Final Project

E158 Introduction to CMOS VLSI Design

April 11, 2001

8-bit Signal Amplifier with Clipping Detection

Jeremy Liu

Tina Wang

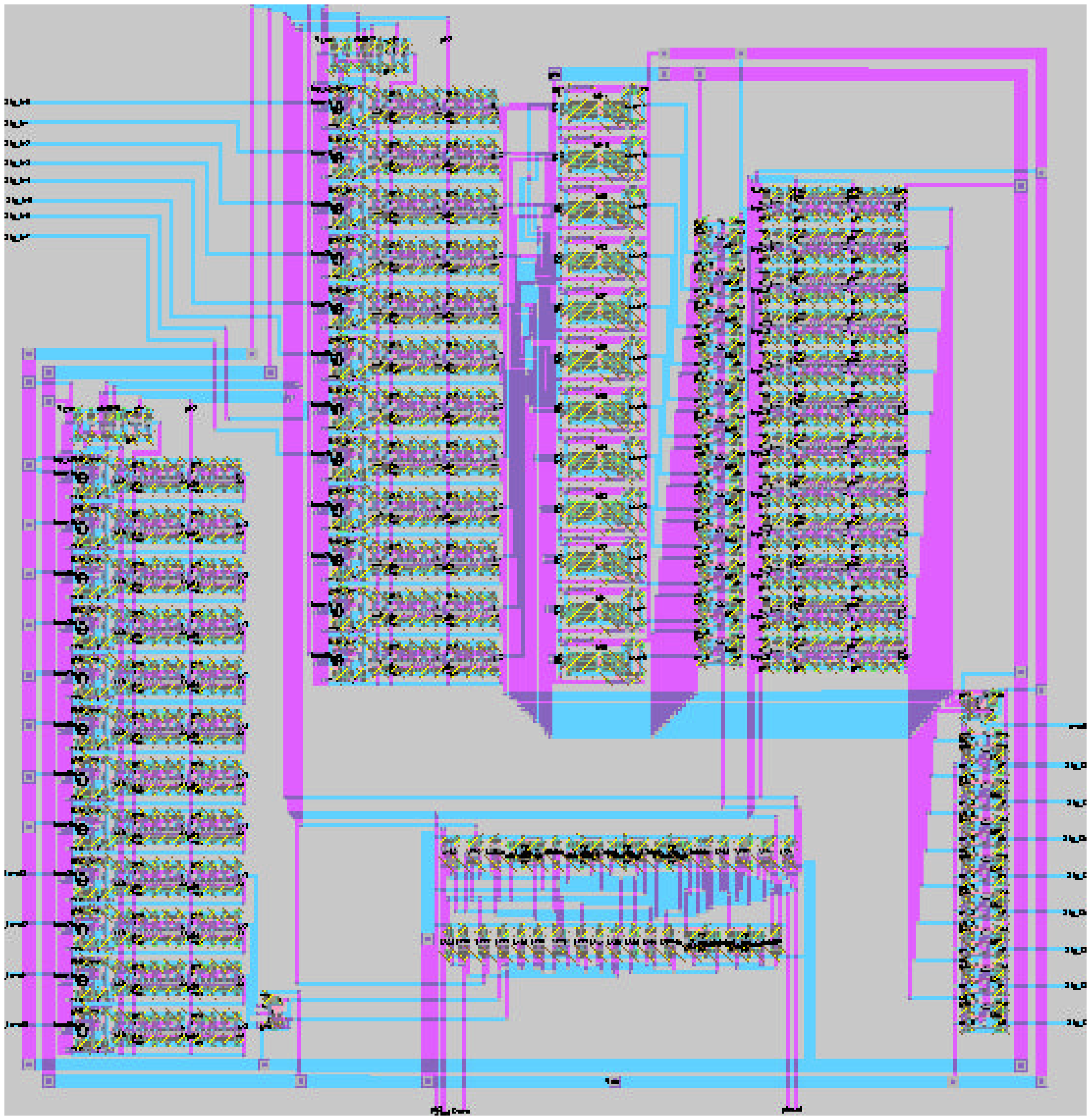


Figure 1: Color Chip Layout

Functional Overview

This is the design for a variable 8-bit signal amplifier with overflow detection. Upon detecting a positive edge on *Start*, the amplifier takes in an 8-bit signal (*Signal_In[7:0]*) and multiplies it by the gain indicated by *Amp_Level[3:0]*. The output is also an 8-bit signal (*Signal_Out[7:0]*) so overflow detection is needed. If the signal overflows, this amplifier chip will clip the output to the maximum possible 8-bit value (8b'1111_1111) and toggle the *Overflow* flag high. When the unit is done amplifying the signal, the *Done* flag will toggle high.

Inputs	**Definitions**
Signal_In[7:0]	8-bit Data signal to be amplified
Amp_Level[3:0]	4-bit Level of amplification (gain)
ph1, ph2	Two-phase clock signals
Start	Starts amplifier on posedge Start signal
Reset	Reset
Outputs	
Signal_Out[7:0]	8-bit Data output of amplified signal
Overflow	High if overflow detected
Done	High when amplifier cycles are complete, also High on Reset.

Table 1: I/O Pins and Definitions

<u>Multiplier</u>

Our amplifier design is based on the following block diagram as discussed in our text. We have scaled it up from a 4x4 multiplier to an 8x4 multiplier and added additional logic for the clipping and overflow detection.

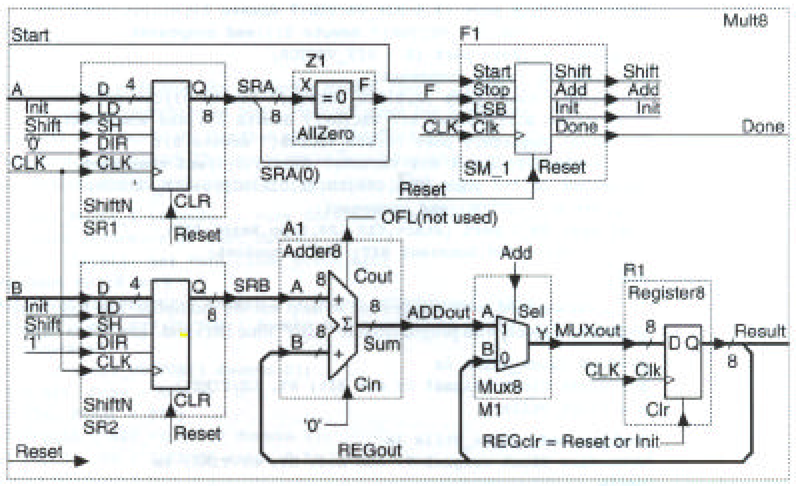


Figure 2: A 4x4 bit multiplier. Source: <u>Application-Specific Integrated Circuits</u> by Michael John Sebastian Smith. 2001. pg. 387.

Signal bus *A* in the above diagram corresponds to our *Amp_Level* and Signal *B* corresponds to our *Signal_In*. In our design we replaced the ShiftN block with a 12-bit shift register. The design works by shifting the multiplier to the right and multiplicand to the left. If the least significant bit of the *Amp_Level* is a 1, then the multiplier will add the value in *Signal_In* to the current value in the result. Otherwise if the least significant bit of *Amp_Level* is 0, then we don't need to add anything.

```
     00011110   Signal_In
     x   0110   Amp_Level
  ---------------------------
            0   1st bit of Amp_Level is 0, Add 0
    00011110    2nd bit is 1, Shift Signal_In left and Add to result
   00011110     3rd bit is 1, Shift Signal_In left again and Add to result
 +       0      4th bit is 0, Add 0
  ---------------------------
     10110100   Signal_Out
```

Figure 3: A Sample Calculation

Since the *Amp_Level* is continuously shifting to the right (with zeros shifting at the left), when the bottom four bits of the Amp Level register are all zero, then we know we are done with the multiplication. The AllZero block in Figure 1 is accomplished using a single NOR4 gate which outputs the signal *Stop*.

