CoolRISC®816
8-bit Microprocessor Core

Hardware and Software Reference Manual

Version 4.5
April 2001

For further information, please contact

XEMICS SA
E mail: info@xemics.com
Web: www.xemics.com

Document History

Date	Version	Who	Changes
4 jul 2000	4.1	AVx	Completely new version.
20 sep 2000	4.2	AVx	pp. 2-8 & 2.9: Modification of Z flag corrected. Section 2.4: Jcc examples corrected.
20 nov 2000	4.3	AVx	p. 2-13: CPL2 operation corrected.
23 mar 2001	4.4	AVx	Many small corrections and clarifications.
05 apr 2001	4.5	CEM	Subtraction operations corrected. SFLAG instruction modified.

Document Nr: R0105-078

© 2001 Xemics, v4.5

Table of Contents

Notations

Chapter 1
Architectural and Functional Overview

Chapter 2
Instruction Set

Chapter 5
Test Capabilities

Chapter 6
Advanced Features

List of Tables

© 2001 Xemics, v4.5

List of Figures

List of Codes

© 2001 Xemics, v4.5

Notations

General Instruction Format

The general instruction format is: `INSTR res, op1 [ , op2 ]`
where `INSTR` is the assembler mnemonic for the instruction and the operands are described below.

Operands

a	Accumulator register
C	Carry flag.
DM[<eaddr>]	Data at effective address <eaddr> in data memory.
GIE	General Interrupt Enable (bit 5 of status register).
PC	Program counter.
STn	(n > 0) n'th level in the hardware stack.
REG REGi REGj REGk	Any 8-bit register (**a**, **r0**, **r1**, **r2**, **r3**, **iph**, **ipl**, **i0h**, **iol**, **i1h**, **i1l**, **i2h**, **i2l**, **i3h**, **i3l**, **stat**). See Figure 1.6 on page 12.
REG[i]	Bit i of register.
REG[m:n]	(m > n) Bit slice of register.
V	Overflow flag.
Z	Zero flag.
ip	16-bit program memory index register, concatenation of **iph** and **ipl** registers.
i0	16-bit data memory index register, concatenation of **i0h** and **i0l** registers.
i1	16-bit data memory index register, concatenation of **i1h** and **i1l** registers.
i2	16-bit data memory index register, concatenation of **i2h** and **i2l** registers.
i3	16-bit data memory index register, concatenation of **i3h** and **i3l** registers.
ix	Any 16-bit data memory index register **i0**, **i1**, **i2**, **i3**.
op1	First operand.
op2	Second operand (optional).
res	Result of an operation.

Operators

:=	Value assignment
<< n	Shift left n bits, vacated bits are filled with zeroes
>> n	Shift right n bits, vacated bits are filled with zeroes

Sub fields and Qualifiers

`MSB, LSB`	8 bits	Most significant byte [15:8], least significant byte [7:0].
`cc:3`	3 bits	Branch condition code.
`jaddr:16`	16 bits	Jump address.
`#data:8`	8 bits	Immediate data.
`#shift:3`	3 bits	Immediate shift value.
`addr:8`	8 bits	Data memory address.
`offset:8`	8 bits	Unsigned offset value.
`cpl2_offset:7`	7 bits	Signed offset value without the sign bit. An offset value of -1 in the instruction is represented as the value `0x7F` in the opcode.
`divn:4`	4 bits	Frequency division ratio.
`#s`	1 bit	Boolean switch.
`#bit:3`		Bit index value (0..7).
`n_data:8`	8 bits	Immediate data, *inverted*.
`n_addr:8`	8 bits	Data memory address, *inverted*.
`n_jaddr:16`	16 bits	Program memory address, *inverted*.

`<regi>`
`<regj>`
`<regk>`
	4 bits	Register selection for `REGi`, `REGj`, and `REGk`, respectively.
		
		Use value: to select: Use value: to select:
		`0000` **`i0l`** `1000` **`ipl`**
		`0001` **`i0h`** `1001` **`iph`**
		`0010` **`i1l`** `1010` **`stat`**
		`0011` **`i1h`** `1011` **`r3`**
		`0100` **`i2l`** `1100` **`r2`**
		`0101` **`i2h`** `1101` **`r1`**
		`0110` **`i3l`** `1110` **`r0`**
		`0111` **`i3h`** `1111` **`a`**

`<ixs>`	2 bits	Data memory index selection. **Use value: to select:** `00` **`i0`** `01` **`i1`** `10` **`i2`** `11` **`13`**
`<eaddr>`		Effective address. Can be one of the following values: `addr:8` immediate address, limited to page 0. `(ix)` indexed address. `(ix, offset:8)` indexed address with offset. `(ix, r3)` indexed address with offset in register **`r3`**. `(ix)+` indexed address with post-incrementation of **`ix`**. `(ix, offset:7)+` indexed address with post-incrementation of **`ix`**. `-(ix)` indexed address with pre-decrementation of the address *and* of **`ix`**. `-(ix, offset:7)` indexed address with pre-decrementation of the address *and* of **`ix`**.

Chapter 1
Architectural and Functional Overview

This chapter introduces the CoolRISC 816 micro-controller core, referred to as CR816 in this document, presents its key features, its architecture and provides a functional overview.

1.1. Introduction

The CR816 is a member of the CoolRISC family of 8-bit micro-controller cores which are designed to be embedded in ASICs, ASSPs and standard products. The CoolRISC cores offer high performance computing for very low voltage, low power consumption and small footprints.

The CR816 has the following general characteristics:

- *Harvard RISC-like architecture*. The instructions to be executed are stored in the program memory and general purpose data as well as the peripherals, are stored in a separate data memory. This structure gives the core the capability to read operands in the data memory simultaneously with one instruction fetch.
- *Register-memory architecture*. Instructions can operate with operands stored either in registers or in the data memory. All arithmetic and logic instructions can be executed with a first operand in a register and a second operand either in data memory or in a second register. The result can be stored either in a third register or in the first one.
- *Memory sizes*. Maximum data memory size is 64 Kbytes. Maximum program memory size is 64 Kinstructions, where one instruction is 22-bit wide.
- *Three-stage pipeline*. One instruction enters the pipeline at each clock cycle and is executed in a maximum of three clock cycles. The pipeline suffers no penalty such as delay slots or branch delays. Thus the clock count per instruction (CPI) is exactly one for any instruction. Also, as the pipeline hardware remains simple (no need to perform branch prediction), the power consumption is minimized.
- *8bx8b multiplier*. The CR816 includes a 8-bit multiplier unit which executes one 8-bit multiplication in one clock cycle.
- *Gated clock design support*. The CR816DL is ready for automatic insertion of gated clocks, hence minimizing power consumption.
- *Low frequency modes*. In order to reduce power consumption, a programmable internal frequency divider is implemented. Division factors of 2, 4, 8 or 16 can be selected. The frequency can be selected by software.
- *Stand-by mode*. The HALT instruction can switch the core in halt mode in which the power consumption is minimal. The internal clock is stopped and almost nothing toggles. The processor can be waken up using either events, interrupts or a reset.
- *Wait mode*. An handshake mechanism is implemented in the CR816 to provide memory sharing between several processors (or DMA controllers). During cycles in which a processor does not access its respective data memory, other processors are free to read or write in it. The management of shared memories is implemented through a request and acknowledge scheme. With this mechanism, the processor is considered as a slave rather than a master. This feature can also be used to access low speed peripherals.

© 2001 Xemics, v4.5

- *On-chip and full scan testing*. A flexible serial test interface for industrial production test of integrated cores is provided. This test interface also provides capabilities to access the peripherals in the data memory space, as well as the program memory. The core also supports full scan testing.
- *Hardware stack*. To optimize performances, a hardware stack of variable depth can be configured prior synthesis or integration.
- *Low power memories*. The CR816 has been designed to use XEMICS' low power memories, e.g. with precharge cycles.
- *Soft and hard core*. The CR816 is available either as a Verilog RTL synthesizable model or as a gate-level/layout hard core.
- *Development tools*. A rich set of development tools including a macro assembler, a source level debugger, a profiler, binary utilities and an ANSI C compiler based on GNU tools is provided. A graphical integrated development environment for Windows is available to run the tools. It also includes a project manager and browser. Finally, a hardware emulator is also available.

1.2. CR816 Architecture

The CR816 is a 3-stage pipeline, 8-bit RISC register-memory processor based on a Harvard architecture. It is characterized by separate instruction and data address busses. The program memory part includes the program memory interface, the program counter and the internal hardware stack. The data memory part includes the data memory interface, 16 8-bit data registers, the ALU (including a 8bx8b multiplier), and the data memory address computation unit. The control unit is responsible to fetch the instructions from the program memory and to generate the various control signals, including the internal gated clock signals, that manage the internal registers, operands and command selection.

Figure 1.1 gives the CR816 core diagram that highlights the main functional blocks of the core. Figure 1.2 gives the CR816 interface signals grouped by function.

 © 2001 Xemics, v4.5

FIGURE 1.1: CR816 core diagram.

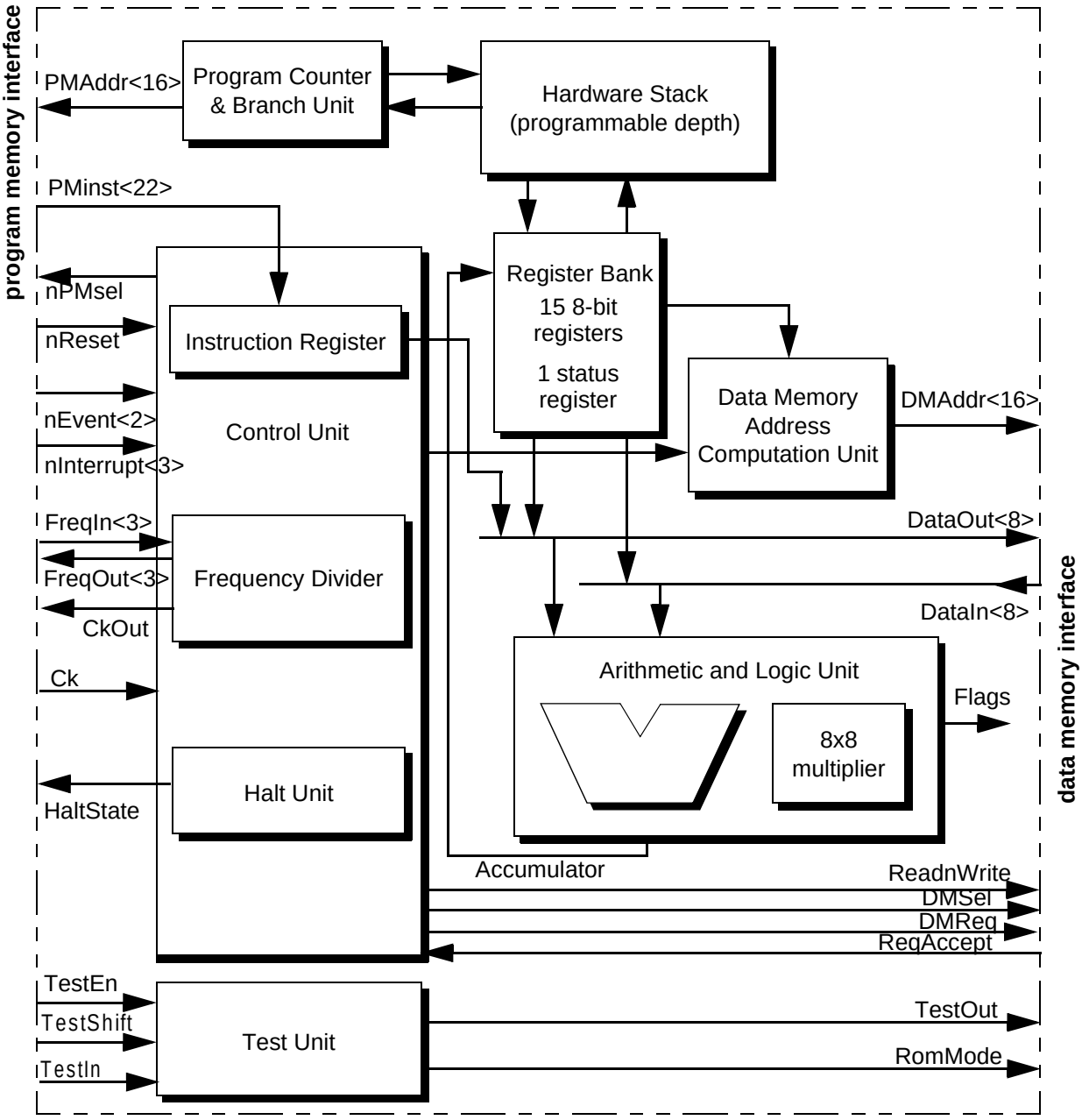
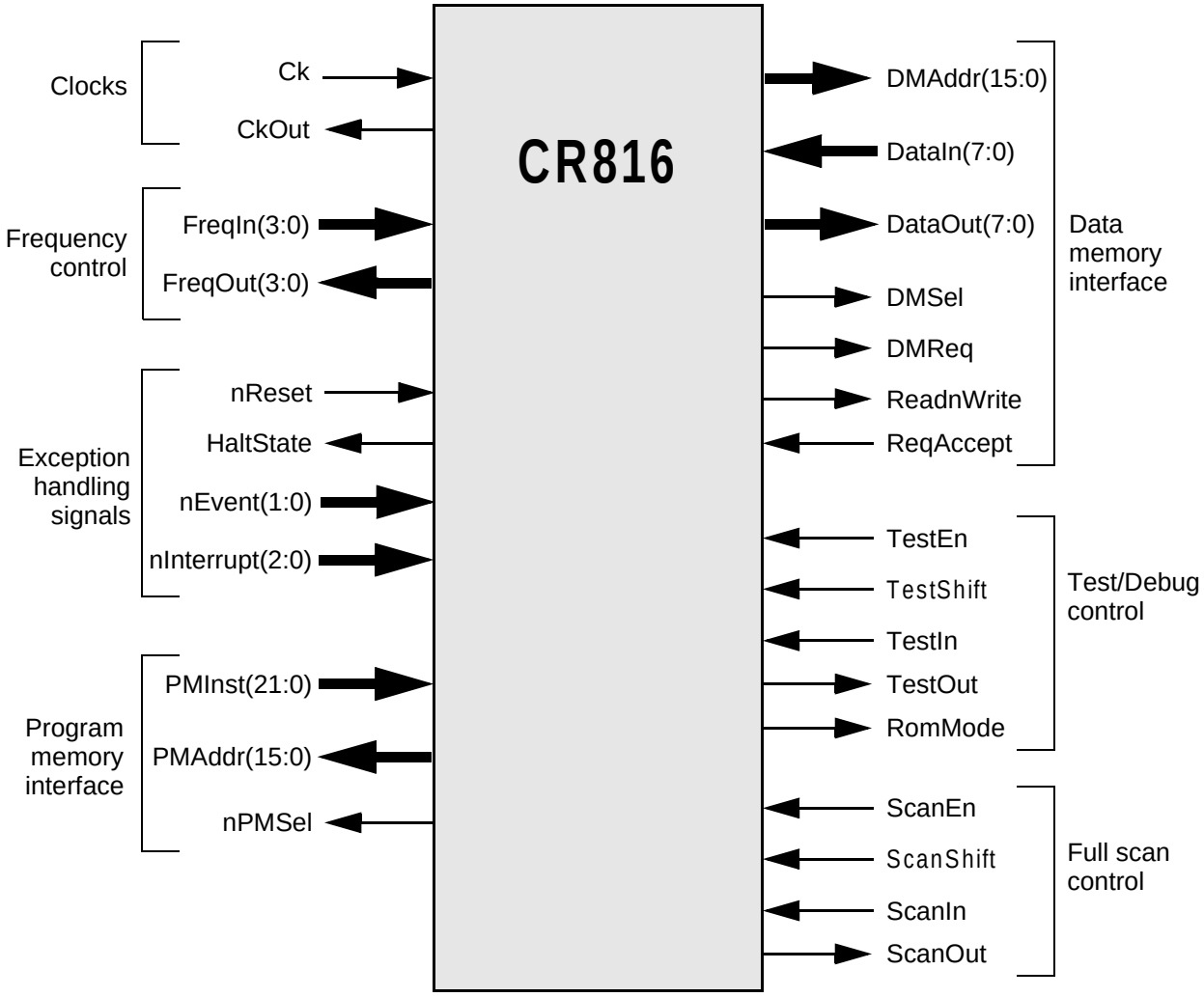


FIGURE 1.2: CR816 interface signals



1.3. Interface Signals Description

This paragraph describes the input/output signals for the CR816 core.

The following conventions are used:

- Signals that are active low have names that start with the prefix “n”. Examples: **`nReset`**, **`nEvent`**. Signal names without this prefix are active high.
- When qualifying a signal, the term “asserted” means that the signal is active, while the term “deasserted” or “negated” means that the signal is inactive regardless of whether the active state is represented by a high or low voltage.
- Signal busses are denoted with the range “(MSB:LSB)” where the index of the most significant bit (MSB) is given first and the index of the least significant bit (LSB) is given last.

1.3.1. Clock Signals

TABLE 1.1: Clock signals.

Name	Type	Description
Ck	input	Main clock.
CkOut	output	Processor internal clock. This signal is related to the **Ck** input clock. When the clock frequency is divided (using the **FreqIn** bus), the **CkOut** signal corresponds to the **Ck** signal divided by the corresponding factor. When the processor is halted, **CkOut** remains at 0.

1.3.2. Clock Frequency Control

The clock frequency can be lowered from 1/2 to 1/16 of the initial clock frequency by hardware with the **FreqIn** input signal or by software through the use of the FREQ instruction. In standard applications the **FreqIn** bus is directly connected to the **FreqOut** bus. See "6.4. Frequency Division".

TABLE 1.3: Frequency control signals.

Name	Type	Description
FreqIn(3:0)	input	Frequency selection. This bus defines the internal processor clock.
FreqOut(3:0)	output	Frequency programming. This bus is the output of an internal register that is programmed by the FREQ instruction.

1.3.3. Exception Handling Signals

These signals control the operation of the core outside its normal processing behaviour. See "Chapter 4 Exception Processing".

TABLE 1.4: Control signals.

Name	Type	Description
nReset	input	Reset of the processor. Asynchronous active low signal.
nEvent(1:0)	input	Event request. The signals set the EV1 and the EV0 flags of the status register. The event signals can be used to wake up the core when it is in Halt mode. They can also be used in conditional loops to test external signals without having to access the data memory bus. The signals must be tied to a logical 1 when not used.

TABLE 1.4: Control signals.

Name	Type	Description
nInterrupt(2:0)	input	Interrupt request. The signals can be used to force a jump (call) to the interrupt sub-routine or to wake up the core when it is in Halt mode. The signals must be tied to a logical 1 when not used.
HaltState	output	Indicates that the core is in Halt mode.

1.3.4. Program Memory Interface

The CR816 core has specific busses for interfacing with program memories (ROM, Flash, etc.).

TABLE 1.5: Program memory interface signals.

Name	Type	Description
PMInst(21:0)	input	Program memory instruction. This signal holds the instruction that is read from the Program memory. One instruction is 22 bits wide. Bit 21 is the most significant bit.
PMAddr(15:0)	output	Program memory address. This signal holds the address of a bus transfer between the core and the Program memory. The address bus is capable of accessing 64K 22-bit instructions. Bit 15 is the most significant bit.
nPMSel	output	Program memory select. This signal indicates, when asserted (low), that the processor is reading instructions. This signal can be used to precharge low power memories when deasserted (high).

1.3.5. Data Memory or Peripheral Interface

Note The precharge signal for low power Program memory, if any, must be generated outside the core, typically by oring the **ck1** clock and the **halt_state** signal.

The CR816 core has specific busses for interfacing with the data memories or peripherals.

TABLE 1.6: Data memory or peripheral interface signals.

Name	Type	Description
DataIn(7:0)	input	Data from memory or peripheral. Signal bus to read data from the external memory or peripheral.
ReqAccept	input	Request accept. When asserted (high), this signal indicates that an access, either read or write, to the data memory or a peripheral has occurred. This signal may be deasserted (low) to access slow peripherals or for DMA and multi-processor communications. In this case, the processor enters into the Wait mode (see "6.8. Wait Mode"). This signal is kept asserted in Normal mode.

TABLE 1.6: Data memory or peripheral interface signals.

Name	Type	Description
DataOut(7:0)	output	Data to memory or peripheral. Signal bus to write data to the external memory or peripheral.
DMAddr(15:0)	output	Data memory or peripheral address. Signal bus that holds the address of the external data memory or peripheral the core wants to access. This signal must be stable before **DMSel** is active and must remain stable after **DMSel** becomes inactive.
DMSel	output	Data memory or peripheral select. When asserted (high), this signal indicates that an access, either read or write, to the data memory or a peripheral is taking place. The access cannot be delayed anymore. This signal is asserted once the **ReadnWrite** and **DMAddr** signals are stable. The data are latched in the peripheral or in the processor at the falling edge of the **DMSel** signal.
DMReq	output	Data memory or peripheral request. When asserted (high), this signal indicates that an access, either read or write, to the data memory or a peripheral is requested by the current instruction. This signal, in conjunction with the **ReadnWrite** and the **DMAddr** signals can be used to control the **ReqAccept** signal in multi-processor configurations. This signal remains asserted as long as the access has not occurred (e.g. in Wait mode). This signal should not be used as a clock signal since glitches may appear during the instruction decoding phase.
ReadnWrite	output	Read/write signal. This signal indicates whether the data access is a read access (**ReadnWrite** asserted/high) or a write access (**ReadnWrite** deasserted/low). This signal should not be used as a clock signal since glitches may appear during the instruction decoding phase.

1.3.6. Test Mode Control

The CR816 provides built-in test capabilities, in addition to the support of full scan testing. It basically consists in a shift register in which instructions are fed in serially. The output of the shift register is a sequence that holds the content of various status flags and internal registers. See "Chapter 5 Test Capabilities".

TABLE 1.7: Test mode control signals.

Name	Type	Description
TestEn	input	Test enable. This signal is used to put the processor in test mode. Test mode is engaged when **TestEn** is asserted (high). In Normal mode this signal must remain deasserted (low).
TestShift	input	Test shift. When **TestEn** is asserted, this signal switches the processor between shift and execute phases. In Normal mode this signal must remain deasserted (low).

TABLE 1.7: Test mode control signals.

Name	Type	Description
TestIn	input	Test shift register input. This signal is used during the test mode operations to shift instructions inside the processor at the rising edge of **Ck**. Each instruction is shifted in with its LSB first.
TestOut	output	Test shift register output. This signal is the output of the internal shift register. It is updated at the rising edge of **Ck**. It is used to check the output of the processor during test mode. In normal mode this signal is forced to 1.
RomMode	output	Indicates that the core is in ROM mode.

1.3.7. Scan Signals

These signals control the full scan testing of the core.

TABLE 1.8: Scan signals.

Name	Type	Description
ScanEn	input	Scan enable. This signal disables all clock gatings and puts the core in scan mode.
ScanShift	input	Scan chain shift.
ScanIn	input	Scan chain input.
ScanOut	output	Scan chain output.

Note the **ScanShift**, **ScanIn** and **ScanOut** signals are only defined at the core interface to ease future scan insertion.

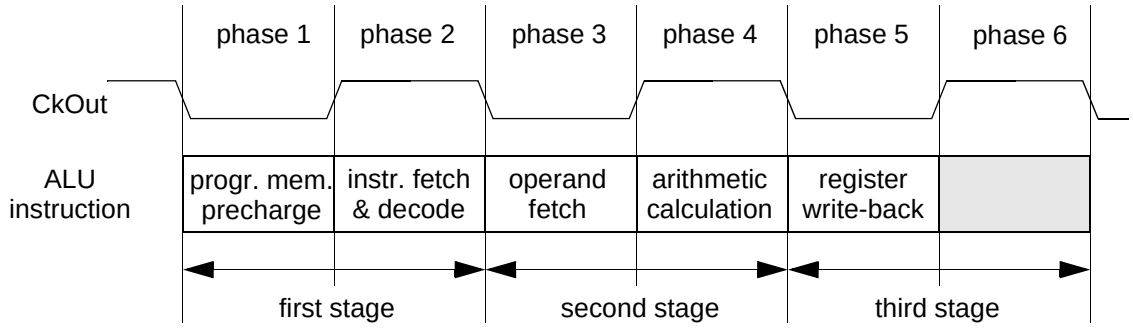
 © 2001 Xemics, v4.5

1.4. Pipeline

The CR816 architecture is based on a three-stage pipeline. An instruction enters the pipeline at each clock cycle and is executed in a maximum of three cycles. The pipeline suffers no penalty such as delay slots, branch delay or branch latency compared to most RISC processors. Thus the number of clock cycles per instruction (CPI) is exactly one, for every kind of instruction. As a result, the number of clock cycles needed to execute a task is the number of executed instructions.

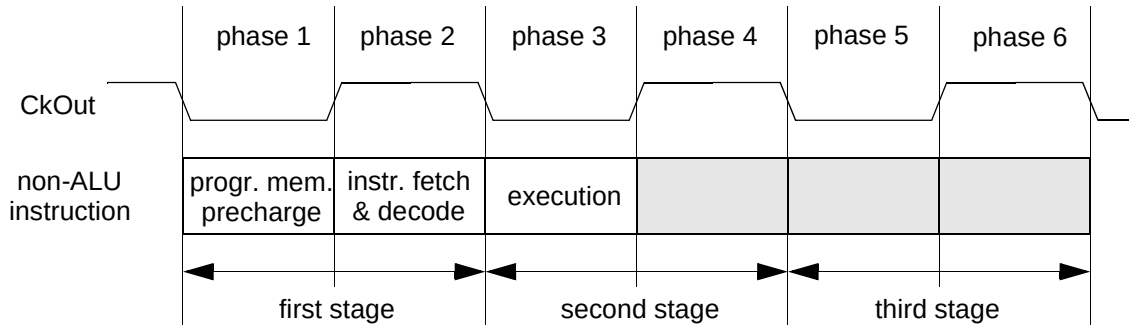
Arithmetic and logic instructions are executed using the three pipeline stages (see Figure 1.3). In the first stage, the program memory is precharged and the instruction is fetched and decoded (phase 2). The second stage decodes the instruction and fetches the operands either in the data memory (read operation) or in the internal register bank (phase 3). These operands are used to perform the ALU instruction (phase 4). The third stage is used to store the result back in the internal register (phase 5). ALU instructions always store the result in internal registers (no write to the data memory).

FIGURE 1.3: Pipeline sequencing for an ALU instruction.



All other instructions (branches, store and miscellaneous instructions) use only the first two stages of the pipeline (see Figure 1.4). In the first stage, the program memory is precharged and the instruction is fetched and decoded (phase 2). For branch conditions (especially conditional branches), the instruction is decoded during phase 2. During phase 3, the instruction is executed. This early decoding is the secret of the high efficiency of the CR816 pipeline. The phase 2 of a conditional branch instruction can be simultaneous with the arithmetic calculation of a previous instruction (phase 4). The conditional branch is decoded at the same time as the flags are coming out of the ALU, allowing the CR816 to make the correct branching at phase 3 (no branch latency).

