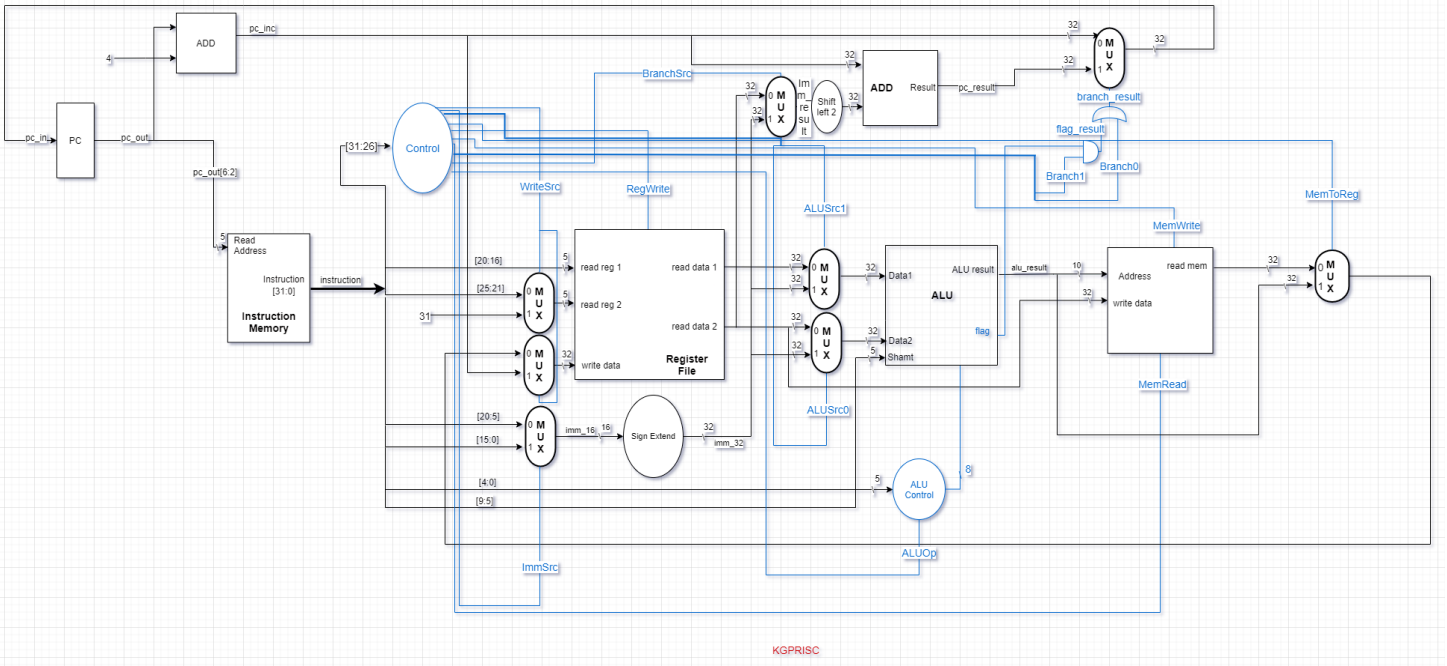
Computer Organization Laboratory

KGP-RISC ISA



GROUP - 7

Battala Vamshi Krishna - 19CS10019

Mahajan Rohan Raj - 19CS10042

Instruction Format:

R-type

opcode [31:26]	dest reg [25:21]	src reg [20:16]	shamt [15:5]	func code [4:0]
6 bit	5 bit	5 bit	11 bit (6 bits are redundant)	5 bit

I-type

opcode [31:26]	dest reg [25:21]	immediate [20:5]	func code [4:0]
6 bit	5 bit	16 bit	5 bit

Load-Store type

opcode [31:26]	dest reg [25:21]	src reg [20:16]	Immediate [15:0]
6 bit	5 bit	5 bit	16 bit

B-type

opcode [31:26]	reg [25:21]	offset [20:5]	func code [4:0]
6 bit	5 bit	16 bit	5 bit

NOTES:

- Maximum number of instructions that can be encoded = $2^5.(2^6-2) = 32*62 = 1984$
  [2 Opcodes are used to encode Load and Store out of 2^6 Opcodes]
- Number of instruction encodings used = 21

OP-CODES & FUNCTION CODES

Instruction	Instruction format	opcode	func code
add	R-type	000000	00000 (0)
comp	R-type	000000	00001 (1)
addi	I-type	001000	00000 (0)
compi	I-type	001000	00001 (1)
and	R-type	000000	00010 (2)
xor	R-type	000000	00011 (3)
shll	I-type	001000	00100 (4)
shrl	I-type	001000	00101 (5)
shllv	R-type	000000	00100 (4)
shrlv	R-type	000000	00101 (5)
shra	I-type	001000	00110 (6)
shrav	R-type	000000	00110 (6)
lw	Load-Store-type	100000 (32)	--------
sw	Load-Store-type	100001 (33)	--------
b	B-type (I)	010000	xxxxx
br	B-type (R)	010001	xxxxx
bltz	B-type (RI)	010010	01000
bz	B-type (RI)	010010	01001
bnz	B-type (RI)	010010	01010
bl	B-type (I)	010011	xxxxx
bcy	B-type (I)	010100	01011
bncy	B-type (I)	010101	01100

