Lanai Instruction Set

This document describes the instruction set of the Lanai CPU.

Table of Contents

General Information

The Lanai is a pipelined, RISC-style, load-store, 32-bit processor.

In the remainder of this specification, we shall refer to 8-bit data units as bytes, to 16-bit units as half-words, and to 32-bit units as words. Pointers to successive words differ by 4, pointers to half-words differ by 2, and pointers to bytes differ by 1. All instructions are 1 word long. All word addresses must be word-aligned. All half-word addresses must be half-word-aligned. Any least-significant bits of an address that would make a memory access non-aligned are ignored. Memory storage is big-endian (most-significant byte is stored at the lowest byte address).

Registers

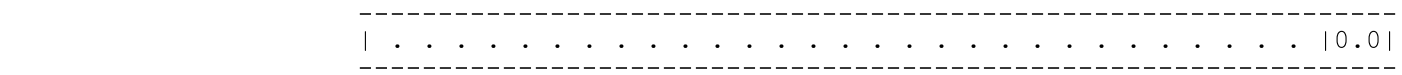
Registers are denoted as `rI', where 0 <= I < 32.

`r0'-`r2'are special registers:

`r0'
contains 0. `r0' may be the destination register of an instruction; if so, the result is discarded.

`r1'
contains 0xFFFFFFFF. `r1' may be the destination register of an instruction; if so, the result is discarded.

`r2'
is the program counter (`pc'):



Instructions that specify `pc' as the destination register modify `pc' in the same way and at the same time as any other register, but they also affect the execution flow. If not written by an instruction during a time-step, the program counter increments by four. The two low-order bits are hard-wired to zero.

`r3-r31'
are general-purpose registers.

The Machine Instruction Set

Concise Encoding Summary

The Lanai machine instructions are encoded as follows, and described in detail below.

```
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+---------------------------------------------------------------+
|0.A.A.A|    Rd   |   Rs1   |F.H| . . . . . . . . . . . . . . . | RI
|1.0.0.S|    Rd   |   Rs1   |P.Q| . . . . . . . . . . . . . . . | RM
|1.0.1.S|    Rd   |   Rs1   |P.Q|   Rs2   |B.B.B|J.J.J.J.J|Y.L.E| RRM
|1.1.0.0|    Rd   |   Rs1   |F.I|   Rs2   |B.B.B|J.J.J.J.J|D.D.D| RR
|1.1.0.1|    Rd   |   Rs1   |F.-|-.-.-.-.-.-.-.-.-.-.-.-.-.-.-|0.1| POPC
|1.1.0.1|    Rd   |   Rs1   |F.-|-.-.-.-.-.-.-.-.-.-.-.-.-.-.-|1.0| LEADZ
|1.1.0.1|    Rd   |   Rs1   |F.-|-.-.-.-.-.-.-.-.-.-.-.-.-.-.-|1.1| TRAILZ
|1.1.1.0|D.D.D| . . . . . . . . . . . . . . . . . . . . . . . |0.I| BR
|1.1.1.0|D.D.D|0.-| dst_reg |-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-|1.I| SCC
|1.1.1.0|D.D.D|1.-|   Rs1   | . . . . . . . . . . . . . . . |1.I| BRR
|1.1.1.1|    Rd   | . . . . |0.S| . . . . . . . . . . . . . . . | SLS
|1.1.1.1|    Rd   | . . . . |1.0| . . . . . . . . . . . . . . . | SLI
|1.1.1.1|    Rd   |   Rs1   |1.1.0.Y.S.E.P.Q| . . . . . . . . . | SPLS
```

The bits denoted by `-' are reserved for future extensions and should be set to 0.

Register Immediate (RI)

Encoding:

```
          ---------------------------------------------------------------
          |0.A.A.A| . . . . | . . . . |F.H| . . . . . . . . . . . . . . . |
          ---------------------------------------------------------------
           opcode     Rd        Rs1                constant (16)
```

Action:

```
          Rd <- Rs1 op constant
```

Except for shift instructions, `H' determines whether the constant is in the high (1) or low (0) word. The other halfword is 0x0000, except for the `AND' instruction (`AAA' = 100), for which the other halfword is 0xFFFF, and shifts (`AAA' = 111), for which the constant is sign extended.

`F' determines whether the instruction modifies (1) or does not modify (0) the program flags.