FIGURE 1.4: Pipeline sequencing for a non-ALU instruction.

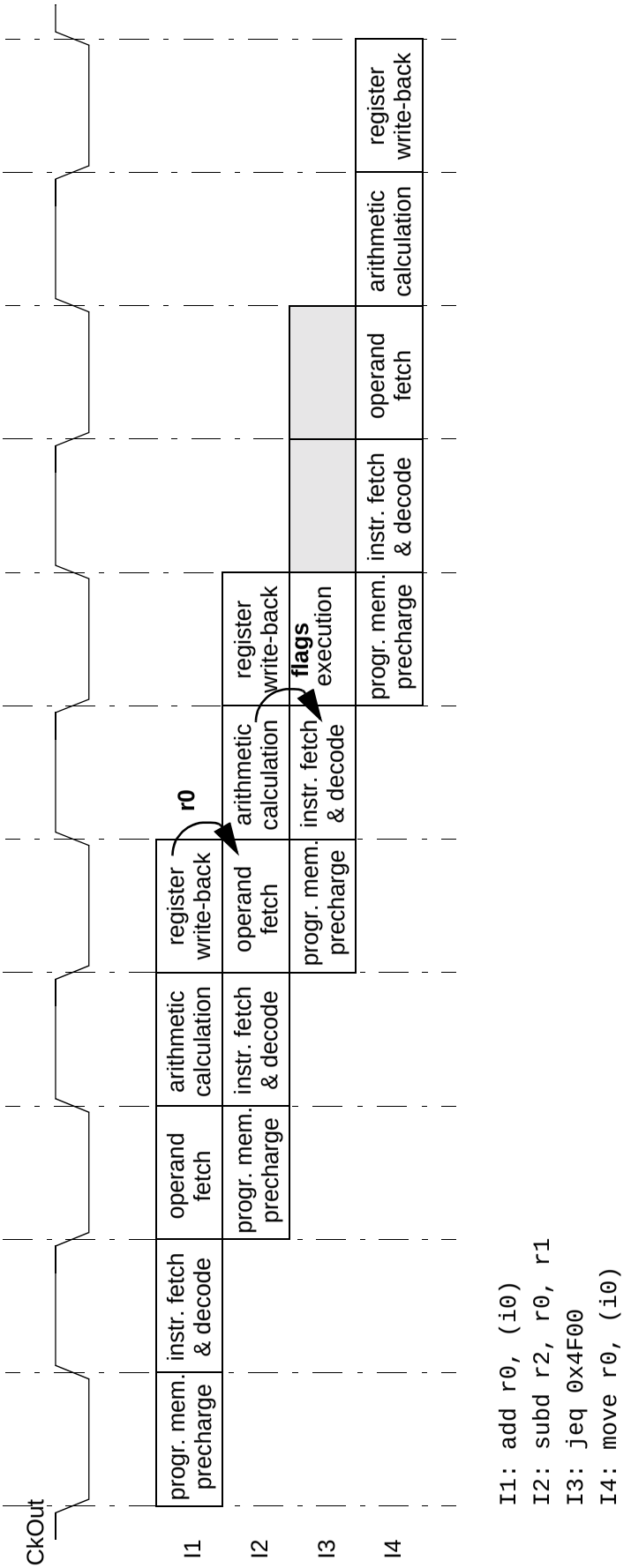


This powerful architecture suppresses the need for either branch delay or branch prediction unit, hence simplifying hardware design. Thus, a CPI = 1 is not a peak value, but rather a characteristic of the CR816 architecture. This makes the pipeline very efficient and low power.

A simple example of the pipeline sequencing is given in Figure 1.5. A first instruction "`add r0, (i0)`" is fetched from the program memory. The two operands are fetched during cycle 2 (one cycle here is two phases). One operand is read directly from the internal register bank while the second is read from the data memory. The result of this calculation is stored in the register **`r0`** at the beginning of cycle 3.
In parallel, during cycle 2, the following instruction "`subd r2, r0, r1`" is fetched. The operand fetch for this instruction is simultaneous with the register write of the previous instruction, so **`r0`** is directly by-passed from the output of the ALU to the operand selection block (first arrow).
The third instruction, "`jeq 0x4F00`" is fetched during cycle 3. This instruction is a conditional branch depending on the result of the previous ALU instruction. As explained before, for branch instructions, the decoding operation takes place during phase 2 (in this case, at the end of cycle 3). To correctly evaluate the branch condition, the flags are by-passed directly from the ALU output to the branch unit (second arrow).

Note	Due to the presence of the pipeline, the execution of some instructions leads to results that are different to what they would be with a non pipelined architecture. These special cases are explained in "6.6. Pipeline Exception".

FIGURE 1.5: Example of pipeline sequencing.

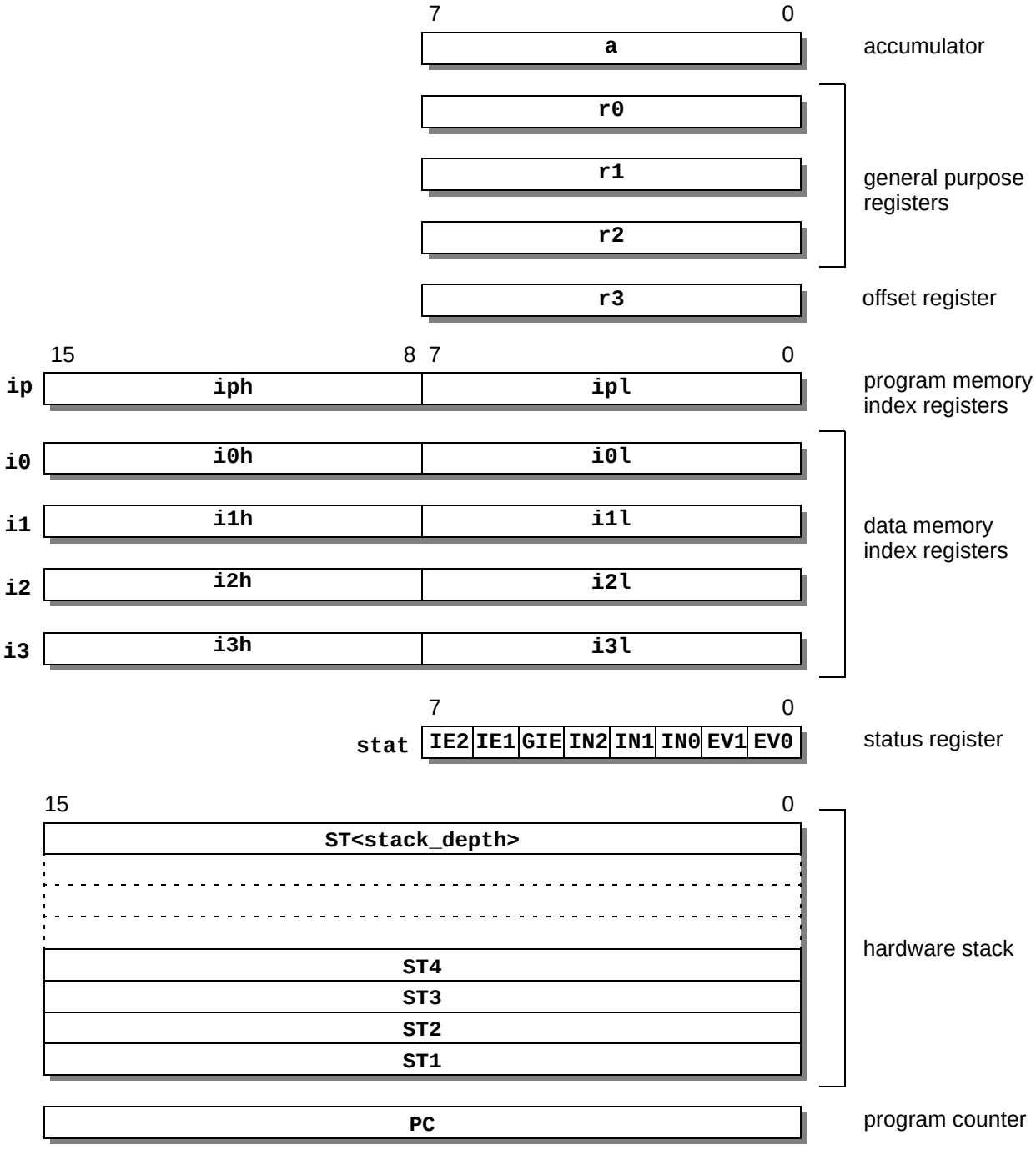


1.5. Programmer's Model

The CR816 contains several internal registers that can be accessed by the programmer. These registers are classified into three categories (Figure 1.6):

- Data registers
- Program counter and stack registers
- Flags

FIGURE 1.6: CR816DL registers.



1.5.1. Data Registers

The CR816 has 15 8-bit registers that can be used for all arithmetic and logic operations as well as for load/store instructions. Furthermore, most of the registers are dedicated for special instructions or for special addressing modes (Table 1.9).

TABLE 1.9: Data registers.
All registers are 8 bit wide, except when explicitly mentioned otherwise.

Name	Type	Description
a	accumulator	This is a pseudo-register that is located at the output of the ALU. The accumulator is modified by every arithmetic or logical operation, including comparison and conditional move. The accumulator contains a copy of the result of the instruction, even if it is not the destination register. Its use in place of other available registers, notably to store temporary results, allows to limit the power consumption.
r0 r1 r2	general purpose registers	These registers can be used for any arithmetic or logical operation as well as for load/store operations.
r3	data memory offset register	This register is the offset register when addressing the data memory. It can also be used for any arithmetic or logical operation as well as for load/store operations.
ip (iph, ipl)	program memory index register	This 16-bit register is used to indirectly address the 64K of program memory. It is also used by branch instructions (CALLS, RETS, POP) to save addresses. The most significant byte **iph** and the least significant byte **ipl** can be used as separate registers. These registers can also be used for any arithmetic or logical operation as well as for load/store operations.
i0 (i0h,i0l) **i1 (i1h,i1l)** **i2 (i2h,i2l)** **i3 (i3h,i3l)**	data memory index registers	These 4 16-bit registers are used as indexes to address the 64K of data memory. The most significant byte **ixh** and the least significant byte **ixl** can be used as separate registers (x=0,1,2,3). These registers can also be used for any arithmetic or logical operation as well as for load/store operations.

1.5.2. Status Register

The status register **stat** is used to control the interrupts and the events. It contains enable bits for the interrupts and the current status of events and interrupts (Table 1.10).

TABLE 1.10: Status register bits.

Bit	Symbol	Name	Description
7	IE2	Level 2 Interrupt Enable	Enables (when set) or disables (when cleared) the interrupt request of level 2.
6	IE1	Level 1 Interrupt Enable	Enables (when set) or disables (when cleared) the interrupt request of level 1.
5	GIE	General Interrupt Enable	This bit enables or disables all interrupt requests (level2, level1 and level0). This bit is cleared at reset to avoid interrupt requests before proper initialization of the application. This bit must be set to 1 to enable interrupts. It is automatically cleared when the core jumps to an interrupt subroutine. This bit is automatically set when returning from an interrupt subroutine by using the RETI instruction. This mecanism avoids multiple interrupt requests within an interrupt subroutine.
4	IN2	Interrupt Request 2	This bit indicates, when set to 1, that an interrupt request of level 2 has occurred. This can happen either by asserting the external **nInterrupt(2)** signal or by setting the corresponding bit to 1 using an ALU or load/store instruction. This bit can only be cleared explicitly. An interrupt of level 2 will take place whenever the bits IN2, IE2 and GIE are set.
3	IN1	Interrupt Request 1	This bit indicates, when set to 1, that an interrupt request of level 1 has occurred. This can happen either by asserting the external **nInterrupt(1)** signal or by setting the corresponding bit to 1 using an ALU or load/store instruction. This bit can only be cleared explicitly. An interrupt of level 1 will take place whenever the bits IN1, IE1 and GIE are set.
2	IN0	Interrupt Request 0	This bit indicates, when set to 1, that an interrupt request of level 0 has occurred. This can happen either by asserting the external **nInterrupt(0)** signal or by setting the corresponding bit to 1 using an ALU or load/store instruction. This bit can only be cleared explicitly. An interrupt of level 0 will take place whenever the bits IN0 and GIE are set.
1	EV1	Event Request 1	This bit indicates, when set to 1, that a event request of level 1 has occurred. This can happen by asserting the external **nEvent(1)** signal or by setting the corresponding bit using a ALU or load/store instruction. This bit can only be cleared explicitly.
0	EV0	Event Request 0	This bit indicates, when set to 1, that a event request of level 0 has occurred. This can happen by asserting the external **nEvent(0)** signal or by setting the corresponding bit using a ALU or load/store instruction. This bit can only be cleared explicitly.

Note All bits of the status register are cleared upon reset.

1.5.3. Program Counter and Stack Registers

The program counter (**PC**) is a 16-bit register that stores the address of the next instruction to be read from the program memory. This register is cleared at reset. When no exception or branch instruction are executed, the program counter is incremented after the fetch of each instruction. Branch instructions cause the PC to contain an immediate value from the index register **ip**. Exceptions cause the PC to contain a predefined, hard-coded, value (see "Chapter 4 Exception Processing").

The hardware stack is a register bank whose size may be selected depending on the application. The default stack depth is 4 levels. Typical stack depths are four or five levels. Each level in the stack can be used to store a return address. These addresses are used for subroutine calls as well as for interrupt processing. The access to the stack is of a LIFO type (Last In First Out). The return address is stored in the stack by pushing it while it is recovered by popping it out (see "6.2. Hardware Stack Depth").
It is also possible to have a variable stack depth by using the data memory and a software stack mechanism (see "6.1. Software Stack").

1.5.4. Flags

Three flags can be set depending on the results of arithmetic operations and of the evaluation of branch conditions: the zero flag (Z), the carry flag (C), and the overflow flag (V) (Table 1.11). These flags are set by the SFLAG instruction and can be read with the RFLAG instruction. For each instruction described in Section 2.1 it is mentioned which flag(s) is(are) modified by the execution of the instruction.

TABLE 1.11: Flags.

Flag	Name	Description
Z	Zero Flag	This flag is set to 1 when the result of an ALU operation is zero. It is also modified when data is moved to a register.
C	Carry Flag	This flag is only modified by arithmetic and shift operations. For shift operations, the flag contains the bit that is shifted out, i.e. the LSB for a right shift and the MSB for a left shift. For arithmetic operations with unsigned numbers, the flag also indicates whether an overflow (addition or equivalent operation, C = 1) or and underflow (subtraction or equivalent operation, C = 0) occurred.
V	Overflow Flag	This flag is only modified by arithmetic and shift operations. For shift and arithmetic operations with signed numbers, the flag is set when an overflow or an underflow occurred. It is cleared otherwise.

Note All flags must be considered as undefined after a multiplication.

1.6. Data Memory Addressing Modes

The data memory is organised as 256 pages, indexed from 0 to 255, of 256 bytes each that can be accessed by two kinds of addressing modes. The direct mode can only access page 0, while four indexed modes can access the whole 64K of data memory. The indexed modes may indifferently use any 16-bit index registers **i0**, **i1**, **i2**, and **i3** that can be splitted into a MSB byte **ixh** and a LSB byte **ixl**, x=0,1,2,3.

See "6.7. Use of Addressing Capabilities" for more specialized use of addressing modes.

1.6.1. Direct Addressing Mode

This mode is limited to access the first 256 bytes of the data memory (page 0). Page 0, also called fast memory, uses the address range 0x0000 to 0x00FF. It is often used for peripheral registers or to store global variables since no index needs to be initialized and the transfer can be done in a single instruction.

The address is directly specified in the instruction opcode. Due to the limited instruction length, only the 8 least significant bits of the opcode are considered.

The assembly syntax for the direct addressing mode is `addr:8` where `addr` is an 8-bit value.

An example of instruction using this mode is:

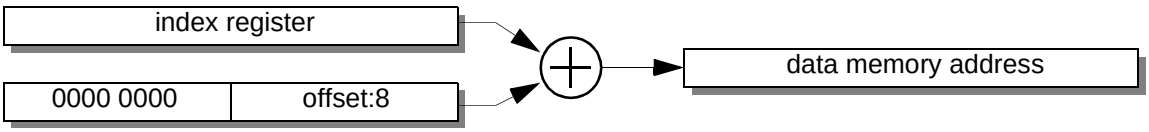
```
MOVE r0, 0x56
```

This instruction moves the content of the data memory at address 0x56 to the register **r0**.

1.6.2. Indexed Addressing Mode with Immediate Offset

In this mode, an 8-bit unsigned offset specified in the instruction is left padded with zeroes to get a 16-bit value and then added to the value in the selected index register to get the effective data memory address (Figure 1.7).

FIGURE 1.7: Indexed addressing mode with immediate offset.



The assembly syntax for this addressing mode is `(ix, offset:8)` where `ix` is either **i0**, **i1**, **i2** or **i3** and `offset` is an 8-bit unsigned value. The notation `(ix)` can be used when the offset is null.

Examples of instructions using this mode are:

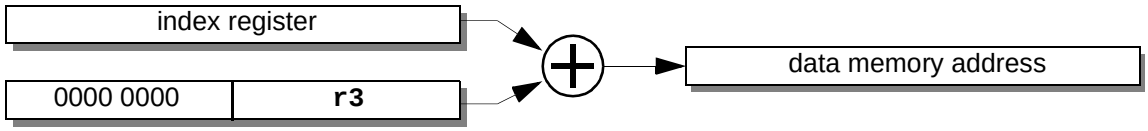
```
MOVE r0, (i0, 0x7E)
MOVE r2, (i1, 0x00)
MOVE r2, (i1)
```

The last two move instructions equivalently use a null offset.

1.6.3. Indexed Addressing Mode with Register Offset

In this mode, a positive offset value is stored in register **r3**. The offset is left padded with zeroes to get a 16-bit value and then added to the value in the selected index register to get the effective data memory address (Figure 1.8).

FIGURE 1.8: Indexed addressing mode with register offset.



The assembly syntax for this addressing mode is `(ix, r3)` where `ix` is either **i0**, **i1**, **i2** or **i3**.

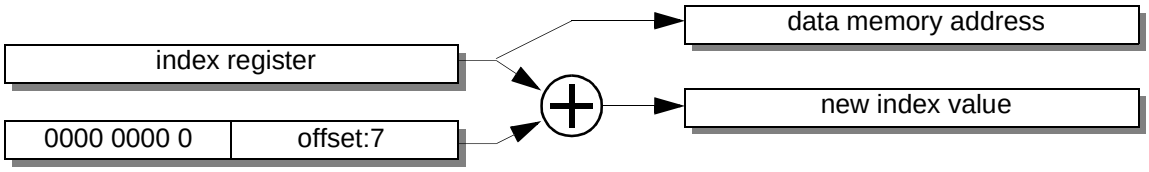
An example using this mode is:

```
MOVE r0, (i3, r3)
```

1.6.4. Indexed Addressing Mode with Post-Incrementation of the Index

In this mode, a 7-bit *positive* offset specified in the instruction is left padded with zeroes to get a 16-bit value. The data memory address is contained in the index register and the offset value is added to the index register after the memory access is done (Figure 1.9).

FIGURE 1.9: Indexed addressing mode with post-incrementation of the offset.



The assembly syntax for this addressing mode is `(ix, offset:7)+` where `ix` is either **i0**, **i1**, **i2** or **i3** and `offset:7` is a 7-bit *positive* value. The notation `(ix)+` can be used when the offset is equal to 1.

Examples of instructions using this mode are:

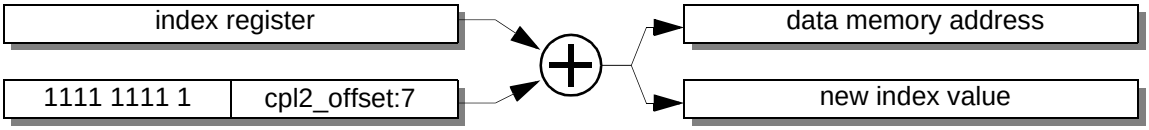
```
MOVE r0, (i0, 0x7E)+
MOVE r2, (i1, 0x01)+
MOVE r2, (i1)+
```

The last two move instructions equivalently use an offset equal to 1.

1.6.5. Indexed Addressing Mode with Pre-Decrementation of the Index

In this mode, a 7-bit *positive* offset specified in the instruction is left padded with ones to get a 16-bit *negative* 2's complement value. This value is added to the value in the selected index register to get the effective data memory address. The index register stores the new index value after the memory access is done (Figure 1.10).

FIGURE 1.10: Indexed addressing mode with pre-decrementation of the offset.



The assembly syntax for this addressing mode is `-(ix, offset:7)` where `ix` is either **`i0`**, **`i1`**, **`i2`** or **`i3`** and `offset:7` is a 7-bit *positive* value. The notation `-(ix)` can be used when the offset is equal to 1.

Examples of instructions using this mode are:

```
MOVE r0, -(i0, 0x7E)   ; 16-bit offset value will be 0xFF82 (= -0x7E)
MOVE r2, -(i1, 0x01)   ; 16-bit offset value will be 0xFFFF (= -0x01)
MOVE r2, -(i1)         ; idem
```

The last two move instructions equivalently use an offset equal to 1.

Chapter 2
Instruction Set

The instruction is tailored to support high-level languages as well as efficient assembler programming. The CR816 with its unique architecture is offering both RISC and CISC instructions to achieve a very dense program code. The instructions can be grouped into four main categories:

- Branch instructions.
- Transfer instructions.
- Arithmetic and logical instructions.
- Special instructions.

The instructions of the CR816 are listed in Table 2.1.

Branch instructions provides different methods of handling program branches. The programmer can either use the internal hardware stack to store the return address or a software stack implemented in the data memory. The hardware stack has a limited size (see "6.2. Hardware Stack Depth") but is very efficient in term of instruction density and power consumption. The software stack provides flexibility for high-level languages and is only limited by the size of the data memory.

Unlike most RISC microprocessors, the CR816 core provides instructions that can perform arithmetic and logical operations with operand stored either in the data memory or in the internal registers. The result is always stored into an internal register and can be transferred later in the data memory.

Furthermore, similarly to classic 8-bit microprocessors, the CR816 architecture provides an accumulator (**a**) located at the ALU output that stores the last ALU result. The accumulator in mapped in the register bank. All arithmetic operations support both signed and unsigned operations. Signed numbers use the 2's complement representation.

Several instruction mnemonics, which do not belong to the CR816's native instruction set are provided by the assembler to ease the writing and reading of assembler code. See "2.2. Assembler Aliases".

© 2001 Xemics, v4.5

TABLE 2.1: CR816 instruction set.
ALU instructions modify the accumulator.

Mnemonic	ALU instruction	Description	Page
ADD	yes	Addition without carry.	2-3
ADDC	yes	Addition with carry.	2-4
AND	yes	Logical AND.	2-5
CALL	no	Jump to subroutine.	2-6
CALLS	no	Jump to subroutine, using **ip** as return address.	2-7
CMP	yes	Unsigned compare.	2-8
CMPA	yes	Signed compare.	2-9
CMVD	yes	Conditional move, if carry clear.	2-10
CMVS	yes	Conditional move, if carry set.	2-11
CPL1	yes	One's complementation.	2-12
CPL2	yes	Two's complementation without carry.	2-13
CPL2C	yes	Two's complementation with carry.	2-14
DEC	yes	Decrementation without carry.	2-15
DECC	yes	Decrementation with carry.	2-16
FREQ	no	Frequency division selection.	2-17
HALT	no	Halt mode selection.	2-18
INC	yes	Increment without carry.	2-19
INCC	yes	Increment with carry.	2-20
Jcc	no	Conditional jump.	2-21
MOVE	yes	Data move.	2-22
MUL	yes	Unsigned multiplication.	2-24
MULA	yes	Signed multiplication.	2-25
NOP	no	No operation.	2-26
OR	yes	Logical OR.	2-27
PMD	no	Program memory dump.	2-28
POP	no	Pop **ip** index from hardware stack.	2-29
PUSH	no	Push **ip** index onto hardware stack.	2-30
RET	no	Return from subroutine.	2-31
RETI	no	Return from interrupt.	2-32
SFLAG	yes	Save flags.	2-33
SHL	yes	Logical shift left without carry.	2-34
SHLC	yes	Logical shift left with carry.	2-35
SHR	yes	Logical shift right without carry.	2-36
SHRA	yes	Arithmetic shift right.	2-37
SHRC	yes	Logical shift right with carry.	2-38
SUBD	yes	Subtraction without carry (op1 - op2).	2-39
SUBDC	yes	Subtraction with carry (op1 - op2).	2-40
SUBS	yes	Subtraction without carry (op2 - op1).	2-41
SUBSC	yes	Subtraction with carry (op2 - op1).	2-42
TSTB	yes	Test bit.	2-43
XOR	yes	Logical exclusive OR.	2-44

2.1. Instruction Set Details

The following pages describe each CR816 instruction in details.

ADD - Addition without Carry

		Operation
Syntax:	`ADD REG, #data:8`	`REG := REG + data`
	`ADD REG, <eaddr>`	`REG := REG + DM[<eaddr>]`
	`ADD REGi, REGj, REGk`	`REGi := REGj + REGk`
	`ADD REGi, REGj`	`REGi := REGj + REGi`

Description: Adds the two operands using binary addition.
The first operand is the destination register into which the result is stored.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if an overflow occurred (signed numbers).
C = 1 if a carry is generated or overflow (unsigned numbers).
C = 0 if underflow (unsigned numbers).
Z = 1 if the result is zero.
Otherwise the flags are cleared.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	Instruction	After	V	C	Z
`r0 = 0x0F`	`ADD r0, #0xF6`	`r0 = a = 0x05`	0	1	0
`r0 = 0x43`	`ADD r0, #0x42`	`r0 = a = 0x85`	1	0	0
`r0 = 0xFF`	`ADD r0, #0x01`	`r0 = a = 0x00`	0	1	1

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1	0	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	1	1	0	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	1	0	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	1	0	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	1	0	0	<regi>				1	1	1	1	1	1	<ixs>	

8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	1	0	0	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	1	0	0	<regk>				<regj>				<regi>			

ADDC - Addition with Carry

Syntax:

	Operation
`ADDC REG, #data:8`	`REG := REG + data + C`
`ADDC REG, <eaddr>`	`REG := REG + DM[<eaddr>] + C`
`ADDC REGi, REGj, REGk`	`REGi := REGj + REGk + C`
`ADDC REGi, REGj`	`REGi := REGj + REGi + C`

Description: Adds the two operands and the Carry flag using binary addition.
The first operand is the destination register into which the result is stored.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if an overflow occurred (signed numbers).
C = 1 if a carry is generated or overflow (unsigned numbers).
C = 0 if underflow (unsigned numbers).
Z = 1 if the result is zero.
Otherwise the flags are cleared.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	**Instruction**	**After**	**V**	**C**	**Z**
`r0 = 0x20, C = 0`	`ADDC r0, #0x20`	`r0 = a = 0x40`	0	0	0
`r0 = 0x80, C = 1`	`ADDC r0, #0x82`	`r0 = a = 0x03`	1	1	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1	0	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	1	1	0	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	1	0	1	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	1	0	1	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	1	0	1	<regi>				1	1	1	1	1	1	<ixs>	

8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	1	0	1	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	1	0	1	<regk>				<regj>				<regi>			

© 2001 Xemics, v4.5

AND - Logical AND

		Operation
Syntax:	`AND REG, #data:8`	`REG := REG AND data`
	`AND REG, <eaddr>`	`REG := REG AND DM[<eaddr>]`
	`AND REGi, REGj, REGk`	`REGi := REGj AND REGk`
	`AND REGi, REGj`	`REGi := REGj AND REGi`

Description: Performs a logical AND between the two operands.
The first operand is the destination register into which the result is stored. This instruction is also used to define the CLRB alias (see "2.2. Assembler Aliases").

