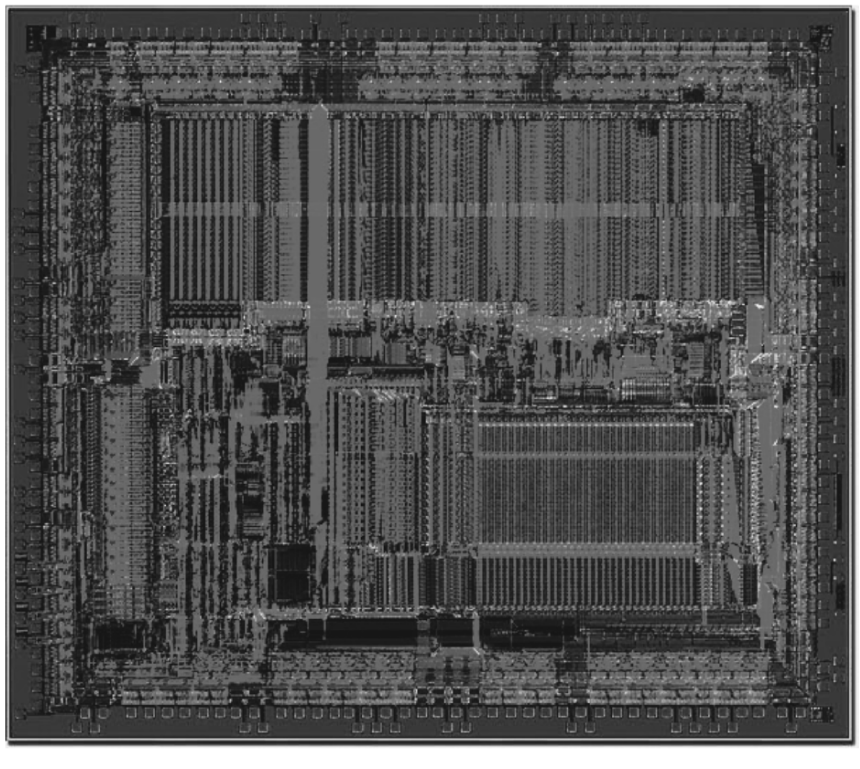
CHAPTER 14

A RISC Design: Synthesis of the MIPS Processor Core



A full die photograph of the MIPS R2000 RISC Microprocessor is shown above. The 1986 MIPS R2000 with five pipeline stages and 450,000 transistors was the world's first commercial RISC microprocessor. Photograph ©1995-2004 courtesy of Michael Davidson, Florida State University, http://micro.magnet.fsu.edu/chipshots.

14 A RISC Design: Synthesis of the MIPS Processor Core

14.1 The MIPS Instruction Set and Processor

The MIPS is an example of a modern reduced instruction set computer (RISC) developed in the 1980s. The MIPS instruction set is used by NEC, Nintendo, Motorola, Sony, and licensed for use by numerous other semiconductor manufacturers. It has fixed-length 32-bit instructions and thirty-two 32-bit general-purpose registers. Register 0 always contains the value 0. A memory word is 32 bits wide.

As seen in Table 14.1, the MIPS has only three instruction formats. Only I-format LOAD and STORE instructions reference memory operands. R-format instructions such as ADD, AND, and OR perform operations only on data in the registers. They require two register operands, Rs and Rt. The result of the operation is stored in a third register, Rd. R-format shift and function fields are used as an extended opcode field. J-format instructions include the jump instructions.

Table 14.1 MIPS 32-bit Instruction Formats.

Field Size	**6-bits**	**5-bits**	**5-bits**	**5-bits**	**5-bits**	**6-bits**
R- Format	Opcode	Rs	Rt	Rd	Shift	Function
I - Format	Opcode	Rs	Rt	Address/immediate value		
J - Format	Opcode	Branch target address				

LW is the mnemonic for the Load Word instruction and SW is the mnemonic for Store Word. The following MIPS assembly language program computes A = B + C.

```
LW  $2, B           ;Register 2 = value of memory at address B
LW  $3, C           ;Register 3 = value of memory at address C
ADD $4, $2, $3      ;Register 4 = B + C
SW  $4, A           ;Value of memory at address A = Register 4
```

The MIPS I-format instruction, BEQ, branches if two registers have the same value. As an example, the instruction BEQ $1, $2, LABEL jumps to LABEL if register 1 equals register 2. A branch instruction's address field contains the offset from the current address. The PC must be added to the address field to compute the branch address. This is called PC-relative addressing.

LW and SW instructions contain an offset and a base register that are used for array addressing. As an example, LW $1, 100($2) adds an offset of 100 to the contents of register 2 and uses the sum as the memory address to read data from. The value from memory is then loaded into register 1. Using register 0, which always contains a 0, as the base register disables this addressing feature.

Table 14.2 MIPS Processor Core Instructions.

Mnemonic	Format	Opcode Field	Function Field	Instruction
Add	R	0	32	Add
Addi	I	8	-	Add Immediate
Addu	R	0	33	Add Unsigned
Sub	R	0	34	Subtract
Subu	R	0	35	Subtract Unsigned
And	R	0	36	Bitwise And
Or	R	0	37	Bitwise OR
Sll	R	0	0	Shift Left Logical
Srl	R	0	2	Shift Right Logical
Slt	R	0	42	Set if Less Than
Lui	I	15	-	Load Upper Immediate
Lw	I	35	-	Load Word
Sw	I	43	-	Store Word
Beq	I	4	-	Branch on Equal
Bne	I	5	-	Branch on Not Equal
J	J	2	-	Jump
Jal	J	3	-	Jump and Link (used for Call)
Jr	R	0	8	Jump Register (used for Return)

A summary of the basic MIPS instructions is shown in Table 14.2. In depth explanations of all MIPS instructions and assembly language programming examples can be found in the references listed in section 14.11.

A hardware implementation of the MIPS processor core based on the example in the widely used textbook, *Computer Organization and Design The Hardware/Software Interface* by Patterson and Hennessy, is shown in Figure 14.1. This implementation of the MIPS performs fetch, decode, and execute in one clock cycle. Starting at the left in Figure 14.1, the program counter (PC) is used to fetch the next address in instruction memory. Since memory is byte addressable, four is added to address the next 32-bit (or 4-byte) word in memory. At the same time as the instruction fetch, the adder above instruction memory is used to add four to the PC to generate the next address. The output of instruction memory is the next 32-bit instruction.

The instruction's opcode is then sent to the control unit and the function code is sent to the ALU control unit. The instruction's register address fields are used to address the two-port register file. The two-port register file can perform two independent reads and one write in one clock cycle. This implements the decode operation.

