Memory Architecture Design of a 32-bit Embedded Processor using VHDL

A
Dissertation submitted to
University of Delhi
In partial
fulfillment of the requirements for
the award of the
Master of Engineering
in Electronics & Communication Engineering,

BY

MANOJ K N
Roll No. 16/E&C/2001
University Roll No 4092

UNDER THE GUIDANCE OF

Prof. ASOK BHATTACHARYYA
AND
MRS. S. INDU

DEPARTMENT OF
ELECTRONICS & COMMUNICATION ENGINEERING
DELHI COLLEGE OF ENGINEERING
DELHI UNIVERSITY

CERTIFICATE

This is to certify that the dissertation entitled ***Memory Architecture Design of a 32-bit Embedded Processor using VHDL*** being submitted by Manoj K N for the degree of M E in Electronics & Communication is a record of his own work carried out under our supervision and guidance. The matter contained in this report has not been submitted for the award of any other degree or diploma.

Mrs. S INDU
Lecturer
Dept of ECE
Delhi College of Engineering

Dr. ASOK BHATTACHARYYA
Professor & H O D
Dept of ECE
Delhi College of Engineering

Department of
Electronics & Communication Engineering
Delhi College of Engineering
Bawana Road, New Delhi – 110 042

ACKNOWLEDGMENT

With great pleasure, I express my sincere thanks to Dr. Asok Bhattacharyya, Professor & HOD, Department of Electronics & Communication Engineering, Delhi College of Engineering for his keen interest and encouragement in completing this project.

I am very much thankful and grateful to Mrs. S. Indu, Lecturer, Department of Electronics & Communication Engineering, for her valuable guidance during this project.

I am also thankful to other staff members of Delhi College of Engineering and all my fellow students, who helped me directly or indirectly in the completion of this project.

Manoj K N

ABSTRACT

Under this project entitled "***Memory Architecture Design of a 32-bit Embedded Processor using VHDL***", the following are modeled using VHDL.

1. A basic 32-bit embedded processor core using **VHDL.** It uses Register Transfer Level (**RTL**) description to model the processor. The processor modeled is based on MIPS Architecture. The design uses a 32 bit register file which is an array of thirty two 32-bit registers uses to store data, a 32bit register to store the instruction, 32-bit register to store operational data at various stage, an **ALU** for arithmetic and logic manipulation and an ALU control which decodes the function to be performed by **ALU**. Control is implemented using a Finite State Machine model. Multiplexers are used to select from different input signals in each functional blocks.

2. Behavioral model of a 32 bit Random Access Memory with burst transfer protocol implemented

3. Behavioral model of a Direct Mapped Write Through Cache Memory block which utilizes the burst transfer protocol while loading from the Random Access Memory.

The simulation is performed using **ACTIVE HDL Version 6.3** of **ALDEC Corporation**.

CONTENTS

1. INTRODUCTION

We are living in a second age industrial revolution, when the availability and processing of information are causing untold changes in our lives. In 1971 Intel produced the first microprocessor, the 4004, which handled data as 4 bit numbers and contained 2250 transistors. It followed this soon with 8008, and within a few years number of companies were making their own microprocessor offerings. By the end of 1970s two trends were emerging for these remarkable devices. One was to scale down in size, if not computing power, the general purpose computer, this led quickly to the first desktop machines. The other, much more revolutionary was to place the microprocessor in products which apparently had nothing to do with computing. They began to find their way into photocopiers, washing machines, and a host of other products. While the first trend led to an inexorable demand for faster and bigger processors, the second placed lower demand on computational power and speed. It wanted physically small and cheap devices, with as much functionality of the system as possible squeezed on to one integrated circuit. Such microprocessors became known as micro controllers and the systems they controlled, embedded system.

MIPS architecture was chosen by Hennessey & Patterson [1] as a vehicle for teaching principle of Computer Architecture. It is a 32-bit Reduced Instruction Set Computer (**RISC**). Here we use the **MIPS** processor for developing a high level abstraction of in **VHDL** first. Then we develop a memory hierarchy to support this processor model to evaluate the importance of Cache Memory in improving the overall memory performance.

Developed more than 20 years ago at Stanford University, the MIPS architecture is a simple, streamlined, highly scalable RISC architecture. Its fundamental characteristics - such as the large number of registers, the number and the character of the instructions, and the visible pipeline delay slots - enable the MIPS architecture to deliver the highest performance per square millimeter and lowest energy consumption in today's SOC designs.

1.1 MIPS32

The MIPS32 Architecture sets a new performance standard for 32-bit embedded processors. The MIPS architecture is the leading embedded architecture because of its robust instruction set, scalability from 32-bits to 64-bits, broad-spectrum of software development tools. The MIPS32 architecture is a superset of the previous MIPS I and MIPS II Instruction Set Architectures (ISA) and incorporates powerful new instructions specifically for embedded applications, as well as proven memory management and privileged mode control mechanisms. By incorporating powerful new features, standardizing privileged mode instructions, and supporting past ISAs, the MIPS32 architecture provides a solid high-performance foundation for all future 32-bit MIPS processor-based development.

The MIPS32 architecture is based on a fixed-length, regularly encoded instruction set and uses a load/store data model. The architecture is streamlined to support optimized execution of high-level languages. Arithmetic and logic operations use a three-operand format, allowing compilers to optimize complex expressions formulation. Availability of 32 general-purpose registers enables compilers to further optimize code generation by keeping frequently accessed data in registers.

Flexibility of its high-performance caches and memory management schemes continues to be a strength of the MIPS architecture. The MIPS32 architecture extends this advantage with well-defined cache control options. The size of the instruction and data caches can range from 256 bytes to 4Mbytes. The data cache can employ either a write-back or write-through policy. A no-cache option can also be specified.

1.2 Features

- Fully MIPS I and MIPS II ISA compatible
- Enhanced with conditional move and data-prefetch instructions
- Standardized DSP operations: multiply (MUL), multiply and add (MADD), and count leading 0/1s (CLZ/O)
- Privileged cache load/control operations
- Robust load/store RISC instruction set with 3-operand instructions in most formats (3 register, 2 registers + immediate), branch/jump options, and delayed jump instructions.
- 32 general purpose 32-bit registers (GPRs)
- Optional floating-point support:
- 32 single precision 32-bit or 16 double precision 64-bit floating point registers (FPRs)
- Floating-point condition code register
- Optional Memory Management Unit with:
    - TLB or BAT address translation mechanisms
    - Programmable page size
    - Optional caches:
    - Instruction and or data cache options
    - Write-back or write-through data-cache options
    - Virtual or physical addressing
    - Enhanced JTAG (EJTAG) support for non-intrusive debug support

MIPS32 compatible processors are intended for high performance, low-power, system-on-a-chip (SOC) embedded applications.

1.3 Applications

Security Devices

- Smart cards
- Smart card readers
- Point of Deployment (POD) devices

Digital Consumer Devices

- Digital Cameras
- Set-top Boxes
- Game Platforms
- DVD Players

Office Automation

- Printers
- Copiers
- Scanners
- Multifunction Peripherals

Other

- Industrial Controllers
- Mass Storage Systems
- Automotive Systems
- Navigation (GPS)
- PC Peripherals
- Graphics Systems
- Dedicated Terminals (POS, ATM, e-cash)

2. INTRODUCTION TO VHDL

As integrated circuit technology has improved to allow more and more components on a chip, digital systems have continued to grow in complexity. As digital systems have become more complex, detailed design of the systems at the gate and flip-flop level has become very tedious and time consuming. For this reason, use of hardware description languages in the digital design process continues to grow in importance. A hardware description language allows a digital process continues to grow in importance. A hardware description language allows a digital system to be designed and debugged at a higher level before conversion to the gate and flip-flop level. Use of synthesis computer-aided design tools to do this conversion is becoming more widespread. This is analogous to writing software programs in a high-level language such as C and then using a compiler to convert the programs to machine language. The two most popular hardware description languages are VHDL and Verilog.

VHDL is a hardware description language used to describe the behavior and structure of digital systems. The acronym VHDL stands for VHSIC Hardware Description Language, and VHSIC in turn stands for very High Speed Integrated Circuit. However, VHDL is a general purpose hardware description language that can be used to describe and simulate the operation of a wide variety of digital systems, ranging in complexity from a few gates to an interconnection of many complex integrated circuits. VHDL was originally developed for the military to allow a uniform method for specifying digital systems. The VHDL language has since become and IEEE standard, and it is widely used in industry.

VHDL can describe a digital system at several different levels-behavioral, data flow, and structural. For example, a binary adder could be described at the behavioral level in terms of its function of adding two binary numbers, without giving any implementation details. The same adder could be described at the data flow level by giving the logic equations for the adder. Finally, the adder could be described at the structural level by specifying the interconnections of the gates that comprise the adder.

VHDL leads naturally to a top-down design methodology, in which the system is first specified at a high level and tested using a simulator. After the system is debugged at this level, the design can gradually be refined, eventually leading to a structural description closely related to the actual hardware implementation. VHDL was designed to be technology independent. If a design is described in VHDL and implemented in today's technology, the same VHDL description could be used as a starting point for a design in some future technology.

2.1 VHDL description of combinational networks

Below given is description of a simple gate network in VHDL. If each gate in the network of figure has a 5-ns propagation delay, the network can be described as follows:

```
C   <=   A and B after 5 ns;
E   <=   C or D after 5 ns;
```

Where A, B, C, D and E are signals. A signal in VHDL usually corresponds to a signal in a physical system. The symbol "<=" is the signal assignment operator, which indicates the value computed on the right side is assigned to the signal on the left side. When these statements are simulated, the first statement will be evaluated any time A or B changes, and the second statement will be evaluated any time C or D changes. Suppose that initially A=1, and B=C=D=E=0. If B changes to 1 at time 0, C will change to 1 at time=5 ns. Then E will change to 1 at time = 10 ns.

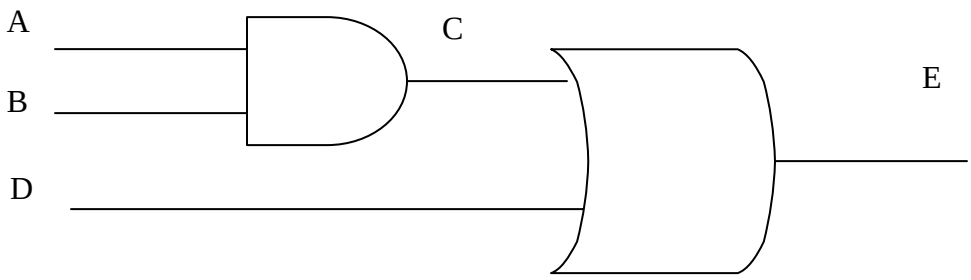


Fig. 2.1 GATE NETWORK

VHDL signal assignment statements, like the ones in the preceding example, are called concurrent statements when they are not contained in a VHDL process or block. The VHDL simulator monitors the right-hand side of each concurrent statement, and any time a signal changes, the expression on the right-hand side is immediately re-evaluated. The new value is assigned to the signal on the left-hand side after an appropriate delay.

When we initially describe a network, we may not be concerned about propagation delays. If we write

```
C<=A and B;

E<=C or D;
```

This implies that the propagation delays are 0 ns. In this case, the simulator will assume an infinitesimal delay referred to as (delta).

Unlike a sequential program, the order of the preceding statements is unimportant. If we write

```
E<=C or D;
C<=A and B;
```

The simulation results would be exactly the same as before. Even if a VHDL program has no explicit loops, concurrent statements may execute repeatedly as if they were in a loop. The VHDL statement

```
CLK <= not  CLK;
```

Will cause a run-time error during simulation. Since there is 0 delay, the value of CLK will change at time 0 +1 ,0+2…,0+3…etc, and real time will never advance.

In general, VHDL is not case sensitive; that is, capital and lowercase letters are treated the same by the compiler and simulator. Thus the statements

```
CLK <= NOT CLK After 10 NS;
```

And

```
CLK <=NOT CLK after 10 ns;
```

Are treated exactly the same. Signal names and other VHDL identifiers may contain letters, numbers, and the underscore character (_). An identifier must start with a letter, and it cannot end with an underscore. Thus C123 and ab_23 are legal identifiers, but 1 ABC and ABC-are not. Every VHDL statement must be terminated with a semicolon.

2.2 Entity-Architecture Pairs

To write a complete VHDL program, we must declare all the input and output signals and specify the type of each signal. As an example, we will describe the full adder of above figure. A complete description must include an entity declaration and an architecture declaration. The entity declaration specifies the inputs and outputs of the adder module:

```
Entity Full Adder is
  Port (X,Y, Cin: in bit;        -- inputs
        Cout, Sum: out bit);     -- outputs

End Full Adder;
```

The words entity, is port, in, out and end are reserved words (or keywords), which have a special meaning to the VHDL compiler. Anything that follows a double dash (--) is a VHDL comment. The port declaration specifies that X,Y and Cin are input signals of type bit, which means it can assume only values of ‘0’ or ‘1’.

The operation of the full adder is specified by an architecture declaration:

```
architecture Equations of Full Adder is
begin

        Sum <=X xor Y xor Cin after 10 ns;
        Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;

end Equations;
```

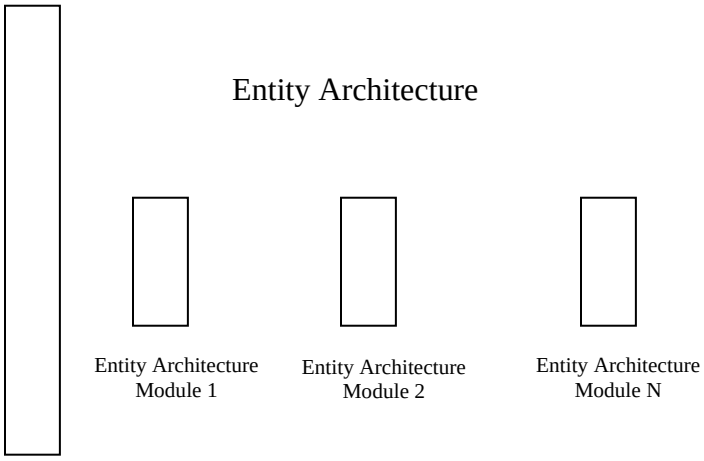


Fig. 2.2 VHDL Program Structure

In this example, the architecture name (Equations) is arbitrary; but the entity name (Full Adder) must match the name used in the associated entity declaration. The VHDL assignment statements for Sum and Cout represent the logic equations for the full adder. Several other architectural descriptions, such as a truth table or an interconnection of gates could have been used instead. In the Cout equation, parenthesis are required around (X and Y), since VDHL does not specify an order of precedence for the logic operators.

When we describe a system in VHDL, we must specify an entity and an architecture at the top level, and also specify an entity and architecture for each of the component modules that are part of the system. Each entity declaration includes a list of interface signals that can be used to connect to other modules or to the outside world. We will use entity declarations of the form

```
entity entity_name is

        [port (interface-signal-declaration);]

end [entity]  [entity_name];
```

The items enclosed in brackets are optional. The interface-signal-declaration normally has the following form;

```
list-of-interface-signals: mode type [:=initial-value]
```

Input signals are of mode in, output signals are of mode out, and bi-directional signals are of mode inout. The optional initial value is used to initialize the signals on the associated list; otherwise, the default initial value is used for the specified type. For example, the port declaration

Port (A, B: in integer :=2; C,D: out bit);
Indicates that A and B are input signals of type integer, which are initially set to 2, and C and D are output signals of type bit, which are initialized by default to '0'.

Associated with each entity is one or more architecture declarations of the form

```
architecture architecture_name of entity_name is

    (declaration)

begin

    architecture body

end [architecture] [architecture_name];
```

2.3 Compilation and Simulation of VHDL code

After describing a digital system in VHDL, simulation of the VHDL code is important for two reasons. First, we need to verify the VHDL code correctly implements the intended design; second, we need to verify that the design meets its specifications. Before the VHDL model of a digital system can be simulated, the VHDL code must first be compiled. The VHDL compiler, also called an analyzer, first checks the VHDL source code to see that it conforms to the syntax and semantic rules of VHDL. If there is a syntax error such as a missing semicolon, or if there is a semantic error such as trying to add two signals of incompatible types, the compiler will output an appropriate error message. The compiler also checks to see that references to libraries are correct. If the VHDL code confirms to all the rules, the compiler generates intermediate code, which can be used by a simulator or by a synthesizer.

In preparation for simulation, the VHDL intermediate code must be converted to a form that can be used by the simulator. This step is referred to as elaboration. During elaboration, ports are created for each instance of a component, memory storage is allocated for the required signals, the interconnections among the port signals are specified, and a mechanism is established for executing the VHDL processes in the proper sequence. The resulting data structure represents the digital

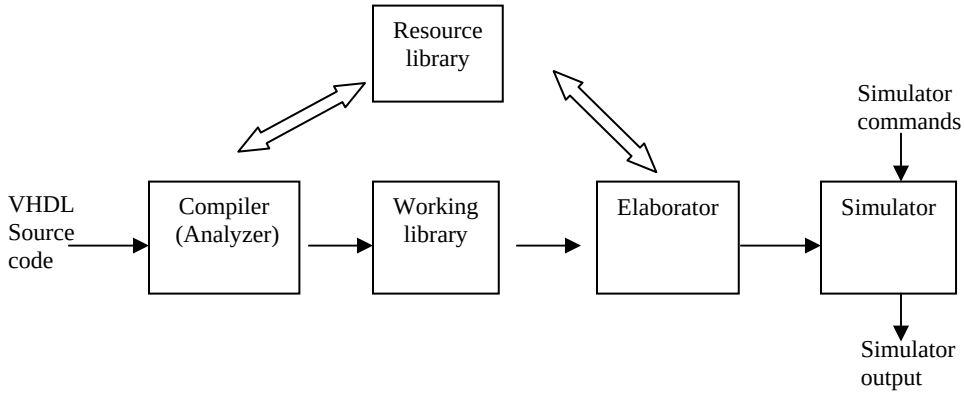


Fig. 2.3 Compilation, Elaboration, and Simulation of VHDL Code

system being simulated. After an initialization phase, the simulator enters the execution phase. The simulator accepts simulation commands, which control the simulation of the digital system and specify the desired simulator output.

2.4 Variables, Signals and Constants

Variables may be used for local storage in process, procedures, and functions. A variable declaration has the form :

Variable list_of_variable_names : type_name [:=initial_value];

Variables must be declared within the process in which they are used and are local to that process. An exception to this rule is shared variables. Signals, on the other hand, must be declared outside of a process. Signals declared at the start of an architecture can be used anywhere within that architecture.

