Chapter 2

ASIC Design Flow

2.1 INTRODUCTION TO ASICs

An ASIC (pronounced "a-sick"; bold typeface defines a new term) is an application-specific integrated circuit —at least that is what the acronym stands for. Before we answer the question of what that means we first look at the evolution of the silicon chip or integrated circuit (IC). Figure 2.1(a) shows an IC package (this is a pin-grid array, or PGA, shown upside down; the pins will go through holes in a printed-circuit board). People often call the package a chip, but, as you can see in Figure 2.1(b), the silicon chip itself (more properly called a die) is mounted in the cavity under the sealed lid. A PGA package is usually made from a ceramic material, but plastic packages are also common.

The physical size of a silicon die varies from a few millimeters on a side to over 1 inch on a side, but instead we often measure the size of an IC by the number of logic gates or the number

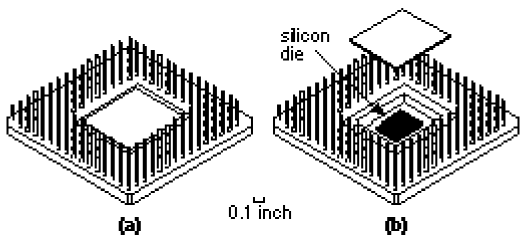


FIGURE 2.1 An integrated circuit (IC). (a) A pin-grid array (PGA) package. (b) The silicon die or chip is under the package lid.

of transistors that the IC contains. As a unit of measure a gate equivalent corresponds to a two-input NAND gate (a circuit that performs the logic function, F = A • B). Often we just use the term gates instead of gate equivalents when we are measuring chip size—not to be confused with the gate terminal of a transistor. For example, a 100 k-gate IC contains the equivalent of 100,000 two-input NAND gates.

A modern submicron CMOS process is now just as complicated as a submicron bipolar or BiCMOS (a combination of bipolar and CMOS) process. However, CMOS ICs have established a dominant position, are manufactured in much greater volume than any other technology, and therefore, because of the economy of scale, the cost of CMOS ICs is less than a bipolar or BiCMOS IC for the same function. Bipolar and BiCMOS ICs are still used for special needs. For example, bipolar technology is generally capable of handling higher voltages than CMOS. This makes bipolar and BiCMOS ICs useful in power electronics, cars, telephone circuits, and so on.

With the advent of VLSI in the 1980s engineers began to realize the advantages of designing an IC that was customized or tailored to a particular system or application rather than using standard ICs alone. Microelectronic system design then becomes a matter of defining the functions that you can implement using standard ICs and then implementing the remaining logic functions (sometimes called glue logic) with one or more custom ICs . As VLSI became possible you could build a system from a smaller number of components by combining many standard ICs into a few custom ICs. Building a microelectronic system with fewer ICs allows you to reduce cost and improve reliability.

2.2 Design Flow

Figure 2.2 shows the sequence of steps to design an ASIC; we call this a design flow . The steps are listed below (numbered to correspond to the labels in Figure 1.10) with a brief description of the function of each step.

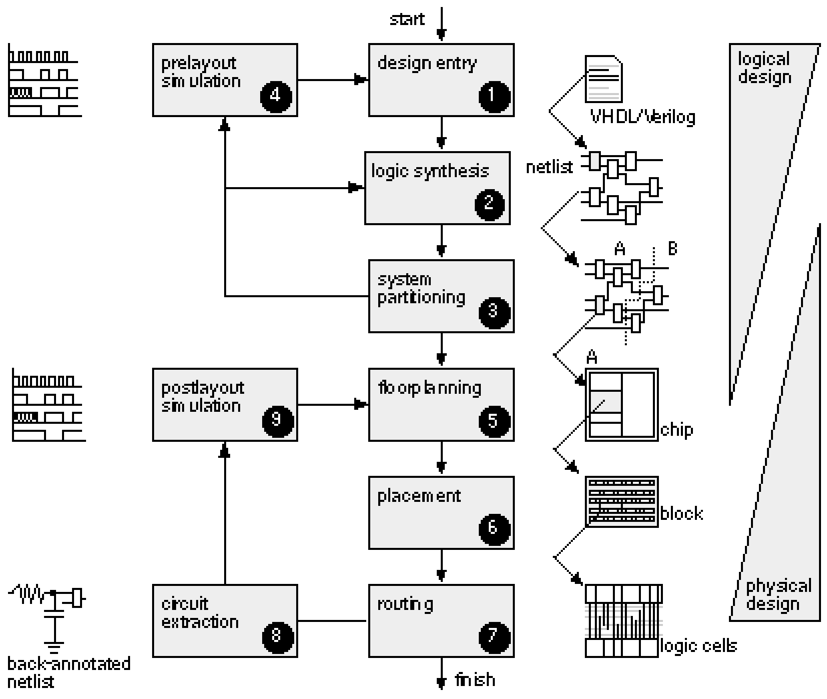


FIGURE 2.2 ASIC design flow.

1. Design entry. Enter the design into an ASIC design system, either using a hardware description language (HDL) or schematic entry .
2. Logic synthesis. Use an HDL (VHDL or Verilog) and a logic synthesis tool to produce a netlist —a description of the logic cells and their connections.
3. System partitioning. Divide a large system into ASIC-sized pieces.
4. Prelayout simulation. Check to see if the design functions correctly.
5. Floorplanning. Arrange the blocks of the netlist on the chip.
6. Placement. Decide the locations of cells in a block.
7. Routing. Make the connections between cells and blocks.
8. Extraction. Determine the resistance and capacitance of the interconnect.

9. Postlayout simulation. Check to see the design still works with the added loads of the interconnect.

Steps 1–4 are part of logical design , and steps 5–9 are part of physical design . There is some overlap. For example, system partitioning might be considered as either logical or physical design. To put it another way, when we are performing system partitioning we have to consider both logical and physical factors.

2.3 ASIC Cell Libraries

The cell library is the key part of ASIC design. For non-programable ASIC you have three choices: the ASIC vendor (the company that will build your ASIC) will supply a cell library, or you can buy a cell library from a third-party library vendor , or you can build your own cell library.

However created, each cell in an ASIC cell library must contain the following:

- A physical layout
- A behavioral model
- A Verilog/VHDL model
- A detailed timing model
- A test strategy
- A circuit schematic
- A cell icon
- A wire-load model
- A routing model

2.4 Viterbi Decoder Project Design

This section describes an ASIC design for a Viterbi decoder using hardware descriptive language as a method for design entry.

2.4.1 Viterbi Encoder

Viterbi encoding is widely used for satellite and other noisy communications channels. There are two important components of a channel using Viterbi encoding: the Viterbi encoder (at the transmitter) and the Viterbi decoder (at the receiver). A Viterbi encoder includes extra information in the transmitted signal to reduce the probability of errors in the received signal that may be corrupted by noise. I shall describe an encoder in which every two bits of a data stream are encoded into three bits for transmission. The ratio of input to output information in an encoder is the rate of the encoder; this is a rate 2/3 encoder. The following equations relate the three encoder output bits (Y_n^2 , Y_n^1 , and Y_n^0) to the two encoder input bits (X_n^2 and X_n^1) at a time nT:

$$Y_n^2 = X_n^2$$

$$Y_n^1 = X_n^1 \text{ xor } X_{n-2}^1$$

$$Y_n^0 = X_{n-1}^1$$

We can write the input bits as a single number. Thus, for example, if $X_n^2 = 1$ and $X_n^2 = 0$, we can write $X_n = 2$. The state machine with two memory elements; which describes the encoder; for the two last input values for X_n^1 : X_{n-1}^1 and X_{n-2}^1 . These two state variables define four states: $\{X_{n-1}^1, X_{n-2}^1\}$, with $S_0 = \{0, 0\}$, $S_1=\{1, 0\}$, S2 = {0, 1}, and S3 = {1, 1}. The 3-bit output Y_n is a function of the state and current 2-bit input X_n .

The following comment is that which I have included in my HDL file describing the rate 2/3 encoder. This model uses two D flip-flops as the state register. When reset (using active-high input signal res) the encoder starts in state S_0.

<u>*Notification:*</u> In the following comment I represent Y_n^2 by Y2N , to be more convineint with the HDL syntax.

/***/

/* module viterbi_encode */

/***/

*/ This is the encoder. X2N (msb) and X1N form the 2-bit input message, XN. Example: if X2N=1, X1N=0, then XN=2. Y2N (msb), Y1N, and Y0N form the 3-bit encoded signal, YN (for a total constellation of 8 PSK signals that will be transmitted). The encoder uses a state machine with four states to generate the 3-bit output, YN, from the 3-bit input, XN. Example: the repeated input sequence XN = (X2N, X1N) = 0,1, 2, 3 produces the repeated output sequence YN = (Y2N, Y1N, Y0N) = 1, 0, 5, 4. */

/***/

Figure 11.3 shows the state diagram for this encoder. The first four rows of Table 2.1 show the four different transitions that can be made from state S_0 . For example, if we reset the encoder and the input is $X_n = 3$ ($X_n^2 = 1$ and $X_n^1 = 1$), then the output will be $Y_n = 6$ ($Y_n^2 = 1$, $Y_n^1 = 1$, $Y_n^0 = 0$) and the next state will be S_1 .

