Single Cycle (MIPS Processor) CPU

Anthony Ramos

Professor Isidor Gertner

CSC343/342

Spring 2021

Contents

Objective

The objective of this lab is to realize a Single Cycle MIPS processor. It will be capable of executing the MIPS instructions defined in table 1 below.

Instruction	Mnemonic	Format	Operation
Add	**add**	R	**R[rd] = R[rs] + R[rt]**
Add Immediate	**addi**	I	**R[rt] = R[rs] + SignExtImm**
Add Immediate Unsigned	**addiu**	I	**R[rt] = R[rs] + SignExtImm**
Add Unsigned	**addu**	R	**R[rd] = R[rs] + R[rt]**
Subtract	**sub**	R	**R[rd] = R[rs] - R[rt]**
Subtract Unsigned	**subu**	R	**R[rd] = R[rs] - R[rt]**
Multiply	**mult**	R	**R[rd] = R[rs] * R[rt]**
Divide	**div**	R	**R[rd] = R[rs] / R[rt]**
And	**and**	R	**R[rd] = R[rs] & R[rt]**
And Immediate	**andi**	I	**R[rt] = R[rs] & ZeroExtImm**
Nor	**nor**	R	**R[rd] = ~(R[rs] | R[rt])**
Or Immediate	**ori**	I	**R[rt] = R[rs] | ZeroExtImm**
Shift Left	**sll**	R	**R[rd] = R[rs] << shamt**
Shift Right	**srl**	R	**R[rd] = R[rs] >> shamt**
Shift Right Arithmetic	**sra**	R	**R[rd] = R[rs] >>> shamt**
Store Word	**sw**	I	**M[R[rs]+SignExtImm] = R[rt]**
Load Word	**lw**	I	**R[rt] = M[R[rs]+SignExtImm]**
Branch if Equal	**beq**	I	**PC = PC+4+[SignExtImm,2b00] If R[rs] == R[rt]**
Branch if Not Equal	**bne**	I	**PC = PC+4+[SignExtImm,2b00] If R[rs] != R[rt]**
Jump	**j**	J	**PC += JumpAddress << 2**

Table 1: Executable instructions of the Single Cycle MIPS processor

Introduction

The workings of a Single Cycle CPU is rather simple in design. In such a CPU, just one instruction is executed per cycle – hence the name. To better understand the CPU design, we must first understand the MIPS instructions that will be implemented. Namely, there are three types of instructions that will need to be implemented: R, I, and J type instructions. They each formatted and defined differently.

R-Type Instructions

Let's first consider the format of *R-Type* instructions which are divided into six fields:

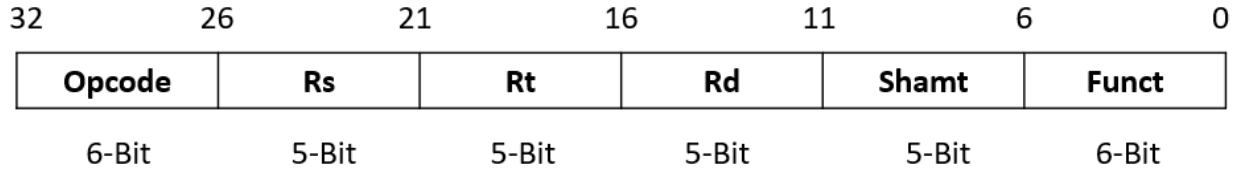

Figure 1: Structure of a *R-Type* instruction

Where the Opcode is 0b000000 for all R-type instructions. Rs, Rt, and Rt fields are the indices (i.e., addresses) of the two source and destination registers respectively. The Shamt field specifies the amount of bits to shift the contents of a register (applies only to shift instructions `sll`, `srl`, and `sra`). Lastly, the Funct field distinguishes among the R-Type instructions. In this lab, we are implementing 11 R-Type instructions.

I-Type Instructions

Let's now consider *I-Type* instructions which is divided into four fields:

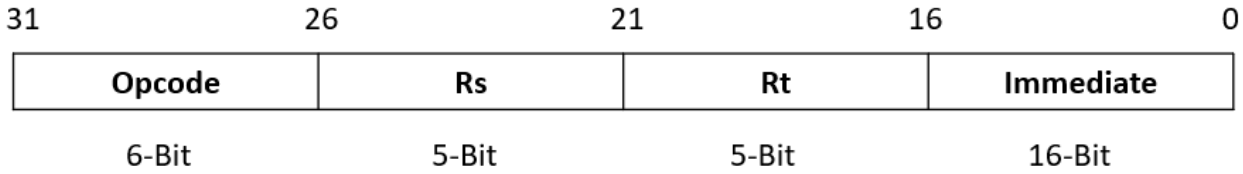

Figure 2: Structure of a *I-Type* instruction

Where the Opcode distinguishes among the I-Type instructions. Rs and Rt are again source indices (i.e., addresses) of two source registers. Lastly, the last 16-Bits are called the immediate and is either zero extended or sign extended to a 32-Bit value depending on the operation (See Table 1). In this lab, we are implementing 8 I-Type instructions.

J-Type Instructions

Lastly, we will examine the J-Type instruction format which is comprised of only two fields: An opcode field to distinguish between the four types of Jump instructions (`j`, `jal`, `jalr`, and `jr`) and the 26-Bit jump target index. However, we are only concerned with the `j` jump instruction.

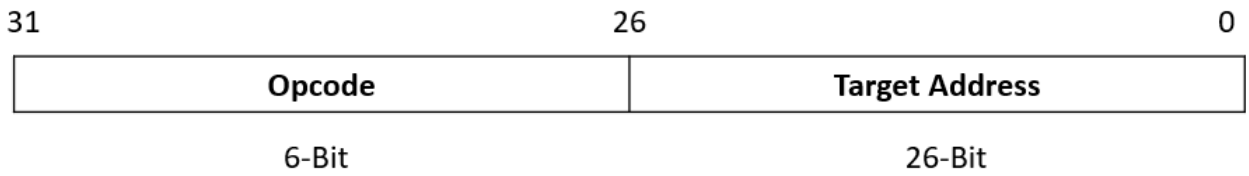

Figure 3: Structure of a *J-Type* instruction

To incorporate the jump instruction into the CPU design, we require a 32-bit address but initially given a 26-Bit target address. The 26-Bit jump target index must be shifted 2 bits to the left (i.e., add 00 to be the lower bits). This will result in a 28-bits index. Next, the upper four bits of next instruction address (i.e. PC + 4) become the high order bits of the jump address which results in a 32-Bit address as shown in figure 4 below.

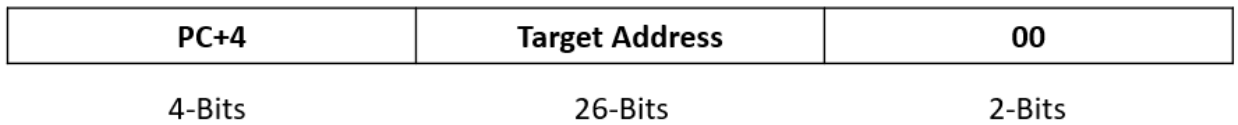

Figure 4: Structure of 32-Bit Target Address instruction

Design Breakdown

Having understood the breakdown of the instruction types that will be implemented, we must now consider *how* each instruction will be implemented. It can be inferred that due to the different types of instruction formats, not all instructions will be handled the same way. Hence, we require some mechanism to organize and control the data paths for each instruction.

The CPU Control Unit (Abstract View)

A control unit will be required to manage *control signals* in order to define the data paths for all the instructions defined in table 1. An abstract view of the control unit is shown in figure 5 below. In addition to defining a data path for a given instruction, the control signals serve as inputs to various multiplexers used throughout the CPU to enforce the data path. This aspect will be discussed in more detail later.

