# Computer Organization and Architecture Laboratory

## KGP MiniRISC Processor

Group 55
Kulkarni Pranav Suryakant - 20CS30029
Vineet Amol Pippal - 20CS30058

# Instruction Set Architecture

## Instruction Format:

### a. R-Type Instructions:

**Fields:**

| | opcode | rs | rt | shamt | not used | funct |
|---|---|---|---|---|---|---|
| size | 6 bits | 5 bits | 5 bits | 5 bits | 6 bits | 5 bits |

**Encoding:**

| Instruction | opcode | func |
|---|---|---|
| 1. add | 000000 | 00001 |
| 2. comp | 000000 | 00101 |
| 3. and | 000000 | 00010 |
| 4. xor | 000000 | 00011 |
| 5. shll | 000000 | 01011 |
| 6. shrl | 000000 | 01001 |
| 7. shllv | 000000 | 11111 |
| 8. shrlv | 000000 | 01101 |
| 9. shra | 000000 | 01000 |
| 10. shrav | 000000 | 01100 |
| 11. diff | 000000 | 00100 |

### b. I-Type Instructions:

**Fields:**

| | opcode | rs | not used | const or address |
|---|---|---|---|---|
| size | 6 bits | 5 bits | 5 bits | 16 bits |

**Encoding:**

| Instruction | opcode |
|---|---|
| 12. addi | 001000 |
| 13. compi | 001001 |

### c. Memory access Instructions:

**Fields:**

| | opcode | rt | rs | const or address |
|---|---|---|---|---|
| size | 6 bits | 5 bits | 5 bits | 16 bits |

**Encoding:**

| Instruction | opcode |
|---|---|
| 14. lw<br>15. sw | 010000<br>011000 |

### d. Jump Instructions: PC relative addressing

**Fields:**

| | opcode | rs | not used | label register |
|---|---|---|---|---|
| size | 6 bits | 5 bits | 5 bits | 16 bits |

**Encoding:**

| Instruction | opcode |
|---|---|
| 16. bltz<br>17. bz<br>18. bnz | 110000<br>110001<br>110010 |

### e. Jump Instructions: Register Addressing

**Fields:**

| | opcode | rs | not used |
|---|---|---|---|
| size | 6 bits | 5 bits | 21 bits |

**Encoding:**

| Instruction | opcode |
|---|---|
| 19. br | 100000 |

### f. Jump Instructions: (Pseudo) Direct Addressing

**Fields:**

| | opcode | Label address |
|---|---|---|
| size | 6 bits | 26 bits |

**Encoding:**

| Instruction | opcode |
|---|---|
| 20. b<br>21. bl<br>22. bcy<br>23. bncy | 101000<br>101011<br>101001<br>101010 |

## Lookup Table for ALU Control:

`Input: opcode, function code`
`Output: 4-bit ALUop code`

ALU allows the following operations: add, and, xor, shift, diff, and a complement.
To generate ALUop code, we can decode function code as follows:

| **Operation** | **ALUop[3]** | **ALUop[2]** | **ALUop[1]** | **ALUop[0]** |
|---|---|---|---|---|
| add | 0 | 0 - no complement<br>1 - complement | 0 | 1 |
| and | 0 | 0 | 1 | 0 |
| xor | 0 | 0 | 1 | 1 |
| shift | 1 | 0 - shamt<br>1 - register | 1 - left<br>0 - right | 1 - logical<br>0 - arithmetic |
| diff | 0 | 1 | 0 | 0 |

Based on the above encoding we can embed ALUop code in the 5 bit function code using it's 4 least significant bits, using this embedding we will get following lookup table for ALU control:

| Operation | Opcode | Funct code | ALUop[3] | ALUop[2] | ALUop[1] | ALUop[0] |
|---|---|---|---|---|---|---|
| add | 000000 | 00001 | 0 | 0 | 0 | 1 |
| comp | 000000 | 00101 | 0 | 1 | 0 | 1 |
| addi | 001000 | - | 0 | 0 | 0 | 1 |
| compi | 001000 | - | 0 | 1 | 0 | 1 |
| and | 000000 | 00010 | 0 | 0 | 1 | 0 |
| xor | 000000 | 00011 | 0 | 0 | 1 | 1 |
| shll | 000000 | 01011 | 1 | 0 | 1 | 1 |
| shrl | 000000 | 01001 | 1 | 0 | 0 | 1 |
| shllv | 000000 | 11111 | 1 | 1 | 1 | 1 |

