

1 ARCHITECTURAL DESCRIPTION

1.1 Registers

The processor contains 15 32-bit *General Registers* (R0 - R14) and one 32-bit *Program Counter* (R15 or PC). Register R14 is also called the *Stack Pointer* (SP). However, this is only an assembly language convention since any of the general registers may be used as a stack pointer.

The processor architecture defines 16 *Special Registers* (P0 - P15), of which eight 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 The low byte of this register contains the flags. The high byte is always zero.	16 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 instruction or a hardware breakpoint.	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:

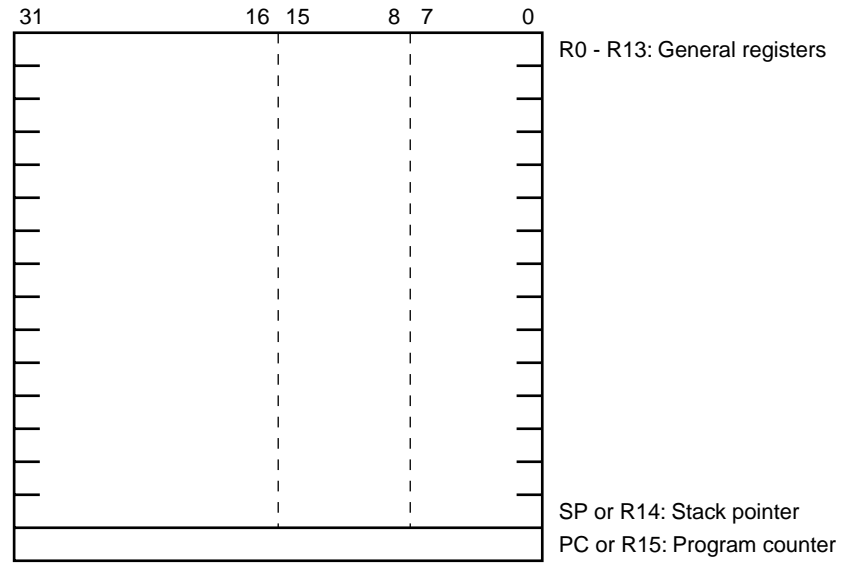


Figure 1-1 General registers

Special registers:

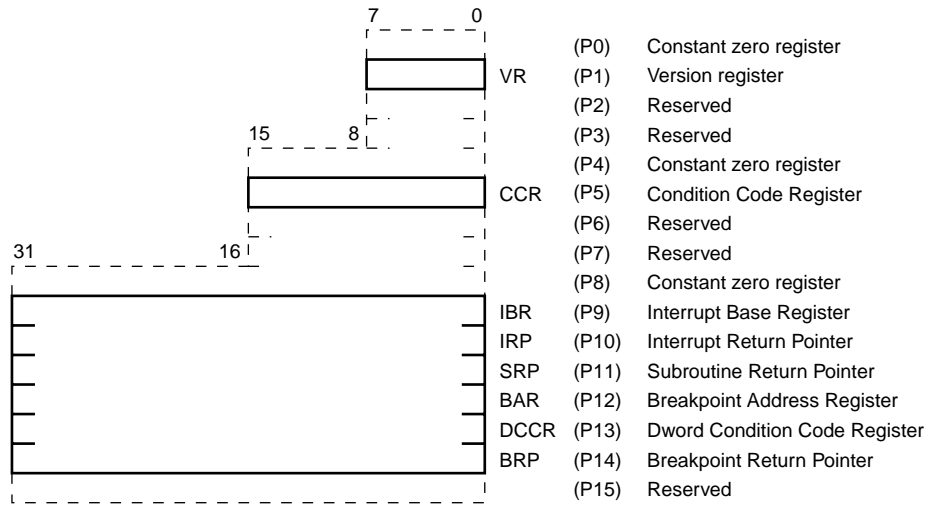


Figure 1-2 Special registers

1.2 Flags and Condition Codes

The Condition Code Register (CCR) and Dword Condition Code Register (DCCR) contain eight different flags. The eight remaining bits are always zero:

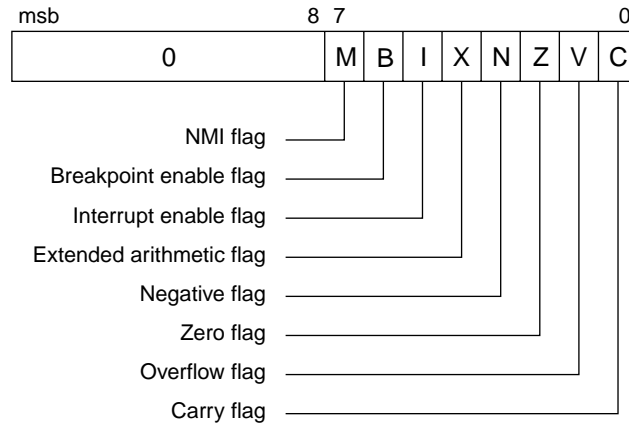


Figure 1-3 The Condition Code Register (CCR)/ Dword Condition Code Register (DCCR)

The X flag which is used for extended arithmetic operations, is described in more detail in section 1.11 *Extended Arithmetic*. 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	\bar{C}
CS	LO	Carry Set	0001	C
NE		Not Equal	0010	\bar{Z}
EQ		Equal	0011	Z
VC		Overflow Clear	0100	\bar{V}
VS		Overflow Set	0101	V
PL		Plus	0110	\bar{N}
MI		Minus	0111	N
LS		Low or Same	1000	$C + Z$
HI		High	1001	$\bar{C} * \bar{Z}$
GE		Greater or Equal	1010	$N * V + \bar{N} * \bar{V}$
LT		Less Than	1011	$N * \bar{V} + \bar{N} * V$
GT		Greater Than	1100	$N * V * \bar{Z} + \bar{N} * \bar{V} * \bar{Z}$
LE		Less or Equal	1101	$Z + N * \bar{V} + \bar{N} * V$
A		Always True	1110	1
		(Reserved)	1111	

Table 1-2 Condition codes

1 Architectural Description

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

General case:
$N = R_{msb}$
$Z = \bar{R}_{msb} * \dots * \bar{R}_{lsb} * (Z + \bar{X})$
Addition: (ADD, ADDQ, ADDS and ADDU)
$N = R_{msb}$
$Z = \bar{R}_{msb} * \dots * \bar{R}_{lsb} * (Z + \bar{X})$
$V = S_{msb} * D_{msb} * \bar{R}_{msb} + \bar{S}_{msb} * \bar{D}_{msb} * R_{msb}$
$C = S_{msb} * D_{msb} + D_{msb} * \bar{R}_{msb} + S_{msb} * \bar{R}_{msb}$
Subtraction: (CMP, CMPQ, CMPS, CMPU, NEG, SUB, SUBQ, SUBS and SUBU)
$N = R_{msb}$
$Z = \bar{R}_{msb} * \dots * \bar{R}_{lsb} * (Z + \bar{X})$
$V = \bar{S}_{msb} * D_{msb} * \bar{R}_{msb} + S_{msb} * \bar{D}_{msb} * R_{msb}$
$C = S_{msb} * \bar{D}_{msb} + \bar{D}_{msb} * R_{msb} + S_{msb} * R_{msb}$
Bit test: (BTST and BTSTQ)
$N = D_n$
$Z = \bar{D}_n * \dots * \bar{D}_{lsb} * (Z + \bar{X})$
Move to CCR: (MOVE s, CCR and POP CCR)
B, I, N, Z, V, C set according to source data.
X always cleared.
M not affected.
Condition Code Manipulation: (SETF and CLEARF)
CCR set or cleared according to mask bits in the instruction.
M can not be cleared.
If X is not on the list, it is cleared.