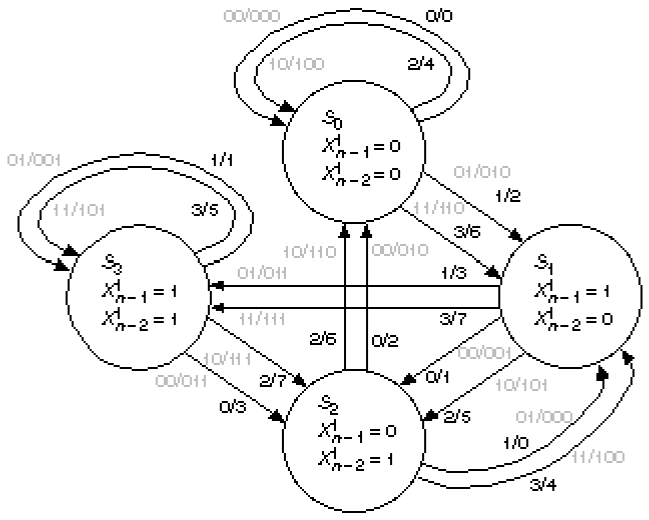


FIGURE 2.3 A state diagram for a rate 2/3 Viterbi encoder. The inputs and outputs are shown in binary as $X_n^2\ X_n^1$ / $Y_n^2 Y_n^1 Y_n^0$, and in decimal as X_n/ Y_n .

					Outputs				
Present state	**Inputs**		**State variables**		Y_n^2	Y_n^1	Y_n^0	**Next state**	
	X_n^2	X_n^1	X_{n-1}^1	X_{n-2}^1	X_n^2	$= X_n^1$ xor X_{n-2}^1	$= X_{n-1}^1$	$\{X_{n-1}^1, X_{n-2}^1\}$	
S_0	0	0	0	0	0	0	0	00	S_0
S_0	0	1	0	0	0	1	0	10	S_1
S_0	1	0	0	0	1	0	0	00	S_0
S_0	1	1	0	0	1	1	0	10	S_1
S_1	0	0	1	0	0	0	1	01	S_2
S_1	0	1	1	0	0	1	1	11	S_3
S_1	1	0	1	0	1	0	1	01	S_2
S_1	1	1	1	0	1	1	1	11	S_3
S_2	0	0	0	1	0	1	0	00	S_0
S_2	0	1	0	1	0	0	0	10	S_1
S_2	1	0	0	1	1	1	0	00	S_0
S_2	1	1	0	1	1	0	0	10	S_1
S_3	0	0	1	1	0	1	1	01	S_2
S_3	0	1	1	1	0	0	1	11	S_3
S_3	1	0	1	1	1	1	1	01	S_2
S_3	1	1	1	1	1	0	1	11	S_3

TABLE 2.1 State table for the rate 2/3 Viterbi encoder.

As an example, the repeated encoder input sequence Xn = 0, 1, 2, 3, ... produces the encoder output sequence Yn = 1, 0, 5, 4, ... repeated. Table 2.2 shows the state transitions for this sequence, including the initialization steps.

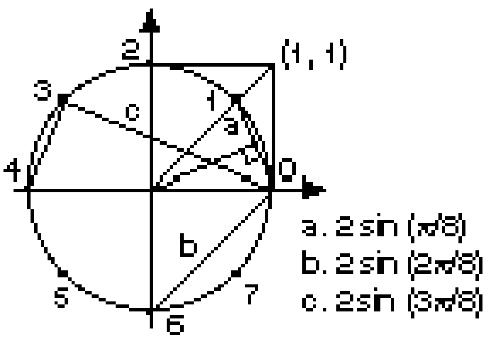


FIGURE 2.4 The signal constellation for an 8 PSK (phase-shift keyed) code.

Time	**Inputs**		State variables		Outputs			**Present state**	**Next state**
ns	X_n^2	X_n^1	X_{n-1}^1	X_{n-2}^1	Y_n^2	Y_n^1	Y_n^0		
0	1	1	x	x	1	x	x	$S_?$	$S_?$
10	1	1	0	0	1	1	0	S_0	S_1
50	0	0	1	0	0	0	1	S_1	S_2
150	0	1	0	1	0	0	0	S_2	S_1
250	1	0	1	0	1	0	1	S_1	S_2
350	1	1	0	1	1	0	0	S_2	S_1
450	0	0	1	0	0	0	1	S_1	S_2
550	0	1	0	1	0	0	0	S_2	S_1
650	1	0	1	0	1	0	1	S_1	S_2
750	1	1	0	1	1	0	0	S_2	S_1
850	0	0	1	0	0	0	1	S_1	S_2
950	0	1	0	1	0	0	0	S_2	S_1

TABLE 2.2 A sequence of transmitted signals for the rate 2/3 Viterbi encoder.

Next we transmit the eight possible encoder outputs (Y_n = 0-7) as **signals** over our noisy communications channel (perhaps a microwave signal to a satellite) using the **signal constellation** show n in Figure 2.4. Typically this is done using **phase-shift keying (PSK)** with each signal position corresponding to a different phase shift in the transmitted carrier signal.

2.4.2 The Received Signal

The noisy signal enters the receiver. It is now our task to discover which of the eight possible signals were transmitted at each time step. First we calculate the distance of each received signal from each of the known eight positions in the signal constellation. Table 2.3 shows the distances between signals in the 8PSK constellation. We are going to assume that there is no noise in the channel to illustrate the operation of the Viterbi decoder, so that the distances in Table 2.3 represent the possible distance measures of our received signal from the 8PSK signals.

The distances, X, in the first column of Table 2.3 are the geometric or algebraic distances. We measure the **Euclidean distance**, $E = X^2$ shown as B (the binary quantized value of E) in Table 2.3. The rounding errors that result from conversion to fixed-width binary are **quantization errors** and are important in any practical implementation of the Viterbi decoder. The effect of the quantization error is to add a form of noise to the received signal.

The following code models the receiver section that digitizes the noisy analog received signal and computes the binary distance measures. Eight binary-distance measures, in0-in7 , are generated each time a signal is received. Since each of the distance measures is 3 bits wide, there are a total of 24 bits ($8 \infty 3$) that form the digital inputs to the Viterbi decoder.

Signal	Algebraic distance from signal 0	**X** = Distance from signal 0	Euclidean distance **E = X^2**	**B** = binary quantized value of **E**	**D** = decimal value of B	Quantization error **Q** = **D** - 1.75 **E**
0	2 sin (0 π / 8)	0.00	0.00	000	0	0
1	2 sin (1 π / 8)	0.77	0.59	001	1	-0.0325
2	2 sin (2 π / 8)	1.41	2.00	100	4	0.5
3	2 sin (3 π / 8)	1.85	3.41	110	6	0.0325
4	2 sin (4 π / 8)	2.00	4.00	111	7	0
5	2 sin (5 π / 8)	1.85	3.41	110	6	0.0325
6	2 sin (6 π / 8)	1.41	2.00	100	4	0.5
7	2 sin (7 π / 8)	0.77	0.59	001	1	-0.0325

TABLE 2.3 Receiver distance measures for an example transmission sequence.

```
/*****************************************************************************/
/* Module Viterbi_distances                                                   */
/*****************************************************************************/
/* This module simulates the front end of a receiver. Normally the received analog signal (with
noise) is converted into a series of distance measures from the known eight possible transmitted
PSK signals: s0,...,s7. We are not simulating the analog part or noise in this version, so we just
take the digitally encoded 3-bit signal, Y, from the encoder and convert it directly to the
distance measures.
 d[N] is the distance from signal = N to signal = 0
 d[N] = (2*sin(N*PI/8))**2 in 3-bit binary (on the scale 2=100)
 Example: d[3] = 1.85**2 = 3.41 = 110
 inN is the distance from signal = N to encoder signal.
 Example: in3 is the distance from signal = 3 to encoder signal.
 d[N] is the distance from signal = N to encoder signal = 0.
 If encoder signal = J, shift the distances by 8-J positions.
```

```
Example: if signal = 2, in0 is d[6], in1 is D[7], in2 is D[0], etc. */
/*****************************************************************************/
```

As an example, Table 2.4 shows the distance measures for the transmitted encoder output sequence Y_n = 1, 0, 5, 4, ... (repeated) corresponding to an encoder input of X_n = 0, 1, 2, 3, ... (repeated).

Time Ns	**Input Xn**	Output **Yn**	**Present state**	**Next state**	in0	in1	in2	in3	in4	in5	in6	in7
0	3	x	$S_?$	$S_?$	x	x	x	x	x	x	x	x
10	3	6	S_0	S_1	4	6	7	6	4	1	0	1
50	0	1	S_1	S_2	1	0	1	4	6	7	6	4
150	1	0	S_2	S_1	0	1	4	6	7	6	4	1
250	2	5	S_1	S_2	6	7	6	4	1	0	1	4
350	3	4	S_2	S_1	7	6	4	1	0	1	4	6
450	0	1	S_1	S_2	1	0	1	4	6	7	6	4
550	1	0	S_2	S_1	0	1	4	6	7	6	4	1
650	2	5	S_1	S_2	6	7	6	4	1	0	1	4
750	3	4	S_2	S_1	7	6	4	1	0	1	4	6
850	0	1	S_1	S_2	1	0	1	4	6	7	6	4
950	1	0	S_2	S_1	0	1	4	6	7	6	4	1

TABLE 2.4 Receiver distance measures for an example transmission sequence.