`AAA' specifies the operation: `add' (000), `addc' (001), `sub' (010), `subb' (011), `and' (100), `or' (101), `xor' (110), or `shift' (111). For the shift, `H' specifies a logical (0) or arithmetic (1) shift. The amount and direction of the shift are determined by the sign extended constant interpreted as a two's complement number. The shift operation is defined only for the range of:

```
     31 ... 0 -1 ... -31
     \      / \        /
       left     right
       shift    shift
```

If and only if the `F' bit is 1, RI instructions modify the condition bits, `Z' (Zero), `N' (Negative), `V' (oVerflow), and `C' (Carry), according to the result. If the flags are updated, they are updated as follows:

`Z'
: is set if the result is zero and cleared otherwise. As a special case, the ‘subb’ instruction will keep ‘Z’ cleared if it was cleared before instruction execution, and will compute it as usual if it was previously set (used to facilitate 64bit comparisons).

`N'
: is set to the most significant bit of the result.

`V'
: For arithmetic instructions (`add', `addc', `sub', `subb') `V' is set if the sign (most significant) bits of the input operands are the same but different from the sign bit of the result and cleared otherwise. For other RI instructions, `V' is cleared.

`C'
: For arithmetic instructions, `C' is set/cleared if there is/is_not a carry generated out of the most significant when performing the twos-complement addition (`sub(a,b) == a + ~b + 1', `subb(a,b) == a + ~b + `C''). For left shifts, `C' is set to the least significant bit discarded by the shift operation. For all other operations, `C' is cleared.

A Jump is accomplished by `Rd' being `pc', and it has one delay slot.

The all-0s word is the instruction `R0 <- R0 + 0', which is a no-op.

Register Register (RR)

Encoding:

```
          ---------------------------------------------------------------
          |1.1.0.0| . . . . | . . . . |F.I| . . . . |B.B.B|J.J.J.J.J|D.D.D|
          ---------------------------------------------------------------
           opcode     Rd        Rs1          Rs2   \       operation     /
```

Action:

```
     `Rd <- Rs1 op Rs2' iff condition DDDI is true (for op!=SELECT)
     `Rd <- (condition DDDI is true) ? Rs1 : Rs2   (for op==SELECT)
```

`DDDI’ is as described for the BR instruction.

`F' determines whether the instruction modifies (1) or does not modify (0) the program flags.

`BBB' determines the operation: `add' (000), `addc' (001), `sub' (010), `subb' (011), `and' (100), `or' (101), `xor' (110), or "special" (111). The `JJJJJ' field is irrelevant except for special.

`JJJJJ' determines which special operation is performed:
`10---' is a logical shift
`11---' is an arithmetic shift
`00000` is the SELECT operation

* The amount and direction of the shift are determined by the contents of `Rs2' interpreted as a two's complement number. Specifically, bits 31 and 4..0 of Rs2 are concatenated to form a 6-bit signed integer, which is interpreted as in the RI instruction.)
* For the SELECT operation, Rd gets Rs1 if condition DDDI is true, Rs2 otherwise.
* All other `JJJJJ' combinations are reserved for instructions that may be defined in the future.

If the `F' bit is 1, RR instructions modify the condition bits, `Z' (Zero), `N' (Negative), `V' (oVerflow), and `C' (Carry), according to the result. All RR instructions modify the `Z', `N', and `V' flags. Except for arithmetic instructions (`add', `addc', `sub', `subb'), `V' is cleared. Only arithmetic instructions and shifts modify `C'. Right shifts clear C.
Note that if the F bit is set, the condition bits are modified even if the (conditional) Rd assignment is not executed.

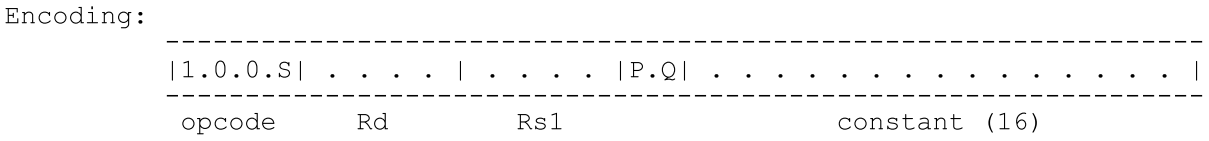
A Jump is accomplished by `Rd' being `pc', and it has one delay slot.

Conditional Jump support is LIMITED to the following cases:
- (op != SELECT) && (DDDI == 0000) - jump to (Rs1 op Rs2)
- (op == SELECT) - jump to either Rs1 or Rs2

If (op != SELECT) && (DDDI != 0000), the behavior is undefined.

