

Design of a Single-Cycle Processor

Sri Sai Nomula
SR No: 22986, MTech ESE,

I OBJECTIVE

The Processor is a 32-bit RISC CPU with 32-bit Address space with RISC type instructions. The processor is a single-cycle processor which completes a Instruction in 1 cycle. The processor supports basic arithmetic and logic instructions, lui, branch instructions, and load-store instructions.

II SPECIFICATIONS

- 32-bit Data width and 32-bit Instruction width.
- 32 General Purpose Registers- R0 to R31 (32 bit each).
- Instructions supported- ADD, SUB, AND, OR, LOAD, STORE, BGE, BEQ, LUI, ADDI.

III INSTRUCTION FORMAT

The first 7 bits of the instruction are used to define the instruction, and the rest of bits are used to define the function or immediate values and Register addresses. The table below shows how each instruction is divided.

Instruction	[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
R-Type	Funct7	RS2	RS1	Funct3	RD	OpCode
Load	Imm(11:5)	Imm(4:0)	RS1	000	RD	OpCode
Store	Imm(11:5)	RS2	RS1	000	Imm(4:0)	OpCode
ADDI	Imm(11:5)	Imm(4:0)	RS1	000	RD	OpCode
BEQ	Imm(11:4)	RS2	RS1	000	Imm(3:0 10)	OpCode
BGE	Imm(11:4)	RS2	RS1	000	Imm(3:0 10)	OpCode
LUI	Imm(31:25)	Imm(24:20)	Imm(19:15)	Imm(14:12)	RD	OpCode

Table 1: Instruction Set Encoding

1. **R-Type Instruction:** The OpCode is 0110011 and the Funct3 field describes the operation ALU should perform (000 - ADD or SUB based on Funct7, 010 - AND, 011 - OR , 100 - XOR). RS1, RS2 and RD fields provides the Register Values which are of 5-bits each. Example - SUB R1 R2 R3 [R1 <- R2 - R3]. Here, Funct3 = ADD(000) and Funct7 = SUB(0100000), RS2 = 00011 , RS1 = 00010 and RD = 00001.
2. **Load-Type Instruction:** The OpCode is 0000011 and the Imm field gives the Value which needs to added to RS1 and access that particular memory location. RS1 and RD fields provides the Register Values which are of 5-bits each. Example - LOAD R2 R1 101 [R2 <- mem[R1 + 101]]. Here, Imm = 101, RS1 = 00001 and RD = 00010.
3. **Store-Type Instruction:** The OpCode is 0100011 and the Imm field gives the Value which needs to added to RS1 and store into that particular memory location. RS1 and RD fields provides the Register Values which are of 5-bits each. Example - STORE R2 R1 101 [R2 -> mem[R1 + 101]]. Here, Imm = 101, RS1 = 00001 and RD = 00010.
4. **ADDI-Type Instruction:** The OpCode is 0010011 and the Imm field gives the Value which needs to added to RS1 and store into RD. RS1 and RD fields provides the Register Values which are of 5-bits each. Example - ADDI R2 R1 101 [R2 <- R1 + 101]. Here, Imm = 101, RS1 = 00001 and RD = 00010.
5. **BEQ(Branch on Equal)-Type Instruction:** The OpCode is 1100011 and the Imm field gives the Value where PC needs to points in case condition satisfies. RS1 and RS2 fields provides the Register Values which are of 5-bits each. Example - BEQ R3 R4 1101 [If R3=R4 then Fetch Instruction from Memory Location 1101]. Here, Imm = 1101, RS1 = 00011 and RS2 = 00100.

6. **BGE(Branch on Greater)-Type Instruction:** The OpCode is 1100011 and the Imm field gives the Value where PC needs to points in case condition satisfies. RS1 and RS2 fields provides the Register Values which are of 5-bits each. Example - BGE R3 R4 1101 [If R3 > R4 then Fetch Instruction from Memory Location 1101]. Here, Imm = 1101, RS1 = 00011 and RS2 = 00100.
7. **LUI-Type Instruction:** The OpCode is 0110111 and the Imm field gives the Upper 12-bits Value that needs to be stored in RD. Example - LUI ABCDE R3[R3 = ABCDE000]. Here, Imm = ABCDE, RD = 00011

