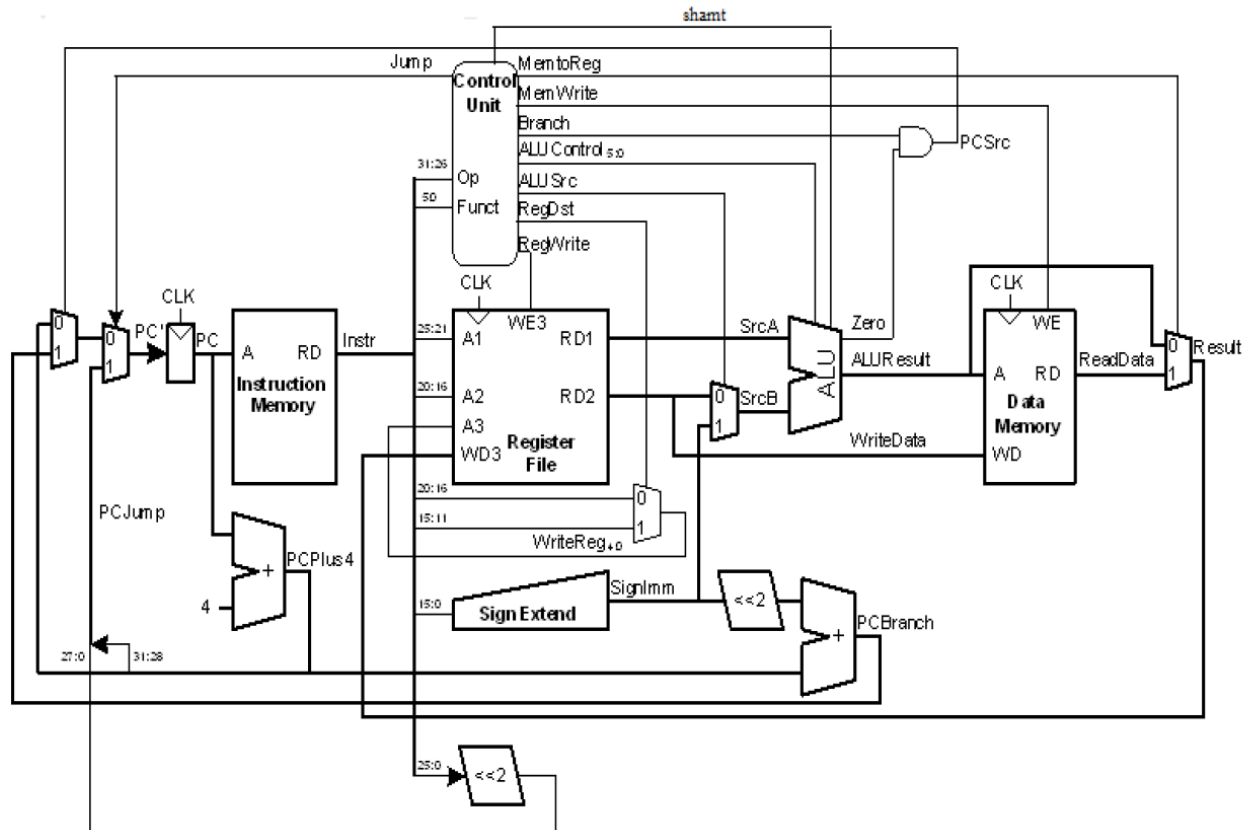


MIPS

Single-Cycle Processor

By: Khaled Ahmed Hamed



1. Introduction

This report documents the design, implementation, and verification of a MIPS single-cycle processor. The design follows the MIPS architecture, which is known for its simplicity and effectiveness. The processor is designed following the RISC-V architecture and adheres to the standard Instruction Set Architecture (ISA) principles as described in the book *"Digital Design and Computer Architecture"* by David Harris and Sarah Harris.

The processor executes each instruction—fetching, decoding, and execution—in a single clock cycle. This design supports so many instructions, classified into five instruction formats.

For more details, visit my repository on GitHub: [GitHub Repo](#)

2. Design Overview

2.1. Processor Components

The processor is composed of several key components, each playing a vital role in executing instructions:

- **ALU (Arithmetic Logic Unit):** Performs arithmetic and logic operations.
- **Register File:** Stores and retrieves values from registers.
- **Data Memory:** Used to read and write data during load/store operations.
- **Instruction Memory:** Holds the program instructions.
- **Control Unit:** Generates control signals based on the opcode and function code of the instruction.
- **Data Path:** Connects all components and handles data flow through the processor.

2.2. Key Modules

- **ALU:** Handles operations like addition, subtraction, AND, OR, and Set Less Than (SLT).
- **Control Unit:** Determines the operation of the processor by generating control signals based on the opcode and function code of the instruction.
- **Data Path:** Manages the flow of data between components based on the control signals.
- **Memory Modules (Instruction and Data):** Provide storage and access to instructions and data.

