**LO-RISC**

**Learning Optimized Reduced Instruction Set Computer**

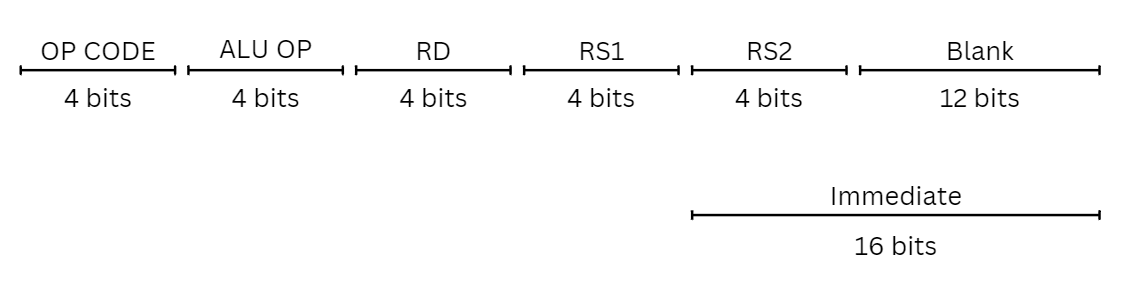
A minimal Instruction Set Architecture designed for speed and simplicity

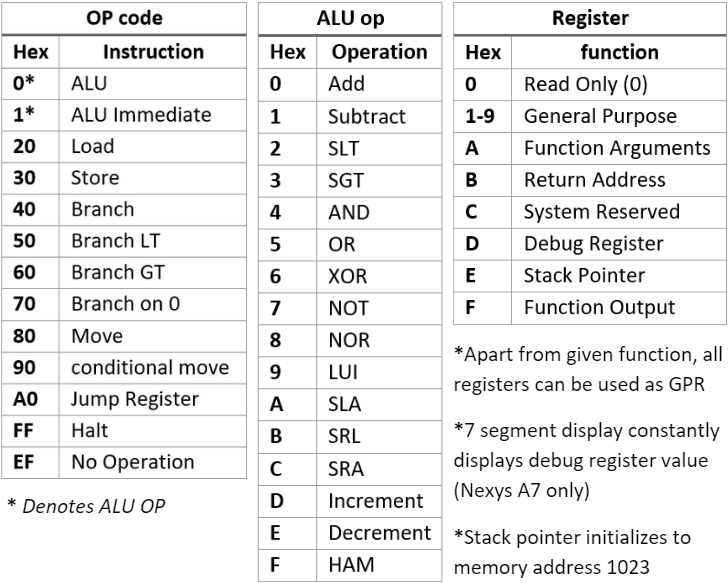
**BY:**

Raaja Das (22CS30043)  
Priyanshu Gaurav (22CS10083)

COA lab Final Project

**Instruction Set**

Each Instruction in LO-RISC is 32 bits long, divided into 8, 4-bit chunks representable by hexadecimal.  
**OP CODE** is connected to the control unit and essentially selects the instruction while **ALU OP** is directly connected to the ALU and selects its function. Three register operand addresses (4-bit each) follow, followed by 12 trailing blank (0) bits. For Immediate instructions **RS2** and Blank Bits are replaced by the **immediate** field.



