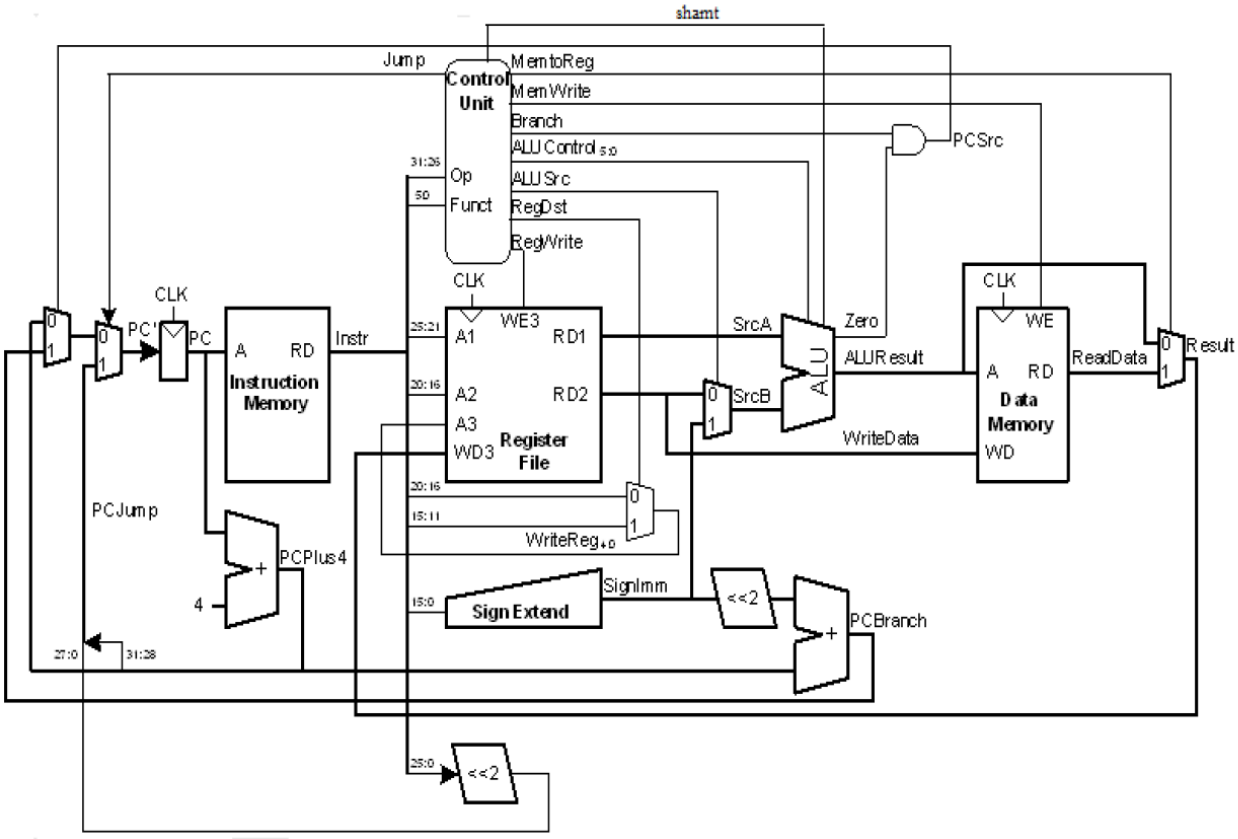
MIPS

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:

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

2.ALU Decoder:

ALU Decoder Truth Table:

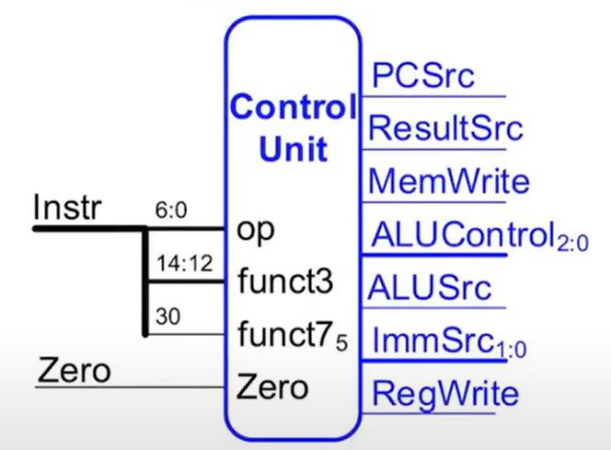
ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

