KGP-RISC ISA

Group: 22
Members: Amit Kumar (20CS30003), Nikhil Saraswat (20CS10039)

Instruction formats:

There are 3 types of instruction formats:

1. R-format (Register format)
2. I-format (Immediate format)
3. J-format (Jump format)

R-format

Field size	6	5	5	5	6	5	Instructions
R-format	op	rs	rt	shamt	func	dc	add, cmp, and, xor, all-shift instructions

Currently, all R-format instructions are kept under the same op-code.

op-code: **000000**

Instruction	**func-code**
add	000001
cmp	000101
diff	000100
and	000010
xor	000011
shll	001100
shrl	001110
shllv	001000
shrlv	001010
shra	001111

I-format

Field-size	6	5	5	16	Instructions
I-format	op	rs	rt/dc	address/immediate	lw, sw, addi, cmpi,

The following table enlists the op-code for all the above instructions

Instruction	**op-code**
lw	010000
sw	011000
addi	001000
cmpi	001001

J-format

Field size	6	26	Instructions
J-format	op	target address	b, br, bl, bcy, bncy

The following table enlists the op-code for all the above instructions

Instruction	**op-code**	**fmt**
br	100000	op | rs | xxxxxxxx |
b	101000	op | label |
bcy	101001	op | label |
bncy	101010	op | label |
bl	101011	op | label |
bltz	110000	op | rs | xx | label |
bz	110001	op | rs | xx | label |
bnz	110010	op | rs | xx | label |

The following table summarizes all the instructions and the associated op-codes

LSB MSB	000	001	010	011	100	101	110	111
000	R-format							
001	addi	cmpi						
010	lw							
011	sw							
100	br							
101	b	bcy	bncy	bl				
110	bltz	bz	bnz					
111								

Register convention:

The architectural design is suggested to be used in a similar manner to the MIPS convention of register use as mentioned:

Symbolic name	**Number**	**use**
zero	0	Constant 0
at	1	Reserved for assembler
v0-v1	2-3	Result registers
a0-a3	4-7	Argument register
t0-t9	8-15, 24-25	Temporary register
s0-s7	16-23	Saved register
k0-k1	26-27	Kernel register
gp	28	Global Data pointer
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return Address

Arithmetic Logic Unit Design

The proposed ALU has an internal adder module, and xor modules and a barrel shifter with lines for direction and type(logical/arithmetic).

A separate input bus is also present for shift-amount to be passed as input to shift module. The shift input can either come directly from the shift input bus or from the 5 LSB's of the register specified in the *shift-variable instructions* .
A 2-1 Mux is hence used to select either of these inputs.

For simplicity, since the ISA doesn't have the subtract command natively, no carry-in has been provided; rather for the complement instruction, a 2-1 MUX is used to switch input-1 with 32'd1 and the adder module is used to compute the complement.

Flags from ALU:

1. Carry
2. Zero result
3. Sign of result (0 for +, 1 for -)

ALL ALU-operations are embedded directly into the funcode or in the operation-code passed on from the ALU-controller module

The ALU design has been included in the documentation and also embedded here:

ALU-operation	**funcode[3]**	**funcode[2]**	**funcode[1]**	**funcode[0]**
forward	0	0	0	0
add	0	compl/not-compl	0	1
and	0	0	1	0
xor	0	0	1	1
shift	1	shamt/reg	right/left	log/arithm

Truth table for ALU-control signals with associated op-codes and func-codes

Operation	**Opcode**	**funcode**	alucode[3]	alucode[2]	alucode[1]	alucode[0]1
add	000000	000001	0	0	0	1
comp	000000	000101	0	1	0	1
addi	001000	-	0	0	0	1
diff	000000	000100	0	1	0	0
compi	001001	-	0	1	0	1
and	000000	000010	0	0	1	0
xor	000000	000011	0	0	1	1
shll	000000	001100	1	1	0	0
shrl	000000	001110	1	1	1	0
shllv	000000	001000	1	0	0	0
shrlv	000000	001010	1	0	1	0
shra	000000	001111	1	1	1	1
shrav	000000	001011	1	0	1	1
lw	010000	-	0	0	0	1
sw	011000	-	0	0	0	1
b	101000		0	0	0	0
br	100000	-	0	0	0	0
bltz	110000	-	0	0	0	0
bz	110001	-	0	0	0	0
bnz	110010	-	0	0	0	0
bl	101011	-	0	0	0	0
bcy	101001	-	0	0	0	0
bncy	101010	-	0	0	0	0

