Computer Architecture

Computer Assignment #2

Seyyed Sina Alizadeh Tabatabai - (ID: 810101477)
Majid Sadeghinezhad Saryazdi - (ID: 810101459)
Prof. : Dr.Safari
Fall 2024

Abstract

A simple single-cycle version of the RISC-V processor with limited available commands implemented using Verilog HDL.

Introduction

In this project, a single-cycle RISC-V processor with limited instruction set is designed. The minute detail of each part is thoroughly discussed in the corresponding section of the report. In order to check the performance of the designed processor, an assembly code for finding the smallest element in an array of ten, 32-bit integers is converted to machine code and then given to the instruction set memory to be executed in the result section

Available Instructions in this implementation:

- R-Type: add, sub, and, or, slt
- I-Type: lw, addi, xori, ori, slti, jalr
- S-Type: sw
- J-Type: jal
- B-Type: beq, bne
- U-Type: lui

The datapath and controller are NOT designed from scratch and are based on the given single-cycle datapath and controller in the cited textbook in the citation section.

The instructions that are not implemented in the textbook version are added by changing the datapath and controller.

Approaching the Problem

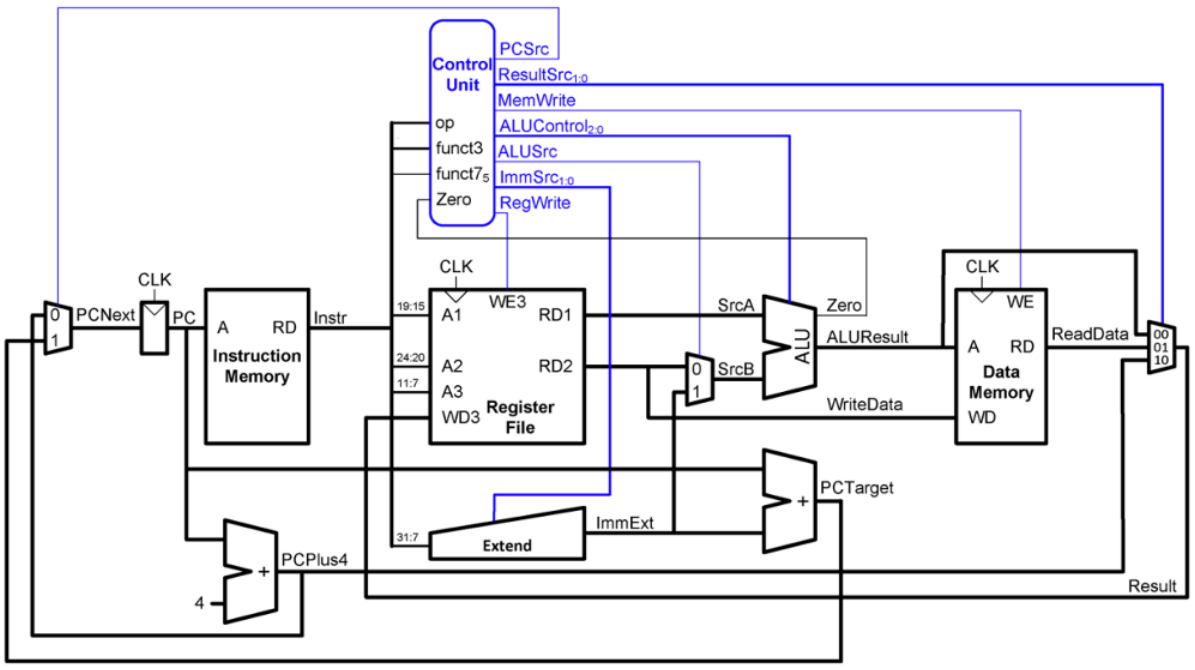
The highlighted instructions are supported by this design and everything was implemented to match the RISC-V documentation instruction formats:

Core Instruction Formats

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ˆ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

The design starts with the base datapath below:



This design lacks the wiring and hardware to support the instructions listed below

- R-Type: sltu
- I-Type: xori, sltiu, jalr
- B-Type: bne
- U-Type: lui

Adding the instructions step-by-step:

- XORI , SLTU : Inorder to add these instructions, since the ALU has two empty instruction slots, no change in the datapath is needed. But the ALU is changed to the format below:

ALU CONTROL [2:0]	ALU RESULT
000	SRC A + SRC B
001	SRC A – SRC B
010	SRC A & SRC B
011	SRC A | SRC B
100	SRC A XOR SRC B
101	SRC A SLT SRC B
110	SRC A SLTU SRC B

- BNE: The changes required for this were previously added to the datapath for adding BEQ instruction, hence, no change is needed.