RTL Code Snippet:

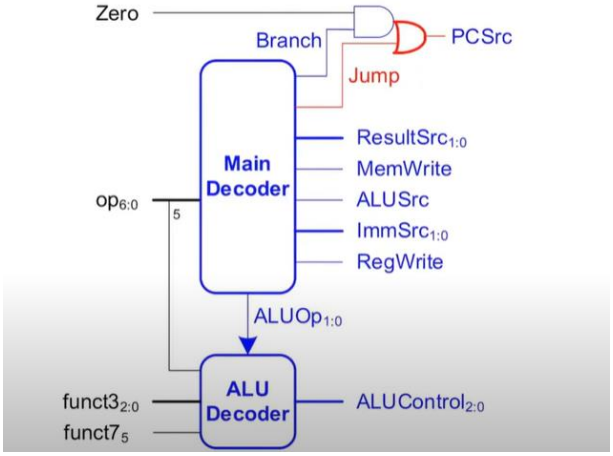
```
module alu_decoder (ALUOp,Funct,ALUControl);
//IO Ports
input [5:0] Funct;
input [1:0] ALUOp;
output reg [2:0] ALUControl;
//Functionality
always @(*) begin
    casex(ALUOp)
        2'b00:ALUControl=3'b010;//Add
        2'b01:ALUControl=3'b110;//Subtract
        2'b1x:begin//R-Type , (2'b1x ==> 2'b10 , 2'b11)
            case(Funct)
                6'b100000:ALUControl=3'b010;//Add
                6'b100010:ALUControl=3'b110;//Subtract
                6'b100100:ALUControl=3'b000;//AND
                6'b100101:ALUControl=3'b001;//OR
                6'b101010:ALUControl=3'b111;//Set Less Than (SLT)
                default:ALUControl=3'bxxx;//Unprovided Case (3'bxxx = 3'bx)
            endcase
        end
        default:ALUControl=3'bxxx;//Unprovided Case (3'bxxx = 3'bx)
    endcase
end
endmodule
```

3.Control Unit Top Module:

High Level View



Low Level View



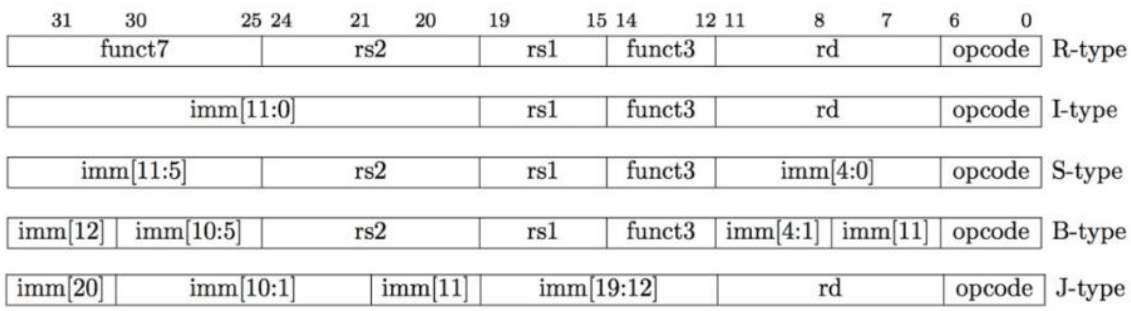
RTL Code Snippet:

```
module control_unit (Opcode,Funct,Zero,MemtoReg,MemWrite,ALUSrc,RegDst,RegWrite,Jump,PCSrc,ALUControl);
//IO Ports
input [5:0] Opcode,Funct;
output MemtoReg,MemWrite,ALUSrc,RegDst,RegWrite,PCSrc,Jump;
output [2:0] ALUControl;
input Zero;
//Internal Signals
wire [1:0] ALUOp;
wire Branch;
//Functionality --> Instantiations
main_decoder MainDecoder (.Opcode(Opcode),.MemtoReg(MemtoReg),.MemWrite(MemWrite),.Branch(Branch),
                          .ALUSrc(ALUSrc),.RegDst(RegDst),.RegWrite(RegWrite),.ALUOp(ALUOp),.Jump(Jump));
alu_decoder AluDecoder (.ALUOp(ALUOp),.Funct(Funct),.ALUControl(ALUControl));
//PC Source
assign PCSrc = Branch & Zero ;
endmodule
```