Datapath and control signals for the ISA

The given document (separate and embedded) shows the complete data paths and control signals for the Instruction set. The data-paths are shown using black

lines and the control signals using the red lines. For simplicity the controller lines have not been joined to the modules themselves rather left open for better visual clarity.

The ISA contains 3 primary control modules:

1. Controller: handles the primary signals to all modules
2. ALU-Controller: handles the ALU-specific control signals
3. Branch-Controller: handles the logic for branch on flag-signals and produces the output if it has to branch or not

There are 3 standard ways in which instruction memory can be referenced in the Instruction set:

1. Direct PC addressing
2. PC-relative addressing
3. (Pseudo)Direct jump addressing

Direct addressing:

This addressing takes place in case of `br` instruction in which case the argument register contains the exact address to jump to.

PC-relative addressing:

This addressing takes in case of any 16-bit Label instruction such as bz, bnz, bltz in which case the absolute address is calculated using this formula:

Address = (PC + 4) + SignExtended(Label)

Pseudo Direct addressing:

This addressing takes in case of any 26-bit label instruction such as b, bl, bcy, bncy in which case the absolute address is calculated using this formula:

Address = {(PC+4)[31:28] , {Label, 2b'00}}

Control signals:

1. *Regwrite*: whether to write into the register file or not
2. *RegDst[1:0]:* destination register for the write-register (can be $ra, rs, rt)
3. *ALUSrc:* Source for the 2nd input to the ALU (can be rt, sgn-extend(imm))
4. *MemRead:* whether to read from Data-memory or not
5. *MemWrite:* whether to write into the Data-memory or not
6. *Mem2Reg[1:0]:* write-data for the register files (can be PC+4, mem[], result_ALU)
7. *LblSel:* select type of addressing for PC-relative and PseudoDirect
8. *JumpAddr*: whether the jump address comes from a source reg (rs) or from a label

Truth table for control signals with associated op-codes and func-codes

Op	Opcode	RegDst	RegWrite	ALUSrc	MemRead	MemWrite	Mem2Reg	LblSel	JumpSel
add	000000	00	1	0	0	0	00	x	x
comp	000000	00	1	0	0	0	00	x	x
addi	001000	00	1	1	0	0	00	x	x
diff	000000	00	1	0	0	0	00	x	x
compi	001001	00	1	1	0	0	00	x	x
and	000000	00	1	0	0	0	00	x	x
xor	000000	00	1	0	0	0	00	x	x
shll	000000	00	1	0	0	0	00	x	x
shrl	000000	00	1	0	0	0	00	x	x
shllv	000000	00	1	0	0	0	00	x	x
shrlv	000000	00	1	0	0	0	00	x	x
shra	000000	00	1	0	0	0	00	x	x
shrav	000000	00	1	0	0	0	00	x	x
lw	010000	01	1	1	1	0	01	x	x
sw	011000	x	0	1	0	1	x	x	x
b	101000	x	0	x	0	0	x	0	0
br	100000	x	0	x	0	0	x	x	1

bltz	110000	x	0	x	0	0	x	1	0
bz	110001	x	0	x	0	0	x	1	0
bnz	110010	x	0	x	0	0	x	1	0
bl	101011	10	1	x	0	0	10	0	0
bcy	101001	x	0	x	0	0	x	0	0
bncy	101010	x	0	x	0	0	x	0	0

Key Points:

- An assembler has been provided with the verilog files for creating binaries compatible with the KGP-RISC architecture.
- Except the Datamemory and Instruction-memory modules all read operations are asynchronous and any write operation on a DFF is synchronized to the clock.
- The data-memory works on the negative edge of the clock to cope up with the delays from the instruction memory and combinational logic with respect to the driving clock

Bubble Sort as a test bench

As a test bench for the given architecture, Bubble Sort is used to sort an array of integers. The semantics for functional call and stack remain the same from MIPS with similar register convention mentioned in the source assembly code. The stack is simulated using the Data-memory with a $sp register used for referencing the base address of the current stack frame.

