Aşağıda tanımlanmış Tek Çevrimli (Single Cycle) MIPS mimarisini VHDL kullanarak ve VHDL simulatör (Xilinx veya benzeri) aracılığıyla gerçekleştiriniz?

Aşağıda verilen MIPS programı geliştirdiğiniz MIPS single cycle mimaride simüle ederek şekilde görüldüğü gibi waveformları elde ediniz?

```
                --PC  : INSTRUCTION
  00: 8C410001;  --00 : LW   $1, 1(2)     --> $1 = MEM (0x02 + 0x01)
                 --                              = MEM (03)       = 0x33
  01: 00A41822;  --04 : SUB $3, $5, $4    --> $3 = 0x05 -   0x04 = 0x01
  02: 00E61024;  --08 : AND $2, $7, $6    --> $2 = 0x07 AND 0x06 = 0x06
  03: 00852025;  --0C : OR  $4, $4, $5    --> $4 = 0x04 OR  0x05 = 0x05
  04: 00C72820;  --10 : ADD $5, $6, $7    --> $5 = 0x06 +   0x07 = 0x0D
  05: 1421FFFA;  --14 : BNE $1, $1, -24   --> NOT TAKEN
  06: 1022FFFF;  --18 : BEQ $1, $2, -4    --> NOT TAKEN
  07: 0062302A;  --1C : SLT $6, $3, $2    --> $6 =  $3  <  $2 = 0x01
                 --                             = 0x01 < 0x06
  08: 10210002;  --20 : BEQ $1, $1, 2     --> TAKEN: 0x33 = 0x33
  09: 00000000;  --24 : NOP                --> NOP
  0A: 00000000;  --28 : NOP                --> NOP
  0B: AC010002;  --2c : SW   $1, 2         --> $1 = 0X33 = MEMORY(02)
  0C: 00232020;  --30 ; ADD $4, $1, $3     --> $4= 0x33 + 0x01 = 0x34
  0D: 08000000;  --34 : JUMP 0             --> JUMP TO PC = 00
END;
```

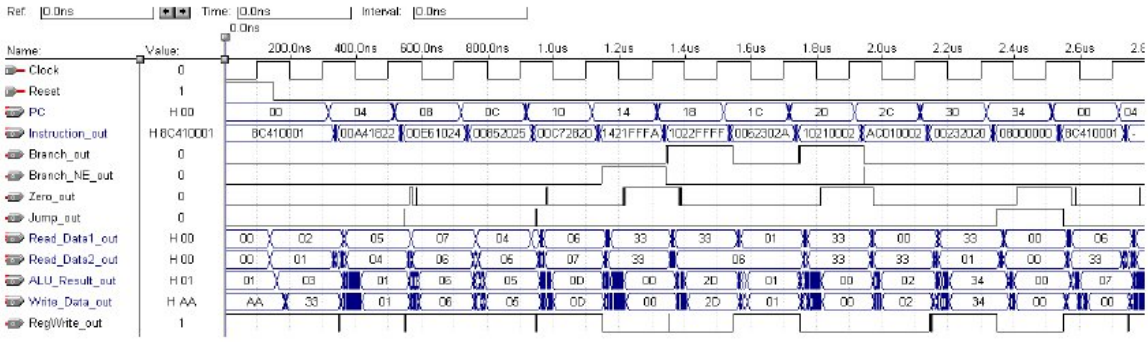


Figure D.1 MIPS Single-cycle Simulation Waveform

## 3.1 THE MIPS INSTRUCTION SET ARCHITECTURE

As mentioned before MIPS is a RISC microprocessor architecture. The MIPS Architecture defines thirty-two, 32-bit general purpose registers (GPRs). Register $r0 is hard-wired and always contains the value zero. The CPU uses byte addressing for word accesses and must be aligned on a byte boundary divisible by four (0, 4, 8, …). MIPS only has three instruction types: I-type is used for the Load and Stores instructions, R-type is used for Arithmetic instructions, and J-type is used for the Jump instructions as shown in Figure 3.2. Table 3.1 provides a description of each of the fields used in the three different instruction types.

