Digital Signal Processors Using VLSI and FPGA's

Jeremy R. Barsten
Jeremy Stockwell

Advisors:
Dr. Vinod B. Prasad
Dr. Thomas L. Stewart

Bradley University Department of Electrical and Computer Engineering

Abstract

This project deals with the design and implementation of a general-purpose signal processor using digital technology and design, namely VHDL, FPGA (Field Programmable Gate Arrays), and VLSI (Very Large Scale Integration). This processor consists of a multiplier and adder, a digital filter, and a "shift-and-rotate" type of data management. Once all stages are completed, the processor will be flashed to a Xilinx FPGA board for testing and demonstration.

Table of Contents

Introduction

In the world today, the digital product industry is ever growing, and the development of digitally based products is rising. Various industries such as audio, video, and cellular industry rely heavily on digital technology. A great part of this deals with digital signal processing. This aspect in engineering has gained increasing interest, especially with much of the world now turning to wireless technology and its applications to keep businesses and industries connected. The world of digital technology is certainly one that will be present for many years to come.

Digital signal processing is an area of electrical engineering that has rapidly grown in the past 30 years. Advances in digital hardware and digital computers have spurred this growth. Current integrated circuit technology, namely very-large-scale integration (VLSI), made it possible to develop smaller, faster, and cheaper special-purpose digital processing. These circuits made it possible to construct digital systems capable of performing the complex digital signal processing tasks that are usually too difficult or too expensive to be performed by analog circuitry. Today, many of the functions usually performed by analog means are now realized by less expensive and more reliable digital hardware[1].

The purpose of this project was to design and construct two high-speed, application-specific digital signal processors (DSP's). One processor was developed by Jeremy Barsten utilizing Very High Speed Integrated Circuit Hardware Descriptive Language (VHDL) to be implemented utilizing a Field

Programmable Gate Array (FPGA). Jeremy Stockwell designed the other in VLSI, and this design will be sent for fabrication on an ASIC chip. However, because of the complexity of the design, the VLSI-based processor was a scaled down version from that done in VHDL.

Overall Design

Digital Signal Processing is the process of manipulating a digital input utilizing a transfer function. This project implemented a second order Direct Form II realization of an infinite-impulse response (IIR) filter. Utilizing this format we derived the following transfer function necessary to process the input signal:

$$y(n)=b_0 \bullet w(n) + b_1 \bullet w(n-1) + b_2 \bullet w(n-2), \text{ where } w(n)=x(n) - a_1 \bullet w(n-1) - a_2 \bullet w(n-2)$$

This simplifies to:

$$Y(n)=b_0 \bullet x(n) + b_1 \bullet x(n-1) + b_2 \bullet x(n-2) - a_1 \bullet y(n-1) - a_2 \bullet y(n-2)$$

The coefficients a_1, a_2, b_0, b_1, and b_2 can be changed to adapt the processor to many different applications. The block diagram for this Direct Form II implementation is shown in Figure 1. Based on this diagram and the above equations, there are four discrete subsystems that must be designed in order to realize this processor. These components are signal converters, a two's complement signed-multiplier, a full adder, and a block of data storage. A top-down design approach was implemented so that each component was designed and tested individually before being added to the overall system. Once all the components were pieced together, the entire DSP would be tested.

[1] Information taken from Digital Signal Processing by John G. Proakis and Dimitris G. Manolakis

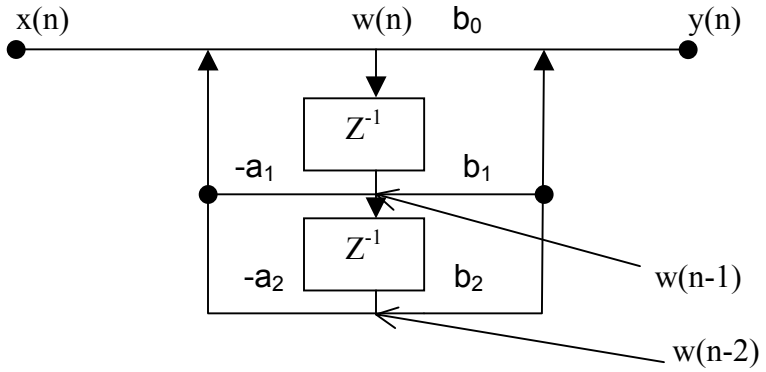

Figure 1: Direct Form II realization of an IIR Filter

Signal Converters

Many of the signals today are analog in nature. However, analog circuits do not provide the reliability and speed of digital circuits. In order to process a signal using a digital circuit, the signal must be converted to a digital one before being sent to the processor. Also, the digital output of the processor must be converted back to an analog one before being passed onto other applications. Therefore, two signal converters are required at each stage of the processor: a analog-to-digital (A/D) converter at the input stage and a digital-to-analog (D/A) at the output stage. The block design representing this is shown in Figure 2.

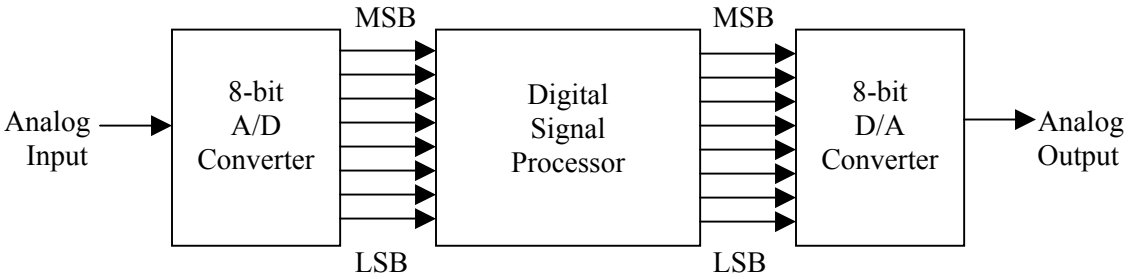

Figure 2: Block diagram showing the connections of the A/D and the D/A converter

The converters chosen for this project are the ones on the FPGA board used in this project. They are part of the circuit board and have already been

interfaced with the FPGA[2]. These will be utilized instead of setting up external circuitry to run the conversions.

Adder and Multiplier Stages

As evident through the transfer function and Figure 1, multiplication and addition are the main functions of digital signal processing. The input values and previous output values must be multiplied by certain coefficients, and then these products must be summed together. Also, each of these must be designed so that they can process negative two's compliment values due to the values of the coefficients.

Adder: VHDL and VLSI Design

Some differences exist between the design in VHDL and VLSI. In VHDL, the first task was to determine the capabilities of the Xilinx compiler. To do this a ripple carry adder (Figure 3) and a carry look-ahead (Figure 4) adder were created with VHDL.

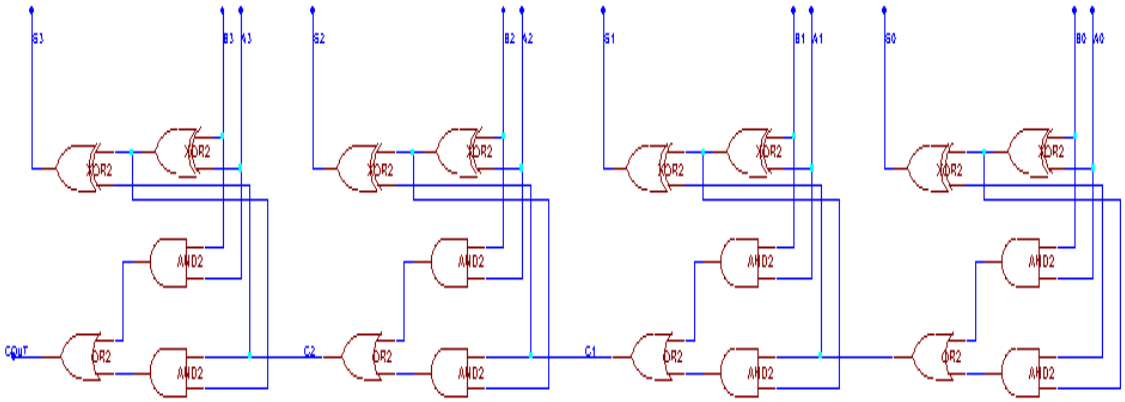

Figure 3: Logical diagram for a 4-bit ripple carry adder.

[2] The FPGA Exansion board used was developed by Brett Marshal and Mike Parker in a previous project.

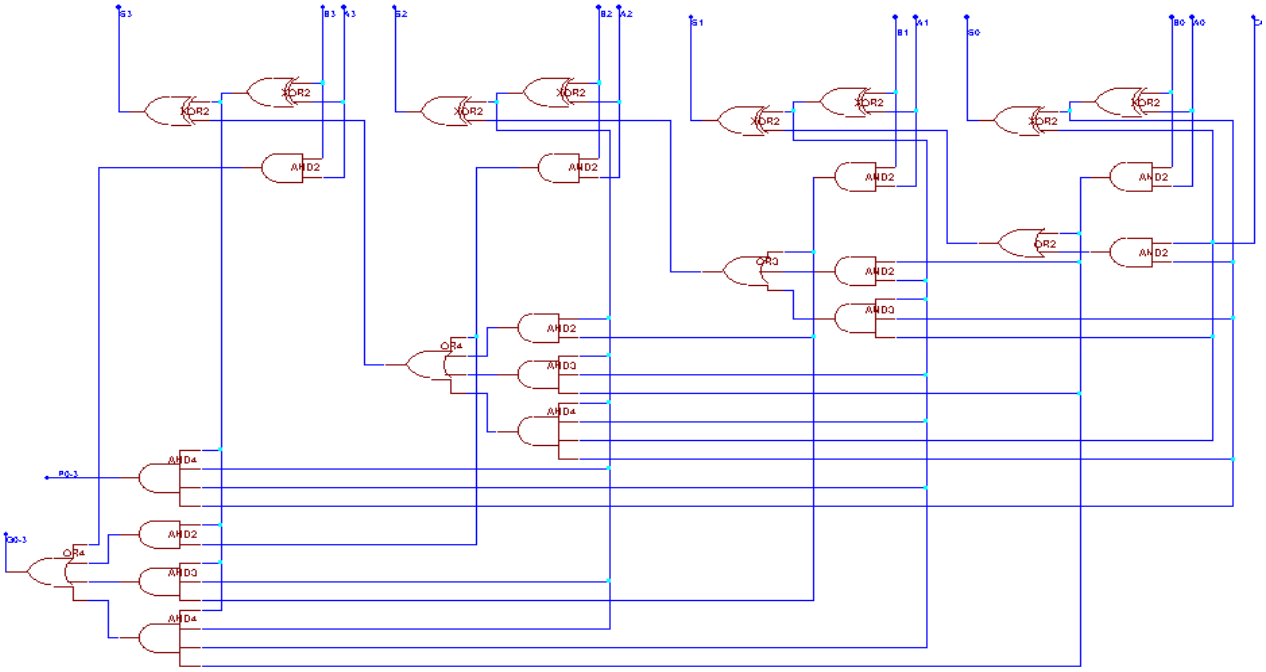

Figure 4: Logical diagram for a 4-bit carry look-ahead adder

A carry ripple adder has a delay of 2n+2 gate delays. To decrease this delay a carry look-ahead adder can be created. This type of adder requires more logic, but the speed advantage is significant. For example, a 16-bit carry ripple adder results in 34 gate delays, while a carry look-ahead adder only has 10 gate delays.

To test the Xilinx compiler a simple adder (a + b = c) was created in VHDL. Then all three adders were simulated and the delays were compared to determine what type of adder the compiler created. The results showed that the complier created a carry look-ahead or equivalent adder.

	Carry Look-ahead Adder	Ripple Adder	Simple Adder
Delay Times	11.05ns	22.43ns	11.05ns

Figure 5: Table of delay times from CLA, carry ripple, and simple adders

To improve the stability and accuracy of the adder, overflow protection was also included. If this was not added and an overflow occurred, then the processor would produce erroneous information.

The design of the adder in VLSI was different than that in VHDL for different reasons. Because of the complexity of the carry-look-ahead adder, a simply ripple-carry adder was used in its place. Not only would design for a carry-look-ahead adder consume the most time, but it would also require a large amount of chip area.