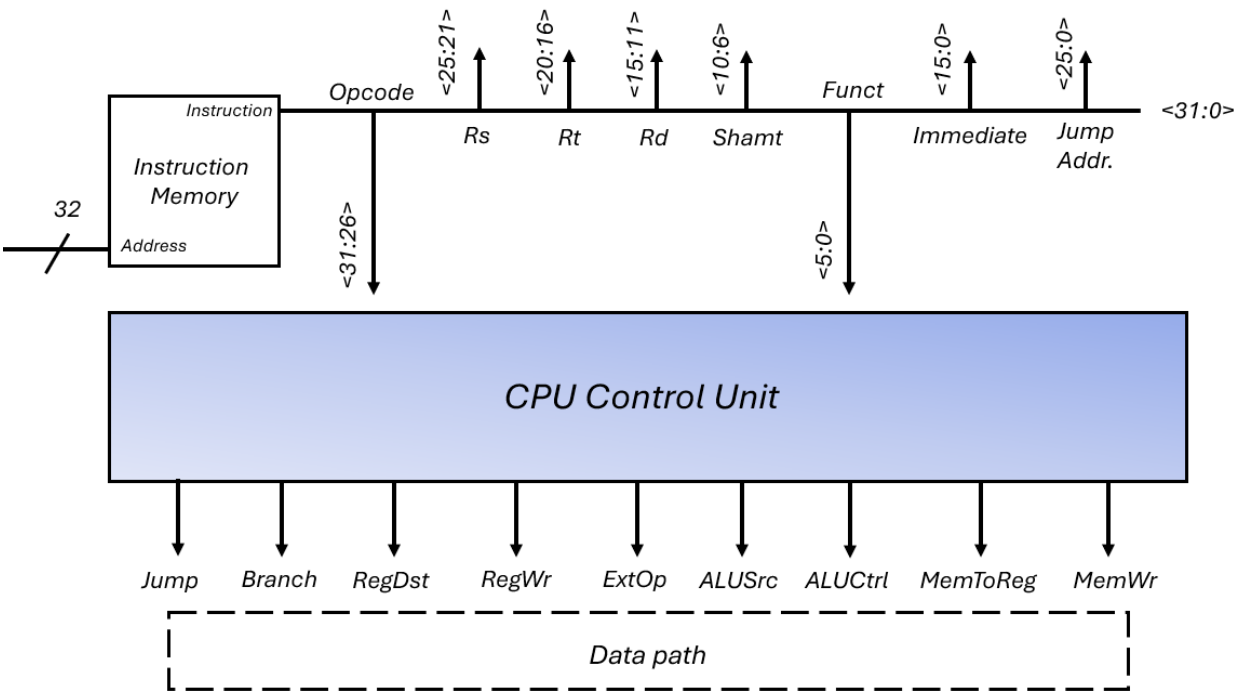

Figure 5: Abstract view of CPU controller

Control Signal Definitions

There are 9 control signals in total that need to be set accordingly. Not all instructions however, need all 9 control signals to be defined. Table 2 shows the meaning of eight of these control signals when they are asserted (i.e., 1) and disserted (i.e., 0).

Signal	Value = 0	Value = 1
`Branch`	PC <= PC + 4	PC <= PC + 4 + {SignExt(Imm), <<2}
`ALUSrc`	Operand 2 <= BusB	Operand 2 <= SignExt(Imm)
`ExtOP`	Zero extend	Sign extend
`RegWr`	No write	The register on the *BusW* input is written with the value on the *Write data* input
`RegDst`	ALUResult <= Rt	ALUResult <= Rd
`MemWr`	No write	Write to memory
`MemRd`	No Read	Read from memory
`MemtoReg`	The value fed to the register *BusW* input comes from the ALU	The value fed to the register *BusW* input comes from the Data Memory
`Jump`	No Jump	Jump to target address

Table 2: Meaning of control signals

The `ALUCtrl` control signal is the ninth signal and is a 4-Bit vector that serves as an input to the Arithmetic Logic Unit (ALU) to perform the necessary arithmetic operation. Table 3 below summarizes all 20 instructions with their corresponding `ALUCtrl` value. Note that the jump instruction is the only instruction that does not depend on the `ALUCtrl` signal.

ALUCtrl	Operation
0000	**`add/addi/addu/addiu/sw/lw`**
0001	**`sub/subu/beq/bne`**
0010	**`mult`**
0011	**`div`**
0100	**`and/andi`**
0101	**`ori`**
0110	**`nor`**
0111	**`sll`**
1000	**`srl`**
1001	**`sra`**

Table 3: Summary of ALU operations

Summary of Control Signals

Table 4 below summarizes the data paths (i.e., the control signal configurations) of every instruction that is to be implemented. For simplicity, R-Type, I-Type, Memory Access, Branch, and Jump instructions are in blue, purple, green, orange, and amber respectively. The setting of control lines is completely determined by the Opcode field if a 32-Bit instruction. For example, the instructions in blue are R-Type and all have the same control signal values.

Operation	Branch	ALUSrc	ExtOP	RegWr	RegDst	MemWr	MemRd	MemtoReg	ALUCtrl	Jump
Add	0	0	X	1	1	0	0	0	0000	0
Addu	0	0	X	1	1	0	0	0	0000	0
Addi	0	1	1	1	0	0	0	0	0000	0
Addiu	0	1	1	1	0	0	0	0	0000	0
Sub	0	0	X	1	1	0	0	0	0001	0
Subu	0	0	X	1	1	0	0	0	0001	0
Mul	0	0	X	1	1	0	0	0	0010	0
Div	0	0	X	1	1	0	0	0	0011	0
And	0	0	X	1	1	0	0	0	0100	0
Andi	0	1	0	1	0	0	0	0	0100	0
Ori	0	1	0	1	0	0	0	0	0101	0
Nor	0	0	X	1	1	0	0	0	0110	0
Sll	0	0	X	1	1	0	0	0	0111	0
Srl	0	0	X	1	1	0	0	0	1000	0
Sra	0	0	X	1	1	0	0	0	1001	0
Lw	0	1	1	1	0	0	1	1	0000	0
Sw	0	1	1	0	X	1	0	X	0000	0
BEQ	1	0	1	0	X	0	0	X	0001	0
BNE	1	0	1	0	X	0	0	X	0001	0
Jump	0	X	X	0	X	0	0	X	XXXX	1

Table 4: Summary of control signals

Identifying R-Type, I-Type, and J-Type Instructions

The R-Type instructions are defined by the same opcode field “000000”. For this reason, we require another mechanism to distinguish between each R-Type instruction. This can be achieved by utilizing the 6-Bit `funct` field of the 32-Bit instruction.

R-Type Instruction	Funct Field
`add`	100000
`addu`	100001
`sub`	100010
`subu`	100011
`mul`	011000
`div`	011010
`and`	100100
`nor`	100111
`sll`	000000
`srl`	000010
`sra`	000011

Table 5: Summary of `funct` codes for R-type instructions

Table 6 summarizes the opcode field values and corresponding instructions.

Instruction	Opcode
`addi`	001000
`addiu`	001001
`andi`	001100
`ori`	001101
`lw`	100011
`sw`	101011
`beq`	000100
`bne`	000101
`j`	000010

Table 5: Summary of opcodes for I-type and J-type instructions

Design Summary

Considering all details discussed so far, the intended design to be realized utilizing Quartus Prime software and VHDL is shown in figure 6 below.

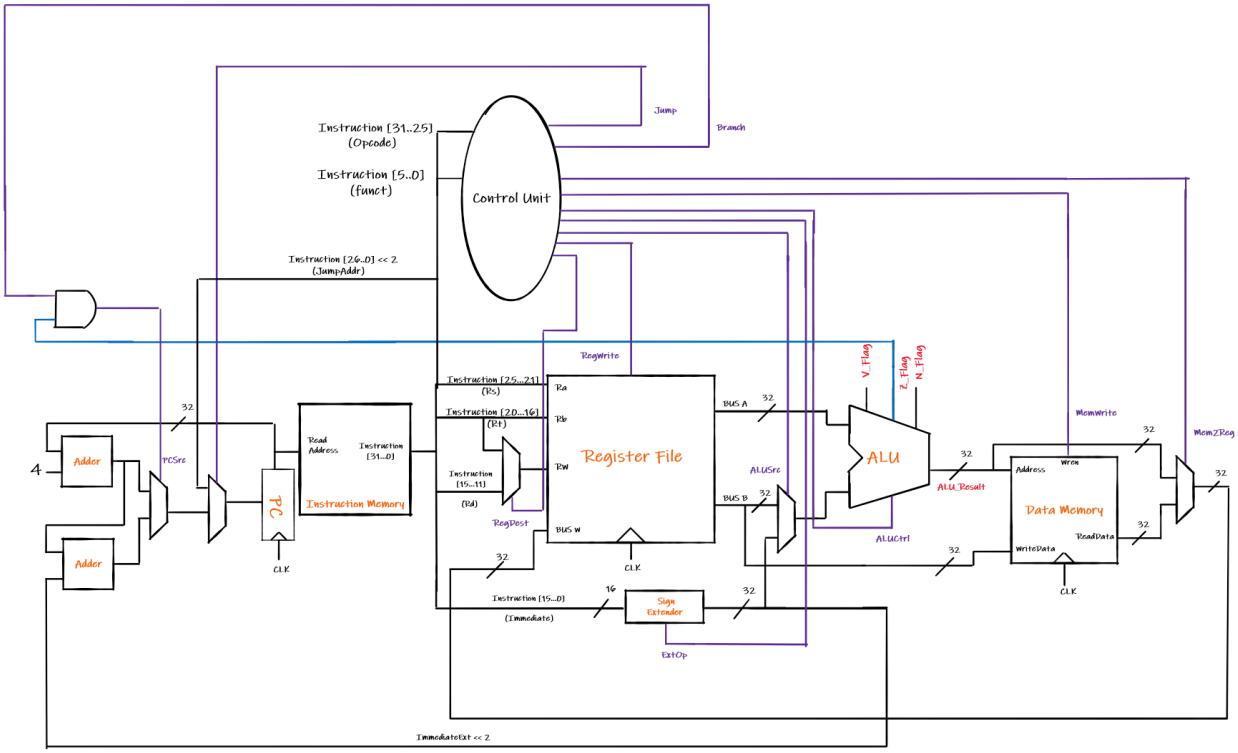

Figure 6: VHDL implementation of Control Unit

CPU Components (VHDL)

Next, we discuss the remaining necessary components to implement the Single Cycle MIPS Processor CPU. We begin with the VHDL implementation of the control unit as previously discussed.