| | | | | | | |
|---|---|---|---|---|---|---|
| shrlv | 000000 | 01101 | 1 | 1 | 0 | 1 |
| shra | 000000 | 01000 | 1 | 0 | 0 | 0 |
| shrav | 000000 | 01100 | 1 | 1 | 0 | 0 |
| lw | 010000 | | 0 | 0 | 0 | 0 |
| sw | 011000 | - | 0 | 0 | 0 | 0 |
| 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 |
| diff | 001111 | - | 0 | 1 | 0 | 0 |
| | | - | | | | |

## Datapath and control signals:

Given below is the complete data paths and control signals for the Instruction set. For simplicity the controller lines have not been joined to the modules themselves and are left open for better visual clarity.

The ISA contains 3 primary control modules:

- **Controller**: handles the primary signals to all modules.
- **ALU-Controller:** handles the ALU-specific control signals.
- **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:

- **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:

$$\text{Address} = \{(PC+4)[31:28] , \{\text{Label}, 2'b00\}\}$$

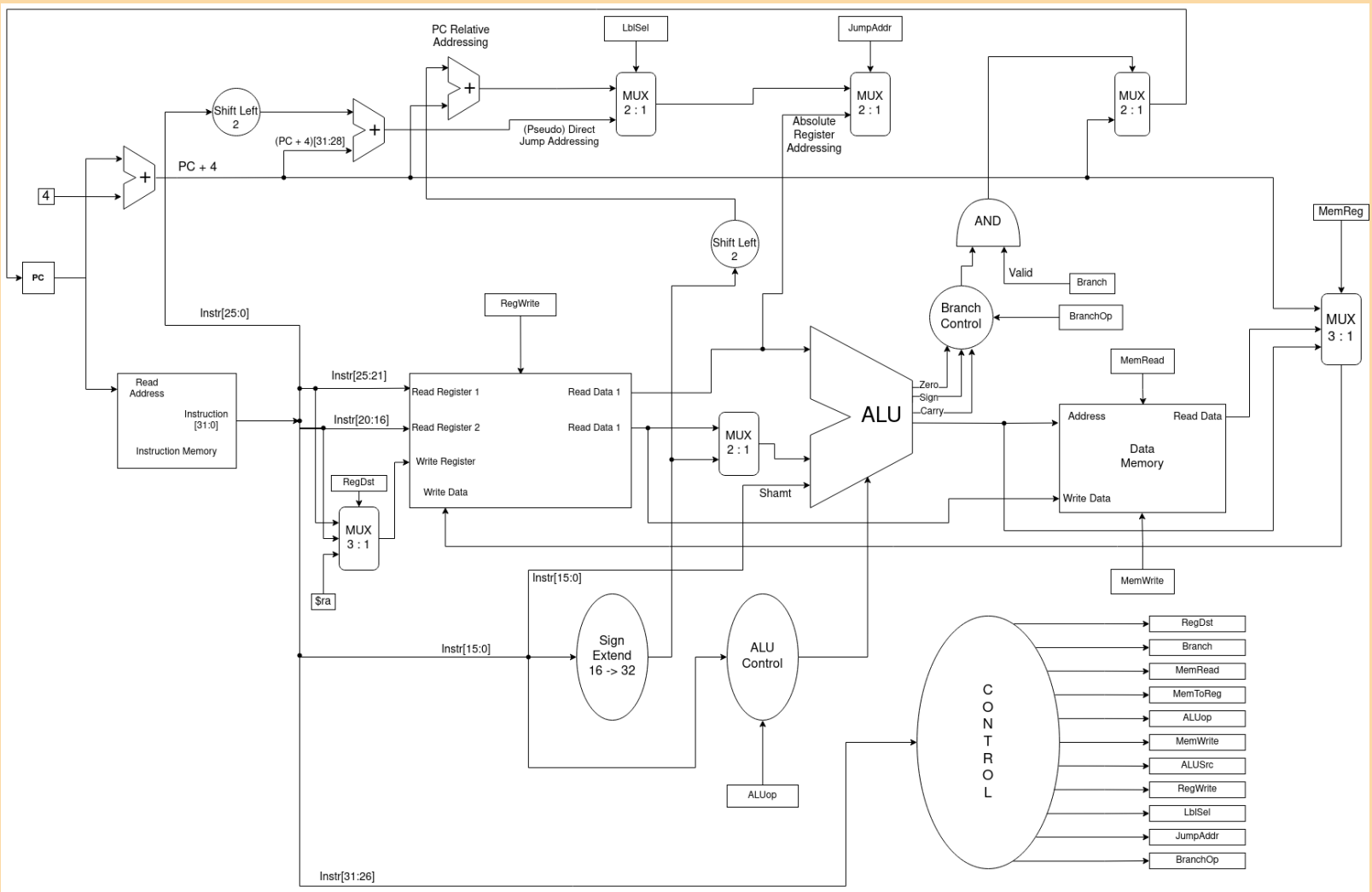
**Control signals:**

- RegWrite: whether to write into the register file or not
- RegDest[1:0]: destination register for the write-register (can be $ra, rs, rt)
- ALUsrc: Source for the 2nd input to the ALU (can be rt, sgn-extend(imm))
- MemRead: whether to read from Data-memory or not
- MemWrite: whether to write into the Data-memory or not
- MemToReg[1:0]: write-data for the register files (can be PC+4, mem[], result_ALU).
- LblSel: select type of addressing for PC-relative and PseudoDirect
- JumpSel: whether the jump address comes from a source reg (rs) or from a label.

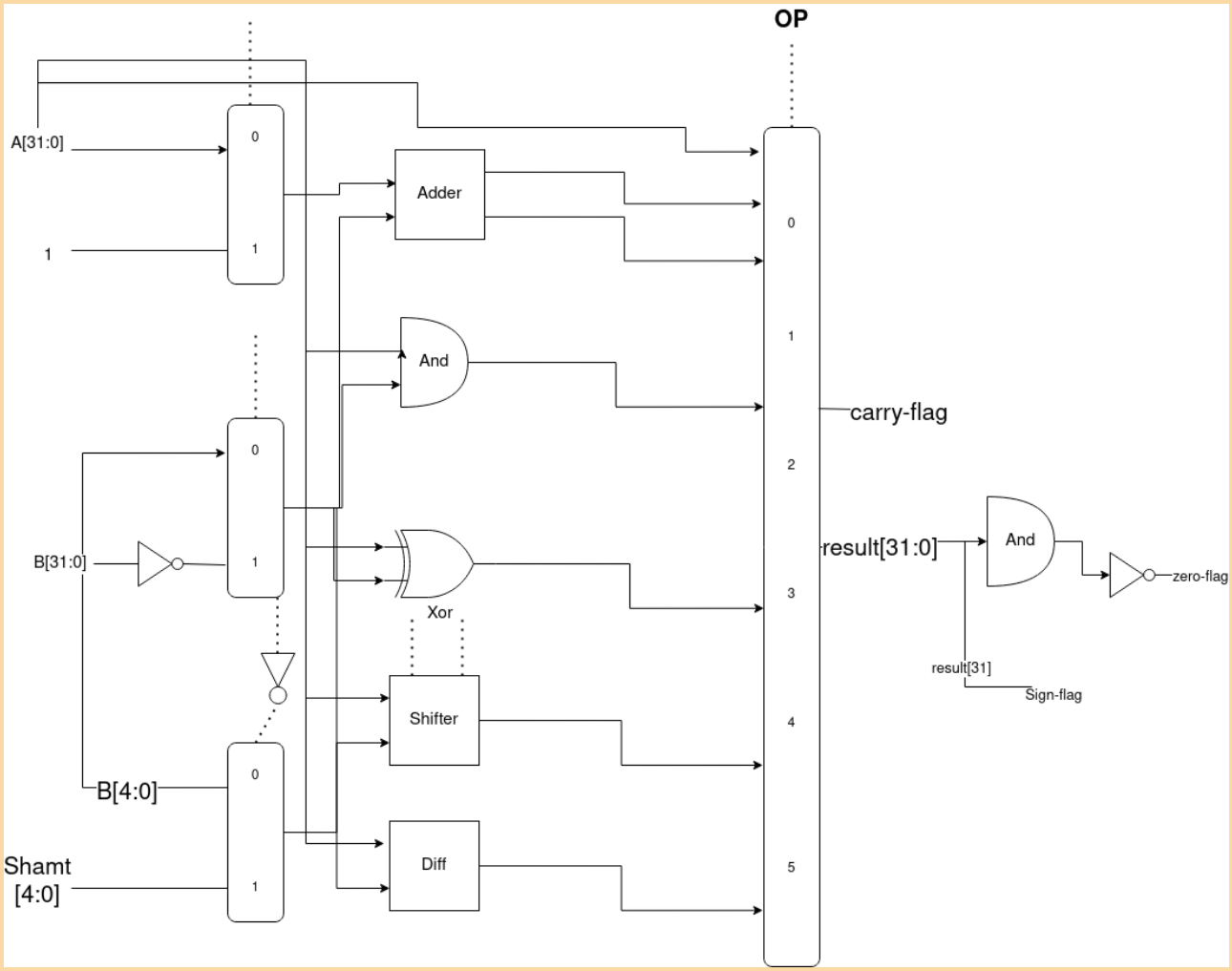
## Lookup table for Datapath Control:

| Op | opcode | reg Dest | RegWrite | ALUsrc | MemRead | MemWrite | MemToReg | Lbl Sel | Jump Sel |
|---|---|---|---|---|---|---|---|---|---|
| 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 |
| compi | 001001 | 00 | 1 | 1 | 0 | 0 | 00 | X | X |
| and | 000000 | 00 | 1 | 1 | 0 | 0 | 00 | X | X |
| xor | 000000 | 00 | 1 | 1 | 0 | 0 | 00 | X | X |
| shll | 000000 | 00 | 1 | 1 | 0 | 0 | 00 | X | X |
| shrl | 000000 | 00 | 1 | 1 | 0 | 0 | 00 | X | X |
| shllv | 000000 | 00 | 1 | 1 | 0 | 0 | 00 | X | X |
| shrlv | 000000 | 00 | 1 | 1 | 0 | 0 | 00 | X | X |
| shra | 000000 | 00 | 1 | 1 | 0 | 0 | 00 | X | X |
| shrav | 000000 | 00 | 1 | 1 | 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 |

# Datapath Diagram:



## ALU Diagram:




## Running Bubble Sort:

To test the given architecture, Bubble Sort has been implemented. The semantics for functional call and stack remain the same from MIPS with similar register convention mentioned in the source assembly code. A pseudocode for the implemented algorithm is as follows:

```
BubbleSort(list):
   for all elements of list
      if list[i] > list[i+1]:
         swap(list[i], list[i+1])

   return list
```

The MIPS code for the given algorithm is as follows:

```
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

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

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

      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, inc_j
      bz $13, inc_j
      bl swap           # swap if arr[j] > arr[j + 1]

inc_j:
      addi $9, 1        # j = j + 1
      b for_j

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

```
end_for_j:
      addi $8, 1         # i = i + 1
      b for_i

end_for_i:
      xor $16, $16
      addi $16, 1
```

And corresponding binary code is given as:

memory_initialization_radix=2;
memory_initialization_vector=
00000010100101000000000000000011,
00000010101101010000000000000011,
00100010101000000000000000001010,
00000001000010000000000000000011,
00000001001010010000000000000011,
00000001010010100000000000000011,
00000001010010000000000000000001,
00000001011101010000000000000101,
00000001010010110000000000000001,
00100001010000000000000000000001,
11000101010000000000000000011110,
00000001001010010000000000000011,
00000001011010110000000000000011,
00000001011010010000000000000001,
00000001011010100000000000000001,
11000101011000000000000000010111,
00000001100011000000000000000011,
00000001100010010000000000000001,
00000001100000000001000000001011,
00000001100101000000000000000001,
01000001100011010000000000000000,
00000000100001000000000000000011,
00000000100011000000000000000001,
00100001100000000000000000000100,
01000001100011100000000000000000,
00000000101001010000000000000011,
00000000101011000000000000000001,
00000001111011100000000000000101,

```
00000001101011110000000000000001,
11000001101000000000000000000010,
11000101101000000000000000000001,
10101100000000000000000000100010,
00100001001000000000000000000001,
10100000000000000000000000001100,
01000000100100100000000000000000,
01000000101100110000000000000000,
01100000101100100000000000000000,
01100000100100110000000000000000,
10000011111000000000000000000000,
00100001000000000000000000000001,
10100000000000000000000000000101,
00000010000100000000000000000011,
00100010000000000000000000000001,
00000000000000000000000000000000;
```