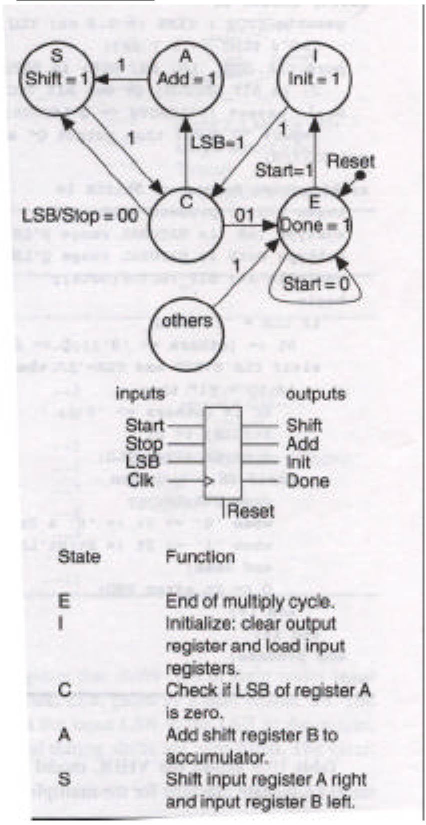
Finally instead of having an 8-bit adder, mux and register, we scaled them all up to 12-bits.

Controller

The SM_1 block in Figure 1 is the controller block for the multiplier. The controller is based off the following state machine. We coded this state machine into Verilog and synthesized it using Synopsis tools for the transistor level layout (see Appendix A for Verilog code).

The state names were changed into longer, more descriptive names.

I	=>	Initialize_state
C	=>	Check_state
A	=>	Add_state
S	=>	Shift_state
E	=>	End_state



State	Function
E	End of multiply cycle.
I	Initialize: clear output register and load input registers.
C	Check if LSB of register A is zero.
A	Add shift register B to accumulator.
S	Shift input register A right and input register B left.

Figure 4: Controller state machine.
Source: Application-Specific Integrated Circuits by Michael John Sebastian Smith. 2001. pg. 386.

The state machine starts in state End_state with *Done* = 1. It will remain in that state until the user starts the amplification cycles by toggling the *Start* signal high.

The initialize state sends Init high for one cycle, then continues on to the Check_state cycle.

In the Check_state cycle, if the current least significant bit of Amp Level register is 1, then we need to do an Add before the next Shift cycle. Otherwise, unless the Stop flag is high, the next state goes directly to the Shift_state. If the Stop flag is high, then we are done with the calculation and we end in End_state. Any Reset signal will also set the state to End_state.

Clipping

The final output from the multiplier is sent into an 8-bit mux for clipping. If the any of the top four bits of the product register are equal to 1, then the result is greater than 8b'1111_1111 and we have an overflow. We are not worried about the 13th carryout bit because the largest possible result we can have would be 8b'1111_1111 multiplied by 4b'1111 which equals 12b'1110_1111_0001.

Therefore the OR4 of the top four bits of the product register is the *Overflow* signal and also acts as the control signal of the 8-bit mux. If there is not an overflow, then the 8-bit mux outputs the bottom 8 bits of the product register. If there is an overflow, then the 8-bit mux will output 8b'1111_1111.

Timing Issues

We used a two-phase clock in our design. The clocked components were the controller, the product register and the two shift registers. The registers were all based on two stage latches, with the first being clocked by ph1 and the second by ph2. This meant that the inputs of the registers were taken at the rising edge of ph1, but that the outputs of the registers did not appear until the rising edge of ph2. The outputs were thus stable_1, which was necessary for the shift registers, which needed a stable_1 signal at the input. The controller signals were clocked by ph2, so that they would also be stable_1, and we could use those signals to qualify ph1 at the registers.

Chip Floorplan

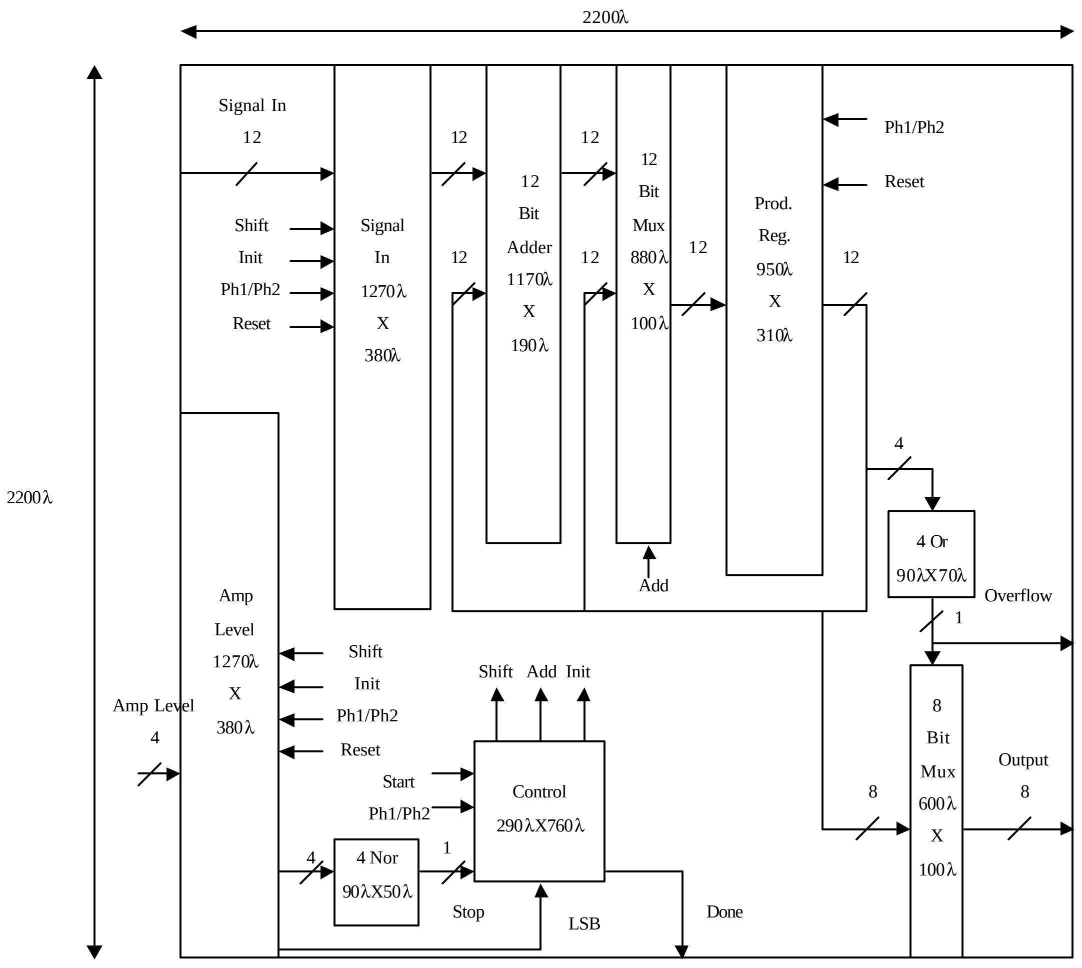


Figure 5: Final Chip Floorplan

Area and Design Time

Leaf Cells	**Est. AREA (h x w)**	**Actual AREA**	**DESIGN TIME**
nor4	$80\lambda \times 120\lambda = 9600\lambda^2$	$70\lambda \times 50\lambda = 3500\lambda^2$	1 hour
std_inv	$60\lambda \times 20\lambda = 1200\lambda^2$	$1200\lambda^2$	given
tri	$80\lambda \times 30\lambda = 2400\lambda^2$	$2400\lambda^2$	given
fulladder	$80\lambda \times 170\lambda = 13600\lambda^2$	$13600\lambda^2$	given
nor3	$60\lambda \times 40\lambda = 2400\lambda^2$	$2400\lambda^2$	given
nand2	$60\lambda \times 30\lambda = 1800\lambda^2$	$1800\lambda^2$	given
Components			
12 Bit Shift Register	$1680\lambda \times 420\lambda = 705600\lambda^2$	$1270\lambda \times 380\lambda = 482600\lambda^2$	12 hours
12 Bit Adder	$1080\lambda \times 170\lambda = 183600\lambda^2$	$1170\lambda \times 190\lambda = 222300\lambda^2$	5 hours
12 Bit Mux	$1080\lambda \times 60\lambda = 64800\lambda^2$	$880\lambda \times 100\lambda = 88000\lambda^2$	3 hours
12 Bit Register	$1200\lambda \times 240\lambda = 288000\lambda^2$	$950\lambda \times 310\ \lambda = 294500\lambda^2$	8 hours
NOR4	$80\lambda \times 120\lambda = 9600\lambda^2$	$90\lambda \times 50\lambda = 4500\lambda^2$	1 hour
Control	$<< 440\lambda \times 1320\lambda$	$290\lambda \times 760\lambda = 220400\lambda^2$	15 hours
OR4	$80\lambda \times 100\lambda = 8000\lambda^2$	$90\lambda \times 70\lambda = 6300\lambda^2$	1 hour
8 Bit Mux	$720\lambda \times 60\lambda = 43200\lambda^2$	$600\lambda \times 100\lambda = 60000\lambda^2$	2 hours
Final Layout			20 hours
TOTAL AREA	$< 2200\lambda \times 2200\lambda$		68 hours

Table 2: Area and Design Time Data

total est area (not including controller): 1,302,800
total actual area (not including controller): 1,158,200

Most of our estimates were pretty close to our actual layout area. We tended to slightly underestimate most component sizes. However, our total estimated area was still larger than the actual component area.