Register Memory (RM)

Encoding:



Action:

```
Rd <- Memory(ea)      (Load)    see below for the
Memory(ea) <- Rd      (Store)   definition of ea.
```

`S' determines whether the instruction is a Load (0) or a Store (1).
If Rd of a Load instruction is used as a source register by any of the 3 immediately following instructions, pipeline will be stalled until Rd has been loaded. Sequential semantics are maintained.

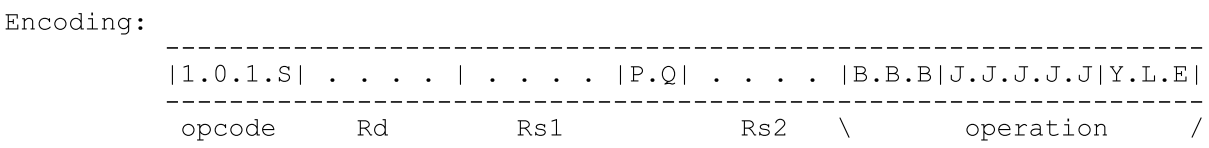
PQ	operation
00	ea = Rs1
01	ea = Rs1, Rs1 <- Rs1 + constant
10	ea = Rs1 + constant
11	ea = Rs1 + constant, Rs1 <- Rs1 + constant

The constant is sign-extended for this instruction.

A Jump is accomplished by `Rd' being `pc', and it has *two* delay slots.

Register Register Memory (RRM)

Encoding:



Action:

```
Rd <- Memory(ea)      (Load)    see below for the
Memory(ea) <- Rd      (Store)   definition of ea.
```

The RRM instruction is identical to the RM instruction except that:

1. `Rs1 + constant' is replaced with `Rs1 op Rs2', where `op' is determined in the same way as in the RR instruction (except that the SELECT operation is NOT supported), and

2. part-word memory accesses are allowed as specified below.

   If `BBB' != 111 (i.e.: For all but shift operations):
   If `YLE' = 01- => fuLl-word memory access
   If `YLE' = 00- => half-word memory access
   If `YLE' = 10- => bYte memory access
   If `YLE' = --1 => loads are zEro extended
   If `YLE' = --0 => loads are sign extended

   If `BBB' = 111 (For shift operations):
   fullword memory access are performed.

All part-word loads write the least significant part of the destination register with the higher-order bits zero- or sign-extended. All part-word stores store the least significant part-word of the source register in the destination memory location.

Conditional Branch (BR)

```
Encoding:
          ---------------------------------------------------------------
          |1.1.1.0|D.D.D| . . . . . . . . . . . . . . . . . . . . . . |0.I|
          ---------------------------------------------------------------
           opcode condition                  constant (23)

Action:
           if (condition) { `pc' <- 4*(zero-extended constant) }
```

The BR instruction is an absolute branch.
The constant is scaled as shown by its position in the instruction word such that it specifies word-aligned addresses in the range [0,2^25-4].

The `DDDI' field selects the condition that causes the branch to be taken (the `I' (Invert sense) bit inverts the sense of the condition):

DDDI	logical function	[code, used for...]
0000	1	[T, true]
0001	0	[F, false]
0010	C AND Z'	[HI, high]
0011	C' OR Z	[LS, low or same]
0100	C'	[CC, carry cleared]
0101	C	[CS, carry set]
0110	Z'	[NE, not equal]
0111	Z	[EQ, equal]
1000	V'	[VC, oVerflow cleared]
1001	V	[VS, oVerflow set]
1010	N'	[PL, plus]
1011	N	[MI, minus]
1100	(N AND V) OR (N' AND V')	[GE, greater than or equal]
1101	(N AND V') OR (N' AND V)	[LT, less than]
1110	(N AND V AND Z') OR (N' AND V' AND Z')	[GT, greater than]
1111	(Z) OR (N AND V') OR (N' AND V)	[LE, less than or equal]

The instruction after the branch instruction is executed regardless of whether the branch is taken or not, i.e. branch has one delay slot. Be very careful if you find yourself wanting to put a branch in a branch delay slot!

Conditional Branch Relative (BRR)

```
Encoding:
          ---------------------------------------------------------------
          |1.1.1.0|D.D.D|1|-|   Rs1   | . . . . . . . . . . . . . . . |1.I|
          ---------------------------------------------------------------
           opcode condition                  constant (16)
Action:
      if (condition) { ‘pc’ <- Rs1 + 4*sign-extended constant) }
```

BRR behaves like BR, except the branch target address is a 18-bit Rs1-relative offset.