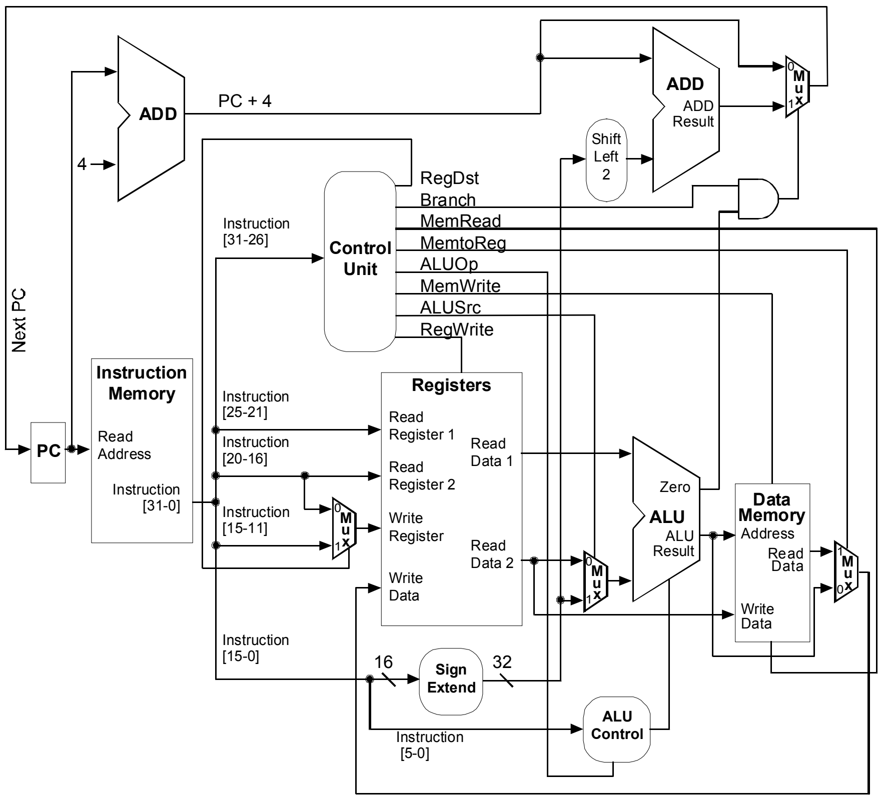


Figure 14.1 MIPS Single Clock Cycle Implementation.

The two outputs of the register file then feed into the data ALU inputs. The control units setup the ALU operation required to execute the instruction. Next, Load and Store instructions read or write to data memory. R-format instructions bypass data memory using a multiplexer. Last, R-format and Load instructions write back a new value into the register file.

PC-relative branch instructions use the adder and multiplexer shown above the data ALU in Figure 14.1 to compute the branch address. The multiplexer is required for conditional branch operations. After all outputs have stabilized, the next clock loads in the new value of the PC and the process repeats for the next instruction.

RISC instruction sets are easier to pipeline. With pipelining, the fetch, decode, execute, data memory, and register file write operations all work in parallel. In a single clock cycle, five different instructions are present in the pipeline. The basis for a pipelined hardware implementation of the MIPS is shown in Figure 14.2.

Additional complications arise because of data dependencies between instructions in the pipeline and branch operations. These problems can be resolved using hazard detection, data forwarding techniques, and branch flushing. With pipelining, most RISC instructions execute in one clock cycle. Branch instructions will still require flushing of the pipeline. Exercises that add pipelining to the processor core are included at the end of the chapter.

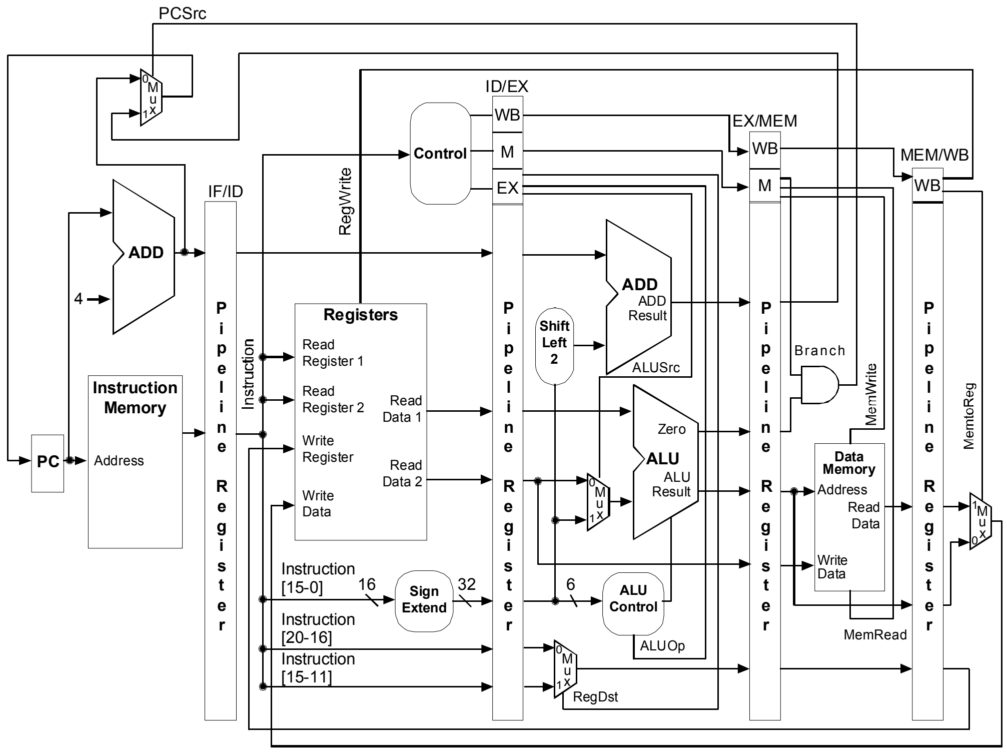


Figure 14.2 MIPS Pipelined Implementation.

14.2 Using VHDL to Synthesize the MIPS Processor Core

A VHDL-synthesis model of the MIPS single clock cycle model from Figure 14.1 will be developed in this section. This model can be used for simulation and implemented using the UP3 board.

The full 32-bit model requires a couple minutes to synthesize. When testing new changes you might want to use the faster functional (i.e. no timing delays) simulation approach before using a full timing delay model. This approach is commonly used on larger models with long synthesis and simulation times.

A two-level hierarchy is used in the model. MIPS.VHD is the top-level of the hierarchy. It consists of a structural VHDL model that connects the five behavioral modules. The five behavioral modules are already setup so that they correspond to the different stages for the MIPS. This makes it much easier to modify when the model is pipelined in later laboratory exercises. For many synthesis tools, hierarchy is also required to synthesize large logic designs. IFETCH.VHD is the VHDL submodule that contains instruction memory and the program counter. CONTROL.VHD contains the logic for the control unit.

IDECODE.VHD contains the multi-ported register file. EXECUTE.VHD contains the data and branch address ALUs. DMEMORY.VHD contains the data memory.

