SIMPLE-CPU INSTRUCTION SET

SC91-A

Table des matières

II.1. Registers....5
II.2. Flags....5
II.3. Instructions, general encoding....6
II.3.1. LOADd – direct value loading into register....6
II.3.2. LOADr – register to register loading....8
II.3.3. LOADm – memory to register data loading....9
II.3.4. STORE – memory writing....10
II.3.5. AND – Logical AND operation....11
II.3.6. OR – Logical OR operation....11
II.3.7. XOR – Logical XOR operation....12
II.3.8. ROT – Register rotation....12
II.3.9. ADDr – Arithmetical addition between registers and more....15
II.3.10. ADDd – Arithmetical addition with a direct value....16
II.3.11. MUL – Arithmetical multiplication....17
II.3.12. DIV – Arithmetical division....18
II.3.13. CPYB – Bit copy between registers....19
II.3.14. JMPd – Unconditional jump to a direct address....20
II.3.15. JMPr – Unconditional jump to a registered address....20
II.3.16. JB – Conditional relative jump....21
II.3.17. PUSH – Stack register value storage....22
II.3.18. POP – Store top stack value in a register....22
II.3.19. CALL – Call a function from its absolute address....23
II.3.20. CALLr – Call a function from a register....23
II.3.21. RET – Return from a function....24
II.3.22. NOP – To do nothing....24
III.1. Loading data into registers....25
III.1.1. LOADd Rdd, Dir16....25
III.1.2. LOADds Rdd, Dir16....25
III.1.3. LOADdl Rdd, Dir16....25
III.1.4. LOADdh Rdd, Dir16....26
III.1.5. LOADm Rdd, (cast)&Rnn....27
III.1.6. LOADr Rdd, (cast)Rnn....28
III.2. Store data into memory....29
III.2.1. STORE &Rmm, (cast) Rnn 29
III.3. Logical operations....30
III.3.1. AND Rdd, Rnn, Rmm....30
III.3.2. OR Rdd, Rnn, Rmm....30
III.3.3. XOR Rdd, Rnn, Rmm....30
III.4. Rotations....31
III.4.1. ROL Rdd, Rnn, step....31
III.4.2. ROR Rdd, Rnn, step....31
III.4.3. RZL Rdd, Rnn, step....31
III.4.4. RZR Rdd, Rnn, step....32
III.4.5. R1L Rdd, Rnn, step....32
III.4.6. R1R Rdd, Rnn, step....32
III.4.7. RCL Rdd, Rnn....33

III.4.8. RCR Rdd, Rnn....33
III.4.9. RSR Rdd, Rnn, step....33
III.5. Arithmetical operations....34
III.5.1. ADDr Rdd, Rnn, Rmm....34
III.5.2. ADCr Rdd, Rnn, Rmm....34
III.5.3. SUBr Rdd, Rnn, Rmm....34
III.5.4. SUCr Rdd, Rnn, Rmm....34
III.5.5. CPMr Rnn, Rmm....35
III.5.6. ADDd Rdd, Rnn, dir16....35
III.5.7. SUBd Rdd, Rnn, dir16....35
III.5.8. MUL Rdd, Rnn, Rmm....35
III.5.9. MULs Rdd, Rnn, Rmm....36
III.5.10. DIV Rdd, Rnn, Rmm....36
III.5.11. DIVs Rdd, Rnn, Rmm....36
III.5.12. IDIV Rdd, Rnn, Rmm....36
III.5.13. DIVs Rdd, Rnn, Rmm....37
III.5.14. MOD Rdd, Rnn, Rmm....37
III.5.15. MODs Rdd, Rnn, Rmm....37
III.6. Bit manipulation....37
III.6.1. CPYB Rdd, Rnn, bitd, bits, mode....37
III.7. Conditionnal and unconditionnal jump....38
III.7.1. JMPd address....38
III.7.2. JMPr Rnn....38
III.7.3. JB Rnn, bit, val, address....38
III.7.4. JZ address....38
III.7.5. JNZ address....39
III.7.6. JS address....39
III.7.7. JNS address....39
III.7.8. JC address....39
III.7.9. JNC address....40
III.7.10. JO address....40
III.7.11. JNO address....40
III.7.12. JSG address....40
III.7.13. JSGE address....41
III.7.14. JSL address....41
III.7.15. JSLE address....41
III.7.16. JG address....41
III.7.17. JGE address....41
III.7.18. JL address....42
III.7.19. JLE address....42
III.8. Stack management....43
III.8.1. PUSH Rnn....43
III.8.2. POP Rnn....43
III.9. Procedure call....43
III.9.1. CALL Address....43
III.9.2. CALLr Rnn....43
III.9.3. RET....44
III.10. No operation instruction....44
III.10.1. NOP....44

I. Architecture description

I.1. Introduction

Simple-CPU is a 32 bits RISC processor with linear memory access using load and store methods. It is based on as less as possible instructions with lots of parameters. This document will so describe natural cpu language with its encoding and then describe a user friendly assembly set of instructions that alias natural one.

Simple-CPU has 32 double word registers (32b width) named R00 to R31. Twenty nine of them are for general use.

R00 is reserved for program counter

R01 is reserved for stack pointer

R02 is used for flags and specific status or conf bits.

R00 and R02 can be used in all instruction like general ones.

All instructions are 32 bits len even if they need less. All instructions need 1 cycle with instruction loading pipeline.

There is no distinction between program space and data space into memory. Instructions allways start on a 4 modulus address but data access can be done by byte, word or double word.

I.2. Instructions type

- Data acces instruction and Register loading

  LOAD : load a byte, word, dword from memory, register or a direct value.

  STOR : store a byte, word, dword into memory.

- Register logic

  AND : standard 32bits logic

  OR : standard 32bits logic

  XRL : standard 32bits logic

  ROT : standard multiple type of bit shifting

- Register arithmetics

  ADD r: standard add/sub with options

  ADDd : dircet value addition

  MUL : standard integer mutiplication storing result into two register

  DIV : standard integer division

- Conditionnal jump and bit manipulation

  JB : jump if specified flag have specified value

  CPY : copy a specific bit into an other bit in a destination register

  JMPd : jump to a sub function

JMPr : jump to a sub function identified by a register

- Sub program jump

  CALL : call of a sub function

  CALLR : call a function identified by a pointer

  RET : return from a subfunction

  PUSH : push register on stack

  POP : pop register on stack

- My favorite

  NOP : to do nothing

II. Natural Instruction set

II.1. Registers

Simple CPU register bank has 32 registers named R00 to R31. Each of 32bits.

R03 to R31 are general use register

R00 is program counter

R01 is stack pointer

R02 is used for flags

As 32x32b represent 1024 flip flop, simple cpu implementation could use less registers. in this case generic registers will be map from R03 to R07 or R03 to R15 depending of your choice. By default all 32 registers are available.

For instruction encoding, registers will be named Rnn where nn will be the register number between 0 to 31, encoded with 5 bits.

All registers can be accessed from instructions.

