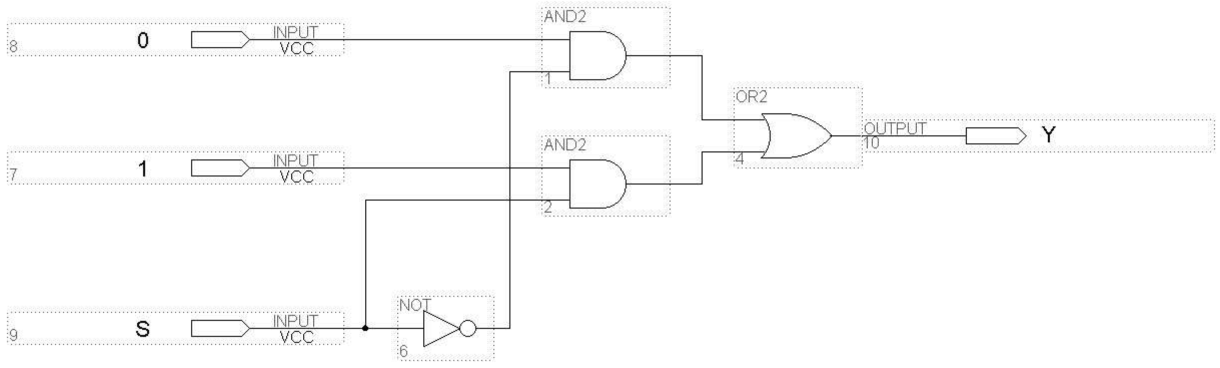


Figure 17 – 2-to-1 multiplexor.

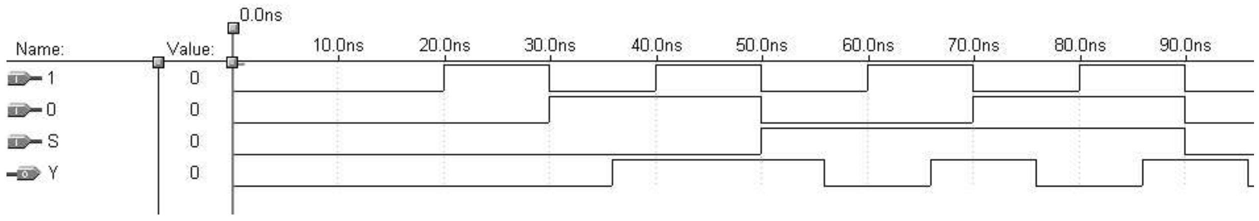


Figure 18 – Simulation of 2-to-1 multiplexor.

Now that the 2-to-1 multiplexor has been designed and tested, it can be instantiated in the design of the 4-to-1 multiplexor. The design is shown in figure 16 and an exhaustive simulation verifying its functionality shown in figure 19.

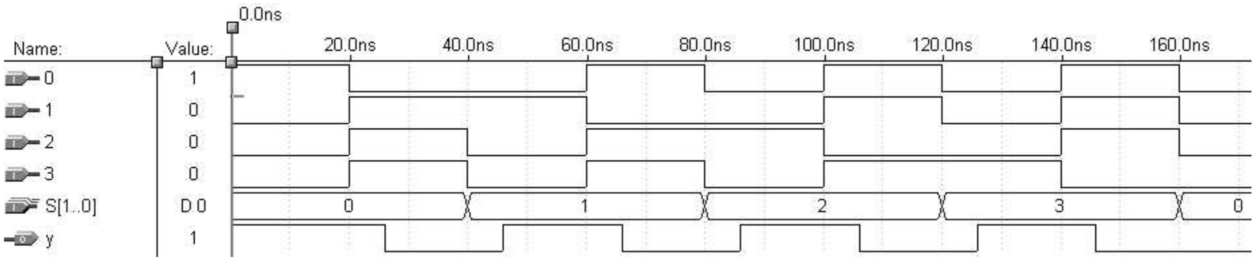


Figure 19 – Simulation of 4-to-1 multiplexor.

The next step is to design the functional units. We'll start with the 4-bit adder/ subtractor which consists of four cascaded full adders, a circuit for performing the two's complement when subtraction is performed, and an overflow detection circuit. The first step is the design of the full adder, which is shown in figure 20.

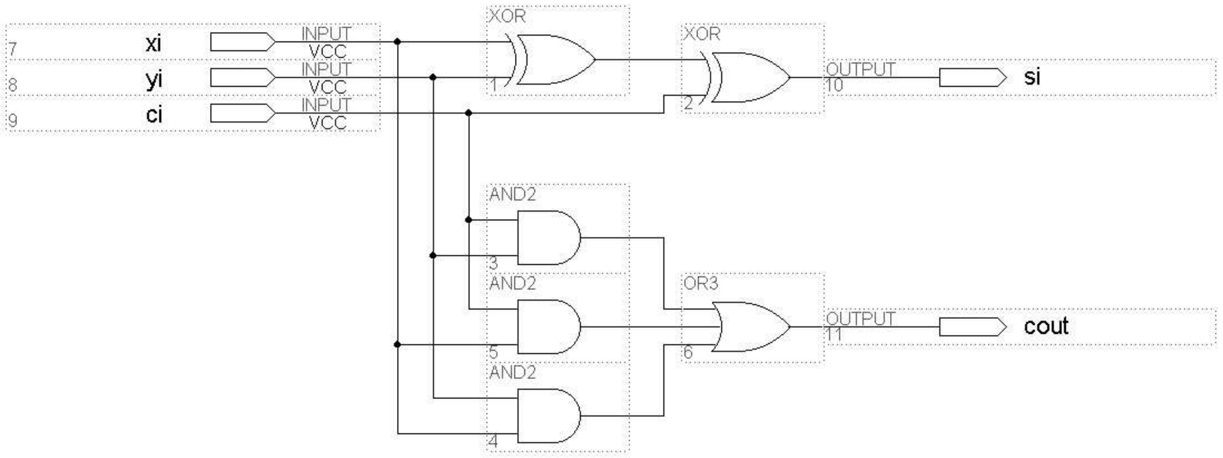


Figure 20- Full adder.

The next step is to thoroughly test the full adder. Due the limited number of inputs and outputs, exhaustive testing will be utilized to ensure correct functionality. The simulation is shown in figure 21.

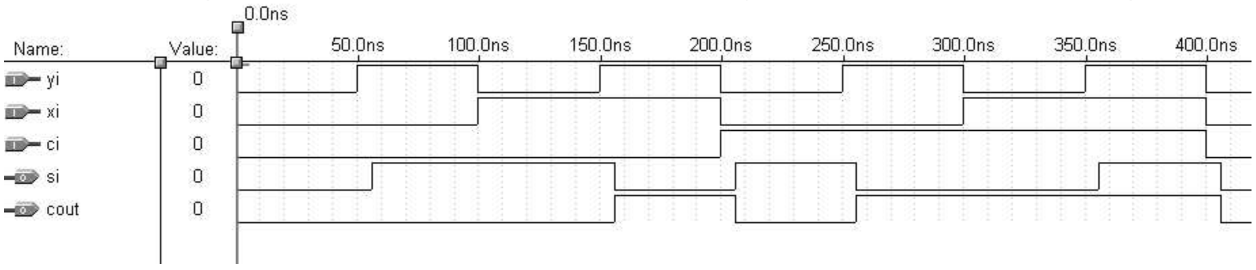


Figure 21 – Full adder simulation.

Now that the full adder has been verified, create the default symbol for the full adder and incorporate four full adders into a 4-bit adder/subtractor with overflow detection. The circuit is shown in figure 22 and the simulation in figure 23. Notice that at 200 ns, the applied inputs produce a negative result, so the output (0xB) corresponds to the correct value (-5).