MIPS code:

```
main:
    xor $20, $20        # base address of array = 0 ($20)
    xor $21, $21
    addi $21, 10        # $21 = n = 10
    xor $8, $8          # $8 = i = 0
    xor $9, $9          # $9 = j = 0

fori:
    xor $10, $10
    add $10, $8
    comp $11, $21
    add $10, $11
    addi $10, 1         # $10 = i - (n - 1) = i - n + 1
    bz $10, exitfori    # if i == n - 1, jump to exitfori
    xor $9, $9          # j = 0

forj:
    xor $11, $11
    add $11, $9
    add $11, $10        # $11 = j + i - n + 1
    bz $11, exitforj    # if j == n - i - 1, jump to exitforj

    xor $12, $12
    add $12, $9
    shll $12, 2         # 4 * j
    add $12, $20        # arr + 4 * j
    lw $13, 0($12)      # $13 = arr[j]
    xor $4, $4
    add $4, $12
    addi $12, 4
    lw $14, 0($12)      # $14 = arr[j + 1]
    xor $5, $5
    add $5, $12

    comp $15, $14
    add $13, $15        # arr[j] - arr[j + 1]
    bltz $13, incj
    bz $13, incj
    bl swap             # swap if arr[j] > arr[j + 1]

incj:
    addi $9, 1          # j = j + 1
    b forj
```

```
swap:
    lw $18, 0($4)
    lw $19, 0($5)
    sw $18, 0($5)
    sw $19, 0($4)
    br $31

exitforj:
    addi $8, 1          # i = i + 1
    b fori

exitfori:
    xor $16, $16
    addi $16, 1         # to indicate completion
```

