Department of Electrical & Computer Engineering

First Semester, 2023/2024

ENCS4370 - COMPUTER ARCHITECTURE

Project #2

Simple Multi Cycle Processor

Prepared by:

Hidaya Mustafa - 1201910

Katya Kobari- 1201478

Rana Odeh - 1201750

Instructor: Aziz Qaroush

Section: 2

Abstract:

This report explores the creation and execution of a Multi-Cycle Processor based on a specified RISC instruction set. The design process involved a thorough analysis of each instruction, identifying the essential components required. Implementation proceeded by constructing these components and determining the corresponding control signals. Rigorous testing was conducted for individual instructions, followed by comprehensive scenarios applied to the processor to validate the results.

Introduction

This project successfully designed, modeled, and simulated a Multi-Cycle Processor adhering to the MIPS architecture using Verilog A systematic approach was employed, involving Modular Development Individual sub-modules of the processor were meticulously designed, coded, and rigorously tested to ensure their independent functionality , and Structural Integration Upon verification, these sub-modules were seamlessly integrated and instantiated within a cohesive structural module, representing the complete processor architecture.

RISC Machines

Reduced Instruction Set Computing represents a design philosophy in computer architecture that emphasizes simplicity and efficiency. RISC machines are characterized by a streamlined set of instructions, each typically executed in a single clock cycle, facilitating faster processing. The architecture relies on a small and fixed set of instructions with uniform formats, aiming to optimize performance by executing instructions quickly and efficiently. By minimizing complexity and focusing on fundamental operations, RISC architectures often achieve higher instruction throughput and are well-suited for tasks requiring frequent and rapid instruction execution. The simplicity of RISC design also facilitates efficient pipelining and parallel processing, contributing to their prominence in various computing devices and systems.

Verilog

Verilog is widely used in the field of digital design and electronic engineering. It serves as a standardized means for specifying the behavior of digital circuits and systems, allowing engineers to model and simulate the functionality of hardware before actual implementation. Verilog enables the design and verification of complex digital systems, including integrated circuits, processors, and other electronic components. The language employs a modular and hierarchical structure, facilitating the organization of designs into manageable modules or blocks. Engineers use Verilog to describe both the structural and behavioral aspects of digital systems, making it a crucial tool in the development process. Additionally, Verilog supports the verification of designs through simulation tools, enabling engineers to identify and rectify potential issues early in the design phase. Its widespread adoption in the semiconductor industry highlights its significance as a key tool in the creation of modern digital electronics.

Multi-Cycle

A multi-cycle Datapath is a computer architecture where instructions are executed in stages over multiple clock cycles. Each stage, such as instruction fetch, decode, execute, and write back, takes one cycle. This design allows instructions to have varying execution times, optimizing resource usage. However, managing dependencies and handling hazards become more complex compared to a single cycle Datapath.

In a multi-cycle processor, each instruction goes through several stages, with each stage completing within a single clock cycle. The stages typically include:

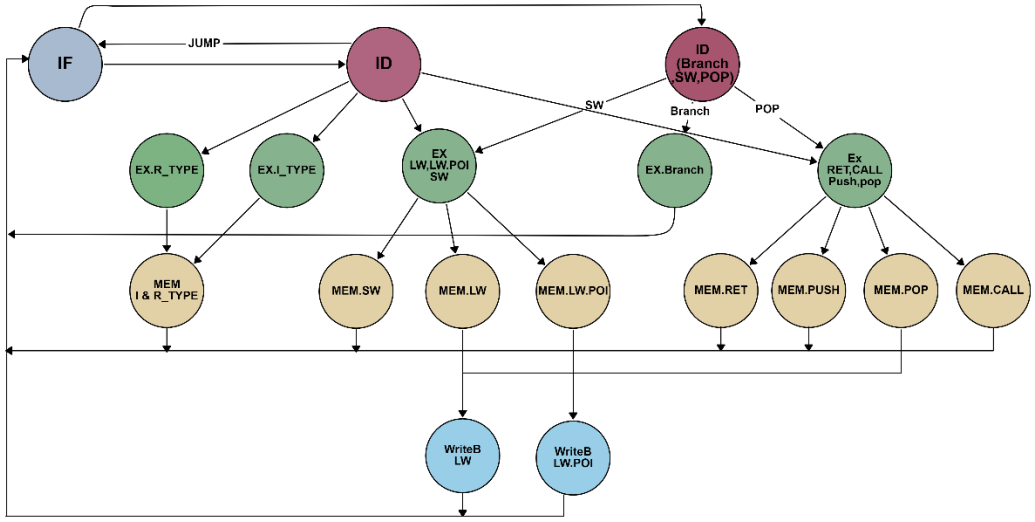
- Instruction Fetch (IF): The processor fetches the instruction from memory using the program counter (PC) and increments the PC to point to the next instruction.
- Instruction Decode (ID): The fetched instruction is decoded to determine the operation to be performed. This stage also involves fetching any necessary operands or data from registers.
- Execution (EX): The actual operation specified by the instruction is performed in this stage. It may involve arithmetic calculations, logical operations, or address computations.
- Memory Access (MEM): If the instruction requires accessing memory, such as loading or storing data, it is performed in this stage. Data is read from or written to memory.
- Write Back (WB): The results of the previous stage are written back to the appropriate register(s). This stage updates the register file with the computed values.