2.4.3 Decoder Model

Verilog code for a Viterbi decoder. The decoder assumes a rate 2/3 encoder, 8 PSK modulation, and trellis coding. The Viterbi module contains nine submodules: subset_decode, metric, compute_metric, compare_select, reduce, pathin, path_memory, and output_decision. The decoder accepts eight 3-bit measures of ||r-si||**2 and, after an initial delay of thirteen clock cycles, the output is the best estimate of the signal transmitted. The distance measures are the Euclidean distances between the received signal r (with noise) and each of the (in this case eight) possible transmitted signals s0 to s7.

<u>D Flip Flop</u>

The Viterbi decoder model presented in this section is written for both simulation and synthesis. The Viterbi decoder makes extensive use of vector D flip-flops (registers).The used flip-flop models supplied with the synthesis tool such as the following:

```
asDff #(3) subout0(in0, sub0, clk, reset);
```

The asDff is a model in the Compass ASIC Synthesizer standard component library. This statement triggers the synthesis of three D flip-flops, with an input vector ina (with a range of three) connected to the D inputs, an output vector sub0 (also with a range of three) connected to the Q flip-flop outputs, a common scalar clock signal, clk, and a common scalar reset signal. The disadvantage of this approach is that the names, functional behavior, and interfaces of the standard components are different for every software system.

```
/******************************************************************************/
/*      Module dff                                                            */
/******************************************************************************/
/* A D flip-flop module. */
module dff(D,Q,Clock,Reset); // N.B. reset is active-low.
output Q; input D,Clock,Reset;
parameter CARDINALITY = 1; reg [CARDINALITY-1:0] Q;
```

```
wire [CARDINALITY-1:0] D;
always @(posedge Clock) if (Reset !== 0) #1 Q = D;
always begin wait (Reset == 0); Q = 0; wait (Reset == 1); end
endmodule
/*************************************************************************/
```

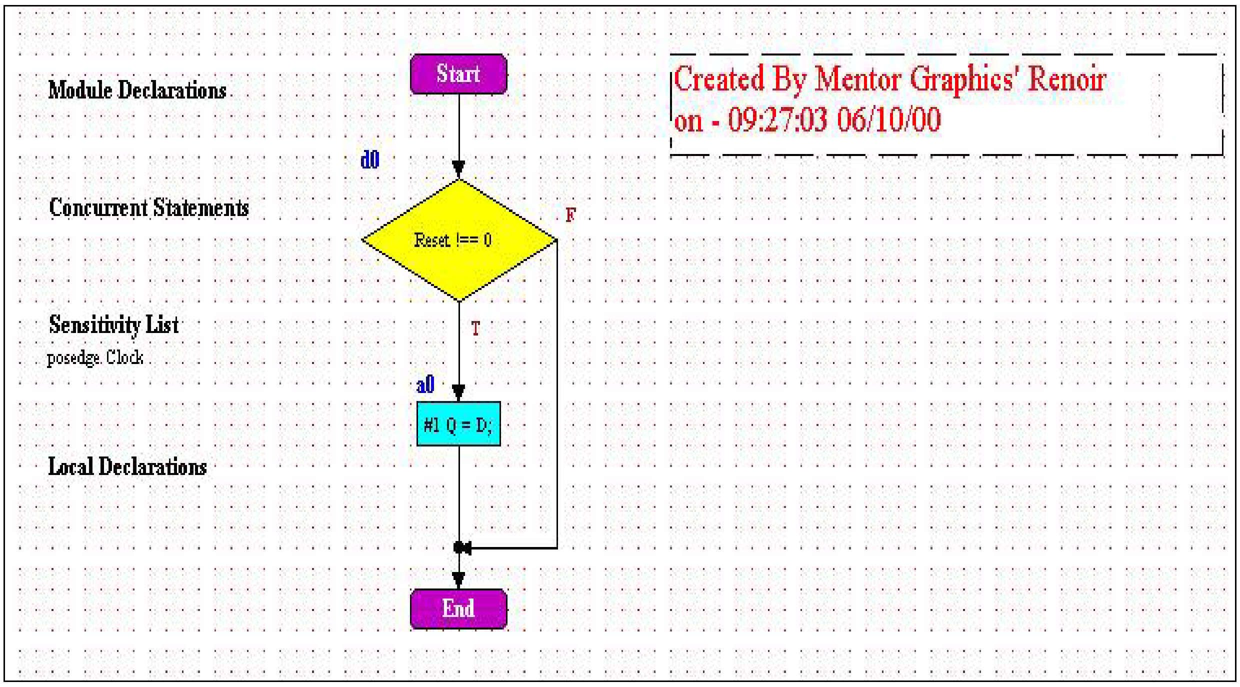


Figure 2.5 Flow chart describing the behavior of the D flip flop.

Viterbi Decoder Submodules

1) Module Subset_decode

```
/*************************************************************************/
/* Module subset_decode                                                   */
/*************************************************************************/
```

/* This module chooses the signal corresponding to the smallest of each set {||r-s0||**2,||r-s4||**2}, {||r-s1||**2, ||r-s5||**2}, {||r-s2||**2,||r-s6||**2}, {||r-s3||**2,||r-s7||**2}. Therefore there are eight input signals and four output signals for the distance measures. The signals sout0, ..., sout3 are used to control the path memory. The statement dff #(3) instantiates a vector array of 3 D flip-flops. */

/***/

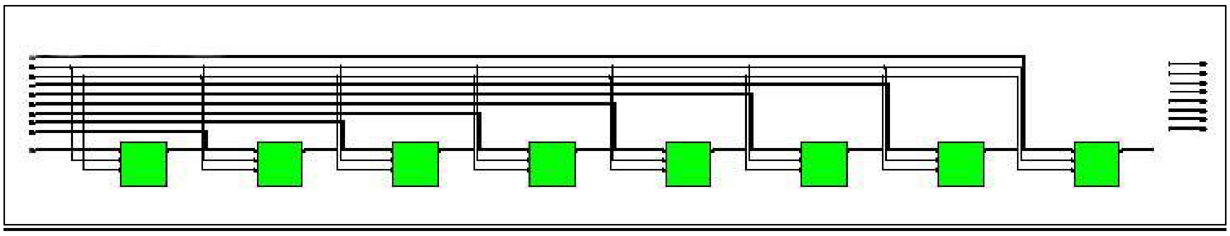


Figure 2.6 Block diagram of the subset_decode module.

2) Module Metric

/**/

/* Module metric */

/**/

/* The registers created in this module (using D flip-flops) store the four path metrics. Each register is 5 bits wide. The statement dff #(5) instantiates a vector array of 5 flip-flops. */

/**/

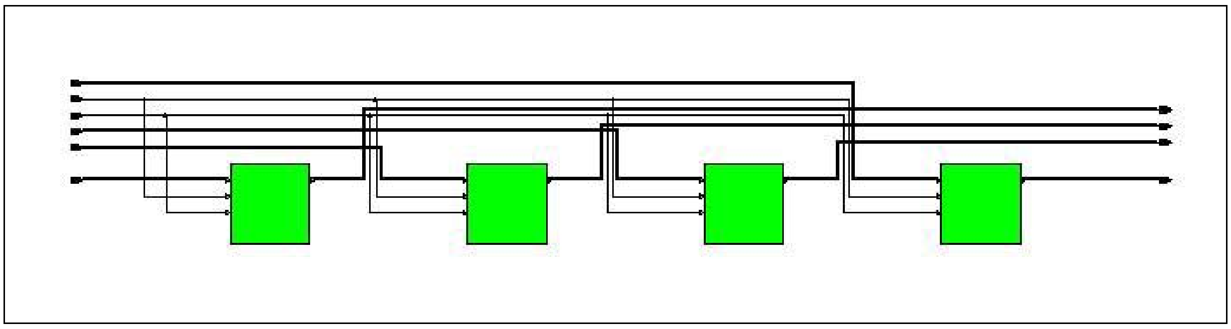


Figure 2.7 Block diagram of the metric module.