IV DATA PATH AND CONTROL PATH

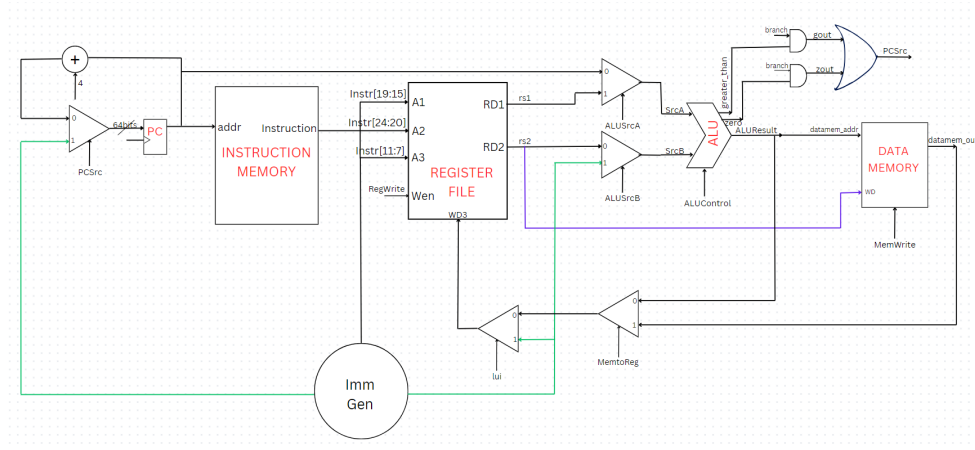


Figure 1: Data Path Block Diagram

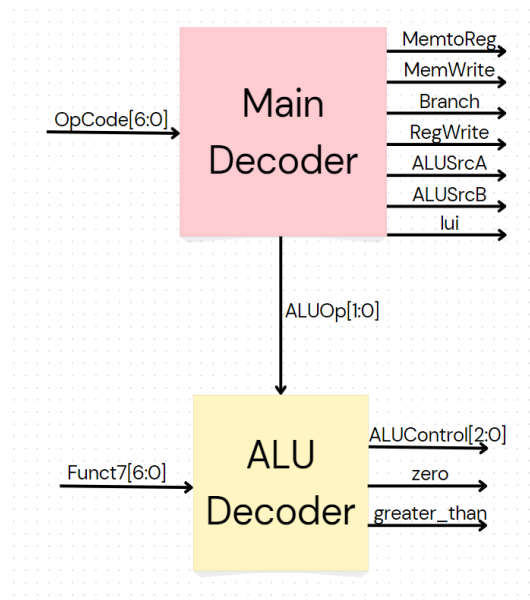


Figure 2: Control Path State Diagram

The Functionality of various control signals are described in the Table Below. The values of these signals change based on the OpCode and the Task they are performing described by the Controller. Most of the control information comes from the opcode, but R-type instructions also use the funct7 and funct3 field to determine the ALU operation. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic, as shown in Figure 2. The main decoder computes most of the outputs from the opcode. It also determines a 2-bit ALUOp signal. The ALU decoder uses this ALUOp signal in conjunction with the funct7 field to compute ALUControl.

Signal	Function
ALUSrcA	To Select Operand-1 to ALU between PC Value and Register Value.
ALUSrcB	To Select Operand-2 to ALU between Register Value and Immediate Value.
ALUControl	To Perform the desired operation of ADD, SUB, AND, OR.
PCSrc	To Select the PCNext Value between PC + 1 and Immediate Address specified.
Z	Used while performing Branch Equal Operation. So, that when Branch is Equal the Zout is set.
G	Used while performing Branch Greater than Operation. So, that when Branch is Greater then Gout is set.
MemtoReg	To Select the data to be written into Register File between ALUResult and Data from Memory.
RegWrite	To Write the data into Register File
MemWrite	To Write the data into Data Memory.

Table 2: Description of Control Signals

Instruction	OpCode	MemtoReg	MemWrite	Branch	RegWrite	ALUSrcA	ALUSrcB	lui
R-Type	0110011	0	0	0	1	1	0	0
LOAD	0000011	1	0	0	1	1	1	0
STORE	0100011	0	1	0	0	1	1	0
ADDI	0010011	0	0	0	1	1	1	0
BEQ	1100011	0	0	1	0	1	0	0
BGE	1100011	0	0	1	0	1	0	0
LUI	0110111	0	0	0	1	X	X	1

Table 3: Main Decoder Truth Table

The control signals for each instruction were described as we built the datapath. Table 3 is a truth table for the main decoder that summarizes the control signals as a function of the opcode.

