

Single Cycle 32-bit RISC-like Processor Using Verilog HDL

P Manoj Kumar (21IE10027)

Self Project : Autumn - 2023 : RISC32

1 32-bit RISC-like Processor Specification

Registers	32 32-bit general-purpose registers R0..R15, organized as a register bank with two read ports and one write port.
Special Register	R0 is a special read-only register that is assumed to contain the value of 0.
Memory	Memory is byte addressable with a 32-bit memory address. All operations are on 32-bit data, and all loads and stores occur from memory addresses that are multiples of 4.
Program Counter	A 32-bit program counter (PC).
Addressing Modes	Register addressing, Immediate addressing, Base addressing for accessing memory (with any of the registers used as base register), PC relative addressing for branch

2 Instruction Set

Type	Instruction	Format	Example
Arithmetic & Logic	ADD	R1, R2, R3	$R1 = R2 + R3$
	SUB	R1, R2, R3	$R1 = R2 - R3$
	AND	R1, R2, R3	$R1 = R2 \& R3$
	OR	R1, R2, R3	$R1 = R2 \mid R3$
	XOR	R1, R2, R3	$R1 = R2 \wedge R3$
	NOT	R1, R2	$R1 = \sim R2$
	SLA	R1, R2, R3	$R1 = R2 \ll R3[0]$
	SRA	R1, R2, R3	$R1 = R2 \gg R3[0]$
	SRL	R1, R2, R3	$R1 = R2 \ggg R3[0]$
Immediate Addressing	ADDI	R1, #25	$R1 = R1 + 25$
	SUBI	R1, #-1	$R1 = R1 - 1$
	SLAI	R1, #1	$R1 = R1 \ll 1$
Load & Store	LD	R1, 10(R2)	$R1 = \text{Mem}[R2 + 10]$
	ST	R1, -2(R3)	$\text{Mem}[R3 - 2] = R1$
Branch	BR	#10	$PC = PC + 10$
	BMI	R1, #-10	$PC = PC - 10 \text{ if } (R1 < 0)$
	BPL	R1, #30	$PC = PC + 30 \text{ if } (R1 > 0)$
	BZ	R1, #-75	$PC = PC - 75 \text{ if } (R1 = 0)$
Register Transfer	MOVE	R1, R2	$R1 = R2$
Program Control	HALT		Halts, waits for interrupt on input pin "INT"
	NOP		No operation

3 Register Bank

Register	Code	Register	Code	Register	Code
R0	00000	R11	01011	R22	10110
R1	00001	R12	01100	R23	10111
R2	00010	R13	01101	R24	11000
R3	00011	R14	01110	R25	11001
R4	00100	R15	01111	R26	11010
R5	00101	R16	10000	R27	11011
R6	00110	R17	10001	R28	11100
R7	00111	R18	10010	R29	11101
R8	01000	R19	10011	R30	11110
R9	01001	R20	10100	R31	11111
R10	01010	R21	10101		

4 ALU Module Description

Input/Output	Description
operand1 (32 bits)	First operand
operand2 (32 bits)	Second operand
mode (4 bits)	Mode selector for the operation
en (1 bit)	Enable signal
out (32 bits)	Result of the operation

Mode	Operation
0000	Addition of operand1 and operand2
0001	Subtraction of operand2 from operand1
0010	Bitwise AND of operand1 and operand2
0011	Bitwise OR of operand1 and operand2
0100	Bitwise XOR of operand1 and operand2
0101	Bitwise negation of operand1
0110	Arithmetic left shift of operand1 by operand2 positions
0111	Logical left shift of operand1 by operand2 positions
1000	Arithmetic right shift of operand1 by operand2 positions
1001	Logical right shift of operand1 by operand2 positions
1010	Addition of operand1 and operand2 with operand2 zero-extended to 32 bits

The operation is performed when the **en** signal is asserted.