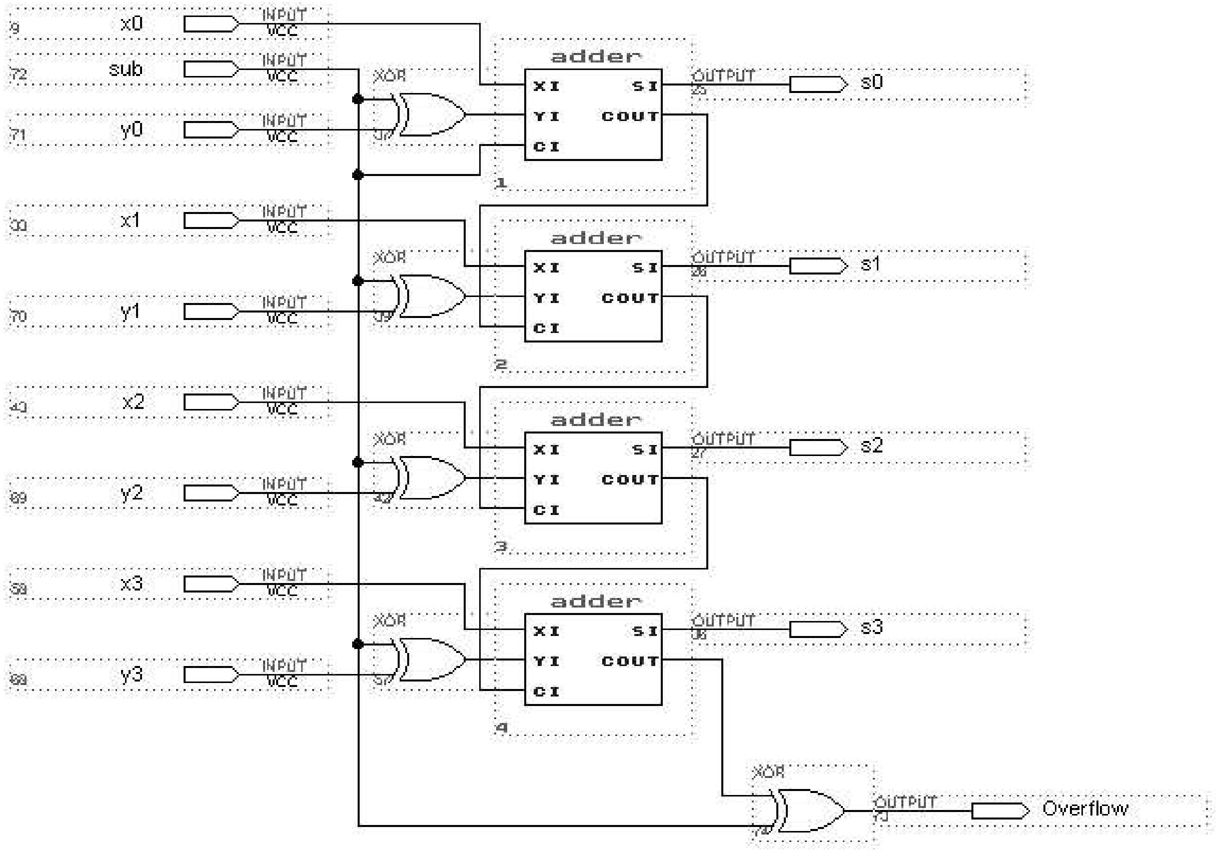


Figure 22 – 4-bit adder/subtractor circuit.

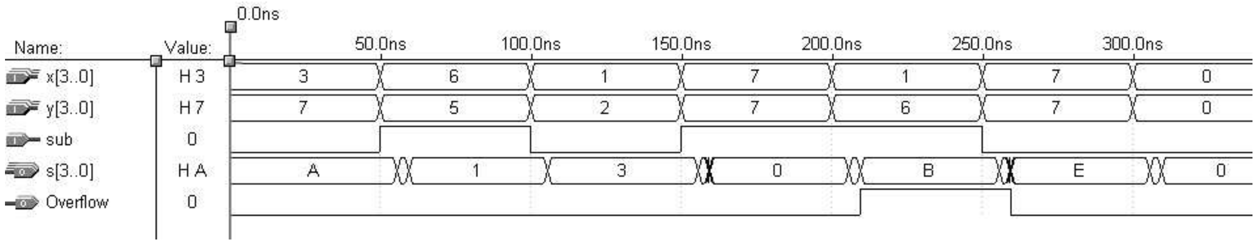


Figure 23 – Simulation of the 4-bit adder/subtractor circuit.

A barrel shifter can be utilized to efficiently implement logical shift left and arithmetic shift right. The barrel shifter consists of a series of multiplexors which route each bit at the input to the correct output, effectively performing the shift operation. The barrel shifter used to perform the logical left shift operation is shown in figure 24, and the corresponding simulation is shown in figure 25. The analogous barrel shifter for the arithmetic right shift is shown along with the simulation in figures 26 and 27 respectively.

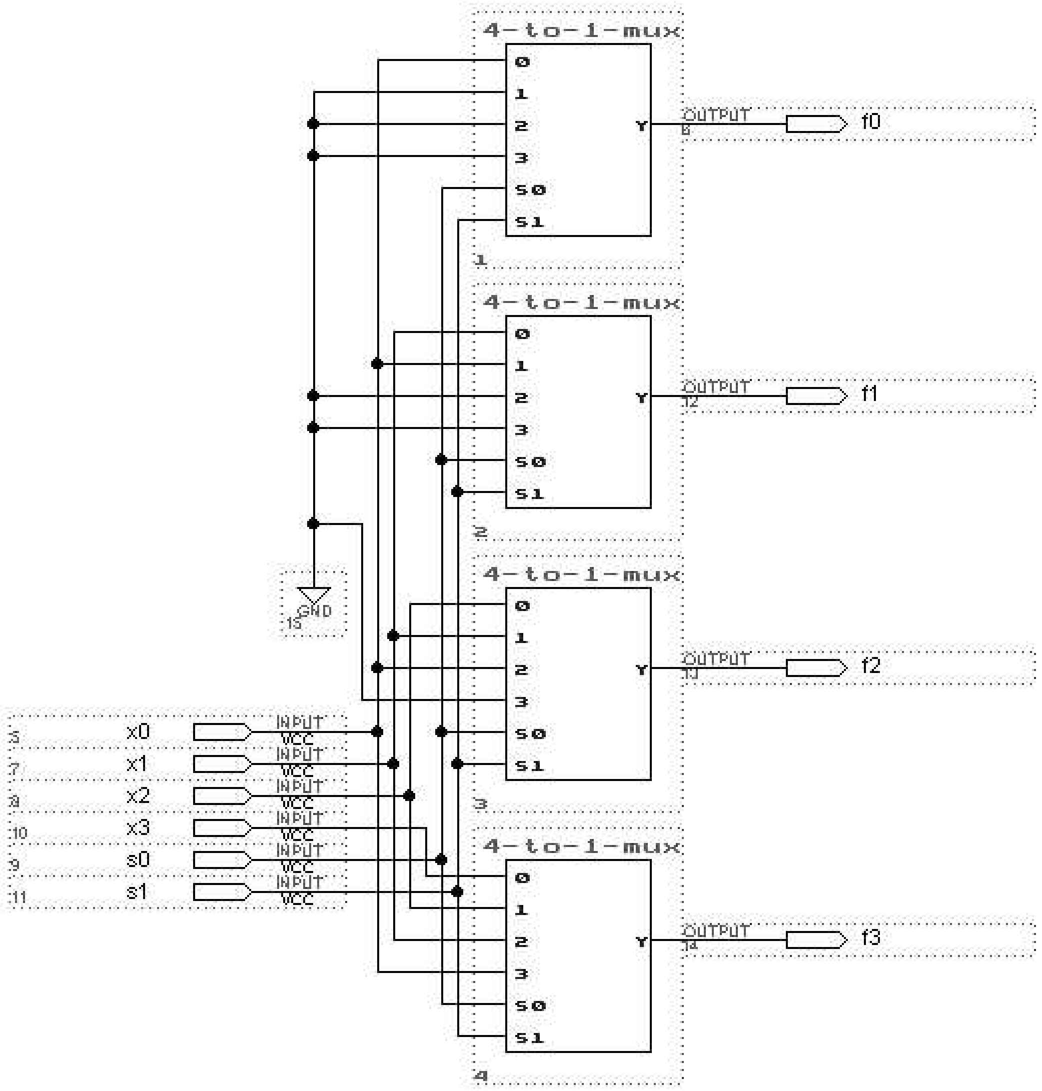


Figure 24 – Logical left shift circuit.