3) Module Compute_metric

```
/***************************************************************************/

/*   Module compute_metric                                                 */

/***************************************************************************/
```

/* This module computes the sum of path memory and the distance for each path entering a state of the trellis. For the four states, there are two paths entering it; therefore eight sums are computed in this module. The path metrics and output sums are 5 bits wide. The output sum is bounded and should never be greater than 5 bits for a valid input signal. The overflow from the sum is the error output and indicates an invalid input signal.*/

```
/***************************************************************************/
```

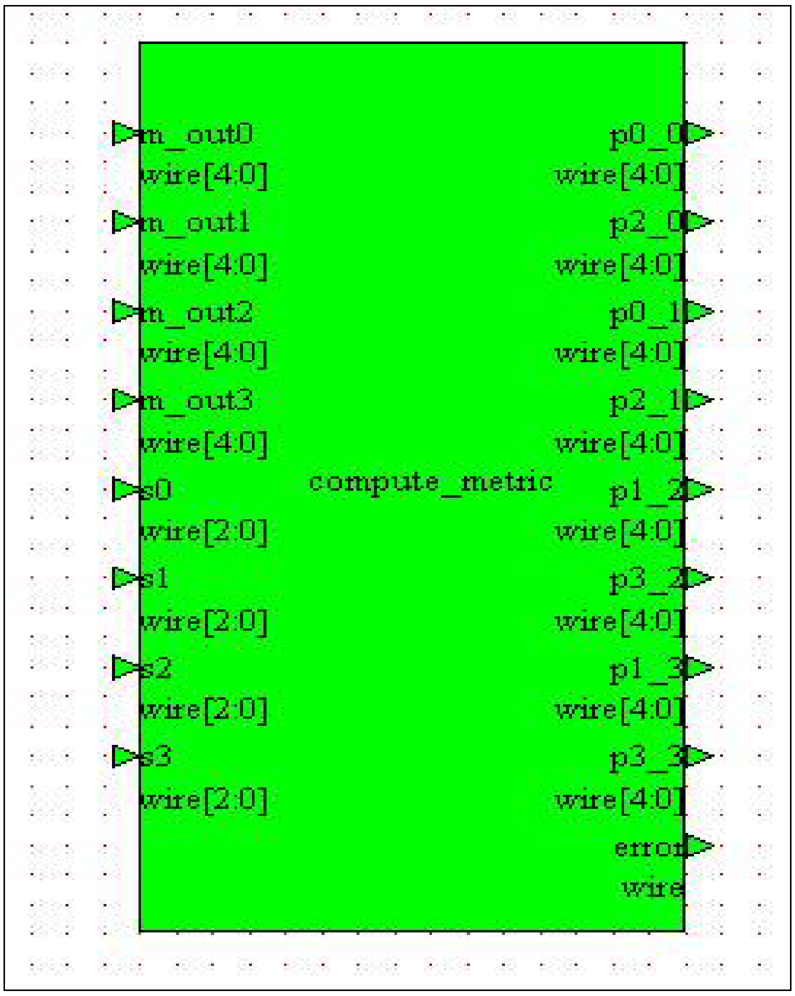


Figure 2.8 Block diagram of the compute_metric module.

4) Module Compare_select

/**/

/* Module compare_select */

/**/

/* This module compares the summations from the compute_metric module and selects the metric and path with the lowest value. The output of this module is saved as the new path metric for each state. The ACS output signals are used to control the path memory of the decoder. */

/**/

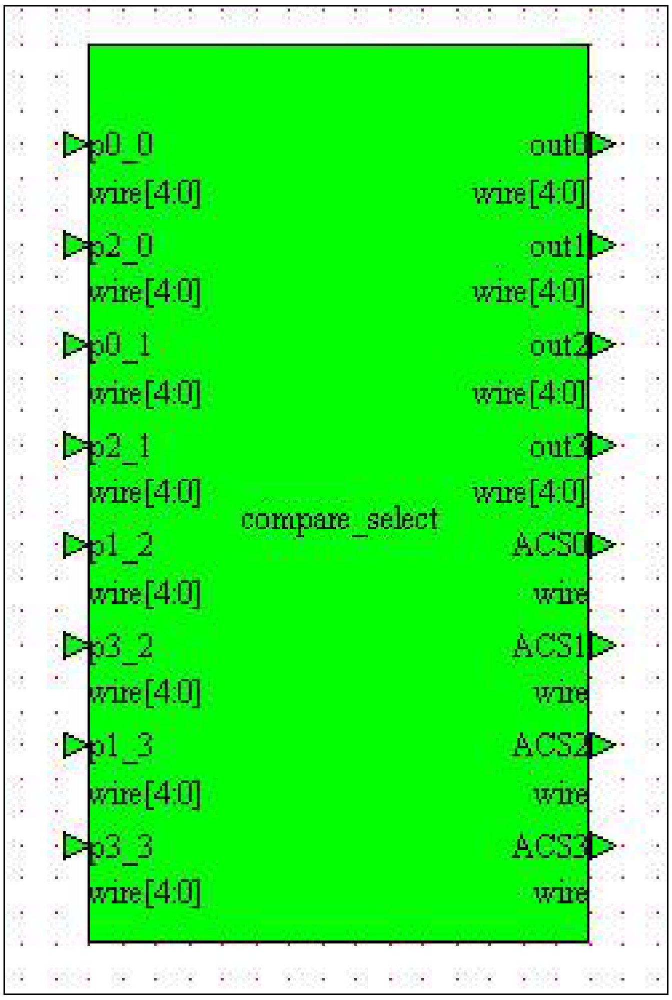


Figure 2.9 Block diagram of the compare_select module

5) Module Path

```
/*****************************************************************************/

/*   Module path                                                              */

/*****************************************************************************/

/* This is the basic unit for the path memory of the Viterbi decoder. It consists of four 3-bit D
flip-flops in parallel. There is a 2:1 mux at each D flip-flop input. The statement dff #(12)
instantiates a vector array of 12 flip-flops. */

/*****************************************************************************/
```

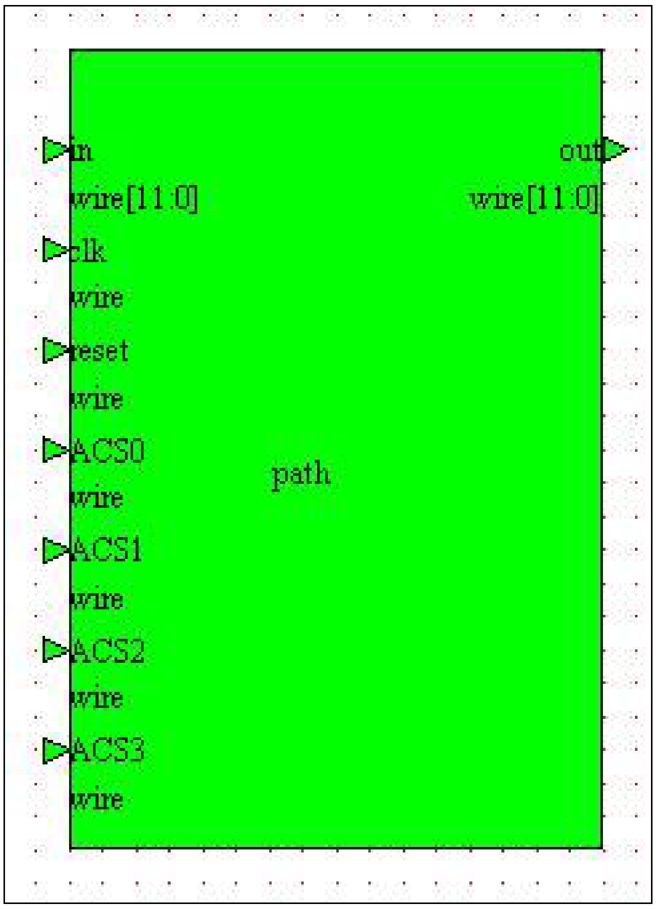


Figure 2.10 Block diagram of the path module

6) Module Path memory

```
/************************************************************/
/*   Module path_memory                                     */
/************************************************************/
/* This module consists of an array of memory elements (D flip-
flops) that store and shift the path memory as new signals are
added to the four paths (or four most likely sequences of
signals). This module instantiates 11 instances of the path
module. */
/************************************************************/
```