5 Instructions and Function Op-code

5.1 Arithmetic and Logical Instructions

Instructions	6-bit opcode	Function
ADD	00 0000	000001
SUB	00 0000	000010
AND	00 0000	000011
OR	00 0000	000100
XOR	00 0000	000101
NOT	00 0000	000110
SLA	00 0000	000111
SLL	00 0000	001000
SRA	00 0000	001001
SRL	00 0000	001010
ADDI	01 0000	-
SUBI	01 0001	-
ANDI	01 0010	-
ORI	01 0011	-
XORI	01 0100	-
NOTI	01 0101	-
SLAI	01 0110	-
SLLI	01 0111	-
SRAI	01 1000	-
SRLI	01 1001	-
MOVE	01 1010	-
LD	10 0001	-
ST	10 0010	-
BR	11 0000	-
BMI	11 0001	-
BPL	11 0010	-
BZ	11 0011	-

5.2 Special Type of Instructions

NOP: 11111111111111111111111111111110

HLT: 11111111111111111111111111111111

6 Instruction Formats

6.1 R-Type Instructions

Field	Bits	Description
Opcode	31-26	00 0000
RS	25-21	Source Register 1
RT	20-16	Source Register 2
RD	15-11	Destination Register
Shamt	10-6	Shift Amount
Func	5-0	Function

6.1.1 Examples

Instruction	Binary Representation
ADD R1, R2, R3	000000 00010 00011 00001 00000 000001
SLA R5, R5, R7	000000 00101 00111 00101 00000 001001

6.2 I-Type Instructions

Field	Bits	Description
Opcode	31-26	opcode for different instructions
RS	25-21	Source Register
RT	20-16	Destination Register
Immediate Data	15-0	16-bit Imm. Data

6.2.1 Examples

Instruction	Binary Representation
ADDI R2, R3, #25	010000 00011 00010 00000000000011001
SLAI R5, R5, #01	010110 01001 01001 00000000000000001
MOVE R10, R5	011010 00101 01010 00000000000000000
LD R2, 10(R6)	100001 00110 00010 00000000000001010
ST R2, -2(R11)	100010 01011 00010 11111111111111110
BMI R5, #-10	110001 00101 00101 11111111111101110
BPL R5, #30	110010 00101 00101 00000000000111110
BZ R8, #-75	110011 01000 01000 11111111011010101

6.3 J-Type Instructions

Field	Bits	Description
Opcode	31-26	opcode for different instructions
Immediate Data	25-0	26-bit Imm. Data

6.3.1 Examples

Instruction	Binary Representation
BR #10	110000 000000000000000000000001010

8.1 Control Signals

IF Stage

Control Signal	Description
LoadPC	Load Program Counter
LoadNPC	Load Next Program Counter
ReadIM	Read Instruction Memory
LoadIR	Load Instruction Register

ID Stage

Control Signal	Description
LoadA	Load Register A
LoadB	Load Register B
LoadIMM	Load Immediate Value

EX Stage

Control Signal	Description
ALUsel1	Select ALU Input 1
ALUsel2	Select ALU Input 2
ALUfunc	ALU Function
LoadALUout	Load ALU Output
Condition	Condition for Branching

MEM Stage

Control Signal	Description
ReadDM	Read Data Memory
WriteDM	Write Data Memory
LoadLMD	Load Data Memory Output

WB Stage

Control Signal	Description
MuxWB	Select Write-Back Source
WriteReg	Write to Register
MuxDest	Select Destination Register
MuxBranch	Mux for Branch Instructions

