# LSA Sim

**Contents**

[**1 LSA Processor**](#_1_LSA_Processor)

[**1.1 Introduction**](#_1.1_Introduction)

[**1.1.1 Notation**](#_1.1.1_Notation)

[**1.1.2 Instruction Set Features**](#_1.1.2_Instruction_Set)

[**1.1.3 Memory**](#_1.1.3_Memory)

[**1.1.4 Registers**](#_1.1.4_Registers)

[**1.2 LSA Opcodes**](#_1.2_LSA_Opcodes)

[**1.2.1 Load pc Relative**](#_1.2.1_Load_pc)

[**1.2.2 Load sp Relative**](#_1.2.2_Load_sp)

[**1.2.3 Store pc Relative**](#_1.2.3_Store_pc)

[**1.2.4 Store sp Relative**](#_1.2.4_Store_sp)

[**1.2.5 Load/Store Word**](#_1.2.5_Load/Store_Word)

[**1.2.6 Mov To/From High Register**](#_1.2.6_Mov_To/From)

[**1.2.7 Alu Operation**](#_1.2.7_Alu_Operation)

[**1.2.8 Shift/Rotate**](#_1.2.8_Shift/Rotate)

[**1.2.9 Load Immed Low/High/Zero**](#_1.2.9_Load_Immed)

[**1.2.10 Branch**](#_1.2.10_Branch)

[**1.2.11 Swap**](#_1.2.11_Swap)

[**1.2.12 Call**](#_1.2.12_Call)

[**1.2.13 Possible future instructions:**](#_1.2.13_Possible_future)

[**1.3 Memory Mapping**](#_1.3_Memory_Mapping)

[**2 Complete list of lsa supported instructions:**](#_2_Complete_list)

# 1 LSA Processor

## 1.1 Introduction

[**Back to Contents**](#TOC)

### 1.1.1 Notation

| **Notation** | **Explanation** |
| --- | --- |
| rx | register x, where x = decimal number between 0-127 |
| rd | destination register |
| rs | source register |
| immed | immediate (integer constant) |
| unsigned\_immed | unsigned immediate (integer constant) positive only |
| signed\_immed | signed immediate (integer constant) can be positive or negative |
| label\_name | Name assigned in lsa-as label instruction |
| 0x1234 | 0x indicates hex number |
| alu | arithmetic logic unit |
| pc | program counter |
| sp | stack pointer |
| n | negative flag bit |
| z | zero flag bit |
| c | carry flag bit |
| v | overflow flag bit |
| pop | take value off stack |
| push | put value on stack |
| [rx] | rx is used as an address in memory space |
| lob | Low order bit/bits |
| hob | High order bit/bits |
| & | Logical .and. |
| | | Logical.or. |
| ^ | Logical.xor. |
| ~ | Logical.not. |
| ++ | increment |
| -- | decrement |
| rx{x:y} | Bits in rx: x=from bit number, y=to bit number |

**Table 1: Notation used in this document**

### 1.1.2 Instruction Set Features

The LSA processor features a load-store architecture RISC instruction set. Registers, memory, and instructions are fixed at 16 bits.

### 1.1.3 Memory

The memory address space for the LSA processor is limited to 16 bits. Upper address space is normally used for access to peripherals. The vector table starts at address 0x0000 and immediately after that sram. Each memory location is 16 bits, address 0x0000 is the first 16 bit data word, 0x0001 is the next 16 bit data word, etc. The term word refers to a 16 bit binary value.

### 1.1.4 Registers

The LSA processor supports 16 general purpose registers, three of which are also special purpose. High Registers can be added, extending to a maximum 128 registers. All registers are 16 bit.