A cellular approach was used in the design of the ripple-carry adder. First, a simple one-bit full adder was designed[3]. This design, shown in Figure 6, consisted of three inputs, two addend bits and carry-in bit, and produced the appropriate carry-out bit and sum bit. This design was first simulated in a logic-design program before it was built and simulated in VLSI. Once functioning correctly, the appropriate CMOS circuit was built in L-Edit Pro and tested by extracting a Spice file. (The VLSI adder cell schematic is shown in Appendix Figure A1.) Simulation proved that this design did work by following the appropriate truth table.

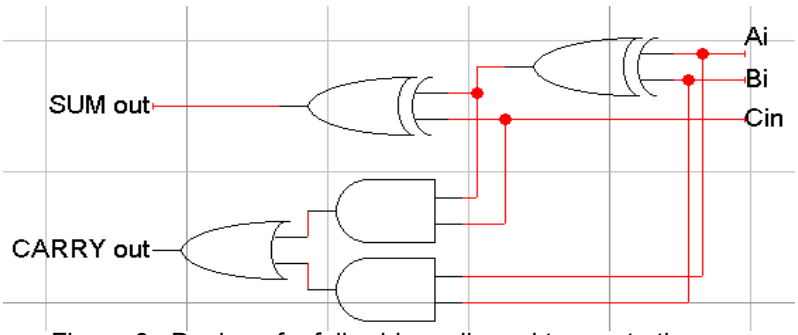

Figure 6: Design of a full-adder cell used to create the ripple-carry adder in VLSI.

[3] Circuit design taken from Physical Design of CMOS Integrated Circuits Using L-Edit by John P. Uyemura.

Once the full-adder cell worked correctly, five identical cells could be cascaded together to form the full 5-bit adder. This adder would take two 5-bit numbers (where the most significant bit of each was the sign bit) and perform the appropriate two's complement addition. This adder would then be placed at the output of the multiplier stage in the overall design for the processor. The logical design for the 5-bit ripple carry adder is shown in Figure 7. Because the VLSI full-adder cell simulated correctly, five identical cells were cascaded together in L-Edit, where the carry-out of each cell was connect to the carry-in of the adjacent cell. When the design was complete, the CMOS design (shown in Figure A2) was tested to show that the ripple-carry adder functioned correctly in VLSI. Several simulations showed that the adder worked for the most part, but some problems did exit with overflow. These points will be discussed later in the paper.

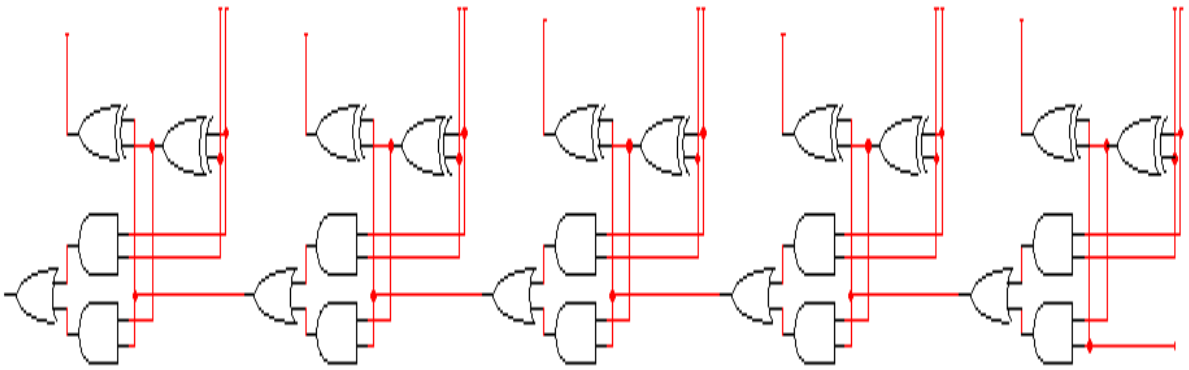

Figure 7: Logical design of a 5-bit ripple carry adder used to create the CMOS signed adder.

Multiplier: VHDL and VLSI Design

Many different multipliers were designed in VHDL to determine the best ratio of size versus speed. Two types of multipliers, serial and parallel, were

investigated. A serial multiplier is easier to create and takes less area on the FPGA, but the speed is decreased nearly n times for a n-bit multiplier. A parallel multiplier has the advantage of speed and would be utilized if area was of no concern.

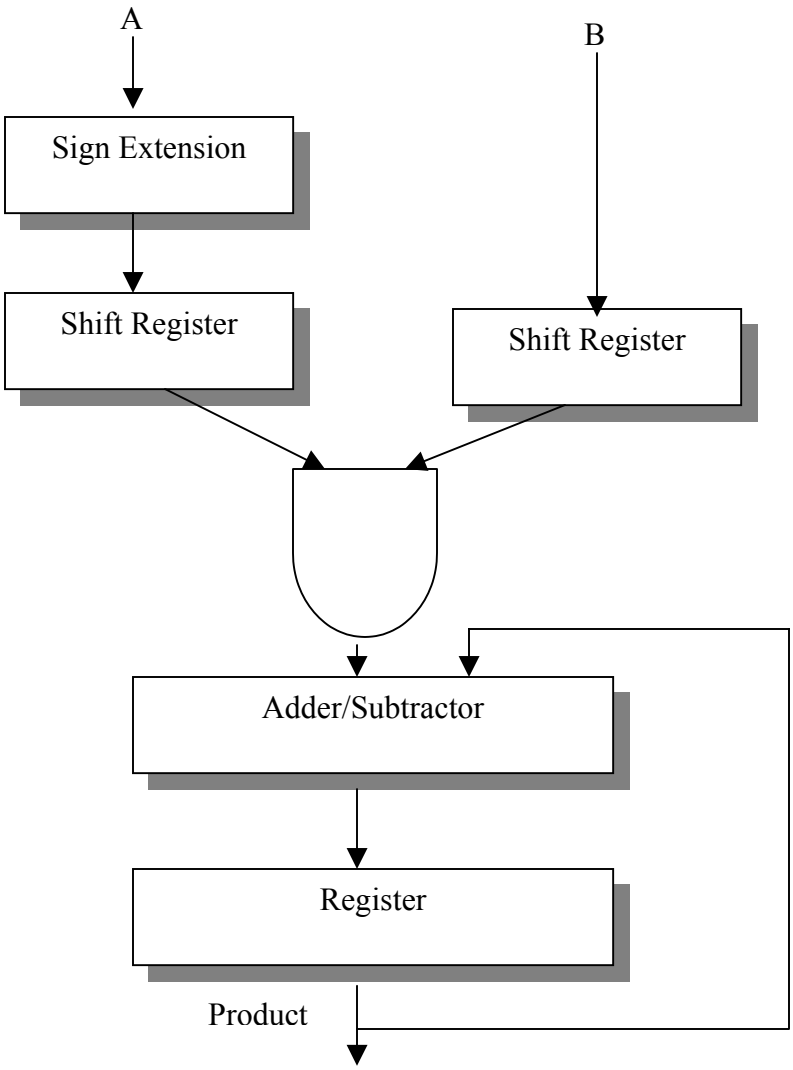

Figure 8: Data path for signed 2's complement serial multiplier

To conclude what course to take, first, a serial multiplier was created. Digital signal processors utilize signed numbers, so a signed multiplier needed to be designed. A parallel multiplier was created using the Xilinx compiler. 4-bit, 8-bit, and 16-bit multipliers were created for both.

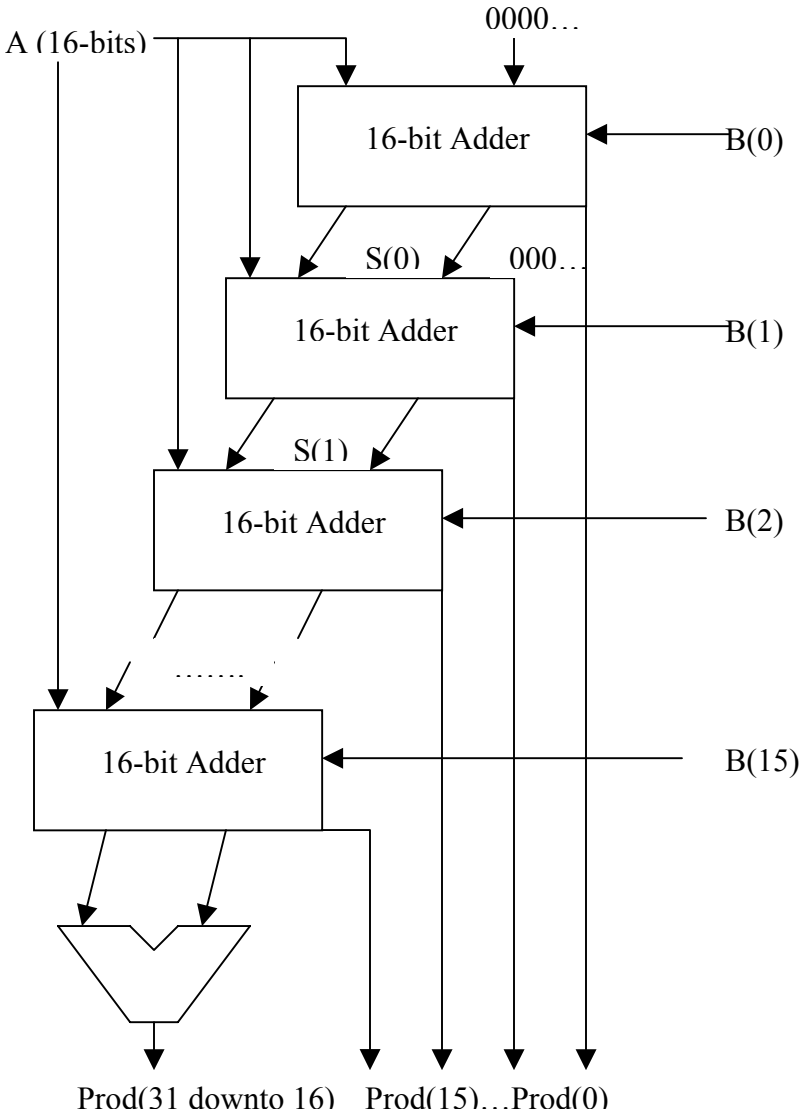

Figure 9: Data path for signed 2's complement parallel multiplier

	4-bit		8-bit		16-bit	
	Area	Delay	Area	Delay	Area	Delay
Serial Multiplier	7.14%	96.88ns	13.52%	210.8ns	24.49%	503.68ns
Parallel Multiplier	10.71%	27.41ns	17.81%	38.76ns	65.40%	72.06ns

Figure10: Table of area used and delay from serial and parallel multipliers.

A parallel multiplier was used because it would, in fact, fit on the chip. To increase the speed even further a modified Booth's Algorithm was used. Booth's algorithm (Figure 11) increases the speed by cutting the number of necessary

Bit			Operation
Yi+1	Yi	Y	
0	0	0	add zero
0	0	1	add X
0	1	0	add X
0	1	1	add 2X
1	0	0	subtract 2X
1	0	1	Subtract X
1	1	0	Subtract X
1	1	1	Subtract 0

Figure 11: Modified Booth's Algorithm

additions in half. The 16-bit Modified Booth's Multiplier had decreased the delay of 60 ns.

One problem that arose with the multiplier was that it would occasionally produce an extra sign bit. It was discovered that this problem only arose when multiplier two identical numbers. Checking for that specific case and adjusting the product accordingly corrected this dilemma.