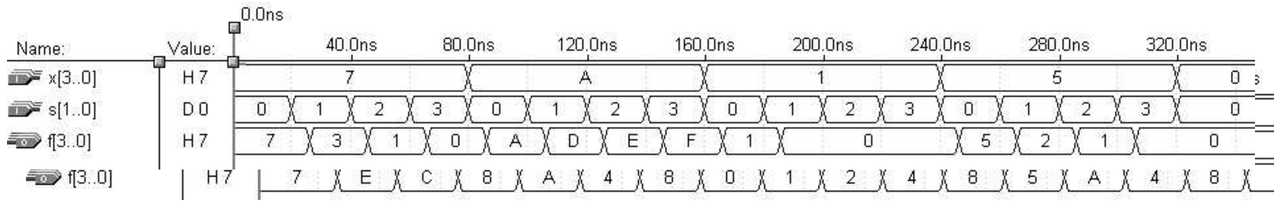


Figure 25 – Simulation of logical left shift circuit.

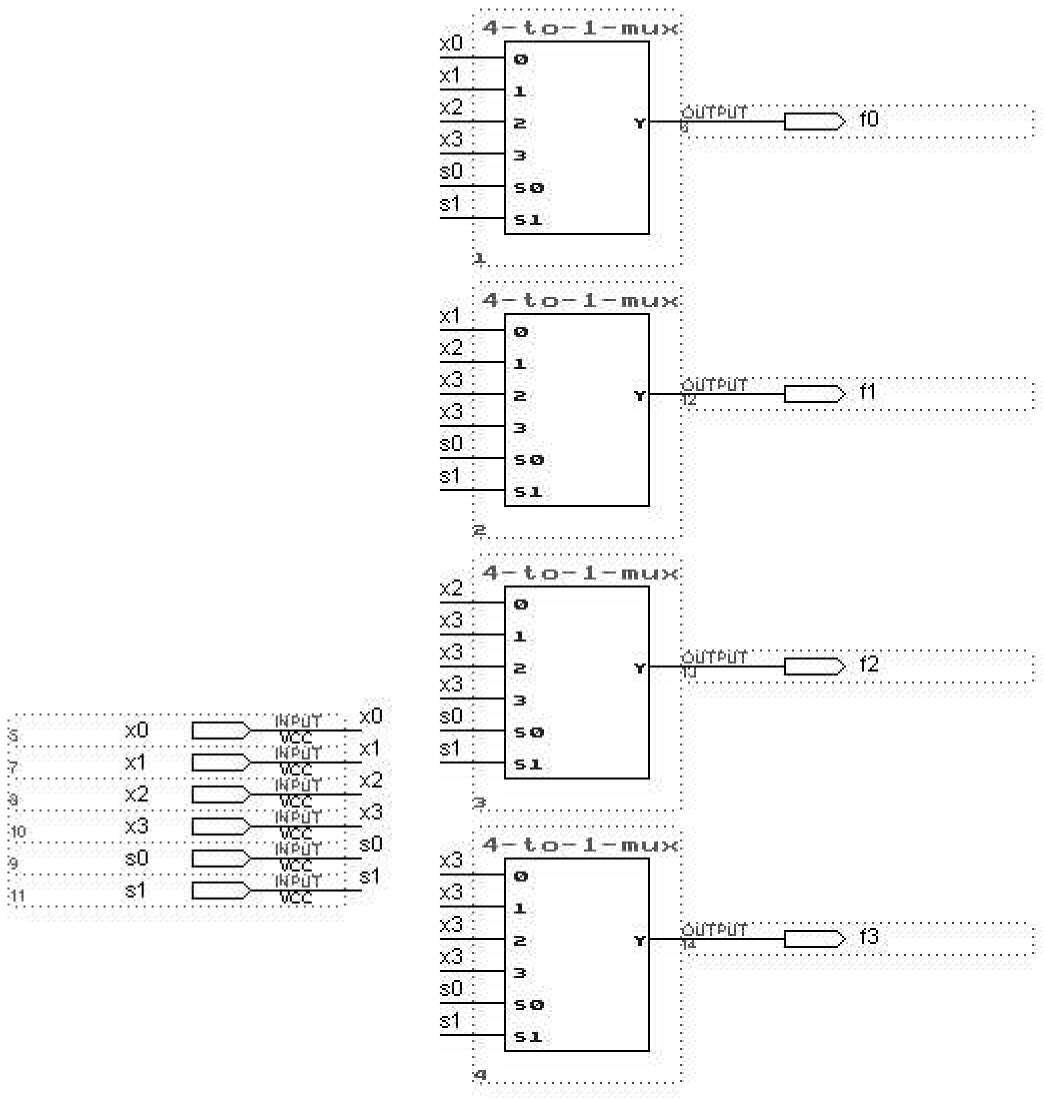


Figure 26 – Arithmetic right shift circuit.

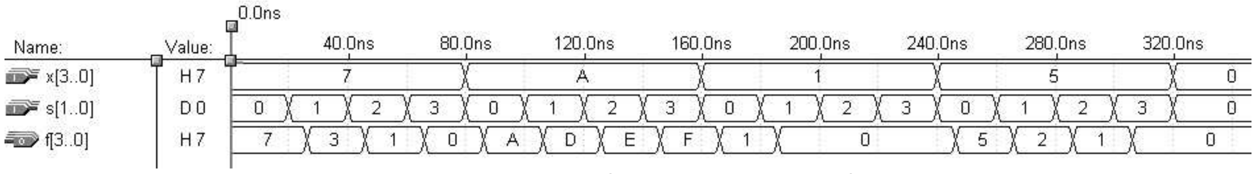


Figure 27 – Simulation of arithmetic right shift circuit.

Now that all the functional components have been designed, the overall design will now be implemented. An output multiplexor will select from the outputs from the functional units to route the proper function's output to the output of the ALU. The output multiplexor actually consists of five 4-to-1 multiplexors: one for F0, F1, F2, F3, and overflow. The output multiplexor is shown in detail in figure 28.