1.Control Unit (Control Signals):

1.Main Decoder:

Main Decoder Truth Table:

| Instruction | Opcode | RegWrite | RegDst | ALUSrc | Branch | MemWrite | MemtoReg | ALUOp | Jump |
|-------------|--------|----------|--------|--------|--------|----------|----------|-------|------|
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 | 0 |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 | 0 |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 | 0 |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 | 0 |
| addi | 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 00 | 0 |
| j | 000010 | 0 | X | X | X | 0 | X | XX | 1 |

RTL Code Snippet:

```
1  module main_decoder (Opcode,MemtoReg,MemWrite,Branch,ALUSrc,RegDst,RegWrite,ALUOp,Jump);
2  //IO Ports
3  input [5:0] Opcode;
4  output MemtoReg,MemWrite,Branch,ALUSrc,RegDst,RegWrite,Jump;
5  output [1:0] ALUOp;
6  //Internal Signal
7  reg [8:0] control_signals ;
8  //Functionality
9  always @(*) begin
10     case(Opcode)
11         6'b000000:control_signals=9'b110000100;//R-Type
12         6'b100011:control_signals=9'b101001000;//Load Word (LW)
13         6'b101011:control_signals=9'b0x101x000;//Store Word (SW)
14         6'b000100:control_signals=9'b0x010x010;//Branch Equal (BEQ)
15         6'b001000:control_signals=9'b101000000;//Add Immediate (addi)
16         6'b000010:control_signals=9'b0xxx0xxx1;//J-Type (Jump)
17         default:control_signals =9'bxxxxxxxxx; //Unprovided Case (9'bxxxxxxxxx = 9'bx)
18     endcase
19 end
20 assign {RegWrite,RegDst,ALUSrc,Branch,MemWrite,MemtoReg,ALUOp,Jump} = control_signals;
21 endmodule
```

2.ALU Decoder:

ALU Decoder Truth Table:

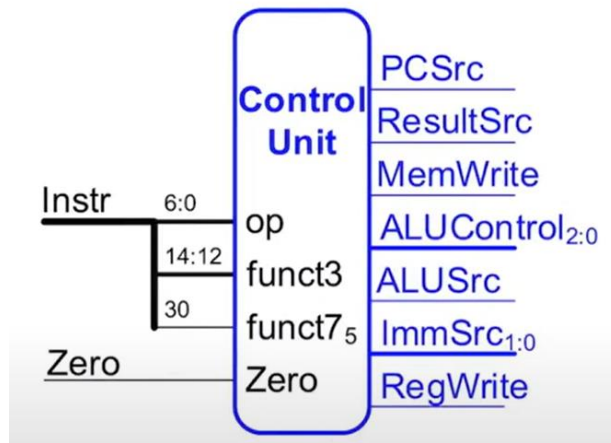
| ALUOp | Func | ALUControl |
|-------|--------------|---------------------|
| 00 | X | 010 (add) |
| X1 | X | 110 (subtract) |
| 1X | 100000 (add) | 010 (add) |
| 1X | 100010 (sub) | 110 (subtract) |
| 1X | 100100 (and) | 000 (and) |
| 1X | 100101 (or) | 001 (or) |
| 1X | 101010 (slt) | 111 (set less than) |

RTL Code Snippet:

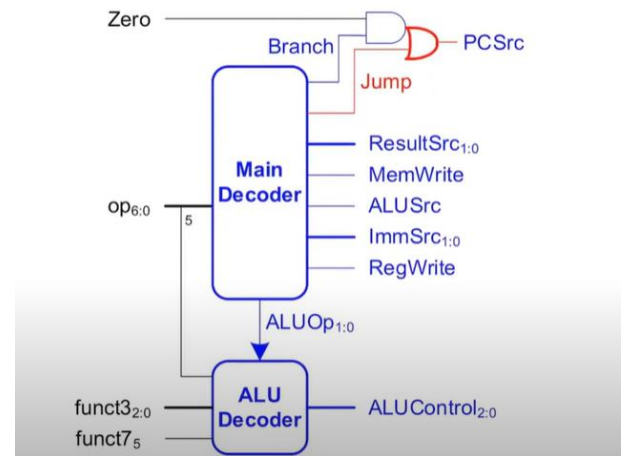
```
1  module alu_decoder (ALUOp,Func,ALUControl);
2  //IO Ports
3  input [5:0] Func;
4  input [1:0] ALUOp;
5  output reg [2:0] ALUControl;
6  //Functionality
7  always @(*) begin
8      casex(ALUOp)
9          2'b00:ALUControl=3'b010;//Add
10         2'b01:ALUControl=3'b110;//Subtract
11         2'b1x:begin//R-Type , (2'b1x ==> 2'b10 , 2'b11)
12             case(Func)
13                 6'b100000:ALUControl=3'b010;//Add
14                 6'b100010:ALUControl=3'b110;//Subtract
15                 6'b100100:ALUControl=3'b000;//AND
16                 6'b100101:ALUControl=3'b001;//OR
17                 6'b101010:ALUControl=3'b111;//Set Less Than (SLT)
18                 default:ALUControl=3'bxxx;//Unprovided Case (3'bxxx = 3'bx)
19             endcase
20         end
21         default:ALUControl=3'bxxx;//Unprovided Case (3'bxxx = 3'bx)
22     endcase
23 end
24 endmodule
```

3.Control Unit Top Module:

High Level View



Low Level View



RTL Code Snippet:

```

1 module control_unit (Opcode, Funct, Zero, MemtoReg, MemWrite, ALUSrc, RegDst, RegWrite, Jump, PCSrc, ALUControl);
2 //IO Ports
3 input [5:0] Opcode, Funct;
4 output MemtoReg, MemWrite, ALUSrc, RegDst, RegWrite, PCSrc, Jump;
5 output [2:0] ALUControl;
6 input Zero;
7 //Internal Signals
8 wire [1:0] ALUOp;
9 wire Branch;
10 //Functionality --> Instantiations
11 main_decoder MainDecoder (.Opcode(Opcode), .MemtoReg(MemtoReg), .MemWrite(MemWrite), .Branch(Branch),
12 .ALUSrc(ALUSrc), .RegDst(RegDst), .RegWrite(RegWrite), .ALUOp(ALUOp), .Jump(Jump));
13 alu_decoder ALUDecoder (.ALUOp(ALUOp), .Funct(Funct), .ALUControl(ALUControl));
14 //PC Source
15 assign PCSrc = Branch & Zero ;
16 endmodule

```

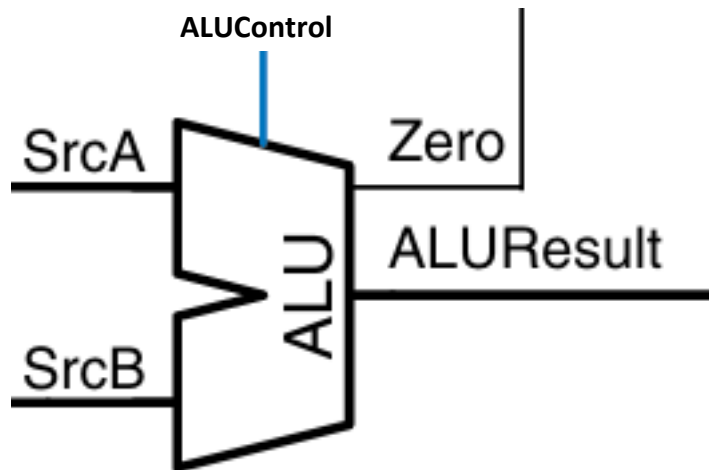
2.Data Path (Functional Blocks):

Every instruction is totally fetched, decoded and executed in a single clock cycle so that it's called a single cycle processor.

Instruction Formats Supported :

| | | | | | | | | | | | | | | | |
|-----------|----|-----------|----|-----|---------|-----|------------|----|----------|----|---------|--------|--------|--------|--------|
| 31 | 30 | 25 | 24 | 21 | 20 | 19 | 15 | 14 | 12 | 11 | 8 | 7 | 6 | 0 | |
| funct7 | | | | rs2 | | rs1 | funct3 | | rd | | | opcode | | R-type | |
| imm[11:0] | | | | | | rs1 | funct3 | | rd | | | opcode | | I-type | |
| imm[11:5] | | | | rs2 | | rs1 | funct3 | | imm[4:0] | | | opcode | | S-type | |
| imm[12] | | imm[10:5] | | rs2 | | rs1 | funct3 | | imm[4:1] | | imm[11] | | opcode | | B-type |
| imm[20] | | imm[10:1] | | | imm[11] | | imm[19:12] | | | rd | | | opcode | | J-type |

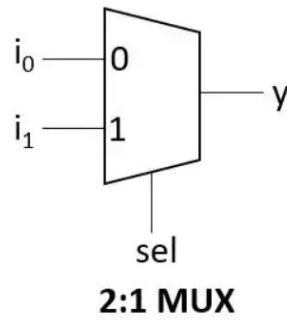
1. ALU (Arithmetic Logic Unit):



RTL Code Snippet:

```
1  module ALU (SrcA,SrcB,ALUControl,ALUResult,Zero);
2  //IO Ports
3  input [31:0] SrcA,SrcB;
4  input [2:0] ALUControl;
5  output reg [31:0] ALUResult;
6  output reg Zero;
7  //Functionality
8  always @(*) begin
9      case(ALUControl)
10         3'b010: ALUResult=SrcA+SrcB;//Add
11         3'b110: ALUResult=SrcA-SrcB;//Subtract
12         3'b000: ALUResult=SrcA&SrcB;//AND
13         3'b001: ALUResult=SrcA|SrcB;//OR
14         3'b111: ALUResult=(SrcA<SrcB)?1:0;//Set Less Than (SLT)
15         default: ALUResult=0;
16     endcase
17     if (ALUResult==0) begin//(ALUResult==0) ==> (!ALUResult)
18         Zero=1;
19     end
20     else begin
21         Zero=0;
22     end
23     //Could also be calculated as : Zero = ~(ALUResult)
24 end
25 endmodule
```

2.MUX (Multiplexer):



RTL Code Snippet:

```
1  module MUX (in0,in1,sel,out);
2  //Parameters
3  parameter WIDTH = 32;
4  //IO Ports
5  input [WIDTH-1:0] in0,in1;
6  input sel;
7  output [WIDTH-1:0] out;
8  //Functionality
9  assign out = (sel) ? in1 : in0 ;
10 endmodule
```

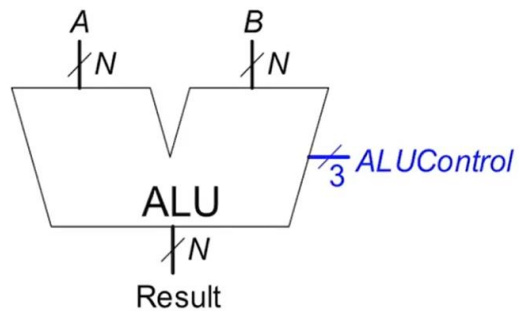
3.Shift Left By 2 :