#### 1.1.4.1 Program Counter (R0)

From a programmers standpoint r0 contains the address of the executing instruction plus 1. A number of instructions, lpc, spc, bpc, etc use r0 without it being specified. Most other instructions allow combinations of any register, including r0. LSA Processor 2

#### 1.1.4.2 Status Register (R1)

This register contains the ALU result flags n, z, v, and c. Conditional branch instructions like bz, bnv, etc. specifically use the bits in this register. When r1 register is specified as the destination of an instruction the result of the instruction operation takes precedence over the flags. For example xor r1,r1 will result in the contents of r1 being a 0x00, the z flag will not be set.

|  |
| --- |
| r1[3:0] = vncz |

All alu operations update the flags any load or other operation that has an rd of r1 will write over the flags. Also rrc and rlc affect the c flag. Other than those cases the flags are not changed by an instruction.

#### 1.1.4.3 Stack Pointer (R2)

This register is the stack pointer. Two instructions, lsp and ssp, are specific to r2 for reaching into the stack. These instructions are designed for stacks that grown down. The LSA processor does not have opcodes for push and pop instructions as the stock store word and load word instructions can perform the same function when r2 is used

|  |
| --- |
| stw [--r2],rs ; push register  ldw rd,[r2++] ; pop register |

#### 1.1.4.4 High Registers

|  |
| --- |
| r0-r15 Low bank  r16-r127 High bank |

Move to/from high register is the only instruction that supports the high bank of registers. Removing support for this instruction can be used to reduce the hardware requirements needed to implement the processor. Allowing this instruction and these registers can make the implementation of a compiler back end easier. Alternatively the hardware implementation does not have to support the full 128 registers it can support 64 or 32 for example by ignoring some of the bits in the instruction

## 1.2 LSA Opcodes

[**Back to Contents**](#TOC)

| **Bits 12-15** | **Bits 8-11** | **Bits 4-7** | **Bits 0-3** | **Instruction** |
| --- | --- | --- | --- | --- |
| 0000 | dddd | iiii | iiii | load pc relative |
| 0001 | dddd | iiii | iiii | load sp relative |
| 0010 | ssss | iiii | iiii | store pc relative |
| 0011 | ssss | iiii | iiii | store sp relative |
| 0100 | dddd | ssss | aaaa | load/store |
| 0101 | llll | ahhh | hhhh | mov to/from high register |
| 0110 | dddd | ssss | aaaa | alu operation |
| 0111 | dddd | sisi | aaaa | shift |
| 1000 | dddd | iiii | iiii | load immed low zero high |
| 1001 | dddd | iiii | iiii | load immed high zero low |
| 1010 | dddd | iiii | iiii | load immed low |
| 1011 | dddd | iiii | iiii | load immed high |
| 1100 | aaaa | siii | iiii | branch pc relative |
| 1101 | aaaa | dddd | ssss | branch register |
| 1110 |  |  |  | (future) |
| 1111 | 1111 | 1111 | 1111 | halt |

**Table 2: Opcode Map**

Bits 12-15 = op code

i = immed

d = dest reg

s = source reg

l = low reg

h = high reg

sisi = ?

a = condition bits

### 1.2.1 Load pc Relative

|  |  |
| --- | --- |
| MAP | 0000 dddd iiii iiii load pc relative |
| SYNTAX | lpc rd,#imm  lpc rd,label\_name |
| FUNCTION | rd = memory[pc+imm] |
| NOTES | 1. pc (r0) is instruction address + 1  2. imm is not sign\_extended, it can only address forward  3. for label\_name form, imm = label\_name{0:7} |

**Examples:**

|  |
| --- |
| lpc rd,#5  lpc rd,#0x7  lpc rd,larry |

### 1.2.2 Load sp Relative

|  |  |
| --- | --- |
| MAP | 0001 dddd iiii iiii load sp relative |
| SYNTAX | lsp rd,#imm |
| FUNCTION | rd = memory[sp+imm] |
| NOTES | 1. imm is not sign extended, it can only address forward  2. labels are pc relative, so there is no ;sp label format |