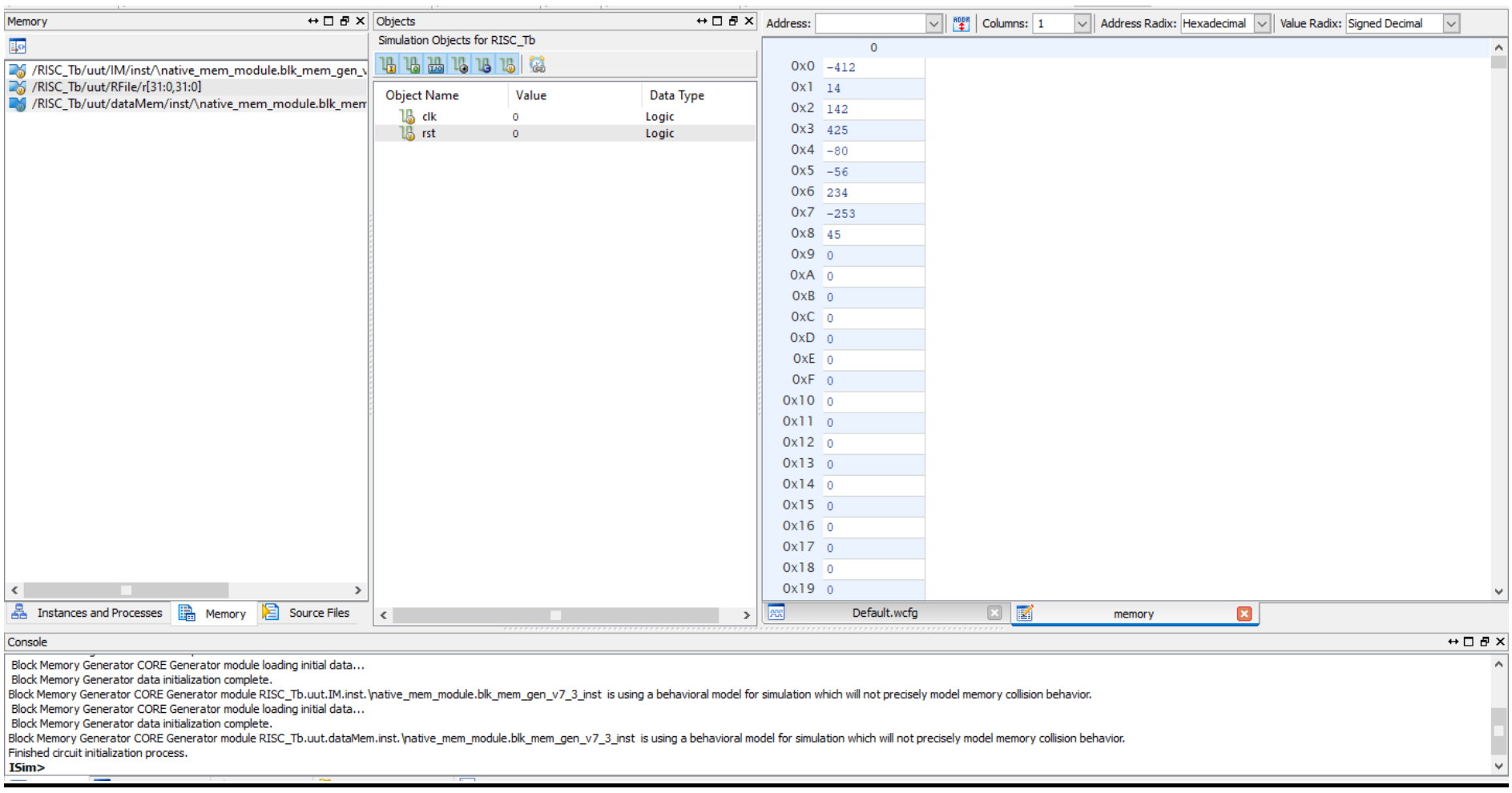
The Byte code:

```
memory_initialization_radix=2;
memory_initialization_vector=
00000010100101000000000001100000,
00000010101101010000000001100000,
00100010101000000000000000001010,
00000001000010000000000001100000,
00000001001010010000000001100000,
00000001010010100000000001100000,
00000001010010000000000000100000,
00000001011101010000000010100000,
00000001010010110000000000100000,
00100001010000000000000000000001,
11000101010000000000000000011110,
00000001001010010000000001100000,
00000001011010110000000001100000,
00000001011010010000000000100000,
00000001011010100000000000100000,
11000101011000000000000000010111,
00000001100011000000000001100000,
00000001100010010000000000100000,
00000001100000000001000110000000,
00000001100101000000000000100000,
01000001100011010000000000000000,
00000000100001000000000001100000,
00000000100011000000000000100000,
00100001100000000000000000000100,
01000001100011100000000000000000,
00000000101001010000000001100000,
00000000101011000000000000100000,
00000001111011100000000010100000,
00000001101011110000000000100000,
11000001101000000000000000000010,
11000101101000000000000000000001,
10101100000000000000000000100010,
00100001001000000000000000000001,
10100000000000000000000000001100,
01000000100100100000000000000000,
01000000101100110000000000000000,
01100000101100100000000000000000,
01100000100100110000000000000000,
10000011111000000000000000000000,
00100001000000000000000000000001,
10100000000000000000000000000101,
00000010000100000000000001100000,
00100010000000000000000000000001;
```