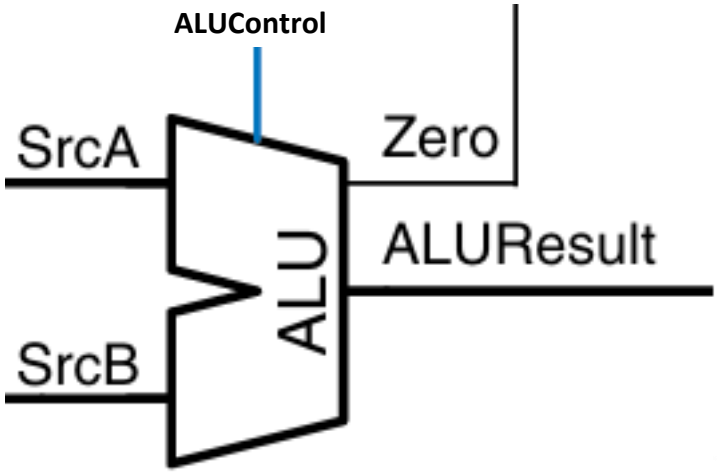
2.Data Path (Functional Blocks):

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 :



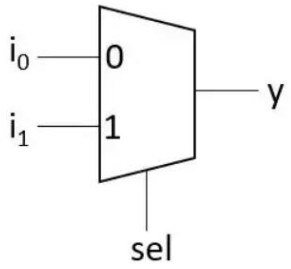
1.ALU (Arithmetic Logic Unit):



RTL Code Snippet:

```
module ALU (SrcA,SrcB,ALUControl,ALUResult,Zero);
//IO Ports
input [31:0] SrcA,SrcB;
input [2:0] ALUControl;
output reg [31:0] ALUResult;
output reg Zero;
//Functionality
always @(*) begin
    case(ALUControl)
        3'b010: ALUResult=SrcA+SrcB;//Add
        3'b110: ALUResult=SrcA-SrcB;//Subtract
        3'b000: ALUResult=SrcA&SrcB;//AND
        3'b001: ALUResult=SrcA|SrcB;//OR
        3'b111: ALUResult=(SrcA<SrcB)?1:0;//Set Less Than (SLT)
        default: ALUResult=0;
    endcase
    if (ALUResult==0) begin//(ALUResult==0) ==> (!ALUResult)
        Zero=1;
    end
    else begin
        Zero=0;
    end
    //Could also be calculated as : Zero = ~|(ALUResult)
end
endmodule
```

2.MUX (Multiplexer):



2:1 MUX

RTL Code Snippet:

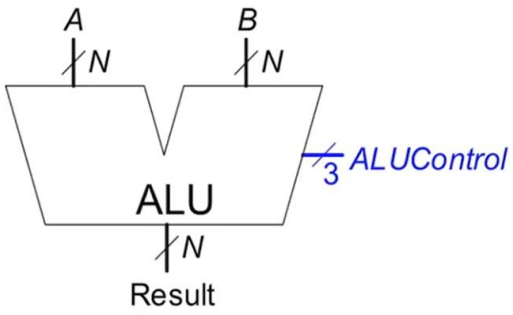
```
1   module MUX (in0,in1,sel,out);
2   //Parameters
3   parameter WIDTH = 32;
4   //IO Ports
5   input [WIDTH-1:0] in0,in1;
6   input sel;
7   output [WIDTH-1:0] out;
8   //Functionality
9   assign out = (sel) ? in1 : in0 ;
10  endmodule
```

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:

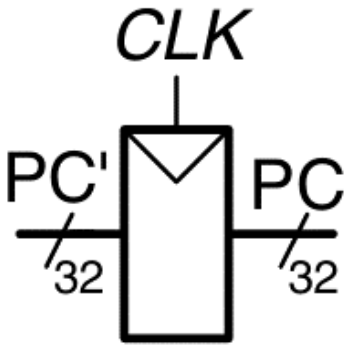
```
1   module adder (in0,in1,out);
2   //IO Ports
3   input [31:0] in0,in1;
4   output [31:0] out;
5   //Functionality
6   assign out = in0 + in1 ;
7   endmodule
```