A signal declaration has the form

signal list_of signal_names : type_name [:=initial_value];

A common form of constant declaration is

constant constant_name : type_name := constant_value;

A constant delay of type time having the value of 5 ns can be defined as

constant delay :time : = 5 ns;

Constants declared at the start of an architecture can be used anywhere within that architecture, but constants declared within a process are local to that process.

Variables are updated using a variable assignment statement of the form

variable_name :=expression;

When this statement is executed, the variable is instantaneously updated with no delay not even a delta delay. In contrast, consider a signal assignment of the form

```
signal_name<=expression [after delay];
```

The expression is evaluated when this statement is executed, and the signal is scheduled to change after delay. If no delay is specified, then the signal is scheduled to be updated after a delta delay.

2.5 Arrays

In order to use an array in VHDL, we must first declare an array type and then declare an array object. For example, the following declaration defines a one-dimensional array type names SHORT_WORD:

```
type SHORT_WORD is array (15 down to 0) of bit;
```

An array of this type has an integer index with a range from 15 down to 0, and each element of the array is of type bit.

Next, we declare array objects of type SHORT_WORD:

```
signal  DATA_WORD  :      SHORT_WORD;

variable  ALT_WORD  :     SHORT_WORD: = "0101010101010101";

constant  ONE_WORD:       SHORT_WORD := (others = '1');
```

DATA_WORD is a signal array of 16 bits, indexed 15 down to 0, which is initialized (by default) to all '0' bits. ALT_WORD is a variable array of 16 bits, which is initialized to alternating 0s and 1s. ONE_WORD is a constant array of 16 bits, all bits are set to 1 by (others => '1'). We can reference individual elements of the array by specifying an index value. For example, ALT_WORD (0) accesses the rightmost bit of ALT_WORD. We can also specify a portion of the array by specifying an index range: ALT_WORD (5 downto 0) accesses the low-order 6 bits of ALT_WORD , which have an initial value of 010101.

The array type and array object declarations illustrated here have the general forms

```
type array_type_name is array index_range of element_type;

signal array_name:  array_type_name [:= initial_values];
```

In the preceding declaration, signal may be replaced with variable or constant.

Multidimensional array types may also be defined with two or more dimensions. The following example defines a two-dimensional array variable, which is a matrix of integers with four rows and three columns:

```
type matrix_4x3 is array (1 to 4, 1 to 3) of integer;
variable matrix_a : matrix_4x3 := ((1,2,3), (4,5,6), (7,8,9), (10,11,12));
```

The variable matrix_a, will be initialized to

```
1   2   3
4   5   6
7   8   9
10  11  12
```

The array element matrix A (3,2) references the element in the third row and second column, which has a value of 8.

2.6 VHDL Operators

Predefined VHDL operators can be grouped into seven classes:

1. Binary logical operators: and or nand nor xor xnor
2. Relational operators : =/ = < <= > >=
3. Shift operators: sll srl sla sra rol ror
4. Adding operators: + - & (concatenation)
5. Unary sign operators : + -
6. Multiplying operators : * / mod rem
7. Miscellaneous operators : not abs **

When parentheses are not used, operators in class 7 have precedence and are applied first, followed by class 6, then class 5 etc. Class 1 operators have lowest precedence and are applied last. Operators in the same class have the same precedence and are applied from left to right in an expression. The precedence order can be changed by using parentheses.

3. IMPLEMENTATION

Any processor will two distinctive parts. One is Datapath and the other one being Control. MIPS datapath can be either single cycle or multicycle.

3.1 Single Cycle

Datapath executes all instructions in 1 clock cycle. This means that no datapath resource can be used more than once per instructions, so any element needed more than once must be duplicated. Inefficient because clock cycle is determined by the longest possible path in the machine (usually load instructions) so the overall performance of the single cycle instruction is not likely to be good as several of the instructions can be completed in a shorter cycle.

3.2 Multicycle

Each instruction is broken into several steps and each of such steps take one clock cycle to executes. This allows functional units to be used more than once per instruction as long as it is used in different clock cycle.

In this project we are modeling a multicycle datapath with finite state machine control for a 32-bit MIPS processor.

In a multicycle implementation, each step in the execution will take 1 clock cycle. The multicycle implementation allows a functional unit to be used more than once per instruction, as long as it is used on different clock cycles. This sharing can help reduce the amount of hardware required. The ability to allow instructions to take different numbers of clock cycles and the ability to share functional units within the execution of a single instruction are the major advantages of a multicycle design.

At the end of a clock cycle, all data that are used in subsequent clock cycles must be stored in a state element. Data used by subsequent instructions in a later clock cycle is stored into one of the programmer-visible state elements (i.e., the register file, the PC, or the memory). In contrast, data used by the same instruction in a later cycle must be stored into one of these additional registers.

Thus the position of the additional registers is determined by the two factors: what combinational units will fit in a clock cycle and what data are needed in later cycles implementing the instruction. In this multicycle design, we assume that the clock cycle can accommodate at most one of the following operations: a memory access, a register file access (two reads or one write), ore an ALU operation. Thus any data produced by one of these three functional units (the memory, the register file, or the ALU) must be saved into a temporary register for use on a later cycle.

3.3 Components of Datapath

A reasonable way to start a datapath design is to examine the major elements to required to execute each class of MIPS instruction. At the end of a clock cycle, all data that are used in subsequent clock cycles must be stored in a state element Data used by subsequent instructions in a Later Clock Cycle is stored into one of the programmer visible state elements (register file, the PC or the memory) where as

the data used by the same instruction in a later cycle is stored into one of the general purpose registers provided after each main functional unit.

The datapath will have the following two types of components.

Combinational Elements (Adder, MUX, ALU)
Storage Elements (Register, Register File)

A multicycle datapath is shown in the figure 3.1. It contains the following

(1) Register array - 32 registers of 32- bit size

(2) Register 32 bit -Memory Data Register

Register A

Register B

ALU Out Register

(3) Registers 32 bit with enable input - These are used where programmer visible registers are needed

(i) Instruction Register (Write is controlled with IRWrite Signal from PSM)
(ii) Program Counter (32 bit) write is controlled with PC Write control signal

(4) Multiplexers All controlled with signals coming from PSM Control unit.

(5) Control Unit

3.4 Register file

The processors 32 registers are stored in a structure called a register file. A register file is collection of register in which any register can be read or written by specifying the number of the register in the file. It has three inputs for selecting the register to be operated (2 for read operation and one in case of write operation)

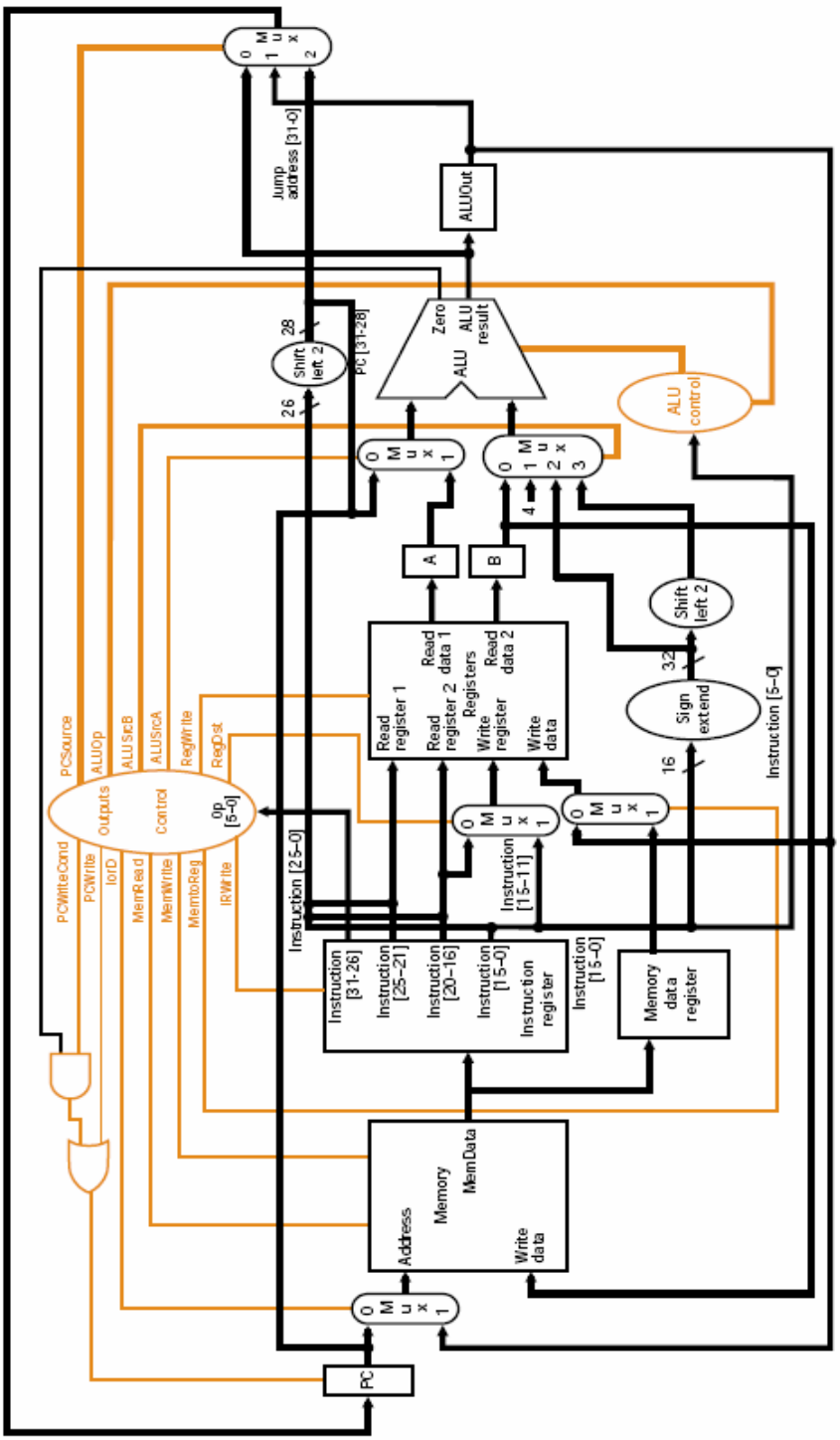


Fig. 3.1 MIPS DATAPATH & CONTROL

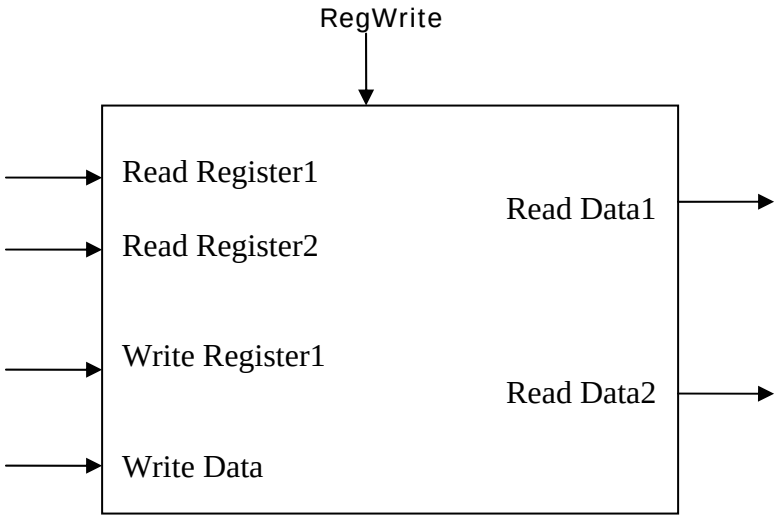


Fig. 3.2 Register File

To write data into the register file, we need 2 inputs one to specify the register number one to supply the data. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes however are controlled by write control signal (Regwrite) which must be asserted for a write to occur at the clock cycle.

3.5 MIPS instruction format

The layout of instruction is called the instruction format. MPIS instructions are classified into three types.

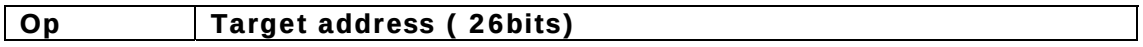
R-type

31					0
Op	**Rs**	**Rt**	**Rd**	**Shamt**	**funct**
6bits	**5 bits**	**5bits**	**5bits**	**5bits**	**6bits**

I-type

31			0
Op	**Rs**	**Rt**	**Immediate**
6 bits	**5bits**	**5bits**	**16bits**

J TYPE



As it can be seen all **MIPS** instructions are 32 bit long.

The different fields are :

- **op - operation of the instruction**
- **rs,rt,rd - source and destination registers**
- **shamt - shift amount**
- **funct - selects the variant of operation in Op field.**

Here we will simulating MIPS datapath for R type (Arithmetic Operations) and I type (Load/Store Operations).

R Type Instructions

R type instructions have an **OP-code** of '0' always. These instructions have three register operands : **rs, rt** and rd. Field **rs** and **rt** are the source registers and rd is the destination. **ALU** function is in the **funct** field and is decoded by the **ALU** control. The **R** type instruction that we implement are **add**, **sub**, **and, or** and **slt** (**set on less than**). The **shamt** field is only used in shift instructions which we will ignore presently.

R type instructions implemented here are

ADD rd, rs, rt => rd = rs + rt

SUB rd, rs, rt => rd = rs - rt

AND rd, rs, rt => rd = rs & rt

OR rd, rs, rt => rd = rs || rt

SLT rd, rs, rt => if rs < rt, rd=1 else rd=0

Fields in the case of **ADD** instruction

As mentioned earlier, for and **R** type instruction the **OP** and **SHAMT** fields will be zero always (first and fifth fields). The second field gives the number of the register that is the first source operand (**rs**) and the third field indicates the second source operand (**rt**). Fourth field indicated the destination where the addition result will be stored (**rd**). Sixth filed defined the function to be performed, in this case it will be 100000 (32) for addition.

The above explanation is valid for other R type instructions as well. The only field that will change is that of function.

R TYPE INSTRUCTION	FUNCTION FIELD ENTRY
ADD	100000 (32)
SUB	100010 (34)
AND	100100 (36)
OR	100101 (37)
SLT	101010 (42)

3.6 ALU CONTROL

The control input to the data path are coming from several fields of Instruction Register and from the **FSM** controller. **ALU** has three control inputs. Only five of the possible eight input combinations are used.

ALU CONTROL INPUT	FUNCTION
000	AND
001	OR
010	ADD
110	SUBTRACT
111	SET ON LESS THAN

Depending on the instruction class the ALU will need to perform one of these five functions. For load word (LW) and store word (SW) instructions we use ALU to compute the memory address by addition. For R type instruction ALU needs to perform for the five functions (Addition, Subtraction, AND, OR and SLT) depending on the value of 6 bit function feed in the lower bits of the instruction.

We can generate the 3 bit ALU Control using a small control unit that has as inputs the function field of the instruction and a 2 bit control field which is ALUOp. Table

illustrates the way in which the ALU Control input is decided using ALUOp and function code.

Instruction OPCODE	ALUOp	Function Field	Desired ALU action	ALU control input
LW	00	XXXXXX	ADD	010
SW	00	XXXXXX	ADD	010
R - TYPE	10	100000	ADD	010
R - TYPE	10	100010	SUBTRACT	110
R - TYPE	10	100100	AND	000
R - TYPE	10	100101	OR	001
R - TYPE	10	101010	SET ON LESS THAN	111

This style of using multiple levels of decoding (i.e. the main control unit generates the ALUOp bits , which then are used as inputs to the ALU control that generates the actual signal to control the ALU Control) can reduce the size of the main control unit which in turn increases the speed of control unit.

3.7 CONTROL UNIT

The control is implemented using a finite state machine. A finite state machine consists of a set of states and directions on how to change states. The directions are defined by next-state functions, which maps the current state and the inputs to a new state. When we use a finite state machine, each state also specifies a set of outputs that are asserted when the machine is in that state. The implementation of a finite state machine usually assumes that all outputs are not explicitly asserted are deasserted. The correct operation of the datapath depends on the fact that a signal that is not explicitly asserted is reasserted, rather than acting as a don't care.

Multiplexer controls are slightly different, since they select one of the inputs whether they are 0 or 1. Thus in the finite state machine, we always specify the setting of all the multiplexer controls that we care about.

The finite state control essentially corresponds to the five steps of execution that will be discussed later. The finite state machine will consists of several parts. Since the first 2 states of FSM are identical for every instruction, the initial two states of the FSM will be common to all instructions. Step 3 through 5 differ, depending on the OPcode. After the execution of the last step for a particular instruction class, the FSM will return to the initial state to begin fetching the next instruction.

