1 Architectural Description

1.1. Registers

The processor contains fourteen 32-bit general registers (R0 - R13), one 32-bit Stack Pointer (R14 or SP), and one 32-bit Program Counter (R15 or PC).

The processor architecture also defines 16 special registers (P0 - P15), ten of which are implemented. The special registers are:

Mnemonic	Reg. no.	Description	Width
	P0	Constant zero register	8 bits
VR	P1	Version Register	8 bits
	P4	Constant zero register	16 bits
CCR	P5	Condition Code Register	16 bits
MOF	P7	Multiply Overflow register	32 bits
	P8	Constant zero register	32 bits
IBR	P9	Interrupt Base Register The upper 16 bits are implemented. The lower 16 bits are always zero.	32 bits
IRP	P10	Interrupt Return Pointer	32 bits
SRP	P11	Subroutine Return Pointer	32 bits
BAR	P12	Breakpoint Address Register This register contains an address for a hardware breakpoint. The breakpoint is enabled with the B flag.	32 bits
DCCR	P13	Dword Condition Code Register The lower 16 bits are the same as the CCR. The upper 16 bits are always zero.	32 bits
BRP	P14	Breakpoint Return Pointer This register contains the return address after a breakpoint, NMI instruction, or a hardware breakpoint.	32 bits
USP	P15	User mode Stack Pointer	32 bits

Table 1-1 Special Registers

Three of the unimplemented special registers (P0, P4 and P8) are reserved as “zero registers”. A read from any of those “registers” returns zero. A write to them has no effect. The zero registers are used implicitly by some instructions (e.g. CLEAR). You will never need to use the zero registers explicitly.

General Registers:

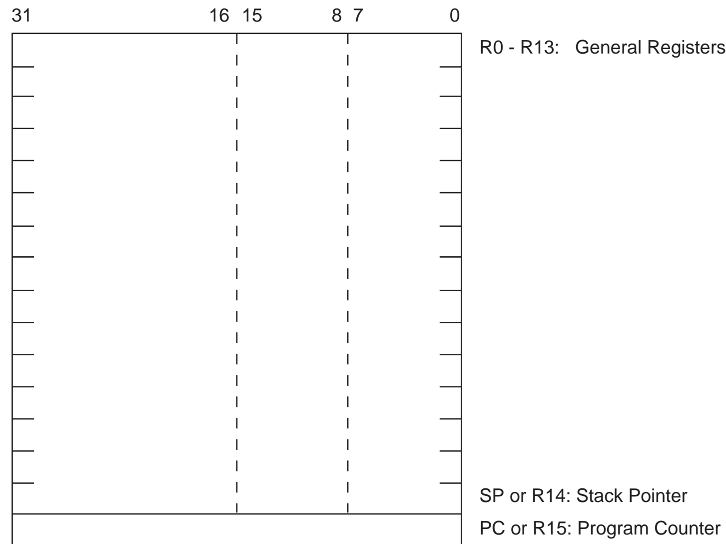

Figure 1-1 General Registers

Special Registers:

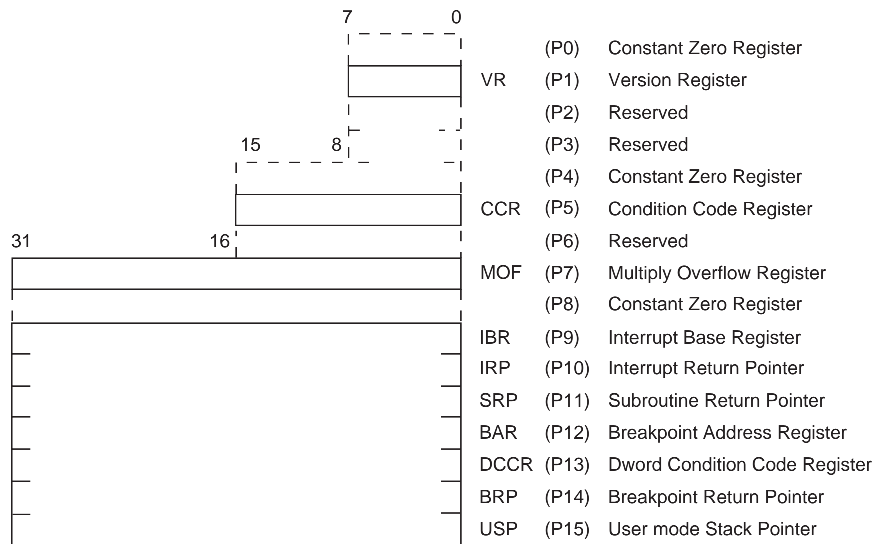

Figure 1-2 Special Registers

1.2. Flags and Condition Codes

The Condition Code Register (CCR) and Dword Condition Code Register (DCCR) for the ETRAX 100LX contain eleven different flags. The remaining bits are always zero:

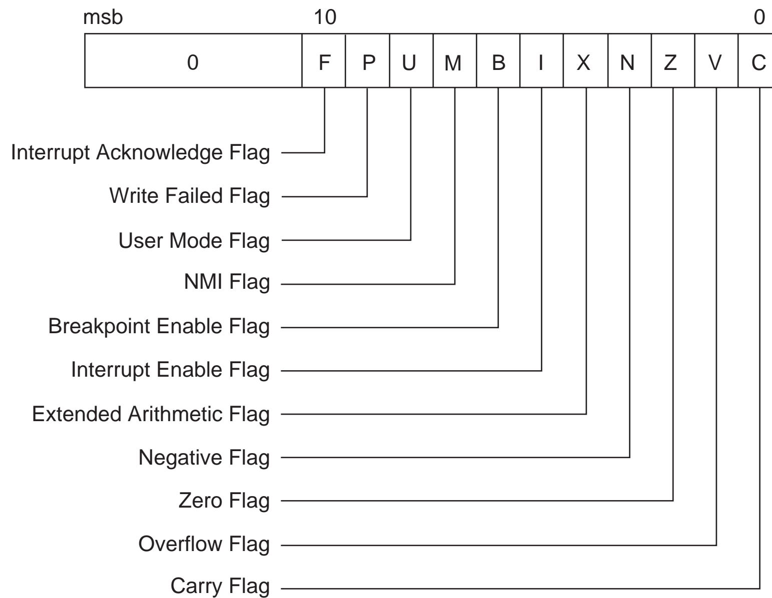

Figure 1-3 The Condition Code Register (CCR)/ Dword Condition Code Register (DCCR)

These flags can be tested using one of the 16 condition codes specified below:

Code	Alt	Condition	Encoding	Boolean function
CC	HS	Carry Clear	0000	$\overline{C}$
CS	LO	Carry Set	0001	C
NE		Not Equal	0010	$\overline{Z}$
EQ		Equal	0011	Z
VC		Overflow Clear	0100	$\overline{V}$
VS		Overflow Set	0101	V
PL		Plus	0110	$\overline{N}$
MI		Minus	0111	N
LS		Low or Same	1000	$C + Z$
HI		High	1001	$\overline{C} * \overline{Z}$
GE		Greater or Equal	1010	$N * V + \overline{N} * \overline{V}$
LT		Less Than	1011	$N * \overline{V} + \overline{N} * V$
GT		Greater Than	1100	$N * V * \overline{Z} + \overline{N} * \overline{V} * \overline{Z}$
LE		Less or Equal	1101	$Z + N * \overline{V} + \overline{N} * V$
A		Always true	1110	1
WF		Write Failed	1111	P

Table 1-2 Condition Codes

The behavior of the flags for different instructions is described in chapter 2. In those cases where the new value of the flag is not specified explicitly, the following applies:

General Case:

$N = R_{msb}$

$Z = \overline{R}_{msb} * \ldots * \overline{R}_{lsb} * (Z + \overline{X})$

Addition: (ADD, ADDQ, ADDS and ADDU)

