

---

# TN8000.04

## *Technical note*

---

# CoolRISC 816 instruction codes and examples

Author : Michel Chevroulet  
For further information please contact:

**XEMICS S.A.**  
**Email:** [info@xemics.com](mailto:info@xemics.com)  
**Web:** <http://www.xemics.com>

# 1 Introduction

This application note concerns all XEMICS products based on the CoolRISC816 core, including all the XE8000 products.

## 2 CoolRISC 816 basics

### 2.1 Introduction

The XE8000 family is built around the CoolRISC 816 processor core. This is a Harvard type RISC processor (Program address is separated from data address). Is it extremely efficient using large instruction words (22 bits), one clock per cycle instruction set (inclusive of multiplication) and efficient pipeline.

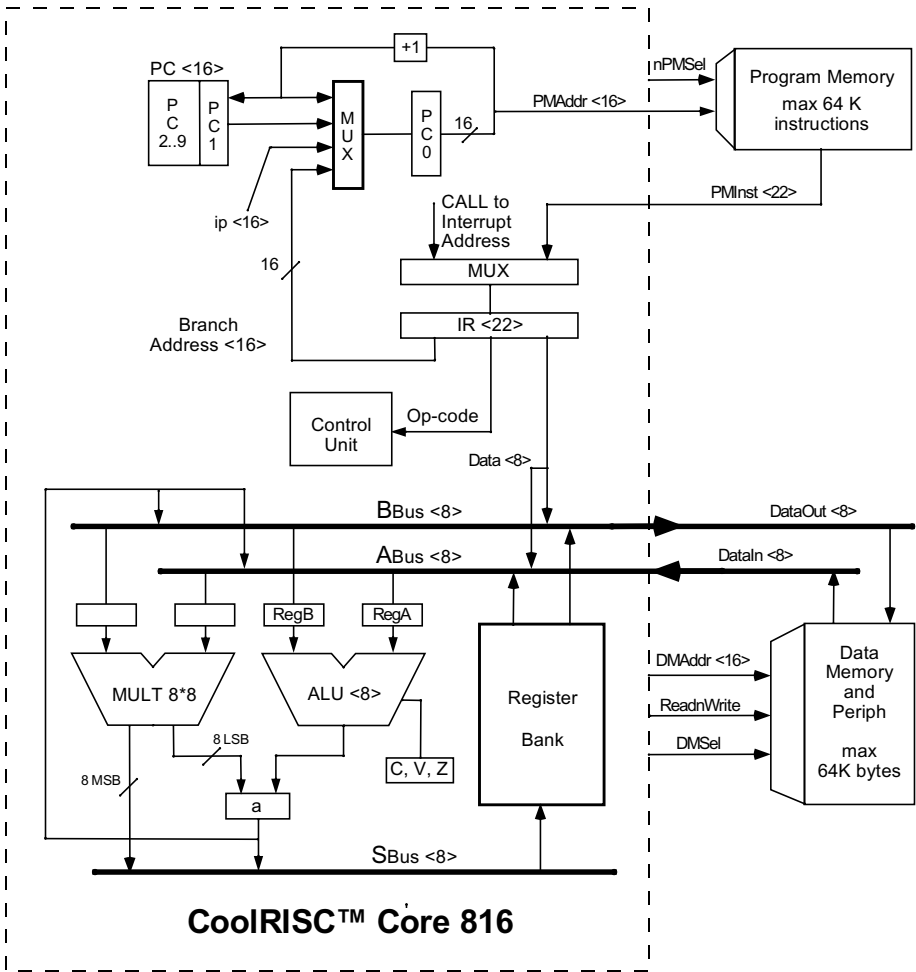


Figure 2.1: CoolRISC 816 core

#### 2.1.1 Pipeline

The CoolRISC architecture is based on a 3-stage pipeline. One instruction enters the pipeline at each clock cycle and executes in a maximum of 3 cycles. The CoolRISC pipeline suffers no penalty such as delay slots or branch delays present in most RISC processors. Thus the clock count per instruction (CPI) is exactly one.

As a result the number of cycles needed to execute a task is easily determined, since it matches the number of executed instructions.

Figure 2.2 shows the timing diagram for the pipeline. Arithmetic instructions go through all three stages of the pipeline, thus occurring in 3 clock cycles. A bypass mechanism is used to avoid any load delay[10]. It should be mentioned that existing 4-bit and 8-bit microprocessors typically need between 4 to 20 clocks per instruction (CPI), because they do not use a pipeline. The efficiency of the CoolRISC architecture is increased by a factor of 4 to 20 compared to these microprocessors.

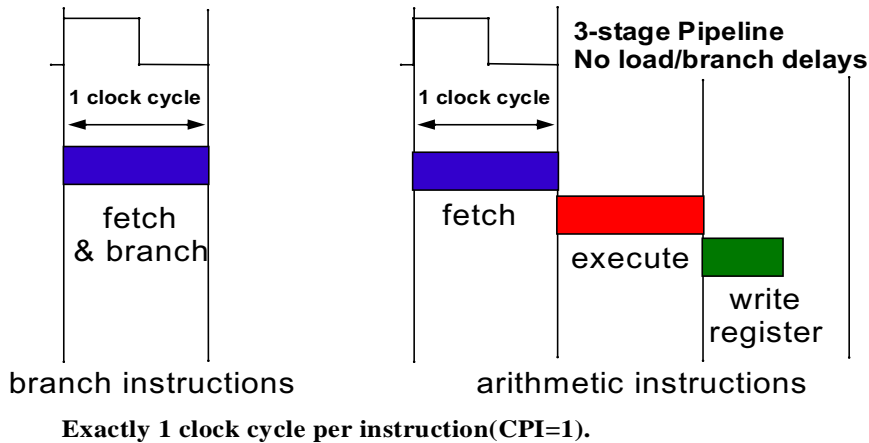


Figure 2.2: CoolRISC Pipeline

Figure 2.2 presents the timing diagram for the execution of different types of instructions.

The first instruction is a typical ALU operation with a first operand in Data Memory (DM) and a second operand in a register. The result is stored in the destination register. During the first clock cycle, the Program Memory (PM) is pre-charged in the first phase and the instruction is read and is decoded in the second phase. During the second clock cycle, the register and the DM are read in phase 3 and the ALU operation is executed in phase 4. The last clock cycle contains only a single phase (phase 5) used to store the result in the destination register.

The second instruction shown is a Data Memory store instruction. The first clock cycle is identical for all instructions. The second clock cycle contains only phase 3 in which the value of a register is written into the DM.

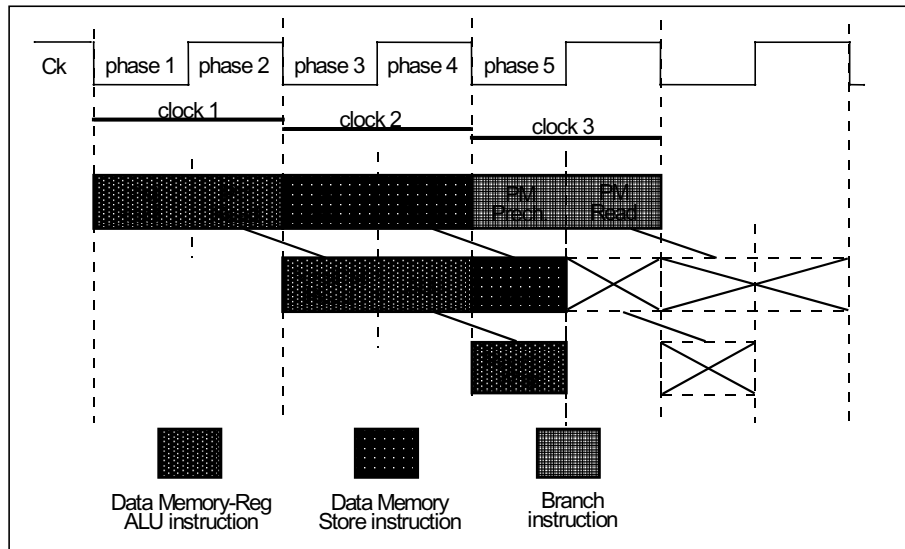


Figure 2.3: Pipeline execution of different instructions

The last instruction shown is a branch instruction. A single clock cycle is necessary for all branch instructions (conditional or unconditional jump, call, return). During phase 2, the next Program Memory address can be determined while considering an already computed test condition (computed during phase 4 of the previous instruction, which is phase 2 of the considered branch instruction). The new address is loaded into the PC at the high-to-low transition of the clock between phase 2 and 3.

Branch instructions executed in one clock do not result in branch delays that generally degrade the pipeline performance [10]. Thus, CPI=1 is not a peak value, but rather a characteristic of the CoolRISC® architecture.

Figure 2.3 shows a Data Memory-reg ALU instruction followed by a DM store instruction. The first instruction stores its result in a register during phase 5 which is phase 3 of the DM store instruction. A bypass mechanism allows the DM store instruction to read the register that is written by the preceding ALU instruction. Such a mechanism does not require load delays.