Control Unit

```
entity Ramos_Control_Unit is
   port(
        Opcode   : in std_logic_vector(5 downto 0); -- 6 Bit Opcode from 32 Bit instruction
        Funct    : in std_logic_vector(5 downto 0); -- 6 Bit function field
        -- Control Signals
        ExtOp  : out std_logic;
        ALUCtrl: out std_logic_vector(3 downto 0);
        RegWr  : out std_logic;
        RegDst : out std_logic; -- 0 => Rt ; 1 => Rd
        ALUSrc : out std_logic; -- 0 => BusB ; 1 => ImmExt
        MemToReg : out std_logic;
        MemWr : out std_logic;
        Branch : out std_logic;
        Jump  : out std_logic
        );
end Ramos_Control_Unit;

architecture arch of Ramos_Control_Unit is
    signal ALUOp : std_logic_vector(2 downto 0); -- ALU Op (used to detemine ALUCtrl)
    begin

    process(Opcode)
    begin
        -- Select Opcode+Function combination for sw, lw, add, addu, addi, addui, sub, subu, and, andi, ori, nor, sll, srl, sra
        -- ALUOp: 0 => R-Type Instructions ; 1 => sw,lw ; 2 =>
        case Opcode is
            when "000000" => -- R-Type Instructions(add, addu, sub, subu, mult, div, and, nor, sll, srl, sra)
                ALUOp <= "000";
                ExtOp <= '0'; --X
                MemWr <= '0'; -- 0
                MemToReg <= '0'; -- 0
                ALUSrc <= '0'; -- 0
                RegWr <= '1'; -- 1
                RegDst <= '1'; -- 1
                Branch <= '0'; -- 0
                Jump <= '0';  -- 0
            when "100011" =>  -- lw
                ALUOp <= "001";
                ExtOp <= '1'; -- 1
                MemWr <= '0'; -- 0
                MemToReg <= '1'; -- 1
                ALUSrc <= '1'; -- 1
                RegWr <= '1'; -- 1
                RegDst <= '0'; -- 0
                Branch <= '0'; -- 0
                Jump <= '0'; -- 0
            when "101011" => -- sw
                ALUOp <= "001";
                ExtOp <= '1'; -- 1
                MemWr <= '1'; -- 1
                MemToReg <= '0'; -- X
                ALUSrc <= '1'; -- 1
                RegWr <= '0'; -- 0
                RegDst <= '0'; -- X
                Branch <= '0'; -- 0
                Jump <= '0'; -- 0
            when "001000" =>  -- addi
                ALUOp <= "001"; --
                ExtOp <= '1'; -- 1
                MemWr <= '0'; -- 0
                MemToReg <= '0'; -- 0
                ALUSrc <= '1'; -- 1
                RegWr <= '1'; -- 1
                RegDst <= '0'; -- 0
                Branch <= '0'; -- 0
                Jump <= '0'; -- 0
            when "001001" =>  --addiu
                ALUOp <= "001"; --
                ExtOp <= '1'; -- 1
                MemWr <= '0'; -- 0
                MemToReg <= '0'; -- 0
                ALUSrc <= '1'; -- 1
                RegWr <= '1'; -- 1
                RegDst <= '0'; -- 0
                Branch <= '0'; -- 0
                Jump <= '0'; -- 0
            when "001100" => -- andi
                ALUOp <= "010"; --
                ExtOp <= '0'; -- 0
                MemWr <= '0'; -- 0
                MemToReg <= '0'; -- 0
                ALUSrc <= '1'; -- 1
                RegWr <= '1'; -- 1
                RegDst <= '0'; -- 0
                Branch <= '0'; -- 0
                Jump <= '0'; -- 0
```

```
            when "001101" => -- ori
                ALUOp <= "100"; --
                ExtOp <= '0'; -- 0
                MemWr <= '0'; -- 0
                MemToReg <= '0'; -- 0
                ALUSrc <= '1'; -- 1
                RegWr <= '1'; -- 1
                RegDst <= '0'; -- 0
                Branch <= '0'; -- 0
                Jump <= '0'; -- 0
            when "000100" => -- beq
                ALUOp <= "101"; --
                ExtOp <= '1'; -- 1
                MemWr <= '0'; -- 0
                MemToReg <= '0'; -- X
                ALUSrc <= '0'; -- 0
                RegWr <= '0'; -- 0
                RegDst <= '0'; -- X
                Branch <= '1'; -- 1
                Jump <= '0'; -- 0
            when "000101" => -- bne
                ALUOp <= "101"; --
                ExtOp <= '1'; -- 1
                MemWr <= '0'; -- 0
                MemToReg <= '0'; -- X
                ALUSrc <= '0'; -- 0
                RegWr <= '0'; -- 0
                RegDst <= '0'; -- X
                Branch <= '1'; -- 1
                Jump <= '0'; -- 0
            when "000010" => -- j
                ALUOp <= "111"; --
                ExtOp <= '0'; -- X
                MemWr <= '0'; -- 0
                MemToReg <= '0'; -- X
                ALUSrc <= '0'; -- X
                RegWr <= '0'; -- 0
                RegDst <= '0'; -- X
                Branch <= '0'; -- 0
                Jump <= '1'; -- 1
            when others =>
                ALUOp <= "ZZZ";
                ExtOp <= 'Z';
                MemWr <= 'Z';
                MemToReg <= 'Z';
                ALUSrc <= 'Z';
                RegWr <= 'Z';
                RegDst <= 'Z';
                Branch <= 'Z';
                Jump <= 'Z';
        end case;
    end process;

    process(ALUOp, Funct)
    begin
      if ALUOp = "000" and Funct = "100000" then -- Add
            ALUCtrl <= "0000";
        elsif ALUOp = "000" and Funct = "100001" then -- Addu
            ALUCtrl <= "0000"; -- Add
        elsif ALUOp = "000" and Funct = "100010" then -- Sub
            ALUCtrl <= "0001";
        elsif ALUOp = "000" and Funct = "100011" then -- Subu
            ALUCtrl <= "0001"; -- Sub
        elsif ALUOp = "000" and Funct = "011000" then -- Mult
            ALUCtrl <= "0010";
        elsif ALUOp = "000" and Funct = "011010" then -- Div
            ALUCtrl <= "0011";
        elsif ALUOp = "000" and Funct = "100100" then -- And
            ALUCtrl <= "0100";
        elsif ALUOp = "000" and Funct = "100111" then -- Nor
            ALUCtrl <= "0110";
        elsif ALUOp = "000" and Funct = "000000" then -- Sll
            ALUCtrl <= "0111";
        elsif ALUOp = "000" and Funct = "000010" then -- Srl
            ALUCtrl <= "1000";
        elsif ALUOp = "000" and Funct = "000011" then -- Sra
            ALUCtrl <= "1001";
        elsif ALUOp = "001" then -- sw, lw, addi, addiu
            ALUCtrl <= "0000"; -- add
        elsif ALUOp <= "010" then --andi
            ALUCtrl <= "0100";
        elsif ALUOp = "100" then -- ori
            ALUCtrl <= "0101";
        elsif ALUOp = "101" then -- beq/bne
            ALUCtrl <= "0001"; -- Sub
        end if;
    end process;
```

Figure 7: VHDL implementation of CPU control signal

Program Counter (PC) Register

The first component is the Program Counter (PC) register which is part of a larger component known as a Next Address Logic (NAL) unit or a Fetch unit. The sole purpose of the PC register is to update the value of the program counter (i.e., the address of the next instruction) at the *rising edge* of the clock signal. The output will be fed into the *instruction memory* component to select the next appropriate instruction to be executed.

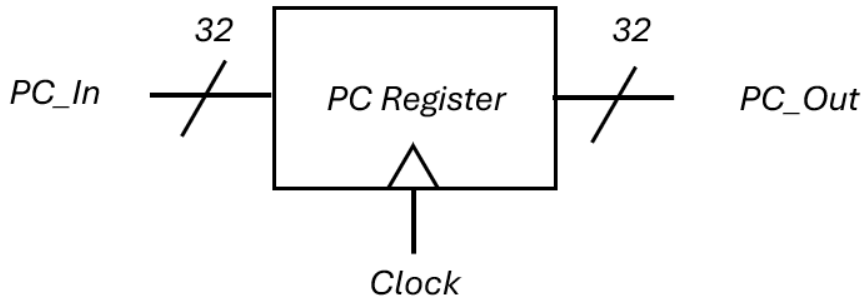

Figure 8a: Program Counter (PC) register symbol

```
entity Ramos_PC_Register is
   port (
          Clock: in std_logic;
          PC_In : in std_logic_vector(31 downto 0);
          PC_Out : out std_logic_vector(31 downto 0));
end Ramos_PC_Register;

architecture arch of Ramos_PC_Register is
begin
    process(Clock)
    begin
        if (rising_edge(Clock)) then
            PC_Out <= PC_In;
        end if;
    end process;
end arch;
```

Figure 8b: VHDL implementation of PC register

Instruction Memory

The instruction memory component will store the machine code instructions of some program to be executed. This component has a memory size of 128 bytes with instruction size of 32-Bits. The instruction to be executed will be determined by the 32-Bit address obtained from the PC register. The selected instruction will then be decoded into the necessary fields as shown in figure 9. These fields will serve as inputs to several other components in the CPU. Note that only one instruction is executed per clock cycle.

