Contemporary Engineering Sciences, Vol. 6, 2013, no. 1, 19 – 34
HIKARI Ltd, www.m-hikari.com

Effective Exploiting of Hardware Description Language as a Tool in Teaching the Digital and Computer Circuits

Fisal Al-Shreef [1], Ahmed Al-Tarazi [2], Hussein Ahmad Al-Ofeishat [3] and Al. Shraideh Khaled [4]

1. King Abdulaziz University, Kingdom of Saudi Arabia
2. Northern Border University, Kingdom of Saudi Arabia
3. Al-balga applied University, Jordan
4. King Abdulaziz University, Kingdom of Saudi Arabia

Abstract

With the increase of knowledge in the field of computer science and technology as well as the accelerating change in these fields, the researchers believe that the enhancement of the output of the educational process at the university level requires not only the adaption of new syllabus , but also implementation of modern tools that help improve the students' knowledge. However, this paper aims to explore the benefits of using Hardware Description Language as a tool for teaching the design of digital and computer circuits. In addition, to present a basic understanding of how the digital and computer circuits work, and how to design multicycle processor by using small components.

Keywords: Processor design, central process unit, CPU, instruction set, Verilog

1. Introduction

Using the integrated circuits in the computer architecture and digital design courses doesn't mean that the students have good understanding of these components. Often, students may have a good idea about the pin configuration, the input and output

signals which they have to send and receive. Using Hardware Description Language will improve their knowledge, because they will design and describe everything, including the internal signals and all the micro details. This is one of the reasons which makes this language very important for education field.
From the codes of the modules, the students understand its architecture and how it works. For instance, the memory and arithmetic logic unit, the control unit, etc. Writing the code for all these modules and dealing with these details will improve their understanding.

Nowadays, the components of computers and digital systems have become so sophisticated as well as understanding these components isn't easy without using simulation software's or high level languages. In addition, the traditional packages don't have capability to deal with these components. For the above mentioned reasons, most of educators have used Hardware description Languages in their courses to learn all the concepts.

There are a lot of packages which used to implement and simulate the digital circuits. Altera and Xilinx companies have produced many different packages like ModelSim, ActiveHDL, Quartus, Webpak….etc. All of these packages use Hardware Description Language to design and implement the complex digital circuits and enable the user to deal with the all details for digital design processes. Moreover, some of these packages allow the users to design their circuits and test the ability to fit their designs into FPGA, then fit them to work according to its function and they offer timing analysis for these circuits like Quartus, Synplfy, etc.
Hardware Description Language is a language used to describe a digital system, for example, a computer or a component of a computer. One may describe a digital system at several levels. Also, an HDL might describe the layout of the wires, resistors and transistors on an Integrated Circuit (IC) chip, i. e., the switch level or it might describe the logical gates and flip flops in a digital system, i. e., and the gate level. Even higher level describes the registers and the transfer of vectors of information between registers. This is called the Register Transfer Level (RTL). Verilog supports all of these levels. However, this handout focuses on only the portions of Verilog which support the RTL level.

2. Verilog and VHDL Language

Verilog is one of the two major Hardware Description Languages (HDL) used by hardware designers in industry and academy. VHDL is the other one. The industry is currently split on which is better. Many feel that Verilog is easier to learn and use than VHDL. As one hardware designer puts it, "I hope the competition uses VHDL." VHDL was made an IEEE Standard in 1987, and Verilog in 1995. Verilog is very C-like and liked by electrical and computer engineers as most learn the C language in college. VHDL is very Ada-like and most engineers have no experience with Ada.
Verilog was introduced in 1985 by Gateway Design System Corporation, now a part of Cadence Design Systems, Inc.'s Systems Division. Until May, 1990, with the

formation of Open Verilog International (OVI), Verilog HDL was a proprietary language of Cadence. Cadence was motivated to open the language to the Public Domain with the expectation that the market for Verilog HDL-related software products would grow more rapidly with broader acceptance of the language. Cadence realized that Verilog HDL users wanted other software and service companies to embrace the language and develop Verilog-supported design tools.
Verilog HDL allows a hardware designer to describe designs at a high level of abstraction such as at the architectural or behavioral level as well as the lower implementation levels (i. e. , gate and switch levels) leading to Very Large Scale Integration (VLSI) Integrated Circuits (IC) layouts and chip fabrication. A primary use of HDL is the simulation of designs before the designer must commit to fabrication. This handout does not cover all of Verilog HDL but focuses on the use of Verilog HDL at the architectural or behavioral levels. The handout emphasizes design at the Register Transfer Level (RTL) [1].