Conditional Set (SCC)

```
Encoding:
          ---------------------------------------------------------------
          |1.1.1.0|D.D.D|0.-| dst_reg |-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-|1.I|
          ---------------------------------------------------------------
           opcode condition

Action:
           dst_reg <- logical function result
```

SCC sets dst_reg to the boolean result (0 or 1) of computing the logical function specified by DDDI, as described in the table for the BR instruction.

Special Load/Store (SLS)

```
Encoding:
          ---------------------------------------------------------------
          |1.1.1.1| . . . . | . . . . |0.S| . . . . . . . . . . . . . . . |
          ---------------------------------------------------------------
           opcode     Rd    addr 5msb's            address 16 lsb's

Action:
          If S = 0 (LOAD):   Rd <- Memory(address);
          If S = 1 (STORE):  Memory(address) <- Rd
```

The instruction pipeline behavior is the same as for RM and RRM instructions.
The two low-order bits of the 21-bit address are ignored. The address is zero extended. Fullword memory accesses are performed.

Special Load Immediate (SLI)

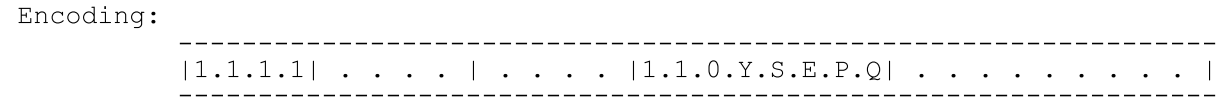
```
Encoding:
          ---------------------------------------------------------------
          |1.1.1.1| . . . . | . . . . |1.0| . . . . . . . . . . . . . . . |
          ---------------------------------------------------------------
           opcode     Rd    const 5msb's          constant 16 lsb's

Action:
          Rd <- constant
```

The 21-bit constant is zero-extended.

Special Part-Word Load/Store (SPLS)

Encoding:



Action:

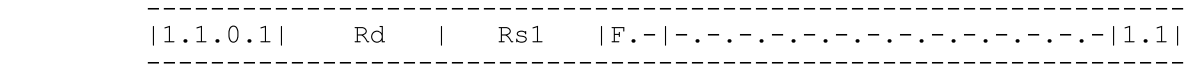
```
If `YS' = 11  (bYte     Store):
     Memory(ea) <- (least significant byte of Rr)
If `YS' = 01  (halfword Store):
     Memory(ea) <- (least significant half-word of Rr)
If `YS' = 10  (bYte     load):  Rr <- Memory(ea)
If `YS' = 00  (halfword load):  Rr <- Memory(ea)
     [Note: here ea is determined as in the the RM instruction. ]
If `SE' = 01 then the value is zEro extended
     before being loaded into Rd.
If `SE' = 00 then the value is sign extended
     before being loaded into Rd.
```

`P' and `Q' are used to determine `ea' as in the RM instruction. The constant is sign extended. The instruction pipeline behavior is the same as for RM and RRM instructions.

All part-word loads write the part-word into the least significant part of the destination register, with the higher-order bits zero- or sign-extended. All part-word stores store the least significant part-word of the source register into the destination memory location.

Count Trailing Zeros (TRAILZ)

Encoding:

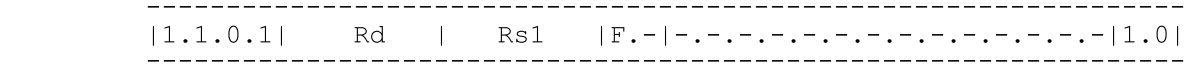


Action:

```
Rd <- Number of Trailing Zeroes in Rs1
```

Count Leading Zeros (LEADZ)

Encoding:

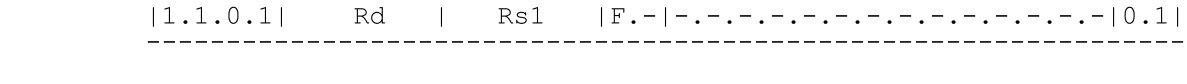


Action:

```
Rd <- Number of Leading Zeroes in Rs1
```

Population Count (POPC)

Encoding:



Action:

```
Rd <- Number of bits set in Rs1
```

Instruction Interpretation

The Lanai is a pipelined processor, so some instructions have delayed effects. From the programmer’s perspective, the Lanai generally appears to execute all operations in program order, except where the Program Counter is concerned.