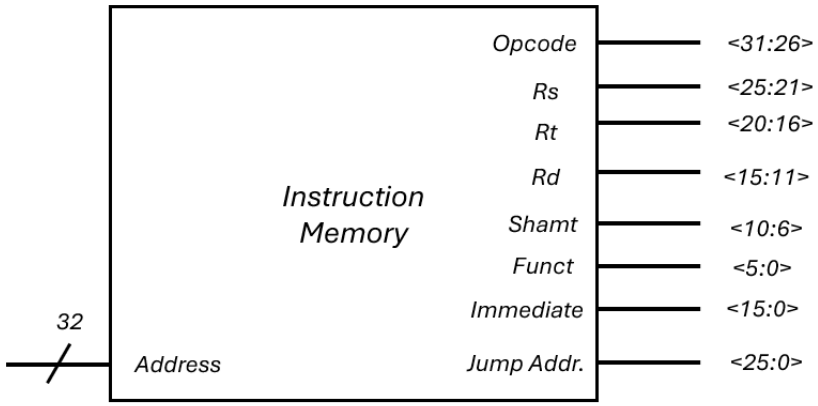

Figure 9a: Instruction Memory block symbol

```
entity Ramos_Instruction_Memory is
    port(
          InstrAddress : in std_logic_vector(31 downto 0); -- 32 Bit instruction address from (PC Register)
          Opcode       : out std_logic_vector(5 downto 0); -- 6 bit Opcode
          Rs, Rt, Rd   : out std_logic_vector(4 downto 0); -- 5 Bit addresses
          Shmat        : out std_logic_vector(4 downto 0); -- 5 Bit shift ammount (for sll & srl)
          Funct        : out std_logic_vector(5 downto 0); -- 6 Bit Function code (for R-Type Instructions)
          Immediate    : out std_logic_vector(15 downto 0); -- 16 Bit Immediate
          Jump_Address : out std_logic_vector(31 downto 0)); -- Effective Jump Address
end Ramos_Instruction_Memory;

architecture arch of Ramos_Instruction_Memory is
type mem is array(0 to 127) of std_logic_vector(31 downto 0);
   signal mem_array: mem;
   signal Instruction : std_logic_vector(31 downto 0);
   begin

   -- Machine code for MIPS program begins here
   -- Each address is incremented by 4
   mem_array(0) <= X"20100007"; -- addu $s0, $zero, 7
      mem_array(4) <= X"2011000A"; -- addu $s1, $zero, 10
   mem_array(8) <= X"02309021"; -- addu $s2, $s1, $s0
   mem_array(12) <= X"02119823"; -- subu $s3, $s0, $s1
   mem_array(16) <= X"12130002"; -- beq $s0, $s3, L2
   mem_array(20) <= X"02300018"; -- mult $s1, $s0
   mem_array(24) <= X"08100008"; -- j L1
   mem_array(28) <= X"0230001A"; -- div $s1, $s0
   mem_array(32) <= X"02508021"; -- addu, $s0, $s2, $s0
   -- Machine code for MIPS program ends here


   Instruction <= mem_array(to_integer(unsigned(InstrAddress)));

   -- Decode Instruction into necessary fields
   Opcode <= Instruction(31 downto 26);
   Rs <= Instruction(25 downto 21);
   Rt <= Instruction(20 downto 16);
   Rd <= Instruction(15 downto 11);
   Shmat <= Instruction(10 downto 6);
   Funct <= Instruction(5 downto 0);
   Immediate <= Instruction(15 downto 0);
   Jump_Address <= InstrAddress(31 downto 28) & Instruction(25 downto 0) & "00";

end arch;
```

Figure 9b: VHDL implementation of Instruction Memory

Sign-Zero Extender

The Sign-Zero Extender component is solely responsible for extending a 16-Bit immediate to a 32-Bit immediate. There are two possible extensions that can occur which is dependent on the `ExtOp` control signal whose behavior is defined in table 2 above.

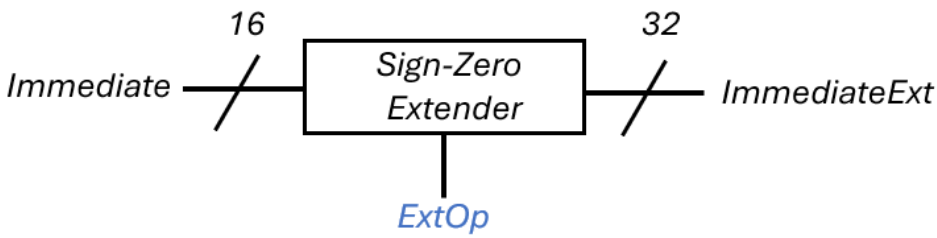

Figure 10a: Sign-Zero Extender block symbol

```
Entity Ramos_SignZero_Ext is
    port( A      : in std_logic_vector(15 downto 0);
          ExtOp  : in std_logic;
          Output: out std_logic_vector(31 downto 0));
end Ramos_SignZero_Ext;

architecture behvaioral of Ramos_SignZero_Ext is
begin
    process(ExtOp)
    begin
      if (ExtOp = '0') then
         Output <= std_logic_vector(resize(unsigned(A), 32));
      elsif (ExtOp = '1') then
         Output <= std_logic_vector(resize(signed(A), 32));
      end if;
   end process;
end behvaioral;
```

Figure 10b: VHDL implementation of Sign-Zero Extender

Adder

The adder component is used to compute the new program counter value (i.e., the address of the next instruction). We require an additional adder to compute the address of a label if a branch or jump is to be taken. Figures 11a and 11b show the block symbol and VHDL code for this component using LPM modules.

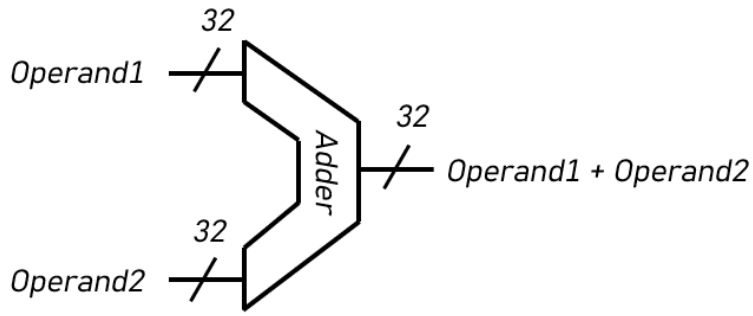

Figure 11a: Adder component

```
ENTITY Ramos_Adder IS
    PORT
    (
        dataa     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        datab     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        result       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END Ramos_Adder;

ARCHITECTURE SYN OF ramos_adder IS

    SIGNAL sub_wire0   : STD_LOGIC_VECTOR (31 DOWNTO 0);

    COMPONENT lpm_add_sub
    GENERIC (
        lpm_direction      : STRING;
        lpm_hint     : STRING;
        lpm_representation       : STRING;
        lpm_type     : STRING;
        lpm_width       : NATURAL
    );
    PORT (
            dataa : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            datab : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            result    : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
    END COMPONENT;

BEGIN
    result     <= sub_wire0(31 DOWNTO 0);

    LPM_ADD_SUB_component : LPM_ADD_SUB
    GENERIC MAP (
        lpm_direction => "ADD",
        lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO",
        lpm_representation => "UNSIGNED",
        lpm_type => "LPM_ADD_SUB",
        lpm_width => 32
    )
    PORT MAP (
        dataa => dataa,
        datab => datab,
        result => sub_wire0
    );
```

Figure 11b: VHDL implementation of LPM Adder

Register File (3 Ported)

The 3 ported Register File is a storage element that consists of three 5-bit registers (two read and one write). Read Registers Ra and Rb are source registers while write register Rw is a destination register. The outputs are Buses A and B, which serve as inputs (i.e., operands) to the Arithmetic Logic Unit (ALU). Bus W serves as an input to the register file itself which contains 32-Bit data to be written to some register. The behavioral of the control signal RegWr is defined in table 2.