Design Specifications and Implementation

Cycle	Action	RTL
1	Instruction Fetch	IR ← InsMemory[PC] PC ← PC + 1
2	Instruction Decode -Fetch registers Jump, Call - Fetch registers (Branch) Compute branch address	 A ← Reg[Rs1], B ← Reg[Rs2] PC ← PC[31:28] || (Imm26) A ← Reg[Rs1], B ← Reg[Rd] ALUout ← PC[31:2] + sign-extend(Imm16)
3	Execution Case 1: Execute R-type ALU Case 2: Execute I-type ALU Case 3: Compute load/store address Case 4: Branch Case 5: Push , Pop , Call , Ret	 ALUOut← A op B ALUOut← A op extend(Imm16) ALUout← A + sign-extend(Imm16) if (Branch) PC ← ALUout ALUout← SP op 1
4	Memory Access -Write ALU result for R& I-type - Access memory for lw,lw.POI - lw.POI - Access memory for store -Access memory for Push -Access memory for Pop -Access memory for Ret -Access memory for Call	 Reg[Rd] ← ALUout Data ← Memory[ALUout] ALUout← A + 1 Memory[ALUout] ← B Memory[SP] ← B Data ← Memory[SP] PC ← Memory[SP] Memory[SP] ← PC + 1
5	Write Back Pop ,Load Lw.POI	 Reg[Rd] ← Data Reg[Rd] ← Data , Reg[Rs1] ← ALUout

Table 0-1

State Diagram :



Cycle1:

PCWrite = 1 since we do want to write into the PC

IRLoad = 1 we do want to write the value read from memory into IR

ALUSrc1 = 00

ALUSrc2 = 00

ALUop = 00

PcSrc = 1

PCwrite = 1 we want the PC to change

RwgWr1 = 0

RegWr2 = 0

MemRead = 0

MemWrite = 0

Cycle2 :

PCWrite = 0

IRLoad = 0

RegSrc = 0

RwgWr1 = 0

RegWr2 = 0

MemRead = 0

MemWrite = 0

Cycle2(Branch,Store,Push):

PCWrite = 0

IRLoad = 0

ALUSrc1 = 00

ALUSrc2 = 10

RegSrc = 1

ALUop = 00

PcSrc = 1

RwgWr1 = 0

RegWr2 = 0

MemRead = 0

MemWrite = 0

ExtOp = 1

Cycle3 (R_Type):

PCWrite = 0

IRLoad = 0

ALUSrc1 = 01

ALUSrc2 = 01

ALUop = op

RwgWr1 = 0

RegWr2 = 0

MemRead = 0

MemWrite = 0

Cycle3 (I_Type):

PCWrite = 0

IRLoad = 0

ALUSrc1 = 01

ALUSrc2 = 10

ALUop = op

RwgWr1 = 0

RegWr2 = 0

ExtOp = op

MemRead = 0

MemWrite = 0

Cycle3 (Branch):

PCWrite = 0

IRLoad = 0

ALUSrc1 = 01

ALUSrc2 = 10

ALUop = 01

RwgWr1 = 0

RegWr2 = 0

MemRead = 0

MemWrite = 0

Cycle3 (RET,CALL,PUSH,POP):

PCWrite = 0

IRLoad = 0

ALUSrc1 = 10

ALUSrc2 = 00

ALUop = op

RwgWr1 = 0

RegWr2 = 0

MemRead = 0

MemWrite = 0

Cycle3 (LW,SW,LW.POI):

PCWrite = 0

IRLoad = 0

ALUSrc1 = 01

ALUSrc2 = 10

ALUop = 00

RwgWr1 = 0

RegWr2 = 0

ExtOp = 1

MemRead = 0

MemWrite = 0

Cycle4 (LW):

PCWrite = 0

IRLoad = 0

ALUop = op

RwgWr1 = 1

RegWr2 = 0

MemSrc1 = 1

MemSrc2 = 0

MemRead = 1

MemWrite = 0

Cycle4 (I & R-Type):

PCWrite = 0

IRLoad = 0

RwgWr1 = 1

RegWr2 = 0

MemRead = 0

MemWrite = 0

WBdata = 0

Cycle4 (LW_POI):

PCWrite = 0

IRLoad = 0

ALUSrc1 = 01

ALUSrc2 = 00

ALUop = 00

RwgWr1 = 1

RegWr2 = 1

MemSrc1 = 1

MemRead = 1

MemWrite = 0

Cycle4 (SW):

PCWrite = 0

IRLoad = 0

RwgWr1 = 0

RegWr2 = 0

MemSrc1 = 1

MemSrc2 = 1

MemRead = 0

MemWrite = 1

Cycle4 (POP):

PCWrite = 0

IRLoad = 0

RwgWr1 = 0

RegWr2 = 0

MemSrc1 = 0

MemRead = 1

MemWrite = 0

Cycle4 (CALL):

PCWrite = 0

IRLoad = 0

RwgWr1 = 0

RegWr2 = 0

MemSrc1 = 0

MemSrc2 = 0

MemRead = 0

MemWrite = 1

Cycle4 (RET):

PCWrite = 0

IRLoad = 0

PcSrc = 3

RwgWr1 = 0

RegWr2 = 0

MemSrc1 = 0

MemRead = 1

Cycle4 (PUSH):