- Branch instructions have one delay slot. (BR, BRR)
- Any move->’pc’ has one delay slot. (RI, RR, POPC, LEADZ, TRAILZ, SCC, SLI)
- Any load->’pc’ has two delay slots. (RM, RRM, SLS)
- For instructions that are neither in a delay-slot, nor a memory store,
  the PC register points at the current instruction. Other cases of using
  %pc are undefined by the lanai ISA.

This simple model is enough to understand most, if not all, compiler-generated assembly.

Writing the Program Counter

A write into PC (just like BR and BRR) has one delay slot, and a load into PC has two.

Consider the function return instruction sequence:

```
        ld -4[%fp],%pc   ; Load return address from stack (two delay slots)
        add %fp,0x0,%sp  ; Restore stack pointer
        ld -8[%fp],%fp   ; Restore frame pointer
```

Two instructions are executed after the load-into-pc.

The Lanai Assembler

General Assembler Info

Registers 0 to 31 are referred to, respectively, as `%r0' to `%r31'. Other recognized names are `%sp' (`%r4'), `%fp' (`%r5'), `%rv' (`%r8'), `%rca' (`%r15'). They stand for, respectively:

"`s'tack `p'ointer,"
"`f'rame `p'ointer,"
"`r'return `v'alue,"
"`r'egister for `c'onstant `a'ddresses."

Instructions modify the flags (Z, N, V, C) only when requested by the .f instruction option.

Operand Types

The following operand types may appear in assembly instructions:

BRABS
An unsigned, 23-bit, immediate absolute branch address or a register. Note that the 2 lsb's must be 0.

BROFF
A signed, 16-bit, immediate relative branch offset. Note that the 2 lsb's must be 0.

AND_CONST
A 32-bit unsigned constant with either the high or low halfword == 0xffff

CONST
A 32-bit unsigned constant with either the high or low halfword == 0

LCONST
A 21-bit unsigned constant

SIGNED_CONST_10
A 10 bit signed constant

SIGNED_CONST
A 16-bit signed constant

SHIFT_CONST
A 6-bit signed constant in the range [-31,31]

OP1
One of "`add', `addc', `sub', `subc', `and', `or', `xor', `sha'"

OP2
One of "`add', `addc', `sub', `subc', `and', `or', `xor', `sh', `sha'"

RDEST
A register

SRC1
A register

SRC2
A register

SRC3
A register

Negative constants may be written with a `-' sign or as a 32-bit constant, which will be truncated appropriately. For example, `-4' and `0xFFFFFFFC' are valid SHIFT_CONSTs.

Instruction Formats

In this section:

* Braces (`<>') indicate optional text. E.g.: "c<onc>at" matches "cat" or "concat" only.

* A vertical bar ("|") indicates a list of alternative matches. E.g.: "walk<s|ing|ed>" matches "walk", "walks", "walking", or "walked"<.

* Braces (`{}') indicate mandatory text. E.g.: "walk{s|ing|ed}" matches "walks", "walking", or "walked" only.

* For all instructions, unless otherwise specified, `pc' <- `pc' + 4 .

Instruction	Machine Instruction
add<.f> SRC1, CONST, RDEST RDEST <- SRC1 + CONST NOTES: (1),(4)	RI
add<.f> SRC1, SRC2, RDEST RDEST <- SRC1 + SRC2 NOTES: (1),(4)	RR

```
addc<.f> SRC1, SRC2, RDEST                          RI
        RDEST <- SRC1 + SRC2 + C
        NOTES: (1),(4) C is the carry flag from %ps
addc<.f> SRC1, CONST, RDEST                         RR
        RDEST <- SRC1 + CONST + C
        NOTES: (1),(4) C is the carry flag from %ps

and<.f> SRC1, AND_CONST, RDEST                      RI
        RDEST <- SRC1 & AND_CONST
        NOTES: (1),(4)
and<.f> SRC1, SRC2, RDEST                           RR
        RDEST <- SRC1 & SRC2
        NOTES: (1),(4)
b?? BRABS                                           BR
        if ( ?? condition is true )
        then %pc <- BRABS
        NOTES: (6)
b??.r BROFF                                         BRR
        if (?? condition is true)
        then %pc <- SRC1 + BROFF
        NOTES: (5), (6)
```

In the ``b??'' instructions above, ``b??'' must be replaced with one of the branch mnemonics in the table below. Each of these branch mnemonics specifies the conditions under which the branch is taken (see BR).