RTL Code Snippet:

```
1  module shift_left_by2 (in,out_shifted);
2  //IO Ports
3  input [31:0] in;
4  output [31:0] out_shifted;
5  //Functionality
6  assign out_shifted = in<<2 ;// (in<<2) ==> {in[29:0],2'b00}
7  endmodule
```

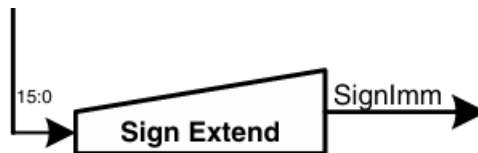
4.Adder (Without Carry) :



RTL Code Snippet:

```
1  module adder (in0,in1,out);
2  //IO Ports
3  input [31:0] in0,in1;
4  output [31:0] out;
5  //Functionality
6  assign out = in0 + in1 ;
7  endmodule
```

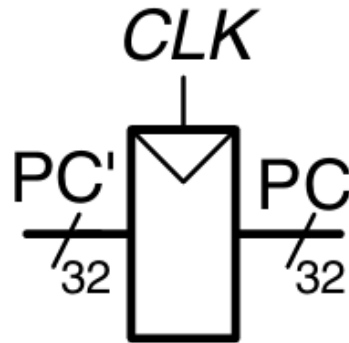
5.Sign Extend:



RTL Code Snippet:

```
1  module sign_extend (in,extended);
2  //IO Ports
3  input [15:0] in;
4  output [31:0] extended;
5  //Functionality
6  assign extended = {{16{in[15]}},in};
7  endmodule
```

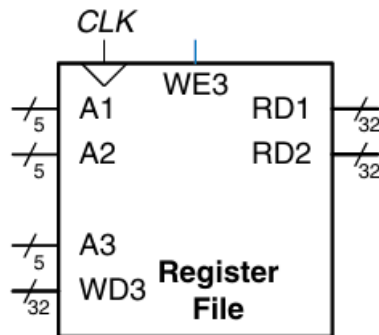

6.D Flip Flop:



RTL Code Snippet:

```
1  module DFF (clk,rst,d,q);
2  //Parameters
3  parameter WIDTH = 32;
4  //IO Ports
5  input [WIDTH-1:0] d;
6  input clk,rst;
7  output reg [WIDTH-1:0] q;
8  //Functionality
9  always @(posedge clk or posedge rst) begin
10     if (rst) begin
11         q<=0;
12     end
13     else begin
14         q<=d;
15     end
16 end
17 endmodule
```

7.Register File:



RTL Code Snippet:

```
1 module register_file (clk,WE3,A1,A2,A3,WD3,RD1,RD2);
2 //IO Ports
3 input clk,WE3;
4 input [4:0] A1,A2;//Read Addresses --> 2^5 = 32 (Depth Size ,Number of registers)
5 input [4:0] A3;//Write Address --> 2^5 = 32 (Depth Size ,Number of registers)
6 input [31:0] WD3;//Input Register --> 32 (Word Size)
7 output [31:0] RD1,RD2;//Output Registers
8 //Register File Body , Three ported register file
9 reg[31:0] RegFile [31:0] ;
10 //Functionality
11 always @(posedge clk) begin
12     if (WE3) begin
13         RegFile[A3]<=WD3;//Write third port on rising edge of clock
14     end
15 end
16 // read two ports combinationaly
17 // register 0 hardwired to 0
18 assign RD1 = (A1!=0) ? RegFile[A1] : 0;
19 assign RD2 = (A2!=0) ? RegFile[A2] : 0;
20 endmodule
```

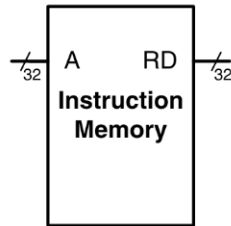
8.Data Path Top Module:

RTL Code Snippet:

```
1 module data_path(clk,rst,MemtoReg,ALUSrc,RegDst,RegWrite,Jump,PCSrc,ALUControl,Zero,ReadData,Instr,ALUResult,WriteData,PC);
2 //Input Ports
3 input clk,rst;//Clock & Reset
4 //Inputs coming from Control Unit
5 input MemtoReg,ALUSrc,RegDst,RegWrite,Jump,PCSrc;
6 input [2:0] ALUControl;
7 //Inputs coming from External Memories
8 input [31:0] ReadData , Instr;
9 //Output Ports
10 output Zero;
11 output [31:0] ALUResult , WriteData , PC ;
12 //Internal Signals
13 wire [31:0] PCPlus4,PCBranch,PCNextBranch,PCNext;
14 wire [31:0] Result;
15 wire [4:0] WriteReg;
16 wire [31:0] SrcA,SrcB;
17 wire [31:0] signimm,signimm_shifted;
18 //Next Program Counter (PC) Logic
19 DFF #(32) PC_FF (clk,rst,PCNext,PC);
20 adder PLUS4 (PC,4,PCPlus4);
21 sign_extend SIGN_EXTEND (Instr[15:0],signimm);
22 shift_left_by2 SHIFTER (signimm,signimm_shifted);
23 adder ADDER (signimm_shifted,PCPlus4,PCBranch);
24 MUX #(32) PC_MUX (PCPlus4,PCBranch,PCSrc,PCNextBranch);
25 MUX #(32) JUMP_MUX (PCNextBranch,{PCPlus4[31:28],Instr[25:0],2'b00},Jump,PCNext);//Mux to handle jump instruction
26 //Register File Logic
27 MUX #(5) WRITE_MUX (Instr[20:16],Instr[15:11],RegDst,WriteReg);
28 register_file REGISTER_FILE (clk,RegWrite,Instr[25:21],Instr[20:16],WriteReg,Result,SrcA,WriteData);
29 //ALU Logic
30 MUX #(32) SrcB_MUX (WriteData,signimm,ALUSrc,SrcB);
31 ALU ALU (SrcA,SrcB,ALUControl,ALUResult,Zero);
32 MUX #(32) RESULT_MUX (ALUResult,ReadData,MemtoReg,Result);
33 endmodule
```