Flags modified:

V	C	Z
no	no	yes

Z = 1 if the result is zero, 0 otherwise.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	Instruction	After	Z
`r1 = 0x0F, a = 0xF0`	`AND a, r1`	`a = 0x0`	`1`
`i0h = 0x99`	`AND i0h, #0x33`	`a = i0h = 0x11`	`0`

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	1	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	0	0	1	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	0	1	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	0	1	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	1	0	<regi>				1	1	1	1	1	1	<ixs>	

with 8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	1	0	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	0	1	0	<regk>				<regj>				<regi>			

CALL - Jump to Subroutine

Syntax:

```
CALL jaddr:16
CALL ip
```

Operation

1) Hardware stack pushed
2) `ST1 := PC+1`
3) `PC` equals either
`jaddr:16` (immediate address) or
`ip` (indexed address)

Description: Jumps to the subroutine whose address is given in the operand.
The program memory address of the instruction immediately following the `CALL` instruction is pushed on top of the hardware stack. The program execution then continues at the address specified in the instruction or in the **`ip`** register.

Flags modified:

V	C	Z
no	no	no

Accu. modified: No.

Example:

Before	Instruction	After
`PC = 0x43E7`	`CALL 0x0A54`	`PC = 0x0A54`
`ST1 = 0x5555`		`ST1 = 0x43E8`
`ST2 = 0x7777`		`ST2 = 0x5555`
		`ST3 = 0x7777`

Instruction format:

with immediate address:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	n_jaddr:16															

with indexed address:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

© 2001 Xemics, v4.5

CALLS - Jump to Subroutine, ip as Return Address

Syntax:

	Operation
`CALLS jaddr:16`	1) `ip := PC + 1`
`CALLS ip`	2) `PC` equals either `jaddr:16` (immediate address) or **`ip`** (indexed address)

Description: Jumps to the subroutine whose address is given in the operand.
The program memory address of the instruction immediately following the `CALLS` instruction is stored in the **`ip`** program memory index register. The program execution then continues at the address specified in the instruction or in the **`ip`** register.
This instruction is similar to the `CALL` instruction except that the hardware stack is not used.
The content of the **`ip`** register must be saved before and restored after each execution of the `CALLS` instruction when using nested subroutine calls.

Flags modified:

V	C	Z
no	no	no

Accu. modified: No.

Examples:

Before	**Instruction**	**After**
`PC = 0x43E7`	`CALLS 0x1FE4`	`PC = 0x1FE4`
`ip = 0x0A54`		`ip = 0x43E8`
`PC = 0x43E7`	`CALLS ip`	`PC = 0x0A54`
`ip = 0x0A54`		`ip = 0x43E8`

Instruction format:

with immediate address:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	n_jaddr:16															

with indexed address:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

CMP - Unsigned Compare

		Operation
Syntax:	`CMP REG, #data:8`	`data - REG`
	`CMP REG, <eaddr>`	`DM[<eaddr>]- REG`
	`CMP REGj, REGk`	`REGk - REGj`

Description: Substracts the value in the first register from the value defined by the second operand and sets the condition codes according to the result.
The registers are not modified. The operands are handled as unsigned values.

Flags modified:

V	C	Z
yes	yes	yes

Flag computation for `CMP d, s` where d (s) is the destination (source) operand:
`C = 0` if `d > s` else `C = 1`
`Z = 0` if `d ≠ s` else `Z = 1`
`V = C AND NOT(Z)`

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	Instruction	After	V	C	Z
`r0 = 0x50`	`CMP r0, #0x62`	`a = 0x12`	1	1	0
`r0 = 0x50`	`CMP r0, #0x99`	`a = 0x49`	1	1	0
`r0 = 0x50`	`CMP r0, #0x50`	`a = 0x00`	0	1	1
`r0 = 0x50`	`CMP r0, #0x47`	`a = 0xF7`	0	0	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	0	0	0	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	0	0	1	<regi>				0	cpl2_offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	0	0	1	<regi>				1	offset:7						

indexed address with offset in register r3:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	0	1	<regi>				1	1	1	1	1	1	<ixs>	

with 8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	0	1	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	0	0	1	<regk>				<regj>				<regi>			

CMPA - Signed Compare

		Operation
Syntax:	`CMPA REG, #data:8`	`data - REG`
	`CMPA REG, <eaddr>`	`DM[<eaddr>]- REG`
	`CMPA REGj, REGk`	`REGk - REGj`

Description: Substracts the value in the first register from the value defined by the second operand and sets the condition codes according to the result.
The registers are not modified. The operands are handled as signed values.

Flags modified:

V	C	Z
yes	yes	yes

Flag computation for `CMPA d, s` where d (s) is the destination (source) operand:
`C = 0` if `d > s` else `C = 1`
`Z = 0` if `d ≠ s` else `Z = 1`
`V = C AND NOT(Z)`

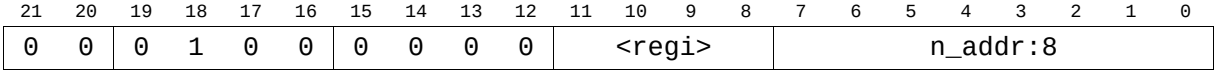
Accu. modified: Yes. Contains the result of the operation.

Examples:

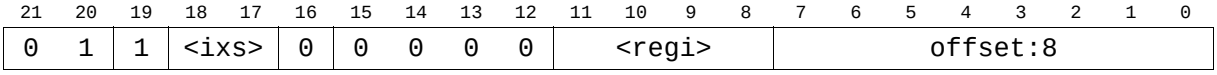
	Before	Instruction	After	V	C	Z
r0 > 0:	r0 = 0x50	CMPA r0, #0x62	a = 0x12	1	1	0
	r0 = 0x50	CMPA r0, #0x50	a = 0x00	0	1	1
	r0 = 0x50	CMPA r0, #0x47	a = 0xF7	0	0	0
	r0 = 0x50	CMPA r0, #0x99	a = 0x49	0	0	0
r0 < 0:	r0 = 0x90	CMPA r0, #0x82	a = 0xF2	0	0	0
	r0 = 0x90	CMPA r0, #0x90	a = 0x00	0	1	1
	r0 = 0x90	CMPA r0, #0xA7	a = 0x17	1	1	0
	r0 = 0x90	CMPA r0, #0x05	a = 0x75	1	1	0

Instruction format:

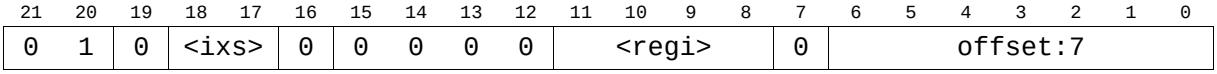
direct addressing:



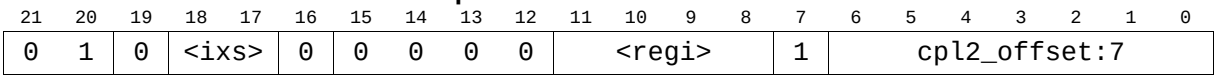
indexed address with offset:



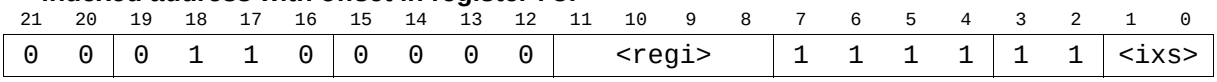
indexed address with offset and post-modification:



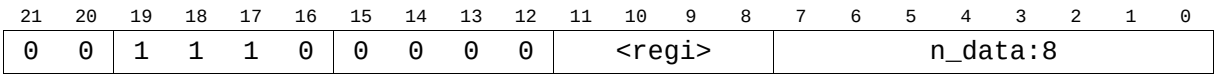
indexed address with offset and pre-modification:



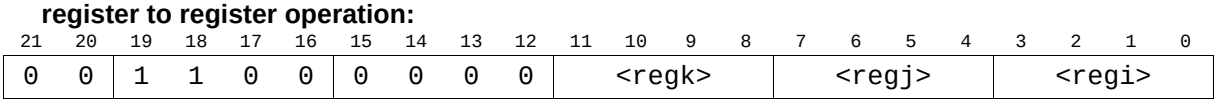
indexed address with offset in register `r3`:



with 8 bit immediate data:



register to register operation:



CMVD - Conditional Move, if Carry Clear

Syntax:

```
CMVD REG, <eaddr>
CMVD REGi, REGj
```

Operation

```
REG  := DM[<eaddr>] iff C = 0
REGi := REGj iff C = 0
```

Description: Conditionally moves the source operand in the destination register if the Carry flag is set to 0.

Flags modified:

V	C	Z
no	no	yes

Z = 1 if the source operand is zero, 0 otherwise.

Accu. modified: Yes. Contains the source operand value.

Examples:

Before	Instruction	After	Z
DM[0] = 88 i3 = 0x0033 C = 0	CMVD ipl, -(i3, 0x33)	ipl = a = 88 i3 = 0x0000	0
DM[0] = 88 i3 = 0x0033 C = 1	CMVD ipl, -(i3, 0x33)	a = 88 i3 = 0x0000	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	1	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	0	0	1	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	0	1	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	0	1	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register r3:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	0	1	0	1	1	1	1	<regj>				<regi>			

© 2001 Xemics, v4.5

CMVS - Conditional Move, if Carry Set

		Operation
Syntax:	`CMVS REG, <eaddr>`	`REG := DM[<eaddr>]` iff `C = 1`
	`CMVS REGi, REGj`	`REGi := REGj` iff `C = 1`

Description: Conditionally moves the source operand in the destination register if the Carry flag is set to 1.

Flags modified:

V	C	Z
no	no	yes

Z = 1 if the source operand is zero, 0 otherwise.

Accu. modified: Yes. Contains the source operand value.

Examples:

Before	Instruction	After	Z

`DM[0] = 88`
`i3 = 0x0033`
`C = 1`
	`CMVD ipl, -(i3, 0x33)`	`ipl = a = 88`
		`i3 = 0x0000`
			0

`DM[0] = 88`
`i3 = 0x0033`
`C = 0`
	`CMVD ipl, -(i3, 0x33)`	`a = 88`
		`i3 = 0x0000`
			0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	1	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	0	0	1	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	0	1	1	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	0	1	1	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	1	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	0	1	1	1	1	1	1	<regj>				<regi>			

CPL1 - One's Complement

		Operation
Syntax:	`CPL1 REG, <eaddr>`	`REG := not DM[<eaddr>]`
	`CPL1 REG`	`REG := not REG`
	`CPL1 REGi, REGj`	`REGi := not REGj`

Description: Computes the one's complement of the source data and stores the result in the destination register.

Flags modified:

V	C	Z
no	no	yes

Z = 1 if the result of the computation is zero
Z = 0 otherwise.

Accu. modified: Yes. Contains the result of the complementation.

Examples:

Before	Instruction	After	Z
`r0 = 0x55`	`CPL1 r0`	`r0 = a = 0xAA`	`0`

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	0	0	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	1	0	0	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	0	0	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	0	0	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	0	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	0	0	0	1	1	1	1	<regj>				<regi>			

© 2001 Xemics, v4.5

CPL2 - Two's Complementation

		Operation
Syntax:	`CPL2 REG, <eaddr>`	`REG := not DM[<eaddr>] + 1`
	`CPL2 REG`	`REG := not REG + 1`
	`CPL2 REGi, REGj`	`REGi := not REGj + 1`

Description: Computes the two's complement of the source data and stores the result in the destination register.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if source operand = 0x80, cleared otherwise.
C = 1 if source operand = 0, cleared otherwise.
Z = 1 if the result of the computation is zero,
Z = 0 otherwise.

Accu. modified: Yes. Contains the result of the complementation.

Examples:

Before	Instruction	After	V	C	Z
`r0 = 0x55`	`CPL2 r0`	`r0 = a = 0xAB`	0	0	0
`r0 = 0x80`	`CPL2 r0`	`r0 = a = 0x80`	1	0	0
`r0 = 0xFF`	`CPL2 r0`	`r0 = a = 0x01`	0	0	0
`r0 = 0x00`	`CPL2 r0`	`r0 = a = 0x00`	0	1	1

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	0	0	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	1	0	0	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	0	0	1	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	0	0	1	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	1	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	0	0	1	1	1	1	1	<regj>				<regi>			

CPL2C - Two's Complementation with Carry

		Operation
Syntax:	`CPL2C REG, <eaddr>`	`REG := not DM[<eaddr>] + C`
	`CPL2C REG`	`REG := not REG + C`
	`CPL2 REGi, REGj`	`REGi := not REGj + C`

Description: Computes the two's complement of the source data and stores the result in the destination register.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if source operand = 0x80, cleared otherwise.
C = 1 if source operand = 0, cleared otherwise.
Z = 1 if the result of the computation is zero,
Z = 0 otherwise.

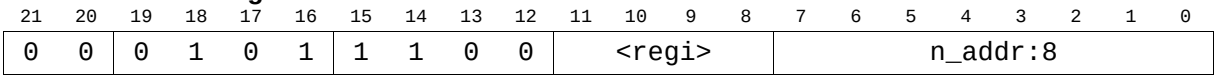
Accu. modified: Yes. Contains the result of the complementation.

Examples:

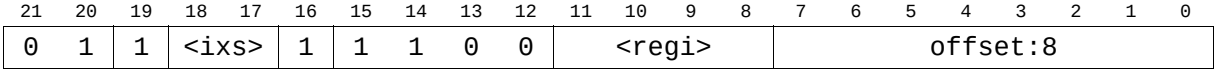
Before	Instruction	After	V	C	Z
`r0 = 0x00, C = 0`	`CPL2C r0`	`r0 = a = 0xFF`	0	0	0
`r0 = 0x00, C = 1`	`CPL2C r0`	`r0 = a = 0x00`	0	1	1
`r0 = 0x80, C = 1`	`CPL2C r0`	`r0 = a = 0x80`	1	0	0
`r0 = 0x55, C = 0`	`CPL2C r0`	`r0 = a = 0xAA`	0	0	0

Instruction format:

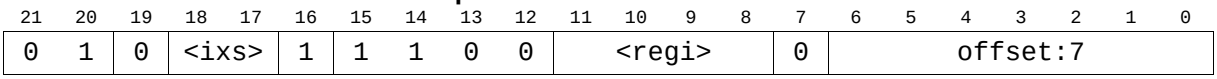
direct addressing:



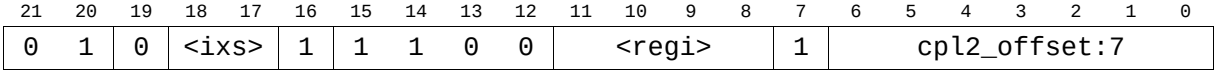
indexed address with offset:



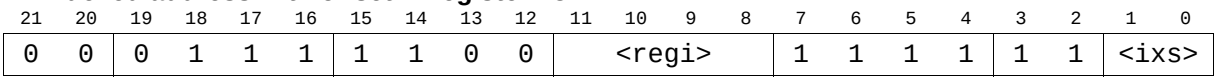
indexed address with offset and post-modification:



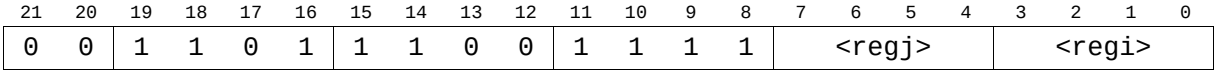
indexed address with offset and pre-modification:



indexed address with offset in register `r3`:



register to register operation:



 © 2001 Xemics, v4.5

DEC - Decrementation without Carry

		Operation
Syntax:	`DEC REG, <eaddr>`	`REG := DM[<eaddr>] - 1`
	`DEC REG`	`REG := REG - 1`
	`DEC REGi, REGj`	`REGi := REGj - 1`

Description: Decrements the source operand and stores the result in the destination register.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if source operand = 0x80, cleared otherwise.
C = 0 if underflow. It is set to 1 otherwise.
Z = 1 if the result of the decrementation is zero,
Z = 0 otherwise.

Accu. modified: Yes. Contains the result of the decrementation.

Examples:

Before	Instruction	After	V	C	Z
`r0 = 0xAA`	`DEC r0`	`r0 = a = 0xA9`	0	1	0
`r0 = 0x55`	`DEC r0`	`r0 = a = 0x54`	0	1	0
`r0 = 0x00`	`DEC r0`	`r0 = a = 0xFF`	0	0	0
`r0 = 0x80`	`DEC r0`	`r0 = a = 0x7F`	1	1	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	0	1	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	1	0	1	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	0	1	1	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	0	1	1	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	1	1	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	0	1	1	1	1	1	1	<regj>				<regi>			

DECC - Decrementation with Carry

		Operation
Syntax:	`DECC REG, <eaddr>`	`REG := DM[<eaddr> + C - 1`
	`DECC REG`	`REG := REG + C - 1`
	`DECC REGi, REGj`	`REGi := REGj + C - 1`

Description: Decrements the source operand only if the Carry flag is set to 0 and stores the result in the destination register.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if source operand = 0x80, cleared otherwise.
C = 0 if underflow. It is set to 1 otherwise.
Z = 1 if the result of the decrementation is zero,
Z = 0 otherwise.

Accu. modified: Yes. Contains the result of the decrementation.

Examples:

Before	Instruction	After	V	C	Z
`r0 = 0x00, C = 0`	`DECC r0`	`r0 = a = 0xFF`	0	0	0
`r0 = 0x00, C = 1`	`DECC r0`	`r0 = a = 0x00`	0	1	1
`r0 = 0x58, C = 0`	`DECC r0`	`r0 = a = 0x57`	0	1	0
`r0 = 0x80, C = 0`	`DECC r0`	`r0 = a = 0x7F`	1	1	0
`r0 = 0xAB, C = 0`	`DECC r0`	`r0 = a = 0xAA`	0	1	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	1	1	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	1	1	1	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	1	1	1	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	1	1	1	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	1	1	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	1	1	1	1	1	1	1	<regj>				<regi>			

© 2001 Xemics, v4.5

FREQ - Frequency Division Selection

Syntax: `FREQ divn:4`

Operation

`FreqOut` `:= divn`

Description: Changes the value of the **`FreqOut`** signal to the specified `divn` value. This signal is used to change the frequency of the processor by dividing the current frequency by the specified ratio (see "6.4. Frequency Division").
`divn` can be one of the following values:

divn:4	Assembler mnemonic	Division ratio	Comment
0000	nodiv	1	Restores original frequency
1000	div2	2	Clk := Clk/2
1100	div4	4	Clk := Clk/4
1110	div8	8	Clk := Clk/8
1111	div16	16	Clk := Clk/16

other values are reserved.

Flags modified:

V	C	Z
no	no	no

Accu. modified: No.

Instruction format:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	divn:4			

HALT - Halt Mode Selection

Syntax: HALT

Description: Halts the processor.
The processor will only halt if no events and no enabled interrupts are active.
See "6.5. Halt Mode".

Flags modified:

V	C	Z
no	no	no

Accu. modified: No.

Instruction format:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1

 © 2001 Xemics, v4.5

INC - Incrementation without Carry

		Operation
Syntax:	`INC REG, <eaddr>`	`REG := DM[<eaddr>] + 1`
	`INC REG`	`REG := REG + 1`
	`INC REGi, REGj`	`REGi := REGj + 1`

Description: Increments the source operand and stores the result in the destination register.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if source operand = 0x7F, cleared otherwise.
C = 1 if overflow. It is set to 0 otherwise.
Z = 1 if the result of the incrementation is zero,
Z = 0 otherwise.

Accu. modified: Yes. Contains the result of the incrementation.

Examples:

Before	Instruction	After	V	C	Z
`r0 = 0xFF`	`INC r0`	`r0 = a = 0x00`	0	1	1
`r0 = 0x55`	`INC r0`	`r0 = a = 0x56`	0	0	0
`r0 = 0xAA`	`INC r0`	`r0 = a = 0xAB`	0	0	0
`r0 = 0x7F`	`INC r0`	`r0 = a = 0x80`	1	0	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	0	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	0	0	0	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	0	0	1	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	0	0	1	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	0	1	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	0	0	1	1	1	1	1	<regj>				<regi>			

INCC - Incrementation with Carry

		Operation
Syntax:	`INCC REG, <eaddr>`	`REG := DM[<eaddr> + C`
	`INCC REG`	`REG := REG + C`
	`INCC REGi, REGj`	`REGi := REGj + C`

Description: Adds the value of the Carry flag to the source operand and stores the result in the destination register.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if source operand = 0x7F, cleared otherwise.
C = 1 if overflow. It is set to 0 otherwise.
Z = 1 if the result of the incrementation is zero,
Z = 0 otherwise.

Accu. modified: Yes. Contains the result of the incrementation.

Examples:

Before	Instruction	After	V	C	Z
`r0 = 0x5C, C = 1`	`INCC r0`	`r0 = a = 0x5D`	0	0	0
`r0 = 0x7F, C = 1`	`INCC r0`	`r0 = a = 0x80`	1	0	0
`r0 = 0x9F, C = 1`	`INCC r0`	`r0 = a = 0xA0`	0	0	0
`r0 = 0xAA, C = 0`	`INCC r0`	`r0 = a = 0xAA`	0	0	0
`r0 = 0xCF, C = 1`	`INCC r0`	`r0 = a = 0xD0`	0	0	0
`r0 = 0xFF, C = 0`	`INCC r0`	`r0 = a = 0xFF`	0	0	0
`r0 = 0xFF, C = 1`	`INCC r0`	`r0 = a = 0x00`	0	1	1

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	1	0	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	0	1	0	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	1	0	1	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	1	0	1	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	1	0	1	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	1	1	1	1	<regj>				<regi>			

© 2001 Xemics, v4.5

Jcc - Jump upon Condition

Syntax:

```
Jcc jaddr:16
Jcc ip
```

Operation

`PC := jump address`, either
`jaddr:16` (immediate address) or
`ip` (indexed address)
if condition is true

Description: Conditionally jumps to the specified address if the condition defined by `cc` is true. The condition may be either the value of a status flag or the result of an unsigned or a signed comparison. The `cc` condition code may be any of the following assembler mnemonics:

cc:3	Assembler mnemonic	Condition	After CMP(A) d, s (d: dest., s: source)
000	JCC	jump if Carry clear	
	JGT		greater than
001	JVC	jump if Overflow clear	
	JGE		greater or equal
010	JZC	jump if Zero clear	
	JNE		not equal
011	JUMP	jump always	
100	JCS	jump if Carry set	
	JLE		less or equal
101	JVS	jump if Overflow set	
	JLT		less than
110	JZS	jump if Zero set	
	JEQ		equal
111	JEVT	jump on event	

The jump on event is executed if either bit 0 (EV0) or bit 1 (EV1) of the status register is set. This instruction can be used to test an external signal without having to access to a peripheral in the data memory.

Flags modified:

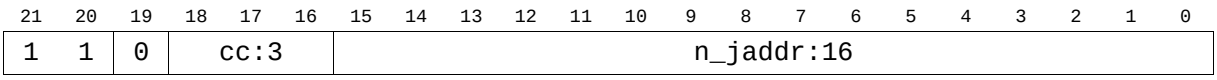
V	C	Z
no	no	no

Accu. modified: No.

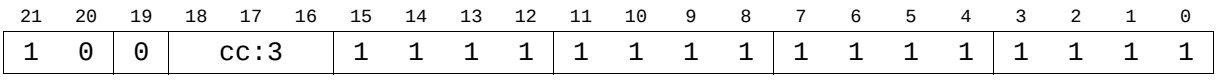
Examples: See "2.4. Instruction Examples".

Instruction format:

with immediate address:



with indexed address:



MOVE - Data Move

		Operation
Syntax:	`MOVE REG, #data:8`	`REG := data`
	`MOVE REGi, REGj`	`REGi := REGj`
	`MOVE REG, <eaddr>`	`REG := DM[<eaddr>]`
	`MOVE <eaddr>, REG`	`DM[<eaddr>] := REG`
	`MOVE addr:8, #data:8`	`DM[00000000,addr:8] := data`

Description: Moves an 8 bit value from the source to the destination.
Direct move from the data memory to the data memory is not possible. In the case an immediate 8 bit data is moved to a 8 bit address (last case above), the 8 most significant bits of the data memory address are forced to zero before the move.

Flags modified:

V	C	Z
no	no	yes/no

The Z flag is modified when the destination of the move is an internal register. `Z = 1` if the moved data is zero, `Z = 0` otherwise.
The Z flag is *not* modified if the destination of the move is the data memory.

Accu. modified: Only if the destination of the move is an internal register.
Contains the moved value.

Examples: See "2.4. Instruction Examples" .

Instruction format:

Destination: data memory

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	<ixs>		<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	<ixs>		<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	<ixs>		<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	<ixs>		<regi>				1	1	1	1	1	1	1	1

8 bit immediate data and immediate address:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	n_data:8								n_addr:8							

 © 2001 Xemics, v4.5

Destination: internal register

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	0	1	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	1	0	1	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	0	1	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	0	1	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	0	1	0	<regi>				1	1	1	1	1	1	<ixs>	

8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	0	1	0	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	0	1	0	1	1	1	1	<regj>				<regi>			

MUL - Unsigned Multiplication

Syntax:

```
MUL REGi, #data:8
MUL REGi, <eaddr>
MUL REGi, REGj, REGk
MUL REGi, REGj
```

Operation

```
REGi := MSB of, a := LSB of:
    REGi * data
    REGi * DM[<eaddr>]
    REGj * REGk
    REGj * REGi
```

Description: Performs an unsigned multiplication between two operands. The result is a 16 bit value whose 8 most significant bits are stored into the destination register and whose 8 least significant bits are stored into the accumulator.
This instruction is executed in a single cycle.
Note that there is no register forwarding for jumps and indexed memory accesses when the result register is **`ix`** or **`ip`**. See "6.9. Register Forwarding Exception".

Flags modified:

V	C	Z
u	u	u

The flags must be considered as undefined after the execution of the instruction.

Accu. modified: Yes. Contains the 8 least significant bits of the result.