V ASSEMBLY PROGRAM

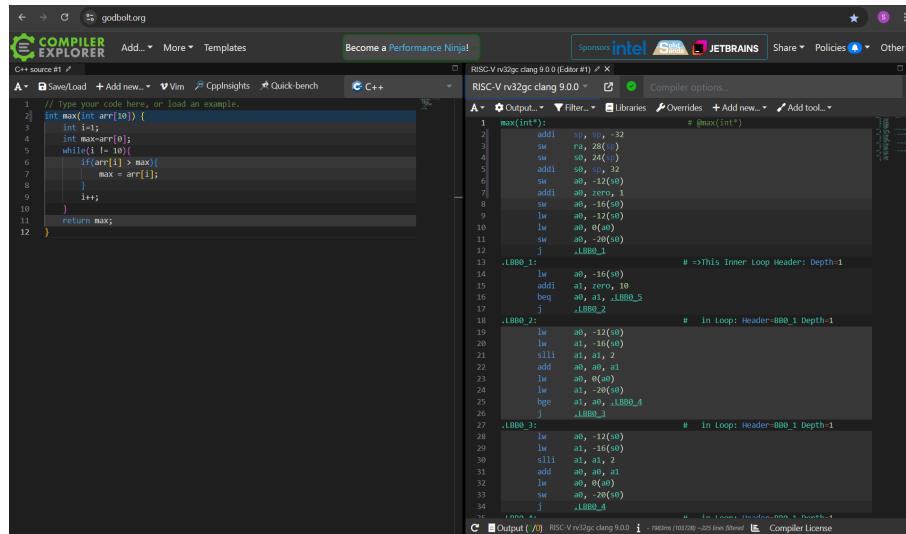
The below assembly program is used to find the maximum number from a set of ten random numbers. This assembly code is written using the instructions declared for this 32-bit RISC-V architecture. The instructions are stored in the first 9 locations of the Instruction Memory and the 10 random numbers are stored in the 10 locations of Data Memory and the final result (Maximum Number) is stored in the R3 of Register File.

Address	Instruction	Representation[HEX Format]
0000	ADDI R1 R0 1	00100093
0001	ADDI R2 R0 10	00A00113
0002	LOAD R3 R0 0	00000183
LOOP : 0003	BEQ R1 R2 STOP	00315763
0004	LOAD R4 R1 0	00008203
0005	ADDI R1 R1 1	00108093
0006	BGE R3 R4 LOOP	0041D363
0007	ADD R3 R4 R0	000201B3
0008	BEQ R1 R2 LOOP	00108363
STOP : 0009	HALT	

Table 4: Assembly Program

VI CONVERTING C CODE TO RISC-V INSTRUCTIONS

- First I used C compiler to write the Code for finding maximum of 10 integers and converted it to RISC-V 32 bit Implementation using Compiler Explorer from godbolt.org as shown in Fig.3



```

1 // Type your code here, or load an example.
2 int max(int arr[10]) {
3     int i;
4     int maxarr[0];
5     while(i != 10){
6         if(arr[i] > max){
7             max = arr[i];
8         }
9         i++;
10    }
11    return max;
12 }

