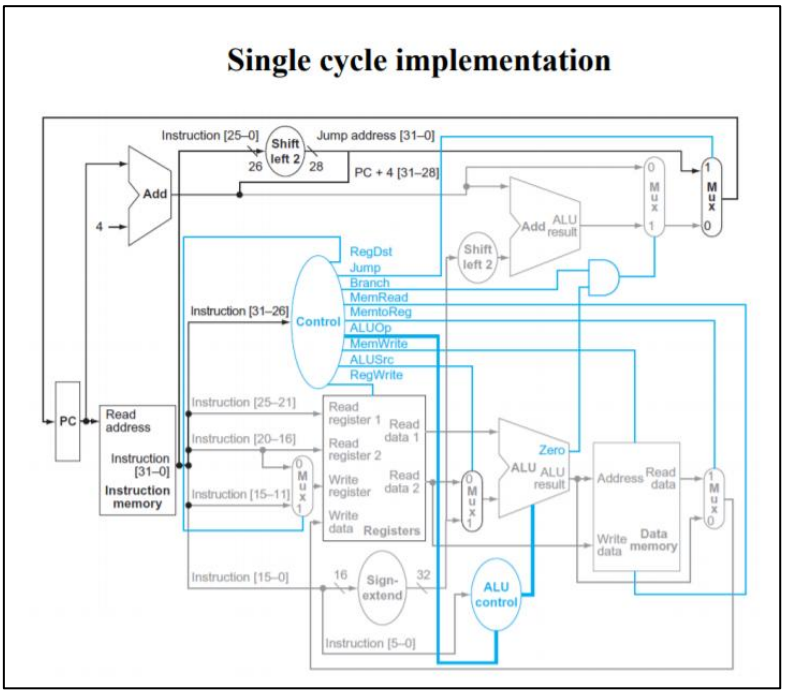
A. Single cycle implementation

1. Introduction

The characteristic of a single-cycle implementation is that each instruction is fully executed before the next instruction begins.

2. Block diagram



3. Example code

```
Loop:
    lw $t1 ,0($t3)
    addi $t5 ,$t2 , 4
    sub $t6 ,$t1 ,$t3
    or $t7 ,$t5 ,$t6
    and $t3,$t1,$t5
    beq $t6 ,$t7,LOOP2
    j LOOP2
Loop2:
    add $t2 ,$t2 ,$t4
    sw $t2 ,20($t3)
    slt $t4,$t0,$t5
    j LOOP
```

4. Main Code

(1) CPU & PC(Main module)

CPU_A is the main module that calls other modules, and the Program Counter (PC) is also defined within this module. The PC address is initially set to 0 (i.e., the first instruction).

When a jump occurs, the jump address is calculated as {PC+4[31:28], address}. The original 26-bit address (a word address) is shifted left by 2 bits to become a 28-bit byte address, resulting in a complete 32-bit jump address.

When a branch occurs (determined based on the Branch and Zero signals), the branch address is calculated as (PC + 4) + offset. The original 16-bit address is first sign-extended to a 32-bit word address, then shifted left by 2 bits to become a 32-bit byte address.