8.2 Main Control Signals According to Function

Instr	ALUsel1	ALUsel2	ALUfunc	Cond	WriteReg	MuxDest	WriteDM	MuxWB	MuxBr
ADD	1	0	0000	11	1	0	0	1	0
SUB	1	0	0001	11	1	0	0	1	0
AND	1	0	0010	11	1	0	0	1	0
OR	1	0	0011	11	1	0	0	1	0
XOR	1	0	0100	11	1	0	0	1	0
NOT	1	0	0101	11	1	0	0	1	0
SLA	1	0	0110	11	1	0	0	1	0
SLL	1	0	0111	11	1	0	0	1	0
SRA	1	0	1000	11	1	0	0	1	0
SRL	1	0	1001	11	1	0	0	1	0
ADDI	1	1	0000	11	1	1	0	1	0
SUBI	1	1	0001	11	1	1	0	1	0
ANDI	1	1	0010	11	1	1	0	1	0
ORI	1	1	0011	11	1	1	0	1	0
XORI	1	1	0100	11	1	1	0	1	0
NOTI	1	1	0101	11	1	1	0	1	0
SLAI	1	1	0110	11	1	1	0	1	0
SLLI	1	1	0111	11	1	1	0	1	0
SRAI	1	1	1000	11	1	1	0	1	0
SRLI	1	1	1001	11	1	1	0	1	0
MOVE	1	1	0000	11	1	1	0	1	0
LD	1	1	0000	11	1	1	0	0	0
ST	1	1	0000	11	0	0	1	1	0
BR	0	0	0000	00	0	0	0	0	1
BMI	0	1	1010	01	0	0	0	0	0
BPL	0	1	1010	00	0	0	0	0	0
BZ	0	1	1010	10	0	0	0	0	0

8.3 Steps for Executing Different Instructions

ADD R1, R2, R3

Stage	Control Signal	Value
IF Stage		
	ReadIM	1
	LoadIR	1
	LoadNPC	1
ID Stage		
	LoadA	1
	LoadB	1
EX Stage		
	ALUfunc	add
	ALUsel1	1
	ALUsel2	0
	LoadALUout	1
MEM Stage		
	LoadPC	1
WB Stage		
	MuxWB	1
	MuxDest	0
	WriteReg	1

LD R2, 10(R6)

Stage	Control Signal	Value
IF Stage		
	ReadIM	1
	LoadIR	1
	LoadNPC	1
ID Stage		
	LoadA	1
	LoadIMM	1
EX Stage		
	ALUfunc	add
	ALUsel1	1
	ALUsel2	1
	LoadALUout	1
MEM Stage		
	LoadPC	1
	ReadDM	1
	LoadLMD	1
WB Stage		
	MuxWB	0
	MuxDest	1
	WriteReg	1

ADDI R1, R2, #10

Stage	Control Signal	Value
IF Stage		
	ReadIM	1
	LoadIR	1
	LoadNPC	1
ID Stage		
	LoadA	1
	LoadIMM	1
EX Stage		
	ALUfunc	add
	ALUsel1	1
	ALUsel2	1
	LoadALUout	1
MEM Stage		
	LoadPC	1
WB Stage		
	MuxWB	1
	MuxDest	1
	WriteReg	1

ST R2, -2(R11)

Stage	Control Signal	Value
IF Stage		
	ReadIM	1
	LoadIR	1
	LoadNPC	1
ID Stage		
	LoadA	1
	LoadIMM	1
EX Stage		
	ALUfunc	add
	ALUsel1	1
	ALUsel2	1
	LoadALUout	1
MEM Stage		
	LoadPC	1
	WriteDM	1
	LoadLMD	0
WB Stage		
	MuxWB	1
	MuxDest	0
	WriteReg	0

MOVE R10, R5

Stage	Control Signal	Value
IF Stage		
	ReadIM	1
	LoadIR	1
	LoadNPC	1
ID Stage		
	LoadA	1
	LoadIMM	1
EX Stage		
	ALUfunc	add
	ALUsel1	1
	ALUsel2	1
	LoadALUout	1
MEM Stage		
	LoadPC	1
WB Stage		
	MuxWB	1
	MuxDest	1
	WriteReg	1

BMI R5, #-10

Stage	Control Signal	Value
IF Stage		
	ReadIM	1
	LoadIR	1
	LoadNPC	1
ID Stage		
	LoadA	1
	LoadIMM	1
EX Stage		
	ALUfunc	add
	ALUsel1	0
	ALUsel2	1
	LoadALUout	1
	Condition	01
MEM Stage		
	LoadPC	1
WB Stage		
	MuxWB	1
	WriteReg	0