As with the adder, the design of the multiplier differed for the VLSI portion of the project. Because of time constraints and complexity issues, a 4-bit by 4-bit cellular multiplier was designed instead of a 16-bit by 16-bit Booth's modified multiplier. Also, a cellular approach was taken when designing the CMOS multiplier as well. A single multiplier cell (Figure 12) was initially designed and tested before the entire multiplier was built. Once functioning correctly, this cell would be arranged in an array to form a 4-bit by 4-bit multiplier, much like the approach taken with the adder. After designing and testing the multiplier cell logically, the equivalent CMOS circuit (Figure A5) was drawn in L-Edit and then simulated in PSpice. Comparing the two simulations produced identical results,

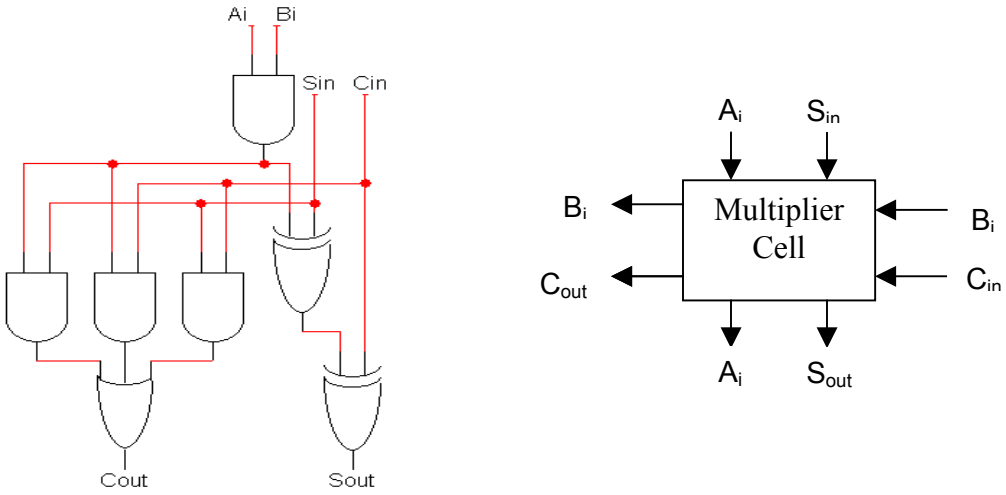

Figure 12: Logic schematic and block diagram of the multiplier cell used in the design of the 8-bit multiplier. The L-Edit schematic is show in Figure A5.

and the full multiplier was ready to be designed.

Unlike the adder, a simple cascaded design could not be used in the case of the multiplier. The cellular array shown in Figure 13 displays the interconnections of the multiplier cells that form the full circuit. In all, sixteen cells made up this design, and the total number of transistors exceeded one thousand. To test the circuit, five different simulations tested the sixteen different combinations for inputs a_0 through a_3 which were multiplied by zero, one, three, seven, and fifteen. The products in simulation were compared to answers from a calculator; the conclusions from these results showed that the multiplier functioned correctly.

However, this multiplier only computed positive numbers, so circuitry needed to be added to the input and output stages so that it would compute negative numbers correctly. The input complement blocks must take a two's complement number and convert it to the corresponding positive number before

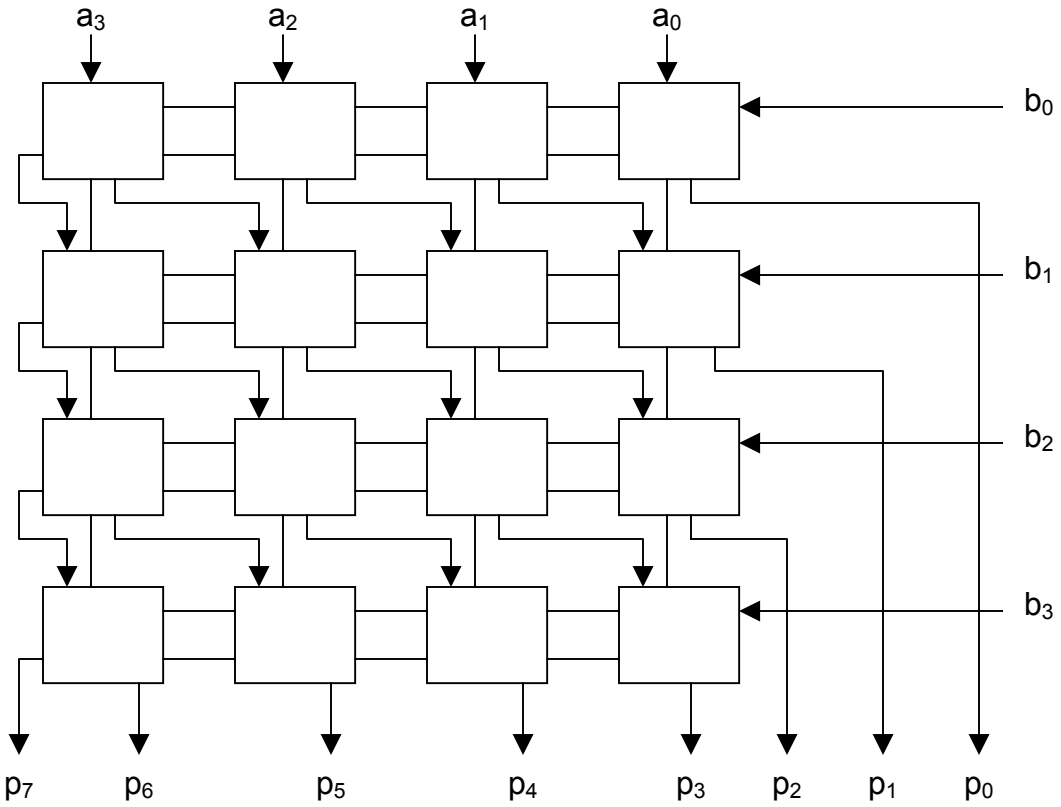

Figure 13: Cellular design of the 4-bit by 4-bit multiplier utilizing the multiplier cell in Figure 5. The VLSI multiplier design is shown in Figure A6.

being sent to the multiplier. The corresponding output complement block would then recognize that the product is either positive or negative based on the inputs and adjust the output value accordingly. To accomplish this, the ripple-carry adder discussed previously and a 2-to-1 multiplexer were added to the input and output stages of the multiplier (Figure 14). To convert to a positive number, the two's complement negative number must undergo bit wise negation; then, one is added to the negated value. The multiplexer uses the sign bit to select the appropriate value to send into the multiplier. At the output, the select bit looks at

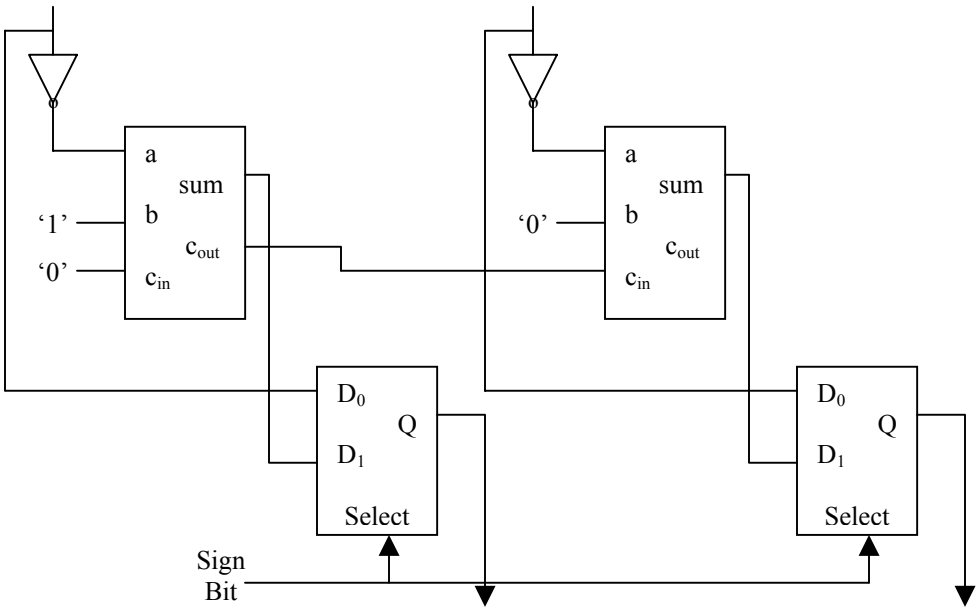

Figure 14: Block diagram of the input/output adjustment circuitry for the two least significant bits of the input or output of the multiplier.

the sign bits of both inputs and the carry-out of the most significant bit of the ripple carry adder used to convert the two's complement number. Based on these signals, the adjustment circuitry selects the appropriate positive number to send through the multiplier. Likewise, the output adjustment circuitry contains circuitry that selects the appropriate value to send out of the multiplier. However, the select bit at this stage looks at the sign bits of both inputs and the carry out of the most significant bit of the two's complement ripple-carry adder. Based on these three signals, the multiplexer selects the appropriate values to send out of the multiplier.

In addition, there were two special cases that needed to corrected. This occurred when the input was $(10000)_2$, or –15 (the least negative input), and when the output was $(100000000)_2$, or –255 (the least negative output). To adjust for these instances, another stage of multiplexers was added to the

adjustment circuitry that specifically looked for these cases. The only time this occurred was when the sign bit of the input or output and the carry-out bit of the two-complement adjustment adder were both HIGH (or 5V). On these occasions, the adjustment circuitry would recognize these numbers and send the correct numbers into or out of the multiplier. These circuits were designed and tested separately. Once both the input and output adjustment circuits were working correctly, their cells were instantiated into the multiplier layout, and the correct connections were made between the appropriate cells. Once connected, the entire layout was checked for design errors, and the circuit file was then extracted and simulated. The results showed that the adjustment circuitry functioned correctly, and the multiplier now produced the correct two's complement numbers when specific inputs were given. (The simulations for both the unsigned multiplier and the multiplier with the adjustment circuitry can be found in Appendix Figures A7 and A8.)

Data Management

This block of the processor handles all of the filter coefficients and previously stored values from the processor. Looking at the equations for the digital filter, there are five coefficients that need to be stored in the data block: b_0, b_1, b_2, a_1, and a_2. Also, the values $\omega(n-1)$ and $\omega(n-2)$ are values that are delayed through the system. These values need to be stored in memory in this data management block so they can be delivered to the adder and multiplier stages at the correct time. This will allow the processor to correctly compute the new output values of the signal. Therefore, this part of memory will be set up in a

"shift-and-rotate" fashion so that when a new value enters the system, the old values will be rotated out and the delayed values can be stored correctly. This flow chart for this data management block is shown in Figure 15.

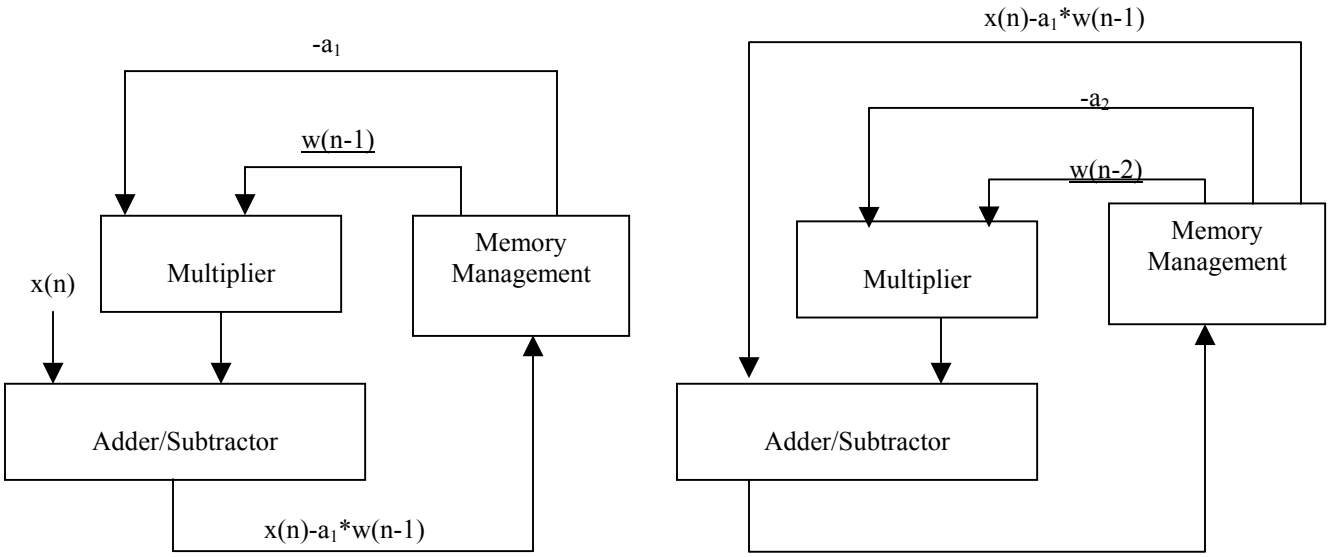

Figure 15: Block diagram showing consecutive steps in the data management.

