The NIISC Instruction Set Architecture

Revision 7

- 16-Bit internal architecture
- 16-Bit and 8-Bit registers
- Byte and Word operations
- 8-Bit and 16-Bit Signed and Unsigned Arithmetic in Binary and Decimal, including Multiply and Divide.

Table of Contents

0 NIISC overview

This instruction set is meant to be a middle ground between overly generalized architectures such as RiSC-16[1] and overly specific architectures such as x86. NIISC is a 16 register, 16-bit computer. The word size is 16 bits. All busses are word size; however, the memory addresses are byte-addresses (i.e., address 0 corresponds to the first byte of main memory, address 1 corresponds to the second byte of memory, etc.). In each instruction 2 bits are dedicated to the instruction mode and 4 bits are dedicated to the opcode. Big endian will be used throughout the instruction set. Because the architecture is intended for educational purposes, the instructions will not be optimized to minimize the size of the semiconductor die.

The mode bits serve no purpose apart from that to organize the opcodes into 4 categorizes such that a 2-bit decoder can be used when implementing this ISA.

The naming convention of the instructions is guided by the final letter in the instruction, which corresponds to the register type that the instruction operates on: 'B' signifies an 8-bit register, while 'W' denotes a 16-bit register. This distinction significantly influences the outcome of the operations. For instance, when handling signed numbers, a signed 8-bit value of **-**128, when interpreted in the context of a 16-bit register, is treated as **+**128. For instructions that don’t end in ‘B’, ‘W’, it means that either register is accepted, as in that it uses 16-bit registers and that padding the 8-bit register with zeros will not affect the result. Similarly For instructions that don’t end in ‘U’ or ‘S’ means that inputting a signed or unsigned number will not alter the result of the operation.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Formats:

RR	Mode	Opcode	Reg a	Reg b	0
I	Mode	Immediate			
R	Mode	Opcode	Reg a	0	
N	Mode	Opcode	0		

1 Instructions

ALU

ADD	00	0000	Reg a	Reg b	0
SUB	00	0001	Reg a	Reg b	0
AND	00	0010	Reg a	Reg b	0
OR	00	0011	Reg a	Reg b	0
MULU	00	0100	Reg a	Reg b	0
MULSW	00	0101	Reg a	Reg b	0
MULSB	00	0110	Reg a	Reg b	0
NOT	00	0111	Reg a	0	

[1] (Jacob)

XOR	00	1000	Reg a	Reg b	0
DIVU	00	1001	Reg a	Reg b	0
DIVSW	00	1010	Reg a	Reg b	0
DIVSB	00	1011	Reg a	Reg b	0
SHL	00	1100	Reg a	Reg b	0
SHR	00	1101	Reg a	Reg b	0
CMP	00	1110	Reg a	Reg b	0
LNOT	00	1111	Reg a	0	

Immediate

IMM	01	Immediate (0 to 0x3FFF)

Conditions

JMP	10	0000	Reg a	0	
JZ	10	0001	Reg a	Reg b	0
JNZ	10	0010	Reg a	Reg b	0

MISC

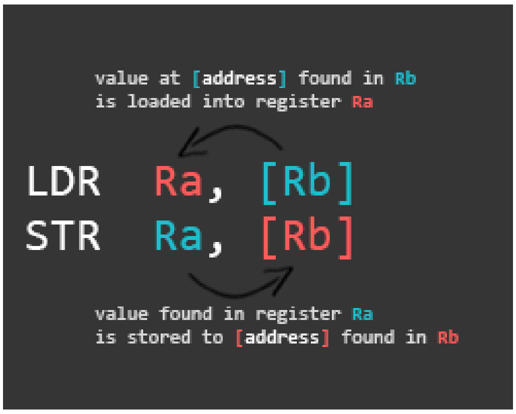
PUSHB	11	0000	Reg a	0	
PUSHW	11	0001	Reg a	0	
POPB	11	0010	Reg a	0	
POPW	11	0011	Reg a	0	
CALL	11	0100	Reg a	0	
RET	11	0101	0		
LDRB	11	0110	Reg a	Reg b	0
LDRW	11	0111	Reg a	Reg b	0
STRB	11	1000	Reg a	Reg b	0
STRW	11	1001	Reg a	Reg b	0
MOV	11	1010	Reg a	Reg b	0

1.1 Description of instruction operators

Mnemonic	Name and format	Mode (Binary)	Opcode (Binary)	Assembly format	Description
			ALU		
ADD	Add RR-type	00	0000	add rA, rB	Add rA with rB and save result in *DX* register
SUB	Sub RR-type	00	0001	sub rA, rB	Subtract rA with rB and save result in *DX* register (rA - rB)
AND	And RR-type	00	0010	and rA, rB	And rA with rB and save result in *DX* register
OR	Or RR-type	00	0011	or rA, rB	Or rA with rB and save result in *DX* register
MULU	Multiplicate unsigned RR-type	00	0100	mul rA, rB	Multiplicate rA with rB and save result in *DX* register
MULSW	Multiplicate signed word RR-type	00	0101	mulsw rA, rB	Multiplicate rA with rB and save result in *DX* register
MULSB	Multiplicate signed byte RR-type	00	0110	mulsb rA, rB	Multiplicate rA with rB and save result in *DX* register
NOT	Not R-type	00	0111	not rA	Not rA and save result in *DX* register
XOR	Xor RR-type	00	1000	xor rA, rB	Xor rA with rB and save result in *DX* register
DIVU	Division unsigned RR-type	00	1001	div rA, rB	Divide rA with rB and save result in *DX* register with remainted in *EX* register (rA / rB)
DIVSW	Division signed word RR-type	00	1010	divsw rA, rB	Divide rA with rB and save result in *DX* register with remainted in *EX* register (rA / rB)
DIVSB	Division signed byte RR-type	00	1011	divsb rA, rB	Divide rA with rB and save result in *DX* register with remainted in *EX* register (rA / rB)
SHL	Bit shift Left RR-type	00	1100	shl rA, rB	Bit shift left rA by rB amount and save in *DX* register
SHR	Bit shift right RR-type	00	1101	shr rA, rB	Bit shift right rA by rB amount and save in *DX* register
CMP	Compare RR-type	00	1110	cmp rA, rB	Subtract rA with rB and save result in *DX* register (rA - rB)
LNOT	Logical not R-type	00	1111	lnot rA	Calculate logical not of rA and store in *DX* register
			Immediate		
IMM	Immediate I-type	01	N/A **	imm imm14	Place immediate value in CX
			Conditions		
JMP	Jump	10	0000	jmp rA	Jump to value stored in rA

	R-type				
JZ	Jump if zero RR-type	10	0001	jz rA, rB	Jump to value stored in rA if rB is zero
JNZ	Jump if not zero RR-type	10	0010	jnz rA, rB	Jump to value stored in rA if rB isn't zero
			MISC		
PUSH	Push R-type	11	0000	push rA	Push rA onto stack
POP	Pop R-type	11	0001	pop rA	Pop stack onto rA
CALL	Call R-type	11	0010	call rA	Call function pointed by rA
RET	Return N-type	11	0011	ret	Return from function
LDR	Load RR-type	11	0100	ldr rA, rB	Load value pointed by rB into rA
STR	Store RR-type	11	0101	str rA, rB	Store value held in rA into rB
MOV	Move RR-type	11	0110	mov rA, rB	mov contents of rB into rA

** The opcode bits of imm are used to input the number, for a total of 14 bits, this way the number can go up to 0x3FFF.



Easier way to visualize the LDR and STR instructions.

Caution:

- SHL, SHR on signed number may produce an unexpected result.
- Implementing MOV improperly may cause it to overwrite data when using 8-Bit registers.
- Using the unsigned multiply (MULU) has a high chance of overflow.
- When CALL is used, the return address is stored in the *RA* register and instruction pointer is over written with the JMP, address
- When RET is used the address stored in *RA* register is copied to the instruction pointer, and I still need to think about implementing function nesting.
- When performing any operation with an 8-Bit register and a 16-Bit register, the computer will default to a 16-Bit register result.

- Certain operations such as LDR, STR, PUSH, POP, can detect that an 8-Bit register is being used and will overwrite only a byte of ram accordingly.

2 Registers

Register num	Register name	Register size (bits)	Description
0	AX	16	•
1	AS	8	•
2	BX	16	•
3	BS	8	•
4	CX	16	• When *imm* is used the result is stored here
5	CS	8	• When *imm* is used the result is stored here
6	DX	16	• When any operation under ALU is used the result is stored here (if an 8-bit register is used the upper 8-bits will be zeros)
7	DS	8	• When any operation under ALU is used the result is stored here
8	EX	16	• When *div* is used the remainder is stored here • Error flags are stored here, ALU overflow, writing where not supposed to (See section 5) (if an 8-bit register is used the upper 8-bits will be zeros)
9	ES	8	• When *div* is used the remainder is stored here • Error flags are stored here, ALU overflow, writing where not supposed to (See section 5)
10	.Data begin (DB)	16	• Points at the end of .text memory segment and at the start of .data memory segment
11	Memory end (ME)	16	• Points at the end of the total amount of RAM
12	Return Address (RA)	16	• When call is used, the return address is stored here (and 1 is added to it)
13	Stack Pointer (SP)	16	• Stores the memory address of the last data element added to the stack
14	Instruction Pointer (IP)	16	• Points to current instruction
15	I/O	16	• Register will receive user input when it is placed in an input position, and it will output to user when put in an output register position

To use any register listed above, simply convert the decimal number into binary, and place it in a register place holder such as rA or rB found in the description of the instructions.

The naming convention of the registers is was chosen arbitrarily, if anyone has a better naming scheme do let me know.

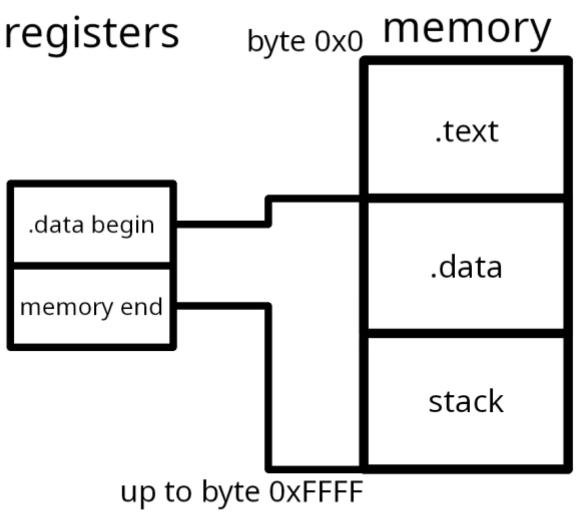
Better diagram to view registers:

	15	7 0
AX:		AS
BX:		BS
CX:		CS
DX:		DS
EX:		ES
.Data begin DB:		
Memory end ME:		
Return address RA:		
Stack pointer SP:		
Instruction pointer IP:		
I/O:		

Essentially registers ending in S (e.g. AS, BS, etc…) are the lower 8 bits of the larger register containing them (e.g. AX, BX, etc… respectively).

The Stack pointer will begin pointing from the first free byte after .data till where the ram runs out. It is important to know the specification of the device currently in use to ensure not running out of stack during the program execution. This can be checked in the *Memory end* register.

Use of .data begin register and memory end register:



3 Memory layout of a program

This instruction set only allows for a single program to be executing at once, as such the memory layout of each and every program will be as follows.

.text (executable code)
.data (initialized data)
stack

There's no heap for security and simplicity reasons.

Both data and text will be decided at compile time, hence the initial position of the stack will also be decided at that same time.

4 Security and reliability

Security and reliability, in this context, are synonyms. The design of the registers allows for the manufacturer to specify the amount of memory included with the product. Moreover it allows for the user to specify the start

All though these features are not necessarily required for the functioning of the processor, it is recommended that manufactures add support for these extra checks to ensure that the memory read / write instructions (such as STR, LDR, PUSH, POP) do not overwrite the .text memory segment nor that write beyond the memory region both of which could cause an unexpected behaviour. It is important to note that it is still possible to access the stack with STR and LDR, and it is possible to access .data memory segment with PUSH and POP, however this behaviour is not intended nor suggested.

5 Error flags

Flags are meant to help users understand why their program is acting unexpectedly.
The following table lists.

Code	Error flag	Description
00000001	arithmetic overflow	This flag is set when an arithmetic operation results in an overflow.
00000010	lower memory limit underflow	This flag is set when the memory being accessed is lower to .data begin register
00000100	upper memory limit overflow	This flag is set when the memory being accessed is above to memory end register

Multiple flags can be had at once. For example, if the register contains 00000011 it means that an arithmetic overflow occurred and that a lower memory limit underflow occurred. (This is a planned feature)