$N = R_{msb}$

$Z = \overline{R}_{msb} * \ldots * \overline{R}_{lsb} * (Z + \overline{X})$

$V = S_{msb} * D_{msb} * \overline{R}_{msb} + \overline{S}_{msb} * \overline{D}_{msb} * R_{msb}$

$C = S_{msb} * D_{msb} + D_{msb} * \overline{R}_{msb} + S_{msb} * \overline{R}_{msb}$

Subtraction: (CMP, CMPQ, CMPS, CMPU, NEG, SUB, SUBQ, SUBS and SUBU)

$N = R_{msb}$

$Z = \overline{R}_{msb} * \ldots * \overline{R}_{lsb} * (Z + \overline{X})$

$V = \overline{S}_{msb} * D_{msb} * \overline{R}_{msb} + S_{msb} * \overline{D}_{msb} * R_{msb}$

$C = S_{msb} * \overline{D}_{msb} + \overline{D}_{msb} * R_{msb} + S_{msb} * R_{msb}$

Multiply: (MULS and MULU)

$N = MOF_{msb}$

$Z = \overline{MOF}_{msb} * \ldots * \overline{MOF}lsb * \overline{R}msb * \ldots * \overline{R}lsb * (Z + \overline{X})$

MULS: $V = ((MOF_{msb} + \ldots + MOF_{lsb}) * \overline{R}_{msb}) + ((\overline{MOF}_{msb} + \ldots + \overline{MOF}_{lsb}) * R_{msb})$

MULU: $V = MOFmsb + \ldots. + MOFlsb$

Bit Test: (BTST and BTSTQ)

$N = D_n$

$Z = \overline{D}_n * \ldots * \overline{D}_{lsb} * (Z + \overline{X})$

Move to Memory:

$P = F * X$

Move to CCR: (MOVE s, CCR and POP CCR)

F, P, U, B, I, N, Z, V, C are set according to source data.

X always cleared.

M not affected.

Condition Code Manipulation: (SETF and CLEARF)

B, I, X, N, V, C are set or cleared according to mask bits in the instruction.

M can be set by SETF, but not be cleared.

If X is not on the list, it is cleared.

F, P are cleared by CLEARF, but are not affected by SETF.

U is not affected.

Table 1-3 Flag Behavior

Explanations:

S_{msb} = Most significant bit of source operand

D_{msb} = Most significant bit of destination operand

D_n = Selected bit in the destination operand

D_{lsb} = Least significant bit of destination operand

R_{msb} = Most significant bit of result operand

R_{lsb} = Least significant bit of result operand

1.3. Data Organization in Memory

Data types supported by the CRIS are:

Name	Description	Size Modifier
Byte	8-bit integer	.B
Word	16-bit integer	.W
Dword	32-bit integer or address	.D

Table 1-4 Data Types Supported by the CRIS

Each address location contains one byte of data. Data is stored in memory with the least significant byte at the lowest address ("little endian"). The CRIS CPU in the ETRAX 100LX has a 32-bit wide data bus. A conversion from 32 bits to 16 bits is performed by the bus interface in the case of an external 16-bit data bus mode.

Data can be aligned to any address. If the data crosses a 32-bit boundary, the CPU will split the data access into two separate accesses. So, the use of unaligned word and dword data will degrade performance.

The figures below show examples of data organization with a 16-bit bus and a 32-bit bus:

Example of a structure layout:

```
struct example
{
   byte  a;
   byte  b;
   word  c;
   dword d;
   byte  e;
   word  f;
   dword g;
};
```

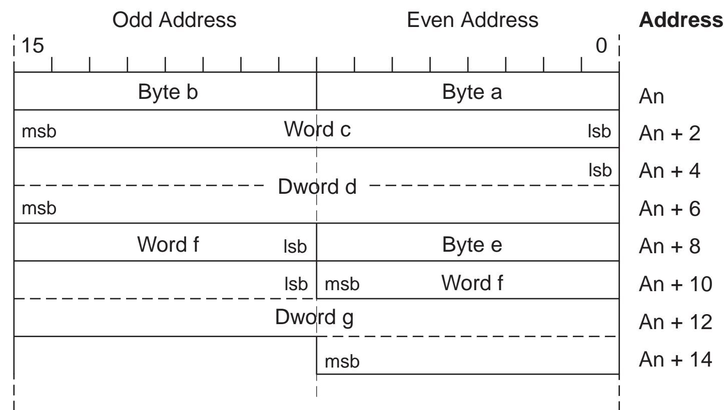

Figure 1-4 Data Organization with a 16-bit Bus

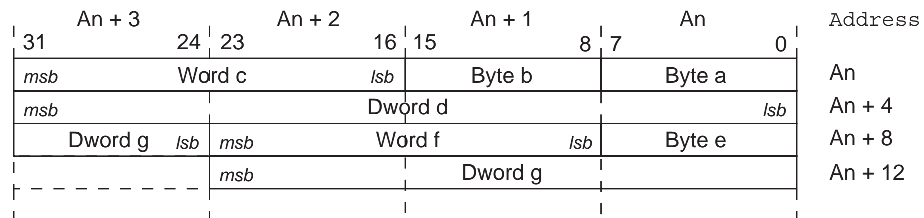

Figure 1-5 Data Organization with a 32-bit Bus

1.4. Instruction Format

The basic instruction word is 16 bits long, and instructions must be word (16 bits) aligned.

When the CPU fetches 32 bits, containing two 16-bit aligned instructions, it saves the upper two bytes in an internal prefetch register. Thus, the CPU will only perform one read for every second instruction when running consecutive code.

The most common instructions follow the same general instruction format:

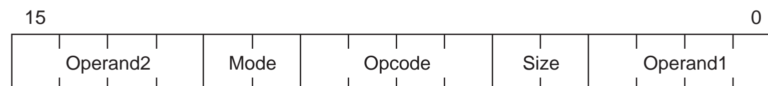

Figure 1-6 General Instruction Format

The basic instruction word can be combined with immediate data and/or Addressing mode prefix words to form more complex instructions, see *section 1.5. Addressing Modes.*

The Opcode field selects which instruction should be executed. For some opcodes, the meaning of the opcode is different depending on its Size and/or Mode field.

The Operand1 field selects one of the operands for the instruction, usually the source operand. Depending on the Mode field, the selected operand is either a general register or a memory location pointed to by the selected register.

The Operand2 field selects the other operand for the instruction, usually the destination operand. The selected operand can be a general or special register, or a condition code.

The Mode field specifies the addressing mode of the instruction. The Mode field affects only the operand of the Operand1 field. The following addressing modes can be specified within the basic instruction word:

Code	Mode
00	Quick immediate mode
01	Register mode
10	Indirect mode
11	Autoincrement mode

Table 1-5 The Mode Field of the Instruction Format

The Size field selects the size of the operation. For most of the instructions, the rest of the register is unaffected by the operation. Three different sizes are available:

Code	Size
00	Byte (8 bits)
01	Word (16 bits)
10	Dword (32 bits)

Table 1-6 The Size Field of the Instruction Format

The Size code 11 is used in conjunction with the Opcode field to encode special instructions that do not need different sizes.

1.5. Addressing Modes

1.5.1 General

The CRIS CPU has four basic addressing modes, which are encoded in the Mode field of the instruction word. The basic addressing modes are:

- Quick Immediate Mode
- Register Mode
- Indirect Mode
- Autoincrement Mode (with Immediate Mode as a special case)

More complex addressing modes can be achieved by combining the basic instruction word with an Addressing mode prefix word. The complex addressing modes are:

- Indexed
- Indexed with Assign
- Offset
- Offset with Assign
- Double Andirect
- Absolute

1.5.2 Quick Immediate Addressing Mode

In the Quick Immediate Addressing Mode, the size and Operand1 fields of the instruction are combined into a 6-bit Immediate value, extended to 32 bits, or interpreted as a 5-bit shift count.

The 6-bit immediate value may be sign or zero extended depending on the instruction.