MIPS is a load/store architecture, meaning that all operations are performed on operands held in the processor registers and the main memory can only be accessed through the load and store instructions (e.g lw, sw). A load instruction loads a value from memory into a register. A store instruction stores a value from a register to memory. The load and store instructions use

the sum of the offset value in the address/immediate field and the base register in the Srs field to address the memory. Arithmetic instructions or R-type include: ALU Immediate (e.g. addi), three-operand (e.g. add, and, slt), and shift instructions (e.g. sll, srl). The J-type instructions are used for jump instructions (e.g. j). Branch instructions (e.g. beq, bne) are I-type instructions which use the addition of an offset value from the current address in the address/immediate field along with the program counter (PC) to compute the branch target address; this is considered PC relative addressing. Table 3.2 shows a summary of the core MIPS instructions.

| Field | Description |
|---|---|
| *opcode* | 6-bit primary operation code |
| *rd* | 5-bit specifier for the destination register |
| *rs* | 5-bit specifier for the source register |
| *rt* | 5-bit specifier for the target (source/destination) register or used to specify functions within the primary *opcode* REGIMM |
| *address/immediate* | 16-bit signed *immediate* used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement |
| *instr_index* | 26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address |
| *sa* | 5-bit shift amount |
| *function* | 6-bit function field used to specify functions within the primary *opcode* SPECIAL |

Table 3.1 MIPS Instruction Fields [11]

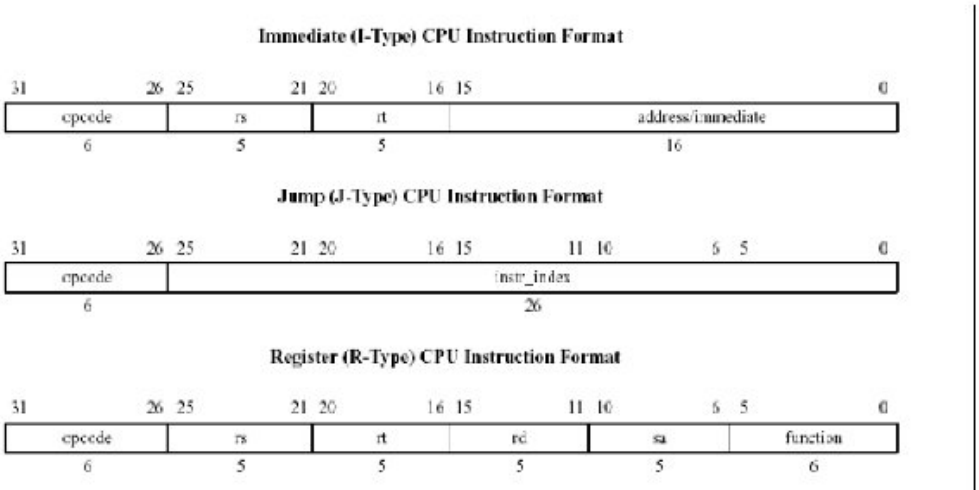


Figure 3.2 MIPS Instruction Types [11]

| Instruction | Symbol | Format | Example | Meaning | Comments |
|---|---|---|---|---|---|
| Add | add | R | add $r1, $r2, $r3 | $r1 = $r2 + $r3 | overflow detected |
| Add Immediate | addi | I | addi $r1, $r2, 100 | $r1 = $r2 + 100 | plus constant |
| Add Unsigned | addu | R | addu $r1, $r2, $r3 | $r1 = $r2 + $r3 | overflow undetected |
| Subtract | sub | R | sub $r1, $r2, $r3 | $r1 = $r2 - $r3 | overflow detected |
| Subtract Unsigned | subu | R | subu $r1, $r2, $r3 | $r1 = $r2 - $r3 | overflow undetected |
| And | and | R | and $r1, $r2, $r3 | $r1 = $r2 & $r3 | bitwise logical and |
| Or | or | R | Or $r1, $r2, $r3 | $r1 = $r2 \| $r3 | bitwise logical or |
| Shift Left Logical | sll | R | sll $r1, $r2, 10 | $r1 = $r2 << 10 | shift left by constant |
| Shift Right Logical | srl | R | srl $r1, $r2, 10 | $r1 = $r2 >> 10 | shift right by constant |
| Set Less Than | slt | R | slt $r1, $r2, $r3 | if ($r2 < $r3) $r1 = 1 else 0 | compare less than |
| Load Word | lw | I | lw $r1, 100($r2) | $r1 = mem($r2 + 100) | load word from mem to reg |
| Store Word | sw | I | sw $r1, 100($r2) | mem($r2 + 100) = $r1 | store word from reg to mem |
| Branch on Equal | beq | I | beq $r1, $r2, 25 | if ($r1 = $r2) goto PC + 4 + 100 | equal test |
| Branch on Not Equal | bne | I | bne $r1, $r2, 25 | if ($r1 != $r2) goto PC + 4 + 100 | not equal test |
| Jump | j | J | j 100 | goto 400 | jump to target address |