3.Memories:

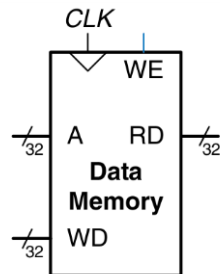
1.Instruction Memory:



RTL Code Snippet:

```
1  module instruction_memory (A,RD);
2  //IO Ports
3  input  [5:0] A;//2^6 = 64 (Depth Size ,Number of registers)
4  output [31:0] RD;//32 (Word Size)
5  //RAM Memory
6  reg [31:0] RAM [63:0];
7  //Functionality
8  assign RD = RAM[A] ;
9  endmodule
```

2.Data Memory:



RTL Code Snippet:

```
1  module data_memory (clk,A,WD,WE,RD);
2  //IO Ports
3  input  [31:0] A;//Write & Read Address
4  output [31:0] RD;//Output Register 32 (Word Size)
5  input  clk ;//Input Clock
6  input  WE;//Write Enable
7  input  [31:0] WD;//Input Register
8  //RAM Memory
9  reg [31:0] RAM [63:0];
10 //Functionality
11 assign RD = RAM[A[31:2]];
12 always @(posedge clk) begin
13     if (WE) begin
14         RAM[A[31:2]]<=WD;
15     end
16 end
17 endmodule
```

3. Processor Components

3.1. ALU (Arithmetic Logic Unit)

- **Functionality:**
 - Performs arithmetic operations (add, sub, etc.), logical operations (and, or, etc.), and comparison operations (slt).
 - The ALU output is determined by the ALUControl signal, which specifies the operation to be performed.

3.2. Register File

- **Functionality:**
 - Contains 32 general-purpose registers.
 - Supports two simultaneous reads and one write operation in each cycle.
 - Register 0 is hardwired to 0.

3.3. Data Memory

- **Functionality:**
 - Supports both read and write operations.
 - Stores data for load and store instructions.

3.4. Instruction Memory

- **Functionality:**
 - Stores the program's instructions.
 - Provides the instruction corresponding to the current program counter (PC) value.

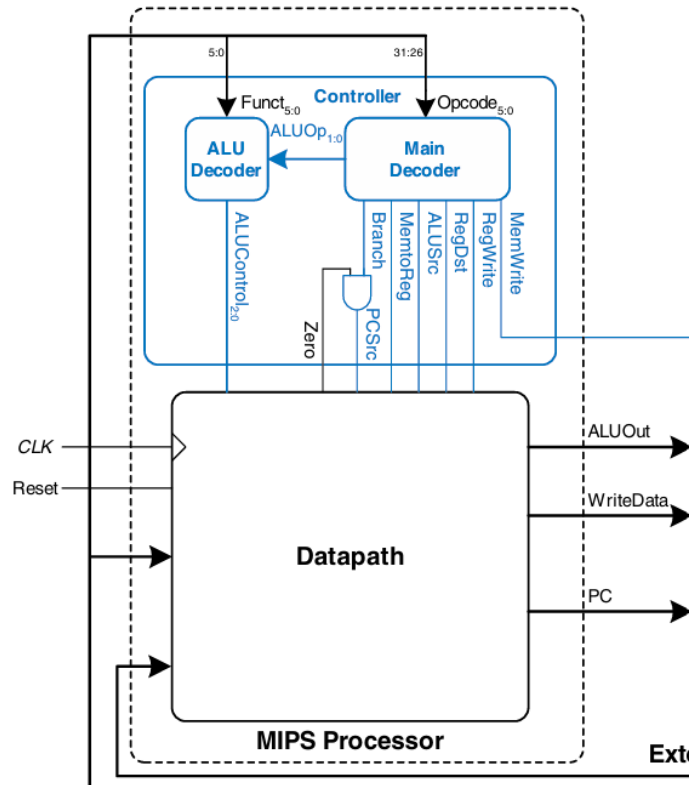
3.5. Control Unit

- **Functionality:**
 - Generates control signals based on the opcode and function code of the instruction.
 - Controls data flow within the processor by generating signals such as ALUSrc, MemWrite, RegWrite, Branch, Jump, etc.

3.6. Data Path

- **Functionality:**
 - Integrates the various components, enabling the processor to fetch, decode, and execute instructions in a single cycle.
 - Manages the flow of data between registers, ALU, and memory.