The biggest difference between estimate and actual area was in the 12-bit Shift Register. We originally estimated the height of one shift register to be 140λ. During the design process, we completely redesigned the shift register (mainly to include NAND's instead of MUX2's) and also came up with less lines running horizontally between the registers than we had originally anticipated. The final height of one shift register became approximately 100 lambda tall with only one signal line running between the registers. The area gained from this size reduction was enough to compensate for our other small area underestimates and still have a smaller total actual size than estimate size.

Verification Results

Leaf Cells	**DRC**	**ERC**	**NCC**
nor4	pass	pass	pass
std_inv	pass	pass	pass
tri	pass	pass	pass
fulladder	pass	pass	pass
nor3	pass	pass	pass
nand2	pass	pass	pass
Components			
12 Bit Shift Register	pass	pass	pass
12 Bit Adder	pass	pass	pass
12 Bit Mux	pass	pass	pass
12 Bit Register	pass	pass	pass
NOR4	pass	pass	pass
Control	pass	pass	pass
OR4	pass	pass	pass
8 Bit Mux	pass	pass	pass
8-bit Signal Amplifier with Clipping	pass	pass	pass

Table 3: Verification Results

Chip Pinout

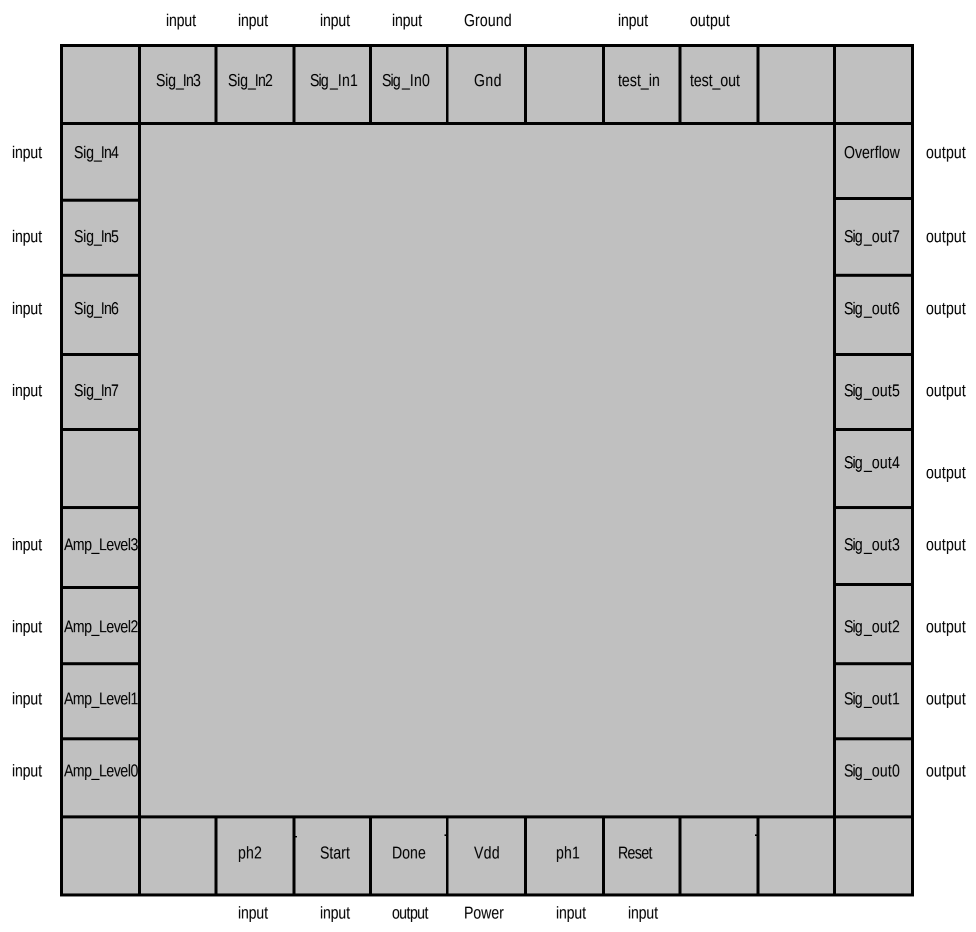


Figure 6: Pad Layout

Test Plan

Upon getting the chip back, the first step in testing this chip would be to test the pins attached to the input and output terminals of an INV gate placed in the extra area of the chip. If the output signal is the inverted input then we know the pins on the chip are active.

The amplifier should not perform any calculations until a posedge signal is detected on *Start*. With a constant clock signal, after a reset to clear all registers, all outputs except for *Done* should be low.

The next step is testing simple operation. The following test vectors can be used to check proper amplification. Pulse *Start* when the data inputs are ready and then wait for the *Done* signal. You should also reset the chip between each test to ensure that the registers are empty before testing.

	Signal_In	**Amp_Level**	**Signal_Out**	**Overflow**
1.	0000 0000	0000	0000 0000	0
2.	0000 0110	0000	0000 0000	0
3.	0000 0110	0001	0000 0110	0
4.	0000 0110	0010	0000 1100	0
5.	0000 1010	0110	0011 1100	0
6.	0010 1101	0011	1000 0111	0
7.	0001 0001	1111	1111 1111	0
8.	1011 0110	0100	1111 1111	1
9.	0001 0001	1001	1001 1001	0
10.	0100 1110	1101	1111 1111	1

Table 4: Test Vectors

Test vector 7 should result in an answer that is the maximum possible value without triggering the overflow detector. Test vectors 8 and 10 should cause an overflow and is used to test the clipping detection of the amplifier. When the result of the amplification exceeds the range of the 8-bit output, the output should be set to the full-scale value and *Overflow* should be true.

Appendix A: Verilog Code for Controller

```
// Tina Wang
// cewang@hmc.edu
// March 20, 2001
// E158: Intro to CMOS VLSI Design

// This is the control module for a 4x8 bit multiplier used in
// an 8-bit signal amplifier. The multiplier multiplies the two numbers
// by performing a series of number shifts and additions. However, since
// this is binary multiplication, we either add the shifted multiplicand
// or we add nothing.

// This Verilog code is based on a state machine and VHDL code from
// the text Application-Specific Integrated Circuits by Michael John
// Sebastian Smith, pg. 386-387.

module SM_2(start, ph1, ph2, LSB, stop, reset,
            init, shift, add, done);

       input         start;       // toggle high to start amplifier
       input         stop;        // Stop=NOR4 of the bottom four bits of the
                                  // Amp level register
                                  //    if stop=1, then Amplifier is done
                                  //     multiplying
       input         LSB;         // The LSB of the Amp Level Register.
       input         reset;
       input         ph1, ph2;    // clocks

       output        init;        // clears product registers and reads in
                                  //     new data into the input registers
       output        shift;       // control signal to shift registers
       output        add;         // control signal to a mux to chose
                                  //     result = result + signal in (shifted)
                                  //     instead of result = result
       output        done;        // output to user signaling end of amplifier
                                  //     cycles

       // multicycle state machine state definitions
       parameter     Initialize_state = 3'b000;
       parameter     Check_state = 3'b001;
       parameter     Add_state = 3'b010;
       parameter     Shift_state = 3'b011;
       parameter     End_state = 3'b100;

       // parameters
       reg                  [2:0]  nextstate_s2;
       reg                  [2:0]  state_s1, state_s2;
       reg                  shift, init, add, done;

       // state register
       always @(ph1 or nextstate_s2)     // master latch
              if (ph1) state_s1 <= nextstate_s2;
       always @(ph2 or state_s1)         // slave latch
              if (ph2) state_s2 <= state_s1;

       // next state logic
       // state transition diagram given in Figure 2 of E158 Final Report
       always @(state_s2 or LSB or start or stop or reset)
              if (reset) nextstate_s2 <= End_state;   // synchronous reset
              else case (state_s2)
                     Initialize_state: nextstate_s2 <= Check_state;
                     Check_state: if (LSB) nextstate_s2 <= Add_state;
```

```
                        else if (!stop) nextstate_s2 <= Shift_state;
                        else nextstate_s2 <= End_state;
                        // If LSB=1, then result = result+Signal_In(shifted)
                        // Else (!Stop) just shift Signal_In and Amp_Level
                        // Else Stop, skip to End_state
                Add_state: nextstate_s2 <= Shift_state;
                Shift_state: nextstate_s2 <= Check_state;
                End_state: if (start) nextstate_s2 <= Initialize_state;
                        else nextstate_s2 <= End_state;
                default: nextstate_s2 <= End_state;
                endcase

        always @(state_s2)
            begin
                // provide default values for signals not specified
                init <= 0;
                add <= 0;
                shift <= 0;
                done <= 0;

                // specify outputs according to Fig. 2 of E158 Final Report
                // each value only stays high for one cycle because
                // of the default values
                case (state_s2)
                        Initialize_state: init <= 1;
                        Add_state: add <= 1;
                        Shift_state: shift <= 1;
                        End_state: done <= 1;
                        default: begin
                                end
                endcase
            end
endmodule
```