II.2. Flags

Flags are stored into R02

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EI																				0	P	C32	C16	C8	O32	O16	O8	S32	S16	S8	Z

- EI – Enable interrupt
- P – Parity (=1 when results has an even number of 1)
- C32, C16, C8 – 32, 16, 8bits parity
- O32, O16, O8 – 32, 16, 8bits overflow
- S32, S16, S8 – 32, 16, 8bits sign
- Z – Zéro
- 0 – always equal to 0

II.3. Instructions, general encoding

All instructions are 32bits length. This solution doesn't preserve memory space but allow to encode in a single instruction 30 bits memory adresse for call, 16 bits direct data for loading and allow to indicate for each instruction a destination, two operands and some few parameters.

Space lost could be saved in flash with compression and as ram memory space do not still being a problem, this choice will help us to have a *simple-cpu*.

Standard instruction encoding is like this :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								*Destination*					*Operand 1*					*Operand 2*					*Parameters & filler*								

First part allow to identify instruction type, this part is encoded from 2 bits to 8 bits.

The last part identify different parameters to condition the instruction results.

third and fourth part are operands register when the second is the destination register. So *Simple-Cpu* can execute operations like R10 = R11 + R12.

In the last part there is the a not assign space in the instruction (filler). This space is fill with some 0 but this is not mandatory. It is reserved for futur extensions or it can be used to check program integrity encoding checksums. This is the dream-it-yourself part.

II.3.1. LOADd – direct value loading into register

Load a direct 16bits width integer into a register. This value can be loaded into the lower or the higher part of the register. Number less than 65535 can be directly loaded into the 32bits register specifying signed or unsigned type.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								*Destination Reg*					*Direct 16bits values*																*mode*		*0*
0xFC								*Rdd*					*0xVVVV*																*[-1]*		*0*

[1] : loading mode

- 00b : *low* – load the 16 bits value into register Rdd bits 0 to 15.
- 01b : *high* – load the 16 bits value into register Rdd bits 16 to 31.
- 10b : *signed* – load the 16 bits value into register Rdd bits 0 to 15. Reset Rdd bits 16 to 31 to value bit 15.
- 11b : *unsigned* - load the 16 bits value into register Rdd bits 0 to 15. Reset Rdd bits 16 to 31 to zero.

Syntax :

```
LOADd        Rdd, value, [low | high | unsigned | signed]
```

Flags :

Impact C16, C8, Sx, O16, O8, Z, P.

O32 and C32 have no sense.

Operations :

(R02) <= 0x01020304

LOADd R02, 0x0102, high encoding : 0xFC100812

LOADd R02, 0x0304, low encoding : 0xFC101820

(R03) <= 0x0102

LOADd R03, 0x0102, unsigned encoding : 0xFC180816

(R04) <= -1

LOADd R04, -1, signed encoding : 0xFC27FFFC

II.3.2. LOADr – register to register loading

Load a register into an other one specifying destination type for 8, 16 or 32 bits loading, signed or unsigned.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								*Destination Reg*					*Source Reg*					*0*	*0*	*0*	*0*	*0*	*mode*			*0*	*0*	*0*	*0*	*0*	*0*
0xFE								*Rdd*					*Rnn*					*0*	*0*	*0*	*0*	*0*	*[1]*			*0*	*0*	*0*	*0*	*0*	*0*

[1] : loading mode

- 000b : *8u* – load Rnn bits 0 to 7 into Rdd bits 0 to 7. Rdd 8 to 31 are set to 0
- 100b : *8s* – load Rnn bits 0 to 7 into Rdd bits 0 to 7. Rdd 8 to 31 are set to Rnn bit 7.
- 001b : *16u* – load Rnn bits 0 to 15 into Rdd bits 0 to 15. Rdd 16 to 31 are set to 0
- 101b : *16s* – load Rnn bits 0 to 15 into Rdd bits 0 to 15. Rdd 16 to 31 are set to Rnn bit 15.
- 010b : *32* – load Rnn into Rdd.

Syntax :

LOADr Rdd, Rnn, [8u | 8s | 16u | 16s | 32]

Flags :

Impact C16, C8, Sx, O16, O8, Z, P.

O32 and C32 have no sense.

Operations :

(R02) <= (byte)(R03)

LOADr R02, R03, 8u encoding : 0xFE10C000

(R02) <= (signed byte)(R03)

LOADr R02, R03, 8s encoding : 0xFE10C100

(R03) <= (R04)

LOADr R03, R04, 32 encoding : 0xFE190080

or : LOADr R03, R04

II.3.3. LOADm – memory to register data loading

Load a value from memory into a register. The LOADm instruction allow to choice between byte, word or double word read; signed and unsigned. Adress to read to optained from a regsiter.

LOADm is not allowed on R00

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								Destination Reg					0	0	0	0	0	Address Reg					mode			0	0	0	0	0	0
0xFD								Rdd					0	0	0	0	0	Rmm					[1]			0	0	0	0	0	0

[1] : loading mode

- 000b : *8u* – load [Rmm] bits 0 to 7 into Rdd bits 0 to 7. Rdd 8 to 31 are set to 0
- 100b : *8s* – load [Rmm] bits 0 to 7 into Rdd bits 0 to 7. Rdd 8 to 31 are set to [Rmm] bit 7.
- 001b : *16u* – load [Rmm] bits 0 to 15 into Rdd bits 0 to 15. Rdd 16 to 31 are set to 0
- 101b : *16s* – load [Rmm] bits 0 to 15 into Rdd bits 0 to 15. Rdd 16 to 31 are set to [Rnn] bit 15.
- 010b : *32* – load [Rmm] into Rdd.

Syntax :

LOADm Rdd, &Rmm, [8u | 8s | 16u | 16s | 32]

Flags :

Impact C16, C8, Sx, O16, O8, Z, P.

O32 and C32 have no sense.

Operations :

(R02) <= (byte) [R03]

LOADm R02, &R03, 8u encoding : 0xFD100600

(R02) <= (signed byte)[R03]

LOADm R02, &R03, 8s encoding : 0xFD100700

(R03) <= [R04] encoding : 0xFD180880

LOADm R03,&R04, 32

or : LOADm R03, &R04

II.3.4. STORE – memory writing

Write a value from a register to the memory. Value can be write in byte, word or double word mode.

Encoding :

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Instruction code								*0*	*0*	*0*	*0*	*0*	*Source Reg*					*Address Reg*					*mode*		*0*	*0*	*0*	*0*	*0*	*0*	*0*
0xEC								*0*	*0*	*0*	*0*	*0*	*Rnn*					*Rmm*					*[1]*		*0*	*0*	*0*	*0*	*0*	*0*	*0*

[1] : writing mode

- 00b : *8* – write Rnn bits 0 to 7 into [Rmm] bits 0 to 7. Rdd 8 to 31 are not set.
- 01b : *16* – write Rnn bits 0 to 15 into [Rmm] bits 0 to 15. Rdd 16 to 31 are not set.
- 10b : *32* – write Rnn into [Rmm].

Syntax :

STORE &Rmm, Rnn, [8 | 16 | 32]

Flags :

Doesn't change flags.

Operations :

STORE &R02, R03, 8 encoding : 0xEC00C400

[R02] <= (byte)(R03)

STORE &R03,R04, 32 encoding : 0xEC010700

or : STORE &R03, R04

[R03] <= (R04) bit 31 to 24

[R03+1] <= (R04) bit 23 to 16

[R03+2] <= (R04) bits 15 to 8

[R03+3] <= (R04) bits 7 to 0

II.3.5. AND – Logical AND operation

Compute a logical AND between two registers and store the results into a third one.

Encoding :

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Instruction code								*Destination Reg*					*Operand 1*					*Operand 2*					*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
0XF1								*Rdd*					*Rnn*					*Rmm*					*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Syntax :

AND Rdd, Rnn, Rmm

Flags :

Impact C16, C8, Sx, O16, O8, Z, P.

O32 and C32 have no sense.

Operations :

(R02) <= (R03) AND (R04)

AND R02, R03, R04 encoding : 0xF110C800

II.3.6. OR – Logical OR operation

Compute a logical OR between two registers and store the results into a third one.

Encoding :

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Instruction code								*Destination Reg*					*Operand 1*					*Operand 2*					*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
0XF2								*Rdd*					*Rnn*					*Rmm*					*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Syntax :

OR Rdd, Rnn, Rmm

Flags :

Impact C16, C8, Sx, O16, O8, Z, P.

O32 and C32 have no sense.

Operations :

(R02) <= (R03) OR (R04)

OR R02, R03, R04 encoding : 0xF210C800

II.3.7. XOR – Logical XOR operation

Compute a logical XOR between two registers and store the results into a third one.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								*Destination Reg*					*Operand 1*					*Operand 2*					0	0	0	0	0	0	0	0	0
0XF3								*Rdd*					*Rnn*					*Rmm*					0	0	0	0	0	0	0	0	0

Syntax :

XOR Rdd, Rnn, Rmm

Flags :

Impact C16, C8, Sx, O16, O8, Z, P.

O32 and C32 have no sense.

Operations :

(R02) <= (R03) XOR (R04)

XOR R02, R03, R04 encoding : 0xF310C800

II.3.8. ROT – Register rotation

Do a rotation or a shift of a register to left or right *direction (d)*. The result is stored into a destination register. Thanks to several parameter this instruction will do left and right bit rotation. *mode* parameter let you choice between multiple type of rotations, when the *step* parameter allow to select the number of bit to move.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								*Destination Reg*					*Source Reg*					*Step*					*Dir*	*mode*			0	0	0	0	0
0XF4								*Rdd*					*Rnn*					*[1]*					*[2]*	*[3]*			0	0	0	0	0

[1] : Step

- How many bits to move : 1 is 1 bit shift or rotation ; 31 is 31 bit rotation.

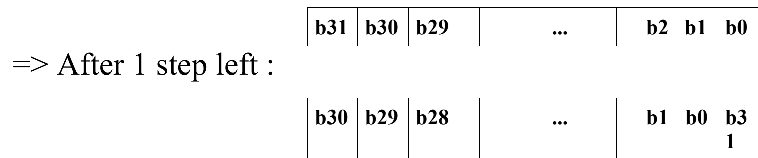
[2] : Direction

- 0b : *left* – shift or rotation on the left.
- 1b : *right* – shift or rotation on the right.

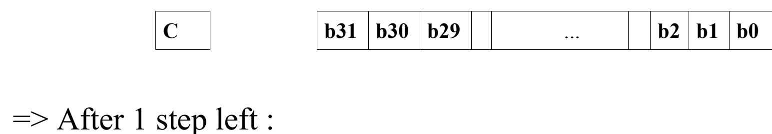
[3] : Mode

- 000b : *rot* – output bit becomes input bit. Only available for step = 1

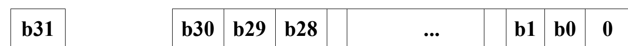
=> After 1 step left :



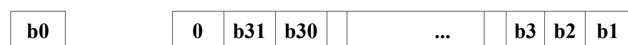
- 010b : *zero* – input bits are zero, last output bit is stored into C.



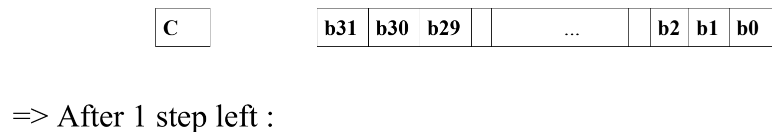
=> After 1 step left :



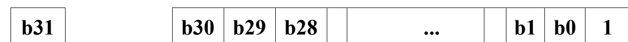
=> After 1 step right :



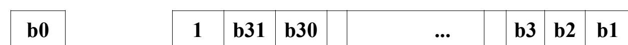
- 011b : *one* – input bits are one, last output bit is stored into C.



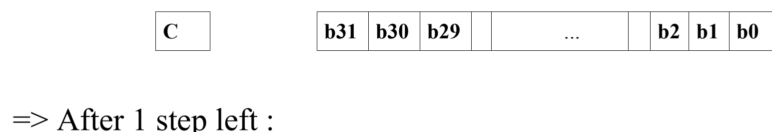
=> After 1 step left :



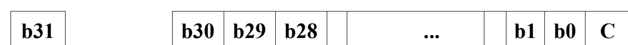
=> After 1 step right :



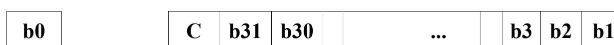
- 100b : *carry* – input bit takes carry value , output bit is stored into Carry. Only available for step = 1



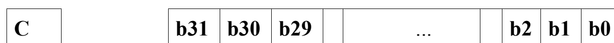
=> After 1 step left :



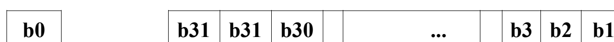
=> After 1 step right :



- 101b : *sign* – input bit's value is sign bit. output bit are stored in Carry. Only right direction make sense.



=> After 1 step right :



Syntax :

ROT Rdd, Rnn, *step, dir, mode*

Flags :

Impact Cx, Sx, Ox, Z, P.

Operations sample:

(R02) <= (R03) << 2

ROT R02, R03, 2, left, zero encoding : 0xF410C440

(R02) <= (R03) rotated right

ROT R02, R03, 1, right, rot encoding : 0xF410C300

II.3.9. ADDr – Arithmetical addition between registers and more

Compute an addition between two registers, store the results into a third one. Depending on *mode* use this instruction can do substraction or compare operations.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								*Destination Reg*					*Operand 1*					*Operand 2*					*mode*			*0*	*0*	*0*	*0*	*0*	*0*
0X80								*Rdd*					*Rnn*					*Rmm*					*[1]*			*0*	*0*	*0*	*0*	*0*	*0*

[1] : mode

- 000b : standard addition Rdd <= Rnn + Rmm ; symbol to use is **"+"**
- 100b : standard substraction Rdd <= Rnn + (-Rmm) ; symbol to use is **"-"**
- 110b : comparaison, flags are updated with (Rnn)+(-Rmm) results ; symbol is **"c"**
- 001b : carry addition Rdd <= Rnn + Rmm + Carry ; symbol is "**c+**"
- 101b : carry substraction Rdd <= Rnn + (- (Rmm+c)) symbol is "**c-**"

Syntax :

ADDr Rdd, Rnn, Rmm, *mode*

Flags :

Impact C, C16, C8, Sx, O, O16, O8, Z, P.

r

Operations :

(R02) <= (R03) + (R04)

ADDr R02, R03, R04, + encoding : 0x8010C800

or : ADDr R02, R03, R04

(R02) <= (R03) – (R04)

ADDr R02, R03, R04, - encoding : 0x8010C900

or : SUBr R02, R03, R04

(flags) <= (flags)((R03) – (R04))

ADDr R02, R03, R04, c encoding : 0x8010C980

or : CMPr R02, R03, R04

II.3.10. ADDd – Arithmetical addition with a direct value

Compute an addition between a source register and a signed 16bits direct value, store the results into a second register.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code			0	0	0	Value		Destination Reg					Operand 1					Value 14' lower bits													
0x5			0	0	0	[1]		Rdd					Rnn					[2]													

[1] and [2] :

- Value to add is *value* = ([1] << 14) | ([2])

Syntax :

ADDd Rdd, Rnn, *dir16*

Flags :

Impact C, C16, C8, Sx, O, O16, O8, Z, P.

Operations :

(R02) <= (R03) + 0x1234

ADDd R02, R03, 0x1234 encoding : 0xA010D234

(R02) <= (R03) – 2

ADDd R02, R03, -2 encoding : 0xA310FFFE

or : SUBd R02, R03, 2

II.3.11. MUL – Arithmetical multiplication

Compute an multiplication between two registers, store the results into a third one. Result, as operand are on 32 bits. Depending on *mode* use this instruction can act on signed or unsigned values into registers.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								*Destination Reg*					*Operand 1*					*Operand 2*					*mode*		*0*	*0*	*0*	*0*	*0*	*0*	*0*
0XF8								*Rdd*					*Rnn*					*Rmm*					*[1]*		*0*	*0*	*0*	*0*	*0*	*0*	*0*

[1] : mode

- 00b : ***signed*** **–** registers used contain signed values.
- 10b : ***unsigned*** **–** registers used contain unsigned values. (default)

Syntax :

MUL Rdd, Rnn, Rmm, *mode*

Flags :

Impact C, C16, C8, Sx, O, O16, O8, Z, P.

Operations :

(R02) <= (R03) * (R04)

MUL R02, R03, R04, unsigned encoding : 0xF810C900

or : MUL R02, R03, R04

II.3.12. DIV – Arithmetical division

Compute an division between two registers, store the results into a third one. Result, as operand are on 32 bits with 16 bits for rest and 16 bits for result. Depending on *mode* use this instruction can act on signed or unsigned values into registers. *Type* parameter allow to choose how result will be store between rest and result.

Encoding :

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Instruction code								*Destination Reg*					*Operand 1*					*Operand 2*					*mode*		*type*		*0*	*0*	*0*	*0*	*0*
0XF9								*Rdd*					*Rnn*					*Rmm*					*[1]*		*[2]*		*0*	*0*	*0*	*0*	*0*

[1] : mode

- 00b : ***signed*** **–** registers used contain signed values.
- 10b : ***unsigned*** **–** registers used contain unsigned values. (default)

[2] : type

- 11b : ***normal*** – rest is stored into destination register 16 highest bits, result into the lower ones.
- 01b : ***rest*** – rest is stored into destination register 32 bits, division result is lost.
- 10b : ***result*** – division result is stored into destination register 32bits, rest is lost.

Syntax :

DIV Rdd, Rnn, Rmm, *mode, type*

Flags :

Impact C, C16, C8, Sx, O, O16, O8, Z, P.

Operations :

(R02) bits 15 to 0 <= (R03) / (R04)

(R02) bits 31 to 16 <= (R03) % (R04)

DIV R02, R03, R04, unsigned, normal encoding : 0xF910C960

or : DIV R02, R03, R04

(R02) <= (signed)(R03)/(04)

DIV R02,R03,R04,signed,result encoding : 0xF910C840

(R02) <= (R03) % (04)

DIV R02,R03,R04,unsigned,rest encoding : 0xF910C920

II.3.13. CPYB – Bit copy between registers

This operation copy a bit from a register to an other one. Bit sourec and destination are user defined. Two *mode* allow to reset destination register before copy or to left it unchanged.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								*Destination Reg*					*Operand 1*					*destination bit*					*mode*		*0*	*0*	*source bit*				
0XE0								*Rdd*					*Rnn*					*5b value*					*[1]*		*0*	*0*	*5b value*				

[1] : mode

- 00b : ***0*** – destination register is firstly cleared.
- 10b : **=** – destination register is left unchanged.

Syntax :

CPYB Rdd, Rnn, *destination bit*, *source bit, mode*

Flags :

C will be initialized with readed bit ; other flags are updated.

Operations :

(R02) <= (((R03) & (1<<6)) >> 6) << 8

(R02 bit 8) <= (R03 bit 6)

CPYB R02, R03,8,6,0 encoding : 0xE010D006

(R02) <= (R02 & (NOT)(1<<8)) | (((R03) & (1<<6)) >> 6) << 8

(R02 bit 8) <= (R02 bit 31-9+7-0) | (R03 bit 6)

CPYB R02, R03,8,6,= encoding : 0xE010D106

II.3.14. JMPd – Unconditional jump to a direct address

Performs an unconditional jump to an address into memory space. Address gives as parameter is a 30 bit direct value. Jump where so aligned on 4 bytes words.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code		*Address to jump to*																													
0		*30 bits highest value*																													

Syntax :

JMPd *destinationAdress*

destination two lowest bits have to be 00.

Flags :

No Impact

Operations :

JMPd 0x12345678 encoding : 0x048D159E

II.3.15. JMPr – Unconditional jump to a registered address

Performs an unconditional jump to an address into memory space. Address is a 30 bit registered value. Jump where so aligned on 4 bytes words.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*Address register*					*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
0XE8								*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*Rmm*					*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Syntax :

JMPr *Rmm*

destination register, two lowest bits will be asumed as 00.

Flags :

No Impact

Operations :

JMPr R02 encoding : 0xE8000400

Next instruction will be *at address (R02) &0xFFFFFFFC*

II.3.16. JB – Conditional relative jump

Performs an conditional and relative jump to an address into memory space. Adress is relative to the next instruction adress (Program Counter value) and point to a 4 byte word address. Address is a signed 16 bits number allowing +/- 32k bytes jump.

Jump condition depend on the value of a specified bit in a specifier register. A mode allow extended tests to perform standard flag condition jumps.

This instruction has many aliases.

Encoding :

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="3">Instruction code</td><td>0</td><td colspan="3">mode</td><td>value</td><td colspan="5">Bit number</td><td colspan="5">Register</td><td colspan="14">Relative address</td></tr>
<tr><td colspan="3">0x6</td><td>0</td><td colspan="3">[1]</td><td>v</td><td colspan="5">bit</td><td colspan="5">Rnn</td><td colspan="14">Addr</td></tr>
</table>

[1] : mode

- 000b : **bit** – Jump if specified bit in specified register equal to *v*
- 010b : **+Z** – Jump as in bit mode or when *Z* flag is equal to 1.
- 011b : **-Z** – Jump as in bit mode and when *Z* flag is equal to 0.
- 110b : **G** – Jump as in bit mode and when S flag is equal O flag.
- 111b : **LE** – Jump as in bit mode or when S flag is not equal to O flag.

Syntax :

JB *mode, Rnn, bit, v,addr*

Flags :

No Impact

Operations :

JB bit, R02,3,1, +0x14 encoding : 0xC1188005

110 0 000 1 00011 00010 00000000000101 (00)

Next instruction will be *at address (PC+0x14) &0xFFFFFFFC* if bit 3 in R02 equal to 1.

JB +Z,R02,5,0,-0x160 encoding : 0xC428BFA8

- 00000001011000 (00)

110 0 010 0 00101 00010 11111110101000 (00)

Next instruction will be *at address (PC-0x160) &0xFFFFFFFC* if bit 5 in R02 equal to 0 or flag Z equal to 1.

II.3.17. PUSH – Stack register value storage

Store the value of a specifed register into the stack. Stack pointer is R01, this instruction store the value of the specified regsiter Rnn into the memory address pointed by R01. Then R01 is added to 4 to point the next stack word space.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								0	0	0	0	0	*Operand 1*					0	0	0	0	1	1	0	0	0	0	0	0	0	0
0XE4								0	0	0	0	0	*Rnn*					0	0	0	0	1	1	0	0	0	0	0	0	0	0

Syntax :

PUSH Rnn

Flags :

No impact

Operations :

[R01] <= (R31) puis (R01) <= (R01) + 4

PUSH R31 encoding : 0xE407C300

II.3.18. POP – Store top stack value in a register

Store the value of the top of the stack into a specifed register. Stack pointer is R01, is firstly decremented to point on the last value in stack then stack value is read and store into the specified register.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								*Destination*					0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
0XE5								*Rdd*					0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0

Syntax :

POP Rdd

Flags :

No impact

Operations :

(R01) <= (R01) - 4 puis (R31) <= [R01]

POP R31 encoding : 0xE5F80280

Special usage :

Return from a procedure call : POP R00

II.3.19. CALL – Call a function from its absolute address

Instruction *CALL* do a call to the function pointed by the given absolute address. It stores the current value of Code Pointer into the top of the stack. Absolute address is 30 bits encoded and so the jump must be done on a 4 modulus address offset.

Encoding :

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Instruction code		*Absolute Address*																													
0x1		*Add*																													

Syntax :

CALL Add

Flags :

No impact

Operations :

[R01] <= [R00] puis (R01) <= (R01)+4 et (R00) <= 0x12345678

CALL 0x123456 encoding : 0x448D159E

II.3.20. CALLr – Call a function from a register

Instruction *CALLr* do a call to the function pointed by the given register. It stores the current value of Code Pointer into the top of the stack. The two left significant bits of the register value are ignored.

Encoding :

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Instruction code								0	0	0	0	0	0	0	0	0	0	*Operand 1*					*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
0XE9								0	0	0	0	0	0	0	0	0	0	*Rmm*					*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Syntax :

CALLr Rmm

Flags :

No impact

Operations :

[R01] <= [R00] puis (R01) <= (R01) + 4 et (R00) <= (R31)

CALLr R31 encoding : 0xE9001FE00

II.3.21. RET – Return from a function

Instruction RET will return back from a procedure. Code pointer will be loaded with the top stack value, stack will be then decremented. This instruction is equal to POP R00.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0XE5								0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

Syntax :

RET

Flags :

No impact

Operations :

(R01) <= (R01) - 4 puis (R00) <= [R01]

RET encoding : 0xE5000400

II.3.22. NOP – To do nothing

This instruction simply do nothing.

Encoding :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction code								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0xFF								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Syntax :

NOP

Flags :

No impact

Operations :

Nothing

NOP encoding : 0xFF000000

III.User friendly instruction set

This instruction set extends general one's with instruction aliases on parameters. So full end user instruction set is described in this section. Detailed binary encoding is described in the last chapter.

III.1. Loading data into registers

III.1.1. LOADd Rdd, Dir16

Load a direct unsigned 16 bit value into the Rdd register. Rdd higher bits are cleared when lower bits are set to the d*ir16* value.

Syntax :	LOADd	Rdd, *dir16*
Exemple :	LOADd	R31,0x8000
	(R31) <= 0x000080000	
Alias :	LOADd	Rdd, *dir16*, unsigned

III.1.2. LOADds Rdd, Dir16

Load a direct signed 16 bit value into the Rdd register. Rdd higher bits are set to *dir16* sign bit when lower bits are set to the d*ir16* value.

Syntax :	LOADds	Rdd, *dir16*
Exemple :	LOADds	R31,0x8000
	(R31) <= 0xFFFF80000	
Alias :	LOADd	Rdd, *dir16*, signed

III.1.3. LOADdl Rdd, Dir16

Load a direct 16 bit value into the Rdd register lower bits. Rdd higher bits stay unchanged when lower bits are set to the d*ir16* value.

Syntax :	LOADdl	Rdd, *dir16*
Exemple :	LOADdl	R31,0x8000
	(R31) <= ((R31) & 0xFFFF0000) | 0x00008000	
Alias :	LOADd	Rdd, *dir16*, low

III.1.4. LOADdh Rdd, Dir16

Load a direct 16 bit value into the Rdd register higher bits. Rdd lower bits stay unchanged when higher bits are set to the d*ir16* value.

Syntax : LOADdh Rdd, *dir16*

Exemple : LOADdh R31,0x8000

(R31) <= ((R31) & 0x0000FFFF) | 0x80000000

Alias : LOADd Rdd, *dir16*, high

III.1.5. LOADm Rdd, *(cast)*&Rnn

Load into Rdd register the value read at (Rnn) address. Rdd can load a char, word or int value, depending on cast add before &Rnn. The *cast* string can be :

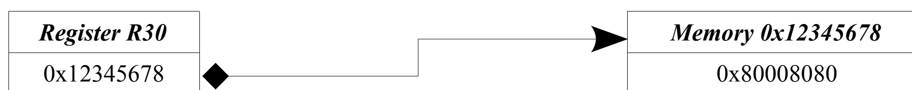
- uchar – for unsigned char loading : then Rdd lowest byte will be loaded with the value when the other bytes will be cleared
- char – for signed char loading : then Rdd lowest byte will be loaded with the value when the other bytes will be set to signed bit of this byte.
- uword – for unsigned word loading : then the two Rdd lowest byte will be loaded with the value when the other bytes will be cleared
- word – for signed word loading : then the two Rdd lowest byte will be loaded with the value when the other bytes will be set to signed bit of this word.

No *cast* indicates the load of a 32 bits value into Rdd.

LOADm is not allowed on R00

Syntax : LOADm Rdd, *(cast)* &Rmm

Exemples :



LOADm R31, &R30

(R31) <= 0x80008080

Alias : LOADd Rdd, &Rmm,32

LOADm R31, (uchar) &R30

(R31) <= 0x00000080

Alias : LOADd Rdd, &Rmm,8u

LOADm R31, (char) &R30

(R31) <= 0xFFFFFF80

Alias : LOADd Rdd, &Rmm,8s

LOADm R31, (uword) &R30

(R31) <= 0x00008080

Alias : LOADd Rdd, &Rmm,16u

LOADm R31, (word) &R30

(R31) <= 0xFFFF8080

Alias : LOADd Rdd, &Rmm,16s

III.1.6. LOADr Rdd, *(cast)*Rnn

Load into Rdd register the value of Rnn. Rdd can load a char, word or an int value, depending on cast add before Rnn. The *cast* string can be :

- uchar – for unsigned char loading : then Rdd lowest byte will be loaded with the value when the other bytes will be cleared
- char – for signed char loading : then Rdd lowest byte will be loaded with the value when the other bytes will be set to signed bit of this byte.
- uword – for unsigned word loading : then the two Rdd lowest byte will be loaded with the value when the other bytes will be cleared
- word – for signed word loading : then the two Rdd lowest byte will be loaded with the value when the other bytes will be set to signed bit of this word.

No *cast* indicates the load of a 32 bits value into Rdd.

Syntax : LOADr Rdd, *(cast)* Rnn

Exemples : Assumes that R30 value is 8008080

LOADr R31, R30

(R31) <= 0x80008080

Alias : LOADr Rdd, Rnn,32

LOADr R31, (uchar) R30

(R31) <= 0x00000080

Alias : LOADr Rdd, Rnn,8u

LOADr R31, (char) R30

(R31) <= 0xFFFFFF80

Alias : LOADd Rdd, Rnn,8s

LOADr R31, (uword) R30

(R31) <= 0x00008080

Alias : LOADd Rdd, Rnn,16u

LOADm R31, (word) R30

(R31) <= 0xFFFF8080

Alias : LOADd Rdd, Rnn,16s

III.2. Store data into memory

III.2.1. STORE &Rmm, *(cast)* Rnn

Load Rnn value into memory address indicated by Rdd value. Value to write can be cast to identify memory write size between char, word or int. *Cast* may be one of the following :

- char – for 1 byte writting : target byte will be loaded with the Rnn less significant byte.
- word – for 2 bytes writting : target two first byte will be loaded with the Rnn two less significant bytes.

No *cast* indicates the load of a 32 bits value into &Rdd.

Syntax : STORE &Rmm, *(cast)* Rnn

Exemples : Assumes that R30 value is 0x01020304 and R31 address is 0x1000

STORE &R31, R30

[0x1000] <= 0x01

[0x1001] <= 0x02

[0x1002] <= 0x03

[0x1003] <= 0x04

Alias : STORE &Rmm, Rnn,32

STORE &R31, (char) R30

[0x1000] <= 0x04

Alias : STORE &Rmm, Rnn,8

STORE &R31, (word) R30

[0x1000] <= 0x03

[0x1001] <= 0x04

Alias : STORE &Rmm, Rnn,16

III.3. Logical operations

III.3.1. AND Rdd, Rnn, Rmm

Load into Rdd register the result of a bit logical AND between Rnn and Rmm.

Syntax : AND Rdd, Rnn, Rmm

Exemple : AND R31, R30, R29

(R31) <= (R31) & (R29)

Alias : direct encoding

III.3.2. OR Rdd, Rnn, Rmm

Load into Rdd register the result of a bit logical OR between Rnn and Rmm.

Syntax : OR Rdd, Rnn, Rmm

Exemple : OR R31, R30, R29

(R31) <= (R31) | (R29)

Alias : direct encoding

III.3.3. XOR Rdd, Rnn, Rmm

Load into Rdd register the result of a bit logical XOR between Rnn and Rmm.

Syntax : XOR Rdd, Rnn, Rmm

Exemple : XOR R31, R30, R29

(R31) <= (R31) ^ (R29)

Alias : direct encoding

III.4. Rotations

III.4.1. ROL Rdd, Rnn, *step*

Realize a left rotation in Rdd register of *step* bits. In left rotations, right input bit is the left output bit for each step.

Syntax : ROL Rdd, Rnn, *step*

Exemple : assumes that R30 value is 0xC0000000

ROL R31, R30, 1

(R31) <= 0x80000001

Alias : ROT Rdd, Rnn, *step*, left, rot

III.4.2. ROR Rdd, Rnn, *step*

Realize a right rotation in Rdd register of *step* bits. In right rotations, left input bit is the right output bit for each step.

Syntax : ROR Rdd, Rnn, *step*

Exemple : assumes that R30 value is 0x80000001

ROR R31, R30, 1

(R31) <= 0xC0000000

Alias : ROT Rdd, Rnn, *step*, right, rot

III.4.3. RZL Rdd, Rnn, *step*

Realizes a left zero shift in Rdd register of *step* bits. In left zero shift, right input bit is allways set to 0. Left output bit is stored into the carry flag bit.

Syntax : RZL Rdd, Rnn, *step*

Exemple : assumes that R30 value is 0x80000001

RZL R31, R30, 1

(R31) <= 0x00000002 C <= 1

Alias : ROT Rdd, Rnn, *step*, left, zero

III.4.4. RZR Rdd, Rnn, *step*

Realizes a right zero shift in Rdd register of *step* bits. In right zero shift, left input bit is allways set to 0. Right output bit is stored into the carry flag bit.

Syntax : RZR Rdd, Rnn, *step*

Exemple : assumes that R30 value is 0x80000001

RZR R31, R30, 1

(R31) <= 0x40000000 C <= 1

Alias : ROT Rdd, Rnn, *step*, right, zero

III.4.5. R1L Rdd, Rnn, *step*

Realize a left one shift in Rdd register of *step* bits. In left one shift, right input bit is allways set to 1. Left output bit is stored into the carry flag bit.

Syntax : ROL Rdd, Rnn, *step*

Exemple : assumes that R30 value is 0x80000001

R0L R31, R30, 1

(R31) <= 0x00000003 C <= 1

Alias : ROT Rdd, Rnn, *step*, left, one

III.4.6. R1R Rdd, Rnn, *step*

Realizes a right one shift in Rdd register of *step* bits. In right one shift, left input bit is allways set to 1. Right output bit is stored into the carry flag bit.

Syntax : ROR Rdd, Rnn, *step*

Exemple : assumes that R30 value is 0x80000001

ROR R31, R30, 1

(R31) <= 0xC0000000 C <= 1

Alias : ROT Rdd, Rnn, *step*, right, one

III.4.7. RCL Rdd, Rnn

Realize a left carry shift in Rdd register of *1* bit. In left carry shift, right input bit is allways set to the carry flag value. Left output bit is stored into the carry flag bit.

Syntax : RCL Rdd, Rnn

Exemple : assumes that R30 value is 0x80000001 and carry set to 1

RCL R31, R30

(R31) <= 0x00000003 C <= 1

Alias : ROT Rdd, Rnn, 1, left, carry

III.4.8. RCR Rdd, Rnn

Realizes a right carry shift in Rdd register of *1* bit. In right carry shift, left input bit is allways set to the carry flag value. Right output bit is stored into the carry flag bit.

