DIGITAL DESIGN VHDL LABORATORY NOTES

CERL/EE

OCTOBER 16, 1996

VERSION 1.01

COPYRIGHT 1996

DR. CECIL ALFORD

TSAI CHI HUANG

CERL / EE

TABLE OF CONTENTS

1. VHDL - HARDWARE DESIGN SOFTWARE APPROACH 6

1.1 INTRODUCTION 6
1.1.1 SEQUENTIAL PROGRAMMING 6
1.1.2 PARALLEL PROGRAMMING 6
1.1.3 OBJECT BASED PROGRAMMING 8
1.1.4 PROGRAMMABLE CIRCUITS 9
1.2 PROBLEMS 10

2. VHDL BEHAVIOR AND STRUCTURE MODELS 13

2.1 INTRODUCTION 13
2.2 PROBLEMS 18

3. STATE MACHINES AND PROGRAMMABLE LOGIC DEVICES 19

3.1 INTRODUCTION 19
3.2 MEALY AND MOORE STATE MACHINES 19
3.3 VHDL PROGRAMS 22
3.3.1 MEALY STATE MACHINE 22
3.3.2 MOORE STATE MACHINE 23
3.4 PROBLEMS 24

4. DIGITAL DEVICE MODELING 26

4.1 INTRODUCTION 26
4.2 SRAM MEMORY 26
4.3 VHDL SRAM MEMORY DESIGN 27
4.4 SRAM VHDL LISTING 28
4.5 PROBLEMS 30

5. DIGITAL DESIGN USING DIVIDE AND CONQUER 32

5.1 INTRODUCTION 32
5.2 VHDL FEATURES 32
5.3 ALU FUNCTIONS 32
5.4 ALU COMPONENT IMPLEMENTATION EXAMPLES 33
5.4.1 RIPPLE CARRY ADDER IMPLEMENTATION 33
5.4.2 PACKAGE *IARITH_PARTS_PKG* LISTING 34

Copyright 1996, CERL / EE, V1.00

5.4.3 *TESTADD.VHD* LISTING 36
5.4.4 CIRCULAR SHIFTER IMPLEMENTATION 37
5.4.5 CIRCULAR SHIFT TRACE DATA 37
5.4.6 PACKAGE *LOGIC_PARTS_PKG* LISTING 37
5.5 ALU IMPLEMENTATION EXAMPLE 40
5.6 PROBLEMS 42

6. ALU DATAPTH IMPLEMENTATION 43

6.1 INTRODUCTION 43
6.2 INSTRUCTION SET ARCHITECTURE (ISA) 43
6.3 A COMPUTER ARCHITECTURE IMPLEMENTATION 43
6.4 IMPLEMENTATION EXAMPLE 44
6.4.1 4X4 REGISTER FILE AND 8X4 MEMORY BLOCK 46
6.4.2 CONTROL MODULE 46
6.4.3 EXECUTION EXAMPLE 47
6.4.4 VHDL LISTING 50
6.4.4.1 regf4x4.vhd 50
6.4.4.2 mem4x2.vhd 52
6.4.4.3 ctrlm.vhd 54
6.4.4.4 BIA.vhd 59
6.5 PROBLEMS 59

7. APPENDICES 60

7.1 INSTALL WARP ON PC 60
7.2 INTRODUCTION TO WARP4 VHDL DEVELOPMENT SYSTEM 60

Copyright 1996, CERL / EE, V1.00

LIST OF FIGURES

Figure 1-1. AND-OR Gates ______ *7*
Figure 1-2. AND-OR Gate Propagation Output ______ *8*
Figure 1-3. PLD Circuit Structure ______ *10*
Figure 1-4. LED Control Circuit ______ *10*
Figure 1-5. LED Segment Control Label ______ *11*
Figure 2-1. Example of Discrete Components ______ *13*
Figure 2-2. RS Flip-flop ______ *14*
Figure 2-3. D Latch ______ *14*
Figure 2-4. D Latch with Reset Control ______ *14*
Figure 2-5. D Flip-flop ______ *15*
Figure 3-1. Mealy and Moore Machine Block Diagram ______ *19*
Figure 3-2. Mealy Machine State Diagram ______ *20*
Figure 3-3. Mealy Machine Timing Diagram ______ *20*
Figure 3-4. Moore Machine State Diagram ______ *21*
Figure 3-5. Moore Machine Timing Diagram ______ *22*
Figure 3-6. Moore Machine to Mealy Machine Conversion ______ *22*
Figure 3-7. Question #1 ______ *24*
Figure 3-8. Question #2 ______ *24*
Figure 3-9. State Flow Diagram ______ *25*
Figure 4-1. SRAM Timing Diagram ______ *27*
Figure 4-2. SRAM Simulation Waveform ______ *27*
Figure 4-3. SRAMvc Timing Diagram ______ *31*
Figure 5-1. ALU Block Diagram ______ *32*
Figure 5-2. An ALU Design ______ *33*
Figure 5-3. Ripple Carry Adder Waveform ______ *34*
Figure 6-1. BIA Block Diagram ______ *45*
Figure 6-2. lw Instruction Timing Diagram ______ *45*
Figure 6-3. 4x4 Register File Block Diagram ______ *46*
Figure 6-4. BIA State Diagram ______ *48*
Figure 6-5. BIA Simulation Timing ______ *50*
Figure 7-1. Project Window ______ *60*
Figure 7-2. Device Window ______ *61*

Copyright 1996, CERL / EE, V1.00

List of VHDL Code

Listing 1-1. Simple VHDL Code for Dataflow Modeling ___ *7*
Listing 1-2. Digital Design Goodness Criteria ___ *9*
Listing 1-3. Example VHDL Code ___ *11*
Listing 2-1. Dataflow Modeling of AND2, OR2 and INV1 Gates ___ *16*
Listing 2-2. D Latch Structure Modeling ___ *16*
Listing 2-3. D Latch Behavior Modeling ___ *17*
Listing 2-4. D Flip Flop Behavior Modeling ___ *18*
Listing 3-1. Mealy Machine VHDL Code ___ *23*
Listing 3-2. Moore Machine VHDL Code ___ *24*
Listing 4-1. SRAM VHDL Code ___ *30*
Listing 5-1. Iarith_parts_pkg VHDL Code Listing ___ *36*
Listing 5-2. Testadd.vhd VHDL Code Listing ___ *36*
Listing 5-3. Circular Shift Trace Listing ___ *37*
Listing 5-4. Logic_parts_pkg VHDL Code Listing ___ *39*
Listing 5-5. Example ALU VHDL Code Listing ___ *42*
Listing 6-1. Regf4x4.vhd VHDL Code ___ *52*
Listing 6-2. Mem4x2.vhd VHDL Code ___ *54*
Listing 6-3. Ctrlm.h VHDL Code ___ *59*

Copyright 1996, CERL / EE, V1.00

1. VHDL - Hardware Design Software Approach

1.1 Introduction

Software programming, no doubt, is a common skill required by all college engineering students. Such knowledge is indispensable for utilizing a computer as a tool for achieving greater engineering goals in both the classroom and the real world. Programming tasks for applications range vary in level of detail and complexity. The readability of programming language constructs span from a very low level, which is almost at the hardware level such as PAL programming, to a high level such as application level programming. Good examples of application programming are database, UNIX shell scripts, DOS batch file, and SPICE circuit simulation. Almost all programming language lie between these two ends. The major difference between programming languages is their programming complexity which affects such things as development time. Sequential programming is most common among programming languages. Examples of sequential languages are BASIC, FORTRAN, Pascal and C. Besides sequential programming, there are other programming paradigms: parallel programming and object based programming. These paradigms are new for most people, and they will be introduced during the process of learning VHDL. VHDL is not much different from conventional programming languages, except it uses sequential, parallel, and object based software paradigms to achieve its goal: namely, digital hardware simulation and synthesis. In other words, a VHDL program is used to specify, to model, and to "create" digital hardware.

VHDL stands for VHSIC (Very High Speed Integrated Circuit) Descriptive Language. VHDL is promoted by the Department of Defense (DOD) mainly to facilitate the standardization of all digital hardware design. VHDL serves as a documentation tool at the beginning to describe a system in a unified manner across the cooperative group of a project. VHDL was started in early 1980 inside of DOD and eventually got standardized by IEEE with the name standard 1076 (IEEE-STD-1076) in 1987. Although further enhancements have been added to the later version of VHDL, for example standard 1164 (IEEE-STD-1164), standard 1076 will be sufficient to learn and explore most of the language feature.

1.1.1 Sequential Programming

This type of programming is characterized by the sequential nature of program flow. Popular programming languages such as Pascal, C, BASIC and FORTRAN are the examples in this category. It is also called imperative programming because programming statements to be executed are imperative. Programs are executed in sequential order and a statement is executed following the statement that precedes it in the proogramm. There is always a clear decision as far as which the next program statement is to be executed next. In summary, in sequential programming, there is no ambiguity as far as the statements execution order is concerned.

In VHDL syntax, the statements after the reserved keyword ***process*** or within the ***process*** block will be executed in sequential order. Typical VHDL sequential statements are condition statements, loop statements, and assignment statements. The syntax of each type of statement can be found in any reasonably thorough VHDL tutorial text book.

1.1.2 Parallel Programming

The parallel programming concept is a more difficult programming paradigm to grasp for people used to sequential programming. The parallel programming concept is straightforward and is expressed through

Copyright 1996, CERL / EE, V1.00

its name; statements are executed in parallel. Moreover, the process of executing statements in parallel mimics the real world phenomena and thus coincide better with the behavior of digital circuits. In parallel processing, the application of parallel programmingg, programs executed in parallel have the advantage of boosting program execution speed versus the equivalent sequential program. In the area of VHDL programming, the goal is to mimic the physical digital system. Due to the fact that hardware operates executed in parallel, to capture the functionality of the hardware, it is important for VHDL to have the features to facilitate the modeling of such behavior.

Parallel program statements, as the name suggests, are the statements that can be executed in parallel. Parallel statements are less intuitive to use due to their nondeterministic nature. Ideally, all parallel statements should be executed at the same time. However, in the popular Von Neumann machine or sequential computer world (eg., PCs and worstations), only sequential statements can be directly executed. Therefore, parallel programming must be simulated on a sequential machine. When simulating parallel statements, due to the fact that statements are executed in parallel and random order, it is crucial that the execution order be irrelevant and not affect the result.

An example of a parallel system is shown in Figure 1-1. It describes a digital hardware circuit where four inputs and one output are involved. Inputs A, B, C and D are fed into two AND gates, and the outputs of these AND gates, E and F, are then fed into an OR gate whose output is shown below(see Figure 1-1).

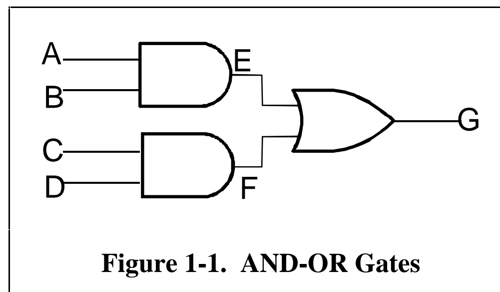


Figure 1-1. AND-OR Gates

A piece of simple VHDL code shown in Listing 1-1 describes the circuit above. Notice the section after *dataflow_machine* ***architecture*** in the code. Three statements, a, *b*, and *c* describe AND and OR logical assignments. These statements can be rearranged in any order, and the simulated output G should still be the same.

```
-- Sum of product example
entity sop is
port (       a, b, c, d:  in bit;
       e, f:  inout bit;
       g:  out bit);
end sop;

architecture dataflow_machine of sop is
begin

-- Begin parallel statements

       g <= e or f;       -- Statement a
       e <= a and b;    -- Statement b
       f <= c and d;     -- Statement c

end dataflow_machine;
```

Listing 1-1. Simple VHDL Code for Dataflow Modeling

Copyright 1996, CERL / EE, V1.00

The result G will be the same if the statements assigning E, F and G outputs were placed in any order. Any data arriving at the inputs A, B, C, and D will pass through the AND and OR gates in continuous time fashion. Whatever shown up at A, B, C or D inputs will be reflected at the intermediate outputs E and F and at the final output G. Thus, output G should change instantly according to the essence of parallel programming. In the world of hardware execution, there is always a delay when data pass through gates. The VHDL code of hardware synthesis is mapped into a Cypress CPLD 375 device and fitted with optimization disabled. The simulation of the mapped Cypress 375 PCLD shows that there is a delay of about 5 ticks between each logical gate. Notice that ticks from VHDL simulation below do not represent any real-time. Instead, a simulation tick is simply a unit of delay of simulation time. In the real world, gates delay are on the order of nanoseconds (10^{-9} sec).

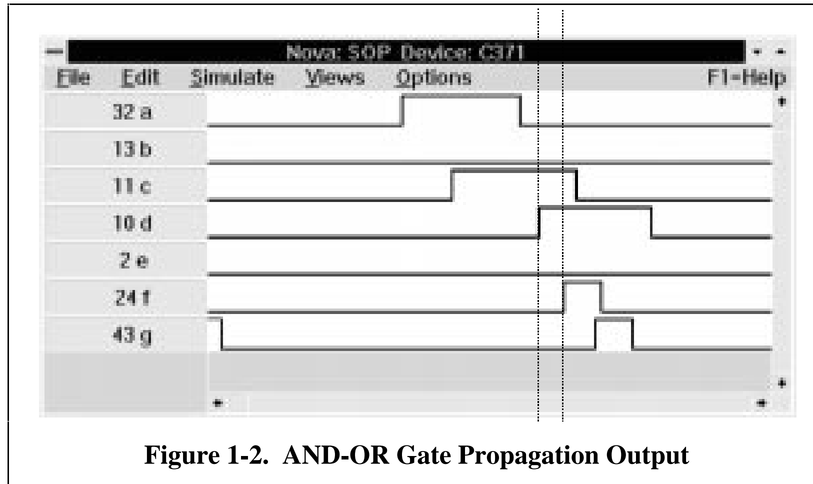


Figure 1-2. AND-OR Gate Propagation Output

Intuitively, the parallel VHDL statements above should be executed in the order: c, a and b. However, the hardware VHDL simulator handles the situation differently. The VHDL simulator executes the parallel statements in clock random order fashion.

Notice that VHDL statements written within the ***process*** block have been executed sequentially. Therefore, all VHDL statements outside of the ***process*** block will be executed in parallel. For example, two ***process*** blocks can be executed in parallel. To resolve sequential statements inside of each ***process*** block, VHDL uses the concept of delta time slots for each clock tick. In each clock tick, the simulator allocates however many sub-ticks called delta for simulating the instantaneous statement execution. Individual parallel statement are being executed at delta ticks. A simulated clock tick will only move forward when the delta tick yields consistent outputs. An example above would be stable E, F, and G outputs.

1.1.3 Object Based Programming

Objected based programming is different from the sequential and parallel paradigms. It should be treated more like a methodology than a paradigm. Objected based methodology basically enforces structure into VHDL. This facilitates the systematic construction of large and complicate digital system. Object based programming also supports the concept of abstract data types enabling the VHDL engineer to construct custom data types and operations and then to store them systematically for future VHDL projects. This is the concept of VHDL code reuse. Since the feature of breaking VHDL code into small components and libraries is already built into the language, it provides the VHDL designer with the structural divide and conquer technique when building larger systems. Moreover, the concept of code reuse definitely saves

Copyright 1996, CERL / EE, V1.00

system development time. In summary, object based features are extremely useful for VHDL code development, management, and reusability.

1.1.4 Programmable Circuits

Digital design technology has evolved greatly in recent years. This can be seen in the field of ever changing microprocessor design where complexity and performance double every year. With improvements to digital hardware technology, the criteria used to determine a design's goodness remain

Correctness	Ability to exactly perform their tasks, as defined by the requirement and specification.
Robustness	Ability to function even in abnormal conditions.
Extendibility	Ability to ease change in specification change.
Reusability	Ability to be reused, in whole or in part, for new applications.
Compatibility	Ability to combine with others.

Listing 1-2. Digital Design Goodness Criteria

the same, and they are listed in Listing 1-2.

These five criteria are borrowed from the book, *Object-oriented Software Construction*, by Bertrand Meyer. Although, they were used to describe software quality, the concepts are also appropriate to be used on hardware. Notice that hardware design technical merits such as speed and cost are not mentioned because they highly depend on the state of technology and market demand.

The word, flexibility, sums up all these criteria above. The practice of digital design nowadays must be flexible. Thus, the trend is to replace ridged discrete logic component with programmable devices to maximize hardware flexibility.

Traditional programmable circuits comes in three flavors of simple combinational logic circuit: Programmable Read Only Memory(PROM), Programmable Logic Array(PLA), and Programmable Array Logic(PAL). All these three devices contain similar hardware structure of AND and OR arrays. Input goes from the AND array to the OR array. The difference is that PROM contains fixed AND and programmable OR arrays, PAL contains programmable AND and fixed OR arrays, and PLA contains programmable AND and OR arrays. (See the figure Figure 1-3.)