9 Verilog Code: Datapath Module

```

1  'include "mux.v"
2  'include "register.v"
3  'include "register_bank.v"
4  'include "instr_mem.v"
5  'include "data_mem.v"
6  'include "adder.v"
7  'include "ALU.v"
8  'include "signext.v"
9  'include "condition.v"
10
11 module datapath(
12     input wire clk,reset,
13     // Input signals from control unit
14     input wire readim,ldir,ldnpc,
15     input wire ldA,ldB,ldimm,
16     input wire [1:0] opcond,
17     input wire alusel1,alusel2,aluen,ldaluout,
18     input wire [3:0] alufunc,
19     input wire seldest,regwrite,writedmem,readdmem,ldlmd,
20     input wire selwb,branch,ldpc,
21
22     // Output signals to control unit
23     output wire [31:0] irout,aluout,
24     output wire [31:0] writedata

```

```

25 );
26
27 // Define wires used in datapath
28 wire [31:0] pcout, npcout, Aout, Bout, immout, lmdout;
29 wire [31:0] instr, pcplus4, pcbranch, pcnext;
30 wire [31:0] Ain, Bin, imm;
31 wire [31:0] destadd;
32 wire [31:0] aluin1, aluin2, result;
33 wire [31:0] dmemout, memmuxout;
34
35 // IF Stage
36 register pcreg(clk, reset, ldpc, pcnext, pcout); // Program Counter Register
37 instr_mem imem(readim, pcout[7:2], instr); // Instruction Memory
38 register irreg(clk, reset, ldir, instr, irout); // Instruction Register
39 adder addpc(pcout, 32'h00000004, pcplus4); // Adder for PC + 4
40 register npcreg(clk, reset, ldnp, pcplus4, npcout); // Next Program Counter
    Register
41 adder bradd(npcout, {{4{irout[25]}}}, irout[25:0], 2'b00, pcbranch); //
    Branch Address Calculation
42 mux muxbr(memmuxout, pcbranch, branch, pcnext); // Mux for Branching
43
44 // ID Stage
45 mux destmux({27'b0, irout[15:11]}, {27'b0, irout[20:16]}, seldest, destadd); //
    Mux for Destination Address
46 register_bank rbank(clk, regwrite, reset, irout[25:21], irout[20:16], destadd
    [4:0], writedata, Ain, Bin); // Register Bank
47 signext ext(instr[15:0], imm); // Sign Extend
48 register A(clk, reset, ldA, Ain, Aout); // Register A
49 register B(clk, reset, ldB, Bin, Bout); // Register B
50 register immreg(clk, reset, ldimm, imm, immout); // Immediate Register
51
52 // EX Stage
53 condition cond(selmem, opcond, Aout); // Condition for Branching
54 mux muxalu1(npcout, Aout, alusel1, aluin1); // Mux for ALU Input 1
55 mux muxalu2(Bout, immout, alusel2, aluin2); // Mux for ALU Input 2
56 ALU alu(aluin1, aluin2, alufunc, aluen, result); // Arithmetic Logic Unit
57 register aluoutreg(clk, reset, ldaluout, result, aluout); // ALU Output
    Register
58
59 // MEM Stage
60 data_mem dmem(clk, reset, writedmem, readdmem, aluout, Bout, dmemout); // Data
    Memory
61 register lmdreg(clk, reset, ldld, dmemout, lmdout); // Load Memory Data
    Register
62 mux memmux(npcout, aluout, selmem, memmuxout); // Mux for Memory Access
63
64 // WB Stage
65 mux wbmux(lmdout, aluout, selwb, writedata); // Mux for Writeback
66
67 endmodule