In all other cases, the PC is simply updated to PC + 4.

```
module CPU_A(clk,PC);

input clk;
wire [31:0]ins_out;
wire RegDst ,Jump, ALUsrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch;
wire [1:0] ALUop;
wire [4:0]write_reg;
wire [31:0]write_data, read_data1, read_data2, address_out, alu_mux_out;
wire [31:0]ALU_out;
wire zero;
wire [3:0]ALU_ctrl;
wire [31:0]DataMem_read_data;
wire [31:0] t0,t1,t2,t3,t4,t5,t6,t7;

output reg [31:0] PC;

initial
	begin
		PC <= 0;
	end

always@(posedge clk)
	begin
		if(Jump)
			begin
				PC <= PC + 4;
				PC <= { PC [31:28] , ins_out [25:0] , 2'b00 } ;
			end
		else
			if(Branch && zero)
				begin
					PC <= PC + 4;
```

```
                        PC <= PC +    (address_out << 2);
                    end
                else
                    PC <= PC + 4;
        end


ins_memory U2(.clk(clk), .read_address(PC), .instruction(ins_out));

control U3(.signal(ins_out[31:26]), .RegDst(RegDst), .Jump(Jump), .ALUsrc(ALUsrc),
.MemtoReg(MemtoReg), .RegWrite(RegWrite), .MemRead(MemRead), .MemWrite(MemWrite), .Branc
h(Branch), .ALUop(ALUop));

register_file U4(.clk(clk) , .RegWrite(RegWrite), .read_reg1(ins_out[25:21]),
                        .read_reg2(ins_out[20:16]), .write_reg(write_reg), .write_data(write_data),
                        .read_data1(read_data1), .read_data2(read_data2),
                        .t0(t0),.t1(t1),.t2(t2),.t3(t3),.t4(t4),.t5(t5),.t6(t6),.t7(t7)
                        );

write_register_mux U5
(.RegDst(RegDst), .rd(ins_out[15:11]), .rt(ins_out[20:16]), .write_reg(write_reg));

sign_extend U6(.address_in(ins_out[15:0]), .address_out(address_out));

reg2alu_mux U7(.alusrc(ALUsrc), .reg2(read_data2), .extended(address_out), .muxo(alu_mux_out));

alu U8(.alu_ctr(ALU_ctrl), .a(read_data1), .b(alu_mux_out), .alu_out(ALU_out), .zero(zero));

alu_control U9(.func(ins_out[5:0]), .aluop(ALUop), .alu_Ctr(ALU_ctrl));

data_memoryU10(.clk(clk), .MemWrite(MemWrite), .MemRead(MemRead), .address(ALU_out), .write
_data(read_data2), .read_data(DataMem_read_data)
                        );
write_data_muxU11(.MemtoReg(MemtoReg), .read_data(DataMem_read_data), .ALU_result(ALU_out
), .write_data(write_data));

endmodule
```

(2) Instruction memory

First, the example code is written into the instruction memory. Then, the instruction to be executed is obtained through the value of read_address(PC).

Since the PC uses a byte address while the instruction memory is indexed by instruction number, the read_address must be divided by 4 to retrieve the correct instruction. ◦

```
module ins_memory(clk,read_address,instruction);

input clk;
input [31:0] read_address;
output reg [31:0] instruction;

reg [31:0] instruction_memoroy [31:0];

initial
    begin
        // Loop
        instruction_memoroy[0] = 32'b100011_01011_01001_0000000000000000;
// lw $t1 ,0($t3)
        instruction_memoroy[1] = 32'b001000_01010_01101_0000000000000100;
// addi $t5 ,$t2 , 4
        instruction_memoroy[2] = 32'b000000_01001_01011_01110_00000_100010;
// sub $t6 ,$t1 ,$t3
        instruction_memoroy[3] = 32'b000000_01101_01110_01111_00000_100101;
// or $t7 ,$t5 ,$t6
        instruction_memoroy[4] = 32'b000000_01001_01101_01011_00000_100100;
// and $t3,$t1,$t5
        instruction_memoroy[5] = 32'b000100_01110_01111_0000000000000001;
// beq $t6 ,$t7,LOOP2
        instruction_memoroy[6] = 32'b000010_00000000000000000000000111;
// j LOOP2
        // Loop2
        instruction_memoroy[7] = 32'b000000_01010_01100_01010_00000_100000;
// add $t2 ,$t2 ,$t4
        instruction_memoroy[8] = 32'b101011_01011_01010_0000000000010100;
// sw $t2 ,20($t3)
        instruction_memoroy[9] = 32'b000000_01000_01101_01100_00000_101010;
// slt $t4,$t0,$t5
        instruction_memoroy[10] = 32'b000010_00000000000000000000000000;
// j LOOP
    end

always@(*)
```

```
    begin
        instruction <= instruction_memoroy[read_address/4];
    end

endmodule
```

(3) Register file & WriteReg_MUX & WriteData_MUX

- Register file

  First, initialize the variables that will be used (i.e., registers $t0 to $t7). Then, based on read_reg1 and read_reg2, the corresponding values read_data1 and read_data2 are retrieved.