14.3 The Top-Level Module

The MIPS.VHD file contains the top-level design file. MIPS.VHD is a VHDL structural model that connects the five component parts of the MIPS. This module could also be created using the schematic editor and connecting the symbols for each VHDL submodule. The inputs are the clock and reset signals. The values of major busses and important control signals are copied and output from the top level so that they are available for easy display in simulations. Signals that are not outputs at the top level will occasionally not exist due to the compilers logic optimizations during synthesis.

```
                                    -- Top Level Structural Model for MIPS Processor Core
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY MIPS IS

   PORT( reset, clock                       : IN     STD_LOGIC;
      -- Output important signals to pins for easy display in Simulator
      PC                                    : OUT  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
      ALU_result_out, read_data_1_out, read_data_2_out,
      write_data_out, Instruction_out       : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
      Branch_out, Zero_out, Memwrite_out,
      Regwrite_out                          : OUT  STD_LOGIC );
END   TOP_SPIM;

ARCHITECTURE structure OF TOP_SPIM IS

   COMPONENT Ifetch
      PORT(   Instruction          : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              PC_plus_4_out        : OUT   STD_LOGIC_VECTOR( 9 DOWNTO 0 );
              Add_result           : IN    STD_LOGIC_VECTOR( 7 DOWNTO 0 );
              Branch               : IN    STD_LOGIC;
              Zero                 : IN    STD_LOGIC;
              PC_out               : OUT   STD_LOGIC_VECTOR( 9 DOWNTO 0 );
              clock,reset          : IN    STD_LOGIC );
   END COMPONENT;

   COMPONENT Idecode
      PORT(   read_data_1          : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              read_data_2          : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              Instruction          : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              read_data            : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              ALU_result           : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              RegWrite, MemtoReg   : IN    STD_LOGIC;
```

```
            RegDst              : IN    STD_LOGIC;
            Sign_extend         : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
            clock, reset        : IN    STD_LOGIC );
END COMPONENT;

COMPONENT control
   PORT(    Opcode              : IN    STD_LOGIC_VECTOR( 5 DOWNTO 0 );
            RegDst              : OUT   STD_LOGIC;
            ALUSrc              : OUT   STD_LOGIC;
            MemtoReg            : OUT   STD_LOGIC;
            RegWrite            : OUT   STD_LOGIC;
            MemRead             : OUT   STD_LOGIC;
            MemWrite            : OUT   STD_LOGIC;
            Branch              : OUT   STD_LOGIC;
            ALUop               : OUT   STD_LOGIC_VECTOR( 1 DOWNTO 0 );
            clock, reset        : IN    STD_LOGIC );
END COMPONENT;

COMPONENT  Execute
   PORT(    Read_data_1         : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
            Read_data_2         : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
            Sign_Extend         : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
            Function_opcode     : IN    STD_LOGIC_VECTOR( 5 DOWNTO 0 );
            ALUOp               : IN    STD_LOGIC_VECTOR( 1 DOWNTO 0 );
            ALUSrc              : IN    STD_LOGIC;
            Zero                : OUT   STD_LOGIC;
            ALU_Result          : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
            Add_Result          : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0 );
            PC_plus_4           : IN    STD_LOGIC_VECTOR( 9 DOWNTO 0 );
            clock, reset        : IN    STD_LOGIC );
END COMPONENT;


COMPONENT dmemory
   PORT(    read_data           : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
            address             : IN    STD_LOGIC_VECTOR( 7 DOWNTO 0 );
            write_data          : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
            MemRead, Memwrite   : IN    STD_LOGIC;
            Clock,reset         : IN    STD_LOGIC );
END COMPONENT;

                             -- declare signals used to connect VHDL components
SIGNAL PC_plus_4             :  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
SIGNAL read_data_1           : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL read_data_2           : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Sign_Extend           : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Add_result            : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL ALU_result            : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL read_data             : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL ALUSrc                : STD_LOGIC;
SIGNAL Branch                : STD_LOGIC;
SIGNAL RegDst                : STD_LOGIC;
```

```
    SIGNAL Regwrite          : STD_LOGIC;
    SIGNAL Zero              : STD_LOGIC;
    SIGNAL MemWrite          : STD_LOGIC;
    SIGNAL MemtoReg          : STD_LOGIC;
    SIGNAL MemRead           : STD_LOGIC;
    SIGNAL ALUop             : STD_LOGIC_VECTOR(  1 DOWNTO 0 );
    SIGNAL Instruction       : STD_LOGIC_VECTOR( 31 DOWNTO 0 );

BEGIN
                                 -- copy important signals to output pins for easy
                                 -- display in Simulator
  Instruction_out   <= Instruction;
  ALU_result_out    <= ALU_result;
  read_data_1_out   <= read_data_1;
  read_data_2_out   <= read_data_2;
  write_data_out    <= read_data WHEN MemtoReg = '1' ELSE ALU_result;
  Branch_out        <= Branch;
  Zero_out          <= Zero;
  RegWrite_out      <= RegWrite;
  MemWrite_out      <= MemWrite;
                                 -- connect the 5 MIPS components
 IFE : Ifetch
    PORT MAP (  Instruction      => Instruction,
                 PC_plus_4_out   => PC_plus_4,
                Add_result       => Add_result,
                Branch           => Branch,
                Zero             => Zero,
                PC_out           => PC,
                clock            => clock,
                reset            => reset );

  ID : Idecode
    PORT MAP (  read_data_1      => read_data_1,
                read_data_2      => read_data_2,
                Instruction      => Instruction,
                read_data        => read_data,
                ALU_result       => ALU_result,
                RegWrite         => RegWrite,
                MemtoReg         => MemtoReg,
                RegDst           => RegDst,
                Sign_extend      => Sign_extend,
                clock            => clock,
                reset            => reset );


  CTL:   control
    PORT MAP (  Opcode           => Instruction( 31 DOWNTO 26 ),
                RegDst           => RegDst,
                ALUSrc           => ALUSrc,
                MemtoReg         => MemtoReg,
                RegWrite         => RegWrite,
                MemRead          => MemRead,
                MemWrite        => MemWrite,
                Branch          => Branch,
                ALUop           => ALUop,
                clock           => clock,
                reset           => reset );

  EXE:  Execute
    PORT MAP ( Read_data_1     => read_data_1,
                Read_data_2     => read_data_2,
                Sign_extend     => Sign_extend,
                Function_opcode => Instruction( 5 DOWNTO 0 ),
                ALUOp           => ALUop,
                ALUSrc          => ALUSrc,
                Zero            => Zero,
                ALU_Result      => ALU_Result,
                Add_Result      => Add_Result,
                PC_plus_4       => PC_plus_4,
                Clock           => clock,
                Reset           => reset );

  MEM:  dmemory
    PORT MAP ( read_data       => read_data,
                address         => ALU_Result,
                write_data      => read_data_2,
                MemRead         => MemRead,
                Memwrite        => MemWrite,
                clock           => clock,
                reset           => reset );
END structure;
```

14.4 The Control Unit

The control unit of the MIPS shown in Figure 14.3 examines the instruction opcode bits and generates eight control signals used by the other stages of the processor. Recall that the high six bits of a MIPS instruction contain the opcode. The opcode value is used to determine the instruction type.