The same general flow was used for both the VLSI and VHDL designs. Seven cycles were used for the entire design in VHDL. The first cycle is used to trigger an analog-to-digital (A/D) converter. A temporary register was also created to store the data until the next cycle. The next five cycles load the data to the multiplier and adder. For each of these cycles the product from the multiplier is fed into adder. For this reason the clock that controls the cycles must be set with a period long enough for the product to be calculated by the multiplier and then summed by the adder. The last two cycles are used to shift the data to be used with the next input. The flow for this process can be seen in Figure 16.

Cycle	Mul1	Mul2	Add		
000	This cycle is used to trigger the A/D convertor				
001	a1	W(n-1)	X(n)		
010	a2	W(n-2)	Temp Reg		
011	b2	W(n-2)	0		
100	b1	W(n-1)	Temp Reg		
101	b0	W(n)	Temp Reg	W(n-1) =>W(n-2)	
110	W(n) => W(n-1)				

Figure 16: Table showing the data management flow for the VHDL code.

The entire VHDL design has been completed and tested successfully. The next step will be to flash this design to the FPGA expansion board. The necessary adjustments and code for this process has been completed, but at this point there has not been an opportunity to verify that the design does in fact work on the board.

More cycles were used for the VLSI design than for the VHDL design. The concurrent and parallel nature of the VHDL design required fewer cycles than the eleven cycles (Figure 17)used in the CMOS processor design. Also, 5-bit registers had to be designed in order to store previously calculated values as well as the different equation coefficients. Finally, a sequential clock circuit consisting of flip-flops was designed to control the flow of data as well as the order in which operations were performed.

The first thing accomplished was the design of the registers. These consisted of five d-type latches where the data flow was controlled by a clock signal. These

Cycle	Mul1	Mul2	Add1	Add2	Product	Sum
1	a_1	$\omega(n-1)$			R_{mul}	
2			x(n)	R_{mul}		R_{add}
3	a_2	$\omega(n-2)$			R_{mul}	
4			R_{add}	R_{mul}		$\omega(n)$
5	b_0	$\omega(n)$			R_{temp}	
6	b_1	$\omega(n-1)$			R_{mul}	
7			R_{temp}	R_{mul}		R_{add}
8	b_2	$\omega(n-2)$			R_{mul}	
9			R_{add}	R_{mul}		y(n)
10	$\omega(n-1)$ -> $\underline{\omega(n-2)}$					
11	$\omega(n)$ -> $\omega(n-1)$					

Figure 17: Table of cycles for the VLSI processor.

latches consisted of simple NAND gates shown in Figure 18[4]. The equivalent CMOS circuit was constructed and simulated. It was found that whenever the clock signal was HIGH, the output followed the input, and when the clock signal was LOW, the output remained at the last input before the next clock transition. Therefore, these registers would be positively triggered latches.

After the registers, the next goal to accomplish was the sequential clock circuitry that would control the flow of data and processes in the processor. This design produced a sequence of eleven clock pulses that controlled various registers and circuits throughout the processor. To accomplish this, twelve d-type flip-flops were cascaded to produce a delayed signal throughout the circuit. To start the sequence, an input pulse must be sent to the first flip-flop, and to

[4] The design for this latch was taken from the website http://www.play-hookey.com.

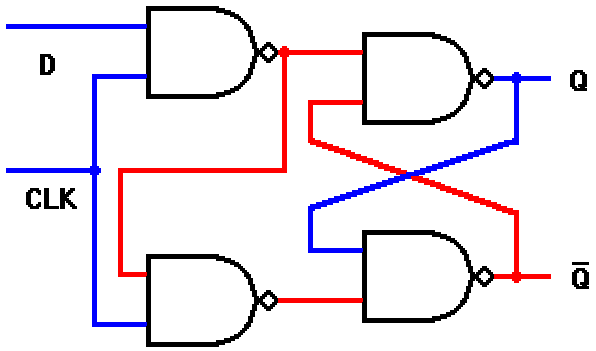

Figure 18: Logical design of the d-type latch used to form the 5-bit registers used to store coefficients and previous values.

continue the sequence, the output of the final flip-flop must be sent to the first one. Because of this, a 2-input OR gate was required at the input of the circuit, as shown in Figure 19. The different pulses were then taken from the output of the appropriate flip-flop.

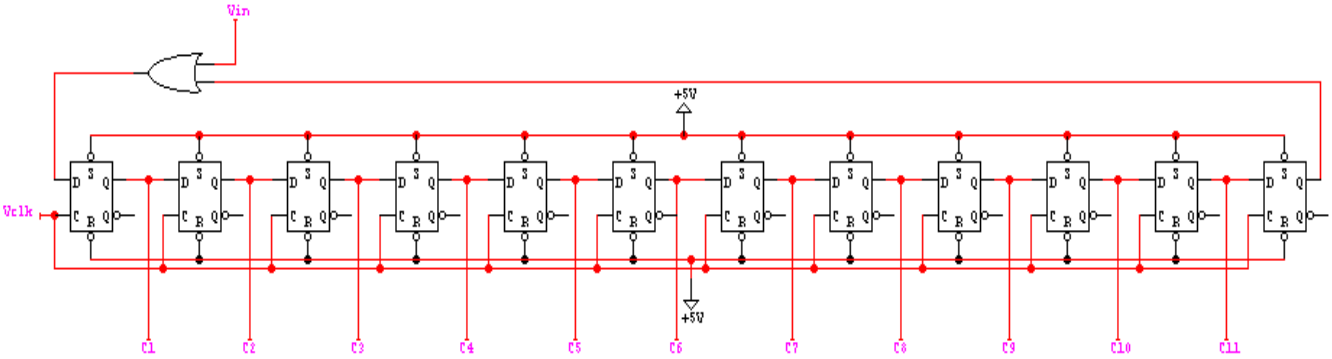

Figure 19: Logical design of the sequential circuit used to control the processes and flow of data in the processor.

To implement this design in VLSI, the filp-flop needed to be designed first. For this, a circuit consisting of NOR gates, seen in Figure 20, was used to create the flip-flop. Once this design was simulated, the overall sequential circuit was assembled according to the logical diagram in Figure 19 and simulated. The results showed that the correct clock pulses were produced at the correct times

according to the clock input. (The VLSI design and simulations can be found in Figures A9 and A10.)

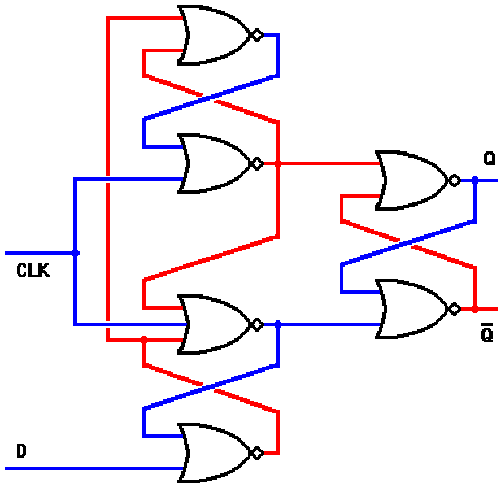

Figure 20: Design of a d-type flip-flop using NOR gates[5].

With both the registers and sequence circuit designed, the VLSI processor (Figure A11) could be connected together. The correct cells were instantiated into one file where the correct connections were made between all circuits (adder, multiplier, registers, and sequential clock circuits). The final design of the processor can be found in the Appendix. All results from this design will be discussed in the next section.

Results and Conclusions

The VHDL processor was successfully tested using Modelsim. The comparison of the Impulse Response of a low-pass filter simulated in Matlab and the same filter implemented on the processor can be seen in Figure 21.

The final delay of the processor was 648 ns, or in other words the speed of the processor was 1.5 MHz. This is ample speed for most applications. Using

higher-ended FPGA's can further increase the speed of the processor.

When programming the FPGA board, problems were encountered. The primary trouble was with the internal clock of the FPGA. The clock would

Matlab	DSP
0.905	0.905
-0.7331	-0.733
0.5938	0.594
-0.481	-0.481
0.3896	0.3895
-0.3156	-0.3156
0.2556	0.2555
-0.207	-0.2071
0.1677	0.1676
-0.1358	-0.1359
0.11	0.11
-0.0891	-0.0891

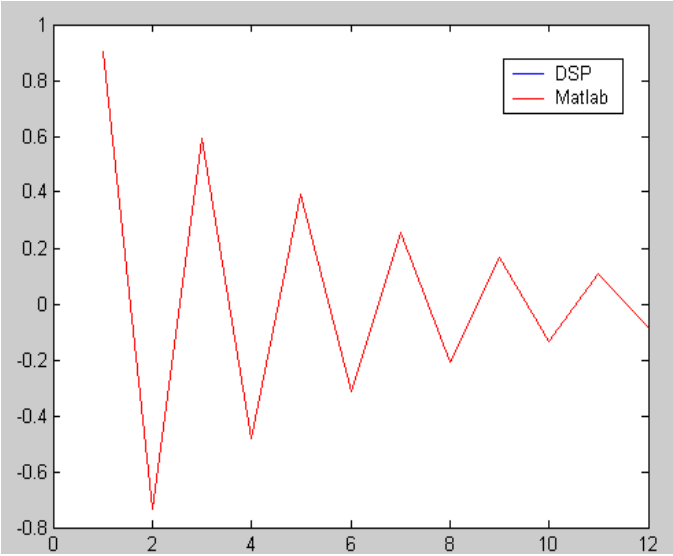

Figure 21: Impulse Response of a low-pass filter implemented using Matlab and the DSP.

not change from its 20 ns period. This is far too fast for the processor. The cycles would change before any information could be processed causing the output to always be zero. The only time results could be seen was when the momentary push-button switch was used. Unfortunately this switch in not debounced, which caused the data to degrade over time as data would be lost when the cycles were run too fast while the switch was bouncing.

Improvements that could be made on the VHDL processor would be to use a FPGA with a higher gate count and better internal clock. With a higher gate count, creating parallel processors can be used to design higher order

[5] This design for the flip flop was taken from the website http://www.play-hookey.com

filters. This would improve the ability of the processor while keeping the speed high.

For the CMOS processor in L-Edit, no overall simulations could be run. However, one problem did occur when investigating the final design of the processor. While looking at the adder simulations, it was found that certain numbers were adding incorrectly. For example, when one was added to fifteen, the adder produced a sum of negative sixteen. Later, the reason for this problem was discovered; no overflow circuitry was added to the adder. This circuit would be very similar to the adjustment circuitry added to the multiplier. However, time constraints did not allow this design to be completed. Further work on this processor would be a design of this overflow circuitry that would adjust the sum of the adder in cases where an erroneous numbers were generated. Also, more investigation could be done regarding the speed of this processor. This parameter is based upon the internal capacitances at the transistor level of the circuit. To determine this, information must be gathered from fabrication labs as well as the design constraints in the layout of the processor itself.

References

"Digital Logic," Play-Hookey Website. http://www.play-hookey.com. 2003.

Manolakis, Dimitris G. and John G. Proakis. Digital Signal Processing: Principles, Algorithms, and Applications. Prentice Hall, Upper Saddle River, NJ: 1996.

Uyemura, John P. Physical Design of CMOS Integrated Circuits Using L-Edit. PWS Publishing, Boston, MA: 1995.