**Examples:**

add $1 $2 $3 ► 0x00123000 ► R1 = R2 + R3

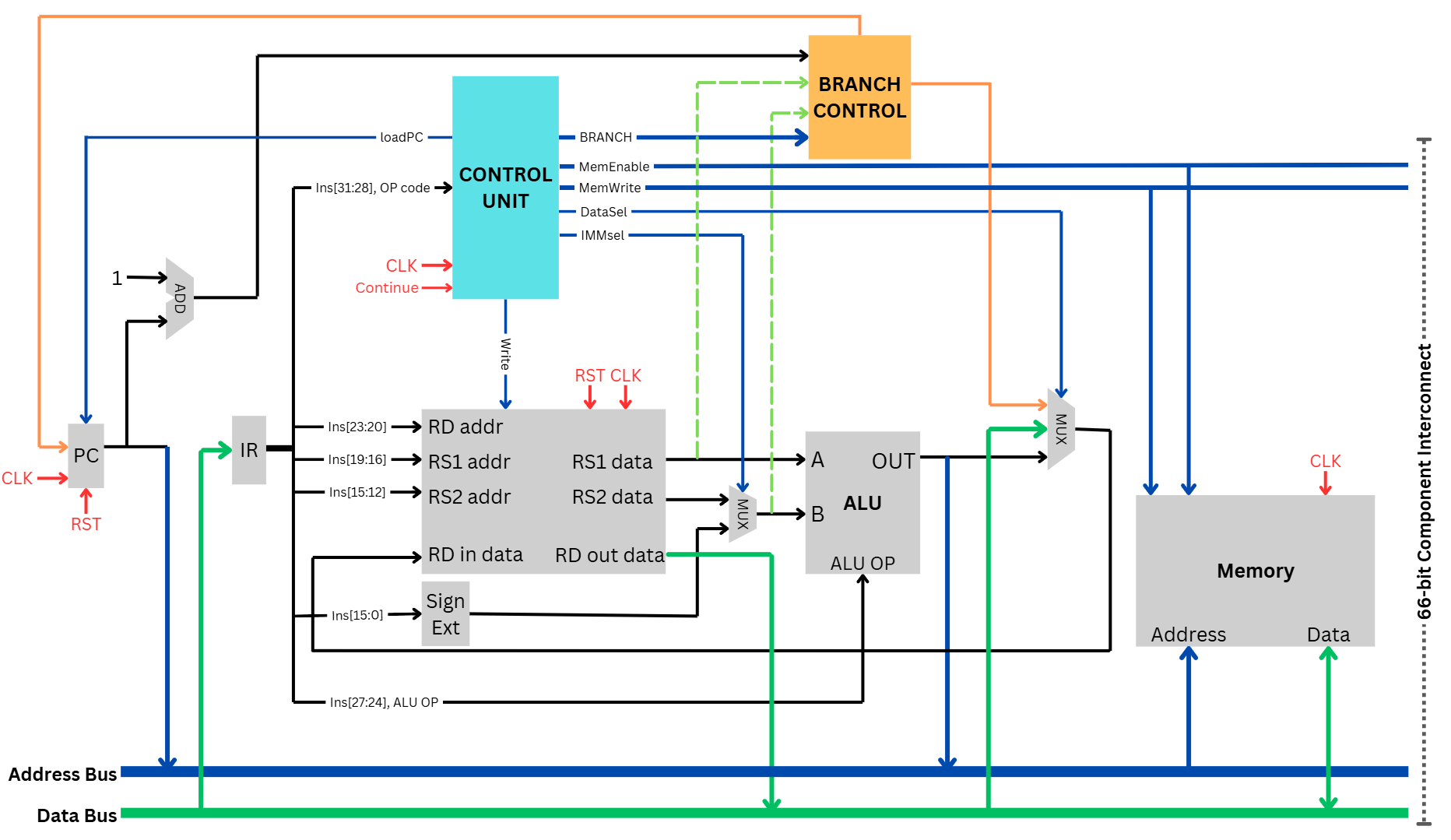
slai $5 $7 1 ► 0x1A570001 ► R5 = R7 << 1

ld $3 8($6) ► 0x20360008 ► R3 = Mem[R6+8]

br #10 ► 0x4000000A ► PC = PC + 10

bmi $5 32 ►0x50050020 ► PC <= PC + 30 if (R5 < 0)

**Data Path**



The CPU uses Von Neuman Architecture for memory access. Instructions and Data share unified system memory.

32-bit **Address** and **Data** buses along with control signals, **Memory Enable** and **Memory Write Enable** make up the 66-pin (32+32+1+1) **Component Interconnect (CI)**. Components such as memory or I/O devices can be connected to the CPU via this interconnect. Tri-State buffers are used to control access to the address and data buses. A **UART I/O module** is used for Input/Output through serial UART and is connected to the CPU via the component interconnect.

All Verilog modules can be found in **‘Verilog Assets/Sources’** directory.