```

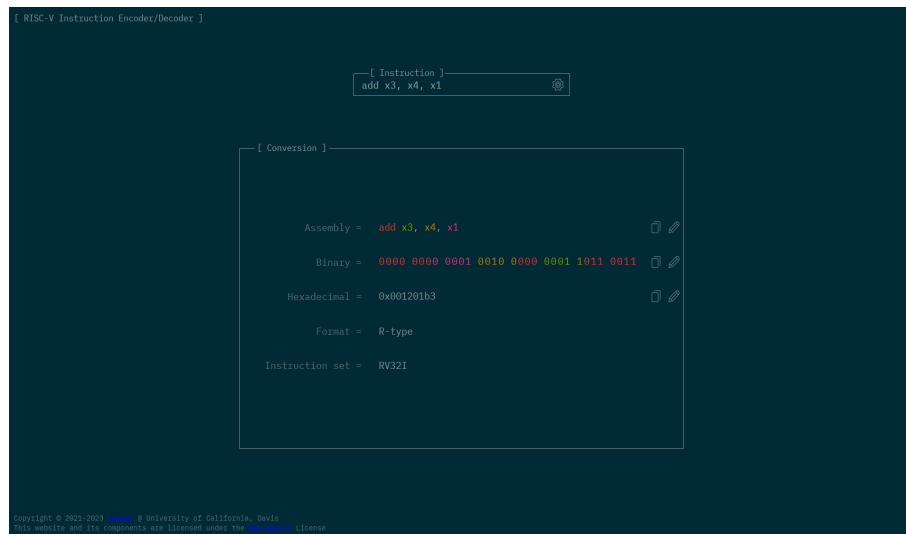
```

1 max(int*):
2     addi    sp, sp, -32
3     sw      ra, 28(sp)
4     sw      zero, 24(sp)
5     addi    a0, sp, 32
6     sw      a0, -12(sp)
7     addi    a0, zero, 1
8     sw      a0, -16(sp)
9     lw      a0, -12(sp)
10    lw      a0, 0(a0)
11    sw      a0, -20(sp)
12    j        .LBB0_1
13 .LBB0_1:
14    lw      a0, -16(sp)
15    addi    a1, zero, 10
16    beq     a0, a1, .LBB0_5
17    j        .LBB0_2
18 .LBB0_2:
19    lw      a0, -12(sp)
20    lw      a1, -16(sp)
21    slliu   a1, a1, 2
22    add     a0, a0, a1
23    lw      a0, 0(a0)
24    lw      a1, -20(sp)
25    bge     a1, a0, .LBB0_4
26    j        .LBB0_3
27 .LBB0_3:
28    lw      a0, -12(sp)
29    lw      a1, -16(sp)
30    slliu   a1, a1, 2
31    add     a0, a0, a1
32    lw      a0, 0(a0)
33    sw      a0, -20(sp)
34    j        .LBB0_4

```

Figure 3: C-Code to Assembly Converter

- Then, using these RISC-V 32 bit Instructions I have converted them to Hex Representation with the help of rvodecjs as shown in Fig.4



```

[ Instruction ]
add x3, x4, x1

```

```

[ Conversion ]

Assembly = add x3, x4, x1
Binary = 0000 0000 0001 0010 0000 0001 1011 0011
Hexadecimal = 0x001201b3
Format = R-type
Instruction set = RV32I

```

Figure 4: RISC-V to Hex Representation

VII TIMING REPORT

The Worst Negative Slack (WNS) is 0.065ns at an operating Time Period of 15ns. Therefore, the Maximum Operating Frequency can be calculated as:

$$f_{max} = \frac{1}{15ns - 0.065ns} = \frac{1}{14.935} = 66.95MHz. \quad (1)$$

The Timing Report is shown in Fig.5

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 0.065 ns	Worst Hold Slack (WHS): 0.302 ns	Worst Pulse Width Slack (WPWS): 3.750 ns	
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns	
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	
Total Number of Endpoints: 11276	Total Number of Endpoints: 11276	Total Number of Endpoints: 1549	
All user specified timing constraints are met.			