Table 3.2 MIPS Core Instructions

## 3.2 MIPS SINGLE-CYCLE PROCESSOR

The MIPS single-cycle processor performs the tasks of instruction fetch, instruction decode, execution, memory access and write-back all in one clock cycle. First the PC value is used as an address to index the instruction memory which supplies a 32-bit value of the next instruction to be executed. This instruction is then divided into the different fields shown in Table 3.1. The instructions opcode field bits [31-26] are sent to a control unit to determine the type of instruction to execute. The type of instruction then determines which control signals are

to be asserted and what function the ALU is to perform, thus decoding the instruction. The instruction register address fields $rs bits [25 - 21], $rt bits [20 - 16], and $rd bits[15-11] are used to address the register file. The register file supports two independent register reads and one register write in one clock cycle. The register file reads in the requested addresses and outputs the data values contained in these registers. These data values can then be operated on by the ALU whose operation is determined by the control unit to either compute a memory address (e.g. load or store), compute an arithmetic result (e.g. add, and or slt), or perform a compare (e.g. branch). If the instruction decoded is arithmetic, the ALU result must be written to a register. If the instruction decoded is a load or a store, the ALU result is then used to address the data memory. The final step writes the ALU result or memory value back to the register file.

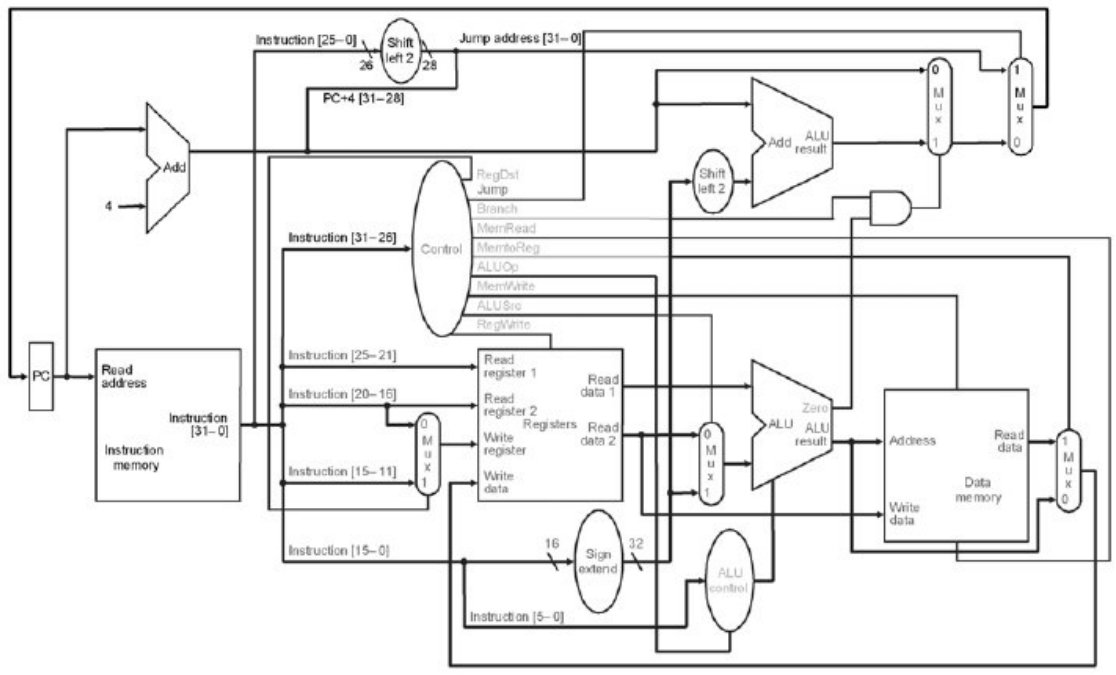


Figure 3.3 MIPS Single-cycle Processor

## 3.3 MIPS SINGLE-CYCLE PROCESSOR VHDL IMPLEMENTATION

The initial task of this project was to implement in VHDL the MIPS single-cycle processor using Altera MAX+PLUS II Text Editor to model the processor developed in [1]. A good VHDL reference and tutorial can be found in the appendices to the book *Fundamentals of Digital Logic with VHDL Design* by Stephen Brown and Zvonko Vranesic [12]. The *IEEE Standard VHDL Language Reference Manual* [13], also helped in the overall design of the VHDL implementation. The first part of the design was to analyze the single-cycle datapath and take note of the major function units and their respective connections.