Appendix B: Simulation Waveforms and Results

Simulation Results

While designing, we simulated each individual cell and component as they were completed, first for the schematics and then for the layout.

For the schematic simulations, all of components simulated correctly except for the controller. Therefore we simulated the circuit without the controller, applying signals as we would expect to see output from the controller. Eventually both parts of the design were working as expected, and then we integrated the controller into the design and did a final simulation.

For the layout simulations, all of the components simulated correctly, including the controller. Upon wiring all the components together in the final solution, the final design simulated correctly so there was no need to simulate the circuit separate from the controller.

Some sample waveforms are included in this appendix. On the controller simulation from the schematic simulations, you can see that *Reset* will send all the control signals except *Done* to low. Upon a posedge from *Start*, *Init* will toggle on and then off for a cycle. Then you should see a series of pulses on *Shift* or *Add* and *Shift* as the multiplier cycles through. Since the controller is not connected to the rest of the circuit, we must set the values for *LSB* and *Stop* by hand. Normally we would expect these values as outputs from the logic in the circuit. Upon detecting a *Stop* signal, all values except *Done* will go low and remain in that state until a *Start* signal is detected.

In the final simulation, we are only worried about the top level inputs and outputs. We put in a series of different combinations to test if the amplifier works or not. There are two simulation waveforms that correspond to the test vectors given in Table 4. We reset before each calculation. Though the Signal_Out data changes during the calculation, we are not concerned with that because the final answer is read from Signal_Out on the posedge of *Done.*

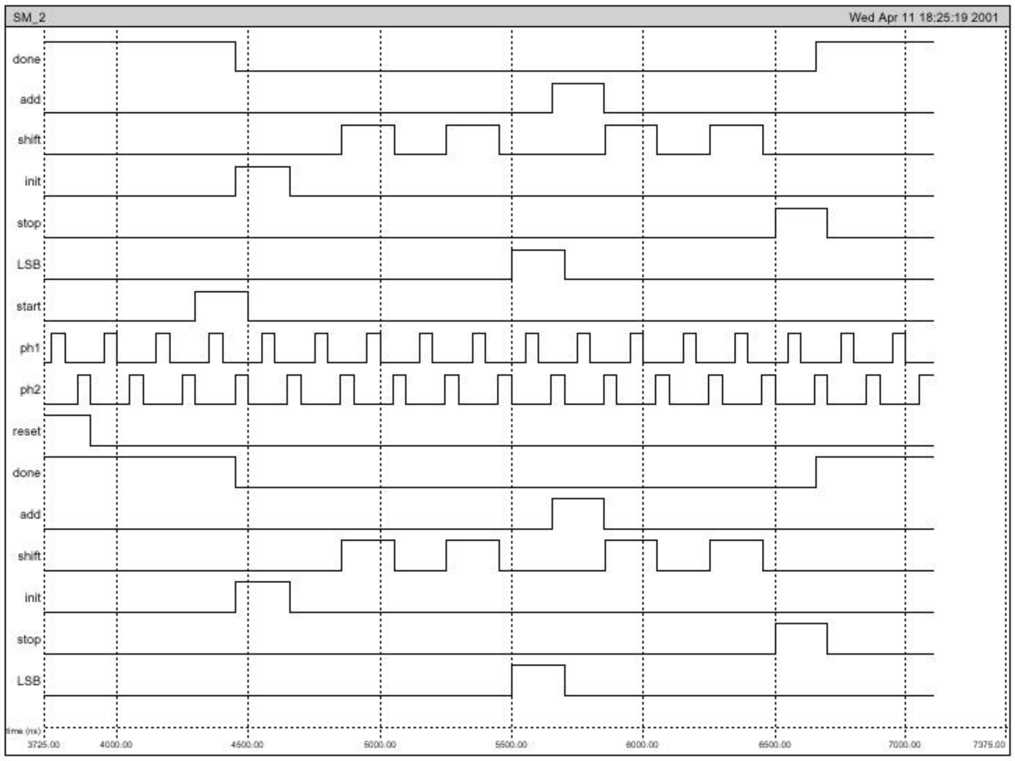


Figure 7: Controller simulation

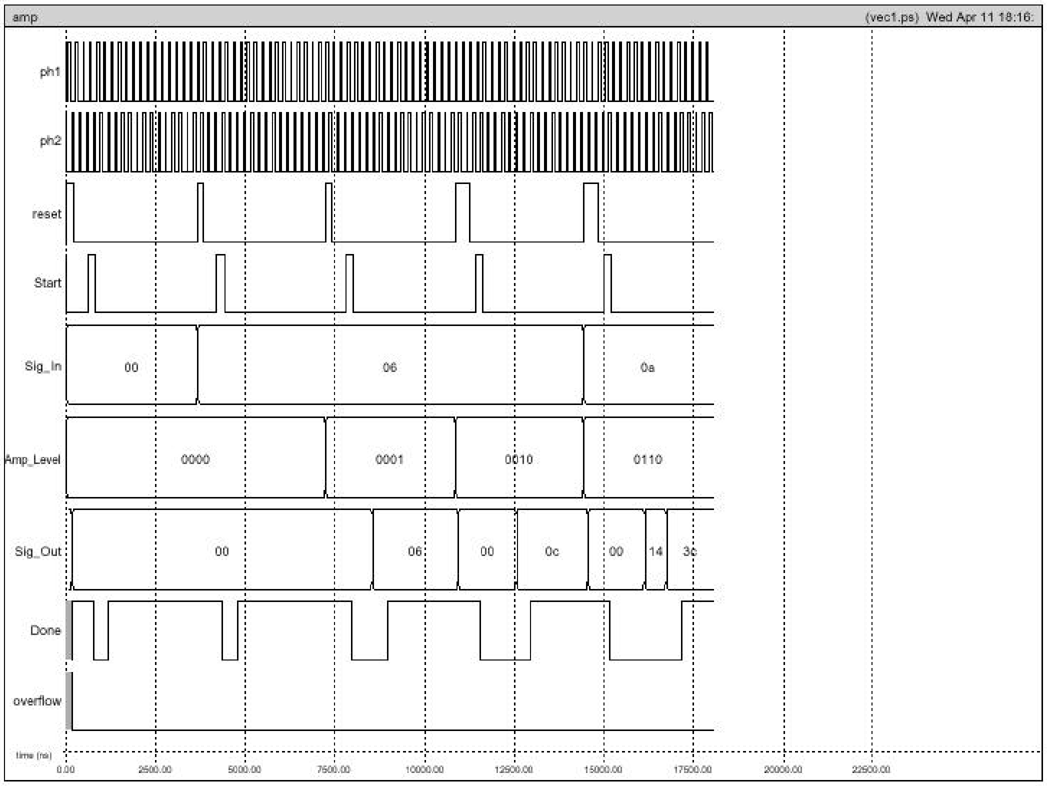


Figure 8: Final simulation (test vectors 1-5)