PCWrite = 0

IRLoad = 0

RwgWr1 = 0

RegWr2 = 0

MemSrc1 = 0

MemSrc2 = 1

MemRead = 0

MemWrite = 1

Cycle 5 (POP,LW):

PCWrite = 0

IRLoad = 0

RwgWr1 = 1

RegWr2 = 0

MemRead = 0

MemWrite = 0

WBdata =1

Cycle 5 (LW_POI):

PCWrite = 0

IRLoad = 0

RwgWr1 = 1

RegWr2 = 1

MemRead = 0

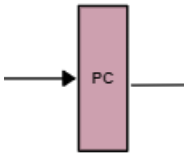
MemWrite = 0

WBdata = 1

Components of a multi-cycle processor data path:

Program Counter (PC):

A register that holds the address of the next instruction to be fetched from memory.



Instruction Memory:

A memory unit that stores the instructions of the program.

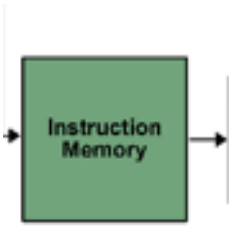


Figure 0-1. Instruction Memory

```
module instruction_memory(clk,addr,out);
    input [31:0] addr;
    output reg [31:0] out;
    reg [31:0] mem [0:1023];
    input clk;
    always @( posedge clk or addr) begin
       out = mem[addr];
    end
     initial begin
        // Set instructions directly in the memory array
        mem[0] = 32'b00000000000001001000000000000000; //AND
        mem[1] = 32'b00000100000001001000000000000000; // ADD
        mem[2] = 32'b00001000000001001000000000000000 ;  // sub
        mem[3] = 32'b00001100000001000000000000000100; //andi
        mem[4] = 32'b00010000000001000000000000000100; //addi
        mem[5] = 32'b00010100000001000000000000000100 ;// lw
        mem[6] = 32'b00011000000001000000000000000100;// lw _poi
        mem[7] = 32'b 00011100000001000000000000000100;// sw
        mem[8] = 32'b00100000000001000000000000000100 ;// bgt
        mem[9] = 32'b 00100100000001000000000000000100; // blt
        mem[10] = 32'b 00101000000001000000000000000100; // beq
        mem[11] = 32'b 00101100000001000000000000000100; // bne
        mem[12] = 32'b 00110000000000000000000000000100 ; // jmp
        mem[13] = 32'b 00110100000000000000000000000100; // call
        mem[14] = 32'b 00111000000000000000000000000000; //ret
        mem[15] = 32'b 00111100010000000000000000000000 ; //push
        mem[16] = 32'b 01000000010000000000000000000000 ; // pop


    end
endmodule

```

Figure 0-2. Instruction Memory Code

This Verilog module implements an instruction memory. The instruction memory is a read-only memory that stores the instructions to be executed by the processor. The module has two inputs and one output: addr, out, and mem. The input address is a 32-bit address that selects an instruction from memory. The output is a 32-bit instruction that is read from memory. The memory mem is a register array of 1024 words, each 32 bits long. The module uses an always block to assign the output to the memory word indexed by the address. The address is byte-addressed, but the memory is word-

addressed, so the lower two bits of the address are ignored. This means the address range is from 0 to 1023, with a step of 4. The memory is initialized with some hard-coded instructions in the initial block. These instructions are written in hexadecimal format.

Register File:

a memory unit that stores the values of the 15 general-purpose registers.

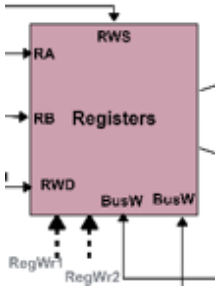


Figure 0-3. Reg File

```
module reg_file(clk,RA,RB,RWD,A,B,RegWr1,RegWr2,WB1,WB2);
    input clk;
    input [3:0] RA, RB,RWD;
    input [31:0] WB1,WB2;
    input RegWr1,RegWr2;
    output reg [31:0] A;
    output reg [31:0] B;
    reg [31:0] registers [0:15];

    always @(posedge clk) begin
        if (RegWr1) begin
            registers[RWD] <= WB1;
        end
         if (RegWr2) begin
            registers[RA] <= WB2;
        end
        A = registers[RA];
        B = registers[RB];
    end
     initial begin
        registers[0] = 32'h00000001;
        registers[1] = 32'h00000001;
        registers[2] = 32'h00000002;
        registers[3] = 32'h00000003;
        registers[4] = 32'h00000004;
        registers[5] = 32'h00000005;
        registers[6] = 32'h00000006;
        registers[7] = 32'h00000007;
        registers[8] = 32'h00000008;
        registers[9] = 32'h00000009;
        registers[10] = 32'h0000000A;
    end

endmodule
```

Figure 0-4. Reg file code

This module implements a registers file, that can store and access data in registers. Registers are small memory units that can hold 32 bits of data each. The code defines 16 registers and how to read or write data to them using addresses and signals, we have two write back buses one writes on RD register and the second write on RS1 register.

ALU:

An arithmetic and logic unit that performs various operations on the operands.