**Performance**

All Instructions take **3 cycles** except load which takes **4 cycles** due to additional writeback state.

**Max Validated Clock Speed:** 100 Mhz

**Max Clock Cycles per Instruction:** 4

**Min System Memory:** 1KB

**Max System Memory:** 16GB

**System Memory in given configuration:** 4 KB

**UART serial baud rate:** 115200

**Max Serial data transfer speed:** 92.16 Kbits/sec

**Assembly**

A LO-RISC assembly file is composed of two parts, data section preceded by .data and instruction section preceded by .text along with macros of the form: num = 4242

**Data:**  
Data entries are composed of the label followed by data type and the corresponding data. Data is placed in data memory sequentially in the order of data entries.

myvar: .int 42

myarr: .arr {3,4,5,7}

mychar: .char 'k'

mystr: .str "Hello"

**Instructions** *(case insensitive)***:**

**Labels** (Eg: Label\_1:) denote specific points in a program used for calculating the effective address for branching.

**The following instructions are available:**

**a)** Arithmetic and logic instructions: ADD, SUB, AND, OR, XOR, NOR, NOT, SL, SRL, SRA,

INC, DEC, SLT, SGT, LUI, HAM. There are corresponding immediate addressing versions

with a suffixing “I” (like ADDI, SUBI, etc.). Assume that all shift instructions can have

either 0 (no shift) or 1 (1-bit shift) as operand. Some example uses are as follows:

add $1 $2 $3 #R1 = R2 + R3

slai $5 $7 1 #R5 = R7 << 1

**b)** Load and store instructions: LD, ST (all load and stores are 32-bits) and use register indexed

addressing (any of the registers R1..R15 can be used). Some example uses are as follows:

ld $1 myvar #r3 = Mem[location of myvar]

ld $3 myarr($2) #R3 = Mem[Location of myarr[$2]]

**c)** Branch instructions: BR, BMI, BPL, BZ. Some example uses are as follows:

br loop #branch to loop

bz $5 lab #Branch to lab if R5 = 0

bmi $5 lab #branch to lab if R5 < 0

bpl $5 lab #branch to lab if R5 > 0

jr $ra #branch to address at RA

**d)** Register to register transfer: MOVE, CMOV. Some example uses are as follows:

move $4 $6 #R4 = R6

cmov $1 $2 $3 #R1 = (R2 < R3) ? R2 : R3

**Pseudo instructions**

**a)** LA loads data memory address into register

la $1 program

#equivalent to:

lui $1 (upper 16 bits of program)

ori $1 $1 (lower 16 bits of program)

**b)** LI loads value into register

li $1 12 #equivalent to: addi $1 $0 12

**b)** JAL to be used for function calls

jal func

# equivalent to:

addi $ra $0 (current address)

br func

**\*See Appendix for Usage and Examples**

UART I/O

An I/O module is used for Serial communications through UART.

* A Serial terminal application such as **Tera Term** or **Minicom** can be used for interacting with the system.
* It operates at a baud rate of 115200 leading to a max effective bandwidth of 92.16 Kbits/sec.
* Memory addresses 4096 and 4097 are reserved for this module. It has two registers, the **command register** and the **data register** at Memory addresses **4096** and **4097** respectively.
* The module starts at idle state, the registers can only be written to in this state.
* The module is controlled through the command register. When command register is set to:
  + 1, it transmits the first byte of data register as an ASCII code
  + 2, it listens for an ASCII transmission and puts the transmission into the data register
  + 3, it transmits the entire data register as a decimal integer
* After the transmission/reception is complete, the command register resets to 0 and the module is idle again.
* The command register can be polled to check the state of transmission/reception.
* Standard subroutines for I/O operations are provided in **‘Programs/lib’** directory.

Appendix

* The Assembler is present as portable executable (**Programs/assembler.exe**) as well as python source code (**Programs/source/assembler.py**). Usage is as follows:
* ./assembler program.s or python source/assembly.py program.s

Outputs will be **program.coe** for loading into memory

* Example programs are present in **‘Programs’** directory