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
```

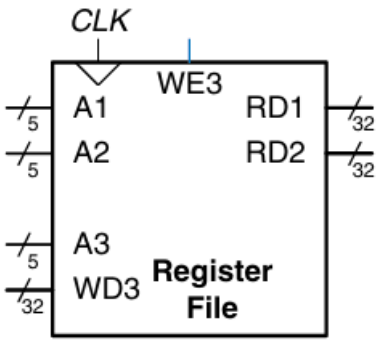
6.D Flip Flop:



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:

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

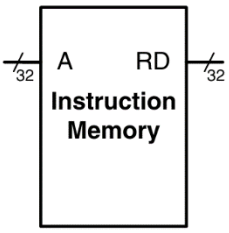
8.Data Path Top Module:

RTL Code Snippet:

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

3.Memories:

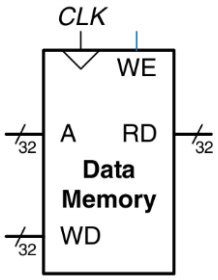
1.Instruction Memory:



RTL Code Snippet:

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

2.Data Memory:



RTL Code Snippet:

```
module data_memory (clk,A,WD,WE,RD);
//IO Ports
input [31:0] A;//Write & Read Address
output [31:0] RD;//Output Register 32 (Word Size)
input clk ;//Input Clock
input WE;//Write Enable
input [31:0] WD;//Input Register
//RAM Memory
reg [31:0] RAM [63:0];
//Functionality
assign RD = RAM[A[31:2]];
always @(posedge clk) begin
    if (WE) begin
        RAM[A[31:2]]<=WD;
    end
end
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.

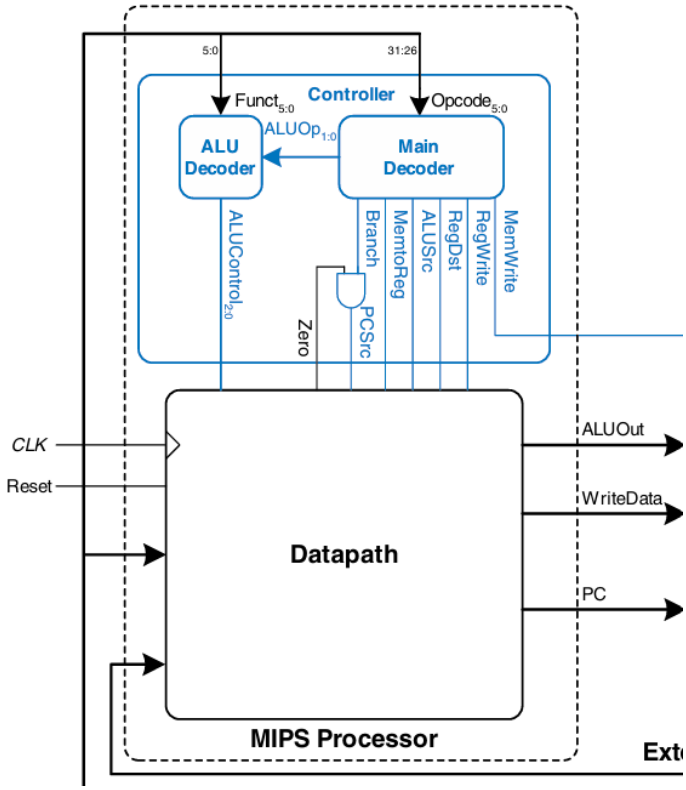
3.5. Control Unit

- **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.

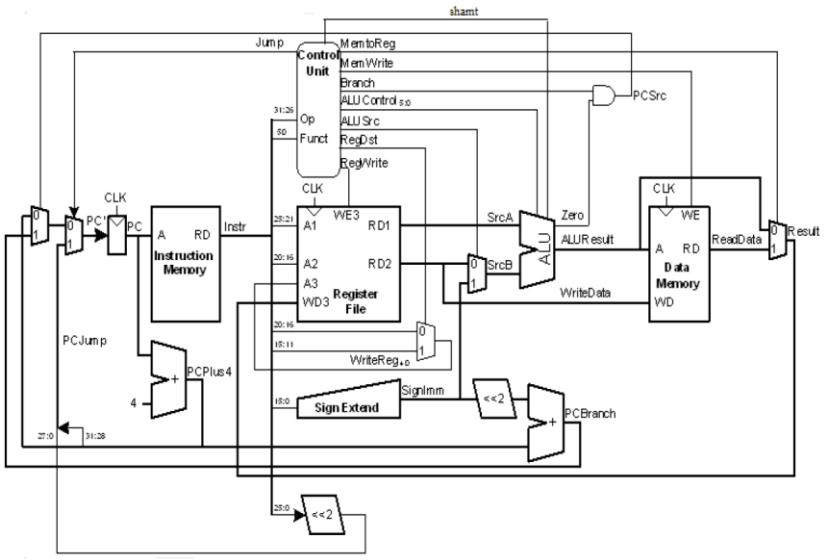
MIPS Processor (Control Unit & Data Path without Memories):