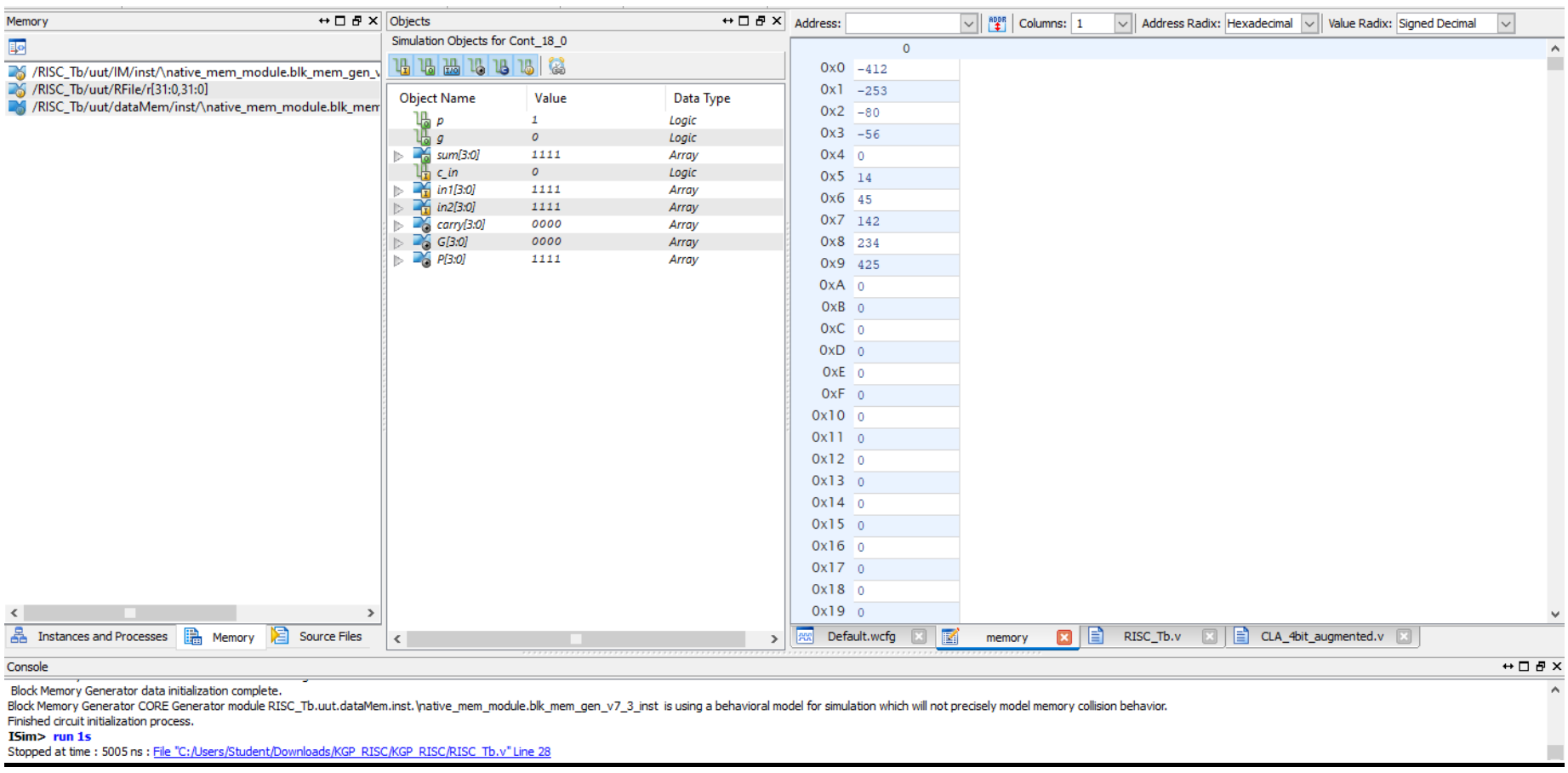
Results:

Bubble Sort:

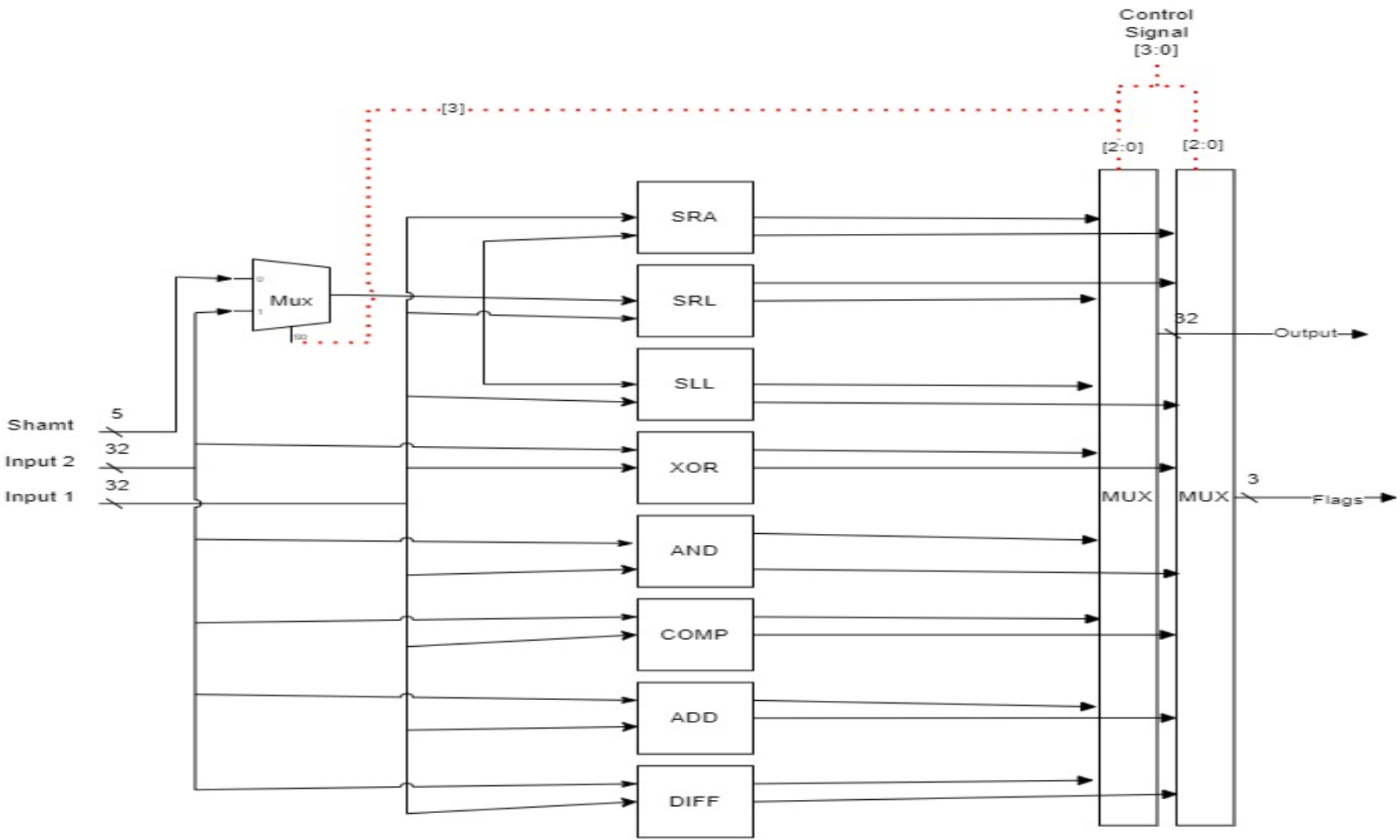
Before sorting:

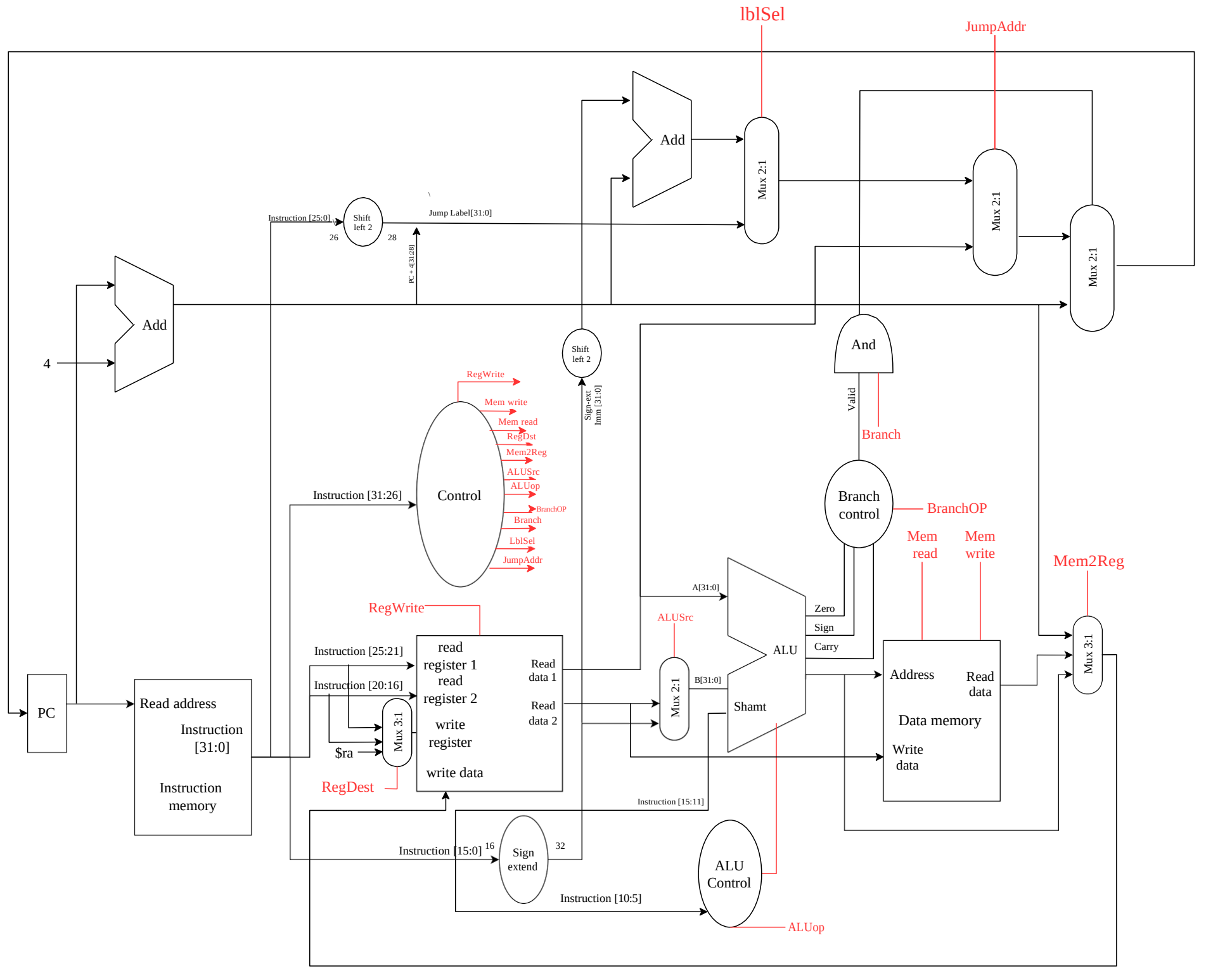


After sorting:



ALU DESIGN KGP-RISC




lblSel
JumpAddr
Add
Mux 2:1
Mux 2:1
Mux 2:1
Instruction [25:0]
Shift left 2
26
28
Jump Label[31:0]
PC + 4[31:28]
Add
4
Shift left 2
Sign-ext Imm [31:0]
And
Valid
Branch
RegWrite
Mem write
Mem read
RegDst
Mem2Reg
ALUSrc
ALUop
BranchOP
Branch
LblSel
JumpAddr
Instruction [31:26]
Control
Branch control
BranchOP
Mem read
Mem write
Mem2Reg
A[31:0]
Zero
Sign
Carry
ALU
ALUSrc
RegWrite
Instruction [25:21]
Instruction [20:16]
read register 1
read register 2
write register
write data
Read data 1
Read data 2
Mux 2:1
B[31:0]
Shamt
Address
Read data
Data memory
Write data
Mux 3:1
PC
Read address
Instruction [31:0]
Instruction memory
Mux 3:1
$ra
RegDest
Instruction [15:11]
Instruction [15:0]
16
Sign extend
32
ALU Control
Instruction [10:5]
ALUop