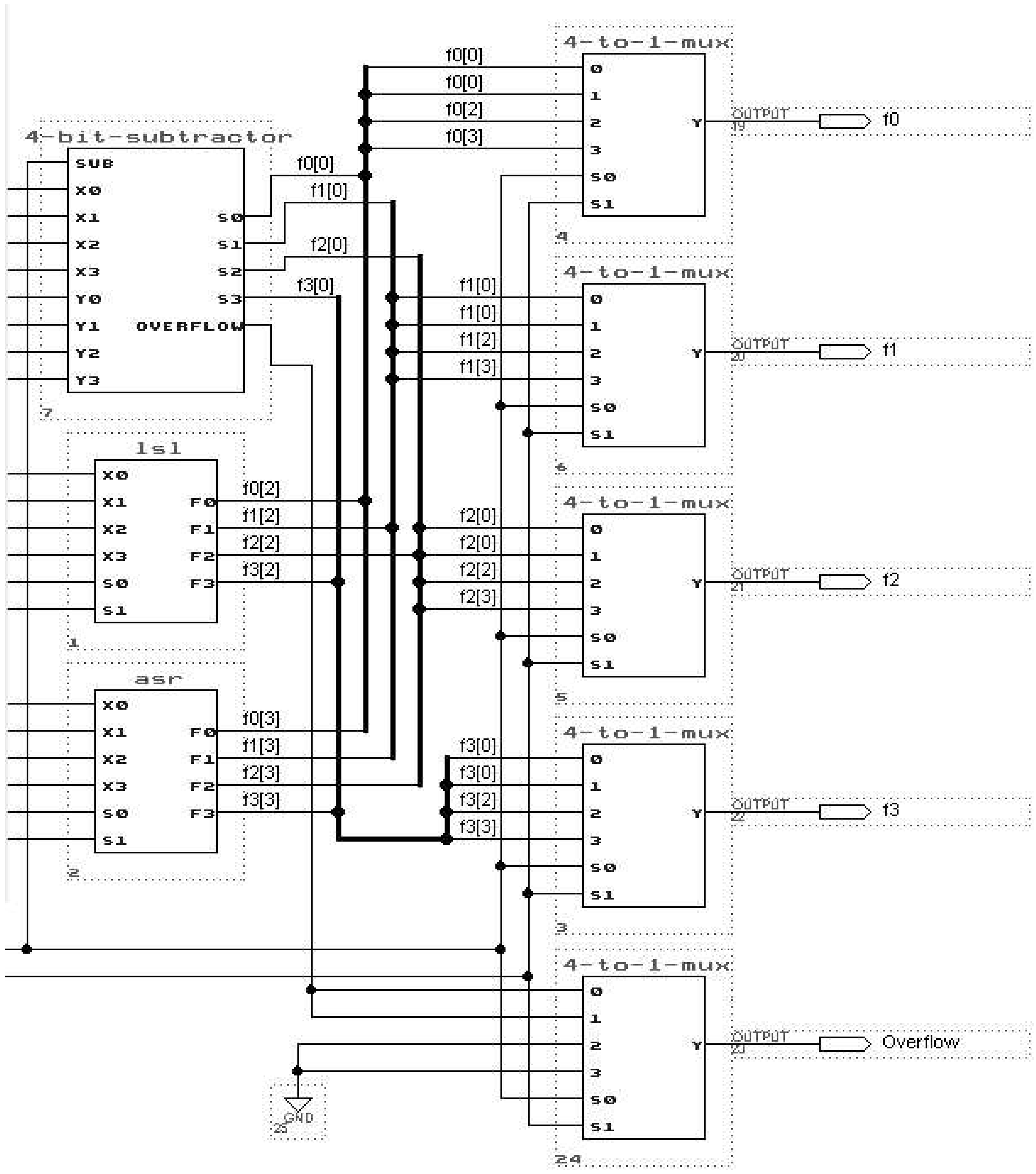


Figure 28 – Output multiplexor.

The darker wires used in connecting the functional unit outputs to the output multiplexor are busses, which help clean up the design, making it easier to read. To implement a bus, right click over the wire, select Line Style, and select the second style from the top (the thick line). When a single connection is made to the bus, right click over the single wire, select Enter Node/Bus Name, and enter a name for that connection. The output multiplexor utilizes four busses: f0, f1, f2, and f3. One bus is used to connect the four outputs from the ALU to the outputs from the various functional units an index is used to specify the individual wires in the bus. For example, the four wires which comprise the bus f0 are referenced by f0[0], f0[1], f0[2], and f0[3].

Once the output multipliexor has been wired to the functional units, the primary inputs should be routed to the appropriate functional units. The overall design is shown in figure 29. Simulation results are shown in figure 30.

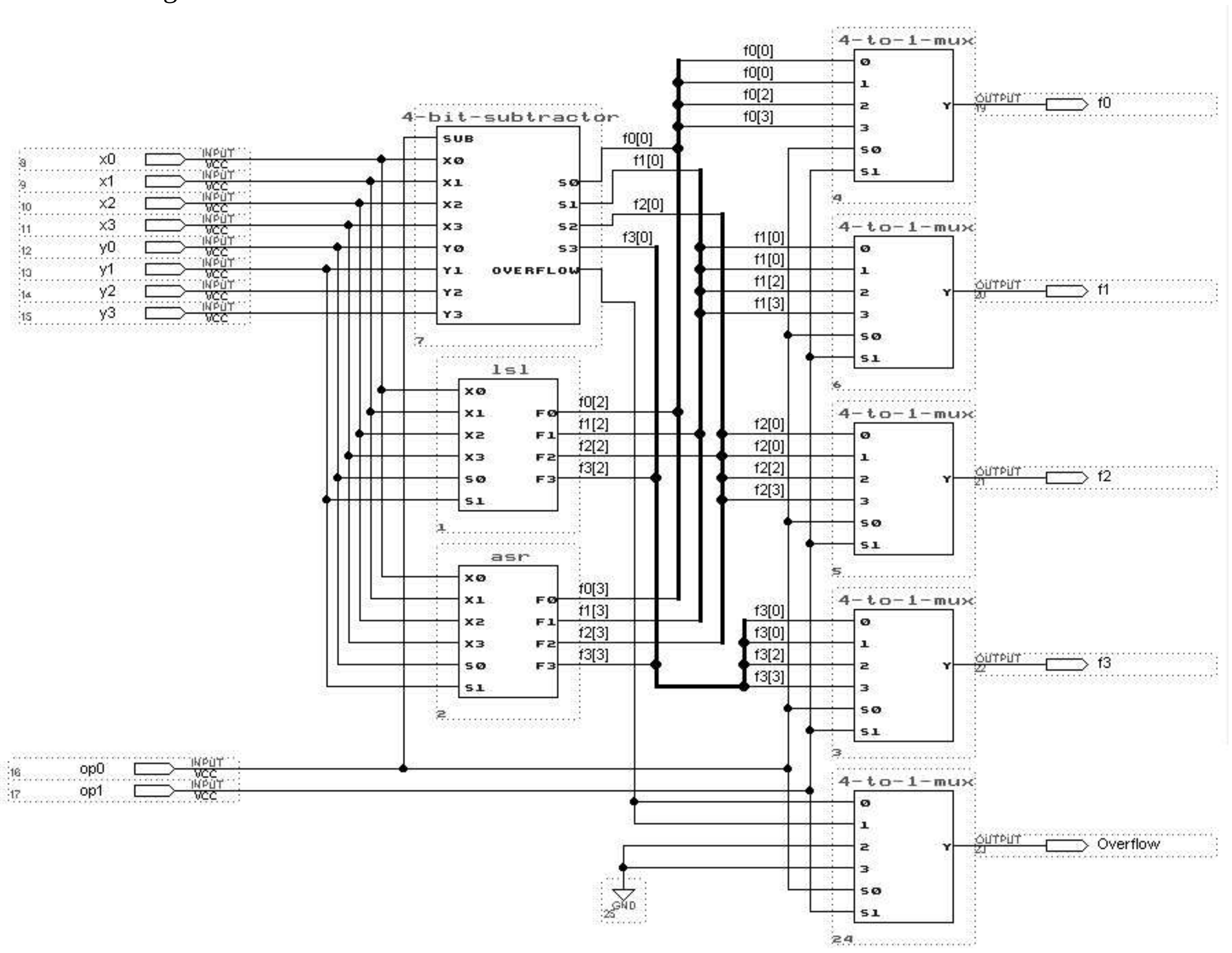


Figure 29 – Overall ALU.