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

The data types supported by 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 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 100 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;
};
```

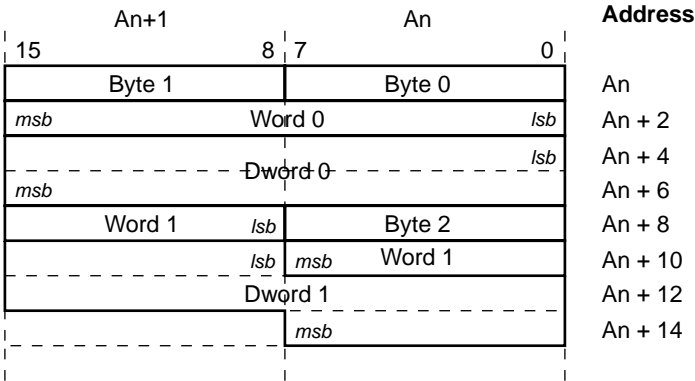


Figure 1-4 Data organization with a 16-bit bus

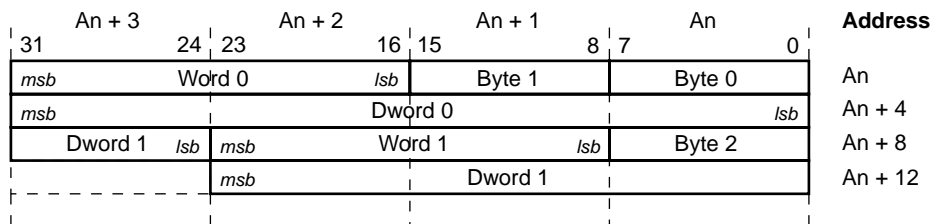


Figure 1-5 Data organization with a 32-bit bus

1.4 Instruction Format

The basic instruction word is 16 bits long. 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:

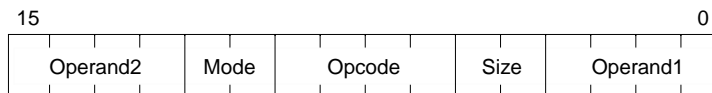


Figure 1-6 General instruction format

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 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 indirect
- 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.

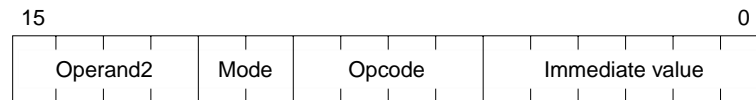


Figure 1-7 Quick immediate addressing mode instruction format

Assembler Syntax: <expression>

Example: 12

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 mode, the operand is contained in the memory location pointed to by the register specified by the Operand1 field.

Assembler Syntax: [Rn]

Example: [R6]

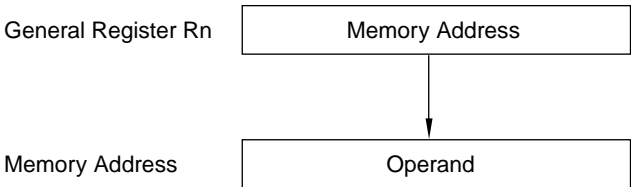


Figure 1-8 Indirect addressing mode

1.5.5 Autoincrement addressing mode

In the autoincrement 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+]

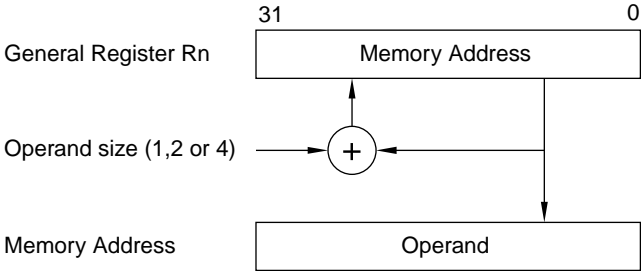


Figure 1-9 Autoincrement addressing mode