3. Limitations of Traditional Programming Languages

There are wide varieties of computer programming languages, from Fortran , C and Java. Unfortunately, they are not adequate to model digital hardware. To understand their limitations, it is beneficial to examine the development of a language. A programming language is characterized by its syntax and semantics. The syntax comprises the grammatical rules used to write a program, and the semantics is the "meaning" associated with language constructs. When a new computer language is developed, the designers first study the characteristics of the underlying processes and then develop syntactic constructs and their associated semantics to model and express these characteristics.

Most traditional general-purpose programming languages, such as C, are modeled after a sequential process. In this process, operations are performed in sequential order, one operation at a time. Since an operation frequently depends on the result of an earlier operation, the order of execution cannot be altered at will. The sequential process model has two major benefits. At the abstract level, it helps the human thinking process to develop an algorithm step by step. At the implementation level, the sequential process resembles the operation of a basic computer model and thus allows efficient translation from an algorithm to machine instructions.
The characteristics of digital hardware, on the other hand, are very different from those of the sequential model. A typical digital system is normally built by smaller parts, with customized wiring that connects the input and output ports of these parts. When a signal changes, the parts connected to the signal are activated and a set of new operations is initiated accordingly. These operations are performed concurrently, and each operation will take a specific amount of time, which represents the propagation delay of a particular part, to complete. After completion, each part updates the value of the corresponding output port.

If the value is changed, the output signal will in turn activate all the connected parts and initiate another round of operations. This description shows several unique characteristics of digital systems, including the connections of parts, concurrent operations, and the concept of propagation delay and timing. The sequential model used in traditional programming languages cannot capture the characteristics of digital hardware, and there is a need for special languages that are designed to model digital hardware. [2]

4. The design [3]

In this paper we will design a multiCycle central processing unit (CPU). During the design operation we will design the internal components for our design depending on the our knowledge about how the multiCycle Processor work.
The processor will be able to handle fifteen different instructions, including R-type, I-type, and J type. This exercise is done to enhance our understanding of processor design, instruction sets, and Verilog code.
First of all, we will design the sub components of the processor, then, we will connect these components together to build our design which is designed to be able execute a variety of instructions in a multicycle implementation. The multicycle implementation breaks instructions down into multiple steps. Each step is designed to take one clock cycle. It allows each functional block to be used more then once per instruction if they are used on different clock cycles. This implementation has several key advantages over a single cycle implementation. First, it can share modules, allowing the use of fewer hardware components. Instead of multiple arithmetic logic units (ALU's) the multicycle implementation uses only one. Only one memory is used for the data and the instructions also. Breaking complex instructions into steps also allows us to significantly increase the clock cycle because we no longer have to base the clock on the instruction that takes the longest to execute. It is also uses several registers to temporarily hold the output of the previous clock cycle. These include an Instruction register, Memory data register, data register, etc.

The multicycle processor breaks down the simple instructions into a series of steps figure (1). These steps typically are the:

1. Instruction fetch step (IF).
2. Instruction decode and Register fetch step (ID)
3. Execution, memory address computation, or branch completion step (EXE)
4. Memory access or R-type instruction completion step (MEM).
5. Memory read completion step (WB)