The MIPS implementation as with all processors, consists of two main types of logic elements: combinational and sequential elements. Combinational elements are elements that operate on data values, meaning that their outputs depend on the current inputs. Such elements in the MIPS implementation include the arithmetic logic unit (ALU) and adder. Sequential elements are elements that contain and hold a state. Each state element has at least two inputs and one output. The two inputs are the data value to be written and a clock signal. The output signal provides the data values that were written in an earlier clock cycle. State elements in the MIPS implementation include the Register File, Instruction Memory, and Data Memory as seen in Figure 3.3. While many of logic units are straightforward to design and implement in VHDL, considerable effort was needed to implement the state elements.

It was determined that the full 32-bit version of the MIPS architecture would not fit onto the chosen FLEX10K70 FPGA. The FLEX10K70 device includes nine embedded array blocks (EABs) each providing only 2,048 bits of memory for a total of 2 KB memory space. The full 32-bit version of MIPS requires no less than twelve EABs to support the processor's register file, instruction memory, and data memory. In order for our design to model that in [1], the data

width was reduced to 8-bit while still maintaining a full 32-bit instruction. This new design allows us to implement all of the processor's state elements using six EABs, which can be handled by the FLEX10K70 FPGA device. Even though the data width was reduced, the design has minimal VHDL source modifications from the full 32-bit version, thus not impacting the instructional value of the MIPS VHDL model.

With our new design, the register file is implemented to hold thirty-two, 8-bit general purpose registers amounting to 32 bytes of memory space. This easily fits into one 256 x 8 EAB within the FPGA. The full 32-bit version of MIPS will require combining four 256 x 8 EABs to implement the register file. The register file has two read and one write input ports, meaning that during one clock cycle, the processor must be able to read two independent data values and write a separate value into the register file. Figure 3.4 shows the MIPS register file. The register file was implemented in VHDL by declaring it as a one-dimensional array of 32 elements or registers each 8-bits wide. (e.g. TYPE register_file IS ARRAY (0 TO 31) OF STD_LOGIC_VECTOR (7 DOWNTO 0) ) By declaring the register file as a one-dimensional array, the requested register address would need to be converted into an integer to index the register file.(e.g. Read_Data_1 <= register_file ( CONV_INTEGER (read_register_address1 (4 DOWNTO 0))) ) Finally, to save from having to load each register with a value, the registers get initialized to their respective register number when the Reset signal is asserted. (e.g. $r1 = 1, $r2 = 2, etc.)