Figure 3.3 in the next page graphically represents the FSM used in the case. The signals that are asserted in each case state are shown within the circle representing the state. The arcs between the states define the next states and are labeled with conditions that select a specific next state when multiple states are possible. The process of branching to different states depending on the instruction

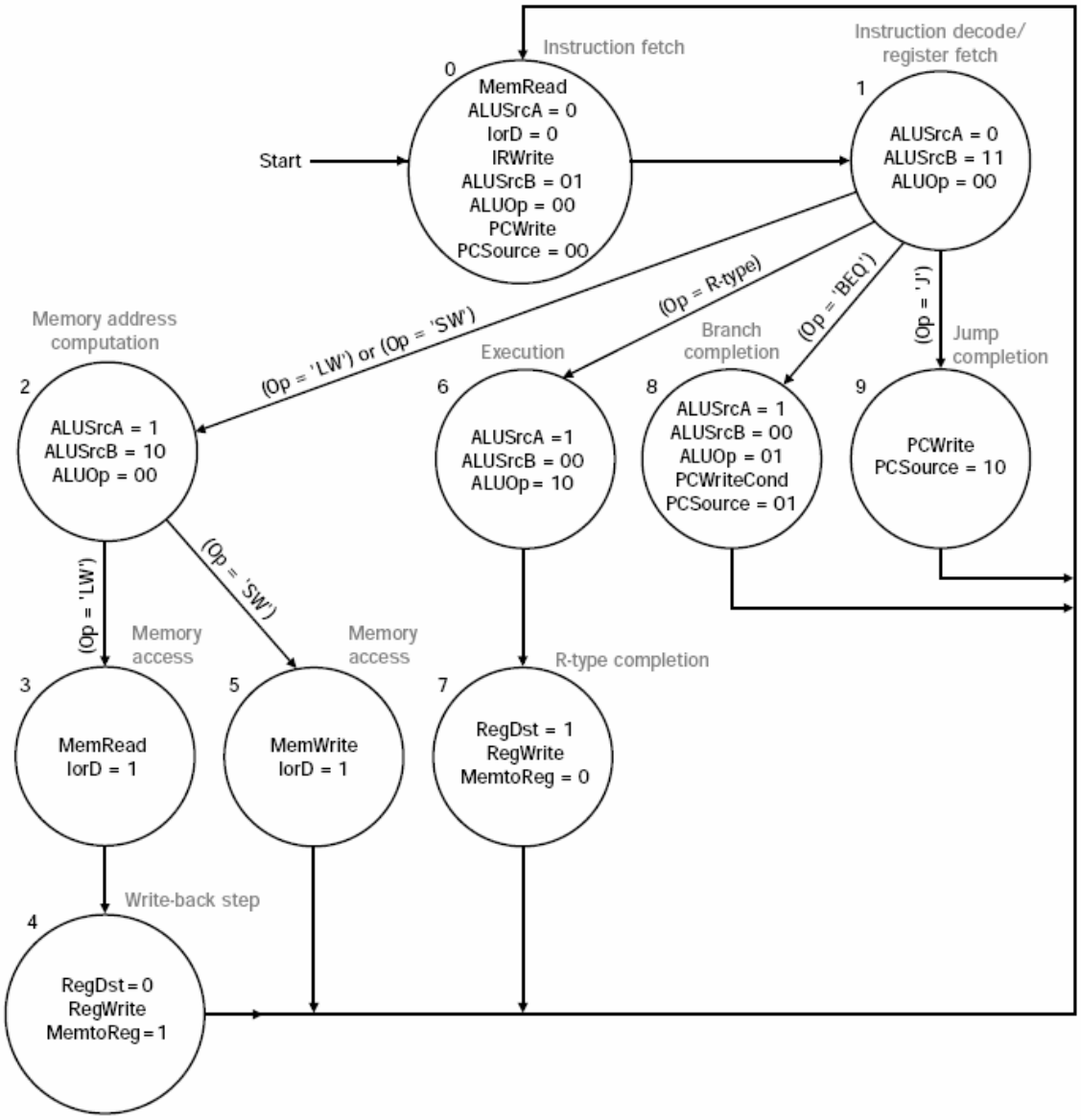


Fig 3.3 The Combined control unit

is called decoding, since the choice of next state and hence the actions that follow, depend on the instruction class.

For memory reference, the first state after fetching the instruction and registers computes the memory address (state 2). To compute the memory address, the ALU input multiplexers must be set so that the first input is the A register while the second is the sign extended displacement field, the result is written into the ALUOut register. After the memory address calculation the memory should be read or written, this requires two different states. If the instruction OPcode is LW, then state 3 does the memory read.

Output of memory is always written into MDR. If it is SW, state 5 does a memory write. In state 3 and 5, the signal IorD is set to 1 to force the memory address to come from the ALU. After performing a write, the SW instruction has completed execution, and next state is 0. If the instruction is a load, however another state (state 4) is needed to write the result from memory into the register file. After this state, corresponding to the memory read completion step, next state is 0.

To implement R type instruction requires two states corresponding to step 3 (execute) and 4 (R type completion). State 6 asserts ALUSrcA and ALUSrcB signal to 00, this forces the two registers that were read from the register file to be used as inputs to the ALU setting ALUOp to cause ALU control unit to use the function field to set the ALU control signed. In state 7, RegWrite signal is asserted to cause register file write.

3.8 Breaking the instruction Execution into Clock Cycles

Given the datapath details, we now need to look at what should happened in each clock cycle of the multicycle execution, since this will determine what additional control signals may be needed, as well as the setting of the control signals.

All the operations listed in one steps occur in parallel within 1 clock cycle while successive steps operate in series in different clock cycles. The limitation of 1 ALU operations, one memory access, one register file access determines what can fit in one step.

Step 1

Instruction Fetch step

IR = Memory (PC)

PC = PC + 4

Operation : Send the PC to the memory as address perform a read and write the instruction into the Instruction Register (IR). Also increment PC BY 4.

To implement this the following control signals are needed

Mem Read = 1
IR Write = 1
IorD = 0 (to select PC as the source of address)
ALUSrcA = 0 (sending PC to ALU)
ALUSrcB = 01 (sending 4 to ALU)
ALUOp = 00 (To make ALU add
PC write = 1 (For storing the incremented instruction address back to PC

The increment of PC and instruction memory access occur in parallel. The new value of PC is not visible until the next clock cycle.

Step 2

Only optimistic actions are performed in this step as the exact nature of instruction is not known yet. So, action performed in this step is access the register file to read register *rs* and *rt* and store the result into registers A and B.

A = Reg [IR(25-21)],

B = Reg [IR (20-16)];

Step 3

Execution, or Memory address computation.

This is the first cycle during which the datapath operation is determined by the instruction class. In all the cases, the ALU is operating on the operands prepared in the previous step, performing one of the following functions depending on the instruction class.

(i) Memory reference:

ALUOut = A+ sign Extend [IR(15-0)]

To implement this the following control signals are needed

ALUSrcA = 1

ALUSrcB = 10

So that output of sign extension unit is used for second ALU input ALU output will be 00 causing ALU to Add.

(ii) Arithmetic – logic Unit (R-type)

ALUOut = A op B

ALU is performing the function specified by the function field on the two values read from the register file. ALUSrcA= 1 and ALUSrcB = 10 causing A & B to be used as

ALU inputs. ALUOp is set to '10' so that function field is used to determine the ALU control signal.

Step 4

Memory Access or R type Instruction completion step: During this step, a load or store instruction accesses memory and a arithmetic logic instruction writes its result when a value is stored into the memory data register (MDR), where it must be used on the next clock cycle.

MDR = Memory (ALUOut)

Or

Memory (ALUOut) = B

In either case, the address used, is the one computed during the previous step and stored in ALUOut.

The signal MemRead or MemWrite is asserted for loads, IorD is set to 1 to force memory address to come from the ALU

In the case of R-type instructions

Reg [IR(15-11)] = ALUOut

Place the content of ALUOut into result register. RegDst is set to 1 (to force rd 15-11) field to be used to select register file entry to write. RegWrite is asserted and MemtoReg is set to '0' so that ALUOut is written to the register file as opposed to the memory data output.

Step 5

Memory read completion step during this step, load complete by writing back the value from memory

Reg [IR(20-16)] = MDR

MemtoReg =1 to write the result from memory

RegWrite = 1 (to cause a write)

RegOut = 0 to choose rt (20-16) as register

4. MEMORY

While designing memory we take advantage of principle of locality by implementing the memory of computer as a memory hierarchy. A memory hierarchy consist of multiple levels of memory with different speeds and sizes. Faster memories are more expensive per bit than the slower memories and thus are usually smaller.

Today there are three primary technologies used in building memory hierarchy. Main memory is implemented using DRAM (Dynamic Random Access Memory) while levels closer to the CPU are implemented using SRAM (Static Random Access Memory). DRAM is less costly per bit than SRAM, although it is substantially slower. The price difference arises because DRAM uses significantly less area per bit of memory and DRAMs thus have larger capacity for the same amount of silicon. Because of the difference of cost and access time, it is advantageous to build memory as hierarchy of levels with faster memory closer to the CPU and slower less expansive memory below that as shown in figure 4.1

The memory system is organized as a hierarchy , a level closer to the process is a subset of any level further away and all data is stored in the lowest level.

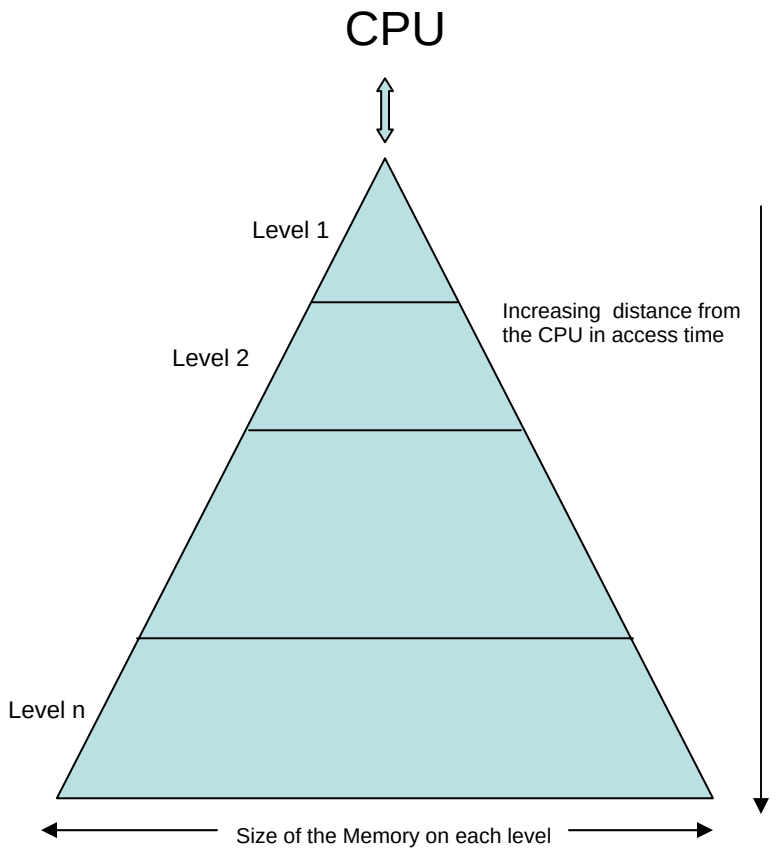
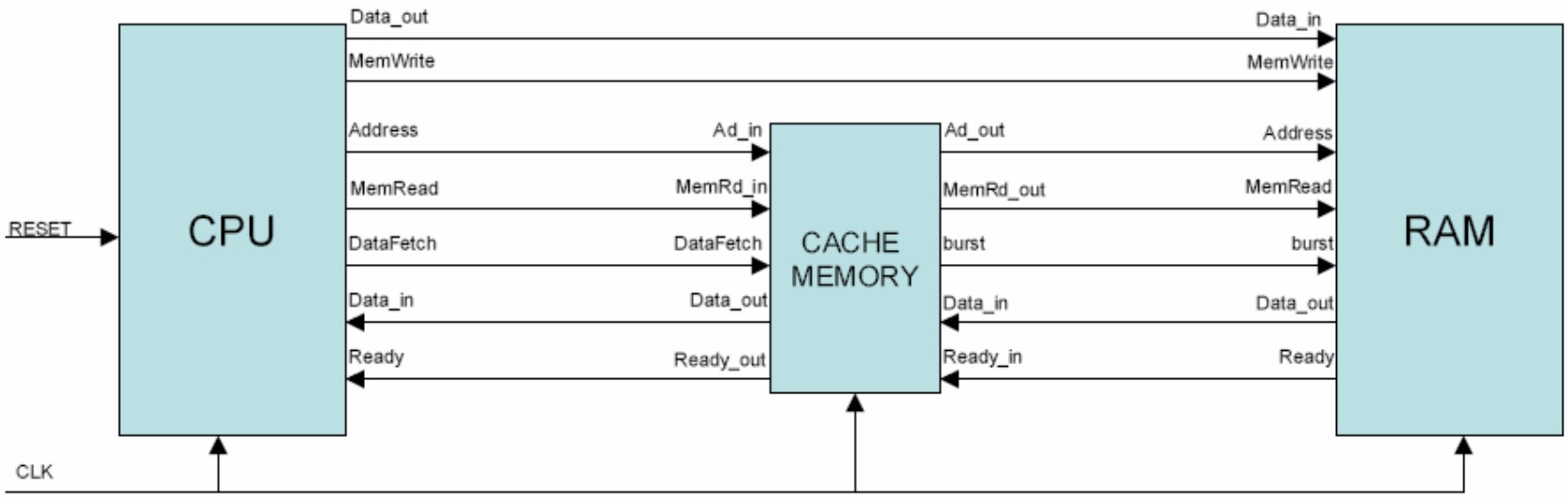


Figure 4.1 Memory Hierarchies



MEMORY HIERARCHY IMPLEMETATION FOR MIPS CPU

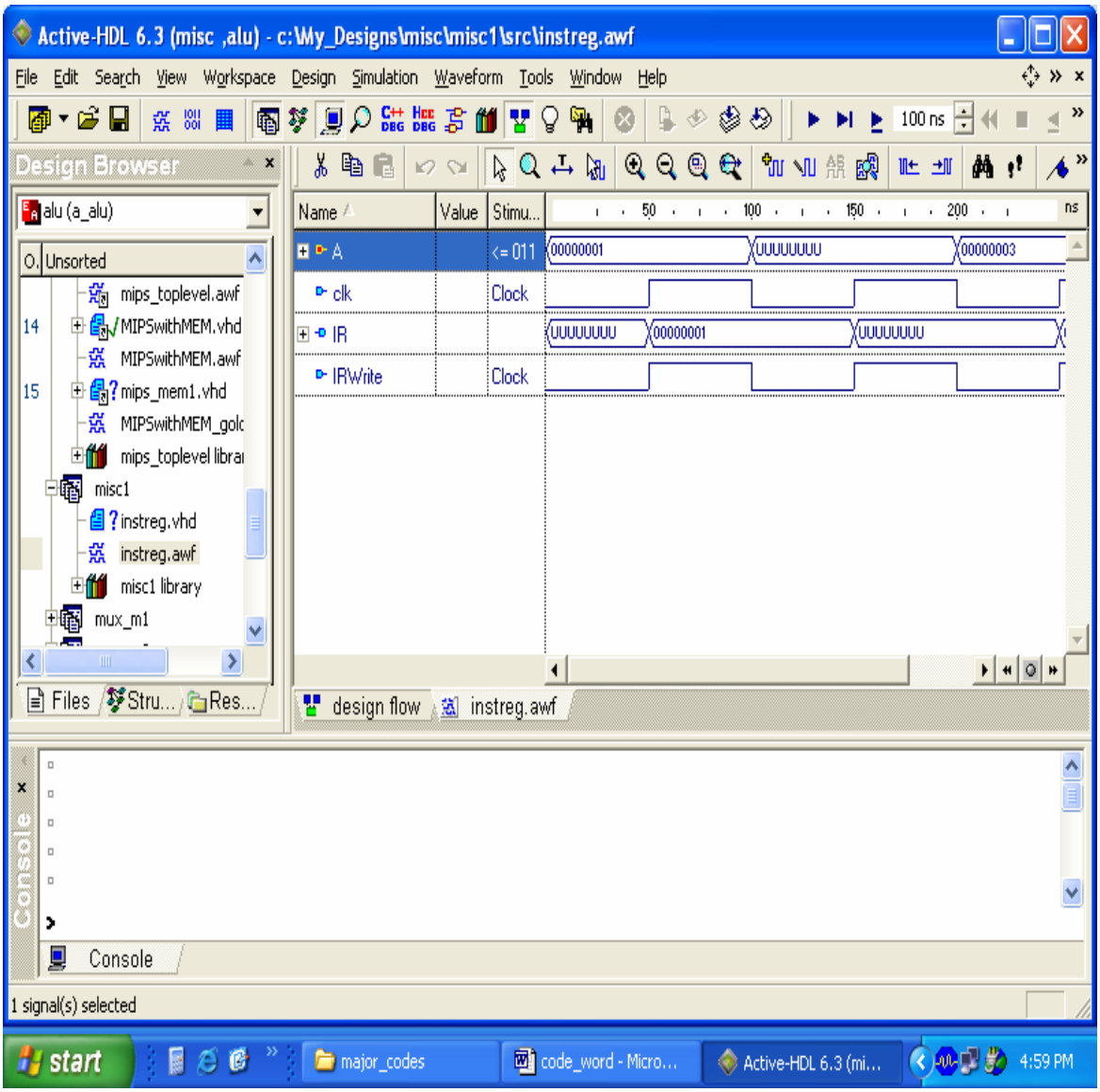
A memory hierarchy can consists of multiple levels, but data is copied between only two adjacent levels at a time. If the data requested by the CPU appears in some block in the upper level, this is called a *hit* and if data is not present in the upper level the request is called a *miss.* Since performance is a major reason for having a memory hierarchy, the speed of hits and misses is important. Hit time is the time to access the upper level of memory which includes the time needed to determine whether the access is a hit or a miss. The miss penalty is the time to replace a block in the upper level with the corresponding block from the lower level plus the time to deliver this block to the CPU.

Program exhibit both temporal locality, the tendency to reuse recently accessed data items and spatial locality, the tendency to refer data items that are close to other recently accessed items. Memory hierarchy takes advantage of temporal locality by keeping more recently accessed data items closer to the CPU. Memory hierarchy takes advantage of spatial locality by moving blocks of multiple contiguous words in memory to upper level of hierarchy.

