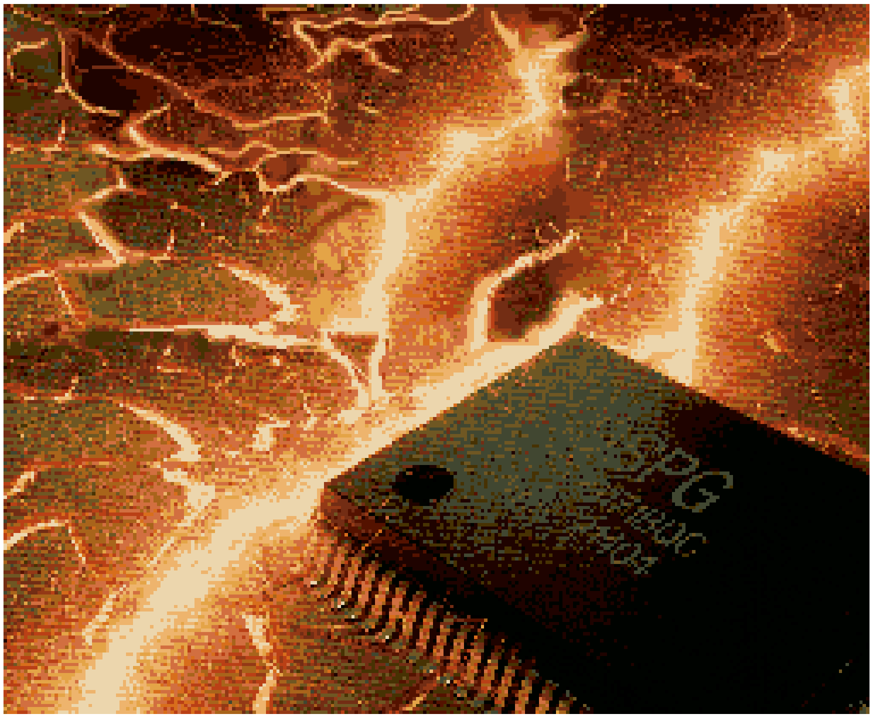
TeakLite®

ARCHITECTURE SPECIFICATION

REVISION 4.41



DSP Group Inc. - DSP Group Ltd.
- CONFIDENTIAL -

This document contains proprietary material of DSP Group, Inc. and DSP Group Ltd. for which there are patents pending. Specifications and information herein are subject to change without notice

TRADEMARKS

PineDSPCore®, OakDSPCore®, TeakDSPCore®, TeakLite® and OCEM® are registered trademarks of DSP Group Inc. and DSP Group Ltd.

SmartCores™ is a trademark of DSP Group Inc. and DSP Group Ltd.

All other trademarks mentioned in this document are property of their respective owners.

Table Of Contents

BLANK PAGE

1. INTRODUCTION

1.1 OVERVIEW

TeakLite® is a 16-bit (data and program) high performance fixed-point DSP core, based on a previous generation of architecture named OakDSPCore®. TeakLite is the first member of the TeakDSPCore® family. It is designed for mid to high-end telecommunications and consumer electronics applications, where low power and portability are major requirements. Among the applications supported by TeakLite are digital cellular telephones (like PDC, GSM, CDMA and TDMA), fast modems advanced facsimile machines, etc.

TeakLite is designed to achieve the best cost-performance factor for a given (small) silicon area. Taken into consideration were ALL elements of 'system-on-chip' requirements, like program size, data memory size, glue logic, power management, etc.

Based on the proven philosophy of previous generations of core technology such as PineDSPCore and OakDSPCore - TeakLite is also designed to be used as an engine for DSP-based application specific ICs. Several aspects of modularity, in RAM, ROM, and I/O, allowing efficient DSP-based ASIC development characterize it. The core consists of the main blocks of a high performance central processing units, including a full featured bit-manipulation unit, RAM and ROM addressing units, and Program control logic. All other peripheral blocks, which are application specific, are defined as part of the user specified logic, implemented around the TeakLite on the same silicon die.

TeakLite is specially aimed at ASIC design style. The core is fully synthesizable; i.e. it can by ported to various foundries and technologies in a full-automated way. The TeakLite core is designed in a single edge clocking design methodology which allows the use of full/partial scan testing methodology, as well as power consumption reduction.

TeakLite has an improved set of DSP and general microprocessor functions designed to meet the applications requirements. Its programming model and instruction-set are aimed at straightforward generation of efficient and compact code. TeakLite has an enhanced instruction set which is upward compatible with the PineDSPCore® instruction set.

TeakLite has power save mechanism to reduce its current consumption, this includes clock shut down during wait state/local data busses, selective module operation and more

TeakLite is available as a synthesizable core for standard cell library, to be utilized as a part of the user’s custom chip design.

1.2 TeakLite OPTIMIZATION

The following is a partial list of functions and algorithms that TeakLite is optimized for. These functions and algorithms play a major role in the above mentioned applications:

- Codebook search - for speech coders based on CELP, VSELP, Multipulse, speech recognition, etc.

- Viterbi decoder - for channel coding in digital cellular, trellis coding for fax and modems

- Digital (complex) modulation/demodulation for de/modulation of fax/modems and digital cellular signals, etc.

- Digital bit-stream manipulation error correcting codes in digital cellular/cordless phone, TDMA, GSM and CDMA, based cordless telephones, etc.

- Adaptive filters - analysis filters for speech coders, lattice filters, equalizers for fax/modem and digital cellular phones, echo canceling, etc.

1.3 ARCHITECTURE HIGHLIGHTS

TeakLite consists of three parallel execution units. The CBU contains a Computation Unit (CU), a Bit Manipulation Unit (BMU), a Data Addressing Arithmetic Unit (DAAU) and a Program Control Unit (PCU). It has two blocks of data RAM/ROM for parallel feeding of the two inputs of the multiplier.

The CU has a 16 by 16 bit multiplier (which performs singed by signed, signed by unsigned or unsigned by unsigned multiplication) supporting single and double precision multiplication. The CU has also a 36-bit ALU and two 36-bit accumulators with access to the two additional accumulators of the BMU.

The BMU consists of a full 36-bit barrel shifter, bit-field operations (BFO) unit, including special hardware for exponent calculation, and two 36-bit accumulators with access to the two accumulators of the CU. Context switching (swapping) between the two set of accumulators is supported.

Powerful zero-overhead looping enables four levels of block repeat (*bkrep*), in addition to an interruptible single word repeat. The pipeline structure was improved in order to gain minimal cycle time. Index based addressing capability is available. Shadow for parts of the status registers and an alternative bank of registers for four of the DAAU registers are available to improve interrupt handling and subroutine nesting. An option for automatic context switching during interrupts is also included.

TeakLite features a software stack, residing in the data memory space. The software stack as well as the index-based addressing capability and the additional accumulators, also supports C-compiler implementation of the TeakLite.

TeakLite includes three maskable interrupts; a NMI interrupt and a Breakpoint interrupt (BI) for emulation support (shares the same interrupt vector of the TRAP).

The core is designed to interface with external memories and peripherals of different speeds, using wait-state mechanism. It supports the steal cycle and the burst modes of the DMA. It is also supports automatic boot procedure and the on-chip emulation module residing off-core.

1.4 DOCUMENT ORGANIZATION

The key features of TeakLite are described in Chapter 2. The core block diagram and detailed descriptions of each block are given in Chapter 3. The TeakLite instruction set and its coding are explained in Chapter 4. Chapter 5 describes the interface to the TeakLite core, including details on reset and power save modes, exception handling, the core pins and provides detailed information on the built and mechanisms that are available for a TeakLite based application specific IC

BLANK PAGE

.

2. KEY FEATURES

2.1 TECHNOLOGY FEATURES

- Fully synthesizable design to any process

- Performance (at typical process):

  - 140Mhz cycle time @ 0.18µ, worst-case conditions
  - 225Mhz cycle time @ 0.18µ, typical conditions

- Low-power dissipation (excluding I/O):

  - 0.14mA/MHz @ 1.8V
  - Slow mode – clock speed and current consumption are reduced linearly, relative to Active Mode by user-defined factor
  - 50nA STOP mode (during “clock shut off”)

- Fully static design.

2.2 ARCHITECTURE FEATURES

- 16-bit fixed-point DSP CORE with high level of modularity:

  - Expandable program ROM and/or RAM
  - Expandable data RAM and/or ROM
  - User-defined registers

- 16 x 16 bit 2's complement parallel multiplier with 32-bit product. Multiplication of signed by signed, signed by unsigned, and unsigned by unsigned

- Single cycle multiply-accumulate instructions

- 36-bit ALU

- 36-bit left/right Barrel Shifter

- Four 36-bit accumulators

- Memory organization:

  - 64K x 16-bit data memory space

  - 64K x 16-bit program memory space
  - On-chip / Off-chip program memory (ROM/RAM)
  - On-Chip / Off-chip data memory (ROM/RAM)
  - Synchronous and/or Asynchronous memory interface, using standard memories
  - Data space is divided into configurable X, Y and Z spaces
  - Data space can be viewed as a single continuous space or as two banks of data memory.

- (4 + 2) x 16-bit general purpose pointer-registers with two dedicated Address Arithmetic Units for data memory (RAM/ROM) and program indirect addressing, circular buffering, and loop counters

- Alternative registers bank for three of the DAAU pointers (and a single configuration register) with individual selectable bank exchanging

- Software stack (with a stack pointer) residing in the data RAM

- Index-based addressing capability

- Option for up to 4 user-defined off core registers

- Automatic context switching by interrupts (with enable/disable feature for each interrupt) using Shadow registers for parts of status registers and swapping between two 36-bit accumulators

- All general and most special purpose registers, arranged as a global register set of 34 registers that can be referenced in most data moves and core operations

- Bit-Field (up to 16 bits) Operations (BFO): Set, Reset and Change Test. These operations are executed directly on registers and data memory content, with no affect on accumulators' content

- Single cycle exponent evaluation of up to 36-bit values

- Enables full normalization operation in two cycles

- Double precision multiplication support

- Max/Min single cycle instruction with pointer latching and modification. Optimized for codebook search and Viterbi decoding

- Single cycle Division step support

- Single cycle data move & shift capability

- Arithmetic and logic shifting capability, according to a shift value stored in a special register, or embedded in the instruction opcode. Conditional shift is also available, as well as rotate left and right operations

- The product register is transferred to the accumulator without scaling, or after scaling by shifting the product 1 bit to the left, 2 bits to the left or 1 bit to the right

- Add/Subtract/Compare of a long immediate value with registers or data memory, with no affect on the accumulator content

- Powerful swapping (14 options) between two sets of accumulators

- Automatic saturation mode on overflow while reading the accumulators' content, or using a special instruction

- Zero-overhead looping by two interruptible mechanisms: Single word Repeat and Block Repeat with four levels of nesting
  - Up to 64 repetitions
  - Instruction to break out of block-repeat

- Memory mapped I/O.

- Support for STOP and SLOW modes of operation for stopping the core or for reducing clock speed, respectively
- Interrupts and exceptions:
  - 1 Reset
  - 1 Non-Maskable interrupt
  - 3 Maskable interrupts
  - 1 TRAP/BI

- Support for code downloading from the data memory space into the program memory space

- Support for DMA

- Support for Program memory protection

- Binary compatibility with the with the OakDSPCore instruction set

- Upward compatibility with the PineDSPCore instruction-set

- On-chip emulation support

- Automatic boot procedure support

- Development/emulation with:
  - 20K words of data memory (10K of Xspace and 10K of Yspace)
  - 32K words of program memory
  - On-chip boot routine

- C-compiler support

3. BLOCK DIAGRAM AND PROGRAMMING MODEL

3.1 GENERAL DESCRIPTION

A high-level block diagram of the TeakLite architecture is shown in Figure 3.1. Block diagrams of the Computation and Bit-manipulation Unit and of the Data Address Arithmetic Unit, are shown in Figures 3.2 and 3.3, respectively.

The main components of the TeakLite are:

• Computation and Bit-manipulation Unit	- CBU
- Accumulators	- a0, a1, b0, b1
- Saturation logic	
- Arithmetic and Logic Unit	- ALU
- Bit Field Operations	- BFO
- Shift Value register	- sv
- Barrel Shifter	
- Exponent logic	- EXP
- Multiplier	
- Input registers	- x, y
- Output register	- p
- Output shifter	
• Data Address Arithmetic Unit	- DAAU
- DAAU registers	- r0 ÷ r3, r4 ÷ r5
- DAAU configuration registers	- cfgi, cfgj
- Software stack pointer	- sp
- Base register	- rb
- Alternative bank of registers	- r0b, r1b, r4b, cfgib
• Minimal/maximal pointer register	- mixp
• Data Buses	- XDB, YDB, PDB, ZDB
• Address Buses	- XAB, YAB, PAB, ZAB

Program Control Unit	- PCU
- Loop Counter	- lc
- Internal Repeat Counter	- repc
• Memories	
- Program memory	
- Data memory	- Xspace, Yspace, Zspace
• Control Registers	
- Status Registers	- st0, st1, st2
- Interrupt context switching	
- Registers (shadow & swap)	
- Internal Configuration register	- icr
- Data Value Match register	- dvm
• User-Defined Registers (off-core)	- ext0 ÷ ext3
• input/output	

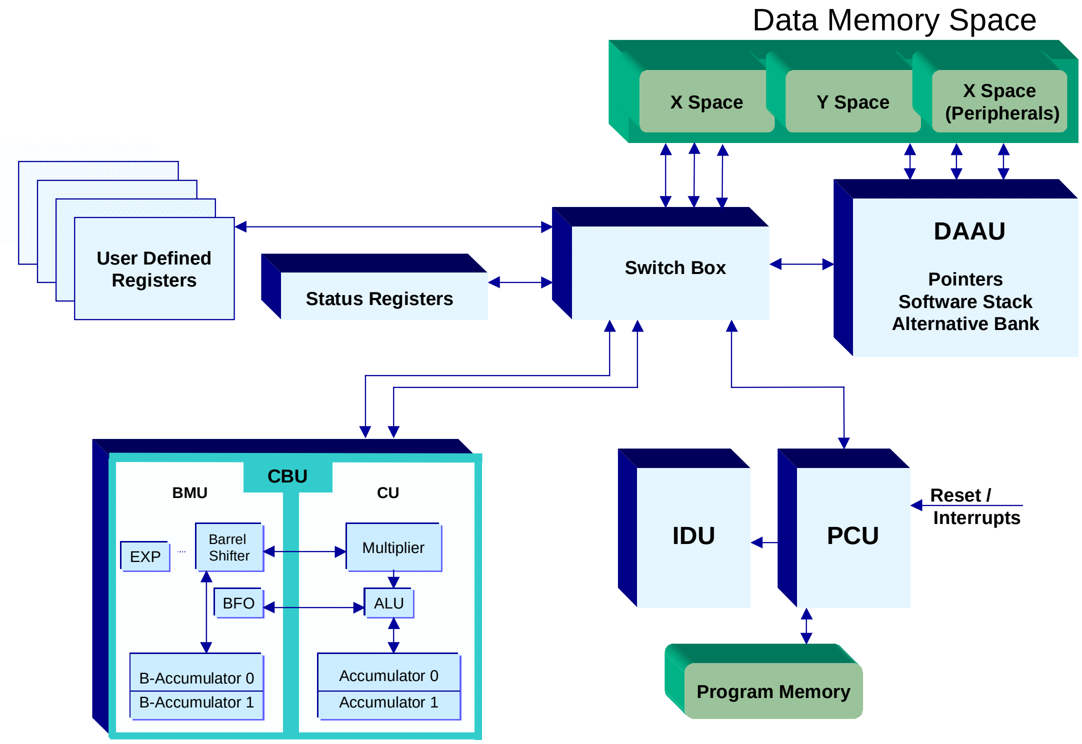


Figure 3.1 TeakLite - Block Diagram

3.2 BUSES

3.2.1 Data Buses

Data is transferred on the Xspace Data Bus (XDB), Yspace Data Bus (YDB), Zspace Data Bus (ZDB) and on the Program Data Bus (PDB).

Both the XDB and the ZDB are implemented as two 16-bit uni-directional buses each, read and write buses. These buses are architecturally connected to all the TeakLite. The XDB is the main data bus where most of the data transfers occur. It is recommended to connect the data memory arrays to the XDB/YDB and memory mapped I/O devices to the ZDB bus.

The YDB is a 16-bit uni-directional bus used for multiplication of instructions, where two data memory locations are read simultaneously, in the same instruction. The bus connects the memory with the input multiplier register (y register).

The PDB is a 16-bit bi-directional bus. Instruction words fetches take place in parallel over PDB.
The bus structure supports register to register, register to memory to register, program to data, program to register and data to program data movements. It is capable of transferring up to 16-bit words within one instruction cycle (m bit for multiplication where two operands read simultaneously).

3.2.2 Address Buses

Addresses are specified for the data memory space on the 16-bit unidirectional buses: Xspace Address Bus (XAB), Yspace Address Bus (YAB) and Zspace Address Bus (ZAB).

Program memory addresses are specified on the 16-bit unidirectional Program Address Bus (PAB).

For memory structure and mapping, refer to section 3.6, Memory Spaces and Organization.

3.3 COMPUTATION AND BIT-MANIPULATION UNIT

The Computation and Bit-manipulation Unit (CBU) contains two main units, the Computation Unit (CU) and the Bit-Manipulation Unit (BMU). Another unit is the saturation unit, which is shared by the CU and the BMU units. A detailed block diagram of the Computation and Bit-manipulation Unit is shown in Figure 3.2.

3.3.1 Computation Unit

The Computation Unit (CU) consists of the Multiplier units, the ALU, and two 36-bit accumulator registers (a0 and a1), as shown in Figure 3.2.

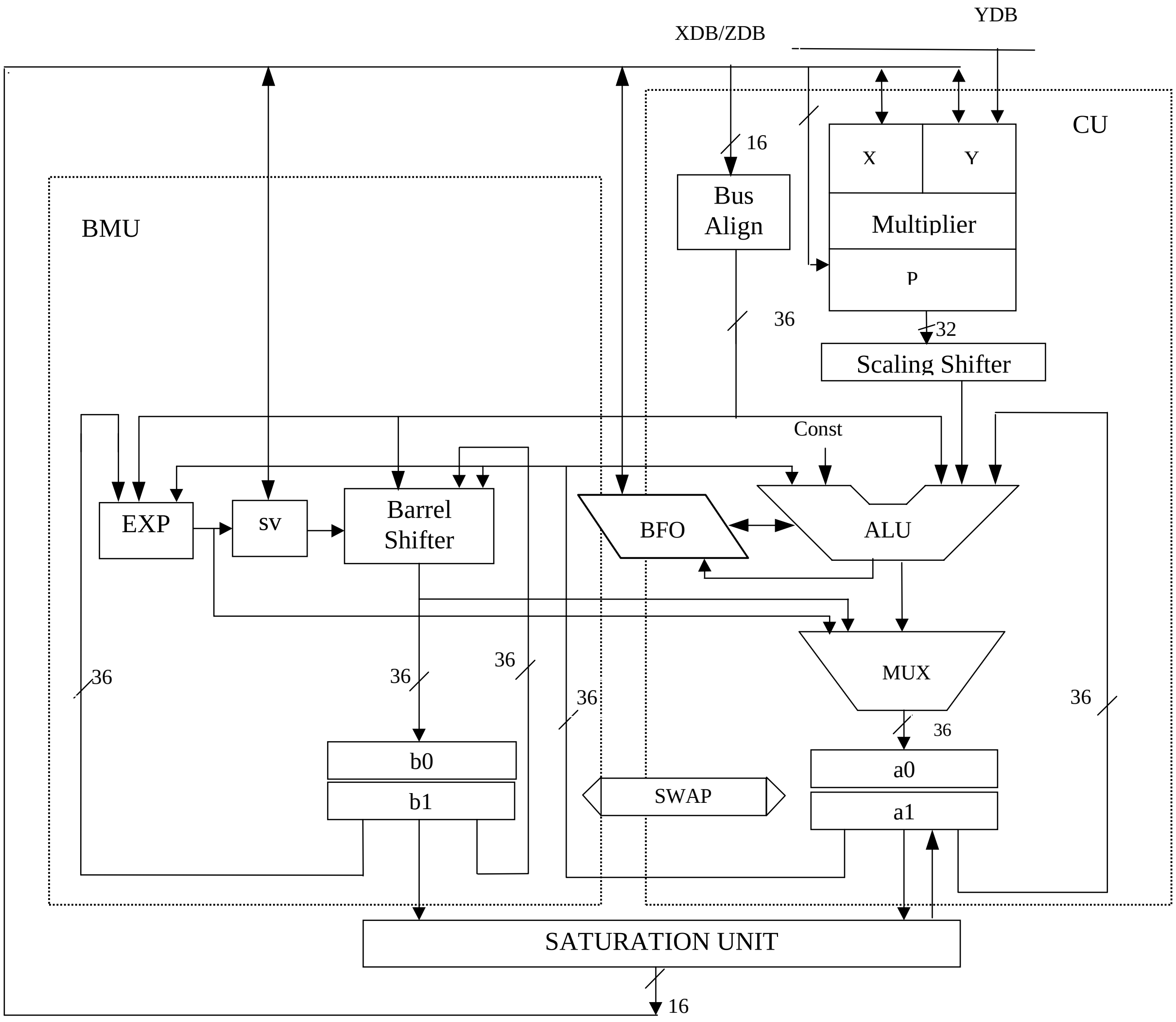


Figure 3.2 Computation and Bit-manipulation Unit Block Diagram

3.3.1.1 Multiplier

The multiplier unit consists of a 16 by 16 to 32 bit parallel 2's complement single-cycle, non-pipeline multiplier, two 16-bit input registers (x and y), a 32-bit output register (p), and a product output shifter. The multiplier performs signed by signed, signed by unsigned, or unsigned by unsigned multiplication. Together with the ALU, the TeakLite can perform a single-cycle multiply-accumulate (*mac*) instruction. The p register is updated only after a multiply instructions and not after a change in the input registers.

The x and y registers are read or written via the XDB/ZDB, while the y register is written via the YDB, as 16-bit operands. The 16-bit Most Significant Portion (MSP) of the p register, ph, can be written by the XDB as an operand. This enables a single-cycle restore of Ph during interrupt service routine. The complete 32-bit p register, sign-extended into 36 bits and shifted by the output shifter, can be used only by the ALU and can be moved only to the a0 and a1 accumulators.

The y and x registers can be also used as general-purpose temporary data registers.

3.3.1.1.1 Product Output Shifter

The p register is sign-extended into 36 bits and then shifted. In addition to pass the data UN-shifted, the output shifter is capable of shifting the data from the p register into the ALU unit by one bit to the right or by one or two bits to the left. In right shift, the sign is extended, whereas in left shift, a zero is appended to the LSBs. Shift operations are controlled by two bits (PS) in status register st1. The shifter also includes an alignment (a 16-bit right shift), for the support of the double precision multiplication (see paragraph 3.3.1.1.2 below).