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

This addressing mode requires the basic instruction word to be preceded by one *addressing mode prefix* word, formatted as shown below:

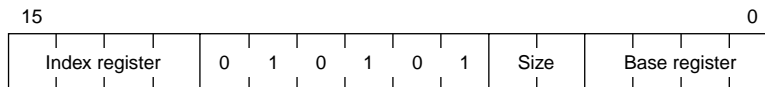


Figure 1-10 Indexed addressing mode prefix format

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: $[R_n + R_m.s]$

Example: $[R6 + R7.B]$

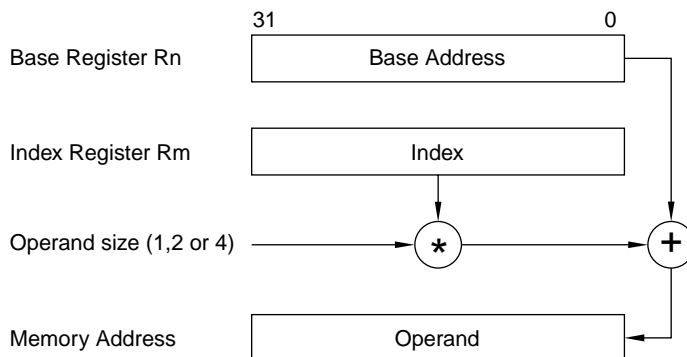


Figure 1-11 Indexed addressing mode

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 for the indexed mode. The selection between indexed 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: $[R_p = R_n + R_m.s]$
 Example: $[R_8 = R_6 + R_7.B]$

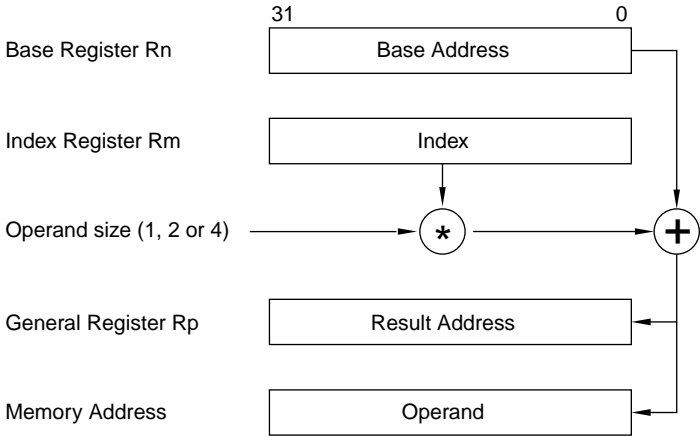


Figure 1-12 Indexed with assign addressing mode

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:

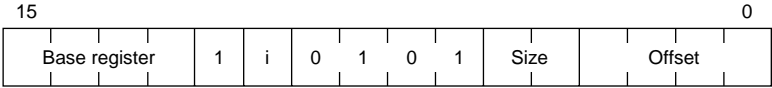


Figure 1-13 Offset addressing mode prefix format

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 ($i = 0$) or autoincrement ($i = 1$) 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:

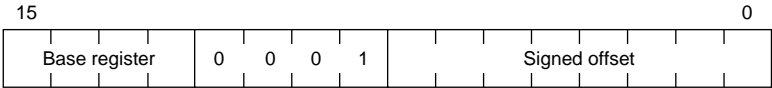


Figure 1-14 Immediate byte offset addressing mode prefix format

Word or dword sized immediate offsets use the general prefix format, with $i = 1$ and $\text{offset} = \text{PC}$. In this case, the immediate offset word(s) will be placed between the prefix word and the basic instruction word, see example below:

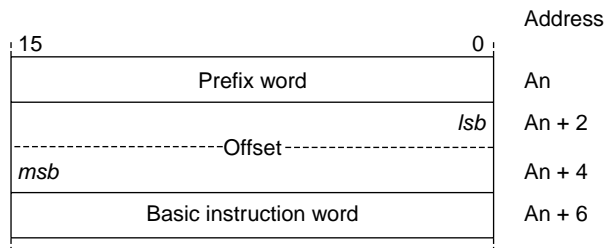


Figure 1-15 Instruction with dword-sized immediate offset