Assembler syntax: `<expression>`
Example: `12`

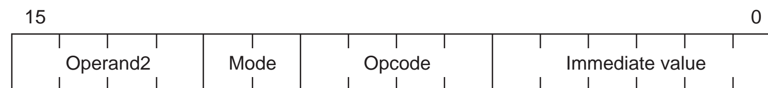

Figure 1-7 Quick Immediate Addressing Mode Instruction Format

1.5.3 Register Addressing Mode

In the Register Addressing Mode, the operand is contained in the register specified by the Operand1 or Operand2 field. The register can be a general register or a special register depending on the instruction.

General Register Addressing Mode

Assembler syntax: `Rn`
Example: `R6`

Special Register Addressing Mode

Assembler syntax: `Pn`
Example: `SRP`

1.5.4 Indirect Addressing Mode

In the Indirect Addressing Mode, the operand is contained in the memory location pointed to by the register specified by the Operand1 field.

Assembler syntax: `[Rn]`
Example: `[R6]`

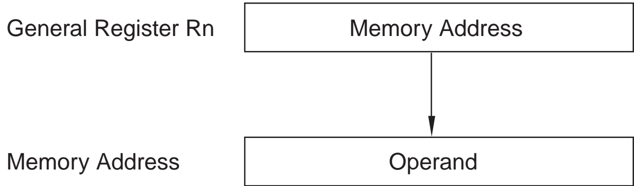

Figure 1-8 Indirect Addressing Mode

1.5.5 Autoincrement Addressing Mode

In the Autoincrement Addressing Mode, the operand is contained in the memory location pointed to by the register specified by the Operand1 field. After the operand address is used, the specified register is incremented by 1, 2 or 4, depending upon the size of the operand.

Assembler syntax: `[Rn+]`
Example: `[R6+]`

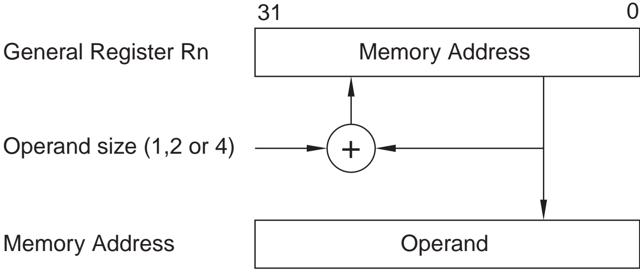

Figure 1-9 Autoincrement Addressing Mode

1.5.6 Immediate Addressing Mode

The Immediate Addressing Mode is a special case of the Autoincrement Addressing Mode, with PC as the address register. The immediate value follows directly after the instruction word. When the immediate data size is byte, PC will be incremented by 2 to maintain word alignment of instructions.

Assembler syntax: `<expression>`
Example: `325`

1.5.7 Indexed Addressing Mode

The Indexed Addressing Mode requires the basic instruction word to be preceded by one Addressing mode prefix word, formatted as shown below:

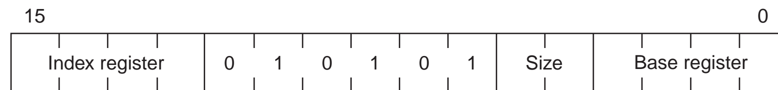

Figure 1-10 Indexed Addressing Mode Prefix Format

The address of the operand is the sum of the contents of the *Base register* and the shifted contents of the *Index register*. The contents of the Index register is shifted left 0, 1 or 2 steps depending upon the Size field of the Addressing mode prefix.

Note that the Size field of the Addressing mode prefix only affects the shift of the index value, not the size of the operand. The size of the operand is selected by the Size field of the basic instruction word.

When PC is used as the Base register, the value used will be the address of the instruction following the modified instruction. When PC is used as the Index Register, the value used will be the address of the modified instruction.

Assembler syntax: `[Rn + Rm.m]`
Example: `[R6 + R7.B]`

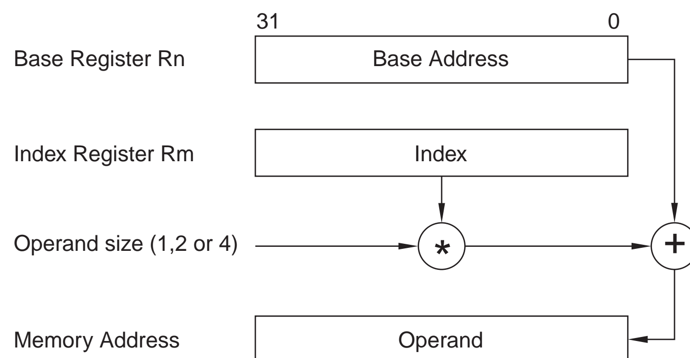

Figure 1-11 Indexed Addressing Mode

1.5.8 Indexed with Assign Addressing Mode

The Indexed with Assign Addressing Mode is similar to the Indexed Addressing Mode. The difference is that the resulting address not only selects the operand, but is also stored to a general register.

The Indexed with Assign Addressing Mode requires a prefix word of the same format as the Indexed Addressing Mode. The selection between Indexed Addressing and Indexed with Assign Addressing Mode is made by the mode field of the basic instruction word:

Code	Addressing Mode
10	Indexed
11	Indexed with assign

Table 1-7

Assembler syntax: `[Rp = Rn + Rm.m]`
Example: `[R8 = R6 + R7.B]`

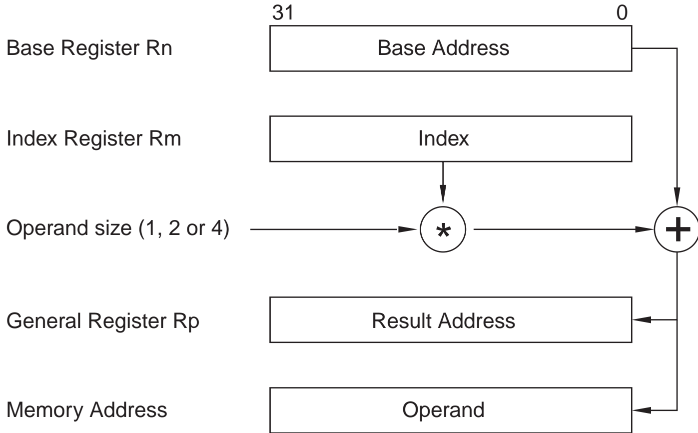

Figure 1-12 Indexed with Assign Addressing Mode

1.5.9 Offset Addressing Mode

This addressing mode requires the basic instruction word to be preceded by one Addressing mode prefix word. The general format for the prefix word is shown below:

15 ... 0

Base Register	1	i	0	1	0	1	Size	Offset

Figure 1-13 Offset Addressing Mode Prefix Format

The address of the operand is the sum of the contents of the Base register and a signed offset. In the general case, the offset is referenced with the indirect (md = 0) or autoincrement (md = 1) mode. The size of the offset can be byte, word or dword.

A special format is used for byte-sized immediate offsets. In this case, the offset is included in the prefix word:

15 ... 0

Base register	0	0	0	1	Signed offset

Figure 1-14 Immediate Byte Offset Addressing Mode Prefix Format

Word or dword sized immediate offsets use the general prefix format, with md = 1 and offset = PC. In this case, the immediate offset word(s) will be placed between the Prefix word and the Basic instruction word, see example below:

15 ... 0	Address
Prefix word	An
Offset (lsb)	An + 2
Offset (msb)	An + 4
Basic instruction word	An + 6

Figure 1-15 Instruction with Dword Sized Immediate Offset

When PC is used as the Base register, the value used will be the address of the Basic instruction word.