3.3.1.1.2 Double Precision Multiplication

The TeakLite supports the double precision multiplication by several multiplication instructions and by an alignment option of the p register. The p register can be aligned (shifting 16 bits to the right) before accumulating the partial multiplication result, in multiply-accumulate instructions (*maa* and *maasu* instructions).
An example of different multiplication operations is in the multiplication of 32 bit by 16 bit fractional numbers. In this case two multiplications are required. Multiplication of the 16 bit number by the lower or upper portion of a 32 bit (double precision) number The signed by unsigned operation is used to multiply or multiply-accumulate the 16 bit signed number by the lower, unsigned portion of the 32 bit number. The signed by signed operation is used to multiply the 16 bit signed number by the upper signed portion of the 32 bit number.

While the signed by signed operation is executed, it is recommended to accumulate the *aligned* (using *maa* instruction) result of the previous signed by unsigned operation. For the multiplication of two double precision (32 bit) numbers, the unsigned by signed operation can be used. If this operation requires a 64 bit result, the unsigned by unsigned operation should be used. For details regarding various multiplication instructions *(mpy, mpysu, mac, macsu, macus, macuu, maa, maasu* and *msu, mpyi*), refer to Chapter 4 on the Instruction Set.

3.3.1.2 aX-Accumulators

Each aX-accumulator is organized as two regular 16-bit registers (a0h, a0l, a1h and a1l) and a 4-bit extension nibble (a0e and a1e). The two portions of each accumulator can be accessed as any other 16-bit data register and can be used as 16-bit source or destination registers in all relevant instructions. The aX-accumulators can serve as the source operand, as well as the destination operand, of the ALU, Barrel Shifter and Exponent units. The extension nibbles of the a0 and a1 accumulators are the MSBs part of status registers st0 and st1 respectively.
The aX-accumulators can be swapped with the bX-accumulators in a single-cycle. Saturation arithmetic is employed to selectively limit overflow from the high portion of an accumulator to the extension bits, when performing a move instruction from one of the accumulators through the XDB/ZDB, or when using the *lim* instruction, which performs saturation on the 36-bit accumulator. For further details regarding saturation, see Paragraph 3.3.3.

Registers aXh and aXl can also be used as general-purpose temporary 16-bit data registers.

3.3.1.2.1 Extension Nibbles

Extension nibbles a0e and a1e offer protection against 32-bit overflows. When the result of an ALU output crosses bit 31, it sets the extension flag (E) in st0, representing crosses of the MSB in aXh. Up to 15 overflows or under flows is possible using the extension nibble. At this point the sign is lost beyond the MSB of the ALU output and/or beyond the MSB of the extension nibble, setting the overflow flag (V) in st0, and also latching the Limit flag (L) in st0. For further details regarding the Status Registers, refer to Paragraph 3.7.2.

3.3.1.2.2 Sign Extension

Sign extension of the 36-bit aX-accumulators is applied when the aX or aXh is written with a smaller size operand.

This occurs when these registers are written from XDB/ZDB, from the ALU or from the Exponent unit in certain CBU operations. Sign extension can be suppressed by specific instructions for details, for further details, refer to Chapter 4.

3.3.1.2.3 Loading of aX-Accumulators

aXl is cleared while loading data into aXh, and aXh is cleared while loading aXl. Loading a full 32-bit value is accomplished via the *addl* or *addh* instructions (see the Instruction Set in Chapter 4). The full 36-bit accumulators can also be loaded using the shift instructions, or with another 36-bit accumulator, in a single-cycle, using the *swap* instruction (see paragraph 3.3.4, Swapping the accumulators, and the Instruction Set in Chapter 4).

3.3.1.3 Arithmetic and Logic Unit

The Arithmetic and Logic Unit (ALU) performs all arithmetic and logic operations on data operands. It is a 36-bit, single-cycle, non-pipelined ALU unit.

The ALU receives one operand from aX (X=0,1), and another operand from either the output shifter of the multiplier, the XDB/ZDB (through bus alignment logic), or from aX. The source operands can be 8 16 or 36 bits. . A joint operation of two aX-accumulators is also a possibility. The source and destination aX-accumulator of an ALU instruction is always the same. The XDB input is used for the transfer of one the register content, an immediate operand, or the content of a data memory location, addressed in direct memory addressing mode, in indirect addressing mode, in index addressing mode or pointed to by the stack pointer, as a source operand. The ALU results are stored in one of the aX-accumulators, or transferred through the XDB/ZDB to one of the registers or to a data memory location. The later is used for addition, subtraction and compare operations between a 16-bit immediate operand and a data memory location or one of the registers, without effecting the accumulators, in two cycles. Refer to the read-modify-write instructions; *addv, subv, cmpv* instructions, in Chapter 4 on the Instruction Set. A Bit-Field Operation (BFO) unit is attached to the ALU and described in details in paragraph 3.3.2.4.

The ALU can perform positive or negative accumulation, addition, subtraction, comparison, logic, and several other operations, most of them in a single cycle. Two’s complement arithmetic’s is employed.

Unless otherwise specified, in all operations between an 8-bit or 16-bit operand and a 36 bit aX, the 16-bit operand will be regarded as the LSP of a 36-bit operand with sign extension for arithmetic operations and zero extension for logical operations.

The *addh*, *subh*, *addl* and *subl* instructions are used when this convention is not adequate in arithmetic operations (refer to these instructions in Chapter 4).

The flags are affected as a result of the ALU output, as well as a result of the BFO or the Barrel shifter operations. In most instructions where the ALU output is transferred to one of the aX-accumulators, the flags represent the aX-accumulator status.

3.3.1.3.1 Rounding

Rounding (by adding 0x8000 to the LSP of the accumulator) can be performed by special instructions, in a single cycle or simultaneously with other operations. Refer to *movr* and *moda* instructions in Chapter 4.

3.3.1.3.2 Division Step

A single cycle division step is supported. For further details refer to *divs* instruction in Chapter 4.

3.3.1.3.3 Logic Operations

The logic operations performed by the ALU are AND, OR and XOR. All logic operations are 36 bits wide. 16-bit operands are zero extended when used in logic operations. The source and destination aX-accumulator of these instructions is always the same. A joint operation of two aX-accumulators is also a possibility. For further details, refer to *and, or*, and *xor* instructions in Chapter 4.

Other logic operations are set, reset, change and test, performed on one of the registers or on the data memory contents. See section 3.3.2.4, Bit-Field Operations.

3.3.1.3.4 Maximum-Minimum Operation

The TeakLite has single cycle maximum/minimum operation involves pointer latching and modification. One of the aX-accumulators, defined in the instruction, holds the maximal value in a *max* instruction, or the minimal value in a *min* instruction. In one cycle the two accumulators are compared, and when a new maximal or minimal number is found, this value is copied to the above-defined accumulator. In the same instruction r0 register (one of the DAAU registers, see paragraph 3.4) can be used, e.g. as a buffer pointer. This register can be post-modified according to the specified mode in the instruction.

When the new maximal or minimal number is found, the r0 pointer is also latched into the 16-bit dedicated minimum/maximum pointer latching (mixp) register - one of the DAAU registers. The maximum operation can also be performed directly on a data memory location pointed at by the r0 register *(maxd* instruction), saving the maximal number in the defined aX-accumulator and latching the r0 value into mixp, in a single cycle. For further details, refer to *max*, *maxd* and *min* instructions in Chapter 4. Regarding the r0 and mixp registers, see section 3.4.

When finding the maximal/minimal value, a few buffer elements can have the same value. The accumulator will save the same value. However, the latched pointer depends on the condition used in the instruction. Upon finding the maximum value, the pointer of the first element, or the last element will be latched, using greater than (>), or equal or greater than (≥) conditions, respectively. Once the minimum value is found, the pointer of the first element or the last element will be latched, corresponding less than (<), or equal or less than (≤) conditions, respectively. All these cases are supported by the *max*, *maxd* and *min* instructions.

3.3.2 Bit-Manipulation Unit

The Bit-Manipulation Unit (BMU) consists of a full 36-bit Barrel Shifter, an EXPonent unit (EXP), a Bit-Field Operation unit (BFO), two 36-bit accumulator registers (b0 and b1), and a Shift Value (SV) register (see Figure 3.2)

3.3.2.1 Barrel Shifter

The Barrel Shifter performs arithmetic shift, logic shift, and rotate operations. It is a 36-bit left and right, single-cycle, non-pipeline Barrel Shifter

The Barrel Shifter receives the source operand from either one of the four accumulators (a0, a1, b0, and b1) or from the XDB/ZDB (through bus alignment logic). The XDB/ZDB input is used for the transfer of one of the registers' contents, or for data memory location, in direct or indirect memory addressing mode. The source operands may be 16 or 36 bits. The destination of the shifted value is one of the four accumulators. The number of shifts is determined by a constant embedded in the instruction opcode or by a value in the sv register.

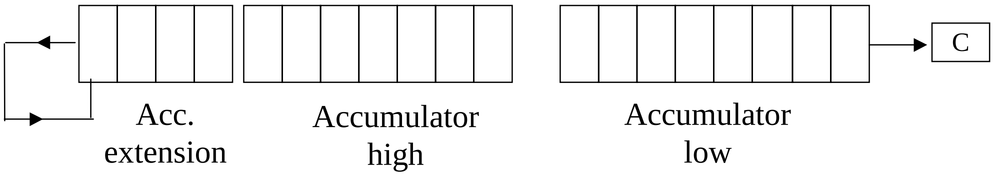
The flags are effected as a result of the Barrel Shifter output, as well as a result of the ALU and BFO outputs. When this result is transferred into one of the accumulators, the flags represent the accumulator status.

3.3.2.1.1 Shifting Operation

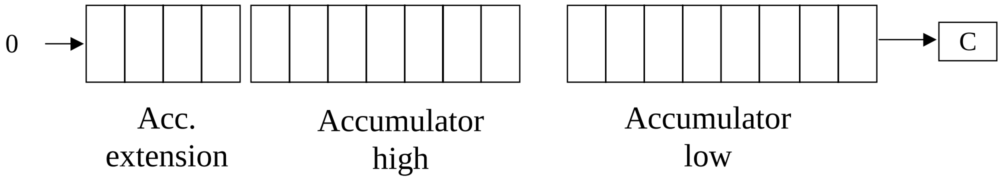
Using the Barrel Shifter opens several shifting options; all of them are implemented in a single cycle.
Each of the four accumulators can be shifted according to a 6-bit signed number, representing -32 ¸ +31 shifts (shift left is a positive number, while shift right is a negative number), embedded in the instruction opcode, into each of the four accumulators. The accumulators can also be shifted **conditionally**, according to the SV register content. In this case the accumulators can be shifted by -36 ¸ +36. This supports the calculation or the number of shifts at run-time, as for example in normalization operations (refer to paragraph 3.3.2.3, Normalization). The source and the destination accumulators can be the same or different. If the accumulators are different, the source accumulator is unaffected. For further details refer to *shfc* and *shfi* instructions in Chapter 4.
In addition to the conditional shifting operations performed when the accumulators are shifted according to the sv register (by -36 ¸+36), the shift and rotate operations included in the *moda* and *modb* instructions are also performed conditionally. *moda* and *modb* includes: 1-bit right and left arithmetic-shift and rotate, and 4-bit right and left arithmetic-shift. For further details refer to *moda* and *modb* instructions in Chapter 4.

Arithmetic left shift and logic left shift perform the same, filling the LSBs with zeros. During an arithmetic right shift the MSBs are sign extended, and during a logic right shift the MSBs are filled with zeros. The shift mode: arithmetic or logic is determined according to the Status mode bit (S) in the ST2 register. It affects all shift instructions.

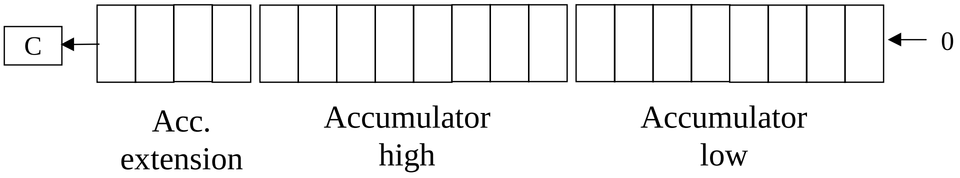
Arithmetic shift right :



Logic Shift right :



Arithmetic and logic shift left :



3.3.2.1.2 Move and Shift Operations

The four accumulators can be loaded in a single cycle, by a shifted value, according to the sv register content.. The accumulators can be loaded from one of the registers or from a data memory location, addressed in direct addressing mode, or in indirect addressing mode, according to the sv register shift value. The shifting capability is +36¸-36, (shift left is a positive number, while shift right is a negative number). The accumulators can also be loaded by one of the DAAU registers (rN registers), shifted according to a constant embedded in the instruction opcode. The shifting capability in this case is 15 to -16. The shift mode, either arithmetic or logic, is determined according to the Status mode bit (S), bit 7 in the st2 register. Refer to *movs* and *movsi* instructions in Chapter 4.

3.3.2.2 Exponent

The exponent unit (EXP) performs exponent evaluation of one of the four accumulators, of a data memory location, or of one of the registers. The result of this operation is a signed 6-bit value, sign-extended into 16 bits and transferred into the Shift Value register (sv). Optionally, it can also be transferred, sign-extended into 36 bits, into one of the aX-accumulators. The source operand is unaffected by this calculation. The source operand can be 36 bits when it is one of the accumulators, or 16 bits when it is a data memory location or one of the registers.

The algorithm for determining the exponent result for a 36-bit number is as follows:
Let N be the number of the sign bits (i.e. the number of MSBs equal to bit 35) found in the evaluated number. The exponent result is N - 5. This means that the exponent is evaluated with respect to bit 32. For a 16-bit operand, the 16 bits are regarded as bits 16 to 31, sign-extended into bits 32 to 35 and then treated as a 36-bit operand. Therefore, in this case the exponent result is always greater than or equal to zero. (See examples in Table 3-1 next page).

A negative result represents a number, for which the extension bits are not identical. The value of a negative result stands for the total of right shifts that should be performed in order to normalize the number, i.e. bit 31 (representing the sign) and bit 30 (representing part of

the magnitude) will be different. A positive result represents an un-normalized number, for which at least the 4 extension bits and the sign bit are the same.
When evaluating the exponent value of one of the accumulators, the positive number is the number of left shifts that should be carried out in order to normalize the source operand . An exponent result equal to zero represents a normalized number.

Examples (including an application in a normalization operation) :

Table 3-1 Exponent Evaluation Examples

Evaluated number **bits 35 31**	**N**	**Exponent result** **N-5**	**Normalized number** **bits 35 31**
0000 0000101...	8	3 (Shift left by 3)	0000 0101.........
1100 10101.......	2	-3 (Shift right by 3)	1111 10010101..
0000 0111000...	5	0 (No shift)	0000 0111000....

Full normalization can be achieved in 2 cycles, using the *exp* instruction, followed by a shift instruction. For further details refer to Section 3.3.2.3, , or to *exp, shfc* and *movs* instructions in Chapter 4.

The exponent unit can also be used for floating point calculations, where it is useful to transfer the exponent result into both the sv register and one of the aX-accumulators.

3.3.2.3 Normalization

Normalization is achieved by employing two techniques. Using the first technique, normalization can be accomplished in two cycles, using two instructions. The first instruction evaluates the exponent value (of one of the registers including the accumulators, or the value of a data memory location). The second instruction shifts the evaluated number, according to the exponent result stored at sv register. The second technique, compatible with PineDSPCore™, uses a normalization step, (*norm* instruction). For Further details refer to Chapter 4.

3.3.2.4 Bit Field Operations

The Bit Field Operation unit (BFO) is used for set, reset, change or test a set of up to 16 bits within a data memory location or within one of the registers. The data memory location can be addressed, using a direct memory or using an indirect address.

The *set, rst* and *chng* instructions are read-modify-write, they require two cycles and two words. The 16- bit immediate mask value is embedded in the instruction opcode. Various testing instructions can be used.
Either zeros testing (*tst0*) or ones (*tst1*) testing of up to 16 bits in a single cycle, can be achieved when the mask is in one of the aXl; or in two cycles when the mask value is embedded in the instruction opcode. Testing instruction (*tstb*) for a specific bit, 1 out of 16, in a data memory location, or in one of the registers in a single cycle, is also available. For further details refer to *set, rst, chng, tst0, tst1, tstb* instructions in Chapter 4.
The Bit-Field Operation unit (BFO) is attached to the ALU. Flags are affected as a result of the bit-field operations, as well as a result of the ALU and the Barrel Shifter operation.

3.3.2.5 bX-Accumulators

Each bX-accumulator is organized as two regular 16-bit registers (b0h, b0l, b1h and b1l) and a 4-bit extension nibble. The two portions of each accumulator can be accessed as 16-bit data registers through the XDB/ZDB bus, and can be used as 16-bit source or destination data registers in relevant instructions. The bX-accumulators can be swapped with the aX-accumulators in a single-cycle. Saturation arithmetic is employed to selectively limit overflow from the high portion of an accumulator to the extension bits, when performing a move instruction from one of the accumulators through the XDB/ZDB. For further details refer to Paragraph 3.3.3.

Each of the bX-accumulators can be a source operand of the Exponent unit, and can be a source operand or a destination operand of the Barrel Shifter.

3.3.2.5.1 Extension Nibbles

The extension nibbles of b0 and b1 offer protection against 32-bit overflows. When the output of the Barrel Shifter crosses bit 31, it sets the extension flag (E) in st0, representing crosses of the MSB at bXh. When the sign is lost beyond the MSB of the Barrel Shifter and/or beyond the MSB of the extension nibble, the overflow flag (V) in st0 is set, and also latched in the Limit flag (L) in st0. For further details refer to Paragraph 3.7.2., Status Registers. The extension bits can be accessed with the assistance of a single cycle shift instruction, or by swapping to the aX- accumulator.

3.3.2.5.2 Sign Extension

A sign extension of the 36-bit bX-accumulators is produced when the bX or bXh is written with a smaller size operand. This occurs when these registers are written from XDB/ZDB or from the Barrel Shifter in shift operations.

3.3.2.5.3 Loading of Bx-Accumulators

bxl is cleared while loading data into bXh, and bXh is cleared while loading bXl. The full 36-bit accumulator can be loaded, in a single-cycle, using the shift instructions or by another 36-bit accumulator, using the *swap* instruction. Refer to section 3.3.4, Swapping the Accumulators, and to Chapter 4).

3.3.2.6 sv Register

The Shift Value (sv) register is a 16-bit register used for shifting operations and exponent calculation. In shift operations it determines the number of shifts, thus enabling the calculations of the number of shifts in a run-time. The exponent result is transferred to the sv register. This register can be used for full normalization by serving as the destination of the exponent calculation, and as the control for the shift. Refer to paragraph 3.3.2.3, Normalization and to Chapter 4.

The sv register can also be used as a general-purpose temporary data register.

3.3.3 Saturation

Saturation arithmetic is employed to selectively limit overflow from the high portion of an accumulator to the extension bits. Saturation is performed when moving from the high portion or low portion of one of the accumulators through the XDB/ZDB, or when using the *lim* instruction, which performs saturation on the 36-bit accumulator. The saturation logic will substitute a "limited" data value having maximum magnitude and the same sign as the source accumulator.

In case saturation occurs while a move instruction (*mov* or *push*) from one of the accumulators (aXh, aXl, bXl or bXh) through the XDB/ZDB happens, the value of the accumulator is not changed. Only the value transferred over the XDB/ZDB is limited to a full-scale 16-bit positive (0x7FFF for aXh or bXh ; 0xFFFF for aXl; or bXl) or negative (0x8000 for aXh or bXh ; 0x0000 for aXl or bXl) value. Limitation will be correctly performed even if the transfer to the XDB/ZDB does not immediately follow the accumulator overflow. When an accumulator is swapped by the *swap* instruction, limitation will be correctly performed when the value is transferred to the XDB/ZDB. The saturation in move instructions can be disabled by the SAT bit in the st0 register. When limitation occurs, the L flag in st0 is set. For further details, refer to paragraph 3.7.2.

The *lim* instruction activates saturation on a 36-bit aX-accumulator. When overflow from the high portion of an aX-accumulator to the extension bits occurs, and a *lim* instruction is executed, the accumulator is limited to a full-scale 32-bit positive (0x7FFFFFFF) or negative (0x80000000) value.

Limitation will be correctly performed even if the *lim* instruction does not immediately follow the accumulator overflow. If an accumulator is swapped by a *swap* instruction, limitation will be correctly performed when the value is operated on by the *lim* instruction. The *lim* instruction can use the same accumulator for both source and destination, or it can use one source aX-accumulator, which will not change, and transfer the limited result into the other aX-accumulator. For further details, refer to the *lim* instruction in Chapter 4. When limiting occurs, the L flag in st0 is set. The SAT bit in st0 has no effect in this instruction. For further details, Refer to Paragraph 3.7.2.

3.3.4 Swapping the Accumulators

The aX-accumulators can be swapped with the bX-accumulators in a single-cycle. It is possible to *swap* two 36-bit registers or all the four 36-bits registers. Swapping is also enabled between a specific aX-accumulator, into one of the bX-accumulator, and in the same cycle from that bX-accumulator into the other aX-accumulator. Similarly swapping is enabled between a specific bX-accumulator, into one of the aX-accumulator, and in the same cycle from that aX-accumulator into the other bX-accumulator.
For a summary regarding the 14 *swap* options and other details, refer to the *swap* instruction in Chapter 4.

Note that during interrupts context switching, the a1 and b1 accumulators are automatically swapped. For further details, refer to section 3.7.3, Interrupt Context Switching.

3.4 DATA ADDRESS ARITHMETIC UNIT (DAAU)

The DAAU performs all address storage and effective address calculations required for addressing data operands in data, in program memory and supporting the software stack pointer. In addition, it supports latching of the modified register in maximum/minimum operations (see paragraph 3.3.1.3.4 Maximum-Minimum Operation). It also supports loop counter operations in conjunction with the *modr* instruction (see Chapter 4) and the R flag (see 3.7.2 Status Registers). This unit operates in parallel with other core resources to minimize address generation overhead. The DAAU performs two types of arithmetic functions; linear or modulo. The DAAU contains six 16-bit address registers (r0¸ r3 and r4¸ r5, also referred to as rN) for indirect addressing, two 16-bit configuration registers (cfgi and cfgj) for modulo and increment/decrement step control, as well as a base register (rb) for supporting index addressing. In addition, it contains a 16-bit stack pointer register (sp), alternative bank registers (r0b, r1b, r4b, cfgib) supported by an individual bank exchange, and a 16-bit minimum/maximum pointer latching register (mixp, see Paragraph 3.3.1.3.4, Maximum-Minimum Operation). The rN and configuration registers are divided into two groups for simultaneous addressing over XAB/ZAB and YAB (or PAB): r0-r3 with cfgi; and r4 as well as r5 with cfgj. Registers from both groups, in addition to rb and sp, can be used for XAB, ZAB and YAB (or PAB) for instructions, which use only one address register. In addition, in these instructions the Xspace, Zspace and Yspace can be viewed as a single continuous data memory space.

The r0, r1, r2, r3, r4, r5, cfgi, cfgj, sp, rb, mixp registers may be read from or written to by the XDB/ZDB as 16-bit operands, thus they can serve as general-purpose registers.

3.4.1 Address Modification

The DAAU can generate two 16-bit addresses every instruction cycle The addresses can be post-modified by two modifiers; linear and modulo modifier. The address modifiers allow data structures in memory (for circular buffers), delay lines, FIFOs, another pointer to the software stack, etc., to be produced.

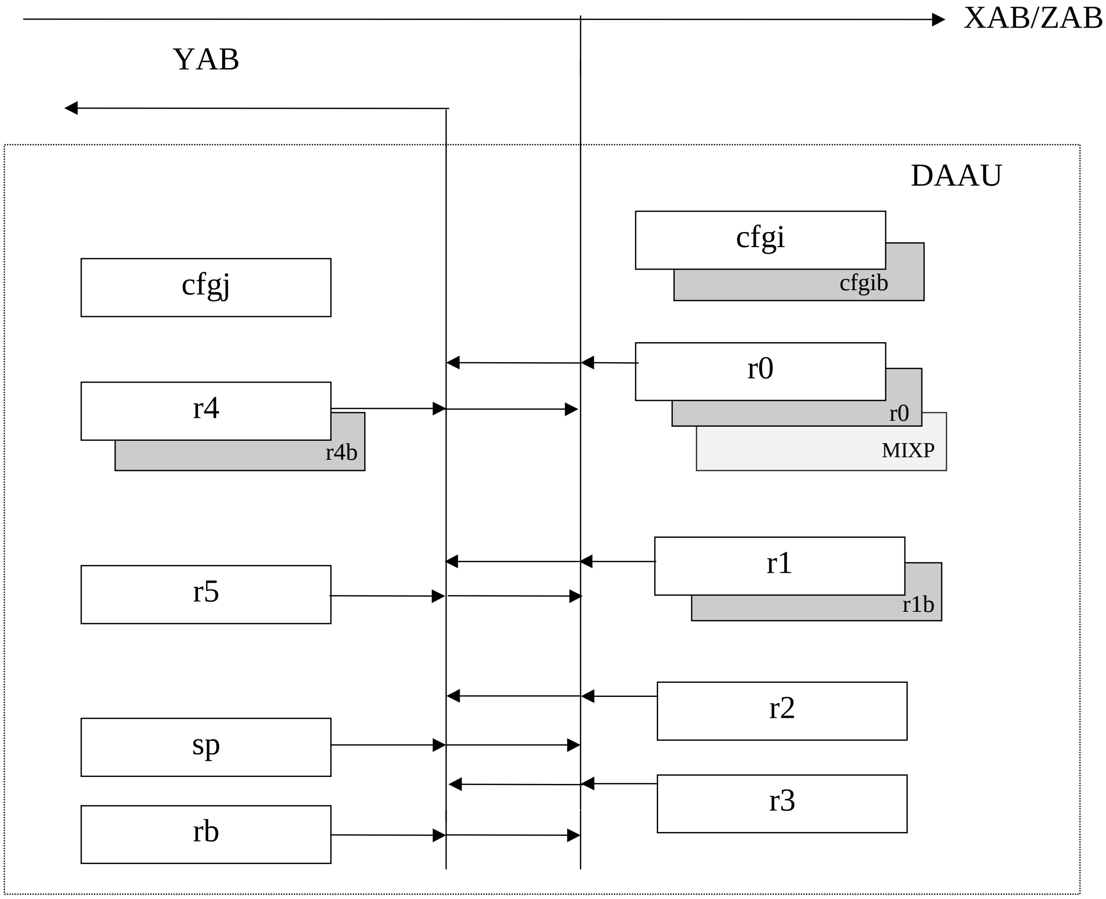


Figure 3-3 Data Address Arithmetic Unit Block Diagram

The rN registers can also be used, in addition to the block-repeat nesting, as loop counters in conjunction with the *modr* instruction (see Chapter 4) and the R flag of st0 (see 3.7.2 Status Registers). Address modification is performed using 16-bit (modulo 65,536) two's complement linear arithmetic.

The range of values of the registers can be regarded as signed (from -32,768 to +32,767) or unsigned (from 0 to +65,536). The same applies to the data space memory map. Refer to 3.6 MEMORY SPACES AND ORGANIZATION. Index addressing capability is also available (refer to 3.4.1.3).

The *modr* instruction has an option for post modification, it is not depended on the Mn bits.

3.4.1.1 Linear (Step) Modifier

During a single instruction cycle, one or two (out of different groups) address registers, rN, can be post incremented/decremented by 1 or added with a 2's complement 7-bit step (from -64 to +63). The selection of linear modifier type (one out of four) is included in the relevant instructions Refer to Paragraph 4.2.2, Conventions for Instruction set). Step values STEPI and STEPJ are stored in the 7 LSB of the configuration register cfgi and cfgj respectively.

Configuration Registers

cfgi

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
mod I									step i						

cfgj

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
mod J									step J						

3.4.1.2 Modulo Modifier

The two modulo arithmetic units can update one or two address registers from different groups within one instruction cycle. They are capable of performing modulo calculations of up to 2**9 (see note 2). Each register can be set independently to be affected or unaffected by the modulo calculation using the six Mn status bits in the st2 register.

Modulo setting-values MODI and MODJ (note the exact definition in the following paragraphs) are stored in 9 MSBs of configuration registers cfgi and cfgj respectively. To carry out accurate modulo calculations, the following constraints must be met (M = modulo factor; q = STEPx, +1, or -1):

a. Only the p LSBs of rN can be modified during modulo operation, where p is the minimal integer that satisfies 2**p ≥ M. rN should be initiated with a number whose p LSBs are less than M.

b. The constrains when the modulo **M is a power of 2** (full modulo operation) are as follows:

- The lower boundary (base address) must contain zeros, at least the k LSBs where k is the minimal integer that satisfies 2**k > M-1
- MODx (x denotes I or J) must be loaded with M-|q|.
- M ≥ q

c. The constrains when the modulo **M is not a power of 2** are as follows :

- The lower boundary (base address) must contain zeros, at least the k LSBs where k is the minimal integer that satisfies 2**k > M-|q|
- MODx (x denotes I or J) must be loaded with M-|q|.
- M must be an integer multiple of q (this is always true for q=+/-1).
- rN should be initiated with a number that contains an integer multiple of |q| or zeros in it's k LSBs.

Note: |q| denotes the absolute value of q.

The **modulo modifier operation**, which is a post-modification of the rN register, is defined as follows:

rN ← 0 in k LSB	; if rN is equal to MODx in k LSBs and q ≥ 0,
rN ← MODx in k LSB	; if rN is equal to 0 in k LSBs and q < 0,
rN (k LSBs) ← rN+q (k LSBs)	; Otherwise

When M = |q| (i.e. MODx = 0) modulo operation is: rN ← rN.

Notes:

1. r0¸r3 can only operate with STEPI and MODI, while r4¸ r5 can operate only with STEPJ and MODJ.

2. The modulo is capable of operating for modulo values greater than 512, when the M-|STEPx| ≤ 511 and all four constrains specified for (c) above are met.

Examples:

(1) M=7 with STEPx=1 (or +1 selected in instruction), MODx=7-1=6, rN=0x0010 (hexa). The sequence of rN values will be: 0x0010, 0x0011, 0x0012, 0x0013, 0x0014, 0x0015, 0x0016, 0x0010, 0x0011...

(2) M=8 with STEPx=2, MODx=8-2=6, rN=0x0010. The sequence of rN values will be: 0x0010, 0x0012, 0x0014, 0x0016, 0x0010,0x 0012,...

(3) M=9 with STEPx=-3, MODx=9-|-3|=6, rN=0x0016. The sequence of rN values will be: 0x0016, 0x0013, 0x0010, 0x0016, 0x0013,...

(4) M=8 with STEPx=3, (2^3 = 8 - full modulo support), MODx=8-3=5, rN=0x0010. The sequence of rN values will be: 0x0010, 0x0013, 0x0016, 0x0011, 0x0014, 0x0017, 0x0012, 0x0015, 0x0010, 0x0013, ...

3.4.1.3 Index Modifier

The TeakLite has short and long Index addressing modes. The Base register is rb, one of the DAAU registers. In the short index addressing mode, the base register along with a 7-bit signed short immediate value (-64 to +63), embedded in the instruction opcode, are used to point at a data memory location in a single cycle. In the long index addressing mode, the base register and a signed 16-bit offset, applied as the second word of the instruction, are used to access the memory in two cycles.

Unlike the linear and modulo addressing modes, in both index-addressing modes, address pre-modification is performed prior to accessing the memory. The base register is unaffected. Indexed addition, subtraction, comparison, *and*, *or*, *xor* and move from/into the pointed data memory location, can be performed in either one or two cycles, using the short or long mode respectively.

The base register can be used as an array pointer, or in conjunction with the Stack Pointer (sp) register. When the stack is used for transferring routine parameters - initializing the base register by the sp value, enables quick access to routine parameters transferred using the stack. The index-addressing mode is useful for supporting the C-compiler.

The rb register is part of the global register set and can be used as a general-purpose register.

3.4.2 Software Stack

The TeakLite contains a software stack, pointed at by a dedicated 16-bit register, the Stack Pointer (sp). The sp contains the address of the top value in the stack, hence it points at the last value pushed onto the stack. The stack is filled from high-memory address to low memory address. A *pop* instruction performs a post-increment, a *push* instruction performs a pre-decrement. The Program Counter (pc) is automatically pushed to the stack, whenever a subroutine call or an interrupt occurs, and popped back upon return from a subroutine or an interrupt. Other values can be pushed and popped using the *push* and *pop* instructions. The top of stack can be read without affecting sp, using a dedicated *mov* instruction.

The software stack can reside anywhere in the data space (Xspace, Zspace, and Yspace), and can be also accessed by any other pointer (r0 to r5 and rb). Please refer to *callr* instruction for further details about the behavior of a stack residing in Zspace

The software stack is useful for supporting the C-compiler. The stack can be used for the transfer or routine parameters. Thus, after initializing the base register (rb) by the sp value, the routine parameters can be referenced by the index mode with the *mov, add, sub, cmp, and, or, xor* instructions. Another support for the transfer of routine parameters is the *retd* instruction, which returns from a subroutine and updates the sp by a short immediate value.

The sp register is part of the global register set (as defined in Paragraph 3.7.1).

3.4.3 Alternative Bank of Registers

The DAAU contains an alternative bank of four registers: r0b, r1b, r4b, and cfgib. For each of the r0/r0b, r1/r1b, r4/r4b, or cfgi/cfgib only one register is accessible at a time. The selection between the current and the alternative register is controlled by a special *banke* instruction, which exchanges (swaps) the contents between the current register and the alternative bank register.

The bank registers are individually selected for exchange, meaning that the *banke* instruction includes a list of registers ought to be exchanged in a single cycle. There are 15 different options to use the four registers of the alternative the bank. Refer to *banke* instruction in Chapter 4.

The individual selectivity of the bank registers contributes to flexibility of the bank registers. The user can decide where each of the alternative registers will be utilized in interrupts, routines, etc.

3.5 PROGRAM CONTROL UNIT (PCU)

The Program Control Unit (PCU) performs instruction fetch, instruction decoding, exception handling, hardware loop control, wait state support and on-chip emulation support. In addition, it controls the internal program memory protection (see Paragraph 3.6.2.1).

The PCU contains the Repeat/Block-Repeat unit, and two 16-bit directly accessible registers: the Program Counter (pc) and the Loop Counter (lc) of the block-repeat unit.

The PCU selects and/or calculates the next address from several possible sources as specified herein:
An incremented pc in a sequential program flow. Jump address in branch or *call* operations; short PC-relative address of 7-bit in a relative branch or *call* operations; start address and exit address of hardware loops. Interrupts vector handling; user write to pc; or the top value on the Stack, pointed to by the sp register upon returning from subroutines and interrupts. The PCU also writes the pc to the top of stack in subroutines and interrupts.

The pc always contains the address of the next instruction. For further information as regards the pipeline method, refer to Chapter 4, Section 4.6.

3.5.1 Repeat and Block Repeat Unit

The Repeat/Block-Repeat unit performs hardware-loop calculations and controls of functions with no overhead, other than the one-time implementation of set-up instructions *rep* or *bkrep*, to initialize the repeat or block-repeat mechanism, respectively. Four nested levels of block-repeat can be performed and the *rep* instruction can be performed inside each one of these levels.

The number of repetitions can be a fixed value embedded in the instruction code or a value transferred from one of the processor's 16-bit registers. This option supports calculations of the number of repetitions in a run-time.

The repeat mechanism contains an internal 16-bit repeat counter, *repc*, and this in order to repeat a single word instruction from 1 to 65536 repetitions. The *repc* counter is readable by the programmer.

In block-repeat operation, the last and first addresses of a loop are stored in 16-bit dedicated registers. A 16-bit dedicated counter, lc, counts the number of loop repetitions (1-65536). In case of nested block-repeats it saves these values in internal registers. The user can access the lc of each level. The programmer cannot access the start, the end address registers as well as the internal shadow registers as registers. An indication of the block-repeat nesting level is a read-only *Block-repeat nesting Counter*, (BC2, BC1, and BC0) in the Internal Configuration Register (icr). See also paragraph 3.7.4, Internal Configuration Register.

The 16-bit block-repeat Loop Counter (lc) is one of the global registers The lc register can be used as an index inside the block-repeat loop or for determining the value of the block-repeat counter when a jump out of the block-repeat loop occurs.

The single instruction repeat can reside in each of the block-repeat levels. Both the repeat and the block-repeat mechanisms are interruptible. For further details as regards specific limitations, refer to *rep* and *bkrep* instruction in Chapter 4.

A *break* instruction can be used for the purpose of stopping each of the four nested levels of a block-repeat. Refer to *break* instructions in chapter 4.

The In-loop (LP) bit in icr is set when a block-repeat is executed and reset upon normal completion of the outer block-repeat loop. When the user resets this bit, it stops the execution of all four levels of block-repeat. For further details as regards the LP bit, refer to Paragraph 3.7.4, Internal Configuration Register. If the LP bit is cleared in the current block repeat loop, the processor is no longer in any of the block loop levels (BC2,BC1,and BC0) bits in icr register are cleared). Therefore, when the last address will be reached there will be no jumps to the first address of the loop, the counter will not be decremented and the processor will continue to the sequential instruction. An exception is when LP is cleared at one of the last three addresses of the block repeat. Should the above mentioned occur, the effect of clearing the lp will take place only in the next loop. An instruction, which reads icr and starts at last address of the outer block-repeat loop will result in an LP bit equal to zero when the last repetition of this outer loop is reached.

The lc register may also serve as a 16-bit general-purpose register for temporary storage.

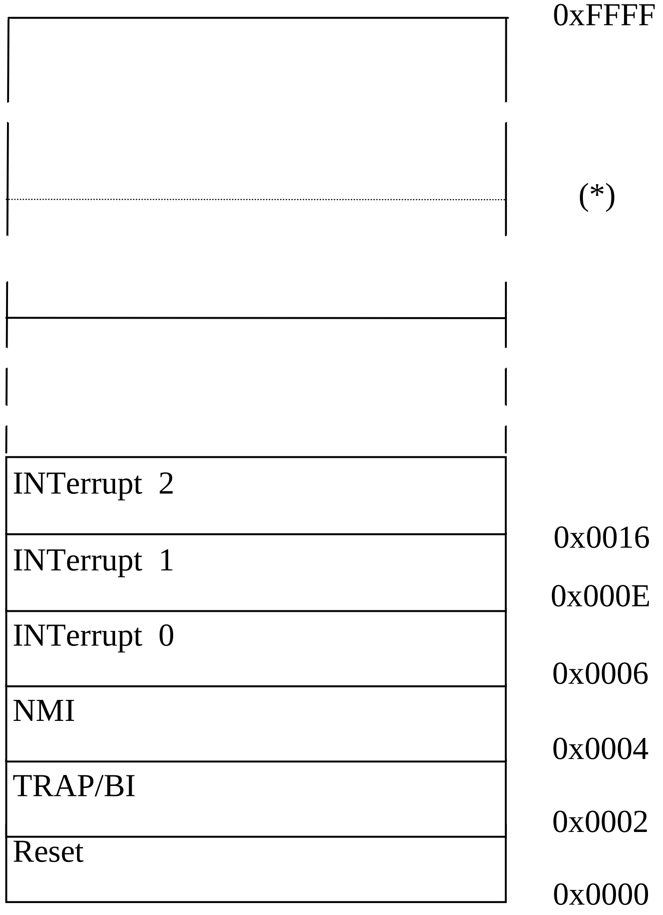
3.6 MEMORY SPACES AND ORGANIZATION

Two independent memory spaces are available, data and program, each one of them is 64K words. The data space is composed from X, Y and Z) .

3.6.1 Program Memory

Addresses 0x0000-0x0016 (see Figure 3.4) are used as interrupt vectors for Reset, TRAP/BI (software interrupt / breakpoint interrupt), NMI (Non-maskable interrupt) and three maskable interrupts (INT0, INT1, INT2). The RESET, TRAP/BI and NMI vectors were separated by two locations, so that branch instructions can be accommodated in those locations if desired. The maskable interrupts were separated by eight locations so that branch instructions, or small and fast interrupt service routines, can be accommodated in those locations. The program memory can be implemented on-chip, and/or off-chip.
The TeakLite supports a wait-state generator for the purpose of interfacing a slow program memory.

The program memory addresses are generated by the PCU.



(*) The program memory can be on-chip or off-chip, according to the user-defined interface logic.

Figure 3.4 Program Memory Map

3.6.1.1 Internal Program Memory Protection

The TeakLite core is designed to support internal program memory protection, even when an external program memory is connected to it. A secured version of an internal program-based chip can be implemented, by employing this design approach. The internal program memory is this way protected against unauthorized reading.. For further details, refer to Chapter 5, the Core Interface, paragraph 5.12, the Program Protection Mechanism.

Once the protection feature is used, the internal program memory must be divided into two parts: the program part, which is hidden from the external user, and the data part, which is accessible. The exact boundary between the program and data portions of the internal program space must be determined when the extent of required internal program space per a specific COMBO is specified. The internal program memory can be fully protected, or partially protected/readable by the external user's program.

3.6.1.2 PSYNC_FAST Mode

In order to improve the frequency of a TeakLite DSP Core system solution and synchronous memories, a special switch, PSYNC_FAST, was inserted in the internal HW design of the core (in the RTL code).

PSYNC_FAST is used to improve the timing of the PPAP (Program Address) output bus of the core, thus making it easier (i.e., less timing critical) to connect the core to synchronous memories. This switch can only be used with the PSYNC switch (which is a RTL switch only), and is not applicable otherwise. However, using the PSYNC_FAST option imposes an extra cycle on certain instructions that breaks the pipeline, as detailed in section 4.4 of Instruction Set Details (*ret, reti, rets*, *retd, retid, pop, mov, br, call, brr, callr)*. This implies that the usage of the PSYNC_FAST switch is recommended for high frequency TeakLite based systems only. For further details, refer to Chapter 5, “Core Interface”.

The Software Development Tools (Debugger and Assembler) have a similar switch to indicate the user and simulate the extra required cycle.

Further details are beyond the scope of this architecture specification and can be found in a special application note.

3.6.2 Data Memory

The TeakLite’s data space is divided into three sections: Xspace, Yspace and Zspace. The three data space configuration is as follows:

- Xspace - located between address 0x0000 and the configurable lower boundary-1.
- Yspace – located between address 0xFFFF and the configurable upper boundary.

- Zspace – located between the configurable upper boundary–1 and the lower boundary.

Boundaries can be configured with a resolution of 1K. Configuration must meet the following requirements:

- Upper boundary > Lower boundary
- Minimal Xspace is 0K words
- Minimal Yspace is 1K words
- Minimal Zspace is zero (i.e. Zspace may not exist).

An input configuration bus of six bits, GXRAMCONFP defines the lower boundary. The upper boundary is defined by another 6 bit input configuration bus, named GYRAMCONFP.

The X, Z and Y spaces were mapped to allow continuos spaces from either direction, i.e. from X to Z and Y as well as from X to Y.

The TeakLite core applies a single stage write buffer when accessing the X and Y spaces. Memory read transactions (from X or Y) precede memory write. Once the buffer is activated (any write transaction to X or Y), it will use the next available cycle in which there is no memory transaction to discharge (an actual memory write transaction).

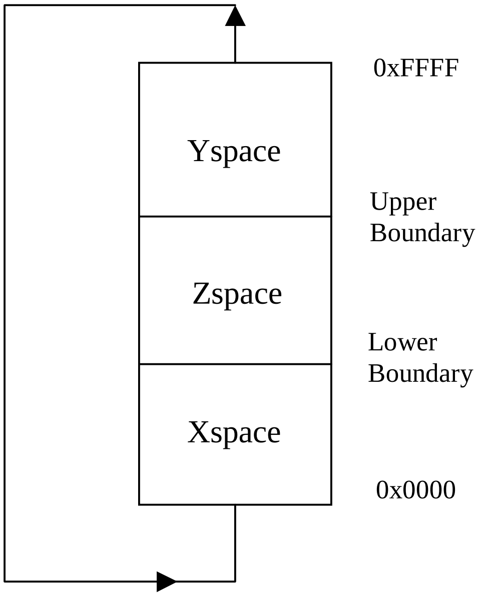


Figure 3.5 Data Memory Map

- In addition, the next memory write transaction (to Xor Y) will empty the current write buffer and reload it simultaneously with a new write transaction.

- It was a design consideration that any un-sequential instruction flow (branch, call, interrupt, return) is executed, that the write buffer would not contain data related to instructions issued prior to the branch, call, interrupt or return.

- In some cases when a sequence of memory write (to X or Y) are followed immediately by a memory read from the same address occur. The data will be fetched from the write buffer and not from the memory.

The above-described sequence (unpredictable write) is unacceptable when addressing a memory mapped I/O device. Such a device may be activated through a memory write. Therefore, in transactions to memory mapped I/O devices, the device must be read directly rather than being by passed by the write buffer. Therefore, the Zspace interface does not include a write buffer, since the interface is slightly different from the X and Y spaces.

The following devices resemble a typical connection to the Zspace:

- Memory mapped I/O devices
- External registers
- Bus structures, implemented to interface other on-chip entities, or used to reach chip pins.
- Shared (core and other entities) data memory RAM arrays, to which the core has a write access (e.g. mail boxes).
- Debug and product test entities (i.e. JTAG module, OCEM).