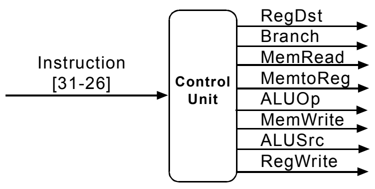


Figure 14.3 Block Diagram of MIPS Control Unit.

```
            -- control module (implements MIPS control unit)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY control IS
  PORT( Opcode            : IN    STD_LOGIC_VECTOR( 5 DOWNTO 0 );
        RegDst            : OUT   STD_LOGIC;
        ALUSrc            : OUT   STD_LOGIC;
        MemtoReg          : OUT   STD_LOGIC;
        RegWrite          : OUT   STD_LOGIC;
        MemRead           : OUT   STD_LOGIC;
        MemWrite          : OUT   STD_LOGIC;
        Branch            : OUT   STD_LOGIC;
        ALUop             : OUT   STD_LOGIC_VECTOR( 1 DOWNTO 0 );
        clock, reset      : IN    STD_LOGIC );
END control;


ARCHITECTURE behavior OF control IS


   SIGNAL  R_format, Lw, Sw, Beq     : STD_LOGIC;


BEGIN
                              -- Code to generate control signals using opcode bits
      R_format    <=  '1'  WHEN  Opcode = "000000"  ELSE '0';
      Lw          <=  '1'  WHEN  Opcode = "100011"  ELSE '0';
      Sw          <=  '1'  WHEN  Opcode = "101011"  ELSE '0';
      Beq         <=  '1'  WHEN  Opcode = "000100"  ELSE '0';

      RegDst      <=  R_format;
      ALUSrc      <=  Lw OR Sw;
      MemtoReg    <=  Lw;
      RegWrite    <=  R_format OR Lw;
      MemRead     <=  Lw;
      MemWrite    <=  Sw;
      Branch      <=  Beq;
      ALUOp( 1 )  <=  R_format;
      ALUOp( 0 )  <=  Beq;

  END behavior;
```

14.5 The Instruction Fetch Stage

The instruction fetch stage of the MIPS shown in Figure 14.4 contains the instruction memory, the program counter, and the hardware to increment the program counter to compute the next instruction address.

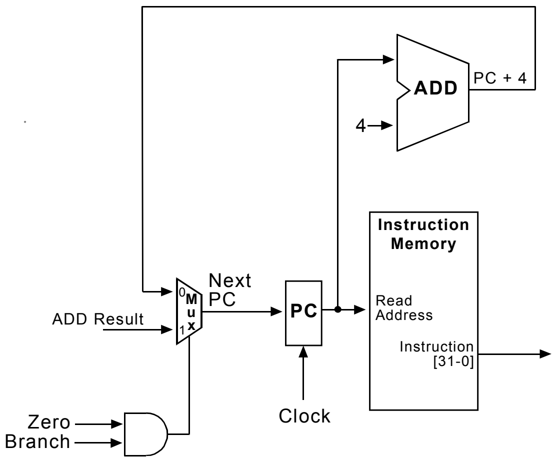


Figure 14.4 Block Diagram of MIPS Fetch Unit.

Instruction memory is implemented using the Altsyncram megafunction. 256 by 32 bits of instruction memory is available. This requires two of the Cyclone chip's M4K RAM memory blocks. Since the Altsyncram memory requires an address register, the PC register is actually implemented inside the memory block. A copy of the PC external to the memory block is also saved for use in simulation displays.

```
                                       -- Ifetch module (provides the PC and instruction
                                       --memory for the MIPS computer)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY altera_mf;
USE altera_mf.altera_mf_components.ALL;

ENTITY Ifetch IS
      PORT( SIGNAL Instruction      : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
            SIGNAL PC_plus_4_out    : OUT  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
            SIGNAL Add_result       : IN   STD_LOGIC_VECTOR( 7 DOWNTO 0 );
            SIGNAL Branch           : IN   STD_LOGIC;
            SIGNAL Zero             : IN   STD_LOGIC;
            SIGNAL PC_out           : OUT  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
            SIGNAL clock, reset     : IN   STD_LOGIC);
END Ifetch;
```

```
ARCHITECTURE behavior OF Ifetch IS
        SIGNAL PC, PC_plus_4          : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
        SIGNAL next_PC                : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
BEGIN
                                      --ROM for Instruction Memory
data_memory: altsyncram

    GENERIC MAP (
       operation_mode => "ROM",
       width_a => 32,
       widthad_a => 8,
       lpm_type => "altsyncram",
       outdata_reg_a => "UNREGISTERED",
                               -- Reads in mif file for initial data memory values
       init_file => "program.mif",
       intended_device_family => "Cyclone")

                               -- Fetch next instruction from memory using PC
    PORT MAP (
                clock0     =>  clock,
                address_a  => Mem_Addr,
                q_a  => Instruction
                );
                             -- Instructions always start on a word address - not byte
       PC(1 DOWNTO 0) <= "00";
                             -- copy output signals - allows read inside module
       PC_out            <= PC;
       PC_plus_4_out     <= PC_plus_4;
                              -- send word address to inst. memory address register
       Mem_Addr <= Next_PC;
                              -- Adder to increment PC by 4
         PC_plus_4( 9 DOWNTO 2 )  <= PC( 9 DOWNTO 2 ) + 1;
         PC_plus_4( 1 DOWNTO 0 )  <= "00";
                              -- Mux to select Branch Address or PC + 4
         Next_PC  <=  X"00" WHEN Reset = '1'  ELSE
                Add_result  WHEN ( ( Branch = '1' ) AND ( Zero = '1' ) )
                ELSE   PC_plus_4( 9 DOWNTO 2 );

                             -- Store PC in register and load next PC on clock edge
    PROCESS
         BEGIN
             WAIT UNTIL ( clock'EVENT ) AND ( clock = '1' );
             IF reset = '1' THEN
                    PC <= "0000000000" ;
             ELSE
                    PC( 9 DOWNTO 2 ) <= Next_PC;
             END IF;
    END PROCESS;
END behavior;
```

The MIPS program is contained in instruction memory. Instruction memory is automatically initialized using the program.mif file shown in Figure 14.5. This initialization only occurs once during download and not at a reset.

For different test programs, the appropriate machine code must be entered in this file in hex. Note that the memory addresses displayed in the program.mif file are word addresses while addresses in registers such as the PC are byte addresses. The byte address is four times the word address since a 32-bit word contains four bytes. Only word addresses can be used in the *.mif files.

```
-- MIPS Instruction Memory Initialization File


Depth = 256;
Width = 32;
Address_radix = HEX;
Data_radix = HEX;
Content
Begin

        -- Use NOPS for default instruction memory values
  [00..FF]: 00000000;              -- nop (sll r0,r0,0)
        -- Place MIPS Instructions here
        -- Note: memory addresses are in words and not bytes
        -- i.e. next location is +1 and not +4


  00: 8C020000;         -- lw $2,0 ;memory(00)=55
  01: 8C030001;         -- lw $3,1 ;memory(01)=AA
  02: 00430820;         -- add $1,$2,$3
  03: AC010003;         -- sw $1,3 ;memory(03)=FF
  04: 1022FFFF;         -- beq $1,$2,-4
  05: 1021FFFA;         -- beq $1,$1,-24

End;
```

Figure 14.5 MIPS Program Memory Initialization File, program.mif.

14.6 The Decode Stage

The decode stage of the MIPS contains the register file as shown in Figure 14.6. The MIPS contains thirty-two 32-bit registers. The register file requires a major portion of the hardware required to implement the MIPS. Registers are initialized to the register number during a reset. This is done to enable the use of shorter test programs that do not have to load all of the registers. A VHDL FOR...LOOP structure is used to generate the initial register values at reset.