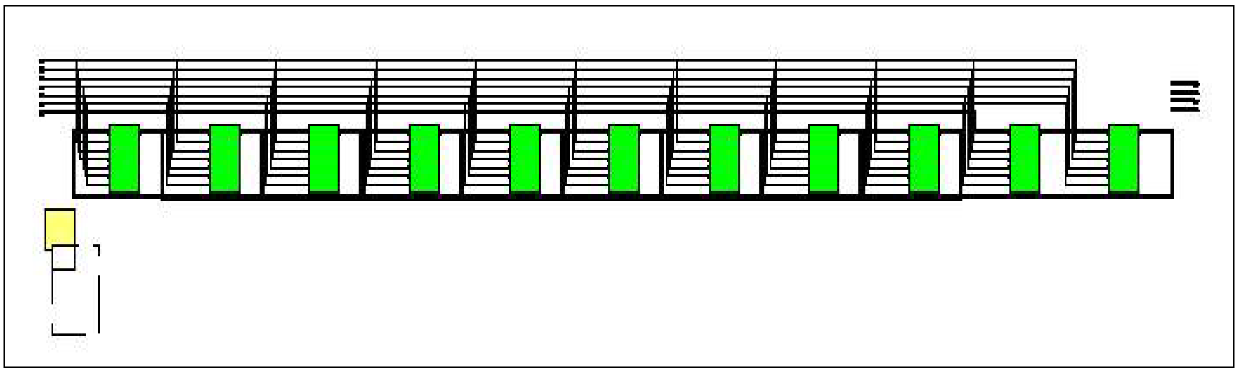
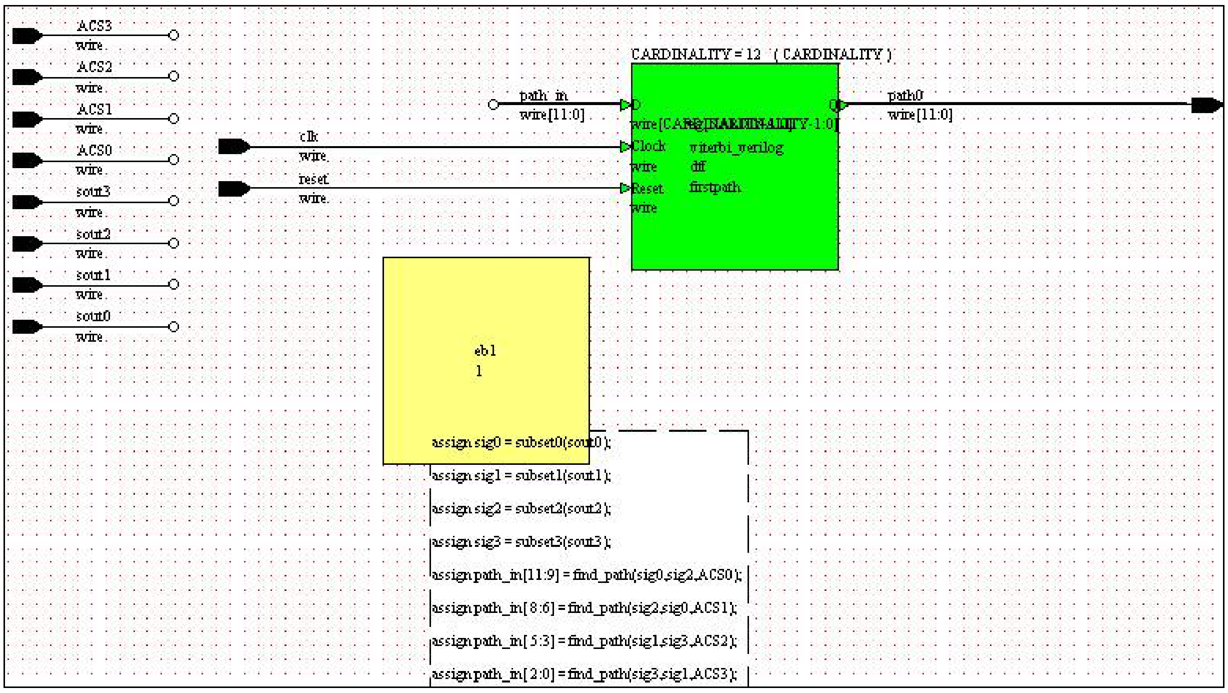


Figure 2.11 Block diagram of the path_memory module.

7) Module Pathin

```
/****************************************************************/
/*   Module pathin                                              */
/****************************************************************/
/* This module determines the input signal to the path for each
of the four paths. Control signals from the subset decoder and
compare select modules are used to store the correct signal.
The statement dff #(12) instantiates a vector array of 12 flip-
flops. */
/****************************************************************/
```



```
assign sig0 = subset0(sout0);
assign sig1 = subset1(sout1);
assign sig2 = subset2(sout2);
assign sig3 = subset3(sout3);
assign path_in[11:9] = find_path(sig0,sig2,ACS0);
assign path_in[8:6] = find_path(sig2,sig0,ACS1);
assign path_in[5:3] = find_path(sig1,sig3,ACS2);
assign path_in[2:0] = find_path(sig3,sig1,ACS3);
```

Figure 2.12 Block diagram of the pathin module.

<u>8) Module Output_decision</u>

```
/***********************************************************/
/*   Module Output_decision                                */
/***********************************************************/
/* This module decides the output signal based on the path that
corresponds to the smallest metric. The control signal comes
from the reduce module. */
/***********************************************************/
```

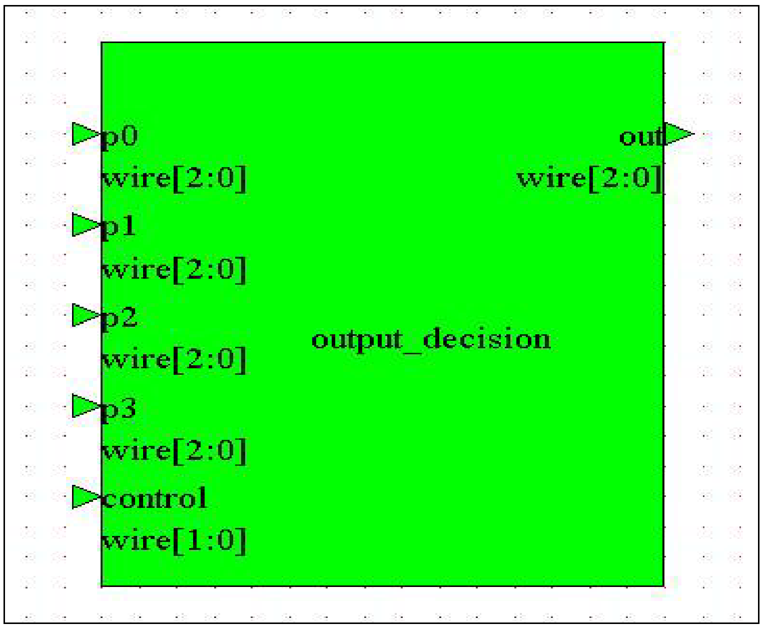


Figure 2.13 Block diagram of the output_decision module.

9) Module Reduce

```
/***************************************************************/
/*   Module reduce                                             */
/***************************************************************/
/* This module reduces the metrics after the addition and
compare operations. This algorithm selects the smallest metric
and subtracts it from all the other metrics. */
/***************************************************************/
```

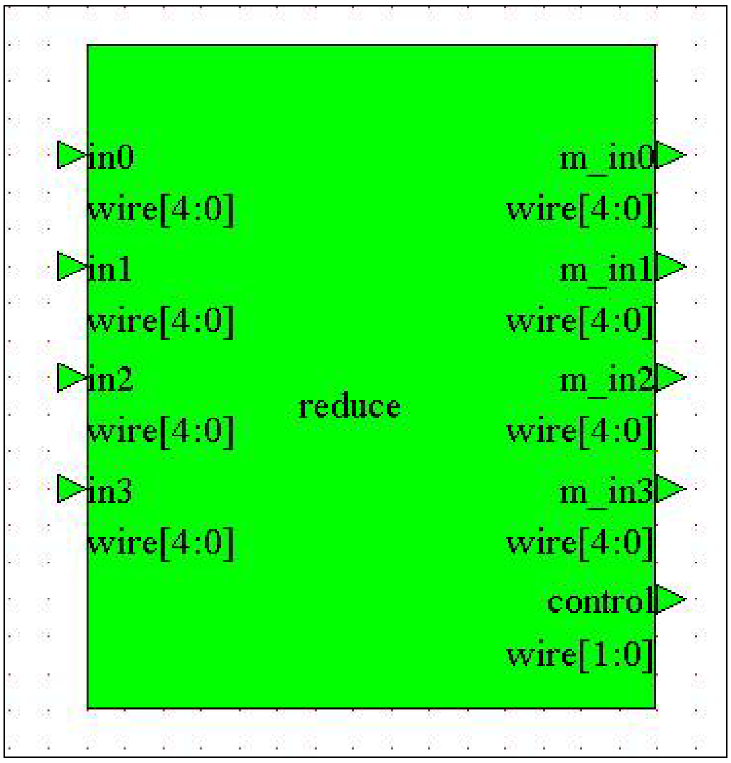


Figure 2.14 Block diagram of the reduce module.

2.4.4 Testing The System

Here is a testbench for the entire system: encoder, receiver front end, and decoder:

```
/**************************************************************/
/* Module viterbi_test_CDD                                     */
/**************************************************************/
/* This is the top-level module, viterbi_test_CDD that models
the communications link. It contains three modules:
viterbi_encode, viterbi_distances, and viterbi. There is no
analog and no noise in this version. The 2-bit message, X, is
encoded to a 3-bit signal, Y. In this module the message X is
generated using a simple counter. The digital 3-bit signal Y is
transmitted, received with noise as an analog signal (not
modeled here), and converted to a set of eight 3-bit distance
measures, in0, ..., in7. The distance measures form the input
to the Viterbi decoder that reconstructs the transmitted signal
Y, with an error signal if the measures are inconsistent. CDD =
counter input, digital transmission, digital reception */
/**************************************************************/
```

The Viterbi decoder takes the distance measures and calculates the most likely transmitted signal. It does this by keeping a running history of the previously received signals in a path memory. The path-memory length of this decoder is 12. By keeping a history of possible sequences and using the knowledge that the signals were generated by a state machine, it is possible to select the most likely sequences.

TABLE 2.5 Output from the Viterbi testbench

```
T    Clk X Y Out Error          t    Clk X Y Out Error
  0  0   3 x x   0              1351 1   1 0 0   0
 50  1   3 x x   0              1400 0   1 0 0   0
 51  1   0 x x   0              1450 1   1 0 0   0
 60  1   0 0 0   0              1451 1   2 5 2   0
100  0   0 0 0   0              1500 0   2 5 2   0
150  1   0 0 0   0              1550 1   2 5 2   0
151  1   1 2 0   0              1551 1   3 4 5   0
```

2.5 Simulation Phase

The simulation of this design is carried out in the environment of **Mentor Graphics** tools, and for the first simulation phase I used ***ModelSim*** simulator to verify the design behavior using the design test bench mentioned in the design phase. A capture of the simulation windows is shown in Figure 2.15.