  The write_reg is determined by the result of the write_reg_mux, and the write_data obtained from the write_data_mux is written into the specified register.

```
module
register_file(clk,RegWrite,read_reg1,read_reg2,write_reg,write_data,read_data1,read_data2,t0,t1,t2,t3,t4,t5,t6,t7);

input clk;

input RegWrite; // Control

input [4:0] read_reg1;
input [4:0] read_reg2;
input [4:0] write_reg;

input [31:0] write_data;
output reg [31:0] read_data1;
output reg [31:0] read_data2;

output reg [31:0] t0,t1,t2,t3,t4,t5,t6,t7;

initial
    begin
        t0 = 0;
        t1 = 1;
        t2 = 2;
        t3 = 3;
        t4 = 4;
        t5 = 5;
        t6 = 6;
        t7 = 7;
    end
```

```
always@(*)
	begin
		case(read_reg1)
		8 : read_data1 = t0;
		9 : read_data1 = t1;
		10 : read_data1 = t2;
		11 : read_data1 = t3;
		12 : read_data1 = t4;
		13 : read_data1 = t5;
		14 : read_data1 = t6;
		15 : read_data1 = t7;
		default : read_data1 = 32'dX;
		endcase
	end

always@(*)
	begin
		case(read_reg2)
		8 : read_data2 = t0;
		9 : read_data2 = t1;
		10 : read_data2 = t2;
		11 : read_data2 = t3;
		12 : read_data2 = t4;
		13 : read_data2 = t5;
		14 : read_data2 = t6;
		15 : read_data2 = t7;
		default : read_data2 = 32'dX;
		endcase
	end

always@(posedge clk)
	begin
		if(RegWrite)
			begin
				case(write_reg)
					8 : t0 <= write_data;
					9 : t1 <= write_data;
					10 : t2 <= write_data;
					11 : t3 <= write_data;
					12 : t4 <= write_data;
					13 : t5 <= write_data;
					14 : t6 <= write_data;
```

```
                        15 : t7 <= write_data;
                endcase
            end
        else
            begin
                t0 <= t0;
                t1 <= t1;
                t2 <= t2;
                t3 <= t3;
                t4 <= t4;
                t5 <= t5;
                t6 <= t6;
                t7 <= t7;
            end
    end

endmodule
```

- WriteReg_MUX

  The value of WriteReg is determined based on the control signal RegDst.

```
module write_register_mux(RegDst,rd,rt,write_reg);

input RegDst;
input [4:0] rd; // R-type destination
input [4:0] rt;

output reg [4:0] write_reg;

always@(*)
    begin
        case(RegDst)
            0 : write_reg = rt;
            1 : write_reg = rd; // R-type
            default : write_reg = 5'dX;
        endcase
    end

endmodule
```

- WriteData_MUX

  The value of write_data is determined based on the control signal MemtoReg.

```
module write_data_mux(MemtoReg,read_data,ALU_result,write_data);

input MemtoReg;

input [31:0] read_data; // lw
input [31:0] ALU_result; // R-format

output reg [31:0] write_data;

always@(*)
    begin
        case(MemtoReg)
            0 : write_data = ALU_result; // R-format
            1 : write_data = read_data; // lw
            default : write_data = 32'dX;
        endcase
    end

endmodule
```