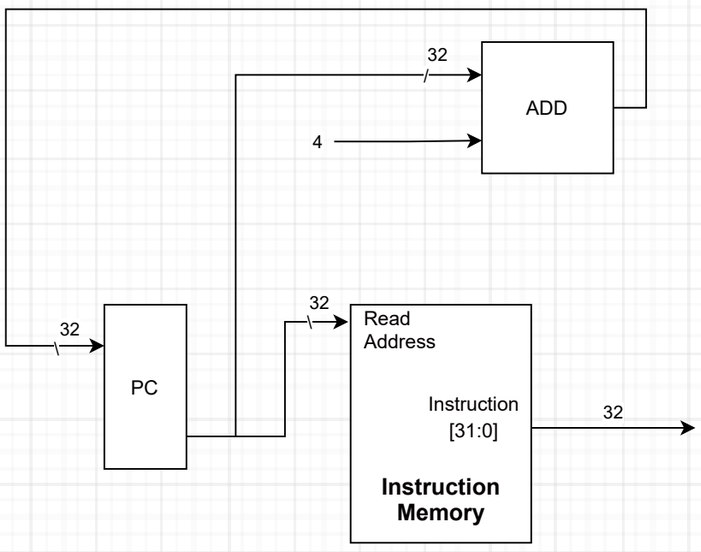
CONTROL

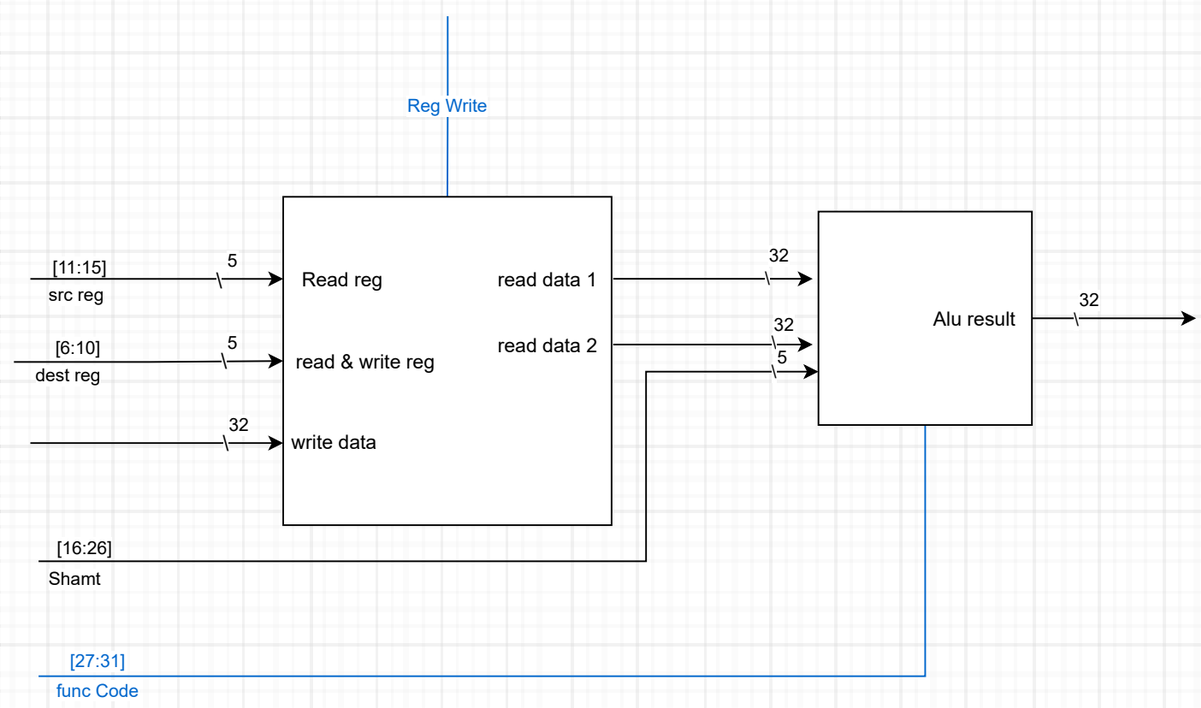
Instr	opcode	WriteSrc	ImmSrc	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	BranchSrc	ALUOp
R-type	000000	0	X	00	1	1	0	0	00	X	1
I-type	001000	0	0	10	1	1	0	0	00	X	1
load	100000	0	1	01	0	1	1	0	00	X	0
store	100001	0	1	01	X	0	0	1	00	X	0
b	010000	0	0	XX	X	0	X	0	X1	1	1
br	010001	0	0	X0	X	0	X	0	X1	0	1
bltz	010010	0	0	X0	X	0	X	0	10	1	1
bz	010010	0	0	X0	X	0	X	0	10	1	1
bnz	010010	0	0	X0	X	0	X	0	10	1	1
bl	010011	1	0	X0	X	0	X	0	10	1	1
bcy	010010	0	0	X0	X	0	X	0	10	1	1
bncy	010010	0	0	X0	X	0	X	0	10	1	1

ALU CONTROL

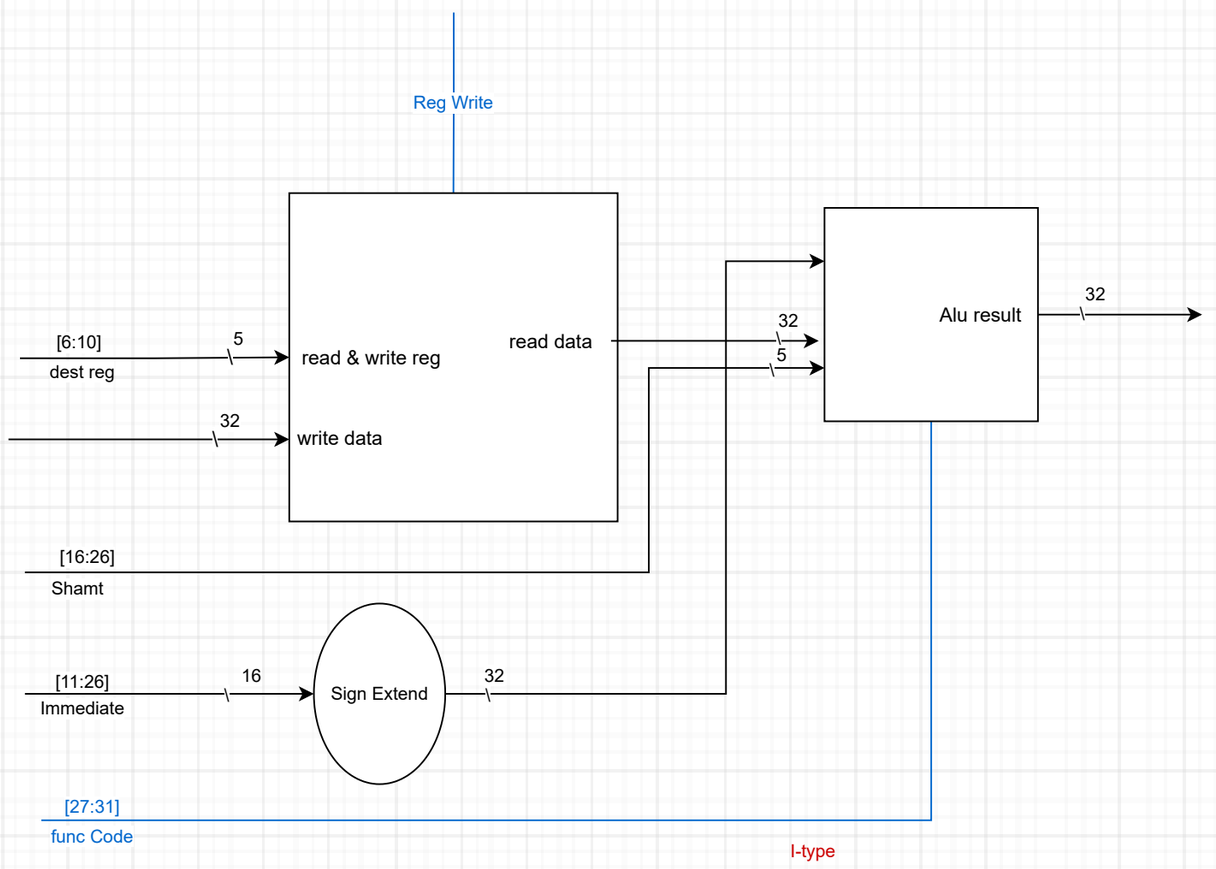
instruction	**Function Code**	**ALUOp**	IsLog	Dir	OPSwitch	FlagSwitch
add, addi	00000 (0)	1	X	X	000	XXX
comp, compi	00001 (1)	1	X	X	011	XXX
and	00010 (2)	1	X	X	001	XXX
xor	00011 (3)	1	X	X	010	XXX
shll, shlv	00100 (4)	1	1	1	100	XXX
shrl, shrlv	00101 (5)	1	1	0	100	XXX
shra, shrav	00110 (6)	1	0	0	100	XXX
load	_____	0	X	X	000	XXX
store	_____	0	X	X	000	XXX
b,br	XXXXX	X	X	X	XXX	XXX
bltz	01000	1	X	X	XXX	100
bz	01001	1	X	X	XXX	011
bnz	01010	1	X	X	XXX	010
bl	XXXXX	X	X	X	XXX	XXX
bcy	01011	1	X	X	XXX	000
bncy	01100	1	X	X	XXX	001