As the CoolRISC pipeline is not affected by branch or load delays [11], the pipeline hardware is simplified (no branch prediction needs to be performed [10]). This makes the CoolRISC® pipeline very efficient and low in power consumption.

### 2.1.2 Gated clocks

The gated clock technique has been extensively used in the CoolRISC® design. It uses the ALU with input and control registers that are loaded only when an ALU operation has to be executed. During the execution of another type of instruction (branch, store, etc...), these registers are not clocked, thus no transitions occur in the ALU. This reduces power consumption.

A similar mechanism is used for the instruction registers. Thus in a branch, which is executed only in the first pipeline stage, no transitions occur in the second and third stages of the pipeline.

Gated clocks can be advantageously combined with the pipeline architecture. When input and control registers have to be implemented to obtain a gated clock ALU, they are naturally used as pipeline registers.

### 2.1.3 Low frequency modes

The internal frequency of the processor can be reduced by a factor of 2, 4, 8 or 16. The division factor is both hardware and software controlled.

The **FREQ** instruction sets the basic division factor which is output on the processor **FreqOut[3:0]** bus. This value can be combined with other signals in an external hardware decoder to compute the final division factor which is then input on the **FreqIn[3:0]** bus.

Power consumption can be further decreased by putting the processor in the low-power standby mode with the **HALT** instruction. It will restart when an Event or an Interrupt occurs.

### 2.1.4 Stand-by Mode

The **HALT** instruction puts the processor in standby mode in which power consumption is minimum. The clock is stopped at the entrance of the processor to prevent any transition in the core.

### 2.1.5 CoolRISC® Core Features

CoolRISC®CoolRISC® Core	Generic8 16	XE88LC01 XE88LC03 XE88LC05
CPI (clock per instruction)	1	1
Pipeline	3 stages	3 stages
Branch/Load delay	no	no
Data Width	8	8
No. of Registers	16	16
Max. Program Memory size	64k * 22	8k * 22
Max. Data Memory size	64k * 8	512 * 8
Instruction size	22	22
No. of Program Memory Index Registers	1	1
No. of Data Memory Index Registers	4	4
No. of Program Memory pages	1 * 64k	1 * 8k
No. of Data Memory pages	256 * 256	2.5 * 256
No. of Data Memory addressing modes	8	8
Software CALL (branch & link)	yes	yes
No. of nested hardware CALL	1..8	4
No. of Interrupt Inputs	3	3
Nested Interrupts	yes	yes
No. of EVENT Inputs (wake-up the CPU)	2	2
Test access	serial	serial
Halt mode	yes	yes
Clock reduction by Software	2..16	2..16
8 by 8 to 16 multiplication in one instruction	yes	yes
Barrel Shifter	yes	yes
Two-Complement capabilities	yes	yes

**Table 2.1: CoolRISC core main characteristics**

## 2.2 Programmer's Model

### 2.2.1 CoolRISC® 816 Architecture

Figure 2.1 shows the CoolRISC® Core 816 architecture which is a 8-bit microprocessor core available with 16 registers and 22-bit wide instructions.

### 2.2.2 Instruction Set

Table 2.2 presents the instruction set of the CoolRISC® 816.

The CoolRISC® provides a RISC instruction set with four main categories:

- branch instructions
- transfer instructions
- arithmetic and logic instructions
- special instructions.

Unlike most RISC microprocessors, the CoolRISC® core provides instructions that 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 the Data Memory or in a second register. The result can be stored either in a third register or in the first one.

Also, unlike other RISC microprocessors and similar to classic 8-bit microprocessors, the CoolRISC® architecture provides an accumulator (**a**) located at the ALU output. This accumulator stores the last ALU result and should be used as an intermediate result for the next ALU operation. This accumulator is mapped in the register bank.

Similarly, both the Branch & Link instruction of RISC microprocessors (Software Call) and the classic hardware Call are provided by the CoolRISC® architecture.

From the programming point of view, CoolRISC® architecture can be used either as a true RISC architecture or as a more classic 8-bit architecture.

The CoolRISC® 816, with its overflow flag (**V**) and its arithmetic instructions (**SHRA**, **CMPA**, **MULA**, **MSHRA**) fully supports signed numbers in the two-complement representation. The **MUL** & **MULA** instructions execute 8 by 8 on 16 bits multiplication. Because the result is on 16 bits, the 8 MSB bits are stored in the destination register and the 8 LSB bits are stored in the accumulator **a**. All the flags (**C**, **Z**, **V**) must be considered as unknown after these instructions.

The multiple shift instructions **MSHL**, **MSHR** & **MSHRA** use the multiplication instructions with an immediate operand. For this reason, the value of **a** is different than the destination register, as in the **MUL** & **MULA** instructions. Thus the shifted “out” bits are never lost (they are either in **a** or in the destination register), and these instructions can be used to split a byte into two bytes, in a single instruction. For example, a “SWAP r0” can be implemented as follows:

```

; r0 = 0xYZ
MSHL          r0, #4          ; r0 <- 0Y, a <- Z0
ADD           r0, a          ; r0 <- ZY
    
```

The conditional move instructions (**CMVD** & **CMVS**) can be used to find the maximum (or minimum) value in a table. If **i0** is a pointer to the table, r0 will contain its maximum value after the following sequence:

```

CMP(A) r0, (i0)
CMVS r0, (i0)+          ; r0 <- DM(i0) if r0 < DM(i0)
CMP(A)r0, (i0)
CMVS r0, (i0)+          ; r0 <- DM(i0) if r0 < DM(i0)
.....
    
```

NAME	Parameters	res	op1	op2	FUNCTION	MODIF.
JUMP	addr:16				PC0 <- addr	-, -, -, -
	ip				PC0 <- ip	
Jcc	addr:16				if cc then PC0 <- addr	
	ip				if cc then PC0 <- ip	
CALL	addr:16				PCn <- PCn-1 (n>1), PC1 <- PC0+1, PC0 <- addr	
	ip				PCn <- PCn-1 (n>1), PC1 <- PC0+1, PC0 <- ip	
CALLS	addr:16				ip <- PC0+1, PC0 <- addr:16	
	ip				ip <- PC0+1, PC0 <- ip	
RET					PCn-1 (n>0) <- PCn	
RETS					PC0 <- ip	
RETI					PCn-1 (n>0) <- PCn, GIE <- 1	
PUSH					PCn <- PCn-1 (n>1), PC1 <- ip, PC0 <- PC0+1	
POP					ip <- PC1, PCn-1 (n>1) <- PCn, PC0 <- PC0+1	
MOVE	reg, data:8	reg	data		res <- op1	-, -, Z, a
	reg1, reg2	reg1	reg2			
	reg, eaddr	reg	eaddr			
	eaddr, reg	eaddr	reg			- , - , - , -
	addr:8, data:8	addr	data			
CMVD	reg1, reg2	reg1	reg2		if C=0 then res <- op1	- , -, Z, a
CMVS	reg, eaddr	reg	eaddr		if C=1 then res <- op1	
SHL	reg1, reg2	reg1	reg2		res(bitn) <- op1(bitn-1) (0<n<8), res(0) <- 0, C <- op1(7)	C, V, Z, a
	reg, eaddr	reg	eaddr			
SHLC	reg1, reg2	reg1	reg2		res(bitn) <- op1(bitn-1) (0<n<8), res(0) <- C, C <- op1(7)	C, V, Z, a
	reg, eaddr	reg	eaddr			
SHR	reg1, reg2	reg1	reg2		res(bitn-1) <- op1(bitn) (0<n<8), res(7) <- 0, C <- op1(0)	C, V, Z, a
	reg, eaddr	reg	eaddr			
SHRC	reg1, reg2	reg1	reg2		res(bitn-1) <- op1(bitn) (0<n<8), res(7) <- C, C <- op1(0)	C, V, Z, a
	reg, eaddr	reg	eaddr			
SHRA	reg1, reg2	reg1	reg2		res(bitn-1) <- op1(bitn) (0<n<8), res(7) <- op1(7), C <- op1(0)	C, V, Z, a
	reg, eaddr	reg	eaddr			
CPL1	reg1, reg2	reg1	reg2		res <- NOT (op1)	-, -, Z, a
	reg, eaddr	reg	eaddr			
CPL2	reg1, reg2	reg1	reg2		res <- NOT (op1) +1, if op1 = 0 then C = 1	C, V, Z, a
	reg, eaddr	reg	eaddr			
CPL2C	reg1, reg2	reg1	reg2		res <- NOT (op1) +C, if op1 = 0 then C = 1	C, V, Z, a
	reg, eaddr	reg	eaddr			
INC	reg1, reg2	reg1	reg2		res <- op1 +1, if overflow then C = 1	C, V, Z, a
	reg, eaddr	reg	eaddr			
INCC	reg1, reg2	reg1	reg2		res <- op1 +C, if overflow then C = 1	C, V, Z, a
	reg, eaddr	reg	eaddr			
DEC	reg1, reg2	reg1	reg2		res <- op1 -1, if underflow then C = 0	C, V, Z, a
	reg, eaddr	reg	eaddr			

**Table 2.2: CoolRISC 816 Instruction Set**

NAME	Parameters	res	op1	op2	FUNCTION	MODIF.
DECC	reg1, reg2	reg1	reg2		res <- op1 -(1 -C), if underflow then C = 0	C, V, Z, a
	reg	reg	reg			
	reg, eaddr	reg	eaddr			
AND	reg, data:8	reg	data	reg	res <- op1 AND op2	-, -, Z, a
	reg1, reg2, reg3	reg1	reg2	reg3		
	reg1, reg2	reg1	reg2	reg1		
	reg	reg	eaddr	reg		
OR	reg, data:8	reg	data	reg	res <- op1 OR op2	-, -, Z, a
	reg1, reg2, reg3	reg1	reg2	reg3		
	reg1, reg2	reg1	reg2	reg1		
	reg	reg	eaddr	reg		
XOR	reg, data:8	reg	data	reg	res <- op1 XOR op2	-, -, Z, a
	reg1, reg2, reg3	reg1	reg2	reg3		
	reg1, reg2	reg1	reg2	reg1		
	reg	reg	eaddr	reg		
ADD	reg, data:8	reg	data	reg	res <- op1 + op2, if overflow then C=1	C, V, Z, a
	reg1, reg2, reg3	reg1	reg2	reg3		
	reg1, reg2	reg1	reg2	reg1		
	reg	reg	eaddr	reg		
ADDC	reg, data:8	reg	data	reg	res <- op1 + op2 + C, if overflow then C=1	C, V, Z, a
	reg1, reg2, reg3	reg1	reg2	reg3		
	reg1, reg2	reg1	reg2	reg1		
	reg	reg	eaddr	reg		
SUBD	reg, data:8	reg	data	reg	res <- op1 -op2, if underflow then C=0	C, V, Z, a
	reg1, reg2, reg3	reg1	reg2	reg3		
	reg1, reg2	reg1	reg2	reg1		
	reg	reg	eaddr	reg		
SUBDC	reg, data:8	reg	data	reg	res <- op1 -op2 - (1-C), if underflow then C=0	C, V, Z, a
	reg1, reg2, reg3	reg1	reg2	reg3		
	reg1, reg2	reg1	reg2	reg1		
	reg	reg	eaddr	reg		
SUBS	reg, data:8	reg	data	reg	res <- op2 -op1, if underflow then C=0	C, V, Z, a
	reg1, reg2, reg3	reg1	reg2	reg3		
	reg1, reg2	reg1	reg2	reg1		
	reg	reg	eaddr	reg		
SUBSC	reg, data:8	reg	data	reg	res <- op2 -op1 - (1-C), if underflow then C=0	C, V, Z, a
	reg1, reg2, reg3	reg1	reg2	reg3		
	reg1, reg2	reg1	reg2	reg1		
	reg	reg	eaddr	reg		
MUL	reg, data:8	reg	data	reg	res <- op1 * op2 (15:8), a <- op1 * op2 (7:0), unsigned	-, -, -, a
	reg1, reg2, reg3	reg1	reg2	reg3		
	reg1, reg2	reg1	reg2	reg1		
	reg	reg	eaddr	reg		
MULA	reg, data:8	reg	data	reg	res <- op1 * op2 (15:8), a <- op1 * op2 (7:0), signed (2 complement)	-, -, -, a
	reg1, reg2, reg3	reg1	reg2	reg3		
	reg1, reg2	reg1	reg2	reg1		
	reg	reg	eaddr	reg		
MSHL	reg, shift:3				a(bitn) <- reg(bitn-shift) for (bitn >= shift), reg(bitn) <- reg (bitn+8-shift) for (bitn < shift)	-, -, -, a
MSHR	reg, shift:3				reg(bitn) <- reg(bitn+shift) for (bitn + shift < 8), a(bitn) <- reg (bitn-8+shift) for (bitn + shift >= 8)	-, -, -, a
MSHRA	reg, shift:3				a <- SHRA(shift,reg), a <- SHL(8-shift,reg), SHRA propagates sign, do not use with shift=0x01	-, -, -, a
CMP	reg, data:8		reg	data	if op2 > op1 then C <- 0, V = C AND NOT(Z), unsigned	C, V, Z, a
	reg1, reg2		reg1	reg2		
	reg, eaddr		reg	eaddr		

**Table 2.2: COOLRISC 816 Instruction Set**

NAME	Parameters	res	op1	op2	FUNCTION	MODIF.
CMPA	reg, data:8		reg	data	if op2 > op1 then C <- 0, V = C AND NOT(Z), signed	C, V, Z, a
	reg1, reg2		reg1	reg2		
	reg, eaddr		reg	eaddr		
TSTB	reg, bit:3				Z <- NOT(reg(bit))	-, -, Z, a
SETB	reg, bit:3				reg(bit) <- 1	-, -, Z, a
CLRB	reg, bit:3				reg(bit) <- 0	-, -, Z, a
INVB	reg, bit:3				reg(bit) <- NOT(reg(bit))	-, -, Z, a
SFLAG					a(7) <- C, a(6) <- C XOR V	-, -, -, a
RFLAG	reg		reg		flags <- op1, SHL op1, SHL a	C, V, Z, a
	eaddr		eaddr			
FREQ	divn:4				set cpu frequency divider	-, -, -, -
HALT					stops CPU	-, -, -, -
NOP					no operation	-, -, -, -
PMD	s:1				if s=1 then starts program dump, if s=0 stops program dump	-, -, -, -

**Table 2.2: CoolRISC 816 Instruction Set**

	Parameters	Data Memory (DM) access	Index update	Addressing mode name
eaddr	addr:8	DM(addr)		Direct addressing
	(ix)	DM(ix)		Indexed addressing
	(ix, offset:8)	DM(ix+offset)		Indexed addressing with immediate offset
	(ix, r3)	DM(ix+r3)		Indexed addressing with register offset
	(ix)+	DM(ix)	ix <- ix+1	Indexed addressing with post-modification of index
	(ix, offset:7)+	DM(ix)	ix <- ix+offset	
	-(ix)	DM(ix-1)	ix <- ix-1	Indexed addressing with pre-modification of index
-(ix, offset:7)	DM(ix-offset)	ix <- ix-offset		

**Table 2.3: CoolRISC 816 addressing modes**

	11 Conditions	Test	
cc	CS	C = 1	
	CC	C = 0	
	ZS	Z = 1	
	ZC	Z = 0	
	VS	V = 1	
	VC	V = 0	
	EV	(EV0 OR EV1) = 1	
	After CMP d, s		
	EQ	d = s	
	NE	d <> s	
	GT	d > s	
	GE	d >= s	
	LT	d < s	
LE	d <= s		

**Table 2.4: CoolRISC 816 conditional jump (Jcc) conditions**

	information type
addr:8	8-bit address
addr:16	16-bit address
ip	Program Memory Index
ix	4 Data Memory (DM) Indexes i0, i1, i2, i3
data:8	8-bit data
offset:8	8-bit positive offset
offset:7	7-bit positive offset
bit:3	3-bit Bit select value : 0..7

**Table 2.5: CoolRISC 816 instruction construction**

	information type
shift:3	3-bit shift value : 2..7
divn:4	0b0000: nodiv, 0b1000: div by 2, 0b1100: div by 4, 0b1110: div by 8, 0b1111: div by 16

**Table 2.5: CoolRISC 816 instruction construction**

	16 Registers	Function
reg reg1 reg2 reg3	r0	
	r1	
	r2	
	r3	DM offset
	i0l	i0[7:0]
	i0h	i0[15:8]
	i1l	i1[7:0]
	i1h	i1[15:8]
	i2l	i2[7:0]
	i2h	i2[15:8]
	i3l	i3[7:0]
	i3h	i3[15:8]
	ipl	ip[7:0]
	iph	ip[15:8]
	stat	status
a	accu	

**Table 2.6: CoolRISC 816 internal registers**

registers organization															register name	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
PC															PC	
iph							ipl								ip	
i0h							i0l								i0	
i1h							i1l								i1	
i2h							i2l								i2	
i3h							i3l								i3	
															a	
															r0	
															r1	
															r2	
															r3	
															stat	
IE2	IE1	GIE	IN2	IN1	IN0	EVI	EV0									stat

**Table 2.7: CoolRISC 816 registers organization**

Interrupt	CALL address
IN0	3
IN1	1
IN2	2

**Table 2.8: CoolRISC 816 interrupts**

The **PUSH & POP** instructions allow the processor to read and write its hardware stack. This can be used to “extend the depth” of the stack when nested interrupts are needed. The software implementation of nested interrupts can be achieved with the following instructions :

```

Interrupt          ; PC1 <- return address
POP                ; ip <- PC1
MOVE eaddr1, ipl   ; eaddr1 <- return. address
MOVE eaddr2, iph   ; eaddr2 <- return. address
.....             ; 1 stack level is now freed
MOVE ipl, eaddr1   ; ipl <- eaddr1
    
```

```

MOVE iph, eaddr2      ; iph <- eaddr2
PUSH                  ; PC1 <- ip
RET(I)                ; PC0 <- PC1
    
```

### 2.2.3 Register bank

The register bank of the 8-bit CoolRISC core 816 is described in Table 2.6 and Table 2.7.

Four data registers and an accu register are available, as well as a two-byte Program Memory Index (**ip**) and four two-byte Data Memory Indexes (**ix**) registers. These Index registers allow the user to address up to 64k instructions and up to 64k Data bytes. **ip** is also used to save the return address with the Software Call instruction (**CALLS**). A Status register (**stat**) is used *only* to control Interrupts and Events. **r3** can also be used as an offset register in the indexed addressing mode of the Data Memory.

If some of the 10 Data & Program Memory Index registers are not used permanently as indexes, they can be used as data registers. Furthermore, if some of them are not used in a given routine, they can be used as temporary data registers, increasing efficiency

Table 2.9 summarises the names and the roles of the registers.

Register Names	Data Regs	Data Mem. Index	Prog. Mem. Index	Soft. Call
a	"X"			
r0	X			
r1	X			
r2	X			
r3	X	DM offset		
i0l	X	X		
i0h	X	X		
i1l	X	X		
i1h	X	X		
i2l	X	X		
i2h	X	X		
i3l	X	X		
i3h	X	X		
ipl	X		X	X
iph	X		X	X
stat				

**Table 2.9: Registers Roles**

### 2.2.4 Program Memory addressing modes

The CoolRISC 816 provides one 16-bit Program Memory index called **ip** in order to address indirectly the 64K of Program Memory (**ipl** for the LSB bits, **iph** for the MSB).

The address field in a direct Jump instruction is of 16 bits. This allows addressing directly to the whole Program Memory space.

### 2.2.5 Data Memory addressing modes

The CoolRISC® 816 provides four 16-bit Data Memory Indexes called **ix** (**i0**, **i1**, **i2** & **i3**) in order to address 64K bytes of Data Memory (**ixl** for the LSB bits, **ixh** for the MSB).

The Data Memory is organised as 256 pages of 256 bytes. The whole memory can be addressed indirectly using *ix*, while only page 0 can be addressed directly.

### 2.2.5.1 Direct addressing

*Data Memory access : DM(addr:8)*

This mode is limited to the addressing of page 0. The field *addr:8* of the corresponding instructions is used as a direct address (DMAAddr[7:0]) while the MSB bits of the Data Memory address (DMAAddr[15:8]) are equal to 0.

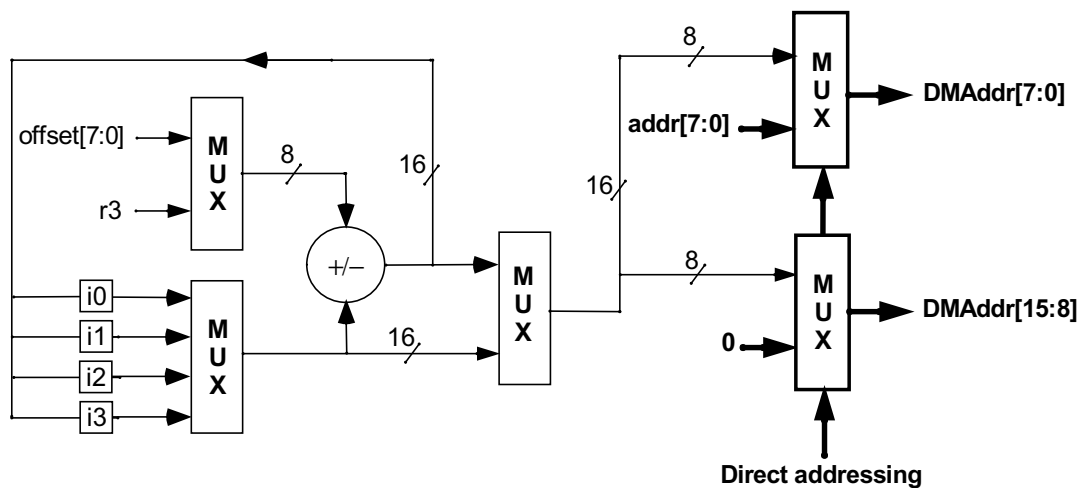


Figure 2.4: Direct addressing

### 2.2.5.2 Indexed addressing

*Data Memory access : DM(ix)*

The complete Data Memory space can be addressed with this mode. The value of the index *ix* is the Data Memory address (DMAAddr[15:0]). The 8 LSB bits (*ixl*) define the offset in the page while the 8 MSB bits (*ixh*) define the page number. Therefore 256 pages of 256 bytes can be addressed.

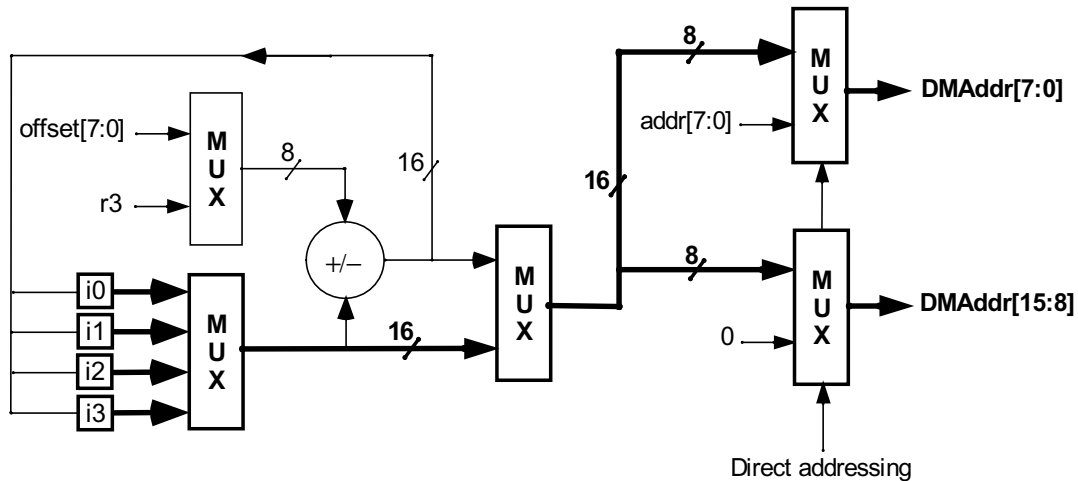


Figure 2.5: Indexed addressing

2.2.5.3 Indexed addressing with an immediate offset

Data Memory access :  $DM(ix+offset:8)$

The 16-bit Data Memory address **DMAddr[15:0]** is calculated by the addition of an 8-bit positive **offset:8** taken in the instruction to one of the 16-bit Index **ix**.

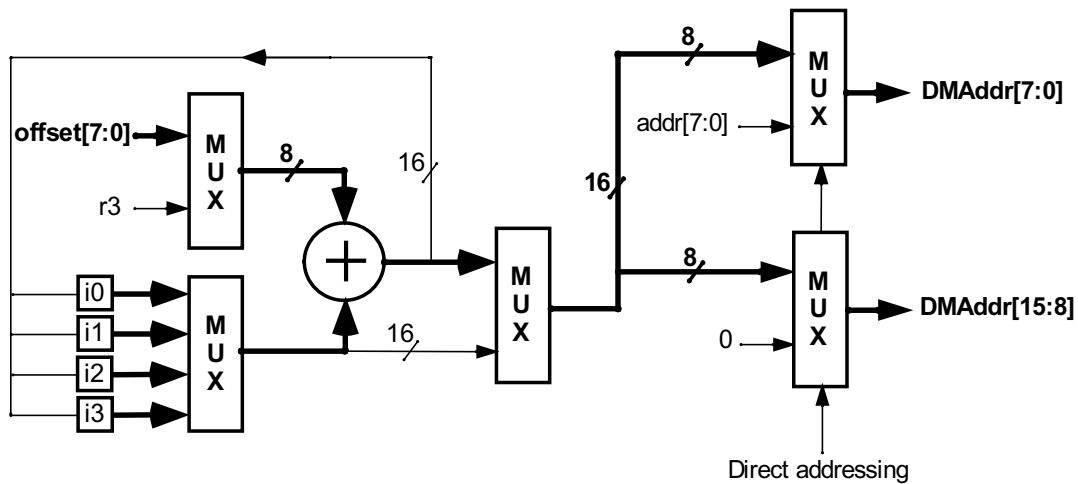


Figure 2.6: Indexed addressing with an immediate offset

2.2.5.4 Indexed addressing with a register offset

Data Memory access :  $DM(ix+r3)$

The 16-bit Data Memory address **DMAddr[15:0]** is calculated by the addition of the 8-bit positive value of the **r3** register to one of the 16-bit Index **ix**.

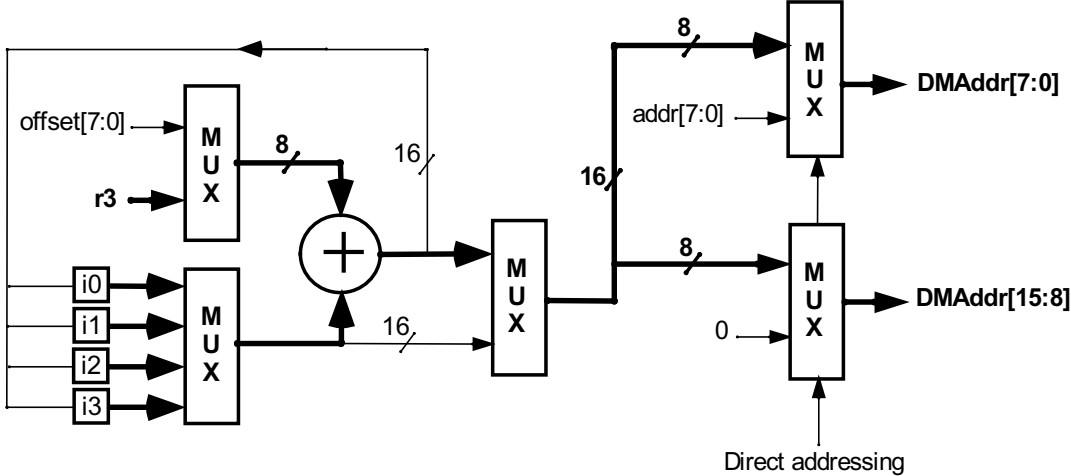


Figure 2.7: Indexed addressing with a register offset

2.2.5.5 Indexed addressing with Post-modification of the Index

Data Memory access :  $DM(ix)$

Index Update :  $ix \leftarrow ix + offset:7$   
Particular case :  $offset:7 = 1$

The 16-bit Data Memory address **DMAddr[15:0]** is given directly by the value of one of the 16-bit Index **ix**. The 16-bit Index **ix** is updated by the addition of the 7-bit positive **offset:7** field in the corresponding instruction.

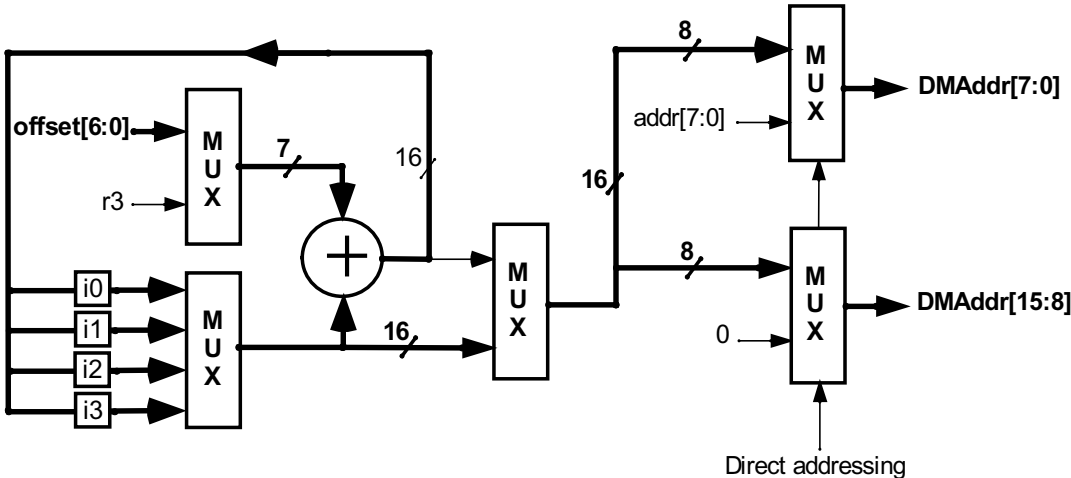


Figure 2.8: Indexed addressing with post-modification of the Index

### 2.2.5.6 Indexed addressing with Pre-modification of the Index

Data Memory access :  $DM(ix-offset:7)$

Index Update :  $ix \leftarrow ix - offset:7$

Particular case :  $offset:7 = 1$

The 16-bit Data Memory address **DMAddr[15:0]** is calculated by the subtraction of the 7-bit positive **offset:7** from the 16-bit Index **ix**. The 16-bit Index **ix** is updated by the subtraction of the 7-bit positive **offset:7** field in the corresponding instruction.

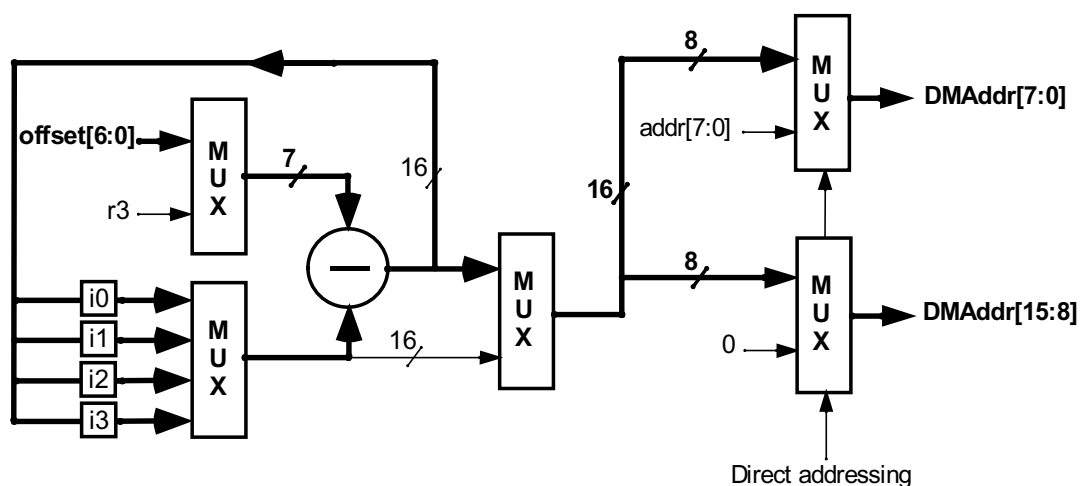


Figure 2.9: Indexed addressing with pre-modification of the index

### 2.2.5.7 Remark on the Indexed addressing

In an Indexed Data Memory access, the Index used for the access may also be used as the destination register for the operation.

In the case of Pre/Post Index modification, the Index **ix** is updated *before* the result of the operation is stored in **ixl** or **ixh**.

### 2.2.6 Flags Z, C & V

The flag **Z** (zero) is modified by all ALU operations (including the **MOVE** into a register).

**Z** = 1 only if the ALU Output (*not the multiplier output*) is equal to 0.

The flags **C** (carry) and **V** (overflow) are only modified by arithmetic, comparison and shift operations.

In shift operations, **C** always contains the “shifted out” bit.

In arithmetic operations with unsigned numbers, **C** indicates whether an overflow or an underflow occurs (can be active either high or low depending on the operation).

In arithmetic and shift operations with signed numbers, **V** indicates whether an overflow or an underflow occurs.

**V** = 1 when overflow (or underflow).

After the comparison instructions “CMP(A) d, s” :

$$C = 0 \text{ if } d > s \quad \text{and} \quad V = C * NOT(Z)$$

### 2.2.7 ALU output register: a

The ALU Output Register (Figure 2.1) named **a** always contains the result of the last ALU operation. It is a temporary register that is *always* modified by ALU operations.

It is addressed as a normal register in the register bank. It should be used for temporary results, *as power-consumption is saved* if this low-power accumulator is used instead of another data register.

### 2.2.8 Program counter

The 16-bit program counter (PC) can address a Program Memory with up to 64K instructions. A hardware stack is provided for efficient subroutine and Interrupt support. The stack depth is designer dependant and can be chosen between 2 and 9. A minimum stack depth of 2 is necessary for Interrupt support as well as for the CALL instruction support. Each additional subroutine or Interrupt stack level has a cost of 500 transistors.

If Interrupts are not used, one more nested hardware CALL is possible with the same hardware.

When the hardware stack is full, Interrupt is disabled until one level of the stack becomes free again (RET or RETI).

Additional subroutine levels are supported with no hardware cost through the use of the Software Call mechanism (CALLS instruction).

### 2.2.9 Branch conditions

After a comparison instruction (**CMP**), 6 conditional branch instructions are possible depending on the comparison result. After the other ALU operations, 6 conditional branch instructions are possible depending on the value of the flags. The **JEV** branch instruction is executed if one (or more) of the Event bits of the Status register is active (equal to 1).

The carry (**C**), the overflow (**V**) and the zero (**Z**) flags result from the "previous" ALU operation (which modified the flags!).

Table 2.10 summarises the different Branch conditions available.

Branch	Test
Branch on ALU result	
JCS	C = 1
JCC	C = 0
JVS	V = 1
JVC	V = 0
JZS	Z = 1
JZC	Z = 0
Branch on Event	
JEV	(EV0 OR EV1) = 1
Branch on CMP result	
JEQ	d = s
JNE	d <> s
JGT	d > s
JGE	d >= s

**Table 2.10: Branch Conditions**

Branch	Test
JLT	$d < s$
JLE	$d \leq s$

**Table 2.10: Branch Conditions**

The Branch **JEV** is executed if one (or more) of the Event bits of the Status register is active (equal to 1).