**Examples:**

|  |
| --- |
| lsp rd,#5  lsp rd,#0x7 |

### 1.2.3 Store pc Relative

|  |  |
| --- | --- |
| MAP | 0010 ssss iiiiiiii store pc relative |
| SYNTAX | spc #imm,rs  spc label\_name,rs |
| FUNCTION | memory[pc+imm] = rs |
| NOTES | 1. pc (r0) is instruction address + 1  2. imm is not sign extended, it can only address forward  3. For Label form, imm = label\_name{0:7} |

**Examples:**

|  |
| --- |
| spc #5,rs  spc #0x7,rs  spc label,rs |

### 1.2.4 Store sp Relative

|  |  |
| --- | --- |
| MAP | 0011 ssss iiiiiiii store sp relative |
| SYNTAX | ssp #imm,rs |
| FUNCTION | memory[sp+imm] = rs |
| NOTES | 1. imm is not sign extended, it can only address forward  2. labels are pc relative, so there is no ssp label format |

**Examples:**

|  |
| --- |
| ssp #5,rs  ssp #0x7,rs |

### 1.2.5 Load/Store Word

#### 1.2.5.1 Load word rs incr/decr

|  |  |
| --- | --- |
| MAP | (see paren below) dddd ssss aaaa load/store |
| SYNTAX | (0000) ldw rd,[rs]  (0001) ldw rd,[rs++]  (0010) ldw rd,[++rs]  (0011) ldw rd,[rs--]  (0100) ldw rd,[--rs] |
| FUNCTION | [rs]: rd = memory [rs]  [rs++]: rd = memory [rs], rs=rs+1  [++rs]: rs=rs+1, rd = memory [rs]  [rs--]: rd = memory [rs], rs=rs-1  [--rs] : rs=rs-1, rd = memory [rs] |

#### 1.2.5.2 Store word rd incr/decr

|  |  |
| --- | --- |
| MAP | (see 1.2) dddd ssss aaaa load/store |
| SYNTAX | (1000) stw [rd] ,rs  (1001) stw [rd++],rs  (1010) stw [++rd],rs  (1011) stw [rd--],rs  (1100) stw [--rd],rs |
| FUNCTION | [rd]: memory [rd] = rs  [rd++]: memory [rd] = rs, rd=rd+1  [++rd]: rd=rd+1, memory [rd]  [rd--]: memory [rd] = rs, rd=rd-1  [--rd] : rd=rd-1, memory [rd] = rs |

### 1.2.6 Mov To/From High Register

|  |  |
| --- | --- |
| MAP | 0101 llll ahhh hhhh mov to/from high register |
| SYNTAX | mov rd,rs |
| FUNCTION | a  0 mov rl,rh  1 mov rh,rl |
| NOTES | 1. rl = r0 - r15  2. rh = r0 - r127 |

### 1.2.7 Alu Operation

|  |  |
| --- | --- |
| MAP | (see paren below) dddd ssss aaaa alu operation |
| SYNTAX | (0000) add rd,rs  (0001) sub rd,rs  (0010) and rd,rs  (0011) dna rd,rs  (0100) or rd,rs  (0101) xor rd,rs  (0110) neg rd,rs  (0111) not rd,rs  (1000) inc rd,rs  (1001) dec rd,rs  (1010) cmp rd,rs  (1011) tst rd,rs |
| FUNCTION | add: rd = rd + rs  sub: rd = rd - rs  and: rd = rd & rs  dna: rd = rd & (~rs)  or: rd = rd | rs  xor: rd = rd ^ rs  neg: rd = 0 - rs  not: rd = ~ rs  inc: rd = rs + 1  dec: rd = rs - 1  cmp: rd = rd - rs  tst: = rd & rs |
| NOTES | 1. All alu operations update the flags:  2. add, sub, neg, inc, and dec use v as signed overflow and c as signed overflow/carry.  3. The others, and, dna, or, xor, not, v and c are zero. n is bit 15 of the result and z is set if result is zero. |