```

10 Verilog Code: Controller Module

```
1 module controller (
2     input wire clk,reset,
3     input wire [31:0] irout,
4
5     output wire readim,ldir,ldnpc,
6     output wire ldA,ldB,ldimm,
7     output wire [1:0] opcond,
8     output wire alusel1,alusel2,aluen,ldaluout,
9     output wire [3:0] alufunc,
10    output wire seldest,regwrite,writedmem,readdmem,ldlmd,
11    output wire selwb,branch,ldpc
12 );
13
14 parameter IF = 3'b000, ID = 3'b001, EX = 3'b010, MEM = 3'b011, WB = 3'b100,
15           HLT = 3'b101;
16 reg [2:0] state;
17 reg [23:0] control_signals;
18 assign {readim,ldir,ldnpc,ldA,ldB,ldimm,opcond,alusel1,alusel2,aluen,
19         ldaluoout,
20         alufunc,seldest,regwrite,writedmem,readdmem,ldlmd,selwb,branch,ldpc
21         } = control_signals;
22
23 always @(posedge clk, negedge reset) begin
24
25     if(reset)
26         control_signals <= 24'b0;
27     else
28         begin
29             case (state)
30             IF: begin
31                 #1
32                 if(irout == 32'hffffffff)
33                     begin
34                         control_signals <= 24'b0;
35                         state <= HLT;
36                     end
37                 else
38                     begin
39                         control_signals[23:21] <= 3'b111;
40                         control_signals[20:0] <= 21'b0;
41                         state <= ID;
42                     end
43             end
44             ID: begin
45                 #1
46                 control_signals[20:18] <= 3'b111;
47                 state <= EX;
48             end
49         end
50     end
51 end
```

```

46         EX: begin
47             #1
48             case (irout[31:30])
49                 2'b00 :
50                     begin
51                         control_signals[17:16] <= 2'b11;
52                         control_signals[15:13] <= 3'b101;
53                         case (irout[5:0])
54                             6'b000001 : control_signals[11:8]
55                                 <= 4'b0000; // ADD
56                             6'b000010 : control_signals[11:8]
57                                 <= 4'b0001; // SUB
58                             6'b000011 : control_signals[11:8]
59                                 <= 4'b0010; // AND
60                             6'b000100 : control_signals[11:8]
61                                 <= 4'b0011; // OR
62                             6'b000101 : control_signals[11:8]
63                                 <= 4'b0100; // XOR
64                             6'b000110 : control_signals[11:8]
65                                 <= 4'b0101; // NOT
66                             6'b000111 : control_signals[11:8]
67                                 <= 4'b0110; // SLA
68                             6'b001000 : control_signals[11:8]
69                                 <= 4'b0111; // SLL
70                             6'b001001 : control_signals[11:8]
71                                 <= 4'b1000; // SRA
72                             6'b001010 : control_signals[11:8]
73                                 <= 4'b1001; // SRL
74                         endcase
75                     end
76                 2'b01 :
77                     begin
78                         control_signals[17:16] <= 2'b11;
79                         control_signals[15:13] <= 3'b111;
80                         case (irout[31:26])
81                             6'b010000 : control_signals[11:8]
82                                 <= 4'b0000; // ADDI
83                             6'b010001 : control_signals[11:8]
84                                 <= 4'b0001; // SUBI
85                             6'b010010 : control_signals[11:8]
86                                 <= 4'b0010; // ANDI
87                             6'b010011 : control_signals[11:8]
88                                 <= 4'b0011; // ORI
89                             6'b010100 : control_signals[11:8]
90                                 <= 4'b0100; // XORI
91                             6'b010101 : control_signals[11:8]
92                                 <= 4'b0101; // NOTI
93                             6'b010110 : control_signals[11:8]
94                                 <= 4'b0110; // SLAI