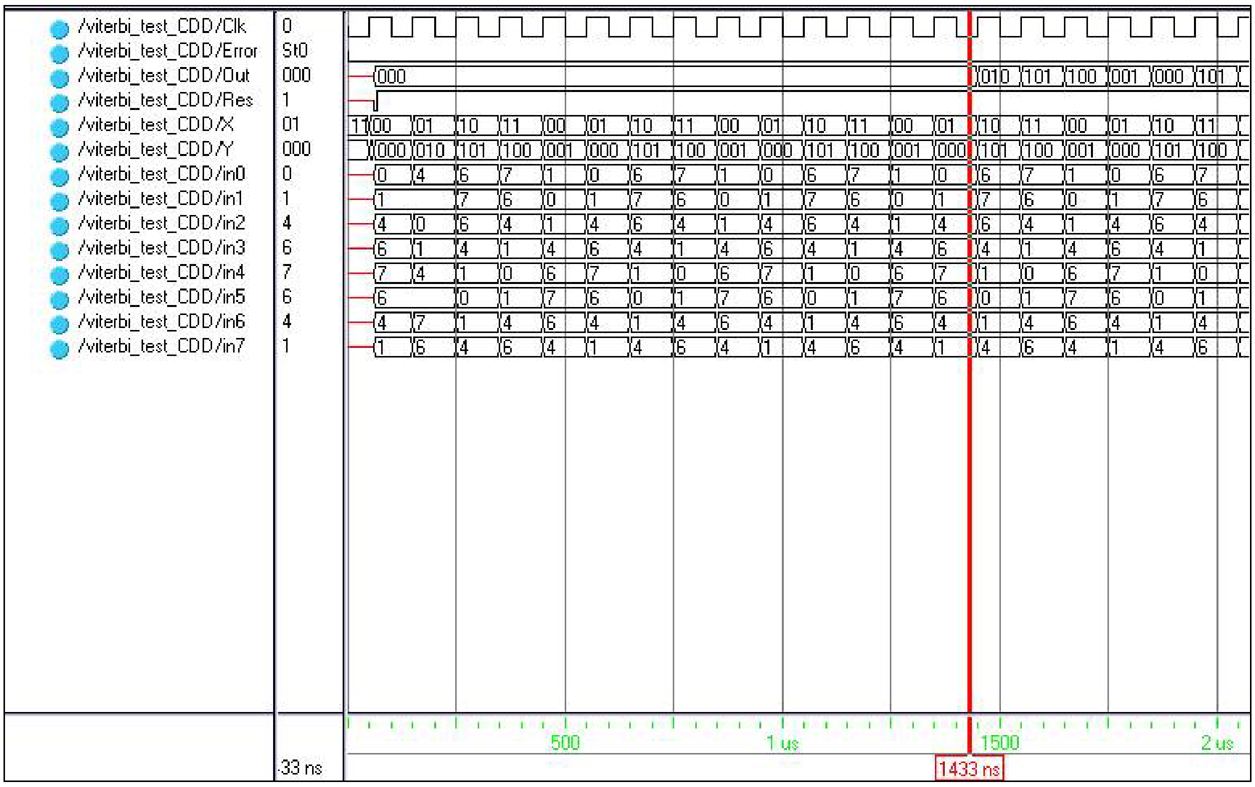


Figure 2.15 Simulation results using the design testbench.

2.6 Synthesis of the Viterbi Decoder

2.6.1 ASIC I/O

Some logic synthesizers can include I/O cells automatically, but the designer may have to use directives to designate special pads (clock buffers, for example). It may also be necessary to use commands to set I/O cell features such as selection of pull-up resistor, slew rate, and so on. Unfortunately there are no standards in this area. Worse, there is currently no accepted way to set these parameters from a HDL. Designers may also use either generic technology-independent I/O models or instantiate I/O cells directly from an I/O cell library. In my project I used the **Mentor Graphics** well known synthesizer ***Leonardo Spectrum*****,** in which one can chose the option (Insert I/O pads).

2.6.2 The Top-Level Model

The synthesizer will take the generic I/O cells and map them to I/O cells from a technology-specific library (AMS library "cyb"). We do not need three-state I/O cells or bidirectional I/O cells for the Viterbi ASIC. Figure 2.16 shows the synthesized Viterbi decoder top view.

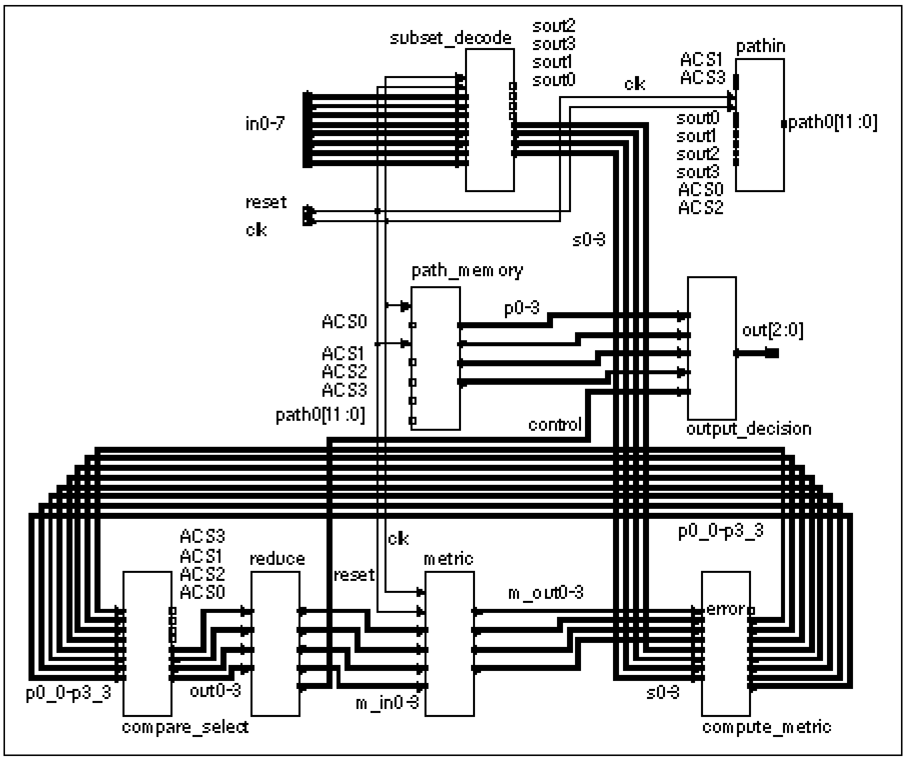


Figure 2.16 Synthesis Viterbi decoder (top level).

The design consists of four components (four component files), each has a view point:

1- Viterbi.

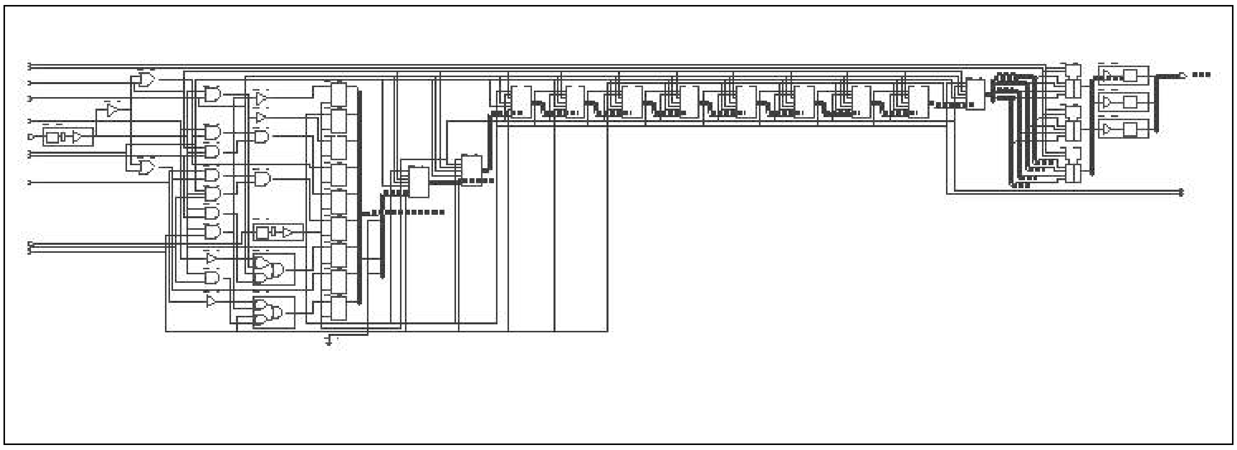


Figure 2.17 Component1 (Viterbi).

2- Subset_decode.