DATA PATHS for Each Instruction Format

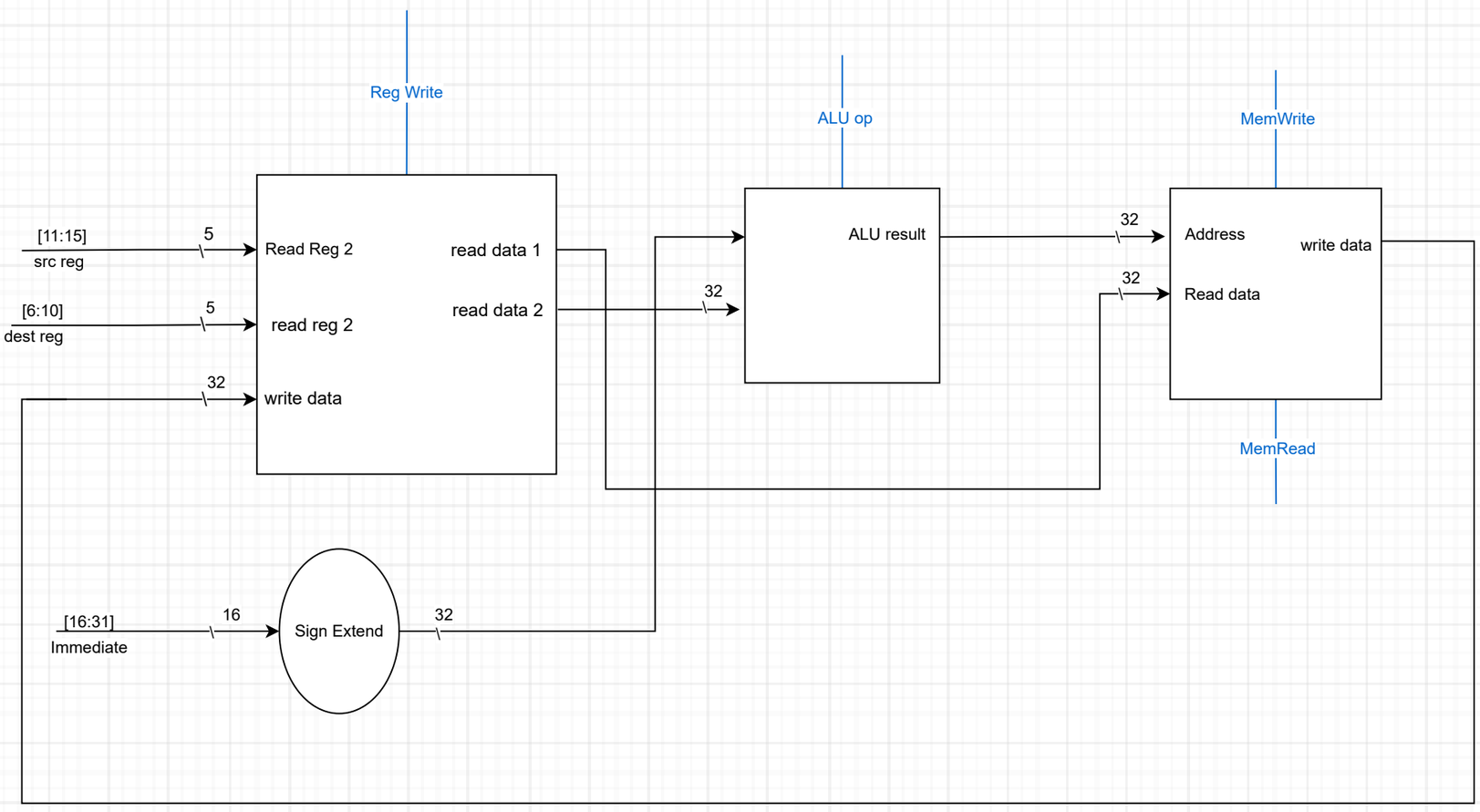




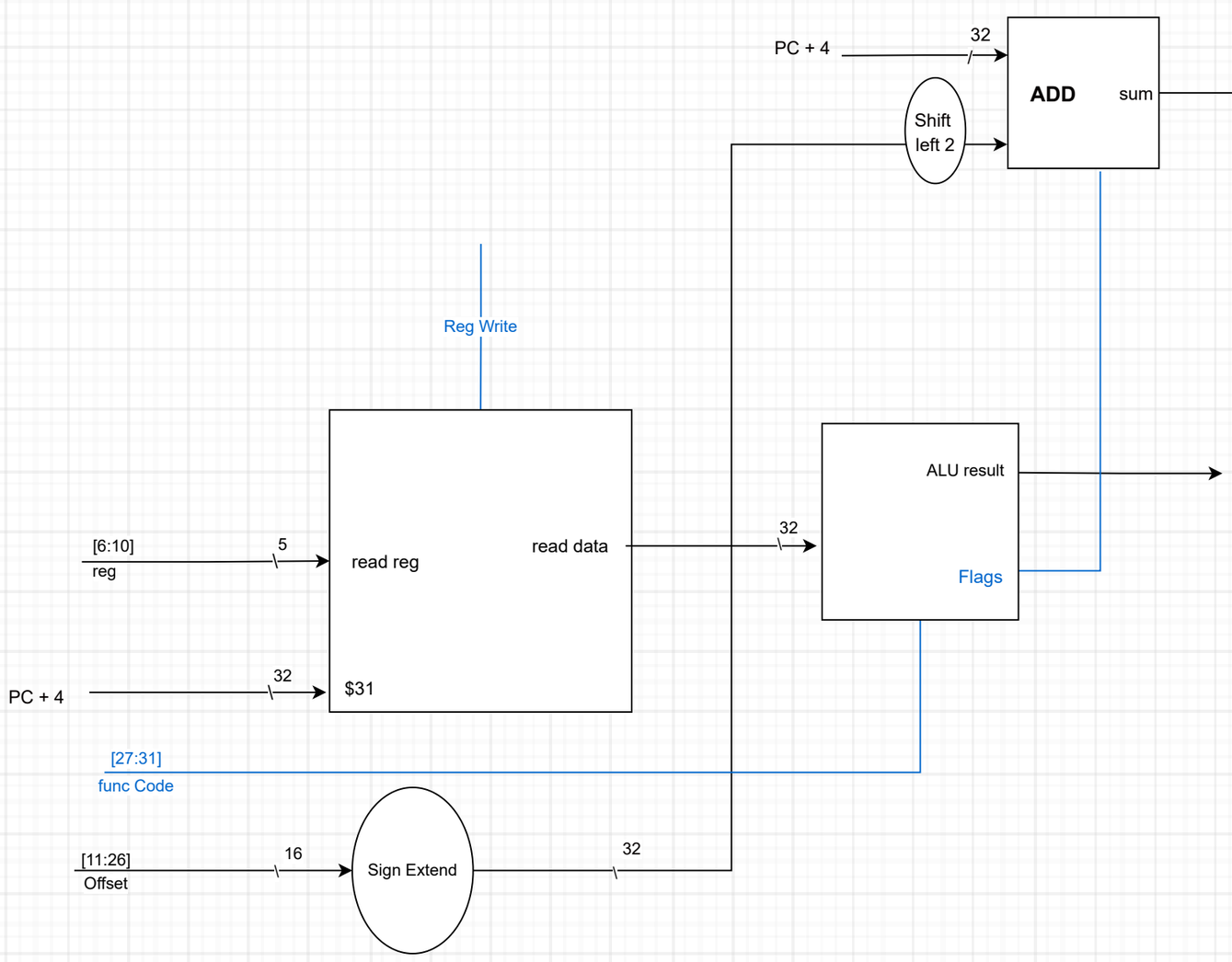
R-type



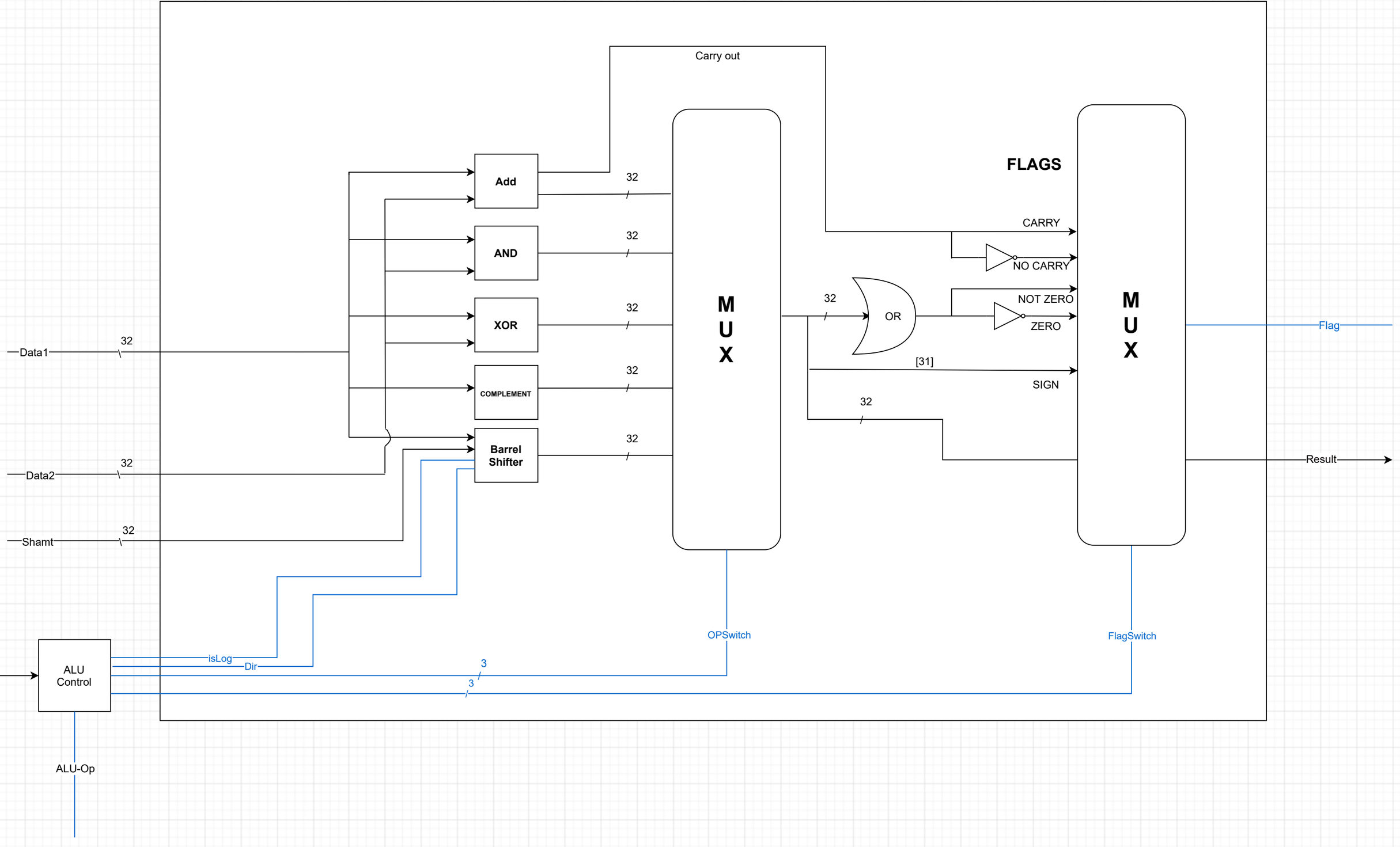
I-type



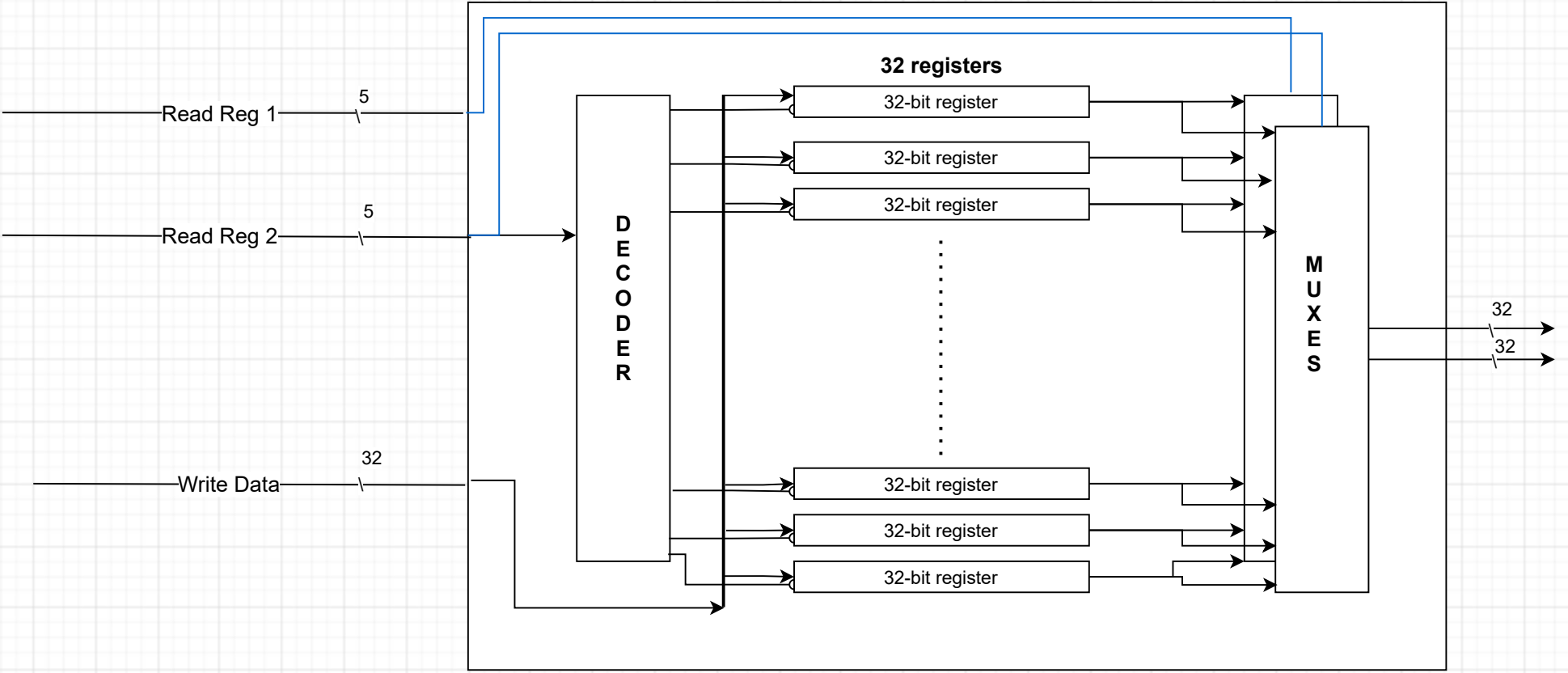
Load-Store-type



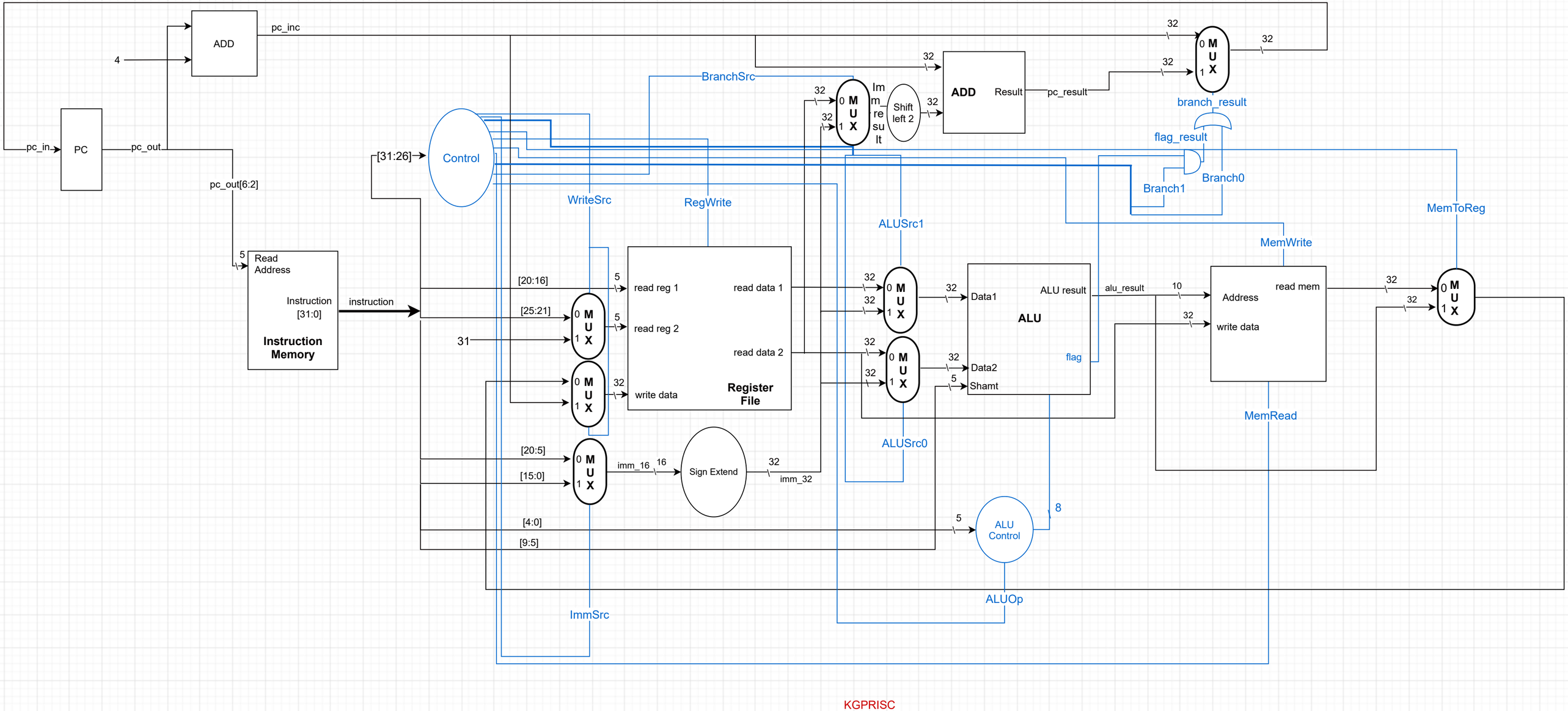
B-type


Carry out
Add
AND
XOR
COMPLEMENT
Barrel Shifter
32
MUX
FLAGS
CARRY
NO CARRY
NOT ZERO
ZERO
OR
[31]
SIGN
MUX
Flag
Result
Data1
Data2
Shamt
FUNC CODE
ALU Control
isLog
Dir
3
OPSwitch
FlagSwitch
ALU-Op

ALU



Register File


ADD
4
pc_inc
PC
pc_in
pc_out
pc_out[6:2]
[31:26]
Control
BranchSrc
WriteSrc
RegWrite
ALUSrc1
ALUSrc0
ImmSrc
MemWrite
MemRead
MemToReg
ALUOp
Read Address
Instruction [31:0]
Instruction Memory
instruction
[20:16]
[25:21]
31
[20:5]
[15:0]
[4:0]
[9:5]
read reg 1
read reg 2
write data
read data 1
read data 2
Register File
imm_16
Sign Extend
imm_32
Imm_result
Shift left 2
ADD
Result
pc_result
branch_result
flag_result
Branch1
Branch0
Data1
Data2
Shamt
ALU result
flag
ALU
alu_result
Address
write data
read mem
ALU Control
MUX
0
1
5
10
16
32
8

KGPRISC

DESIGN COMPONENTS

INSTRUCTION FETCH :