```

```

78         6'b010111 : control_signals[11:8]
79             <= 4'b0111; // SLLI
80         6'b011000 : control_signals[11:8]
81             <= 4'b1000; // SRAI
82         6'b011001 : control_signals[11:8]
83             <= 4'b1001; // SRLI
84         6'b011010 : control_signals[11:8]
85             <= 4'b0000; // MOVE
86     endcase
87 end
88 2'b10 :
89     begin
90         control_signals[17:16] <= 2'b11;
91         control_signals[15:13] <= 3'b111;
92         case (irout[31:26])
93             6'b100001 : control_signals[11:8]
94                 <= 4'b0000; // LD
95             6'b100010 : control_signals[11:8]
96                 <= 4'b0000; // ST
97         endcase
98     end
99 2'b11 :
100     begin
101         control_signals[15:13] <= 3'b011;
102         case (irout[31:26])
103             6'b110000 : {control_signals
104                 [17:16], control_signals[11:8]}
105                 <= {2'b11, 4'b1010}; // BR
106             6'b110001 : {control_signals
107                 [17:16], control_signals[11:8]}
108                 <= {2'b01, 4'b1010}; // BMI
109             6'b110010 : {control_signals
110                 [17:16], control_signals[11:8]}
111                 <= {2'b00, 4'b1010}; // BPL
112             6'b110011 : {control_signals
113                 [17:16], control_signals[11:8]}
114                 <= {2'b10, 4'b1010}; // BZ
115         endcase
116     end
117 endcase
118 #2 control_signals[12] <= 1'b1;
119 state <= MEM;
120 end
121 MEM:begin
122     #1
123     case (irout[31:26])
124         6'b100001 : control_signals[5:3] <= 3'b011; //
125                     LD
126         6'b100010 : control_signals[5:3] <= 3'b100; //
127                     ST

```

```

112         endcase
113         if(irout[31:26] == 6'b110000)
114             control_signals[1:0] <= 2'b11;
115         else
116             control_signals[1:0] <= 2'b01;
117             state <= WB;
118         end
119     WB: begin
120         #1
121         case (irout[31:30])
122             2'b00 : {control_signals[7:6], control_signals
123                     [2]} <= 3'b011;
124             2'b01 : {control_signals[7:6], control_signals
125                     [2]} <= 3'b111;
126             2'b10 :
127                 begin
128                     case (irout[31:26])
129                         6'b100001 : {control_signals[7:6],
130                                     control_signals[2]} <= 3'b110;
131                                     // LD
132                         6'b100010 : {control_signals[7:6],
133                                     control_signals[2]} <= 3'b001;
134                                     // ST
135                     endcase
136                 end
137             2'b11 : {control_signals[7:6], control_signals
138                     [2]} <= 3'b000;
139             default: {control_signals[7:6], control_signals
140                     [2]} <= 3'b000;
141         endcase
142         state <= IF;
143     end
144     HLT: begin
145         #1 control_signals <= 24'b0;
146         state <= HLT;
147     end
148     default: state <= IF;
149 endcase
150 end
151 endmodule

```

11 Verilog Code: Top Module

```
1  'include "datapath.v"
2  'include "controller.v"
3  module mips (
4      input wire clk,reset,
5      output wire [31:0] writedata
6  );
7  wire readim,ldir,ldnpc;
8  wire ldA,ldB,ldimm;
9  wire [1:0] opcond;
10 wire alusel1,alusel2,aluen,ldaluout;
11 wire [3:0] alufunc;
12 wire seldest,regwrite,writedmem,readdmem,ldlmd;
13 wire selwb,branch,ldpc;
14 wire [31:0] irout,aluout;
15
16 datapath dpath(clk,reset,readim,ldir,ldnpc,ldA,ldB,ldimm,opcond,alusel1,
17     alusel2,aluen,ldaluout,alufunc,seldest,regwrite,writedmem,readdmem,ldlmd,
18     selwb,branch,ldpc,irout,aluout,writedata);
19 controller ctrl(clk,reset,irout,readim,ldir,ldnpc,ldA,ldB,ldimm,opcond,
20     alusel1,alusel2,aluen,ldaluout,alufunc,seldest,regwrite,writedmem,
21     readdmem,ldlmd,selwb,branch,ldpc);
22
23 endmodule
```

12 Problem Solving Using the Processor

12.1 Finding GCD of Two Numbers

To find the greatest common divisor (GCD) of two numbers, we'll use registers *R1* and *R2* to hold the first and second numbers, respectively. Additionally, we'll store these values in `data_memory[0]` and `data_memory[1]` respectively. Finally, the calculated GCD will be stored in `data_memory[2]`.