Appendix A: L-Edit Circuits and Simulations

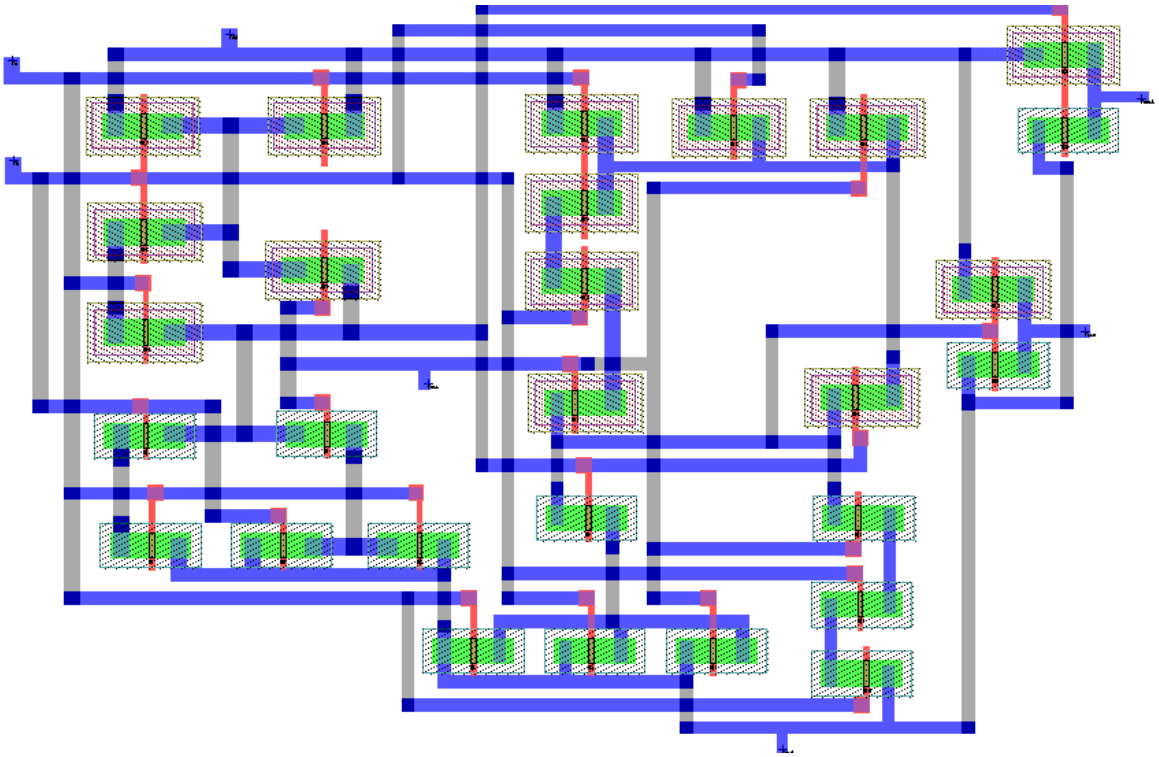

Figure A1: CMOS layout of the adder cell used to create the 5-bit ripple-carry adder used in the VLSI processor.

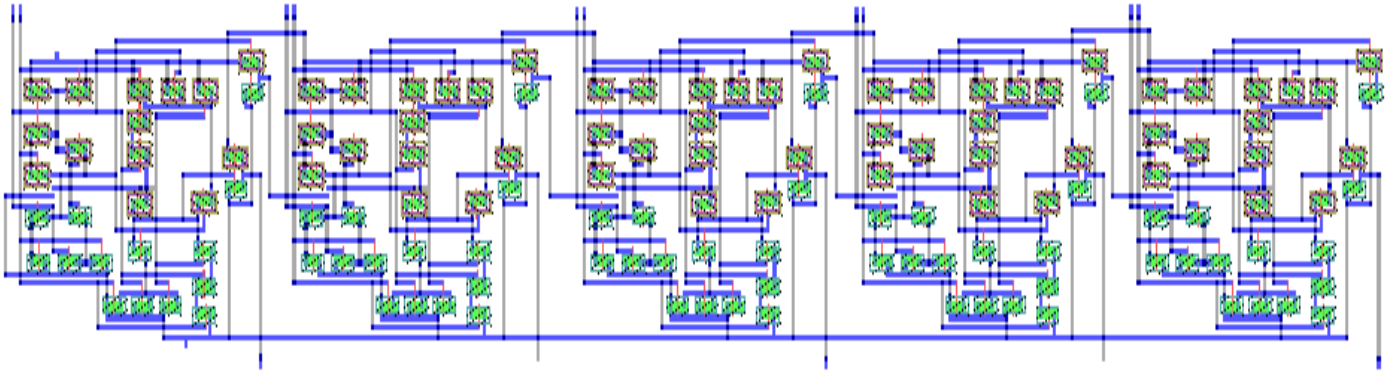

Figure A2: CMOS layout of the 5-bit ripple-carry adder created though cell instantiation used in the VLSI processor.

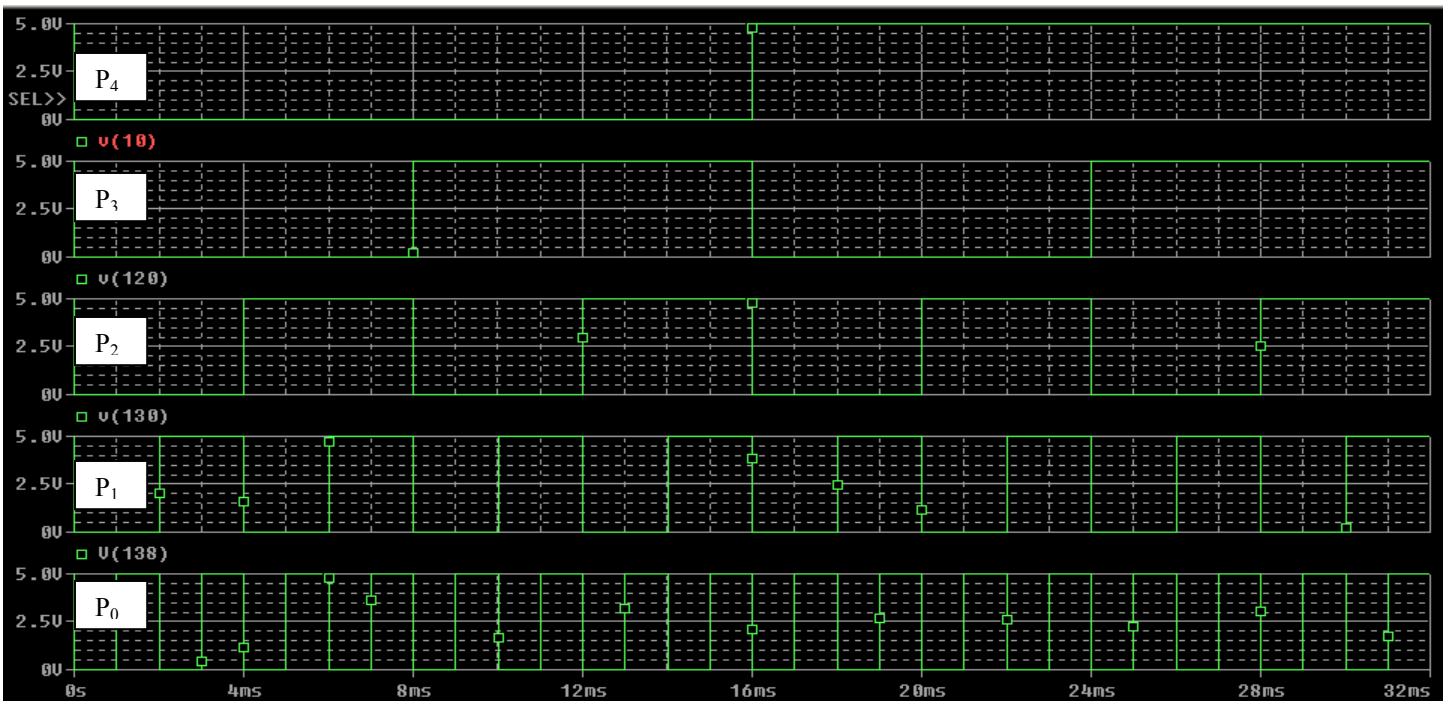

Figure A3: PSpice simulation of the CMOS 5-bit ripple-carry adder where 32 different combinations for a_0-a_3 are added with zero.

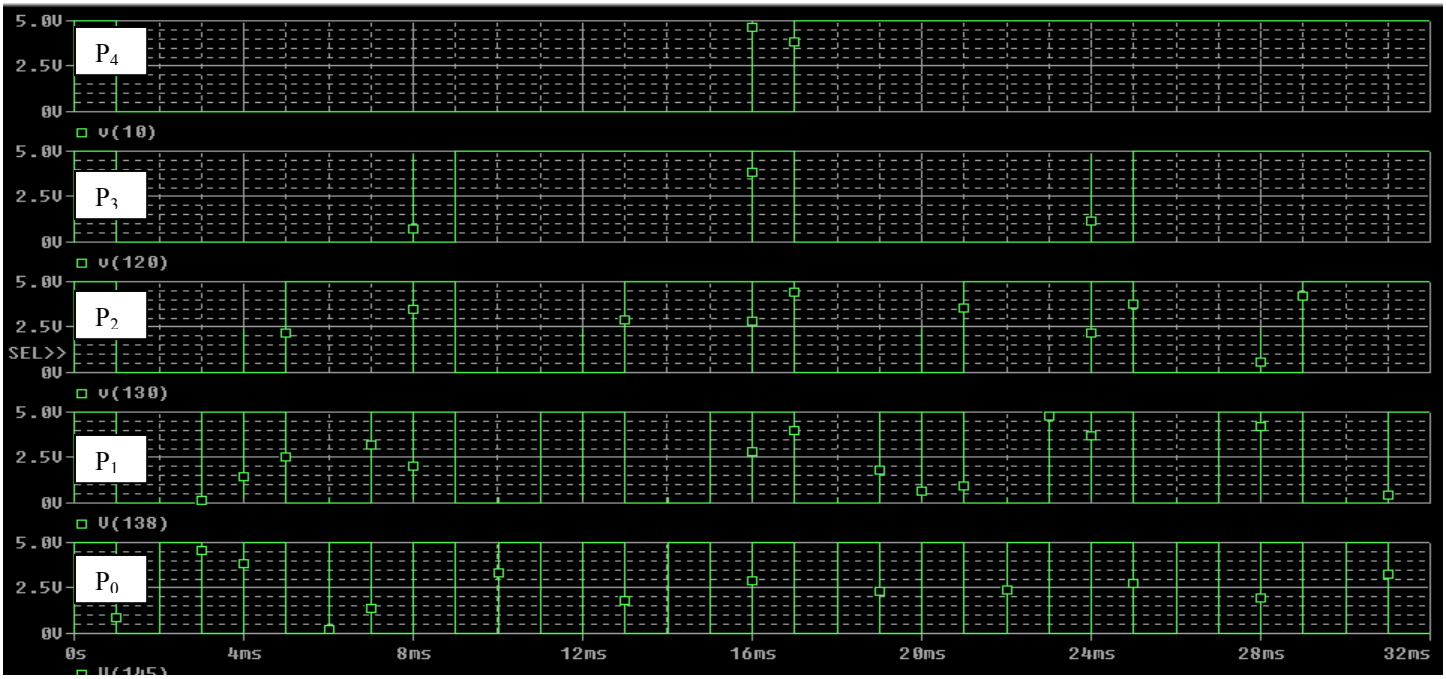

Figure A4: PSpice simulation of the CMOS 5-bit ripple-carry adder where 32 different combinations for a_0-a_3 are added with -1.

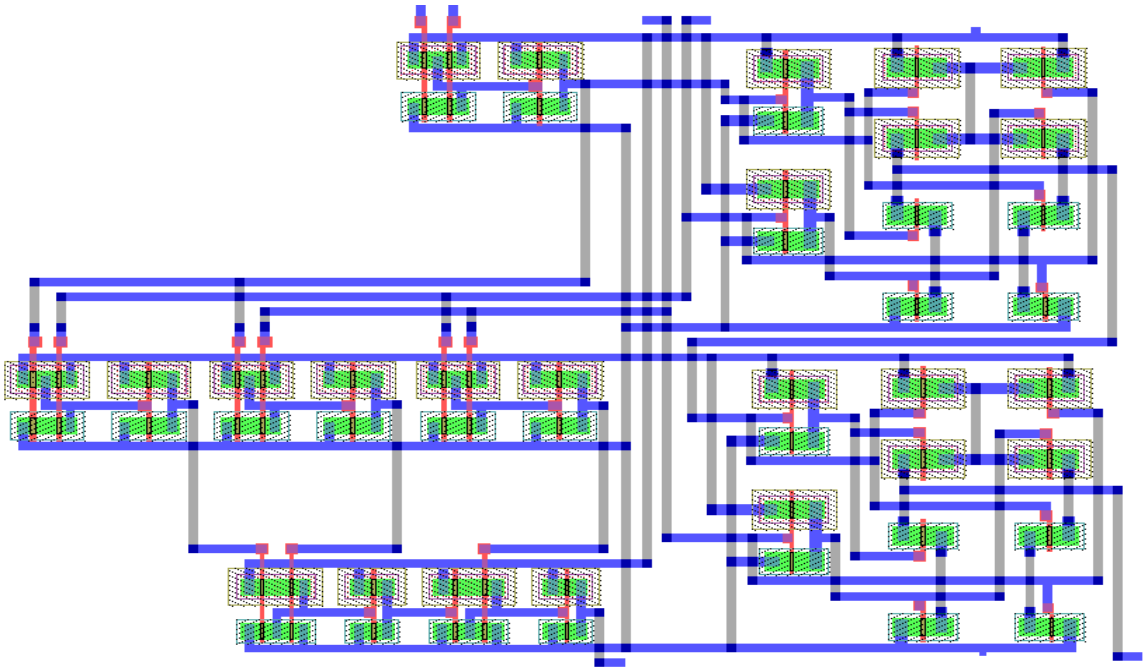

Figure A5: L-Edit layout of the CMOS multiplier cell.