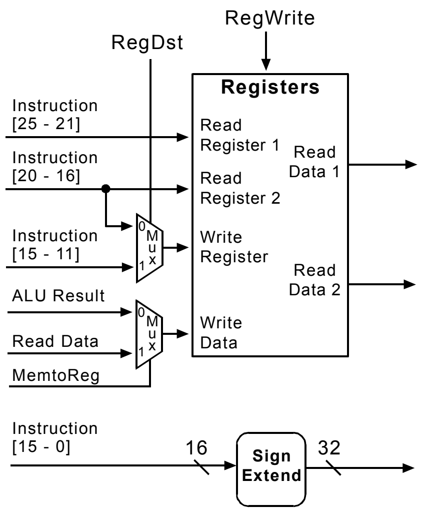


Figure 14.6 Block Diagram of MIPS Decode Unit.

```
                                    -- Idecode module (implements the register file for
LIBRARY IEEE;                       -- the MIPS computer)
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY Idecode IS
     PORT(    read_data_1     : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              read_data_2     : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              Instruction     : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              read_data       : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              ALU_result      : IN    STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              RegWrite        : IN    STD_LOGIC;
              MemtoReg        : IN    STD_LOGIC;
              RegDst          : IN    STD_LOGIC;
              Sign_extend     : OUT   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
              clock,reset     : IN    STD_LOGIC );
END Idecode;
```

```
ARCHITECTURE behavior OF Idecode IS
TYPE register_file IS ARRAY ( 0 TO 31 ) OF STD_LOGIC_VECTOR( 31 DOWNTO 0 );

    SIGNAL register_array: register_file;
    SIGNAL write_register_address        : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
    SIGNAL write_data                    : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
    SIGNAL read_register_1_address       : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
    SIGNAL read_register_2_address       : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
    SIGNAL write_register_address_1      : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
    SIGNAL write_register_address_0      : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
    SIGNAL Instruction_immediate_value : STD_LOGIC_VECTOR( 15 DOWNTO 0 );

BEGIN
    read_register_1_address      <= Instruction( 25 DOWNTO 21 );
    read_register_2_address      <= Instruction( 20 DOWNTO 16 );
    write_register_address_1     <= Instruction( 15 DOWNTO 11 );
    write_register_address_0     <= Instruction( 20 DOWNTO 16 );
    Instruction_immediate_value  <= Instruction( 15 DOWNTO 0 );
                                 -- Read Register 1 Operation
    read_data_1 <= register_array( CONV_INTEGER( read_register_1_address) );
                                 -- Read Register 2 Operation
    read_data_2 <= register_array( CONV_INTEGER( read_register_2_address) );
                                 -- Mux for Register Write Address
    write_register_address <= write_register_address_1
            WHEN RegDst = '1'  ELSE write_register_address_0;
                                 -- Mux to bypass data memory for Rformat instructions
    write_data <= ALU_result( 31 DOWNTO 0 )
            WHEN ( MemtoReg = '0' )  ELSE read_data;
                                 -- Sign Extend 16-bits to 32-bits
    Sign_extend <= X"0000" & Instruction_immediate_value
       WHEN Instruction_immediate_value(15) = '0'
       ELSE     X"FFFF" & Instruction_immediate_value;

PROCESS
    BEGIN
        WAIT UNTIL clock'EVENT AND clock = '1';
        IF reset = '1' THEN
                                 -- Initial register values on reset are register = reg#
                                 -- use loop to automatically generate reset logic
                                 -- for all registers
            FOR i IN 0 TO 31 LOOP
                register_array(i) <= CONV_STD_LOGIC_VECTOR( i, 32 );
            END LOOP;
                                 -- Write back to register - don't write to register 0
        ELSIF RegWrite = '1' AND write_register_address /= 0 THEN
            register_array( CONV_INTEGER( write_register_address)) <= write_data;
        END IF;
    END PROCESS;
END behavior;
```

14.7 The Execute Stage

The execute stage of the MIPS shown in Figure 14.7 contains the data ALU and a branch address adder used for PC-relative branch instructions. Multiplexers that select different data for the ALU input are also in this stage.

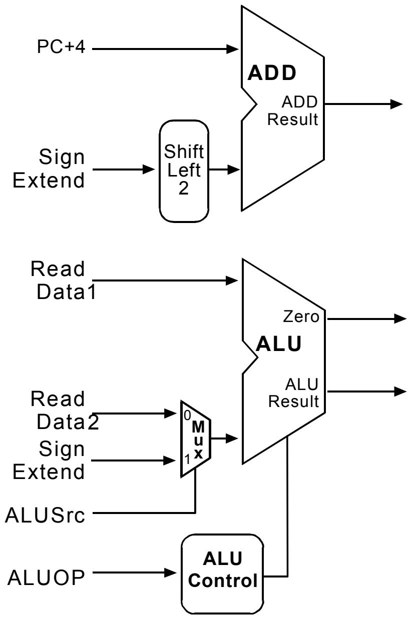


Figure 14.7 Block Diagram of MIPS Execute Unit.

```
-- Execute module (implements the data ALU and Branch Address Adder
-- for the MIPS computer)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY Execute IS
    PORT( Read_data_1     : IN   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
          Read_data_2     : IN   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
          Sign_extend     : IN   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
          Function_opcode : IN   STD_LOGIC_VECTOR( 5 DOWNTO 0 );
          ALUOp           : IN   STD_LOGIC_VECTOR( 1 DOWNTO 0 );
          ALUSrc          : IN   STD_LOGIC;
          Zero            : OUT  STD_LOGIC;
          ALU_Result      : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
          Add_Result      : OUT  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
          PC_plus_4       : IN   STD_LOGIC_VECTOR( 7 DOWNTO 0 );
          clock, reset    : IN   STD_LOGIC );
END Execute;
```

```
ARCHITECTURE behavior OF Execute IS
SIGNAL Ainput, Binput            : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL ALU_output_mux            : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Branch_Add                : STD_LOGIC_VECTOR( 8 DOWNTO 0 );
SIGNAL ALU_ctl                   : STD_LOGIC_VECTOR( 2 DOWNTO 0 );
BEGIN
     Ainput <= Read_data_1;
                                        -- ALU input mux
    Binput <= Read_data_2
         WHEN ( ALUSrc = '0' )
         ELSE  Sign_extend( 31 DOWNTO 0 );
                                        -- Generate ALU control bits
    ALU_ctl( 0 ) <= ( Function_opcode( 0 ) OR Function_opcode( 3 ) ) AND ALUOp(1 );
    ALU_ctl( 1 ) <= ( NOT Function_opcode( 2 ) ) OR (NOT ALUOp( 1 ) );
    ALU_ctl( 2 ) <= ( Function_opcode( 1 ) AND ALUOp( 1 )) OR ALUOp( 0 );
                                        -- Generate Zero Flag
    Zero <= '1'
         WHEN ( ALU_output_mux( 31 DOWNTO 0 ) =  X"00000000"  )
         ELSE '0';
                                        -- Select ALU output for SLT
    ALU_result <=  X"0000000" & B"000" & ALU_output_mux( 31 )
         WHEN  ALU_ctl = "111"
         ELSE   ALU_output_mux( 31 DOWNTO 0 );
                                        -- Adder to compute Branch Address
    Branch_Add <= PC_plus_4( 9 DOWNTO 2 ) +  Sign_extend( 7 DOWNTO 0 ) ;
    Add_result    <= Branch_Add( 7 DOWNTO 0 );

PROCESS ( ALU_ctl, Ainput, Binput )
    BEGIN
                                  -- Select ALU operation
    CASE ALU_ctl IS
                                        -- ALU performs ALUresult = A_input AND B_input
          WHEN "000"     => ALU_output_mux   <= Ainput AND Binput;
                                        -- ALU performs ALUresult = A_input OR B_input
         WHEN "001"      => ALU_output_mux   <= Ainput OR Binput;
                                        -- ALU performs ALUresult = A_input + B_input
         WHEN "010"      => ALU_output_mux   <= Ainput + Binput;
                                        -- ALU performs ?
         WHEN "011"      => ALU_output_mux   <= X"00000000" ;
                                        -- ALU performs ?
         WHEN "100"      => ALU_output_mux   <= X"00000000" ;
                                        -- ALU performs ?
         WHEN "101"      => ALU_output_mux   <= X"00000000" ;
                                        -- ALU performs ALUresult = A_input - B_input
         WHEN "110"      => ALU_output_mux   <= Ainput - Binput;
                                        -- ALU performs SLT
         WHEN "111"      => ALU_output_mux   <= Ainput - Binput ;
         WHEN OTHERS => ALU_output_mux   <= X"00000000" ;
    END CASE;
  END PROCESS;
 END behavior;
```