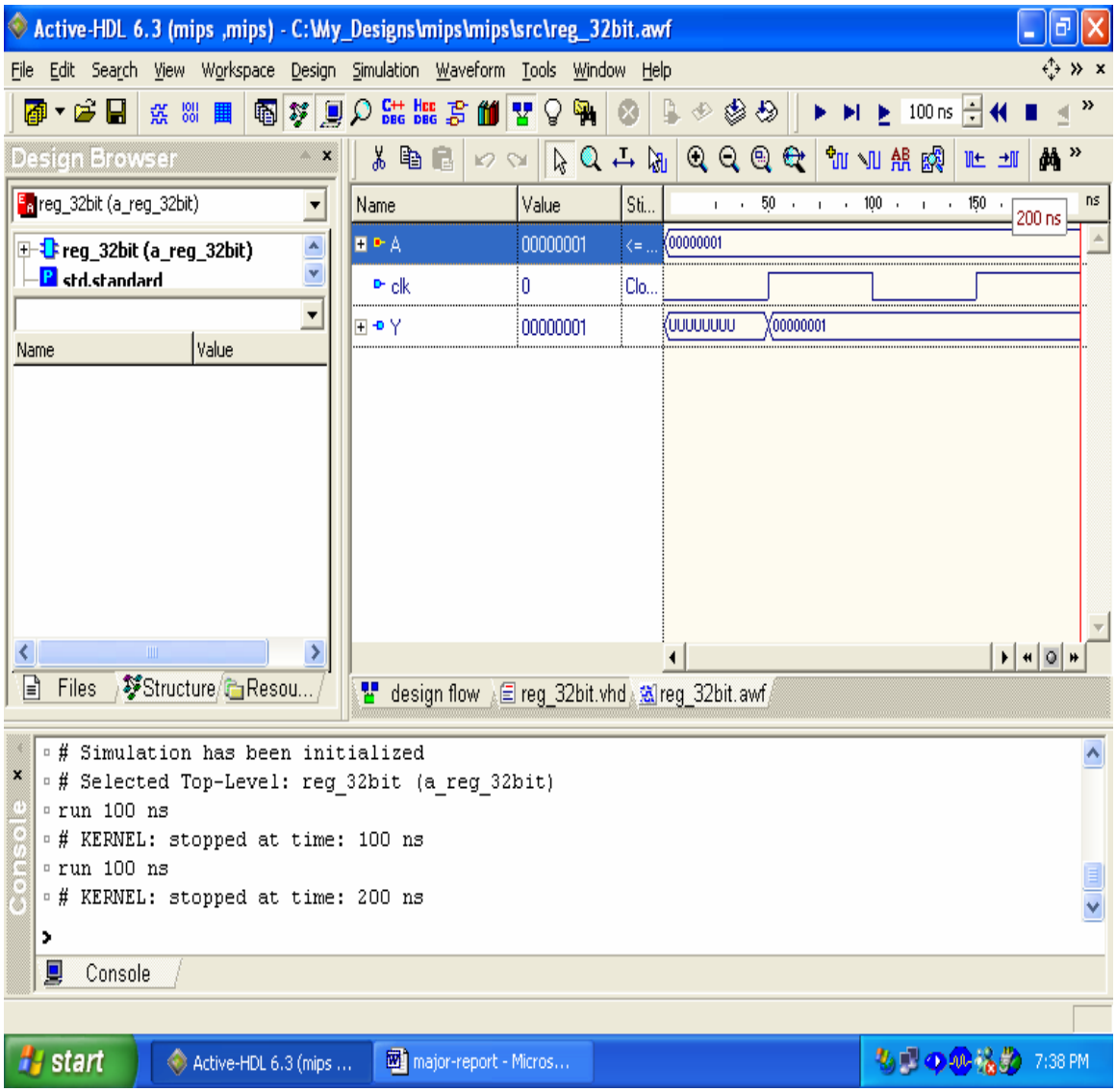
In our design a main memory is containing both data and instruction area. A write through direct mapped cache is designed for data memory. All instructions are maintained in the main memory itself. As soon as the first data access is initiated, data from adjacent locations are also copied to the cache memory using burst transfer.

5. VHDL CODES & SIMULATION RESULTS

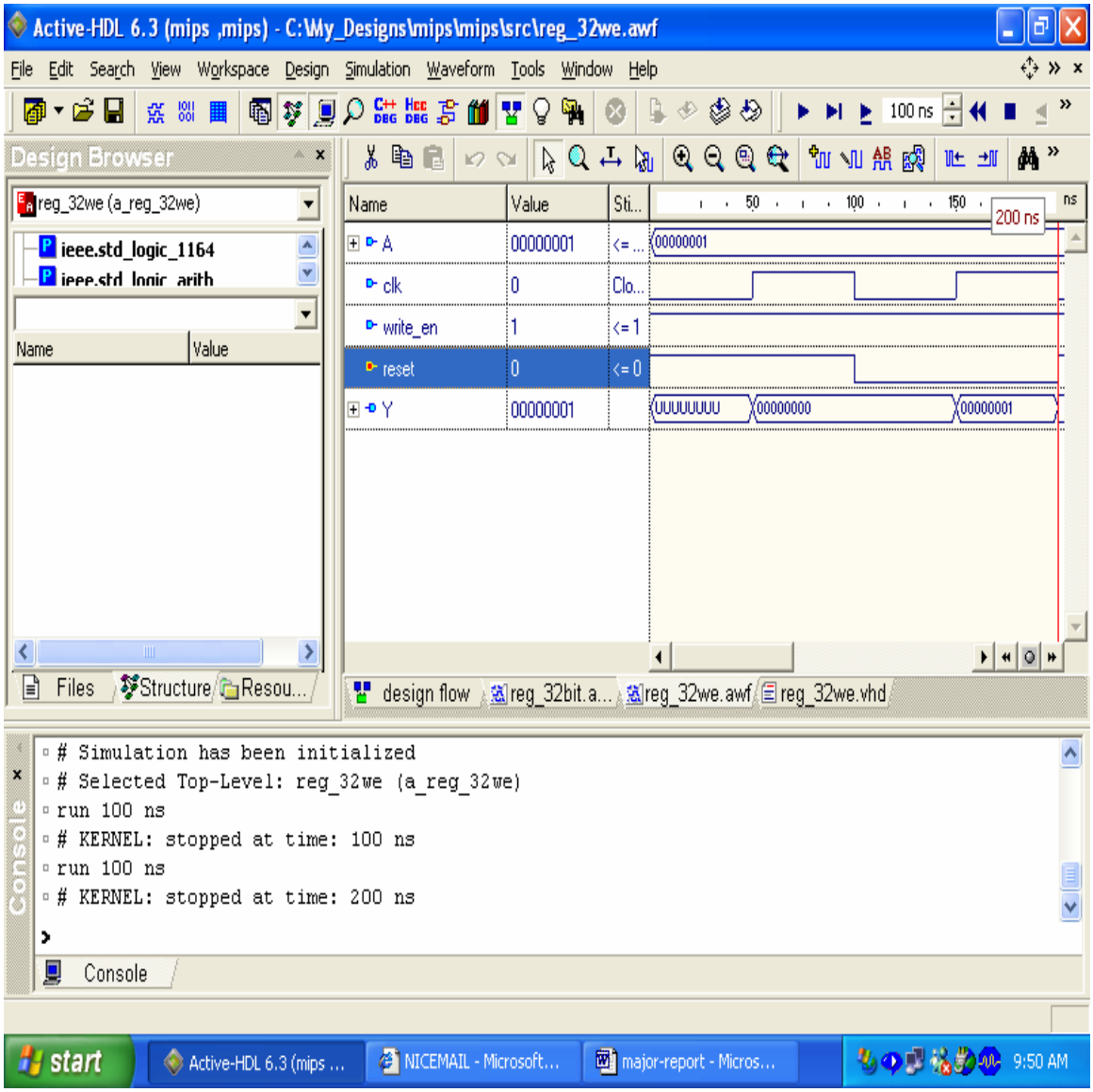
```
----------------------------------------------------
-- *****simulation of mips processor  ********
-- 32 bit instruction register
----------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_bit.all;
----------------------------------------------------
entity instreg is
port(   A               :       IN std_logic_vector(31 downto 0);
        clk             :       IN std_logic;
        IRWrite         :       IN bit;
        IR              :       OUT unsigned (31 downto 0));
end instreg;
----------------------------------------------------
architecture a_instreg of instreg is
begin
        process
        begin
        wait on clk until rising_edge(clk)and IRWrite ='1' ;
        IR <= unsigned (to_bitvector(A));
        end process;
end a_instreg;
----------------------------------------------------
```


Active-HDL 6.3 (misc ,alu) - c:\My_Designs\misc\misc1\src\instreg.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
100 ns
Design Browser
alu (a_alu)
Unsorted
mips_toplevel.awf
MIPSwithMEM.vhd
MIPSwithMEM.awf
mips_mem1.vhd
MIPSwithMEM_golc
mips_toplevel libra
misc1
instreg.vhd
instreg.awf
misc1 library
mux_m1
Files
Stru...
Res...
Name
Value
Stimu...
50
100
150
200
ns
A
<= 011
00000001
UUUUUUUU
00000003
clk
Clock
IR
UUUUUUUU
00000001
UUUUUUUU
IRWrite
Clock
design flow
instreg.awf
Console
Console
1 signal(s) selected
start
major_codes
code_word - Micro...
Active-HDL 6.3 (mi...
4:59 PM

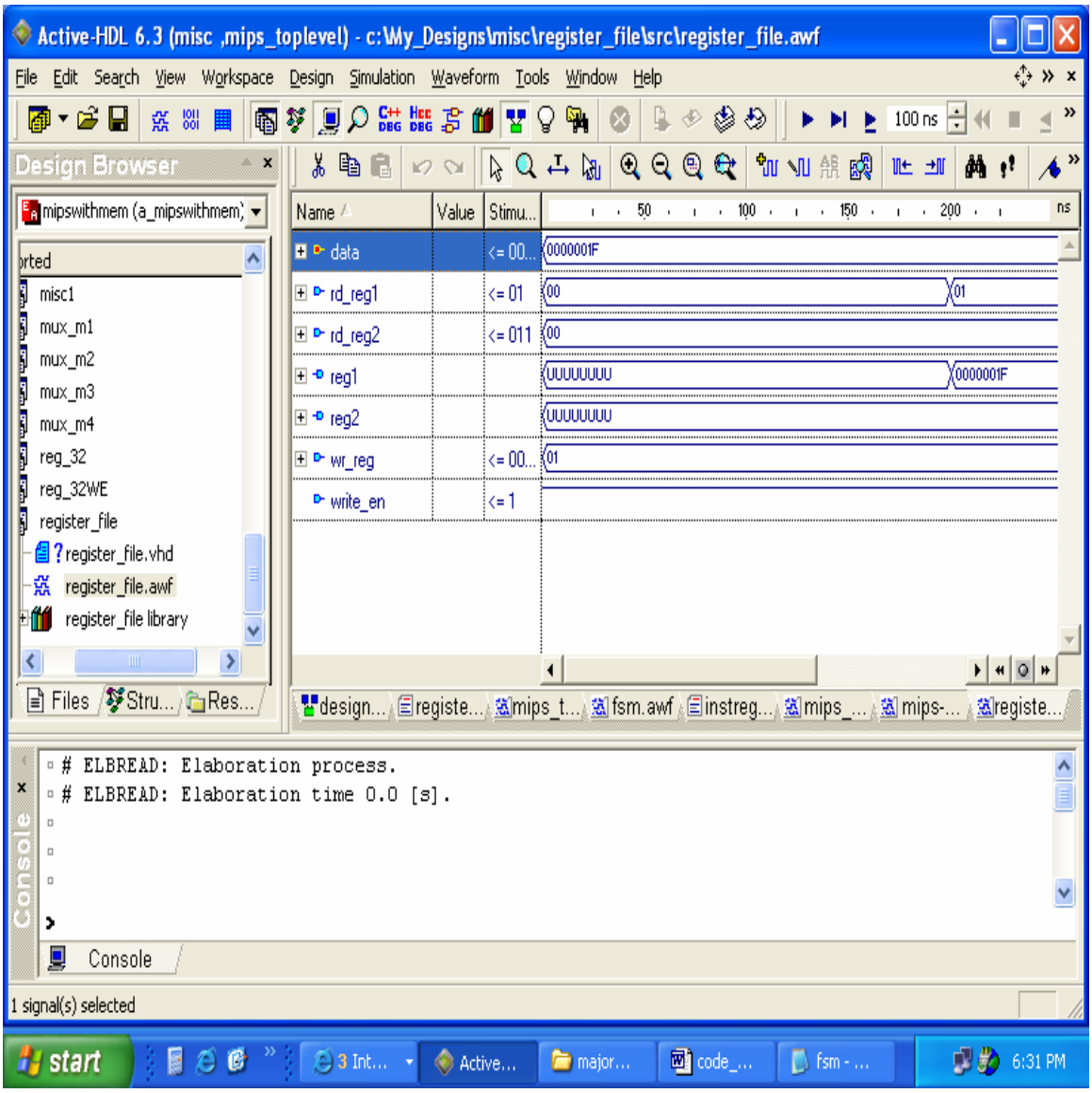
```
--------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
--------------------------------------------------

entity reg_32bit is

        port(
                A               :       IN std_logic_vector(31 downto 0);
                clk             :       IN std_logic;
                Y               :       OUT std_logic_vector(31 downto 0));

end reg_32bit;
--------------------------------------------------

architecture a_reg_32bit of reg_32bit  is

begin

        process
        begin
        wait until clk'event and clk= '1';
        Y <= A;
        end process;

end a_reg_32bit;

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


Active-HDL 6.3 (mips ,mips) - C:\My_Designs\mips\mips\src\reg_32bit.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
100 ns
Design Browser
reg_32bit (a_reg_32bit)
reg_32bit (a_reg_32bit)
std.standard
Name
Value
Files
Structure
Resou...
Name
Value
Sti...
50
100
150
ns
200 ns
A
00000001
clk
0
Clo...
Y
00000001
UUUUUUUU
00000001
design flow
reg_32bit.vhd
reg_32bit.awf
Simulation has been initialized
Selected Top-Level: reg_32bit (a_reg_32bit)
run 100 ns
KERNEL: stopped at time: 100 ns
run 100 ns
KERNEL: stopped at time: 200 ns
Console
Console
start
Active-HDL 6.3 (mips ...
major-report - Micros...
7:38 PM

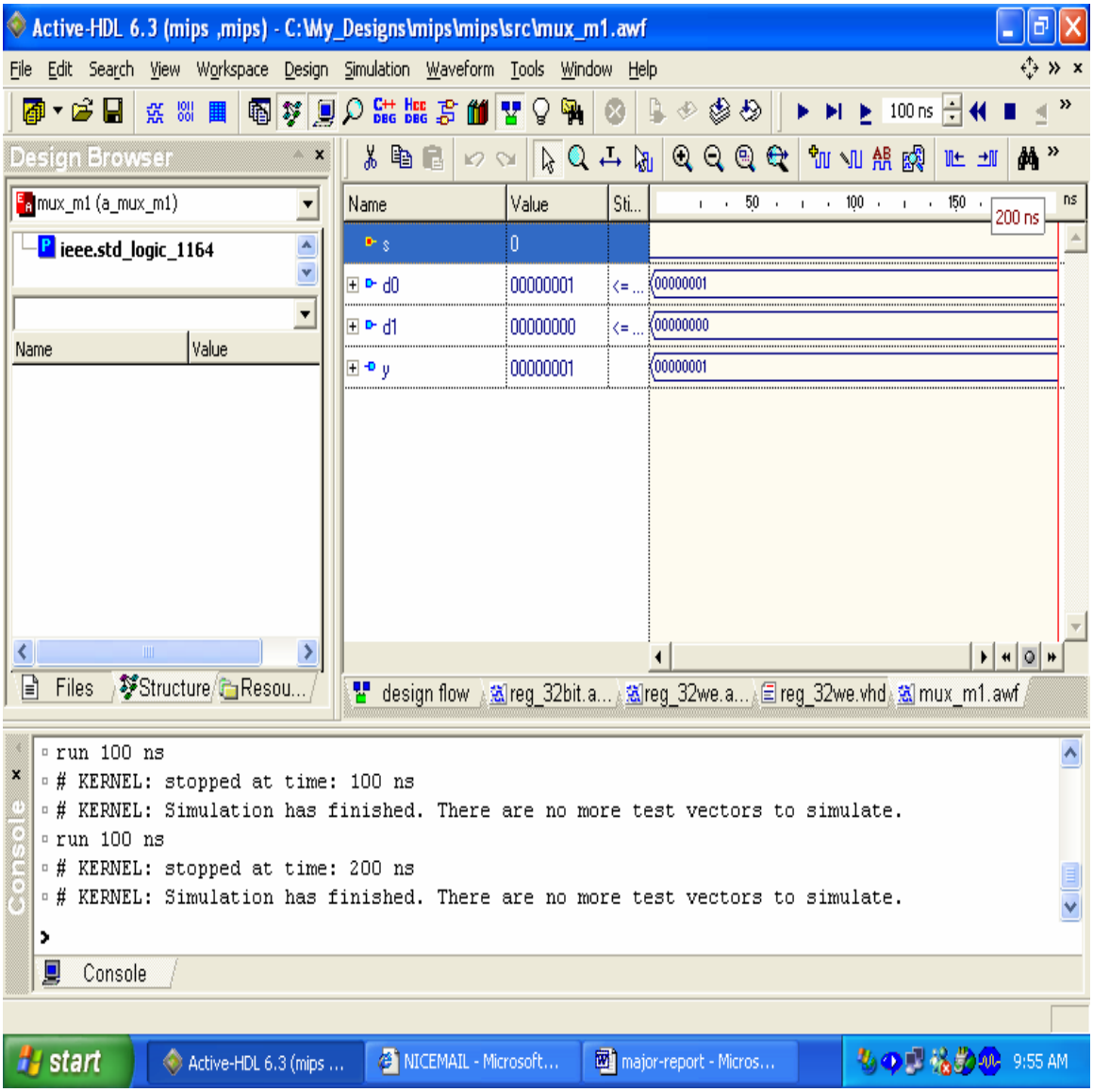
```
--------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
--------------------------------------------------
entity reg_32we is
port(
	A		:	in std_logic_vector(31 downto 0);
	clk		:	in std_logic;
	write_en	:	in bit;
	reset		:	in bit;
	Y		:	out std_logic_vector (31 downto 0));
end reg_32we;
--------------------------------------------------
architecture a_reg_32we of reg_32we is
begin
	process (clk)
	begin
	if clk = '1' and clk'event then
		if reset = '1' then
			Y <= x"0000_0000";
		elsif write_en ='1'  then
			Y<= A;
		end if;
	end if;
	end process;
end a_reg_32we;
--------------------------------------------------
```


Active-HDL 6.3 (mips ,mips) - C:\My_Designs\mips\mips\src\reg_32we.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
100 ns
Design Browser
reg_32we (a_reg_32we)
ieee.std_logic_1164
Name
Value
Files
Structure
Resou...
Name
Value
Sti...
50
100
150
ns
200 ns
A
00000001
<=...
00000001
clk
0
Clo...
write_en
1
<=1
reset
0
<=0
Y
00000001
UUUUUUUU
00000000
00000001
design flow
reg_32bit.a...
reg_32we.awf
reg_32we.vhd
Simulation has been initialized
Selected Top-Level: reg_32we (a_reg_32we)
run 100 ns
KERNEL: stopped at time: 100 ns
run 100 ns
KERNEL: stopped at time: 200 ns
Console
Console
start
Active-HDL 6.3 (mips ...
NICEMAIL - Microsoft...
major-report - Micros...
9:50 AM