Figure A6: L-Edit layout of the 4-bit by 4-bit multiplier put together by instantiated multiplier cells.

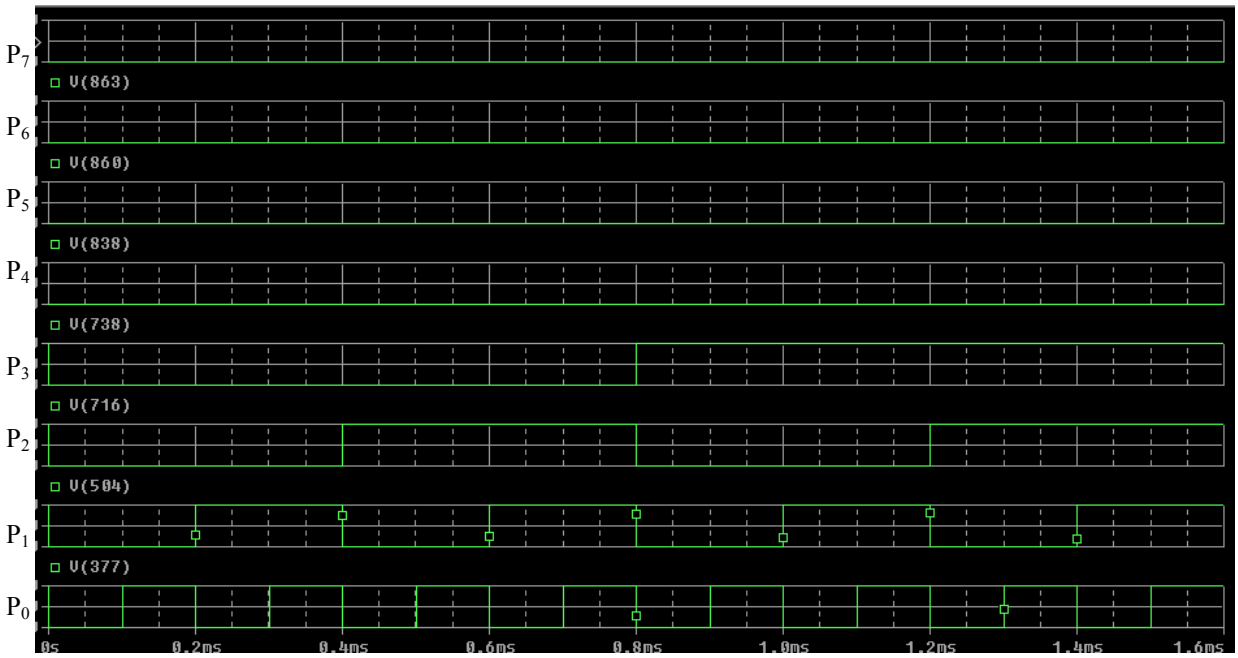

Figure A7: PSpice simulation for the multiplier where 16 different combinations for a_0-a_3 are multiplied by 1.

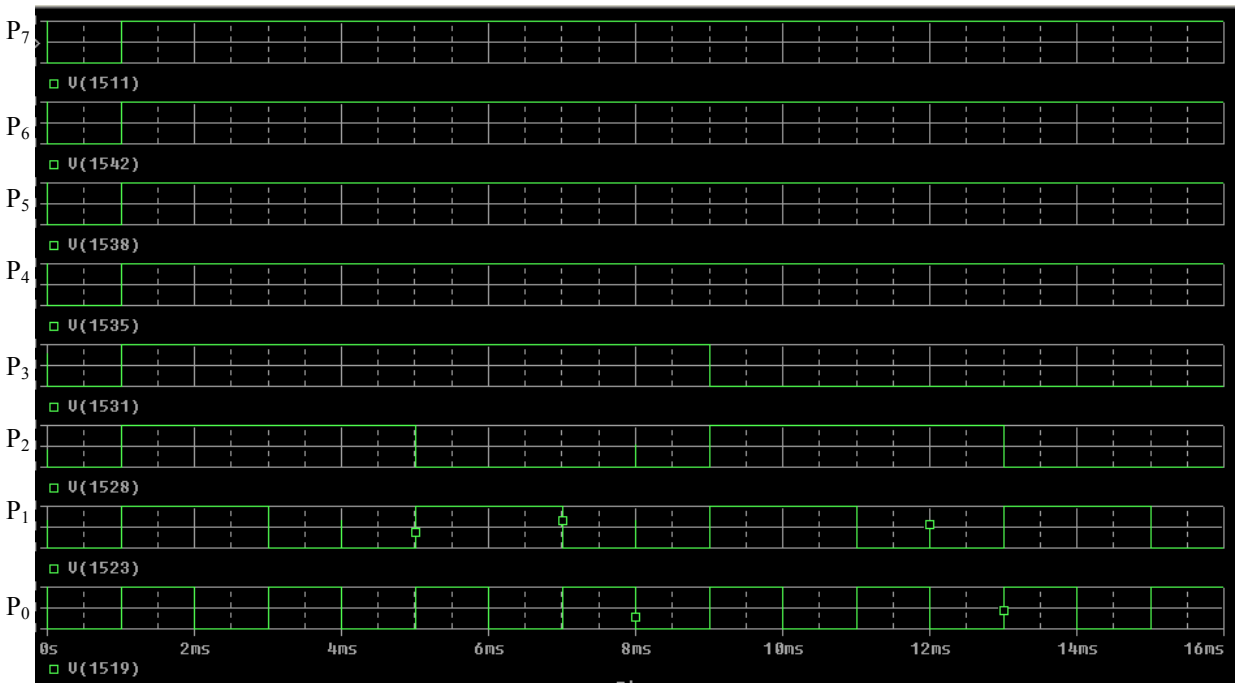

Figure A8: PSpice simulation for the multiplier where 16 different combinations for a_0-a_3 are multiplied by -1.

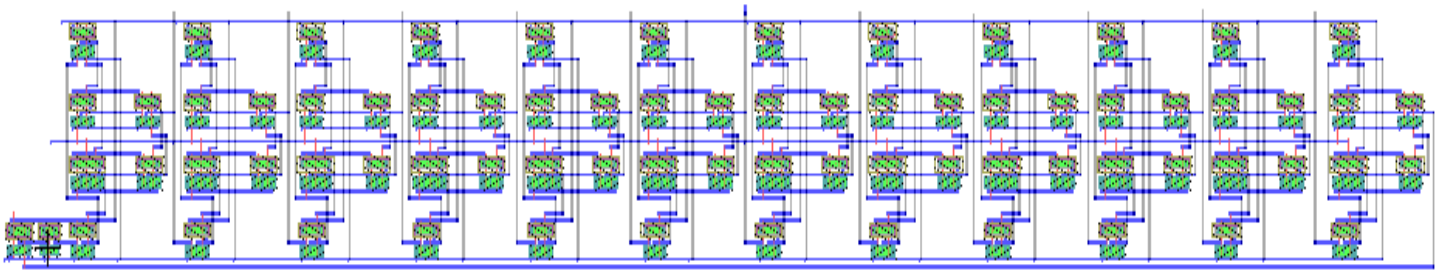

Figure A9: L-Edit layout of the sequential clock-controller circuit using 12 d-type flip-flops.

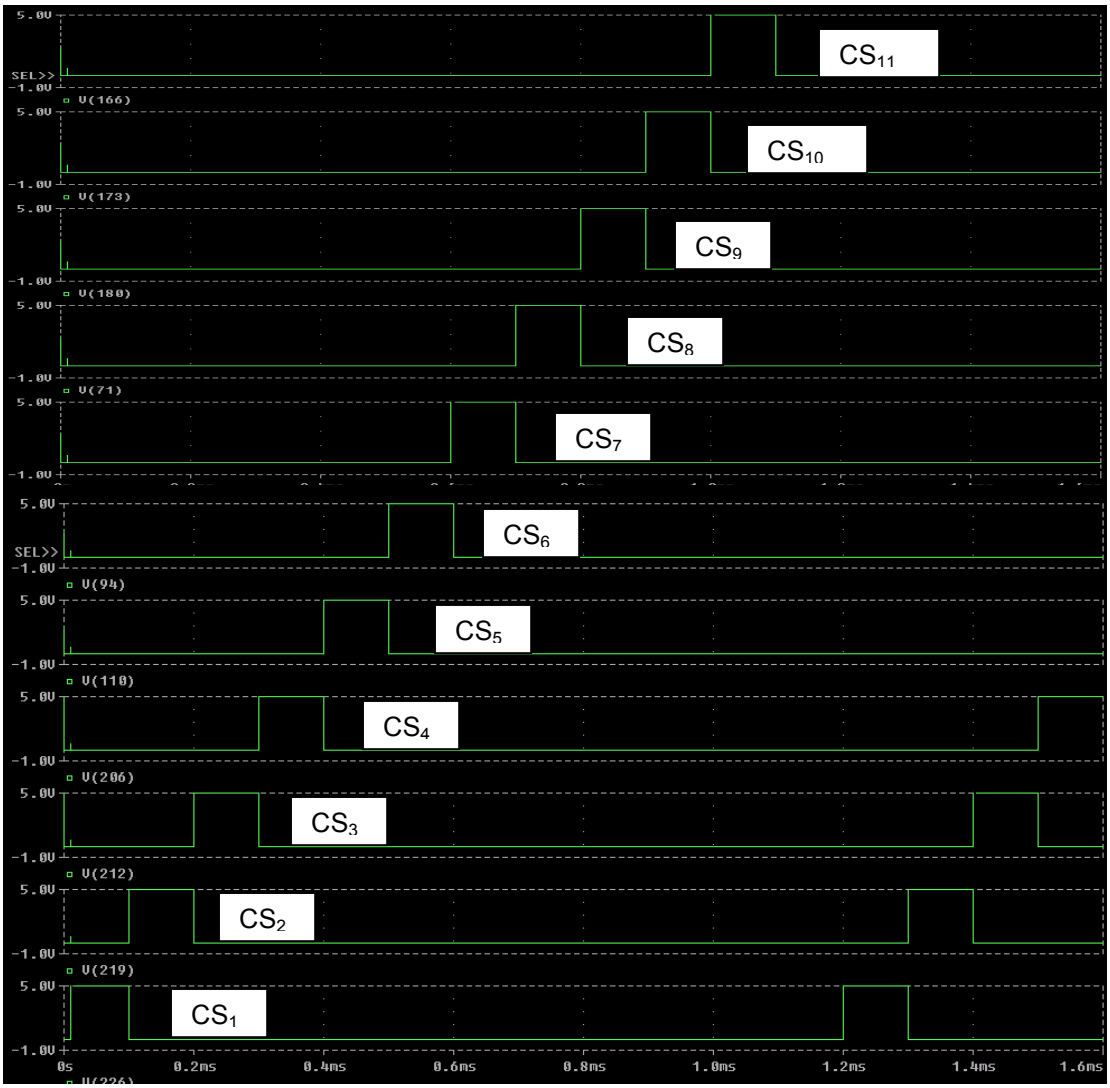

Figure A10: PSpice simulation showing the 11 different clock pulses used to control data flow and processes in the VLSI processor.