### 2.2.10 Call, Branch and Link

In most RISC microprocessors, only a Branch & Link mechanism is available. This mechanism saves the return address in a register. The programmer is responsible to save this address in the Data Memory before a new Software Call is used.

CoolRISC provides both Hardware and Software Call mechanisms. The Hardware Call requires  $N > 1$  PC stack level. The Software Call stores the return address in a particular register which is the two-byte Program Memory Index **ip**. Therefore, the programmer has to save **ip** (if it is used) before a Software Call (**CALLS**).

The designer has to trade-off the number of hardware PCs and the number of supplementary instructions executed to save/restore the Program Memory Index used as Branch & Link register.

### 2.2.11 Events and Interrupts

The 8-bit Status register (**stat**) contains the following booleans:

bit0:	Event nb 0 ( <b>EV0</b> )
bit1:	Event nb 1 ( <b>EV1</b> )
bit2:	Interrupt nb 0 ( <b>IN0</b> )
bit3:	Interrupt nb 1 ( <b>IN1</b> )
bit4:	Interrupt nb 2 ( <b>IN2</b> )
bit5:	Enable bit for all Interrupts ( <b>GIE</b> )
bit6:	Enable bit for <b>IN1</b> ( <b>IE1</b> )
bit7:	Enable bit for <b>IN2</b> ( <b>IE2</b> )

Events and Interrupts are Boolean flags which can be either hardware or software modified. A negative pulse on one of the **nEvent[1:0]** or **nInterrupt[2:0]** pins will set the corresponding bit to 1. In addition, a write to the Status register can either set or reset any of these bits. Therefore Interrupts and Events can be forced by software (see next chapter: “Pipeline Exception”). Note however that **a** is modified when “Software Interrupt” is generated (by an ALU operation).

Interrupts force a **CALL** to a fixed address (one specific address for each interrupt), save the Program counter (PC) on the stack (which will be restored by the **RETI** or **RET** instruction), and start the processor if it is in the HALT mode. The designer must save the accumulator **a**, the flag **C** and the working registers (which are used in the Interrupt routine) at the beginning of the Interrupt routine, and restore them at the end as follows :

```

MOVE eaddr1, a                ;save a & Z
SFLAG                        ;a ( C & V
MOVE eaddr2, a                ;save C & V
MOVE eaddr3, ri               ;save ri
.....
MOVE ri, eaddr3               ;restore ri
RFLAG eaddr2                  ;restore C & V
MOVE a, eaddr1                ;restore a & Z
    
```

A disabled Interrupt (corresponding Enable bit at 0) cannot force a **CALL**, and cannot **START** the CPU if it is in the HALT mode. However, the request is taken into account and will be executed as soon as the corresponding Enable bit is set to 1, unless the interrupt bit has been successfully cleared by software in the meantime.

The general Interrupt Enable bit **GIE** (**stat**[5]), if 0, disables any Interrupt with the same principle as above.

When a **CALL** to an Interrupt routine is executed, **GIE** is automatically cleared in order to prevent the **CPU** executing another **CALL** to the same (or another) Interrupt.

When the **RETI** instruction is executed, **GIE** is automatically set to 1 in order to allow Interrupts to occur again. In order to return from an Interrupt, **RET** must be used instead of **RETI** if the programmer does not want to change the value of **GIE**.

The programmer may allow nested Interrupts by setting **GIE** to 1. But each time a new **CALL** to an Interrupt is executed, a new level of the hardware stack is used.

When the *hardware stack is full*, *Interrupts are disabled* independently of the value of **GIE**. As soon as a level of the stack is freed (**RET**, **RETI**), a pending Interrupt can be executed.

An action on the **nEvent**[1:0] pins restarts the processor if it is in the **HALT** mode. In contrast to the Interrupt, an Event does not force a call to a predefined address. It should be used as a handshake facility.

The **HALT** instruction is only effective if all Event bits and all non-masked Interrupt bits are cleared.

The **nEvent** and **nInterrupt** lines are active low, but a "short" negative pulse is sufficient to set the Event or Interrupt bit in the status register.

Clearing an Event or an Interrupt bit is only possible if the corresponding input line is not active. This allows the execution of an Interrupt routine as long as the corresponding input line is active.

In the Interrupt routine, it is recommended first to deactivate the Interrupt line (by commanding the corresponding peripheral to release the line), and then to clear the corresponding Interrupt bit. Thus, it will be possible to receive a new Interrupt (on the same input) as soon as the Interrupt bit is cleared.

If several Interrupts are pending, they are executed in order of priority.

Only **GIE** is reset by a hardware Reset. The other booleans must be initialised by the programmer.

Table 2.11 shows the call addresses and the priorities of the Interrupts.

Inter. nb.	CALL ad.	Priority
Inter. nb 0	3	Highest
Inter. nb 1	1	Medium
Inter. nb 2	2	Lowest

**Table 2.11: CALL addresses and priorities**

### 2.2.12 Pipeline exception

If an interrupt bit is set by the software (write into **stat**) the pipeline causes the next instruction to be executed "before" the CPU executes the interrupt routine. This allows the supply of a parameter to a "trap" as follows :

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

If an Event bit is set by software, and a "JUMP on Event" (**JEV**) is the next instruction, *the first instruction will be ignored by the second.*

These are the only delays caused by the CoolRISC® pipeline.

#### 2.2.13 HALT mode

The **HALT** instruction turns the processor into stand-by mode, in which power consumption is minimum. The clock is stopped at the entrance to the processor to prevent any transition in the core. Only an Event, an Interrupt or a hardware Reset are able to wake up the microprocessor.

The **HALT** instruction is only effective if all Event bits and all enabled Interrupt bits are inactive (low).

When the processor stops because of a **HALT** instruction, the previous instruction is totally executed before the stand-by mode occurs. The next instruction will only begin when the processor restarts.

#### 2.2.14 Hardware Reset

When the **nReset** signal goes low, the general Interrupt Mask bit **GIE** is reset, the CPU restarts if it was in the HALT mode, the frequency division factor is set to 1x, the PC stack is emptied, and the Test mode is reset as well as the Program Memory Dump mode.

Furthermore, at each rising edge of the external clock, a JUMP to address 0 is forced. The instruction located at address 0 will be executed when the **nReset** signal returns high.

#### 2.2.15 Low frequency modes

As explained in chapter 1.6, the processor internal frequency can be reduced by a factor of 2, 4, 8 or 16. The division factor is both hardware and software controlled.

The **FREQ** instruction sets the basic division factor which is output on the **FreqOut[3:0]** bus. The instruction that follows **FREQ** is already executed with the new processor frequency.

### 3 Programming the CoolRISC816 core

#### 3.1 Introduction

This chapter gives examples on instruction executions including limits and special cases.  
Next chapter contains the instruction codes, useful for the debugging of programs.