```
-------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_bit.all;
-------------------------------------------------
entity register_file is
	port(
		write_en	:	in bit;
		rd_reg1		:	in unsigned (4 downto 0);
		rd_reg2		:	in unsigned (4 downto 0);
		wr_reg		:	in unsigned (4 downto 0);
		data		:	in std_logic_vector(31 downto 0);
		reg1		:	out std_logic_vector(31 downto 0);
		reg2		:	out std_logic_vector(31 downto 0));
end register_file;
-------------------------------------------------------

architecture a_register_file of register_file is
begin



	regtag: process ( rd_reg1, rd_reg2, wr_reg, data, write_en) is

   type reg_array is array (0 to 31 ) of std_logic_vector(31 downto 0);
	variable reg_file : reg_array;
	variable index1,index2,index3 : natural;

	begin

		reg_file(0) := X"00000001";
		reg_file(1) := X"00000002";
		reg_file(2) := X"00000003";
		reg_file(3) := X"00000004";
		reg_file(6) := X"00000000";
		--write port
		if write_en = '1' then
			index3 := to_integer(wr_reg);
			reg_file(index3) := data;
		end if;

		--read port1
		index1 := to_integer(rd_reg1);
		reg1 <= reg_file(index1) ;

		--read port2

		index2 := to_integer(rd_reg2);
		reg2 <= reg_file(index2) ;
	end process regtag;
end a_register_file;
 -------------------------------------------------------
```


Active-HDL 6.3 (misc ,mips_toplevel) - c:\My_Designs\misc\register_file\src\register_file.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Design Browser
mipswithmem (a_mipswithmem)
misc1
mux_m1
mux_m2
mux_m3
mux_m4
reg_32
reg_32WE
register_file
register_file.vhd
register_file.awf
register_file library
Files Stru... Res...
Name Value Stimu...
data <= 00... 0000001F
rd_reg1 <= 01 00 01
rd_reg2 <= 011 00
reg1 UUUUUUUU 0000001F
reg2 UUUUUUUU
wr_reg <= 00... 01
write_en <= 1
50 100 150 200 ns
design... registe... mips_t... fsm.awf instreg... mips_... mips-... registe...
ELBREAD: Elaboration process.
ELBREAD: Elaboration time 0.0 [s].
Console
1 signal(s) selected
start
3 Int...
Active...
major...
code_...
fsm - ...
6:31 PM

```
-----------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_bit.all;
-----------------------------------------------
entity mux_m1 is
        port (
                s       :       in bit;
                d0      :       in std_logic_vector (31 downto 0);
                d1      :       in std_logic_vector (31 downto 0);
                y       :       out unsigned (31 downto 0));
end  mux_m1;
-----------------------------------------------
architecture a_mux_m1 of mux_m1 is

begin

process (s,d0,d1)

begin
        case  s is
        when '0' =>
        y <= unsigned (to_bitvector(d0));
        when '1' =>
        y <= unsigned (to_bitvector(d1));
        end case;
end process;

end architecture a_mux_m1;
-----------------------------------------------
```


Active-HDL 6.3 (mips ,mips) - C:\My_Designs\mips\mips\src\mux_m1.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Design Browser
mux_m1 (a_mux_m1)
ieee.std_logic_1164
Name
Value
Files
Structure
Resou...
Name
Value
Sti...
50
100
150
200 ns
ns
s
0
d0
00000001
d1
00000000
y
00000001
design flow
reg_32bit.a...
reg_32we.a...
reg_32we.vhd
mux_m1.awf
run 100 ns
KERNEL: stopped at time: 100 ns
KERNEL: Simulation has finished. There are no more test vectors to simulate.
run 100 ns
KERNEL: stopped at time: 200 ns
KERNEL: Simulation has finished. There are no more test vectors to simulate.
Console
start
Active-HDL 6.3 (mips ...
NICEMAIL - Microsoft...
major-report - Micros...
9:55 AM

```
-----------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_bit.all;
-----------------------------------------------
entity mux_m2 is
        port (
                s       :       in bit;
                d0      :       in unsigned (4 downto 0);
                d1      :       in unsigned (4 downto 0);
                y       :       out unsigned (4 downto 0));
end mux_m2;
-----------------------------------------------
architecture a_mux_m2 of mux_m2 is

begin
process (s, d0,d1)
begin

        case  s is
                when '0' =>
                y <=  d0 ;
                when '1'=>
                y  <= d1 ;
        end case;
end process;

end a_mux_m2;

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

```
-------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
-------------------------------------------------
entity mux_m3 is
	port (
		s	:	in	bit;
		d0	:	in	std_logic_vector (31 downto 0);
		d1	:	in	std_logic_vector (31 downto 0);
		y	:	out	std_logic_vector (31 downto 0));
end mux_m3;
-------------------------------------------------
architecture a_mux_m3 of mux_m3 is

begin

process (s, d0, d1)
begin
	case s is
		when '0' =>
		y <= d0;
		when '1' =>
		y <= d1;

	end case;

end process;

end a_mux_m3;

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

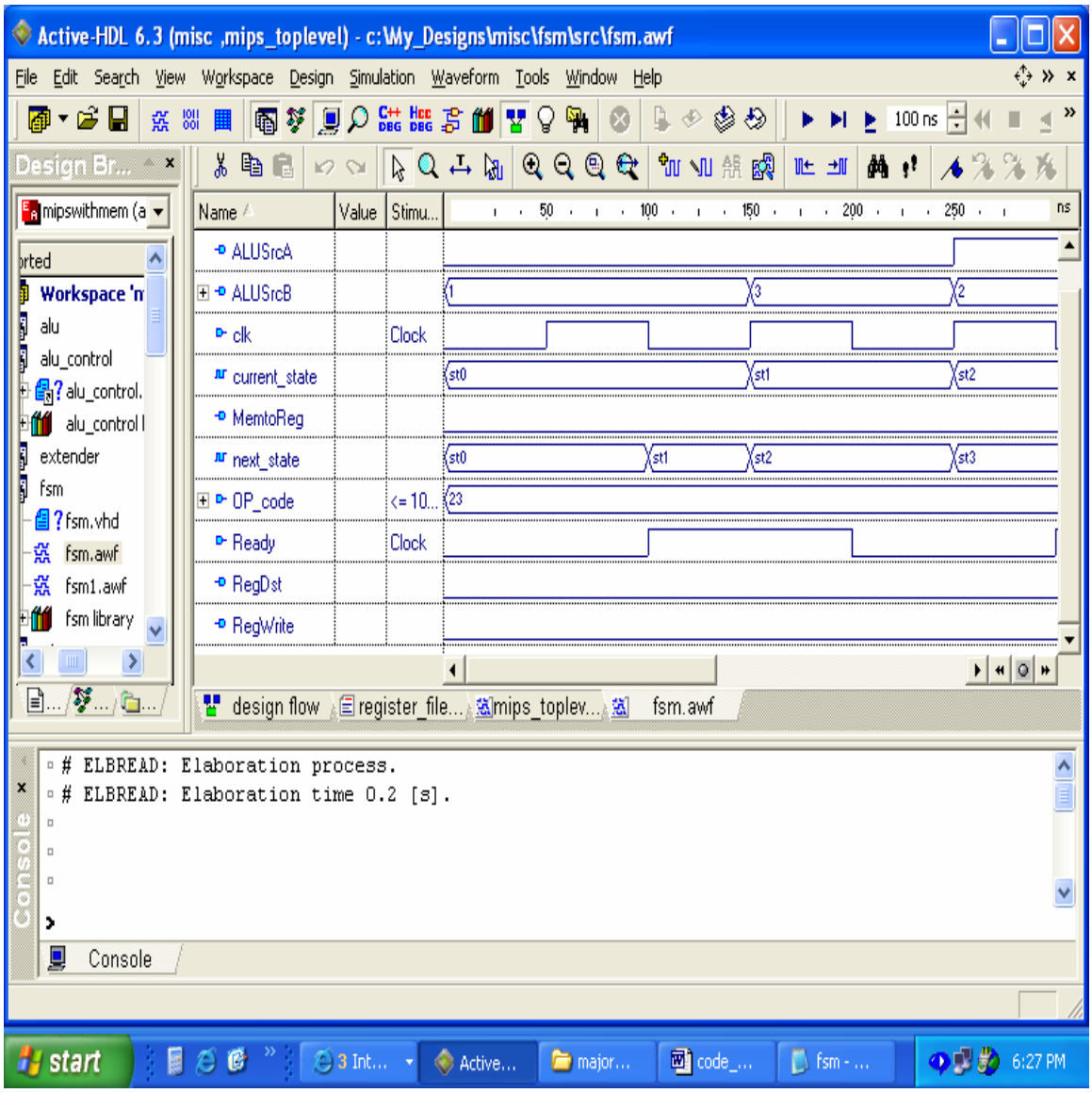
```
-----------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
-----------------------------------------------
entity mux_m4 is
	port (
		s	:	in bit_vector(1 downto 0);
		d0	:	in std_logic_vector (31 downto 0);
		d1	:	in std_logic_vector (31 downto 0);
		d2	:	in std_logic_vector (31 downto 0);
		d3	:	in std_logic_vector (31 downto 0);

		y	:	out std_logic_vector (31 downto 0));
end mux_m4;
-----------------------------------------------
architecture a_mux_m4 of mux_m4 is

begin

process (s, d0, d1,d2,d3)
begin
	case s is
		when "00" =>
		y <= d0;
		when "01" =>
		y <= d1;
		when "10" =>
		y <= d2;
		when "11" =>
		y <= d3;


	end case;

end process;

end a_mux_m4;

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

```
-------------------------------------------------------
-- ***simulation of MIPS Processor  ***********

-- Finite State Machine for MIPS Processor
-------------------------------------------------------
LIBRARY ieee;
USE IEEE.std_logic_1164.ALL;
use IEEE.numeric_bit.all;
-------------------------------------------------
ENTITY fsm    IS
PORT (
clk             : IN    std_logic;
OP_code         : IN    unsigned ( 5 DOWNTO 0 );
Ready           : IN    bit;
RegDst          : OUT   bit;
ALUSrcA         : OUT  bit;
ALUSrcB         : OUT  bit_vector ( 1 DOWNTO 0 );
MemtoReg        : OUT  bit;
IorD            : OUT  bit;
ALUOp           : OUT  bit_vector ( 1 DOWNTO 0 );
PCSource        : OUT  bit_vector ( 1 DOWNTO 0 );
RegWrite        : OUT   bit;
MemRead         : OUT  bit;
MemWrite        : OUT   bit;
IRWrite         : OUT  bit;
PCWrite         : OUT  bit);
END fsm;
-----------------------------------------------------
ARCHITECTURE a_fsm OF  fsm  IS

TYPE state IS (ST0,ST1,ST2,ST3,ST4,ST5,ST6,ST7);
SIGNAL current_state,next_state : state;

BEGIN

Proc1: PROCESS
       BEGIN
       CASE current_state IS

       WHEN ST0 =>

              ALUOp          <= "00";
              RegWrite       <= '0';
              MemRead        <= '1';
              MemWrite       <= '0';
              IRWrite        <= '1';
              PCWrite        <= '1', '0' after 200ns;
              ALUSrcB        <= "01";
              PCSource       <= "00";
              ALUSrcA        <= '0';
```

```
        IorD            <= '0';

        WAIT UNTIL ready'EVENT  and ready = '1';
        next_state <= ST1;

WHEN ST1 =>
        ALUOp           <= "00";
        RegWrite        <= '0';
        MemRead         <= '0';
        MemWrite        <= '0';
        IRWrite         <= '0';
        PCWrite         <= '0';
        ALUSrcB         <= "11";
        ALUSrcA         <= '0';
        if OP_code = "000000" then
                next_state <= ST6;
        else if OP_code = "100011"
             or OP_code = "101011" then    --LW(35) or SW(43)

                next_state <= ST2;
        end if;
        end if;


WHEN ST2 =>

        ALUOp           <= "00";
        RegWrite        <= '0';
        MemRead         <= '0';
        MemWrite        <= '0';
        IRWrite         <= '0';
        PCWrite         <= '0';

        ALUSrcB         <= "10";
        ALUSrcA         <= '1';
        if OP_code = "100011" then          --LW(35)
                next_state <= ST3;
        else
                next_state <= ST5;  --SW(43)

        end if;

WHEN ST3 =>

        ALUOp           <= "00";
        RegWrite        <= '0';
        MemRead         <= '1';
        MemWrite        <= '0';
        IRWrite         <= '0';
        PCWrite         <= '0';

        IorD            <= '1';
```

```
        WAIT UNTIL ready'EVENT  and ready = '1';

                next_state <= ST4;

WHEN ST4 =>

        ALUOp       <= "00";
        RegWrite    <= '1';
        MemRead     <= '0';
        MemWrite    <= '0';
        IRWrite     <= '0';
        PCWrite     <= '0';
        MemtoReg    <= '1';
        RegDst      <= '0';

                next_state    <= ST0;

WHEN ST5 =>

        ALUOp       <= "00";
        RegWrite    <= '0';
        MemRead     <= '0';
        MemWrite    <= '1';
        IRWrit      <= '0';
        PCWrit      <= '0';
        IorD        <= '1';

        WAIT UNTIL ready'EVENT  and ready = '1';

                next_state    <= ST0;

WHEN ST6 =>

        ALUOp       <= "10";
        RegWrite    <= '0';
        MemRead     <= '0';
        MemWrite    <= '0';
        IRWrite     <= '0';
        PCWrite     <= '0';

        ALUSrcA     <= '1';
        ALUSrcB     <= "00";

        next_state <= ST7;

WHEN ST7 =>
        ALUOp       <= "00"       ;
        RegWrite    <= '1';
        MemRead     <= '0';
        MemWrite    <= '0';
        IRWrite     <= '0';
        PCWrite     <= '0';
        RegDst      <= '1';
```

```
                MemtoReg     <= '0';
                        next_state <= ST0;
END CASE;
WAIT ON current_state;
END PROCESS;

Proc2: PROCESS (clk)
                BEGIN
                        IF clk'event AND clk =  '1' THEN
                                current_state <= next_state ;
                        END IF;
                END PROCESS;

        END ARCHITECTURE  a_fsm;
-------------------------------------------------------
```