### 1.2.8 Shift/Rotate

#### 1.2.8.1 Shift/Rotate Register

|  |  |
| --- | --- |
| MAP | (see paren below) dddd sisi aaaa shift |
| SYNTAX | (0000) lsr rd,rs  (0001) asr rd,rs  (0010) lsl rd,rs  (0011) ror rd,rs  (0100) rol rd,rs  (0101) rrc rd,rs  (0110) rlc rd,rs |
| FUNCTION | lsr: Shift bits in rd right rs positions, place rs 0’s in hob of rd. (shifted out lob go to bit bucket) (logical shift right)  asr: Save bit 15 of rd., Shift bits in rd right one position, place saved bit in bit 15, (shifted out lob goes to bit bucket), repeat rs-1 times (arithmetic shift right)  lsl: Shift bits in rd left rs positions, place rs 0’s in lob of rd. (shifted out hob go to bit bucket) (logical shift left)  ror: Save rs lob of rd, shift bits in rd right rs positions, place saved lob in hob of rd (rotate right)  rol: Save rs hob of rd, shift bits in rd left rs positions, place saved hob in lob of rd, (rotate left)  rrc: Save c, Place lob of rd in c, Shift bits in rd right one position, place saved c in hob, repeat rs-1 times (rotate right through carry)  rlc: Save c, Place hob of rd in c, Shift bits in rd left one position, place saved c in lob, repeat rs-1 times (rotate left through carry) |
| NOTES | 1. c = carry bit in r1  2. rs of 0 = noop  3. For register shifts rs = sisi  4. Only the lower 4 bits of rs are used for register shifts and rotates (number of shifts is 1-16) |

#### 1.2.8.2 Shift/Rotate Immediate

|  |  |
| --- | --- |
| MAP | (see paren below) dddd sisi aaaa shift |
| SYNTAX | (1000) lsr rd,#imm  (1001) asr rd,#imm  (1010) lsl rd,#imm  (1011) ror rd,#imm  (1100) rol rd,#imm  (1101) rrc rd,#imm  (1110) rlc rd,#imm |
| FUNCTION | lsr: Shift bits in rd right imm positions, place imm 0’s in hob of rd. (shifted out lob go to bit bucket) (logical shift right)  asr: Save bit 15 of rd. Shift bits in rd right one position, place saved bit in rd bit 15, (shifted out lob goes to bit bucket), repeat imm -1 times (arithmetic shift right)  lsl: Shift bits in rd left imm positions, place imm 0’s in lob of rd. (shifted out hob goes to bit bucket) (logical shift left)  ror: Save rs lob of rd, shift remaining bits in rd right rs positions, place saved lob in hob of rd (rotate right)  rol: Save rs hob of rd, shift remaining bits in rd left rs positions, place saved hob in lob of rd (rotate left)  rrc: Save c, Place lob of rd in c, Shift bits in rd right one position, place saved c in hob, repeat imm -1 times (rotate right through carry)  rlc: Save c, Place hob of rd in c, Shift bits in rd left one position, place saved c in lob, repeat imm -1 times (rotate left through carry) |
| NOTES | 1. For immediate shifts sisi = imm (number of shifts is 1-16)  2. imm of 0 = noop  3. c = carry bit in r1 |

### 1.2.9 Load Immed Low/High/Zero

|  |  |
| --- | --- |
| MAP | 1000 dddd iiiiiiii load immed low zero high |
| SYNTAX | llz rd,#imm |
| FUNCTION | rd{15:8} = 0  rd{7:0} = imm |

|  |  |
| --- | --- |
| MAP | 1001 dddd iiiiiiii load immed high zero low |
| SYNTAX | lhz rd,#imm |
| FUNCTION | rd{15:8} = imm  rd{7:0} = 0 |

|  |  |
| --- | --- |
| MAP | 1010 dddd iiiiiiii load immed low |
| SYNTAX | ll rd,#imm |
| FUNCTION | rd{15:8} = no change  rd{7:0} = imm |

|  |  |
| --- | --- |
| MAP | 1011 dddd iiiiiiii load immed high |
| SYNTAX | lh rd,#imm |
| FUNCTION | rd{15:8} = imm  rd{7:0} = no change |