Examples:

Before	Instruction	After

`r1 = 0x55, r2 = 0xAA`
`DM[A2] = 0x55,`
`a = 0xAA`
	`MUL r0, r1, r2`
	`MUL a, 0xA2`
		`r0 = 0x38, a = 0x72`
		`a = 0x72`

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1	1	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	1	1	1	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	1	1	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	1	1	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	1	1	0	<regi>				1	1	1	1	1	1	<ixs>	

8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	1	1	0	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	1	1	0	<regk>				<regj>				<regi>			

 © 2001 Xemics, v4.5

MULA - Signed Multiplication

Syntax:

```
MULA REGi, #data:8
MULA REGi, <eaddr>
MULA REGi, REGj, REGk
MULA REGi, REGj
```

Operation

```
REGi := MSB of, a := LSB of:
    REGi * data
    REGi * DM[<eaddr>]
    REGj * REGk
    REGj * REGi
```

Description: Performs a signed multiplication between two operands. The result is a 16 bit value whose 8 most significant bits are stored into the destination register and whose 8 least significant bits are stored into the accumulator.
This instruction is executed in a single cycle.
Note that there is no register forwarding for jumps and indexed memory accesses when the result register is **`ix`** or **`ip`**. See "6.9. Register Forwarding Exception".

Flags modified:

V	C	Z
u	u	u

The flags must be considered as undefined after the execution of the instruction.

Accu. modified: Yes. Contains the 8 least significant bits of the result.

Examples:

Before	Instruction	After

`r1 = 0x55, r2 = 0xAA`
`DM[A2] = 0x55,`
`a = 0xAA`
	`MULA r0, r1, r2`
	`MULA a, 0xA2`
		`r0 = 0xE3, a = 0x72`
		`a = 0x72`

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	1	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	0	1	1	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	1	1	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	1	1	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	1	0	<regi>				1	1	1	1	1	1	<ixs>	

8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	1	1	0	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	1	1	0	<regk>				<regj>				<regi>			

NOP - No Operation

Syntax: NOP

Description: No operation.

Flags modified:

V	C	Z
no	no	no

Accu. modified: No.

Instruction format:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Operation

No operation

OR - Logical OR

		Operation
Syntax:	`OR REG, #data:8`	`REG := REG OR data`
	`OR REG, <eaddr>`	`REG := REG OR DM[<eaddr>]`
	`OR REGi, REGj, REGk`	`REGi := REGj OR REGk`
	`OR REGi, REGj`	`REGi := REGj OR REGi`

Description: Performs a logical OR between the two operands.
The first operand is the destination register into which the result is stored.

Flags modified:

V	C	Z
no	no	yes

Z = 1 if the result is zero, 0 otherwise.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	**Instruction**	**After**	**Z**
`r1 = 0x0F, a = 0xF0`	`OR a, r1`	`a = 0xFF`	`0`
`i0h = 0x99`	`OR i0h, #0x33`	`a = i0h = 0xBB`	`0`

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	0	1	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	1	0	1	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	0	1	1	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	0	1	1	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	0	1	1	<regi>				1	1	1	1	1	1	<ixs>	

with 8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	0	1	1	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	0	1	1	<regk>				<regj>				<regi>			

PMD - Program Memory Dump

Syntax: `PMD #s`

Operation

if s =	0	1
ROM mode is:	off	on

Description: Sets the ROM mode on or off.
This instruction has no effect when the processor is in Test mode. See "5.3. ROM Mode".

Flags modified:

V	C	Z
no	no	no

Accu. modified: No.

Instruction format:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	1	1	0	1	1	1	#s	1	1	1	1	1	1	1	1

 © 2001 Xemics, v4.5

POP - Pop `ip` Index from Hardware Stack

Syntax: POP

Operation

iph := MSB of ST1
ipl := LSB of ST1
PC := PC + 1
STi := ST(i+1)

Description: Retrieves the 16 bit value located at register ST1 of the stack and stores it into the **ip** register.
The stack is popped one level after the execution of the instruction. This instruction allows the access to the hardware stack without modifying the program flow (i.e. no call or jump is necessary). See "6.2. Hardware Stack Depth" and "6.3. Task Switching".

Flags modified:

V	C	Z
no	no	no

Accu. modified: No.

Example:

Before	Instruction	After
ip = 0x2222	POP	ip = 0x0A54
PC = 0x43E7		PC = 0x43E8
ST1 = 0x0A54		ST1 = 0x3333
ST2 = 0x3333		ST2 = 0x5555
ST3 = 0x5555		ST3 = 0x7777
ST4 = 0x7777		

Instruction format:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

PUSH - Push `ip` Index onto Hardware Stack

Syntax: PUSH

Operation

```
PC  := PC + 1
ST1 := ip
ST(i+1) := STi, i > 1
```

Description: Stores the value of the ip register on register ST1 of the stack.
The stack is pushed one level after the execution of the instruction. See "6.2. Hardware Stack Depth".

Flags modified:

V	C	Z
no	no	no

Accu. modified: No.

Example:

Before	Instruction	After
ip = 0x0A54	PUSH	ip = 0x0A54
PC = 0x43E7		PC = 0x43E8
ST1 = 0x3333		ST1 = 0x0A54
ST2 = 0x5555		ST2 = 0x3333
ST3 = 0x7777		ST3 = 0x5555
		ST4 = 0x7777

Instruction format:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

 © 2001 Xemics, v4.5

RET - Return from Subroutine

Syntax: RET

Operation

PC := top of hardware stack
Hardware stack popped one level

Description: Returns from the execution of a subroutine.
The returned address is popped from the top of the hardware stack and stored into the program counter.

Flags modified:

V	C	Z
no	no	no

Accu. modified: No.

Example:

Before	Instruction	After
	RET	PC = 0x43E8
ST1 = 0x43E8		ST1 = 0x5555
ST2 = 0x5555		ST2 = 0x7777
ST3 = 0x7777		ST3 = 0x9999
ST4 = 0x9999		

Instruction format:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

RETI - Return from Interruption

Syntax: `RETI`

Operation

PC := top of hardware stack
Hardware stack popped one level
`GIE := 1`

Description: Returns from the execution of an interrupt subroutine.
The returned address is popped from the top of the hardware stack and stored into the Program Counter. In addition, the `GIE` flag is set. See "4.2. Interrupts and Events".

Flags modified:

V	C	Z
no	no	no

Accu. modified: No.

Example:

Before	Instruction	After
	`RETI`	`PC = 0x43E8`
`ST1 = 0x43E8`		`ST1 = 0x5555`
`ST2 = 0x5555`		`ST2 = 0x7777`
`ST3 = 0x7777`		`ST3 = 0x9999`
`ST4 = 0x9999`		

Instruction format:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

 © 2001 Xemics, v4.5

SFLAG - Save Flags

Syntax: `SFLAG`

Operation

```
a[7] := C flag
a[6] := C flag XOR V flag
a[5] := stack full flag
a[4] := stack empty flag
```

Description: Saves various flags in the accumulator **a**, namely: the Carry flag, the V flag, the hardware stack full flag and the hardware stack empty flag. All other bits in the accumulator have an undefined value.
This instruction is is typically used for checking for free space before pushing data onto the hardware stack (see "6.2. Hardware Stack Depth") or for context switching (see "6.3. Task Switching").
The Z flag can be modified because SFLAG changes the value of **a**.

Flags modified:

V	C	Z
no	no	yes

Accu. modified: Yes.

Examples:

Before	Instruction	After
`a = 0b01010101, C = 1, V = 0`	`SFLAG`	`a = 0b11001010`
`a = 0b00110011, C = 1, V = 1`	`SFLAG`	`a = 0b10001001`
`a = 0b00111000, C = 0, V = 0`	`SFLAG`	`a = 0b00001100`
`a = 0b00000011, C = 0, V = 1`	`SFLAG`	`a = 0b01000001`

In the above examples it is assumed that the hardware stack is neither full nor empty.

Instruction format:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

SHL - Logical Shift Left without Carry

		Operation
Syntax:	`SHL REG, <eaddr>`	`REG[7:1] := (DM[<eaddr>] << 1)[7:1]` `REG[0] := 0`
	`SHL REGi, REGj`	`REGi[7:1] := (REGj << 1)[7:1]` `REGi[0] := 0`
	`SHL REG`	`REG[7:1] := (REG << 1)[7:1]` `REG[0] := 0`

Description: Performs a logical left shift of one position of the source operand. The result is stored in the destination register. The least significant bit of the result is cleared. The Carry flag takes the value of the most significant bit of the operand.

Flags modified:

V	C	Z
yes	yes	yes

`C = op1[7]`
`Z = 1` if the result is zero, `0` otherwise.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	Instruction	After	V	C	Z
`r1 = 0xC0`	`SHL r1`	`r1 = a = 0x80`	0	1	0
`r1 = 0xBF`	`SHL r1`	`r1 = a = 0x7E`	1	1	0
`r1 = 0x3F`	`SHL r1`	`r1 = a = 0x7E`	0	0	0
`r1 = 0x40`	`SHL r1`	`r1 = a = 0x80`	1	0	0
`r1 = 0x80`	`SHL r1`	`r1 = a = 0x00`	1	1	1

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	0	1	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	1	0	1	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	0	1	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	0	1	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	1	0	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	0	1	0	<regk>				<regj>				<regi>			

© 2001 Xemics, v4.5

SHLC - Logical Shift Left with Carry

Syntax:

	Operation
`SHLC REG, <eaddr>`	`REG[7:1] := (DM[<eaddr>] << 1)[7:1]` `REG[0] := C`
`SHLC REGi, REGj`	`REGi[7:1] := (REGj << 1)[7:1]` `REGi[0] := C`
`SHLC REG`	`REG[7:1] := (REG << 1)7:1]` `REG[0] := C`

Description: Performs a logical left shift of one position of the source operand. The result is stored in the destination register. The least significant bit of the result is filled with the value of the Carry flag before the operation. The Carry flag takes the value of the most significant bit of the operand.

Flags modified:

V	C	Z
yes	yes	yes

`C = op1[7]`
`Z = 1` if the result is zero, `0` otherwise.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	**Instruction**	**After**	**V**	**C**	**Z**
`r1 = 0x00, C = 0`	`SHLC r1`	`r1 = a = 0x00`	0	0	1
`r1 = 0x55, C = 0`	`SHLC r1`	`r1 = a = 0xAA`	1	0	0
`r1 = 0xAA, C = 1`	`SHLC r1`	`r1 = a = 0x55`	1	1	0
`r1 = 0xFF, C = 1`	`SHLC r1`	`r1 = a = 0xFF`	0	1	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	1	1	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	1	1	1	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	1	1	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	1	1	1	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	1	0	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	1	1	0	<regk>				<regj>				<regi>			

SHR - Logical Shift Right without Carry

Syntax:

	Operation
`SHR REG, <eaddr>`	`REG[6:0] := (DM[<eaddr>] >> 1)[6:0]` `REG[7] := 0`
`SHR REGi, REGj`	`REGi[6:0] := (REGj >> 1)[6:0]` `REGi[7] := 0`
`SHR REG`	`REG[6:0] := (REG >> 1)[6:0]` `REG[7] := 0`

Description: Performs a logical right shift of one position of the source operand. The result is stored in the destination register. The most significant bit of the result is cleared. The Carry flag takes the value of the least significant bit of the operand.

Flags modified:

V	C	Z
yes	yes	yes

`C = op1[0]`
`Z = 1` if the result is zero, `0` otherwise.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	Instruction	After	V	C	Z
`r1 = 0xAA`	`SHR r1`	`r1 = a = 0x55`	0	0	0
`r1 = 0x55`	`SHR r1`	`r1 = a = 0x2A`	0	1	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	1	1	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	0	1	1	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	1	1	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	1	1	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	1	1	0	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	1	0	<regk>				<regj>				<regi>			

© 2001 Xemics, v4.5

SHRA - Arithmetic Shift Right

Syntax:

	Operation
`SHRA REG, <eaddr>`	`REG[6:0] := (DM[<eaddr>] >> 1)[6:0]` `REG[7] := op1[7]`
`SHRA REGi, REGj`	`REGi[6:0] := (REGj >> 1)[6:0]` `REGi[7] := op1[7]`
`SHRA REG`	`REG[6:0] := (REG >> 1)[6:0]` `REG[7] := op1[7]`

Description: Performs an arithmetic right shift of one position of the source operand. The result is stored in the destination register. The result is sign extended: the most significant bit of the result is filled with the most significant bit of the operand. The Carry flag takes the value of the least significant bit of the operand.

Flags modified:

V	C	Z
yes	yes	yes

`V = 0`
`C = op1[0]`
`Z = 1` if the result is zero, `0` otherwise.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	**Instruction**	**After**	**V**	**C**	**Z**
`r1 = 0xAA`	`SHRA r1`	`r1 = a = 0xD5`	`0`	`0`	`0`
`r1 = 0x55`	`SHRA r1`	`r1 = a = 0x2A`	`0`	`1`	`0`
`r1 = 0xFF`	`SHRA r1`	`r1 = a = 0xFF`	`0`	`1`	`0`
`r1 = 0x00`	`SHRA r1`	`r1 = a = 0x00`	`0`	`0`	`1`

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	0	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	0	0	0	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	0	0	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	0	0	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	0	0	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	0	0	0	<regk>				<regj>				<regi>			

SHRC - Logical Shift Right with Carry

Syntax:

	Operation
`SHRC REG, <eaddr>`	`REG[6:0] := (DM[<eaddr>] >> 1)[6:0]` `REG[7] := C`
`SHRC REGi, REGj`	`REGi[6:0] := (REGj >> 1)[6:0]` `REGi[7] := C`
`SHRC REG`	`REG[6:0] := (REG >> 1)[6:0]` `REG[7] := C`

Description: Performs a logical right shift of one position of the source operand. The result is stored in the destination register. The most significant bit of the result is filled with the Carry flag. The Carry flag takes the value of the least significant bit of the operand.

Flags modified:

V	C	Z
yes	yes	yes

`C = op1[0]`
`Z = 1` if the result is zero, `0` otherwise.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	**Instruction**	**After**	**V**	**C**	**Z**
`r1 = 0xAA, C = 0`	`SHRC r1`	`r1 = a = 0x55`	0	0	0
`r1 = 0x55, C = 1`	`SHRC r1`	`r1 = a = 0xAA`	0	1	0
`r1 = 0x00, C = 0`	`SHRC r1`	`r1 = a = 0x00`	0	0	1
`r1 = 0x00, C = 1`	`SHRC r1`	`r1 = a = 0x80`	0	0	0
`r1 = 0xFF, C = 1`	`SHRC r1`	`r1 = a = 0xFF`	0	1	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	1	0	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		1	0	1	0	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	1	0	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		1	0	1	0	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	1	0	0	<regi>				1	1	1	1	1	1	<ixs>	

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	0	<regk>				<regj>				<regi>			

© 2001 Xemics, v4.5

SUBD - Subtraction without Carry (op1 - op2)

		Operation
Syntax:	`SUBD REG, #data:8`	`REG := data - REG`
	`SUBD REG, <eaddr>`	`REG := DM[<eaddr>] - REG`
	`SUBD REGi, REGj, REGk`	`REGi := REGj - REGk`
	`SUBD REGi, REGj`	`REGi := REGj - REGi`

Description: Subtracts the operand 2 from the operand 1.
The first operand is the destination register into which the result is stored.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if a signed overflow occurred.
C = 1 if a carry is generated.
C = 0 if an unsigned overflow occured.
Z = 1 if the result is zero.
Otherwise the flags are cleared.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	**Instruction**	**After**	**V**	**C**	**Z**
`r0 = 0x56`	`SUBD r0, #0x12`	`r0 = a = 0xBC`	0	0	0
`r0 = 0x56`	`SUBD r0, #0x90`	`r0 = a = 0x3A`	1	1	0
`r0 = 0x12`	`SUBD r0, #0x56`	`r0 = a = 0x44`	0	1	0
`r0 = 0x90`	`SUBD r0, #0x56`	`r0 = a = 0xC6`	1	0	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	0	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	0	1	0	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	1	0	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	1	0	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	0	0	<regi>				1	1	1	1	1	1	<ixs>	

8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	1	0	0	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	1	0	0	<regk>				<regj>				<regi>			

SUBDC - Subtraction with Carry (op1 - op2)

Syntax:

	Operation
`SUBDC REG, #data:8`	`REG := data - REG - (1 - C)`
`SUBDC REG, <eaddr>`	`REG := DM[<eaddr>] - REG - (1 - C)`
`SUBDC REGi, REGj, REGk`	`REGi := REGj - REGk - (1 - C)`
`SUBDC REGi, REGj`	`REGi := REGj - REGi - (1 - C)`

Description: Subtracts the operand 2 and the invert of the Carry flag from the operand 1. The first operand is the destination register into which the result is stored.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if a signed overflow occurred.
C = 1 if a carry is generated.
C = 0 if an unsigned overflow occured.
Z = 1 if the result is zero.
Otherwise the flags are cleared.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	**Instruction**	**After**	**V**	**C**	**Z**
`r0 = 0x77, C = 0`	`SUBDC r0, #0x07`	`r0 = a = 0x8F`	0	0	0
`r0 = 0x77, C = 1`	`SUBDC r0, #0x07`	`r0 = a = 0x90`	0	0	0
`r0 = 0x07, C = 0`	`SUBDC r0, #0x77`	`r0 = a = 0x6F`	0	1	0
`r0 = 0x07, C = 1`	`SUBDC r0, #0x77`	`r0 = a = 0x70`	0	1	0
`r0 = 0xA5, C = 0`	`SUBDC r0, #0x6C`	`r0 = a = 0xC6`	1	0	0
`r0 = 0x5A, C = 0`	`SUBDC r0, #0xC6`	`r0 = a = 0x6B`	1	1	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	0	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	0	1	0	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	1	0	1	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	1	0	1	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	0	1	<regi>				1	1	1	1	1	1	<ixs>	

8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	1	0	1	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	1	0	1	<regk>				<regj>				<regi>			

© 2001 Xemics, v4.5

SUBS - Subtraction without Carry (op2 - op1)

		Operation
Syntax:	`SUBS REG, #data:8`	`REG := REG - data`
	`SUBS REG, <eaddr>`	`REG := REG - DM[<eaddr>]`
	`SUBS REGi, REGj, REGk`	`REGi := REGk - REGj`
	`SUBS REGi, REGj`	`REGi := REGi - REGj`

Description: Subtracts the operand 1 from the operand 2.
The first operand is the destination register into which the result is stored.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if a signed overflow occurred.
C = 1 if a carry is generated.
C = 0 if an unsigned overflow occured.
Z = 1 if the result is zero.
Otherwise the flags are cleared.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	Instruction	After	V	C	Z
`r0 = 0x56`	`SUBS r0, #0x12`	`r0 = a = 0x44`	0	1	0
`r0 = 0x56`	`SUBS r0, #0x90`	`r0 = a = 0xC6`	1	0	0
`r0 = 0x12`	`SUBS r0, #0x56`	`r0 = a = 0xBC`	0	0	0
`r0 = 0x90`	`SUBS r0, #0x56`	`r0 = a = 0x3A`	1	1	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	1	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	0	0	1	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	0	1	1	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	0	1	1	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	1	1	<regi>				1	1	1	1	1	1	<ixs>	

8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	1	1	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	0	1	1	<regk>				<regj>				<regi>			

SUBSC - Subtraction with Carry (op2 - op1)

Syntax:

	Operation
`SUBSC REG, #data:8`	`REG := REG - data - (1 - C)`
`SUBSC REG, <eaddr>`	`REG := REG - DM[<eaddr>] - (1 - C)`
`SUBSC REGi, REGj, REGk`	`REGi := REGk - REGj - (1 - C)`
`SUBSC REGi, REGj`	`REGi := REGi - REGj - (1 - C)`

Description: Subtracts the operand 1 and the invert of the Carry flag from the operand 2. The first operand is the destination register into which the result is stored.

Flags modified:

V	C	Z
yes	yes	yes

V = 1 if a signed overflow occurred.
C = 1 if a carry is generated.
C = 0 if an unsigned overflow occured.
Z = 1 if the result is zero.
Otherwise the flags are cleared.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	**Instruction**	**After**	**V**	**C**	**Z**
`r0 = 0x77, C = 0`	`SUBSC r0, #0x07`	`r0 = a = 0x6F`	0	1	0
`r0 = 0x77, C = 1`	`SUBSC r0, #0x07`	`r0 = a = 0x70`	0	1	0
`r0 = 0x07, C = 1`	`SUBSC r0, #0x77`	`r0 = a = 0x9F`	0	0	0
`r0 = 0x07, C = 0`	`SUBSC r0, #0x77`	`r0 = a = 0x8F`	0	0	0
`r0 = 0xC6, C = 1`	`SUBSC r0, #0x5A`	`r0 = a = 0x6C`	1	1	0
`r0 = 0x6C, C = 0`	`SUBSC r0, #0xA5`	`r0 = a = 0xC6`	1	0	0

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	1	1	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	0	1	1	1	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	1	1	1	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	0	1	1	1	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	1	1	<regi>				1	1	1	1	1	1	<ixs>	

8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	1	1	1	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	1	1	1	<regk>				<regj>				<regi>			

© 2001 Xemics, v4.5

TSTB - Test Bit

Syntax: `TSTB REG, #bit:3`

Operation

`REG AND bit_mask`

Description: Performs a logical AND between the register and a mask.
The mask is an immediate value that is created from the bit index value (an integer number between 0 and 7). The selected bit is set while the other bits are cleared. The register is not modified by the instruction. This instruction is used to test the value of a single bit in a register.

Flags modified:

V	C	Z
no	no	yes

`Z = 1` if the tested bit is 0
`Z = 0` if the tested bit is 1

Accu. modified: Yes. Contains the result of the AND operation.

Examples:

Before	Instruction	After	Z
`r0 = 0x16`	`TSTB r0, #2`	`a = 0x04`	`0`
`r0 = 0x16`	`TSTB r0, #3`	`a = 0x00`	`1`

Instruction format:

8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	1	1	1	<regi>				n_data:8							

XOR - Logical Exclusive OR

		Operation
Syntax:	`XOR REG, #data:8`	`REG := REG XOR data`
	`XOR REG, <eaddr>`	`REG := REG XOR DM[<eaddr>]`
	`XOR REGi, REGj, REGk`	`REGi := REGj XOR REGk`
	`XOR REGi, REGj`	`REGi := REGj XOR REGi`

Description: Performs a logical exclusive OR between the two operands.
The first operand is the destination register into which the result is stored.

Flags modified:

V	C	Z
no	no	yes

Z = 1 if the result is zero, 0 otherwise.

Accu. modified: Yes. Contains the result of the operation.

Examples:

Before	**Instruction**	**After**	**Z**
`r1 = 0x0F, a = 0xF0`	`XOR a, r1`	`a = 0xFF`	`0`
`i0h = 0x99`	`XOR i0h, #0x33`	`a = i0h = 0xAA`	`0`

Instruction format:

direct addressing:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	0	0	0	<regi>				n_addr:8							

indexed address with offset:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	<ixs>		0	1	0	0	0	<regi>				offset:8							

indexed address with offset and post-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	0	0	0	<regi>				0	offset:7						

indexed address with offset and pre-modification:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	<ixs>		0	1	0	0	0	<regi>				1	cpl2_offset:7						

indexed address with offset in register `r3`:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	0	0	0	<regi>				1	1	1	1	1	1	<ixs>	

with 8 bit immediate data:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	0	0	0	<regi>				n_data:8							

register to register operation:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	0	0	0	<regk>				<regj>				<regi>			

© 2001 Xemics, v4.5

2.2. Assembler Aliases

The CR816 assembler provides several mnemonics, called *aliases*, for some often used operations. Aliases do not have their own opcode, but use the opcode of other instructions. They are interpreted by the assembler which translates them into CR816DL's native instructions. Table 2.2 lists the available assembler aliases.

TABLE 2.2: CR816 assembler aliases.

Mnemonic	Description
CLRB	Clear register bit.
INVB	Invert register bit.
MSHL MSHR MSHRA	Multiple shift left. Multiple shift right. Multiple arithmetic shift right.
RETS RFLAG	Return from subroutine using a software stack. Restore C and V flags.
SETB	Set register bit.

These aliases are defined to ease the work of the programmer by using comprehensive mnemonics. For example the CLRB alias can be used to clear a specified bit of a register. The assembler will generate the opcode of an AND for this mnemonic.

2.2.1. CLRB - Clear Register Bit

Usage: `CLRB REG, #bit:3`

This alias is used to clear the specified bit (index number 0 to 7) of a register. It is equivalent to the native instruction:

```
AND REG, #data
```

where `data` is a 8 bit word with all zeroes except the selected bit which is set to one.

2.2.2. SETB - Set Register Bit

Usage: `SETB REG, #bit:3`

This alias is used to set the specified bit (index number 0 to 7) of a register. It is equivalent to the native instruction:

```
OR REG, #data
```

where `data` is an 8 bit word with all zeroes except the selected bit which is set to one.

2.2.3. INVB - Invert Register Bit

Usage: `INVB REG, #bit:3`

This alias is used to invert the specified bit (index number 0 to 7) of a register. It is equivalent to the native instruction:

```
XOR REG, #data
```

where `data` is an 8 bit word with all zeroes except the selected bit which is set to one.

2.2.4. MSHL - Multiple Shift Left

Usage: `MSHL REG, #shift:3`

This assembler instruction is used to shift left the specified register multiple times. The shift count is specified as a number between 0 and 7. The alias is equivalent to the native instruction:

```
MUL REG, #data
```

where `data` is a 8 bit word equal to 2^{shift}. The result of the multiple shift is stored in the *accumulator*.

> **Note** The flags must be considered as undefined after the execution of the alias. See " MUL - Unsigned Multiplication" on page 24.

As an example, here is the implementation of a swap between the 4-bit MSB and 4-bit LSB of an 8-bit register with the `MSHL` alias:

```
MSHL r0, #4
ADD  r0, a
```

2.2.5. MSHR - Multiple Shift Right

Usage: `MSHR REG, #shift:3`

This assembler instruction is used to shift right the specified register multiple times. The shift count is specified as a number between 0 and 7. The alias is equivalent to the native instruction:

```
MUL REG, #data
```

where `data` is an 8 bit word equal to $2^{8-shift}$. The result of the multiple shift is stored in the specified *destination register*.

> **Note** The flags must be considered as undefined after the execution of the alias. See " MUL - Unsigned Multiplication".

2.2.6. MSHRA - Multiple Arithmetic Shift Right

Usage: `MSHRA REG, #shift:3`

This assembler instruction is used to shift right the specified register multiple times. The shift count is specified as a number between 2 and 7 or possibly 0 (see the note below). The alias is equivalent to the native instruction:

```
MULA REG, #data
```

where `data` is an 8 bit word equal to $2^{8-shift}$. The result of the multiple shift is stored in the specified *destination register*.

> **Note** The flags must be considered as undefined after the execution of the alias. See " MULA - Signed Multiplication".