IF	ID	EXE	MEM	WB

Fig (1) show the five stages

During the instruction fetch step the multicycle processor fetches instructions from the memory and computes the address of the next instruction, by incrementing the program counter (PC).
During the second step, the Instruction decode and register fetch step, we decode the instruction to figure out what type it is: memory access, R-type, I-type, branch. The format of these individual instructions will be discussed later.
The third step, the Execution, memory address computation, or branch completion step functions in different ways depending on what type of instruction the processor is executing. For a memory access instruction the ALU computes the memory address. An R-type instruction uses this third step to perform the actual arithmetic. This third step is the last step for branch and jump instructions. It is the step where the next PC address is computed and stored.
The fourth step only takes place in load word, store word, R-type, and I-type instructions. This step is when the load and store word instructions access the memory and use an arithmetic-logical instruction to write its result. Values are either loaded from memory and stored into the memory data register, or loaded from a register and stored back into the memory. This fourth step is the last step for R-type and I-type instructions.
For R and I type instructions this is the step where the result from the ALU computation is stored back into the destination register.
Only load instructions need the fifth step to finish up. This is the memory read completion step. In a load instruction the value of the memory data register is stored back into the register file.
These different steps are all controlled and orchestrated by the “brain” of the multicycle CPU. This “brain” is the controller. The controller is a finite state machine that works with the Opcode to walk the rest of the components through all the different steps, or states. The controller controls when each register is allowed to write and controls which operation the ALU is performing.

4.1 The Instruction Set:
The multicycle CPU can handle a total of 15 instructions. Nine of these were also available on the single-cycle CPU implementation. Of these, there are five R-type instructions: add, subtract, and, or, and set less than. There are three I-Type instructions: load word, store word, and branch on equal. The jump instruction is also supported. Six additional instructions were added to our multicycle design. These are: nor, xor, or immediate, and immediate, add immediate, and store less than immediate. Nor and xor were added as R-type instructions. The additional R-type instructions were implemented by including by augmenting the ALU. They have the same op-code as other R-type instructions, but the function code dictates the operation that is being performed. The four additional I-Type instructions were implemented by wiring the opcode to the control module. This is necessary because there is no function code in the instruction. Code had to be added to the control code to read these specific opcodes. A pair of states were also added to the state machine to handle these immediate instructions. Figure (2) show the instruction format.

OPCODE	RS	RT	RD	SHAMT	FUNCT

R – Type Instruction

OPCODE	RS	RT	IMMEDIATE

I – Type Instruction

OPCODE	ADDRESS

J – Type Instruction

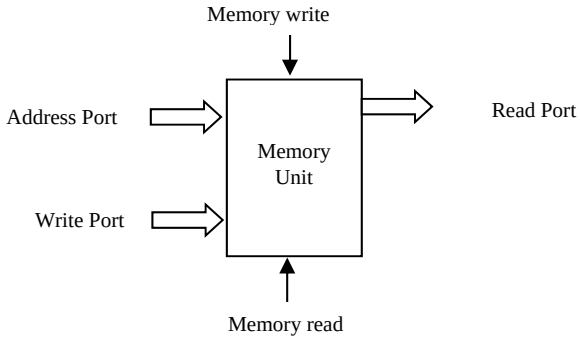
Fig (2) Instruction Format

4.2 The Components of the Design: [1,2,3,4]

The following units are the main components of our design. We will describe each one of them, and then we will connect them together by top module to build our processor. The complete code for the design is available on the website (http://www.hii.edu.ly/index.php?option=com_docman&task=cat_view&gid=76&Itemid=32.

4.2.1 Data Memory Unit:

Each processor need to memory to hold the data and the program which we want to execute. The following block shows the memory unit.



The Verilog module

// the name of the module and the input and output signals

```
Module DataMemory(Clk, Address, WriteData, MemWrite, MemRead, MemData);
input [31:0] Address, WriteData;
input MemRead, MemWrite, Clk;
output [31:0] MemData; wire [31:0] Mem
reg [31:0] RegFile [512:0];

always @ (posedge Clk) begin
if(MemWrite==1'b1)begin
RegFile[Address] <= WriteData;
end
end
```