Immediate Offset Addressing Mode

Assembler syntax: [Rn + <expression>]
Example: [R6 + 27]

Indirect Offset Addressing Mode

Assembler syntax: [Rn + [Rm].m]
Example: [R6 + [R7].B]

Autoincrement Offset Addressing Mode

Assembler syntax: [Rn + [Rm+].m]
Example: [R6 + [R7+].B]

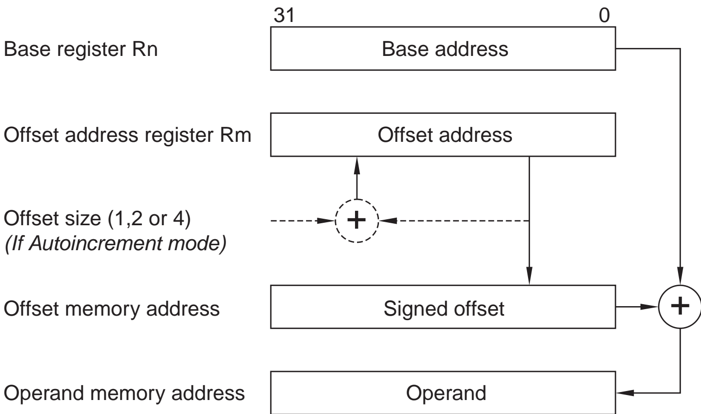

Figure 1-16 Offset Addressing Mode (general case)

1.5.10 Offset with Assign Addressing Mode

The Offset with assign addressing mode is similar to the Offset addressing mode. The difference is that the resulting address not only selects the operand, but is also stored to a general register.

The Offset with assign mode requires a prefix word of the same format as for the Offset mode. The selection between the Offset and the Offset with assign addressing mode is made by the Mode field of the basic instruction word:

Code	Addressing Mode
10	Offset
11	Offset with assign

Table 1-8

Immediate Offset with Assign Addressing Mode

Assembler syntax: [Rp = Rn + <expression>]
Example: [R8 = R6 + 27]

Indirect Offset with Assign Addressing Mode

Assembler syntax: [Rp = Rn + [Rm].m]
Example: [R8 = R6 + [R7].B]

Autoincrement Offset with Assign Addressing Mode

Assembler syntax: [Rp = Rn + [Rm+].m]
Example: [R8 = R6 + [R7+].B]

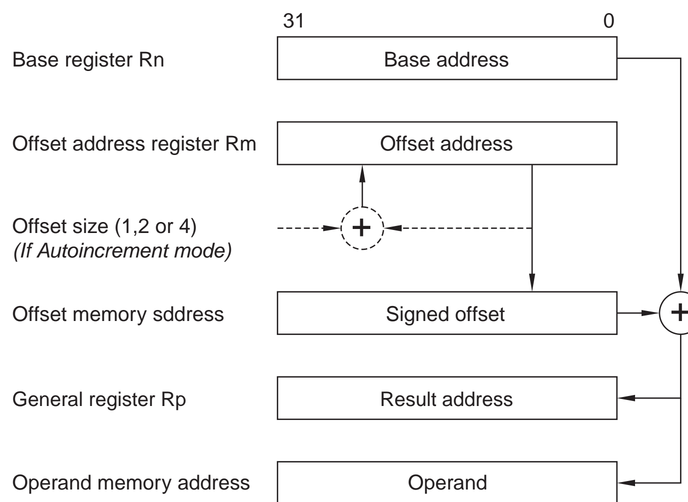

Figure 1-17 Offset with Assigned Addressing Mode (general case)

1.5.11 Double Indirect Addressing Mode

The Double indirect addressing mode requires the basic instruction word to be preceded by one Addressing mode prefix word, formatted as shown below:

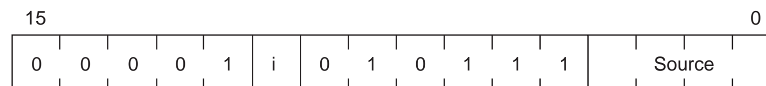

Figure 1-18 Double Indirect Addressing Mode Prefix Format

In the Double indirect addressing mode, the register specified by the Source field of the prefix word points to a memory address that contains the address of the operand. The specified register may be left unchanged (md = 0) or incremented by 4 after it is used (md = 1).

Double Indirect Addressing Mode

Assembler syntax: `[[Rn]]`
Example: `[[R6]]`

Double Indirect with Autoincrement Addressing Mode

Assembler syntax: `[[Rn+]]`
Example: `[[R6+]]`

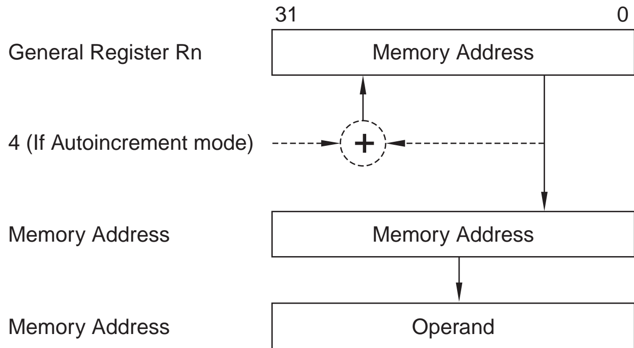

Figure 1-19 Double Indirect Addressing Mode

1.5.12 Absolute Addressing Mode

The Absolute Addressing Mode is a special case of the Double Indirect with Autoincrement Mode, with PC as the source register. The Absolute address will be placed between the Prefix word and the Basic instruction word:

Assembler syntax: `[<expression>]`
Example: `[3245]`

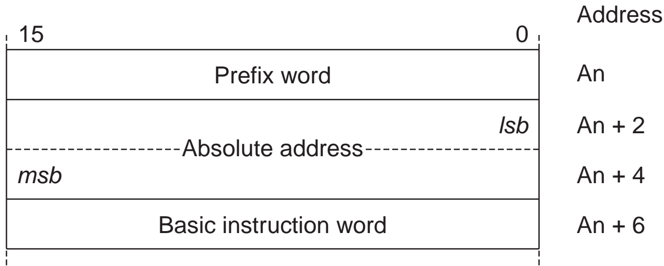

Figure 1-20 Instruction with Absolute Address

1.5.13 Multiple Addressing Mode Prefix Words

The CRIS CPU is designed to accept multiple consecutive addressing mode prefix words, where the calculated address from the first Prefix word replaces the Operand1 field of the second Prefix word. This can be done in an unlimited number of levels.

The addressing modes resulting from consecutive prefix words are not supported by the assembler or the disassembler.

1.6. Branches, Jumps and Subroutines

1.6.1 Conditional Branch

The *Bcc* instruction (where cc represents one of the 16 condition codes described in section 1.2) is a conditional relative branch instruction. If the specified condition is true, a signed immediate offset is added to the PC.

The Bcc instruction exists in two forms, one with an 8-bit offset contained within the basic instruction word, and one with a 16-bit immediate offset following directly after the instruction word. The assembler automatically selects between the 8-bit offset and the 16-bit offset form.

The Bcc instruction is a delayed branch instruction. This means that the instruction following directly after the Bcc instruction will always be executed, even if the branch is taken. The instruction position following the Bcc instruction is called a delay slot.

Example:

```
    :
    MOVEQ     4,R0
LOOP:
    BNE       LOOP
    SUBQ      1,R0          ; Delay slot instruction, executed
                            ; even if the branch is taken.
    :
```

The branch to LOOP will be taken 4 times, and register R0 decremented by 1 after each turn. After leaving the loop, R0 will have the value -1.

There are some restrictions as to which instructions can be placed in the delay slot. Valid instructions for the delay slots are all instructions except:

- Bcc
- BREAK/JBRC/JIR/JIRC/JMPU/JSR/JSRC/JUMP
- RET/RETB/RETI
- Instructions using Addressing mode prefix words.
- Immediate addressing other than Quick Immediate

The maximum offset range that can be reached by the Bcc instruction directly is -32768 - +32766. If a larger offset is needed, the branch must be combined with a jump to reach the branch target. The assembler resolves this situation automatically, and inserts the necessary code. The assembler can optionally give a warning message each time it makes this adjustment.

1.6.2 Jump instructions

The *JUMP* instruction is an unconditional absolute jump instruction. This instruction can be used with all different addressing modes described in section 1.5. *Addressing Modes*, except Quick Immediate. The resulting operand is taken as the jump target address, and is stored to PC.