> **Note** A shift of 1 should be avoided as this would lead to the execution of the native instruction `MULA REG, #0x80`. As 0x80 is treated as a negative number the actual result would be the two's complement of the expected result. This case is not detected by the assembler.

2.2.7. RETS - Return From Subroutine

Usage: `RETS`

This assembler instruction is used to return from a subroutine when a software stack is used. It is equivalent to the native instruction:

```
JUMP ip
```

2.2.8. RFLAG - Restore Flags

Usage: `RFLAG REG` or `RFLAG <eaddr>`

This assembler instruction is used to restore the C and V flags, during context switching for example. It is equivalent to the native instruction:

```
SHL a, REG     or     SHL a, <eaddr>
```

The Carry flag takes the value of the bit 7 of the operand and the Overflow flag takes the value of the XOR between the bit 7 and the bit 6 of the operand. This instruction is often used together with the `SFLAG` instruction to store and restore context.

Note	The `RFLAG` instruction modifies the accumulator and the three flags C, V and Z.

Note	The `SFLAG` instruction saves the stack empty and the stack full flags of the hardware stack. These flags are only modified by stack operations (`CALL`, `RET`, `RETI`, `PUSH`, `POP`). They are *not* modified by the `RFLAG` instruction.

2.3. Instruction Set Quick Reference

TABLE 2.3: CR816 instruction set summary.

Instr.	Parameters	res	op1	op2	Operation	Modif.[a] C V Z a
JUMP	jaddr:16 ip				PC := jaddr	- - - -
Jcc					if cc is true then PC := jaddr or ip	
CALL					STn := STn-1 (n>1) ST1 := PC + 1 PC := jaddr or ip	
CALLS					ip := PC + 1 PC := jaddr or ip	
RET					PC := ST1, STn-1 := STn (n>1)	- - - -
RETS					PC := ip	
RETI					PC := ST1, STn-1 := STn (n>1) GIE := 1	
PUSH					PC := PC + 1, ST1 := ip STn+1 := STn (n>1)	- - - -
POP					ip := ST1, PC := PC + 1 STn := STn+1	
MOVE	reg, #data:8 regi, regj reg, <eaddr>	reg regi reg	data regj <eaddr>		res := op1	- - Z a
	<eaddr>, reg addr:8, #data:8	<eaddr> addr	reg data			- - - -
CMVD	reg, <eaddr> regi, regj	reg regi	<eaddr> regj		if C = 0 then res := op1	- - Z a
CMVS					if C = 1 then res := op1	
SHL	regi, regj reg reg, <eaddr>	regi reg reg	regj reg <eaddr>		res := op1 << 1, res[0] := 0 C := op1[7]	C V Z a
SHLC					res := op1 << 1, res[0] := C C := op1[7]	
SHR					res := op1 >> 1, res[7] := 0 C := op1[0]	
SHRC					res := op1 >> 1, res[7] := C C := op1[0]	
SHRA					res := op1 >> 1 res[7] := op1[7], C := op1[0]	
CPL1					res := NOT(op1)	- - Z a
CPL2					res := NOT(op1) + 1 C := 1 if op1 = 0	C V Z a
CPL2C					res := NOT(op1) + C C := 1 if op1 = 0	
INC					res := op1 + 1 C := 1 if overflow	
INCC					res := op1 + C C := 1 if overflow	
DEC					res := op1 - 1 C := 0 if underflow	
DECC					res := op1 - (1 - C) C := 0 if underflow	

TABLE 2.3: CR816 instruction set summary.

Instr.	Parameters	res	op1	op2	Operation	Modif.[a] C V Z a
AND	reg, #data:8 regi, regj, regk regi, regj reg, <eaddr>	reg regi regi reg	data regj regj <eaddr>	reg regk regi reg	res := op1 AND op2	- - Z a
OR					res := op1 OR op2	
XOR					res := op1 XOR op2	
ADD					res := op1 + op2, C := 1 if overflow	C V Z a
ADDC					res := op1 + op2 + C C := 1 if overflow	
SUBD					res := op1 - op2 C := 0 if underflow	
SUBDC					res := op1 - op2 - (1 - C) C := 0 if underflow	
SUBS					res := op2 - op1 C := 0 if underflow	
SUBSC					res := op2 - op1 - (1 - C) C := 0 if underflow	
MUL					res := (op1 * op2)[15:8] a := (op1 * op2)[7:0]	u u u a
MULA					res := (op1 * op2)[15:8] a := (op1 * op2)[7:0]	
MSHL	reg, #shift:3	reg			a := reg << shift reg := reg >> (8 - shift)	u u u a
MSHR					reg := reg >> shift a := reg << (8 - shift)	
MSHRA					reg := SHRA shift a := reg << (8 - shift)	
CMP	reg, #data:8 regi, regj reg, <eaddr>		data regj <eaddr>	reg regi reg	a := op1 - op2 C := 0 if op2 > op1 else C := 1 V := C AND NOT(Z)	C V Z a
CMPA					a := op1 - op2 C := 0 if op2 > op1 else C := 1 V := C AND NOT(Z)	
TSTB	reg, #bit:3				Z := NOT(reg[bit])	- - Z a
SETB					reg[bit] := 1	
CLRB					reg[bit] := 0	
INVB					reg[bit] := NOT reg[bit]	
SFLAG					a[7] := C, a[6] := C XOR V a[5] := stack full a[4] := stack empty	- - - a
RFLAG	reg <eaddr>		reg <eaddr>		SHL a, op1	C V Z a
FREQ	divn:4				clk := clk/n with n = 1, 2, 4, 8 or 16	- - - -
HALT					halts the processor	- - - -
NOP					no operation	
PMD	#s				if s=1 then ROM mode := on else ROM mode := off	- - - -

a. "-" means "not affected", "u" means "undefined".

TABLE 2.4: Instruction set decoding table.

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	NOP
1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	RET
1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	RETI
1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	POP
1	1	1	0	1	0	n_addr:16																CALLS
1	1	1	0	0	1	n_addr:16																CALL
1	1	0	cc:3			n_addr:16																Jcc
1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	PUSH
1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	CALLS
1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	CALL
1	0	0	cc:3			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	Jcc
0	1	1	ix:2			alu_op:5				reg:4				offset:8								1)
0	1	0	ix:2			alu_op:5				reg:4				(cpl2_)offset:8								2)
0	0	1	1	1	0	alu_op:4				reg:4				n_data:8								3)
0	0	1	1	0		alu_op:5				reg_op2:4				reg_op1:4				reg_res:4				4)
0	0	1	0	1	1	1	1	1	0	1	1	1	s	1	1	1	1	1	1	1	1	PMD
0	0	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	HALT
0	0	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	divn:4				FREQ
0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	SFLAG
0	0	0	1	1		alu_op:5				reg:4				1	1	1	1	1	1	ix:2		5)
0	0	0	1	0		alu_op:5				reg:4				n_addr:8								6)
0	0	0	0	1	1	1	0	ix:2		reg:4				1	1	1	1	1	1	1	1	7)
0	0	0	0	1	1	0	1	ix:2		reg:4				(cpl2_)offset:8								8)
0	0	0	0	1	0	1	1	ix:2		reg:4				offset:8								9)
0	0	0	0	0	1	1	0	1	1	reg:4				n_addr:8								10)
0	0	0	0	0	0	n_data:8								n_addr:8								11)

1) Indexed ALU operation with immediate offset.
2) Indexed ALU operation with pre- or post-modification of the index.
3) ALU operation with immediate data.
4) ALU operation between registers.
5) ALU operation with offset in register **r3**.
6) ALU operation with 8 bit immediate address.
7) MOVE to data memory with offset in register **r3**.
8) MOVE to data memory with pre- or post-modification of the index.
9) MOVE to data memory with immediate offset.
10) MOVE to data memory with 8 bit immediate address.
11) Immediate MOVE to data memory with 8 bit data and 8 bit address.

TABLE 2.5: Opcodes

reg:4	Code	Function
r0	1 1 1 0	
r1	1 1 0 1	
r2	1 1 0 0	
r3	1 0 1 1	DM offset
i0l	0 0 0 0	i0[7:0]
i0h	0 0 0 1	i0[15:8]
i1l	0 0 1 0	i1[7:0]
i1h	0 0 1 1	i1[15:8]
i2l	0 1 0 0	i2[7:0]
i2h	0 1 0 1	i2[15:8]
i3l	0 1 1 0	i3[7:0]
i3h	0 1 1 1	i3[15:8]
ipl	1 0 0 0	ip[7:0]
iph	1 0 0 1	ip[15:8]
stat	1 0 1 0	status
a	1 1 1 1	accu

cc:3	Test	Code
JUMP	-	0 1 1
JCS	C = 1	1 0 0
JCC	C = 0	0 0 0
JZS	Z = 1	1 1 0
JZC	Z = 0	0 1 0
JVS	V = 1	1 0 1
JVC	V = 0	0 0 1
JEV	event	1 1 1
After CMP(A) d, s :		
JEQ	d = s	1 1 0
JNE	d ≠ s	0 1 0
JGT	d > s	0 0 0
JGE	d ≥ s	0 0 1
JLT	d < s	1 0 1
JLE	d ≤ s	1 0 0

ix:2	Code
i0	0 0
i1	0 1
i2	1 0
i3	1 1

divn:4	Code
nodiv	0 0 0 0
div2	1 0 0 0
div4	1 1 0 0
div8	1 1 1 0
div16	1 1 1 1

alu_op:5	Code
MOVE	0 1 0 1 0
CMVD	1 0 0 1 0
CMVS	1 0 0 1 1
SHL	1 1 0 1 0
SHLC	1 1 1 1 0
SHR	1 0 1 1 0
SHRC	1 0 1 0 0
SHRA	1 0 0 0 0
CPL1	1 1 0 0 0
CPL2	1 1 0 0 1
CPL2C	1 1 1 0 0
AND	0 0 0 1 0
OR	0 1 0 1 1
XOR	0 1 0 0 0

alu_op:5	Code
INC	1 0 0 0 1
INCC	1 0 1 0 1
DEC	1 1 0 1 1
DECC	1 1 1 1 1
ADD	0 1 1 0 0
ADDC	0 1 1 0 1
SUBD	0 0 1 0 0
SUBDC	0 0 1 0 1
SUBS	0 0 0 1 1
SUBSC	0 0 1 1 1
CMP	0 0 0 0 1
CMPA	0 0 0 0 0
MUL	0 1 1 1 0
MULA	0 0 1 1 0

alu_op:4	Code
MOVE	1 0 1 0
AND	0 0 1 0
OR	1 0 1 1
XOR	1 0 0 0
TSTB	1 1 1 1
ADD	1 1 0 0
ADDC	1 1 0 1
SUBD	0 1 0 0
SUBDC	0 1 0 1
SUBS	0 0 1 1
SUBSC	0 1 1 1
CMP	0 0 0 1
CMPA	0 0 0 0
MUL	1 1 1 0
MULA	0 1 1 0

2.4. Instruction Examples

Instruction	Parameters before execution				C	Parameters after execution				C	V	Z	a
JUMP 0x0A54	PC = 0x43E7				C	PC = 0x0A54				-	-	-	-
JUMP ip	PC = 0x43E7	ip = 0x0A54	iph = 0x0A	ipl = 0x54	C	PC = 0x0A54				-	-	-	-
JCS(JLE) ip	PC = 0x43E7	ip = 0x3A54	iph = 0x3A	d ≤ s	1	PC = 0x3A54				-	-	-	-
JCS(JGT) ip	PC = 0x43E7	ip = 0x3A54	iph = 0x3A	d ≤ s	0	PC = 0x43E8				-	-	-	-
JCC(JGT) ip	PC = 0x43E7	ip = 0x3A54	iph = 0x3A	d > s	0	PC = 0x3A54				-	-	-	-
JCC(JLE) ip	PC = 0x43E7	ip = 0x3A54	iph = 0x3A	d > s	1	PC = 0x43E8				-	-	-	-
JZS(JEQ) 0x1FE4	PC = 0x43E7		Z = 1	d = s	C	PC = 0x1FE4				-	-	-	-
JVS(JNE) 0x1FE4	PC = 0x43E7		V = 0	d = s	C	PC = 0x43E8				-	-	-	-
JVC(JNE) 0x1FE4	PC = 0x43E7		V = 0	d ≠ s	C	PC = 0x1FE4				-	-	-	-
JZC(JEQ) 0x1FE4	PC = 0x43E7		Z = 1	d ≠ s	C	PC = 0x43E8				-	-	-	-
JLT 0xF2E5	PC = 0x43E7		C*nZ = 1	d < s	C	PC = 0xF2E5				-	-	-	-
JLT 0xF2E5	PC = 0x43E7		C*nZ = 0	d < s	C	PC = 0x43E8				-	-	-	-
JGE 0xF2E5	PC = 0x43E7		C*nZ = 0	d ≥ s	C	PC = 0xF2E5				-	-	-	-
JGE 0xF2E5	PC = 0x43E7		C*nZ = 1	d ≥ s	C	PC = 0x43E8				-	-	-	-
JEV 0x0001	PC = 0xFFFF	stat[0] = 1	OR/AND	stat[1] = 1	C	PC = 0x0001				-	-	-	-
JEV 0x0001	PC = 0xFFFF	stat[0] = 0	AND	stat[1] = 0	C	PC = 0				-	-	-	-
CALL 0x0A54	PC = 0x43E7	ST1 = 0x5555	ST2 = 0x7777		C	PC = 0x0A54	ST1 = 0x43E8	ST2 = 0x5555	ST3 = 0x7777	-	-	-	-
RET	ST1 = 0x43E8	ST2 = 0x5555	ST3 = 0x7777	ST4 = 0x9999	C	PC = 0x43E8	ST1 = 0x5555	ST2 = 0x7777	ST3 = 0x9999	-	-	-	-
RETI	ST1 = 0x43E8	ST2 = 0x5555	ST3 = 0x7777	stat[5] = 0	C	PC = 0x43E8	ST1 = 0x5555	ST2 = 0x7777	stat[5] = 1	-	-	-	-
CALLS 0x1FE4	PC = 0x43E7	ip = 0x0A54	iph = 0x0A	ipl = 0x54	C	PC = 0x1FE4	ip = 0x43E8	iph = 0x43	ipl = 0xE8	-	-	-	-
CALLS ip	PC = 0x43E7	ip = 0x0A54	iph = 0x0A	ipl = 0x54	C	PC = 0x0A54	ip = 0x43E8	iph = 0x43	ipl = 0xE8	-	-	-	-
RETS	PC = 0x43E7	ip = 0x0A54	iph = 0x0A	ipl = 0x54	C	PC = 0x0A54	ip = 0x0A54	iph = 0x0A	ipl = 0x54	-	-	-	-
PUSH	PC = 0x43E7	ip = 0x0A54	iph = 0x0A	ipl = 0x54	C	PC = 0x43E8	ip = 0x0A54	iph = 0x0A	ipl = 0x54	-	-	-	-
	ST1 = 0x3333	ST2 = 0x5555	ST3 = 0x7777			ST1 = 0x0A54	ST2 = 0x3333	ST3 = 0x5555	ST4 = 0x7777				
POP	PC = 0x43E7	ip = 0x2222	iph = 0x22	ipl = 0x22	C	PC = 0x43E8	ip = 0x0A54	iph = 0x0A	ipl = 0x54	-	-	-	-
	ST1 = 0x0A54	ST2 = 0x3333	ST3 = 0x5555	ST4 = 0x7777		ST1 = 0x3333	ST2 = 0x5555	ST3 = 0x7777					
FREQ div8		ClkμP = Ck			C		ClkμP = Ck/8			-	-	-	-
FREQ nodiv		ClkμP = Ck/8			C		ClkμP = Ck			-	-	-	-

Instruction	Parameters before execution				C	Parameters after execution				C	V	Z	a
SFLAG	a = 0b01010101			V = 0	1	a = 0b11101010				1	0	0	a
SFLAG	a = 0b00110011			V = 1	1	a = 0b10011001				1	1	0	a
SFLAG	a = 0b00111000			V = 0	0	a = 0b00011100				0	0	0	a
SFLAG	a = 0b00000011			V = 1	0	a = 0b01000001				0	1	0	a
RFLAG r0	r0 = 0b11000000				C	a = 0b10000000				1	0	0	a
RFLAG 0x27	DM[27] = 0b10111111				C	a = 0b01111110				1	1	0	a
RFLAG (i0)+	DM[i0] = 0b00111111				C	a = 0b01111110				0	0	0	a
RFLAG -(i1)	DM[i1-1] = 0b01000000				C	a = 0b10000000				0	1	0	a
MOVE stat, #13					C	stat = a = 13				-	-	0	a
MOVE r0, iph	iph = 0x0				C	r0 = a = 0x0				-	-	1	a
MOVE r1, 0x67	DM[67] = 0x32				C	r1 = a = 0x32				-	-	0	a
MOVE i0h, (i0)	DM[426] = 43	i0 = 0x0426	i0h = 0x04	i0l = 0x26	C	i0h = a = 43	i0 = 0x4326	i0h = 0x43	i0l = 0x26	-	-	0	a
MOVE i0l, (i0, 0x11)	DM[A2C7] = 55	i0 = 0xA2B6	i0h = 0xA2	i0l = 0xB6	C	i0l = a = 55	i0 = 0xA255	i0h = 0xA2	i0l = 0x55	-	-	0	a
MOVE r3, (i1, r3)	DM[A2C7] = 177	r3 = 0x11	i1h = 0xA2	i1l = 0xB6	C	r3 = a = 177	i1 = 0xA2B6	i1h = 0xA2	i1l = 0xB6	-	-	0	a
MOVE r2, (i1)+	DM[37] = 22	i1 = 0x0037	i1h = 0x00	i1l = 0x37	C	r2 = a = 22	i1 = 0x0038	i1h = 0x00	i1l = 0x38	-	-	0	a
MOVE i2l,(i2,0x24)+	DM[EE] = 33	i2 = 0x00EE	i1h = 0x00	i2l = 0xEE	C	i2l = a = 33	i2 = 0x0133	i2h = 0x01	i2l = 0x33	-	-	0	a
MOVE i3h, -(i3)	DM[03FF] = A8	i3 = 0x0400	i3h = 0x04	i3l = 0x00	C	i3h = a = 48	i3 = 0x48FF	i3h = 0x48	i3l = 0xFF	-	-	0	a
MOVE ipl,-(i3,0x33)	DM[0] = 88	i3 = 0x0033	i3h = 0x00	i3l = 0x33	C	ipl = a = 88	i3 = 0x0000	i3h = 0x00	i3l = 0x00	-	-	0	a
CMVD ipl,-(i3,0x33)	DM[0] = 88	i3 = 0x0033	i3h = 0x00	i3l = 0x33	0	ipl = a = 88	i3 = 0x0000	i3h = 0x00	i3l = 0x00	-	-	0	a
CMVD ipl,-(i3,0x33)	DM[0] = 88	i3 = 0x0033	i3h = 0x00	i3l = 0x33	1	a = 88	i3 = 0x0000	i3h = 0x00	i3l = 0x00	-	-	0	a
CMVS ipl,-(i3,0x33)	DM[0] = 88	i3 = 0x0033	i3h = 0x00	i3l = 0x33	1	ipl = a = 88	i3 = 0x0000	i3h = 0x00	i3l = 0x00	-	-	0	a
CMVS ipl,-(i3,0x33)	DM[0] = 88	i3 = 0x0033	i3h = 0x00	i3l = 0x33	0	a = 88	i3 = 0x0000	i3h = 0x00	i3l = 0x00	-	-	0	a
MOVE 0xF2, #133					C	DM[F2] = 133				-	-	-	-
MOVE 0x125, a	a = 7				C	DM[125] = 7				-	-	-	-
MOVE (i0), i0h	i0h = 0xE4	i0 = 0xE447	i0h = 0xE4	i0l = 0x47	C	DM[E447] = 0xE4	i0 = 0xE447	i0h = 0xE4	i0l = 0x47	-	-	-	-
MOVE (i0, 0xFF), ipl	ipl = 0x155	i0 = 0x3302	i0h = 0x33	i0l = 0x02	C	DM[3401] = 155	i0 = 0x3302	i0h = 0x33	i0l = 0x02	-	-	-	-
MOVE (i2, r3), r0	r0 = 0x47	r3 = 0xFF	i2h = 0x33	i2l = 0x02	C	DM[3401] = 47	i0 = 0x3302	i0h = 0x33	i0l = 0x02	-	-	-	-
MOVE (i3)+, r1	r1 = 0	i3 = 0x00FF	i3h = 0x00	i3l = 0xFF	C	DM[FF] = 0	i3 = 0x0100	i3h = 0x01	i3l = 0x00	-	-	-	-
MOVE (i2,0x14)+,stat	stat = 0x18	i2 = 0xFFEE	i2h = 0xFF	i2l = 0xEE	C	DM[FFEE] = 18	i2 = 0x0002	i2h = 0x00	i2l = 0x02	-	-	-	-
MOVE -(i3), r2	r2 = 111	i3 = 0x0100	i3h = 0x01	i3l = 0x00	C	DM[FF] = 111	i3 = 0x00FF	i3h = 0x00	i3l = 0xFF	-	-	-	-
MOVE -(i3, 0x46), i2l	i2l = 189	i3 = 0x0045	i3h = 0x00	i3l = 0x45	C	DM[FFFF] = 189	i3 = 0xFFFF	i3h = 0xFF	i3l = 0xFF	-	-	-	-

Instruction	Parameters before execution				C	Parameters after execution				C	V	Z	a
SHL r1	r1 = 0b11000000				C	r1 = a 0b10000000				1	0	0	a
SHL a, ipl	ipl = 0b01000011				C	A = 0b10000110				0	1	0	a
SHLC r0,i0l	i0l = 0b00111100				1	r0 = a = 0b01111001				0	0	0	a
SHLC r0	r0 = 0b10000011				0	r0 = a = 0b00000110				1	1	0	a
SHR a	a = 0b11000000				C	a = 0b01100000				0	0	0	a
SHR r1, iph	iph = 0b01000011				C	r1 = a = 0b00100001				1	0	0	a
SHRC r0	r0 = 0b00111100				1	r0 = a = 0b10011110				0	0	0	a
SHRC r1, a	a = 0b10000011				0	r1 = a = 0b01000001				1	0	0	a
SHRA a	a = 0b10000000				C	a = 0b11000000				0	0	0	a
SHRA r3, i3h	i3h = 0b01111111				C	r3 = a = 0b00111111				1	0	0	a
CPL1 a	a = 0b01010101				C	a = 0b10101010				-	-	0	a
CPL2 r0, r1	r1 = 0b01010101				C	r0 = a = 0b 10101011				0	0	0	a
CPL2 a	a = 0b10000000				C	a = 0b10000000				0	1	0	a
CPL2C r2	r2 = 0b01010101				0	r2 = a = 0b10101010				0	0	0	a
CPL2C a	a = 0b00000000				1	a = 0b00000000				1	0	1	a
MSHL i2l, #4	i2l = 0x34				C	i2l = 0x03	a = 0x40			u	u	u	a
MSHL r0, #2	r0 = 0xF3				C	r0 = 0x03	a = 0xCC			u	u	u	a
MSHR i2l, #4	i2l = 0x34				C	i2l = 0x03	a = 0x40			u	u	u	a
MSHR r0 , #2	r0 = 0xF3				C	r0 = 0x3C	a = 0xC0			u	u	u	a
MSHRA r3, #4	r3 = 0xF5				C	r3 = 0XFF	a = 0x50			u	u	u	a
MSHRA a, #4	a = 0xF5				C		a = 0x50			u	u	u	a
MUL r0, r1, r2	r1 = 0x55	r2 = 0xAA			C	r0 = 0x38	a = 0x72			u	u	u	a
MUL i2l, #0x10	i2l = 0x34				C	i2l = 0x03	a = 0x40			u	u	u	a
MUL a, 0xA2	DM[A2] = 0x55	a = 0xAA			C		a = 0x72			u	u	u	a
MULA r0, r1, r2	r1 = 0x55	r2 = 0xAA			C	r0 = 0xE3	a = 0x72			u	u	u	a
MULA r3, #0x10	r3 = 0xF5				C	r3 = 0XFF	a = 0x50			u	u	u	a
CMP r0, r1	r1 = 0xB4	r0 = 0xB6		d > s	C	a = 0xFE				0	0	0	a
CMP a, r0	r0 = 0x80	a = 0x7E		d < s	C	a = 0x02				1	1	0	a
CMPA r0, r1	r1 = 0xB4	r0 = 0xB6		d < s	C	a = 0xFE				1	1	0	a
CMPA a, r0	r0 = 0x80	a = 0x7E		d > s	C	a = 0x02				0	0	0	a
CMPA r3, r3				d ≤ s	C	a = 0				1	0	1	a

Instruction	Parameters before execution				C	Parameters after execution				C	V	Z	a
AND a, r1	r1 = 0x0F	a = 0xF0			C	a = 0				-	-	1	a
AND i0h, #0x33	i0h = 0x99				C	i0h = a = 0x11				-	-	0	a
OR a, r1	r1 = 0x0F	a = 0xF0			C	a = 0xFF				-	-	0	a
OR i0h, #0x33	i0h = 0x99				C	i0h = a = 0xBB				-	-	0	a
XOR a, r1	r1 = 0x0F	a = 0xF0			C	a = 0xFF				-	-	0	a
XOR i0h, #0x33	i0h = 0x99				C	i0h = a = 0xAA				-	-	0	a
TSTB r0, #2	r0 =0xFF				C	a = 0x04				-	-	0	a
TSTB a, #7	a = 0x7F				C	a = 0				-	-	1	a
SETB i0l, #0	i0l = 0				C	i0l = a = 0x01				-	-	0	a
SETB a, #6	a = 0x40				C	a = 0x40				-	-	0	a
CLRB r1, #0	i0l = 0x01				C	r1 = a = 0x00				-	-	1	a
CLRB a, #4	a = 0xEF				C	a = 0xEF				-	-	0	a
INVB iph, #5	iph = 0xFF				C	iph = a = 0xDF				-	-	0	a
INVB a, #3	a = 0x00				C	a = 0x08				-	-	0	a
INC a, (i0)	DM[A2B6] = FF	i0 = 0xA2B6	i0h = 0xA2	i0l = 0xB6	C	a = 0	i0 = 0xA2B6	i0h = 0xA2	i0l = 0xB6	1	0	1	a
INCC r0	r0 = 0xFF				0	r0 = a = 0XFF				0	0	0	a
INCC a	a = 0x7F				1	a = 0x80				0	1	0	a
DEC r1, (i0, 0xFF)	DM[3401] = 0	i0 = 0x3302	i0h = 0x33	i0l = 0x02	C	r1 = a = 0xFF				0	0	0	a
DECC a	a = 0				1	a = 0				1	0	0	a
DECC a	a = 0x80				0	a = 0x7F				1	1	0	a
ADD r0, 0xAF	DM[AF] = 0xF6	r0 = 0x0F			C	r0 = a = 0x05				1	0	0	a
ADD iph, r2	r2 = 0x42	iph = 0x43			C	iph = a = 0x85				0	1	0	a
ADDC i0l, i0l	i0l = 0x20				0	i0l = a = 0x40				0	0	0	a
ADDC a, r1	r1 = 0x80	a = 0x82			1	a = 0x03				1	1	0	a
SUBD r0, r1	r1 = 0xB4	r0 = 0xB6			C	r0 = a = 0xFE				0	0	0	a
SUBD a, r0	r0 = 0x80	a = 0x7E			C	a = 0x02				1	1	0	a
SUBDC i0h, 0x18	DM[18] = 0x28	i0h = 0x07			1	i0h = a = 0x21				1	0	0	a
SUBDC ipl, a	a = 0x42	ipl = 0xBC			0	ipl = a = 0x85				0	1	0	a
SUBS r3, a, r2	a = 0x7E	r2 = 0x80			C	r3 = a = 0x02				1	1	0	a
SUBSC a, ipl	ipl = 0xBC	a = 0x42			0	a = 0x85				0	1	0	a

Chapter 3
Memory and Peripheral Interfaces

This chapter describes the available memory and peripheral interfaces and their related timing diagrams. The CR816 has separate program memory interface and data memory interface. The data memory interface may also be used to access peripherals.