In the instruction fetch phase, a Block RAM(BRAM) module is instantiated which acts as the instruction memory. The instructions (source code) are fed to the memory during the initialization phase of the BRAM with a .coe file. The module takes as input a PC (program counter) value and a clock signal, and at every positive edge of the clock, it outputs the instruction (of 32-bits) present in the memory at the location specified by the PC value. The BRAM module has the signals clock(clka), reset(rsta), enable(ena), write-enable,address(for reading or writing)(wea), write-data-input(dina) as input signals and read-data-output(douta) as output signals.

Further, the instruction gets split into different components based on the type of instruction (r,i,ls,b):

- Opcode
- rs
- rt
- shamt
- immediate/offset
- funcCode

It is clear from the register allocation document that the above set of bits are overlapping. This is because depending on the opcode, only a certain set will be used at a time. So, this module clubs all of the instructions as per the definitions, and depending on the flags set by the Control Unit (which in turn take the opcode as input) various other modules of the processor use the above buses.

REGISTER FILE : (register_file.v)

This is a sequential module and takes clk and rst as input. This module holds 32 32-bit registers. It has three inputs of which two(read_reg1 and read_reg2) are 5-bit inputs as they are used to locate the registers whose values will be given as output. The values of the registers given by the above two addresses are given as output. If the RegWrite flag(from Control Unit) is enabled then the 32-bit input over the write_data bus will be written to the register pointed to by read_reg2. The two outputs are the values read at read_reg1 and read_reg2.

CONTROL : (control.v)

This is a combinational module that takes reset, opcode(6 bit) as inputs and outputs all the control lines in the datapath : ALUSrc, Branch, WriteSrc, ImmSrc, MemToReg, RegWrite, MemRead, MemWrite, BranchSrc, ALUOp. The matchings are made according to the table provided above. All the control lines get disabled when reset is enabled.

ALU CONTROL : (alu_control.v)

This module is activated whenever the ALUOp control line is enabled. This is a combinational module which takes reset, ALUOp, funcCode(last 5 bits in instruction encoding) as input and produces isLog (to determine whether shift is logical or not), dir (to determine the direction of shift), opSwitch and flagSwitch. opSwitch is further given as select input to the MUX determining the result of operations in ALU. flagSwitch is further used a select input to the MUX determining the flag to be produced for branch instructions.

ALU : (alu.v)

This is a purely combinational module that takes one input from the register Bank, one from the above MUX and gives a 32-bit output after performing an operation which is controlled by the alu_control select instruction which is of four bits and the default output is zero. The ALU performs the following operations:

- Addition
- Compliment
- Shift
- AND
- XOR

And at each instance, sets the following flags: - sign, carry and zero.
Further, these flags are passed through a MUX to get the required flag during branch instructions.

DATA MEMORY : (data_memory.v)

This is the module simulating the RAM but in the given implementation, we have an array of 1024 registers which are 32-bits each. In this module, there is a 10-bit address bus which is the result of the ALU. and a 32-bit data write BUS which is the output from the Register Bank. In case the MemWrite flag is true then data from the output of the register bank is written to the data. If the MemRead flag is true then the Data Memory outputs the contents of the location pointed to by the address input to the Data Memory.

Other Components:

- Basic components such as AND gate, OR gate, MUXes (5bit_2to1, 6bit_2to1, 32bit_2to1), 32-bit Adders, 32-bit shifters, 16to32 sign extension blocks were also made and implemented in the datapath.