Input and output declaration

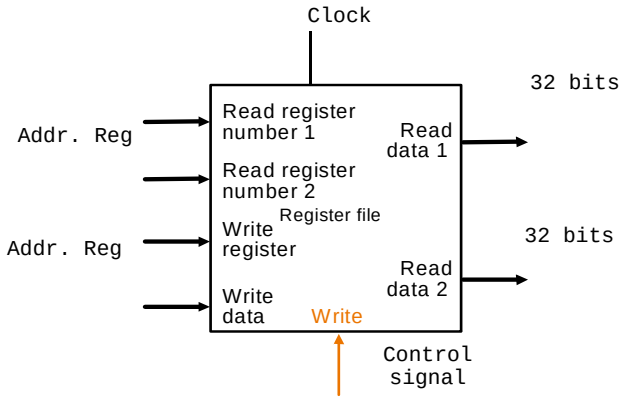
the write and read operation

```
assign MemData = (MemRead==1'b1)? RegFile[Address] : 0;
initial begin //load in data and instructions of program
RegFile[0] <= 32'd8;
RegFile[1] <= 32'd1;
RegFile[2] <= 32'd1;
RegFile[128] <= 32'h8c030000;
RegFile[132] <= 32'h8c040001;
RegFile[136] <= 32'h8c050002;
RegFile[140] <=32'h8c010002;
RegFile[144] <=32'h10600004;
RegFile[148] <=32'h00852020;
RegFile[152] <=32'h00852822;
RegFile[156] <=32'h00611820;
RegFile[160] <=32'h1000fffb;
RegFile[164] <=32'hac040006;
end
Endmodule
```

Store the data and the code here

4.2.2 Register File Unit:

It contains some of 32bits registers which used for store the operands for the instructions in these registers.



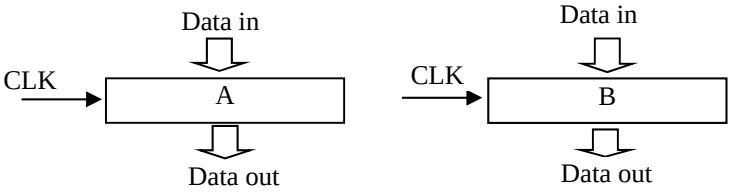
The module:

```
module RegFile(RA,RB,W,WE,WD,RDA,RDB,CLK);
input [4:0] RA, RB, W;
input [31:0] WD;
input WE, CLK;
output [31:0] RDA, RDB;
reg [31:0] RegFile [31:0];
reg [5:0] i;
always @(posedge CLK)
begin
```

// the code here

end
endmodule

4.2.3 A & B registers:
They are two registers which used for ALU operands. Both register can get the data from the memory or from the other register and pass it to ALU.

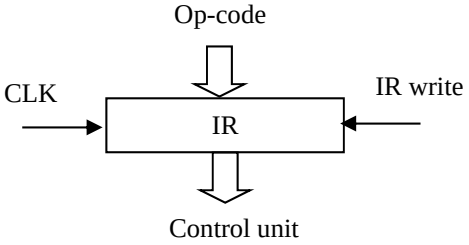


The Module:
module ABregister(DataIn, DataOut ,CLK);
input [31:0] DataIn;
input CLK;
output [31:0] DataOut;
reg [31:0] DataOut;
always @(posedge CLK)
begin

// the code here

end
endmodule

4.2.4 IR Register (instruction register):
It is 32bits register which used for holding the Instruction after fetching it from the memory to decode.



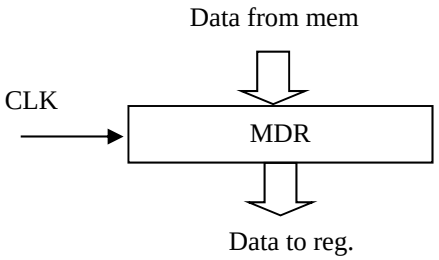
The Module:
module IR(IRWrite, DataIn, DataOut ,CLK);
input [31:0] DataIn;
input CLK, IRWrite;
output [31:0] DataOut;
reg [31:0] DataOut;
always @(posedge CLK)
begin

// the code here

end
endmodule

4.2.5 MDR (memory data register):
It is 32bits register which used for holding the data (operands) after reading it from the memory.