RTL Code Snippet:

```
module MIPS (clk,rst,ReadData,Instr,ALUResult,WriteData,MemWrite,PC);
//IO Ports
input clk,rst;
input [31:0] ReadData,Instr;
output [31:0] ALUResult,WriteData,PC;
output MemWrite;
//Internal Signals
wire MemtoReg,ALUSrc,RegDst,RegWrite,Jump,PCSrc;
wire [2:0] ALUControl;
wire Zero ;
//Functionality --> Instantiations
data_path DATA_PATH (.clk(clk),.rst(rst),.MemtoReg(MemtoReg),.ALUSrc(ALUSrc),.RegDst(RegDst),
                     .RegWrite(RegWrite),.Jump(Jump),.PCSrc(PCSrc),.ALUControl(ALUControl),.Zero(Zero),
                     .ReadData(ReadData),.Instr(Instr),.ALUResult(ALUResult),.WriteData(WriteData),.PC(PC));
control_unit CONTROL_UNIT (.Opcode(Instr[31:26]),.Funct(Instr[5:0]),.Zero(Zero),.MemtoReg(MemtoReg),.MemWrite(MemWrite),
                           .ALUSrc(ALUSrc),.RegDst(RegDst),.RegWrite(RegWrite),.Jump(Jump),.PCSrc(PCSrc),.ALUControl(ALUControl));
endmodule
```

Full MIPS Single-Cycle Architecture Design:



RTL Code Snippet:

```
module ARCH (clk,rst,WriteData,DataAddress);
//IO Ports
input clk,rst;
output [31:0] WriteData,DataAddress;
//Internal Signals
wire MemWrite ;
wire [31:0] PC, ReadData , Instr;
//Functionality --> Instantiations of Processor and Memories
MIPS PROCESSOR (.clk(clk),.rst(rst),.ReadData(ReadData),.Instr(Instr),.ALUResult(DataAddress),.WriteData(WriteData),.MemWrite(MemWrite),.PC(PC));
instruction_memory INSTRUCTION_MEMORY (.A(PC[7:2]),.RD(Instr));
data_memory DATA_MEMORY (.clk(clk),.A(DataAddress),.WD(WriteData),.WE(MemWrite),.RD(ReadData));
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:

```
module ARCH_tb();
    //Signals Declaration
    reg clk;
    reg reset;
    wire [31:0] writedata, dataadr;
    // Instantiate device to be tested
    ARCH ARCH_DUT (clk, reset, writedata, dataadr);

    // Initialize test
    initial begin
        reset = 1;
        #22;
        reset = 0;
    end

    // Generate clock to sequence tests
    always begin
        clk = 1;
        #5;
        clk = 0;
        #5;
    end

    // Load memory with test data
    initial begin
        $readmemh ("mem.dat", ARCH_DUT.INSTRUCTION_MEMORY.RAM);
    end