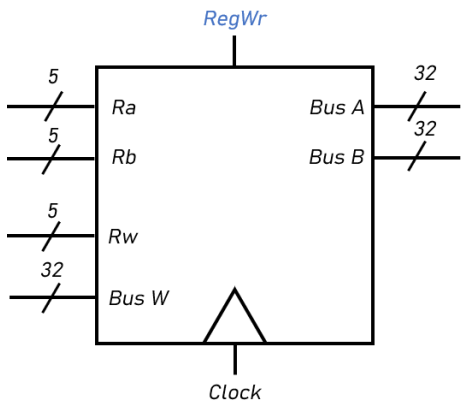

Figure 12a: 3-Ported Register File block symbol

```
ENTITY Ramos_RegisterFile is
   port(
      Clock: in std_logic;
      RegWr : in std_logic;
      BusW : in std_logic_vector(31 downto 0);
      Rw  : in std_logic_vector(4 downto 0);
      Ra : in std_logic_vector(4 downto 0);
      Rb : in std_logic_vector(4 downto 0);
      BusA : out std_logic_vector(31 downto 0);
      BusB : out std_logic_vector(31 downto 0));
end Ramos_RegisterFile;

architecture arch of Ramos_RegisterFile is
   type reg is array(0 to 31) of std_logic_vector(31 downto 0);
   signal reg_array: reg;
      begin

   process(Clock, RegWr)
   begin
   if (rising_edge(Clock)) then
      if (RegWr = '1') then
         reg_array(to_integer(unsigned(Rw))) <= BusW;
      end if;
   end if;
   end process;

   BusA <= X"00000000" when Ra = "00000" else reg_array(to_integer(unsigned(Ra)));
   BusB <= X"00000000" when Rb = "00000" else reg_array(to_integer(unsigned(Rb)));

end arch;
```

Figure 12b: VHDL implementation 3-Ported Register File

Arithmetic Logic Unit (ALU)

The ALU will perform the appropriate arithmetic operation depending on the value of the `ALUCtrl` signal. Namely, it is able to perform the operations defined in table 3. The inputs to the ALU are 32-Bit operands. The outputs of the ALU are two 32-bit upper and lower result registers. This was done to compensate for how the results of multiplication and division operations are stored in two 32-Bit *Hi* and *Lo* registers. In addition, overflow, negative, zero, and carry out flags are outputs and relevant only to addition/subtraction operations.

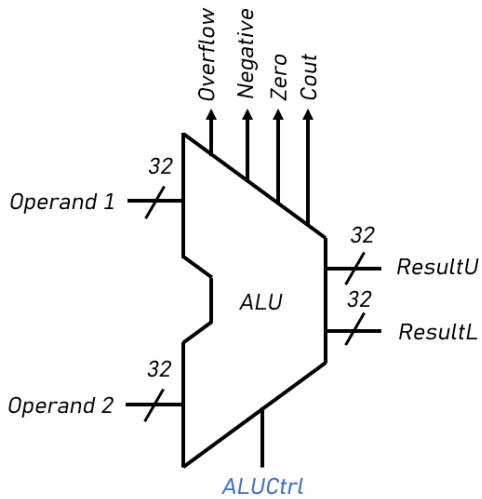

Figure 13a: ALU block symbol

```
-- This file defines the behavior of an ALU
Library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.RAMOS_ALU_COMPONENTS_PACKAGE.ALL;

entity Ramos_ALU is
   port(
        ALUCtrl    : in std_logic_vector(3 downto 0); -- Input comes from ALUControl
        Operand1   : in std_logic_vector(31 downto 0); -- BusA
        Operand2   : in std_logic_vector(31 downto 0); -- BusB  or ImmExt
        ALUResultU : out std_logic_vector(31 downto 0); -- Upper 32-Bit Result (For Mult and Div only!)
        ALUResultL : out std_logic_vector(31 downto 0); -- Lower 32-Bit Result
        V_FLAG, Z_FLAG, N_FLAG, Cout : out std_logic); -- ALU Output flags
end Ramos_ALU;

architecture arch of Ramos_ALU is
   signal Overflow, OpSel : std_logic;
   signal Result2 : std_logic_vector(63 downto 0); -- Product
   signal Result1, Result4, Result5, Result6, Result7, Result8, Result9 : std_logic_vector(31 downto 0);
   signal Result2U, Result2L, Result3U, Result3L : std_logic_vector(31 downto 0);

   begin
    process(ALUCtrl) -- For NBit_Adder_Sub
    begin
        if (ALUCtrl = "0000") then
            OpSel <= '0'; -- Add
        elsif (ALUCtrl = "0001") then
            OpSel <= '1'; -- Sub
        end if;
    end process;

    U1: Ramos_NBit_Add_Sub port map (Operand1, Operand2, '0', OpSel, Result1, Cout, V_FLAG, Z_FLAG, N_FLAG); -- add/sub with no carry-in

    U2: Ramos_Multiply_LPM port map(Operand1, Operand2, Result2); -- Multiply
    -- Split 64-Bit product into two 32-bit upper and lower results.
    Result2U <= Result2(63 downto 32);
    Result2L <= Result2(31 downto 0);
    U3: Ramos_Division_LPM port map(Operand1, Operand2, Result3L, Result3U); -- Divide
    U4: Ramos_Bitwise_AND port map (Operand1, Operand2, Result4); -- Bitwise AND/ANDI
    U5: Ramos_Bitwise_OR port map (Operand1, Operand2, Result5); -- Bitwise ORI
    U6: Ramos_Bitwise_NOR port map (Operand1, Operand2, Result6); -- Bitwise NOR
    U7: Ramos_Bitwise_LeftShift port map(Operand2, Result7);-- Sll
    U8: Ramos_Bitwise_RightShift port map(Operand2, Result8);-- Srl
    U9: Ramos_Bitwise_RightShiftArith port map(Operand2, Result9);-- Sra
```

```
process(ALUCtrl, Result1, Result2U, Result2L, Result3U, Result3L, Result4, Result5, Result6, Result7, Result8, Result9)
    begin
        case ALUCtrl is
          when "0000" =>
            ALUResultU <= X"00000000";
            ALUResultL <= Result1; -- Add/Addu/Addi/Addiu/lw/sw
          when "0001" =>
            ALUResultU <= X"00000000";
            ALUResultL <= Result1; -- Sub/Subu/beq/bne
          when "0010" => -- Mult
            ALUResultU <= Result2U;
            ALUResultL <= Result2L;
          when "0011" => -- Div
            ALUResultU <= Result3U; -- Remainder
            ALUResultL <= Result3L; -- Quotient
          when "0100" =>
            ALUResultU <= X"00000000";
            ALUResultL <= Result4; -- and/andi
          when "0101" =>
            ALUResultU <= X"00000000";
            ALUResultL <= Result5; -- ori
          when "0110" =>
            ALUResultU <= X"00000000";
            ALUResultL <= Result6; -- nor
          when "0111" =>
            ALUResultU <= X"00000000";
            ALUResultL <= Result7; -- sll
          when "1000" =>
            ALUResultU <= X"00000000";
            ALUResultL <= Result8; -- srl
          when "1001" =>
            ALUResultU <= X"00000000";
            ALUResultL <= Result9; -- sra
          when others =>
            ALUResultU <= (others => 'Z');
            ALUResultL <= (others => 'Z');
        end case;
    end process;

end arch;
```

Figure 13b: VHDL implementation of ALU

```
package Ramos_ALU_Components_Package is

    component Ramos_NBit_Add_Sub
    Port ( A, B                    : in std_logic_vector(31 downto 0);
           Cin                     : in std_logic;
           Opcode                  : in std_logic; -- Add when 0 / subtract when 1
           S                       : out std_logic_vector(31 downto 0);
           Cout                    : out std_logic;
           V_FLAG, Z_FLAG, N_FLAG  : out std_logic);
    end component;

    component Ramos_Division_LPM
    Port (
      denom     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      numer     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      quotient    : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
      remain      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
    end component;

    component Ramos_Multiply_LPM
    Port
    (
      dataa     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      datab     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      result      : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
    );
    end component;

    component Ramos_Bitwise_AND
    port ( A, B   : in std_logic_vector(31 downto 0);
           result : out std_logic_vector(31 downto 0));
    end component;

    component Ramos_Bitwise_NOR
    port ( A, B   : in std_logic_vector(31 downto 0);
           result : out std_logic_vector(31 downto 0));
    end component;

    component Ramos_Bitwise_OR
    port ( A, B   : in std_logic_vector(31 downto 0);
           result : out std_logic_vector(31 downto 0));
    end component;


    component Ramos_Bitwise_RightShift
    port ( A  : in std_logic_vector(31 downto 0);
           result : out std_logic_vector(31 downto 0));
    end component;

    component Ramos_Bitwise_RightShiftArith
    port ( A  : in std_logic_vector(31 downto 0);
           result : out std_logic_vector(31 downto 0));
    end component;

   component Ramos_Adder -- Adder to compute instruction addresses
   PORT(
      dataa     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      datab     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      result    : OUT STD_LOGIC_VECTOR (31 DOWNTO 0));
   end component;

end Ramos_ALU_Components_Package;
```

Figure 13c: ALU Components Package

Data Memory

The Data Memory component allows for the reading and writing data to/from memory. This component has a 64-Bit memory size and data word size of 32-Bits. The 32-bit *Address* input is the memory location (i.e., address) to be selected. The *Data In* input, stores the data to be written if the control signal `MemWr` is asserted and the clock is at the rising edge. If `MemRd` is asserted, then we read from memory (i.e., the value stored in *Data Out*).