The module:
module MDR(DataIn, DataOut ,CLK);
input [31:0] DataIn;
input CLK;
output [31:0] DataOut;
reg [31:0] DataOut;
always @(posedge CLK)
begin



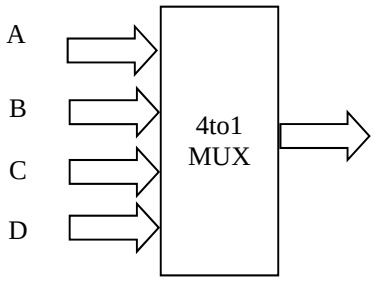
// the code here

end
endmodule

4.2.6 32bits 4 to 1 multiplexer:

It used for select which data source will connect to ALU unit.

The Module:
module Four2One(Control, A, B, C, D, Out);
input [1:0] Control;
input [31:0] A, B, C, D;
output [31:0] Out;
reg [31:0] temp1, temp2, Out;
always @ (Control, A, B, C, D)
begin



// the code here

end
endmodule

4.2.7 Sign extender:

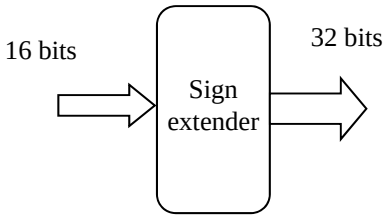
This unit used for keep the sign for the operands at changing the width of these operands in some instructions.

The module:

```
module SignExtend(In, Out);
input [15:0] In;
output [31:0] Out;

// the code here

endmodule
```



4.2.8 Control Unit:

This unit is very important. it responsible on generation all the control signals in the processor. It receives the operation code for the instruction from IR and decodes it, and then it generates the control signals which depend on the type of instruction itself. The controller is very complex and usually used the finite state machine (FSM) to design the controllers. Figure (3) show the finite state machine describes our controller.

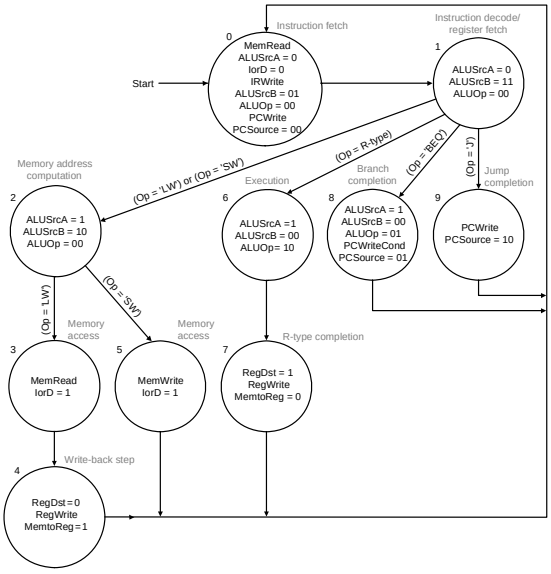


Fig (3) finite state machine for Control Unit

The Module:

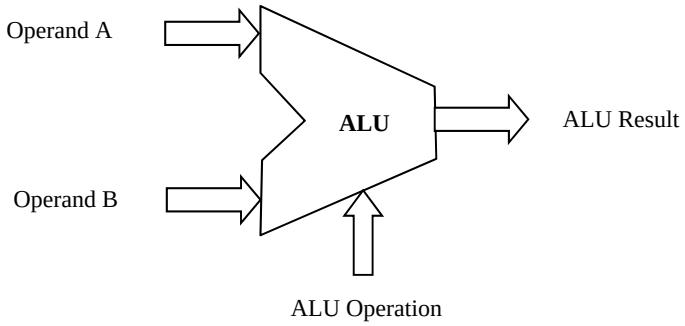
```
module controller (Clk, Reset, Op, PCWriteCond, PCWrite, lorD,
MemRead, MemWrite,
Memorex, IRWrite, PCSource, ALUOp, ALUSrcB, ALUSrcA,
 RegWrite, RegDst);
input [5:0] Op; input Clk, Reset;
output [1:0] ALUOp, ALUSrcB, PCSource;
output PCWriteCond, PCWrite, lorD, MemRead, MemWrite, MemtoReg,
IRWrite, ALUSrcA, RegWrite, RegDst; reg [1:0] ALUOp, ALUSrcB,
PCSource;
reg PCWriteCond, PCWrite, lorD, MemRead, MemWrite, MemtoReg,
IRWrite,
ALUSrcA, RegWrite, RegDst; reg [4:0] state =0, nextstate;
parameter S0=0; parameter S1=1; parameter S2=2; parameter S3=3;
 parameter S4=4; parameter S5=5;
parameter S6=6; parameter S7=7; parameter S8=8; parameter S9=9;
parameter S10=10; parameter S11=11;
always@(posedge Clk)
begin

// the code here
End
endmodule
```