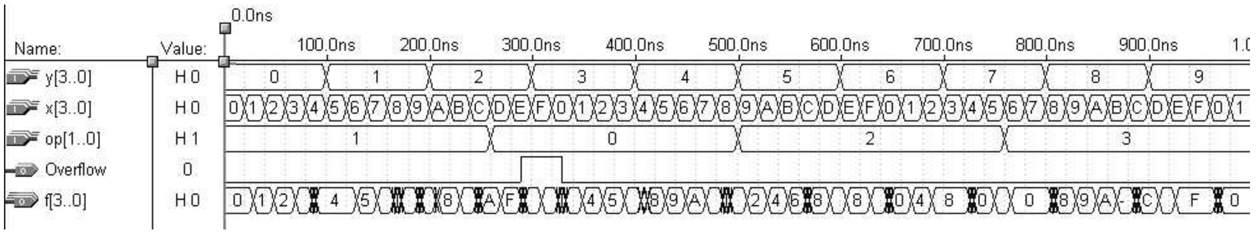
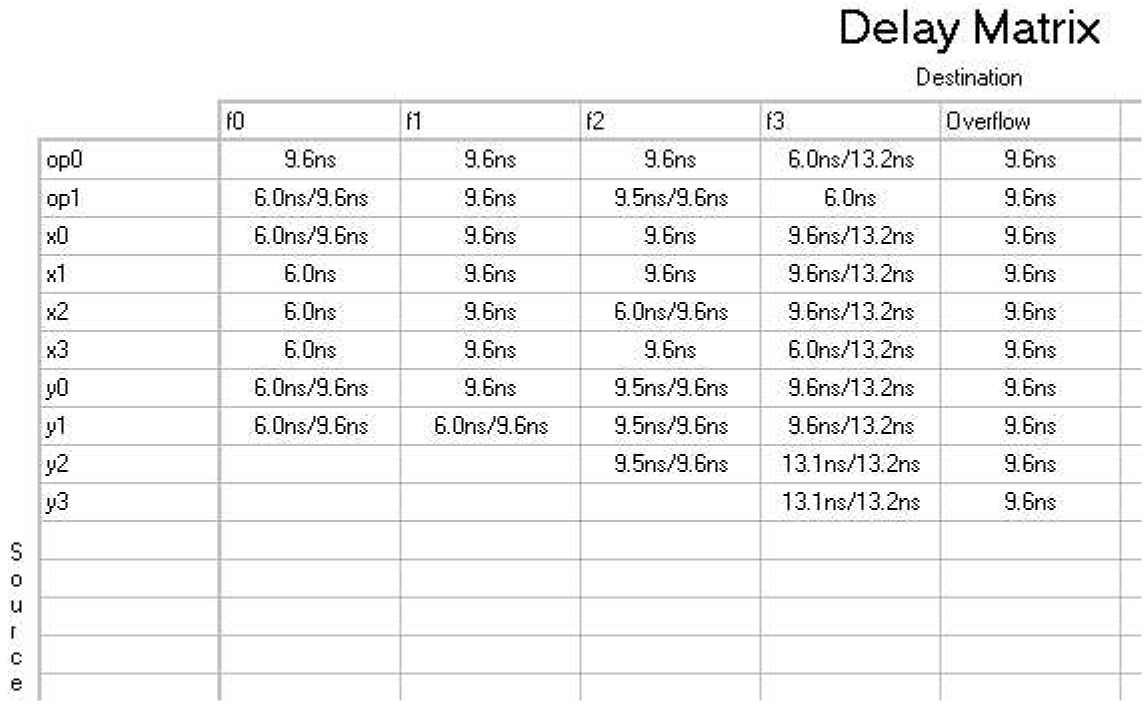


Figure 30 – Simulation of ALU.

Timing analysis reveals that the worst case delay along the critical path is 13.2 ns, as shown in figure 31. This can be improved by replacing the ripple carry adder/subtractor with a carry-lookahead adder/subtractor, as shown in figures 32 and 33. One of the powerful features of Max+plus II is the ease with which a module can be modified in a complicated design. In this example, the carry-lookahead adder/subtractor can be incorporated into the design by creating a symbol for the adder/subtractor, and updating the symbol in the overall ALU design by selecting Symbol → Update Symbol → All Symbols in the File. Once this is done, the carry lookahead adder/subtractor is part of the design. The new timing analysis shown in figure 34 reveals a 27 % improvement in the speed of the ALU.



Delay Matrix

Destination

Source	f0	f1	f2	f3	Overflow
op0	9.6ns	9.6ns	9.6ns	6.0ns/13.2ns	9.6ns
op1	6.0ns/9.6ns	9.6ns	9.5ns/9.6ns	6.0ns	9.6ns
x0	6.0ns/9.6ns	9.6ns	9.6ns	9.6ns/13.2ns	9.6ns
x1	6.0ns	9.6ns	9.6ns	9.6ns/13.2ns	9.6ns
x2	6.0ns	9.6ns	6.0ns/9.6ns	9.6ns/13.2ns	9.6ns
x3	6.0ns	9.6ns	9.6ns	6.0ns/13.2ns	9.6ns
y0	6.0ns/9.6ns	9.6ns	9.5ns/9.6ns	9.6ns/13.2ns	9.6ns
y1	6.0ns/9.6ns	6.0ns/9.6ns	9.5ns/9.6ns	9.6ns/13.2ns	9.6ns
y2			9.5ns/9.6ns	13.1ns/13.2ns	9.6ns
y3				13.1ns/13.2ns	9.6ns

Figure 31 – Timing analysis of ALU.