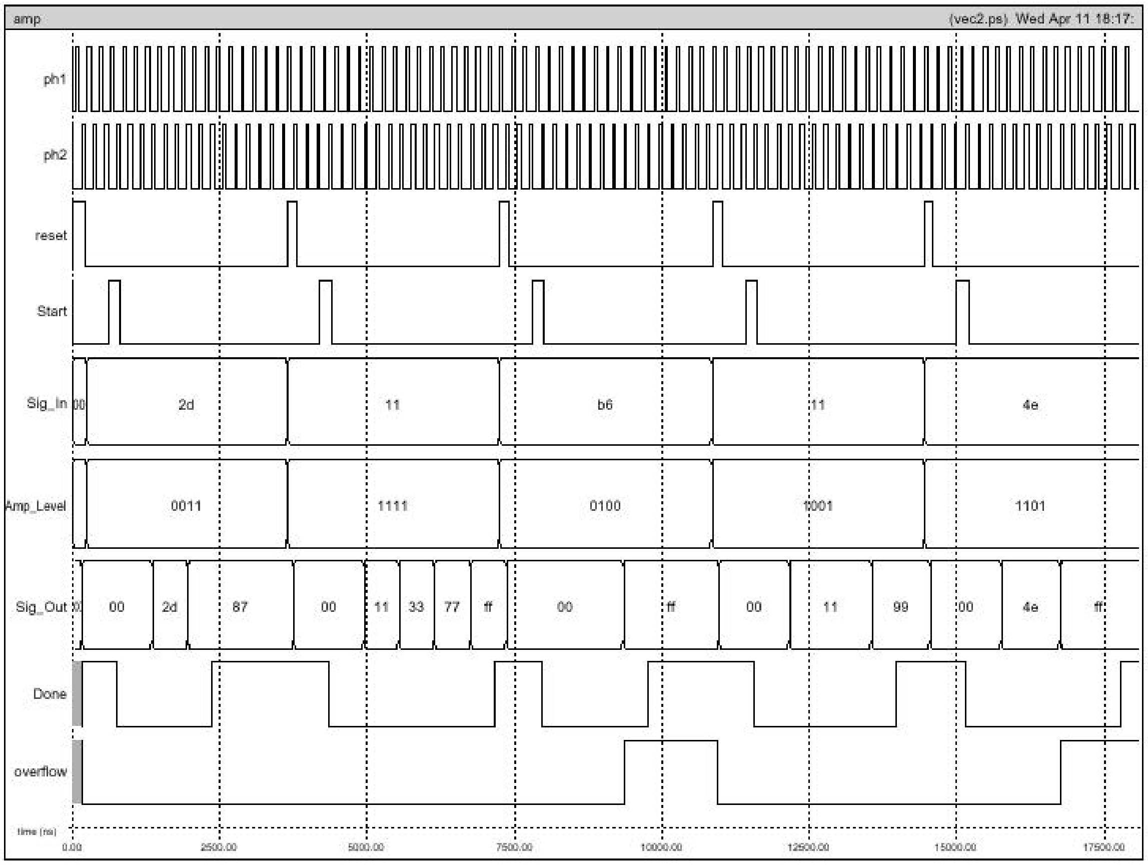


Figure 9: Final simulation (test vectors 6-10)

Appendix C: Schematics

The following pages contain the complete schematics for our design.