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: $[Rn + \langle \text{expression} \rangle]$

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: $[R6 + [R7+].B]$

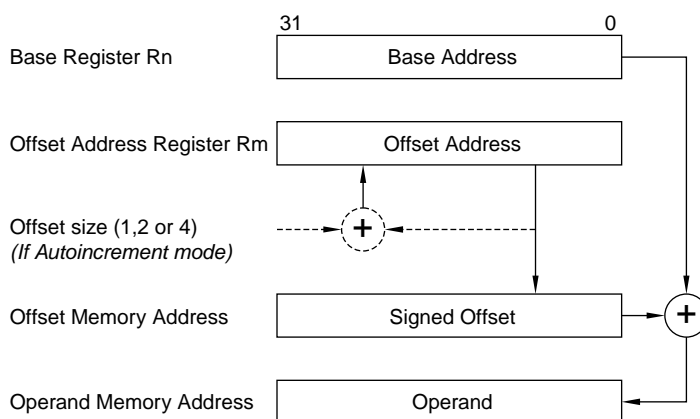


Figure 1-16 Offset addressing mode (general case)

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]

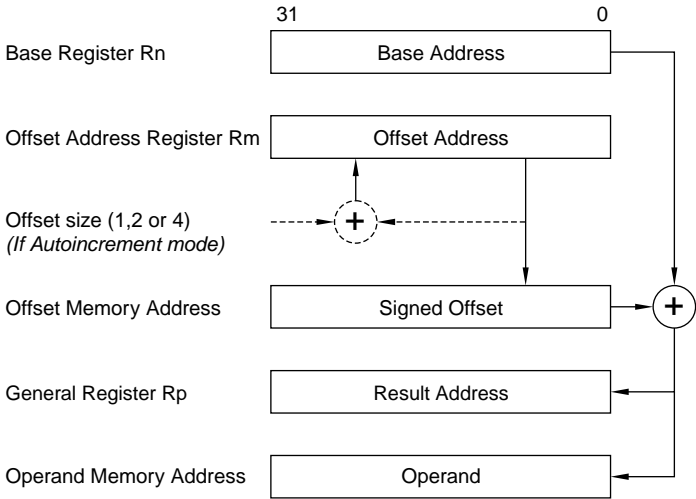


Figure 1-17 Offset with assigned addressing mode (general case)

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:

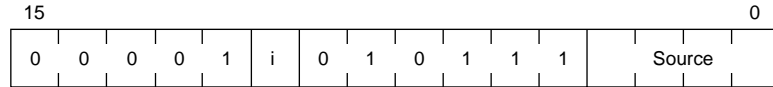


Figure 1-18 Double indirect addressing mode prefix format

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 ($i = 0$) or incremented by 4 after it is used ($i = 1$).

Double indirect addressing mode

Assembler Syntax: $[[Rn]]$

Example: $[[R6]]$

Double indirect with autoincrement addressing mode

Assembler Syntax: $[[Rn+]]$

Example: $[[R6+]]$

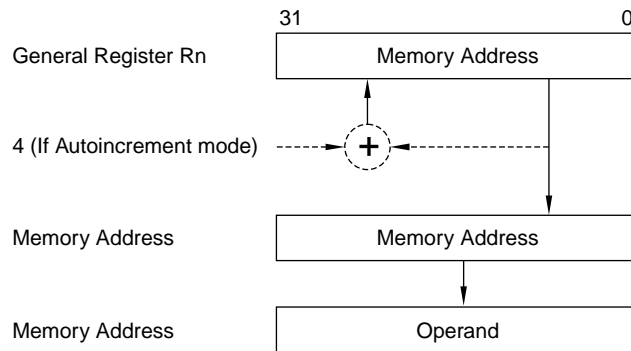


Figure 1-19 Double indirect addressing mode

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:

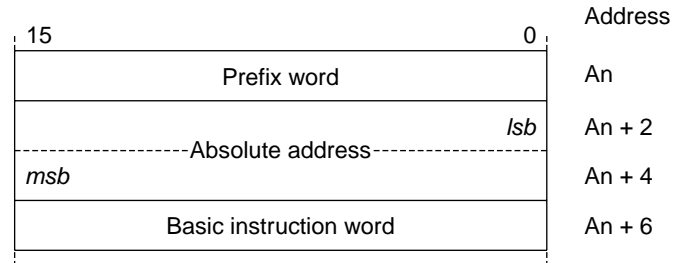


Figure 1-20 Instruction with absolute address

Assembler Syntax: [<expression>]

Example: [3245]

1.5.13 Multiple addressing 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/JIR/JSR/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 instruction

The *JUMP* instruction is an unconditional absolute jump instruction. The jump 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
```