1.2.9 ALU Unit:

It is the unit where all the arithmetic and logical operations executed.

Also it used for resisters increment and addresses computations etc.



The Module:

```
module MainALU(DataA, DataB, Operation, ALUResult, Zero);
input [31:0] DataA, DataB; input [3:0] Operation;
output [31:0] ALUResult; output Zero;
reg [31:0] ALUResult; reg Zero;
always @ (Operation, DataA, DataB)
begin

// the code here
End
Endmodule
```

1.2.10 CPU Module:

It is the top level module which connects all the last modules together to build the multicycle processor.

Top Module:

```
module cpu(cycle, PC, inst, alu_out, mem_out, clock, regdst, aluop,
alusrc, branch, memread,
memwrite, regwrite, memtoreg, zero); input clock;
output[31:0] cycle, PC, inst, alu_out, mem_out;
output regdst, alusrc, branch, memread, memwrite, regwrite, memtoreg;
output[1:0] aluop; output zero; reg[31:0] cycle=32'd0;
always @ (posedge clock) begin

// the code here

End
Endmodule
```

Figure (4) shows the complete design which explains all the modules that used and how they are connected to each other.

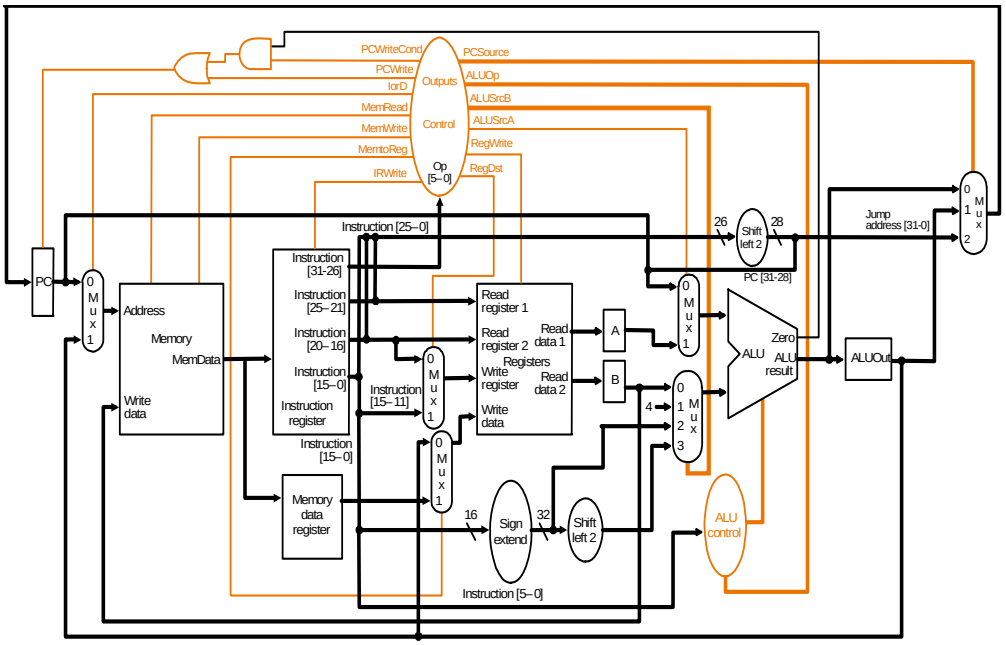


Fig (4) the complete design

1.2.11 Testing and Results:
1.2.12

To get results we have to test the design. Our test bench here is very simple, it is just for generate the clock signal. Also the program which we use saved in memory unit. It is a simple program which used for calculate Fibonacci numbers. The operation code for this program saved in the memory module.