Syntax : RCR Rdd, Rnn

Exemple : assumes that R30 value is 0x80000001 and carry set to 1

RCR R31, R30

(R31) <= 0xC0000000 C <= 1

Alias : ROT Rdd, Rnn, 1, right, carry

III.4.9. RSR Rdd, Rnn, *step*

Realizes a right sign shift in Rdd register of 1 bits. In right sign shift, left input bit is allways set to the register sign bit value. Right output bit is stored into the carry flag bit.

Syntax : RCR Rdd, Rnn

Exemple : assumes that R30 value is 0x80000001 and carry set to 1

RCR R31, R30

(R31) <= 0xC0000000 C <= 1

Alias : ROT Rdd, Rnn, *step*, right, sign

III.5. Arithmetical operations

III.5.1. ADDr Rdd, Rnn, Rmm

Store into Rdd the results of the Rnn and Rmm values addition.

Syntax : ADDr Rdd, Rnn, Rmm

Exemple : ADDr R31, R30, R29

(R31) <= (R30) + (R29)

Alias : ADDr Rdd, Rnn, Rmm, +

III.5.2. ADCr Rdd, Rnn, Rmm

Store into Rdd the results of the Rnn, Rmm and carry flag values addition.

Syntax : ADCr Rdd, Rnn, Rmm

Exemple : ADCr R31, R30, R29

(R31) <= (R30) + (R29) + Carry

Alias : ADDr Rdd, Rnn, Rmm, c+

III.5.3. SUBr Rdd, Rnn, Rmm

Store into Rdd the results of the Rnn and Rmm values substraction.

Syntax : SUBr Rdd, Rnn, Rmm

Exemple : SUBr R31, R30, R29

(R31) <= (R30) - (R29)

Alias : ADDr Rdd, Rnn, Rmm, -

III.5.4. SUCr Rdd, Rnn, Rmm

Store into Rdd the results of the Rnn, Rmm and carry flag values substraction.

Syntax : SUCr Rdd, Rnn, Rmm

Exemple : SUCr R31, R30, R29

(R31) <= (R30) - (R29) - Carry

Alias : ADDr Rdd, Rnn, Rmm, c-

III.5.5. CPMr Rnn, Rmm

Load flags with the results of the Rnn and Rmm substraction.

Syntax : CPMr Rnn, Rmm

Exemple : CPMr R30, R29

(R02) <= compute_flags((R30) – (R29))

Alias : ADDr R02, Rnn, Rmm, c

III.5.6. ADDd Rdd, Rnn, *dir16*

Store into Rdd the result of the Rnn and signed *dir16* addition.

Syntax : ADDd Rdd, Rnn, *dir16*

Exemple : ADDd R31, R30, 0x1234

(R31) <= (R30) + 0x1234

Alias : ADDr Rdd, Rnn, *dir16*

III.5.7. SUBd Rdd, Rnn, *dir16*

Store into Rdd the result of the Rnn and signed *dir16* substraction.

Syntax : SUBd Rdd, Rnn, *dir16*

Exemple : SUBd R31, R30, 0x1234

(R31) <= (R30) - 0x1234

Alias : ADDr Rdd, Rnn, - *dir16*

III.5.8. MUL Rdd, Rnn, Rmm

Store into Rdd the result of the Rnn and Rmm unsigned multiplication.

Syntax : MUL Rdd, Rnn, Rmm

Exemple : MUL R31, R30, R29

(R31) <= (int)((R30) * (R29))

Alias : MUL Rdd, Rnn, Rmm, unsigned

III.5.9. MULs Rdd, Rnn, Rmm

Store into Rdd the result of the Rnn and Rmm signed multiplication.

Syntax : MULs Rdd, Rnn, Rmm

Exemple : MULs R31, R30, R29

(R31) <= (int)((R30) * (R29))

Alias : MUL Rdd, Rnn, Rmm, signed

III.5.10. DIV Rdd, Rnn, Rmm

Store into Rdd higher word the rest of the unsigned Rnn by Rmm division. The result of that division is stored into the lower word.

Syntax : DIV Rdd, Rnn, Rmm

Exemple : DIV R31, R30, R29

(R31) bit 31 to 16 <= (R30) % (R29)

(R31) bit 15 to 00 <= (R30) / (R29)

Alias : DIV Rdd, Rnn, Rmm, unsigned, normal

III.5.11. DIVs Rdd, Rnn, Rmm

Store into Rdd higher word the rest of the signed Rnn by Rmm division. The result of that division is stored into the lower word.

Syntax : DIVs Rdd, Rnn, Rmm

Exemple : DIVs R31, R30, R29

(R31) bit 31 to 16 <= (R30) % (R29)

(R31) bit 15 to 00 <= (R30) / (R29)

Alias : DIV Rdd, Rnn, Rmm, signed, normal

III.5.12. IDIV Rdd, Rnn, Rmm

Store into Rdd lower word the result of the unsigned Rnn by Rmm division.

Syntax : IDIV Rdd, Rnn, Rmm

Exemple : IDIV R31, R30, R29

(R31) <= (R30) / (R29)

Alias : DIV Rdd, Rnn, Rmm, unsigned, result

III.5.13. IDIVs Rdd, Rnn, Rmm

Store into Rdd lower word the result of the signed Rnn by Rmm division.

Syntax : IDIVs Rdd, Rnn, Rmm

Exemple : IDIVs R31, R30, R29

(R31) <= (R30) / (R29)

Alias : DIV Rdd, Rnn, Rmm, signed, result

III.5.14. MOD Rdd, Rnn, Rmm

Store into Rdd higher word the rest of the unsigned Rnn by Rmm division.

Syntax : MOD Rdd, Rnn, Rmm

Exemple : MOD R31, R30, R29

(R31) <= (R30) % (R29)

Alias : DIV Rdd, Rnn, Rmm, unsigned, rest

III.5.15. MODs Rdd, Rnn, Rmm

Store into Rdd higher word the rest of the signed Rnn by Rmm division.

Syntax : MODs Rdd, Rnn, Rmm

Exemple : MODs R31, R30, R29

(R31) <= (R30) % (R29)

Alias : DIV Rdd, Rnn, Rmm, signed, rest

III.6. Bit manipulation

III.6.1. CPYB Rdd, Rnn, *bitd, bits, mode*

Copy the specifier *bits* from Rnn to the Rdd specified *bitd* clearing (*mode* = “0”) or keeping (*mode* = “=”) the other destination bits.

Syntax : CPYB Rdd, Rnn, *bitd*, *bits*, *mode*

Exemple : CPYB R31, R30, 1, 5, 0

(R31) <= (((R30) & 0x00000020) >> 5) << 1

Alias : direct encoding

III.7. Conditionnal and unconditionnal jump

III.7.1. JMPd *address*

Jump to the specified 32 bit address. This last has to be aligned on an int.

Syntax : JMPd *address*

Exemple : JMPd *myFunction* *(0x12345678)*

Jump to address of *myFunction* (0x12345678 & 0xFFFFFFFC)

Alias : direct encoding

III.7.2. JMPr *Rnn*

Jump to the value's specified register address. This last has to be aligned on an int.