Active-HDL 6.3 (misc ,mips_toplevel) - c:\My_Designs\misc\fsm\src\fsm.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Design Br...
mipswithmem (a
Workspace 'n
alu
alu_control
alu_control.
alu_control
extender
fsm
fsm.vhd
fsm.awf
fsm1.awf
fsm library
Name
Value
Stimu...
50
100
150
200
250
ns
ALUSrcA
ALUSrcB
clk
current_state
MemtoReg
next_state
OP_code
Ready
RegDst
RegWrite
Clock
st0
st1
st2
st3
design flow
register_file...
mips_toplev...
fsm.awf
ELBREAD: Elaboration process.
ELBREAD: Elaboration time 0.2 [s].
Console
start
3 Int...
Active...
major...
code_...
fsm - ...
6:27 PM

```
-------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_bit.all;
---------------------------------------
entity mips_toplevel is
        port(
                ready         : in bit;
                clk           : in std_logic;
                reset         : in bit;
                MemRead       : out bit;
                MemWrite      : out bit;
                DataFetch     : out bit;
                Address       : out unsigned (31 downto 0);
                data_in       : in std_logic_vector(31 downto 0);
                data_out      : out std_logic_vector(31 downto 0)
           );
end mips_toplevel;
-----------------------------------------------

architecture mips_toplevel of mips_toplevel is

        component instreg is

        port(
               A        : in std_logic_vector(31 downto 0);
               clk      : in std_logic;
               IRWrite : in     bit;
               IR       : out  unsigned(31 downto 0));

        end  component instreg;

        component  register_file is
        port(
                write_en : in bit;
                rd_reg1  : in unsigned (4 downto 0);
                rd_reg2  : in unsigned (4 downto 0);
                wr_reg   : in unsigned (4 downto 0);
                data     : in std_logic_vector(31 downto 0);
                reg1     : out std_logic_vector(31 downto 0);
                reg2     : out std_logic_vector(31 downto 0)
           );
        end component       register_file;


        component reg_32bit is

        port(
                        a       : in std_logic_vector(31 downto 0);
                        clk     : in std_logic;
                        y       : out std_logic_vector(31 downto 0));

        end component reg_32bit;
```

```
component reg_32we is

        port(
                a          :     in std_logic_vector(31 downto 0);
                clk        :     in std_logic;
                write_en   :     in bit;
                reset      :     in bit;
                y          :     out std_logic_vector (31 downto 0));

        end component reg_32we;


        component fsm       is
        port (
                clk             : in     std_logic;
                OP_Code         : in     unsigned ( 5 downto 0 );
                ready           : in    bit;
                RegDst          : out    bit;
                ALUSrcA         : out    bit;
                ALUSrcB         : out    bit_vector ( 1 downto 0 );
                MemtoReg        : out    bit;
                IorD            : out    bit;
                PCSource        : out    bit_vector ( 1 downto 0 );
                ALUOp           : out    bit_vector ( 1 downto 0 );
                RegWrite        : out    bit;
                MemRead         : out    bit;
                MemWrite        : out    bit;
                IRWrite         : out    bit;
                PCWrite         : out    bit);

        end component fsm;

        component extender is
         port(
                 ext_in : in unsigned (15 downto 0);
                 ext_out : out std_logic_vector(31 downto 0)
            );
        end component extender;


        component alu_control is
        port(
                alu_op          :        in bit_vector (1 downto 0);
                func            :        in unsigned (5 downto 0);
                alu_control     :        out bit_vector (2 downto 0));
        end component alu_control;


        component alu is
        port (
```

```
        a               : in    std_logic_vector ( 31 downto 0 );
        b               : in    std_logic_vector ( 31 downto 0 );
        alu_control     : in    bit_vector ( 2 downto 0 );
        alu_result      : out   std_logic_vector ( 31 downto 0 ));
    end component alu;



    component  mux_m1 is
                port (
                s       :       in bit;
                d0      :       in std_logic_vector (31 downto 0);
                d1      :       in std_logic_vector (31 downto 0);
                y       :       out unsigned (31 downto 0));
    end  component mux_m1;


    component mux_m2 is
        port (
                s       :       in bit;
                d0      :       in unsigned (4 downto 0);
                d1      :       in unsigned (4 downto 0);
                y       :       out unsigned (4 downto 0));
    end component mux_m2;

    component mux_m3 is
        port (
                s       :       in bit;
                d0      :       in std_logic_vector (31 downto 0);
                d1      :       in std_logic_vector (31 downto 0);
                y       :       out std_logic_vector (31 downto 0));
    end component mux_m3;

    component mux_m4 is
        port (
                s       :       in bit_vector (1 downto 0);
                d0      :       in std_logic_vector (31 downto 0);
                d1      :       in std_logic_vector (31 downto 0);
                d2      :       in std_logic_vector (31 downto 0);
                d3      :       in std_logic_vector (31 downto 0);
                y       :       out std_logic_vector (31 downto 0));
    end component mux_m4;

signal ir_out : unsigned(31 downto 0);
signal mux2_out : unsigned(4 downto 0);
signal mux3_out : std_logic_vector(31 downto 0);
signal mux6_out : std_logic_vector(31 downto 0);
signal mdr_out :  std_logic_vector(31 downto 0);
signal extender_out :  std_logic_vector(31 downto 0);
signal regfile_out1 :  std_logic_vector(31 downto 0);
signal regfile_out2 :  std_logic_vector(31 downto 0);
signal rega_out :  std_logic_vector(31 downto 0);
signal regb_out :  std_logic_vector(31 downto 0);
```

```
signal pc_out :  std_logic_vector(31 downto 0) ;
signal alu_in1 :  std_logic_vector(31 downto 0);
signal alu_in2 :  std_logic_vector(31 downto 0);
signal alu_out :  std_logic_vector(31 downto 0);
signal regalu_out :  std_logic_vector(31 downto 0);
signal fsm_pcwrite : bit;
signal fsm_iord : bit;
signal fsm_memtoreg : bit;
signal fsm_irwrite : bit;
signal fsm_alusrca : bit;
signal fsm_regwrite : bit;
signal fsm_regdst : bit;
signal fsm_aluop : bit_vector(1 downto 0);
signal fsm_alusrcb : bit_vector(1 downto 0);
signal fsm_pcsource : bit_vector(1 downto 0);
signal alu_control_out : bit_vector(2 downto 0);
signal mem_ready : bit;


begin

ir:     component instreg
        port map (
        a               => data_in,
        clk             => clk,
        irwrite => fsm_irwrite,
        ir              => ir_out);


mdr:component reg_32bit
        port map (
        a       => data_in,
        clk     => clk,
        y       => mdr_out);

m2: component mux_m2
        port map(
                        s       =>      fsm_regdst,
                        d0      =>      ir_out(20 downto 16),
                        d1      =>      ir_out(15 downto 11),
                        y       =>      mux2_out);

m3:     component mux_m3
        port map(
                        s       =>      fsm_memtoreg,
                        d0      =>      regalu_out,
                        d1      =>      mdr_out,
                        y       =>      mux3_out);

rf:     component  register_file
        port map(
                write_en        => fsm_regwrite,
                rd_reg1         => ir_out(25 downto 21),
```

```
                rd_reg2      => ir_out(20 downto 16),
                wr_reg       => mux2_out,
                data         => mux3_out,
                reg1         => regfile_out1,
                reg2         => regfile_out2 );

ext:    component extender
                port map (
                ext_in       => ir_out(15 downto 0),
                ext_out              => extender_out);

rega:   component reg_32bit
                port map (
                a       => regfile_out1,
                clk     => clk,
                y       => rega_out);

regb:   component reg_32bit
                port map (
                a       => regfile_out2,
                clk     => clk,
                y       => regb_out);

data_out <= regb_out;

m4:             component mux_m4
                port map (
                        s       => fsm_alusrcb,
                        d0      => regb_out,
                        d1      => x"0000_0004",
                        d2      => extender_out,
                        d3      => x"0000_0000",
                        y       => alu_in2 );

m5:     component mux_m3
                port map (
                s       => fsm_alusrca,
                d0      => pc_out,
                d1      => rega_out,
                y       => alu_in1);


alc:    component alu_control
                port map(
                alu_op       => fsm_aluop,
                func         => ir_out (5 downto 0),
                alu_control  => alu_control_out );

al:             component alu
                port map (
                a                    => alu_in1,
                b                    => alu_in2,
                alu_control  => alu_control_out,
```

```
            alu_result     => alu_out  );

regal:  component reg_32bit
            port map (
            a       => alu_out,
            clk     => clk,
            y       => regalu_out);

m6:         component mux_m4
            port map (
                  s       => fsm_pcsource,
                  d0      => alu_out,
                  d1      => regalu_out,
                  d2      => x"0000_0000",
                  d3      => x"0000_0000",
                  y       => mux6_out );

pc:         component reg_32we
            port map (
            a                 => mux6_out,
            clk               => clk,
            reset       => reset,
            write_en    => fsm_pcwrite,
            y                 => pc_out);


m1:         component mux_m1
            port map (
            s       => fsm_iord,
            d0      => pc_out,
            d1      => regalu_out,
            y       => address);

fs:         component fsm
            port map (
            clk               => clk,
            op_code           => ir_out(31 downto 26),
            ready             => ready,
            regdst            => fsm_regdst,
            alusrca           => fsm_alusrca,
            alusrcb           => fsm_alusrcb,
            memtoreg          => fsm_memtoreg,
            iord              => fsm_iord,
            pcsource          => fsm_pcsource,
            aluop             => fsm_aluop,
            regwrite          => fsm_regwrite,
            memread           => memread,
            memwrite          => memwrite,
            irwrite           => fsm_irwrite,
            pcwrite           => fsm_pcwrite);

      datafetch <= fsm_iord;
```

```
        end mips_toplevel;
-------------------------------------------------
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_bit.all;
------------------------------------------------------------
entity mips_mem is

port   (
address        : in unsigned ( 31 downto 0);
clk            : in std_logic;
data_in        : in std_logic_vector (31 downto 0);
data_out       : out std_logic_vector (31 downto 0);
burst          : in bit;
ready          : out bit;
MemWrite       : in bit;
MemRead        : in bit);
end entity mips_mem;

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

architecture a_mips_mem  of mips_mem is


constant mem_size : natural := 65536;
constant access_time  : delay_length := 200 ns;
constant access_time_burst  : delay_length := 50 ns;
constant propagation_delay : delay_length := 2 ns;

begin

ram1:  process is

constant high_address : natural := mem_size - 1;
type mem_array is array (natural range <>) of unsigned (31 downto 0);
variable ram : mem_array ( 0 to high_address/4)
                      := (others => X"0000_0000");
variable address_byte, address_word : natural;
variable write_enable : boolean;

procedure program_load_proc is

begin
        ram(0)  := "10001100110001110000000000101100";
        ram(1)  := "10001100110001110000000000110000";
        ram(2)  := "10001100110001110000000000110100";
        ram(3)  := "10001100110001110000000000111000";
        ram(4)  := "10001100110001110000000000111100";
        ram(5)  := "10001100110001110000000001000000";
        ram(6)  := "10001100110001110000000001000100";
        ram(7)  := "10001100110001110000000001001000";
        ram(8)  := "10001100110001110000000001001100";
        ram(9)  := "10001100110001110000000001010000";
        ram(11) := "11111111111111111111111111111111";
        ram(12) := "00000000000000000000000000000001";
```

```
        ram(13) := "00000000000000000000000000000011";
        ram(14) := "00000000000000000000000000000111";
        ram(15) := "00000000000000000000000000011111";
        ram(16) := "00000000000000000000000111111111";
        ram(17) := "00000000000000000111111111111111";
        ram(18) := "00000000000000011111111111111111";
        ram(19) := "00000000000000000000000000000001";

end program_load_proc;

procedure write_cycle_proc is
begin
ram(address_word) := unsigned (to_bitvector(data_in));
end write_cycle_proc;

procedure read_cycle_proc is
begin
data_out <= to_X01(bit_vector(ram(address_word)));
end read_cycle_proc;

begin

program_load_proc;

ready <= '0'after propagation_delay ;

loop

wait on clk until rising_edge(clk)and (MemWrite= '1' or MemRead ='1');

address_byte := to_integer(address);
write_enable := MemWrite ='1';
if address_byte <= high_address then
        address_word := address_byte/4;

        if write_enable then
                write_cycle_proc;
                wait for access_time;

        else
                wait for access_time;
                read_cycle_proc;
        end if;
        wait until rising_edge(clk);
        ready <= '1' after propagation_delay;
        wait until rising_edge(clk);
        ready <= '0' after propagation_delay;

        while burst = '1' loop
                address_word := address_word + 1;
                wait until rising_edge(clk);
                if write_enable then
                        write_cycle_proc;
```

```
                wait for access_time_burst;
            else
                wait for access_time_burst;
                read_cycle_proc;

            end if;
            wait until rising_edge(clk);
            ready <= '1' after propagation_delay;
            wait until rising_edge(clk);
            ready <= '0' after propagation_delay;
            end loop;

        end if;
        end loop;
end process ram1;

end a_mips_mem;
-------------------------------------------------
```

```
-----------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_bit.ALL;
-----------------------------------------------
entity mipswithmem is
        port(
        clk     : in    STD_LOGIC;
        reset   : in    bit;
        result  : OUT  STD_LOGIC_VECTOR(31 downto 0)
        );

end entity mipswithmem;
-------------------------------------------------

architecture a_mipswithmem of  mipswithmem is


 COMPONENT  mips_toplevel is
         port(
        ready : in bit;
        clk : in STD_LOGIC;
        reset : in bit;
        MemRead : out bit;
        MemWrite : out bit;
        address : out unsigned (31 downto 0);
        data_in : in STD_LOGIC_VECTOR(31 downto 0);
        data_out : out STD_LOGIC_VECTOR(31 downto 0)
            );
end  COMPONENT mips_toplevel;

COMPONENT mips_mem is

        port   (
        address : in unsigned ( 31 downto 0);
        clk             : in std_logic;
        data_in         : in std_logic_vector (31 downto 0);
        data_out: out std_logic_vector (31 downto 0);
        burst   : in bit;
        ready   : out bit;
        MemWrite: in bit;
        MemRead : in bit);
end COMPONENT mips_mem;

signal mem_ready_signal, write_signal, read_signal : bit;
signal address_signal : unsigned ( 31 downto 0);
signal data_in_signal, data_out_signal : std_logic_vector (31 downto 0);

begin


CPU:  component mips_toplevel
         port map (
```

```
            ready       => mem_ready_signal,
            clk         => clk,
            reset       => reset,
            MemRead     => read_signal,
            MemWrite    => write_signal,
            address     => address_signal,
            data_in     => data_in_signal,
            data_out    => data_out_signal
        );


mem: component mips_mem

    port map   (
            address     => address_signal,
            clk         => clk,
            data_in     => data_out_signal,
            data_out    => data_in_signal,
            burst       => '0' ,
            ready       => mem_ready_signal,
            MemWrite    => write_signal,
            MemRead     => read_signal      );


end  architecture a_mipswithmem;
-------------------------------------------------------
```

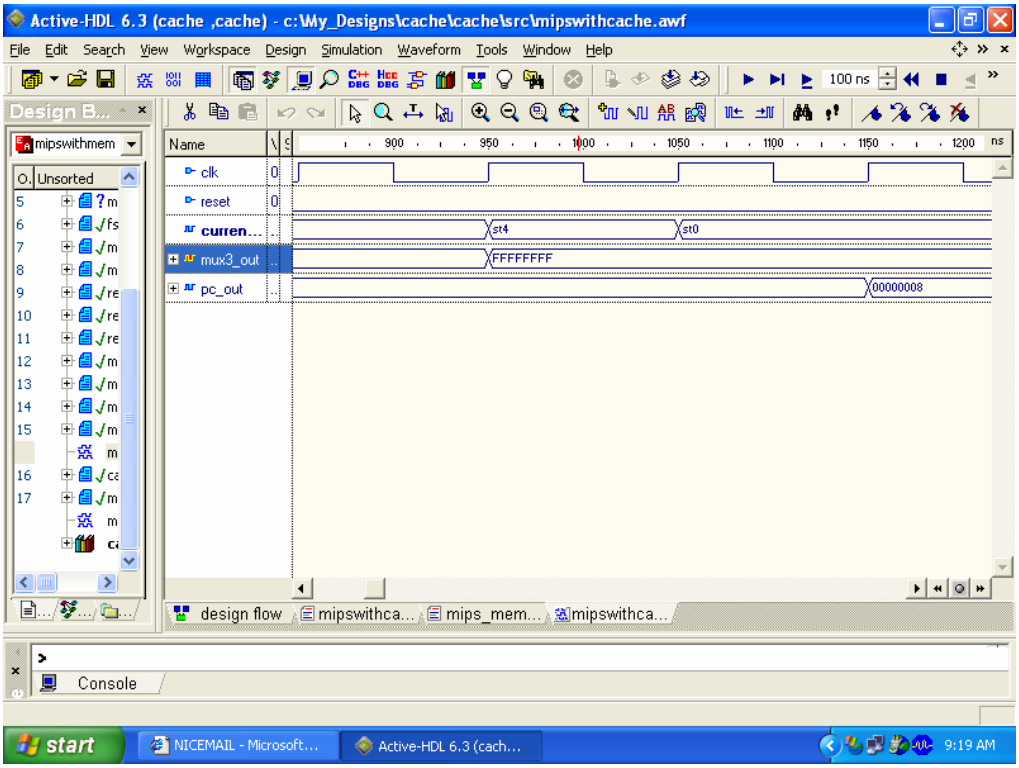
```
-----------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_bit.all;
-----------------------------------------------
entity cache_mem is
        port(
                DataFetch : in bit;
                MemRd_in : in bit;
                Ready_in : in bit;
                Ready_out : out bit;
                clk : in std_logic;
                ad_in : in unsigned(31 downto 0);
                data_in : in STD_LOGIC_VECTOR(31 downto 0);
                burst : out bit;
                MemRd_out : out bit;
                data_out : out STD_LOGIC_VECTOR(31 downto 0);
                ad_out : out unsigned(31 downto 0)
           );
end cache_mem;

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

architecture cache_mem of cache_mem is
begin
process is
        type cache_array is array (natural range <>) of STD_LOGIC_VECTOR (31
downto 0);
        variable cache_data : cache_array ( 0 to 9):= (others => X"0000_0000");
        type address_array is array (natural range <>) of unsigned (31 downto 0);
        variable cache_address : address_array  ( 0 to 9):= (others =>
X"0000_0000");

        variable cache_valid_data : natural :=0;
        variable cache_index : natural;
        variable cache_access : boolean;

        procedure cache_load is
        variable address : unsigned (31 downto 0);

        begin
                        MemRd_out <= MemRd_in;
                        ad_out <= ad_in;
                        address := ad_in;
                        burst <= '1';
                        Ready_out <= '0';
                        for i in 1 to 9 loop
                                wait until rising_edge(clk) and Ready_in = '1';
                                cache_data(i) := data_in;
                                cache_address(i) := address;
                                address := address + 4;
                                ad_out <= address;
                        end loop;
```

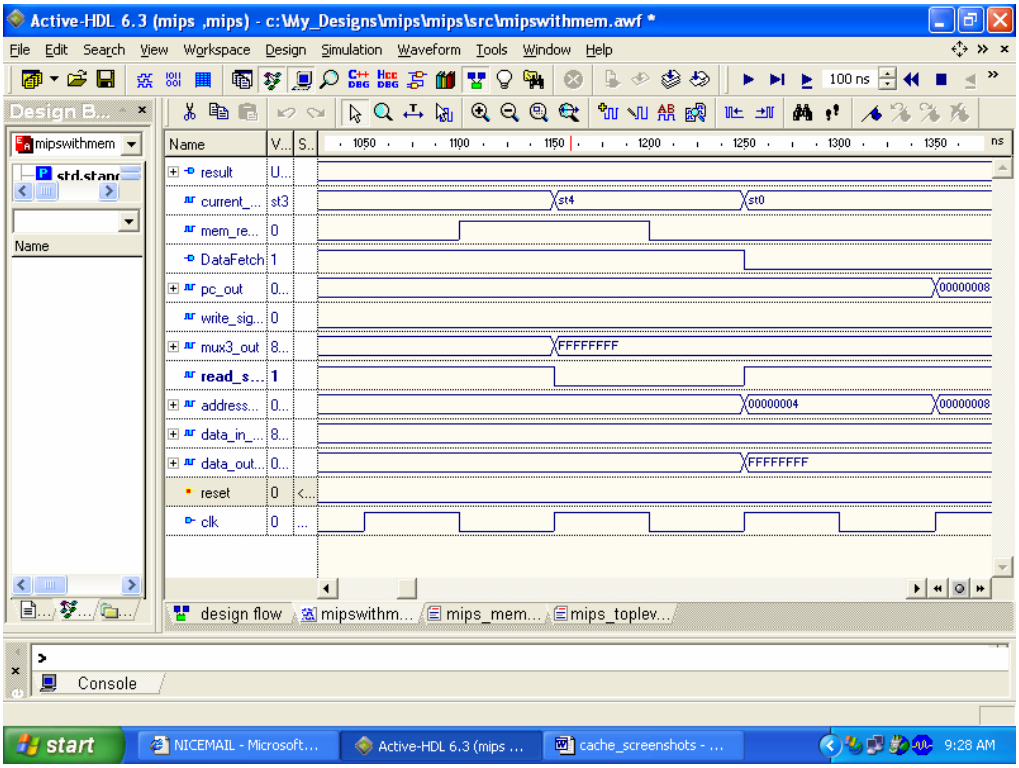
```
                cache_valid_data := 1;
                burst <= '0';
                data_out <= cache_data(1);
                wait until rising_edge(clk);
                Ready_out <= '1';
                wait until rising_edge(clk);
                Ready_out <= '0';

        end procedure cache_load;


begin
        Ready_out <= '0';
        loop
                wait until rising_edge(clk) and MemRd_in ='1';
                cache_access := DataFetch = '1' and MemRd_in = '1';
                if cache_access         then
                        if cache_valid_data = 0 then
                                cache_load;
                        else
                                for i in 0 to 9 loop
                                        if cache_address(i) = ad_in  then
                                                cache_index := i;
                                        end if;
                                end loop;
                                data_out <= cache_data(cache_index);
                                wait until rising_edge(clk);
                                Ready_out <= '1';
                                wait until rising_edge(clk);
                                Ready_out <= '0';
                        end if;
                else
                        ad_out <= ad_in;
                        MemRd_out <= MemRd_in;
                        Ready_out <= Ready_in;
                        data_out <= data_in;
                        burst <= '0';
                end if;
        end loop;

end process;



end cache_mem;
```