14.8 The Data Memory Stage

The data memory stage of the MIPS core shown in Figure 14.8 contains the data memory. To speed synthesis and simulation, data memory is limited to 256 locations of 32-bit memory. Data memory is implemented using the Altsyncram megafunction. Memory write cycle timing is critical in any design. The Altsyncram function requires an internal address register with a clock. In this design, the falling clock edge is used to load the data memories internal address register. The rising clock edge starts the next instruction. Two M4K RAM blocks are used for data memory. Two M4K RAM blocks are also used for the 32-bit instruction memory.

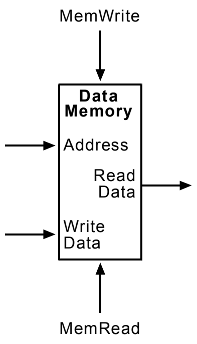


Figure 14.8 Block Diagram of MIPS Data Memory Unit.

```
--  Dmemory module (implements the data
-- memory for the MIPS computer)

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;
LIBRARY altera_mf;
USE altera_mf.atlera_mf_components.ALL;

ENTITY dmemory IS
    PORT(  read_data              : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
           address                : IN   STD_LOGIC_VECTOR( 7 DOWNTO 0 );
           write_data             : IN   STD_LOGIC_VECTOR( 31 DOWNTO 0 );
           MemRead, Memwrite      : IN   STD_LOGIC;
           clock, reset           : IN   STD_LOGIC );
END dmemory;

ARCHITECTURE behavior OF dmemory IS
SIGNAL write_clock : STD_LOGIC;
BEGIN
```

```
    data_memory: altsyncram
    GENERIC MAP (
        operation_mode => "SINGLE_PORT",
        width_a => 32,
        widthad_a => 8,
        lpm_type => "altsyncram",
        outdata_reg_a => "UNREGISTERED",
                                -- Reads in mif file for initial data memory values
        init_file => "dmemory.mif",
        intended_device_family => "Cyclone"lpm_widthad    =>      8
    )
    PORT MAP (
        wren_a => memwrite,
        clock0 => write_clock,
        address_a => address,
        data_a => write_data,
        q_a => read_data        );
                                -- Load memory address & data register with write clock
        write_clock <= NOT clock;
END behavior;
```

MIPS data memory is initialized to the value specified in the file dmemory.mif shown in Figure 14.9. Note that the address displayed in the dmemory.mif file is a word address and not a byte address. Two values, 0x55555555 and 0xAAAAAAA, at byte address 0 and 4 are used for memory data in the short test program. The remaining locations are all initialized to zero.

```
-- MIPS Data Memory Initialization File
Depth = 256;
Width = 32;
Content
Begin
                -- default value for memory
  [00..FF] : 00000000;
                -- initial values for test program
  00 : 55555555;
  01 : AAAAAAAA;
End;
```

Figure 14.9 MIPS Data Memory Initialization File, dmemory.mif.

14.9 Simulation of the MIPS Design

The top-level file MIPS.VHD is compiled and used for simulation of the MIPS. It uses VHDL component instantiations to connect the five submodules. The values of major busses and important control signals are output at the top level for use in simulations. A reset is required to start the simulation with PC = 0. A clock with a period of approximately 200ns is required for the simulation.

Memory is initialized only at the start of the simulation. A reset does not re-initialize memory.

The execution of a short test program can be seen in the MIPS simulation output shown in Figure 14.10. The program loads two registers from memory with the LW instructions, adds the registers with an ADD, and stores the sum with SW. Next, the program does not take a BEQ conditional branch with a false branch condition. Last, the program loops back to the start of the program at PC = 000 with another BEQ conditional branch with a true branch condition.

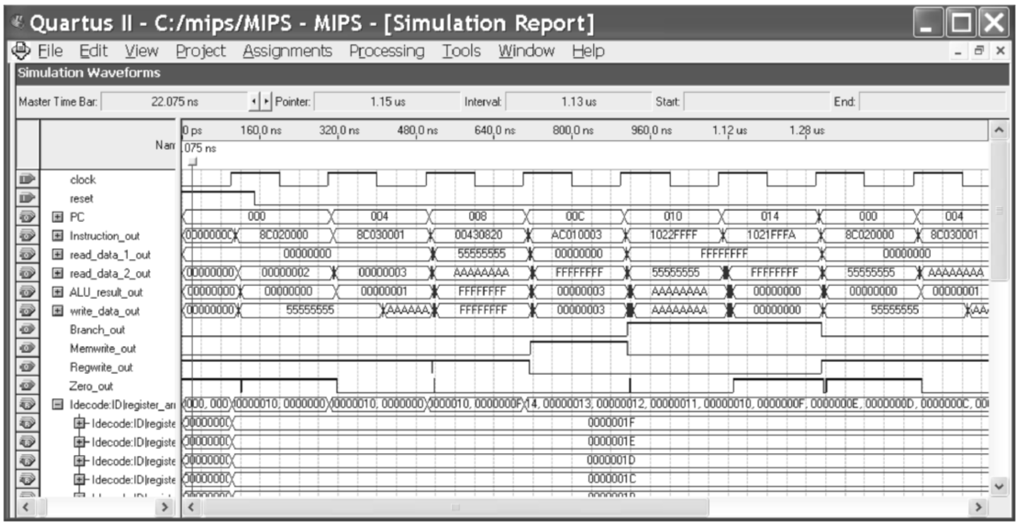
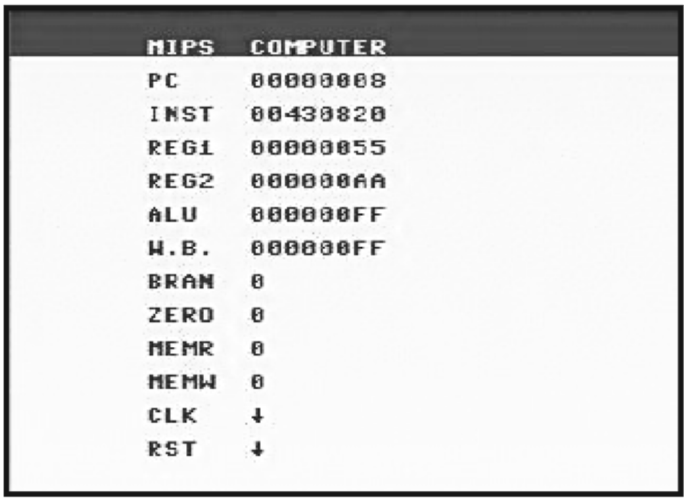