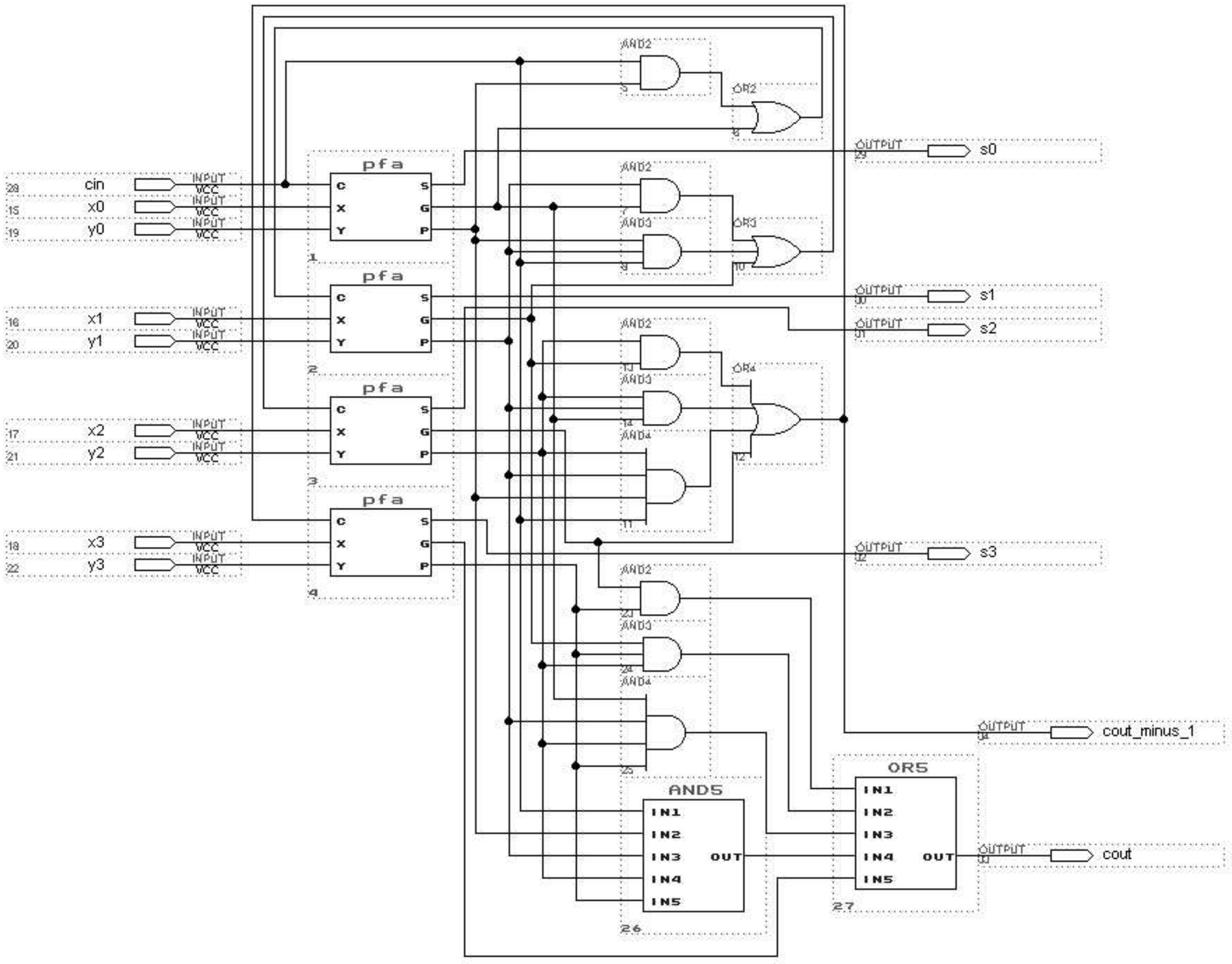


Figure 32 – Carry lookahead adder.

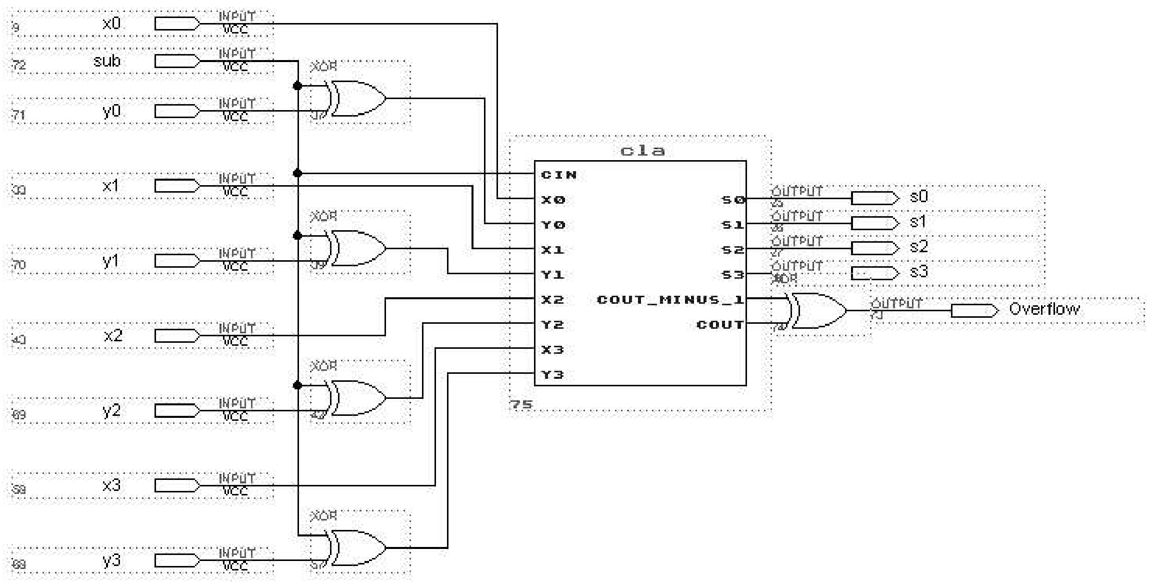
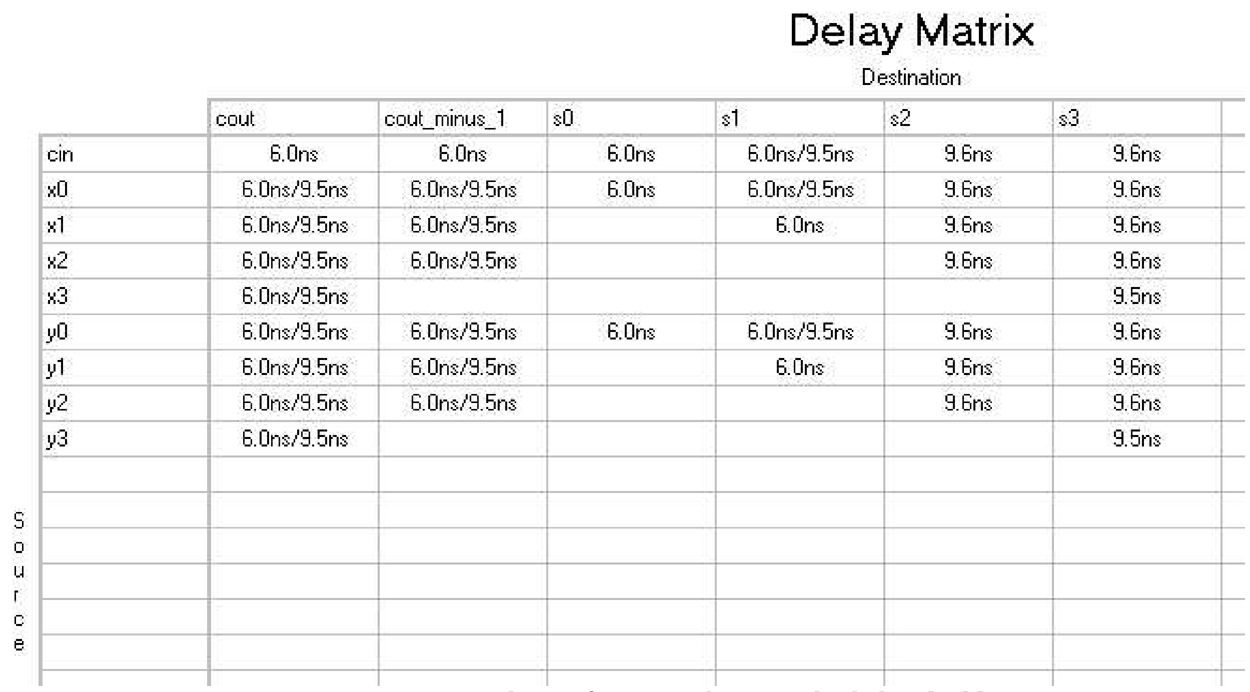


Figure 33 – 4-bit adder/subtractor circuit implemented with a carry-lookahead adder.



Delay Matrix

Destination

Source

	cout	cout_minus_1	s0	s1	s2	s3
cin	6.0ns	6.0ns	6.0ns	6.0ns/9.5ns	9.6ns	9.6ns
x0	6.0ns/9.5ns	6.0ns/9.5ns	6.0ns	6.0ns/9.5ns	9.6ns	9.6ns
x1	6.0ns/9.5ns	6.0ns/9.5ns		6.0ns	9.6ns	9.6ns
x2	6.0ns/9.5ns	6.0ns/9.5ns			9.6ns	9.6ns
x3	6.0ns/9.5ns					9.5ns
y0	6.0ns/9.5ns	6.0ns/9.5ns	6.0ns	6.0ns/9.5ns	9.6ns	9.6ns
y1	6.0ns/9.5ns	6.0ns/9.5ns		6.0ns	9.6ns	9.6ns
y2	6.0ns/9.5ns	6.0ns/9.5ns			9.6ns	9.6ns
y3	6.0ns/9.5ns					9.5ns

Figure 34 – Timing analysis of ALU with a carry lookahead adder.

Finite State Machine Example

In this example, a 2-bit up/down counter will be designed and implemented using Max+plus II. A Moore model will be used in the design. The finite state diagram is shown in figure 35.