Figure (5) shows the results of this example which we get by using ModelSim package. The figure is wave form which describes all the signals and busses which used in the modules. From the wave form the students can trace or see the signals and the data and the addresses to understand the mechanism of any module, also they can from the wave form debug the design if there is any mistakes or wrong signals and correct the mistakes. Also, the students can modify the code to compute the Fibonacci for different numbers.

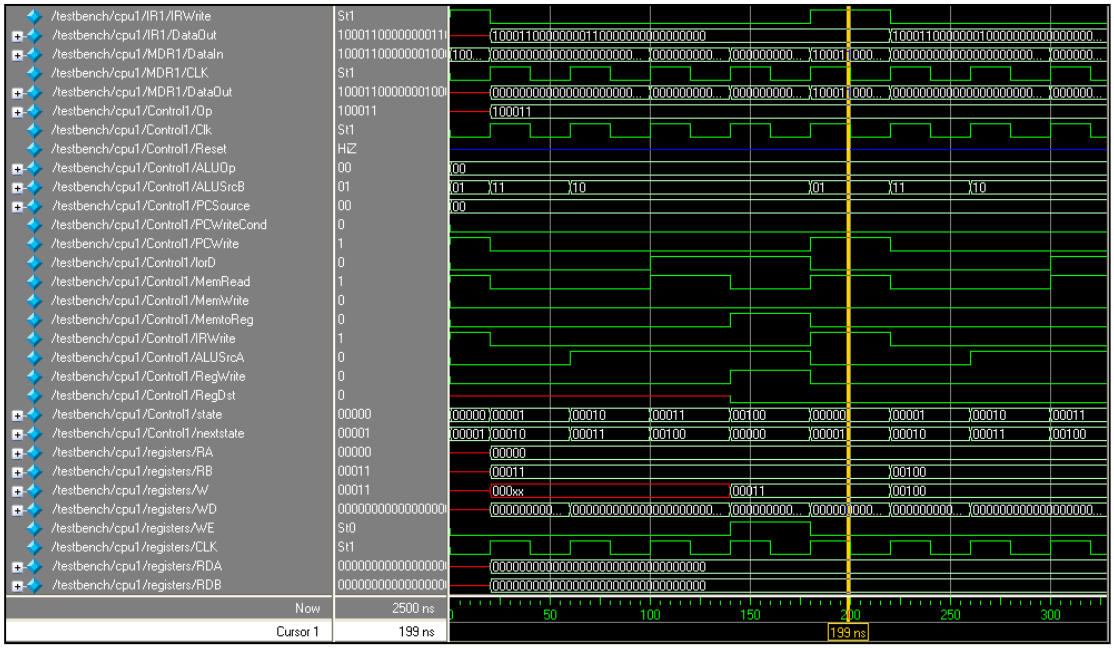


Fig (5) the wave form for results

5. Conclusion

Using Hardware Description Language to design the digital circuits is very useful to understand how the digital circuits work. Using this language allow the students to get deep understanding about the internal signals and operations which are invisible by using hardware circuits. Also the packages which used in this language like quartus and simplify offer timing analysis for the circuits and many other functions. In our design we design the units for multicycle processor, during the design operations we described the behavior of these units which allow to these units to work as we need.

References

[1] “Verilog HDL: A Guide to Design and Synthesis”, Samir Palntkar, McGraw-Hill, 2000.

[2] “VHDL Programming By Examples”, Douglas L. Perry, McGraw-Hill, 2002.

[3] “CMOS VLSI Design”, Weste and Harris, Addison-Wesley, 2005.

[4] “Verilog Quickstart”, James M. Lee, Kluwer Academic Publishers, 2002.

Received: November, 2012