Copyright 1996, CERL / EE, V1.00

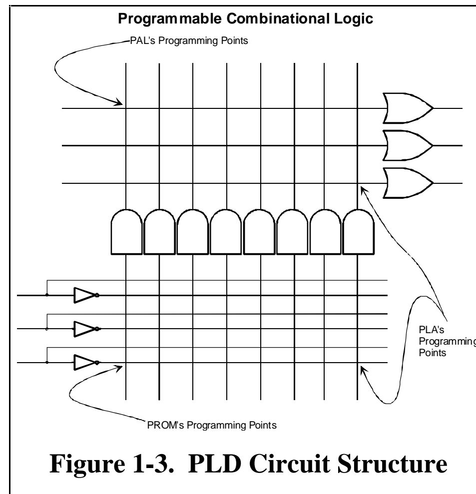


Figure 1-3. PLD Circuit Structure

Most of the time, a digital hardware designer doesn't need to know the circuit structure of the programmable logic device (PLD) that he is using except he is conscious about the design's speed and capacity. For example, he might want to try to "fit" a large piece of circuit into a given PLD or to select a type of fast PLD to meet a timing limit. Digital hardware can be programmed to fit a circuit into a PLD using hardware programming languages such as CUPL, Abel, PALism, and VHDL.

For more complex PLDs (CPLDs), memory elements such as flip flop devices are involved and they will be introduced later.

1.2 Problems

1) Binary Coded Decimal (BCD) is a useful decimal number representation using binary numbers. Many times, it is the only representation that digital hardware uses to communicate with humans since humans are custom to the decimal numbering system. One example is to use BCD to control the decimal LED display panel of a calculator. Each LED decimal digit has eight segments and each segment contains an LED source as shown in Figure 1-4.

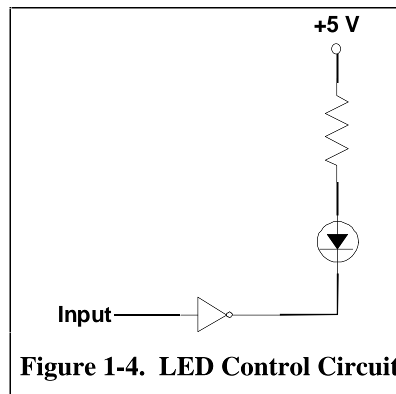


Figure 1-4. LED Control Circuit

At high input "1", current flows through the LED and resistor, and the LED glows. The typical current is about 20mA, and voltage drops 1.6 Volts(V) across the LED and 3 V across the resistor. If input is low "0", no current will flow through LED and it will become dim.

The LED segment specification is illustrated in Figure 1-5.

Copyright 1996, CERL / EE, V1.00

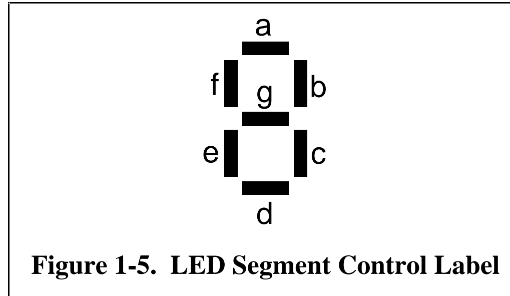


Figure 1-5. LED Segment Control Label

The input consists of 5 bits, and the output consists of 7 bits. Out of 5 input bits, 4 bits are used to specify binary value, and one bit is used to blank all LED segments. For all output bits, one bit is used for controlling each of the segment, and one bit is used to indicate overflow of the input binary number, for example binary input greater than 9. The LED logic specification is shown in Table 1-1.

An LED segment control decoder is used to control a one-digit LED display panel. Construct such a decoder using VHDL. This can be done using ***if*** statements nested inside of a ***process*** statement or ***when*** statements alone. The form of such decoder could look like the VHDL code segment in Listing 1-3.

```
entity XXX is
        port( .... );
end XXX;
architecture XXXX of XXX is
begin

-- option 1
option1: process ( input)
begin
        if <condition> then
                <output> <= <input>;
        elsif <condition> then
                <output> <= <input>;
        .....
        else
                <output> <= <input>;
        end if;
end process option1;

-- Or use option 2
output <= <input> when <condition> else
                <input> when <condition> else
                .....
                <input> when <condition> else
                <input>;
end XXXX;
```

Listing 1-3. Example VHDL Code

After completing the implementation, test the decoder for various inputs and simulate the resulting output using the NOVA simulator. The simulation session can be saved into the trace file with the “PSD” extension. This is done by using the mouse to select *FILE* and then *Write Trace (*.PSD)* on the top of the NOVA menu bar. In the saved PSD file, annotate the clock tick that corresponds to the decoder action. For example, the input that leads to the output display.

Copyright 1996, CERL / EE, V1.00

INPUT **OUTPUT**

Comment	Blank	in1	in2	in3	in4	Overflow	a	b	c	d	e	f	g
Blank	1	x	x	x	x	0	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	1	0	1	0	0	1	1	1	1
2	0	0	0	1	0	0	0	0	1	0	0	1	0
3	0	0	0	1	1	0	0	0	0	0	1	1	0
4	0	0	1	0	0	0	1	0	0	1	1	0	0
5	0	0	1	0	1	0	0	1	0	0	1	0	0
6	0	0	1	1	0	0	0	1	0	0	0	0	0
7	0	0	1	1	1	0	0	0	0	1	1	1	1
8	0	1	0	0	0	0	0	0	0	0	0	0	0
9	0	1	0	0	1	0	0	0	0	1	1	0	0
a	0	1	0	1	0	1	0	0	0	0	0	1	0
b	0	1	0	1	1	1	1	1	0	0	0	0	0
c	0	1	1	0	0	1	0	1	1	0	0	0	1
d	0	1	1	0	1	1	1	0	0	0	0	1	0
e	0	1	1	1	0	1	0	0	1	0	0	0	0
f	0	1	1	1	1	1	0	1	1	1	0	0	0

Table 1-1. LED-Control Truth Table

Copyright 1996, CERL / EE, V1.00

2. VHDL Behavior and Structure models

2.1 Introduction

The ability to "remember" is important in the field of digital systems. The digital memory element is a necessary and fundamental block for building complex digital systems. Despite its importance, it takes no more than basic digital logic for its construction. There are many ways to construct a digital memory element using basic logic gates such as NOT, AND, and OR.

There are three ways to model a digital system using VHDL. A piece of VHDL hardware can be modeled in structural, data flow and behavior methodologies. In this lesson, methods of building memory elements such as latches and flip flops are studied using both single and mixed methodologies.

A natural way to describe discrete components is to use a data flow model. Discrete gates such as a one input and one output NOT gate, INV1, two inputs and one output AND gate, AND2, and two inputs and one output OR gate, OR2, can be modeled fairly easily, and they are illustrated in Figure 2-1. The logic gates in Figure 2-1 have their inputs and outputs labeled. These components will be used to illustrate more examples of elaborate logic devices, such as a digital memory element.

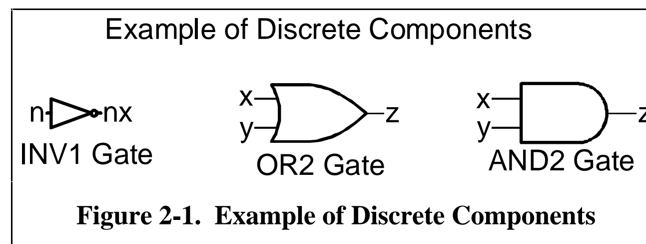


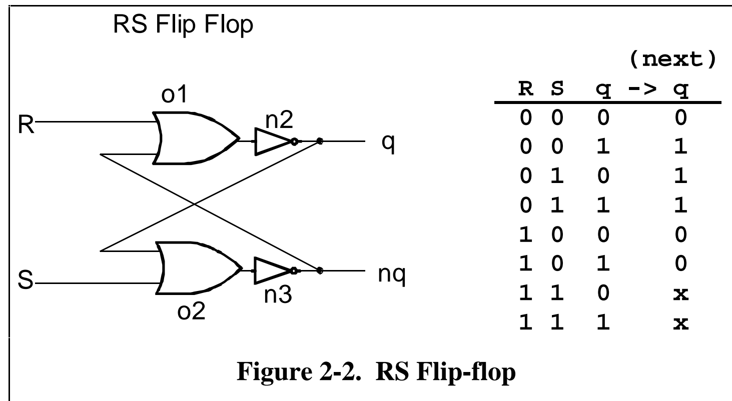
Figure 2-1. Example of Discrete Components

There are many types of memory elements that one can create using the basic gates shown in Figure 2-1. A simple memory block that one can make is an RS flip flop which uses two OR2 and INV1 gates. For the sake of consistency with the previous example, NOR2 is not included for now. Even though OR2 is used to create an RS flip flop, DeMorgan's law would also permit AND2 to be used as well.

Notice the flip flop is named RS because the flip and flop can be set (output=1) or reset (output=0) depending on the inputs (R and S). When both R and S inputs are set to zero, outputs (*q* and *nq*) will be suspended to the previous values of R and S. Be aware that both R and S, at the same time, can't be set to one for two reasons. The first reason is that once R and S become one, output *q* and *nq* will no longer be complementary to each other. Both *q* and *nq* will become zero, and this violates the assumption of complementing flip flop output. Second, once the output *q* and *nq* become zero, the next state after R and S change, outputs *q* and *nq* are impossible to predict due to the feedback of *q* and *nq* values. See Figure 2-2 for the RS flip flop logic truth table.

Although an RS flip flop is easy to implement, it is not an ideal memory device due to its incomplete input states. From the perspective of a digital designer, it is cumbersome to guarantee the partial input state. A better way to use the same memory block is to modify the RS flip flop to take care of the partial input problem. One simple solution is to add two AND2 gates at the front of RS flip flop to prevent R and S from becoming one at the same time. The resulting circuitry is called a D latch. See Figure 2-3 for this modified memory element..

Copyright 1996, CERL / EE, V1.00



R	S	q	-> q (next)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	x
1	1	1	x

Figure 2-2. RS Flip-flop

A D latch functions very similar to an RS flip flop because it remembers past states. However, it works differently because output is triggered by the level of the input signal instead of the input signal edge. As a matter of fact, a flip flop works differently from a latch because the flip flop output signal is triggered at the clock edge and the latch output signal is triggered at the level of clock signal.

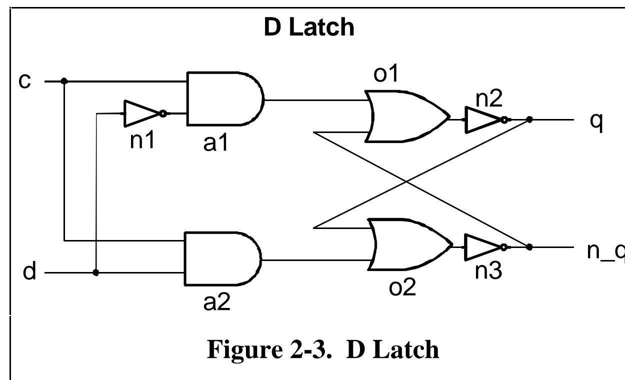


Figure 2-3. D Latch

Digital circuitry during the beginning of operation, especially memory elements, will go to erroneous states unless provision is made for startup operation. At the initial stage of the memory operation, the value stored inside of the memory element is unknown, and, if this unknown is to be applied to rest of the system, the system will almost be guaranteed to produce error. The modification to D latch is shown in Figure 2-4. The new latch get reset circuitry so the internal value stored can be set to zero (low) whenever the reset signal goes high.

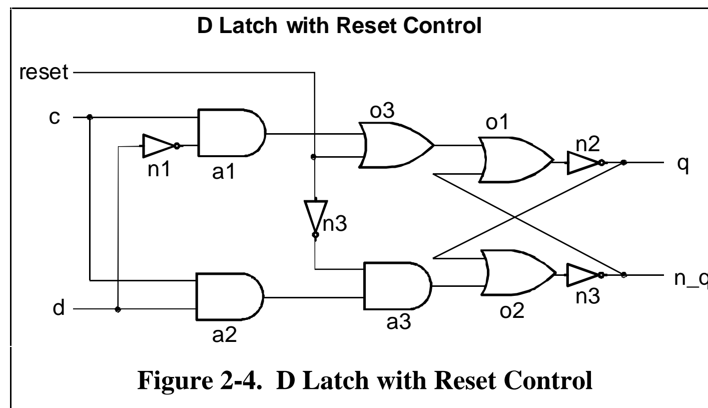


Figure 2-4. D Latch with Reset Control

although the difference between flip flops and latches have been discussed, it is important to note the application side of these memory elements. In the world of digital design, the majority of digital system is

Copyright 1996, CERL / EE, V1.00

running synchronously with the system clock. Since flip flops activate at an instance in time to acquire data, relatively speaking, flip flops are more useful than latches because the output of flip flops is held constant for a full clock cycle instead of half a clock cycle.

Cascading two D latches can produce a D flip flop. The clock signal that runs through these two latches is inverted so data can pass through the latches in pipeline fashion. Due to the holding of input data at the lower phase clock signal, input signal of the first latch is preserved for the instance when the clock signal goes down. Simultaneously, the signal output from the first latch is acquired by the second latch, and the signal is held constant until the second latch's clock goes down. A D flip flop drawing made by cascading two D latches is shown below in Figure 2-5.

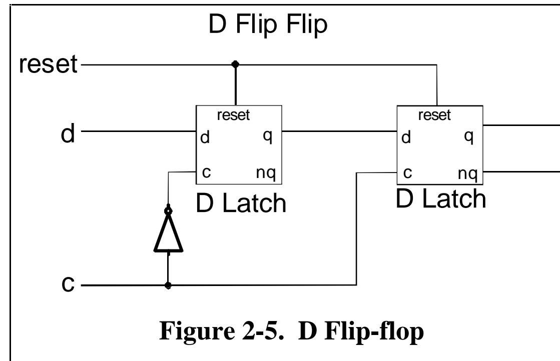


Figure 2-5. D Flip-flop

Now the memory basics have been explained, and it is time to model some of the memory block using VHDL. The difference between structure, dataflow, and behavior models will be shown in later examples. For example, in Listing 2-1, each discrete logic component, AND2, OR2 and INV1, is defined at the beginning using the dataflow model as state before. Notice that VHDL programming structure is declared with ***entity*** statement first and followed by an ***architecture*** statement. Signals defined in the ***entity*** block under the keyword ***port*** are visible inside of the architecture block.

```
-- INV1 Gate
--
entity inv1 is
	port(	x: in bit;
		nx: out bit);
end inv1;

architecture behav of inv1 is
begin
	nx <= not x;
end behav;

--AND2 Gate
--
entity and2 is
	port(	x, y: in bit;
		z: out bit);
end and2;

architecture behav of and2 is
begin
	z <= x and y;
end behav;

-- OR2 Gate
--
entity or2 is
	port(	x, y: in bit;
		z: out bit);
end or2;
```


Copyright 1996, CERL / EE, V1.00

```
architecture behav of or2 is
begin
        z <= x or y;
end behav;
```

Listing 2-1. Dataflow Modeling of AND2, OR2 and INV1 Gates

As its name implies, structures modeling uses the gates defined above to connect to others to make into a system. The process of describing the system structure in gate connection is shown in Listing 2-2. The VHDL code after the gate definitions shown in Listing 2-1 shows how to connect gates, and this is done through the use of VHDL statement ***map***.

```
--
-- structure modeling of D latch
--

entity lat_ds1 is
        port(        c, d: in bit; -- input data
                     q, nq: inout bit);
end lat_ds1;

architecture struct of lat_ds1 is

        signal i1_nx: bit;
        signal i2_nx: bit;
        signal i3_nx: bit;
        signal a1_z: bit;
        signal a2_z: bit;
        signal o1_z: bit;
        signal o2_z: bit;

        component and2
                port(x, y: in bit; z: out bit);
        end component;

        component or2
                port(x, y: in bit; z: out bit);
        end component;

        component inv1
                port(x: in bit; nx: out bit);
        end component;

begin

-- structure connection

a1: and2  port map(c, i1_nx, a1_z);
a2: and2  port map(c, d, a2_z);
o1: or2   port map(a1_z, nq, o1_z);
o2: or2   port map(a2_z, q, o2_z);
i1: inv1  port map(d, i1_nx);
i2: inv1  port map(o1_z, q);
i3: inv1  port map(o2_z, nq);

end struct;
```

Listing 2-2. D Latch Structure Modeling

The variables inside of the mapping statements are declared after the D latch architecture declaration, ***architecture*** *struct of lat_ds1 is*, and before the ***begin*** statement. These signals can only be used here, and they are not accessible by other architecture code blocks.

Copyright 1996, CERL / EE, V1.00

Gate components used here are also required to be declared. The declaration of a component is similar to C, FORTRAN or Pascal programming where all procedures or subroutines are required to be declared before use. However, it is not exactly the same as doing programming because each component declared could be physical part rather than a conceptual abstract. For example, the ports declared by component do have corresponding physical meaning, namely wires to carry signals. This concept should be understood from the digital hardware engineer perspective.

Now, consider taking the same D latch circuit and describing or modeling it behaviorally in VHDL. This code is shown in Listing 2-3. As we have noticed, the VHDL code here is shorter than its corresponding structure model because the circuit behavior is directly described. It is a higher level modeling, and it can be done easily if the designer knows exactly how his circuit behaves. Many times, the digital circuit designer knows how the circuit should behave but is not sure if the actual circuit behaves as he predicts. Therefore, it is useful to construct the circuit as a structure model and verifies its working order. Behavior modeling saves time and it is based on the assumption that the designer knows his circuit's behavior well. If the assumption about the circuit is wrong, then behavior modeling can lead to a disastrous result. Remember, behavior modeling should be used as much as possible when modeling digital circuits because it will save much development time.

