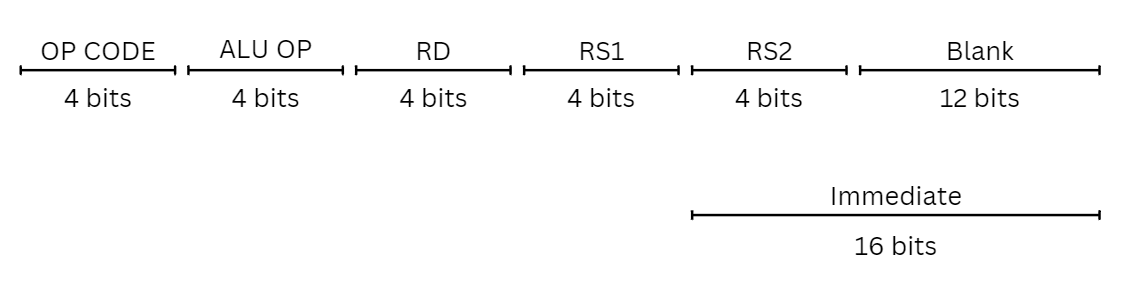
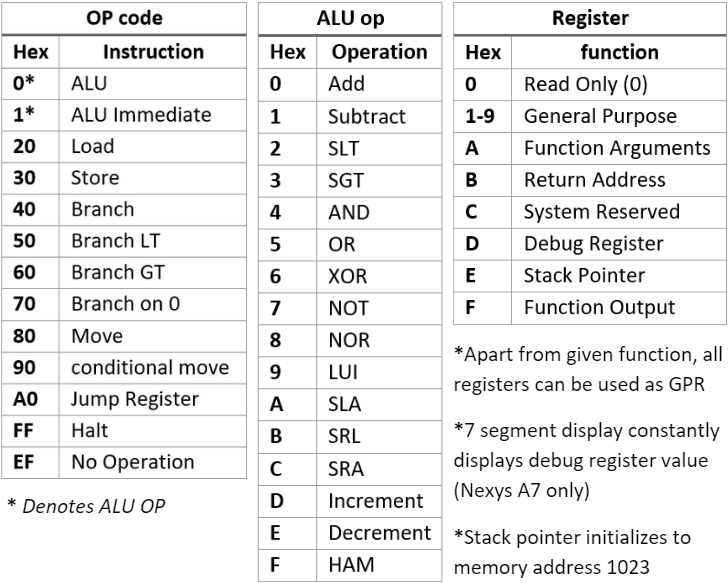
**KGP RISC**

**Instruction Set**

Each Instruction in KGP 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)

**Assembly**

A KGP RISC assembly file is composed of two parts, data section preceded by .data and instruction section preceded by .text

**Data:**  
data entries are composed of the label followed by data type and the corresponding data

myvar: .int 42

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

mychar: .char 'k'

mystr: .str "Hello World"

**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 less\_than #Branch to less\_than 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

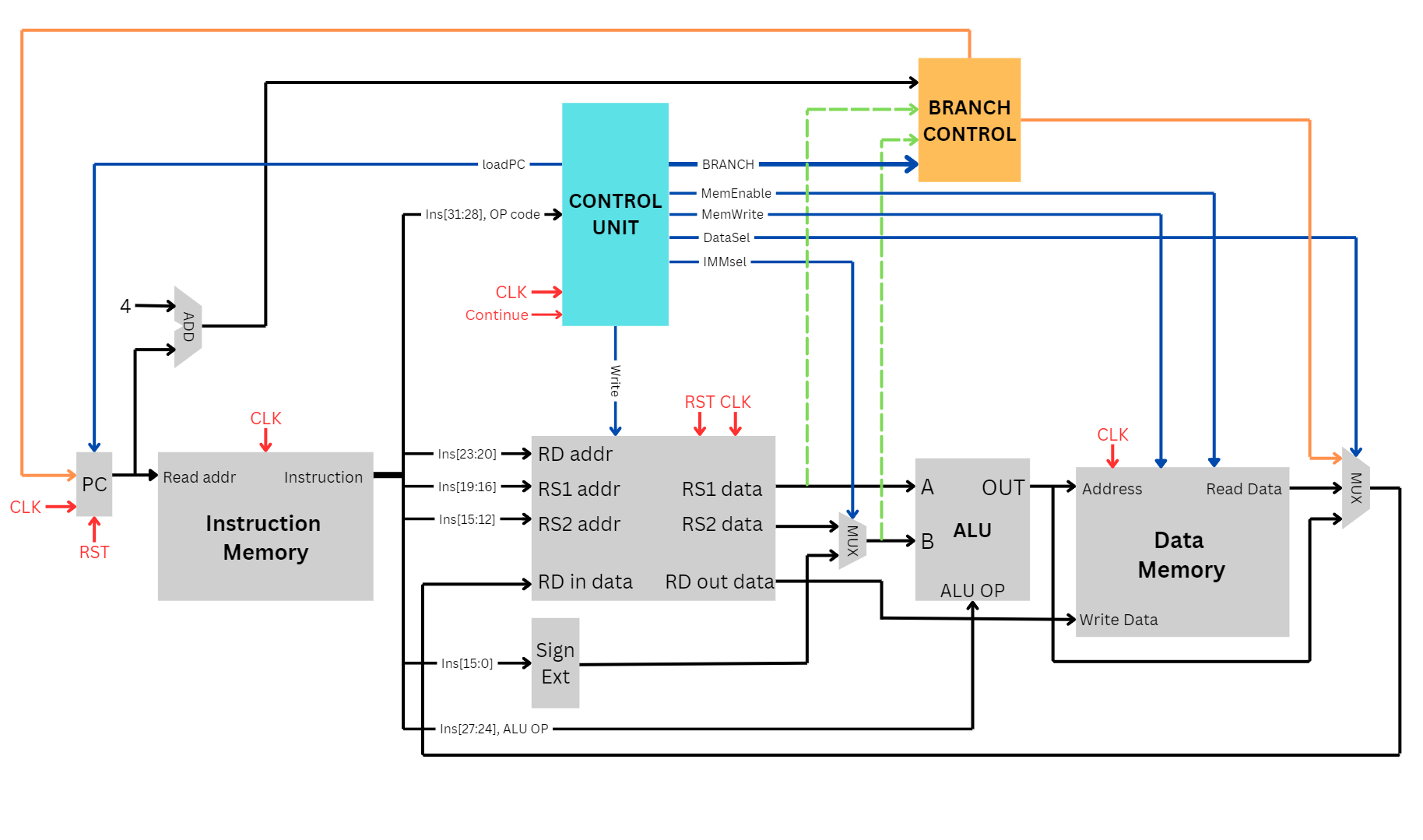
# equivalent to:

addi $ra $0 (current address)

br func

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

**Data Path**



**Performance**

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

**Max Clock Cycles per Instruction:** 4

\* Timing Details (Valid for Nexys A7 only)

**CPU only**

* **Max propagation delay:** 5.494 ns
* **Max theoretical Clock Speed:** 91.01 Mhz
* **Max Validated Clock Speed:** 83.33 Mhz

**With 7 segment display**

* **Max propagation delay:** 11.334 ns
* **Max theoretical Clock Speed:** 44.12 Mhz
* **Max Validated Clock Speed:** 41.66 Mhz

Appendix

* All Verilog modules can be found in **‘Verilog Assets/Sources’** directory
* The Assembler is present as portable executable (assembler.exe) as well as python source code (assembler.py) in **‘Programs’** directory. Usage is as follows:
* ./assembler program.s or python assembly.py program.s

Outputs will be **inst.coe** for instruction memory and **data.coe** for data memory

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