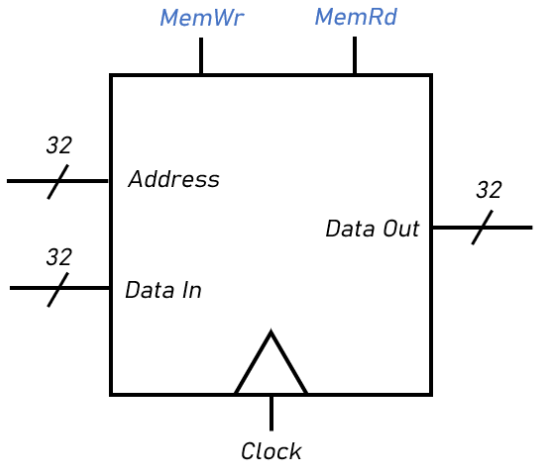

Figure 14a: Data Memory block symbol

```
ENTITY Ramos_DataMemory is
   port(
      clock: in std_logic;
      MemWr : in std_logic; -- Control signal
      MemRd : in std_logic; -- Control signal
      dataWr : in std_logic_vector(31 downto 0);
      address: in std_logic_vector(31 downto 0);
      dataRd : out std_logic_vector(31 downto 0));
   end Ramos_DataMemory;

architecture arch of Ramos_DataMemory is
   signal MemAddr : std_logic_vector(31 downto 0);
   type mem is array(0 to 63) of std_logic_vector(31 downto 0);
   signal mem_array: mem :=((others => (others=> '0')));

   begin
   MemAddr <= address(31 downto 0);
   process(clock)
   begin
      if(rising_edge(Clock)) then
         if (MemWr = '1') then -- Write to memory
            mem_array(to_integer(unsigned(MemAddr))) <= dataWr;

         end if;
      end if;
   end process;
   dataRd <= mem_array(to_integer(unsigned(MemAddr))) when (MemRd ='1') else x"00000000"; -- Read from memory

end arch;
```

Figure 14b: VHDL implementation of Data Memory

2-to-1 (32-Bit) Multiplexers

Lastly, multiple 2 to 1 multiplexers will be utilized throughout the CPU. They will help control the flow of the data path. Namely, we require 4 32-Bit and 1 5-Bit multiplexers.

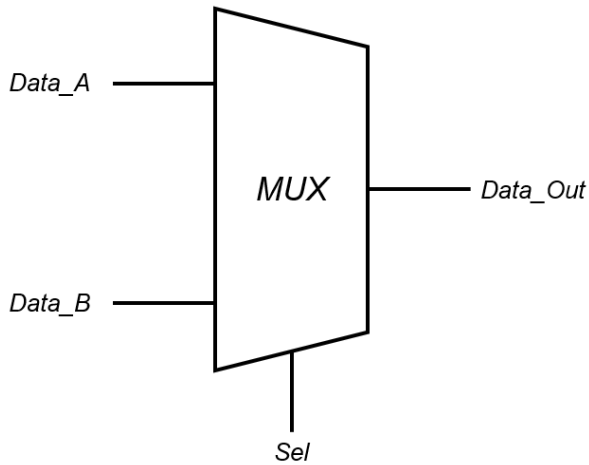

Figure 15a: 2 to 1 Multiplexer block symbol

```
entity Ramos_2to1_Mux is
   port ( D0, D1 : in STD_LOGIC_VECTOR(31 downto 0);
          SEL    : in STD_LOGIC;
          OUT1   : out STD_LOGIC_VECTOR(31 downto 0));
end Ramos_2to1_Mux;

architecture behavioral of Ramos_2to1_Mux is
begin
   OUT1(0) <= (D0(0) AND (NOT SEL)) OR (D1(0) AND SEL);
   OUT1(1) <= (D0(1) AND (NOT SEL)) OR (D1(1) AND SEL);
   OUT1(2) <= (D0(2) AND (NOT SEL)) OR (D1(2) AND SEL);
   OUT1(3) <= (D0(3) AND (NOT SEL)) OR (D1(3) AND SEL);
   OUT1(4) <= (D0(4) AND (NOT SEL)) OR (D1(4) AND SEL);
   OUT1(5) <= (D0(5) AND (NOT SEL)) OR (D1(5) AND SEL);
   OUT1(6) <= (D0(6) AND (NOT SEL)) OR (D1(6) AND SEL);
   OUT1(7) <= (D0(7) AND (NOT SEL)) OR (D1(7) AND SEL);
   OUT1(8) <= (D0(8) AND (NOT SEL)) OR (D1(8) AND SEL);
   OUT1(9) <= (D0(9) AND (NOT SEL)) OR (D1(9) AND SEL);
   OUT1(10) <= (D0(10) AND (NOT SEL)) OR (D1(10) AND SEL);
   OUT1(11) <= (D0(11) AND (NOT SEL)) OR (D1(11) AND SEL);
   OUT1(12) <= (D0(12) AND (NOT SEL)) OR (D1(12) AND SEL);
   OUT1(13) <= (D0(13) AND (NOT SEL)) OR (D1(13) AND SEL);
   OUT1(14) <= (D0(14) AND (NOT SEL)) OR (D1(14) AND SEL);
   OUT1(15) <= (D0(15) AND (NOT SEL)) OR (D1(15) AND SEL);
   OUT1(16) <= (D0(16) AND (NOT SEL)) OR (D1(16) AND SEL);
   OUT1(17) <= (D0(17) AND (NOT SEL)) OR (D1(17) AND SEL);
   OUT1(18) <= (D0(18) AND (NOT SEL)) OR (D1(18) AND SEL);
   OUT1(19) <= (D0(19) AND (NOT SEL)) OR (D1(19) AND SEL);
   OUT1(20) <= (D0(20) AND (NOT SEL)) OR (D1(20) AND SEL);
   OUT1(21) <= (D0(21) AND (NOT SEL)) OR (D1(21) AND SEL);
   OUT1(22) <= (D0(22) AND (NOT SEL)) OR (D1(22) AND SEL);
   OUT1(23) <= (D0(23) AND (NOT SEL)) OR (D1(23) AND SEL);
   OUT1(24) <= (D0(24) AND (NOT SEL)) OR (D1(24) AND SEL);
   OUT1(25) <= (D0(25) AND (NOT SEL)) OR (D1(25) AND SEL);
   OUT1(26) <= (D0(26) AND (NOT SEL)) OR (D1(26) AND SEL);
   OUT1(27) <= (D0(27) AND (NOT SEL)) OR (D1(27) AND SEL);
   OUT1(28) <= (D0(28) AND (NOT SEL)) OR (D1(28) AND SEL);
   OUT1(29) <= (D0(29) AND (NOT SEL)) OR (D1(29) AND SEL);
   OUT1(30) <= (D0(30) AND (NOT SEL)) OR (D1(30) AND SEL);
   OUT1(31) <= (D0(31) AND (NOT SEL)) OR (D1(31) AND SEL);

end behavioral;
```

Figure 15b: VHDL implementation of 2 to 1 Multiplexer (32-Bit)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Ramos_2to1_Mux_5Bit is
   port ( D0, D1 : in STD_LOGIC_VECTOR(4 downto 0);
          SEL    : in STD_LOGIC;
          OUT1   : out STD_LOGIC_VECTOR(4 downto 0));
end Ramos_2to1_Mux_5Bit;

architecture behavioral of Ramos_2to1_Mux_5Bit is
begin
   OUT1(0) <= (D0(0) AND (NOT SEL)) OR (D1(0) AND SEL);
   OUT1(1) <= (D0(1) AND (NOT SEL)) OR (D1(1) AND SEL);
   OUT1(2) <= (D0(2) AND (NOT SEL)) OR (D1(2) AND SEL);
   OUT1(3) <= (D0(3) AND (NOT SEL)) OR (D1(3) AND SEL);
   OUT1(4) <= (D0(4) AND (NOT SEL)) OR (D1(4) AND SEL);
end behavioral;
```

Figure 15c: VHDL implementation of 2 to 1 Multiplexer (5-Bit)

Quartus Design and Testing

Quartus Designs

The Single Cycle (MIPS Processor) CPU was realized in two ways. One implementation involved creating schematic blocks for each individual component and connecting each component accordingly as designed in figure 6 above which can then generate the VHDL code automatically (not shown due to length).

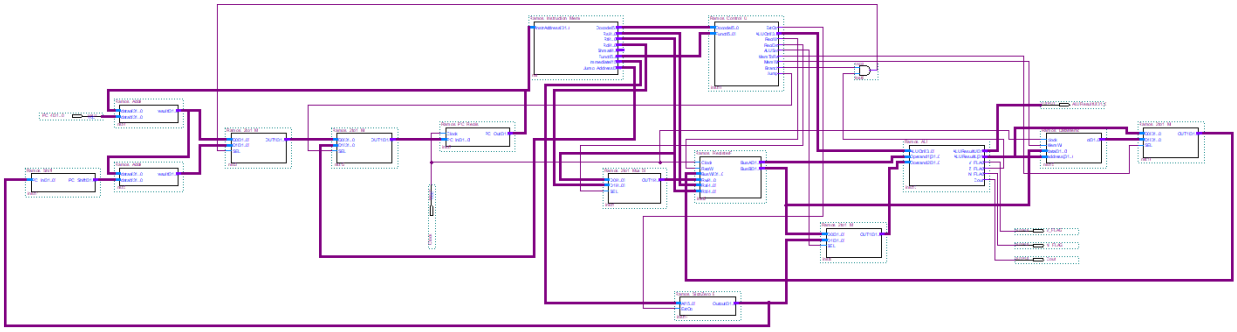

Figure 16a: Quartus Implementation of Single Cycle (MIPS Processor) CPU

The second implementation involved manually writing the VHDL code and connecting components utilizing *port maps* as shown in figure 16b. The logic is fundamentally the same as the design above.