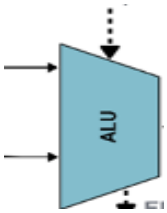


Figure 0-5. ALU

```verilog
// alu module
module alu(a,b,op,result,flags);
    input [31:0] a,b;
    input [1:0] op;
    output reg [31:0] result;
    output reg [2:0]flags;
    always @(*) begin
        case (op)
            2'b00:result = a + b;
            2'b01: begin
                result = a - b;
                flags[2] = (a[31] ^ b[31]) & (a[31] ^ result[31]); // Overflow
                flags[1]= (a < b); // Carry
            end
            2'b10:result = a & b;
        endcase
        if (result == 32'b0 )
            flags[0]=1; // Zero
        else
             flags[0]=0;
    end
endmodule
```

Figure 0-6.Alu code

This module implements an arithmetic and logic unit (ALU) for a 32-bit processor. An ALU is a circuit that can perform different operations on two input values, such as addition, subtraction, and bitwise AND.
We assign the value for Zero , carry , overflow flag --> whether branch is taken or not

Data Memory:

A memory unit that stores the data of the program.

Figure 0-7. Data Memory

```verilog
module Data_memory (clk,addr,data_in,MemRd,MemWr,data_out);
    input clk;
    input  [31:0] addr;
    input  [31:0] data_in;
    input  MemRd;
    input  MemWr;
    output reg [31:0] data_out;
    reg [31:0] mem [0:1023];

    always @(posedge clk) begin
        if (MemWr) begin
            mem[addr] = data_in;
        end
        else if (MemRd) begin
            data_out = mem[addr];
        end
    end
    // Initialize memory with some random values
    initial begin
        integer i;
        for (i = 0; i < 1024; i = i + 1) begin
            mem[i] = i;
        end
    end
endmodule
```

Figure 0-8. Data Memory Code

This module implements a data memory for a 32-bit processor. The data memory can store and load data for the processor using a clock signal, an address, a data input, a data output, and two signals that enable reading or writing. The data memory has 1024 words, each 32 bits long. The code can

read or write one word in one clock cycle by checking the enabled signals and using the address to select the word.

Extender:

A logic unit that sign/zero -extends the 16-bit immediate value to 32 bits.

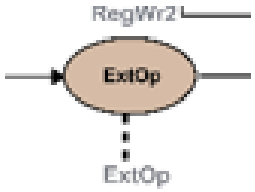


Figure 0-9. Extender

```
1  module extender (in,op,out);
2      input [15:0] in;
3      input op;
4      output reg [31:0] out;
5
6      always @(*) begin
7          case (op)
8              1'b0: out = {18'b0, in[13:0]}; // unsigned
9              1'b1: out = {{18{in[13]}}, in[13:0]}; // signed
10         endcase
11     end
12 endmodule
13
```

Figure 0-10. Extender code

This Module implements extender unit, where we use for immediate values. An extender makes a binary number longer without changing its meaning. A binary number is a number made of only 0s and 1s. When we make a binary number longer, we copy the first bit (0 or 1) many times and add it to the front. This is called sign extension or zero extension.
Our data path had binary numbers that were 16 bits long. We used the extender to make them 32 bits long. The extender could do sign extension or zero extension. It depended on the control signal ExtOp. If ExtOp was 1, it did sign extension. If ExtOp was 0, it did zero extension.

Multiplexers

A logic unit that selects one of the inputs based on the control signals.

Figure 0-11. MUX

```
module mux4_1 (
  input  [31:0] in0, in1, in2,in3,
  input  [1:0] sel,
  output reg [31:0] out);
 always @(*)begin
if (sel == 2'b00)
    out = in0;
  else if (sel == 2'b01)
    out = in1;
  else if (sel == 2'b10)
    out = in2;
  else
    out = in3;
 end
endmodule
```

```
module mux2_1 (
  input  [31:0] in0, in1,
  input  sel,
  output reg [31:0] out
);
 always @(*)begin
if (sel == 1'b0)
    out = in0;
else
    out = in1;
 end
endmodule
```

Figure 0-12. MUX CODE

This module implements a multiplexer, a multiplexer is a combinational circuit with multiple data inputs and a single output, the selection of which depends on control or select inputs. In our project, we incorporated three 2x1 multiplexers in the design of the multicycle processor, as well as three 4x1 multiplexers. These multiplexers were strategically employed at points where the system needed to choose among multiple options based on specific requirements at that moment.

Main Control

This module implements the main control unit for the multi-cycle processor. The main control unit is responsible for generating the control signals for the different stages of the processor, such as instruction fetch, decode, execute, memory access, and write back. The module has the following inputs and outputs

Op → The 6-bit input that specifies the opcode of the instruction.

Stage→ The 3-bit input that indicates the current stage.

aluOp → The 2-bit output that selects the operation for the arithmetic and logic unit (ALU).

ALUSrc1→ selects the first ALU source as PC , A or SP .

ALUSrc2 → selects the 2nd ALU source as (1) , B or extended immediate

extOp → The output that enables the sign extension for immediate values.

regWr1 and regWr2 → The outputs that enable the write operations to the register file, where regWr1 enables write in the Rd register and regWr2 enables write in Rs1.

RegSrc → The output that selects the second source on the register file (Rd or Rs2).

memSrc1 → This signals select the address for data memory SP or ALU_res.

memSrc2 → This signals select the data_in for data memory ALU_res or B.

memRd and memWr → The outputs that enable the read and write operations to the data memory.

WbData → The output that selects the source for the write-back data.