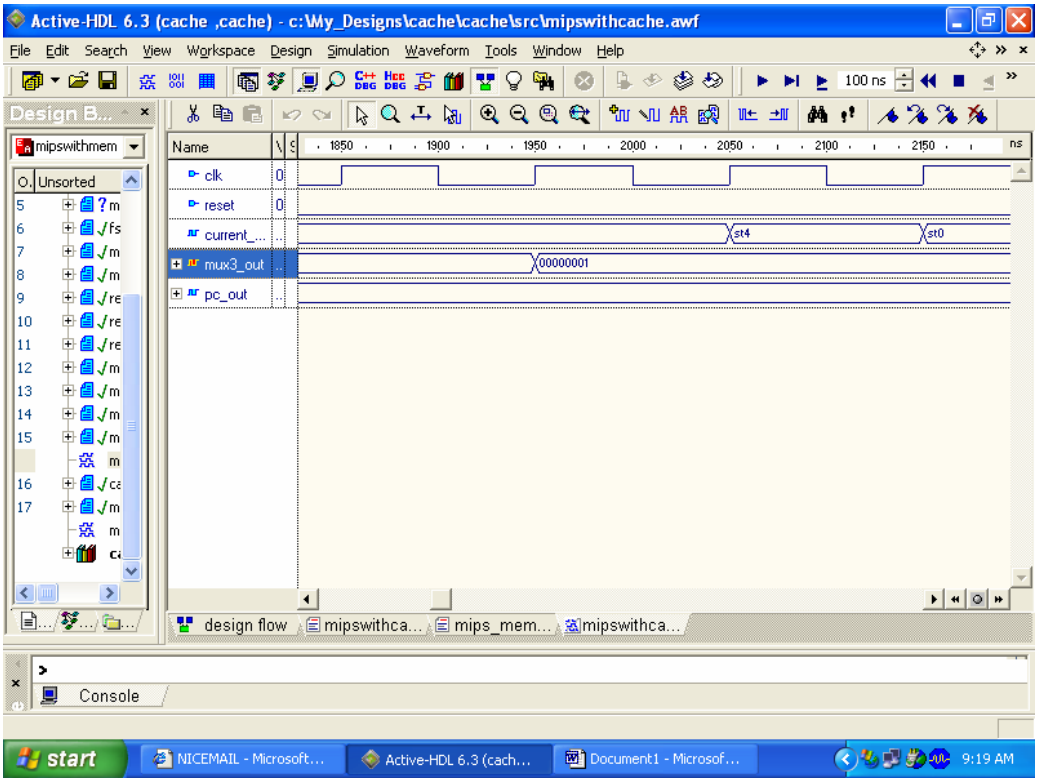
```
----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_bit.ALL;
----------------------------------------
entity mipswithcache is
        port(
        clk     : in    STD_LOGIC;
        reset   : in    bit;
        result  : OUT  STD_LOGIC_VECTOR(31 downto 0)
        );

end mipswithcache;

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

architecture mipswithcache of mipswithcache is

component  mips_toplevel is
         port(
                ready : in bit;
                clk : in STD_LOGIC;
                reset : in bit;
                MemRead : out bit;
                MemWrite : out bit;
                DataFetch : out bit;
                address : out unsigned (31 downto 0);
                data_in : in STD_LOGIC_VECTOR(31 downto 0);
                data_out : out STD_LOGIC_VECTOR(31 downto 0)
           );
end component  mips_toplevel;

component   mips_mem is

        port   (address         : in unsigned ( 31 downto 0);
                clk             : in std_logic;
                data_in         : in std_logic_vector (31 downto 0);
                data_out        : out std_logic_vector (31 downto 0);
                burst           : in bit;
                ready           : out bit;
                MemWrite        : in bit;
                MemRead         : in bit);
end component  mips_mem;

component cache_mem is
         port(
                DataFetch     : in bit;
                MemRd_in      : in bit;
                Ready_in      : in bit;
                Ready_out     : out bit;
                clk           : in std_logic;
                ad_in         : in unsigned(31 downto 0);
                data_in       : in STD_LOGIC_VECTOR(31 downto 0);
```

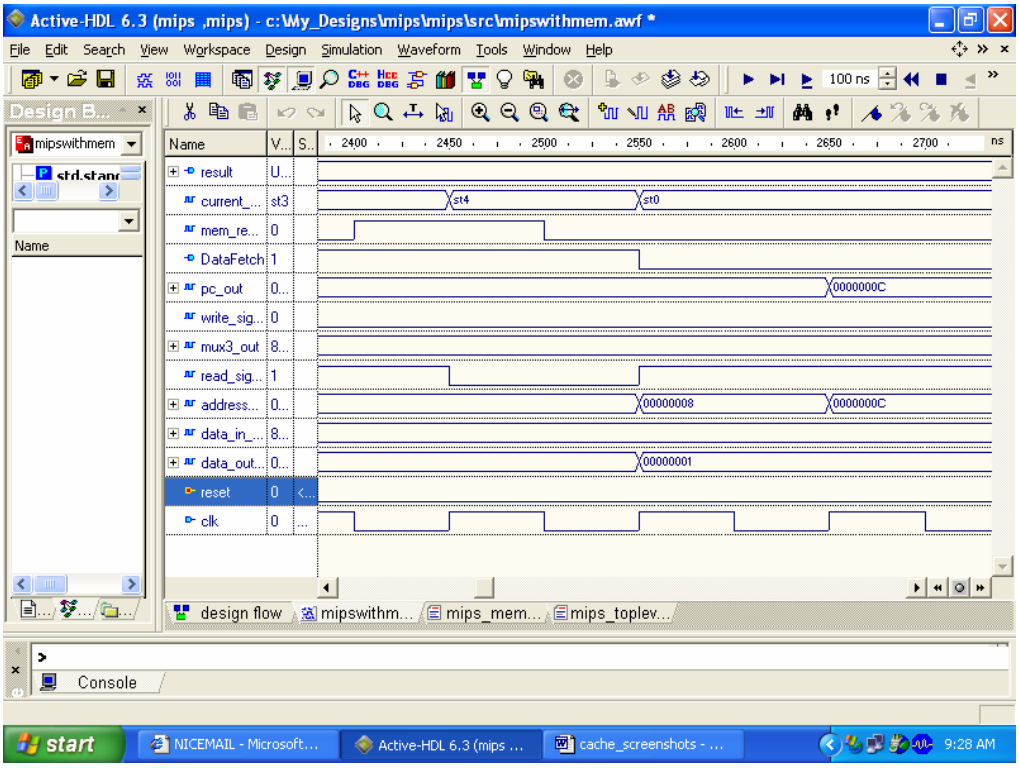
```
            burst       : out bit;
            MemRd_out   : out bit;
            data_out    : out STD_LOGIC_VECTOR(31 downto 0);
            ad_out      : out unsigned(31 downto 0)
        );
end component cache_mem;

signal cpu_data_out : STD_LOGIC_VECTOR(31 downto 0);
signal cpu_MemWrite_out : bit;
signal cpu_address_out : unsigned(31 downto 0);
signal cpu_MemRead_out : bit;
signal cpu_Ready_in : bit;
signal cpu_data_in : STD_LOGIC_VECTOR(31 downto 0);
signal cpu_DataFetch : bit;
signal cache_ad_out : unsigned(31 downto 0);
signal cache_MemRd_out : bit;
signal cache_Ready_in : bit;
signal cache_data_in : STD_LOGIC_VECTOR(31 downto 0);
signal cache_burst_out : bit;


begin

cpu:   component  mips_toplevel
            port map(
            ready       => cpu_Ready_in,
            clk         => clk ,
            reset       => reset,
            MemRead     => cpu_MemRead_out,
            MemWrite    => cpu_MemWrite_out,
            DataFetch   => cpu_DataFetch,
            address     => cpu_address_out,
            data_in     => cpu_data_in,
            data_out    => cpu_data_out
        );


mem:  component   mips_mem
            port map (
            address     => cache_ad_out,
            clk         => clk,
            data_in     => cpu_data_out,
            data_out    => cache_data_in,
            burst       => cache_burst_out,
            ready       => cache_Ready_in ,
            MemWrite    => cpu_MemWrite_out,
            MemRead     => cache_MemRd_out);


cac:   component cache_mem
            port map(
             DataFetch  => cpu_DataFetch,
             MemRd_in   => cpu_MemRead_out,
```

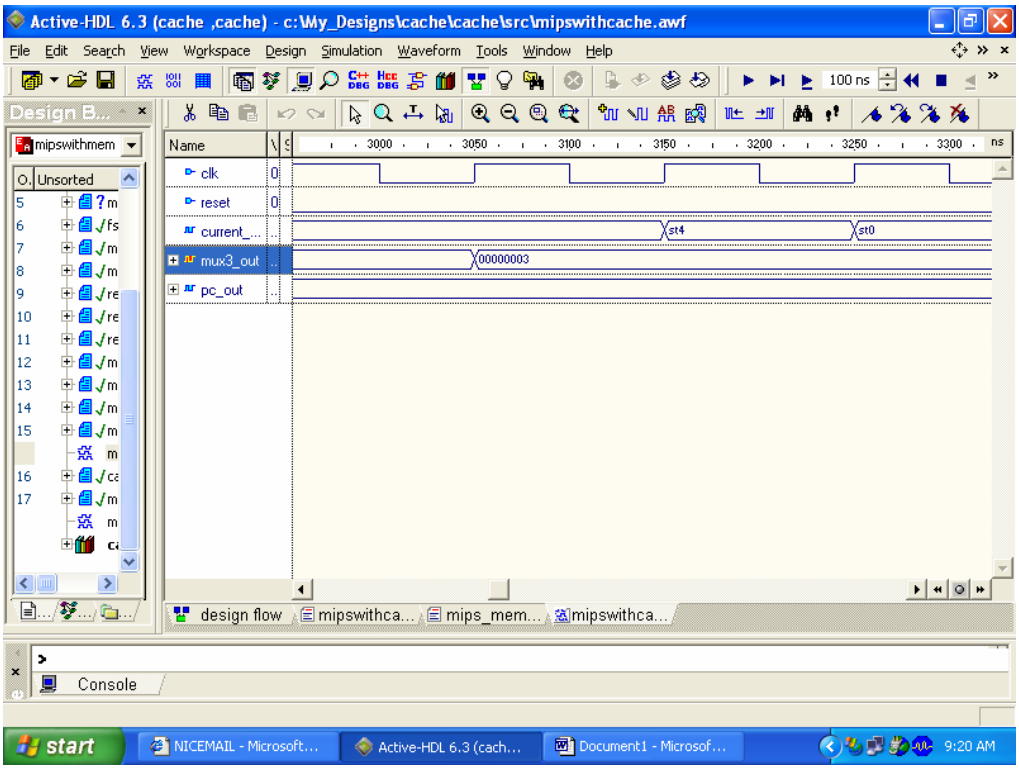
```
            Ready_in     => cache_Ready_in,
            Ready_out    => cpu_Ready_in,
            clk          => clk,
            ad_in        => cpu_address_out,
            data_in      =>cache_data_in,
            burst        => cache_burst_out,
            MemRd_out => cache_MemRd_out,
            data_out     => cpu_data_in,
            ad_out       => cache_ad_out
        );

end mipswithcache;
```

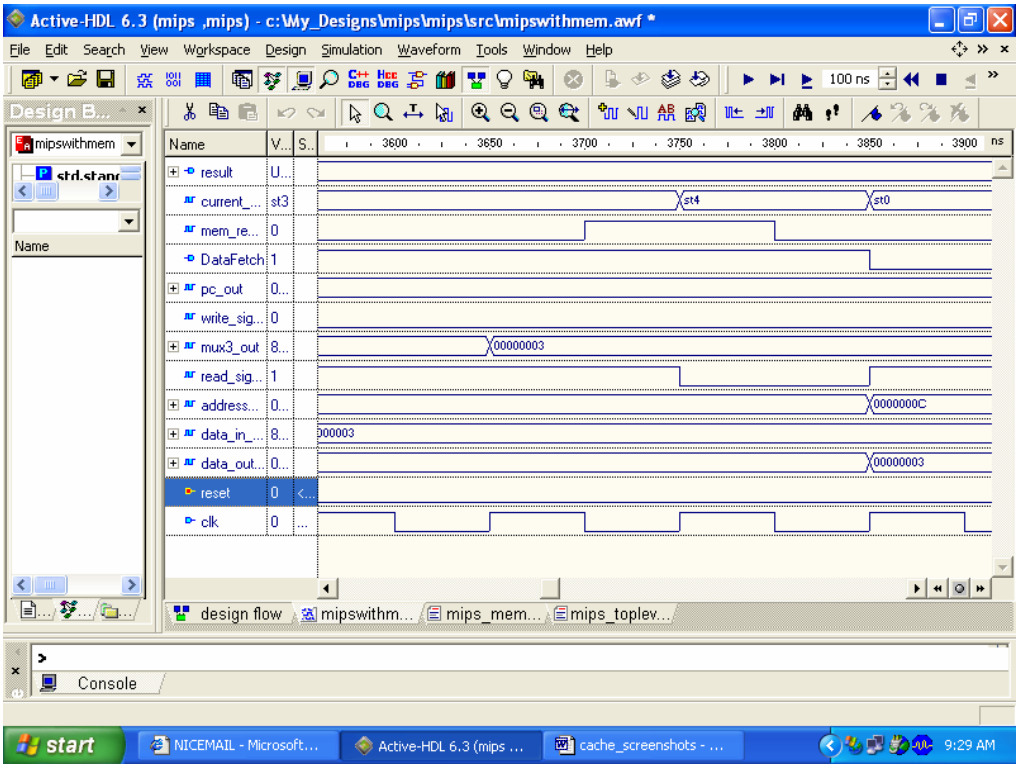

Active-HDL 6.3 (cache ,cache) - c:\My_Designs\cache\cache\src\mipswithcache.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Design B...
mipswithmem
Name
clk
reset
curren...
mux3_out
pc_out
st4
st0
FFFFFFFF
00000008
design flow
mipswithca...
mips_mem...
mipswithca...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (cach...
9:19 AM

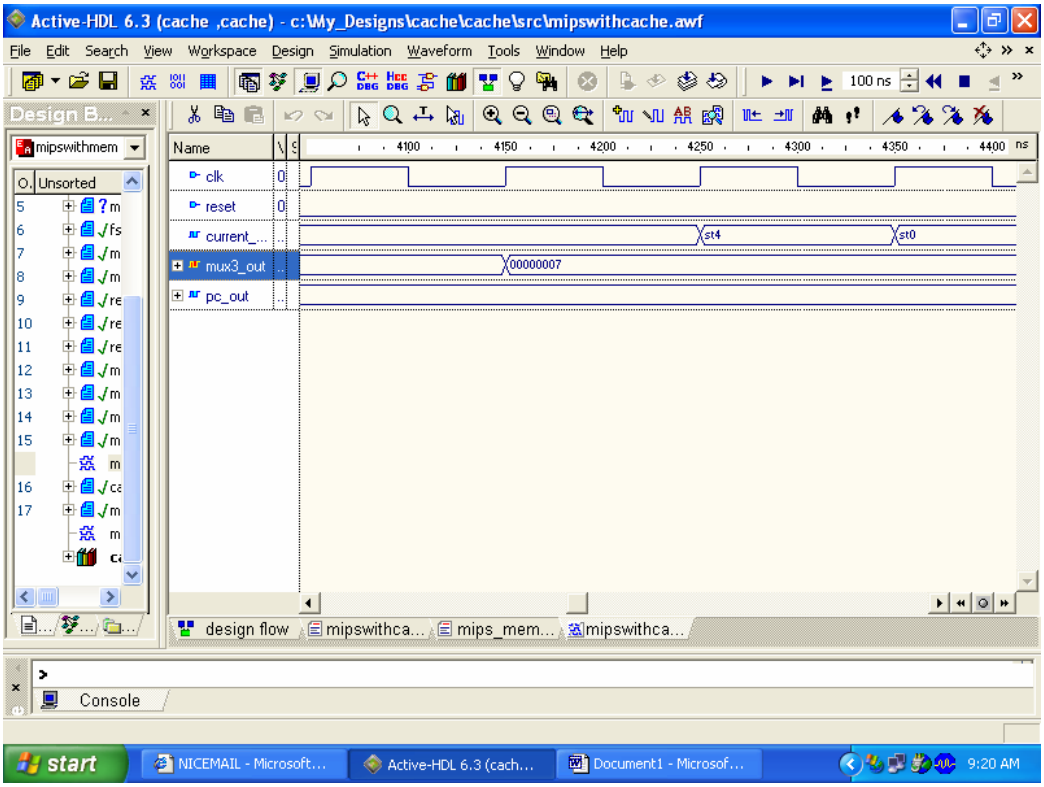

Active-HDL 6.3 (mips ,mips) - c:\My_Designs\mips\mips\src\mipswithmem.awf *
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Design B...
mipswithmem
Name
result
current_...
mem_re...
DataFetch
pc_out
write_sig...
mux3_out
read_s...
address...
data_in_...
data_out...
reset
clk
st4
st0
00000004
00000008
FFFFFFFF
design flow
mipswithm...
mips_mem...
mips_toplev...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (mips ...
cache_screenshots - ...
9:28 AM