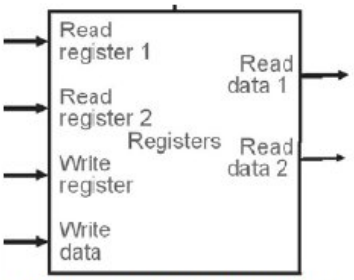


Figure 3.4 MIPS Register File

Altera MAX+PLUS II is packaged with a Library of Parameterized Modules (LPM) that allow one to implement RAM and ROM memory in Altera supported PLD devices. With our design this library was used to declare the instruction memory as a read only memory (ROM) and the data memory as a random access memory (RAM). Using the lpm_rom component from the LPM Library, the Instruction memory is declared as a ROM and the following parameters are set: the width of the output data port parameter lpm_width is set to 32-bits, the width of the address port parameter lpm_widthad is set to 8-bits, and the parameter lpm_file is used to declare a memory initialization file (.mif) that contains ROM initialization data. This allows us to set the indexed address data width to 8-bits, the instruction output to 32-bits wide, and enables us to initialize the ROM with the desired MIPS program to test the MIPS processor implementation. With these settings, four 256 x 8 EABs are required to implement the instruction memory. An example of the MIPS instruction memory can be seen in Figure 3.5 and the VHDL code implementation can be seen in Figure 3.6.

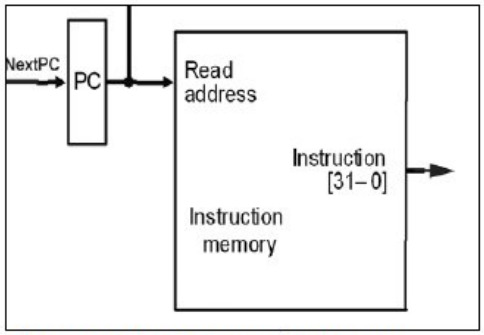

Figure 3.5 MIPS Instruction Memory

```
Instr_Memory: LPM_ROM

GENERIC MAP(
LPM_WIDTH               => 32,
LPM_WIDTHAD             => 8,
LPM_FILE                => "instruction_memory.mif",
LPM_OUTDATA             => "UNREGISTERED",
LPM_ADDRESS_CONTROL     => "UNREGISTERED")

PORT MAP (
        address         => PC,
        q               => Instruction );
```
Figure 3.6 VHDL – MIPS Instruction Memory

The data memory is declared using the lpm_ram_dq component of the LPM library. This component is chosen because it requires that the memory address to stabilize before allowing the write enable to be asserted high. The input Address width (lpm_widthad) and the Read Data

output width (lpm_width) are both declared as 8-bit wide, in lieu of our altered design. Using these settings allows us to use one 256 x 8 EAB instead of the 4 combined EABs required for the full 32-bit version of MIPS. An example of the MIPS data memory can be seen in Figure 3.7 and the VHDL code implementation can be seen in Figure 3.8.

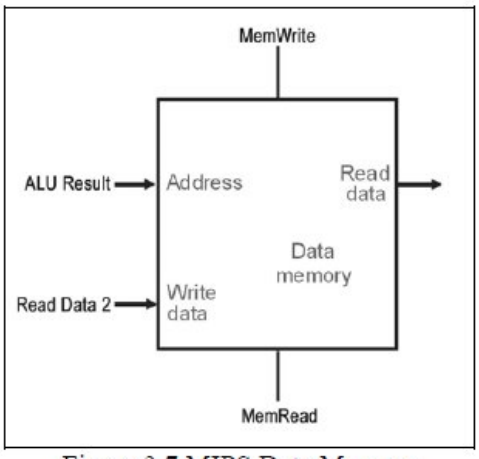


Figure 3.7 MIPS Data Memory

```
Data Memory : LPM_RAM_DQ

        GENERIC MAP(
                LPM_WIDTH            => 8,
                LPM_WIDTHAD          => 8,
                LPM_FILE             => "data memory.mif",
                LPM_INDATA           => "REGISTERED",
                LPM_ADDRESS_CONTROL  => "UNREGISTERED",
                LPM_OUTDATA          => "UNREGISTERED")

        PORT MAP(
                inclock      => Clock,
                data         => Write Data,
                address      => Address,
                we           => LPM_WRITE,
                q            => Read Data);
```

Figure 3.8. VHDL – MIPS Data Memory

Once we determined how to declare the state elements of the MIPS processor it was time to implement the rest of the logic devices in VHDL. Because the final task is to pipeline the single-cycle implementation of the MIPS processor, we decided to modularize the single-cycle

implementation into the five different VHDL modules to be fully utilized later in the pipelined implementation of the MIPS processor. The five modules are: Instruction Fetch, Instruction Decode, Control Unit, Execution, and Data Memory as shown in Figure 3.9.