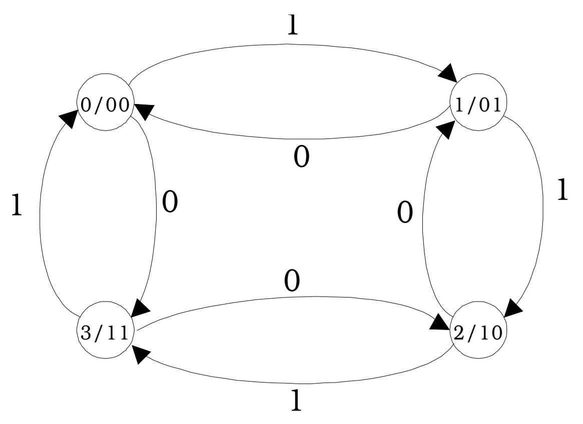


Figure 35 – 2-bit up/down counter.

A block diagram of the counter is shown in figure 36. Note that the output decoder is note depicted. This is because the outputs (Z_1, Z_0) are the same as the current state (Q_1, Q_0). The equations describing the next state decoder are shown in figure 37.

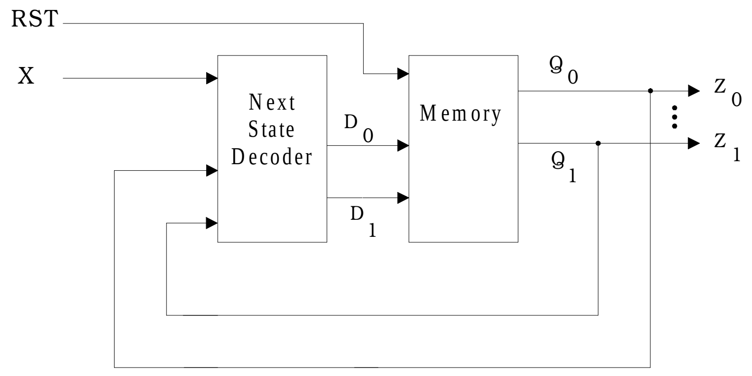


Figure 36 – Block diagram of 2-bit up/down counter.

$$D_0 = Q_0'$$
$$D_1 = X'Q_1'Q_0' + XQ_1Q_0' + X'Q_1Q_0 + XQ_1'Q_0$$

Figure 37 – Next state decoder equations.

Now that the design has been specified at a high level, attention can be given to low level details. We'll start with the memory block. The four states shown in the state diagram require two flip-flops. D-flip-flops with asynchronous reset (RST) will be used. The flip-flop design and simulation are shown in figures 38 and 39 respectively.

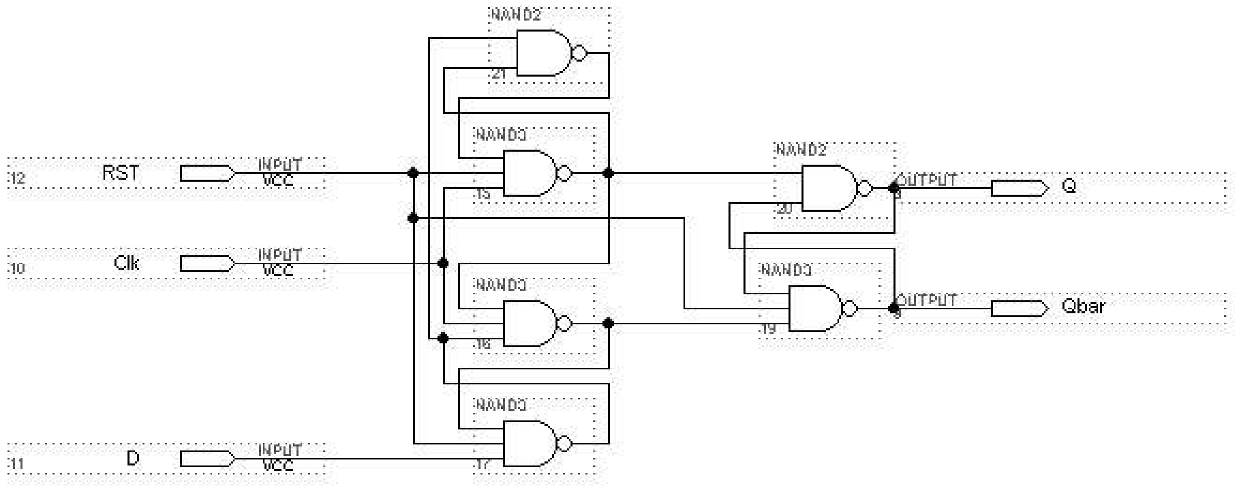


Figure 38 – D-flip-flop with asynchronous reset.

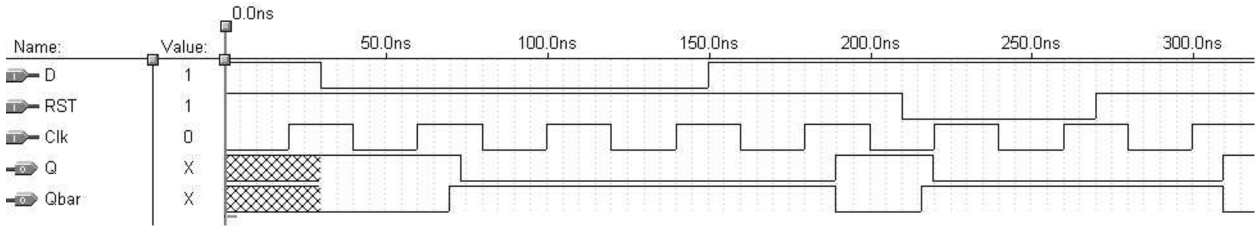


Figure 39 – Simulation of D-flip-flop.

Now that the flip-flops have been designed and tested, the next state decoder must be implemented. The decoder is shown in figure 40, with the associated simulation in figure 41.

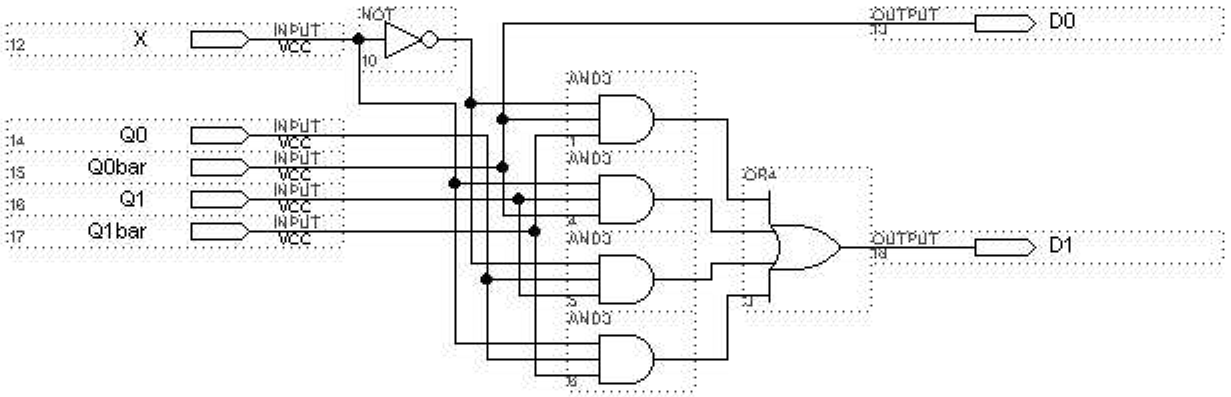


Figure 40 – Next state decoder implementation.