(4) Control

Based on the opcode, the values of the various control signals in the Control unit are output.

```
module control(signal,
RegDst,Jump,ALUsrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUop);

input [5:0] signal;
output reg RegDst,ALUsrc,Jump,MemtoReg,RegWrite,MemRead,MemWrite,Branch;
output reg[1:0] ALUop;

always@(signal)
    begin
        case(signal)
            6'b000000:begin //R format
                        RegDst=1'b1;
                        ALUsrc=1'b0;
                        Jump=1'b0;
                        MemtoReg=1'b0;
                        RegWrite=1'b1;
```

```
                MemRead=1'b0;
                MemWrite=1'b0;
                Branch=1'b0;
                ALUop=2'b10;
            end
        6'b100011:begin // lw
                RegDst=1'b0;
                ALUsrc=1'b1;
                Jump=1'b0;
                MemtoReg=1'b1;
                RegWrite=1'b1;
                MemRead=1'b1;
                MemWrite=1'b0;
                Branch=1'b0;
                ALUop=2'b00;
            end
        6'b101011:begin // sw
                RegDst=1'bX;
                ALUsrc=1'b1;
                Jump=1'b0;
                MemtoReg=1'bX;
                RegWrite=1'b0;
                MemRead=1'b0;
                MemWrite=1'b1;
                Branch=1'b0;
                ALUop=2'b00;
            end
        6'b000100:begin // beq
                RegDst=1'bX;
                ALUsrc=1'b0;
                Jump=1'b0;
                MemtoReg=1'bX;
                RegWrite=1'b0;
                MemRead=1'b0;
                MemWrite=1'b0;
                Branch=1'b1;
                ALUop=2'b01;
            end
        6'b001000:begin // addi
                RegDst=1'b0;
                ALUsrc=1'b1;
                Jump=1'b0;
```

```
                    MemtoReg=1'b0;
                    RegWrite=1'b1;
                    MemRead=1'b0;
                    MemWrite=1'b0;
                    Branch=1'b0;
                    ALUop=2'b00;
                end
            6'b000010:begin // jump
                    RegDst=1'bX;
                    ALUsrc=1'bX;
                    Jump=1'b1;
                    MemtoReg=1'bX;
                    RegWrite=1'b0;
                    MemRead=1'b0;
                    MemWrite=1'b0;
                    Branch=1'b0;
                    ALUop=2'bXX;
                end
            default: begin
                    RegDst=1'bX;
                    ALUsrc=1'bX;
                    MemtoReg=1'bX;
                    RegWrite=1'bX;
                    MemRead=1'bX;
                    MemWrite=1'bX;
                    Branch=1'bX;
                    ALUop=2'bXX;
                end
        endcase
    end

endmodule
```

(5) Sign-extend

The 16-bit address or immediate value is sign-extended to 32 bits before it can be used in the ALU operation.

```
module sign_extend(address_in,address_out);

input [15:0] address_in;
output [31:0] address_out;

assign address_out = { {16{address_in[15]}} , address_in}; //  正數補 0  負數補 1
```

```
endmodule
```

(6) ALU & ALU_Control & ALU_MUX

- ALU

The ALU_Control signal determines the type of operation that the ALU will perform.

```
module alu( alu_ctr,a,b,alu_out,zero);

    input [31:0] a,b;
    input[3:0] alu_ctr;
    output reg [31:0] alu_out;
    output reg zero;

    always@(a or b or alu_ctr)
        begin
            case (alu_ctr)
                4'b0000: alu_out = a & b;
                4'b0001: alu_out = a | b;
                4'b0010: alu_out = a + b;
                4'b0110: alu_out = a - b;
                4'b0111: alu_out = a < b;
                4'b1100: alu_out = a ~^ b;
                default: alu_out = 32'dX;
            endcase
    end

    always@(alu_out)
        begin
            if (alu_out==32'd0)
                zero=1'b1;
            else
                zero=1'b0;
            end

endmodule
```