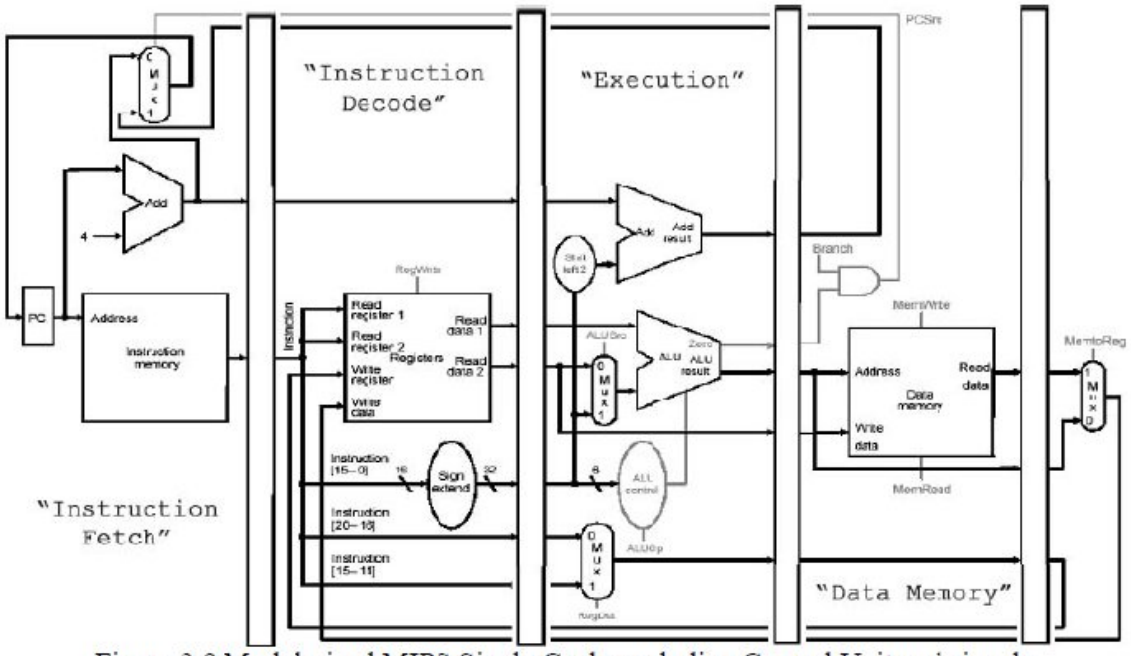

Figure 3.9 Modularized MIPS Single Cycle excluding Control Unit and signals.

With the decision to use five different modules to implement the single-cycle MIPS processor, the VHDL design becomes a two-level hierarchy. The top-level of the hierarchy is a structural VIIDL file that connects the all five components of the single-cycle implementation, while the bottom-level contains the behavioral VHDL models of the five different components. Appendix C contains the top-level structural VHDL code for the MIPS single-cycle processor.

### 3.3.1 INSTRUCTION FETCH UNIT

The function of the instruction fetch unit is to obtain an instruction from the instruction memory using the current value of the PC and increment the PC value for the next instruction as

shown in Figure 3.10. Since this design uses an 8-bit data width we had to implement byte addressing to access the registers and word address to access the instruction memory. The instruction fetch component contains the following logic elements that are implemented in VHDL: 8-bit program counter (PC) register, an adder to increment the PC by four, the instruction memory, a multiplexor, and an AND gate used to select the value of the next PC. Appendix C contains the VHDL code used to create the instruction fetch unit of the MIPS single-cycle processor.