Execution Facts:

- Instructions were written in separate .txt files. We have written a code in python that takes these instructions and converts them into an array of 32-bit binary strings. Then this array is fed into .coe file.
- Apart from the instructions to be performed we have added two no-op instructions, one in the beginning and the other at the end of instructions. This is useful so that when the program needs to exit it performs no-op instructions as many times as required. This won't affect the model because the no-op instructions do nothing.

- Write instructions in memory and register file are triggered at the neg-edge of the clock cycle so that the operation does not clash with the next read instructions (which are triggered at pos-edge of the clock).

  Instruction files to perform **GCD of two numbers :**

.txt file	.coe file

addi $t0, 52
addi $t1, 39
 	
bz $t0, return_b
 	
while:
bz $t1, return_a
and $t2, $zero
add $t2, $t1
comp $t3, $t0
add $t2, $t3
bltz $t2, aGb
comp $t3, $t0
add $t1, $t3
b while
 	
aGb:
comp $t4, $t1
add $t0, $t4
b while
 	
return_a:
and $t4, $zero
add $t4, $t0
b exit
 	
return_b:
and $t4, $zero
add $t4, $t1
b exit
 	
exit:
	memory_initialization_radix=2;
	memory_initialization_vector=
	00000000000000000000000000000000,
	00100001000000000000001011100000,
	00100001001000000000100101100000,
	00000001001010000000000000000000,
	10000101001010110000000000000010,
	10000001011010110000000000000010,
	00000001011010000000000000000001,
	00000001000010110000000000000000,
	01001001000111111111111100001001,
	00000000000000000000000000000000;

DESIGN COMPONENTS

INSTRUCTION FETCH :

In the instruction fetch phase, a Block RAM(BRAM) module is instantiated which acts as the instruction memory. The instructions (source code) are fed to the memory during the initialization phase of the BRAM with a .coe file. The module takes as input a PC (program counter) value and a clock signal, and at every positive edge of the clock, it outputs the instruction (of 32-bits) present in the memory at the location specified by the PC value. The BRAM module has the signals clock(clka), reset(rsta), enable(ena), write-enable,address(for reading or writing)(wea), write-data-input(dina) as input signals and read-data-output(douta) as output signals.

Further, the instruction gets split into different components based on the type of instruction (r,i,ls,b):

- Opcode
- rs
- rt
- shamt
- immediate/offset
- funcCode

It is clear from the register allocation document that the above set of bits are overlapping. This is because depending on the opcode, only a certain set will be used at a time. So, this module clubs all of the instructions as per the definitions, and depending on the flags set by the Control Unit (which in turn take the opcode as input) various other modules of the processor use the above buses.

REGISTER FILE : (register_file.v)

This is a sequential module and takes clk and rst as input. This module holds 32 32-bit registers. It has three inputs of which two(read_reg1 and read_reg2) are 5-bit inputs as they are used to locate the registers whose values will be given as output. The values of the registers given by the above two addresses are given as output. If the RegWrite flag(from Control Unit) is enabled then the 32-bit input over the write_data bus will be written to the register pointed to by read_reg2. The two outputs are the values read at read_reg1 and read_reg2.

CONTROL : (control.v)

This is a combinational module that takes reset, opcode(6 bit) as inputs and outputs all the control lines in the datapath : ALUSrc, Branch, WriteSrc, ImmSrc, MemToReg, RegWrite, MemRead, MemWrite, BranchSrc, ALUOp. The matchings are made according to the table provided above. All the control lines get disabled when reset is enabled.

ALU CONTROL : (alu_control.v)

This module is activated whenever the ALUOp control line is enabled. This is a combinational module which takes reset, ALUOp, funcCode(last 5 bits in instruction encoding) as input and produces isLog (to determine whether shift is logical or not), dir (to determine the direction of shift), opSwitch and flagSwitch. opSwitch is further given as select input to the MUX determining the result of operations in ALU. flagSwitch is further used a select input to the MUX determining the flag to be produced for branch instructions.

ALU : (alu.v)

This is a purely combinational module that takes one input from the register Bank, one from the above MUX and gives a 32-bit output after performing an operation which is controlled by the alu_control select instruction which is of four bits and the default output is zero. The ALU performs the following operations:

- Addition
- Compliment
- Shift
- AND
- XOR

And at each instance, sets the following flags: - sign, carry and zero.
Further, these flags are passed through a MUX to get the required flag during branch instructions.

DATA MEMORY : (data_memory.v)

This is the module simulating the RAM but in the given implementation, we have an array of 1024 registers which are 32-bits each. In this module, there is a 10-bit address bus which is the result of the ALU. and a 32-bit data write BUS which is the output from the Register Bank. In case the MemWrite flag is true then data from the output of the register bank is written to the data. If the MemRead flag is true then the Data Memory outputs the contents of the location pointed to by the address input to the Data Memory.

Other Components:

- Basic components such as AND gate, OR gate, MUXes (5bit_2to1, 6bit_2to1, 32bit_2to1), 32-bit Adders, 32-bit shifters, 16to32 sign extension blocks were also made and implemented in the datapath.

Execution Facts:

- Instructions were written in separate .txt files. We have written a code in python that takes these instructions and converts them into an array of 32-bit binary strings. Then this array is fed into .coe file.
- Apart from the instructions to be performed we have added two no-op instructions, one in the beginning and the other at the end of instructions. This is useful so that when the program needs to exit it performs no-op instructions as many times as required. This won't affect the model because the no-op instructions do nothing.

- Write instructions in memory and register file are triggered at the neg-edge of the clock cycle so that the operation does not clash with the next read instructions (which are triggered at pos-edge of the clock).