Examples:

```
JUMP      R3          ; Jump target is the address contained
                      ; in register R3.

JUMP      346         ; Jump to address 346.

JUMP      [346]       ; Read jump target address from memory
                      ; address 346.

JUMP      [SP+]       ; Pop jump target address from stack.
                      ; This is useful as a subroutine
                      ; return instruction, see 1.6.5.

JUMP      [PC+R3.D]   ; Jump via jump table. The contents of
.DWORD    L0          ; register R3 is used as an index for
.DWORD    L1          ; the table.
:
.DWORD    Ln
```

The *JMPU* instruction is similar to JUMP except that JMPU causes a transition to user mode if the U flag is set, while JUMP never affects the operation mode. JMPU can not be used with the register addressing mode.

In contrast to the Bcc instruction, the JMPU and JUMP instructions take action immediately.

1.6.3 Implicit jumps

For many of the instructions in the CRIS instruction set, PC can be specified as the destination operand. When PC is used in this way, the result of the instruction will act as a jump target address.

The CPU will, in this case, require an extra execution cycle to compute the new address, but the instruction following the implicit jump instruction will not be executed.

The most useful instructions for implicit jumps are ADD, ADDS, ADDU, SUB, SUBS and SUBU, which result in unconditional relative jumps, see example in 1.6.4.

The following instructions *do not* support PC as the destination operand:

ADDI,	BOUND,	DSTEP,	LSL,	LSLQ,	LSR,
LSRQ,	MSTEP,	MULS,	MULU,	NEG,	NOT,
Scc,	SWAP				

1.6.4 Switches and Table Jumps

A common element in many high level languages is the *switch* statement. A typical switch construct in C can look like this:

```
switch (sel_val)

{
    case 6:
                a = b + c;
                break;
    case 7:
                d = a * (c - b) + 2;
                break;
    case 8:
                b = a + c + d;
                break;
    default:
                c = a + b;
                break;
}
```

A switch construct in the CRIS assembler can be implemented in several different ways. Two examples based on jump tables are shown below. The first example uses a table of absolute addresses, the second example one uses relative addressing.

Example of a switch construct with a table of absolute addresses:

```
        MOVE       [sel_val],R0    ; Load selector value to R0.
        SUBQ       6,R0            ; Adjust table index by subtracting
                                   ; the lowest selector value.
        BOUND.D    3,R0            ; Adjust index to point to the default
                                   ; case if it is out of range.
        JUMP       [PC+R0.D]       ; Table jump:
        .DWORD     L6              ; Address to case 6
        .DWORD     L7              ; Address to case 7
        .DWORD     L8              ; Address to case 8
        .DWORD     L_DEF           ; Address to default case
L6:
        :
        (Perform case 6)
        :
        BA         L_END           ; Break
        Op or NOP                  ; Delay slot
L7:
        :
        (Perform case 7)
        :
        BA         L_END           ; Break
        Op or NOP                  ; Delay slot
L8:
        :
        (Perform case 8)
        :
        BA         L_END           ; Break
        Op or NOP                  ; Delay slot
L_DEF:
        :
        (Perform default case)
        :
L_END:
```

Example of a switch construct with a table of relative addresses (this is the model used by the CRIS GNU C Compiler):

```
        MOVE       [sel_val],R0       ; Load selector value to R0.
        SUBQ       6,R0               ; Adjust table index by subtracting
        BOUND.D    3,R0               ; the lowest selector value.
        ADDS.W     [PC+R0.W],PC       ; Adjust index to point to the default
                                      ; case if it is out of range.
                                      ; Implicit relative table jump:
L_TABLE:
        .WORD      L6 - L_TABLE       ; Address to case 6
        .WORD      L7 - L_TABLE       ; Address to case 7
        .WORD      L8 - L_TABLE       ; Address to case 8
        .WORD      L_DEF - L_TABLE    ; Address to default case
L6:
        :
        (Perform case 6)
        :
        BA         L_END              ; Break
        Op or NOP                     ; Delay slot
L7:
        :
        (Perform case 7)
        :
        BA         L_END              ; Break
        Op or NOP                     ; Delay slot
L8:
        :
        (Perform case 8)
        :
        BA         L_END              ; Break
        Op or NOP                     ; Delay slot
L_DEF:
        :
        (Perform default case)
        :
L_END:
```

1.6.5 Subroutines

The JSR instruction of the CRIS CPU does not automatically push the return address for a subroutine on the stack. Instead, the return address is stored in a special register called the Subroutine Return Pointer (SRP).

For terminal subroutines (subroutines that do not call other subroutines), the return address can be kept in the SRP throughout the subroutine. In this way, the overhead for a subroutine call can be reduced to two single-cycle instructions.

For non-terminal subroutines, the contents of the SRP must be explicitly pushed on the stack. It is preferred that this is done as the first instruction of the subroutine.

This method results in two different ways of returning from a subroutine. Note that the RET instruction is a delayed jump with one delay slot, but the JUMP instruction is performed immediately. See examples below:

Terminal Subroutine

```
SUB_ENTRY:
    :                              ; Pushing of SRP is not needed.
    :
    (Perform desired function)
    :
    :
    RET                            ; Return: Take address from SRP.
    Op or NOP                      ; Delay slot after return.
```

Non-terminal Subroutine

```
SUB_ENTRY:
    PUSH        SRP                ; Pushing of SRP on to the stack.
    :
    (Perform desired function)
    :
    :
    JUMP        [SP+]              ; Return: Take address from stack.
```

1.6.6 The JBRC, JIRC and JSRC Subroutine Instructions

The subroutine instruction, Jump to Subroutine with Context (JSRC), adds 4 to the return address stored to the SRP register. This leaves four bytes unused between the JSRC instruction and the return point. These four bytes can, for example, be used for C++ exception handling information.

Figure 1-21 The JSRC Instruction

In the case of immediate addressing, the unused bytes are placed after the immediate value:

Figure 1-22 Immediate Addressing of JSRC

The Jump to Breakpoint Routine with Context (JBRC) instruction, and the Jump to Interrupt Routine with Context (JIRC) instruction act just like JSRC except that instead of storing the return address to the SRP register, JBRC stores the return address to the BRP register, and JIRC stores the return address to the IRP register.

1.7. MMU Support

1.7.1 Overview

To support the Memory Management Unit (MMU) incorporated with the ETRAX 100LX, a number of features have been included in the CRIS architecture:

- The CPU can be in one of two different operation modes: User mode and Supervisor mode. The MMU uses the operation mode to select the appropriate mapping between logical and physical addresses.

- The Bus fault is a mechanism that can interrupt the CPU in any cycle, not only at instruction boundaries. This is needed because the MMU can get a page miss in any cycle. The bus fault mechanism also gives a straightforward way to include single step capability.

- With the introduction of the bus fault mechanism, integral read-write operations can not be achieved by just disabling the interrupt. Instead, another method is used, see section 1.13. *Integral Read-Write Operations*.

The user and supervisor modes have different stack pointers. In both modes, the user mode stack pointer can be referenced as USP, while the currently active stack pointer is referenced as SP (or R14). Thus, in user mode, SP and USP refer to the same register while in supervisor mode, they are separate registers.

Note that the U flag does not indicate the current mode. The U flag is set by bus faults, interrupts, and BREAK instructions depending on the preceding mode. It is used by the instructions that affect the operation mode (JMPU, RBF, RETB, and RETI) to determine which mode will be selected.

The following CRIS instructions are included specifically for MMU support:

- SBFS (Save Bus Fault Status)
- RBF (Return from Bus Fault)
- JMPU (Jump, set user mode if U flag is set)

The SBFS and RBF instructions are used at the entry and exit of the bus fault interrupt routine. They save and restore a 16 byte CPU status record containing the information necessary to resume the operation that was interrupted by the bus fault.