Figure 14.10 Simulation of MIPS test program.

14.10 MIPS Hardware Implementation on the FPGA Board

A special version of the top level of the MIPS, VIDEO_MIPS.VHD, is identical to MIPS.VHD except that it also contains a VGA video output display driver. As seen in Figure 14.11, this driver displays the hexadecimal value of major busses in the MIPS processor on a monitor. The video character generation technique used is discussed in Chapter 10. On the FPGA boards, it also displays the PC in the LCD or LED displays. All FPGA boards use pushbuttons for the clock and reset inputs. The clock pushbutton toggles the processor clock so you can see the data changes occurring on each clock edge as you step through MIPS machine instructions. This top-level module should be used instead of MIPS.VHD after the design has been debugged in simulations. The final design with video output is then downloaded to the FPGA chip on the board. The video driver uses two M4K RAM embedded memory blocks for format and character font data.

After simulation with MIPS.VHD, recompile using VIDEO_MIPS.VHD and download the design to the FPGA board for hardware verification. Attach a VGA monitor to the board's VGA connector. Any changes or additions made to top level signal names in MIPS.VHD and other modules as suggested in the exercises will need to also be cut and pasted into VIDEO_MIPS.VHD.



```
MIPS  COMPUTER
PC    00000008
INST  00430820
REG1  00000055
REG2  000000AA
ALU   000000FF
W.B.  000000FF
BRAN  0
ZERO  0
MEMR  0
MEMW  0
CLK   ↓
RST   ↓
```

Figure 14.11 MIPS with Video Output generated by UP3 Board.

14.11 For Additional Information

The MIPS processor design and pipelining are described in the widely-used Patterson and Hennessy textbook, *Computer Organization and Design The Hardware/Software Interface*, Third Edition, Morgan Kaufman Publishers, 2005. The MIPS instructions are described in Chapter 2 and Appendix A of this text. The hardware design of the MIPS, used as the basis for this model, is described in Chapters 5 and 6 of the Patterson and Hennessy text.

SPIM, a free MIPS R2000 assembly language assembler and PC-based simulator developed by James Larus, is available free from http://www.cs.wisc.edu/~larus/spim.html . The reference manual for the SPIM simulator contains additional explanations of all of the MIPS instructions.

The MIPS instruction set and assembly language programming is also described in J. Waldron, *Introduction to RISC Assembly Language Programming*, Addison Wesley, 1999, and Kane and Heinrich, *MIPS RISC Architecture*, Prentice Hall, 1992.

A SMALLER VERSION OF THE MIPS PROCESSOR FOR THE UP1 AND UP2 IS PROVIDED ON THE DVD. 8-BIT DATA IS USED WITH ONLY 8 REGISTERS.

14.12 Laboratory Exercises

1. Use VHDL to synthesize the MIPS single clock cycle design in the file TOP_SPIM.VHD. After synthesis and simulation perform the following steps:

   Display and print the timing diagram from the simulation. Verify that the information on the timing diagram shows that the hardware is functioning correctly. Examine the test program in IFETCH.VHD. Look at the program counter, the instruction bus, the register file and ALU outputs, and control signals on the timing diagram and carefully follow the execution of each instruction in the test program. Label the important values for each instruction on the timing diagram and attach a short write-up explaining in detail what the timing diagram shows relative to each instruction's execution and correct operation.

   Return to the simulator and run the simulation again. Examine the ALU output in the timing diagram window. Zoom in on the ALU output during execution of the add instruction and see what happens when it changes values. Explain exactly what is happening at this point. Hint: Real hardware has timing delays.

2. Recompile the MIPS model using the VIDEO_MIPS.VHD file, which generates video output. Download the design to the FPGA board. Attach a VGA monitor to the FPGA board. Single step through the program using the pushbuttons.

3. Write a MIPS test program for the AND, OR, and SUB instructions and run it on the VHDL MIPS simulation. These are all R-format instructions just like the ADD instruction. Modifications to the memory initialization files, program.mif and dmemory.mif, (i.e. only if you use data from memory in the test program) will be required. Registers have been preloaded with the register number to make it easy to run short test programs.

4. Add and test the JMP instruction. The JMP or jump instruction is not PC-relative like the branch instructions. The J-format JMP instruction loads the PC with the low 26 bits of the instruction. Modifications to the existing VHDL MIPS model will be required. For a suggested change, see the hardware modifications on page 313 of *Computer Organization and Design The Hardware/Software Interface*.

5. Add and test the BNE, branch if not equal, instruction. Modifications to the existing VHDL MIPS model will be required. Hint: Follow the implementation details of the existing BEQ, branch if equal, instruction and a change to add BNE should be obvious. Both BEQ and BNE must function correctly in a simulation. Be sure to test both the branch and no branch cases.

6. Add and test the I-format ADDIU, add immediate unsigned, instruction. Modifications to the existing VHDL MIPS model will be required.

7. Add and test the R-format SLT, set if less than, instruction. As an example SLT $1, $2, $3 performs the operation, If $2<$3 Then $1 = 1 Else $1 = 0. SLT is used before BEQ or BNE to implement the other branch conditions such as less than or greater.

8. Pipeline the MIPS VHDL simulation. Test your VHDL model by running a simulation of the example program shown in Figure 6.21 using the pipeline hardware shown in Figure 6.27 in *Computer Organization and Design The Hardware/Software Interface*. To minimize changes, pipeline registers must be placed in the VHDL module that generates the input to the pipeline. As an example, all of the pipeline registers that store control signals must be placed in the control module. Synthesize and check the control module first, since it is simple to see if it works correctly when you add the pipeline flip-flops. Use the following notation which minimizes changes to create the new pipeline register signals, add a "D_" in front of the signal name to indicate it is the input to a D flip-flop used in a pipeline register. Signals that go through two D flip-flops would be "DD_" and three would be "DDD_". As an example, *instruction* would be the registered version of the signal, D_*instruction.*

   Add pipeline registers to the existing modules that generate the inputs to the pipeline registers shown in the text. This will prevent adding more modules and will not require extensive changes to the MIP.VHD module. Add signal and process statements to model the pipeline modules – see the PC in the ifetch.vhd module for an example of how this can work. A few muxes may have to be moved to different modules.