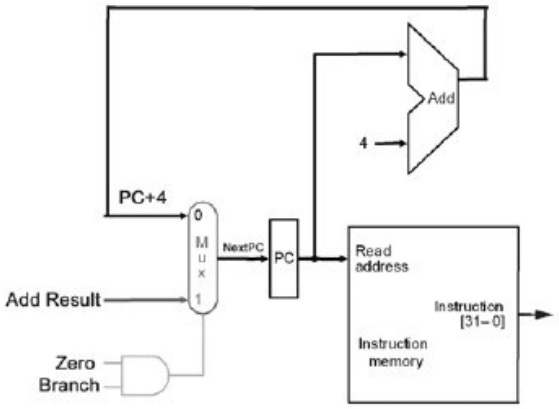


Figure 3.10 MIPS Instruction Fetch Unit

### 3.3.2 INSTRUCTION DECODE UNIT

The main function of the instruction decode unit is to use the 32-bit instruction provided from the previous instruction fetch unit to index the register file and obtain the register data values as seen in Figure 3.11. This unit also sign extends instruction bits [15 - 0] to 32-bit. However with our design of 8-bit data width, our implementation uses the instruction bits [7 – 0] bits instead of sign extending the value. The logic elements to be implemented in VHDL include several multiplexors and the register file, that was described earlier. Appendix C contains the VHDL code used to create the instruction decode unit of the MIPS single-cycle processor.

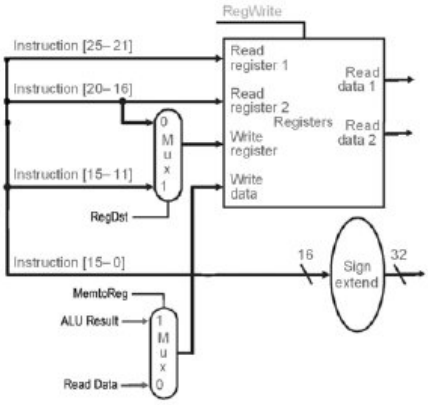


Figure 3.11 MIPS Instruction Decode Unit

### 3.3.3 THE CONTROL UNIT

The control unit of the MIPS single-cycle processor examines the instruction opcode bits [31 – 26] and decodes the instruction to generate nine control signals to be used in the additional modules as shown in Figure 3.12. The RegDst control signal determines which register is written to the register file. The Jump control signal selects the jump address to be sent to the PC. The Branch control signal is used to select the branch address to be sent to the PC. The MemRead control signal is asserted during a load instruction when the data memory is read to load a register with its memory contents. The MemtoReg control signal determines if the ALU result or the data memory output is written to the register file. The ALUOp control signals determine the function the ALU performs. (e.g. and, or, add, sbu, slt) The MemWrite control signal is asserted when during a store instruction when a registers value is stored in the data memory. The ALUSrc control signal determines if the ALU second operand comes from the register file or the sign extend. The RegWrite control signal is asserted when the register file

needs to be written. Table 3.3 shows the control signal values from the instruction decoded. Appendix C contains the VHDL code for the MIPS single-cycle control unit.

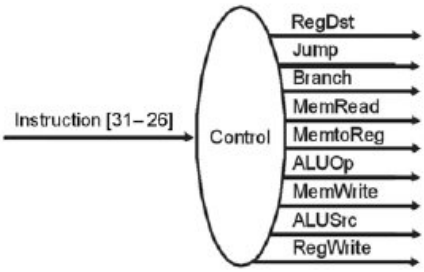


Figure 3.12 MIPS Control Unit

| Instruction | RegDst | ALUSrc | MemtoReg | RegWrite | MemRead | MemWrite | Branch | ALUOp1 | ALUOp0 |
|---|---|---|---|---|---|---|---|---|---|
| R-type | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| beq | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

Table 3.3 MIPS Control Signals

### 3.3.4 EXECUTION UNIT

The execution unit of the MIPS processor contains the arithmetic logic unit (ALU) which performs the operation determined by the ALUop signal. The branch address is calculated by adding the PC+4 to the sign extended immediate field shifted left 2 bits by a separate adder. The logic elements to be implemented in VHDL include a multiplexor, an adder, the ALU and the ALU control as shown in Figure 3.9 Appendix C contains the VHDL code used to create the execution unit of the MIPS single-cycle processor.