    // Check results
    always @(negedge clk) begin
        //Test Stimiulus Generator
        $display("PC = %d, Instr = %h, ALUResult = %d, WriteData = %d, DataAddress = %d, MemWrite = %b",
                 ARCH_DUT.PC, ARCH_DUT.Instr, ARCH_DUT.DataAddress, writedata, dataadr, ARCH_DUT.MemWrite);
        if (ARCH_DUT.MemWrite) begin
            if (dataadr == 84 && writedata == 7) begin
                $display("Simulation Succeeded");
                $stop;
            end else if (dataadr !== 80) begin
                $display("Simulation Failed");
                $stop;
            end
        end
    end
endmodule
```

QuestaSim Simulation:

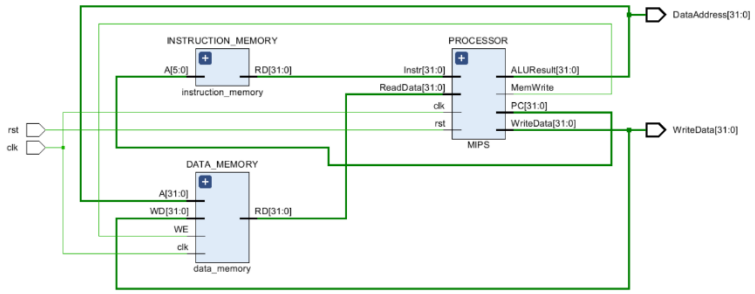
Transcript Snippet:

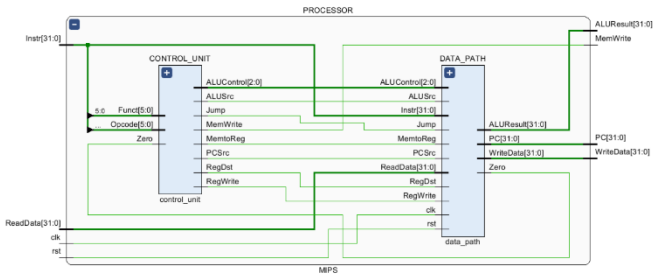
```
# PC =          0, Instr = 20020005, ALUResult =          5, WriteData =          x, DataAddress =          5, MemWrite = 0
# PC =          0, Instr = 20020005, ALUResult =          5, WriteData =          5, DataAddress =          5, MemWrite = 0
# PC =          0, Instr = 20020005, ALUResult =          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
```