   The control module should contain all of the control pipeline registers – 1, 2, or 3 stages of pipeline registers for control signals. Some control signals must be reset to zero, so use a D flip-flop with a synchronous reset for these pipeline registers. This generates a flip-flop with a Clear input that will be tied to Reset. Critical pipeline registers with control signals such as regwrite or memwrite should be cleared at reset so that the pipeline starts up correctly. The MIPS instruction ADD $0, $0, $0 is all zeros and does not modify any values in registers or memory. It is used to initialize the IF/ID pipeline at reset. Pipeline registers for instruction and data memory outputs can also be added by modifying options in the Altsyncram megafunction.

   The data memory clocking scheme might also change with pipelining. In Dmemory.vhd, the data memory address and data inputs are already pipelined inside the altsyncram function used for data memory (this is why it has a clock input). You will need to take this into account when you pipeline your design. High speed memory writes almost always require a clock and the design in the textbook skips over this point – since they do not have their design running on real hardware. As an example, in the Quartus software you can’t even have altsyncram memory without a clock!

   Currently in the original single cycle design, data memory uses NOT CLOCK as the clock input so that there is time to get both the correct ALU result loaded into the memories internal address and data pipeline registers (first half of clock cycle) and write to memory (second half of clock cycle).

Once you pipeline the model, you will probably want to have your data memory clock input use CLOCK instead of NOT CLOCK for the fastest clock cycle time. With NOT CLOCK you would be loading the ALU Result into the pipeline register in the middle of the clock cycle (not the end) – so it would slow down the clock cycle time on real hardware.

Since there is already a pipeline register in the data memory inputs, don't add another one in the address or data input paths to data memory, if you switch NOT CLOCK to CLOCK. You will still need to delay the ALU result two clocks (with two pipeline registers) for the register file write back operation.

Sections 6.2 and 6.3 of Computer Organization and Design The Hardware/Software Interface contain additional background information on pipelining.

9. Once the MIPS is pipelined as in problem 8, data hazards can occur between the five instructions present in the pipeline. As an example consider the following program:

```
Sub    $2,$1,$3
Add    $4,$2,$5
```

The subtract instruction stores a result in register 2 and the following add instruction uses register 2 as a source operand. The new value of register 2 is written into the register file by SUB $2,$1,$3 in the write-back stage after the old value of register 2 was read out by ADD $4,$2,$5 in the decode stage. This problem is fixed by adding two forwarding muxes to each ALU input in the execute stage. In addition to the existing values feeding in the two ALU inputs, the forwarding multiplexers can also select the last ALU result or the last value in the data memory stage. These muxes are controlled by comparing the rd, rt, and rs register address fields of instructions in the decode, execute, or data memory stages. Instruction rd fields will need to be added to the pipelines in the execute, data memory, and write-back stages for the forwarding compare operations. Since register 0 is always zero, do not forward register 0 values.

Add forwarding control to the pipelined model developed in problem 8. Test your VHDL model by running a simulation of the example program shown in Figure 6.29 using the hardware shown in Figures 6.32 of *Computer Organization and Design The Hardware/Software Interface* by Patterson and Hennessy.

Two forwarding multiplexers must also be added to the Idecode module so that a register file write and read to the same register work correctly in one clock cycle. If the register file write address equals one of the two read addresses, the register file write data value should be forwarded out the appropriate read data port instead of the normal register file read data value. Section 6.4 of *Computer Organization and Design The Hardware/Software Interface* contains additional background information on forwarding.

10. Add LW/SW forwarding to the pipelined model. This will allow an LW to be followed by an SW that uses the same register. It is possible since the MEM/WB register contains the load instruction register write data in time for use in the MEM stage of the store. Write a test program and verify correct operation in a simulation.

11. When a branch is taken, several of the instructions that follow a branch have already been loaded into the pipeline. A process called flushing is used to prevent the execution of these instructions. Several of the pipeline registers are cleared so that these instructions do not store any values to registers or memory or cause a forwarding operation. Add branch flushing to the pipelined MIPS VHDL model as shown in Figures 6.38 of the Computer Organization and Design The Hardware/Software Interface by Patterson and Hennessy. Note that two new forwarding multiplexers at the register file outputs (not shown in the Figure, currently at ALU inputs) are needed to eliminate the new Branch data hazards that appear when the branch comparator is moved into to the decode stage. Section 6.6 *of Computer Organization and Design The Hardware/Software Interface* contains additional background information on branch hazards.

12. Use the timing analyzer to determine the maximum clock rate for the pipelined MIPS implementation, verify correct operation at this clock rate in a simulation, and compare the clock rate to the original non-pipelined MIPS implementation.

13. Redesign the pipelined MIPS VHDL model so that branch instructions have 1 delay slot as seen in Figure 6.40 (i.e. one instruction after the branch is executed even when the branch is taken). Rewrite the VHDL model of the MIPS and test the program from the problem 10 assuming 1 delay slot. Move instructions around and add nops if needed.

14. Add the overflow exception hardware suggested at the end of Chapter 6 in Figure 6.42 of *Computer Organization and Design The Hardware/Software Interface* by Patterson and Hennessy. Add an overflow circuit that produces the exception with a test program containing an ADD instruction that overflows. Display the PC and the trap address in your simulation. For test and simulation purposes make the exception address 40 instead of 40000040. Section 6.8 of *Computer Organization and Design The Hardware/Software Interface* contains additional background information on exceptions.

15. Investigate using two Altsyncram memory blocks to implement the register file in IDECODE. A single Altsyncram block can be configured to do a read and write in one clock cycle (dual port). To perform two reads, use two Altsyncrams that contain the same data (i.e. always write to both blocks).

16. Add the required instructions to the model to run the MIPS bubble sort program from Chapter 3 of *Computer Organization and Design The Hardware/Software Interface*.

After verifying correct operation with a simulation, download the design to the FPGA board and trace execution of the program using the video output. Sort this four element array 4, 3, 5, 1.

17. Add programmed keyboard input and video output to the sort program from the previous problem using the keyboard, vga_sync, and char_rom FPGAcores. Use a dedicated memory location to interface to I/O devices. Appendix A.36-38 of *Computer Organization and Design The Hardware/Software Interface* contains an explanation of MIPS memory-mapped terminal I/O.

18. The MIPS VHDL model was designed to be easy to understand. Investigate various techniques to increase the clock rate such as using two dual-port memory blocks for the register file, moving hardware to different pipeline stages to even out delays, or changing the way memory is clocked. Additional fitter effort settings may also help. Use the timing analysis tools to evaluate design changes.

19. Develop a VHDL synthesis model for another RISC processor's instruction set. Possible choices include the Nios, Microblaze, Picoblaze, PowerPC, ARM, SUN SPARC, the DEC ALPHA, and the HP PARISC. DVD Appendix D of Computer Organization and Design The Hardware/Software Interface contains information on several RISC processors. Earlier hardware implementations of the commercial RISC processors designed before they became superscalar are more likely to fit on a FPGA.