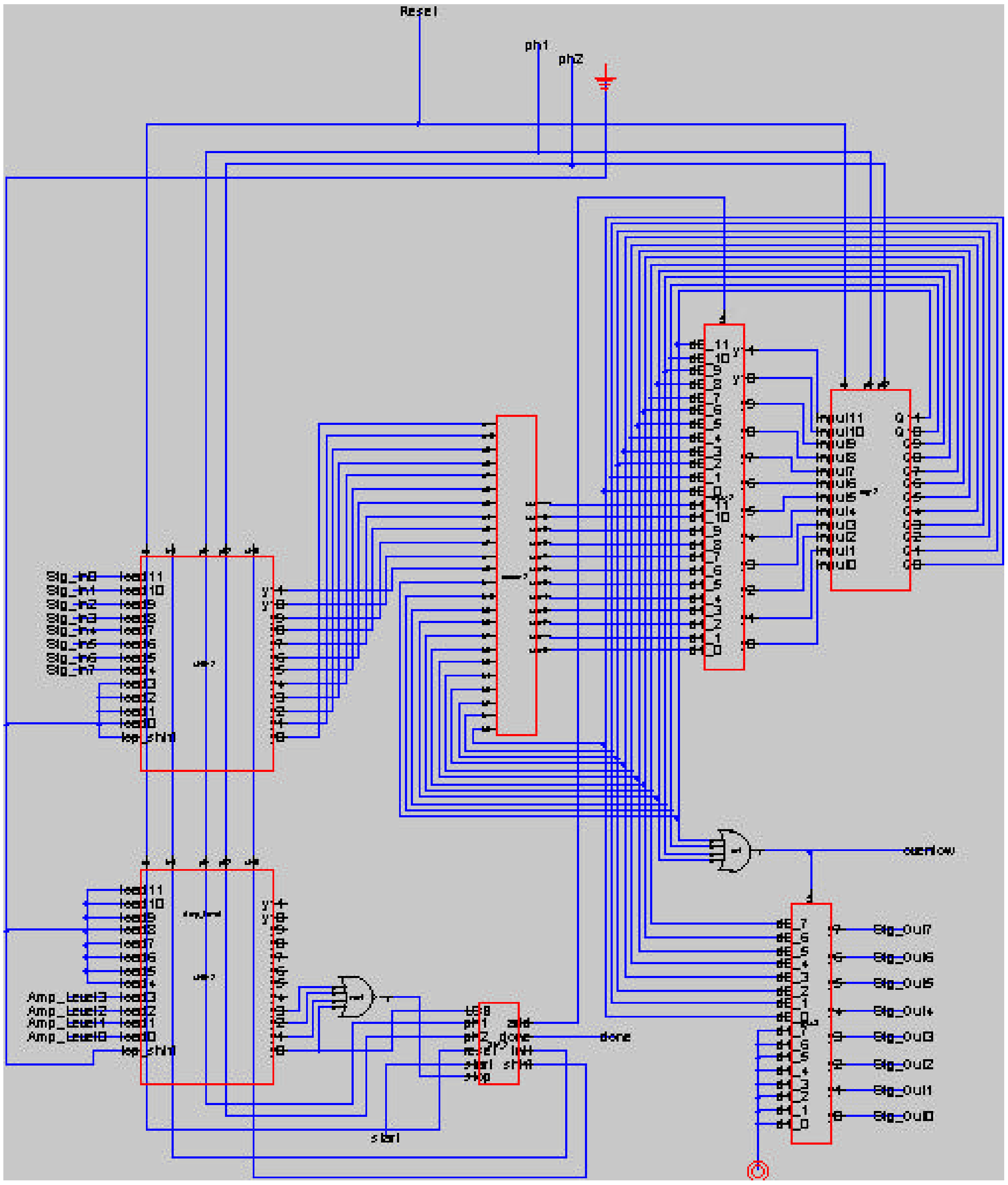


Figure 10: 8-bit signal amplifier with clipping

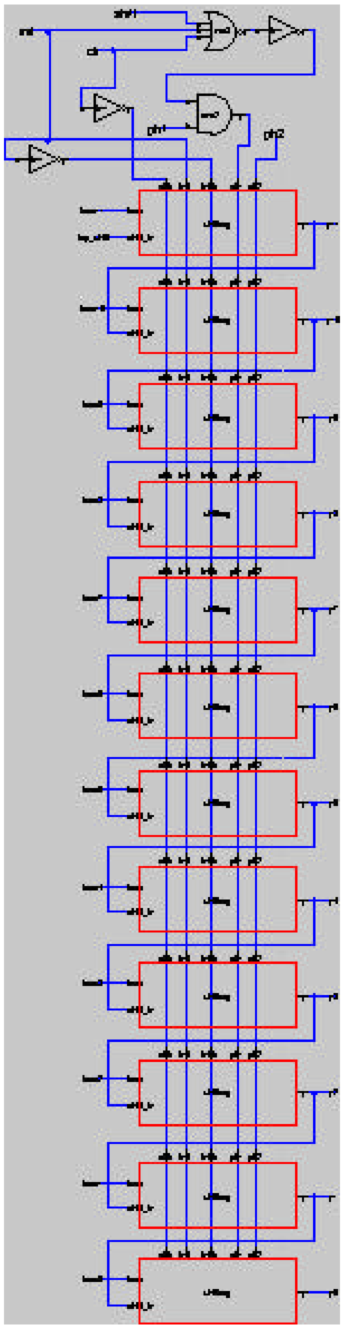


Figure 11: Shift12

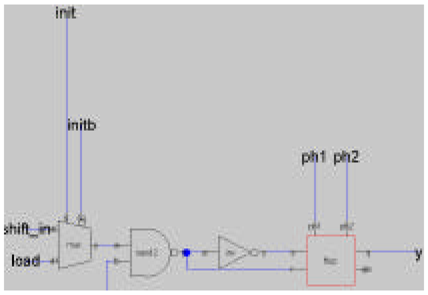


Figure 12: ShiftReg

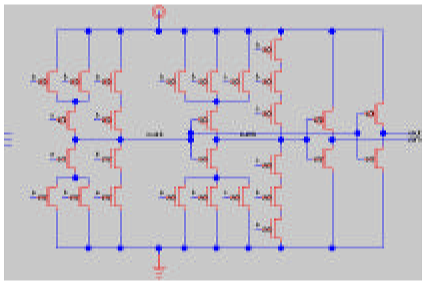


Figure 13: fulladder

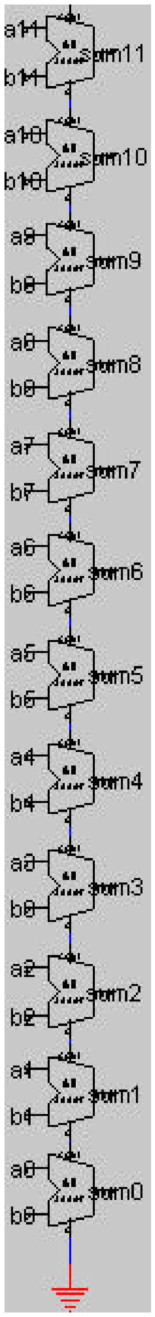


Figure 14: Adder12

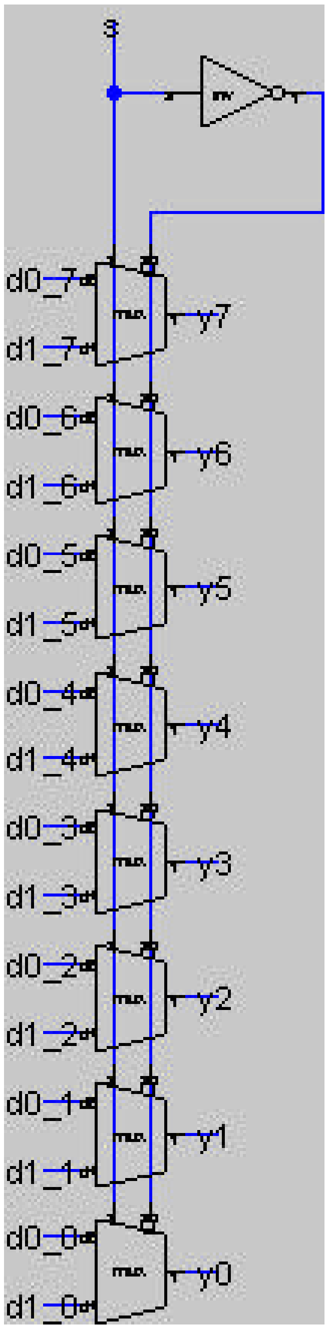


Figure 15: mux8

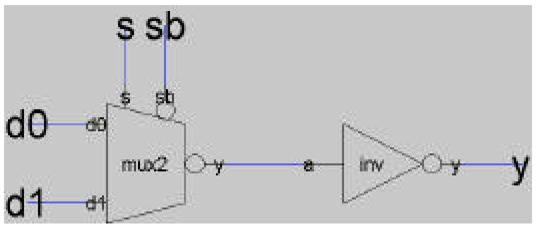


Figure 16: mux