```vhdl
entity Ramos_CPU is
   port(
         Clock : in std_logic;
         Result: out std_logic_vector(63 downto 0) -- 64-Bit ALU Result
      );

end Ramos_CPU;

architecture arch of Ramos_CPU is
   signal Opcode, Funct          : std_logic_vector(5 downto 0);
   signal Rs, Rt, Rd, Shmat      : std_logic_vector(4 downto 0);
   signal Immediate              : std_logic_vector(15 downto 0);
   signal JumpAddr               : std_logic_vector(31 downto 0);
   -- Control Signals
   signal ExtOp, RegWr, RegDst, ALUSrc, MemWr, MemToReg, Branch, Jump : std_logic;
   signal PCSrc : std_logic;
   signal ALUCtrl : std_logic_vector(3 downto 0);
   -- ALU Signals
   signal Operand1, Operand2, ALUResultU, ALUResultL, BusB: std_logic_vector(31 downto 0);
   signal V_FLAG, Z_FLAG, N_FLAG, Cout : std_logic;
   -- Multiplexer Outputs
   signal Rw   : std_logic_vector(4 downto 0);
   signal BusW : std_logic_vector(31 downto 0);
   signal NextAddr : std_logic_vector(31 downto 0);
   -- Data Memory Signals
   signal MemDataOut : std_logic_vector(31 downto 0);
   -- Instruction Fetch Signals
   signal ImmExt, ImmExtShift : std_logic_vector(31 downto 0);
   signal Addr1, Addr2 : std_logic_vector(31 downto 0);
   signal PC_Out : std_logic_vector(31 downto 0);
   signal CurrAddr,CurrAddrTemp : std_logic_vector(31 downto 0);

   begin
   -- Fetch and decode current instruction from Instruction Memory and set data paths via control unit
   PC: Ramos_PC_Register port map(Clock, CurrAddr, NextAddr);
   InstrMem: Ramos_Instruction_Memory port map(NextAddr, Opcode, Rs, Rt, Rd, Shmat, Funct, Immediate, JumpAddr); -- Decode instruction into fields
   CU: Ramos_Control_Unit port map(Opcode, Funct, ExtOp, ALUCtrl, RegWr, RegDst, ALUSrc, MemToReg, MemWr, Branch, Jump); -- Set all control signals
   -- Prepare instruction executing by selecting destination address and values of both operands
   M1: Ramos_2to1_Mux_5Bit port map(Rt, Rd, RegDst, Rw); -- If RegDst = 0, then Rw <= Rt, else Rw <= rd
   RefFile: Ramos_RegisterFile port map(Clock, RegWr, BusW, Rw, Rs, Rt, Operand1, BusB); -- Determine value of first operand1 and busB
   M2: Ramos_2to1_Mux port map(BusB, ImmExt, ALUSrc, Operand2); -- If ALUSrc = 0, then operand2 <= BusB, else operand2 <= ImmExt
   -- Execute Arithimetic Operation
   ALU: Ramos_ALU port map(ALUCtrl, Operand1, Operand2, ALUResultU, ALUResultL, V_FLAG, Z_FLAG, N_FLAG, Cout);
   PCSrc <= Branch AND Z_FLAG; -- Define PCSrc control signal
   -- Write to Data Memory
   DataMem: Ramos_DataMemory port map(Clock, MemWr, BusB, ALUResultU, MemDataOut);
   M3: Ramos_2to1_Mux port map(ALUResultL, MemDataOut ,MemToReg, BusW); -- If MemToReg = 0, then BusW <= ALUResultL, else BusW <= MemDataOut
```

```vhdl
   -- Instruction Fetch Unit (Get address of next instrction)
   SZExt: Ramos_SignZero_Ext port map(Immediate, ExtOp, ImmExt); -- Compute the sign extended Imm. field
   ImmExtShift <= to_stdlogicvector(to_bitvector(ImmExt) sll 2); -- Multiply by 4 via shift ImmExt left by 2 (ImmExtShift <= ImmExt << 2)
   PC_Adder: Ramos_Adder port map(NextAddr, X"00000004", Addr1); -- Compute Addr1 <= PC + 4
   Branch_Adder: Ramos_Adder port map(Addr1, ImmExtShift, Addr2); -- Compute Branch address Addr2 <= PC+4+ImmExtShift
   M4: Ramos_2to1_Mux port map(Addr1, Addr2, PCSrc, CurrAddrTemp); -- If PCSrc <= 0, then NextAddr = Addr1, else NextAddr <= Addr2
   M5: Ramos_2to1_Mux port map(CurrAddrTemp, JumpAddr, Jump, PC_Out); -- If Jump = 0, then NextAddr = PC_Out, else NextAddr = JumpAddr
   Result(63 downto 32) <= ALUResultU;
   Result(31 downto 0) <= ALUResultL;
end arch;
```

Figure 16d: Single Cycle (MIPS Processor) CPU VHDL Implementation

Pre-Simulation

Prior to simulating the Single Cycle (MIPS) Processor, some machine instructions are required. To obtain these machine instructions, we require a MIPS assembly program. Hence, a program, or multiple programs, can be written in MARS simulator such that each instruction can be implemented. We begin with a simple program to compute some simple arithmetic, execute an if-else statement (via beq/bne), and perform a jump. Figures 17a and 17b show the assembly code and machine instructions of a simple MIPS test.

```
# MIPS Test Program #1
.text
addi $s0, $zero, 7
addi $s1, $zero, 10
addu $s2, $s1, $s0 # $s2 = $s1 + $s0
subu $s3, $s0, $s1 # $s3 = $s0 - $s1
beq $s0, $s3, Else # If equal, go to label Else, If $s0 = $s3, execute lines 8 and 9.
mult $s1, $s0 # $s1 * $s0
j L1 # Jump to label L1 unconditionally
Else:
div $s1, $s0 # $s1 / $s0

L1:
addu $s0, $s2, $s0
```

Figure 17a: MIPS assembly test program

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x20100007	addi $16,$0,0x00000007	3: addi $s0, $zero, 7
	0x00400004	0x2011000a	addi $17,$0,0x0000000a	4: addi $s1, $zero, 10
	0x00400008	0x02309021	addu $18,$17,$16	5: addu $s2, $s1, $s0 # $s2 = $s1 + $s0
	0x0040000c	0x02119823	subu $19,$16,$17	6: subu $s3, $s0, $s1 # $s3 = $s0 - $s1
	0x00400010	0x12130002	beq $16,$19,0x00000002	7: beq $s0, $s3, Else # If equal, go to label Else, If $s0 = $s3, execute lines 8 and 9.
	0x00400014	0x02300018	mult $17,$16	8: mult $s1, $s0 # $s1 * $s0
	0x00400018	0x08100008	j 0x00400020	9: j L1 # Jump to label L1 unconditionally
	0x0040001c	0x0230001a	div $17,$16	11: div $s1, $s0 # $s1 / $s0
	0x00400020	0x02508021	addu $16,$18,$16	14: addu $s0, $s2, $s0

Figure 17b: Machine Instructions MIPS assembly test program

The machine instructions are then preloaded into the *Instruction Memory* component as shown in figure 18. Utilizing a memory array of size 128-Bytes, each machine instruction is defined at a 32-Bit address starting at 0 and spaced out by 4-Bytes.