3.1. Program Memory Interface

The program memory interface is used by the processor to read instructions from the program memory. Figure 3.1 describes the program memory access. After the **CkOut** clock goes low, the program memory address **PMAddr** changes and the program memory select **nPMSel** becomes high. With XEMICS program memories this causes a precharge phase during which the program memory output bus **PMInst** is held high (a precharge at a low level would hold the **PMInst** bus low). Table 3.1 gives the minimum or maximum timing requirements to meet.

Note	The program memory is on the critical path of the processor since instructions are partially decoded during phase 2 (see "1.4. Pipeline").

FIGURE 3.1: Program memory access.

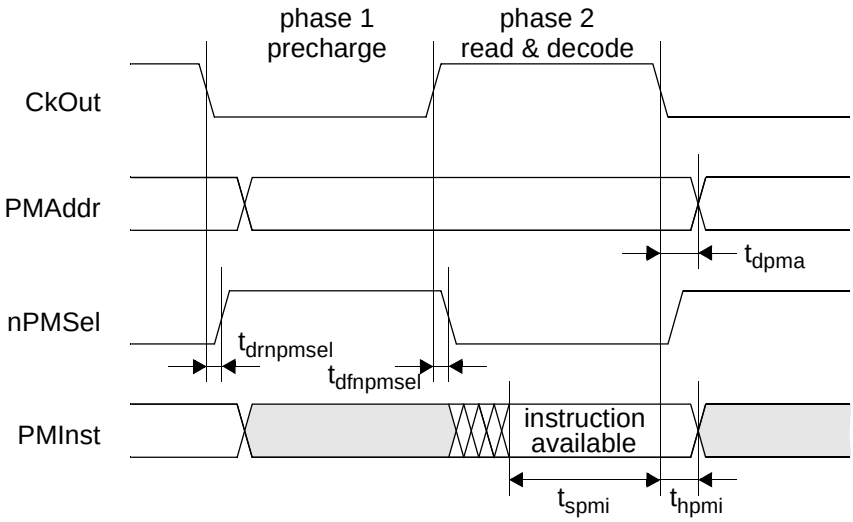


TABLE 3.1: Program memory access timing requirements.

Timing parameter	Description	Min	Max
t_{dpma}	**CkOut** fall to **PMAddr** delay		x
$t_{drnpmsel}$	**CkOut** fall to **nPMSel** rise delay		x
$t_{dfnpmsel}$	**CkOut** rise to **nPMSel** fall delay		x
t_{spmi}	**PMInst** setup time	x	
t_{hpmi}	**PMInst** hold time	x	

3.2. Data Memory (and Peripheral) Interface

A data memory access is performed in two steps:

1) The processor requests an access to the data memory by asserting the **DMReq**, **ReadnWrite** and **DMAddr** signals. The **DMReq** and **ReadnWrite** signals are asserted at the same time, while the **DMAddr** signal is asserted with some setup time.

2) The access to the data memory starts at the falling edge of the **CkOut** signal, provided the **ReqAccept** signal is asserted (the **ReqAccept** signal must be asserted before the end of phase 2). The **DMSel** signal is asserted during the data memory access.
A deasserted **ReqAccept** signal forces the processor to enter the Wait mode (see "6.8. Wait Mode").

For a write access, the **DataOut** bus is driven by the processor while the **DMSel** signal is active. The data must be stored in the data memory or the peripheral at the falling edge of **DMSel**.
For a read access, the **DataIn** bus must be driven by the data memory or the addressed peripheral. The data must be put on the **DataIn** bus before the **DMSel** signal falls as the processor latches the data on the **DataIn** bus at the falling edge of **DMSel**.

Figure 3.2 describes the data memory or peripheral access. Table 3.2 gives the minimum or maximum timing requirements to meet.

FIGURE 3.2: Data memory or peripheral access.

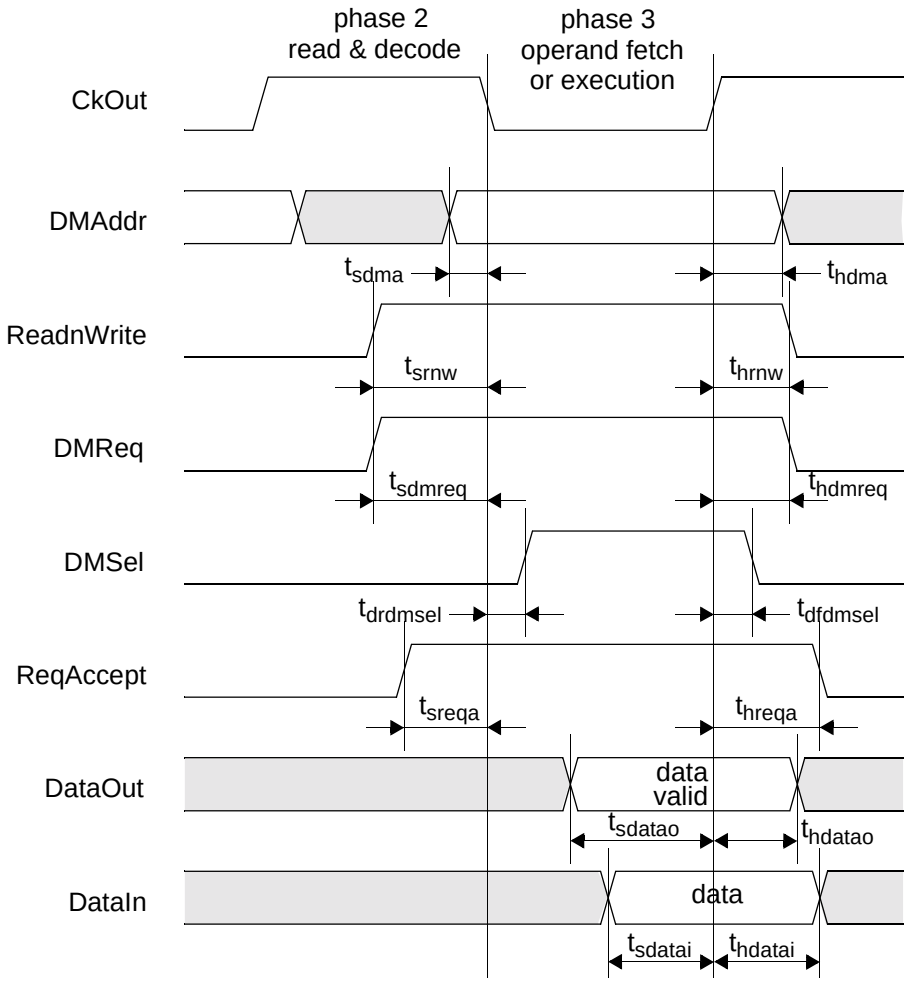


TABLE 3.2: Data memory access timing requirements.

Timing parameter	Description	Min	Max
t_{sdma}	**DMAddr** setup time	x	
t_{hdma}	**DMAddr** hold time	x	
t_{srnw}	**ReadnWrite** setup time	x	
t_{hrnw}	**ReadnWrite** hold time	x	
t_{sdmreq}	**DMReq** setup time	x	
t_{hdmreq}	**DMReq** hold time	x	
$t_{drdmsel}$	**CKout** to **DMSel** rise delay		x
$t_{dfdmsel}$	**CKout** to **DMSel** fall delay		x
t_{sreqa}	**ReqAccept** setup time	x	
t_{hreqa}	**ReqAccept** hold time	x	
t_{sdatao}	**DataOut** setup time	x	
t_{hdatao}	**DataOut** hold time	x	
t_{sdatai}	**DataIn** setup time	x	
t_{hdatai}	**DataIn** hold time	x	

Chapter 4
Exception Processing

This chapter describes the operations of the processor when an exception occurs. Both hardware and software exceptions may interrupt the normal flow of program execution. Hardware exceptions are related to the core interface signals **nReset**, **nInterrupt** and **nEvent**. Software exceptions are generated by writing into the status register. The processor state can be determined by reading the same register. Table 4.1 gives a summary of the exception capabilities.

TABLE 4.1: Exception capabilities summary.

Exception	Modification of execution flow	Restart from Halt mode	Generation by software
reset	yes	yes	no
interrupt	yes[a]	yes[a]	yes
event	no	yes	yes

a. Interrupts must be enabled.

4.1. Reset

The reset exception has the highest priority and is generated by asserting the **nReset** signal low. The exception has the following effects:

- The program counter (**PC**) is cleared.
- Pending interrupts and all interrupt enable flags (i.e. GIE, IE1, IE2) are cleared.
- The stack is initialized.
- The **FreqOut** signal is set to the nodiv (0x0) value.
- The Test mode and the ROM mode are disabled (see "Chapter 5 Test Capabilities").
- All registers are cleared.

The processor remains in reset mode as long as the **nReset** signal is asserted. The **nReset** signal must remain asserted during at least one clock cycle. It must be deasserted while **CkOut** is low with some setup time, otherwise a level 0 interrupt will occur. Figure 4.1 gives the reset timing diagram. Table 4.2 gives the reset timing requirement.

FIGURE 4.1: Reset timing diagram.

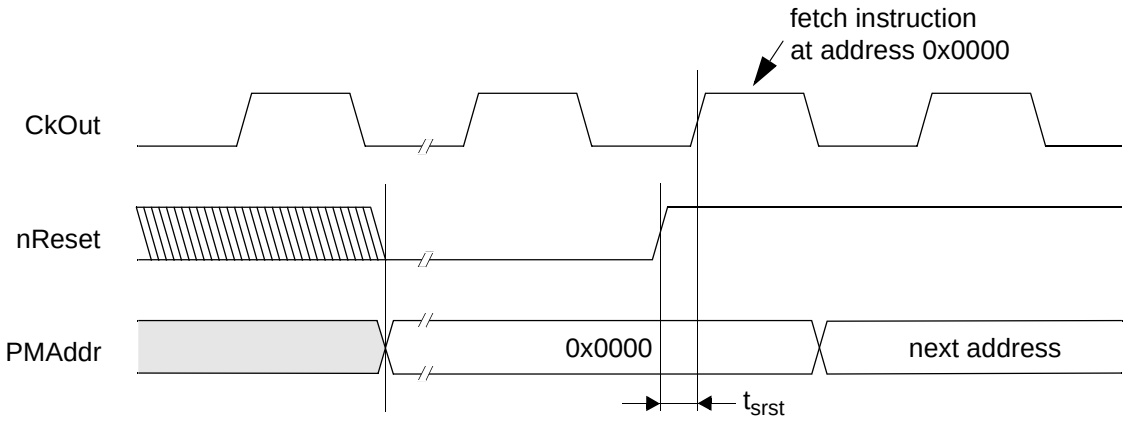


TABLE 4.2: Reset timing requirement.

Timing parameter	Description	Min	Max
t_{srst}	**nReset** setup time	x	

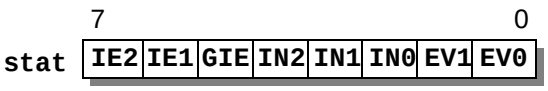
4.2. Interrupts and Events

Interrupts and events are boolean flags which can be modified either by hardware or by software. A negative pulse on the **nInterrupt** or **nEvent** pins will set the corresponding flag in the **stat** register to 1. The flags can be set or reset by writing into the **stat** register (Figure 4.2). This allows interrupts and events to be generated by software.

Interrupts force a CALL to a fixed interrupt vector (Table 4.4) and save the program counter (**PC**) onto the hardware stack. The processor will be restarted if it were in Halt mode. The corresponding flag in the **stat** register is set.

Events are generally used to restart the processor from the Halt mode without jumping to an interrupt subroutine. Events can also be combined with the JEV (jump on event) instruction (see "Jcc - Jump upon Condition" on page 2-21) with minimum hardware implications. The two event inputs have the same priority and the same effect on the processor behaviour, except the event bit position in the status register. Events have the same timing requirements as interrupts.

FIGURE 4.2: Status register layout.



For a complete description of the status register, see "1.5.2. Status Register".

4.2.1. Timing Requirements for Interrupts and Events

Interrupt and events must be driven while **CkOut** is low, as shown in Figure 4.3. As long as the interrupt enable bits are set, the interrupt call will occur at the next falling edge of the clock.

Interrupts and events may remain asserted (low) for several clock cycles. The corresponding bit in the status register will be set as long as the signals remain asserted. Thus it is necessary to clear the interrupt source (for example, an interrupt controller or peripheral register) before clearing the bit in the status register.

Figure 4.3 gives the timing diagrams for the interrupt and the event signals. Table 4.3 gives the timing requirements for the interrupt and the event signals.

FIGURE 4.3: Interrupts and events timing requirements.

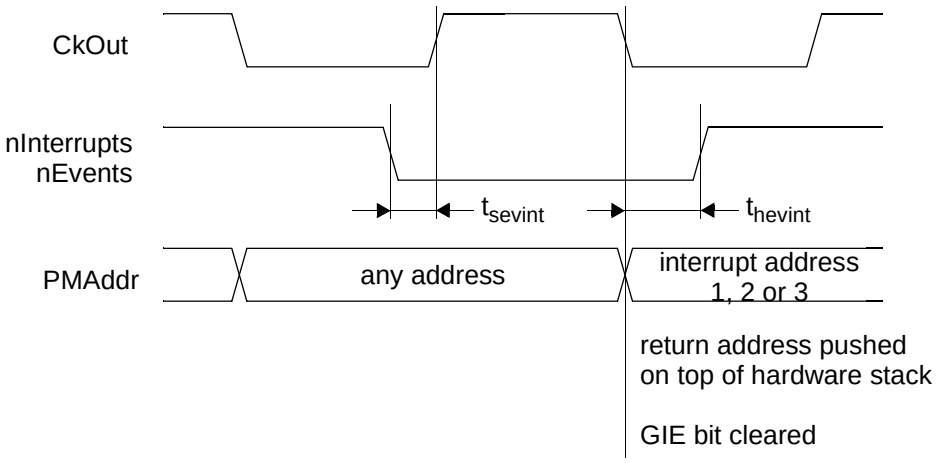


TABLE 4.3: Interrupts and events timing requirement.

Timing parameter	Description	Min	Max
t_{sevint}	**nInterrupt** and **nEvent** setup time	x	
t_{hevint}	**nInterrupt** and **nEvent** hold time	x	

Interrupts and events can be also generated by setting the corresponding bits in the status register with processor instructions (`MOVE`, `OR`, etc.). Interrupts or events generated by hardware (on the **nInterrupt** or **nEvent** signals) or generated by software act exactly the same way. See Paragraph 6.6 for pipeline exception and more information on software generated interrupts.

4.2.2. Interrupt Subroutine Calls

When an interrupt is received by the core that meets the requirements described in Paragraph 4.2.1 the corresponding bit of the status register is set, namely:

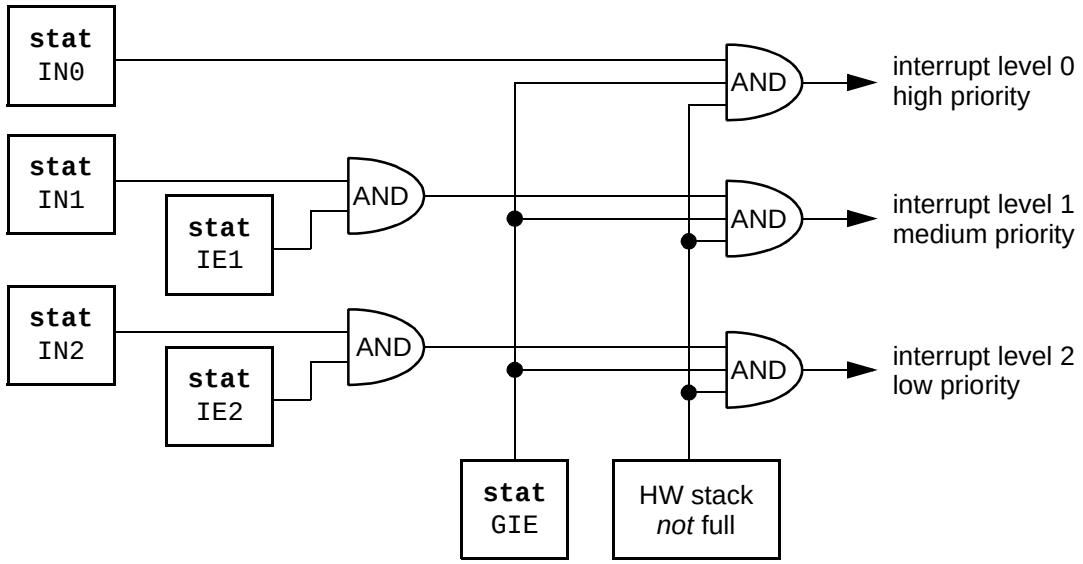
- Interrupt request generated on signal **ninterrupt(0)** sets the `IN0` bit (bit 2) of the status register.
- Interrupt request generated on signal **ninterrupt(1)** sets the `IN1` bit (bit 3) of the status register.
- Interrupt request generated on signal **ninterrupt(2)** sets the `IN2` bit (bit 4) of the status register.

Interrupt requests 1 and 2 can be enabled/disabled independently by the IE1 bit (bit 6) and IE2 (bit 7) of the status register. The GIE bit (bit 5) is a General Interrupt Enable bit that may be used to enable or disable the three interrupt requests. See Paragraph 1.5.2 for more details on the status register.

Note Interrupts requests are disabled when the hardware stack is full and independently of the value of the GIE bit. As soon as one level of stack is freed (with the RET, RETI or POP instruction), the pending interrupt with the highest priority is executed.

The enabling of interruptions is determined according to some priority scheme that is described in Figure 4.4.

FIGURE 4.4: Interrupt enabling priority scheme.



When the control unit of the processor receives one or several interrupts, the processor jumps to the address of the interrupt with the highest priority (see Figure 4.3). The addresses for the different interrupts are given in Table 4.4. When the processor jumps to the interrupt subroutine, the General Interrupt Enable bit (GIE) of the status register is automatically cleared to avoid recursive interrupt requests during interrupt processing. As the processor jumps to the interrupt subroutine, the return address is pushed onto the hardware stack.

Note Remember that interrupt requests remain active as long as the **nInterrupt** signal remains asserted.

The RETI instruction is used to return from the interrupt sub-routine. This instruction pops the address of the next instruction from the hardware stack and automatically sets the GIE bit. Because GIE is cleared by reset, no interrupt can occur until the user explicitly sets the GIE bit in the status register. Also, if the GIE bit is cleared prior to returning from the interrupt, it will be set again as the RETI instruction is executed.

Note The RET instruction should rather be used when interrupts must remain disabled after the interrupt processing.

If a high priority interrupt occurs while the processor is handling a low priority interrupt, the pending interrupt must wait until the GIE bit is enabled, usually by the RETI instruction. Experienced programmers may implement more complex interrupt processing by setting explicitly GIE during the interrupt subroutine (and return with RET instead of RETI).

The interrupt priority is used only to select which interrupt will be processed when multiple interrupt requests occur simultaneously.

4.2.3. Processor Vector Table

The address 1, 2 and 3 of the program memory are reserved for interrupt subroutine calls. Generally, the first four addresses of the program memory are reserved as the *processor vector table*. The address 0 of the program memory contains a jump to the start-up routine.

TABLE 4.4: Processor vector table.

Address	Accessed by	Description	Priority
0	**nReset**	reset, start-up address	maximal, above interrupts
1	IN1	interrupt level 1	medium
2	IN2	interrupt level 2	low
3	IN0	interrupt level 0	high

4.2.4. Halt Mode

Interrupts and events can be used to wake up the processor from the Halt mode. Timing requirements for this operation are the same as the timing requirements for normal interrupts (Figure 4.3 and Table 4.3). See "6.5. Halt Mode" for more details on the wake-up from Halt mode using exceptions.

4.2.5. Context Saving

Since an interrupt may occur any time during normal program execution, there is no way to know which processor registers are used by the user program. For this reason, all resources that will be modified in the interrupt service routine must be saved upon entering and restored when leaving the service routine. The flags (C, V, Z) and the accumulator (**a**) must always be saved, since most instructions will modify them. Other registers must only be saved when they are modified in the interrupt service routine. It is however good practice to save all of them anyway.

There is a particular order to follow when saving resources. Register **a** should be saved first, followed by the flags and then any other register. Depending on the data handling strategy, registers may be either saved on the software stack or in reserved memory locations. Code 4.1 shows how the context may be saved on the software stack.

CODE 4.1: Context saving on the stack.

```
        ; We will use the same software stack as the main application
        ; since we are not allowed to modify the stack pointer before
        ; the accumulator has been saved. i0 is our stack pointer.

int0_service:
        MOVE    -(i0), a        ; save accumulator
        SFLAG                   ; transfer flags into accumulator
        MOVE    -(i0), a        ; save flags
        MOVE    -(i0), r0
        MOVE    -(i0), r1
        ...                     ; we may save other registers as well
        ; service routine processing
        ...
        ; restore the context
        MOVE    r1, (i0)+
        MOVE    r0, (i0)+
        MOVE    a, (i0)+
        RFLAG   a
        MOVE    a, (i0)+
        RETI
```

Code 4.2 shows a code fragment where registers are saved in hard-coded memory locations. In order to minimize the number of reserved locations, one may want to use fixed location to save the accumulator, the flags and an index pointer, then use this index pointer to save the remaining registers in another memory location, using the full 64K addressing capabilities of the processor.

CODE 4.2: Context saving in reserved memory space.

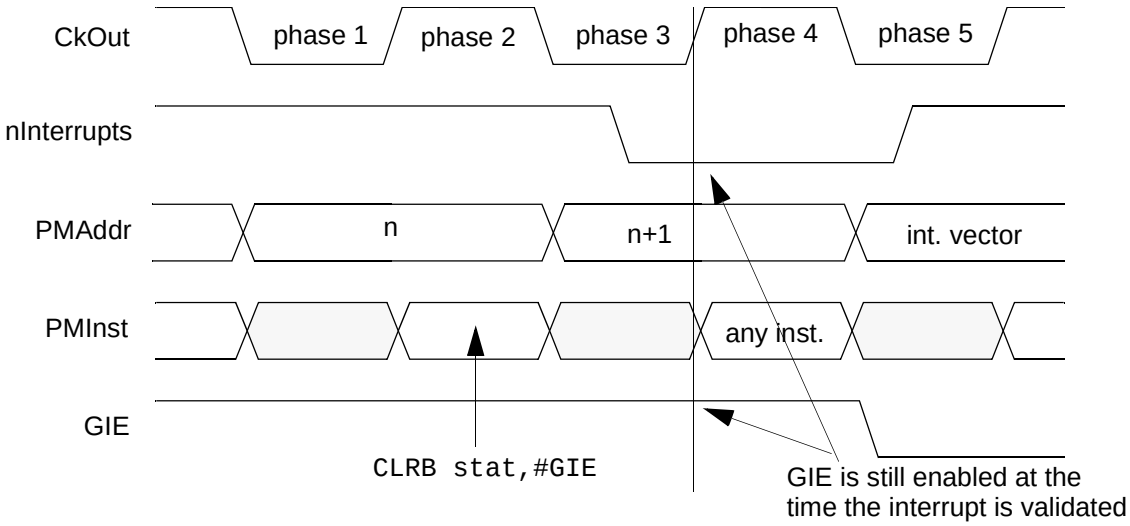
```
        ; Let's say we have reserved memory location 0x40-0x44 for
        ; context saving.

int0_service:
        MOVE    0x40, a         ; save accumulator
        SFLAG                   ; transfer flags into accumulator
        MOVE    0x41, a         ; save flags
        MOVE    0x42, r0
        MOVE    0x43, r1
        ...                     ; we may save other registers as well
        ; service routine processing
        ...
        ; restore the context
        MOVE    r1, 0x43
        MOVE    r0, 0x42
        MOVE    a, 0x41
        RFLAG   a
        MOVE    a, 0x40
        RETI
```

Note Instructions used for saving **a** and the flags should not modify them. In particular, transferring them to some other unused registers or to the hardware stack would not be suitable, since instructions to do that do modify the flags.

4.2.6. Disabling Interrupts

Interrupts can be enabled or disabled by writing in the EN2, EN1 and GIE bits of the status register. Because of the pipeline structure, a disabling instruction may be missed if an interrupt occurs at the exact same time as shown on Figure 4.5. Disable instruction is started in phase 1. Because this is an ALU operation, it needs three cycles to execute so the write-back operation takes place in phase 5. If an interrupt occurs during phase 3, it will be validated since the GIE bit will not be cleared until phase 5. The GIE bit will be cleared since the CLRB finishes execution before the interrupt routine is executed but it will be restored to 1 by the RETI instruction upon interrupt return.

FIGURE 4.5: Interrupt disabling timing.



Code 4.3 gives a code example which can be used to securely disable interrupts.

CODE 4.3: Disable interrupt example.

```
main:
        ...
        ; we want to disable interrupts from this point

disable:
        CLRB    stat, #GIE    ; clear general enable bit
                              ; if an interrupt occurs here, the following
                              ; code will execute with GIE=1, so let's
        TSTB    stat, #GIE    ; check it
        JZC     disable

secure:
                              ; now we are certain that GIE has been
                              ; cleared
        ...                   ; do some sensitive processing

restore:
        SETB    stat, #GIE    ; interrupts are restored
        ...
```

Note See "6.6. Pipeline Exception" for a similar instruction delay case when dealing with software exceptions.

Chapter 5
Test Capabilities

The CR816 architecture provides a flexible serial test interface for debugging and industrial testing of CoolRISC based products. This dedicated serial interface consists of the 4 signals, **TestIn**, **TestOut**, **TestShift** and **TestEn**, that must be used in combination with the **Ck** signal. This interface provides an access to the instruction register, to various flags and to registers.