    1) Instruction files to perform **GCD of two numbers :**

.txt file	.coe file

addi $t0, 52
addi $t1, 39
 	
bz $t0, return_b
 	
while:
bz $t1, return_a
and $t2, $zero
add $t2, $t1
comp $t3, $t0
add $t2, $t3
bltz $t2, aGb
comp $t3, $t0
add $t1, $t3
b while
 	
aGb:
comp $t4, $t1
add $t0, $t4
b while
 	
return_a:
and $t4, $zero
add $t4, $t0
b exit
 	
return_b:
and $t4, $zero
add $t4, $t1
b exit
 	
exit:
	memory_initialization_radix=2;
	memory_initialization_vector=
	00000000000000000000000000000000,
	00100001000000000000011010000000,
	00100001001000000000010011100000,
	01001001000000000000000111101001,
	01001001001000000000000101101001,
	00000001010000000000000000000010,
	00000001010010010000000000000000,
	00000001011010000000000000000001,
	00000001010010110000000000000000,
	01001001010000000000000001101000,
	00000001011010000000000000000001,
	00000001001010110000000000000000,
	01000000000111111111111011100000,
	00000001100010010000000000000001,
	00000001000011000000000000000000,
	01000000000111111111111010000000,
	00000001100000000000000000000010,
	00000001100010000000000000000000,
	01000000000000000000000001100000,
	00000001100000000000000000000010,
	00000001100010010000000000000000,
	01000000000000000000000000000000,
	00000000000000000000000000000000;

Execution Results:

Instruction Memory

	0	1	2	3
0x0	00000000000000000000000000000000	00100001000000000000011010000000	00100001001000000000010011100000	01001001000000000000000111101001
0x4	01001001001000000000000101101001	00000001010000000000000000000010	00000001010010010000000000000000	00000001011010000000000000000001
0x8	00000001010010110000000000000000	01001001010000000000000001101000	00000001011010000000000000000001	00000001001010110000000000000000
0xC	01000000000111111111111011100000	00000001100010010000000000000001	00000001000011000000000000000000	01000000000111111111111010000000
0x10	00000001100000000000000000000010	00000001100010000000000000000000	01000000000000000000000001100000	00000001100000000000000000000010
0x14	00000001100010010000000000000000	01000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000
0x18	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000
0x1C	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000

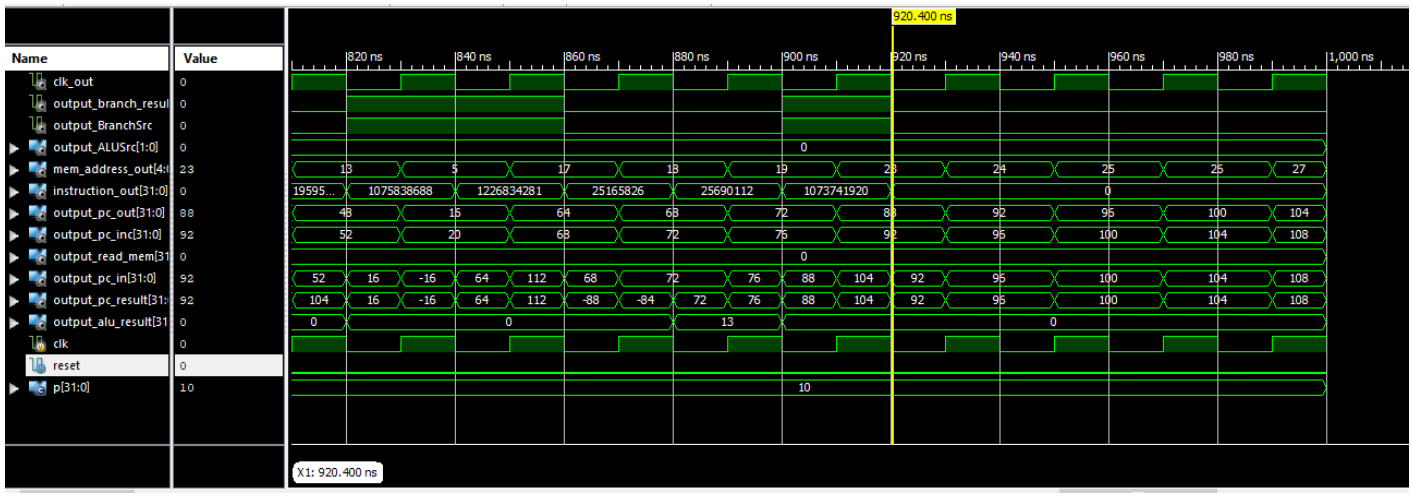
Initial and final instructions are no-op instructions (0).

	0	1	2	3
0x1F	0	0	0	0
0x1B	0	0	0	0
0x17	0	0	0	0
0x13	0	0	0	0
0xF	0	0	0	13
0xB	-13	0	0	13
0x7	0	0	0	0
0x3	0	0	0	0

Register File

The result of the GCD of two numbers is stored in $t4, which is 13 in this case gcd(39,52).

Simulation



2) Instruction files of **Random program** that verifies the working of **lw,sw,comp,bz** instructions**:**

.txt file	.coe file

label:
addi $t0, 23
addi $t1, 75
add $t1, $t0
sw $t1, (2)$t3
lw $t3, (2)$t3
comp $t3, $t0
add $t0, $t3
bz $t0, label
	memory_initialization_radix=2;
	memory_initialization_vector=
	00000000000000000000000000000000,
	00100001000000000000001011100000,
	00100001001000000000100101100000,
	00000001001010000000000000000000,
	10000101001010110000000000000010,
	10000001011010110000000000000010,
	00000001011010000000000000000001,
	00000001000010110000000000000000,
	01001001000111111111111100001001,
	00000000000000000000000000000000;

	0	1	2	3
0x0	0	553648864	555747680	19398656
0x4	2234187778	2171273218	23592961	17498112
0x8	1226833673	0	0	0
0xC	0	0	0	0
0x10	0	0	0	0
0x14	0	0	0	0
0x18	0	0	0	0
0x1C	0	0	0	0

Instruction Memory

	0	1	2	3
0x1F	0	0	0	0
0x1B	0	0	0	0
0x17	0	0	0	0
0x13	0	0	0	0
0xF	0	0	0	0
0xB	-23	0	588	0
0x7	0	0	0	0
0x3	0	0	0	0

Register File

	0	1	2	3
0x4F	0	0	0	0
0x4B	0	0	0	0
0x47	0	0	0	0
0x43	0	0	0	0
0x3F	0	0	0	0
0x3B	0	0	0	0
0x37	0	0	0	0
0x33	0	0	0	0
0x2F	0	0	0	0
0x2B	0	0	0	0
0x27	0	0	0	0
0x23	0	0	0	0
0x1F	0	0	0	0
0x1B	0	0	0	0
0x17	0	0	0	0
0x13	0	0	0	0
0xF	0	0	0	0
0xB	0	0	0	0
0x7	0	0	0	0
0x3	0	98	0	0

DATA MEMORY

98, which is the value stored in $t1 is now stored in data memory at location 2($t3) i.e. 2

Simulation