inst.	branch condition	branches if true
bt	true	1
bf	false	0
bhi|bugt	high	C AND Z'
bls|bule	low or same	C' OR Z
bcc|bult	carry clear	C'
bcs|buge	carry set	C
bne	not equal	Z'
beq	equal	Z
bvc	overflow cleared	V'
bvs	overflow set	V
bpl	plus	N'
bmi	minus	N
bge	greater than or equal	(N AND V) OR (N' AND V')
blt	less than	(N AND V') OR (N' AND V)
bgt	greater than	(N AND V AND Z') OR (N' AND V' AND Z')

```
Instruction                                          Machine Instruction
-------------------------------------------------------------------------
<u>ld SIGNED_CONST[SRC1], RDEST                      RM
        RDEST    <- mem(SRC1+SIGNED_CONST)
        NOTES: (2), (3)
<u>ld SRC2[SRC1], RDEST                              RRM
        RDEST    <- mem(SRC1+SRC2)
        NOTES: (2),(3)
<u>ld{.h|.b} SIGNED_CONST_10[SRC1], RDEST            SPLS
        RDEST    <- mem(SRC1+SIGNED_CONST_10)
        NOTES: (2),(3)

<u>ld SIGNED_CONST[*SRC1], RDEST                     RM
        RDEST    <- mem(SRC1+SIGNED_CONST)
        SRC1 <- SRC1+SIGNED_CONST
        NOTES: (2),(3)
ld [{--|++}SRC1], RDEST                              RM
        RDEST    <- mem(SRC1 {-|+} 4)
        SRC1 <- SRC1 {-|+} 4
        NOTES: (2),(3)
<u>ld.h [{--|++}SRC1], RDEST                         SPLS
        RDEST    <- mem(SRC1 {-|+} 2)
        SRC1 <- SRC1 {-|+} 2
        NOTES: (2),(3)
<u>ld.b [{--|++}SRC1], RDEST                         SPLS
        RDEST    <- mem(SRC1 {-|+} 1)
        SRC1 <- SRC1 {-|+} 1
        NOTES: (2),(3)
<u>ld SRC2[*SRC1], RDEST                             RRM
        RDEST    <- mem(SRC1+SRC2)
        SRC1 <- SRC1+SRC2
        NOTES: (2),(3)
<u>ld{.h|.b} SIGNED_CONST_10[*SRC1], RDEST           SPLS
        RDEST    <- mem(SRC1+SIGNED_CONST_10)
        SRC1 <- SRC1+SIGNED_CONST_10
        NOTES: (2),(3)

<u>ld SIGNED_CONST[SRC1*], RDEST                     RM
        RDEST    <- mem(SRC1)
        SRC1 <- SRC1+SIGNED_CONST
        NOTES: (2),(3)
ld [SRC1{--|++}], RDEST                              RM
        RDEST    <- mem(SRC1)
        SRC1 <- SRC1 {-|+} 4
        NOTES: (2),(3)
<u>ld.h [SRC1{--|++}], RDEST                         SPLS
        RDEST    <- mem(SRC1)
        SRC1 <- SRC1 {-|+} 2
        NOTES: (2),(3)
<u>ld.b [SRC1{--|++}], RDEST                         SPLS
        RDEST    <- mem(SRC1)
        SRC1 <- SRC1 {-|+} 1
        NOTES: (2),(3)
<u>ld SRC2[SRC1*], RDEST                             RRM
        RDEST    <- mem(SRC1)
        SRC1 <- SRC1+SRC2
        NOTES: (2),(3)
<u>ld{.h|.b} SIGNED_CONST_10[SRC1*], RDEST           SPLS
        RDEST    <- mem(SRC1)
        SRC1 <- SRC1+SIGNED_CONST_10
        NOTES: (2),(3)

ld [SRC1 OP2 SRC2], RDEST                            RRM
        RDEST    <- mem(SRC1 OP2 SRC2)
        NOTES: (2),(3)
ld [*SRC1 OP2 SRC2], RDEST                           RRM
```

```
        RDEST    <- mem(SRC1 OP2 SRC2)
        SRC1 <- SRC1 OP2 SRC2
        NOTES: (2),(3)
ld [SRC1* OP2 SRC2], RDEST                          RRM
        RDEST    <- mem(SRC1)
        SRC1 <- SRC1 OP2 SRC2
        NOTES: (2),(3)

ld [LCONST], RDEST                                  SLS
        RDEST <- mem(LCONST)
        NOTES: (2),(3)

popc SRC1, RDEST                                    POPC
        RDEST <- Number of bits set in SRC1

leadz SRC1, RDEST                                   LEADZ
        RDEST <- Number of Leading Zeroes in SRC1

trailz SRC1, RDEST                                  TRAILZ
        RDEST <- Number of Trailing Zeroes in SRC1

Instruction                                         Machine Instruction