In the field of VLSI design, the translation of a behavior model to a structure model is called *hardware synthesis*, and it relies much on the software intelligence. During the process of mapping a VHDL circuit to the actual hardware, it is best to specify as much structure as possible if the speed and the size requirement of the circuit are a concern. As an analogy, think of a programmer who wants his code to be as fast and efficient as possible. The programmr does better write the code in assembly (or even better - machine language) instead of relying on a the high level language compiler. For the exercises here, problems mainly focus on translating VHDL code to hardware. In other words, VHDL will be used as a programming language to map into hardware. Again, remember the concept of *hardware synthesis*. In *hardware synthesis*, the digital designer needs to be conscious of the final form of the mapped circuit.

```
--
-- behavior modeling of D latch
--


entity lat_db1 is
port(       c, d: in bit;       -- input data
            q, nq: inout bit
);
end lat_db1;

architecture behavior of lat_db1 is

begin

-- behavior model

behavior: process (c)
begin
      if c='1' then
            q      <= d;
            nq     <= not d;
      end if;

end process behavior;

end behavior;
```

Listing 2-3. D Latch Behavior Modeling

Copyright 1996, CERL / EE, V1.00

Another example of behavior modeling is again the D flip flop construction using two D latches which was illustrated earlier in Figure 2-5. Listing 2-4, however, shows the construction of a D flip flop without using any D latches. By using the behavior modeling technique, the VHDL code length was reduced drastically. A circuit similar to a D flip flop in size would be tedious to describe totally described in a structural model.

```
--
-- behavior modeling of D latch
--
entity rdffb is
port(
        reset: in bit;       -- reset signal
        c, d: in bit;        -- input data, c=clock, d=data
        q, nq: out bit ); -- output q and not q
end rdffb;

-- behavior model
architecture behavior of rdffb is
begin

  behavior: process (c, reset)
  begin
        if reset='1' then
                q      <= '0';
                nq     <= '1';
        elsif (c='1' and c'event) then
                q      <= d;
                nq     <= not d;
        end if;

  end process behavior;

end behavior;
```

Listing 2-4. D Flip Flop Behavior Modeling

2.2 Problems

1) Implement a D latch with reset using VHDL structure modeling. Demonstrate its functionality in the Cypress NOVA simulator by resetting the circuit, latching in a high value, and resetting again at the next clock cycle when the input is kept high. Save the simulation session to a stimulus file (*.SIM) and trace file (*.PSD) using the NOVA FILE menu items.

2) A T flip flop can be defined as a flip flop that toggles the output at the edge of the clock cycle, for example, rising or falling edge. Implement T flip flop behaviorally. Again, save the simulation session in both stimulus file and trace file.

3) Implement an RS flip flop using an AND gate, and model it structurally. Demonstrate and show its functionality.

4) In VHDL, implement the circuit described below using two D flip flops with /RESET input. The circuit should first be implemented in a behavior model and, then, in a structure model. Show the functionality using the VHDL simulator. (Hint: Utilize /RESET input).

   INPUT: Two input lines, A and B.
   OUTPUT: One output C.
   CONDITIONS:
   a) The output goes to 1 on every positive transition of the A line.
   b) The output goes to 0 on every positive transition of the B line.
   c) The output must be capable of set to 1 again after it being to 0.
   d) Input B takes precedence over Input A to affect the output.

Copyright 1996, CERL / EE, V1.00

3. State Machines and Programmable Logic Devices

3.1 Introduction

Simple programmable devices that we saw earlier can be extended into state machine by a combining memory elements (such as latches and flip flops) with combinational logic elements such as an AND-OR array. A classical example of constructing state machine uses a PROM and latches. Although the construction is simple, the design lacks flexibility for testing. In addition, the use of discrete PROM and latches for creating state machine suffers performance loss due to the hardware interconnection overhead. Modern state machine construction uses programmable devices such as PLDs that integrate memory and discrete elements into one block of hardware real-estate.

3.2 Mealy and Moore State Machines

There are two classical state machine constructions, and they are described in most of the digital design textbooks. They are Mealy and Moore state machines. It can be seen from Figure 3-1 that both state machines are very similar. The Moore machine uses only the output of a delay element to generate the system output whereas the Mealy machine uses both the system input and a delay element output for system output. In theory, both machines are equivalent, so one can convert a synchronized digital hardware design from a Mealy to a Moore state machines or vice versa. Such a statement is not true in the real world of digital design. The Moore state machine generates output based only on its current state. Therefore, the Moore machine output is usually synchronized with the system clock or the clock edge of the output D flip flop (the delay element). In the Mealy machine, because output depends on both the input and current state, output will change whenever the input changes. In other words, in the Mealy system, when output is released at a clock edge, the input is usually latched and considered prior to the release of latched output.

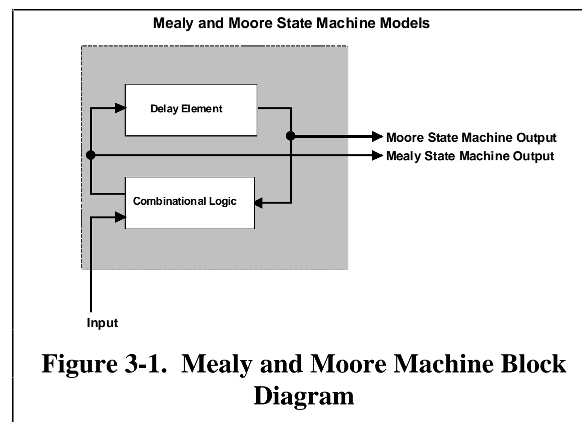


Figure 3-1. Mealy and Moore Machine Block Diagram

The Mealy machine is more difficult to understand than the Moore machine. Because the Mealy machine outputs based on the input and the current state, the correct initiating output process in a Mealy machine is first to change the input value. Due to the dependency of input and the current state, Mealy machine output changes next. However, the change at the input will also trigger the change at the machine state. At last, the output will change again due the new state. Notice that Mealy machine output only at the edge of the state transition, see Figure 3-3. whereas the Moore machine output coincide with the state change, see Figure 3-5.

An VHDL example of a rising edge detector will be constructed using both machines. First, the state diagram of such device as a Mealy machine is shown in Figure 3-2.

Copyright 1996, CERL / EE, V1.00

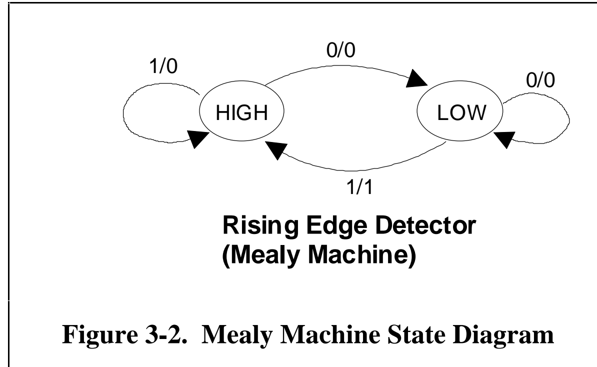


Figure 3-2. Mealy Machine State Diagram

There are two states in the state machine: HIGH and LOW. Notation *x/y* next to an arc between states stands for input/output values during the state transition. The machine advances to the next state when it sees an input *x*. In mean time, the machine also outputs *y*. The VHDL code for the state machine shown in Figure 3-2 is listed in Listing 3-1.

Listing 3-1 contains the usual VHDL components such as ***entity***, ***architecture*** declaration and the VHDL body. A ***type*** statement after the ***architecture*** declaration declares the enumerated type *StateType*. It is advantageous to use VHDL enumerated type so each state of the state machine can have meaningful name (for example, HIGH and LOW). Two signal identifiers *present_state* and *next_state* are declared and used like any identifiers, except they can only be assigned an enuerated state value, HIGH or LOW. The use of the identifiers *present_state* and *next_state* can be found in the ***process*** blocks: *state_machine*, *state_clocking*, and *mealy_output* process statements. These blocks of the VHDL program execute their statements in parallel. In particular, depending on the input, *state_machine* controls the state to which to advance. *Mealy_output* generates output depending on the current state and input. *State_clocking* dictates the synchronization between the state machine and clock rising edge. The timing diagram is shown in Figure 3-3. This figure shows that input triggers output instantly and that present_state does not change its value until the beginning of the next rising clock edge. Thus the timing diagram, Mealy machine timing diagram, agrees with which statement stated above where outputs at the state transition of clock rising edges. In addition, it also shows that Mealy machine output is not registered and the machine will reflect whatever input value it sees on its output.

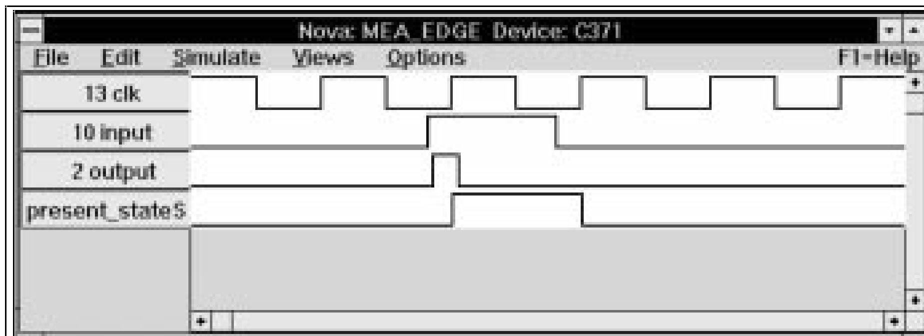


Figure 3-3. Mealy Machine Timing Diagram

Given a Mealy machine, it is possible to convert it to an equivalent Moore machine using a deterministic algorithm. With the differences of Mealy and Moore machines in mind, the algorithm consists of the folllowing steps:

Copyright 1996, CERL / EE, V1.00

1) Put the state machine diagram of edge detector in Figure 3-2 into table form, as shown in Table 3-1. Realize that the table entry is constructed based on state transitions. Because there are four transitions, four table entries (four columns) are shown.

2) Combine the last two rows of table entries, output state and output value, to form new Moore machine states. According to the table, three unique states result: HIGH0, HIGH1, and LOW0.

Table 3-1.

Entry #	**1**	**2**	**3**	**4**
Input State	HIGH	LOW	LOW	HIGH
Input Value	1	1	0	0
Output State	HIGH	HIGH	LOW	LOW
Output Value	0	1	0	0

3) Create the new Moore machine state table from Table 3-1 using the new states. For example, the Mealy state machine transition table entry column 1 contains four values: input state, input value, output state, and output value. As shown before, output state and output value are used to generate the corresponding Moore machine state, HIGH0. Input state and input value are to be used to generate the corresponding Moore machine state transitions. Here, Mealy input state HIGH is replaced by the Moore state HIGH0 and HIGH1 (HIGH plus all possible outputs). Now, two corresponding Moore state transitions are derived; HIGH0 and input 1 goes to HIGH0, HIGH1 and input 1 goes to HIGH0. The same process applies to the other entry columns (2, 3, and 4), see Table 3-2.

Table 3-2.

Moore States	**Input 0**	**Input 1**	**Output Value**
LOW0	LOW0	HIGH1	0
HIGH0	LOW0	HIGH0	0
HIGH1	LOW0	HIGH0	1

4) Make a Moore machine state diagram based on this state transition table, as shown in Figure 3-4.

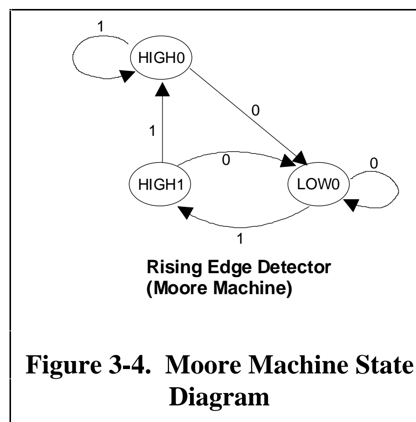


Figure 3-4. Moore Machine State Diagram

5) Describe the Moore machine in VHDL constructs, as shown in Listing 3-2.

Copyright 1996, CERL / EE, V1.00

6) Express the Moore machine operation in the form of a timing diagram. The timing Diagram of the derived Moore machine is shown in Figure 3-5. In the figure, the output signal coincides with the state transition, as is expected from a Moore state machine.

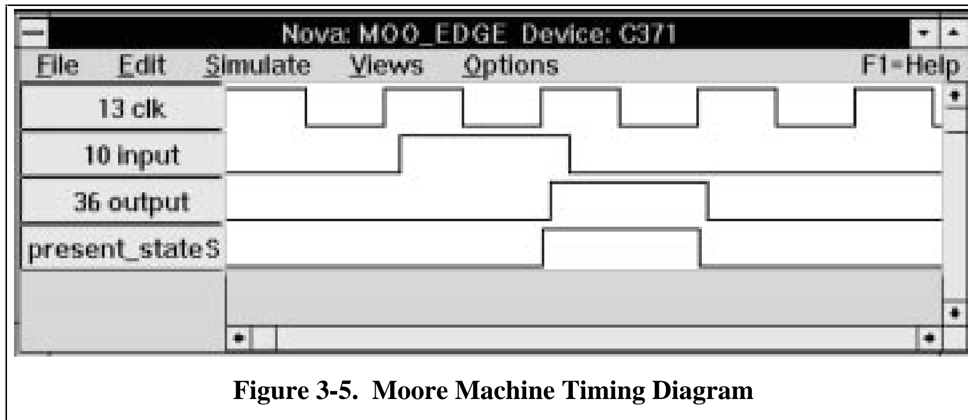


Figure 3-5. Moore Machine Timing Diagram

The inverse process of converting a Moore state machine to a Mealy state machine is simpler. The steps are rather trivial and consist of changing every state output in the Moore machine to the out going transition output of the corresponding Mealy machine. This process effectively converts a machine from a state output state machine to a transition output state machine. See Figure 3-6.

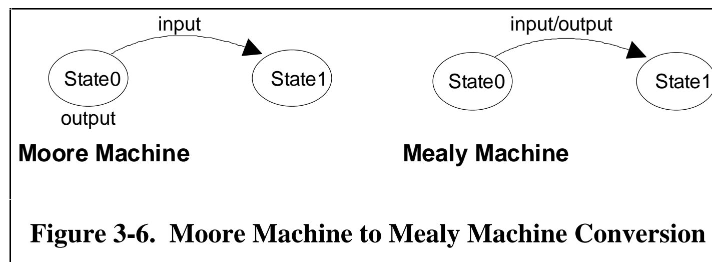


Figure 3-6. Moore Machine to Mealy Machine Conversion

When selecting Mealy versus Moore state machine types for a design, one needs to keep in mind the advantages and disadvantages of each construction. For a Mealy machine, system equations are intuitive to derive because input is readily available. Due to its simplicity, a Mealy machine contains fewer memory elements than a Moore machine. However, because of input dependency, Mealy output also depend highly on the input. For slow input, this phenomenon can translate into a performance bottleneck. Moore machine, on the other hand, does not suffer such input dependency since it “buffers” the system input. However, the tradeoff will be that Moore machine needs more hardware to construct.

3.3 VHDL Programs

3.3.1 Mealy State Machine

```
entity mea_edge is port (
        reset, clk, input: in bit;
        output:     inout bit;
        mea_output:     out bit );
end mea_edge;
------------------------------------------------------------------
architecture mealy_machine of mea_edge is

type StateType is (LOW, HIGH);
signal present_state, next_state: StateType;

begin
-------------------------------------------------------------------
```


Copyright 1996, CERL / EE, V1.00

```
state_machine: process(present_state) begin
  case present_state is
    when LOW   =>   if (input='0') then
                      next_state      <= LOW;
                    else
                      next_state <= HIGH;
                    end if;
    when HIGH  =>   if (input='0') then
                      next_state      <= LOW;
                    else
                      next_state      <= HIGH;
                    end if;
    end case;
end process state_machine;
---------------------------------------------------------------------
state_clocking: process(clk) begin
  if (clk'event and clk='1') then
    present_state <= next_state;
    mea_output  <= output;
  end if;
end process state_clocking;
----------------------------------------------------------------------
mealy_output: process(input, present_state) begin
  if (present_state=LOW and input='1') then
    output <= '1';
  else
    output <= '0';
  end if;
end process mealy_output;

end mealy_machine;
```

Listing 3-1. Mealy Machine VHDL Code

3.3.2 Moore State Machine

```
entity moo_edge is port (
        reset, clk, input: in bit;
        output:      out bit );
end moo_edge;

architecture moore_machine of moo_edge is

type StateType is (LOW, HIGH0, HIGH1);
signal present_state, next_state: StateType;

begin

state_machine: process(present_state) begin

  case present_state is
    when LOW   =>   if (input='0') then
                      next_state      <= LOW;
                    else
                      next_state <= HIGH1;
                    end if;
    when HIGH0 =>   if (input='0') then
                      next_state      <= LOW;
                    else
                      next_state      <= HIGH1;
                    end if;
    when HIGH1 =>   if (input='0') then
                      next_state      <= LOW;
                    else
                      next_state      <= HIGH0;
                    end if;
    end case;
end process state_machine;
```


Copyright 1996, CERL / EE, V1.00

```
state_clocking: process(clk) begin
  if (clk'event and clk='1') then
    present_state <= next_state;
  end if;
end process state_clocking;


moore_output: process(present_state) begin
  if (present_state=HIGH1) then
    output <= '1';
  else
    output <= '0';
  end if;
end process moore_output;

end moore_machine;
```

Listing 3-2. Moore Machine VHDL Code

3.4 Problems

1) Convert the Moore machine in Figure 3-7 into a Mealy machine. Implement the Mealy machine in VHDL and simulate it with Cypress NOVA simulator.