#### 3.2 Instruction Examples

INSTRUCTION	PARAMETERS BEFORE EXECUTION OF THE INSTRUCTION					PARAMETERS AFTER EXECUTION OF THE INSTRUCTION					C	V	Z	a
JUMP 0x0A54	PC0 = 0x43E7				c	PC0 = 0x0A54					-	-	-	-
JUMP ip	PC0 = 0x43E7	ip = 0x0A54	iph = 0x0A	ipl = 0x54	c	PC0 = 0x0A54					-	-	-	-
JCS(JLE) ip	PC0 = 0x43E7	ip = 0x3A54	iph = 0x3A	(d (s)	1	PC0 = 0x3A54					-	-	-	-
JCS(JLE) ip	PC0 = 0x43E7	ip = 0x3A54	iph = 0x3A	(d (s)	0	PC0 = 0x43E8					-	-	-	-
JCC(JGT) ip	PC0 = 0x43E7	ip = 0x3A54	iph = 0x3A	(d > s)	0	PC0 = 0x3A54					-	-	-	-
JCC(JGT) ip	PC0 = 0x43E7	ip = 0x3A54	iph = 0x3A	(d > s)	1	PC0 = 0x43E8					-	-	-	-
JZS(JEQ) 0x1FE4	PC0 = 0x43E7		Z = 1	(d = s)	c	PC0 = 0x1FE4					-	-	-	-
JVS(JEQ) 0x1FE4	PC0 = 0x43E7		V = 0	(d = s)	c	PC0 = 0x43E8					-	-	-	-
JVC(JNE) 0x1FE4	PC0 = 0x43E7		V = 0	(d (s)	c	PC0 = 0x1FE4					-	-	-	-
JZC(JNE) 0x1FE4	PC0 = 0x43E7		Z = 1	(d (s)	c	PC0 = 0x43E8					-	-	-	-
JLT 0xF2E5	PC0 = 0x43E7		C*nZ = 1	(d < s)	c	PC0 = 0xF2E5					-	-	-	-
JLT 0xF2E5	PC0 = 0x43E7		C*nZ = 0	(d < s)	c	PC0 = 0x43E8					-	-	-	-
JGE 0xF2E5	PC0 = 0x43E7		C*nZ = 0	(d (s)	c	PC0 = 0xF2E5					-	-	-	-
JGE 0xF2E5	PC0 = 0x43E7		C*nZ = 1	(d (s)	c	PC0 = 0x43E8					-	-	-	-
JEV 0x0001	PC0 = 0xFFFF	stat[0] = 1	OR / AND	stat[1] = 1	c	PC0 = 0x0001					-	-	-	-
JEV 0x0001	PC0 = 0xFFFF	stat[0] = 0	AND	stat[1] = 0	c	PC0 = 0					-	-	-	-
CALL 0x0A54	PC0 = 0x43E7	PC1 = 0x5555	PC2 = 0x7777		c	PC0 = 0x0A54	PC1 = 0x43E8	PC2 = 0x5555	PC3 = 0x7777					
RET	PC1 = 0x43E8	PC2 = 0x5555	PC3 = 0x7777	PC4 = 0x9999	c	PC0 = 0x43E8	PC1 = 0x5555	PC2 = 0x7777	PC3 = 0x9999					
RETI	PC1 = 0x43E8	PC2 = 0x5555	PC3 = 0x7777	stat[5] = 0	c	PC0 = 0x43E8	PC1 = 0x5555	PC2 = 0x7777	stat[5] = 1					
CALLS 0x1FE4	PC0 = 0x43E7	ip = 0x0A54	iph = 0x0A	ipl = 0x54	c	PC0 = 0x1FE4	ip = 0x43E8	iph = 0x43	ipl = 0xE8					
CALLS ip	PC0 = 0x43E7	ip = 0x0A54	iph = 0x0A	ipl = 0x54	c	PC0 = 0x0A54	ip = 0x43E8	iph = 0x43	ipl = 0xE8					
RETS	PC0 = 0x43E7	ip = 0x0A54	iph = 0x0A	ipl = 0x54	c	PC0 = 0x0A54	ip = 0x0A54	iph = 0x0A	ipl = 0x54					
PUSH	PC0 = 0x43E7	ip = 0x0A54	iph = 0x0A	ipl = 0x54	c	PC0 = 0x43E8	ip = 0x0A54	iph = 0x0A	ipl = 0x54					
		PC1 = 0x3333	PC2 = 0x5555	PC3 = 0x7777			PC1 = 0x0A54	PC2 = 0x3333	PC3 = 0x5555	PC4 = 0x7777				
POP	PC0 = 0x43E7	ip = 0x2222	iph = 0x22	ipl = 0x22	c	PC0 = 0x43E8	ip = 0x0A54	iph = 0x0A	ipl = 0x54					
		PC1 = 0x0A54	PC2 = 0x3333	PC3 = 0x5555	PC4 = 0x7777		PC1 = 0x3333	PC2 = 0x5555	PC3 = 0x7777					
FREQ div8		CkuP = Ck			c		CkuP = Ck/8							
FREQ nodiv		CkuP = Ck/8			c		CkuP = Ck							

Table 3.1: Instruction examples

INSTRUCTION	PARAMETERS BEFORE EXECUTION OF THE INSTRUCTION					PARAMETERS AFTER EXECUTION OF THE INSTRUCTION					C	V	Z	a
					C									
SFLAG	a = 0b01010101			V = 0	1	a = 0b11101010								
SFLAG	a = 0b00110011			V = 1	1	a = 0b10011001								
SFLAG	a = 0b00111000			V = 0	0	a = 0b00011100								
SFLAG	a = 0b00000011			V = 1	0	a = 0b01000001								
RFLAG r0	r0 = 0b11000000				c	a = 0b10000000								
RFLAG 0x27	DM(27) =	0b10111111			c	a = 0b01111110								
RFLAG (i0)+	DM(i0) =	0b00111111			c	a = 0b01111110								
RFLAG -(i1)	DM(i1-1) =	0b01000000			c	a = 0b10000000								
MOVE stat, (13)					c	stat = a = 13					-	-	0	a
MOVE r0, iph	iph = 0				c	r0 = a = 0					-	-	1	a
MOVE r1, 0x67	DM(67) = 32				c	r1 = a = 32					-	-	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) = 0x55	i0 = 0xA2B6	i0h = 0xA2	i0l = 0xB6	c	i0l = a = 0x55	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) = 0x33	i2 = 0x00EE	i2h = 0x00	i2l = 0xEE	c	i2l = a = 0x33	i2 = 0x0133	i2h = 0x01	i2l = 0x33	-	-	0	a	
MOVE i3h, -(i3)	DM(03FF) = 0xA8	i3 = 0x0400	i3h = 0x04	i3l = 0x00	c	i3h = a = 0x48	i3 = 0x48FF	i3h = 0x48	i3l = 0xFF	-	-	0	a	
MOVE ipl, -(i4, 0x33)	DM(0) = 88	i4 = 0x0033	i4h = 0x00	i4l = 0x33	c	ipl = a = 88	i4 = 0x0000	i4h = 0x00	i4l = 0x00	-	-	0	a	
CMVD ipl, -(i4, 0x33)	DM(0) = 88	i4 = 0x0033	i4h = 0x00	i4l = 0x33	0	ipl = a = 88	i4 = 0x0000	i4h = 0x00	i4l = 0x00	-	-	0	a	
CMVD ipl, -(i4, 0x33)	DM(0) = 88	i4 = 0x0033	i4h = 0x00	i4l = 0x33	1	ipl = a = 88	i4 = 0x0000	i4h = 0x00	i4l = 0x00	-	-	0	a	
CMVS ipl, -(i4, 0x33)	DM(0) = 88	i4 = 0x0033	i4h = 0x00	i4l = 0x33	0	a = 88	i4 = 0x0000	i4h = 0x00	i4l = 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 = 155	i0 = 0x3302	i0h = 0x33	i0l = 0x02	c	DM(3401) = 155	i0 = 0x3302	i0h = 0x33	i0l = 0x02	-	-	-	-	
MOVE (i2, r3), r0	r0 = 47	r3 = 0xFF	i2h = 0x33	i2l = 0x02	c	DM(3401) = 47	i2 = 0x3302	i2h = 0x33	i2l = 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 = 18	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 -(i4, 0x46), i2l	i2l = 189	i4 = 0x0045	i4h = 0x00	i4l = 0x45	c	DM(FFFF) = 189	i4 = 0xFFFF	i4h = 0xFF	i4l = 0xFF	-	-	-	-	
SHL r1	r1 = 0b11000000				c	r1 = a = 0b10000000				1	0	0	0	a
SHL a, ipl	ipl = 0b01000011				c	a = 0b10000110				0	1	0	0	a
SHLC r0, i0l	i0l = 0b00111100				1	r0 = a = 0b01111001				0	0	0	0	a
SHLC r0	r0 = 0b10000011				0	r0 = a = 0b00000110				1	1	0	0	a
SHR a	a = 0b11000000				c	a = 0b01100000				0	0	0	0	a
SHR r1, iph	iph = 0b01000011				c	r1 = a = 0b00100001				1	0	0	0	a
SHRC r0	r0 = 0b00111100				1	r0 = a = 0b10011110				0	0	0	0	a
SHRC r1, a	a = 0b10000011				0	r1 = a = 0b01000001				1	0	0	0	a
SHRA a	a = 0b10000000				c	a = 0b11000000				0	0	0	0	a

Table 3.1: Instruction examples

INSTRUCTION	PARAMETERS BEFORE EXECUTION OF THE INSTRUCTION				PARAMETERS AFTER EXECUTION OF THE INSTRUCTION									
					C						C	V	Z	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 = 0b10101011					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 = 0xFS			c	r3 = 0xFF	a = 0xS0				u	u	u	a
MSHRA a, (4		a = 0xFS			c		a = 0xS0				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	r2 = 0xE3	a = 0x72				u	u	u	a
MULA r3, (0x10		r3 = 0xFS			c	r3 = 0xFF	a = 0xS0				u	u	u	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		r1 = 0x1			c	r1 = a = 0					-	-	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

**Table 3.1: Instruction examples**

INSTRUCTION	PARAMETERS BEFORE EXECUTION OF THE INSTRUCTION				PARAMETERS AFTER EXECUTION OF THE INSTRUCTION								
					C					C	V	Z	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
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

Table 3.1: Instruction examples



## 4 CoolRISC® 816 Instruction Codes

### 4.1 Instructions

#### 4.1.1 Branch

<b>1 1 0</b>	cc:3	not_addr:16
--------------	------	-------------

**JUMP** addr      PC0 <- addr  
**JCC** addr      if cc then PC0 <- addr

#### 4.1.2 Indexed Branch :

<b>1 0 0</b>	cc:3	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
--------------	------	---------------------------------

**JUMP** ip      PC0 <- ip  
**RETS**      PC0 <- ip (*soft. return*)  
**JCC** ip      if cc then PC0 <- ip

#### 4.1.3 Subroutine CALL :

<b>1 1 1 0 0 1</b>	not_addr:16
--------------------	-------------

**CALL** addr      PCn <- PCn-1(n>1), PC1 <- PC0 + 1, PC0 <- addr

#### 4.1.4 Indexed subroutine CALL :

<b>1 0 1 0 0 1 1 1 1</b>	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
--------------------------	---------------------------------