Syntax : JMPr Rnn

Exemple : Assumes that R30 value is 0x12345678

JMPr R30 *(0x12345678)*

Jump to address (0x12345678 & 0xFFFFFFFC)

Alias : direct encoding

III.7.3. JB Rnn, *bit, val, address*

Jump to the specified signed 16 bit direct address value depending on value of specified bit into Rnn. Jump if values match.

Syntax : JB Rnn, *bit, val, address*

Exemple : assumes that R30 value is 0x00000001

JB R30, 0, 1, + 0x1000

Jump to R00 current address + 0x1000 as R30 bit 0 is equal to 1.

Alias : JB bit, Rnn, *bit*, *val*, *address*

III.7.4. JZ *address*

Jump to the specified signed 16 bit direct address value when zero flag is set to 1.

Syntax : JZ *address*

Exemple : assumes that R02 value is 0x00000001

JZ +0x1000

Jump to R00 current address + 0x1000 as flag Z is set to 1.

Alias : JB bit, R02, 0, 1, *address*

III.7.5. JNZ *address*

Jump to the specified signed 16 bit direct address value when zero flag is set to 0.

Syntax : JNZ *address*

Exemple : assumes that R02 value is 0x00000000

JNZ +0x1000

Jump to R00 current address + 0x1000 as flag Z is set to 0.

Alias : JB bit, R02, 0, 0, *address*

III.7.6. JS *address*

Jump to the specified signed 16 bit direct address value when sign flag is set to 1.

Syntax : JS *address*

Exemple : JS +0x1000

Jump to R00 current address + 0x1000 when S = 1

Alias : JB bit, R02, 3, 1, *address*

III.7.7. JNS *address*

Jump to the specified signed 16 bit direct address value when sign flag is set to 0.

Syntax : JNS *address*

Exemple : JNS +0x1000

Jump to R00 current address + 0x1000 when S = 0

Alias : JB bit, R02, 3, 0, *address*

III.7.8. JC *address*

Jump to the specified signed 16 bit direct address value when carry flag is set to 1.

Syntax : JC *address*

Exemple : JC +0x1000

Jump to R00 current address + 0x1000 when C = 1

Alias : JB bit, R02, 9, 1, *address*

III.7.9. JNC *address*

Jump to the specified signed 16 bit direct address value when carry flag is set to 0.

Syntax : JNC *address*

Exemple : JNC +0x1000

Jump to R00 current address + 0x1000 when C = 0

Alias : JB bit, R02, 9, 0, *address*

III.7.10. JO *address*

Jump to the specified signed 16 bit direct address value when overflow flag is set.

Syntax : J0 *address*

Exemple : J0 +0x1000

Jump to R00 current address + 0x1000 when O = 1

Alias : JB bit, R02, 6, 1, *address*

III.7.11. JNO *address*

Jump to the specified signed 16 bit direct address value when overflow flag is cleared.

Syntax : JNO *address*

Exemple : JNO +0x1000

Jump to R00 current address + 0x1000 when O = 0

Alias : JB bit, R02, 6, 0, *address*

III.7.12. JSG *address*

Jump to the specified signed 16 bit direct address value when in the last signed operation ADDr or ADDd (and aliases) Rnn is greater than Rmm.

Syntax : JSG *address*

Exemple : JSG +0x1000

Alias : JB G, R02, 0, 0, *address*

III.7.13. JSGE *address*

Jump to the specified signed 16 bit direct address value when in the last signed operation ADDr or ADDd (and aliases) Rnn is greater or equal than Rmm.

Syntax : JSGE *address*

Exemple : JSGE +0x1000

Alias : JB G, R02, 11, 0, *address*

III.7.14. JSL *address*

Jump to the specified signed 16 bit direct address value when in the last signed operation ADDr or ADDd (and aliases) Rnn is lower than Rmm.

Syntax : JSL *address*

Exemple : JSL +0x1000

Alias : JB LE, R02, 11, 1, *address*

III.7.15. JSLE *address*

Jump to the specified signed 16 bit direct address value when in the last signed operation ADDr or ADDd (and aliases) Rnn is lower or equal than Rmm.

Syntax : JSLE *address*

Exemple : JSLE +0x1000

Alias : JB LE, R02, 0, 1, *address*

III.7.16. JG *address*

Jump to the specified signed 16 bit direct address value when in the last unsigned operation ADDr or ADDd (and aliases) Rnn is greater than Rmm.

Syntax : JG *address*

Exemple : JG +0x1000

Alias : JB -Z, R02, 9, 0, *address*

III.7.17. JGE *address*

Jump to the specified signed 16 bit direct address value when in the last unsigned operation ADDr or ADDd (and aliases) Rnn is greater or equal than Rmm.

Syntax : JGE *address*

Exemple : JGE +0x1000

Alias : JB bit, R02, 9, 0, *address*

III.7.18. JL *address*

Jump to the specified signed 16 bit direct address value when in the last unsigned operation ADDr or ADDd (and aliases) Rnn is lower than Rmm.

Syntax : JL *address*

Exemple : JL +0x1000

Alias : JB bit, R02, 9, 1, *address*

III.7.19. JLE *address*

Jump to the specified signed 16 bit direct address value when in the last unsigned operation ADDr or ADDd (and aliases) Rnn is lower or equal than Rmm.

Syntax : JLE *address*

Exemple : JLE +0x1000

Alias : JB +Z, R02, 9, 1, *address*

III.8. Stack management

III.8.1. PUSH Rnn

Push register value (Rnn) into the top of the stack indicated by (R01), then increment (R01) to point the next top stack address.

Syntax : PUSH Rnn

Exemple : PUSH R31

&(R01) <= (R31) then (R01) <= (R01) + 4

Alias : direct encoding

III.8.2. POP Rnn

Decrement stack pointer then pop top stack value into register (Rnn).

Syntax : POP Rnn

Exemple : POP R31

(R01) <= (R01) - 4 then (R31) <= &(R01)

Alias : direct encoding

III.9. Procedure call

III.9.1. CALL *Address*

Jump to the specified address after pushing code pointer (R00) register.

Syntax : CALL *AbsoluteAddress*

Exemple : CALL 0x12345678

&(R01) <= (R00) then (R01) <= (R01) + 4

Jump to *AbsoluteAddress* & 0xFFFFFFFC

Alias : direct encoding

III.9.2. CALLr Rnn

Jump to the address specified by the Rnn value after pushing code pointer (R00) register.

Syntax : CALL Rnn

Exemple : CALL R31

&(R01) <= (R00) then (R01) <= (R01) + 4

Jump to (R31) & 0xFFFFFFFC

Alias : direct encoding

III.9.3. RET

Jump back to caller. Return from a procedure call.

Syntax : RET

Exemple : RET

(R01) <= (R01) - 4 then (R00) <= &(R01)

Alias : POP R00

III.10. No operation instruction

III.10.1. *NOP*

Do nothing.

Syntax : NOP

Exemple : NOP

No action.

Alias : direct encoding