JMPU is intended for return from ordinary interrupt routines where the IRP (or BRP) has been pushed on the stack. By looking at the U flag, JMPU can return to the operation mode that was valid before the interrupt occurred. In the case where the return address from the interrupt routine is kept in the IRP or BRP register, the RETI or RETB instructions will, in the same way, return to the correct operation mode.

This document only describes the CRIS CPU architecture features for MMU support. For information about the ETRAX 100LX Memory Management Unit itself, and for the single step capability, see the ETRAX 100LX Designer’s Reference Manual.

These MMU support features are not available in CRIS implementations prior to the ETRAX 100LX.

1.7.2 Protected registers and flags

A few registers and flags need to be protected from being modified while the CPU is in user mode. The protected registers and flags are:

- IBR (Interrupt Base Register)
- BAR (Breakpoint Address Register)
- M flag (NMI Enable Flag)
- B flag (HW Breakpoint Enable Flag)
- I flag (Interrupt Enable Flag)

An attempt to modify a protected register while in user mode will just be silently denied. It will not cause any exception. The protected registers are readable in both user and supervisor modes.

1.7.3 Transition Between Operation Modes

A transition between the user and supervisor modes can take place for the following reasons:

Transition to User Mode:

- JMPU with the U flag set
- RBF with the U flag set
- RETI with the U flag set
- RETB with the U flag set

Transition to Supervisor Mode:

- System reset
- BREAK instruction
- Interrupt (including NMI and HW break)
- Bus fault

The stack pointers will be automatically exchanged at a transition between the user and supervisor modes.

1.7.4 Bus fault sequence

When an external unit (e.g. MMU) signals a Bus Fault, the CPU will interrupt immediately at the end of the CPU clock cycle and enter a Bus Fault sequence.

The Bus Fault sequence is similar to the ordinary interrupt sequence, see section 1.8. *Interrupts*.The steps in the sequence are:

1. Bus Fault INTA cycle. This cycle will be an idle bus cycle. The following is a pseudo code description of the bus fault INTA cycle operations:

```
if (current mode == user mode)

{

  U flag = 1;

  Exchange stack pointers;

}

else

{

  U flag = 0;

}

current mode = supervisor mode;

F flag = 1;

hidden CPU status registers = current CPU status;
```

2. Interrupt vector read cycle. In this cycle the CPU will read the interrupt vector for the Bus Fault interrupt routine. If the bus fault was caused by the single step unit, the interrupt vector number will be 0x20, otherwise it will be 0x2e. If both the MMU and single step bus fault occur at the same time, single step will have priority.

3. Start execution of the Bus Fault interrupt routine at the address given by the interrupt vector.

When entering into the Bus Fault interrupt routine, the internal CPU status is present in hidden CPU status registers. This status has to be saved to the memory using the SBFS instruction as the first instruction in the interrupt routine.

1.7.5 Format of the CPU status record

The format of the CPU status record is as follows:

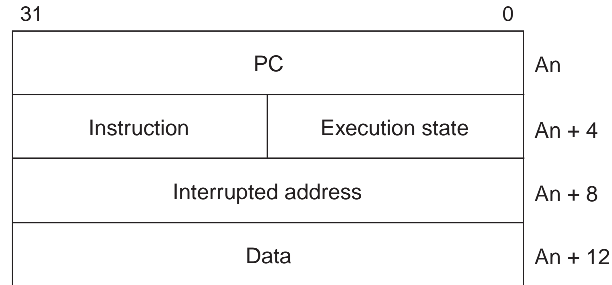

Figure 1-23

PC Field

First, the PC Field contains the value of PC immediately after the interrupted cycle. For example, if the bus fault occurs on an instruction fetch at address A in a linear instruction stream, the PC field will contain the value A + 2.

Execution State Field

The Execution State Field contains a number of flags that enables the CPU to restart in the correct execution state. The flags are:

Bit Number	Flag Name	Description
15 - 9	Reserved	These bits are written as 0's by SBFS. To ensure compatibility with future implementations, these bits should not be modified by the SW. If you generate the CPU status record by the SW (not using a status record saved with SBFS), these bits should be set to 0's. The bits are ignored by the current implementation of the RBF instruction.
8	Old F flag	This bit is set according to the status of the F flag immediately after the interrupted cycle (i.e. before it was set by the bus fault). This bit is ignored by the RBF instruction.
7	User mode flag	This bit is set according to the status of the U flag immediately after the interrupted cycle (i.e. before it was modified by the bus fault).
6	Arithmetic extend flag	This bit is set according to the status of the X flag immediately after the interrupted cycle.
5	Unaligned flag	Set if the interrupted cycle was the second cycle of an unaligned data read or write.
4	Data cycle flag	Set if the interrupted cycle was a data read or write (as opposed to an instruction fetch).
3	RETI/RETB delay slot flag	Set if the interrupted cycle was a delay slot of a RETI or RETB instruction that should take effect.
2	Delay slot flag	Set if the interrupted cycle was a delay slot of a taken branch, or a delay slot of a RET, RETI or RETB instruction that should take effect.
1	Address prefix flag	Set if the interrupted instruction was preceded by an address prefix.
0	Interrupt vector flag	Set if the interrupted cycle was an interrupt vector read cycle. This bit is ignored by the RBF instruction.

Table 1-9 Execution State Field Flags

Instruction field

If the interrupted cycle was a data read or write (i.e. not an instruction fetch), the Instruction Field contains the opcode of the interrupted instruction. In case the interrupted instruction was a MOVEM, the destination field (bit 15-12) of the instruction will hold the register number currently in transfer when the instruction was interrupted.

If the interrupted cycle was an instruction fetch, the instruction field will contain the invalid data that was fetched during the interrupted cycle. In this case, the field will be ignored by the RBF instruction.

Interrupted Address Field

The Interrupted Address Field contains the address of the data entity in transfer during the interrupted cycle. For instruction fetches and for aligned data read/write cycles, this is always the same as the address output from the CPU during the interrupted cycle. But for the second cycle of an unaligned data transfer, this field will contain the address that was output from the CPU during the cycle that came before the interrupted cycle.

Example:

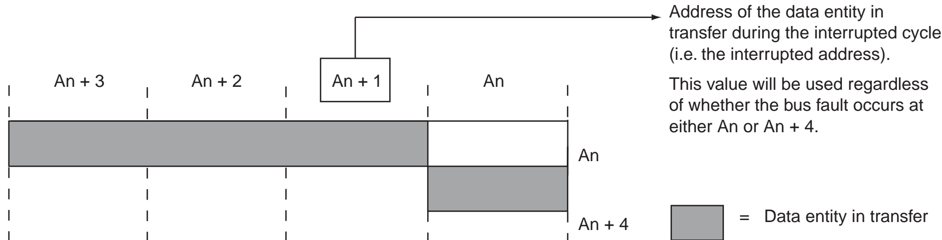

Figure 1-24

Data Field

Finally, the Data Field will have different meaning depending on the type of cycle that was interrupted:

Type of Interrupted Cycle	Definition of the Data Field
Instruction fetch cycle, not preceded by an address prefix	The data field contains the ALU result of the previous instruction. This data is ignored by the RBF instruction.
Instruction fetch cycle preceded by an address prefix	The data field contains the address that was calculated by the address prefix.
Aligned data read cycle, or first cycle of an unaligned data read	The data field contains the invalid data that was read in the interrupted cycle. This data is ignored by the RBF instruction.
Second cycle of an unaligned data read	The lower part of the data field contains the valid data that was read in the first cycle of the data read. The upper part of the data field will contain the invalid data read in the interrupted cycle. The RBF instruction will use the lower part and ignore the upper part of the data field.
Data write cycle	The data field will contain the data that was going to be written in the interrupted cycle.