3.6.3 Memory Addressing Modes

3.6.3.1 Following is a description of the five data memory addressing modes provided:

a. **Short Direct Addressing Mode**: Eight bits embedded in the instruction opcode as LSBs plus eight bits from status register st1 (see Paragraph 3.7.2, Status Registers) as MSB, constitute the 16 bit address to the Data memory, Hence, the pages are of 256 words each. For example, page 0 corresponds addresses 0 to 255 in Xspace, page 1 from 256 to 511 in Xspace, and page 255 from -256 to -1 in Yspace. Any memory location in the 64K-word data space, can directly be accessed in a single-cycle.

b. **Long Direct Addressing Mode**: Sixteen bits embedded in the instruction opcode as the second word of the instruction, are used as the 16-bit address of the Data memory. Any memory location in the 64K-word data space, can directly be accessed in two cycles.

c. **Indirect Addressing Mode**: The rN registers of the DAAU are used as 16 bit addresses for indirect addressing the Xspace and Yspace. Some instructions use two registers, simultaneously, addressing a memory location in Xspace and another in Yspace, both addressed in indirect addressing mode.

d. **Short Index addressing mode** : The base register rb plus an index value (offset7, a short immediate value embedded in the instruction opcode), are used for index-based indirect addressing the Xspace, Yspace or Zspace. The index value can be from -64 to +63. The actual address is rb+offset7 though, the contents of rb is kept unaffected. For further details, refer to paragraph 3.4.1.3, Index Modifier.

e. **Long Index Addressing Mode**: The base register rb plus a 16-bit immediate index value (embedded in the second instruction opcode word), are used for index-based indirect addressing the Xspace or Yspace. The index value can be from -32768 to +32767. The contents of rb remain unaffected. For further details, refer to paragraph 3.4.1.3, Index Modifier.

The software stack located in the data memory is addressed using the stack pointer (sp) register.

3.6.3.2 The Addressing program memory is accomplished by:

a. **Indirect Addressing Mode**: The rN registers of the DAAU and the accumulator can be used for addressing the program memory in specific instructions.

b. **Special Relative Addressing Mode**: Special Branch-Relative (*brr*) and call-Relative (*callr*) instructions support jumping, relative to the pc (from pc-63 to pc+64).

3.7 PROGRAMMING MODEL AND REGISTERS

Most of TeakLite visible registers are arranged as a global register set of 34 registers, which can be accessed by most data moves and core operations. The registers are listed below, organized according to the units' partition. Additional details concerning each register can be found in the description of each unit and in the following paragraphs.

3.7.1 Programming Model

The following figures describe all TeakLite registers. Registers, which are part of the global register set, referred to as REG in Chapter 4’s instruction set, are **bold-faced**.

CBU Registers:

CU Registers:

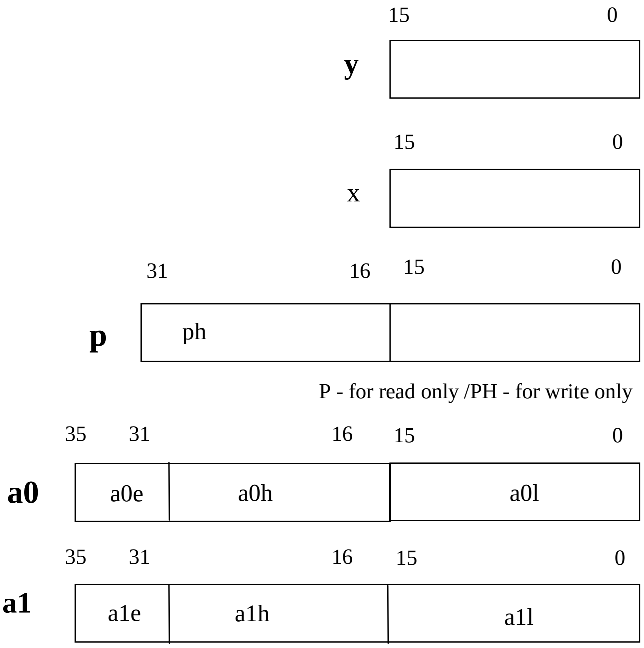


Figure 3-6 The TeakLite Registers

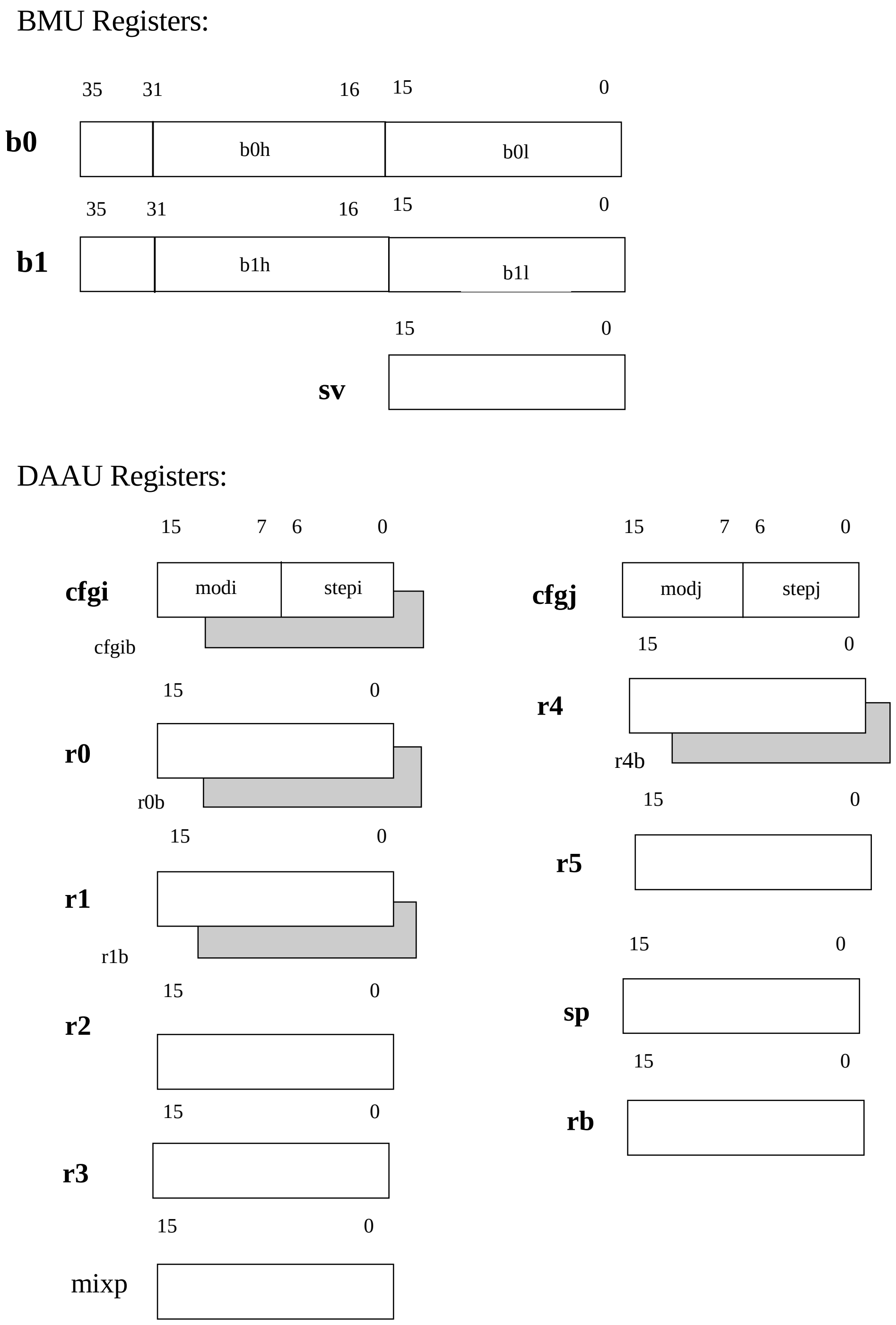
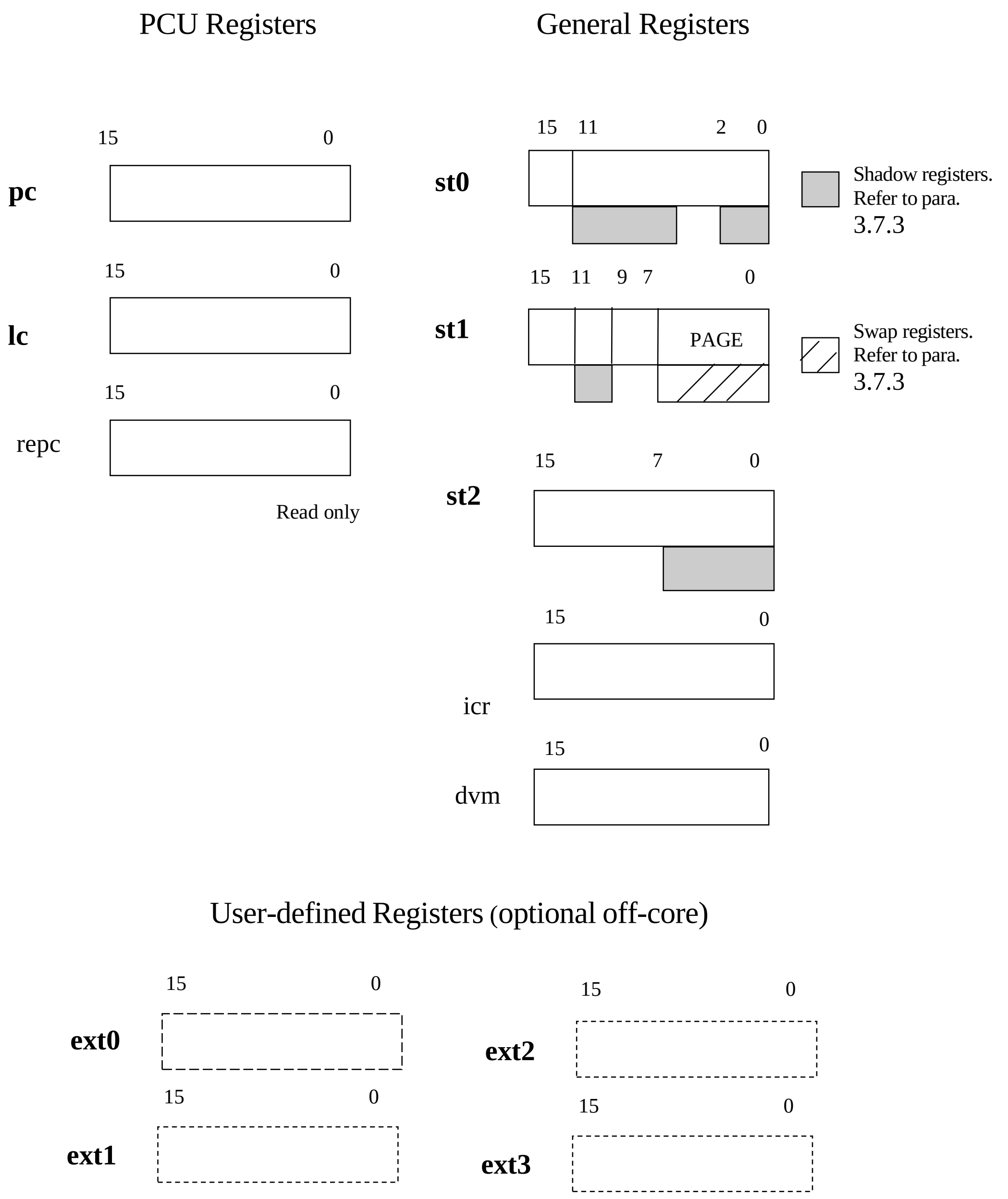


Figure 3-6 The TeakLite Registers (Cont'd)



PCU Registers

General Registers

User-defined Registers (optional off-core)

Figure 3-6 The TeakLite Registers (Cont’d)

3.7.2 Status Registers

Three status registers are available for the purpose of holding flags, status bits, control bits, user I/O bits, and paging bits aimed at direct addressing. The contents of each register and their field definitions are described below.

3.7.2.1 Status Registers Format

st0

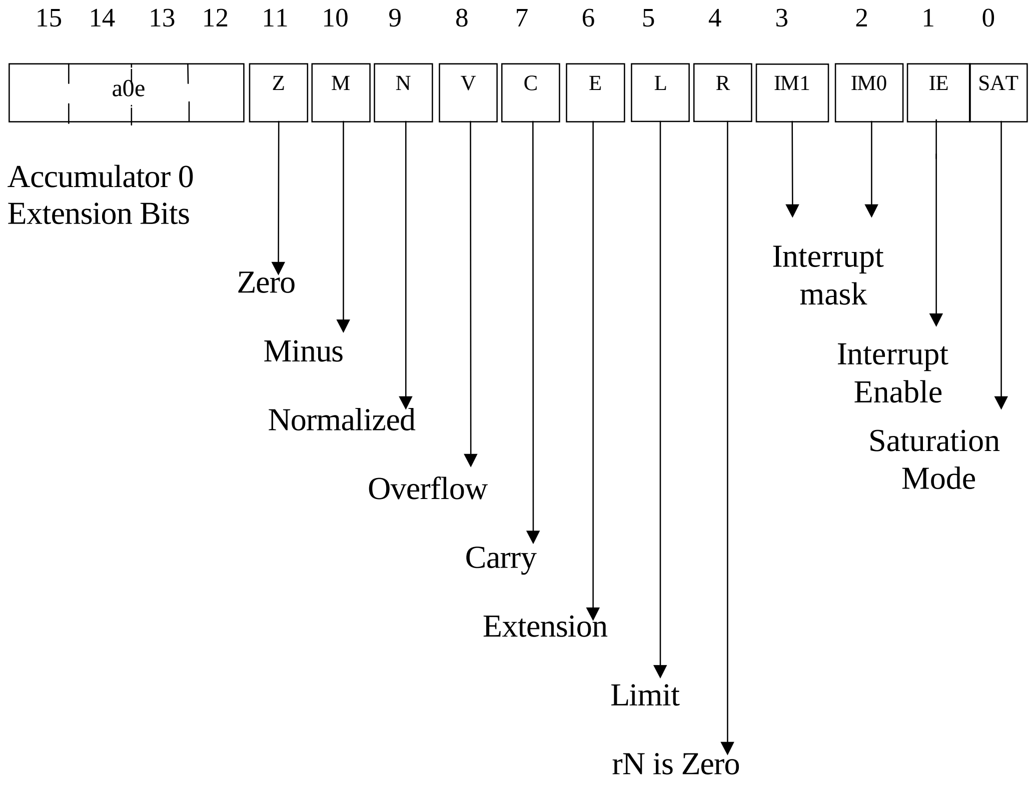


Figure 3-7 Status Registers Format

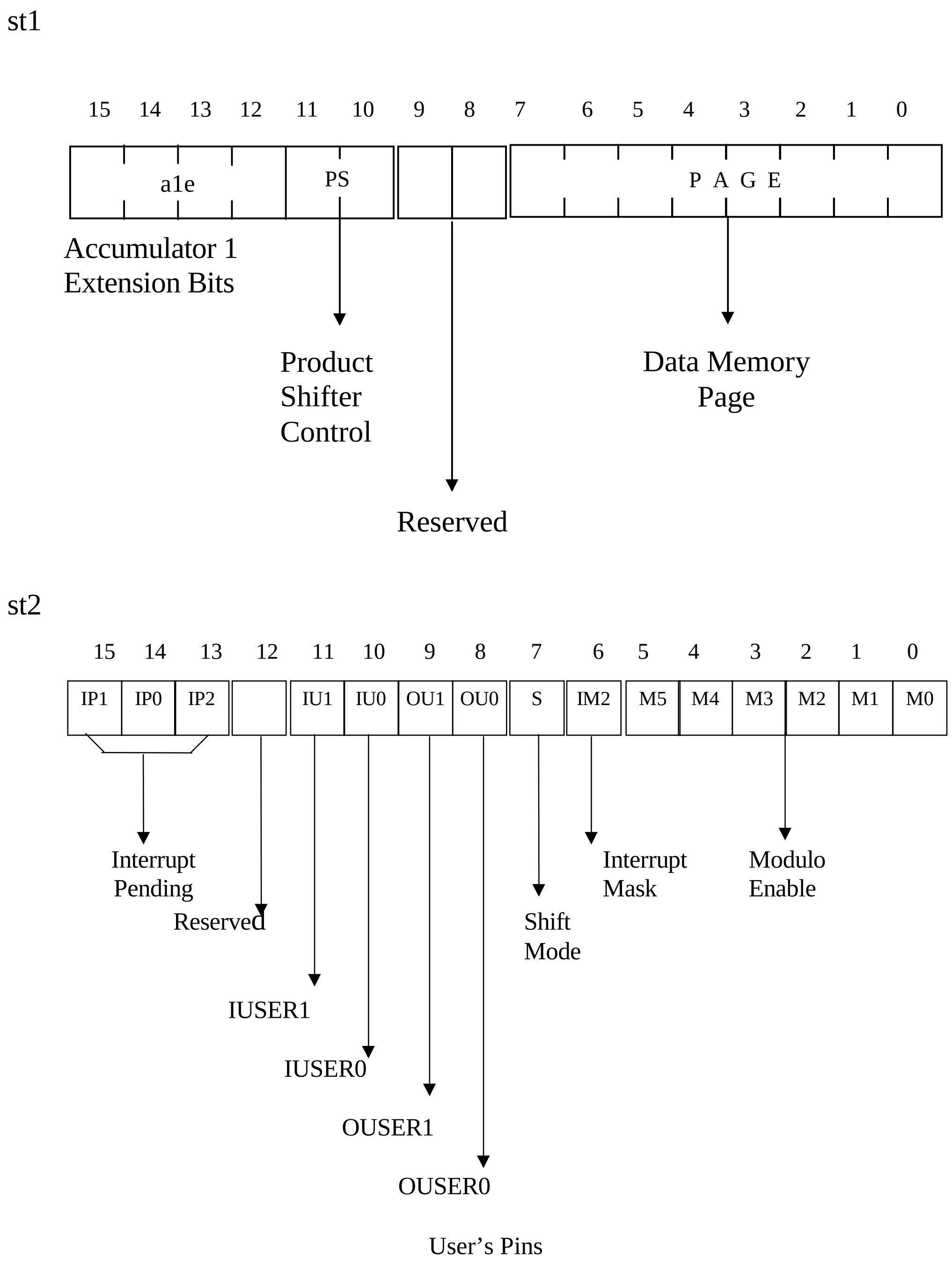


Figure 3-7 Status Registers Format (Cont'd)

Notation:

*	Written as zero, read as don’t care.
~	Not
∪	Or
∩	And
⊕	Exclusive-or

3.7.2.2 Status Register Field Definition

Following are the definitions and behavior patterns of the status registers' fields. Writing into one of the status registers can also modify most of the fields.

The flags (Z, M, N, V, C, E, and L) indicate the result of the last ALU, BFO or Barrel Shifter operation. When one of these outputs is latched into one of the destination accumulators, the flags generally indicate the destination accumulator status.

Some specific instructions (e.g. *max*, *mi*n) have a different effect than those listed under the status register fields. For further details refer to Chapter 4.

st0 Register

Zero (Z) - Bit 11
Set if the ALU/BFO/Barrel Shifter output, generated by the last instruction, equals zero; cleared otherwise. This flag is also used to indicate the result of the test bit/s instructions (*tst0*, *tst1*, *tstb*).

A processor reset clears the zero flag.
Writing into st0 can modify the zero flag.

Minus (M) - Bit 10
Set if the ALU/BFO/Barrel Shifter output, generated by the last instruction, is a negative; cleared otherwise. The Minus flag is the same as the MSB of the output (bit 35).

A processor reset clears the Minus flag.
Writing into st0 can modify the Minus flag.

Normalized (N) - Bit 9

Set if the 32 LSBs of the ALU/Barrel Shifter output, generated by the last instruction, are normalized; cleared otherwise. i.e. set if $Z \cup ((\text{bit } 31 \oplus \text{bit } 30) \cap \sim E)$.

A processor reset clears the normalized flag.
Writing into st0 can modify the normalized flag

Overflow (V) - Bit 8

Set if an arithmetic overflow (36-bit overflow) occurs after an arithmetic operation; cleared otherwise. It indicates that the result of an operation cannot be represented in 36 bits.

A processor reset clears the overflow flag.
Writing into st0 can modify the overflow flag.

Carry (C) - Bit 7

Set if an addition operation generates a carry, or if a subtraction operation generates a borrow; cleared otherwise. It also accepts the rotated bit or the last bit shifted out of the 36-bit result.

The carry flag is cleared during processor reset.
The carry flag can be modified by writing into st0.

Extension (E) - Bit 6

Set if bits 35-31 of the ALU/Barrel Shifter output, generated by the last instruction, are not identical; cleared otherwise. If the E flag is cleared, it indicates that the 4 MSBs of the output, are the sign-extension of bit 31 and can be ignored.

The extension flag is cleared by a processor reset.
The extension flag can be modified by writing into st0.

Limit (L) - Bit 5

The L flag fulfill two functions: it latches the overflow (V) flag and indicates limitation during accumulator move or *lim* operations.

Set if the overflow flag was set (overflow latch), or a limitation occurred when performing a move instruction (*mov* or *push*) from one of the accumulators (aXh, aXl, bXl or bXh) through the data bus, or when limitation occurred when the *lim* instruction was executed. Otherwise it is not affected.

The limit flag is cleared by a processor reset.
The limit flag can be modified by writing into st0.

rN register is zero (R) - Bit 4
This flag is affected by the *modr* and *norm* instructions. The R flag is set if the result of the rN modification operation (rN ; rN+1; rN-1; rN+S) is zero; cleared otherwise. The R flag status is latched until one of the above instructions is used.

The R flag is cleared by a processor reset.
The R flag can be modified by writing into st0.

Interrupt0, interrupt1 Mask (IM1, IM0) - Bits 3, 2
IM0 - Interrupt mask for INT0
IM1 - Interrupt mask for INT1
Clear - disable the specific interrupt
Set - enable the specific interrupt

The interrupt mask bits are cleared by a processor reset.
The interrupt mask bits can be modified by writing into st0.

Interrupt Enable (IE) - Bit 1
Clear - disable all maskable interrupts
Set - enable all maskable interrupts

The interrupt enable bit is cleared during processor reset.
The interrupt enable bit can be modified by the instructions *eint* (enable interrupts), *dint* (disable interrupts), by using *reti*/*retid* for returning from one of the maskable interrupt service routines, or by writing into st0.

Saturation Mode (SAT) - Bit 0
Clear - enable saturation when transferring the accumulator’s contents is transferred on the data bus.
Set - disable the saturation mode.

<u>Note</u>: this bit has no affect on the *lim* instruction.

The saturation enable bit is cleared by a processor reset.
The saturation enable bit can be modified by writing into st0.

st1 Register

Product Shifter control (PS) - Bits 11, 10
The Product Shifter control bits, control the scaling shifter at the p register output as follows:

Table 3-3 PS Files and the Related Number of Shifts

ps bits		Number of shifts
bit 11	bit 10	
0	0	No shift
0	1	Shift right by one
1	0	Shift left by one
1	1	Shift left by two

The PS bits are cleared by a processor reset.
The PS bits can be modified by writing into st1.

Reserved bits - Bit 9, 8
These bits are written as zero, read as don't care.

Data memory space Page (PAGE) - Bits 7, 6, 5, 4, 3, 2, 1, 0
Used for direct data memory addressing mode. Refer to paragraph 3.6.3, Memory Addressing Modes.