MIPS Processor (Control Unit & Data Path without Memories):



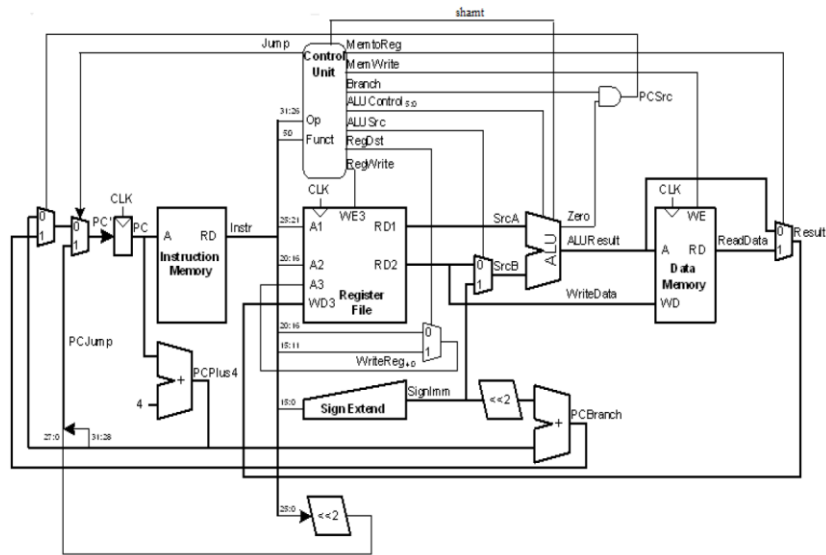
RTL Code Snippet:

```

1 module MIPS (clk,rst,ReadData,Instr,ALUResult,WriteData,MemWrite,PC);
2 //IO Ports
3 input clk,rst;
4 input [31:0] ReadData,Instr;
5 output [31:0] ALUResult,WriteData,PC;
6 output MemWrite;
7 //Internal Signals
8 wire MemtoReg,ALUSrc,RegDst,RegWrite,Jump,PCSrc;
9 wire [2:0] ALUControl;
10 wire Zero;
11 //Functionality --> Instantiations
12 data_path DATA_PATH (.clk(clk),.rst(rst),.MemtoReg(MemtoReg),.ALUSrc(ALUSrc),.RegDst(RegDst),
13 .RegWrite(RegWrite),.Jump(Jump),.PCSrc(PCSrc),.ALUControl(ALUControl),.Zero(Zero),
14 .ReadData(ReadData),.Instr(Instr),.ALUResult(ALUResult),.WriteData(WriteData),.PC(PC));
15 control_unit CONTROL_UNIT (.Opcode(Instr[31:26]),.Funct(Instr[5:0]),.Zero(Zero),.MemtoReg(MemtoReg),.MemWrite(MemWrite),
16 .ALUSrc(ALUSrc),.RegDst(RegDst),.RegWrite(RegWrite),.Jump(Jump),.PCSrc(PCSrc),.ALUControl(ALUControl));
17 endmodule

```

Full MIPS Single-Cycle Architecture Design:



RTL Code Snippet:

```

1 module ARCH (clk,rst,WriteData,DataAddress);
2 //IO Ports
3 input clk,rst;
4 output [31:0] WriteData,DataAddress;
5 //Internal Signals
6 wire MemWrite ;
7 wire [31:0] PC, ReadData , Instr;
8 //Functionality --> Instantiations of Processor and Memories
9 MIPS_PROCESSOR (.clk(clk),.rst(rst),.ReadData(ReadData),.Instr(Instr),.ALUResult(DataAddress),.WriteData(WriteData),.MemWrite(MemWrite),.PC(PC));
10 instruction_memory INSTRUCTION_MEMORY (.A(PC[7:2]),.RD(Instr));
11 data_memory DATA_MEMORY (.clk(clk),.A(DataAddress),.WD(WriteData),.WE(MemWrite),.RD(ReadData));
12 endmodule

```

4. Simulation and Testing

4.1. Testbench Design

- **Purpose:** To validate the correct functioning of the RISC-V processor by simulating its operation with a predefined set of instructions.
- **Features:**
 - Initializes the processor and loads a program into the instruction memory.
 - Checks the output against expected results after each instruction is executed.

4.2. Simulation Results

- **Test Programs:**
 - The processor was tested with a variety of programs, including arithmetic operations, memory access (load/store), and branch/jump instructions.
- **Outcome:**
 - The processor correctly executed all the instructions in the test programs.
 - The final states of the registers and memory matched the expected outcomes, verifying the processor's correct operation.

Assembly and machine code for MIPS test program:

| # | Assembly | Description | Address | Machine |
|---------|----------------------|-----------------------|---------|----------|
| main: | addi \$2, \$0, 5 | # initialize \$2 = 5 | 0 | 20020005 |
| | addi \$3, \$0, 12 | # initialize \$3 = 12 | 4 | 2003000c |
| | addi \$7, \$3, -9 | # initialize \$7 = 3 | 8 | 2067fff7 |
| | or \$4, \$7, \$2 | # \$4 <= 3 or 5 = 7 | c | 00e22025 |
| | and \$5, \$3, \$4 | # \$5 <= 12 and 7 = 4 | 10 | 00642824 |
| | add \$5, \$5, \$4 | # \$5 = 4 + 7 = 11 | 14 | 00a42820 |
| | beq \$5, \$7, end | # shouldn't be taken | 18 | 10a7000a |
| | slt \$4, \$3, \$4 | # \$4 = 12 < 7 = 0 | 1c | 0064202a |
| | beq \$4, \$0, around | # should be taken | 20 | 10800001 |
| | addi \$5, \$0, 0 | # shouldn't happen | 24 | 20050000 |
| around: | slt \$4, \$7, \$2 | # \$4 = 3 < 5 = 1 | 28 | 00e2202a |
| | add \$7, \$4, \$5 | # \$7 = 1 + 11 = 12 | 2c | 00853820 |
| | sub \$7, \$7, \$2 | # \$7 = 12 - 5 = 7 | 30 | 00e23822 |
| | sw \$7, 68(\$3) | # [80] = 7 | 34 | ac670044 |
| | lw \$2, 80(\$0) | # \$2 = [80] = 7 | 38 | 8c020050 |
| | j end | # should be taken | 3c | 08000011 |
| | addi \$2, \$0, 1 | # shouldn't happen | 40 | 20020001 |
| end: | sw \$2, 84(\$0) | # write adr 84 = 7 | 44 | ac020054 |

Contents of memfile.dat:

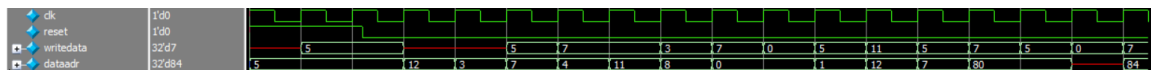
20020005
2003000c
2067fff7
00e22025
00642824
00a42820
10a7000a
0064202a
10800001
20050000
00e2202a
00853820
00e23822
ac670044
8c020050
08000011
20020001
ac020054

If successful, it should write the value 7 to address 84

Testbench RTL Code Snippet:

```
1  module ARCH_tb();
2      //Signals Declaration
3      reg clk;
4      reg reset;
5      wire [31:0] writedata, dataadr;
6      // Instantiate device to be tested
7      ARCH ARCH_DUT (clk, reset, writedata, dataadr);
8
9      // Initialize test
10     initial begin
11         reset = 1;
12         #22;
13         reset = 0;
14     end
15
16     // Generate clock to sequence tests
17     always begin
18         clk = 1;
19         #5;
20         clk = 0;
21         #5;
22     end
23
24     // Load memory with test data
25     initial begin
26         $readmemh ("mem.dat", ARCH_DUT.INSTRUCTION_MEMORY.RAM);
27     end
28
29     // Check results
30     always @(negedge clk) begin
31         //Test Stimulus Generator
32         $display("PC = %d, Instr = %h, ALUResult = %d, WriteData = %d, DataAddress = %d, MemWrite = %b",
33             ARCH_DUT.PC, ARCH_DUT.Instr, ARCH_DUT.DataAddress, writedata, dataadr, ARCH_DUT.MemWrite);
34         if (ARCH_DUT.MemWrite) begin
35             if (dataadr == 84 && writedata == 7) begin
36                 $display("Simulation Succeeded");
37                 $stop;
38             end else if (dataadr != 80) begin
39                 $display("Simulation Failed");
40                 $stop;
41             end
42         end
43     end
44 endmodule
```

QuestaSim Simulation:



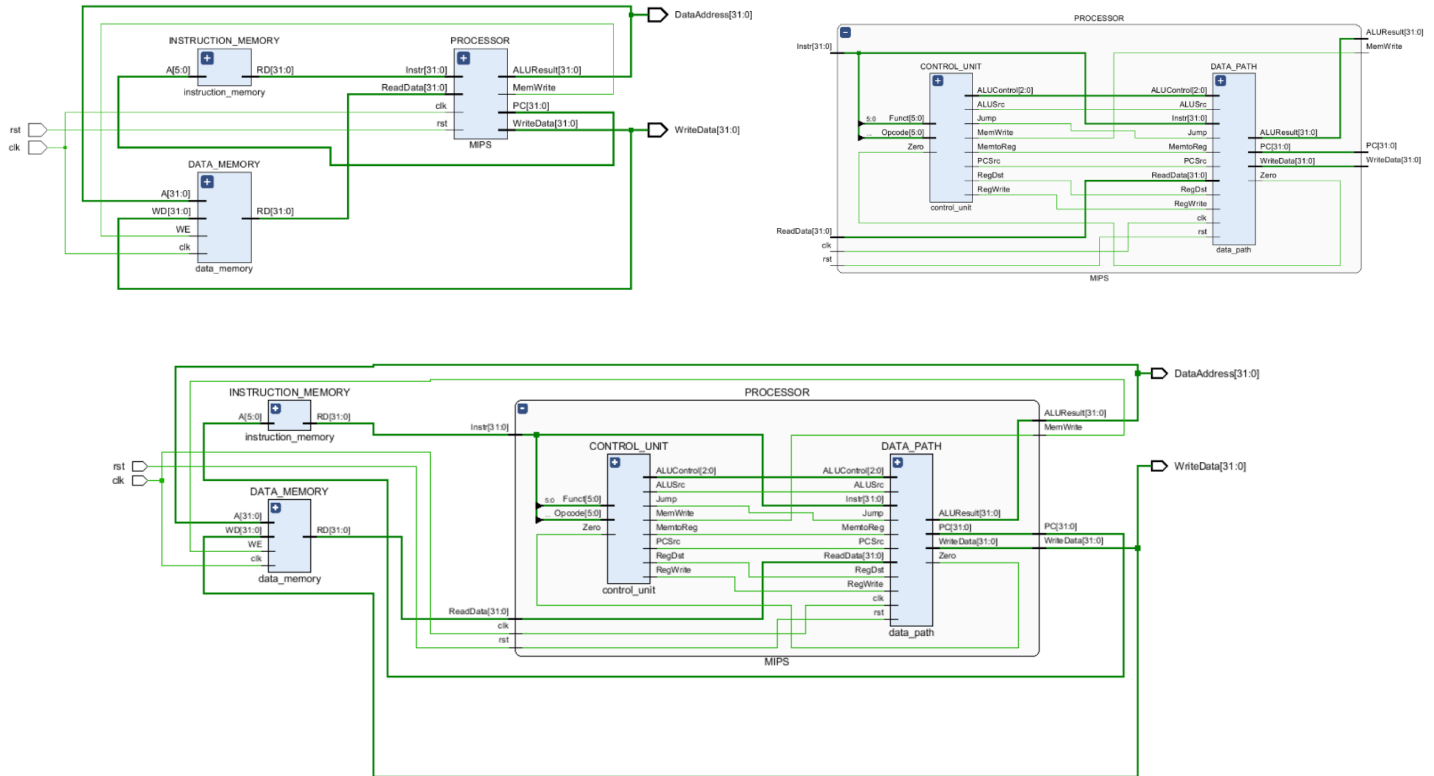
Transcript Snippet:

```
# PC = 0, Instr = 20020005, ALUResult = 5, WriteData = x, DataAddress = 5, MemWrite = 0
# PC = 0, Instr = 20020005, ALUResult = 5, WriteData = 5, DataAddress = 5, MemWrite = 0
# PC = 0, Instr = 20020005, ALUResult = 5, WriteData = 5, DataAddress = 5, MemWrite = 0
# PC = 4, Instr = 2003000c, ALUResult = 12, WriteData = x, DataAddress = 12, MemWrite = 0
# PC = 8, Instr = 2067fff7, ALUResult = 3, WriteData = x, DataAddress = 3, MemWrite = 0
# PC = 12, Instr = 00e22025, ALUResult = 7, WriteData = 5, DataAddress = 7, MemWrite = 0
# PC = 16, Instr = 00642824, ALUResult = 4, WriteData = 7, DataAddress = 4, MemWrite = 0
# PC = 20, Instr = 00a42820, ALUResult = 11, WriteData = 7, DataAddress = 11, MemWrite = 0
# PC = 24, Instr = 10a7000a, ALUResult = 8, WriteData = 3, DataAddress = 8, MemWrite = 0
# PC = 28, Instr = 0064202a, ALUResult = 0, WriteData = 7, DataAddress = 0, MemWrite = 0
# PC = 32, Instr = 10800001, ALUResult = 0, WriteData = 0, DataAddress = 0, MemWrite = 0
# PC = 40, Instr = 00e2202a, ALUResult = 1, WriteData = 5, DataAddress = 1, MemWrite = 0
# PC = 44, Instr = 00853820, ALUResult = 12, WriteData = 11, DataAddress = 12, MemWrite = 0
# PC = 48, Instr = 00e23822, ALUResult = 7, WriteData = 5, DataAddress = 7, MemWrite = 0
# PC = 52, Instr = ac670044, ALUResult = 80, WriteData = 7, DataAddress = 80, MemWrite = 1
# PC = 56, Instr = 8c020050, ALUResult = 80, WriteData = 5, DataAddress = 80, MemWrite = 0
# PC = 60, Instr = 08000011, ALUResult = x, WriteData = 0, DataAddress = x, MemWrite = 0
# PC = 68, Instr = ac020054, ALUResult = 84, WriteData = 7, DataAddress = 84, MemWrite = 1
# Simulation Succeeded
```


Elaboration:

Using VIVADO Xilinx 2018 to make schematic

Schematic:



Do File:

```
vlib work
vlog main_decoder.v alu_decoder.v control_unit.v ALU.v MUX.v shift_left_by2.v adder.v sign_extend.v
DFF.v register_file.v data_path.v instruction_memory.v data_memory.v MIPS.v ARCH.v ARCH_tb.v
vsim -voptargs=+acc work.ARCH_tb
add wave *
run -all
#quit -sim
```

5. Conclusion

This 32-bit single-cycle MIPS processor successfully executes instructions categorized into 5 formats. The processor's design was validated through simulation, demonstrating its ability to fetch, decode, and execute instructions in a single clock cycle.

The design adheres to the principles of the MIPS architecture, offering a simplified and efficient approach to processor design. Future work could involve extending the processor to a pipelined or multi-cycle design for enhanced performance.

تم بحمد الله

سُورَةُ النَّجْمِ

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

وَأَن لَّيْسَ لِلْإِنسَانِ إِلَّا مَا سَعَى ﴿٢٩﴾

سُورَةُ التَّوْبَةِ

وَقُلْ أَعْمَلُوا فَسِيرَی اللَّهِ عَمَلَكُمْ وَرَسُولُهُ وَالْمُؤْمِنُونَ ۖ
وَسَتُرَدُّونَ إِلَىٰ عِلْمِ الْغَيْبِ وَالشَّهَادَةِ فَيُنَبِّئُكُمْ بِمَا كُنْتُمْ
تَعْمَلُونَ ﴿١٠٥﴾