Table 1-10 Data Field

1.7.6 Programming Examples

Go to user mode for the first time:

```
MOVE    CCR, Rn
OR.W    0x100, Rn
MOVE    Rn, CCR                         ; Set U flag
MOVE    user_stack_pointer, USP
JMPU    user_mode_program_entry
```

Bus fault routine:

```
SBFS [SP=SP-16]
PUSH DCCR
PUSH registers
:
:
POP registers
POP DCCR
RBF [SP+]
```

Disabling interrupt from user mode programs:

In user mode, the I flag is prevented from being changed. This is in general desired to avoid that user mode programs lock out interrupts. If a user mode program needs to disable interrupts, this can be achieved by using the BREAK instruction. You can for example reserve BREAK 0 for this purpose. (The same mechanism can also be used for other more complicated system calls.)

User mode program:

```
    :
    BREAK 0                   ; Jump to breakpoint0_entry
    :                         ; and save return address in BRP.
```

Breakpoint code:

```
breakpoint0_entry:
    RETB                      ; Return immediately
    DI                        ; Disable interrupts in the delay slot.
```

1.8. Interrupts

The CRIS CPU uses vectorized interrupts that are generated either externally to, or internally by, the ETRAX 100LX. The interrupt acknowledge sequence consists of the following steps:

1 Perform an INTA cycle, where the 8-bit vector number is read from the bus.

2 Store the contents of PC to the Interrupt Return Pointer (IRP). Note that the return address is not automatically pushed on the stack.

3 Read the interrupt vector from the address [IBR + <vector number> * 4].

4 Start the execution at the address pointed to by the interrupt vector.

The Interrupt Base Register (IBR) has bits 31-16 implemented. The remaining bits are always zero.

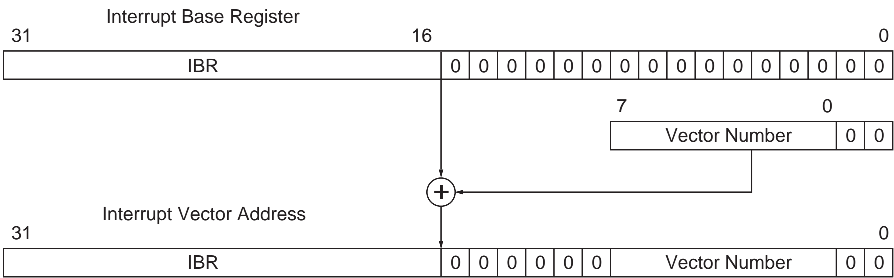

Figure 1-25 Interrupt Vector Address Calculation

The interrupt acknowledge sequence of the CRIS CPU does not automatically push the condition codes and the interrupt return address on the stack. The interrupt return address is stored in the Interrupt Return Pointer (IRP). If nested interrupts are used, the IRP must be pushed on the stack as the first instruction of the interrupt routine. The Condition Code Register (CCR) must always be pushed at the start of an interrupt routine, and restored at the end.

The Interrupt enable flag is unaffected by the interrupt sequence. However a new interrupt will not be enabled until after the first instruction of the interrupt routine.

Also, all transfers to and from Special Registers will disable interrupts until the next instruction is executed. In this way, the IRP and CCR or DCCR can always be pushed on the stack before a new interrupt is allowed, see examples on the next page.

Note that the RETI instruction is a delayed jump with one delay slot, but the JMPU instruction is performed directly:

Single Level Interrupts

```
INT_ENTRY:
    PUSH      DCCR               ; Push condition codes onto the stack.
    DI                           ; Disable interrupts.
    SUBQ      stack_offset,SP    ; Reserve stack for used registers.
    MOVEM     Rn,[SP]            ; Save registers.
    :
    (Perform desired function)
    :
    MOVEM     [SP+],Rn           ; Restore registers.
    RETI                         ; Return: Take address from IRP.
    POP       DCCR               ; Restore condition codes (this is
                                 ; placed in the delay slot of the
                                 ; RETI instruction).
```

Nested Interrupts

```
INT_ENTRY:
    PUSH      IRP                ; Push return address onto the stack.
    PUSH      DCCR               ; Push condition codes onto the stack.
    SUBQ      stack_offset,SP    ; Reserve stack for used registers.
                                 ; ← Interrupts are enabled here.
    MOVEM     Rn,[SP]            ; Save registers.
    :
    (Perform desired function)
    :
    MOVEM     [SP+],Rn           ; Restore registers.
    POP       DCCR               ; Restore condition codes.
                                 ; ← Interrupts are disabled here
                                 ; until after the return from
                                 ; interrupt.
    JMPU      [SP+]              ; Return from interrupt.
```

Interrupts (including NMI and HW break) update the U flag according to the current operating mode, and perform a transition to supervisor mode. The transition will take place in the INTA cycle so that the interrupt vector is read in supervisor mode. An interrupt will also set the F flag.

A special case occurs if there is a bus fault in the interrupt vector read cycle. The CPU can handle the bus fault, and a separate bit is set in the CPU status record. The interrupt sequence can, however, not be automatically restarted by the RBF instruction. This case does not have to be considered for MMU functionality because a bus fault on the interrupt vector table would make it impossible to reach the bus fault interrupt routine anyway. For single step, this case has to be checked for and taken care of by the single step SW.

1.8.1 NMI

The Non Maskable Interrupt (NMI) is handled in the same way as the normal interrupt except for the following three differences:

1. The return address is stored in the Breakpoint Return Pointer (BRP) instead of the IRP.
2. The NMI is enabled/disabled by the M flag instead of the I flag. The M flag can be set with the SETF M instruction. Move to CCR/DCCR has no effect. Once set, the M flag can only be cleared by an NMI acknowledge cycle or system reset.
3. The INTA cycle will be an idle bus cycle, and the vector number 0x21 is generated internally in the CPU.

1.9. Software Breakpoints

The CRIS CPU has a breakpoint instruction (BREAK n). This instruction saves the current value of PC in the Breakpoint Return Pointer (BRP) register, and performs a jump to address (IBR + 8*n).

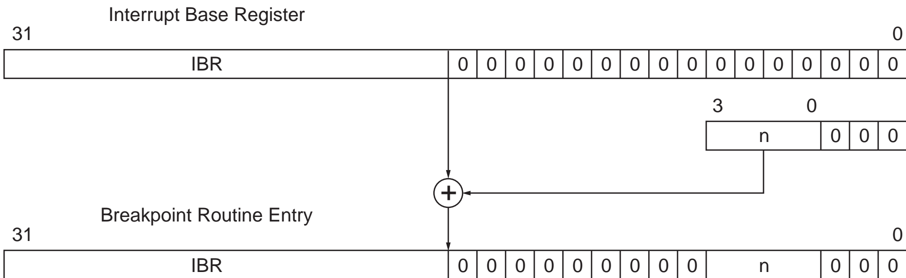

Figure 1-26 Software Breakpoint Address Calculation

1.10. Hardware Breakpoint Mechanism

The CPU contains a hardware breakpoint mechanism. The hardware breakpoint address is loaded in the Breakpoint Address Register (BAR), and the hardware breakpoint mechanism is enabled by setting the Breakpoint enable flag B (see 1-1 and Figure 1-3).

For each CPU read or write cycle, the address is compared with the contents of the BAR register. In order to detect a read or write in the dword (and not just a single

byte) of the address location, bit 1 and 0 are ignored in the comparison. Bit 31 is also ignored in the comparison since that bit handles the cache in the ETRAX 100LX(address bit 31 set will bypass the cache and directly access the main memory).

An address hit is handled in the same way as an NMI with interrupt vector number 0x20, except that a breakpoint hit is not affected by the M flag.

The hardware breakpoint mechanism is disabled after reset.

1.11. Multiply and Divide

1.11.1 General

The ETRAX 100LX implementation of the CRIS CPU has two multiply instructions: Signed Multiply (MULS) and Unsigned Multiply (MULU). For compatibility with CRIS implementations not supporting multiply instructions, multiply operations can also be performed using a sequence of Multiply Step (MSTEP) instructions.

There are no divide instructions, so divide operations are performed by a sequence of Divide Step (DSTEP) instructions.