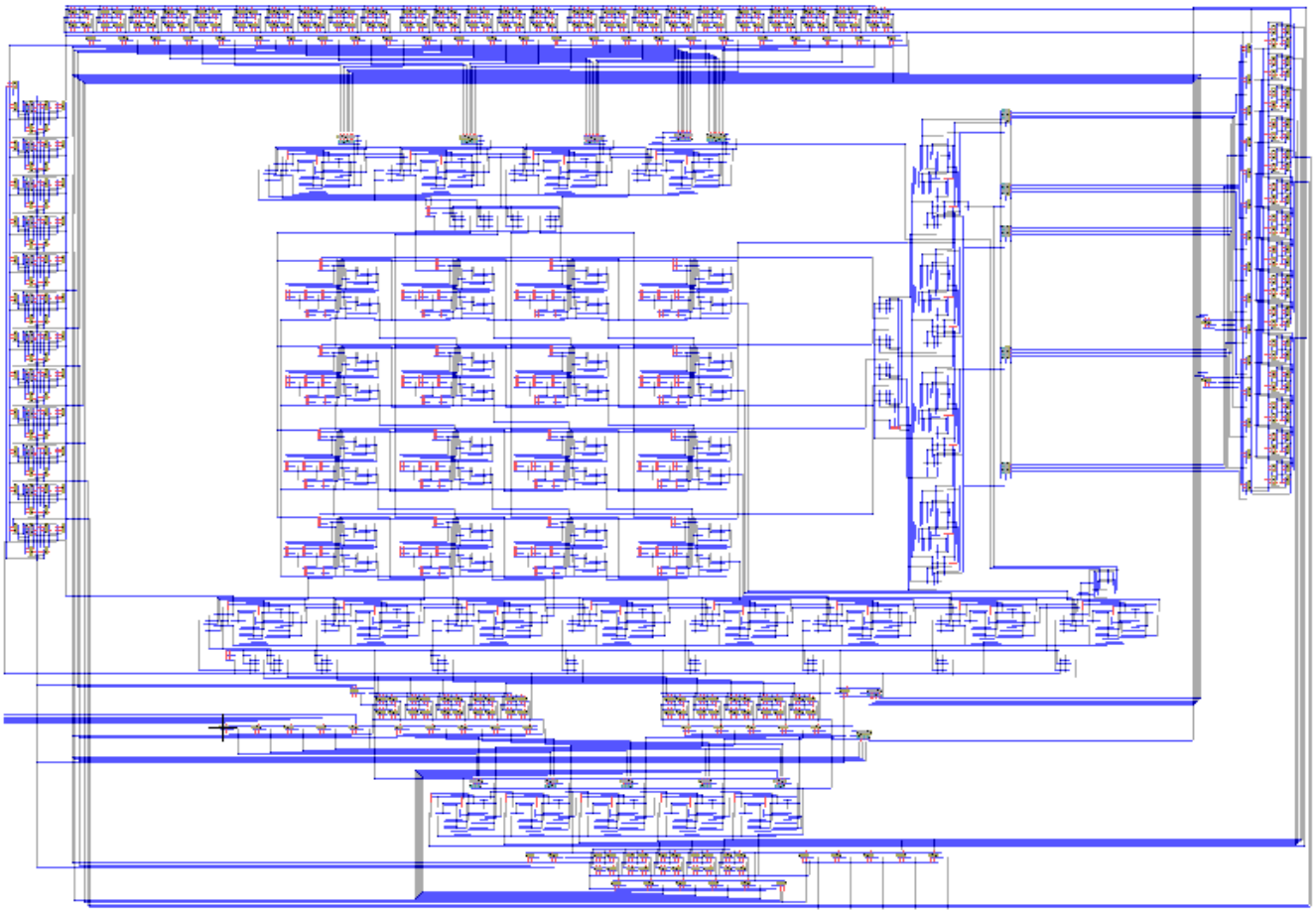

Figure A11: Final CMOS layout of the VLSI digital signal processor created in L-Edit Pro.

Appendix B: VHDL Code for the Digital Signal Processor

Easy Adder

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity add16 is
port(
            a :   in std_logic_vector(15 downto 0);
            b     :     in std_logic_vector(15 downto 0);
            cin   :     in std_logic;
            sum   : out std_logic_vector(15 downto 0);
            cout : out std_logic);
end entity add16;

architecture circuits of add16 is
signal temp_sum : std_logic_vector(16 downto 0);
begin

            sum<=temp_sum(15 downto 0);
            cout<=temp_sum(16);--carry out

process(a,b,cin)--carry adjust
begin
if (cin='0') then
            temp_sum<=('0' & a)+('0' & b);
            else
            temp_sum<=('0'& a)+('0' & b) + "0000000000000001";
end if;
end process;



end architecture circuits;
```

Carry Look Ahead Adder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity add16 is
port(
            a :   in std_logic_vector(15 downto 0);
            b     :     in std_logic_vector(15 downto 0);
            cin   :     in std_logic;
            sum   : out std_logic_vector(15 downto 0);
            cout : out std_logic);
end entity add16;

architecture circuits of add16 is
component fcadd3
      port(c_in   :     in std_logic;
                  a     :     in    std_logic_vector(2 downto 0);
                  b     :     in    std_logic_vector(2 downto 0);
                  sum   :     out   std_logic_vector(2 downto 0);
                  e     :     out std_logic;
                  r     :     out   std_logic);
end component;

component fcadd2
      port(c_in   :     in std_logic;
                  a     :     in    std_logic_vector(1 downto 0);
                  b     :     in    std_logic_vector(1 downto 0);
                  sum   :     out std_logic_vector(1 downto 0);
                  e     :     out std_logic;
                  r     :     out   std_logic);
end component;

signal c3,c6,c9,c12,c15 :     std_logic;
--attribute synthesis_off of c3,c6,c9,c12,c15 : signal is true;
signal e0,e1,e2,e3,e4,e5      :     std_logic;
--attribute synthesis_off of e0,e1,e2,e3,e4,e5 : signal is true;
signal r0,r1,r2,r3,r4,r5 : std_logic;
--attribute synthesis_off of r0,r1,r2,r3,r4,r5 : signal is true;
signal vss : std_logic:='0';
begin

u1: fcadd3 port map (cin,a(2 downto 0),b(2 downto 0),sum(2 downto
0),e0,r0);
u2: fcadd3 port map (c3,a(5 downto 3),b(5 downto 3),sum(5 downto
3),e1,r1);
u3: fcadd3 port map (c6,a(8 downto 6),b(8 downto 6),sum(8 downto
6),e2,r2);
u4: fcadd3 port map (c9,a(11 downto 9),b(11 downto 9),sum(11 downto
9),e3,r3);
u5:   fcadd2 port map   (c12, a(13 downto 12),b(13 downto 12),sum(13
downto 12),e4,r4);
u6:   fcadd2 port map   (c15, a(15 downto 14),b(15 downto 14),sum(15
downto 14),e5,r5);
```

```
c3<=e0 or (r0 and cin);
c6<=e1 or (r1 and e0) or (r1 and r0 and cin);
c9<=e2 or (r2 and e1) or (r2 and r1 and e0) or (r2 and r1 and r0 and
cin);
c12<=e3 or (r3 and e2) or (r3 and r2 and e1) or (r3 and r2 and r1 and
e0) or (r3 and r2 and r1 and r0 and cin);
c15<=e4 or (r4 and e3) or (r4 and r3 and e2) or (r4 and r3 and r2 and
e1) or (r4 and r3 and r2 and r1 and e0) or
       (r4 and r3 and r2 and r1 and r0 and cin);
cout<=e5 or (r5 and e4) or (r5 and r4 and e3) or (r5 and r4 and r3 and
e2) or (r5 and r4 and r3 and r2 and e1) or
            (r5 and r4 and r3 and r2 and r1 and e0) or (r5 and r4 and r3
and r2 and r1 and r0 and cin);

end circuits;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity fcadd2 is
      port(c_in   :     in std_logic;
                  a     :     in    std_logic_vector(1 downto 0);
                  b     :     in    std_logic_vector(1 downto 0);
                  sum   :     out std_logic_vector(1 downto 0);
                  e     :     out std_logic;
                  r     :     out   std_logic);
end fcadd2;

architecture archfcadd2 of fcadd2 is
begin
      sum(0)<=a(0) xor b(0) xor c_in;
      sum(1)<=a(1) xor b(1) xor ((a(0) and b(0)) or (a(0) and c_in) or
(b(0) and c_in));
      e<=(a(1) and b(1)) or ((a(1) or b(1)) and (a(0) and b(0)));
end archfcadd2;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity fcadd3 is
      port(c_in   :     in std_logic;
                  a     :     in    std_logic_vector(2 downto 0);
                  b     :     in    std_logic_vector(2 downto 0);
                  sum   :     out   std_logic_vector(2 downto 0);
                  e     :     out std_logic;
                  r     :     out   std_logic);
end fcadd3;

architecture archfcadd3 of fcadd3 is
signal c1,c2:     std_logic;
begin
```

```
      sum(0)<=a(0) xor b(0) xor c_in;
      sum(1)<=a(1) xor b(1) xor c_in;
      sum(2)<=a(2) xor b(2) xor c_in;
      c1<=(a(0) and b(0)) or ((a(0) or b(0)) and c_in);
      c2<=(a(1) and b(1)) or ((a(1) or b(1)) and (a(0) and b(0))) or
((a(1) or b(1)) and (a(0) or b(0)) and c_in);
      e<=(a(2) and b(2)) or ((a(2) or b(2)) and (a(1) and b(1))) or
((a(2) or b(2)) and (a(1) or b(1)) and (a(0) or b(0)));
      r<=(a(2) or b(2)) and (a(1) or b(1)) and (a(0) or b(0));
end archfcadd3;
```

Ripple Carry Adder

```
Library IEEE;
use IEEE.std_logic_1164.all;

entity add16 is
port(
            a :   in std_logic_vector(15 downto 0);
            b     :     in std_logic_vector(15 downto 0);
            cin   :     in std_logic;
            sum   : out std_logic_vector(15 downto 0);
            cout : out std_logic);
end entity add16;

architecture circuits of add16 is
signal c : std_logic_vector (0 to 14); --internal carry signals
begin
      a0: entity work.pfa port map(a(0),b(0),cin,sum(0),c(0));
      stage: for i in 1 to 14 generate
            as: entity work.pfa port map(a(i),b(i),c(i-1),sum(i),c(i));
      end generate stage;
      a15: entity work.pfa port map(a(15),b(15),c(14),sum(15),cout);
end architecture circuits;
```

Shift Multiplier

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

   entity mul4x4 is
   port(multiplier: in std_logic_vector(7 downto 0);
      multiplicand: in std_logic_vector(7 downto 0);
      product: out std_logic_vector(15 downto 0);
      clock: in std_logic);
   end mul4x4;

  architecture jrb of mul4x4 is

     signal mdreg, mrreg : std_logic_vector(7 downto 0);
      signal adderin : std_logic_vector(7 downto 0);
     signal adderout : std_logic_vector(8 downto 0);
     signal ppreg : std_logic_vector(16 downto 0);
     signal clr_mr ,load_mr ,shift_mr: std_logic;
     signal clr_md ,load_md : std_logic;
     signal clr_pp ,load_pp ,shift_pp: std_logic;

     signal mulstate : natural range 0 to 16;

  begin


  registers: process begin

     if rising_edge(clock) then

     --register to hold multiplicand during multiplication
           if clr_md = '1' then
             mdreg <= (others => '0');
         elsif load_md = '1' then
           mdreg <= multiplicand;
        else
           null;
       end if;
            end if;

     --register/shifter to hold multiplier
     if clr_mr = '1' then
        mrreg <= (others => '0');
     elsif load_mr = '1' then
        mrreg <= multiplier;
     elsif shift_mr = '1' then
        mrreg<='0' & mrreg(7 downto 1);
     else
        null;
     end if;
     --register/shifter accumulates partial product values
     if clr_pp = '1' then
        ppreg <= (others => '0');
```

```
      elsif load_pp = '1' then
         ppreg(16 downto 8) <= adderout; --add to top half
      elsif shift_pp = '1' then
         ppreg<='0' & ppreg(16 downto 1);
      else
         null;
      end if;

  end process;

--adder
if (mrreg(0)=1) then

       adderin <= mdreg;
       adderout <= ('0' & ppreg(15 downto 8)) + ('0' & adderin);
--connect ppreg to product output
       product <= ppreg(15 downto 0);
       else (others => 0);

end if;

  mul_state: process begin
     wait until rising_edge(clock);
     if mulstate < 16 then mulstate <= mulstate + 1;
     else mulstate <= 0;
     end if;
  end process;