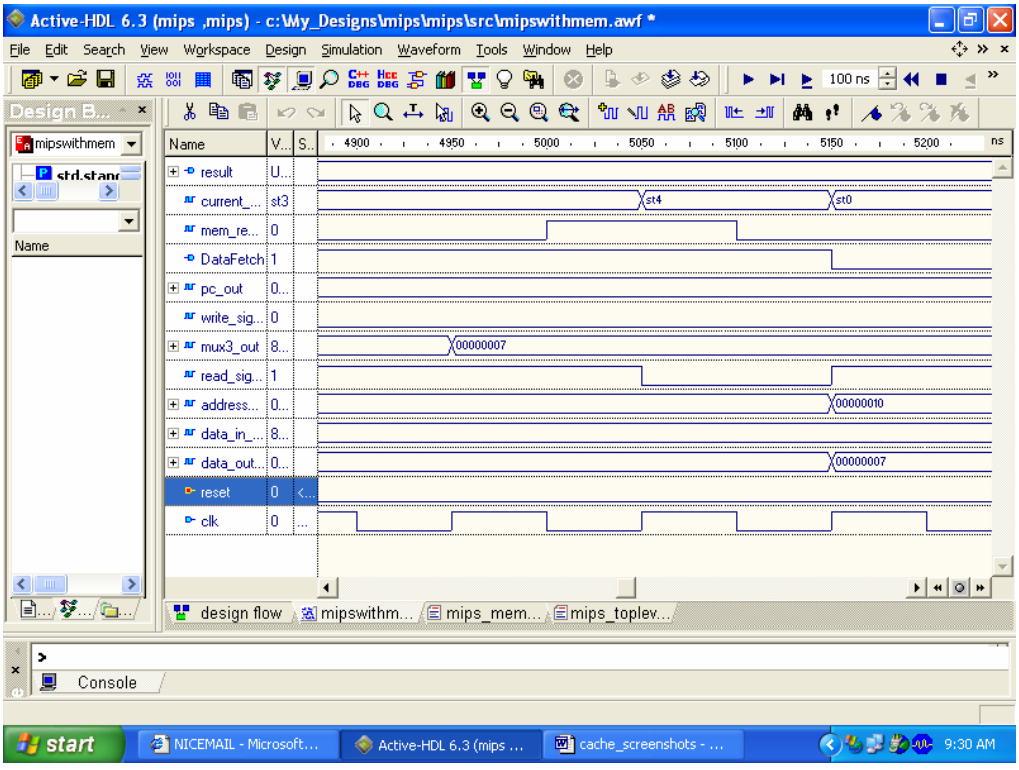

Active-HDL 6.3 (cache ,cache) - c:\My_Designs\cache\cache\src\mipswithcache.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Design B...
mipswithmem
Unsorted
Name
clk
reset
current_...
mux3_out
pc_out
st4
st0
00000001
design flow
mipswithca...
mips_mem...
mipswithca...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (cach...
Document1 - Microsof...
9:19 AM

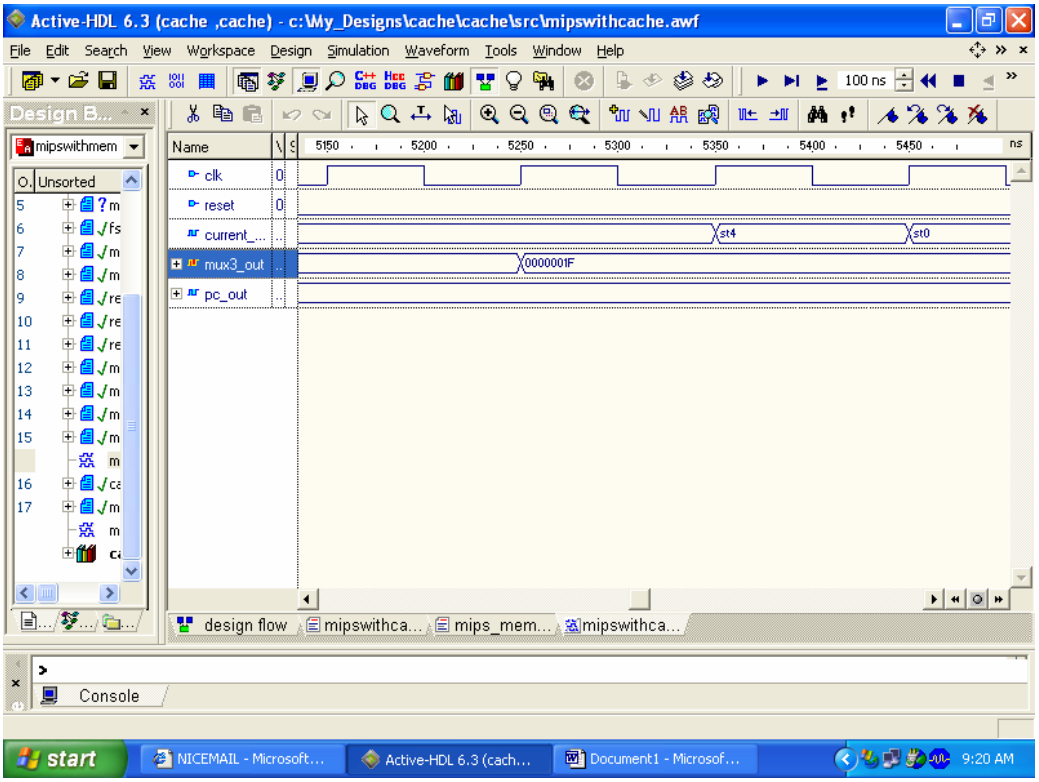

Active-HDL 6.3 (mips ,mips) - c:\My_Designs\mips\mips\src\mipswithmem.awf *
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Design B...
mipswithmem
Name
result
current_...
mem_re...
DataFetch
pc_out
write_sig...
mux3_out
read_sig...
address...
data_in_...
data_out...
reset
clk
st4
st0
0000000C
00000008
00000001
design flow
mipswithm...
mips_mem...
mips_toplev...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (mips ...
cache_screenshots - ...
9:28 AM

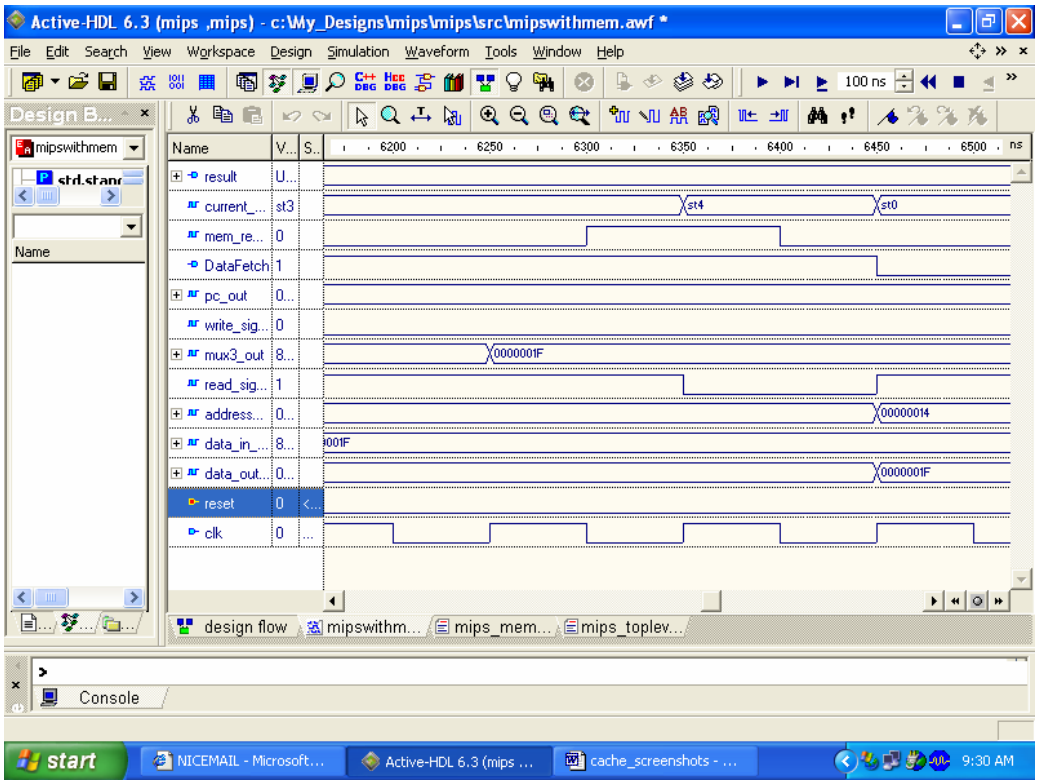

Active-HDL 6.3 (cache ,cache) - c:\My_Designs\cache\cache\src\mipswithcache.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Design B...
mipswithmem
Name
clk
reset
current_...
mux3_out
pc_out
00000003
design flow
mipswithca...
mips_mem...
mipswithca...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (cach...
Document1 - Microsof...
9:20 AM

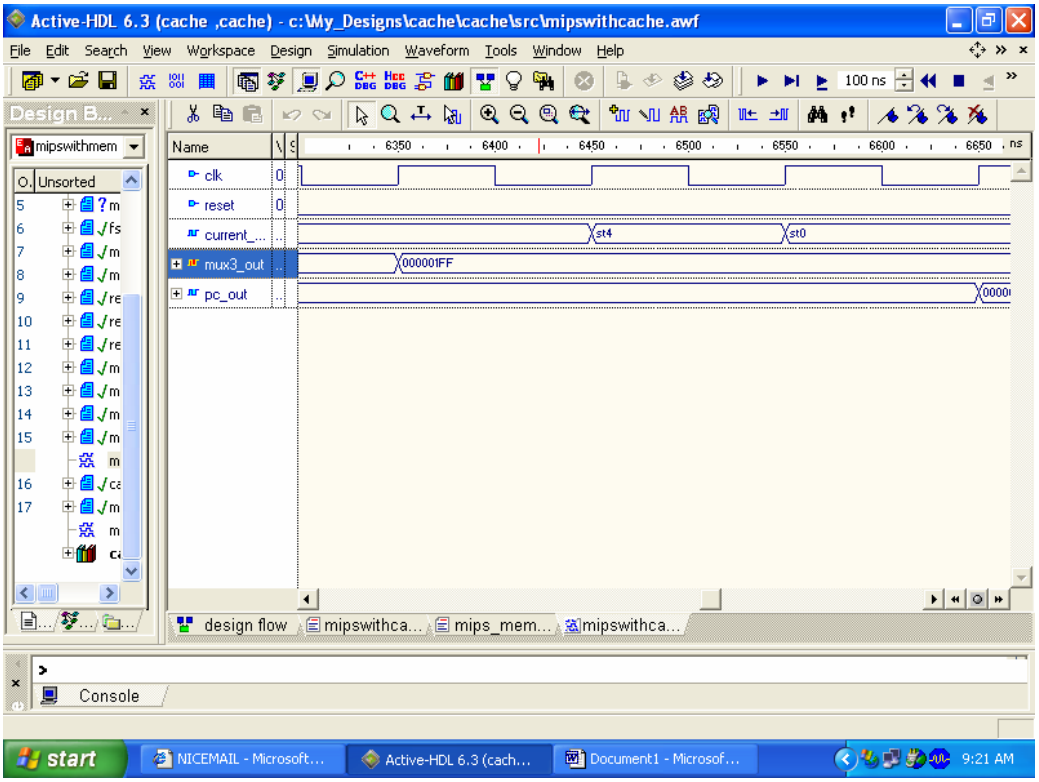

Active-HDL 6.3 (mips ,mips) - c:\My_Designs\mips\mips\src\mipswithmem.awf *
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Design B...
mipswithmem
Name
result
current_...
mem_re...
DataFetch
pc_out
write_sig...
mux3_out
read_sig...
address...
data_in_...
data_out...
reset
clk
00000003
0000000C
design flow
mipswithm...
mips_mem...
mips_toplev...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (mips ...
cache_screenshots - ...
9:29 AM

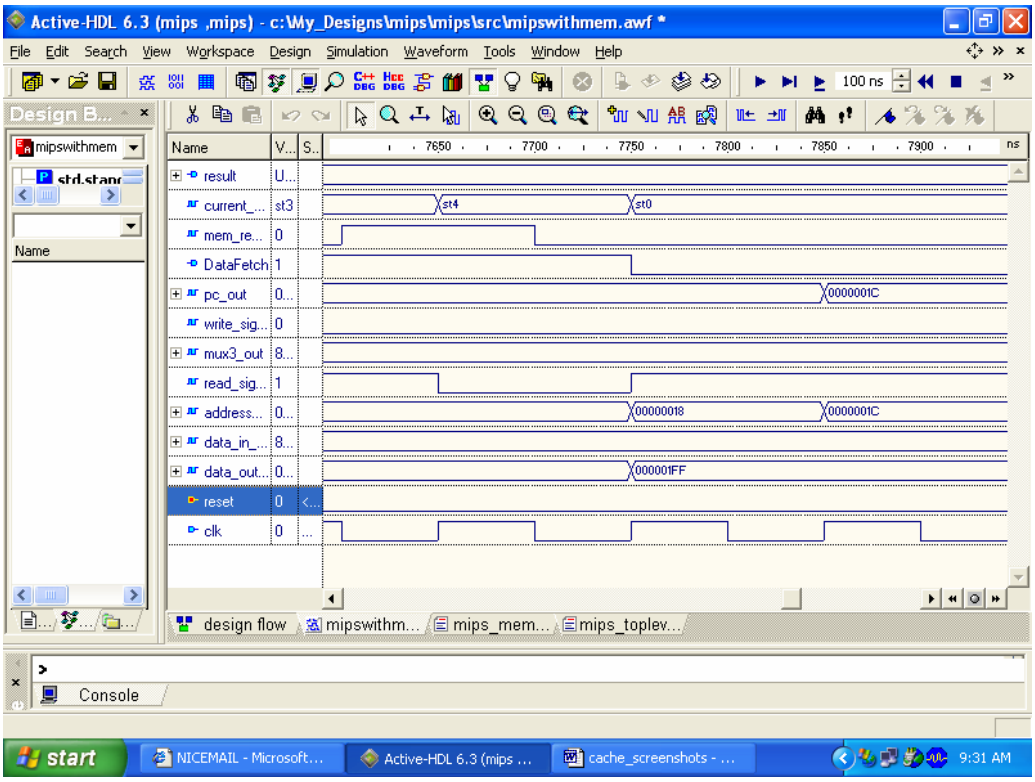

Active-HDL 6.3 (cache ,cache) - c:\My_Designs\cache\cache\src\mipswithcache.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
100 ns
Design B...
mipswithmem
Unsorted
Name
clk
reset
current_...
mux3_out
pc_out
st4
st0
00000007
design flow
mipswithca...
mips_mem...
mipswithca...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (cach...
Document1 - Microsof...
9:20 AM


Active-HDL 6.3 (mips ,mips) - c:\My_Designs\mips\mips\src\mipswithmem.awf *
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
100 ns
Design B...
mipswithmem
Name
result
current_...
mem_re...
DataFetch
pc_out
write_sig...
mux3_out
read_sig...
address...
data_in_...
data_out...
reset
clk
st3
st4
st0
00000007
00000010
00000007
design flow
mipswithm...
mips_mem...
mips_toplev...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (mips ...
cache_screenshots - ...
9:30 AM


Active-HDL 6.3 (cache ,cache) - c:\My_Designs\cache\cache\src\mipswithcache.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
100 ns
Design B...
mipswithmem
Unsorted
Name
clk
reset
current_...
mux3_out
pc_out
st4
st0
0000001F
design flow
mipswithca...
mips_mem...
mipswithca...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (cach...
Document1 - Microsof...
9:20 AM


Active-HDL 6.3 (mips ,mips) - c:\My_Designs\mips\mips\src\mipswithmem.awf *
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
100 ns
Design B...
mipswithmem
Name
result
current_...
mem_re...
DataFetch
pc_out
write_sig...
mux3_out
read_sig...
address...
data_in_...
data_out...
reset
clk
st4
st0
0000001F
00000014
design flow
mipswithm...
mips_mem...
mips_toplev...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (mips ...
cache_screenshots - ...
9:30 AM


Active-HDL 6.3 (cache ,cache) - c:\My_Designs\cache\cache\src\mipswithcache.awf
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Design B...
mipswithmem
Name
clk
reset
current_...
mux3_out
pc_out
st4
st0
000001FF
design flow
mipswithca...
mips_mem...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (cach...
Document1 - Microsof...
9:21 AM


Active-HDL 6.3 (mips ,mips) - c:\My_Designs\mips\mips\src\mipswithmem.awf *
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Design B...
mipswithmem
Name
result
current_...
mem_re...
DataFetch
pc_out
write_sig...
mux3_out
read_sig...
address...
data_in_...
data_out...
reset
clk
st3
st4
st0
0000001C
00000018
000001FF
design flow
mipswithm...
mips_mem...
mips_toplev...
Console
start
NICEMAIL - Microsoft...
Active-HDL 6.3 (mips ...
cache_screenshots - ...
9:31 AM

6. CONCLUSION

The designed 32-bit Processor is simulated with Main Memory and Cache Memory included. The instruction area in the Main Memory is loaded with the program code to be executed. Data area in the Main Memory is filled with following nine constants: FF, 01, 03, 07, 1F, 1FF, 7FFF, 3FFFF and 01. Sequence of Instructions executed is Load Word (*lw*) to load consecutive words in the data memory to Register number 8 in Register file.

Design is initialized with Reset input set to '1'. At 100 ns, Reset input is changed to '0' and simulation is executed till 10 microseconds.

When the microprocessor and the memory are configured in the design and applied all the inputs, the Microprocessor starts working by fetching the instructions stored in the Memory and executing each instruction. The result is analyzed by monitoring the output and the corresponding current state of the finite state control. In a *lw* instruction ST4 of FSM correspond to the register write of result.

First the simulation is done with Processor and Main Memory connected together in the design. After noting the resulted timing, during the next simulation Cache Memory also is included in the design. Access time delay of 200ns and burst access time delay of 50ns is set as the Memory variable during compilation. The below given table illustrates the timing comparison of both the simulations.

Data Fetched from Memory	**Time at which the fetched data is written to microprocessor register *when Data Cache memory excluded***	**Time at which the fetched data is written to microprocessor register *when Data Cache memory included***
FF	**1150 ns**	**950 ns**
01	**2450 ns**	**2050 ns**
03	**3750 ns**	**3150 ns**
07	**5050 ns**	**4250 ns**
1F	**6350 ns**	**5350 ns**
1FF	**7650 ns**	**6450 ns**

Thus it is concluded that the simulation is functioning as desired and there is an improvement in the data access response when Data Cache is included in the design.

7. SCOPE FOR FURTHER IMPROVEMENT

In the present design a basic model of Cache Memory is developed for data memory area only. The model developed can be further modified to handle Cache misses. Moreover for Instruction Memory also Cache Memory can be developed. Data path can be modified to handle jump instructions as well. Once that is done, some sort of benchmarking program can be written and executed in a loop to evaluate the overall Cache performance.

8. REFERENCES

1. Hennessy. J, and D. Patterson. "Computer Organization and Design: The hardware/software interface", Morgan Kaufmann, 1998.

2. Bhaskar. J, "A VHDL Primer", Pearson Education Asia, 1999.

3. Robert K Dueck, "Digital design with CPLD application and VHDL", Thomson Delmar Learning,2001.

4. Douglas Perry, "VHDL Programming by Example", Tata McGraw Hill,2000.

5. Peter. J. Ashenden, "Designers guide to VHDL", Morgan Kaufmann,2002.

6. Charles. H. Roth Jr. , "Digital Systems Design using VHDL" , Thomson Learning , 2004

7. Home page of MIPS Technologies, Inc, www.mips.com