Figure 5: Timing Report for Single-Cycle Processor

VIII SIMULATION RESULTS

The Waveform after Simulation is shown in Fig.6

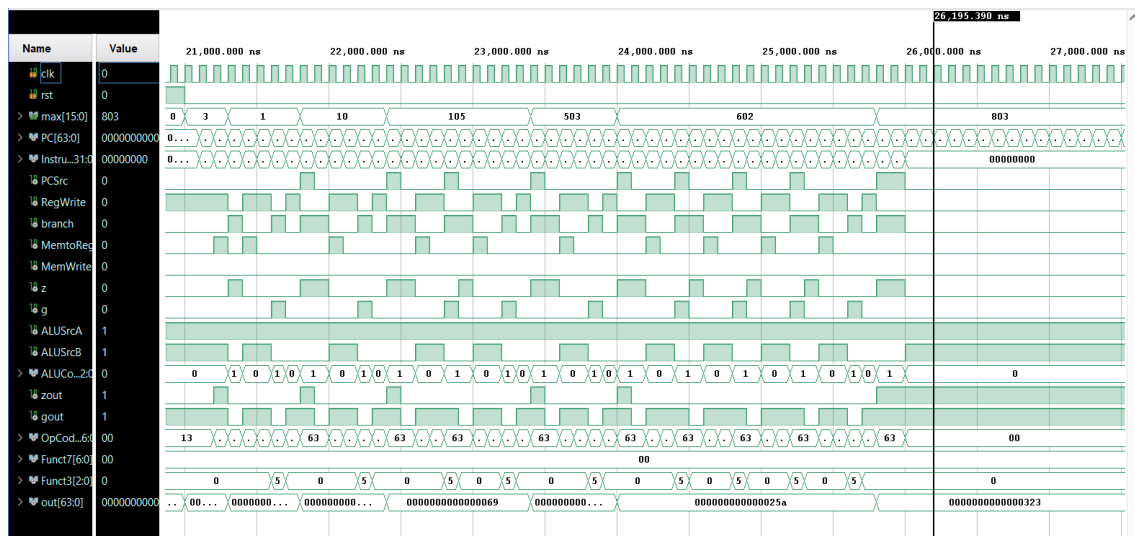


Figure 6: Simulation for Single-Cycle Processor

IX RESOURCE UTILIZATION

The component statistics of the Single-Cycle Processor designed using Verilog Code gets synthesized into the Hardware as shown below in Fig.7 and Fig.8

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Bonded IOB (106)	BUFGCTRL (32)
singlecycle	1575	524	640	331	755	551	1024	18	1
Data_Path (datapath)	1575	524	640	331	755	551	1024	0	0
ALU (ALU)	14	0	0	0	22	14	0	0	0
data_memory (Data_	1090	0	512	256	319	66	1024	0	0
instruction_memory	11	0	0	0	7	11	0	0	0
PCreg (flipflops_64)	1	12	0	0	4	1	0	0	0
reg_file (Register_Fil	468	512	128	75	470	468	0	0	0

Figure 7: Components statistics of Single-Cycle Processor

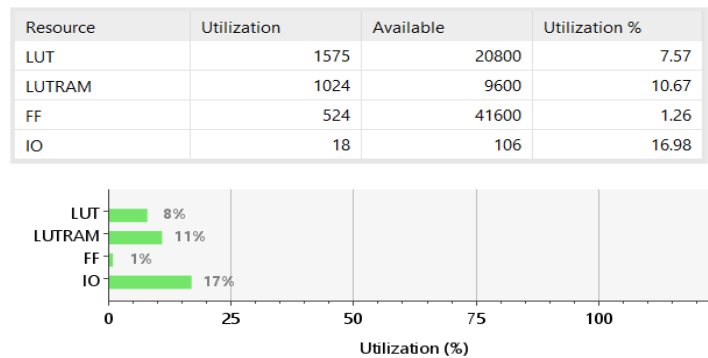


Figure 8: Components Summary of Single-Cycle Processor

X POWER REPORT

The Power Report of the Single Cycle Processor is shown in Fig.9

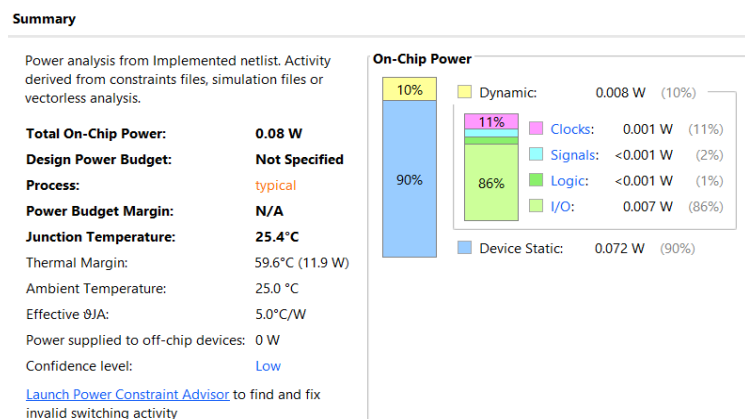


Figure 9: Power Report of Single-Cycle Processor

XI CONCLUSION

The designed 32-bit RISC Single Cycle CPU was then verified using the assembly code for finding the maximum value in a set of 10 random Numbers and output was displayed on Basys3 FPGA Board using its 7-Segment display. Algorithm to find maximum number was written in C and got the required assembly/binary code using Compiler Explorer and then used RVCodec to get the Hex Representation of the RISC-V Instructions.

REFERENCES

- [1] Digital Design and Computer Architecture 2nd edition” - David Mooney Harris and Sarah L Harris
- [2] ”Compiler Explorer to Convert C-Code to RISC-V 32 bit Instructions”
- [3] ”RVCodec to get the Hex representation of RISC-V Instructions”