- ALU_Control

The ALU_Control signal determines the type of operation that the ALU will perform.

```
module alu_control( func,aluop,alu_Ctr);
input [5:0] func;
input [1:0] aluop;
output reg [3:0] alu_Ctr;
```

```
always@(func or aluop)
	begin
		case (aluop)
			2'b00: alu_Ctr=4'b0010; // lw,sw
			2'b01: alu_Ctr=4'b0110; // beq
			2'b10:
				begin
					if(func==6'b100000) // add
						alu_Ctr=4'b0010;
					else if (func==6'b100010) // sub
						alu_Ctr=4'b0110;
					else if (func==6'b100100) // AND
						alu_Ctr=4'b0000;
					else if (func==6'b100101) // OR
						alu_Ctr=4'b0001;
					else if (func==6'b101010) // slt
						alu_Ctr=4'b0111;
					else
						alu_Ctr=4'b1111;
				end
			default: alu_Ctr=4'b1111;
		endcase
	end

endmodule
```

- ALU_MUX

  The value of ALUSrc determines the input to the ALU.

```
module reg2alu_mux(alusrc,reg2,extended,muxo);

input alusrc;
input [31:0]reg2,extended;
output reg [31:0] muxo;

always@(alusrc or reg2 or extended)
	begin
		if(alusrc)
			muxo = extended;
		else
			muxo = reg2;
```

```
    end

endmodule
```

(7) Data memory

First, set up a data_memory array containing 32 entries, each of which is 32 bits wide, and assign initial values to each data entry. Then, based on the result of the ALU operation and the values of MemWrite and MemRead, determine whether to read data from memory and write it to the register, or write data from the register to memory.

```
module data_memory(clk,MemWrite,MemRead,address,write_data,read_data);

input clk;

input MemWrite; // Control for sw
input MemRead; // Control for lw

input [31:0] address; //    rt address
input [31:0] write_data; // sw
output reg [31:0] read_data; // lw

reg [31:0] memory_data [31:0]; // 32 個 register 的 32-bit data

initial
    begin
        memory_data[0] = 0;
        memory_data[1] = 1;
        memory_data[2] = 2;
        memory_data[3] = 3;
        memory_data[4] = 4;
        memory_data[5] = 5;
        memory_data[6] = 6;
        memory_data[7] = 7;
        memory_data[8] = 0;
        memory_data[9] = 0;
        memory_data[10] = 0;
        memory_data[11] = 0;
        memory_data[12] = 0;
        memory_data[13] = 0;
        memory_data[14] = 0;
        memory_data[15] = 0;
        memory_data[16] = 0;
        memory_data[17] = 0;
        memory_data[18] = 0;
```

```
            memory_data[19] = 0;
            memory_data[20] = 0;
            memory_data[21] = 0;
            memory_data[22] = 0;
            memory_data[23] = 0;
            memory_data[24] = 0;
            memory_data[25] = 0;
            memory_data[26] = 0;
            memory_data[27] = 0;
            memory_data[28] = 0;
            memory_data[29] = 0;
            memory_data[30] = 0;
            memory_data[31] = 0;
      end

always@(negedge clk)
      begin
            if(MemWrite && ! MemRead) // sw
                  memory_data[address] <= write_data;
            else
                  memory_data[address] <= memory_data[address];
      end

always@(*)
      begin
            if(MemRead && !MemWrite) // lw
                  read_data = memory_data[address];
            else
                  read_data = 32'dX;
      end

endmodule
```

5. Testbench

```
`timescale 1ns / 1ps

module CPU_A_tb;

reg clk;
wire [31:0] PC;

CPU_A test( .clk(clk) , .PC(PC));
```

```
initial
	begin
		clk = 1'b0;
		#20;
		forever #20 clk = ~clk;
	end

initial
	begin
		#880; $finish;
	end

endmodule
```

6. Waveform

- Changes in Control, registers ($t0-$t7), and Data memory