1.11.2 Multiply using MULS and MULU

The MULS and MULU instructions are fast (2 cycle) multiply operations. The multiply is performed on 32 by 32 bits, giving a 64-bit result. The lower 32 bits are stored to the destination register specified with the instruction, while the upper 32 bits are stored in the Multiply Overflow (MOF) register.

For multiply with byte or word sized operands, the operands are extended to 32 bits before the multiply. Sign extend is used with Signed Multiply (MULS), while zero extend is used with Unsigned Multiply (MULU).

1.11.3 Multiply Using MSTEP

In addition to the MULS and MULU instructions, multiply operations can be performed using the MSTEP instruction. The MSTEP instruction does the following:

1. Shifts the destination register one step to the left.
2. If the N flag is set, adds the source operand.
3. Updates the flags.

The example below shows a 16-bit by 16-bit unsigned multiply with 32-bit result:

```
MUL_BEGIN:
    MOVU.W     [value1],R0       ; Move first operand to a register,
                                 ; and clear the upper 16 bits.
    MOVU.W     [value2],R1       ; Move second operand to a register.
    LSLQ       16,R1             ; Shift left, clear the lower 16 bits
                                 ; of the result register, and set the
                                 ; N flag according to msb of value2.
    MSTEP      R0,R1             ; Perform 16 iterations of the MSTEP
    MSTEP      R0,R1             ; instruction. Each iteration sets
    MSTEP      R0,R1             ; the N flag for next step.
    MSTEP      R0,R1
    MSTEP      R0,R1
    MSTEP      R0,R1
    MSTEP      R0,R1
    MSTEP      R0,R1
    MSTEP      R0,R1
    MSTEP      R0,R1
    MSTEP      R0,R1
    MSTEP      R0,R1
    MSTEP      R0,R1
    MSTEP      R0,R1
    MSTEP      R0,R1
    MSTEP      R0,R1             ; The last iteration. The result is in R1.
```

1.11.4 Divide

Divide operations can be performed using the DSTEP instruction. The DSTEP instruction does the following:

1 Shifts the destination register one step to the left.

2 If the destination register is unsigned-greater-than or equal to the source operand, the source operand is subtracted from the destination register.

16-bit by 16-bit unsigned divide example:

```
DIV_BEGIN:
    MOVU.W    [num],R1          ; Move numerator to a register,
                                ; and clear the upper 16 bits.
    MOVU.W    [denom],R0        ; Move denominator to a register.
    LSLQ      16,R0             ; Shift left, clear the lower 16 bits
                                ; of the denominator register.
    SUBQ      1,R0              ; Subtract one from the denominator.
    DSTEP     R0,R1             ; Perform 16 iterations of the DSTEP
    DSTEP     R0,R1             ; instruction.
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1
    DSTEP     R0,R1             ; The last iteration. The quotient is
                                ; in the lower half of R1, and the
                                ; remainder is in the upper half of of R1.
```

1.12. Extended arithmetic

Extended arithmetic (arithmetic with more than 32 bits) is supported by using the X flag. The X flag is set by the AX (SETF X) instruction, and is cleared by all other instructions.

When the X flag is set, instructions involving an addition or subtraction are modified in the following ways:

1 The C flag is added to the result of an addition, and subtracted from the result of a subtraction. This is valid even if the addition/subtraction result is not the result operand of the instruction.

2 If the result operand is zero, the Z flag will maintain its old value instead of being set.

3 The change of the Z flag behaviour is valid for all instructions that affect the Z flag except:

CLEARF,
MOVE to CCR/DCCR,
POP CCR/DCCR,
SETF

The addition/subtraction of the C flag affects the following instructions:

ABS,	ADD,	ADDI,	ADDQ,	ADDS,
ADDU,	BOUND,	CMP,	CMPQ,	CMPS,
CMPU,	DSTEP,	MSTEP,	NEG,	SUB,
SUBQ,	SUBS,	SUBU		

The address calculation in addressing mode prefixes is not affected.The AX instruction disables the interrupts until the next instruction to ensure that the X flag is not cleared by an interrupt routine before it is used. Below are two examples of extended arithmetic.

Add a 48-bit signed value contained in R3:R2 to a 64 bit value stored in R1:R0:

```
EXT_ADD:
    ADD.D     R2,R0             ; Add the low dwords.
    AX                          ; Set the X flag.
    ADDS.W    R3,R1             ; Add the upper 16 source bits.
```

Test if a 40-bit value contained in R1:R0 is zero:

```
EXT_TEST:
    TEST.D    R0                 ; Test the lower 32 bits.
    AX
    TEST.B    R1                 ; Test upper 8 bits.
```

1.13. Integral Read-Write Operations

Since a bus fault can interrupt the CPU in any bus cycle (except INTA), it is not possible to ensure the integrity of a piece of code just by disabling the interrupts or by only using instructions that lock out interrupts between them. Instead, integral read-write operations can be implemented by using the Load-locked, Store-conditional principle:

```
Start:

Initialize lock;

Read variable;

Modify variable;

Write back variable if and only if the sequence hasn't been interrupted;

Go to Start if write failed;
```

The F and P flags, and the branch instruction Branch on Write Failed (BWF), are used to test whether the write succeeded or failed. See section 1.2. *Flags and Condition Codes*.

The F flag is set by the BREAK instruction, when the CPU performs an interrupt acknowledge, or when a bus fault sequence occurs. The P flag is set when a write to memory fails because of broken integrity.

The F and P flags are cleared by the CLEARF instruction regardless of the list of flags. F and P are not affected by the SETF instruction.

A write to memory can be made conditional by setting the X flag in the instruction before the write. This will affect all instructions that write to memory, except SBFS.

Pseudo code for instructions that write to memory will be:

```
if (F & X)

{

  P = 1;

}

else

{

  write to memory;

}
```

The BWF instruction has the action: Branch if P is set. It has the same opcodes as the normal branch instruction, and the condition field of the instruction (bits 15 - 12) is 1111 (binary).

A code example of how the features can be used to implement a test-and-clear function is shown below:

```
START_LOCK:CLEARF                         ; CLEARF with an empty list will
LOCK_LOOP: MOVE.b [memory_location], R0 ; clear F, P and X flags.
           AX                            ; Save data in R0 for future analysis.
           CLEAR.b [memory_location]     ; Make the clear conditional.
           BWF LOCK_LOOP                 ; Loop back if clear failed.
           CLEARF                        ; Use delay slot to
                                         ; reinitialize F and P flags.
```

Still, more complicated things can be done in the loop, as long as the data can be written in one single CPU cycle. With some extra care about where the MMU page boundaries are placed, it is also possible to use write instructions that need several CPU cycles (e.g. unaligned dword writes, or MOVEM instructions).

1.14. Reset

The following registers are initialized after reset:

Register	Value (hex)
VR	<version number>
CCR	0000
DCCR	00000000
IBR	00000000

Table 1-11 Registers Initialized After Reset

All other registers have unknown values after reset.

After reset, the ETRAX 100LX CPU starts execution at a particular address depending on the boot method:

Register	Value (hex)
PROM	80000002
Net	380000f0
Parallel port	380000f0
Serial port	380000f4

Table 1-12 Boot Methods

1.15. Version Identification

Different versions of the CRIS CPU can be identified by reading the Version Register (VR). The version register is an 8-bit read-only register that contains the CPU version number. The contents of the CRIS VR Register are:

Value	Chip Name	Part No	Note
0	ETRAX-1	13425	
1	ETRAX-2	13576	
2	ETRAX-3	13873	
3	ETRAX-4	14517	
4, 5, 6, 7			Reserved for future chips in the ETRAX-1 family.
8	ETRAX 100 version 1	15822	
9	ETRAX 100 version 2	16284	
10	ETRAX 100LX E1	17511	
11	ETRAX 100LX E2	17854	
11	ETRAX 100LX E3	18816	
11	ETRAX 100LX E3	19322	Lead (Pb) free.
12, 13, 14, 15			Reserved for future chips in the ETRAX 100LX family.
16 - 255			Not assigned.

Table 1-13 CRIS VR Register