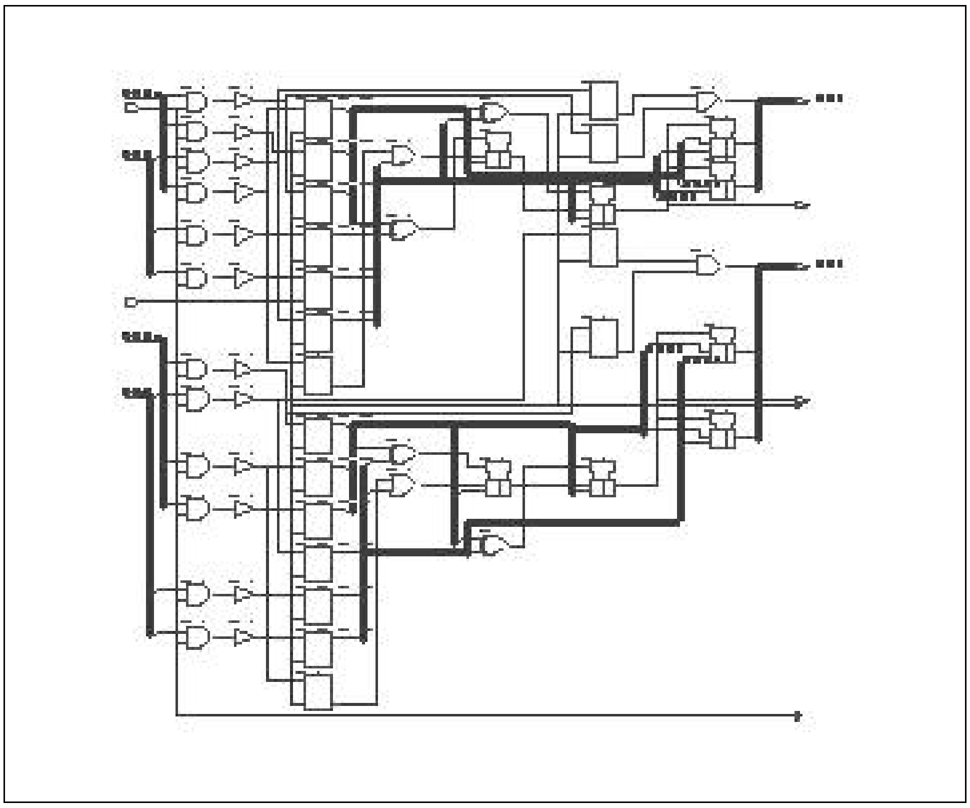


Figure 2.18 Component2(Subset_decode).

3- Path

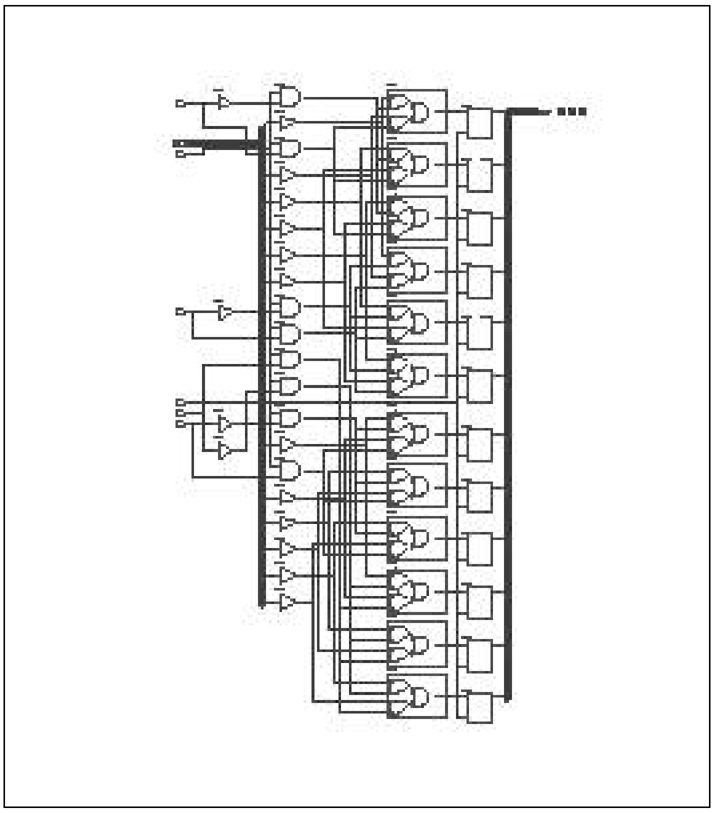


Figure 2.19 Component3(Path).

4- Metric

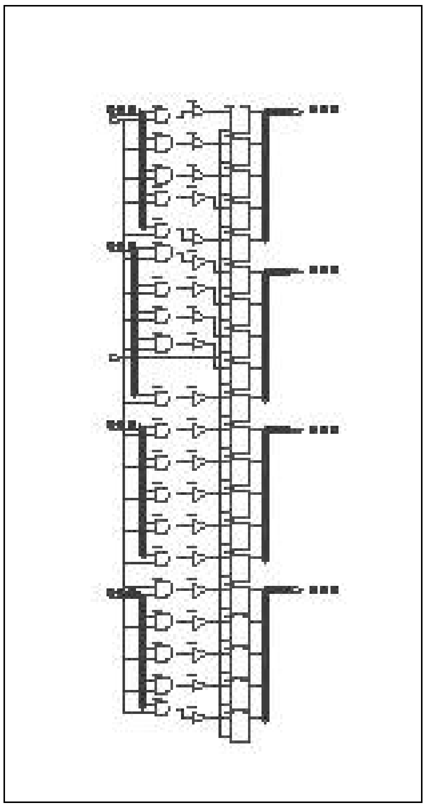


Figure 2.20 Component4(Metric).

After completing this phase, another phase starts which is the layout generation. This phase will start with begin with generating a viewpoint for the ***IC_Station*** tool of **Mentor Graphics,** we first have to insert the power pads. This is done using the ***Design Architect*** tool, which allows the designer to insert the power pads inside the schematic applet of the top-level components. Figure 2.21 shows the top level schematics generated by the ***Schematics Generator*** tool, then opened from the ***Design Architect*** tool.

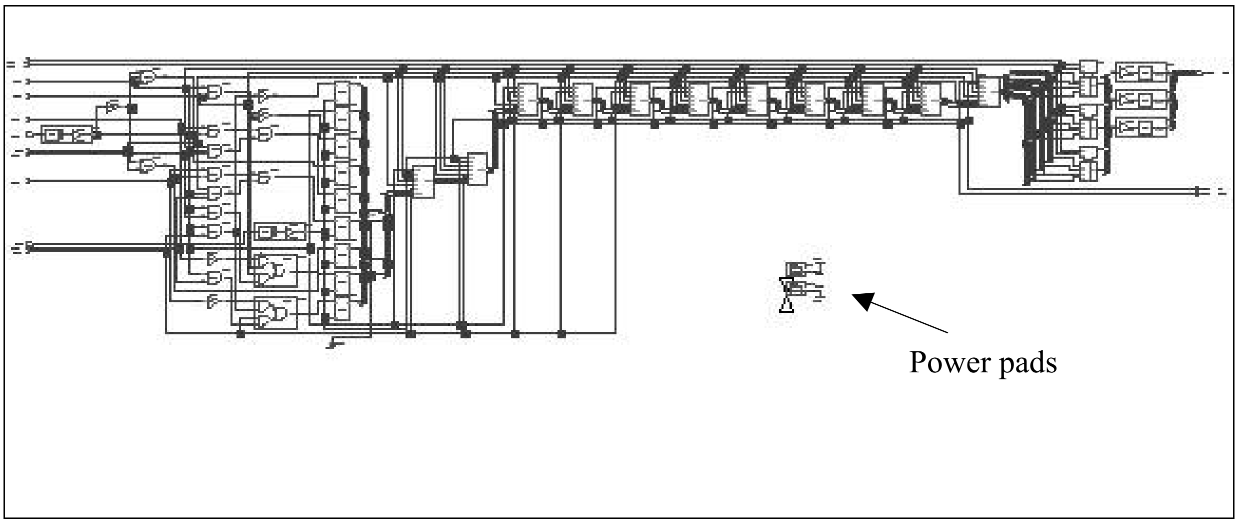


Figure 2.21 Top level design after power pads insertion manually.

2.7 Layout Phase

First step is the automated layout flow involves the ***IC_Station*** is creating the layout cell, then comes some other steps:

1- Floorplanning (auto).
2- Place and route (autoplace and route for cells and ports in case of core limited layout plus corner cells in case of pad_limited layouts).
3- Autorouting.

By these steps, the layout will be established and surrounded by I/O pads (in case the created cell is pad_limited), and without I/O pads (in case the created cell is core_limited, which is the case designs included in larger designs and not to be fabricated as a stand alone IC. Figure 2.22 shows the pad_limited cell representing the layout of the Viterbi decoder IC.