In contrast to the Bcc instruction, the JUMP instruction takes 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,  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:
```

1 Architectural Description

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

```
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.
ADDS.W    [PC+R0.W],PC     ; 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 New Subroutine instructions in the ETRAX 100

Two new instructions, *Jump to Subroutine* (JSRC) and *Jump to Interrupt Routine* (JIRC) have been added to the ETRAX 100. The new JSRC and JIRC instructions will add 4 to the return address stored to SRP or IRP. This leaves four Bytes unused between the JSRC/JIRC instruction and the return point. These four Bytes can, for example, be used for C++ exception handling information.



Figure 1-21 The new JSRC and JIRC instructions in the ETRAX 100

1 Architectural Description

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

JSR/JIR instructions	A	JSRC/JIRC instructions	A
Immediate jump target address	A + 2	Immediate jump target address	A + 2
Return to here	A + 6	Unused	A + 6
		Return to here	A + 10

Figure 1-22 Immediate addressing of the new jump instructions

A 'jump to breakpoint routine' instruction, JBRC, has also been added. JBRC acts just like the JSRC and JIRC instructions, except that it stores the return address in the BRP register instead of the SRP or IRP register.

1.7 Interrupts

The CRIS CPU uses vectorized interrupts that are generated either externally to, or internally by, the ETRAX 100. 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 bit 31-16 implemented. The remaining bits are always zero.

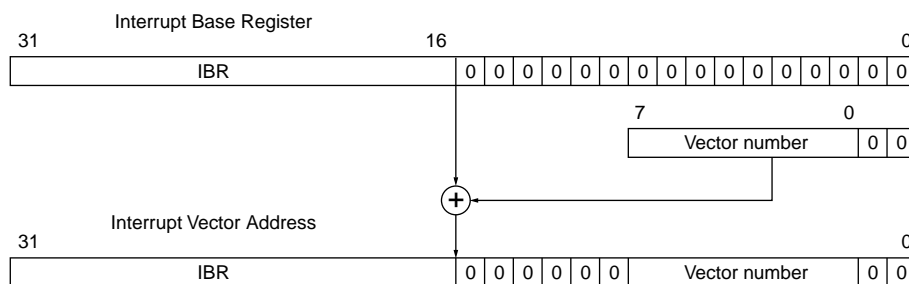


Figure 1-23 Interrupt vector address calculation

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 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 JUMP instruction is performed directly. See examples below:

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.
    JUMP      [SP+]                ; Return from interrupt.
```

Note that the sequences described above will also restore the correct state if the interrupt routine is called by software (using the JIR or JIRC instruction). If the interrupts were disabled at the time of the JIR or JIRC instruction, the interrupts will be disabled throughout the execution of the called interrupt routine, and remain disabled after return.

1.7.1 NMI

The Non Maskable Interrupt (NMI) is handled in the same way as the normal interrupt except for the following two 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 has no effect. Once set, the M flag can only be cleared by an NMI acknowledge cycle or system reset.

1.8 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).

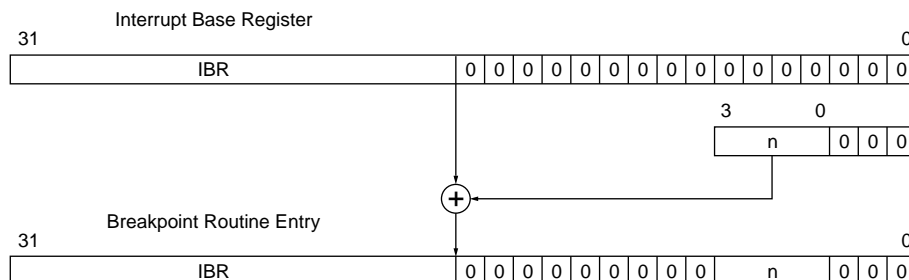


Figure 1-24 Software breakpoint address calculation

1.9 Hardware Breakpoint Mechanism

The CPU contains a hardware breakpoint mechanism. The hardware breakpoint address is loaded in the BAR register (Breakpoint Address Register), and the hardware breakpoint mechanism is enabled by setting the Breakpoint enable flag B (see table 1-1 *Special registers*, on page 13).

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 (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.

1.10 Multiply and Divide

1.10.1 General

The CRIS CPU has no complete multiply and divide instructions. Multiply and divide operations are performed by a sequence of multiply-step (MSTEP) or divide step (DSTEP) instructions.

1.10.2 Multiply

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.

16-bit by 16-bit unsigned multiply example:

```
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
    :                                                ; instruction. Each iteration sets
    (16 MSTEP instructions) ; the N flag for next step.
    :
    MSTEP     R0,R1              ; The last iteration. The result is
                                ; in R1.
MUL_END:
```

1.10.3 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
    :                               ; instruction.
    (16 DSTEP instructions)
    :
    DSTEP    R0,R1          ; The last iteration. The quotient is
                               ; in the lower half of R1, and the
                               ; remainder is in the upper half of
                               ; R1.

DIV_END:
```

1.11 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.

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

CCR, CLEARF, MOVE, POP CCR, 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. Example 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.
```

1.12 Reset

1.12.1 Normal case

After reset, the ETRAX 100 CPU starts the execution at address 80000002. The following registers are initialized after reset:

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

Table 1-9

All other registers have unknown values after reset.

1.13 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	Token ring.
3	ETRAX-4	14517	SCSI interface.
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, 11, 12, 13, 14, 15			Reserved for future chips in the ETRAX 100 family.
16			Not assigned.

Table 1-10