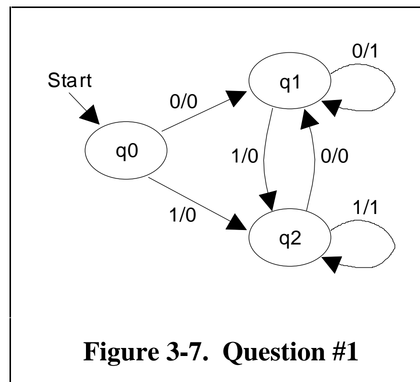


Figure 3-7. Question #1

2) Simulate the state machine shown in Figure 3-8 with VHDL and describe what it does (Hint: See next question).

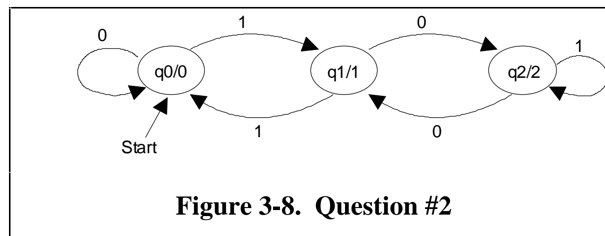


Figure 3-8. Question #2

3) Design a Mealy state machine that computes the remainder of value 5 instead 3. Draw the state diagram and simulate it with VHDL. Test the machine using input 3, 4, 5, 6, 76, and 101.

4) Design the state machine of the state flow diagram in Figure 3-9 and the output table in Table 3-3.

Copyright 1996, CERL / EE, V1.00

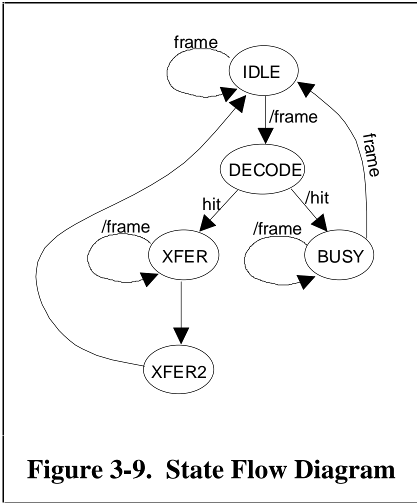


Figure 3-9. State Flow Diagram

Table 3-3.

State / Output	OE	GO	ACT
IDLE	0	0	0
DECODE	0	0	0
BUSY	0	0	1
XFER	1	1	1
XFER2	1	0	1

Copyright 1996, CERL / EE, V1.00

4. Digital Device Modeling

4.1 Introduction

Several VHDL concepts have been addressed in the previous lessons. VHDL topics covered in the previous lessons are various digital circuit modeling methodologies: structural, behavioral, and dataflow models. In addition to digital modeling, the text also covered the implementation of two types of state machines: Moore and Mealy state machines using the VHDL ***process*** statement. The ***Process*** statement is a unique feature in comparison to conventional imperative programming languages such as C, Pascal, FORTRAN, Ada, C++, etc. Due to the concurrent nature of VHDL programming, ***Process*** blocks execute/simulate in parallel. Statements inside of the ***process*** block, on the other hand, are executed sequentially. Parallel simulation is important because real world digital systems operate and behave in parallel. In the following lesson, this same concept of parallel executing ***process*** blocks will be addressed again. The emphasis here will be more towards the solving of real world application problems such as digital component modeling.

4.2 SRAM Memory

Memory plays an important role in the advancement of modern computers. When design a computing system, different types of memory are hierarchically structured into layers according to their data handling speed. The CPU is usually the fastest component and it is surrounded by various levels of memory. The various types of memory work in conjunction with the CPU: for example, on-chip and off-chip memory, memory chips surround the CPU on the motherboard, hard disk memory, and secondary memory device such as tape backups. To make computer efficient so the CPU will not be bogged down by the slower memory, different speeds of memory need to be used. Memory used by CPU are classified in general as SRAM (static RAM) and DRAM (dynamic RAM). SRAM memory runs faster than DRAM, and it interfaces directly with the CPU or the CPU on chip memory such as the L1 cache of the system. DRAM, on the other hand, is slower than SRAM by an order and requires more control circuitry, but has the advantages of smaller cell size and lower cost.

DRAM memory is denser than SRAM due to its circuit simplicity. Unlike SRAM which uses flip flops, DRAM stores information in capacitors, and information will only be kept until the charges in the capacitors leak out. As a consequence, more effort is needed to design DRAM “refresh circuitry” to preserve the stored DRAM information. Refreshing DRAM means re-charging the capacitors.

SRAM memory is much simpler to use than DRAM because it does not require a “refresh” operation. A generalized SRAM timing diagram is shown in Figure 4-1. SRAM is an asynchronous device. Read and write operations depend on the chip’s selecting signal *nce* shown in Figure 4-1. Typical timing parameters found in an SRAM data book include signal *nce* to various signal edges, which also shown in Figure 4-1.

The SRAM signals shown in Figure 4-1 are described in Table 4-1. The signal type entry in the table describes the signal direction attribute such as input, output or bidirectional. The line *bidata* is set to high impedence when it is not used. By doing so, *bidata* will not cause data line conflict.

Timing parameters for an SRAM are also illustrated in Figure 4-1. They are the minimum times starting from the signal *nce* to various signals and they are the same for both read and write operations. Signal *nwrite* indicates read or write operation. If the SRAM is in write mode, *bidata* will be the input for capturing data. Oppositely, if the SRAM is in read mode, *bidata* will be used to output data.

Copyright 1996, CERL / EE, V1.00

Figure 4-1. SRAM Timing Diagram

Table 4-1. SRAM Signal Description

Signal Name	Signal Type	Signal Description
NCE	IN	Not chip select signal.
NWrite	IN	Not chip write or chip read signal.
Address	IN	Use for read and write address.
Data	IN/OUT	Use for data input and data output.

4.3 VHDL SRAM Memory Design

SRAM modeling can be implemented using the above timing information and previous VHDL lessons. A VHDL program that models a 2 bit by 4 word SRAM device is listed in Section 4.4. This program was compiled and simulated using the Cypress Warp tool. The simulation waveform is shown in Figure 4-2. From the figure, it can be seen that the data value 2 was written into the memory location 3 (*bidata*(1)='1' and *bidata*(0)='0') and read back afterwards.

Figure 4-2. SRAM Simulation Waveform

Several VHDL concepts were used in this program. The first is the use of the ***generic*** statement. The ***Generic*** statement is similar to constant declaration in programming languages except it is more useful

Copyright 1996, CERL / EE, V1.00

due to the hardware design application. Notice that we have discussed a VHDL method for declaring constants in earlier lessons. ***Generic*** statement is different because it specifies where those constant are valid. In other words, the ***generic*** statement defines the scope of its constants. The ***Generic*** statement is to be used inside of an ***entity*** statement (see previous lessons), and the architecture body associated with this entity will be able to see and use these generic constants. It is a very powerful concept because it allows generalized hardware modeling. In the SRAM example, the size of the SRAM does not need to be fixed at 8 words deep and 2 bits per word. Different size SRAMs can be created by changing only the declaration inside of the ***generic*** statement.

The second concept is the use of the sub-program ***procedure*** statement. Although the ***component*** statement was introduced first, the ***procedure*** statement is the preferred construct due to its simplicity. The component structure is very much the same as the architecture structure. It uses an entity statement to specify input and output signals and an architecture body to define the relationship(s) of these signals. Declaration of a component is also needed in the main architecture body. As clear as the component concept may seem it, there are many hassles in creating and using ***component*** for simulation. Unless the ***component*** is useful or has already been created and put into a library, a ***procedure*** is the prefer choice.

Another reason why a ***procedure*** construct is preferred over a ***component*** construct is for ease of understanding VHDL code. In general, VHDL code for structure modeling is less likely to be comprehended than the corresponding behavior modeling which is naturally associated with ***procedure*** statement. The fact that behavior modeling is easier to grasp was shown in the previous lesson where a D flip-flop was represented in different VHDL models.

The third concept is the use of library routines. The ***use*** statement can be found between the ***entity*** statement and ***architecture*** statement. Since the Cypress VHDL compiler is mainly geared towards hardware synthesis, the behavior modeling aspect of Cypress VHDL compiler is weak because it doesn't have good support for modeling a resistor. For example, in the SRAM modeling below, to multiplex bi-directional line, *bidata*, component tristate buffer, *bufoe*, from the Cypress library *rtlpkg* would have to be used.

4.4 SRAM VHDL Listing

```
----------------------------------------------------------------------
-- Entity statement
--
-- Specifies input, output and bidirectional signals
-- Specifies
----------------------------------------------------------------------
entity sram0 is

    generic(
        address_width  : integer := 2;   -- column address width
        address_size   : integer := 4;
        word_width     : integer := 2;   -- word width
        zero_word : bit_vector := "00"  -- To nullify a memory
    );

    port(
        address      : in bit_vector (address_width-1 downto 0);
        nwrite       : in bit;
        nce          : in bit;
        bidata       : inout x01z_vector (word_width-1 downto 0)
    );
end sram0;


use work.rtlpkg.all;
    -- bufoe(x, oe, y, yfb): Three state buffer with feedback.
use work.int_math.all;  -- See Cypress VHDL synthesis reference page 4-86
        --  i2bv(i,w): converts integer i to w width binary
```


Copyright 1996, CERL / EE, V1.00

```
        --      signal.
        --  bv2i(a):   inverse of function i2bv



----------------------------------------------------------------------

architecture  sram0arch of sram0 is

    type memory_elmt is array (address_size downto 0) of
        bit_vector (address_width downto 0);


----------------------------------------------------------------------
-- Read from the memory
----------------------------------------------------------------------

    procedure ReadMemory
        ( nce, nwrite: in bit;
        address: in bit_vector (address_width-1 downto 0);
        memory_word: in memory_elmt;
        dataout: out bit_vector (word_width-1 downto 0)) is

    variable i, w: integer;
    variable addressi: integer;
    begin
        if (nce='0') and nce'event then
            if (nwrite='1') then
                addressi := bv2i(address);
                for i in 0 to address_size-1 loop
                    if(i=addressi) then
                        for w in 0 to word_width-1 loop
                            dataout(w) <= memory_word(i)(w);
                        end loop;
                    end if;
                end loop;
            end if;
        end if;
    end ReadMemory;

----------------------------------------------------------------------
-- Write to the memory
----------------------------------------------------------------------
    procedure WriteMemory
        ( nce, NWrite: in bit;
        Address: in bit_vector (address_width-1 downto 0);
        memory_word: out memory_elmt;
        datain: in bit_vector (word_width-1 downto 0)) is
    variable i, w: integer;
    variable addressi: integer;
    begin
        if (nce='0') and nce'event then
            if (nwrite='0') then
                addressi := bv2i(address);
                for i in 0 to address_size-1 loop
                    if(i=addressi) then
                        for w in 0 to word_width-1 loop
                            memory_word(i)(w) <= datain(w);
                        end loop;
                    end if;
                end loop;
            end if;
        end if;
    end WriteMemory;


----------------------------------------------------------------------
-- Main program
----------------------------------------------------------------------
```


Copyright 1996, CERL / EE, V1.00

```
signal datain, dataout: bit_vector (word_width-1 downto 0);
signal memory_word:     memory_elmt;
signal w: integer;
begin

-- Memory read cycle

    process (nce)
    begin
        ReadMemory(nce, nwrite, address, memory_word, dataout);
    end process;

-- Memory write cycle

    process (nce)
    begin
        WriteMemory(nce, nwrite, address, memory_word, datain);
    end process;

-- Tristate input and output

    bif0: bufoe port map(dataout(0), nwrite, bidata(0), datain(0));
    bif1: bufoe port map(dataout(1), nwrite, bidata(1), datain(1));


end sram0arch;
```

Listing 4-1. SRAM VHDL Code

4.5 Problems

1) The Cypress Warp tool uses the time unit tick. Figure 4-2 shows an example of tick value 68. From Figure 4-1, use the Warp simulator to determine the minimum times for signals $T_{nce2datain}$, $T_{nce2nwr}$ and $T_{nce2addr}$ that produce a valid $T_{nce2dataout}$. Also, what is the value of $T_{nce2dataout}$?

2) The timing diagram in Figure 4-3 below shows a modified SRAM that contains an extra field for storing a valid bit. This memory is to be named SRAMvc (SRAM with valid control) memory. To put data into a memory location, the signal *nvalid* has to be set to zero prior to signal *nce*'s going during the SRAMvc reading cycle, if the signal *nvalid* at the requested address location is high, *bidata* should output zero. During the SRAMvc writing cycle, data will write into the memory along with the *nvalid* signal. If the signal *nvalid* is high, the maximum data value (all bits set to one) should write into the corresponding memory location.

   Before using SRAMvc, a signal *nreset* should be applied to invalidate signal *nvalid* all memory locations (for example, *nvalid*<=‘0’). In result, all *nvalid* bits will be set to zero, and all *bidata* value will be set to max. See Figure 4-3 for the timing relationship of *nreset* and *nce*.

   Construct SRAMvc using VHDL and prove its functional by simulation. The simulator should produce timing waveforms similar to those shown in Figure 4-3.

   From the simulation, find all the timing parameters in Figure 4-3 based on the simulation of your SRAMvc VHDL model.

Copyright 1996, CERL / EE, V1.00

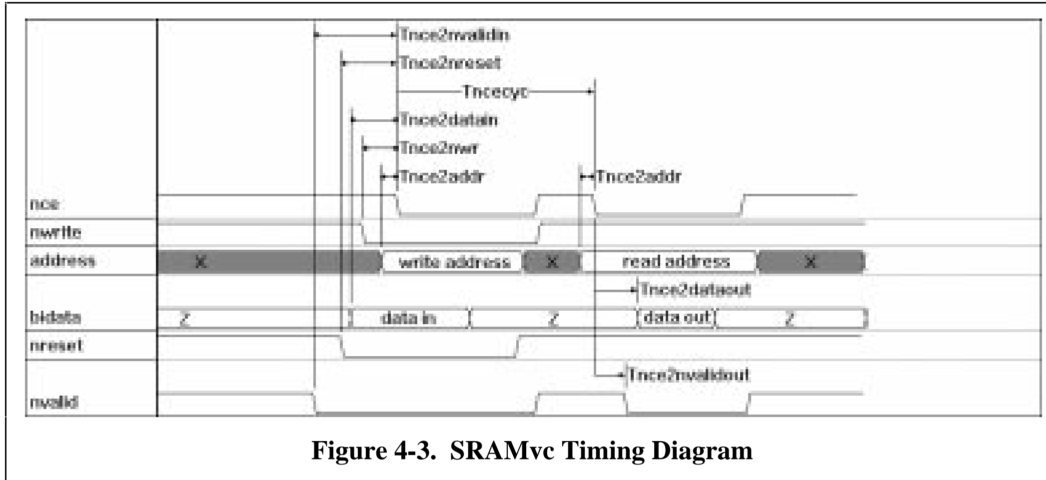


Figure 4-3. SRAMvc Timing Diagram

Copyright 1996, CERL / EE, V1.00

5. Digital Design using Divide and Conquer

5.1 Introduction

As the demand for digital system complexity increases, one needs to rely on traditional divide and conquer methodologies for constructing complex digital systems. VHDL contains features that support the divide and conquer technique for designing complex system design.

Some of the VHDL features that that permit the designer to build sub-systems have already been introduced in previous lessons: for example, the use of VHDL ***component*** block and ***procedure*** and ***functions*** statements. In addition, there is a benefit to breaking a large problem into many smaller problems because the solution for each small problem can be reused and applied to other problems similar in nature. For example, the design of an automobile fuel mixer controller might require the use of a two stage 5 bit D flip flop. By coincidence, this same odd device (2-stages, 5-bit D flip flop) is also used inside traffic light controller. One can imagine that the design for this device should not have to be implemented twice, provided the design tool (I.e. VHDL) allows it. This is the concept of hardware reuse, and for a long time it has been employed in the *object oriented* software world. In this lesson, the VHDL ***package*** and ***library*** between which support “hardware reuse” will be introduced.

5.2 VHDL features

The use of VHDL ***component*** and ***procedure*** statements in previous lessons gave the VHDL designer a systematic way of breaking a large and complicated problem into smaller, and simpler problems to solve. The VHDL ***package*** clause feature lets the digital designer pack tested digital parts into a library (storage). Then, he/she can use the VHDL keyword ***use*** or ***library*** to retrieve these parts for any design afterwards. This enables designer to build more complex systems and also encourages the digital design divide and conquer methodologies.

5.3 ALU Functions

The Arithmetic logic unit (ALU) is the most important part of the computation engine because it directly affects the system performance. All computational arithmetic or logic operations are done in the ALU. Therefore, it is clear that ALU output latency will affect the overall input to output delay. Figure 5-1 shows the general block diagram of an ALU.