The PAGE bits can be modified by a *load* instruction, *lpg* instruction or by writing into st1.

st2 Register

Interrupt Pending Status (IP1, IP0, IP2) - Bits 15, 14, 13
IP0 - Interrupt pending for INT0
IP1 - Interrupt pending for INT1
IP2 - Interrupt pending for INT2
The interrupt pending bit is set when the corresponding interrupt request is active. The bit reflects the interrupt pin level regardless of the mask bits.

The IPx bits are read only.

Reserved bit - Bit 12
This bit is written as zero, read as don't care.

IUSER0, IUSER1 (IU1, IU0) - Bits 11, 10
The IUSERx bits reflect the logic state of the corresponding user input pins.

The IUSERx bits are read only bits.

OUSER0, OUSER1 (OU1, OU0) - Bits 9, 8
The OUSERx bits define the logic state of the corresponding user output pins.

The OUSERx bits are cleared by a processor reset.
The OUSERx bits can be modified by writing into st2.

Shift mode (S) - Bit 7
The Shift mode bit defines the shift method. Affects in all shift instructions : *shfc*, *shfi*, *moda*, *modb*, *movs*, and *movsi*. (Refer to section 3.3.2.1.1, Shifting Operations.)
Clear - the shift instruction will perform an arithmetic shift
Set - the shift instruction will perform a logic shift

The Shift mode bit is cleared by a processor reset.
The Shift mode bit can be modified by writing into st2.

Interrupt2 Mask (IM2) - Bit 6
Interrupt mask for INT2
Clear - disable interrupt2
Set - enable interrupt2

The interrupt mask bit is cleared by a processor reset.
The interrupt mask bit can be modified by writing into st2.

Modulo Enable (M5, M4, M3, M2, M1, M0) - Bits 5, 4, 3, 2, 1, 0
Cleared Mn bit - when using the corresponding rN register, the rN register will be modified as specified by the instruction, regardless of the modulo option.
Set Mn bit - when using the corresponding rN register, the rN register will be modified as specified by the instruction using the suitable modulo.

Note:

> The *modr* instruction is the only instruction that can use one of the rN registers without being affected by the corresponding Mn bit, using a special option field.

The Mn bits are cleared by a processor reset.
The Mn bits can be modified by writing into st2.

3.7.3 Interrupt Context Switching

When a program is interrupted by an interrupt service routine, it is necessary to save those registers used by the service routine, so that the interrupted program resumes operation properly. To reduce the involved overhead, a context switching mechanism can be used for each of the following interrupts: NMI, INT0, INT1 and/or INT2. The decision if a specific interrupt is using the context switching mechanism or not is determined according to the corresponding bit in the Internal Configuration Register (ICR). Refer to Paragraph 3.7.4, ICR contents and to the *mov* instruction in Chapter 4.

When an interrupt that uses context switching is accepted, context switching occurs *automatically* without any payments in interrupt latency. When returning from this interrupt service routine, context switching should be used to restore the original register values automatically. Refer to the *reti* and *cntx* instructions in Chapter 4.

Context switching involves three parallel mechanisms: push/pop from/to dedicated shadow bits, swap of a dedicated page register, and swap between two specific accumulators.

The following register bits are saved automatically as shadow bits, i.e. one stack level register. This means that the data bits can be pushed to or popped from the shadow registers: st0[0] , st0[2¸11] , st1[10¸11] , st2[0¸7] (see figure).

The Page bits, st1[0¸7] are swapped to an alternative register. This means that when an interrupt is accepted, the current page is saved into the alternative register, while the previous (stored) value of the page is restored. So that it can be used without additional initialization. When returning from the interrupt, the interrupt page is saved again into the alternative register, for the next interrupt, and the page used before entering the interrupt service routine, is swapped back to st1.

st0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	a 0 e				Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT
Shadow					Z	M	N	V	C	E	L	R	IM1	IM0		SAT

st1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	a1e				P	S	*	*	PAGE							
Shadow					P	S										
Swap									PAGE							

st2

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	IP1	IP0	IP2	*	IU1	IU0	OU1	OU0	S	IM2	M5	M4	M3	M2	M1	M0
Shadow									S	IM2	M5	M4	M3	M2	M1	M0

Accumulators

36 bit b1 accumulator	⇐ SWAP ⇒	36 bit a1 accumulator

* Written as zero, read as don't care.

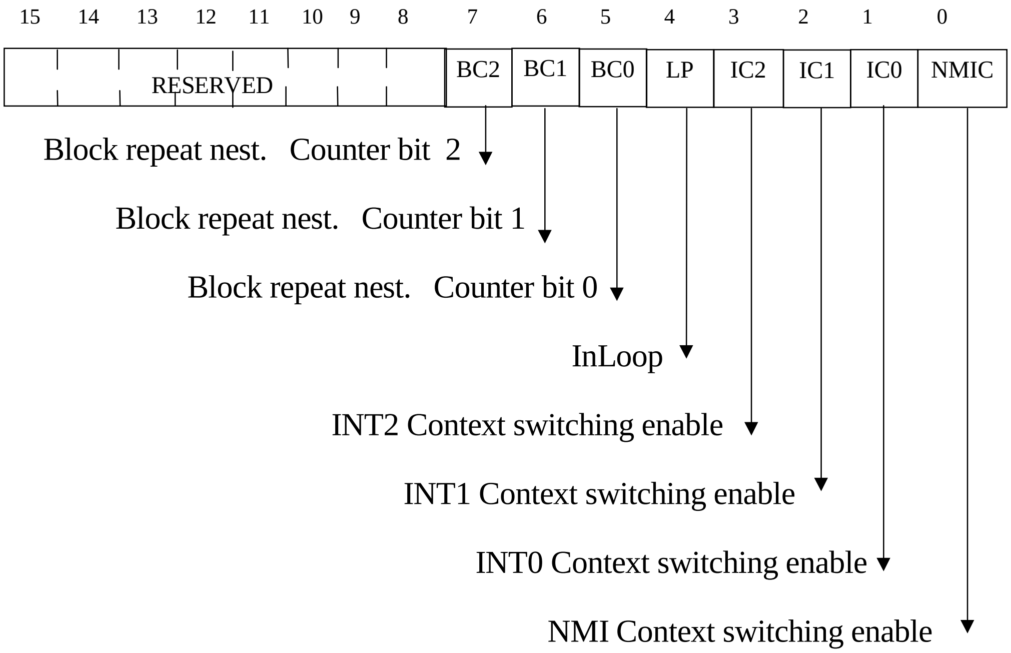
The a1 and b1 accumulators are automatically swapped. Therefore, it is very convenient to use b1 to store data needed for interrupt routines. This data will be transferred automatically into the a1 accumulator on interrupt service for interrupts which use the context switching mechanism, and transferred back while returning from the interrupt service routine.

A context switching activation instruction is also available. For further details, refer to *cntx* instruction in Chapter 4.

3.7.4 Internal Configuration Register

The internal configuration register (icr) includes context switching bits, a block-repeat indication and processor status bits.

ICR


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
RESERVED
BC2
BC1
BC0
LP
IC2
IC1
IC0
NMIC
Block repeat nest. Counter bit 2
Block repeat nest. Counter bit 1
Block repeat nest. Counter bit 0
InLoop
INT2 Context switching enable
INT1 Context switching enable
INT0 Context switching enable
NMI Context switching enable

The reserved bits (bits 8-15) are read as don't care.

Block repeat nesting Counter (BC2, BC1, BC0) - bits 7, 6, 5

Holds the current block repeat loop nesting level according to :

Table 3-4 Block-repeat counter state description

BC2	BC1	BC0	Block-repeat counter state description
0	0	0	Not within a block-repeat loop
0	0	1	Within first block-repeat level (the most outer loop)
0	1	0	Within second block-repeat level
0	1	1	Within third block-repeat level
1	0	0	Within fourth block-repeat level (the most inner loop)

The BCx bits are cleared by a processor reset, and due to disabling the block-repeat mechanism by clearing the LP bit (i.e. when all block-repeats are aborted).

The BCx bits are read only.

INLOOP (LP) - bit 4

Set if a block repeat is performed; cleared otherwise.

When data is transferred into the icr, the LP bit is affected or unaffected in the following manner :

'1' - The LP bit and the block-repeat nesting counter are cleared.
'0' - The LP bit is unaffected.

Clearing this bit, causes a break from the **four** levels of block-repeat, hence clearing the block repeat nesting counter (BCx) bits.

The inloop bit is cleared by a processor reset.
The inloop bit can be cleared by writing ('1') into icr.

See also paragraph 3.5.1, Repeat and Block-Repeat Unit.

To break out from one block-repeat level ,refer to *break* instruction in Chapter 4.

Context switching enable (IC2, IC1, IC0, NMIC) - bits 3, 2, 1, 0

IC2 - INT2 Context switching enable
IC1 - INT1 Context switching enable
IC0 - INT0 Context switching enable
NMIC - NMI Context switching enable

Set - Enable context switching during the beginning of the corresponding interrupt.
Clear - Disable context switching during the beginning of the corresponding interrupt.

The ICx bits and NMIC are cleared during processor reset.
The ICx bits and NMIC can be modified by writing into the icr register.

3.7.5 Data Value Match Register (dvm)

The Data Value Match (dvm) register is part of on-core support for on-chip emulation. This register can be used by an on-chip emulation module, residing off-core, for a generation of a break-point on a data value match. A data value match occurs when the dvm register content is the same as the data on XDB/ZDB. In order to enable comparison for any transaction, and since the data on XDB/ZDB is not always transferred off-core, the dvm register is implemented as a part of the core.

This register is also used upon servicing the TRAP/BI routine: The pc content is transferred into the dvm in addition to the software stack. The dvm content can be transferred from/into the accumulators. Refer to *mov* instruction in chapter 4.

3.7.6 User-Defined Registers

The core supports up to 4 user-definable registers, which can be located off-core (in the COMBO). This feature enables future expansion of the core. These registers appear in the data register fields of all relevant instructions. While implementing these registers, external computation units can be loaded with data and read at the end of the computation, directly into internal registers in a single cycle. Operations such as parity calculation can easily be performed in a few cycles in parallel with the TeakLite. The hardware for performing such operations will be designed as part of the COMBO. Refer to Chapter 5, paragraph 5.8, User Defined Registers.

3.8 INPUT and OUTPUT

Memory mapped I/O is used, mapped in the data space. The major I/O operations are performed and controlled by the COMBO. Refer to paragraph 3.6.1 in this chapter.

Two special input bits and two output bits are available as status bits in status register st2. Conditional instructions can be executed according to the two input bits.

The optional external registers can also be used as direct and fast (single cycle) I/O. Refer to paragraph 3.7.6, User-Defined Registers.

BLANK PAGE

4. INSTRUCTION SET AND CODING

4.1 INTRODUCTION

This chapter is an overview and a detailed description of the TeakLite instruction set definition and coding, as well as a complete description regarding the function of each instruction. The pipeline method is covered briefly.

4.2 NOTATION AND CONVENTIONS

4.2.1 Notation

The instruction and register name are lower-case.

The following notations are used in this chapter:

Registers:

rN	= Address registers: r0, r1, r2, r3, r4, r5
rI	= Address registers: r0, r1, r2, r3
rJ	= Address registers: r4, r5
aX	= a0 or a1
aXl	= a-accumulator-low (LSP), X = 0, 1
aXh	= a-accumulator-high (MSP), X = 0, 1
aXe	= a-accumulator extension, X = 0, 1
bX	= b0 or b1
bXl	= b-accumulator-low (LSP), X = 0, 1
bXh	= b-accumulator-high (MSP), X = 0, 1
ac	= a0, a1, a0h, a1h, a0l, a1l
bc	= b0, b1, b0h, b1h, b0l, b1l
ab	= a0, a1, b0, b1
cfgX	= Configuration registers of DAAU (MODI or MODJ, STEPI or STEPJ), X = I, J
sv	= Shift Value register

Registers (Cont’d)

sp = Stack Pointer
pc = Program Counter
lc = Loop Counter (of block repeat)

extX = External registers, x = 0, 1, 2, 3 (user definable registers)

REG = a0, a1, a0h, a1h, a0l, a1l, b0, b1, b0h, b1h, b0l, b1l, r0, r1, r2, r3, r4, r5, rb, y, p or ph, sv, sp, pc, lc, st0, st1, st2, cfgi, cfgj, ext0, ext1, ext2, ext3.

x = x (multiplier input) register
mixp = Minimum/maximum pointer
icr = Internal Configuration Register
repc = Repeat Counter
dvm = Data Value Match register

Number representation: ___ decimal
0b___ , 0B___ binary
0x___ , 0X___ hexadecimal

Data and Program Operands:

The following table lists the data and the program operands : number of bits, operand range, including the assembler mnemonics and an example for each operand. See also Chapter 3, Section 3.6.3, Memory Addressing Modes.

Table 4-1 Operand Name, Size and Assembler Syntax

Operand	Number of bits	Assembler Syntax			Example
		Decimal	Hexadecimal	Binary	
Data Operands					
# signed short immediate	2's complement 8 bits	# -128 ... 127	# -0x80 ... 0x7f	# -0b10000000 ... 0b01111111	*mov #-12,r0*
# signed 6 bit immediate	2's complement 6 bits	# -32 ... 31	# -0x20 ... 0x1f	# -0b100000 ... 0b011111	*shfi b0,a0,#-4*
# signed 5 bit immediate	2's complement 5 bits	# -16 ... 15	# -0x10 ... 0xf	# -0b10000 ... 0b01111	*movsi r1,a0,#3*
# unsigned 9 bit immediate	unsigned 9 bits	# 0 ... 511	# 0x0 ... 0x1ff	# 0b0000000 ... 0b111111111	*load #270,modi*
# unsigned short immediate	unsigned 8 bits	# 0 ... 255	# 0x0 ... 0xff	# 0b0000000 ... 0b11111111	*add #0b10, a0*
# unsigned 7 bit immediate	unsigned 7 bits	# 0 ... 127	# 0x0 ... 0x7f	# 0b0000000 ... 0b01111111	*load # 3, stepj*
# unsigned 5 bit immediate	unsigned 5 bits	# 0 ... 31	# 0x0 ... 0x1f	# 0b00000 ... 0b011111	*mov #0x5, icr*
# unsigned 2 bit immediate	unsigned 2 bits	# 0 ... 3	# 0x0 ... 0x3	# 0b00 ... 0b11	*load #0b11, ps*
# bit number	unsigned 4 bits	# 0 ... 15	# 0x0 ... 0xf	# 0b0000 ... 0b01111	*tstb r0, #12*
## long immediate	2's complement 16 bits	## -32768 ... 32767	## -0x8000 ... 0x7fff	## -0b1000..00 ...0b01111..111	*mov ## -0x6000,a0*
##offset	unsigned 16 bits	## 0 ... 65535	## 0x0... 0xffff	## 0b0 ... 0b1111..111111	*mov ## 0xf000,r0*
#offset7	2's complement 7 bits	-64 ... 63	-0x40 ... 0x3f	-0b1000000 ... 0b0111111	*add (rb-5),a1*

Table 4-1 Operand Name, Size and Assembler Syntax (Cont'd)

Operand	Number of bits	Assembler Syntax			Example
		Decimal	Hexadecimal	Binary	
Program Operands					
Direct address [direct address]	unsigned 8 bits (offset in page)	0 ... 255	0x0 ... 0xff	0b0000000 ... 0b11111111	*add 120, a1*
[##direct address]	unsigned 16 bits	0 ... 65535	0x0 ... 0xffff	0b0 ... 0b1111..111111	*sub [##var1],a0*
Address	unsigned 16 bits	0 ... 65535	0x0... 0xffff	0b0 ... 0b1111..111111	*call 0x5000*

(*) Negative numbers can also be written as four hexadecimal digits. For example: -0x80 can be written as 0xff80; -0x20 can be written as 0xffe0.

Option fields:

eu- Extension unaffected. Optional field in the *mov direct address, axh,[eu]* instruction.

When mentioned, the data is transferred into aXh without affecting aXe. When not mentioned, the data is transferred into aXh with sign-extension into aXe.

context - Context switching. Optional field in the *reti* instruction.

When mentioned, it means automatic context switching.
When not mentioned, it means without context switching.

dmod - Disable modulo. Optional field in the *modr* instruction.

When mentioned, the rN is post-modified with modulo modifier disabled. When not mentioned, the post-modification of rN is affected by the Mn bit.

Table 4-2 Condition field (*cond*)

Mnemonics	Description	Condition
true	Always	
eq	Equal to zero	$Z = 1$
neq	Not equal to zero	$Z = 0$
gt	Greater than zero	$M = 0 \cap Z = 0$
ge	Greater or equal to zero	$M = 0$
lt	Less than zero	$M = 1$
le	Less or equal to zero	$M = 1 \cup Z = 1$
nn	Normalize flag is cleared	$N = 0$
v	Overflow flag is set	$V = 1$
c	Carry flag is set	$C = 1$
e	Extension flag is set	$E = 1$
l	Limit flag is set	$L = 1$
nr	R flag is cleared	$R = 0$
niu0	Input user pin 0, IUSER0, is cleared	
iu0	Input user pin 0, IUSER0, is set	
iu1	Input user pin 1, IUSER1, is set	

Other:

(x)	= Contents of x
|	= One of the options should be included
[ ]	= Optional field in the instruction
<x>	= Specific notes
→	= Is assigned to
>>	= Shift right
<<	= Shift left

exp(x) = Exponent of x
_ = Not
∪ = Or
∩ = And

Flags Notation:

The effect of each instruction on the flags is described by the following notation:

*	The flag is affected by the execution of the instruction.
-	The flag is not affected by the instruction.
1 or 0	The flag is unconditionally set or cleared by the instruction.

st0 bits	11	10	9	8	7	6	5	4
Flag	Z	M	N	V	C	E	L	R

For flag definitions, refer to Section 3.7.2.2, Status Register Field Definitions.

4.2.2 Conventions

a. The arithmetic operations are performed in 2's complement.

b. **Post-modification** of **rN** registers is possible in the following instructions:
 - instructions which use an indirect addressing mode
 - *modr*
 - *norm*
 - *max, maxd, min* (use r0 only)

In the above mentioned instructions, the contents of rN register is post-modified as follows:
Options controlled by instruction:
rN, rN+1, rN-1, rN+step
Options controlled by configuration registers cfgX:
Step size: STEPI, STEPJ - 2's complement 7 bits (-64 to 63)
Modulo size: MODI, MODJ - unsigned 9 bits (1 to 512)

Options controlled by st2:
For each rN register, the option should be defined whether modulo is enabled or disabled. A default option during reset is a disable modulo. In case MODI or MODJ are used, the relative Mn bit must be set, (the only exception for this is in the *modr* instruction, when an optional field for disabling the modulo is applied) For further details as regards the modulo arithmetic unit refer to section 3.4.1.2, Modulo Modifier.

Whenever the operand field in the instruction includes the option of (rN), it means that the rN can be post-modified in one of the four options.
Assembler syntax: (rN) , (rN)+, (rN)- , (rN)+s
For example: *mov (r0)-,r1 ; mac (r4)+,(r0)+s,a0 ; add (r2),a1 ; modr (r5)-*

c **Direct addressing** mode assembler syntax:
The syntax when a **1** word instruction is used is either *direct address* or *[direct address]*. (This instruction uses the page bits contents).
The syntax when a **2** word instruction is used (even if the address is an 8 bit value) is *[## direct address]*.

d. The MSP of the p register (**ph**) is a write only register. The 32-bit **p register** is updated after a multiplication. It can be read only by transferring it to the ALU, that is, it can be moved into aX or be an operand for arithmetic and logic operations. When transferring it into the ALU, it is sign-extended to 36 bits. This enables the user to store and restore the p register.

e. The **p register** is used as a **source** operand for different instructions as described below: As one of the REG registers; in the *moda* instruction - *pacr* function; in multiplication instructions where the p register is added or subtracted from one of the accumulators. When using the p register as a source operand, it always means using the **'shifted p register'**. The shifted p register means that the p register is sign-extended into 36 bit and then shifted as defined in the PS field, status register st1. In right shifts the sign is extended, whereas in left shift, a zero is appended to the LSB. The contents of the p register remain unchanged. In two multiplication instructions, *maa* and *maasu*, the p register is also aligned, i.e. after the p register is sign-extended and shifted according to the PS field, it is also shifted by 16 to the right.

e. All move instructions using the accumulator (**aX** or **bX**) as destination cause sign extension. All instructions that use the accumulator-low (**aXl** or **bXl**) as a destination, will clear the accumulator-high and the accumulator-extension. Therefore, they are sign extension suppressed. All instructions using the accumulator-high (**aXh** or **bXh**) as a destination, will clear the accumulator-low and are sign extended. An exception is *mov direct address ,axh,[eu]*, when the 'eu' option is used, it moves data into accumulator-high without sign extension (the accumulator-extension aXe is unaffected).

f. In all **arithmetic operations** between **16-bit registers and aX** (36 bits), the 16-bit register will be regarded as the 16 low-order bits of a 36-bit operand that is sign extended in the MSBs.

g. It is recommended that the **flags** be used immediately after the instruction that updated them. Otherwise, careful programming is required (some flags may be changed in the meantime).

h. The **condition** field is almost always an optional field, except when the condition field is followed by another optional field as in the *reti* instruction. When the condition field is the last field of the instruction, then when the condition is missing, the condition is **true**.
Examples: *shr4 true* is the same as *shr4* ,
but in *reti true, context* the *true* cannot be omitted.

i. General Restrictions:

I. Arithmetic and logic operations (but not bit manipulation operations) must not be performed with the **same accumulator** as the source (soperand) and the destination (doperand).

Example: *add a0,a0* is not allowed (*shfc a0,a0* is allowed).

II. An instruction, which immediately follows an instruction that modifies the *rb* register, may not use the index addressing mode. The only exception is when *rb* is modified using a long immediate operand (*mov ##long immediate, rb*).

j. Whenever a register is written and the same register is used in the next instruction (except for *movd* instructions) for address generation, the next instruction is a two cycle instruction. Re-ordering the instructions can be used to avoid this restriction.

k. Two *nop* instructions should follow instructions which use the pc as a destination register, except after *mov ##long immediate, pc*, where only one *nop* is needed.

l. PSYNC_FAST is an internal switch within the hardware design (RTL code) and in the Software Development Tools (Debugger, Assembler) used in order to improve the timing of the PPAP (Program Address) output bus of the core, this makes it easier (i.e. less timing critical) to connect the core to synchronous memories. The PSYNC_FAST switch can only be used in conjunction with the PSYNC switch (which is an RTL switch only), and is not applicable otherwise. However using the PSYNC_FAST option imposes an extra cycle to some instructions that breaks the pipeline, as detailed in section 4.4 on Instruction Set Details. This means that the usage of the PSYNC_FAST switch is recommended for high frequency TeakLite based systems only. Please refer to Chapter 5 on Core Interface for more details.

m. Whenever an instruction writes to one of the accumulators or part of it (an exception is when moving long immediate data value into the accumulator), and the next instruction uses part of this accumulator (low or high) as an information to decide on the next program address value (instructions - calla, mov abXl/h,PC, movp (aXl),REG, rep abXl/h), this next instruction is extended by one more cycle . Re-ordering of the instructions can be used to avoid this extra cycle. This convention is relevant only when the PSYNC_FAST switch is set, otherwise, the next instruction remains with the same cycle count.