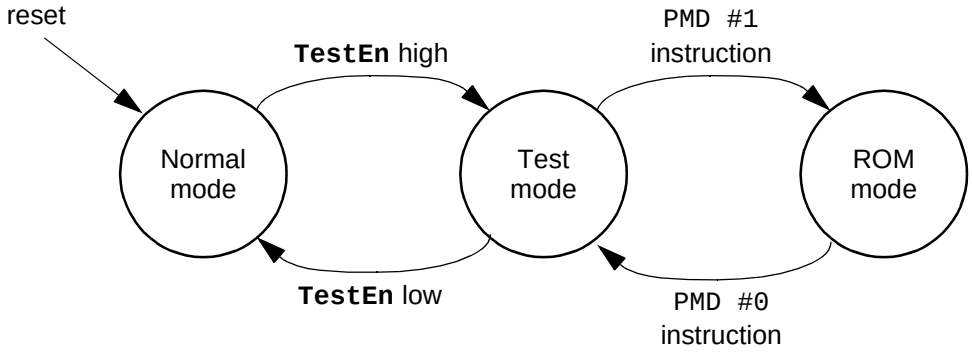
5.1. CR816 Operating Modes

The CR816 processor can be used in three different modes:

- **Normal mode**. This mode corresponds to the normal operating condition; the processor reads the instructions from the program memory and executes them at every clock cycle. In this mode, the serial test interface is not used. The **TestEn** signal must be tied to 1.
- **Test mode**. This mode uses the serial interface to shift instructions inside the processor. The program memory is bypassed and the shifted instructions are executed instead. This gives total freedom to examine processor resources and data memory.
- **ROM mode**. This mode uses the serial interface to shift the content of the program memory out of the chip. This mode may only be entered from the Test mode. In this mode, the instructions are not executed by the processor but are merely shifted out.

The Normal mode is the mode in which the processor enters after a reset. To go to the Test mode, the **TestEn** signal must be asserted. The processor can be switched between Test mode and Normal mode any time by cycling the **TestEn** signal. This is summarized in Figure 5.1.

FIGURE 5.1: CR816 operating modes.



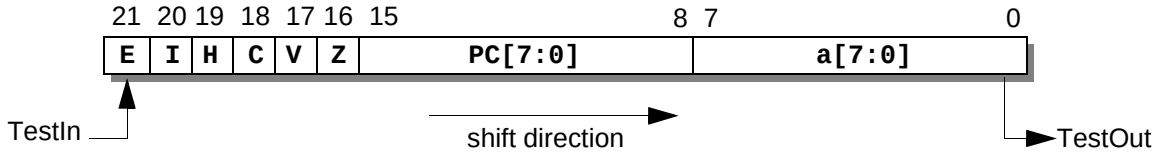
5.2. Test Mode

In Test mode, instructions are not read from the program memory but instead are serially shifted in through the **TestIn** pin into an internal shift register. The instructions are then executed in the same way as they would be in Normal mode. At the end of the execution, the shift register is loaded with internal flags and registers (see Figure 5.2). As the next instruction gets shifted in, the data appears on the **TestOut** test output. Test mode is entered by asserting the **TestEn** input high.

5.2.1. Test Shift Register

The test shift register is 22-bit wide and is clocked at the rising edge of **Ck**. The **TestShift** signal determines the shift register mode of operation, namely: load (**TestShift** low) or shift (**TestShift** high). The shift register layout is shown in Figure 5.2.

FIGURE 5.2: Test shift register layout.



All data captured in the shift register give a snapshot of the pipeline. Because several instructions are in the pipeline at one time, not all gathered data belong to the same instruction, as shown in Figure 5.3. Informations shifted out of **TestOut** should be interpreted as follows:

E: Pending events

I: Pending interrupts

H: Current **HaltState** signal

C: Carry flag from instruction n-2

V: Overflow flag from instruction n-2

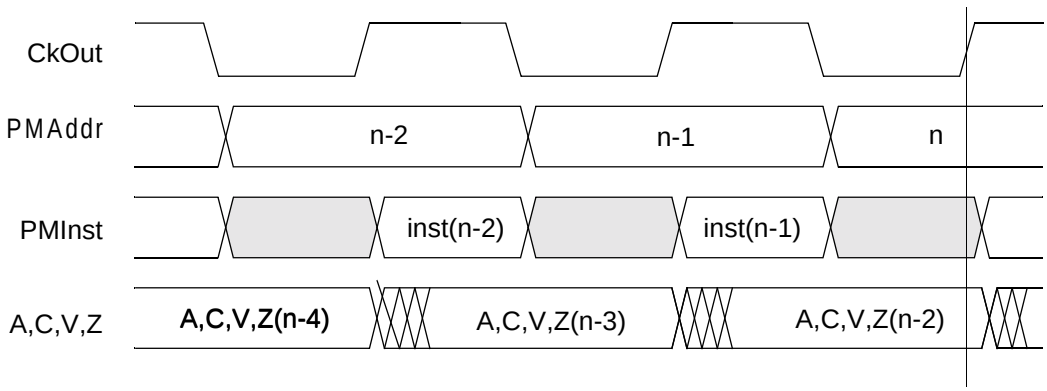
Z: Zero flag from instruction n-2

PC: Current lower byte of program counter

a: Accumulator from instruction n-2

Pending interrupts and events show the current status after masking (see Figure 4.4).

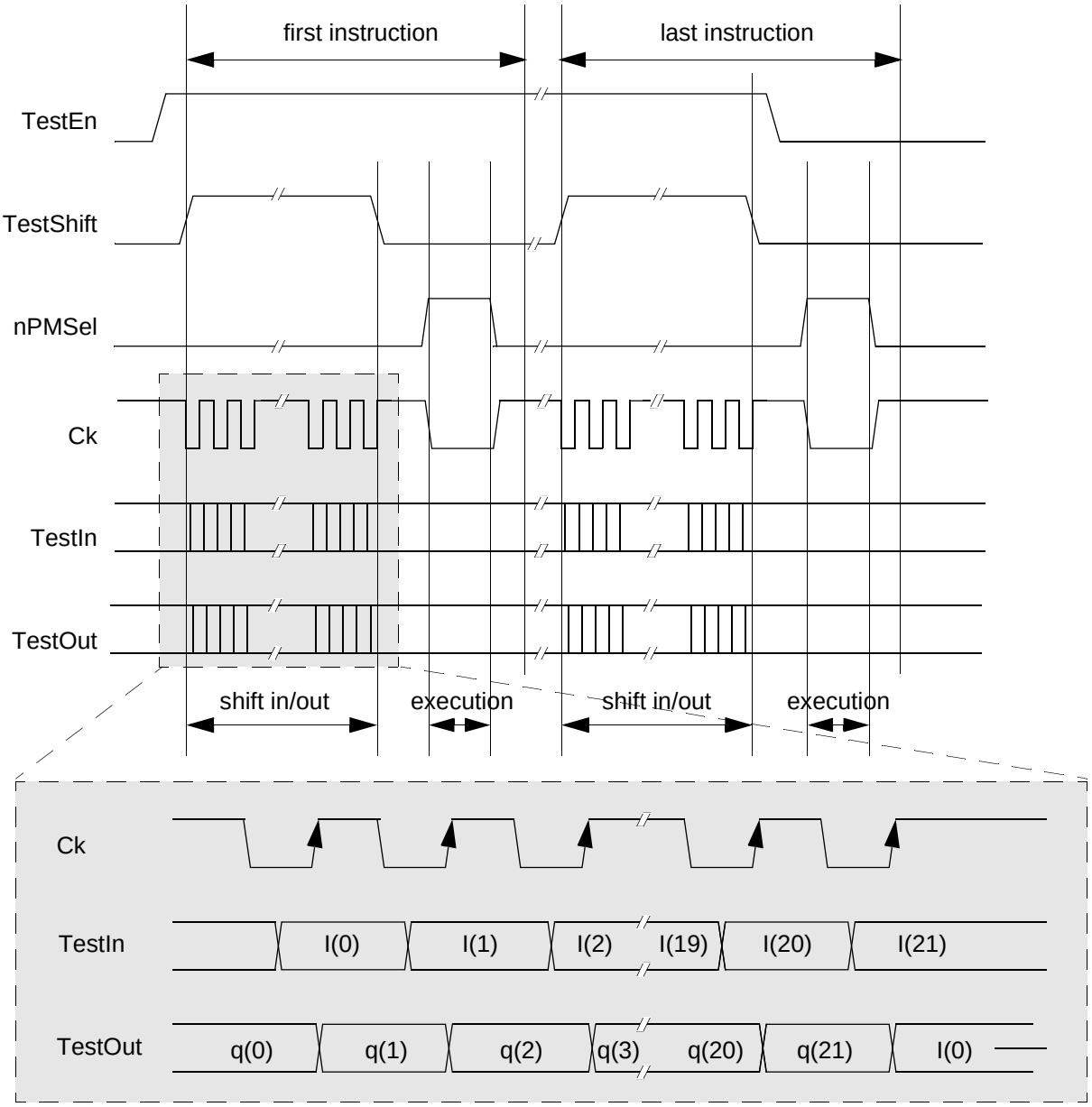
FIGURE 5.3: Pipeline snapshot.



5.2.2. Executing Instructions

Test mode sequencing is represented in Figure 5.4. The **TestEn** signal must be held high during the entire process. To shift an instruction in the processor, the **TestShift** signal must be held high, and the clock must be cycled 22 times, once for each bit presented on the **TestIn** input (LSB first). The **TestShift** signal must then return to zero for one clock cycle for the execution to take place.

FIGURE 5.4: Shift and execution of instructions in Test mode.



I(21:0): shifted in instruction
q(21:0): shifted out data

Some remarks are in order:

- To access the CR816 hardware stack, use PUSH and POP instructions to transfer data between the stack and the **ip** register, then use any ALU operation to copy **ip** into **a**.
- Data registers can be observed by copying them into **a** using any ALU operation.

- To read the data memory, load the **i0l** and **i0h** registers with the desired address and execute an instruction which access the data memory using the indexed mode, for example the following instruction would do the job: `MOVE r0,(i0)`. The content of an address in page zero can be accessed using the 8-bit direct addressing mode.
- To write in the data memory, use also the indexed mode and perform a store, like for example with the instruction `MOVE (i0),a`.
- Remember that the expected data will appear on **TestOut** only when the *second next* instruction is shifted in the processor.

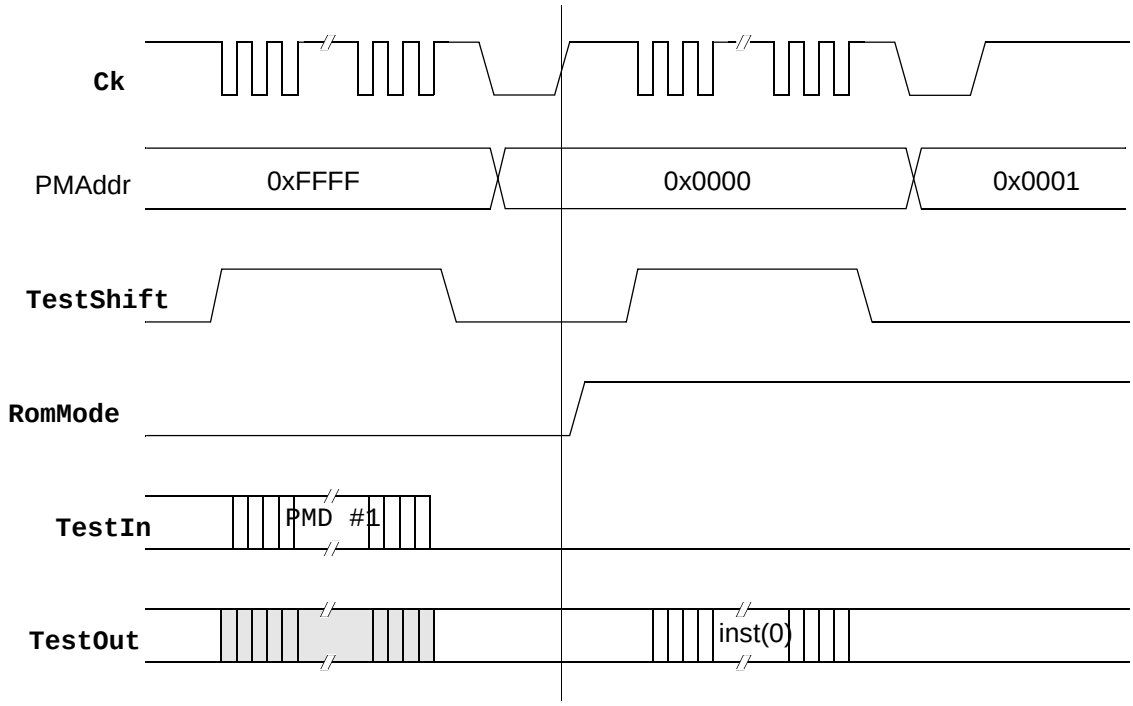
5.3. ROM Mode

The ROM mode can be used to test the program memory in production with no additional hardware. The ROM mode can only be entered from the Test mode, by shifting the `PMD #1` instruction inside the processor. When this instruction is decoded and executed, the processor enters ROM mode and the **RomMode** output signal goes to the high level.

Note	When the CR816 is in ROM mode, any instruction that is shifted in the core using the serial interface is ignored. The processor executes a NOP at each clock cycle.

Once in ROM mode, the processor reads the content of the program memory, starting at the current address of the program counter. Alternating shift and execute cycles allows to shift out the entire memory content. Figure 5.5 gives the sequence to enter the ROM mode. The processor quits the ROM mode when a `PMD #0` instruction is encountered in the program memory or a reset sequence is applied.

FIGURE 5.5: Rom mode sequencing.



Memory dump can start at any address, provided that a jump instruction is used to modify the **PC** register prior to entering ROM mode. The first instruction is shifted out after the first execution cycle has been executed

Note	The `PMD #0` is the only instruction that may be executed in ROM mode. Generally, this instruction is written at the last address of the physical program memory.

5.4. Test Signals Timing Requirements

Figure 5.6 defines the timing constraints and Table 5.1 gives the timing requirements that are related to the test interface signals.

FIGURE 5.6: Test signals timing constraints.

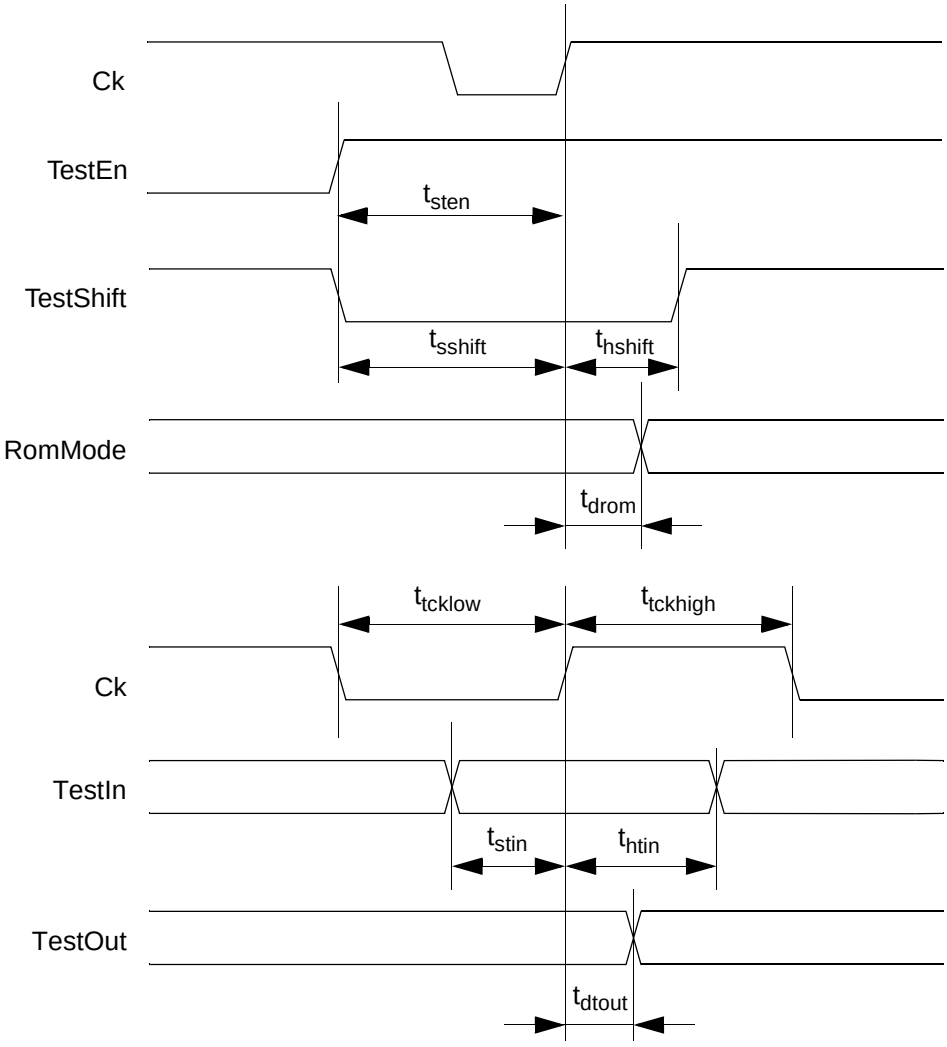


TABLE 5.1: Test signals timing requirements.

Timing parameter	Description	Min	Max
t_{cklow}	**Ck** low pulse width	x	
t_{ckhigh}	**Ck** high pulse width	x	
t_{sten}	**TestEn** setup time	x	
t_{sshift}	**TestShift** setup time	x	
t_{hshift}	**TestShift** hold time	x	
t_{stin}	**TestIn** setup time	x	
t_{htin}	**TestIn** hold time	x	
t_{dtout}	**TestOut** delay		x
t_{drom}	**RomMode** delay		x

5.5. Scan Testing

The CR816 interface provides the scan signals **ScanEn**, **ScanShift**, **ScanIn** and **ScanOut** (see "1.3. Interface Signals Description"). Only the **ScanEn** signal is used in a delivered soft core to disable clock gatings and to put the core in scan mode. **ScanEn** must be asserted (high) before starting scan testing. The other signals have still to be appropriately connected by inserting the scan chain.

Note The soft core version of the CR816 only includes clock gating to handle the Halt mode and the Test mode. A full clock gating has to be automatically inserted by the appropriate synthesis tool. Hence care should be taken during automatic scan insertion to ensure a correct use of the **ScanEn** signal with internally gated clocks. Recall that the **ScaEn** signal is used to disable all clock gating and to put the core in scan mode. This is unfortunately tool dependent.

Chapter 6
Advanced Features

6.1. Software Stack

The hardware stack that is available in the CR816 may not be appropriate for complex applications such as multi-process operating systems or use of high-level languages as it has a fixed depth (see "6.2. Hardware Stack Depth"). For these reasons and when the available data memory is large enough, a software stack is preferred.
The software stack uses the data memory to store values like return addresses of calling subroutines or the parameters passed to a subroutine. The size of the software stack is only limited by the size of the physical memory available.
When using a software stack, the hardware stack inside the CR816 is only used to store return addresses when processing interrupts. The hardware stack may also be reserved for the operating system to increase the processing speed.
The CR816 provides built-in instructions to ease the implementation and the use of a software stack.

6.1.1. Stack Access

The software stack can be accessed with the indexed addressing mode. For example, the stack can be implemented so that the **i0** register is always pointing to the data on the top of the stack. To write a new data, the index must be pre-decremented. When a data has been read, the index is post-incremented. This may be done with the instructions in Code 6.1.

CODE 6.1: Software stack push/pop assembly instructions.

```
MOVE -(i0), #data     ; push data onto the stack

MOVE REG, (i0)+       ; pop data out of the stack
```

6.1.2. Subroutine Calls

The CALLS instruction (instead of the CALL instruction) must be used to call a subroutine using the software stack. This instruction does not use the hardware stack but stores the return address in the **iph** and **ipl** registers. The RETS instruction (instead of the RET instruction) must be used to return from the subroutine. Code 6.2 gives an example of a subroutine call using a software stack.

Note	See the C compiler documentation to learn how to use the software stack in a C program.

CODE 6.2: Subroutine call using a software stack.

```
        ; main program
        ; (i0) is the software stack pointer
        ;
        ...
        MOVE   -(i0),#0x56
        MOVE   -(i0),#0x17    ; pushes parameters on the stack
        CALLS  sum            ; add parameters and put the result in r0
        ADD    i0l, #2
        INCC   i0h            ; deletes the parameters from the stack
        ...
; subroutine
sum:
        MOVE   -(i0),ipl
        MOVE   -(i0),iph      ; pushes the return address on the stack
        MOVE   r0, (i0, 2)    ; gets subroutine parameters
        ADD    r0, (i0, 3)    ; performs the calculation
        MOVE   iph, (i0)+
        MOVE   ipl, (i0)+     ; gets the return address
        RETS                  ; returns from subroutine
```

6.2. Hardware Stack Depth

The hardware stack is a register bank stored inside the CR816. The size of this bank is given by a parameter whose value may be defined prior to the synthesis of the core through a parameter called STACK_DEPTH in the RTL description. The default value of the STACK_DEPTH parameter is 4 and typical values are stack depths of four or five levels. Each level in the stack can be used to store a return address for a subroutine call as well as for interrupt processing. The stack is a LIFO, Last In First Out, type. The return address is stored in the stack by pushing it while it is recovered by popping it out. To have an unlimited stack, a software stack using the data memory is preferred (see "6.1. Software Stack").

A protection mechanism has been implemented inside the CR816 to avoid jumps to interrupt subroutines when the stack is full. An internal *stack full flag* is cleared at reset. An internal counter is incremented each time a value is pushed on the stack and decremented each time a value is popped out. When the counter reaches the stack depth value, the internal full stack flag becomes set and interrupt requests are disabled until a value is popped out of the stack (which clears the flag). Reciprocally an internal *stack empty flag* is available. Both the stack full and stack empty flags can be read by using the SFLAG instruction (see "SFLAG - Save Flags" on page 2-33).

6.3. Task Switching

The aim of this paragraph is to describe the instructions and their order that are necessary to store, and to restore, the current status of the processor (context switching). Only the internal resources of the CR816 are treated here. Application resources, like variables in memory are not considered.

The following internal resources can be stored or restored by context switching:

- Hardware stack (whose depth must be know by the programmer).
- Flags (C, V, Z).
- Program counter.
- Registers (**`r0`** - **`r3`**, **`i0l`** - **`i3h`**, **`iph`**, **`ipl`**, **`stat`**, **`a`**).

There is a particular order to follow when saving the resource. The register **a** should be saved first since most instructions modify it. Then the flags should be saved using the SFLAG instruction. After that, other registers may be saved in any order. The program counter is saved in a way that is dependent of the way the task switching is performed, i.e. interrupt, subroutine call or direct switch.

Note	Saving the status register may be optional since in a multi-process environment, the use of the status register should be reserved to the operating system. Care should be taken however not to generate soft interrupts when restoring the status register.

The examples of context saving and context restoring given below (Code 6.3 and Code 6.4, respectively) make the assumption that a space in the data memory has been reserved to store the content of the resources of the CR816. Since direct addressing on 16 bits is not possible, some resources must be stored first in page zero and then moved later to their final location.

CODE 6.3: Example of context saving.

```
; the address range used is 0x40-0x44 (must be in page zero)

MOVE 0x40, a      ; save the accumulator (this also saves the Z flag)
SFLAG             ; move the C and V flags in a
MOVE 0x41, a      ; save the previous saved C and V flags in memory
MOVE 0x42, stat   ; save the status register
MOVE 0x43, i0l    ; save i0l
MOVE 0x44, i0h    ; save i0h

; now that we saved the i0l and i0h registers,
; we can use the indexed addressing mode

MOVE i0l, #0x80   ; let's assume a free space at 0x0280 where
MOVE i0h, #0x02   ; we can save the resources
MOVE (i0)+, i1l
MOVE (i0)+, i1h
MOVE (i0)+, i2l
MOVE (i0)+, i2h
MOVE (i0)+, i3l
MOVE (i0)+, i3h
MOVE (i0)+, ipl
MOVE (i0)+, iph
MOVE (i0)+, r0
MOVE (i0)+, r1
MOVE (i0)+, r2
MOVE (i0)+, r3

; now save the content of the hardware stack
; here a stack depth of four is considered
; note: this is not required for interrupts

POP
MOVE (i0)+, ipl
MOVE (i0)+, iph
POP
MOVE (i0)+, ipl
MOVE (i0)+, iph
POP
MOVE (i0)+, ipl
MOVE (i0)+, iph
POP
MOVE (i0)+, ipl
MOVE (i0)+, iph

; move values stored in [MEM_0:MEM_4] to their new locations

MOVE a,     0x44
MOVE (i0)+, a
MOVE a,     0x43
MOVE (i0)+, a
MOVE a,     0x42
MOVE (i0)+, a
MOVE a,     0x41
MOVE (i0)+, a
MOVE a,     0x40
MOVE (i0)+, a     ; (i0) is now 0x0299
```

CODE 6.4: Example of context restoring.

```
; to restore the context, we must make the same things as for the saving,
; but in the reverse order

MOVE i0l, #0x99    ; this is the top of the area where we have stored
MOVE i0h, #0x02    ; the resources of this process

MOVE a, -(i0)
MOVE 0x40, a
MOVE a, -(i0)
MOVE 0x41, a
MOVE a, -(i0)
MOVE 0x42, a
MOVE a, -(i0)
MOVE 0x43, a
MOVE a, -(i0)
MOVE 0x44, a

; restore the hardware stack

MOVE iph, -(i0)
MOVE ipl, -(i0)
PUSH
MOVE iph, -(i0)
MOVE ipl, -(i0)
PUSH
MOVE iph, -(i0)
MOVE ipl, -(i0)
PUSH
MOVE iph, -(i0)
MOVE ipl, -(i0)
PUSH

; restore the registers

MOVE r3,  -(i0)
MOVE r2,  -(i0)
MOVE r1,  -(i0)
MOVE r0,  -(i0)
MOVE iph, -(i0)
MOVE ipl, -(i0)
MOVE i3h, -(i0)
MOVE i3l, -(i0)
MOVE i2h, -(i0)
MOVE i2l, -(i0)
MOVE i1h, -(i0)
MOVE i1l, -(i0)

; restore the registers used to switch the context

MOVE i0h,  0x44
MOVE i0l,  0x43
MOVE stat, 0x42
RFLAG 0x41         ; restore the C & V flags
MOVE a, 0x40       ; restore the accumulator and the Z flag
```

6.4. Frequency Division

The CR816 has a built-in frequency divider that can be used to divide the frequency of the internal clock of the processor, hence allowing a longer access time to both data and program memory and reducing the power consumption. The frequency divider does not insert any latency and the change of the division ratio does not insert any wait states.

The frequency divider can be typically used for the following applications:

- Power consumption reduction. When it is known that the processor will not have a lot of events to handle for a given time, reducing the frequency can be used when the Halt mode is inappropriate. The processor can very quickly return to its high frequency to process incoming data.
- To increase the available access time. It is possible to split the program memory into a high-speed small-size memory for frequently accessed instructions, and a low-speed large-size memory for the rest of the program.