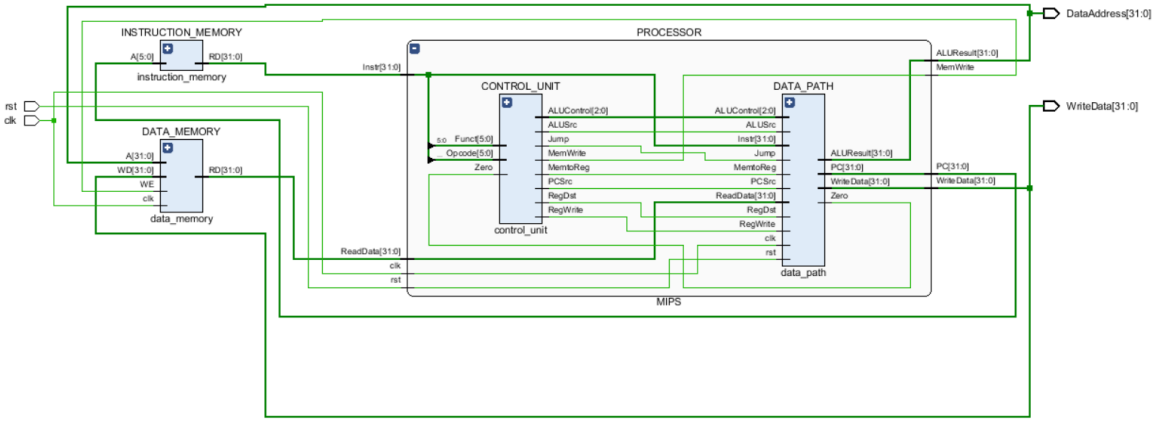
Elaboration:

Using VIVADO Xlinix 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.

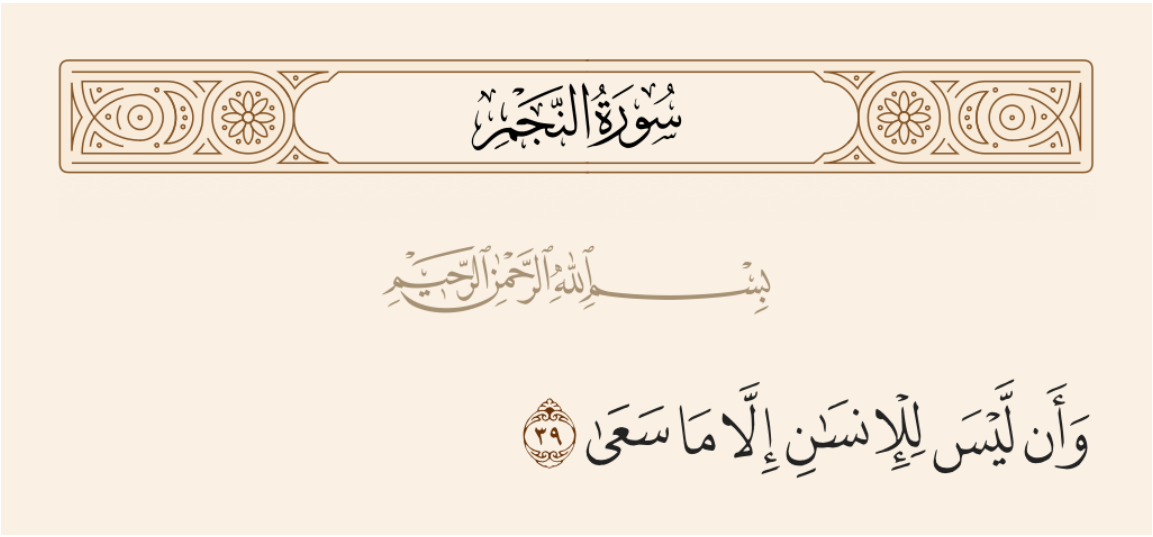
References:

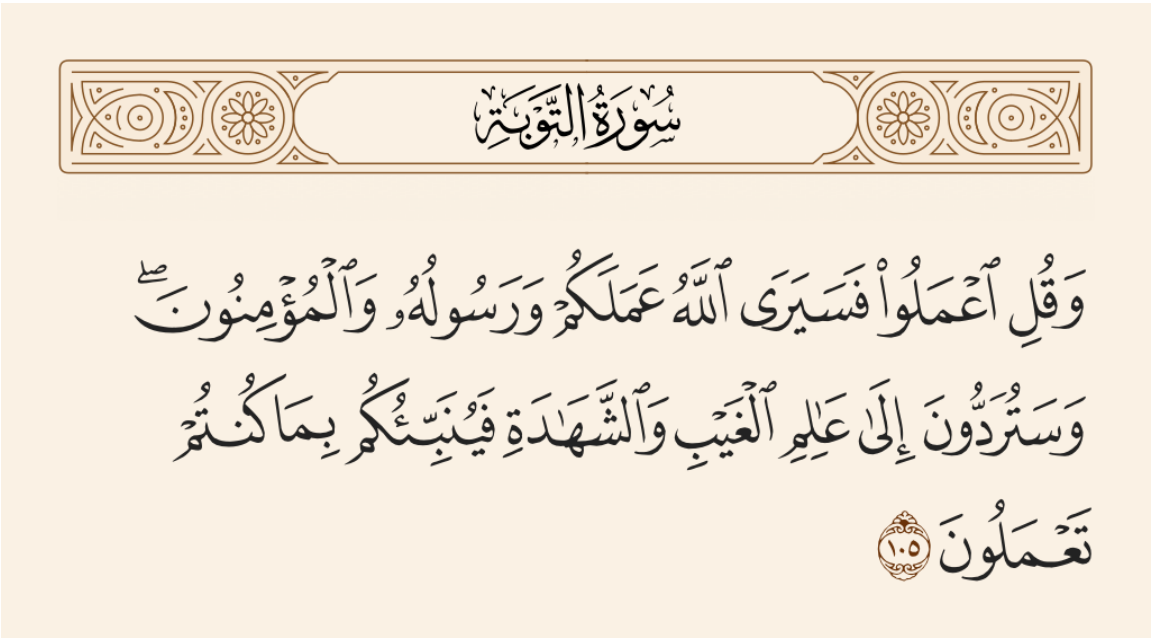
"Digital Design and Computer Architecture" by David Harris and Sarah Harris

https://www.youtube.com/watch?v=lrN-uBKooRY&list=PLhA3DoZr6boVQy9Pz-aPZLH-rA6DvUidB

https://edisciplinas.usp.br/pluginfile.php/7910542/mod_resource/content/1/Digital%20Design%20and%20Computer%20Architecture.pdf

تم بحمد الله


سُورَةُ النَّجۡمِ
بِسۡمِ ٱللَّهِ ٱلرَّحۡمَٰنِ ٱلرَّحِيمِ
وَأَن لَّيۡسَ لِلۡإِنسَٰنِ إِلَّا مَا سَعَىٰ ٣٩


سُورَةُ التَّوۡبَةِ
وَقُلِ ٱعۡمَلُواْ فَسَيَرَى ٱللَّهُ عَمَلَكُمۡ وَرَسُولُهُۥ وَٱلۡمُؤۡمِنُونَۖ
وَسَتُرَدُّونَ إِلَىٰ عَٰلِمِ ٱلۡغَيۡبِ وَٱلشَّهَٰدَةِ فَيُنَبِّئُكُم بِمَا كُنتُمۡ
تَعۡمَلُونَ ١٠٥