PCWrite → The output that enables the program counter to update to the next instruction address.

We use assign statements to generate the control signals based on the values of op and stage. The module also uses always blocks to assign the values of aluOp, aluSrc1, and aluSrc2 using case statements. The module can handle different types of instructions, such as arithmetic, logic, load, store, branch, push, pop, ret, and call.

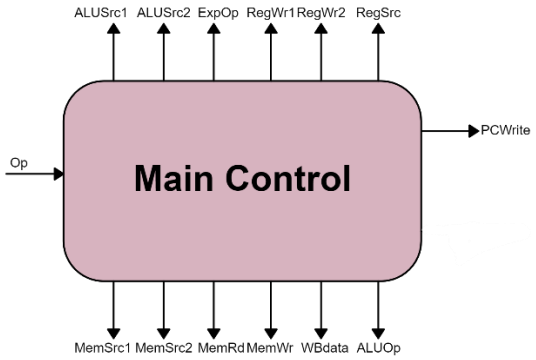


Figure 0-13:Main Control Unit

```
module main_control(stage,op,aluOp,aluSrc1,aluSrc2,extOp,regWr1,regWr2,regSrc,memSrc1,memSrc2,memRd,memWr,wbData,pcWrite);
input [5:0] op;
input [2:0]stage;  //aluop
output reg [1:0] aluOp,aluSrc1,aluSrc2;
output reg extOp,regWr1,regWr2,memSrc1,memSrc2,memRd,memWr,wbData,regSrc,pcWrite;
//stage 1----------
assign pcWrite = (stage==3'b001);
assign extOp=((stage==3'b010 && ((op==6'b001000))||(op==6'b001001))|| (op==6'b001010)
|| (op==6'b001011))  || (stage==3'b011 && (op==6'b000100 || op==6'b000101 || op==6'b000110 || op==6'b000111  )) ) ;  //branch or addi or lw or sw or lw poi
assign memRd= (stage==3'b100 && (op==6'b000101 || op==6'b000110 || op==6'b001110 || op==6'b010000));
assign memWr= (stage==3'b100 && (op==6'b000111 || op==6'b001101|| op==6'b001111 ));
assign regWr1= (stage==3'b101 && (op==6'b000101 || op==6'b000110||op==6'b000000|| op==6'b000001|| op==6'b000010 || op==6'b000011|| op==6'b000100 ));
assign regWr2= (stage==3'b101 && (op==6'b000110));
assign memSrc1=(stage==3'b100 && (op==6'b000101 || op==6'b000110|| op==6'b000111 ));
assign memSrc2=(stage==3'b100 && (op==6'b001111 || op==6'b000111));
assign regSrc=(stage==3'b010 && (op==6'b001111 || op==6'b000111 || op==6'b001000||op==6'b001001|| op==6'b001010 || op==6'b001011  ));
assign wbData= (stage == 3'b101 && (op==6'b000101 || op==6'b000110 || op==6'b010000 ));
always @(op,stage)begin
    if((stage == 3'b100 && op==6'b001101) || (stage == 3'b001 )||
        (stage == 3'b010 && (op==6'b001000||op==6'b001001|| op==6'b001010 || op==6'b001011 )) )
          aluSrc1 = 2'b00;
    else if ((stage == 3'b100 && op==6'b000110) || stage == 3'b011 && (op==6'b000000 || op==6'b000001 || op==6'b000010 ||
        op==6'b000011 || op==6'b000100 || op==6'b000101 || op==6'b000110 ||  op==6'b000111 || op==6'b001000|| op==6'b001001 || op==6'b001010 || op==6'b001011 ))
          aluSrc1=2'b01;
     else if (stage == 3'b011  && (op==6'b001101 || op==6'b001110 || op==6'b001111||op==6'b010000 ))
          aluSrc1 = 2'b10;
    end
always @(op,stage)begin
    if((stage == 3'b001) || (stage == 3'b011 && (op==6'b001101 ||
        op==6'b001110 || op==6'b001111||op==6'b010000)) ||( stage == 3'b100 && op==6'b000110))
          aluSrc2 = 2'b00;
       else if (stage == 3'b011 && (op==6'b000000 || op==6'b000001 ||op==6'b000010 || op==6'b001000|| op==6'b001001 || op==6'b001010 || op==6'b001011))
          aluSrc2=2'b01;
     else if ((stage == 3'b010 && (op==6'b001000||op==6'b001001||
        op==6'b001010 || op==6'b001011))  || (stage == 3'b011 && (op==6'b000011||op==6'b000100|| op==6'b000101 || op==6'b000110 || op==6'b000111)) )
          aluSrc2=2'b10;
    end
```

```
always @(op,stage)begin
    if((stage == 3'b001) || (stage == 3'b011 && (op==6'b001101 ||
        op==6'b001110 || op==6'b001111||op==6'b010000)) ||( stage == 3'b100 && op==6'b000110))
          aluSrc2 = 2'b00;
       else if (stage == 3'b011 && (op==6'b000000 || op==6'b000001 ||op==6'b000010 || op==6'b001000|| op==6'b001001 || op==6'b001010 || op==6'b001011))
          aluSrc2=2'b01;
     else if ((stage == 3'b010 && (op==6'b001000||op==6'b001001||
        op==6'b001010 || op==6'b001011))  || (stage == 3'b011 && (op==6'b000011||op==6'b000100|| op==6'b000101 || op==6'b000110 || op==6'b000111)) )
          aluSrc2=2'b10;
    end
always @(op,stage)begin
    if((stage == 3'b001) || (stage == 3'b010 && (op==6'b001000 ||
        op==6'b001001 || op==6'b001010||op==6'b001011)) ||
        (stage == 3'b011 && (op==6'b000001 || op==6'b000011 ||
        op==6'b000101||op==6'b000110 ||op==6'b000111 ||op==6'b001101 ||op==6'b001111)) || (stage == 3'b100 && (op==6'b000110)))
          aluOp = 2'b00;
       else if ((stage == 3'b011 && (op==6'b000010 || op==6'b001110 ||op==6'b010000 ||op==6'b001000|| op==6'b001001 || op==6'b001010 || op==6'b001011 )))
          aluOp=2'b01;
     else if ((stage == 3'b011 && (op==6'b000000)))
          aluOp=2'b10;
    end

endmodule
```