Note There is a main difference between the Halt mode and the use of frequency division. In Halt mode, the processor does not execute any instruction and consumes almost no power, while using the frequency division allows the processor to still execute instructions, for example polling peripherals, but with some power consumption reduction (which is linear with the frequency).

Note Low speed peripherals can also use the Wait mode instead of the frequency division. See "6.8. Wait Mode" for more details.

The frequency divider provides the following division ratios: 1/1, 1/2, 1/4, 1/8, and 1/16.

The clock signal **Ck** is the input of the frequency divider. The output signal **CkOut** is the clock signal used inside the processor. Its frequency depends on the division ratio. All signals to/from the CR816 (except the serial test interface: **TesEnk**, **TestShift**, **TestIn**, **TestOut** and **RomMode**) are defined relatively to **CkOut**.
Two buses are used to control the frequency of the CR816: **FreqIn(3:0)** and **FreqOut(3:0)**. **FreqOut** is the output of an internal four bit register that is programmed by the FREQ instruction. Any value written in this register will appear at the output. This register is cleared at reset.
FreqIn is the input of the frequency divider. Often this input is directly connected to the **FreqOut** bus, allowing a frequency change by software. It is however possible to connect **FreqIn** to a control logic which selects a frequency division ratio, depending, for example on the value of the **PMAddr** bus.

Note In any case, the modification of the **FreqIn** signal *must occur when the clock is low*, with the respect of certain timing requirements (see "6.4.1. Timing Diagrams and Requirements").

The value set on the **FreqIn** bus will determine the frequency division ratio as given in Table 6.1. Other values than the one given in the table are reserved for future use.

TABLE 6.1: Frequency division settings.

FreqIn	Division
0000	no division
1000	1/2
1100	1/4
1110	1/8
1111	1/16

6.4.1. Timing Diagrams and Requirements

Figure 6.1 gives the timing diagrams for the signals involved in frequency division. Table 6.2 gives the timing requirements to meet.

FIGURE 6.1: FreqIn and FreqOut timing diagrams.

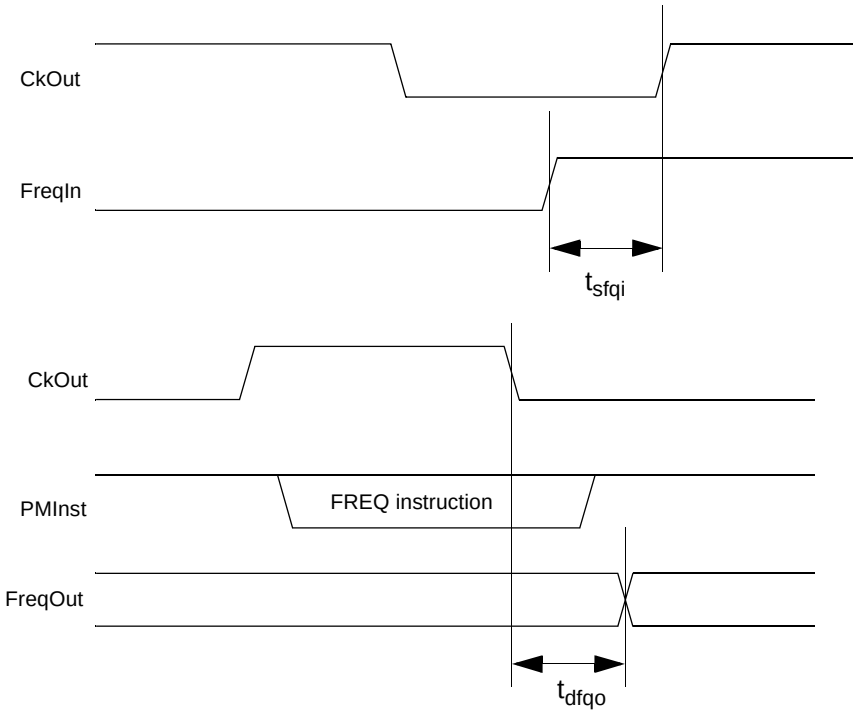


TABLE 6.2: Frequency signals timing requirements.

Timing parameter	Description	Min	Max
t_{sfqi}	**FreqIn** setup time w.r.t. **CkOut** rising	x	
t_{dfqo}	**FreqOut** hold time w.r.t. **CkOut** falling		x

6.5. Halt Mode

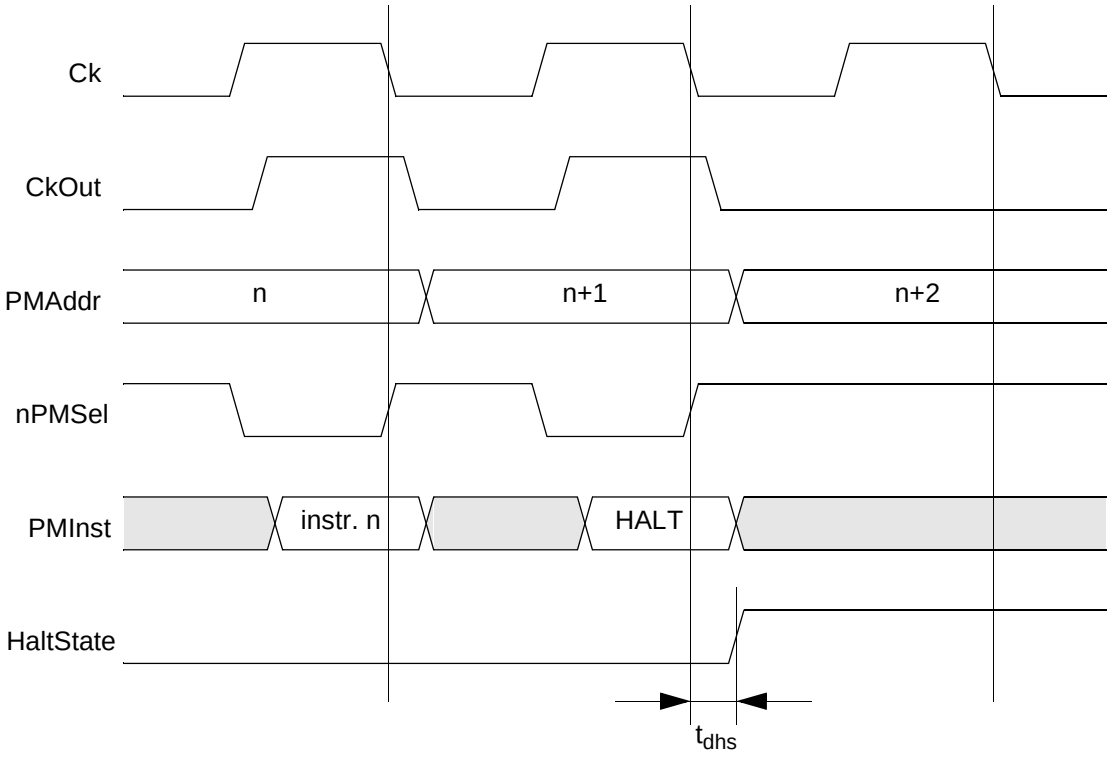
The HALT instruction can switch the CR816 in Halt mode in which the power consumption is minimal. The internal clock is stopped and almost nothing toggles. The processor can be waked up using either events, interrupts or a reset.

When the processor is in Halt mode, the **HaltState** signal is asserted, otherwise it is negated. **CkOut** remains at 0 when the processor is in Halt mode. If not in Halt mode, **CkOut** depends on **Ck** and of the current frequency division ratio.

6.5.1. Halting the Processor

To halt the processor, simply execute the HALT instruction. The processor will then go in Halt mode at the next falling edge of **CkOut**. The instructions preceding the HALT instruction are always terminated before the HALT instruction is executed.
The processor will not enter in Halt mode if interrupt requests or events are active. See 6.5.2 for more details about restarting from the Halt mode. See 4.2.2 for mode details on interrupt request enabling.
Figure 6.2 gives the timing diagram for halting the processor.

FIGURE 6.2: Timing diagram for halting the processor.



6.5.2. Restarting the Processor from Halt Mode

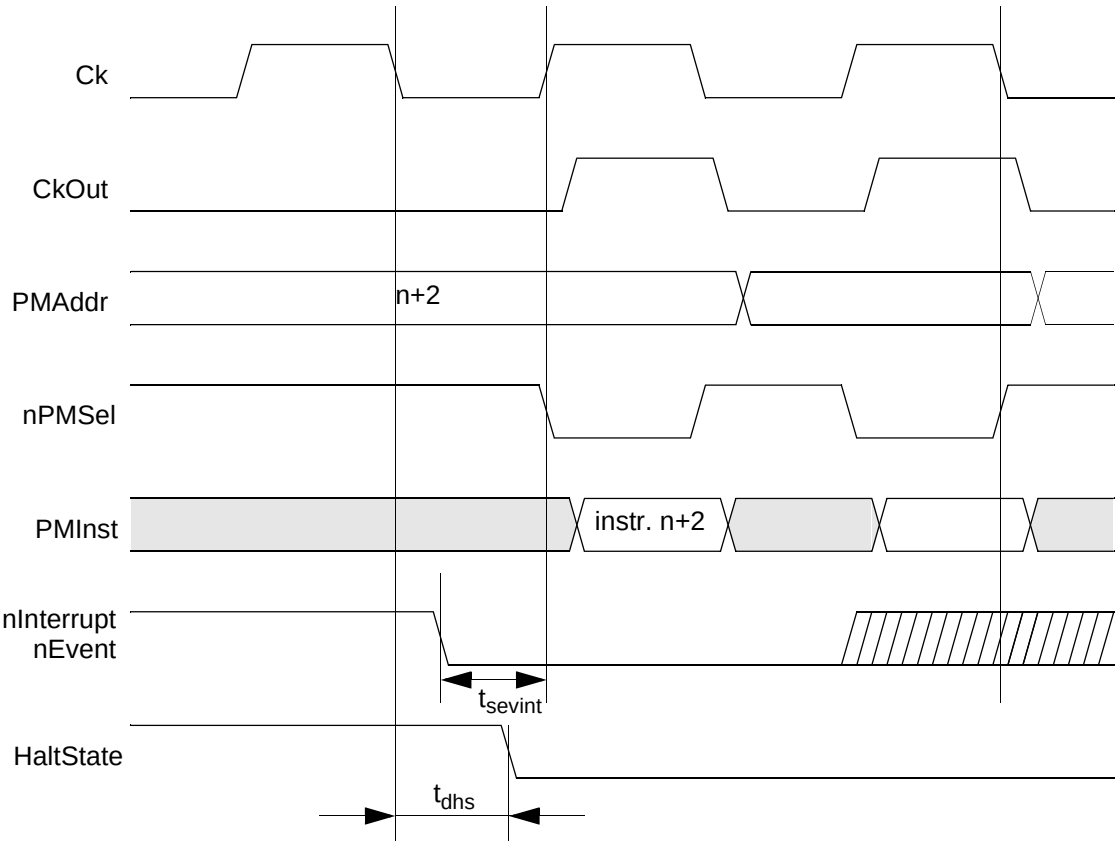
Figure 6.3 gives the timing diagram of the processor restarting from Halt mode after reception of an interrupt or an event. Interrupts and event must meet the timing requirement given in "4.2.1. Timing Re-

quirements for Interrupts and Events". Remember that interrupts must be properly enabled to wake up the processor. For example when the GIE flag is off, the only way to restart the processor is to do a reset.

Note	Software generated interrupts or events cannot restart the processor since the processor cannot execute instructions.

A reset can restart the processor from the Halt mode, but the program counter will be cleared. In this case, the state of the application at the time the halt occurred cannot be directly recovered.

FIGURE 6.3: Timing diagram for restarting the processor (continued from Figure 6.2).



6.5.3. Timing Requirements

TABLE 6.3: Halt mode timing requirements.

Timing parameter	Description	Min	Max
t_{dhs}	**HaltState** set and clear delay after **Ck** falling		x

6.6. Pipeline Exception

The three stage pipeline of the CR816 causes several instruction sequences to work in a different way than expected. These instructions are mostly related to interrupt and event signals. For "normal" instructions, the pipeline is completely transparent.

If an interrupt bit is set by software (e.g. write into the **`stat`** register with a `MOVE stat`) the pipeline causes the next instruction to be executed *before* the processor jumps to the interrupt subroutine. This allows one to supply a parameter to a “trap” as in Code 6.5.

CODE 6.5: Supplying a parameter to an interrupt subroutine.

```
SETB    stat, #4            ; trap
MOVE    a, #parameter       ;
```

If an event bit is set by software (e.g. write into the **`stat`** register with a `MOVE stat`) and if a `JEV` (jump on event) instruction immediately follows the move, the jump on event will act as if the move has not been executed, since the write into the **`stat`** register will occur only once the `JEV` has been executed. The move takes three cycles to be executed and the `JEV` only one.

Note See “"4.2.6. Disabling Interrupts" for a similar instruction delay case when dealing with hardware exceptions.

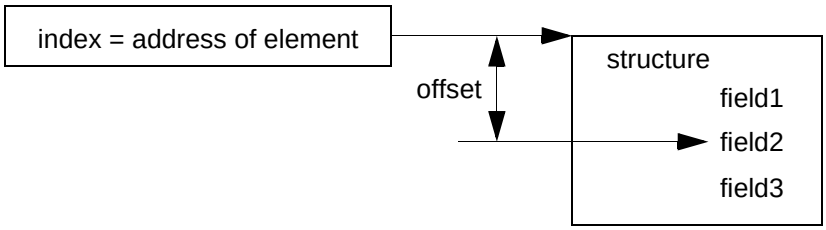
6.7. Use of Addressing Capabilities

The *direct addressing mode* can only access the first 256 bytes of the data memory. This memory area is called "Page Zero". It is often used for peripherals or for global variables, because it is possible to write an immediate 8-bit value in this zone with a single instruction. No internal register is allocated by the application for this mode.

The *indexed addressing mode with offset* is the most general addressing mode. The user may use any arithmetic or logic operation to calculate an index value, used as an address. Because the offset is determined for one application, this addressing mode suits well for relative accesses, inside structures for example (Figure 6.4).

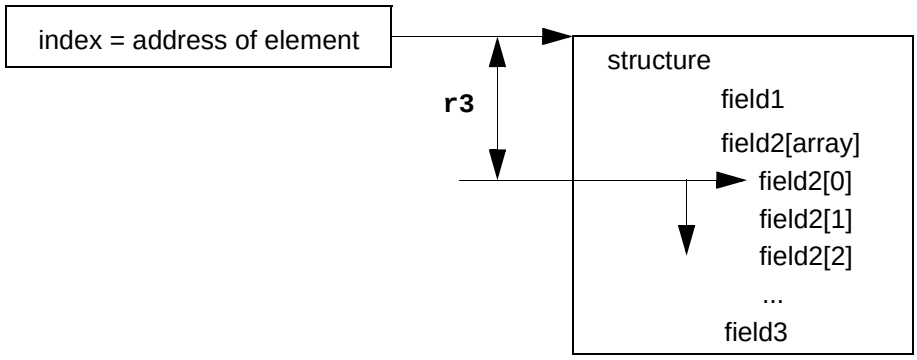
Note Neither `MUL`, `MULA`, `MSHR`, `MSHL` nor `MSHRA` instruction should be used to compute an index.

FIGURE 6.4: Use of indexed addressing mode with offset.



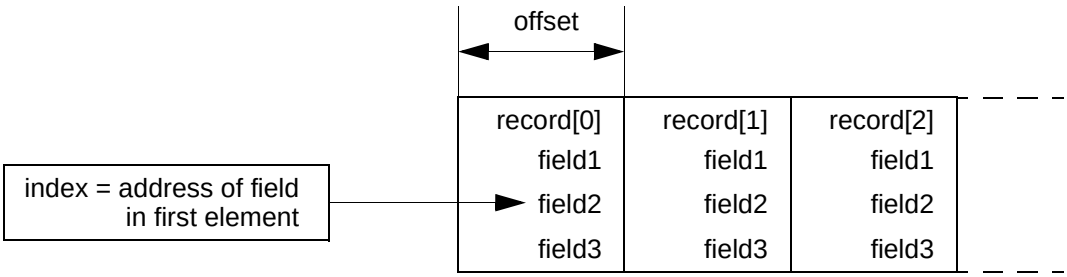
The *indexed address mode with offset in register* **`r3`** is especially well suited to fill areas inside a structure or inside a reserved page (Figure 6.5).

FIGURE 6.5: Use of indexed addressing mode with offset in register `r3`.



The *addressing mode with pre or post modification* can be used to implement data stacks inside the data memory (see "6.1. Software Stack"). These modes can also be useful to fill a particular field inside a structure that is itself inside an array of structure (Figure 6.6).

FIGURE 6.6: Use of addressing mode with pre or post modification.



For example, the operation:

```
record[1].field2 = value
```

can be realized with the instruction:

```
MOVE   (i0, size_of(record))+, value
```

with **`i0h`** and **`i0l`** initialized with the address of the field of the first element, i.e.:

```
i0 = &record[0].field2
```

6.8. Wait Mode

The Wait mode provides a way to slow down the CR816 accesses to the data memory without the need to modify the processor's frequency or to halt it. The Wait mode mechanism can be used only when the processor requests an access to the data memory, either in Read or Write mode. It can be used in multi-processor systems to share memory between multiple processors (or DMA controllers) or to handle low speed peripherals.

The Wait mode is controlled by the **DMReq**, **DMSel** and **ReqAccept** signals (see "3.2. Data Memory (and Peripheral) Interface"). The other signals of the data memory interface are not directly used to control the mechanism and their state will be kept as long as the processor is waiting.

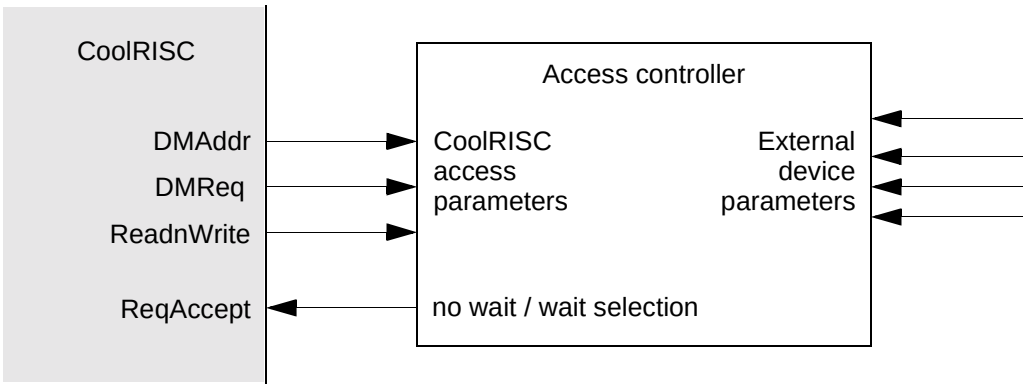
Note None of the CR816DL signals goes into high impedance when the processor is waiting. It is the responsibility of the system designer to implement multiplexers to choose between the two (or more) sources.

6.8.1. Wait Mode Control

The **DMReq** signal indicates, when asserted, that the processor will make an access to the data memory during the next phase (when the **CkOut** signal will go low). This signal is directly generated by the instruction decoder and may be subject to change during the decoding phase. A minimum setup time is however guaranteed before **CkOut**'s falling edge. This time is not only dependant on the clock frequency and on the time at which the instruction is provided to the core, but also on the decoding time of the core.

Before **CkOut**'s falling edge, the **DMReq**, **DMAddr**, and **ReadnWrite** signals are all stable and allow an external access controller to choose to insert wait states or not (Figure 6.7).

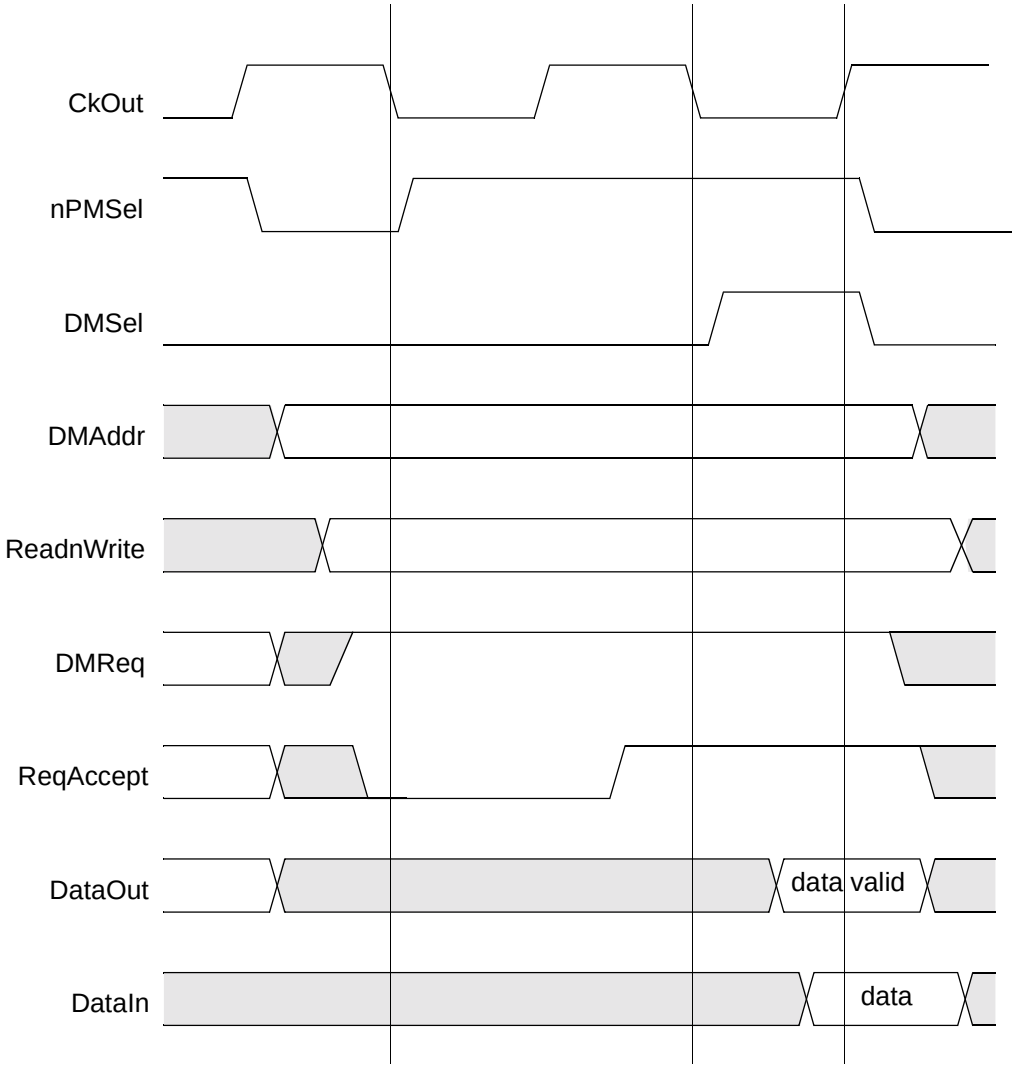
FIGURE 6.7: Access controller for wait states insertion.



If wait states are needed, the **ReqAccept** signal must be deasserted before **CkOut**'s falling edge with a minimum setup time. If **ReqAccept** is not deasserted before that time, the processor will proceed to the data memory access. The **DMSel** signal is only active when the access takes place. Otherwise, the processor will insert a wait state. Wait states will be inserted as long as **ReqAccept** remains deasserted. When **ReqAccept** is finally asserted, the access takes place and the processor continues with the normal execution of the program. The **nPMSel** signal remains deasserted as long as the processor is in Wait mode. Figure 6.8 gives the timing diagram for the Wait mode with one wait state.

Note Interrupts occurring during the Wait mode will have no effect until the access is finished.

FIGURE 6.8: Wait mode timing diagram with one wait state.



6.8.2. Timing Requirements

FIGURE 6.9: DMReq and ReqAccept timing constraints in Wait mode.

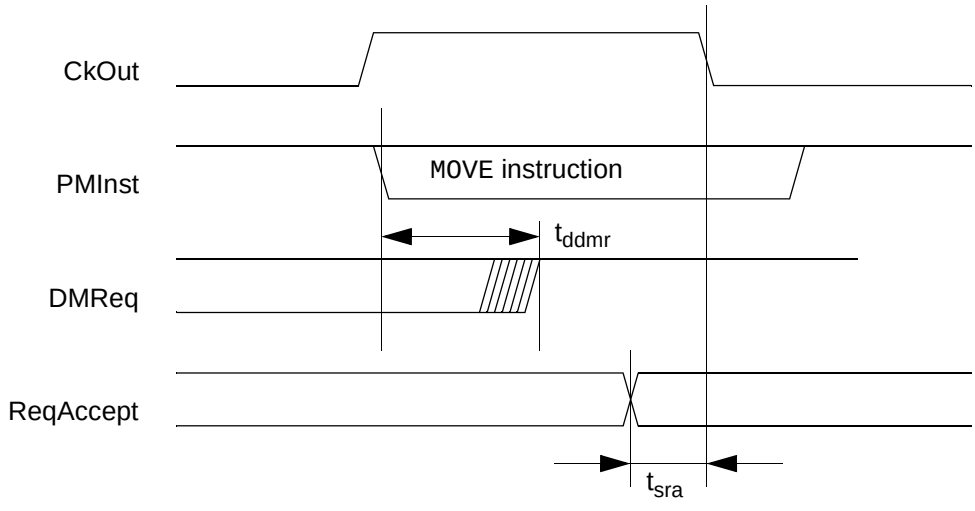


TABLE 6.4: Wait mode timing requirements.

Timing parameter	Description	Min	Max
t_{ddmr}	**DMReq** stable after instruction is stable		x
t_{sra}	**ReqAccept** setup time w.r.t. **CkOut** falling	x	

6.9. Register Forwarding Exception

As explained in "1.4. Pipeline", bypass mechanisms are implemented to avoid branch or load delays. There is however one exception when the multiplier is used to compute a program or data memory index (i.e. with **ip**, **ix** registers). In this particular case the program should wait for at least one cycle after the MUL or MULA operation before accessing the memory, either by inserting a NOP instruction or any instruction that does not execute a multiplication operation. The assembly code extracts in Code 6.6 are therefore forbidden.

CODE 6.6: Incorrect use of multiplication operation.

```
MUL     i0l, r0
MOVE    a, (i0)
; or
MUL     ipl, r0
JUMP    ip
```

©XEMICS 2002
All rights reserved. Reproduction in whole or in part is prohibited without the prior written consent of the copyright owner. The information presented in this document does not form part of any quotation or contract, is believed to be accurate and reliable and may be changed without notice. No liability will be accepted by the publisher for any consequence of its use. Publication thereof does not convey nor imply any license under patent or other industrial or intellectual property rights.
XEMICS PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF XEMICS PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE UNDERTAKEN SOLELY AT THE CUSTOMER'S OWN RISK.
Should a customer purchase or use XEMICS products for any such unauthorized application, the customer shall indemnify and hold XEMICS and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs damages and attorney fees which could arise.