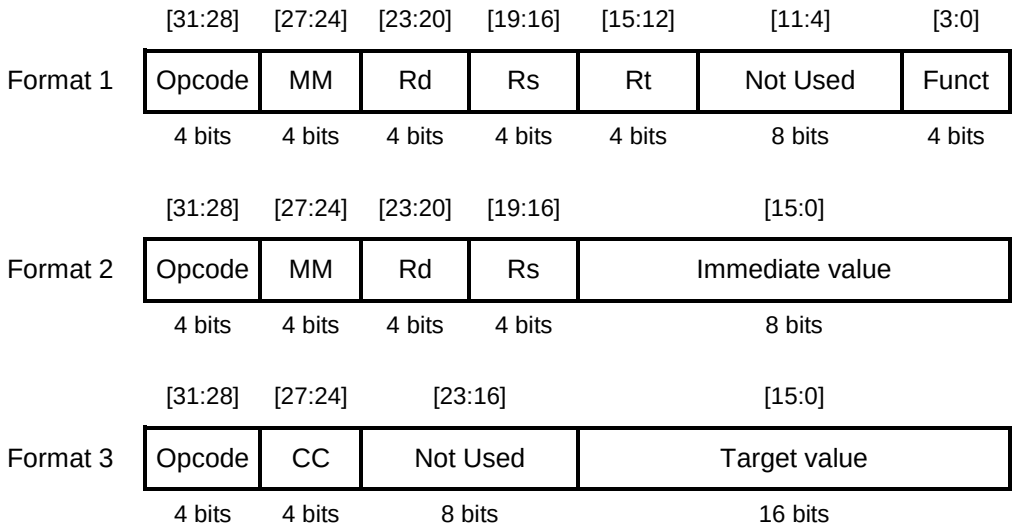
SISC Instruction Set

Overview:



Instruction	*Operation*	*Format*	*Opcode*	*MM/CC*	*Funct*
NOP	None.	3	0000 *(0x0)*	N/A	N/A
ADI	Rd ← Rs + (sign ext.) imm	2	1000 *(0x8)*	1000 *(0x8)*	N/A
ADD	Rd ← Rs + Rt	1	1000 *(0x8)*	0000 *(0x0)*	0001 *(0x1)*
SUB	Rd ← Rs – Rt	1	1000 *(0x8)*	0000 *(0x0)*	0010 *(0x2)*
NOT	Rd ← ~Rs	1	1000 *(0x8)*	0000 *(0x0)*	0100 *(0x4)*
OR	Rd ← Rs ∨ Rt	1	1000 *(0x8)*	0000 *(0x0)*	0101 *(0x5)*
AND	Rd ← Rs ∧ Rt	1	1000 *(0x8)*	0000 *(0x0)*	0110 *(0x6)*
XOR	Rd ← Rs ⊕ Rt	1	1000 *(0x8)*	0000 *(0x0)*	0111 *(0x7)*
RTR	Rd ← Rs ⩾ [Rt]	1	1000 *(0x8)*	0000 *(0x0)*	1000 *(0x8)*
RTL	Rd ← Rs ⩽ [Rt]	1	1000 *(0x8)*	0000 *(0x0)*	1001 *(0x9)*
SHR	Rd ← Rs >> [Rt]	1	1000 *(0x8)*	0000 *(0x0)*	1010 *(0xA)*
SHL	Rd ← Rs << [Rt]	1	1000 *(0x8)*	0000 *(0x0)*	1011 *(0xB)*
BRA	PC ← target (if taken)	3	0100 *(0x4)*	CC	N/A
BRR	PC ← (PC+1) + offset (if taken)	3	0101 *(0x5)*	CC	N/A
BNE	PC ← target (if taken)	3	0110 *(0x6)*	CC	N/A
BNR	PC ← (PC+1) + offset (if taken)	3	0111 *(0x7)*	CC	N/A
LDX	Rd ← Mem[Rs + imm]	2	0001 *(0x1)*	1000 *(0x8)*	N/A
LDA	Rd ← Mem[imm]	2	0001 *(0x1)*	0000 *(0x0)*	N/A
LDP	Rd ← Mem[Rs + imm] Rs ← Rs + imm	2	0001 *(0x1)*	1001 *(0x9)*	N/A
LDR	Rd ← Mem[Rs] Rs ← Rs + imm	2	0001 *(0x1)*	0001 *(0x1)*	N/A

STX	Mem[Rs + imm] ← Rd	2	0010 *(0x2)*	1000 *(0x8)*	N/A
STA	Mem[imm] ← Rd	2	0010 *(0x2)*	0000 *(0x0)*	N/A
STP	Mem[Rs + imm] ← Rd Rs ← Rs + imm	2	0010 *(0x2)*	1001 *(0x9)*	N/A
STR	Mem[Rs] ← Rd Rs ← Rs + imm	2	0010 *(0x2)*	0001 *(0x1)*	N/A
SWP	Rs ← Rd Rd ← Rs	2	0011 *(0x3)*	N/A	N/A
HLT	Execution stops.	3	1111 *(0xF)*	N/A	N/A

Details/Notes:

R0: Register zero (R0) is a dummy register designation. If used as the source register, a 32-bit all zero operand value is generated. If used as the destination register, any instruction results are discarded.

PC: Since memory addressing is to the word, the PC is incremented by 1 after instruction fetch. This can also be seen in the BRR and BNR instructions where the incremented PC is added to the offset if the branch is taken.

CC:

Upon executing ADD and SUB instructions, the ALU outputs a four-bit status code that describes the result of the addition or subtraction, and are saved in the status register (statreg.v) for use in branch instructions. The four bits are referred to as C, V, N, and Z:

- Bit 3 (Carry): This bit of the status register is set high when the operation generates a carry or borrow.
- Bit 2 (oVerflow): This bit is set high when the addition or subtraction causes an overflow or underflow.
- Bit 1 (Negative): This bit is set high when the result of the addition or subtraction *should* be negative. Note that if the addition of two positive numbers overflows, this bit will **not** be set high, because the result *should* be positive. Similarly, if the addition of two negative numbers underflows, it **will** be set high, though the ALU output will be a positive number.
- Bit 0 (Zero): This bit is set high when the addition or subtraction results in zero.

BRA, BRR, BNE AND BNR:

For the branch instructions, the CC field of the instruction in combination with the current status register contents determine if the branch is taken or not. If the branch is not taken, the PC is incremented by 4 to point to the next instruction following the branch instruction. If the branch is taken, the PC is loaded with the branch target address.

In the case of logical-positive branches (BRA & BRR), the branch is taken when *any* of the instruction's CC bits are set to 1 AND the corresponding bits in the status register are also 1 (i.e. (CC & STAT) != 0).

To illustrate this, consider the case when we wish to add two registers, then branch if the result of that addition is less than or equal to zero. We would use a BRA or BRR instruction with the CC = 0011, because we want to take the branch if *either* the negative status bit or the zero status bit is set in the status register.

Similarly, for logical-negative branches (BNE & BNR), the branch is *not* taken if the status register bit is set high for any CC bit set to 1 (i.e. (CC & STAT) == 0).

If, in the above example, we wished instead to take the branch whenever the addition was greater or equal to zero, we would use a BNE or BNR instruction with the CC = 0010. That way, the only case in which we will *not* take the branch is when the ALU sets the negative status bit and we will therefore branch for all ALU outputs that are zero or greater.

Lastly, it should be noted that a BNE or BNR instruction with the CC = 0000 is effectively an unconditional branch.

IMM:

In the ADD immediate instruction, the 16-bit immediate value is sign-extended to create a 32-bit operand to be added to Rs. If, for some reason, you need an immediate value that does not fit in 16 bits, use a temporary register to hold the top 16 bits, shift left 16 places, and add to the lower 16 bits.

Because the PC is 16 bits, the 16-bit immediate value is sufficient to address all instruction memory. For the relative branch instructions, the 16-bit immediate value is a signed value. However, for the load and store instructions where the immediate value is added to the source register value, the immediate value is sign extended to create a 32-bit operand to be added to Rs in the ALU. The ALU output is then truncated to 16 bits to form the data memory address.

LDP, LDR, STP and STR Instructions:

For the LDP and STP instructions, if the IMM value is +1/-1, the effective addressing mode is +/-(Rs), i.e. pre-increment or pre-decrement. For the LDR and STR instructions, if the IMM value is +1/-1, the effective addressing mode is (Rs)+/-, i.e. post-increment or post-decrement.

Pseudo Instructions:

There are many common functions that are accomplished by the use of specific opcodes, register, and immediate values. Some of them are as follows.

Pseudo Instruction	Real Instruction	Comments
CLR Rd	ADD Rd,R0,R0	Clear a register
LRI Rd,imm	ADI Rd,R0,imm	Load a register with sign extended immediate value
INC Rd	ADI Rd,Rd,0x0001	Increment a register
DEC Rd	ADI Rd,Rd,0xFFFF	Decrement a register
STI Rd,Rs	STX Rd,Rs,0x0000	Store indirect; M[Rs] ← Rd
LDI Rd,Rs	LDX Rd,Rs,0x0000	Load indirect; Rd ← M[Rs]
NEG Rd	SUB Rd,R0,Rd	Negate Rd; i.e. Rd ← -Rd