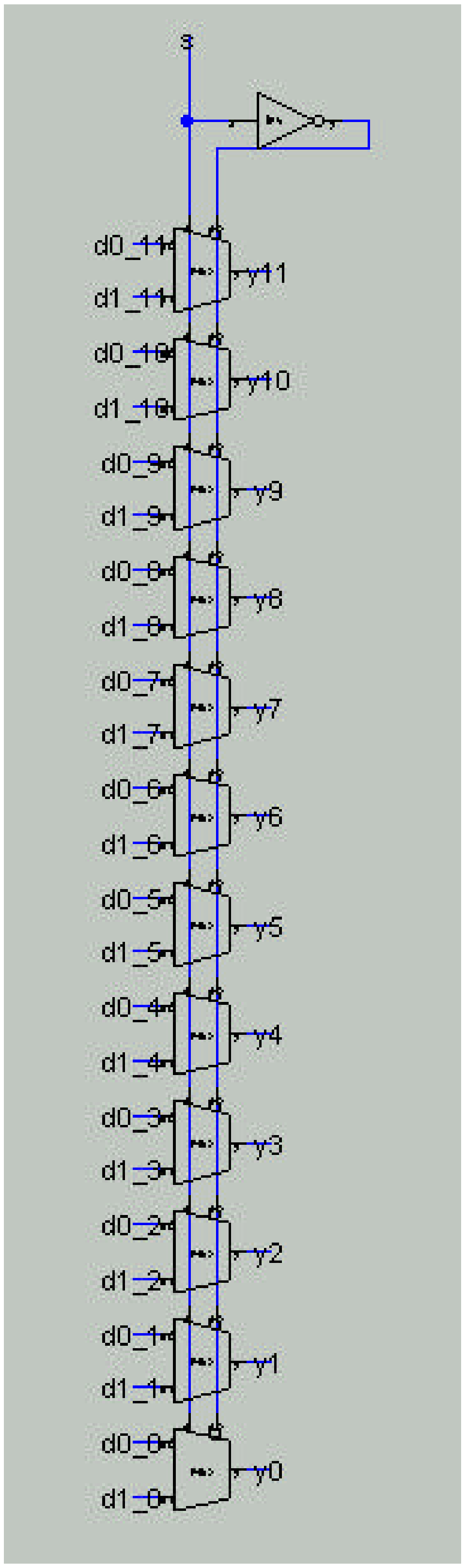


Figure 17: mux12

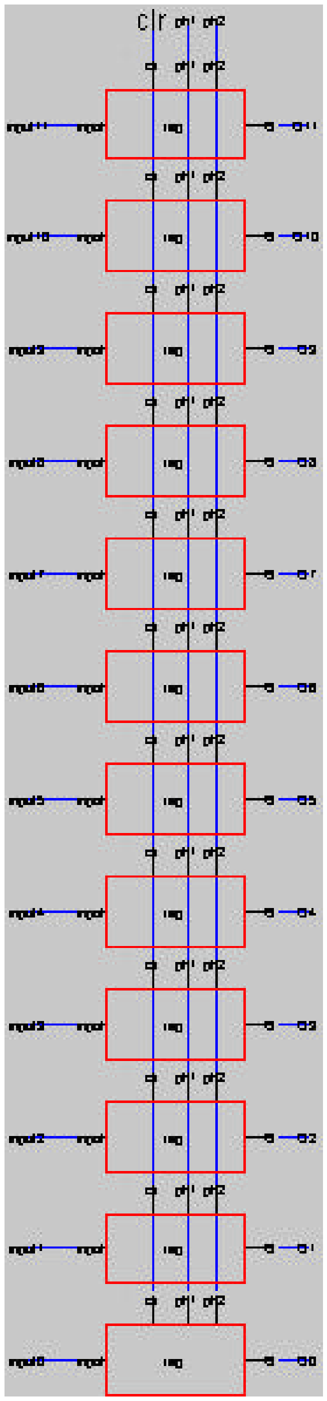


Figure 18: reg12

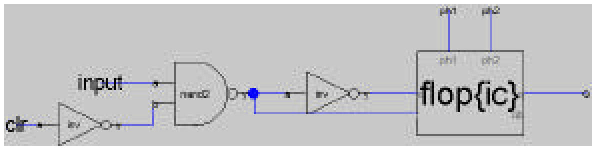


Figure 19:reg

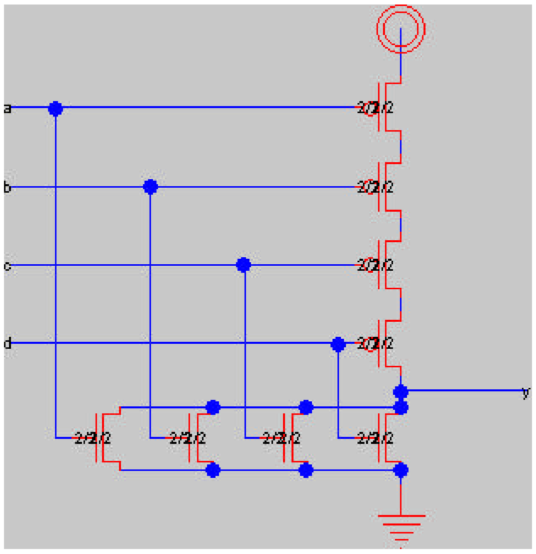


Figure 20: NOR4

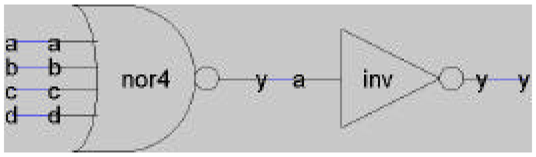


Figure 21: OR4

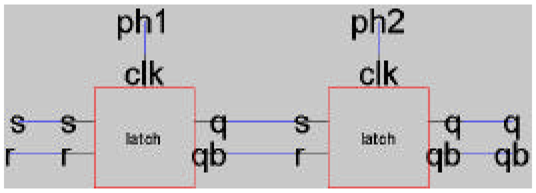


Figure 22: flop

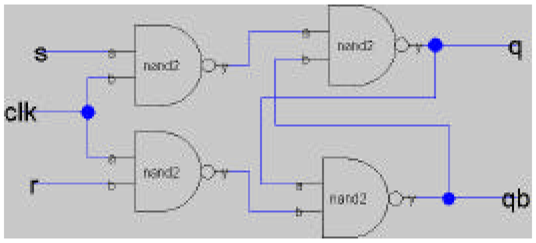


Figure 23: latch

Appendix D: Layouts

The following pages contain the complete layouts for our design.