-------------------------------------------------------------------
OP.CC[.f] SRC1, SRC2, RDEST                         RR
        if (condition(CC) == true)
            RDEST <- src1 op src2
        if (.f is specified)
                update flags.
  where OP is one of {add,addc,sub,subb,and,or,xor,sha,shl}
  and CC is one of {t,f,hi,ugt,ls,ule,cs,uge,ne,eq,vc,vs,pl,mi,ge,lt,gt}.
  RDEST must not be equal to PC unless CC is "t" (unconditional jump).

SEL.CC[.f] SRC1, SRC2, RDEST                        RR
        if (condition(CC) == true)
            RDEST <- src1
        else
            RDEST <- src2
        if (.f is specified)
                update flags.
  with CC one of {t,f,hi,ugt,ls,ule,cs,uge,ne,eq,vc,vs,pl,mi,ge,lt,gt}
  RDEST may be equal to PC.

mov CONST,SRC1                                      RI
        SRC1 <- CONST
        NOTES: (1)
mov SRC2,SRC1                                       RR
        SRC1 <- SRC2
        NOTES: (1)
mov LCONST,SRC1                                     SLI
        SRC1 <- LCONST
        NOTES: (1)
mov AND_CONST,SRC1                                  RI
        SRC1 <- AND_CONST
        NOTES: (1)

nop                                                 RI
        (does nothing)

or<.f> SRC1, CONST, RDEST                           RI
        RDEST <- SRC1 | SRC2
        NOTES: (1),(4)
or<.f> SRC1, SRC2, RDEST                            RR
        RDEST <- SRC1 | SRC2
        NOTES: (1),(4)

popc SRC1, RDEST                                    POPC
        RDEST <= Number of bits set in SRC1


s?? RDEST                                           SCC
        RDEST <- condition
        The condition is specified as for b??

sh<.f> SRC1, SHIFT_CONST, RDEST                     RI
        RDEST <- SRC1 << SHIFT_CONST
        NOTES: (1),(4), (7) logical shift performed
sh<.f> SRC1, SRC2, RDEST                            RR
        IF(31>=SRC2>=0)THEN
                RDEST <- SRC1 << SHIFT_CONST
        ELSE IF(0>SRC2>=-31)
                RDEST <- SRC1 >> -SHIFT_CONST
        ELSE
                result undefined
        NOTES: (1),(4), logical shift performed


Instruction                                         Machine Instruction
-------------------------------------------------------------------
sha<.f> SRC1, SHIFT_CONST, RDEST                    RI
        RDEST <- SRC1 << SHIFT_CONST
        NOTES: (1),(4), (7) arithmetic shift performed
sha<.f> SRC1, SRC2, RDEST                           RR
        IF(31>=SRC2>=0)THEN
                RDEST <- SRC1 << SHIFT_CONST
        ELSE IF(0>SRC2>=-31)
                RDEST <- SRC1 >> -SHIFT_CONST
        ELSE
                result is undefined
        NOTES: (1),(4)
st SRC1, SIGNED_CONST[SRC3]                         RM
        mem(SRC3+SIGNED_CONST) <- SRC1
        NOTES: (3)
st{.h|.b} SRC1, SIGNED_CONST_10[SRC3]               SPLS
        mem(SRC3+SIGNED_CONST_10) <- SRC1
        NOTES: (3)
st<.h|.b> SRC1, SRC2[SRC3]                          RRM
        mem(SRC3+SRC2) <- SRC1
        NOTES: (3)

st SRC1, SIGNED_CONST[*SRC3]                        RM
        mem(SRC3+SIGNED_CONST) <- SRC1
        SRC3 <- SRC3+SIGNED_CONST
        NOTES: (3)
```

```
st{.h|.b} SRC1, SIGNED_CONST_10[*SRC3]          SPLS
        mem(SRC3+SIGNED_CONST_10) <- SRC1
        SRC3 <- SRC3+SIGNED_CONST_10
        NOTES: (3)
st SRC1, [{--|++}SRC3]                          RM
        mem(SRC3 {-|+} 4) <- SRC1
        SRC3 <- SRC3 {-|+} 4
        NOTES: (3)
st.h    SRC1, [{--|++}SRC3]                     SPLS
        mem(SRC3 {-|+} 2) <- SRC1
        SRC3 <- SRC3 {-|+} 2