- JALR: Based on the RISC-V documentation:

jal	Jump And Link	J	1101111			rd = PC+4; PC += imm
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm

A path for handling JAL already exists in the previous datapath (where PC is added with Imm and is put into PC). But a way of handling JALR is to be designed.
There already is a path to add RD1 with Imm, hence, this path is used to put the result in PC.
This requires the signal, PCSrc, to become 2-bits instead of 1-bit to be able to handle the extra input for the multiplexer. These changes are displayed with this color in the next section.

- LUI:

lui	Load Upper Imm	U	0110111			rd = imm << 12

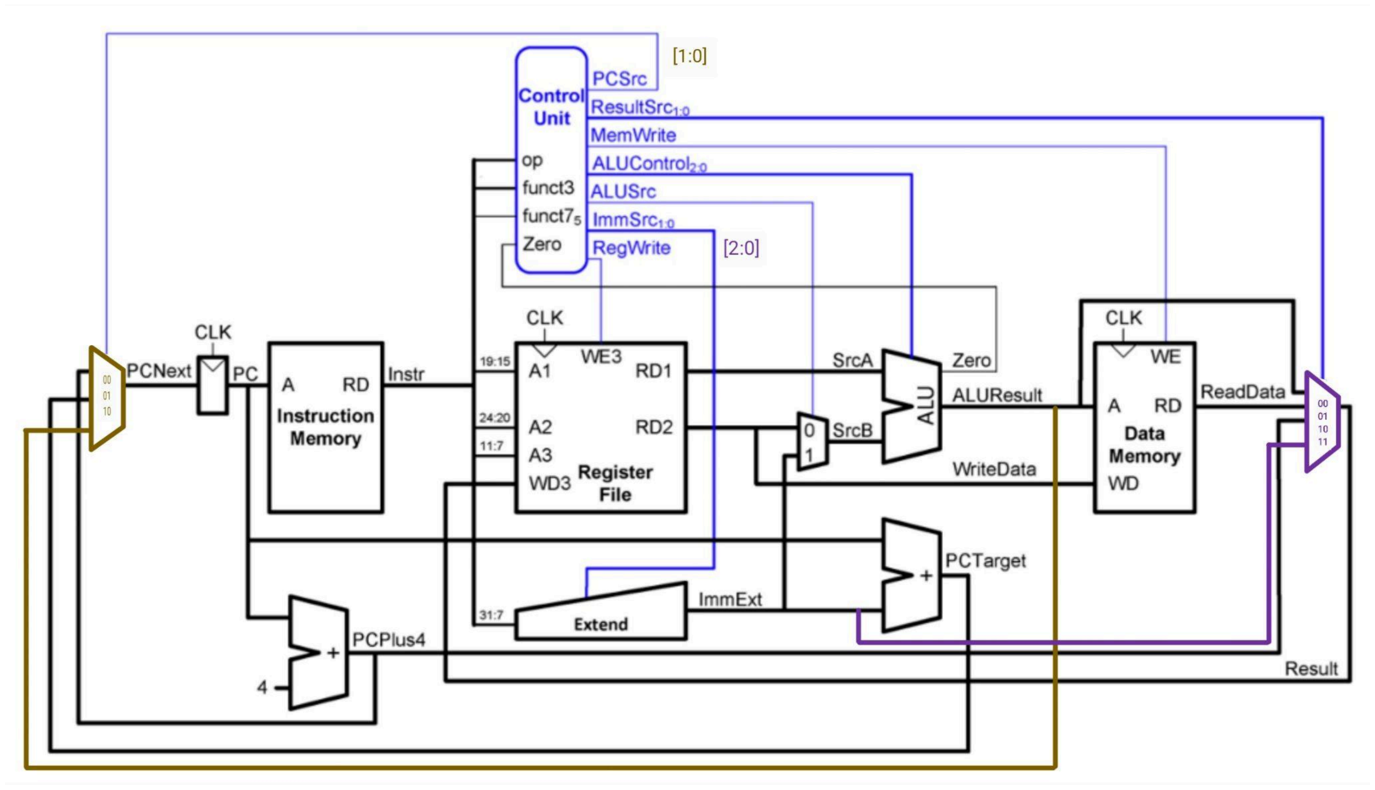
Firstly, a path is to be added to give the output, ImmExt, to WD3. Which is handled by adding a new input to the result multiplexer. Since this is a new instruction type which was not implemented in the datapath before, the Extend block must be improved and 1 bit must be added to the ImmSrc signal (making it a 3-bit signal). The changes are displayed with this color in the next section.
Based on core instruction formats, the extend block is designed as below:

Type	ImmExt	ImmSrc
I	{ 20{ Ins[31] } , Ins[31:20] }	000
S	{ 20{ Ins[31] } , Ins[31:25] , Ins[11:7] }	001
B	{ 19{ Ins[31] } , Ins[31] , Ins[7] , Ins[30:25] , Ins[11:8] , 1'b0 }	010
J	{ 11{ Ins[31] } , Ins[31] , Ins[19:12] , Ins[20] , Ins[30:21] , 1'b0 }	011
U	{ Ins[31:12] , 12'b0 }	100

The controller signals are determined in the next section.

Final Datapath and Controller

- Data Path



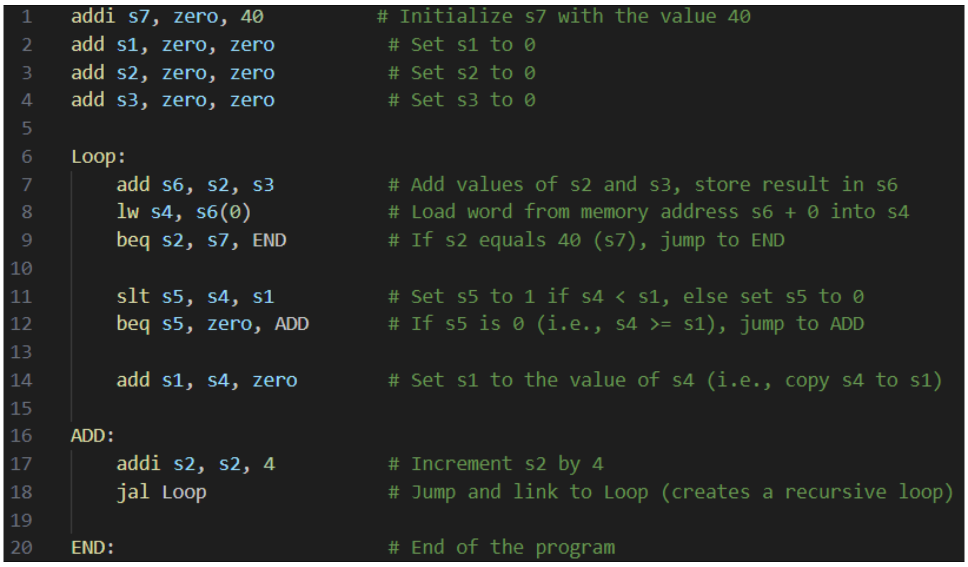
- Controller

For the controller, using the final datapath, and the RISC-V documentation, the control signals are issued as below:

OP	Func3	Func7	Inst	PCSrc	ResultSrc	MemWrite	ALUControl	ALUSrc	ImmSrc	RegWrite
51	0	0	add	00	00	0	000	0	—	1
51	0	32	sub	00	00	0	001	0	—	1
51	6	0	or	00	00	0	011	0	—	1
51	7	0	and	00	00	0	010	0	—	1
51	2	0	slt	00	00	0	101	0	—	1
51	3	0	sltu	00	00	0	110	0	—	1
19	0	-	addi	00	00	0	000	1	000	1
19	4	-	xori	00	00	0	100	1	000	1
19	6	-	ori	00	00	0	011	1	000	1
19	2	-	slti	00	00	0	101	1	000	1
19	3	-	sltiu	00	00	0	110	1	000	1
3	2	-	lw	00	01	0	—	1	000	1
35	2	-	sw	00	–	1	000	1	001	0
99	0	-	Beq	Zero?01:00	–	0	001	0	010	0
99	1	-	Bne	Zero?00:01	–	0	001	0	010	0
111	-	-	Jal	01	10	0	—	-	011	1
103	0	-	Jalr	10	10	0	000	1	000	1
55	-	-	lui	00	11	0	—	-	100	1