### 1.2.10 Branch

|  |  |
| --- | --- |
| MAP | 1100 aaaa siiiiiii branch pc relative |
| SYNTAX | bz #imm |
| FUNCTION | If z =1, pc = (pc+1)+sign\_extend (imm) from a programmers perspective |

|  |  |
| --- | --- |
| MAP | (see paren below) aaaa dddd ssss branch register |
| SYNTAX | (0000) b,rs  (0001) bz,rs  (0010) bnz,rs  (0011) bc,rs  (0100) bnc,rs  (0101) bn,rs  (0110) bnn,rs  (0111) bv,rs  (1000) bnv,rs  (1001) bsg,rs  (1010) bsl,rs |
| FUNCTION | b: pc = rs (unconditional branch)  bz: if z = 1, pc = rs  bnz: if z = 0, pc = rs  bc: if c = 1, pc = rs (unsigned greater than or equal)  bnc: if c = 0, pc = rs (unsigned less than)  bn: if n=1, pc = rs  bnn: if n – 0, pc = rs  bv: if v = 1, pc = rs  bnv: if v = 0, pc = rs  bsg: if (n xor v) = 0, pc = rs (signed greater or equal)  bsl: if (n xor v) = 1, pc = rs (signed less than) |

### 1.2.11 Swap

|  |  |
| --- | --- |
| MAP | ???????????? |
| SYNTAX | Swap rd,rs |
| FUNCTION | rd{0:7} = rs{8:15}  rd {8:15} = rs{0:7} (swap halves) |

### 1.2.12 Call

Call is special, is only encoded using the branch register format

|  |  |
| --- | --- |
| MAP | 1101 aaaa dddd ssss branch register |
| SYNTAX | call rd,rs |
| FUNCTION | rd = pc  pc = rs |
| NOTES | 1. To return, use ldw pc,rs or b rs where rs is the rd from the call. |

### 1.2.13 Possible future instructions:

adc add with carry

sbb subtract with borrow

mul multiply takes three registers though

div divide takes three registers though

nor or with result inverted

nand and with result inverted

Called it dna because may want to add a bic and bis where the operand is a 4 bit number

bic rd,#imm clears the bit specified

bis rd,#imm sets the bit specified

## 1.3 Memory Mapping

[**Back to Contents**](#TOC)

The LSA processor is designed around a ram only model. Having 16 bit registers the memory space is limited to 64KWords (128KBytes)

### 1.3.1 Resets And Interrupts

Reset vectors start at address 0x0000

0x0000 reset

0x0001 interrupt (not supported yet, TODO) others are tbd

The reset/interrupt vector contains a branch instruction to the handler.

Implementations can then choose to use less memory, 12 bits for 4096 words of ram, etc.

Registers (uart, timers, etc) are in the 0xF000 range, maybe 0xE000 if that is not enough, so with a full 16 bit memory space 1/16th of that memory is not available.

### 1.3.2 Ram/Rom

Ideally a microcontroller will want to have the program in rom. A simple implementation would be on reset to have the rom copied over to ram then release reset on the processor.

Another implementation might be to have the lower half of memory, or a fraction thereof be ram and the upper half or portion be rom. Before booting the hardware needs to write the address of the entry point in rom (ideally the first rom location). The boot code would then add an interrupt vector if so desired.