4.3 INSTRUCTION SET SUMMARY

Following is a summary of the TeakLite instructions set organized by instruction groups and by alphabetical order. Most of the Teaklite instructions are performed within a single cycle. The instruction time is denoted by Xc, where X is the number of cycles. Instructions that have a long immediate, long direct, or long index operand, as one of the operand options, require an extra cycle while using such operand, and thus denoted in particular. Some instructions that break the pipeline have different execution timing
whether the condition set for is met or not (e.g. call instruction), these instructions are marked appropriately.

4.3.1 Instruction Set Summary by Instruction Group

Following is a summary of the TeakLite instruction set organized in instruction groups. The last column indicates the relevant page in Chapter 4 where further details are given.

ARITHMETIC AND LOGIC INSTRUCTIONS		Cycles	Page
add	Add	1c / long in 2c	4-26
sub	Subtract	1c / long in 2c	4-130
or	OR	1c / long in 2c	4-101
and	And	1c / long in 2c	4-31
xor	Exclusive - OR	1c / long in 2c	4-140
cmp	Compare	1c / long in 2c	4-48
add1	Add to low accumulator	1c	4-28
sub1	Subtract from Low Accumulator	1c	4-132
addh	Add to High Accumulator	1c	4-27
subh	Subtract from High Accumulator	1c	4-131
cmpu	Compare Unsigned	1c	4-49
addv	Add Long Immediate value to register or a data memory location	2c	4-29
subv	Subtract Long Immediate Value from a Register or a Data Memory Location	2c	4-133
cmpv	Compare Long immediate Value to a Register or a Data Memory Location	2c	4-50
moda	Modify aX-Accumulator Conditionally Modifications:	1c	4-73

shr	-	Shift Right
shr4	-	Shift Right by 4
dh1	-	Shift Left
shl4	-	Shift Left by 4
ror	-	Rotate Right through Carry
not	-	Logical Not
neg	-	2’s Complement
clr	-	Clear
copy	-	Copy other Accumulator

moda (Cont'd)

Modifications:

rnd	-	Round Upper 20 bits
pacr	-	Product Move and Round
clrr	-	Clear and Round
inc	-	Increment by One
dec	-	Decrement by One

		Cycles	Page
norm	Normalize	2c	4-98
divs	Division Step	1c	4-55
max	Maximum between Two aX-Accumulators	1c	4-69
maxd	Maximum between data Memory Location and aX-Accumulators	1c	4-70
min	Minimum between Two aX-Accumulators	1c	4-72
lim	Limit aX-Accumulator	1c	4-59

MULTIPLY INSTRUCTIONS

		Cycles	Page
mpy	Multiply	1c / long in 2c	4-91
mpysu	Multiply Signed by Unsigned	1c / long in 2c	4-94
mac	Multiply and Accumulate Previous Product	1c / long in 2c	4-64
macsu	Multiply Signed by Unsigned and Accumulate Previous Product		4-65
macus	Multiply Unsigned by Signed and Accumulate Previous Product	1c / long in 2c	4-66
macuu	Multiply Unsigned by Unsigned and Accumulate Previous Product	1c / long in 2c	4-67

		Cycles	Page
maa	Multiply and Accumulate Aligned Previous Product	1c / long in 2c	4-62
maasu	Multiply Signed by Unsigned and Accumulate Aligned Previous Product	1c / long in 2c	4-63
msu	Multiply and Subtract Previous Product	1c / long in 2c	4-95
mpyi	Multiply Signed Short Immediate (mpys)	1c	4-92
sqr	Square	1c	4-128
sqra	Square and Accumulate Previous Product	1c	4-129

BMU INSTRUCTIONS

set	set bit-field	2c	4-118
rst	Reset bit-field	2c	4-116
chng	Change bit-field	2c	4-44
tst0	Test bit-field for Zeros	1c/2c	4-137
tst1	Test bit-field for Ones	1c/2c	4-138
tstb	Test Specific Bit	1c	4-139
shfc	Shift Accumulators according to Shift Value Register Conditionally	1c	4-120
shfi	Shift Accumulators by an Immediate Shift Value	1c	4-122
modb	Modify bX-Accumulator Conditionally Modifications: *shr* - Shift Right *sh4* - Shift Right by 4 *sh1* - Shift Left *sh14* - Shift Left by 4 *ror* - Rotate Right through Carry *rol* - Rotate Left through Carry *clr* - Clear	1c	4-77
exp	Evaluate the Exponent Value	1c	4-57

MOVE INSTRUCTIONS

		Cycles	Page
mov	Move Data	1c / long in 2c *	4-81
movp	Move from Program Memory into Data Memory	3c	4-86
movd	Move from Data Memory into Program Memory	4c	4-85
movs	Move and Sift according to Shift Value Register	1c	4-88
movsi	Move and Shift according to an Immediate Shift Value	1c	4-90
movr	Move to Accumulator and Round	1c	4-87
push	Push Register or Long Immediate Value onto Stack	1c	4-104
pop	Pop from Stack into Register	1c *	4-103
swap	Swap Ax Register and bX accumulators	1c	4-135
banke	Bank Exchange	1c	4-33

LOOP INSTRUCTIONS

rep	Repeat Next Instruction	1c *	4-105
bkrep	Block Repeat	2c	4-34
break	Break from a Block-Repeat	1c	4-38

BRANCH / CALL INSTRUCTIONS *

br	Conditional Branch	2c / 3c cond. met[1]	4-37
brr	Conditional Relative Branch	2c	4-39
call	Conditional call Subroutine	2c / 3c cond. met[1]	4-40
callr	Relative Conditional Call Subroutine	2c	4-42
calla	Call Subroutine a location Specified by aX Accumulator	3c	4-41
ret	Return Conditionally	2c / 3c cond. met[1]	4-107
retd	Delayed Return	1c	4-108
reti	Return form Interrupt Conditionally	2c / 3c cond. met[1]	4-109
retid	Delayed Return form Interrupt	1c	4-110
rets	Return and Adjust Stack Pointer with a short Immediate Offset	3c	4-112

CONTROL AND MISCELLANEOUS INSTRUCTIONS		Cycles	Page
nop	No Operation	1c	4-97
modr	Modify rN	1c	4-80
eint	Enable Interrupt	1c	4-56
dint	Disable Interrupt	1c	4-54
trap	Software Interrupt	2c	4-136
load	Load Specific Fields into Registers modXm stepX, ps, page, (1pg)	1c	4-60
cntx	Context Switching Store or Restore	1c	4-51

cond. met[1] means - 3 cycles instruction when the condition is met.

* - Refer to 4.4 INSTRUCTION SET DETAILS for cycle count when using PSYNC_FAST mode.

4.3.2 Instruction Set Summary by an Alphabetical Order

Following is a summary of the TeakLite instruction set presented in alphabetical order. This is the same order as the one used in the instruction details, given in this chapter. The last column indicates the number of words specific instruction requires.

Table 4-3 TeakLite Instruction Set Summary

Inst.	Description	Cycles	Words
add	Add	1 /long in 2c	1 /long in 2w
addh	Add to High Accumulator	1	1
addl	Add to Low Accumulator	1	1
addv	Add Long Immediate Value to a Register or a Data Memory Location	2	2
and	AND	1 /long in 2c	1 /long in 2w
banke	Bank Exchange	1	1
bkrep	Block Repeat	2	2
br	Conditional Branch *	2 cond. not met 3 cond. Met	2
break	Break from Block-Repeat	1	1
brr	Conditional Relative Branch *	2	1
call	Conditional Call Subroutine *	2 cond. not met 3 cond met	2
calla	Call Subroutine at Location Specified by the aX-Accumulator	3	1
callr	Relative Conditional Call Subroutine *	2	1
chng	Change Bit-field	2	2
clr	Clear Accumulator (moda/modb)	1	1
clrr	Clear and Round aX-Accumulator (moda)	1	1
cmp	Compare	1 /long in 2c	1 /long in 2w

Table 4-3 TeakLite Instruction Set Summary (Cont'd)

Inst.	Description	Cycles	Words
cmpu	Compare Unsigned	1	1
cmpv	Compare Long Immediate Value to a Register or a Data Memory Location	2	2
cntx	Context Switching Store or Restore	1	1
copy	Copy other aX-Accumulator (moda)	1	1
dec	Decrement aX-Accumulator by One (moda)	1	1
dint	Disable Interrupt	1	1
divs	Division Step	1	1
eint	Enable Interrupt	1	1
exp	Evaluate the Exponent Value	1	1
inc	Increment aX-Accumulator by One (moda)	1	1
lim	Limit aX-Accumulator	1	1
load	Load Specific Fields into Registers	1	1
lpg	Load The Page Bits (load)	1	1
maa	Multiply and Accumulate Aligned Previous Product	1 /long in 2c	1 /long in 2w
maasu	Multiply Signed by Unsigned and Accumulate Aligned Previous Product	1 /long in 2c	1 /long in 2w
mac	Multiply and Accumulate Previous Product	1 /long in 2c	1 /long in 2w
macsu	Multiply Signed by Unsigned and Accumulate Previous Product	1 /long in 2c	1 /long in 2w
macus	Multiply Unsigned by Signed and Accumulate Previous Product	1 /long in 2c	1 /long in 2w
macuu	Multiply Unsigned by Unsigned and Accumulate Previous Product	1 /long in 2c	1 /long in 2w

Table 4-3 Teaklite Instruction Set Summary (Cont'd)

Inst.	Description	Cycles	Words
max	Maximum between Two aX-Accumulators	1	1
maxd	Maximum between Data Memory Location and aX-Accumulator	1	1
min	Minimum between Two aX-Accumulators	1	1
moda	Modify aX-Accumulator Conditionally	1	1
modb	Modify bX-Accumulator Conditionally	1	1
modr	Modify rN	1	1
mov	Move Data *	1 /long in 2c	1 /long in 2w
movd	Move from Data Memory into Program Memory	4	1
movp	Move from Program Memory into Data Memory	3	1
movr	Move to Accumulator and Round	1	1
movs	Move and Shift according to Shift Value Register	1	1
movsi	Move and Shift according to an Immediate Shift Value	1	1
mpy	Multiply	1 /long in 2c	1 /long in 2w
mpyi *mpys*	Multiply Signed Short Immediate	1	1
mpysu	Multiply Signed by Unsigned	1 /long in 2c	1 /long in 2w
msu	Multiply and Subtract Previous Product	1 /long in 2c	1 /long in 2w
neg	2's Complement of aX-Accumulator (moda)	1	1
nop	No Operation	1	1
norm	Normalize	2	1

Table 4-3 Teaklite Instruction Set Summary (Cont'd)

Inst.	Description	Cycles	Words
not	Logical Not (moda)	1	1
or	OR	1 /long in 2c	1 /long in 2w
pacr	Product Move and Round to aX-Accumulator (moda)	1	1
pop	Pop from Stack into Register	1 *	1
push	Push Register or Long Immediate Value onto Stack	1 /long in 2c	1 /long in 2w
rep	Repeat Next Instruction	1 *	1
ret	Return Conditionally	2cond not met 3 cond met *	1
retd	Delayed Return	1 *	1
reti	Return from Interrupt Conditionally	2cond not met 3 cond met *	1
retid	Delayed Return from Interrupt	1 *	1
rets	Return and Adjust Stack Pointer with a Short Immediate Offset	3 *	1
rnd	Round Upper 20 bits of aX-Accumulator (moda)	1	1
rol	Rotate Accumulator Left through Carry (moda/modb)	1	1
ror	Rotate Accumulator right Carry (moda/modb)	1	1
rst	Reset Bit-field	2	2
set	Set Bit-field	2	2
shfc	Shift Accumulators according to Shift Value Register Conditionally	1	1
shfi	Shift Accumulators by an Immediate Shift Value	1	1

Table 4-3 Teaklite Instruction Set Summary (Cont'd)

Inst.	Description	Cycles	Words
shl	Shift Left (moda/modb)	1	1
shl4	Shift Left by 4 (moda/modb)	1	1
shr	Shift Right (moda/modb)	1	1
shr4	shift Right by 4 (moda/modb)	1	1
sqr	Square	1	1
sqra	Square and Accumulate Previous Product	1	1
sub	Subtract	1 /long in 2c	1 /long in 2w
subh	Subtract from High Accumulator	1	1
subl	Subtract from Low Accumulator	1	1
subv	Subtract Long Immediate Value from a Register or a Data Memory Location	2	2
swap	Swap aX and bX Accumulators	1	1
trap	Software Interrupt	2	1
tst0	Test Bit-Field for Zeros	1 /long in 2c	1 /long in 2w
tst1	Test Bit-Field for Ones	1 /long in 2c	1 /long in 2w
tstb	Test Specific Bit	1	1
xor	Exclusive-Or	1 /long in 2c	1 /long in 2w

*** -** Refer to 4.4 INSTRUCTION SET DETAILS for cycle count when using PSYNC_FAST mode.

4.4 INSTRUCTION SET DETAILS

This section contains a detailed description of each instruction. It includes the instruction syntax, description of operation, operand details, effect on flags, number of execution cycles, and other relevant notes and exceptions.

The coding of each instruction can be found in section 4.5. The order in which instructions are presented, is as per section 4.3, Instruction Set Summary.

4.4.1 Instruction Example

Following is an example of the terminology used in the instruction set. The specific example given is that of the *add* instruction.

For directives and operators supported by the assembler and linker refer, to " Teaklite Assembler and Linker User Guide".

Add **Add** ← Instruction mnemonic and description

add operand , aX ← Instruction mnemonics

Operation: aX + operand → aX

Source operand 1	Source operand 2	Destination operand

The instruction has **two source operands**, which are added in the ALU, the ALU output is latched into the **destination operand**.

In this instruction, the flags are affected according to the ALU output, they are reflecting the status of the destination accumulator.

'aX' means one of the accumulators a0 or a1. This accumulator is both source operand and destination operand.

The **'Operand'** field, in this instruction, is the other source operand, and can be one of the following options;

operand: REG <1>
(rN)
direct address
[##direct address]
#unsigned short immediate
##long immediate
(rb+#offset7)
(rb+##offset)

REG -

Unless otherwise specified REG is one of the 34 TeakLite registers : a0, a1, a0h, a1h, a0l, a1l, b0, b1, b0h, b1h, b0l, b1l, r0, r1, r2, r3, r4, r5, rb, y, p or ph, sv, sp, pc, lc, st0, st1, st2, cfgi, cfgj, ext0, ext1, ext2 and ext3. The contents of the source register is added to the accumulator. The operation result, the ALU output, is placed at the accumulator.

Example:
add r1,a0

	Before execution	After execution
a0	0x1001	0x1008
rl	0x7	0x7

Unsigned Short Immediate - #Unsigned Short Immediate

The 8-bit (positive number) is one of the source operands. The 8-bit value is added, right-justified, to the accumulator. The operation result i.e., the ALU output, is placed in the accumulator.

Example:
add #255,a0

	Before execution	After execution
a0	1001	1256

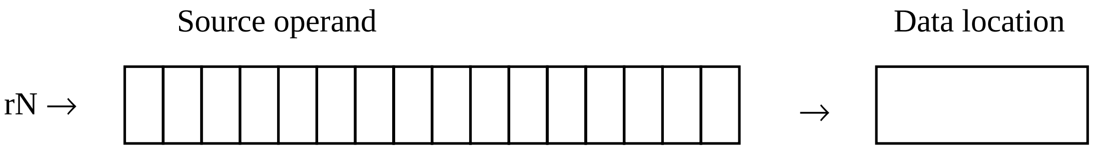
Long Immediate - ##Long Immediate

The 16-bit value is one of the source operands. This value is added, right-justified and sign-extended, to the accumulator. The operation result, i.e., the ALU output, is placed in the accumulator.

Example:
add ##0xffff,a1

	Before execution	After execution
a1	0x20	0x1F

Indirect addressing - (rN), (rN)+, (rN)-, (rN)+s



One of the DAAU registers (r0, r1, r2, r3, r4, r5) points at one of the 64k data words. The data location contents, pointed at by the register, is the source operand - added to the accumulator. The operation result, i.e., the ALU output, is placed into the accumulator.

The rN register is modified in the following way after the instruction is:
(rN) - no update
(rN)+ - rN is auto incremented
(rN)- - rN is auto decremented
(rN)+s - rN is auto incremented/auto decremented by the offset s

Each of these modifications can use the modulo option. For further details regarding the post-modification, see paragraph 4.2.2 (2).

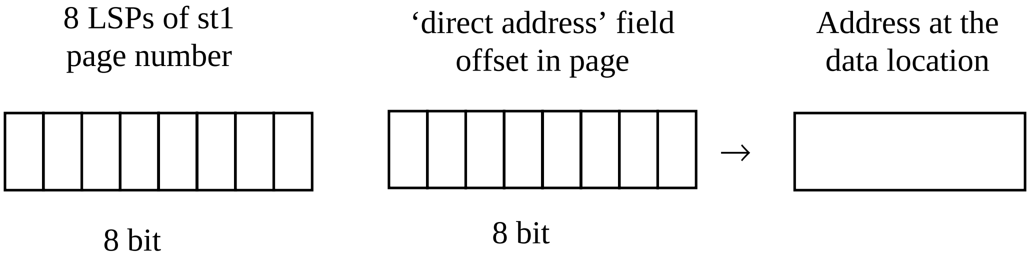
Example:
add (r1)+s,a0

	Before execution	After execution
a0	0x1001	0x1101
rl	7	9
Data location 7	0x100	0x100
cfgi meaning S = 2	2	2

St2 meaning no modulo option for rl	ϕϕϕ0 ϕ - don't care	ϕϕϕ0

Short Direct Address - direct address/[Direct address]

The data location ,i.e., one of the 64k data words, is one of the source operands. The 16-bit data location contains the page number in st1 register and the 'direct address' field - the offset in the page. The data location contents is added to the accumulator. The operation result from the ALU output, is placed in one of the accumulators.



Example:

add @Var2,a1 (or *add [2],a1* or *add 2,a1)*

Assuming *Var2* is a variable whose address equals 0x0102. For @ operator refer to "TeakLite Assembler and Linker User Guide".

	Before execution	After execution
a1	0xFFFFFFFFF	0
st1 - meaning page 1	0xF301	0x0301
Data location 0x102	1	1

Long Direct Address - [##direct address]

The data location specified as the second word of the instruction, is one of the source operands. The 16-bit data location is one of any of the 64k data words. (page bits are not used in this mode).

Example:

add [##Tmp],a0 (or *add [##0x1050],a0)*

Assuming *Tmp* is a variable whose address is equal to 0x1050.

a0	0xFFFFFFFFFF	0
Data location 0x1050	1	1

Short Index Addressing Mode - (rb+#offset7)

The offset 7 value is added to the rb register contents and together they generate an address of one of the 64k data words. This data memory location is the source operand, which is added to the accumulator. Offset7 is a 2's complement 7 bit number, between -64 to 63. The base register, rb, is unaffected by this operation.

Example: add (rb-#20), a0

	Before execution	After execution
a0	0x6000	0x5FFF
rb	50	50
	0xFFFF	0xFFFF

Long index addressing mode- (rb+##offset)

The 16 bit offset value is added to the *rb* register contents and together they generate an address of one of the 64k data words. This data memory location is the source operand added to the accumulator. Offset is a 2's complement 16 bit number. The base register, *rb*, is unaffected by this operation.

Example:
add (rb+##0x1000),a1

	Before execution	After execution
a1	0x6000	0x6010
rb	0x0050	0x0050
Data location 0x1050	0x0010	0x0010

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

From the 'affects flags' field in this example, it can be observed that all the flags are affected, except for the R flag. See paragraph 3.7.2.2, Status Register Field Definition.

Cycles: Number of cycles.
Cycle means a machine cycle - one clock cycle, which can be stretched to more than one in case of wait states. It is stretched until the end of the wait interval.

Words: Number of program words.

4.4.2 Instructions Details

The following is a detailed description of the TeakLite instructions, presented in alphabetical order.

add Add add

add operand , aX

Operation: aX + operand → aX

operand:		
	REG	<1>
	(rN)	
	direct address	
	[##direct address]	
	#unsigned short immediate	
	##long immediate	
	(rb+offset7)	
	(rb+##offset)	

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1
2 when the instruction is a two words instruction

Words: 1
2 when the operand is: *##long immediate* or *(rb+##offset) or [##direct address]*

<1> The REG cannot be bX.

addh Add to High Accumulator addh

addh operand , aX

Operation: aX + operand*2^16 → aX
The aXl is unaffected.

operand: REG <1>
(rN)
direct address

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1

Words 1

<1> The REG cannot be: aX, bX, p.

addl Add to Low Accumulator addl

addl operand ,aX

Operation: aX + operand → aX
The operand is sign-extension suppressed.

operand: REG <1>
(rN)
direct address

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1

Words: 1

<1> The REG cannot be: aX, bX, p.

addv Add Long Immediate Value to a Register or a Data Memory Location addv

addv ##long immediate , operand

Operation: operand + ##long immediate → operand
The operand and the long immediate values are sign-extended . If the operand is not a part of an accumulator (aXl, aXh, aXe, bXl, bXh) then the accumulators are **unaffected**. If the operand is a part of an accumulator, only the addressed part is affected.

operand: REG <1>
(rN)
direct address

Affects flags: When the operand is not st0: <2> , <3>

Z	M	N	V	C	E	L	R
*	*	-	-	*	-	-	-

Z, M, C are a result of the 16-bit operation.
M is affected by bit 15.

When the operand is **st0**:
st0 (including the flags) accepts the addition result, regardless of a0e bits.

Cycles: 2

Words: 2

<1> The REG cannot be : aX, bX, p, pc.
Note that aX can be used in *add ##long immediate,aX* instruction.

<2> When adding a long immediate value to st0, st0 (including the flags) accepts the ALU output. When adding a long immediate value to st1, the flags are affected by the ALU output, as usual.

addv Add Long Immediate Value to a Register or a Data Memory Location (Continued) addv

<3> Note that when the operand is a part of an accumulator, only the addressed part is affected.
For example, if the instruction *addv ##long immediate, a0l* generates a carry, the carry flag is set though, a0h is unchanged. On the other hand, the instruction *addl ##long immediate, a0l* (with same a0 and immediate values) changes the a0h and affects the carry flag according to bit 36 of the ALU output.

and AND and

and operand , aX

Operation: If operand is aX or shifted p

```
aX[35-0] AND operand → aX[35-0]

If operand is unsigned short immediate
  aX[7-0 ] AND operand → aX[7-0]
    aX[15-8] →aX[15-8]          <1>
    0 → aX[35-16]

  If operand is REG, (rN), long immediate
    aX[15-0] AND operand → aX[15-0]
    0 → aX[35-16]
```

It should be noted that if the operand is one of the a-accumulators or the shifted p register it is ANDed with the destination accumulator.