**CALL** ip      PCn <- PCn-1(n>1), PC1 <- PC0 + 1, PC0 <- ip

#### 4.1.5 Software subroutine CALL :

<b>1 1 1 0 1 0</b>	not_addr:16
--------------------	-------------

**CALLS** addr      ip <- PC0 + 1, PC0 <- addr

#### 4.1.6 Indexed software subroutine CALL :

<b>1 0 1 0 1 0 1 1 1</b>	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
--------------------------	---------------------------------

**CALLS** ip      ip <- PC0 + 1, PC0 <- ip

#### 4.1.7 RETURN from Subroutine

<b>1 1 1 1 1 1 0 0 1</b>	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
--------------------------	---------------------------------

**RET**      PCn-1 <- PCn(n>0)

**4.1.8 RETURN from Interrupt**

<b>1 1 1 1 1 1 0 0 0</b>	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
--------------------------	---------------------------------

**RETI** PCn-1<- Pcn(n>0), GIE <- 1

**4.1.9 Hardware Stack PUSH**

<b>1 0 1 1 0 1 1 1 1</b>	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
--------------------------	---------------------------------

**PUSH** PCn <- PCn-1(n>1), PC1 <- ip, PC0 <- PC0 + 1

**4.1.10 Hardware Stack POP**

<b>1 1 1 1 1 0 1 0 1</b>	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
--------------------------	---------------------------------

**POP** ip <- PC1, PCn-1<- PCn(n>1), PC0 <- PC0 + 1

**4.1.11 Alu Operations between a Register and an Immediate Data**

<b>0 0 1 1 1 0</b>	alu_op:4	reg:4	not_data:8
--------------------	----------	-------	------------

**ALU\_OP** reg, #data reg <- data alu\_op reg

**4.1.12 Alu Operations between registers :**

<b>0 0 1 1 0</b>	alu_op:5	op2:4	op1:4	res:4
------------------	----------	-------	-------	-------

*res, op1 & op2 = reg:4*

**ALU\_OP** res, op1, op2

res <- op1 alu\_op op2

**ALU\_OP** res, op1

res <- op1 alu\_op res (*op2=res*)

res <- alu\_op op1

*(unary alu\_op)*

**ALU\_OP** res

res <- alu\_op res (*op1=res*)

*(unary alu\_op)*

**4.1.13 Alu Operations between the Data Memory (DM) and Registers :**

<b>0 0 0 1 0</b>	alu_op:5	reg:4	not_addr:8
------------------	----------	-------	------------

*addr[15:8]=0*

**ALU\_OP** reg, addr

reg <- DM(addr[7:0]) alu\_op reg

reg <- alu\_op DM(addr[7:0])

*(unary alu\_op)*

**4.1.14 Indexed Alu Operation (between the DM and Registers) with an immediate offset :**

<b>0 1 1</b>	ix:2	alu_op:5	reg:4	offset:8
--------------	------	----------	-------	----------

*Positive offset*

**ALU\_OP** reg, (ix, offset)

reg <- DM(ix+offset) alu\_op reg

reg <- alu\_op DM(ix+offset)

*(unary alu\_op)*

**ALU\_OP** reg, (ix)

*with offset=0*

reg <- DM(ix) alu\_op reg

reg <- alu\_op DM(ix)

*(unary alu\_op)*



**4.1.21 Indexed Data Memory (DM) STORE with a register offset :**

<b>0 0 0 0 1 1 1 0</b>	ix:2	reg:4	<b>1 1 1 1 1 1 1 1</b>
------------------------	------	-------	------------------------

**MOVE** (ix, r3), reg      DM(ix+r3) <- reg (*r3=8-bit positive offset*)

**4.1.22 Frequency of the Microprocessor**

<b>0 0 1 0 1 1 1 0 1 1</b>	<b>1 1 1 1 1 1 1 1</b>	divn:4
----------------------------	------------------------	--------

**FREQ** divn      CkuP <- Ck/n, with n =1, 2, 4, 8 or 16

**4.1.23 Stand-by mode**

<b>0 0 1 0 1 1 1 1 0 1</b>	<b>1 1 1 1 1 1 1 1 1 1 1 1</b>
----------------------------	--------------------------------

**HALT**      CkuP <- 0

**4.1.24 Save the Flags**

<b>0 0 1 0 1 1 0 1 1 1</b>	<b>1 1 1 1 1 1 1 1 1 1 1 1</b>
----------------------------	--------------------------------

**SFLAG**      accu <- flags

**4.1.25 No Operation**

<b>1 1 1 1 1 1 1 1 1 1 1 1 1 1</b>	<b>1 1 1 1 1 1 1 1 1 1</b>
------------------------------------	----------------------------

**NOP**
**4.1.26 Program Memory Dump**

<b>0 0 1 0 1 1 1 1 1 0</b>	1 1 1	S	<b>1 1 1 1 1 1 1 1</b>
----------------------------	-------	---	------------------------

**PMD** #s      Starts the Program Memory Dump if S=1, ends it if S=0

**4.2 Instruction tables**

reg:4	code	Function	cc:3	Test	Code	ix:2	code
r0	1 1 1 0		JUMP	-	0 1 1	i0	00
r1	1 1 0 1		JCS	C = 1	1 0 0	i1	01
r2	1 1 0 0		JCC	C = 0	0 0 0	i2	10
r3	1 0 1 1	DM offset	JZS	Z = 1	1 1 0	i3	11
i0l	0 0 0 0	i0[7:0]	JZC	Z = 0	0 1 0		
i0h	0 0 0 1	i0[15:8]	JVS	V = 1	1 0 1		
i1l	0 0 1 0	i1[7:0]	JVC	V = 0	0 0 1	divn:4	code
i1h	0 0 1 1	i1[15:8]	JEV	EVENT	1 1 1	nodiv	0 0 0 0
i2l	0 1 0 0	i2[7:0]	After CMP(A) d, s :			div2	1 0 0 0
i2h	0 1 0 1	i2[15:8]	JEQ	d = s	1 1 0	div4	1 1 0 0

**Table 4.1: Modifier codes**

i3l	0 1 1 0	i3[7:0]	JNE	d <> s	0 1 0	div8	1 1 1 0
i3h	0 1 1 1	i3[15:8]	JGT	d > s	0 0 0	div16	1 1 1 1
ipl	1 0 0 0	ip[7:0]	JGE	d >= s	0 0 1		
iph	1 0 0 1	ip[15:8]	JLT	d < s	1 0 1		
stat	1 0 1 0	status	JLE	d <= s	1 0 0		
a	1 1 1 1	accu					

**Table 4.1: Modifier codes**

alu_op:5	code	alu_op:5	code	alu_op:4	code
MOVE	0 1 0 1 0	INC	1 0 0 0 1	MOVE	1 0 1 0
CMVD	1 0 0 1 0	INCC	1 0 1 0 1	AND	0 0 1 0
CMVS	1 0 0 1 1	DEC	1 1 0 1 1	OR	1 0 1 1
		DECC	1 1 1 1 1	XOR	1 0 0 0
SHL	1 1 0 1 0				
SHLC	1 1 1 1 0	ADD	0 1 1 0 0	ADD	1 1 1 0
SHR	1 0 1 1 0	ADDC	0 1 1 0 1	ADDC	1 1 0 1
SHRC	1 0 1 0 0	SUBD	0 0 1 0 0	SUBD	0 1 0 0
SHRA	1 0 0 0 0	SUBDC	0 0 1 0 1	SUBDC	0 1 0 1
		SUBS	0 0 0 1 1	SUBS	0 0 1 1
CPL1	1 1 0 0 0	SUBSC	0 0 1 1 1	SUBSC	0 1 1 1
CPL2	1 1 0 0 1				
CPL2C	1 1 1 0 0	CMP	0 0 0 0 1	CMP	0 0 0 1
		CMPA	0 0 0 0 0	CMPA	0 0 0 0
AND	0 0 0 1 0				
OR	0 1 0 1 1	MUL	0 1 1 1 0	MUL	1 1 1 0
XOR	0 1 0 0 0	MULA	0 0 1 1 0	MULA	0 1 1 0

**Table 4.2: Instruction codes**

Some assembler instructions are executed as immediate data alu operations :

NAME	code	alu_op	NAME	code	alu_op
TSTB	1 1 1 1	AND			
SETB	1 0 1 1	OR	MSHL	1 1 1 0	MUL
CLRB	0 0 1 0	AND	M SHR	1 1 1 0	MUL
INVB	1 0 0 0	XOR	M SHRA	0 1 1 0	MULA

**Table 4.3: Assembler instructions executed as immediate data alu operations**

**RFLAG** (restore flags) is equivalent to the **SHL** alu\_op, but **accu** is always the destination

© XEMICS, 2001

All rights are reserved. Reproduction 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.