  --assign control signal values based on state
  state_decoder: process(mulstate)
  begin
     --assign defaults, all registers refresh
     clr_mr <= '0';
     load_mr <= '0';
     shift_mr <= '0';
     clr_md <= '0';
     load_md <= '0';
     clr_pp <= '0';
     load_pp <= '0';
     shift_pp <= '0';
     if mulstate = 0 then
        load_mr <= '1';
        load_md <= '1';
        clr_pp <= '1';
     elsif mulstate mod 2 = 0 then   --mulstate = 2,4,6,8 ....
        shift_mr <= '1';
        shift_pp <= '1';
     else --mulstate = 1,3,5,7......
        load_pp <= '1';
     end if;
  end process state_decoder;

end jrb;
```

Fold Multiplier

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;


entity mult is
      port(
                  clk : in std_logic;
                  reset : in std_logic;
                  a : in std_logic_vector(15 downto 0);
                  b : in std_logic_vector(15 downto 0);
                  prod : out std_logic_vector(31 downto 0);
                  enter :     in std_logic);
end mult;

architecture jrb of mult is

component sgnext
      port(
                  a : in std_logic_vector(15 downto 0);
                  b : out std_logic_vector(31 downto 0));
end component;

signal count : integer range 0 to 16;--counts cycles
signal aext       :     std_logic_vector(31 downto 0);
signal areg       :     std_logic_vector(31 downto 0);--register for a
signal breg       :     std_logic_vector(15 downto 0);--register for b
signal prodreg          : std_logic_vector(31 downto 0);--register for
product

begin

U0    :     sgnext
            port map(a=>a, b=>aext);

process(clk,reset)
begin
      if (reset='1') then --reset counter and done
            count<= 0;
      elsif (clk'event and clk='1') then
            if (enter='1') then
                  count<=1;
                  prodreg<=(others =>'0');
                  areg<=aext;
                  breg<=b;
            elsif(count/=0) then
                  if (breg(0)='1')then
                        if (count=16) then
                              prodreg<=prodreg - areg;
                        else prodreg<=(prodreg + areg);
                        end if;
```

```
                  end if;
                        if (count=16) then
                              count<=0;
                        else count<=count+1;
                        end if;
            areg<=areg(30 downto 0) & '0';
            breg<='0'& breg(15 downto 1);
            end if;
      end if;
end process;

prod<=prodreg;

end jrb;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity sgnext is
      port(
            a : in std_logic_vector(3 downto 0);
            b : out std_logic_vector(7 downto 0));
end sgnext;

architecture jrb of sgnext is
begin

gen1: for i in natural range 3 downto 0 generate
      b(i)<=a(i);
end generate;

gen2: for i in natural range 7 downto 4 generate
      b(i)<=a(3);
end generate;
end jrb;
```

Booth Parallel Multiplier

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mul16 is  --16 by 16 two's comp multiplier
port(
      a     :     in std_logic_vector(15 downto 0);
      b     :     in std_logic_vector(15 downto 0);
      p     :     out std_logic_vector(31 downto 0));
end entity mul16;

architecture circuits of mul16 is
signal zer : std_logic_vector(15 downto 0) :=x"0000"; --zeros
signal mul0 : std_logic_vector(2 downto 0);
subtype word is std_logic_vector(15 downto 0);
type ary is array (0 to 7) of word;
signal s : ary;
begin
      mul0<=a(1 downto 0) & '0';
      a0: entity work.booth port map(mul0,b,zer,s(0),p(1 downto 0));
      a1: entity work.booth port map(a(3 downto 1),b,s(0),s(1),p(3
downto 2));
      a2: entity work.booth port map(a(5 downto 3),b,s(1),s(2),p(5
downto 4));
      a3: entity work.booth port map(a(7 downto 5),b,s(2),s(3),p(7
downto 6));
      a4: entity work.booth port map(a(9 downto 7),b,s(3),s(4),p(9
downto 8));
      a5: entity work.booth port map(a(11 downto 9),b,s(4),s(5),p(11
downto 10));
      a6: entity work.booth port map(a(13 downto 11),b,s(5),s(6),p(13
downto 12));
      a7: entity work.booth port map(a(15 downto 13),b,s(6),p(31 downto
16),p(15 downto 14));
end architecture circuits;


library IEEE;
use IEEE.std_logic_1164.all;

entity booth is --special adder for signed mult
port(
      a     :     in std_logic_vector(2 downto 0);  --booth multiplier
      b     :     in std_logic_vector(15 downto 0); --multiplicand
      sum_in      :     in std_logic_vector(15 downto 0);
      sum_out     :     out std_logic_vector(15 downto 0);
      prod  :     out std_logic_vector(1 downto 0));
end entity booth;

architecture circuits of booth is
subtype word is std_logic_vector(15 downto 0);
signal bb : word;
signal psum : word;
signal b_bar: word;
```

```
signal two_b : word;
signal two_b_bar : word;
signal cout : std_logic;
signal cin : std_logic;
signal topbit : std_logic;
signal topout : std_logic;
signal nc1  : std_logic;

begin
      two_b<=b(14 downto 0)&'0';
      b_bar<=not b;
      two_b_bar<=b_bar(14 downto 0)&'0';
      bb<=b when a="001" or a="010"     --5 intput multiplexor
            else two_b when a="011"
            else two_b_bar when a="100"
            else b_bar when a="101" or a="110"
            else x"0000";
      cin<='1' when a="001" or a="101" or a="110"
            else '0';
      topbit<=b(15) when a="001" or a="010" or a="011"
            else b_bar(15) when a="100" or a="101" or a="110"
            else '0';
a1:entity work.add16 port map(sum_in,bb,cin,psum,cout);
a2:entity work.pfa port map(sum_in(15),topbit,cout,topout,nc1);
      sum_out(13 downto 0)<=psum(15 downto 2);
      sum_out(15)<=topout;
      sum_out(14)<=topout;
      prod<=psum(1 downto 0);
end architecture circuits;

Library IEEE;
use IEEE.std_logic_1164.all;

entity pfa is
port(
            a     :     in std_logic;
            b     :     in std_logic;
            cin : in std_logic;
            s     :     out std_logic;
            cout  :     out std_logic);
end entity pfa;

architecture circuits of pfa is
begin
      s<=a xor b xor cin;
      cout<=(a and b) or (a and cin) or (b and cin);
end architecture circuits;
```

Data Management

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity data_manage is --data management for DSP Processor
port(
            clk   : in std_logic;
            X_in : in std_logic_vector(15 downto 0);
            count_out : out std_logic_vector(2 downto 0);--used to check
the counting cycles to see if clock is working

            Y_out : out std_logic_vector(15 downto 0));
end entity data_manage;

architecture circuits of data_manage is
signal count : std_logic_vector(2 downto 0) := "000"; --counter for mux
signal a1 : std_logic_vector(15 downto 0) :="0000000000000000";  --a1
coefficient
signal a2 : std_logic_vector(15 downto 0) :="1001100001010010";  --a2
coefficient
signal b0 : std_logic_vector(15 downto 0) :="0000000000000000";  --b0
coefficent
signal b1 : std_logic_vector(15 downto 0) :="0111001111010111";  --b1
coefficent
signal b2 : std_logic_vector(15 downto 0) :="0111001111010111";  --b2
coefficent
signal temp_reg : std_logic_vector(15 downto 0); --temp register to
store summed values
signal wn : std_logic_vector(15 downto 0):="0000000000000000";  --Wn
register
signal wn_1 : std_logic_vector(15 downto 0):="0000000000000000";  --Wn-1
register
signal wn_2 : std_logic_vector(15 downto 0):="0000000000000000";  --Wn-2
register
signal mul1 : std_logic_vector(15 downto 0);  --signals sent to
multiplier
signal mul2 : std_logic_vector(15 downto 0);
signal add  :     std_logic_vector(15 downto 0);
signal product    :     std_logic_vector(31 downto 0);--signal from
multiplier
signal temp: std_logic_vector(16 downto 0);
signal tempcarry: std_logic_vector(15 downto 0);
begin
U0: entity work.mul16 port map(mul1,mul2,product);


count_out<=count;


Clock: process(clk)
begin
      if rising_edge (clk) then
```

```
            if (count/="110") then
                  count<=count+"001";
            else count<="000";  -- resets count
            end if;
      end if;
 end process;

Mux: process(count)
begin
 if (count="001") then
      mul1<=a1;
      mul2<=wn_1;
      add<=x_in;
      elsif (count="010") then
            mul1<=a2;
            mul2<=wn_2;
            add<=temp_reg;
      elsif (count="011") then
            wn<=temp_reg;--stores wn
            mul1<=b2;
            mul2<=wn_2;
            add<=x"0000";
      elsif (count="100") then
            mul1<=b1;
            mul2<=wn_1;
            add<=temp_reg;
      elsif (count="101") then
            mul1<=b0;
            mul2<=wn;
            add<=temp_reg;
            wn_2<=wn_1;
      elsif (count="110") then
--          y_out<=temp_reg;
            Y_out<=(not temp_reg(15)) & temp_reg(14 downto 0);
            Wn_1<=wn;   --stores wn to wn-1
      end if;
end process;


adderprocess:process (clk)--adder process used to ensure it doesn't add
until mult is done
begin                                                     --and also
used to correct the extra sign bit problem if mul1 and mul2 are the same
      if falling_edge(clk) then
            if (mul1=mul2) then
                  temp<=('0' & add)+('0' & product(29 downto 14));

                  tempcarry<=('0' & add(14 downto 0))+('0' & product(28
downto 14));
            else
                  temp<=('0' & add)+('0' & product(30 downto 15));

                  tempcarry<=('0' & add(14 downto 0))+('0' & product(29
downto 15));

            end if;
      end if;
```

```
end process adderprocess;

processcorrect:process(temp)--process used to correct overflow prob
begin
      if (temp(16)<tempcarry(15)) then
            temp_reg<="0111111111111111";--sets to highest positive
      elsif (temp(16)>tempcarry(15)) then
            temp_reg<="1000000000000000";--sets to lowest negative

      else
            temp_reg<=temp(15 downto 0);
      end if;
end process processcorrect;



end architecture circuits;
```

Expansion Board Code

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

entity expansion is
port(
            clk1 : in std_logic; --clock for dsp
--          clk2  :     in std_logic; --clock for 74LS139
            X_in  :     in std_logic_vector(9 downto 0);
            count_out : out std_logic_vector(1 downto 0); --output for
74ls139
            Dec_out : out std_logic_vector(3 downto 0)); --output to
display driver
            constant limit : integer := 300;
end entity expansion;

architecture circuits of expansion is
signal count      :     std_logic_vector(1 downto 0) := "00";  --counter
for 74ls139
signal countclk : integer := 0;
signal y_out :    std_logic_vector(15 downto 0);  --output from dsp
signal dec_temp : std_logic_vector(3 downto 0); --temp reg for inside
process
signal X_adj      :     std_logic_vector(15 downto 0);  --used to make
X_in 16 bits
signal clkadj : std_logic :='0'; ---used to slow down the 20ns internal
clock
signal clkadjtemp : std_logic :='0';
begin
      X_adj<=(X_in & "000000");  --add 6 0's to right of X_in
      clkadj<=clkadjtemp;
U0: entity work.data_manage port map (clkadj, X_adj, y_out);

      count_out<=count;
      Dec_out<=dec_temp;


process(clk1)
begin
if (clk1='1' and clk1'event) then
      if countclk<limit then
            countclk <= countclk+1;
      elsif countclk=limit then
            countclk<=0;
            clkadjtemp<=not clkadjtemp;
            end if;
end if;



end process;
```

```
process(clk1)
begin
      if rising_edge(clk1) then
            if (count="11") then
                  count<= "00";
            else count<=count + "01";
            end if;
      end if;
end process;

process(count)  --mux for display decoder
begin
            if (count="00") then
                  dec_temp<=Y_out(15 downto 12);
            elsif (count="01") then
                  dec_temp<=Y_out(7 downto 4);
            elsif (count="10") then
                  dec_temp<=Y_out(11 downto 8);
            elsif (count="11") then
                  dec_temp<=Y_out(3 downto 0);
            end if;
end process;


end architecture circuits;
```