If the operand is short immediate, the operand is zero-extended to form a 36-bit operand, then ANDed with the destination accumulator. Bits15-8 are unaffected; other bits of the accumulator are cleared. <1>

If the operand is a 16-bit register or a long immediate value, the operand is zero-extended to form a 36-bit operand, then ANDed with the accumulator. Hence, the upper bits of the accumulator are cleared by this instruction.

operand: REG <2>
(rN)
direct address
[##direct address]
#unsigned short immediate
##long immediate
(rb+offset7)
(rb+##offset)

and AND (Continued) and

Affects flags:

Z	M	N	V	C	E	L	R
*	*	-	-	-	*	-	-

Z flag is set if all the bits at the ALU output are zeroed, otherwise cleared. Note that when the operand is unsigned short immediate, ALU output bit[35:8] = 0 bits[7:0] = aX[7:0] AND operand

Cycles: 1
2 when the instruction is a two words instruction

Words: 1
2 when the operand is: *##long immediate* or
(rb+##offset) or *[##direct address]*

<1> The instruction *and #unsigned short immediate,aX* can be used for the purpose of clearing some of the low-order bits at a 16-bit destination.
For example: *mov ram, aX*
and #unsigned short immediate, aX mov aX, ram

Using the *and* instruction, bits 15-8 are unaffected, therefore the high-order bits at the destination do not change.

See also the ***rst* instruction**.

In addition, this instruction can be used for BIT TEST, test one of the low-order bits of a destination accumulator (e.g. at accumulator-low)
For example: *and #unsigned short immediate, aX*
br address, eq or *br address, neq* (check the zero flag)

See also the ***tstb* instruction**.

<2> The REG cannot be bX.

banke Bank Exchange banke

banke [r0] [,r1] [,r4] [,cfgi] <1>

Exchange the registers that appear in the exchange list.

Affects flags: No

Cycles: 1

Words: 1

<1> The number and order of the above mentioned registers,(on the exchange list), may vary. Following are some Assembler syntax examples:

banke r0
banke r1,cfgi
banke r1,r0
banke cfgi,r1,r4

<2> For further details, refer to paragraph 3.4.3, Alternative Bank of Registers.

bkrep Block Repeat bkrep

bkrep operand , add

Operation: operand → lc <2>
1 → LP status bit
BCx + 1 → BCx

Begins a block repeat that is to be repeated operand+1 times.

The repetition range is from 1 to 65536.

The first block address is the address after the *bkrep* instruction, and the last block address is the address specified by the 'add' field. <3>

The operand is inserted into the loop counter register (lc). The inloop status bit LP is set - indicating a block repeat loop. The block repeat nesting level counter is incremented by one.

The repeated block is interruptible.

operand: #unsigned short immediate	<2>
REG	<4>,<5>

Affects flags: No

Cycles: 2

Words: 2

<1> This instruction can be nested. Four levels of block repeat can be used.

<2> When using an unsigned short immediate operand, the number of repetitions is between 1 to 256. When transferring the #unsigned short immediate number into the lc Register, it is copied to the low-order 8 bits of the lc. The high-order 8 bits are zero-extension of the low-order bits.

bkrep Block Repeat (Cont'd) bkrep

<3> In case the last instruction at the block repeat is:

a. One word instruction - 'add' is the address of this instruction.

b. Two words instruction - 'add' is the address of the second word of the instruction.

<4> In the block-repeat outer level - the REG cannot be aX, bX, p.

In other block-repeat levels - the REG cannot be aX, bX, p, lc. Note that the Assembler can not check the restriction on lc Register in a nested block-repeat.

<5> The data read while reading lc during a block repeat loop execution is of the loop counter. If the outer block repeat loop has normally completed its turn with the contents of lc of 0; if it was completed using *break*, the contents of lc will be the value of the loop counter at the break point.

<6> The block-repeat minimum length is 2 words.

<7> Restrictions:

1. *Break* cannot start at the last address of the block repeat loop :.

2. The following instructions cannot start at (last address-1) of the block repeat loop : *mov* soperand, icr ; *mov* icr, ab.

3. The following instructions cannot start at the two last addresses of the block repeat loop : *br, brr, call, callr, calla, ret, reti, rets, retd, retid, bkrep, rep, trap* instruction with pc or lc as a destination, instruction with lc or icr as a source.

4. The following instructions cannot start at (last address-2) of the block repeat loop : instructions with lc as a destination, *mov* soperand, icr.

bkrep Block Repeat (Continued) bkrep

5. The following instructions cannot start at (last address-3) of the block repeat loop: *set/rst/chng/addv/subv* with lc as a destination.

6. Note that illegal instruction sequences are also restricted as the last and first instructions of a block-repeat loop.

7. Two block-repeat loops cannot have the same last address.

8. The following instruction cannot start at the three last addresses of the block repeat loop: *mov soperand* , *pc* (including *pop pc* and *movp(aXl),pc*)

br Conditional Branch br

br address [, cond]

Operation: If condition then
address → pc

If the condition is met, branch to the program memory location specified by 'address'.

Affects flags: No

Cycles:
If PSYNC_FAST,
3 if branch did not occur
4 if branch occur

Else: 2 if branch did not occur
3 if branch occur

Words: 2

<1> If the condition is met, 'address' is the address/label of the new program memory location. The 'address' is the second word of the instruction.

break Break from Block-Repeat break

break

This instruction is used in order to break out of the current block-repeat loop. The internal registers, which contain the first address, last address and loop-counter are popped.

Affects flags: No

Cycles: 1

Words: 1

<1> The *break* instruction cannot be the last instruction of a block-repeat loop.

<2> A *break* at the outer level does not change lc and resets LP bit.

brr Relative Conditional Branch brr

brr relative address [,cond]

Operation: If condition then
'address of brr inst.'+ relative address + 1 → pc

If the condition is met, a branch is performed on program memory locations: *brr* instruction' + 'relative address' + 1
The offset range is -63 to 64. (Offset range is 'relative address'+1)

Affects flags: No

Cycles:	
If PSYNC_FAST, then:	3
Else:	2
Words:	1

<1> Assembler syntax:

brr relative address [,cond]
or
brr $-relative address [,cond]

or
brr label [,cond]

Where 'label' is the new program memory location. The instruction word includes the 'relative address' calculated by the Assembler as follows:
(label address) - (*brr* address) - 1.

call Conditional Call Subroutine call

call address [,cond]

Operation: If condition then

```
sp - 1    → sp
pc        → (sp)
address   → pc
```

If the condition is met, the stack pointer (sp) is pre-decrement, the program counter (pc) is pushed into the software stack and a branch is performed on the program memory location specified by 'address'.

Affects flags: No

Cycles:

If PSYNC_FAST,

3 if condition not met
4 if condition met

Else:

2 if condition not met
3 if condition met

Words: 2

<1> If the condition is met, 'address' is the address/label of the new program memory location. The 'address' is the second word of the instruction.

calla Call Subroutine at Location Specified by the aX-Accumulator calla

calla aXl

Operation:	sp -1 → sp pc → (sp) aXl → pc
	Call subroutine indirect (address from aXl). The stack pointer (sp) is pre-decrement. The program counter (pc) is pushed into the software stack and a branch is executed to the address pointed by accumulator-low. This instruction can be used to perform computed subroutine calls.
Affects flags:	No
Cycles:	3
Words:	1

callr Relative Conditional Call Subroutine callr

callr relative address [, cond]

Operation: If condition then
sp - 1 → sp
pc → (sp)
'the *callr* inst.' + relative address + 1 → pc

If the condition is met, the stack pointer (sp) is pre-decrement, the program counter (pc) is pushed into the software stack and a branch is executed to the following program memory location:
'the *callr* instruction' + 'relative address' + 1 The offset range is -63 to 64. (Offset range is 'relative address'+1).

Affects flags: No

Cycles:

If PSYNC_FAST, then:	3
Else:	2

Words: 1

<1> Assembler syntax:
callr $+relative address [,cond]
or
callr $-relative address [,cond]

or
callr label [,cond]

Where 'label' is the new program memory location.
The instruction word includes the 'relative address' calculated by the Assembler as (label address) - (*callr* address) - 1.

<2> In case the stack is residing in Zspace, The writing of pc to the stack memory would still be executed regardless the state of the condition (met

or not), The sp register however will be modified only when the condition is met and thus a call is executed.

i.e. the next write (push) to the stack would write over the data written by this dummy write. This behavior is relevant for Zspace only, and is not applicable for a stack residing in Xspace and Yspace, due to the different nature of the Zspace interface.

chng Change Bit-field chng

chng ##long immediate ,operand

Operation: operand XOR ##long immediate → operand

Change specific bit-field in a 16-bit operand according to a long immediate value. The long immediate value contains ones in the bit-field location, where the bits will be changed.

If the operand is not a part of an accumulator (aXl, aXh, aXe, bXl, bXh) then the accumulators are unaffected. If the operand is a part of an accumulator, only the addressed part would be affected.

The operand and the long immediate values are sign-extension suppressed.

operand: REG <1>
(rN)
direct address

Affects flags when the operand is not st0

Z	M	N	V	C	E	L	R
*	*	-	-	-	-	-	-

When the operand is st0 :
The specified bits are changed according to the bit-field in the long immediate value, regardless of whether the a0e bits have been changed or not.

Cycles: 2

Words: 2

chng Change Bit-field (Cont'd) chng

<1> The REG cannot be : aX, bX, p, pc.

<2> When changing the a0e bits (*chng ##long immediate,st0*), the flags are affected according to the long immediate value. When changing the a1e bits (*chng ##long immediate,st1*) the flags are affected according to the ALU output.

clr Clear Accumulator clr

clr aX [,cond] See ***moda*** instruction.

Operation: aX = 0

or

clr bX [,cond] See ***modb*** instruction.

Operation: bX = 0

clrr Clear and Round aX-Accumulator clrr

clrr aX [,cond]

Operation: aX = 0x8000

See ***moda*** instruction.

cmp Compare cmp

cmp operand , aX

Operation: aX - operand

operand: REG <1>
(rN)
direct address
[##direct address]
#unsigned short immediate
##long immediate
(rb+offset7)
(rb+##offset)

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1
2 when the instruction is a two-word instruction

Words: 1
2 when the operand is: *##long immediate* or
(rb + ##offset) or *[##direct address]*

<1> The REG cannot be bX.

cmpu Compare Unsigned cmpu

cmpu operand , aX

Operation: aX - operand
The operand is sign-extension suppressed.

operand: REG <1>
(rN)
direct address

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1

Words: 1

<1> The REG cannot be: aX, bX, p.

<2> In order to compare aX with an unsigned 16-bit operand, bits [35-16] of the accumulator should be cleared (it can be done by using the mov → aXl instruction or other instructions).

cmpv Compare Long Immediate Value to a Register or a Data Memory Location cmpv

cmpv ##long immediate , operand

Operation: operand - ##long immediate
The operand and the long immediate values are sign-extended.

operand: REG <1>
(rN)
direct address

Affects flags: <2>

Z	M	N	V	C	E	L	R
*	*	-	-	*	-	-	-

Z, M, C are an output of the 16-bit operation.
M is affected by bit 15.

Cycles: 2

Words: 2

<1> The REG cannot be : aX, bX, p, pc.
Note that aX can be used in *cmp ##long immediate,aX* instruction.

<2> Note that when using *subv ##long immediate, st0* and *cmpv ##long immediate, st0* the flags are set differently.

cntx Context Switching Store or Restore cntx

cntx s | r

Operation: This instruction activates the context switching mechanism.

s - store the shadow/swap bits and swap a1 and b1 accumulators contents.
Bits: st0[0], st0[11÷2], st1[11÷10] and st2[7÷0], are pushed to their shadow bits.

The page bits st1[7÷0] are swapped with their alternative register.

a1 is transferred into b1, b1 is transferred into a1.

r - restore the shadow/swap bits and swap a1 and b1 accumulators contents.
Bits: st0[0], st0[11÷2],st1[11÷10] and st2[7÷0]are popped from their shadow bits.

The page bits st1[7÷0] are swapped with their alternative register.

a1 is transferred into b1, b1 is transferred into a1.

Affects flags: In store, flags represents the data transferred into a1

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

In restore, flags are written from their shadow bits.

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	*

Cycles: 1

Words: 1

copy Copy other aX-Accumulator copy

copy aX [,cond]

Operation: $aX = \overline{aX}$

See *moda* instruction

dec Decrement aX-Accumulator by One dec

dec aX [,cond]

Operation: aX = aX-1

See *moda* instruction

dint Disable Interrupt dint

dint

Operation:	0 → IE
	IE bit is cleared - Disable interrupts.
Affects flags:	No
Cycles:	1
Words:	1

divs Division Step divs

divs direct address ,aX

Operation:

```
aX - ( direct address*2^15 ) → ALU output
If ALU output < 0
    then  aX = aX * 2
        else  aX = ALU output * 2 + 1
```

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

Cycles: 1

Words: 1

<1> The 16-bit dividend is placed at accumulator-low; the accumulator-high and the accumulator-extension, are cleared.
The divisor is placed at the direct address.
For a 16-bit division, *divs* should be performed 16 times. After 16, times the quotient is in the accumulator-low and the remainder is in the accumulator-high.
The dividend and the divisor should both be positive.

eint Enable Interrupt eint

eint

Operation:	1 → IE
	IE bit is set - Enable interrupts.
Affects flags:	No

Cycles:	1
Words:	1

exp Evaluate the Exponent Value exp

exp soperand [, aX]

Operation when using *exp soperand* :

Exponent (soperand) → sv
The soperand is unaffected.

Operation when using *exp soperand,aX* :

Exponent (soperand) → sv and aX
The soperand is unaffected.

Evaluate the exponent value of the operand REG: (one of the registers) or data memory location, using an indirect addressing mode. The exponent result, a signed 6 bits value, is sign-extended and is written into the Shift Value register (sv) and optionally into one of the aX-accumulators. The source operand remains unchanged.

soperand :	REG	<1>
	(rN)	

Affects flags: No

Cycles: 1

Words: 1

<1> The REG cannot be p.

<2> Refer to paragraph 3.3.2.2, Exponent.

<3> The instruction following an *exp* instruction cannot move from/to sv register - sv register can be used only in *shfc* and *movs* instructions.

inc Increment aX-Accumulator by One inc

inc aX [,cond]

Operation: aX = aX+1

See *moda* instruction

lim Limit aX-Accumulator lim

lim aX [, aX]

Operation when using *lim* $\overline{aX}$:

```
If aX > 0x07FFFFFFF then
    aX = 0x07FFFFFFF
else
    If aX < 0xF80000000 then
        aX = 0xF80000000
    else
        aX is unaffected
```

Operation when using *lim* aX,$\overline{aX}$:

```
aX is unaffected
If aX > 0x07FFFFFFF then
    $\overline{aX}$ = 0x07FFFFFFF
else
    If aX < 0xF80000000 then
        $\overline{aX}$ = 0xF80000000
        else
        $\overline{aX}$ = aX
```

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	-	0	*	-

L flag is set when limitation occurs.

Cycles: 1

Words: 1

load Load Specific Fields into Registers load

load #immediate, operand

load #unsigned immediate 8 bit , **page** <1>

Operation: The page bits, which are the low-order 8 bits of st1, are loaded with an 8-bit constant (0 to 255).

load #unsigned immediate 9 bit , **modi**
load #unsigned immediate 9 bit , **modj**

Operation: The modX bits, which are the high-order 9 bits of cfgX, are loaded with a 9-bit constant (0 to 511, which means 1 to 512 modulo options).

load #immediate 7 bit , **stepi**
load #immediate 7 bit , **stepj**

Operation: The stepX bits, which are the low-order 7 bits of cfgX, are loaded with a 7-bit constant.

load #unsigned immediate 2 bit , **ps**

Operation: The ps status bits, bits 10 and 11 of st1, are loaded with a 2-bit constant.

The following table describes the number of values, which have to be loaded in order to receive the required number of shifts.

Table 4 - Loaded ps Values

ps bits		Number of shifts
bit 11	bit 10	
0	0	no shift
0	1	shift right by one
1	0	shift left by one
1	1	shift left by two

Affects flags: No

Cycles: 1

Words: 1

<1> The Assembler syntax permits the use of *lpg #unsigned short immediate*, which is the equivalent of *load #unsigned short immediate, page*.

lpg Load the Page Bits lpg

lpg #unsigned short immediate <1>

The page bits, which are the low-order 8 bits of st1, are loaded with an 8-bit constant (0 to 255).

See ***load*** instruction

maa Multiply and Accumulate Aligned Previous Product maa

maa operand1 , operand2 , aX

Operation: aX + aligned & shifted p → aX <1>
operand1 → y <2>
operand2 → x
signed y * signed x → p

operand1 , operand2:
y , direct address
y , (rN)
y , REG <3>
(rJ) , (rI) <4>
(rN) , ##long immediate

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

<1> Aligned & shifted p register means that the previous product is sign-extended into 36 bits, then shifted as defined in the ps field, status register st1, and then aligned, with sign-extension, 16 bit to the right.

<2> y → y means that y retains its value.

<3> The REG cannot be aX, bX, p.

<4> The multiplication in *maa (rJ), (rI), aX* is between Xspace/Zspace and Yspace only, where rJ points at Yspace, rI points at Xspace/Zspace.

maasu Multiply Signed by Unsigned and Accumulate Aligned Previous Product maasu

maasu operand1 , operand2 , aX

Operation:	aX + aligned & shifted p → aX	<1>
	operand1 → y	<2>
	operand2 → x	
	signed y * unsigned x → p	

operand1 , operand2:	y , (rN)	
	y , REG	<3>
	(rJ) , (rI)	<4>
	(rN) , ##long immediate	

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

<1> Aligned & shifted p register means that the previous product is sign-extended into 36 bits, then shifted as defined in the ps field, status register st1, and then aligned, with sign-extension, 16 bit to the right.

<2> y → y means that y retains its value.

<3> The REG cannot be aX, bX, p.

<4> The multiplication in *maasu (rJ), (rI), aX* is between and Xspace/Zspace only, where rJ points at Yspace, rI points at Xspace/Zspace.

mac Multiply and Accumulate Previous Product mac

mac operand1 , operand2 , aX

Operation: aX + shifted p → aX <1>
operand1 → y <2>
operand2 → x
signed y * signed x → p

operand1 , operand2:
y , direct address
y , (rN)
y , REG <3>
(rJ) , (rI) <4>
(rN) , ##long immediate

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

<1> Shifted p register means that the previous product is sign-extended into 36 bits, then shifted as defined in the ps field, status register st1.

<2> y → y means that y retains its value.

<3> The REG cannot be aX, bX, p.

<4> The multiplication in *mac (rJ), (rI), aX* is between Xspace/Zspace and Yspace only, where rJ points at Yspace, rI points at Xspace/Zspace.

macsu Multiply Signed by Unsigned and Accumulate Previous Product macsu

macsu operand1 , operand2 , aX

Operation:	aX + shifted p → aX	<1>
	operand1 → y	<2>
	operand2 → x	
	signed y * unsigned x → p	

operand1 , operand2:	y , direct address	
	y , (rN)	
	y , REG	<3>
	(rJ) , (rI)	<4>
	(rN) , ##long immediate	

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

<1> Shifted p register means that the previous product is sign-extended into 36 bits, then shifted as defined in the ps field, status register st1.

<2> y → y means that y retains its value.

<3> The REG cannot be aX, bX, p.

<4> The multiplication at *macsu (rJ),(rI),aX* is between Xspace/Zspace and Yspace only, where rJ points at Yspace, rI points at Xspace/Zspace.

macus Multiply Unsigned by Signed and Accumulate Previous Product macus

macus operand1 , operand2 , aX

Operation: aX + shifted p → aX <1>
operand1 → y <2>
operand2 → x
unsigned y * signed x → p

operand1 , operand2:

y , (rN)	
y , REG	<3>
(rJ) , (rI)	<4>
(rN) , ##long immediate	

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

<1> Shifted p register means that the previous product is sign-extended into 36 bits, then shifted as defined in the pc field, status register st1.

<2> y → y means that y retains its value.

<3> The REG cannot be aX, bX, p.

<4> The multiplication in *macus (rJ),(rI),aX* is between Xspace/Zspace and Yspace only, where rJ points at Yspace, rI points at Xsapce/Zspace.

macuu Multiply Unsigned by Unsigned and Accumulate Previous Product macuu

macuu operand1 , operand2 , aX

Operation: aX + shifted p → aX <1>
operand1 → y <2>
operand2 → x
unsigned y * unsigned x → p

operand1 , operand2: y , (rN)
y , REG <3>
(rJ) , (rI) <4>
(rN) , ##long immediate

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

<1> Shifted p register means that the previous product is sign-extended into 36 bits, then shifted as defined in the ps field, status register st1.

<2> y → y means that y retains its value.

<3> The REG cannot be aX, bX, p.

<4> The multiplication in *macuu (rJ),(rI),aX* is between Xspace/Zspace and Yspace only, where rJ points at Yspace, rI points at Xspace/Zspace.

macuu Multiply Unsigned by Unsigned and Accumulate Previous Product (Continued) macuu

<5> After using this instruction the p register cannot be reconstructed. During an interrupt service routine, which uses the p register, the p register should be saved before it is used, and restored before returning from the interrupt. However, the p register cannot be reconstructed after a *macuu* instruction. Therefore, it is recommended to place *a dint* instruction before the *macuu* instruction and an *eint* instruction after the instruction which uses the in the p register's output (the unsigned product).

<6> The instruction which uses the p register or the 'shifted p register' as a source operand, after a *macuu* instruction, uses the unsigned output in p register, zero extended into 36 bits and then shifted as defined in the ps field, status register st1. This situation will be maintained until a new signed product is generated or a new value is written into the *ph*.

max Maximum between Two aX-Accumulators max

max aX , (r0) , ge | gt

aX - the maximal value
mixp register - location of the maximal value

Operation when using *ge*:

```
If $a\overline{X} \geq aX$ then
   $aX = a\overline{X}$
   mixp = r0
r0 is post-modified as specified
```

Operation when using *gt*:

```
If $a\overline{X} > aX$ then
   $aX = a\overline{X}$
   mixp = r0
r0 is post-modified as specified
```

This instruction is used for finding the maximal value between the two aX-accumulators. In case the maximal value should be updated, it saves the new maximal value in the defined accumulator ,aX , and saves the r0 pointer value in mixp register. The r0 register is post-modified as specified in the instruction, regardless of whether the new maximal value is updated or not.

Affects flags:

Z	M	N	V	C	E	L	R
-	*	-	-	-	-	-	-

Note: M is set when the max value is found and the accumulator and mixp register were updated; cleared otherwise.

Cycles: 1

Words: 1

<1> mixp cannot be read in the instruction following the *max* instruction.

maxd Maximum between Data Memory Location and aX-Accumulator maxd

maxd aX , (r0) , ge | gt

aX - the maximal value
mixp register - location of the maximal value

Operation when using *ge*:

```
If (r0 ) ≥ aX then
    aX   = (r0)
    mixp =  r0
r0 is post-modified as specified
```

Operation when using *gt*:

```
If (r0) > aX then
  aX   = (r0)
  mixp =  r0
r0 is post-modified as specified
```

This instruction is used in the search for the maximal value between a data memory location pointed at by r0 and one of the aX-accumulators. In case where r0 points at a larger (or larger or equal) value, than the accumulator, the new maximal value is transferred into the defined accumulator (aX) and the r0 pointer is transferred into the mixp register. The r0 register is post-modified as specified in the instruction, regardless of whether the new maximal value is updated or not.