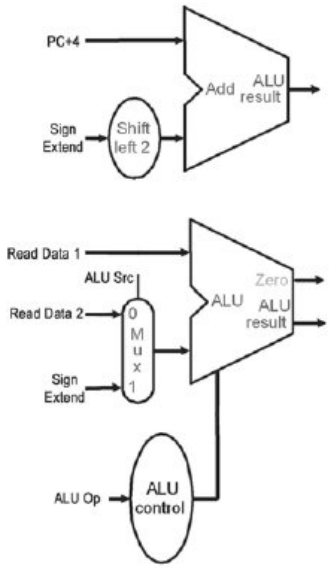


FIGURE 3.13 MIPS EXECUTION UNIT

### 3.3.5 DATA MEMORY UNIT

The data memory unit is only accessed by the load and store instructions. The load instruction asserts the MemRead signal and uses the ALU Result value as an address to index the data memory. The read output data is then subsequently written into the register file. A store instruction asserts the MemWrite signal and writes the data value previously read from a register into the computed memory address. The VHDL implementation of the data memory was described earlier. Figure 3.14 shows the signals used by the memory unit to access the data memory. Appendix C contains the complete VHDL code used to create the memory state of the MIPS single-cycle processor. Appendix D shows an example of MIPS single-cycle being simulated using Altera MAX+PLUS II waveform editor.

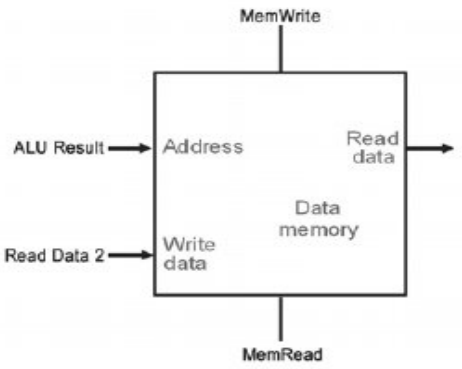


Figure 3.14 MIPS Data Memory Unit

## REFERENCES


[1] Patterson, D. A., Hennessy, J. L., *Computer Organization and Design: The Hardware/Software Interface*, 2nd edition, Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[2] Xilinx, *XC4000 Series Field Programmable Gate Arrays Product Specification*, ver. 1.6, 1999.

[3] Altera, *FLEX 10K Embedded Programmable Logic Device Family Data Sheet*, ver. 4.2., 2003.

[4] Altera, *MAX 7000 Programmable Logic Device Family Data Sheet*, ver. 6.02, 2002.

[5] MIPS Technologies, www.mips.com.

[6] SPIM, http://www.cs.wisc.edu/~larus/spim.html.

[7] Altera, *MAX+PLUS II Getting Started Manual*, ver. 6.0, 1995.

[8] Diab, H., Demashkieh, I., "A reconfigurable microprocessor teaching tool", *IEEE Proceedings A*, vol. 137, issue 5, September 1990.

[9] Gray, J., *Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip*, 2000.

[10] Altera, *University Program UP2 Development Kit User Guide*, ver. 3.0, 2003.

[11] MIPS Technologies, *MIPS32™ Architecture For Programmers Volume I: Introduction to the MIPS32™ Architecture*, rev. 2.0, 2003.

[12] Brown, S., Vranesic, Z., *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill Publishers, 2002.

[13] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE, New York, NY, 2002. IEEE Standard 1076-2002.

[14] Land. B, *Electrical Engineering 475 Microprocessor Architectures*, http://instruct1.cit.cornell.edu/Courses/ee475/

[15] Takahashi, R., Ohiwa, H., "Situated Learning on FPGA for Superscalar Microprocessor Design Education", *IEEE Proceedings of the 16th Symposium on Integrated Circuits and System Design*, 2003.

[16] Altera, *ByteBlaster MV Parallel Port Download Cable*, ver. 3.3, 2002.

[17] Larus, J. R., "*SPIM S20: A MIPS R2000 Simulator*", 1993.