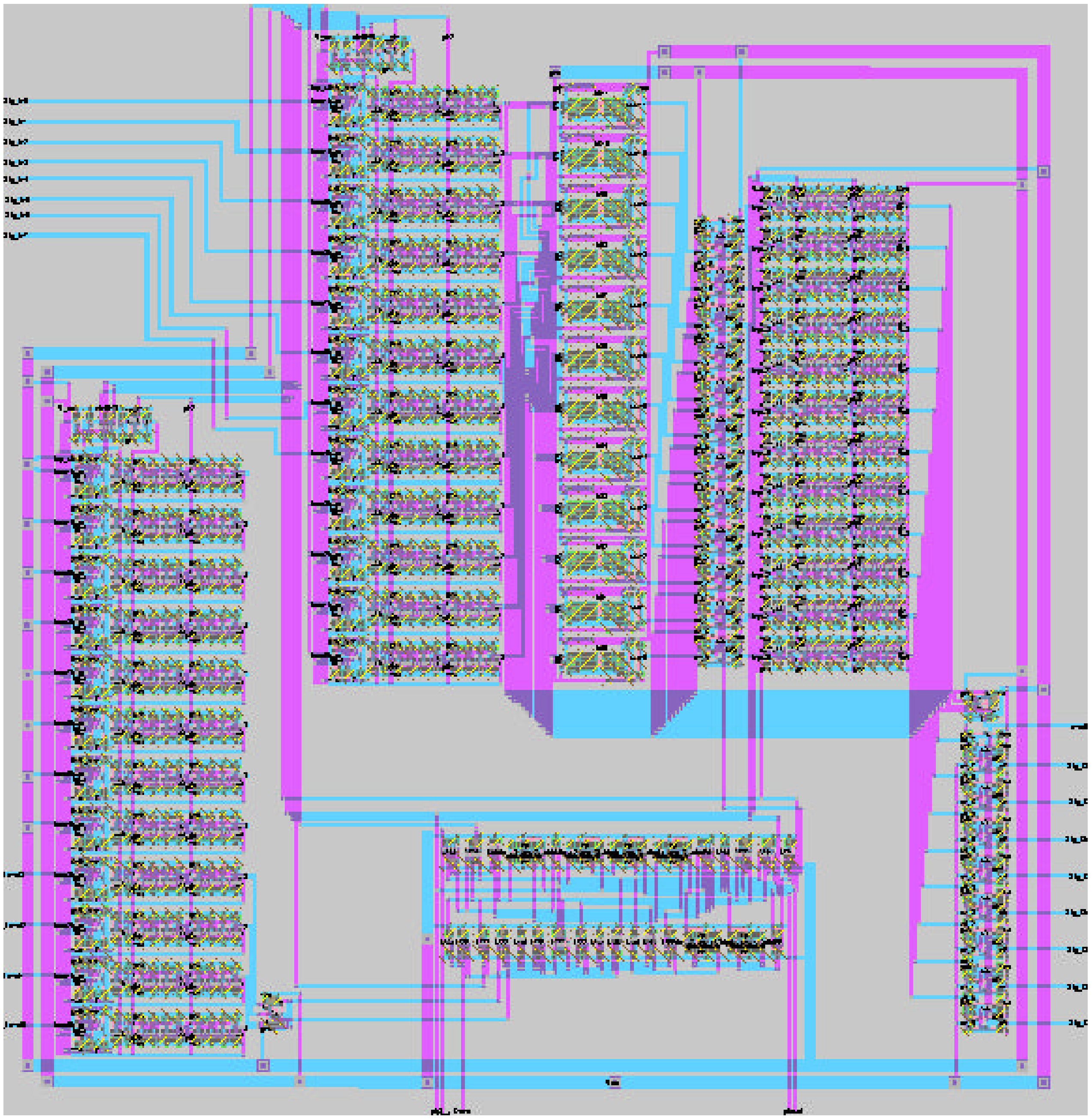


Figure 24: 8-bit signal amplifier with clipping

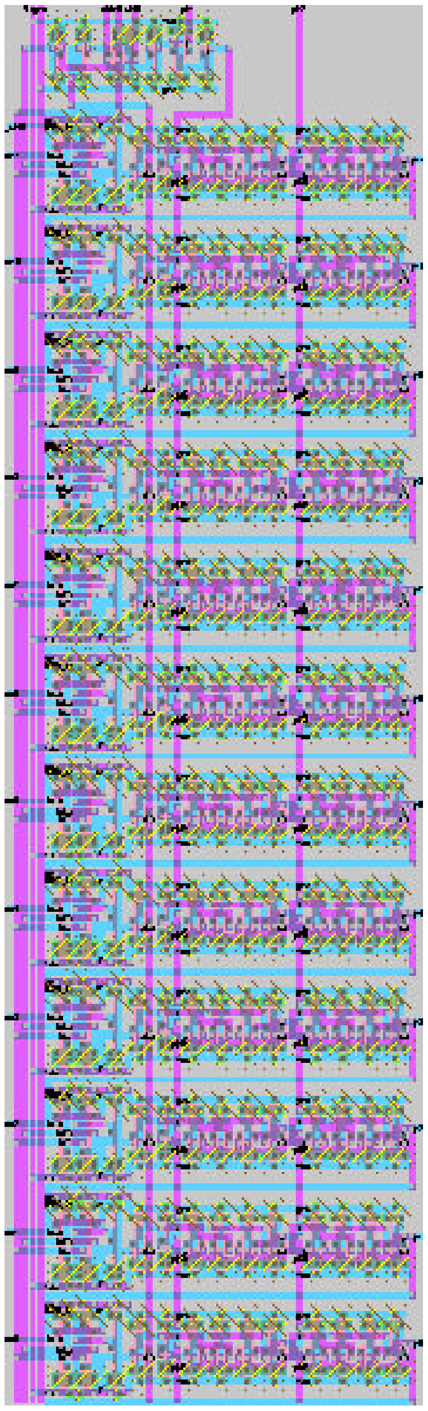


Figure 25: shift12

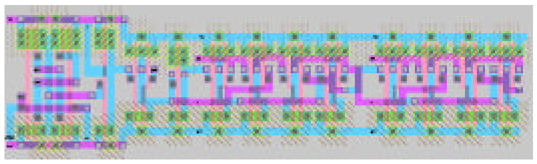


Figure 26: shiftreg

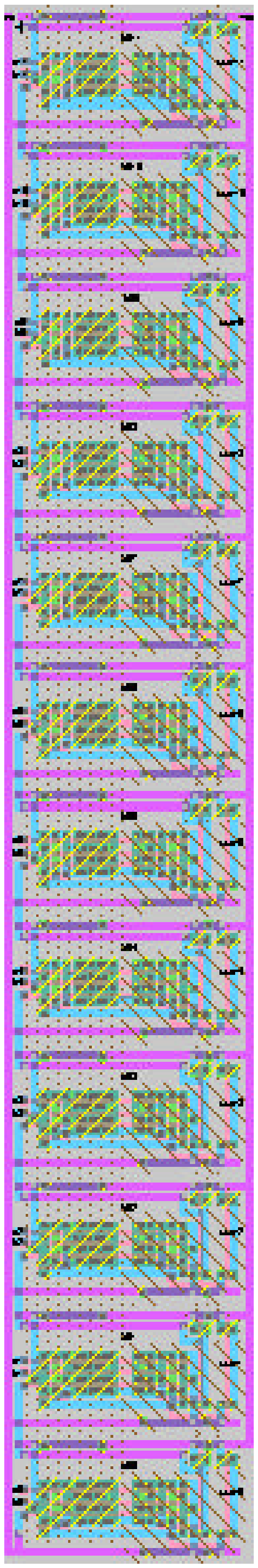


Figure 27: adder12

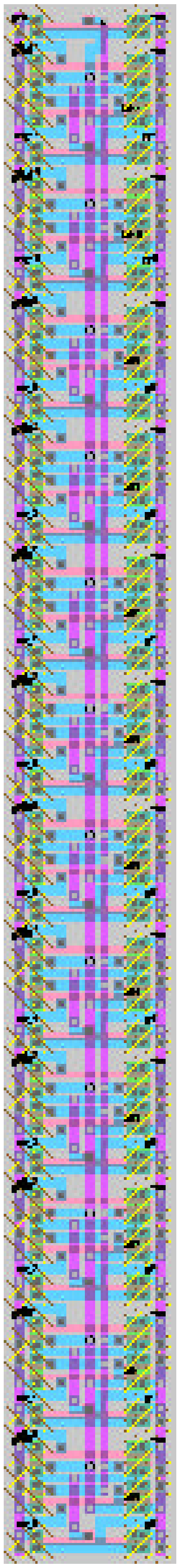


Figure 28: mux12

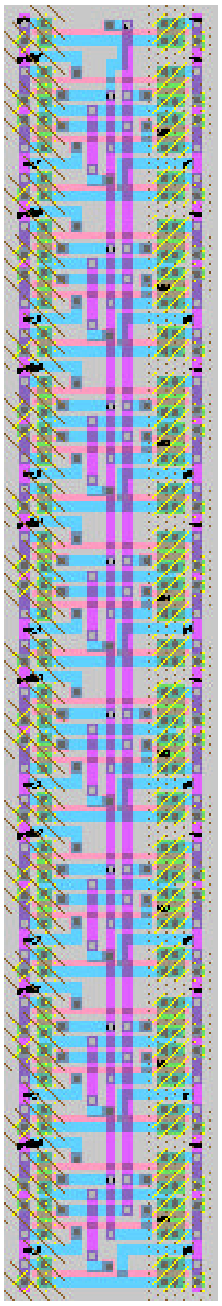


Figure 29: mux8

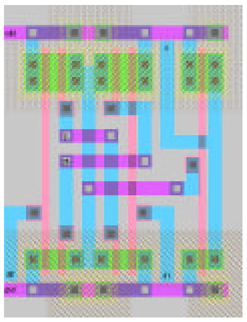


Figure 30: mux

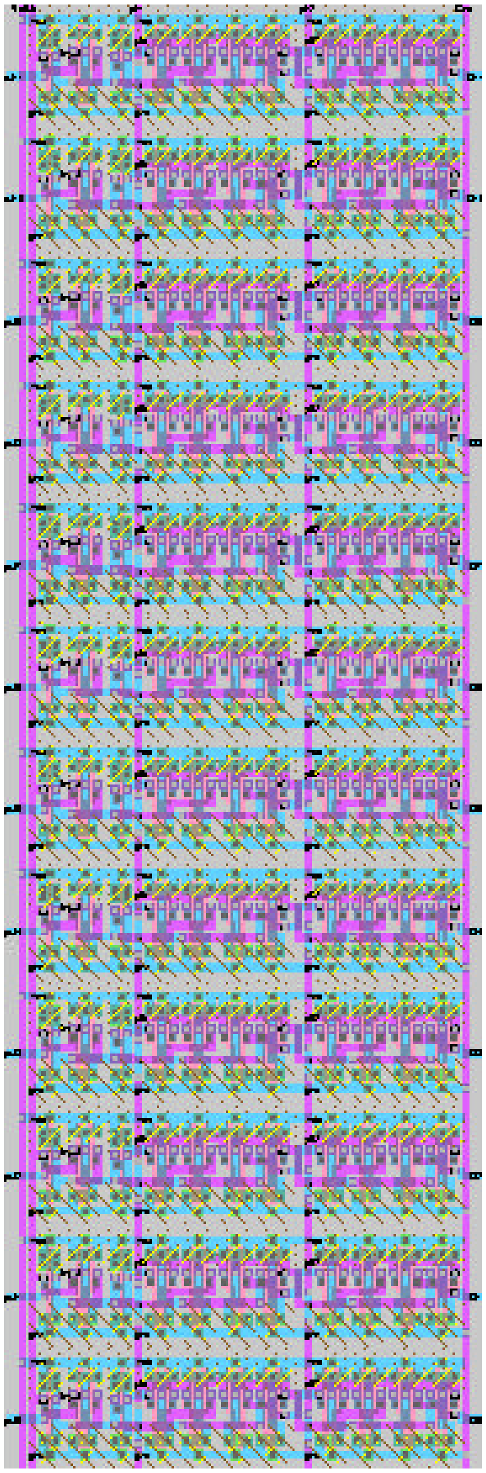


Figure 31: reg12

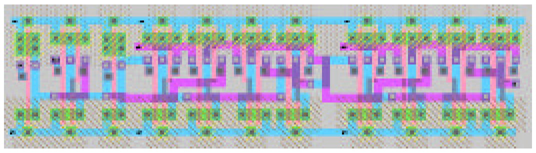


Figure 32: reg

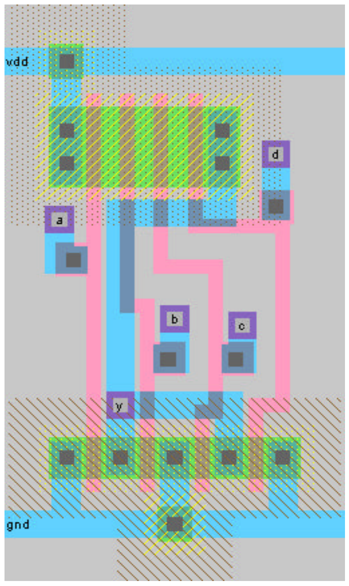


Figure 33: NOR4

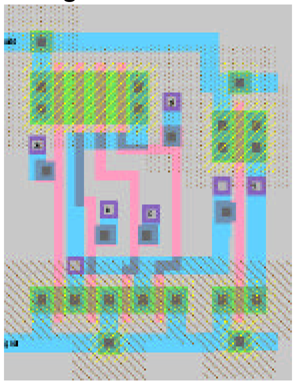


Figure 34: OR4

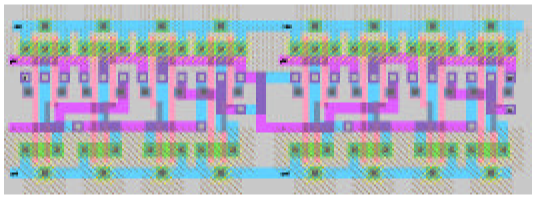


Figure 35: flop

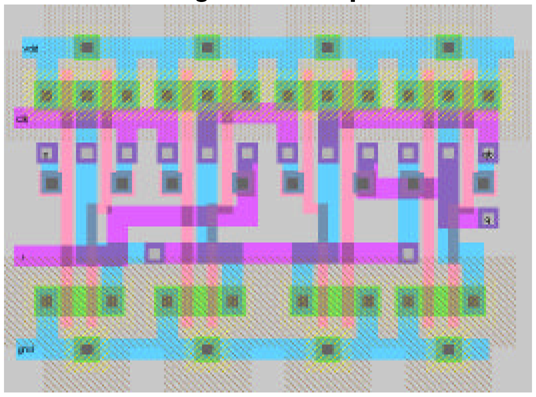


Figure 36: latch