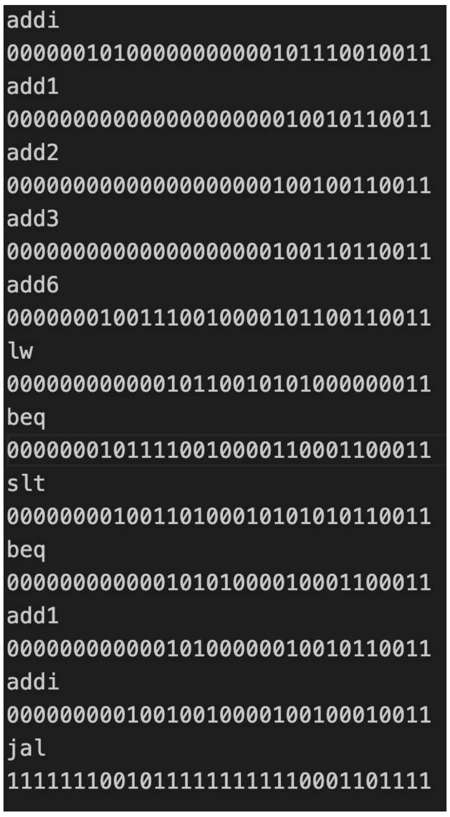
Results (Assembly Code Check)

In this section, the goal is to check the design's validity. The assembly code written below finds the smallest element in a 10, 32-bit element array.



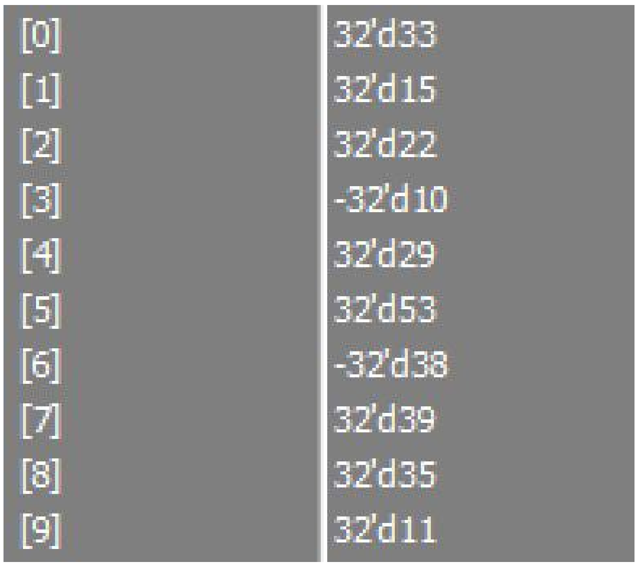
```
addi s7, zero, 40           # Initialize s7 with the value 40
add s1, zero, zero           # Set s1 to 0
add s2, zero, zero           # Set s2 to 0
add s3, zero, zero           # Set s3 to 0

Loop:
    add s6, s2, s3           # Add values of s2 and s3, store result in s6
    lw s4, s6(0)             # Load word from memory address s6 + 0 into s4
    beq s2, s7, END          # If s2 equals 40 (s7), jump to END

    slt s5, s4, s1           # Set s5 to 1 if s4 < s1, else set s5 to 0
    beq s5, zero, ADD        # If s5 is 0 (i.e., s4 >= s1), jump to ADD

    add s1, s4, zero         # Set s1 to the value of s4 (i.e., copy s4 to s1)

ADD:
    addi s2, s2, 4           # Increment s2 by 4
    jal Loop                 # Jump and link to Loop (creates a recursive loop)

END:                         # End of the program
```

This code must be first changed to machine code in order to be tested.
Conversion to machine code:

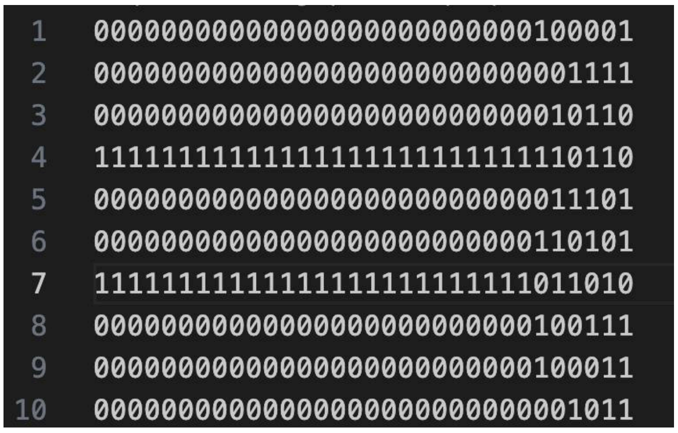


```
addi
00000010100000000000101110010011
add1
00000000000000000000100101110011
add2
00000000000000000000100100110011
add3
00000000000000000000100110110011
add6
00000001001110010000101100110011
lw
00000000000010110010101000000011
beq
00000001011110010000110001100011
slt
00000000100110100010101010110011
beq
00000000000010101000010001100011
add1
00000000000010100000100101110011
addi
00000000100100100000100100010011
jal
11111110010111111111110001101111
```