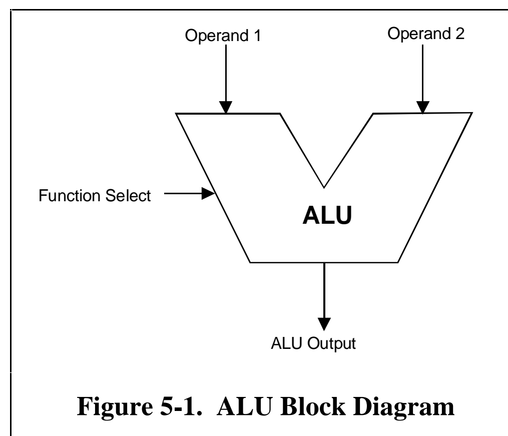


Figure 5-1. ALU Block Diagram

The types of functions built into an ALU depends on the nature of computation. For example, a number crunching ALU such as a digital signal processor (DSP) will devote a good portion of ALU resources for floating point number manipulation such as floating point add, multiply, etc. An intelligent embeded

Copyright 1996, CERL / EE, V1.00

controller, on the other hand, would probably have an overall balanced arithmetic and logic functions, with emphasis on integer manipulation.

There are many ways to construct an ALU. During the 1980's, when Large Scale Integration (LSI) Integrated Chip (IC) and Medium Scale Integration IC were popular, bit-slice ALU design was the main technology due to its modularity for LSI and MSI design. The bit-slice design concept was used because it reduced computations to individual bits as to minimize each bit's overlapping functionality. In the current trend of integrated circuit technology (e.g. VLSI), ALU designers focus less on the ALU's bit versatility and more towards the its overall speed performance. An ALU design example is shown in Figure 5-2.

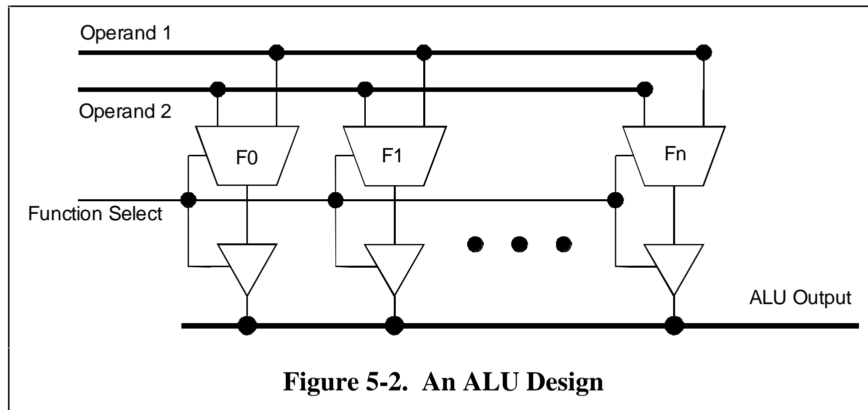


Figure 5-2. An ALU Design

Figure 5-2 is a simplified ALU design, nevertheless, it shows a good overall ALU structure. In the ALU, function boxes, F0, F1, to Fn are dedicated for specified ALU functions such as add, multiply, shift, etc. The inputs to each function box are delivered by two buses, operand1 and operand2. Output of each function box is tapped onto the output bus via tri-state buffers. In addition to the resulting data outputs, there are also ALU flag outputs such as divide by zero, arithmetic overflow, underflow, etc. Each function box is controlled by the control input bits. When functions are similar in nature, for example, left circular and right circular shifts, control bits are used to distinguish these functions.

There are many arithmetic and logic functions that a computer architecture designer might like to put into an ALU. However, due to design requirements and technology limitations the designer must compromise all the factors for an optimal area and speed tradeoff.

5.4 ALU Component Implementation Examples

For simple laboratory work, one might like to design a 4-bit, 2's complement ALU which consists a 4-bit adder for integer arithmetic, left circular shift, right circular shift, and bitwise AND and OR logic functions.

VHDL supports modular hardware implementation by using the objected oriented software concept of reusability. In VHDL, code or program pieces such as a ***procedure***, ***function*** or ***component*** can be stored into a ***library*** for later use. Whenever the parts are needed, they can be linked into the main VHDL program and reused again. The VHDL feature that supports part reuse is called ***Package***, and it is described in the ALU implementation example.

5.4.1 Ripple Carry Adder Implementation

A ripple carry adder is the simplest adder and it is easiest to build. To implement a 4-bit adder, one starts by paralleling a half adder with full adders. Ripple carry adder implementation can be found in many digital design text books. this laboratory, half adder and full adder components were implemented and

Copyright 1996, CERL / EE, V1.00

they have been placed inside of the library, *Arith_parts_pkg package*. The listing of the *Arith_parts_pkg package* is shown in Section 5.4.2.

In *Arith_parts_pkg*, components half_adder and full_adder were combined to produce a 4-bit ripple carry adder *ripp_adder4*. One example of hardware reuse is to test the functionality of *ripp_adder4* without re-implementing the same module. This is done by creating a VHDL module, *addtest.vhd*, which is listed in Section 5.4.3. In file *addtest.vhd*, after the ***entity*** declaration, the line beginning with the VHDL key word *use* tells the VHDL compiler that all modules (*procedure*, *function*, and *component*) inside of the package *iarith_parts_pkg* are visible to file *addtest.vhd* and therefore are available for use.

Only the module *ripp_adder4* is used structurally inside of the *addtest* module. Remaining modules such as *full_adder* and *half adder* are not shown inside of the *addtest* module; nevertheless, they were important in building the ripple carry adder *ripp_adder4*. The test wave form which illustrates the adder's functionality is shown in Figure 5-3. In Figure 5-3, *ain* is set to "1111" and *bin* is set to "0011". The output *aout* becomes "0010" with the overflow flag set to 1. Notice the glitches on the at *aout* lines caused by the asynchronous *ain* and *bin* input signals.

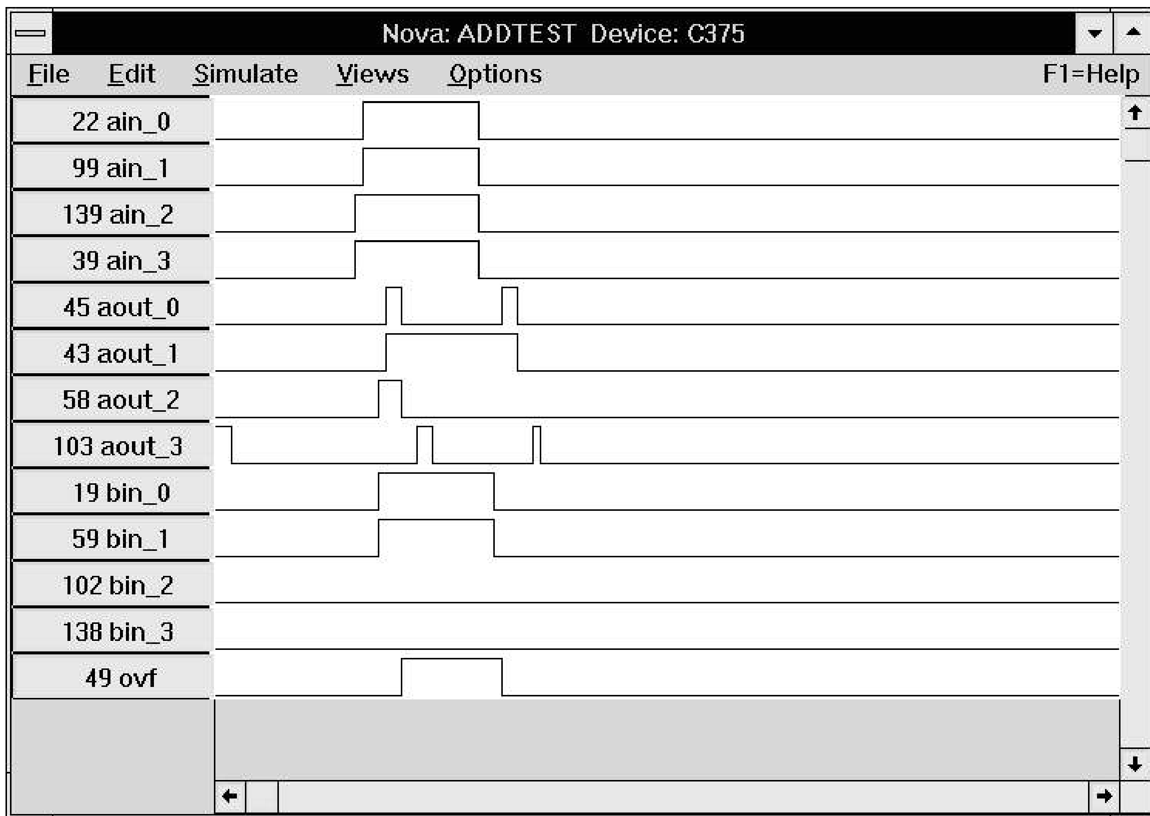


Figure 5-3. Ripple Carry Adder Waveform

5.4.2 Package *Iarith_parts_pkg* Listing

```
package iarith_parts_pkg is

    component half_adder
        port(signal Ain, Bin:    in bit;
                    signal Cout, Sout:  out bit);
    end component;

    component full_adder
        port(signal Cin, Ain, Bin:  in bit;
```


Copyright 1996, CERL / EE, V1.00

```
                signal Cout, Sout:     out bit);
        end component;

        component ripp_adder4
                port(signal Ain, Bin:   in bit_vector (3 downto 0);
                        signal Sout:       out bit_vector (3 downto 0);
                        signal OverFlow:   out bit);
        end component;
end iarith_parts_pkg;

----------------------------------------------------------
-- Half adder component

entity half_adder is
  port(signal Ain, Bin:  in bit;
    signal Cout, Sout:   out bit);
end half_adder;

architecture half_adder_arch of half_adder is

begin
  process (Ain, Bin)
  begin
    if Ain='1' and Bin='1' then
      Sout  <= '0';
      Cout  <= '1';
    else
      Cout  <= '0';
      Sout  <= Ain or Bin;
    end if;
        end process;

end half_adder_arch;

----------------------------------------------------------
-- Full adder component

entity full_adder is

  port(signal Cin, Ain, Bin:  in bit;
       signal Cout, Sout:     out bit);

end full_adder;

architecture full_adder_arch of full_adder is
begin

  process (Ain, Bin, Cin)
  begin
    if Ain='1' and Bin='1' and Cin='1' then
      Sout  <= '1';
      Cout  <= '1';
    elsif Cin='0' then
      if Ain='1' and Bin='1' then
        Sout  <= '0';
        Cout  <= '1';
      else
        Sout  <= Ain or Bin;
        Cout  <= '0';
      end if;
    else  -- Cin='1' and either Ain='1' or Bin='1'
      Sout  <= not (Ain or Bin);
      Cout  <= Ain or Bin;
    end if;
  end process;

end full_adder_arch;

----------------------------------------------------------
```


Copyright 1996, CERL / EE, V1.00

```
-- 4 bit adder component

entity ripp_adder4 is
        port(signal Ain, Bin:   in bit_vector (3 downto 0);
                        signal Sout:       out bit_vector (3 downto 0);
                        signal OverFlow:   out bit);
end ripp_adder4;

architecture ripp_adder4_arch of ripp_adder4 is
signal Cout: bit_vector (3 downto 1);
begin

  ha0: half_adder
    port map(Ain=>Ain(0), Bin=>Bin(0),
                                        Cout=>Cout(1), Sout=>Sout(0));


  fa1: full_adder
    port map(Cin=>Cout(1), Ain=>Ain(1), Bin=>Bin(1),
            Cout=>Cout(2), Sout=>Sout(1));

  fa2: full_adder
    port map(Cin=>Cout(2), Ain=>Ain(2), Bin=>Bin(2),
            Cout=>Cout(3), Sout=>Sout(2));

  fa3: full_adder
    port map(Cin=>Cout(3), Ain=>Ain(3), Bin=>Bin(3),
            Cout=>OverFLow, Sout=>Sout(3));


end ripp_adder4_arch;
```

Listing 5-1. Iarith_parts_pkg VHDL Code Listing

5.4.3 *Testadd.vhd* Listing

```
---------------------------------------------------
-- to test ripple adder routine using component
-- "ripp_adder4" of vhdl file iar.vhd.
---------------------------------------------------
entity addtest is

        generic( data_wd: integer := 3);

        port (      Ain: in bit_vector (data_wd downto 0);
                    Bin: in bit_vector (data_wd downto 0);
                    Aout:out bit_vector(data_wd downto 0);
                    OVF: out bit);
end addtest;

use work.iarith_parts_pkg.all;

architecture addtest_arch of addtest is

begin

                op4: ripp_adder4 port map(Ain, Bin, Aout, OVF);

end;
```

Listing 5-2. Testadd.vhd VHDL Code Listing

Copyright 1996, CERL / EE, V1.00

5.4.4 Circular Shifter Implementation

The circular shifter unit is more difficult to implement than simple bitwise logical operations. To make the implementation more challenging, the shifter was implemented in generalized fashion so it could accommodate various bus widths. Again, by applying the concept of reuse, the implementation of the circular shifter was put into a library for test. The library that contains the shifter is called *logic_parts_pkg*, and the VHDL code is listed in Section 5.4.6. Again, the testing of circular shift logic can be performed similar to the ripple carry adder test. The trace file for doing two right circular shifts on *ain* bits "0010" is "1000" and is shown in the Section 5.4.5. Notice that the output "1000" took 3 ticks to come out after applying inputs *ain* and *bin*. *Ain* is the bit to be shifted and *bin* is the shifted value. In the *cshift* component of *logic_parts_pkg*, input bit *RL* was used to indicate shift direction: RL='0' for left circular shift and '1' for right circular shift.

5.4.5 Circular Shift Trace Data

```
NOVA Simulation Printout

File: TEST.psd -  View: PINS and REGS

Printout produced: Thu Jun 13 01:46:57 1996



      b b a a a a a a a a
      i i i i i o o o o o
      n n n n n u u u u u
      _ _ _ _ _ _ t t t t
      0 1 0 1 2 3 _ _ _ _
      | | | | | | 0 1 2 3
      | | | | | | | | | |
^^^^^^^^^^^^^^^^^^^^^^^^^^^
   0: 0 1 0 0 1 0 L L L L
   1: 0 1 0 0 1 0 L L L L
   2: 0 1 0 0 1 0 L L L L
   3: 0 1 0 0 1 0 H L L L
   4: 0 1 0 0 1 0 H L L L
   5: 0 1 0 0 1 0 H L L L
   6: 0 1 0 0 1 0 H L L L
   7: 0 1 0 0 1 0 H L L L
   8: 0 1 0 0 1 0 H L L L
   9: 0 1 0 0 1 0 H L L L
  10: 0 1 0 0 1 0 H L L L
  11: 0 1 0 0 1 0 H L L L
  12: 0 1 0 0 1 0 H L L L
  13: 0 1 0 0 1 0 H L L L
  14: 0 1 0 0 1 0 H L L L
```

Listing 5-3. Circular Shift Trace Listing

5.4.6 Package *Logic_parts_pkg* Listing

```
package logic_parts_pkg is
    component cshift

        generic (data_wd: integer:= 3);

        port (Ain: in bit_vector (data_wd downto 0);
                    Sin: in bit_vector (1 downto 0);
                    RL : in bit;
                    Sout: out bit_vector(data_wd downto 0));
    end component;

    component bAND
        port (Ain: in bit_vector (3 downto 0);
                    Bin: in bit_vector (3 downto 0);
                    Cout: out bit_vector (3 downto 0));
```


Copyright 1996, CERL / EE, V1.00

```
        end component;

        component bOR
                port (Ain: in bit_vector (3 downto 0);
                                Bin: in bit_vector (3 downto 0);
                                Cout: out bit_vector (3 downto 0));
        end component;

end logic_parts_pkg;


-- Use Cypress library

use work.bv_math.all;
use work.int_math.all;


-----------------------------------------------------
-- Circular right or left shift
--
-- Input:
-- Ain -  Input bits
-- Sin -  Number of shift
-- RL  -  0: left circular shift
--        1: right circular shift
-- Output:
-- Sout - Output bits
-----------------------------------------------------
entity cshift is

        generic(data_wd: integer := 3);

        port( Ain: in bit_vector (data_wd downto 0);
                        Sin: in bit_vector (1 downto 0);
                        RL : in bit;
                        Sout: out bit_vector(data_wd downto 0));

end cshift;


-----------------------------------------------------
architecture cshift_arch of cshift is

signal temp: bit_vector (data_wd downto 0);
begin

process (Ain, Sin, RL)
variable i, j, k, l: integer := 0;
begin

-- i is the shifted value

        for i in 0 to data_wd loop

                if i=bv2i(Sin) then

                        for j in 0 to data_wd loop

-- Find right circular shift index k

                                if RL='1' then
                                        k := j+(data_wd+1)-i;

-- Find left circular shift index k

                                else
                                        k := j+i;
                                end if;
```


Copyright 1996, CERL / EE, V1.00

```
-- Assign Output

                        if k > data_wd then
                                k := k-(data_wd+1);
                        end if;

                        for l in 0 to data_wd loop
                                if k=l then
                                        temp(l) <= Ain(j);
                                end if;
                        end loop;

                end loop;

        end if;

    end loop;

    Sout <= temp;

end process;



end cshift_arch;

-----------------------------------------------------
-- Bitwise AND
----------------------------------------------------
entity bAND is
    port( Ain: in bit_vector (3 downto 0);
                        Bin: in bit_vector (3 downto 0);
                        Cout: out bit_vector (3 downto 0));
end bAND;

----------------------------------------------------
architecture bAND_arch of bAND is
begin

process (Ain, Bin)
begin
    Cout(3 downto 0) <= Ain(3 downto 0) and Bin(3 downto 0);
end process;

end bAND_arch;


-----------------------------------------------------
-- Bitwise OR
-----------------------------------------------------
entity bOR is
    port(Ain: in bit_vector (3 downto 0);
                Bin: in bit_vector (3 downto 0);
                Cout: out bit_vector (3 downto 0));
end bOR;

----------------------------------------------------
architecture bOR_arch of bOR is
begin

process (Ain, Bin)
begin
    Cout(3 downto 0) <= Ain(3 downto 0) or Bin(3 downto 0);
end process;

end bOR_arch;
```

Listing 5-4. Logic_parts_pkg VHDL Code Listing

Copyright 1996, CERL / EE, V1.00

5.5 ALU Implementation Example

Using components from Section 5.4 and the block diagram in Figure 5-2, one should be able effortlessly to construct an ALU. See the ALU VHDL code in Listing 5-5.

```
use work.logic_parts_pkg.all;
use work.iarith_parts_pkg.all;


-----------------------------------------------------
-- alu_comp
--
-- Input:
--
-- Ain, Bin - Input bits
-- sel_bit - Calculation type selection bit
--        0001 - Bitwise AND operation
--        0010 - Bitwise OR operation
--        0100 - Left circular shift
--        0101 - Right circular shift
--        1000 - Integer add.
--
-- Output:
--
-- flags  - Bit 0 for Ain all zero flag, all time
--        Bit 1 for Bin all zero flag, all time
--        Bit 2 for latched adder overflow flag
-----------------------------------------------------
entity alu_comp is

    generic ( data_wd: integer := 3);

    port( Ain, Bin: in bit_vector (data_wd downto 0);
            sel_bit:  in bit_vector (data_wd downto 0);
            clk:          in bit;
            Aout:     out bit_vector (data_wd downto 0);
            flags:        out bit_vector (2 downto 0));
end alu_comp;

-----------------------------------------------------
architecture alu_body of alu_comp is


-----------------------------------------------------
-- calc_ctrl
--
-- Calculation type decoder
--
-- logic_ctrl - * Bit 0 for Bitwise AND tri-buffer
--            * Bit 1 for Bitwise OR tri-buffer
--            * Bit 2,3 for circular shift controller
--              Bit 4 for circular shift tri-buffer
--            * Bit 5 for integer add tri-buffer
-----------------------------------------------------
    procedure calc_ctrl
        (sel_bit:     in bit_vector (3 downto 0);
         ctrl:   out bit_vector (5 downto 0)) is
    begin

        case sel_bit is
            when "0001" =>  -- Bitwise AND operation
                ctrl <= "000001";
            when "0010" =>  -- Bitwise OR operation
                ctrl <= "000010";
            when "0100" =>  -- Left circular shift operation
```


Copyright 1996, CERL / EE, V1.00

```
                ctrl <= "010100";
            when "0101" =>  -- Right circular shift operation
                ctrl <= "011000";
            when "1000" =>  -- Integer add operation
                ctrl <= "100000";
            when others =>  -- Invalid operation
                ctrl <= "000000";
        end case;
    end calc_ctrl;


    signal Ain_lat, Bin_lat: bit_vector (data_wd downto 0);
    signal sel_bit_lat: bit_vector (3 downto 0);
    signal ctrl: bit_vector (5 downto 0);
    signal bAND_out:  bit_vector (data_wd downto 0);
    signal bOR_out:   bit_vector (data_wd downto 0);
    signal cshift_out:bit_vector (data_wd downto 0);
    signal ripp_adder4_out: bit_vector (data_wd downto 0);
    signal overflow: bit;


begin

-- Synchronize data

    process(clk)
    begin
        if clk='1' and clk'event then
            Ain_lat(3 downto 0) <= Ain(3 downto 0);
            Bin_lat(3 downto 0) <= Bin(3 downto 0);
            sel_bit_lat(3 downto 0) <= sel_bit(3 downto 0);
        end if;
    end process;


-- Input select line decode

    process(sel_bit_lat)
    begin
        calc_ctrl(sel_bit_lat, ctrl);
    end process;


-- Operations assignment

    op1: bAND port map(Ain_lat, Bin_lat, bAND_out);
    op2: bOR port map(Ain_lat, Bin_lat, bOR_out);
    op3: cshift port map(Ain_lat, Bin_lat (1 downto 0), ctrl(3), cshift_out);
    op4: ripp_adder4 port map(Ain_lat, Bin_lat, ripp_adder4_out, overflow);

-- Output stage

    process(ctrl)
    begin
        if ctrl(0)='1' then
            Aout <= bAND_out;
        elsif ctrl(1)='1' then
            Aout <= bOR_out;
        elsif ctrl(4)='1' then
            Aout <= cshift_out;
        elsif ctrl(5)='1' then
            Aout <= ripp_adder4_out;
        else
            Aout <= Ain;
        end if;
    end process;


-- Setting alu flags
```


Copyright 1996, CERL / EE, V1.00

```
        process(Ain)
        begin
                if      Ain="0000" then
                        flags(0) <= '1';
                else
                        flags(0) <= '0';
                end if;
        end process;

        process(Bin)
        begin
                if      Bin="0000" then
                        flags(1) <= '1';
                else
                        flags(1) <= '0';
                end if;
        end process;



-- To latch overflow flag

        process(ctrl(5))
        begin
                if ctrl(5)='1' then
                        flags(2) <= overflow;
                end if;
        end process;


end alu_body;
```

Listing 5-5. Example ALU VHDL Code Listing

5.6 Problems

1) Test component *cshift* by constructing a VHDL module *cshtest.vhd.* Test both shift directions, left and right and make sure the test covers the circular wrap effect. The test vectors should be collected in a trace file after running the *cshtest.vhd* simulation. In the trace file, write a comment at the side of the test vectors that corresponding to each distinct operating period.

   Specifically, how long is the input to output delay. Does the delay depend on the number of shift and/or shift direction?

2) Implement a 4-bit arithmetic shift component *ashift. Ashift* takes a similar input to *cshift* except it does an arithmetic shift instead of a circular shift. This component should also be included in the package *logic_parts_pkg*.

3) Implement a 4-bit subtract unit that utilizes components already built, such as the adder. Notice that one can subtract two numbers by complementing the subtractant then and add the numbers. Again, this unit should be included in the package *iarith_parts_pkg*.

4) Implement the ALU shown in Figure 4.18 of Computer Organization and Design by Patterson and Hennessy using the control logic described in Figure 4.19.

   What is the worstcase operational propagation delay in this device that you just implemented?

Copyright 1996, CERL / EE, V1.00

6. ALU Datapth Implementation

6.1 Introduction

After learning ALU modeling in VHDL, one can carry it further by designing an ALU datapath and eventually building a complete CPU. Again, the divide and conquer approach will be used to illustrate the example. A CPU datapath consists of several major components: register file, data memory, instruction memory, and control module. To model the overall CPU datapath activity, it is necessary first to implement and model each component separately. VHDL is the nature candidate for such a task because it supports the building of small modules, for example with the. ***package*** concept.

6.2 Instruction Set Architecture (ISA)

Computer architecture design requires technology from both software and hardware areas. By conventional practice, a computer system is built in a hierarchy of hardware to software layers. The abstract layer where hardware and software interact is called the instruction set architecture (ISA). How ISA is designed and implemented is important because it affects the machine in every aspect, such as compatibility, performance, and cost. Usually, depending on the end user application, the machine architect selects the best possible trade off for its design. By utilizing the same instruction set, one can ensure compatibility problem among different machines. However, this choice might limit the machine in performance because it must follow the same architecture path as other machines using the instruction set.

To illustrate ISA directly, think as the machine's instruction set. A machine's instruction set plays important role in both software and hardware layers. On the hardware side, it sits on top and directly controls how the machine manipulates data. In software side, it is used by all software running on the machine because it is the lowest possible level. Thus, the ISA is obviously the software and hardware interface level. Moreover, the instruction set is abstract, and it serves as a good starting point for one to begin his/her computer architecture design.

6.3 A Computer Architecture Implementation

Perhaps MIPS is the most popular RISC machine used in academia for studying computer architecture. This laboratory will take no exception by borrowing concepts from the the MIPS architecture and implementing example datapath of a CPU in VHDL. This architecture is to be called BIA (Borrowed and Incomplete Architecture).

Similar to the MIPS instruction format, the BIA instruction format is constant but shorter in length 16 bits. To steal more MIPS ideas, BIA instructions can be separated into six fields (see Table 6-1). Notice that the field lengths sum to 16 bits.

Field	Description	Length	Bit Location
OP	Opcode	4 bits	[15:12]
rs	Operand 1	2 bits	[11:10]
rt	Operand 2	2 bits	[9:8]
rd	Distention	2 bits	[7:6]
sh	Shift amount	2 bits	[5:4]

Copyright 1996, EE / CERL, V1.00

funct	Function	4 bits	[3:0]

Table 6-1. BIA Instruction Field

Shamelessly, BIA also contains three types of instructions: R-type, I-type and J-type, where R instructions are for register type operations, I instructions are for data transfer operations and J instructions for unconditional instruction jumps. Since the purpose of BIA is to demonstrate the ability to model a CPU datapath in VHDL, the BIA architectural functionality need not to be complete. Only a few instructions are sufficient for modeling purposes, and they are shown in Table 6-2.

Instructions	Description	Format	OP code	Function
lw	load word	I	14	x
sw	store word	I	15	x
add	add	R	0	1
sub	subtract	R	0	2
or	logical or	R	0	3
sll	logical left shift	R	0	5
srl	logical right shift	R	0	6
andi	and immediate	I	12	x
ori	or immediate	I	13	x

Table 6-2. BIA Instructions

6.4 Implementation Example

The BIA CPU block diagram shown in Figure 6-1 contains four major components: register file, ALU, memory block, and control module. The BIA ALU implementation is simple and straight forward. Most of the ALU functions, (for example, add, circular shift, logic operations, etc.), have already been presented in the previous VHDL exercise so there is no need to re-implementing them. Since the main focus of this laboratory is to learn the design of a modern microprocessor CPU, supporting logic that surrounds the ALU, (such as datapaths and memory element entities), will be the main focus of this lesson. Again, these items are shown in Figure 6-1.

Copyright 1996, EE / CERL, V1.00

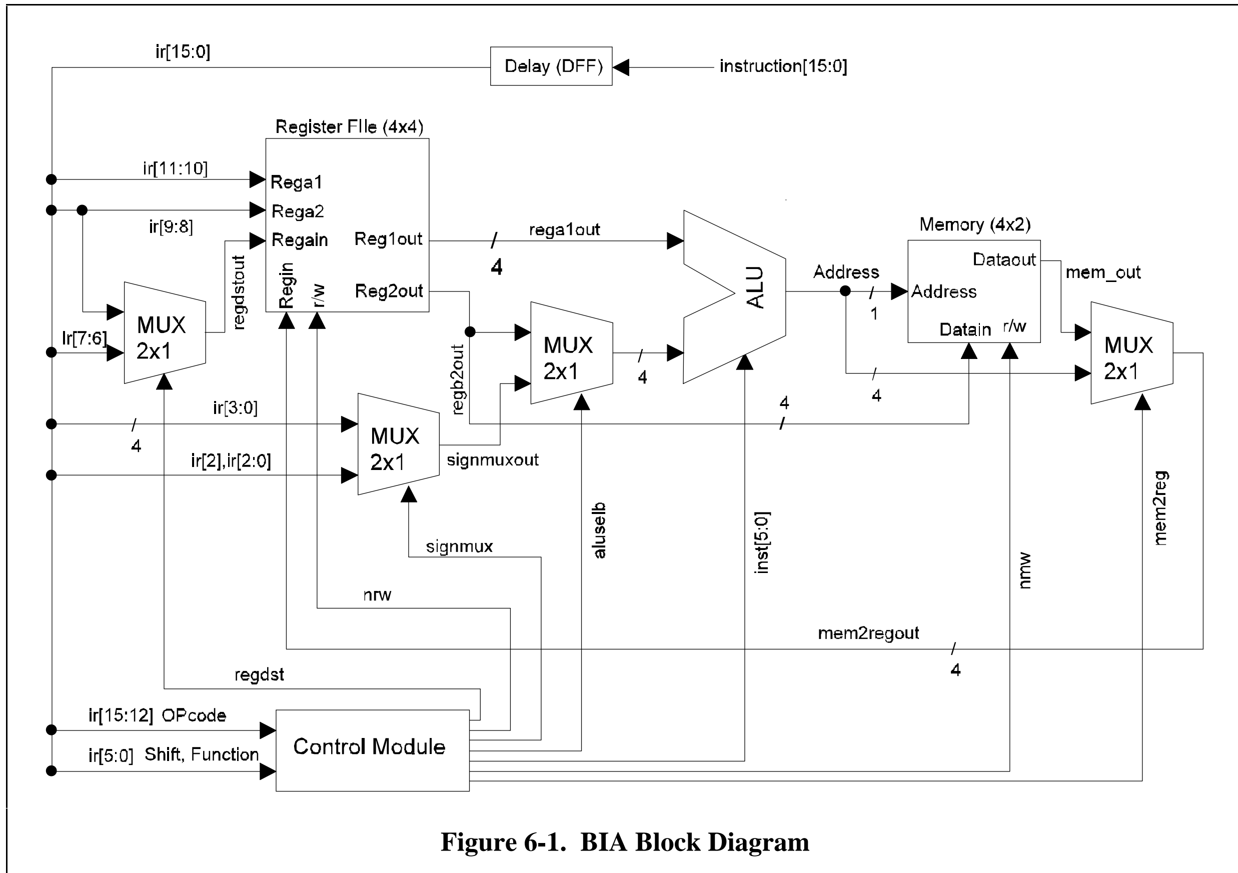


Figure 6-1. BIA Block Diagram

All BIA instructions shown in Table 6-2 take one clock cycle to execute, with the exception of the *lw* (load word) instruction which needs two clock cycles. The clocked elements are the register file and memory in the BIA datapath. The *lw* operation is triggered by setting the instruction signal a*luop* to 14. Next, the load memory effective address is calculated. The calculation is done by adding the register content to the address offset which is stored in the immediate field of *lw* instruction. Next, *lw* reads the memory by supplying it with the effective address. Finally, the content of the memory is written back into a particular register which is specified in the instruction word. The timing diagram of the *lw* instruction along with other control signals is in drawn in Figure 6-2. In the figure, the BIA execution path coarsely resembles to a typical CPU instruction cycle: Decode ($T_{opcodeDelay}$), Fetch ($T_{memoutDelay}$), and Execute ($T_{nregwriteHold}$).

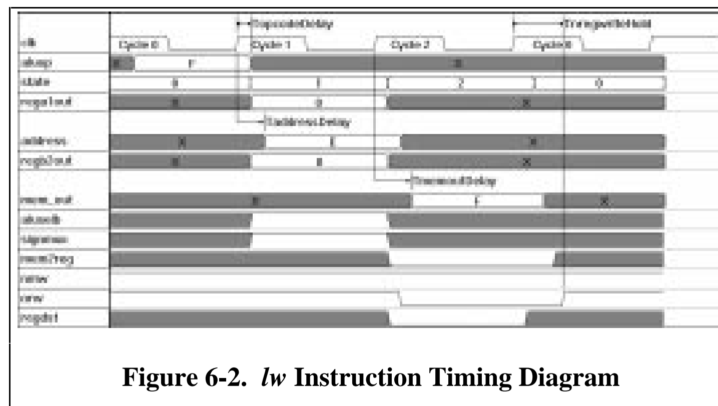


Figure 6-2. *lw* Instruction Timing Diagram

Due to the complexity of the Cypress VHDL tool, only a fraction of an ALU block is implemented in BIA, and it is shown in the VHDL listing. Important ALU functions such as add and circular shift are not

Copyright 1996, EE / CERL, V1.00

included because the current CPLD fitter only supports up to 128 micro-cell devices. The BIA VHDL implementation which is shown in a later section includes everything shown in Figure 6-1 with a basic ALU block that mimic ALU functions.

6.4.1 4x4 Register File and 8x4 Memory Block

A Register file is not much different from a memory block as seen a few lessons earlier. A register file is made by an array of registers, and each register bit is built from a D flip-flop. A Register file is rather similar to a memory block in that they both are data storage devices. The difference between a register file and a memory block is that a register file contains two read ports instead of one. Moreover, a register file is a faster device because it is placed physically near the ALU. Memory, on the other hand, is located farther from the ALU or even sometimes external to the CPU chip. No less important than the ALU block, register file performance is also critical to how a carfully designed CPU performs.

The logic block diagram of the BIA register file is shown in Figure 6-3. Since BIA is a much reduced MIPS CPU, only four 4-bit wide registers are implemented. The overall BIA register file is made up of multiplexers, decoders, and D flip-flops. Two addresses, REG1a and REG2a, are used to select any two of the registers for reading. Data are output on the corresponding REG1 and REG2 ports. When writing to the register file, data to be written is input from REGin, and the writing register address from REGina is decoded to enable the corresponding register. When reading and writing from the same register, the reading port gets the value prior to the writing of the new value. In other words, the value returned to the reading port will be the value written in an earlier clock cycle. The VHDL source code can be found in a later sections.

Both the register file and memory block in BIA are clocked memory elements. To write or read from these devices, control signals such as the read/write signal, address, and write data lines need to be synchronized with a rising clock edge. This synchronization can be seen in the timing diagram Figure 6-2.

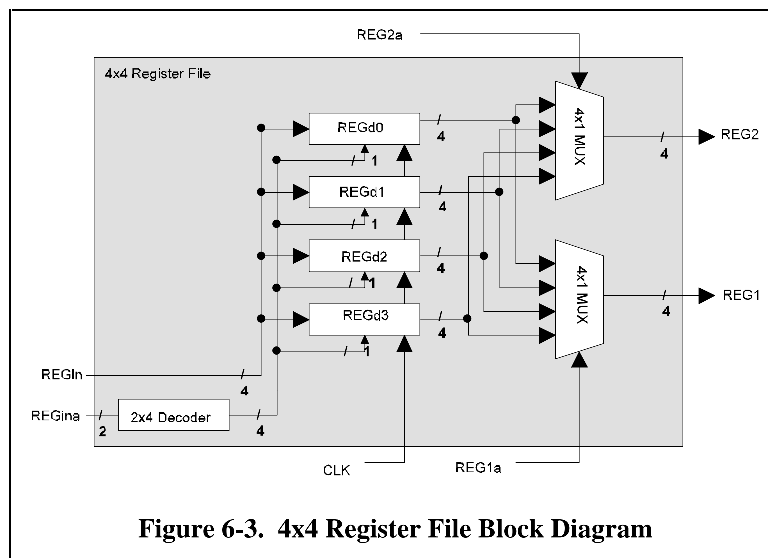


Figure 6-3. 4x4 Register File Block Diagram

6.4.2 Control Module

When given instructions, BIA executes according to the state diagram shown in Figure 6-4 below. Notice that BIA instructions are divided into four types: *lw* (load word), *sw* (store word), *ori* (or immediate), and R (or register operations). Each instruction type has distinct execution flow and consequential control signal values. The VHDL file *ctrlm.vhd* which is shown in 6.4.4.3 contains a VHDL entity, *st-mach* that

Copyright 1996, EE / CERL, V1.00

implements functions dictating the BIA state flow. Entity *ctrl_module* uses the state value produced in *st_mach* to generate control signals that go to every module of the BIA CPU.

6.4.3 Execution Example

An example session of running BIA is shown in Figure 6-5. To see how the BIA datapath handles instruction and data traffic, one also needs to be familiar with the BIA block diagram of Figure 6-1. The session was captured by executing five BIA instructions. A BIA instruction is first fed into the input port *inst*. Next, the instructions is latched into registers *ir*. Notice that *ir* is associated with the state machine that holds instruction (opcode+operand) for some number of clock cycles. For example, the R-type, *sw*, and *ori* instructions take two cycles to complete and the *lw* instruction takes three. This instruction latch state machine is implemented in entity *instruction_latch* and it can be found in the listing for *bia.vhd*.

After an instruction has been latched into the instruction register *ir*, each instruction field, such as opcode and operands, is sent to the corresponding function block. For example, in the typical R-type instruction, after the opcode is identified to be an R-type instruction, operands are sent to the register file (fast two-port read memory) for retrieving register values and getting them to the next processing stage, namely the ALU. If the instruction in execution is simple R-type instruction, the result from the ALU stores directly back into the register file, and it takes two cycles (cycle 07 and 08 in Figure 6-5). If the instruction is a memory transfer instruction, depending on load word or store word, it can take either two or three clock cycles. Remember that both the register file and memory are clocked elements, therefore, to store a register word into memory, the first clock cycle is needed to release the data from the register and the second clock cycle to store the data into the memory. Similarly in a load word operation, the first clock cycle is to read data from the register (address), the second clock cycle to retrieve data, and the third clock cycle to store back to the register. Again, all of these processes are executed when running the BIA code in Table 6-3 below.

Copyright 1996, EE / CERL, V1.00

Instruction	**Comment**	**Opcode** [15:12]	**rs** [11:10]	**rt** [9:8]	**rd** [7:6]	**sh** [5:4]	**function** [3:0]
ori r3, r0, #06	; $3 <- $0 OR 6	1101	00	00	11	00	0110
ori r2, r0, #04	; $2 <- $0 OR 4	1101	00	00	10	00	0001
or r1, r2, r3	; $1 <- $2 OR $3	0000	10	11	01	00	0010
sw r1, #4(r0)	; Memory[$0+4] = $1	1111	00	00	01	00	0001
lw r3, #4(r0)	; $3 = Memory[$0+4]	1110	00	00	11	00	0001
nop	; no op	0001	00	00	00	00	0000

Table 6-3. BIA Code

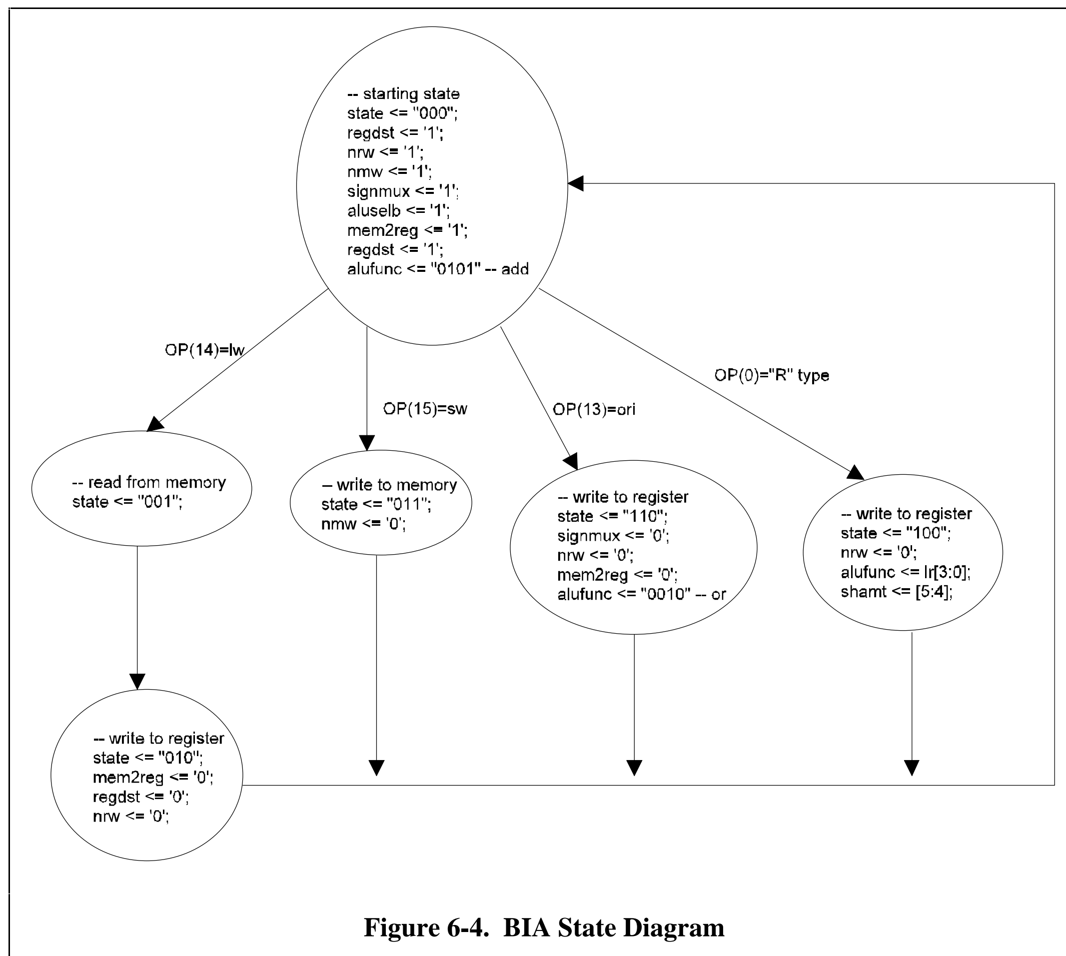


Figure 6-4. BIA State Diagram

The simulation run of the operations in Figure 6-5 contains two fields, *inst* and *ir*. *Inst* is the input instruction corresponding to the code above. *Ir* is the instruction latched corresponding to the *inst* value. For example, at clock cycle 2, instruction 0xd0c6 is input from *inst* and then latched by the *ir* register for

Copyright 1996, EE / CERL, V1.00

two cycles. The reason that instructions are latched is because control signals at every stage or clock cycle used by various modules are generated from the instruction field. To execute each BIA instruction correctly, the instruction needs to be frozen during that instruction cycle. For example, instructions such as *ori*, *or*, and *lw* need to be constant for two clock cycles, and the instruction *lw* needs three clock cycles. Again the timing diagram in Figure 6-5 shows *ir* was assigned value 0xd0c6 for two cycles (cycle 3 and 4). Notice that this operation is different from the pipeline concept to be learned later.

The signals *regf4x4_regd1*, *regf4x4_regd2*, and *regf4x4_regd3* are a register file that contains registers 1, 2 and 3. Notice that register 0 always contains a zero value, and therefore it was not shown in the figure. Memory contents are stored in the signals *mem2x4_regd0* and *mem2x4_regd1*. A memory word is 4 bits wide, However, there are only two words implemented in BIA because of Cypress the VHDL compiler limitation (BIA with more memory words cannot be fitted and simulated properly).

Although a more complete ALU can be used, the ALU implemenetd here contains only a few simple operations such as the *or*, *ori*, *and*, and *andi* instructions. Due to the limited amount of logic that one can fit into a Cypress CPLD family part, only these representative operations are used and shown for purposes of this example. Similar is of the case in the memory word depth. The BIA module originally contain a 4 bit wide and 8 word deep (4x8) module. But, the fitter can’t get that BIA logic into a 37x CPLD. Therefore, the memory block size was reduced to 4x2 to make the logic fit.

Another comment about simulating these six BIA instructions in Table 6-3 is that the Cypress VHDL simulator, NOVA, takes about three minutes to do the simulation on a Petium-90 processor with 16 Megabytes of RAM in the Windows 95 platform. NOVA is not a graphically incremental simulator. The user does not see timing waveform change incrementally. The final wave shows only at the end of the simulation. In a typical simulation session, one would first start the simulation and let it run. While the machine is simulating, output timing waveforms are frozen for approximately three minutes, and then at last, the waveforms are updated all at once when the simulation completes.

Copyright 1996, EE / CERL, V1.00

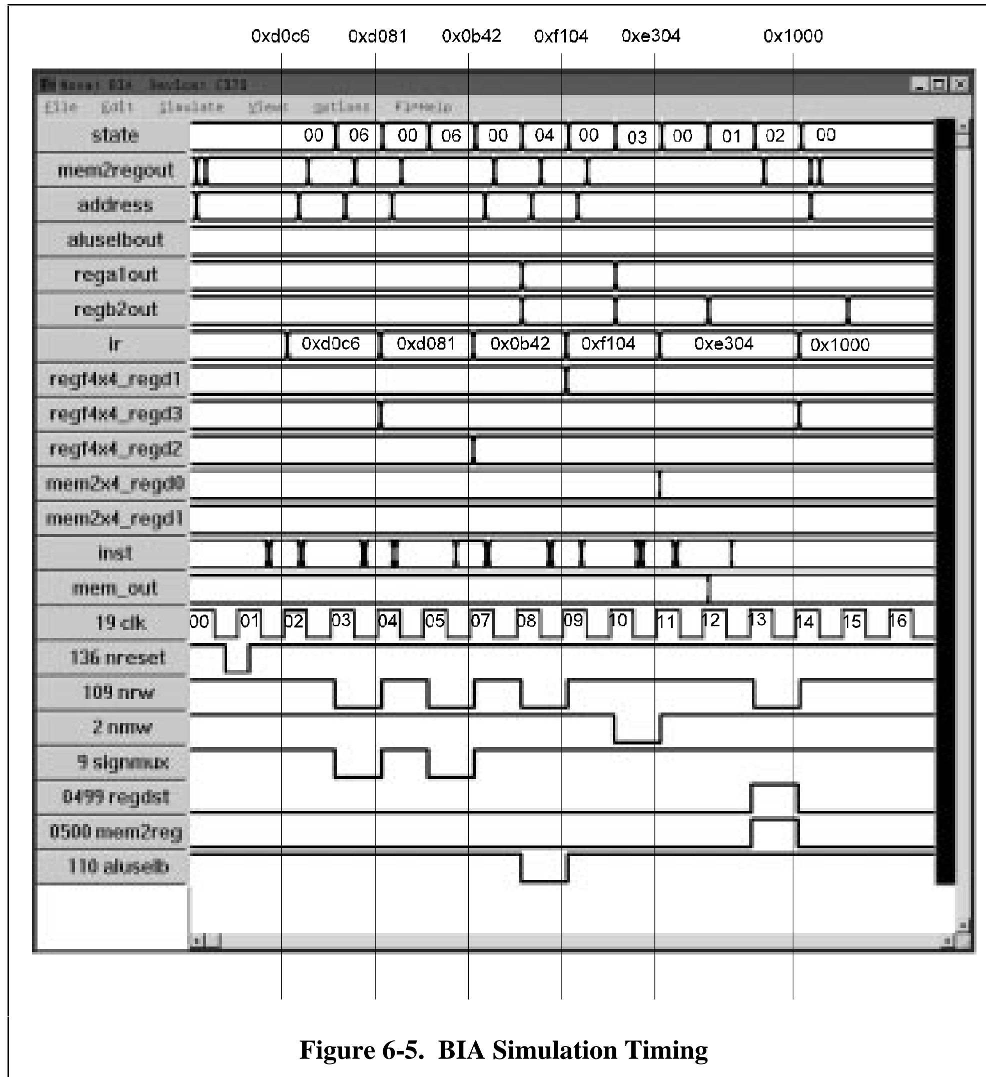


Figure 6-5. BIA Simulation Timing

6.4.4 VHDL Listing

Sections below list VHDL code for major components and modules that make BIA architecture.

6.4.4.1 regf4x4.vhd

Below is the VHDL listing for the 4x4 register file block.

```
package regf_type is

    constant bw: integer := 3;
    constant adw: integer := 1;

component regf4x4

    port(
        clk   :           in     bit;
      nreset: in bit;
        nrw: in bit;
```


Copyright 1996, EE / CERL, V1.00

```
        regin:          in      bit_vector(bw downto 0);
        reg1, reg2:     out     bit_vector(bw downto 0);
        reg1a, reg2a, regina:
                        in      bit_vector(adw downto 0));
end component;

end regf_type;


use work.regf_type.all;

entity regf4x4 is

  port(
    clk    : in     bit;
   nreset: in bit;
  nrw: in bit;
    regin: in     bit_vector(bw downto 0);
    reg1, reg2:     out     bit_vector(bw downto 0);
    reg1a, reg2a, regina:
                        in      bit_vector(adw downto 0));

end regf4x4;



architecture regf4x4_arch of regf4x4 is
signal regd0, regd1, regd2, regd3: bit_vector (bw downto 0);
begin

process (clk)
variable i: integer;
begin

    if clk'event and clk='1' then
        if nreset='0' then
            regd0 <= "0000";
            regd1 <= "0000";
            regd2 <= "0000";
            regd3 <= "0000";
        else

-- writing to register
            if nrw='0' then

                if regina="00" then
                    regd0<=regin;
                elsif regina="01" then
                    regd1<=regin;
                elsif regina="10" then
                    regd2<=regin;
                else
                    regd3<=regin;
                end if;
            end if;

-- reading from register

-- register 1 read
            if reg1a="00" then
--                  reg1<=regd0;
                reg1<="0000";
            elsif reg1a="01" then
                reg1<=regd1;
            elsif reg1a="10" then
                reg1<=regd2;
            else
                reg1<=regd3;
            end if;
```


Copyright 1996, EE / CERL, V1.00

```
-- register 2 read
                if reg2a="00" then
--                      reg2<=regd0;
                        reg2<="0000";
                elsif reg2a="01" then
                        reg2<=regd1;
                elsif reg2a="10" then
                        reg2<=regd2;
                else
                        reg2<=regd3;
                end if;
            end if;

        end if;

end process;

end;
```

Listing 6-1. Regf4x4.vhd VHDL Code

6.4.4.2 mem4x2.vhd

Below is the construction of both 2x4 and 8x4 memory blocks in VHDL. Although the 2x4 memory block is the one used in *BIA.vhd*, the 8x4 memory block illustrates a more generic method of constructing memory of any word depth.

```
package memory_type is

        constant bw: integer := 3;
        constant adw: integer := 1;
        constant total_word: integer := 3;

        type one_mem_data is array (bw downto 0)of bit;
        type mem_data is array (total_word downto 0) of one_mem_data;


component mem2x4
        port(       clk: in bit;
                    mdatain:   in bit_vector(3 downto 0);
                    mdataout: out bit_vector(3 downto 0);
                    address:   in bit;
                    nw:        in bit);
end component;



component mem8x4
        port(       clk: in bit;
                    mdatain:   in one_mem_data;
                    mdataout: out one_mem_data;
                    address:   in bit_vector (adw downto 0);
                    nw:        in bit);

end component;


end memory_type;

use work.int_math.all;
use work.memory_type.all;
```


Copyright 1996, EE / CERL, V1.00

```
entity mem8x4 is

        port(       clk: in bit;
                    mdatain:   in one_mem_data;
                    mdataout: out one_mem_data;
                    address:   in bit_vector (adw downto 0);
                    nw:          in bit);

end mem8x4;

architecture mem8x4_arch of mem8x4 is
signal mword: mem_data;
signal madd: bit_vector (adw downto 0);
begin

-- clocked memory!

process (clk)
variable j: integer;
begin
        if clk'event and clk='1' then

-- write process
                if nw='0' then
                        for j in 0 to total_word loop
                                if address=i2bv(j,2) then
                                        mword(j) <= mdatain;
                                end if;
                        end loop;
                else
                        for j in 0 to total_word loop
                                if address=i2bv(j,2) then
                                        mdataout <= mword(j);
                                end if;
                        end loop;
                end if;

        end if;

end process;

end mem8x4_arch;
-----------------------------------------------------------------------------
use work.memory_type.all;
entity mem2x4 is
        port(       clk: in bit;
                    mdatain:   in bit_vector(3 downto 0);
                    mdataout: out bit_vector(3 downto 0);
                    address:   in bit;
                    nw:          in bit);
end mem2x4;

architecture mem2x4_arch of mem2x4 is


signal word0, word1: bit_vector(3 downto 0);

begin

process (clk)
begin
        if clk'event and clk='1' then
                if address='0' then
                        if nw='0' then
                                word0 <= mdatain;
                        else
                                mdataout <= word0;
```


Copyright 1996, EE / CERL, V1.00

```
                    end if;
               else
                    if nw='0' then
                         word1 <= mdatain;
                    else
                         mdataout <= word1;
                    end if;
               end if;
          end if;
end process;

end mem2x4_arch;
```

Listing 6-2. Mem4x2.vhd VHDL Code

6.4.4.3 ctrlm.vhd

The VHDL program *ctrl_pkg* below the VHDL code section *main* contains BIA state control logic. It dictates BIA state given independent input such as instructions. At each BIA state, *ctrlm.vhd* also generates control signals to control other logic blocks such as ALU, memory, and the register block. These processes comprise in the component *st_mach*. Notice that st_mach execute according to the flow diagram in Figure 6-4.

One more important piece of code is the *alu_module* block. *alu_module* is very similar to the one described in the previous lesson except it contains operations custom to BIA architecture. Also, operations such as addition and circular shift are not included (commented out) due the CPLD capacity.

The last remaining components are simple multiplexers of various bit widths.

```
package ctrl_pkg is

component alu_module
     port (
          ain, bin: in bit_vector (3 downto 0);
          sel_bit: in bit_vector (3 downto 0);
          shamt: in bit_vector (1 downto 0);
          cout: out bit_vector (3 downto 0);
          flags: out bit_vector (2 downto 0));
end component;

component mux1_2x1
     port (
          ain, bin: in bit;
          cout: out bit;
          sel: in bit);
end component;

component mux4_2x1
     port (
          ain, bin: in bit_vector (3 downto 0);
          cout: out bit_vector (3 downto 0);
          sel: in bit);
end component;

component mux2_2x1
     port (
          ain, bin: in bit_vector (1 downto 0);
          cout: out bit_vector (1 downto 0);
          sel: in bit);
end component;

component st_mach
```


Copyright 1996, EE / CERL, V1.00

```
        port( state: inout bit_vector (2 downto 0);
                aluop: in bit_vector (3 downto 0);
                clk: in bit;
                nreset: in bit;
                ir3_0: in bit_vector (3 downto 0);
                aluselb: out bit;
                alufunc: out bit_vector (3 downto 0);
                mem2reg: out bit;
                regdst: out bit;
                nregwr: out bit;
                nmemwr: out bit;
                signmux: out bit);

end component;

end ctrl_pkg;
---------------------------------------------------------------

use work.logic_parts_pkg.all;
use work.iarith_parts_pkg.all;

entity alu_module is
        port (
                ain, bin: in bit_vector (3 downto 0);
                sel_bit: in bit_vector (3 downto 0);
                shamt: in bit_vector (1 downto 0);
                cout: out bit_vector (3 downto 0);
                flags: out bit_vector (2 downto 0));
end alu_module;

architecture alu_module_arch of alu_module is
signal bAND_out, bOR_out: bit_vector (3 downto 0);
signal lcshift_out, rcshift_out: bit_vector (3 downto 0);
signal ripp_adder4_out: bit_vector (3 downto 0);
signal apass_out, bpass_out: bit_vector (3 downto 0);
signal overflow, b_zero, b_one: bit;
begin

op1: bAND port map(ain, bin, bAND_out);
op2: bOR port map(ain, bin, bOR_out);
b_zero <= '0';
b_one <= '1';

process (shamt)
begin
        if shamt="00" then
                lcshift_out <= "0000";
                rcshift_out <= "0000";
        end if;
end process;


-- use reg2 pass for memory transfer
ripp_adder4_out <= bin;

-- op3: cshift port map(ain, shamt, b_zero, lcshift_out);
-- op4: cshift port map(ain, shamt, b_one, rcshift_out);
-- op5: ripp_adder4 port map(ain, bin, ripp_adder4_out, overflow);

   op6: apass_out <= ain;
   op7: bpass_out <= bin;

process(sel_bit)
begin
        if sel_bit="0001" then
                cout <= bAND_out;
        elsif sel_bit="0010" then
                cout <= bOR_out;
        elsif sel_bit="0011" then
```


Copyright 1996, EE / CERL, V1.00

```
            cout <= lcshift_out;
        elsif sel_bit="0100" then
            cout <= rcshift_out;
        elsif sel_bit="0101" then
            cout <= ripp_adder4_out;
        elsif sel_bit="0110" then
            cout <= apass_out;
        elsif sel_bit="0110" then
            cout <= bpass_out;
        else
            cout <= "1111";
        end if;
end process;

-- flags(0) <= overflow;
flags(0) <= '0';
flags(1) <= '0';
flags(2) <= '0';

end alu_module_arch;

----------------------------------------------------------------
entity mux1_2x1 is
        port(
            ain, bin: in bit;
            cout: out bit;
            sel: in bit);
end mux1_2x1;

architecture mux1_2x1_arch of mux1_2x1 is
begin

process (sel)
begin
        if sel='0' then
            cout <= ain;
        else
            cout <= bin;
        end if;
end process;

end mux1_2x1_arch;



----------------------------------------------------------------
entity mux4_2x1 is
        port(
            ain, bin: in bit_vector (3 downto 0);
            cout: out bit_vector (3 downto 0);
            sel: in bit);
end mux4_2x1;

architecture mux4_2x1_arch of mux4_2x1 is
begin

process (sel)
begin
        if sel='0' then
            cout <= ain;
        else
            cout <= bin;
        end if;
end process;

end mux4_2x1_arch;
```


Copyright 1996, EE / CERL, V1.00

```
entity mux2_2x1 is
        port(
                ain, bin: in bit_vector (1 downto 0);
                cout: out bit_vector (1 downto 0);
                sel: in bit);
end mux2_2x1;

architecture mux2_2x1_arch of mux2_2x1 is
begin
process (sel)
begin
        if sel='0' then
                cout <= ain;
        else
                cout <= bin;
        end if;
end process;
end mux2_2x1_arch;



----------------------------------------------------------------
-- state machine
----------------------------------------------------------------

entity st_mach is

        port( state: inout bit_vector (2 downto 0);
                aluop: in bit_vector (3 downto 0);
                clk: in bit;
                nreset: in bit;
                ir3_0: in bit_vector (3 downto 0);
                aluselb: out bit;
                alufunc: out bit_vector (3 downto 0);
                mem2reg: out bit;
                regdst: out bit;
                nregwr: out bit;
                nmemwr: out bit;
                signmux: out bit);

end st_mach;

architecture st_mach_arch of st_mach is
begin



process(clk)
begin
        if clk'event and clk='1' then
                if nreset='0' then
                        state <= "000";
                        alufunc <= "0101";
                        regdst <= '1';
                        nregwr <= '1';
                        nmemwr <= '1';
                        signmux <= '1';
                        aluselb <= '1';
                        mem2reg <= '1';
                        regdst <= '1';
                else
-- lw instruction
                        if state="000" and aluop="1110" then
                                state <= "001";
                        elsif state="001" and aluop="1110" then
                                state <= "010";
                                mem2reg <='0';
                                regdst <='0';
                                nregwr <= '0';
```


Copyright 1996, EE / CERL, V1.00

```
-- sw instruction
            elsif state="000" and aluop="1111" then
                state <= "011";
                nmemwr <= '0';

-- ORI instruct
            elsif state="000" and aluop="1101" then
                state <= "110";
                alufunc <= "0010";
                nregwr <= '0';
                signmux <= '0';

-- R-type instruction
            elsif state="000" and aluop="0000" then
                state <= "100";
                alufunc <= ir3_0 (3 downto 0);
                aluselb <= '0';
                nregwr <= '0';

-- back to starting state
            else
                state <= "000";
                alufunc <= "0101";
                regdst <= '1';
                nregwr <= '1';
                nmemwr <= '1';
                signmux <= '1';
                aluselb <= '1';
                mem2reg <= '1';
                regdst <= '1';

            end if;
        end if;
    end if;

end process;

end st_mach_arch;

----------------------------------------------------------------
-- test program
----------------------------------------------------------------
entity cmtest is
    port (
        whatop: inout bit_vector (3 downto 0);
        state: inout bit_vector (2 downto 0);
        ir3_0: in bit_vector (3 downto 0);
        aluop: in bit_vector (3 downto 0);
        alufunc: inout bit_vector (3 downto 0);
        aluselb: out bit;
        mem2reg: out bit;
        regdst: out bit;
        nregwr: out bit;
        signmux: out bit;
        nmemwr: out bit;
        clk: in bit;
        dummy: out bit;
        nreset: in bit );
end cmtest;

architecture cmtest_arch of cmtest is
begin

    state1: st_mach port map (state, aluop, clk, nreset, ir3_0, aluselb, alufunc, mem2reg,
                                    regdst,nregwr, nmemwr,
                                    signmux);
end cmtest_arch;
```


Copyright 1996, EE / CERL, V1.00

Listing 6-3. Ctrlm.h VHDL Code

6.4.4.4 BIA.vhd

The VHDL code below describes the top-most BIA structure. *BIA.vhd* connects all major component blocks with extra glue logic to make these blocks compatible. For example, an instruction is fed through the instruction register *ir*. *ir* input is used by a few components such as *alu_module* and *st_mach* listed in previous section. However, *ir* is one clock cycle behind the actual input *inst* shown in Figure 6-5 due to latching. When latching *inst*, *ir* is required to hold its value for either two or three clock cycles depending on the instruction. Therefore, the state machine *ir_process* inside of *bia.vhd* is used for this purpose. Moreover, *ir_process* also delays the reset signal because of the delayed or latched instruction input into *ir*.

See problems section for *BIA.vhd* VHDL code listing.

6.5 Problems

1) The program BIA.vhd is not given in section 6.4.4.4. However, the entity and the architecture header statements are given. Please use all the information from this lesson to complete the program *BIA.vhd*. The *BIA.vhd* JEDEC file is given to demonstrate the simulation.

```
use work.regf_type.all;
use work.ctrl_pkg.all;
use work.memory_type.all;

entity bia is
        port (
                nrw, nmw: inout bit;
                aluselb, signmux: inout bit;
                nreset_delay: inout bit;
                clk, nreset: in bit;
                inst: in bit_vector(15 downto 0);
                regb2out, rega1out, aluselbout: inout bit_vector(3 downto 0);
                address: inout bit_vector (3 downto 0);
                mem2regout: inout bit_vector (3 downto 0);
                state: inout bit_vector (2 downto 0));
end bia;

architecture bia_arch of bia is


signal pass: bit_vector (1 downto 0);
signal ir: bit_vector (15 downto 0);
signal regdstout: bit_vector (1 downto 0);
signal mem_out: bit_vector ( 3 downto 0);
signal flags: bit_vector (2 downto 0);
signal temp: bit_vector (3 downto 0);
signal regdst, mem2reg: bit;
signal alufunc: bit_vector (3 downto 0);

begin

        <your vhdl code goes here>

end bia_arch;
```

Simulate your compiled program in the NOVA simulator and produce a timing waveform similar to the one in Figure 6-5.

Copyright 1996, EE / CERL, V1.00

7. APPENDICES

The sections below describe miscellaneous procedures for beginners to use the Cypress VHDL WARP2 version 4.0 development system. The WARP2 program supports both Window 3.1X and Window 95.

7.1 Install Warp on PC

Cypress VHDL WARP2 version 4.0 requires the use of a CD ROM drive to install. Since the program itself is big, the WARP2 program requires around 70 Mbytes of hard disk space on the PC. The following steps are needed to setup the WARP program for a PC.

1) Go to the CD ROM drive and under the directory \pc, run setup.

2) Answer all questions the setup procedure asks, (for example, name, serial number, etc.).

3) Install everything by clicking the continue, next, and yes buttons as appropriate, (e.g., installing WIN32S system [for Window 3.1X], WARP documentation, etc.).

4) After you are done installing WARP documents and programs, install the acrobat reader so you can retrieve the on-line WARP user guide and reference manual. To install the acrobat reader, go to the directory \pc\acroread\disk1 and run setup.

5) At last, you need to copy files uguide.pdf (WARP user guide) and refmanl.pdf (WARP reference manual) from \doc directory in the CD to a directory on your computer so the Acrobat reader can use them.

6) Now, the WARP2 system should be ready to go.

7.2 Introduction to WARP4 VHDL Development System

1) Cypress VHDL WARP2v4.0 comes with an integrated project manager, ***Galaxy***, in which the VHDL code developer can edit, compile, debug, simulate, and test the VHDL program under the same environment. This section will give you a brief introduction on how to use the VHDL project manager.

Figure 7-1. Project Window

2) Go into the Windows environment WARP program group, and select the ***Galaxy*** program. This should enable the ***Galaxy*** program to pop up an integrated project development window. Click on the

OK button if this were the first time WARP got start. After that, a project window will appear on the screen. See Figure 7-1.

3) Click on the top ***Galaxy*** menu bar item **Project,** then select **New** command, and another window will pop on the screen to prompt for the project name. Type *prj_name* for the project name. After that, another interactive project development window named *prj_name* will pop on the screen.

4) Now click on the **New** button inside of the **Edit** box on right hand side of the *prj_name* project window to start the editor.

5) Enter the VHDL code such as the one from lesson one, Listing 1-1.

6) After typing in the VHDL program, click on the top **File** button and select the **Save As** command to save the file into *fil_name.vhd*. Quit the editor after the file has been saved. This should return the program control back to the project manager window *prj_name*.

7) To include the VHDL program in the project manager window *prj_name* for compiling, click on the **Files** command on the top menu bar and select the **Add** button to enable the VHDL program add window. In the add window, first select the VHDL file by clicking on it (e.g. *fil_name.vhd*) in the left box, and then click on the → button to move the VHDL file to the right box. Select the **OK** button to go back to the project window afterwards.

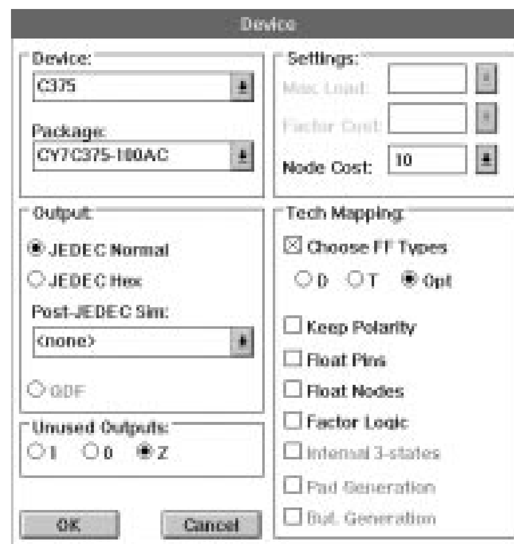


Figure 7-2. Device Window

8) To open a device window, click on the **Device** button inside of the Synthesis options box which is located at the bottom right corner of the project window. In the device window, see Figure 7-2, select C375 on the top left corner under the device category. Click on the **OK** button to go back to the project window again.

9) In the project window, click on the **Set top** button to tell the project manager that the VHDL program *fil_name* just entered is the main program for simulation.

10) Now, the program *fil_name* is ready to compile and simulate. To compile the VHDL program, first select the file *fil_name.vhd*, then click the **Smart** button inside of the compile box in project window. Another window should pop up and display the compilation status.

11) If there were VHDL syntax errors, one can easily go back to the VHDL program by selecting the **Error** buttons that got the magnifying glass icon on the top of the compilation window menu bar.

12) After the compilation process is completed, quit the compilation window by clicking on the **Close** button on the top of compilation window menu bar.

13) To start the ***NOVA*** simulator, click on the top **Tools** button and select the **NOVA** command to get to the ***NOVA*** window. In the ***NOVA*** window, click on the **Files** button and select **Open** to specify the simulation object file with the file extension **.jed*. In this example, *fil_name.jed* is used.

14) In the ***NOVA*** simulator, one can examine the circuit's functionality by studying the input and output relationship. To simulate the circuit, one needs to set the input waveform(s) first, then, click on the top menu bar **Simulate** item and select the **execute** command to simulate. Hopefully, in a short period, ***NOVA*** will display the simulated waveform in red.