Instruction	Binary Code
ADDI R1, R0, #143;	010000_00000_00001_0000000010001111
ADDI R2, R0, #78;	010000_00000_00010_0000000001001110
ST R1, 0(R0);	100010_00000_00001_0000000000000000
ST R2, 1(R0);	100010_00000_00010_0000000000000001
SUB R3, R1, R2; (W)	000000_00001_00010_00011_00000_000010
BMI R3, #3; // if R1<R2 (X)	110001_00011_00011_0000000000000011
BPL R3, #6; // if R1>R2 (Y)	110010_00011_00011_00000000000000110
BZ R3, #8; // if R1=R2 (Z)	110011_00011_00011_00000000000001000
HLT;	11111111111111111111111111111111
SUB R3, R2, R1; // if R1<R2 (X)	000000_00010_00001_00011_00000_000010
MOVE R2, R3;	011010_00011_00010_0000000000000000
BR #-8; (W)	110000_111111111111111111111111000
HLT;	11111111111111111111111111111111
MOVE R1, R3; // if R1>R2 (Y)	011010_00011_00001_0000000000000000
BR #-11; (W)	110000_1111111111111111111111110101
HLT;	11111111111111111111111111111111
ST R1, 2(R0); // if R1=R2 (Z)	100010_00000_00001_00000000000000010
HLT;	11111111111111111111111111111111

12.2 Sorting a set of 10 integers using bubble sort

The program implements the bubble sort algorithm to sort a set of 10 integers stored in the array `arr`. The array, starting at address 100, contains the elements [20, 50, 10, 30, 70, 40, 60, 80, 100, 90]. In the testbench, the entire array is loaded into Data Memory for sorting.

Instruction	Binary Code
ADDI R1, R0, #100; // arr[0]	010000_00000_00001_00000000001100100
MOVE R2, R0; // i=0	011010_00000_00010_0000000000000000
MOVE R3, R0; // j=0	011010_00000_00011_0000000000000000
ADDI R4, R0, #10; // n=10	010000_00000_00100_00000000000001010
ADDI R5, R0, #10; // n-i for inner loop	010000_00000_00101_00000000000001010
MOVE R6, R1; // for iterating addr by i	011010_00001_00110_0000000000000000
MOVE R7, R1; // for iterating addr by j	011010_00001_00111_0000000000000000
SUBI R4, R4, #1; // decrement n	010001_00100_00100_00000000000000001
MOVE R3, R0; // outer_loop // j=0	011010_00000_00011_0000000000000000
SUBI R5, R5, #1; // decreasing size for inner_loop	010001_00101_00101_00000000000000001
ADD R7, R0, R1; // resetting addr itr j	000000_00000_00111_00000_00000_000010
LD R8, 0(R7); // inner_loop // arr[j]	100001_00111_01000_0000000000000000
ADDI R7, R7, #1; // addr itr j += 1	010000_00111_00111_00000000000000001
LD R9, 0(R7); // arr[j+1]	100001_00111_01001_0000000000000000
ADDI R3, R3, #1; // j++	010000_00011_00011_00000000000000001
SUB R10, R8, R9; // R10 = R8 - R9;	000000_01000_01001_01010_00000_000010
BMI R10, #3; // if R8 < R9 then Branch	110001_01010_01010_00000000000000011
ST R8, 0(R7); // swap	100010_00111_01000_0000000000000000
ST R9, -1(R7);	100010_00111_01001_11111111111111111
LD R9, 0(R7);	100001_00111_01001_0000000000000000
SUB R11, R3, R5;	000000_00011_00101_01011_00000_000010
BZ R11, #1; // Exiting from inner_loop	110011_01011_01011_00000000000000001
BR #-12; // Address to inner_loop	110000_11111111111111111111110100
ADDI R2, R2, #1; // After Exiting From inner_loop	010000_00010_00010_00000000000000001
SUB R12, R2, R4;	000000_00010_00100_01100_00000_000010
BZ R12, #1; // Program Completed	110011_01100_01100_00000000000000001
BR #-19; // outer_loop	110000_111111111111111111111101101
HLT;	111111111111111111111111111111111