Op	ALUOp	ALU_Code
AND	AND	01
ADD	ADD	00
SUB	SUB	10

ANDI	AND	01
ADDI	ADD	00
LW	ADD	00
LW.POI	ADD	00
SW	ADD	00
BGT	SUB	10
BLT	SUB	10
BEQ	SUB	10
BNE	SUB	10
JMP	x	x
CALL	ADD	00
RET	SUB	10
PUSH	ADD	00
POP	SUB	10

Test :

```
run 800 ns
# KERNEL: ------------------ADD--------------------
# KERNEL: Cycle 1 : aluOp=00, aluSrc1=00, aluSrc2=00, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=1
# KERNEL: Cycle 2 : aluOp=00, aluSrc1=00, aluSrc2=00, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=0
# KERNEL: Cycle 3 : aluOp=00, aluSrc1=01, aluSrc2=01, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=0
# KERNEL: Cycle 4 : aluOp=00, aluSrc1=01, aluSrc2=01, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=0
# KERNEL: Cycle 5 : aluOp=00, aluSrc1=01, aluSrc2=01, extOp=0, regWrl=1, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=0
# KERNEL: ------------------BEQ--------------------
# KERNEL: Cycle 1 : aluOp=00, aluSrc1=00, aluSrc2=00, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=1
# KERNEL: Cycle 2 : aluOp=00, aluSrc1=00, aluSrc2=10, extOp=1, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=1, pcWrite=0
# KERNEL: Cycle 3 : aluOp=01, aluSrc1=01, aluSrc2=01, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=0
# KERNEL: Cycle 4 : aluOp=01, aluSrc1=01, aluSrc2=01, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=0
# KERNEL: Cycle 5 : aluOp=01, aluSrc1=01, aluSrc2=01, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=0
# KERNEL: ------------------STORE--------------------
# KERNEL: Cycle 1 : aluOp=00, aluSrc1=00, aluSrc2=00, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=1
# KERNEL: Cycle 2 : aluOp=00, aluSrc1=00, aluSrc2=00, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=1, pcWrite=0
# KERNEL: Cycle 3 : aluOp=00, aluSrc1=01, aluSrc2=10, extOp=1, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=0
# KERNEL: Cycle 4 : aluOp=00, aluSrc1=01, aluSrc2=10, extOp=0, regWrl=0, regWr2=0, memSrc1=1, memSrc2=1, memRd=0, memWr=1, wbData=0, regSrc=0, pcWrite=0
# KERNEL: Cycle 5 : aluOp=00, aluSrc1=01, aluSrc2=10, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=0
# KERNEL: ------------------PUSH--------------------
# KERNEL: Cycle 1 : aluOp=00, aluSrc1=00, aluSrc2=00, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=1
# KERNEL: Cycle 2 : aluOp=00, aluSrc1=00, aluSrc2=00, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=1, pcWrite=0
# KERNEL: Cycle 3 : aluOp=00, aluSrc1=10, aluSrc2=00, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=0
# KERNEL: Cycle 4 : aluOp=00, aluSrc1=10, aluSrc2=00, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=1, memRd=0, memWr=1, wbData=0, regSrc=0, pcWrite=0
# KERNEL: Cycle 4 : aluOp=00, aluSrc1=10, aluSrc2=00, extOp=0, regWrl=0, regWr2=0, memSrc1=0, memSrc2=0, memRd=0, memWr=0, wbData=0, regSrc=0, pcWrite=0
# RUNTIME: Info: RUNTIME_0068 main_control_tb.v (117): $finish called.
# KERNEL: Time: 200 ns,  Iteration: 0,  Instance: /main_control_tb,  Process: @INITIAL#15_0@.
# KERNEL: stopped at time: 200 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
```

PC Control

The Verilog module pc_control controls the two-bit output signal pc_src using input signals op and flags. Internal wires branch, jump, and ret are assigned based on conditions involving op and flags. using conditional statements. If the branch condition is true, pc_src is set to 2, if jump is true, it's set to 0, if ret is true, it's 3. Otherwise, pc_src is set to 1. This module serves as control logic for program counter behavior in a computer architecture.