# 2 Complete list of lsa supported instructions:

## 2.1 Load/store pc/sp immediate:

[**Back to Contents**](#TOC)

**Load pc relative immediate (rd = pc + imm)**

|  |
| --- |
|  |
| **lpc rd,#imm** |
|  |

**Load pc relative label (rd = pc + imm, imm = 8 lob of label\_name)**

|  |
| --- |
|  |
| **lpc rd,label\_name** |
|  |

**load sp** **relative immediate (rd = sp + imm)**

|  |
| --- |
|  |
| **lsp rd,#imm** |
|  |

**Store pc relative immediate (memory(pc+imm) = rs)**

|  |
| --- |
|  |
| **spc #imm,rs** |
|  |

**Store pc relative label(memory(pc+imm) = rs, imm = 8 lob of label\_name)**

|  |
| --- |
|  |
| **spc label\_name,rs** |
|  |

**Store sp relative immediate (memory(sp+imm) = rs)**

|  |
| --- |
|  |
| **ssp #imm,rs** |
|  |

## 2.2 Load/Store Word Incr/Decr

[**Back to Contents**](#TOC)

**Load word post increment rs, ( rd = memory[rs], rs=rs+1**

|  |
| --- |
|  |
| **ldw rd,[rs++] (ldw rd,[r2++] = pop)** |
|  |

**Load word pre increment rs, ( rs=rs+1, rd = memory [rs]**

|  |
| --- |
|  |
| **ldw rd,[++rs]** |
|  |

**Load word post decrement rs, ( rd = memory[rs], rs=rs-1**

|  |
| --- |
|  |
| **ldw rd,[rs--]** |
|  |

**Load word pre decrement rs, ( rs=rs-1, rd = memory [rs]**

|  |
| --- |
|  |
| **ldw rd,[--rs]** |
|  |

**Load word (rd = memory[rs])**

|  |
| --- |
|  |
| **ldw rd,[rs]** |
|  |

**Store word**

|  |
| --- |
|  |
| **stw [rd] ,rs** |
|  |

**Store word post increment rd,( memory[rd] = rs, rd=rd+1)**

|  |
| --- |
|  |
| **stw [rd++],rs** |
|  |

**Store word pre increment rd, ( rd=rd+1, memory [rd] =rs)**

|  |
| --- |
|  |
| **stw [++rd],rs** |
|  |

**Store word post decrement rd, ( memory[rd] = rs, rd=rd-1)**

|  |
| --- |
|  |
| **stw [rd--],rs** |
|  |

**Store word pre decrement rd, ( rd=rd-1, memory [rd] = rs)**

|  |
| --- |
|  |
| **stw [--rd],rs (stw [--r2],rs = push)** |
|  |

## 2.3 ALU instructions

[**Back to Contents**](#TOC)

**Add rd=rs+rd**

|  |
| --- |
|  |
| **add rd,rs** |
|  |

**Subtract ( rd=rd-rs)**

|  |
| --- |
|  |
| **sub rd,rs** |
|  |

**Logical And ( rd=rd&rs)**

|  |
| --- |
|  |
| **and rd,rs** |
|  |

**Logical reverse and ( rd=rd&(~rs) )**

|  |
| --- |
|  |
| **dna rd,rs** |
|  |

**Logical or ( rd=rd|rs)**

|  |
| --- |
|  |
| **or rd,rs** |
|  |

**Logical xor ( rd=rd^rs)**

|  |
| --- |
|  |
| **xor rd,rs** |
|  |

**Logical negative ( rd=0-rs)**

|  |
| --- |
|  |
| **neg rd,rs** |
|  |

**Logical not ( rd=~rs)**

|  |
| --- |
|  |
| **not rd,rs** |
|  |

**Increment ( rd=rs+1)**

|  |
| --- |
|  |
| **inc rd,rs** |
|  |

**Decrement ( rd=rs-1)**

|  |
| --- |
|  |
| **dec rd,rs** |
|  |

## 2.4 Shift/Rotate Instructions

[**Back to Contents**](#TOC)