The values of the array's elements in decimal and binary:

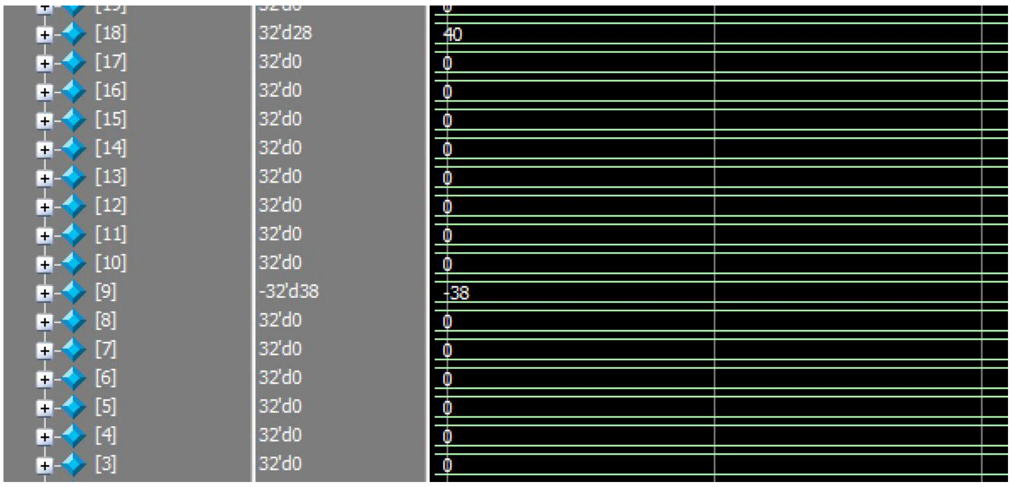


[0]	32'd33
[1]	32'd15
[2]	32'd22
[3]	-32'd10
[4]	32'd29
[5]	32'd53
[6]	-32'd38
[7]	32'd39
[8]	32'd35
[9]	32'd11



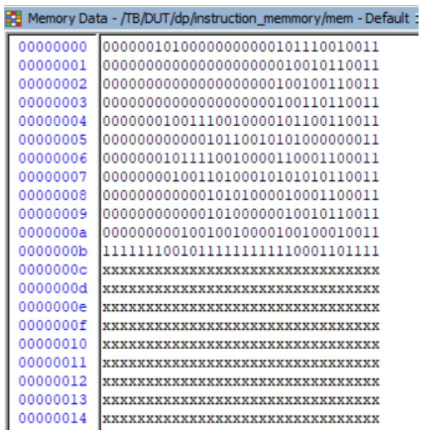
```
00000000000000000000000000100001
00000000000000000000000000001111
00000000000000000000000000010110
11111111111111111111111111110110
00000000000000000000000000011101
00000000000000000000000000110101
11111111111111111111111111011010
00000000000000000000000000100111
00000000000000000000000000100011
00000000000000000000000000001011
```

The output of the testbench, given the instructions and data provided in preceding paragraphs:

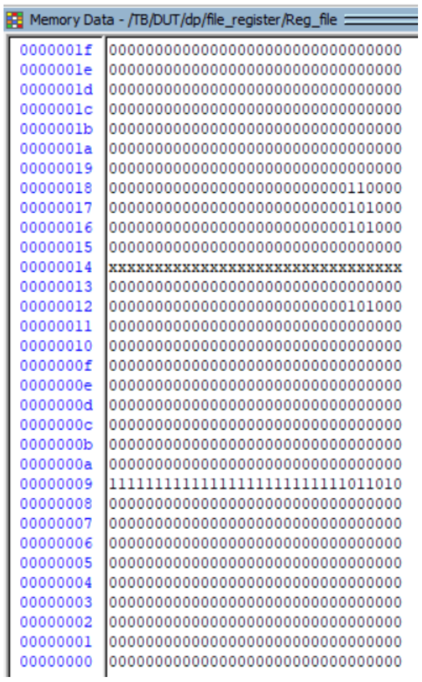


```
[18]  32'd28    40
[17]  32'd0     0
[16]  32'd0     0
[15]  32'd0     0
[14]  32'd0     0
[13]  32'd0     0
[12]  32'd0     0
[11]  32'd0     0
[10]  32'd0     0
[9]   -32'd38   -38
[8]   32'd0     0
[7]   32'd0     0
[6]   32'd0     0
[5]   32'd0     0
[4]   32'd0     0
[3]   32'd0     0
```

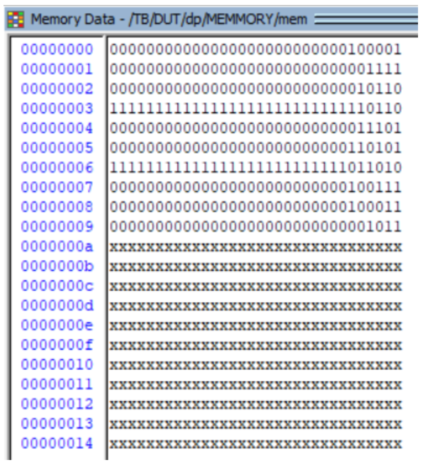
It can be seen that the smallest element (-38) is stored in the s1 register.
Memory files after code execution:



```
Memory Data - /TB/DUT/dp/instruction_memmory/mem - Default

00000000  00000010100000000000101110010011
00000001  00000000000000000000010010110011
00000002  00000000000000000000100100110011
00000003  00000000000000000000100110110011
00000004  00000001001110010000101100110011
00000005  00000000000010110010101000000011
00000006  00000001011110010000110001100011
00000007  00000000100110100010101010110011
00000008  00000000000010101000010001100011
00000009  00000000000010100000010010110011
0000000a  00000000010010010000100100010011
0000000b  11111110010111111111110001101111
0000000c  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000d  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000e  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000010  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000011  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000013  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000014  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```



```
Memory Data - /TB/DUT/dp/file_register/Reg_file

0000001f  00000000000000000000000000000000
0000001e  00000000000000000000000000000000
0000001d  00000000000000000000000000000000
0000001c  00000000000000000000000000000000
0000001b  00000000000000000000000000000000
0000001a  00000000000000000000000000000000
00000019  00000000000000000000000000000000
00000018  00000000000000000000000000110000
00000017  00000000000000000000000000101000
00000016  00000000000000000000000000101000
00000015  00000000000000000000000000000000
00000014  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000013  00000000000000000000000000000000
00000012  00000000000000000000000000101000
00000011  00000000000000000000000000000000
00000010  00000000000000000000000000000000
0000000f  00000000000000000000000000000000
0000000e  00000000000000000000000000000000
0000000d  00000000000000000000000000000000
0000000c  00000000000000000000000000000000
0000000b  00000000000000000000000000000000
0000000a  00000000000000000000000000000000
00000009  11111111111111111111111111011010
00000008  00000000000000000000000000000000
00000007  00000000000000000000000000000000
00000006  00000000000000000000000000000000
00000005  00000000000000000000000000000000
00000004  00000000000000000000000000000000
00000003  00000000000000000000000000000000
00000002  00000000000000000000000000000000
00000001  00000000000000000000000000000000
00000000  00000000000000000000000000000000
```



```
Memory Data - /TB/DUT/dp/MEMMORY/mem

00000000  00000000000000000000000000100001
00000001  00000000000000000000000000001111
00000002  00000000000000000000000000010110
00000003  11111111111111111111111111110110
00000004  00000000000000000000000000011101
00000005  00000000000000000000000000110101
00000006  11111111111111111111111111011010
00000007  00000000000000000000000000100111
00000008  00000000000000000000000000100011
00000009  00000000000000000000000000001011
0000000a  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000b  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000d  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000e  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000010  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000011  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000013  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000014  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Citation

Harris, S. L., & Harris, D. (2021). Digital Design and RISC-V Computer Architecture Textbook.
https://doi.org/10.1109/wcae53984.2021.9707615
https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf