SISA (*Simple Instruction Set Architecture*), un lenguaje RISC de 16 bits:

- 25 instrucciones de tamaño fijo de 16 bits.

  De cálculo (Aritmético lógicas y de Comparación): `AND`, `OR`, `XOR`, `NOT`, `ADD`, `SUB`, `SHA`, `SHL`, `CMPLT`, `CMPLE`, `CMPEQ`, `CMPLTU`, `CMPLEU`, `ADDI`.

  De movimiento (Inmediato, Memoria y Entrada/Salida): `MOVI`, `MOVHI`, `LD`, `LDB`, `ST`, `STB`, `IN`, `OUT`

  De ruptura de secuencia: condicional, `BZ` y `BNZ`, e incondicional, `JALR` (para llamadas y retorno de subrutinas).

- Estado del computador:

  PC (Program Counter),

  8 registros de 16 bits (R0, R1,... , R7),

  Memoria de datos e instrucciones de 2^16 bytes accesible a byte y a word (2 bytes).

  Espacio de direcciones de E/S con 256 puertos de entrada y 256 de salida de 16 bits cada uno.

- Usamos el siguiente convenio: los bits dentro de un campo de bits (Registro, Instrucción, Byte o Word en memoria, o campos de menos bits dentro de estos) se numeran siendo el bit 0, el de la derecha, el bit de menor peso en las interpretaciones como número natural (binario) o entero (Ca2, complemento a dos).

- Los operandos fuente y destino para las instrucciones de cálculo son siempre registros (R0, R1,... R7), excepto en la instrucción `ADDI`, en la que el segundo operando es un inmediato de 6 bits en Ca2.

- Instrucciones de "*load*" (`LD`, `LDB`) y "*store*" (`ST`, `STB`) para copiar words y bytes entre memoria y registros. El modo de direccionamiento para las cuatro instrucciones es "registro base más un desplazamiento de 6 bits en Ca2". Los accesos a words (`LD` y `ST`) son alineados a palabras (2 bytes) con formato de almacenamiento *Little Endian*: la dirección es par (el hardware pone a cero el bit de menor peso de la dirección calculada sumando `Ra` más la extensión de signo del desplazamiento de 6 bits, `N6`) y apunta al byte de menor peso de los dos bytes que forman la palabra en memoria. En la dirección siguiente (la dirección par + 1, la dirección impar) está el byte de más peso. No obstante, no se detecta ni se produce una excepción en caso de intento de acceso no alineado, si la dirección `Ra + SE(N6)` es impar simplemente se accede a la dirección que resulta de poner a 0 el bit de menor peso de la dirección calculada (para obtener el byte de menor peso) y a la dirección siguiente (para el de más peso).

- Tipos de datos. Los operados fuente de las instrucciones de cálculo son palabras de 16 bits. No se detecta si el resultado de las operaciones aritméticas es o no representable en 16 bits ni para la interpretación como números naturales en binario ni para números enteros en Ca2 (no detección de Carry ni Overflow). Hay dos instrucciones diferentes para la división por potencias de dos de naturales y de enteros, en las que tampoco se detecta si el resultado es representable o no en 16 bits. Hay instrucciones diferentes para la comparación de naturales y para la de enteros y en estas instrucciones el booleano resultante de la comparación (0 para FALSO y 1 para VERDADERO) siempre es correcto (aunque haciendo una resta de los operandos se hubiera producido carry y/o overflow). El segundo operando de la instrucción `ADDI` es un entero de 6 bits en Ca2, así como el desplazamiento para el cálculo de la dirección de memoria en las instrucciones "*load*" (`LD`, `LDB`) y "*store*" (`ST`, `STB`). El desplazamiento en las instrucciones de ruptura de secuencia condicional relativa al PC es de 8 bits en Ca2, que se multiplica por 2 antes de sumarlo al PC actualizado. El PC se actualiza a PC+2 durante la búsqueda de cada instrucción.

- Resumen de las 25 instrucciones SISA:

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0	**Mnemonic**	**Example Assembler**	**Example Operation** (in pseudo C language)
0 0 0 0	a a a	b b b	d d d	f f f	AND, OR, XOR, NOT, ADD, SUB, SHA, SHL	NOT Rd, Ra ADD Rd, Ra, Rb	Rd=~Ra; Rd=Ra+Rb;
0 0 0 1	a a a	b b b	d d d	f f f	CMPLT, CMPLE, -, CMPEQ, CMPLTU, CMPLEU, -, -	CMPLE Rd, Ra, Rb CMPLEU Rd, Ra, Rb	if(Ra_s<Rb_s)Rd=1 else Rd=0; if(Ra_u<Rb_u)Rd=1 else Rd=0;
0 0 1 0	a a a	d d d	n n n n n n		ADDI	ADDI Rd, Ra, N6	Rd=Ra+SE(N6);
0 0 1 1	a a a	d d d	n n n n n n		LD	LD Rd, N6(Ra)	Rd=Mem_w[(Ra+SE(N6))&(~1)]
0 1 0 0	a a a	b b b	n n n n n n		ST	ST N6(Ra), Rb	Mem_w[(Ra+SE(N6))&(~1)]=Rb
0 1 0 1	a a a	d d d	n n n n n n		LDB	LDB Rd, N6(Ra)	Rd=SE(Mem_b[Ra+SE(N6)]);
0 1 1 0	a a a	b b b	n n n n n n		STB	STB N6(Ra), Rb	Mem_b[Ra+SE(N6)]=Rb<7..0>;
0 1 1 1	a a a	d d d	x x x x x x		JALR	JALR Rd, Ra	PC=PC+2;tmp=Ra&(~1);Rd=PC;PC=tmp
1 0 0 0	a a a	0	n n n n n n n n		BZ	BZ Ra, N8	PC=PC+2;if(Ra==0)PC=PC+SE(N8)*2
		1			BNZ	BNZ Ra, N8	PC=PC+2;if(Ra!=0)PC=PC+SE(N8)*2
1 0 0 1	d d d	0	n n n n n n n n		MOVI	MOVI Rd, N8	Rd=SE(N8);
	a a a d d d	1			MOVHI	MOVHI Rd, N8	Rd<15..8>=N8;
1 0 1 0	d d d	0	n n n n n n n n		IN	IN Rd, N8	Rd=Input[N8];
	a a a	1			OUT	OUT N8, Ra	Output[N8]=Ra;
1 0 1 1 1 1 x x	x x x x x x x x x x x x				Future Extensions		

SISA (Simple Instruction Set Architecture) Reference Sheet

Assembler	Operation (in pseudo C language)[1]	Foot Note	Format	Op. Code	e	F	Name
AND Rd, Ra, Rb	Rd=Ra&Rb	2	3R	0	-	0	Bitwise logical And
OR Rd, Ra, Rb	Rd=Ra|Rb	2	3R	0	-	1	Bitwise logical Or
XOR Rd, Ra, Rb	Rd=Ra^Rb	2	3R	0	-	2	Bitwise logical Xor
NOT Rd, Ra	Rd=~Ra	2	3R	0	-	3	Bitwise logical Not
ADD Rd, Ra, Rb	Rd=Ra+Rb	2	3R	0	-	4	Add
SUB Rd, Ra, Rb	Rd=Ra-Rb	2	3R	0	-	5	Subtraction
SHA Rd, Ra, Rb	$Rd=Ra_s$*2**Rb<4..0>$_s$	2, 3, 4	3R	0	-	6	Shift arithmetic
SHL Rd, Ra, Rb	$Rd=Ra_u$*2**Rb<4..0>$_s$	2, 3, 4	3R	0	-	7	Shift logic
CMPLT Rd, Ra, Rb	if(Ra_s<Rb_s)Rd=1 else Rd=0	2	3R	1	-	0	Compare if less than
CMPLE Rd, Ra, Rb	if(Ra_s<=Rb_s)Rd=1 else Rd=0	2	3R	1	-	1	Compare if less or equal
CMPEQ Rd, Ra, Rb	if(Ra==Rb)Rd=1 else Rd=0	2	3R	1	-	3	Compare if equal
CMPLTU Rd, Ra, Rb	if(Ra_u<Rb_u)Rd=1 else Rd=0	2	3R	1	-	4	Compare less than for unsigned
CMPLEU Rd, Ra, Rb	if(Ra_u<=Rb_u)Rd=1 else Rd=0	2	3R	1	-	5	Compare if less or equal for unsigned
ADDI Rd, Ra, N6	Rd=Ra+SE(N6)	2, 5, 6	2R	2	-	-	Add immediate
LD Rd, N6(Ra)	Rd=Mem_w[(Ra+SE(N6))&(~1)]	2, 5, 6, 8	2R	3	-	-	Load
ST N6(Ra), Rb	Mem_w[(Ra+SE(N6))&(~1)]=Rb	2, 5, 6, 8	2R	4	-	-	Store
LDB Rd, N6(Ra)	Rd=SE(Mem_b[Ra+SE(N6)])	2, 5, 6, 7	2R	5	-	-	Load byte
STB N6(Ra), Rb	Mem_b[Ra+SE(N6)]=Rb<7..0>	2, 4, 5, 6, 7	2R	6	-	-	Store byte
JALR Rd, Ra	PC=PC+2;tmp=Ra&(~1);Rd=PC;PC=tmp	9	2R	7	-	-	Jump and link register
BZ Ra, N8	PC=PC+2;if(Ra==0)PC=PC+SE(N8)*2		1R	8	0	-	Branch on zero
BNZ Ra, N8	PC=PC+2;if(Ra!=0)PC=PC+SE(N8)*2		1R	8	1	-	Branch on not zero
MOVI Rd, N8	Rd=SE(N8)	2, 5, 6	1R	9	0	-	Move immediate
MOVHI Rd, N8	Rd<15..8>=N8	2, 4	1R	9	1	-	Move immediate high
IN Rd, N8	Rd=Input[N8]	2, 5	1R	A	0	-	Input
OUT N8, Ra	Output[N8]=Ra	2, 5	1R	A	1	-	Output

1) All operands and results are of 16 bits except the power of 2 in SHA and SHL that are a 5 bit signed integer in two-complement. Ra, Rb, Rd is the content of source general registers a and b and destination register d (a, b, d $\in$ {0, 1, …, 7}). Rk_s is the content of register k considered as an signed integer (two-complement). Rk_u is the content of register k considered as an unsigned integer (binary). When no subindex s or u appears is because the operation is the same for signed or unsigned integer (as add or sub) or because is not required any interpretation (as for bit-wise logical operations).

2) In addition to the specified action, PC is updated to point next instruction: PC=PC+2.

3) ** represents the operator exponential or power of.

4) Rk<x..y>$_z$ is the bit field from bit x down to bit y of register k interpreted as signed integer (with subindex z=s) or as unsigned integer (with subindex z=u).

5) N6 or N8 is a number coded in the instruction in a field of 6 or 8 bits respectively. Depending on the instruction the field is interpreted as unsigned or signed integer. It is expressed in assembler language as decimal or hexadecimal,

6) SE(X) is the sign extension of the field of bits X to form a 16 bits operand (X is interpreted as a signed integer in two-complement).

7)Mem_b[k] is the byte of Memory with address k (the 64 KBytes Memory has a byte address space with 16 bit addresses).

8) Mem_w[k] is the word (16 bits) of Memory with byte addresses k and k+1, for k even (the address of the 8 less significant bits of the word: little endian format).

9) tmp is a temporal storage for the jump target address (always an even address). Rd and Ra may specify the same register: the jump targed address (the old register value) is temporaly stored before the new register value (the updated PC / the address of the instruction following the JALR instruction / the return address) is assigned.

Instruction Formats

Format (15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0)	Instruction Mnemonic
3R (Three register format):	
c c c c | a a a | b b b | d d d | f f f	AND, OR, XOR, NOT, ADD, SUB, SHA, SHL CMPLT, CMPLE, -, CMPEQ, CMPLTU, CMPLEU, -, -
2R (Two register format):	
c c c c | a a a | b b b | n n n n n n	ST, STB
c c c c | a a a | d d d | n n n n n n	ADDI, LD, LDB, JALR
1R (One register format):	
c c c c | a a a | e | n n n n n n n n	BZ, BNZ, OUT
c c c c | d d d | e | n n n n n n n n	MOVI, MOVHI, IN