```
-- Machine code for MIPS program begins here
-- Each address is incremented by 4
mem_array(0) <= X"20100007"; -- addu $s0, $zero, 7
   mem_array(4) <= X"2011000A"; -- addu $s1, $zero, 10
mem_array(8) <= X"02309021"; -- addu $s2, $s1, $s0
mem_array(12) <= X"02119823"; -- subu $s3, $s0, $s1
mem_array(16) <= X"12130002"; -- beq $s0, $s3, L2
mem_array(20) <= X"02300018"; -- mult $s1, $s0
mem_array(24) <= X"08100008"; -- j L1
mem_array(28) <= X"0230001A"; -- div $s1, $s0
mem_array(32) <= X"02508021"; -- addu, $s0, $s2, $s0
-- Machine code for MIPS program ends here
```

Figure 18: Loading MIPS Machine Instructions onto Instruction Memory

ModelSim Simulation

At the start of the simulation show in figure 19a, the address of the first instruction (i.e., 0x00000000), is loaded in signal `CurrAddr`. The first expected instruction to be fetched has machine code 0x20100007. At each instruction fetch, the selected instruction is then decoded to define the necessary field values (in blue). Next, the control signals (in yellow) are set accordingly depending on the opcode value and `funct` value (for R-Type instructions only) to set the data path. The values of the two operands are defined and the operation defined by the `ALUCtrl` signal is performed. The result is stored onto two 32-Bit high and low registers and outputted as a 64-Bit resultant (in red). Once an instruction is fully executed, we must fetch the next instruction (in orange) for execution. There are always three possible values for the address of the next instruction: The current address incremented by 4, the branch address, and the jump address. This entire process occurs in just one clock cycle per instruction.

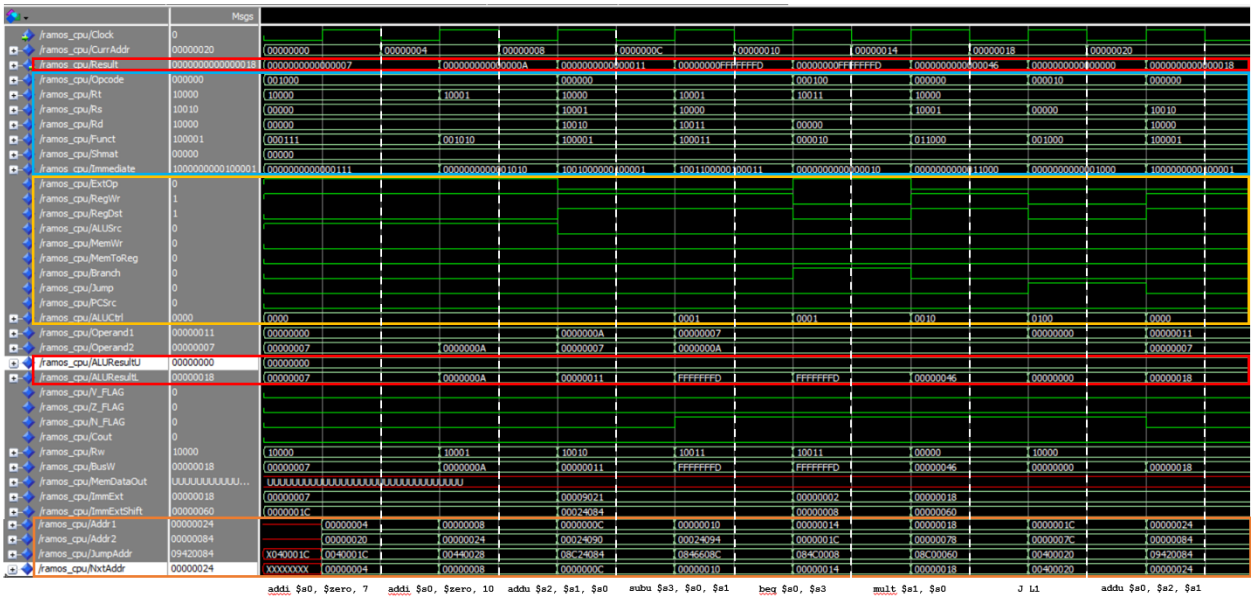

Figure 19a: ModelSim Simulation Waveforms of MIPS program of figure 17a

The simulation results can be verified by executing the program shown in figure 17a in MARS simulator. Figure 19b shows the values of the registers throughout the program. From this, the correctness of the simulation above can be verified.

$s0	16	0x00000007
$s1	17	0x0000000a
$s2	18	0x00000011
$s3	19	0xfffffffd
hi		0x00000000
lo		0x00000046
$s0	16	0x00000018

Figure 19b: Register values throughout the program

Design Pitfalls

It should be stated that some instructions could not be correctly verified. Namely, difficulties arose when testing the `lw` and `sw` instructions. Furthermore, while the bitwise operations where not discussed, they were tested and verified in previous labs. They are shown in figures 20a to 20.

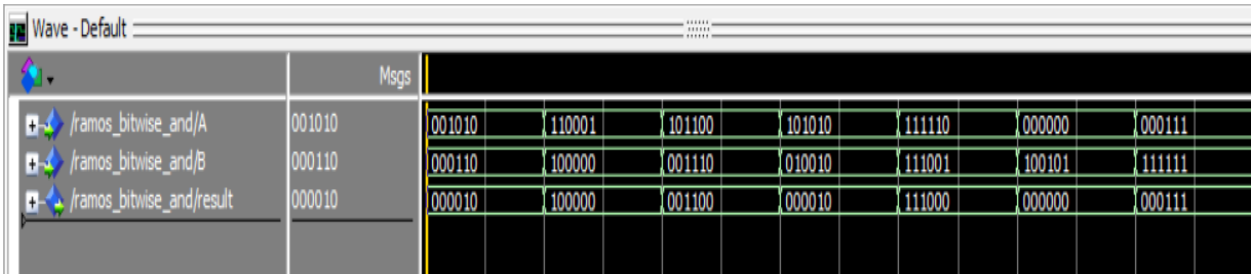

Figure 20a: Simulation of bitwise AND

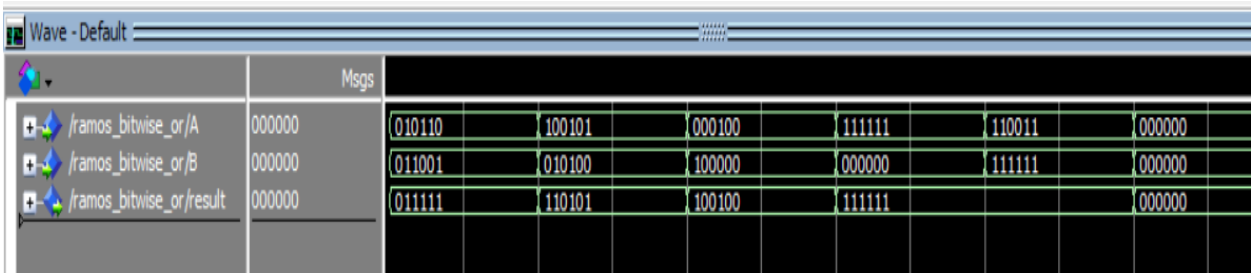

Figure 20b: Simulation of bitwise OR

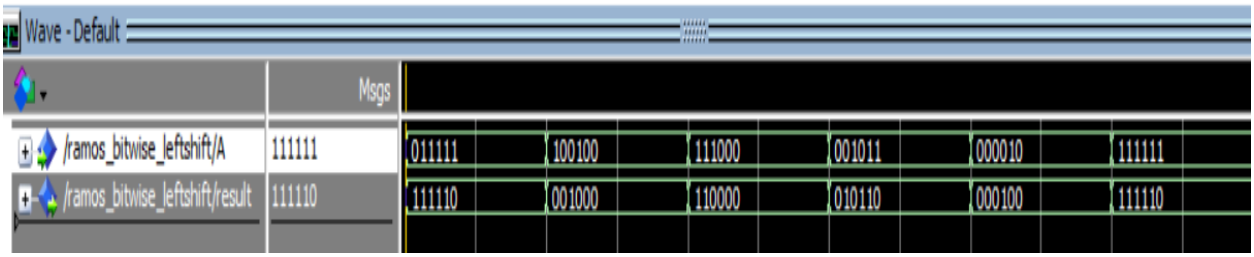

Figure 20c: Simulation of bitwise sll operation

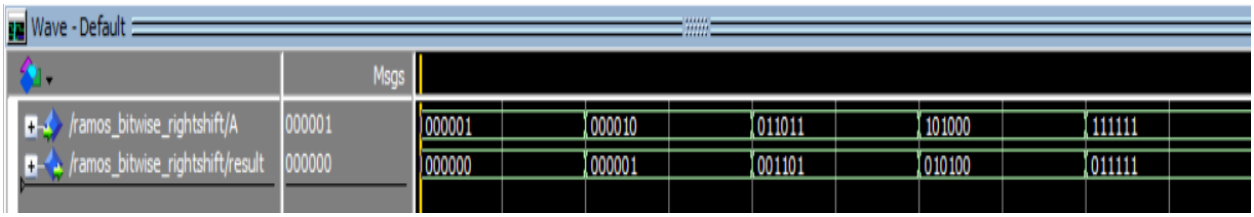

Figure 20d: Simulation of bitwise srl operation

Conclusion

Despite its simplicity and ease of design, the Single Cycle CPU is not desired in today's world of processors. This is because, its biggest demerit is its overall inefficiency. Namely since the clock cycle (i.e, the execution time) is solely determined by the time it takes to execute the longest instruction. In terms of this design, the longest instruction is the load word instruction as its data path passes through five components in the CPU: Instruction Memory → Register File → ALU → Data Memory → Register File. While we cannot reduce the execution time of an instruction, but we can increase the overall *throughput* by implementing a so called "Pipelining" technique, which enable the CPU to execute multiple instructions within a single clock cycle.