**Logical shift right register(**[**See 1.2.8.1**](#_1.2.8.1_Shift/Rotate_Register)**)**

|  |
| --- |
|  |
| **lsr rd,rs** |
|  |

**Arithmetic shift right register(**[**See 1.2.8.1**](#_1.2.8.1_Shift/Rotate_Register)**)**

|  |
| --- |
|  |
| **asr rd,rs** |
|  |

**Logical shift left register(**[**See 1.2.8.1**](#_1.2.8.1_Shift/Rotate_Register)**)**

|  |
| --- |
|  |
| **lsl rd,rs** |
|  |

**Rotate right register(**[**See 1.2.8.1**](#_1.2.8.1_Shift/Rotate_Register)**)**

|  |
| --- |
|  |
| **ror rd,rs** |
|  |

**Rotate left register(**[**See 1.2.8.1**](#_1.2.8.1_Shift/Rotate_Register)**)**

|  |
| --- |
|  |
| **rol rd,rs** |
|  |

**Rotate right carry register(**[**See 1.2.8.1**](#_1.2.8.1_Shift/Rotate_Register)**)**

|  |
| --- |
|  |
| **rrc rd,rs** |
|  |

**Rotate left carry register(**[**See 1.2.8.1**](#_1.2.8.1_Shift/Rotate_Register)**)**

|  |
| --- |
|  |
| **rlc rd,rs** |
|  |

**Logical shift right immediate(**[**See 1.2.8.2**](#_1.2.8.2_Shift/Rotate_Immediate)**)**

|  |
| --- |
|  |
| **lsr rd,#imm** |
|  |

**Arithmetic shift right immediate(**[**See 1.2.8.2**](#_1.2.8.2_Shift/Rotate_Immediate)**)**

|  |
| --- |
|  |
| **asr rd,#imm** |
|  |

**Logical shift left immediate(**[**See 1.2.8.2**](#_1.2.8.2_Shift/Rotate_Immediate)**)**

|  |
| --- |
|  |
| **lsl rd,#imm** |
|  |

**Rotate right immediate(**[**See 1.2.8.2**](#_1.2.8.2_Shift/Rotate_Immediate)**)**

|  |
| --- |
|  |
| **ror rd,#imm** |
|  |

**Rotate left immediate(**[**See 1.2.8.2**](#_1.2.8.2_Shift/Rotate_Immediate)**)**

|  |
| --- |
|  |
| **rol rd,#imm** |
|  |

**Rotate right carry immediate(**[**See 1.2.8.2**](#_1.2.8.2_Shift/Rotate_Immediate)**)**

|  |
| --- |
|  |
| **rrc rd,#imm** |
|  |

**Rotate left carry immediate(**[**See 1.2.8.2**](#_1.2.8.2_Shift/Rotate_Immediate)**)**

|  |
| --- |
|  |
| **rlc rd,#imm** |
|  |

## 2.5 Load Immediate Low/High

[**Back to Contents**](#TOC)

**Load immediate Low , zero High (rd{15:8}= 0, rd{7:0} = imm)**

|  |
| --- |
|  |
| **llz rd,#imm** |
|  |

**Load immediate High , zero Low (rd{15:8}= imm, rd{7:0} = 0)**

|  |
| --- |
|  |
| **lhz rd,#imm** |
|  |

**Load immediate Low (rd{7:0} = imm, rd{15:8} = unchanged)**

|  |
| --- |
|  |
| **ll rd,#imm** |
|  |

**Load immediate High (rd{15:8}= imm, rd{7:0} = unchanged)**

|  |
| --- |
|  |
| **lh rd,#imm** |
|  |

## 2.6 Branch Instructions

[**Back to Contents**](#TOC)