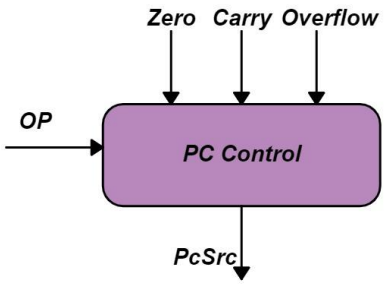


Figure 0-14:Pc Control Unit

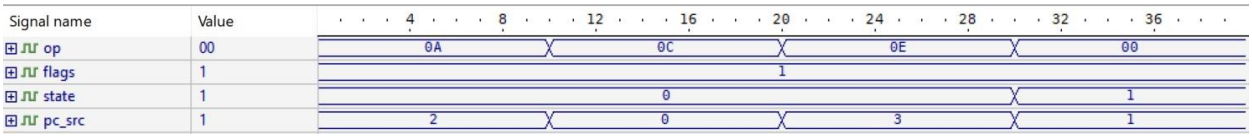
```
module pc_control(state,op,flags,pc_src);
    input [5:0] op;
    input [2:0] flags;// [0]-->zero, [1]--> carry ,[2]-->overFlow.
    input [2:0] state;
    output reg [1:0] pc_src ;
    wire branch, jump,ret,next;
    assign branch = ((op== 6'b001010 && flags[0]) || (op ==6'b 001011 && !flags[0])||
    (op == 6'b001000 && !flags[1]  && !flags[2])|| (op == 6'b001001 && flags[1] ) ) ;
    assign jump = (op==6'b001100 ||op==6'b001101 ) ; //jump or call
    assign ret=(op==6'b001110) ;
    assign next = (state == 3'b001);

    always @(*)begin
       if(branch)
           pc_src=2'b10;
        else if (jump)
           pc_src=2'b00;
      else if (ret)
         pc_src=2'b11;
       else if (next)
         pc_src=2'b01;

    end
endmodule
```

Op	Zero flag	Carry	Overflow	PCSrc
JMP	x	x	x	1
Call	x	x	x	1
Ret	x	x	x	2
BEQ	0	x	x	0
BEQ	1	x	x	3
BNE	0	x	x	3
BNE	1	x	x	0
BGT	x	0	0	3
BGT	x	1	1	0
BLT	x	1	x	3
BLT	x	0	x	0
Other	x	x	x	0

Test :




```
# KERNEL: (branch)pc_src = 10
# KERNEL: (jump)pc_src = 00
# KERNEL: (ret)pc_src = 11
# KERNEL: (pc+1)pc_src = 01
```

Datapath

Finaly assembling all the components in Datapath based on all the Instructions.

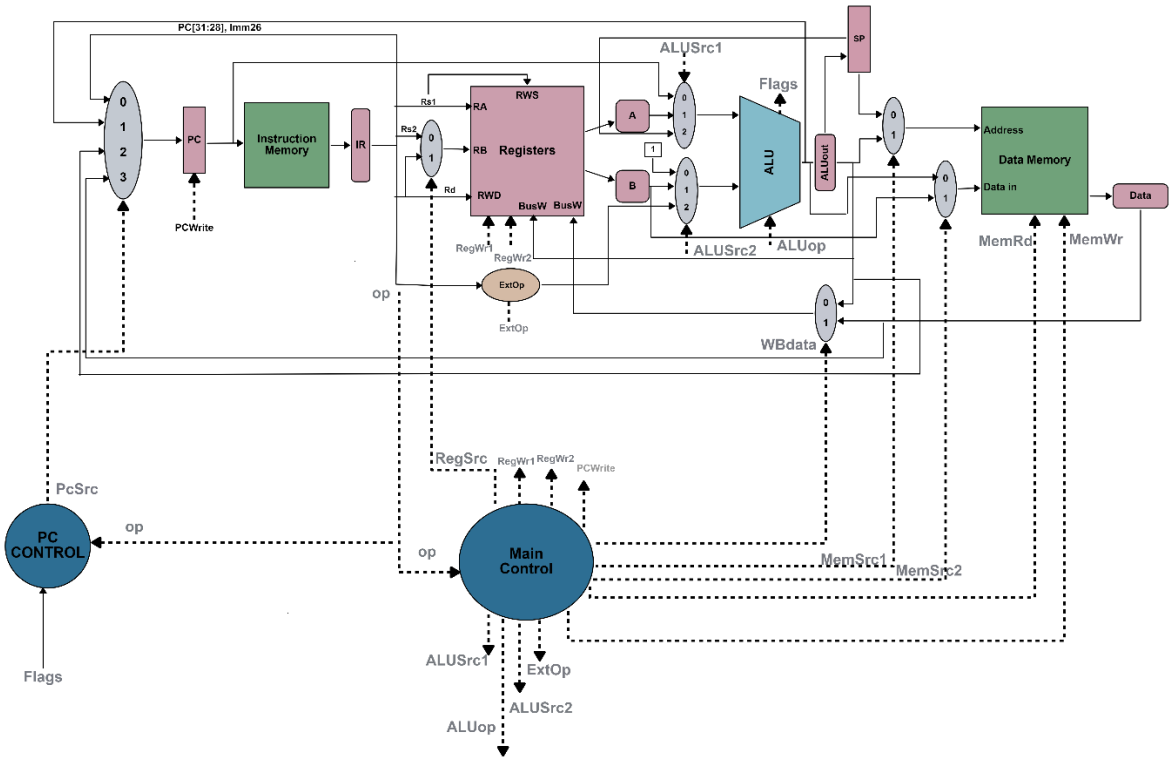


Figure 0-15:Datapath

Testing

Sample instructions for test:

and --> 000000 | 0000| 0001| 0010 | 00000000000000

Add --> 000001 | 0000| 0001| 0010 | 00000000000000

Sub --> 000010 | 0000| 0001| 0010 | 00000000000000

ANDI--> 000011 | 0000| 0001 | 0000000000000001 | 00

ADDI --> 000100| 0000| 0001 | 0000000000000001 | 00

lw--> 000101 | 0000| 0001 | 0000000000000001 | 00

lW POI --> 000110 | 0000| 0001 | 0000000000000001 | 00

sW--> 000111 | 0000| 0001 | 0000000000000001 | 00

BGT--> 001000 | 0000| 0001 | 0000000000000001 | 00

blt --> 001001 | 0000| 0001 | 0000000000000001 | 00

BEQ --> 001010 | 0000| 0001 | 0000000000000001 | 00

BNE --> 001011 | 0000| 0001 | 0000000000000001 | 00

JMP --> 001100 | 00000000000000000000000100

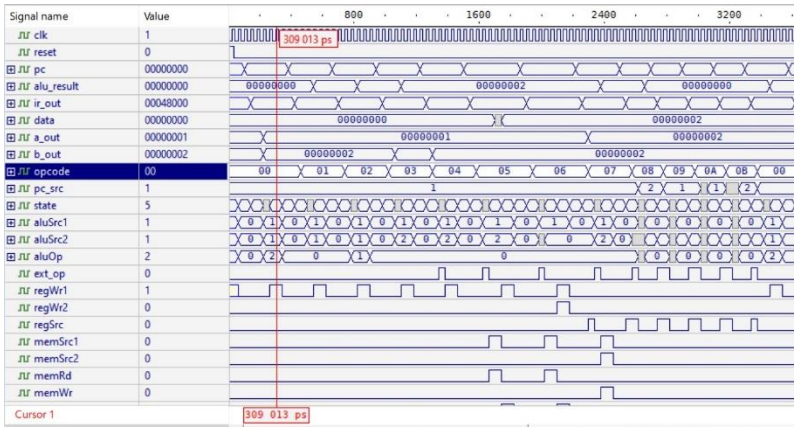
call --> 001101 | 00000000000000000000000100

RET --> 001110 | 00000000000000000000000000

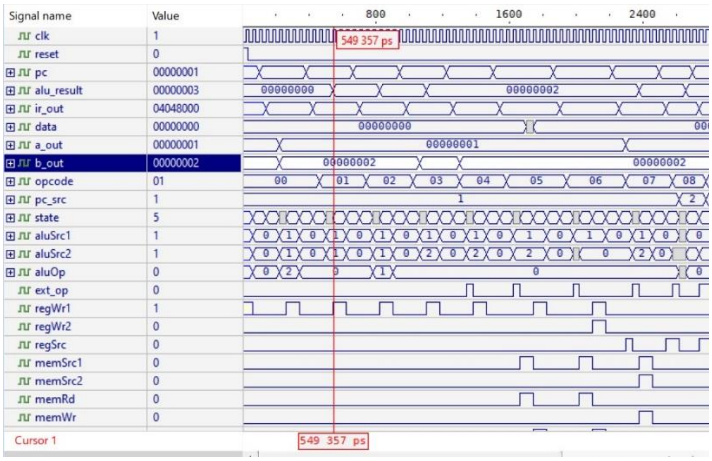
Push --> 001111 | 0001 | 00000000000000000000

pop --> 010000 | 0001 | 00000000000000000000

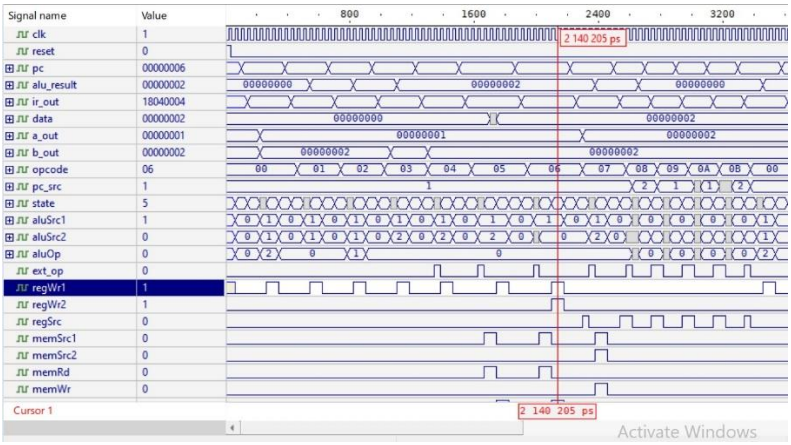
AND :



ADD :



LW.POI:



Team Work

We have done all work together; it was all shared between us.

Conclusion

A multi-cycle processor is a processor that can execute an instruction in five clock cycle. This allows the processor to use fewer hardware resources and simplify the control logic, as different stages of the instruction can share the same functional units. However, a multi-cycle processor also has some drawbacks, such as lower performance, increased complexity of data and control hazards, and reduced instruction-level parallelism. Therefore, a multi-cycle processor is a trade-off between hardware cost and performance.