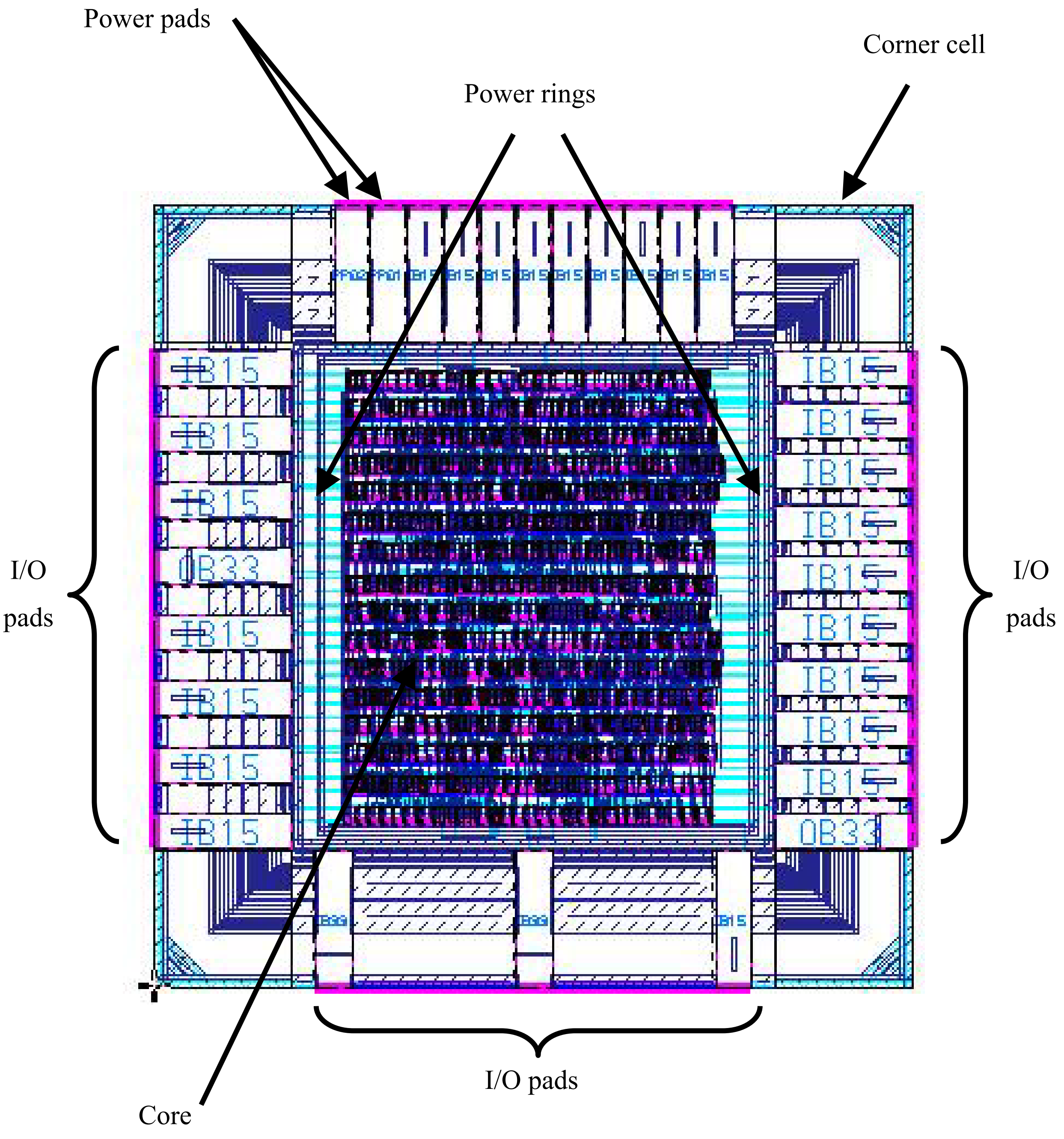


Figure 2.22 Viterbi decoder layout.

2.8 Design Rule Check (DRC)

Sometimes the routing between cells violates the IC design rules, hence we use the design rule checker to catch these violations. If such case occurs, then one must fix such errors manually. For my design I found no design rules' errors except in the I/O, which happens to happen, but in such case one must not think of fixing these DRC's because such cells are well designed, tested, manufactured and working properly in all the IC's that used the same library as I did (AMS 0.8 micron).

Reporting the cell:

1-Layers:

```
/*********************************************************/
Layers Used in Viterbi decoder layout
/*********************************************************/

Layer Number       Layer Name
*****************************
 1                  NTUB
 4                  DIFF
 6                  FIMP
 10                 POLY1
 12                 NPLUS
 13                 PPLUS
 16                 CONT
 17                 MET1
 18                 VIA
 19                 MET2
 20                   PAD
 22                   CELBOX
 27                   DIFCUT
 32                   DIODE
 38                   RESDEF
 39                   RESTRM
 51                   CESIG1
 52                   CESIG2
 54                   CETXT
 60                   TEXT
 61                   CEPWR1
 62                   CEPWR2

/***********************************************************/
```

<u>2- Cells</u>: Number of library cells used in this design = 1182 cell.

4- <u>Transistors:</u> Number of transistors in the design = 10370.

<u>5- Design area:</u> 5.5 millimeter square (without compaction).

2.9 Cell Compaction

We can also optimize the design layout area, or simply compacting the cell. This could result in extra DRC's sometimes, but for the current design; it had no extra DRC's and a compaction ratio of about 10%. The new area became equal to 5 millimeter square.

2.10 Back Annotation (BA)

It means post simulation that has to be done in order to make sure that delays (component and routing delays) didn't affect the design so far that can result in changing the behavior of it. I used ***QuickSim*** tool of **Mentor Graphics.** The simulation results were almost typical to that obtained pre to layout implementation, which means that the layout parasitical effects are almost negligible. Figure 2.21 shows the simulation results on ***QuickSim***.

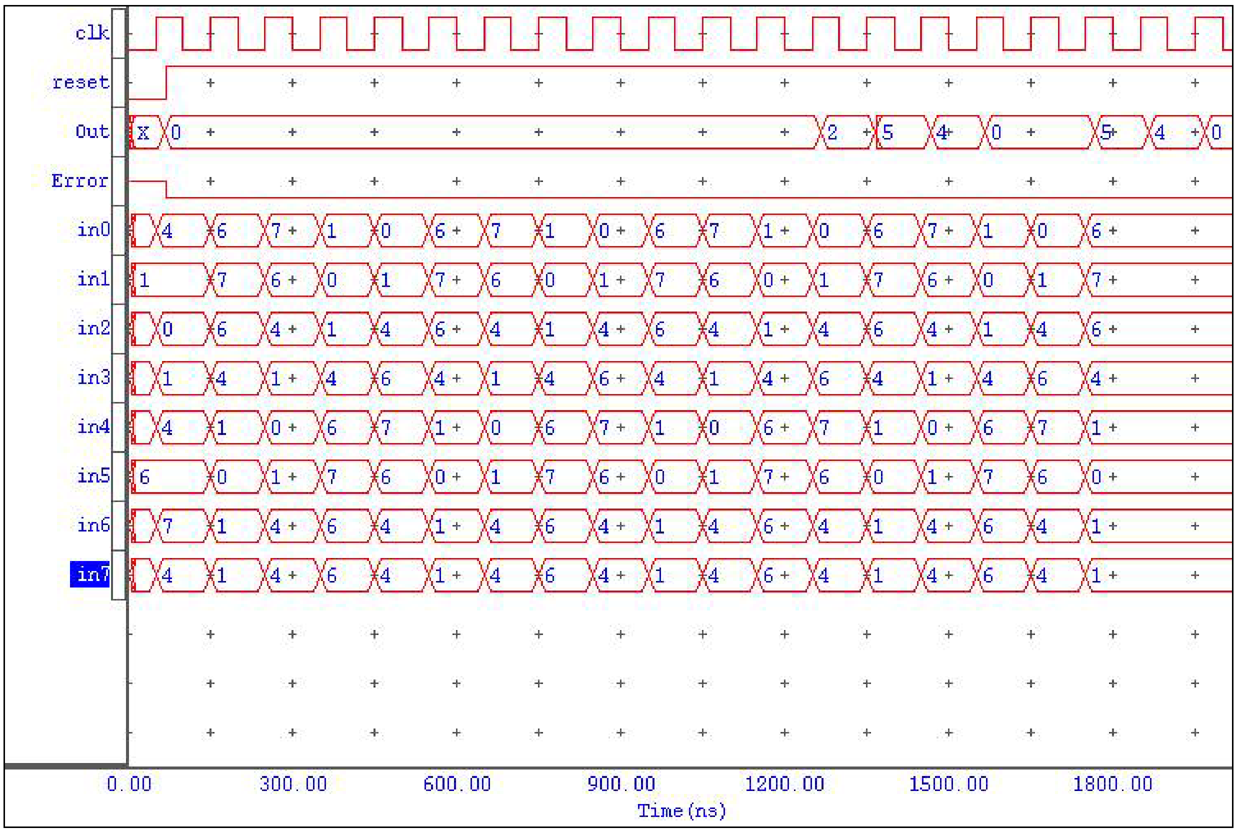


Figure 2.23 Back Annotation results on QuickSim.

2.11 Layout Versus Schematics (LVS)

This phase makes sure that your layout achieves your schematics' functionality. This operation is an automated one, and the result that tells you that the operation has been succeeded is a smiling face appears in the output file. The operation succeeded in the Viterbi design.