**Branch unconditional immediate(pc = signed\_imm)**

|  |
| --- |
|  |
| **b #signed\_imm** |
|  |

**Branch if zero immediate (if z = 1, pc = signed\_imm)**

|  |
| --- |
|  |
| **bz #signed\_imm** |
|  |

**Branch if not zero immediate (if z = 0, pc = signed\_imm)**

|  |
| --- |
|  |
| **bnz #signed\_imm** |
|  |

**Branch if carry immediate (if c = 1, pc = signed\_imm) (unsigned greater or equal)**

|  |
| --- |
|  |
| **bc #signed\_imm** |
|  |

**Branch if not carry immediate (if c = 0, pc = signed\_imm) (unsigned less)**

|  |
| --- |
|  |
| **bnc #signed\_imm** |
|  |

**Branch if negative immediate (if n = 1, pc = signed\_imm)**

|  |
| --- |
|  |
| **bn #signed\_imm** |
|  |

**Branch if not negative immediate (if n = 0, pc = signed\_imm)**

|  |
| --- |
|  |
| **bnn #signed\_imm** |
|  |

**Branch if signed overflow immediate(if v = 1, pc = signed\_imm)**

|  |
| --- |
|  |
| **bv #signed\_imm** |
|  |

**Branch if not signed overflow immediate (if v = 0, pc = signed\_imm)**

|  |
| --- |
|  |
| **bnv #signed\_imm** |
|  |

**Branch signed greater or equal immediate (if n .xor. v) = 0 (n == v) , pc = signed\_imm)**

|  |
| --- |
|  |
| **bsg #signed\_imm** |
|  |

**Branch signed less than immediate (if n .xor. v) = 1 (n!= v) , pc = signed\_imm)**

|  |
| --- |
|  |
| **bsl #signed\_imm** |
|  |

**Branch unconditional register (pc=rs)**

|  |
| --- |
|  |
| **b rs** |
|  |

**Branch if zero register (if z = 1, pc=rs)**

|  |
| --- |
|  |
| **bz rs** |
|  |

**Branch if not zero register (if z = 0, pc=rs)**

|  |
| --- |
|  |
| **bnz rs** |
|  |

**Branch if carry register (if c = 1, pc=rs) (unsigned greater or equal)**

|  |
| --- |
|  |
| **bc rs** |
|  |

**Branch if not carry register (if c = 0, pc=rs) (unsigned less)**

|  |
| --- |
|  |
| **bnc rs** |
|  |

**Branch if negative register(if n = 1, pc=rs)**

|  |
| --- |
|  |
| **bn rs** |
|  |

**Branch if not negative register (if n = 0, pc=rs)**

|  |
| --- |
|  |
| **bnn rs** |
|  |

**Branch if signed overflow register(if v = 1, pc=rs)**

|  |
| --- |
|  |
| **bv rs** |
|  |

**Branch if not signed overflow register (if v = 0, pc=rs)**

|  |
| --- |
|  |
| **bnv rs** |
|  |

**Branch signed greater or equal register( if (n .xor. v) = 0 (n == v) , pc=rs)**

|  |
| --- |
|  |
| **bsg rs** |
|  |

**Branch signed less than register (if (n .xor. v) = 1 (n!= v) , pc=rs)**

|  |
| --- |
|  |
| **bsl rs** |
|  |

## 2.7 Misc Instructions

[**Back to Contents**](#TOC)

**Call (rd = pc, pc=rs)**

|  |
| --- |
|  |
| **call rd,rs** |
|  |

**Stop**

|  |
| --- |
|  |
| **Halt** |
|  |

**Compare (rd = rd - rs)**

|  |
| --- |
|  |
| **cmp rd,rs** |
|  |

**Test (rd = rd&rs)**

|  |
| --- |
|  |
| **tst rd,rs** |
|  |

**Swap(rd{0:7} = rs{8:15}, rd{8:15} = rs{0:7})**

|  |
| --- |
|  |
| **swap rd,rs** |
|  |

**Move (rd=rs)**

|  |
| --- |
|  |
| **mov rd,rs both low (r0-r15)** |
|  |

**Move (rd=rs)**

|  |
| --- |
|  |
| **mov rd,rs one high (r0-r127) one low (r0-r15) (optional instruction, might not be implemented)** |
|  |