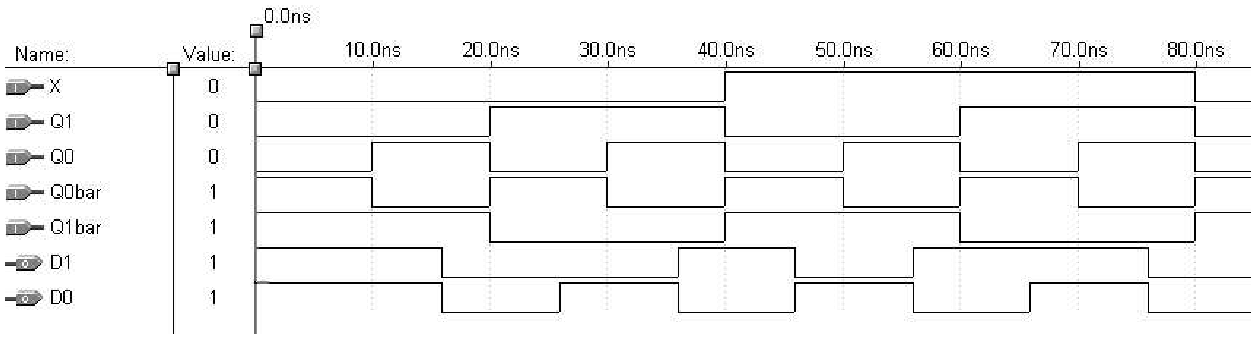


Figure 41 – Simulation of next state decoder.

The next state decoder and flip-flops can now be instantiated in the counter by creating symbols for each of them, inserting the symbols into the design of the counter, and then making the proper connections, as shown in figure 42. Functional verification of the counter is shown in figure 43.

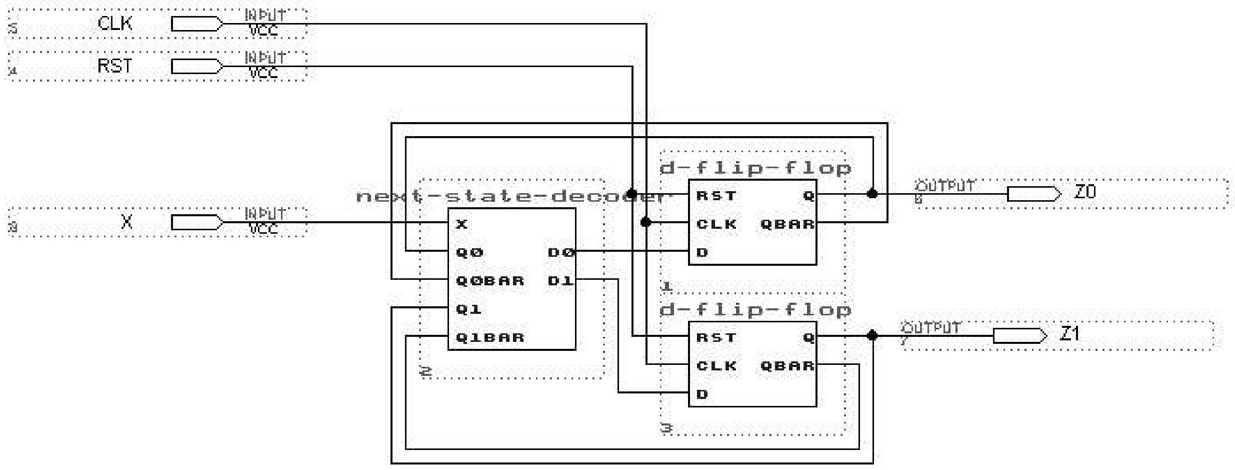


Figure 42 – 2-bit up/down counter.

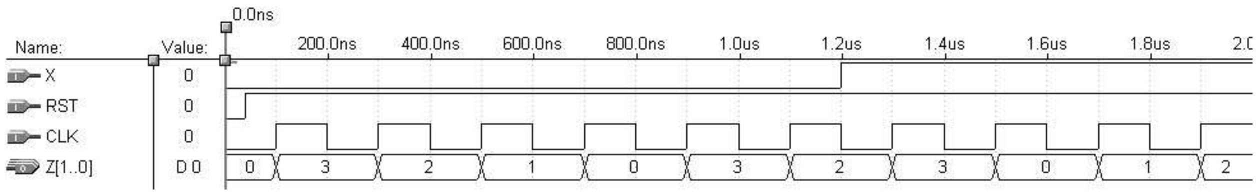


Figure 43 – Functional simulation of counter.

Design Using Off-the-shelf Components

There are various libraries in Max+plus II which allow off the shelf components to be used to implement a design. In this example, TTL 7400 series chips will be used to design a 3-to-8 decoder from a dual 2-to-4 decoder (74139). Select the 74139 from the *mf* library. Using the same library, add an inverter (7404). Add three input terminals from the primitives library (prim) and eight output terminals. Wire the circuit as shown in figure 44. A simulation of the 3-to-8 decoder is shown in figure 45.

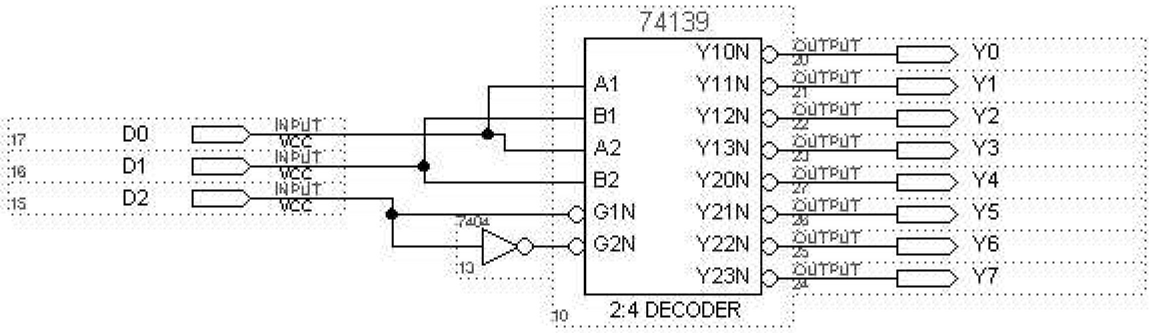


Figure 44 – 3-to-8 decoder implemented with a dual 2-to-4 decoder

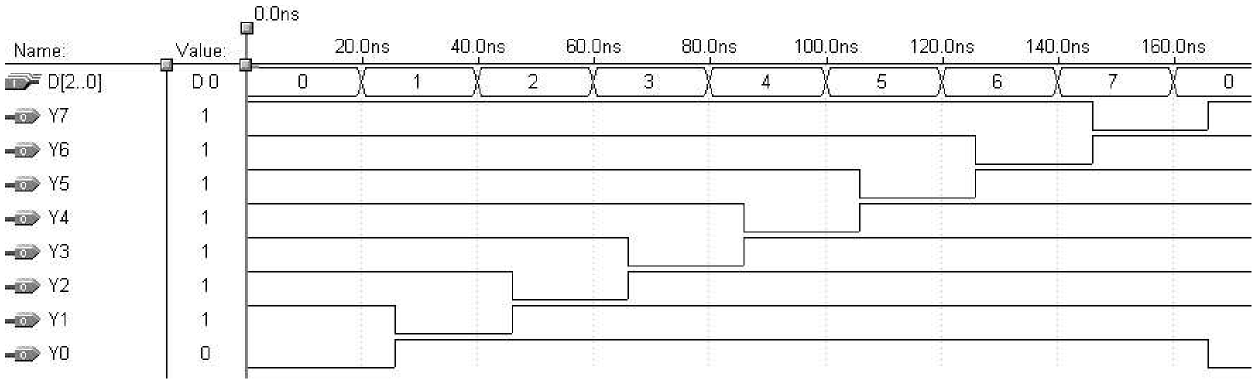


Figure 45 – Simulation of the 3-to-8 decoder