        NOTES: (3)
st.b    SRC1, [{--|++}SRC3]                     SPLS
        mem(SRC3 {-|+} 1) <- SRC1
        SRC3 <- SRC3 {-|+} 1
        NOTES: (3)
st<.h|.b> SRC1, SRC2[*SRC3]                     RRM
        mem(SRC3+SRC2) <- SRC1
        SRC3 <- SRC3+SRC2
        NOTES: (3)
```

```
Instruction                                     Machine Instruction
-------------------------------------------------------------------
st      SRC1, SIGNED_CONST[SRC3*]               RM
        mem(SRC3) <- SRC1
        SRC3 <- SRC3+SIGNED_CONST
        NOTES: (3)
st{.h|.b} SRC1, SIGNED_CONST[SRC3*]             SPLS
        mem(SRC3) <- SRC1
        SRC3 <- SRC3+SIGNED_CONST
        NOTES: (3)
st      SRC1, [SRC3{--|++}]                     RM
        mem(SRC3) <- SRC1
        SRC3 <- SRC3 {-|+} 4
        NOTES: (3)
st.h    SRC1, [SRC3{--|++}]                     SPLS
        mem(SRC3) <- SRC1
        SRC3 <- SRC3 {-|+} 2
        NOTES: (3)
st.b    SRC1, [SRC3{--|++}]                     SPLS
        mem(SRC3) <- SRC1
        SRC3 <- SRC3 {-|+} 1
        NOTES: (3)
st<.h|.b> SRC1, SRC2[SRC3*]                     RRM
        mem(SRC3) <- SRC1
        SRC3 <- SRC3+SRC2
        NOTES: (3)

st<.h|.b> RDEST, [SRC1 OP2 SRC2]                RRM
        mem(SRC1 OP2 SRC2) <- RDEST
st<.h|.b> RDEST, [*SRC1 OP1 SRC2]               RRM
        mem(SRC1 OP1 SRC2) <- RDEST
        src1 <- src1 op2 src2
st<.h|.b> RDEST, [SRC1* OP2 SRC2]               RRM
        mem(SRC1) <- RDEST
        src1 <- src1 op2 src2

st      RDEST, [LCONST]                         SLS
        mem(LCONST) <- RDEST
```

```
Instruction                                     Machine Instruction
-------------------------------------------------------------------
sub<.f> SRC1, CONST, RDEST                      RI
        RDEST <- SRC1 - CONST
        NOTES: (1),(4)
sub<.f> SRC1, SRC2, RDEST                       RR
        RDEST <- SRC1 - SRC2
        NOTES: (1),(4)

subb<.f> SRC1, CONST, RDEST                     RI
        RDEST <- SRC1 - CONST + C
        NOTES: (1),(4) C is the carry bit
subb<.f> SRC1, SRC2, RDEST                      RR
        RDEST <- SRC1 - SRC2 + C
        NOTES: (1),(4) C is the carry bit

xor<.f> SRC1, CONST, RDEST                      RI
        RDEST <- SRC1 XOR CONST
        NOTES: (1),(4)
xor<.f> SRC1, SRC2, RDEST                       RR
        RDEST <- SRC1 XOR SRC2
        NOTES: (1),(4)
```

Notes

1. If the destination register is `pc', one more instruction will be executed before execution continues at the location specified by the address stored to `pc' by this instruction.

2. a. Memory loads into `pc' have *two* delay slots. That is, two more instructions will be executed after the instruction performing the load before execution resumes at the location specified by the value loaded into `pc'.

   b. The byte loaded by a `ld.b' or the halfword loaded by a `ld.h' instruction is sign-extended to 32 bits before being saved in RDEST. The byte loaded by a `uld.b' or the halfword loaded

by a `uld.h' instruction is zero-extended to 32 bits before being saved in RDEST.

c. RDEST in a `ld' instruction is not changed until after the following instruction. For further information, see RM.

3. `.h' => halfword memory access
   `.b' => byte memory access
4. `.f' => modify the flags

5. Relative branches branch relative to the current `pc'. The current `pc' generally contains the address of the current instruction being executed.

6. Branches have a delay slot. See PC.

7. Here, a right shift is performed if SHIFT_CONST is negative.