Affects flags:

Z	M	N	V	C	E	L	R
-	*	-	-	-	-	-	-

Note: M is set when the max. value is found and the accumulator and mixp register were updated; cleared otherwise.

maxd Maximum between Data Memory Location and aX-Accumulator (Cont'd) maxd

Cycles:	1
Words:	1

<1> mixp cannot be read in the instruction following the *maxd* instruction.

min Minimum between Two aX-Accumulators min

min aX , (r0) , le | lt

aX - the minimal value
mixp register - location of the minimal value

Operation when using *le*:

```
If a$\overline{X}$ $\le$ aX then
   aX = a$\overline{X}$
   mixp = r0
r0 is post-modified as specified
```

Operation when using *lt*:

```
If a$\overline{X}$ < aX then
   aX = a$\overline{X}$
   mixp = r0
r0 is post-modified as specified
```

This instruction is used in the search for the minimal value between the two aX-accumulators. In case the minimal value should be updated, it saves the new minimal value in the defined accumulator (aX) and saves the r0 pointer value in the mixp register. The r0 register is post-modified as specified at the instruction, regardless of the updates or non-updates of the new minimal value.

Affects flags:

Z	M	N	V	C	E	L	R
-	*	-	-	-	-	-	-

Note: M is set when the min value is found and the accumulator and mixp register were updated; cleared otherwise.

Cycles: 1

Words: 1

<1> mixp cannot be read in the instruction following the *min* instruction.

moda Modify aX-Accumulator Conditionally moda

moda func , aX [,cond]
or <1>
func aX [,cond]

Operation: If the co~~nd~~ition is met then
aX is modified by 'func'

The accumulator and the flags are modified according to the function field only when the condition is met.

func:	shr	aX = aX >> 1
	shl	aX = aX << 1
	shr4	aX = aX >> 4
	shl4	aX = aX << 4
	ror	Rotate aX right through carry
	rol	Rotate aX left through carry
	clr	aX = 0
	copy	aX = aX
	neg	aX = -aX
	not	aX = not(aX)
	rnd	Round upper 20 bits of the aX aX = aX+0x8000
	pacr	aX = shifted p + 0x8000 <2>
	clrr	aX = 0x8000
	inc	aX = aX + 1
	dec	aX = aX - 1

Affects flags: See below.

Cycles: 1

Words: 1

<1> The Assembler syntax makes it possible to omit the *mod*a, e.g. *shr a0* is equivalent to *moda shr,a0*.

<2> Shifted p register means that the p register is sign-extended to 36 bits and then shifted as defined in the ps field, status register *st1*.

moda Modify aX-Accumulator Conditionally (Cont'd) moda

shr - Shift right one bit.

shr4 - Shift right four bits.
The output obtained as a result of this operation is the same as received after executing *shr* four times.

Shl - Shift left one bit.

shl4 - Shift left four bits.
The output obtained as a result of this operation is the same as after executing *shl* four times.

Arithmetic shift is performed when the S status bit is cleared. **Logic** shift is performed when the S status bit is set.
See paragraphs 3.3.2.1.1, Shifting Operations and 3.7.2.2, Status Register Field Definition.

Affects flags:

When **arithmetic shift** is performed:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

C - Set according to the last bit (*shr* - bit 0, *shr4* - bit 3, *shl* - bit35, *shl4* - bit32) shifted out of the operand.

V - *shl*/*shl4*: Cleared if the operand is being shifted, can be presented in 35/31 bits for *shl/shl4*, respectively; set otherwise.
shr/*shr4*: Always cleared.

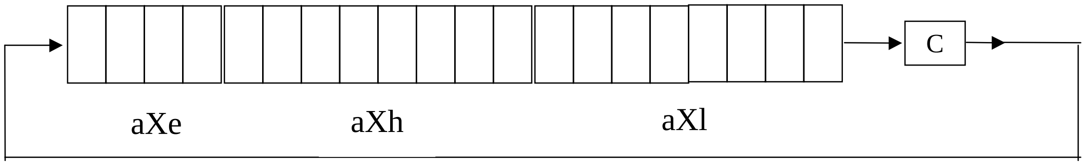
When **logic shift** is performed:

Z	M	N	V	C	E	L	R
*	*	*	-	*	*	-	-

moda Modify aX-Accumulator Conditionally (Cont'd) moda

C - Set according to the last bit (*shr* - bit 0, *shr4* - bit 3, *shl* - bit35, *shl4* - bit32) shifted out of the operand.

ror

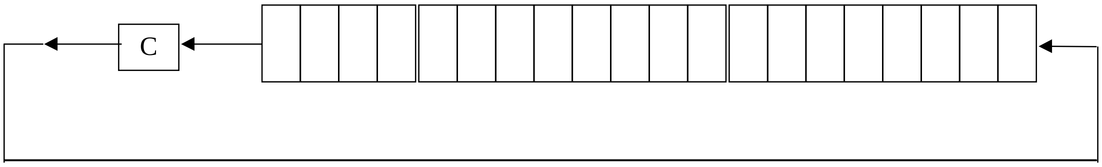


Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	*	*	-	-

C - Set according to the LSB (bit 0) shifted out of the operand.

rol



Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	*	*	-	-

C - Set according to the MSB (bit 35) shifted out of the operand.

moda Modify aX-Accumulator Conditionally (Cont'd) moda

not, copy, clr, clrr

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

neg, rnd, pacr

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

inc, dec

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

modb Modify bX-Accumulator Conditionally modb

modb func , bX [,cond]
or <1>
func bX [,cond]

Operation: If the condition is met then
bX is modified by 'func'

The accumulator and the flags are modified according to the function field only when the condition is met.

func:		
	shr	bX = bX >> 1
	sh1	bX = bX << 1
	shr4	bX = bX >> 4
	shl4	bX = bX << 4
	ror	Rotate bX right through carry
	rol	Rotate bX left through carry
	clr	bX = 0

Affects flags: See below.

Cycles: 1

Words: 1

<1> The Assembler syntax makes it possible to omit the *modb* , e.g. *shr b0* is equivalent to *modb shr,b0*.

shr - Shift right one step.

shr4 - Shift right four steps.
The output obtained as a result of this operation is the same as after executing *shr* four times.

shl - Shift left one step.

shl4 - Shift left four steps.
The output obtained as a result of this operation is the same as after executing *shl* four times.

modb Modify bX-Accumulator Conditionally (Cont’d) modb

shr / shr4/ shl / shl4

Arithmetic shift is performed when the S status bit is cleared. **Logic** shift is performed when the S status bit is set. See paragraphs 3.3.2.1.1, Shifting Operations and paragraph 3.7.2.2, Status Register Field Definition.

Affects flags:

When **arithmetic shift** is performed:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

C - Set according to the last bit (*shr* - bit 0, *shr4* - bit 3, *shl* - bit35, *shl4* - bit32) shifted out of the operand.

V - *shl*/*shl4*: Cleared if the operand is being shifted, can be presented in 35/31 bits for *shl/shl4* respectively; set otherwise.

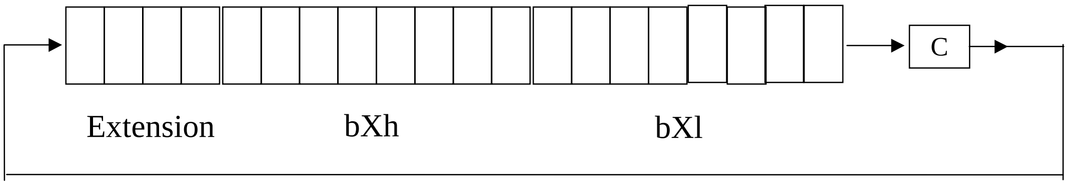
shr/*shr4*: Always cleared.

When **logic shift** is performed:

Z	M	N	V	C	E	L	R
*	*	*	-	*	*	-	-

C - Set according to the last bit (*shr* - bit 0, *shr4* - bit 3, *shl* - bit35, *shl4* - bit32) shifted out of the operand.

ror



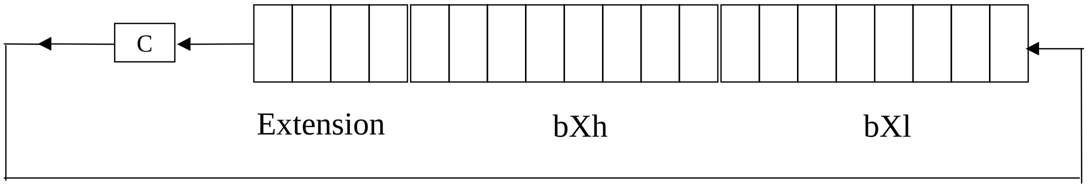
modb Modify bX-Accumulator Conditionally (Cont’d) modb

Affect flags:

Z	M	N	V	C	E	L	R
*	*	*	-	*	*	-	-

C - Set according to the LSB (bit 0) shifted out of the operand.

rol



Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	*	*	-	-

C - Set according to the MSB (bit 35) shifted out of the operand.

clr

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

modr Modify rN modr

modr (rN) [,dmod]

Operation when using *modr (rN)* :

rN is modified, as specified, and affected by the corresponding Mn bit.

Operation when using *modr (rN),dmod* :

rN is modified, as specified, with modulo disable.

Affects flags:

Z	M	N	V	C	E	L	R
-	-	-	-	-	-	-	*

R flag is set if the 16-bit rN register turns zero following post-modification; otherwise cleared.

Cycles: 1

Words: 1

<1> This instruction can be used also for loop control.
Example: *modr (r0)-*
brr add , nr.

mov Move Data mov

mov soperand , doperand

Operation: soperand → doperand

soperand,doperand:	REG	, REG	<1>,<2>,<3>,<4>
	REG	, (rN)	<1>,<2>,<5>
	(rN)	, REG	<4>,<5>
	mixp	, REG	<4>,<6>
	REG	, mixp	<1>,<2>,<6>
	icr	, ab	
	x	, ab	
	dvm	, ab	
	repc	, ab	
	aXl	, x	
	bXl	, x	
	aXl	, dvm	
	bXl	, dvm	
	REG	, icr	<7>,<8>
	rN	, direct address	
	aXl,	, direct address	
	aXh,	, direct address	
	bXl	, direct address	
	bXh	, direct address	
	y	, direct address	
	rb	, direct address	
	sv	, direct address	
	direct address	, rN	
	direct address	, aX	
	direct address	, aXl	
	direct address	, aXh [,eu]	<10>
	direct address	, bX	
	direct address	, bXl	
	direct address	, bXh	
	direct address	, y	
	direct address	, rb	
	direct address	, sv	

mov Move Data (Cont’d) mov

```
                    [##direct address], aX
                    aXl , [##direct address]

soperand,doperand :
                    (sp)          ,  REG                  <4>,<6>
                    (rb+ #offset7) ,  aX
                    (rb+ ##offset) ,  aX
                    aXl , (rb+#offset7)
                    aXl , (rb+ ##offset)

                    ##long immediate  , REG               <4>

                    #unsigned short immediate, aXl
                    #signed short immediate , aXh
                    #signed short immediate , rN          <9>
                    #signed short immediate , y           <9>
                    #signed short immediate , rb          <9>
                    #signed short immediate , extX        <9>
                    #signed short immediate , sv          <9>
                    #unsigned imm. (5 bits),icr           <7>,<8>
```

Affects flags: **No** effect when doperand is not ac, bc, st0, or when soperand is not aXl, aXh, bXl, bXh

When **soperand** is **aXl, aXh, bXl or bXh** :

Z	M	N	V	C	E	L	R
-	-	-	-	-	-	*	-

When **doperand** is **ac or bc**:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

If **doperand** is **st0**, st0 (including the flags) accepts the transferred data.

mov Move Data (Cont'd) mov

Cycles:

If PSYNC_FAST, and soperand is a data memory location or the st0/st1 register, and doperand is PC then :

2

Else: 1

2 when the instruction is a two-word instruction

Words: 1

2 when the operand is: *##long immediate* or *(rb+##offset) or [##direct address]*

<1> The 32-bit p register can be transferred (through the product output shifter) only to aX (*mov p,aX*). ph is a write-only register, therefore the soperand cannot be ph.

<2> The 36-bit accumulators can be a soperand only in the *mov ab,ab* instruction.

<3> With *mov reg,reg* the soperand cannot be the same as the doperand.

<4> When the doperand is the pc register, two *nop* instructions must be placed after the *mov soperand, pc* instruction, except for the instruction *mov ##long immediate, pc* where only one *nop* is necessary.

<5> It is not allowed to move data from a memory location (pointed at by one of the rN registers) to the same rN register (and vice versa) with post modification. (See also <11> below).

<6> The REG cannot be bx.

<7> Enable or disable of context switching (by a write to icr), takes effect after the next sequential instruction (e.g. when the user enables context switching for a specific interrupt, if the same interrupt is accepted immediately after the write to icr, it will not activate the context switching mechanism).

<8> A *mov soperand,icr* cannot be followed by a *bkrep* instruction.

<9> Loading the doperand with a short immediate number causes sign-extension.

<10> The eu field is an optional field.
eu = accumulator extension is unaffected. See the following table.

mov Move Data(Cont'd) mov

Instruction Fields		**Accumulator Fields Contents (following an instruction)**		
Accumulator	eu	Extension bits	16 MSB aXh / bXh	16 LSB aXl / bXl
aX / bX	-	Sign-extended	sign-extended	DATA
aXl / bXl	-	Cleared	cleared	DATA
aXh / bXh	-	Sign-extended	DATA	cleared
aXh	eu	Unaffected	DATA	cleared

<11> Conventions:

- The instruction in program memory address 0x0100: mov pc, ram

  After execution (RAM)=0x0101

- *mov (r0),r0*

	Before Execution	After Execution
r0	0x20	1000
RAM address 0x20	1000	1000

movd Move from Data Memory into Program Memory movd

movd (rI), (rJ)

Operation:

rI - points at a data memory location
rJ - points at a program memory location <1>
(rI) → (rJ)
rI is post-modified as specified
rJ is post-modified as specified

Move a word from a data memory location, pointed at by rI, into a program memory location, pointed at by rJ. <1>

Affects flags: No

Cycles: 4

Words: 1

<1> By no means should register rJ point at the *movd* instruction address or at (*movd* address) + 1.

movp Move from Program Memory into Data Memory movp

movp soperand , doperand

Operation: soperand - points at program memory location
soperand → doperand

Move a word from a Program memory location pointed at by soperand, into a data memory location pointed at by doperand, or into REG. When using aX as soperand, the address is defined by aX-accumulator-low.

soperand , doperand:		
	(aXl) , REG	<1>,<2>
	(rN) , (rI)	

Affects flags: No effect when doperand is not **ac**, **st0**.

When doperand is **ac** :

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

If the doperand is **st0**, st0 (including the flags) accepts the pointed program memory contents.

Cycles: 3

Words: 1

<1> When the operand REG is the pc register, two *nop* instructions must be placed after the *movp (ax),pc* instruction.

<2> The REG cannot be bX.

movr Move to Accumulator and Round movr

movr operand , aX
movr operand , abXh

Operation:

When using *mov operand,aX*
operand + 0x8000 → aX

operand : REG <1>
(rN)

When using *mov operand,abXh*
operand * 2^{16}+ 0x8000 → abX

operand : (rN) <2>

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1

Words: 1

<1> The REG cannot be bX.

<2> The rN operand can be r0, r2, r4 or r5. All post modification options are supported.

movs Move and Shift according to Shift Value Register movs

movs operand , ab

Operation: The operand is sign-extended to 36 bit

```
If  $0 < sv \le 36$  then
      operand << sv → ab

If  $-36 \le sv < 0$  then
      operand >> |sv| → ab

If sv = 0  then
      operand → ab
```

operand : REG <1>
(rN)
direct address

Affects flag

When **arithmetic shift** is performed:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

V - If $-36 \le sv \le 0$ (shift right) then V is cleared.
If $0 < sv < 36$ (shift left) and the operand, before being shifted, can be presented by (36 - sv) bits, then V is cleared; set otherwise.
If sv = 36 (shift left) and the operand ≠ 0, then V is set; cleared otherwise.

When logic shift is performed:

Z	M	N	V	C	E	L	R
*	*	*	-	*	*	-	-

Note**:** If sv = 0, the C flag is cleared.

movs Move and Shift according to Shift Value Register (Cont'd) movs

Cycles:	1
Words:	1

<1> The REG cannot be p.

<2> When operand is ab, the Assembler translates it into a *shfc* instruction, e.g., *shfc a0,b0,true*

movsi Move and Shift according to an Immediate Shift Value movsi

movsi operand , ab , #signed 5 bit immediate

Operation: The operand is sign-extended to 36 bit

```
If  0 < #immediate ≤ 15  then
      operand << #immediate → ab

If  -16 ≤ #immediate < 0  then
      operand >> #|immediate → ab

If #immediate = 0  then
      operand → ab
```

operand : rN
y
rb

Affects flags:

When **arithmetic shift** is performed:

Z	M	N	V	C	E	L	R
*	*	*	0	*	*	-	-

When **logic shift** is performed:

Z	M	N	V	C	E	L	R
*	*	*	-	*	*	-	-

Note: If #immediate = 0, the C flag is cleared.

Cycles: 1

Words: 1

mpy Multiply mpy

mpy operand1 , operand2

Operation:	operand1 → y operand2 → x signed y * signed x → p	<1>

operand1 , operand2:

y	, direct address	
y	, (rN)	
y	, REG	<2>
(rJ)	, (rI)	<3>
(rN)	, ##long immediate	

Affects flags: No

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

<1> y → y means that y retains its value.

<2> The REG cannot be aX, bX, p.

<3> The multiplication in *mpy (rJ),(rI)* is between Xspace/Zspace and Yspace only, where rJ points at Yspace, rI points at Xspace/Zspace.

mpyi Multiply Signed Short Immediate mpyi

mpyi y, #signed short immediate

Operation:	#signed short immediate → x signed y * signed x → p
Affects flags:	No
Cycles:	1
Words:	1

<1> For PineDSPCore compatibility, the Assembler syntax allows the use of the mnemonics *mpys y,#signed short immediate* which is equivalent to *mpyi y,#signed short immediate*.

mpys Multiply Signed Short Immediate mpys

mpys y, #signed short immediate

Operation: #signed short immediate → x
signed y * signed x → p

See ***mpyi*** instruction.

mpysu Multiply Signed by Unsigned mpysu

mpysu operand1 , operand2

Operation:	operand1 -> y operand2 -> x signed y * unsigned x -> p	<1>

operand1 , operand2:			
	y	, (rN)	
	y	, REG	<2>
	(rJ)	, (rI)	<3>
	(rN)	, ##long immediate	

Affects flags: No

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

<1> y → y means that y retains its value.

<2> The REG cannot be aX, bX, p.

<3> The multiplication at *mpysu (rJ),(rI)* is between Xspace/Zspace and Yspace only, where rJ points at Yspace, rI points at Xspace/Zspace.

msu Multiply and Subtract Previous Product msu

msu operand1 , operand2 , aX

Operation: aX - shifted p → aX <1>
operand1 → y <2>
operand2 → x
signed y * signed x → p

operand1 , operand2:			
	y	, direct address	
	y	, (rN)	
	y	, REG	<3>
	(rJ)	, (rI)	<4>
	(rN)	, ##long immediate	

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

<1> Shifted p register means that the previous product is sign-extended into 36 bits, then shifted as defined in the PS field, status register st1.

<2> y -> y means that y retains its value.

<3> The REG cannot be aX, bX, p.

<4> The multiplication at *msu (rj)*,*(ri*),aX is between Xspace/Zspace and Yspace only, where rJ points at Yspace, rI points at Xspace/Zspace.

neg 2's Complement of aX-Accumulator neg

neg aX [,cond]

Operation: aX = -aX

See ***moda*** instruction.

nop No Operation nop

nop

Operation:	No operation
Affects flags:	No
Cycles:	1
Words:	1

norm Normalize norm

norm aX , (rN)

Operation:

```
If N = 0 (aX is not normalized)
then aX = aX * 2
      rN is modified as specified
   else nop
         nop
```

This instruction is used to normalize the signed number in the accumulator. Affects the rN register when normalization is accomplished

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	*

The R flag is updated in *norm* instruction ONLY when rN pointer is modified.
C is set or cleared as in *shl* (*moda*).

Cycles: 2

Words: 1

<1> The *norm* instruction uses the N flag to determine whether shift or *nop*, are to be selected. Hence, when using *norm* in the first iteration, the flag must be updated according to aX.

<2> To normalize a number using the *norm* instruction, the *norm* instruction can be used along with a *rep* instruction.

Example:

```
rep    #n
norm   a0,(r0)+
```

Another method is to use the N flag for conditional branch.

Example:

```
nrm:  norm   a0, (r0)+
      brr    nrm, nn
```

norm Normalize (Cont'd) norm

<3> Normalization can also be accomplished by using the *exp* and shift instructions**.** For further details, refer to paragraph 3.3.2.3, Normalization.

not Logic Not not

not aX [,cond]

Operation: aX = not (aX)

See ***moda*** instruction.

or OR or

or operand , aX

Operation: If operand is aX or shifted p

aX[35-0] OR operand → aX[35-0]

If operand is REG, (rN),

unsigned short immediate, long immediate

aX[15-0] OR operand → aX[15-0]

aX[35-16] aX → [35-16]

If the operand is one of the aX-accumulators or the shifted p register, it is ORed with the destination accumulator.

If the operand is a 16-bit register or an immediate value, the operand is zero-extended to form a 36 bits operand, then ORed with the accumulator. Consequently, the upper bits of the accumulator are unaffected by this instruction.

operand:

REG <1>
(rN)
direct address
[##direct address]
#unsigned short immediate
##long immediate
(rb+offset7)
(rb+##offset)

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

Cycles: 1

2 when the instruction is a two-word instruction

Words: 1

2 when the operand is: *##long immediate* or *(rb+##offset)* or *[##direct address]*

<1> The REG cannot be bX.

pacr Product Move and Round to aX-Accumulator pacr

pacr aX [,cond]

Operation: aX = shifted p + 0x8000

See ***moda*** instruction.

pop Pop from Stack into Register pop

pop REG

Operation: (sp) → REG <1> , <2>
sp + 1 → sp

The top of stack is popped into one of the registers (REG) and the stack pointer, sp, is post-incremented.

Affects flags: When REG is **ac** :

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

If the REG is **st0**, st0 (including the flags) accepts the popped data.

Cycles:

If PSYNC_FAST, and REG is PC then :	2
Else:	1

Words: 1

<1> When popping to p, the data transferred into p-high (ph)

<2> The REG cannot be sp, bX.

<3> When *pop* instruction is following one of the following instructions:

- *mov soperand, sp (except* for *mov ##long immediate,sp)*
- *movp (aXl), sp ; addv/subv/set/rst/chng ##long immediate, sp*.

An extra cycle is added

<4> When REG is pc, two *nop* instructions must be placed after the pop instruction.

push Push Register or Long Immediate Value onto Stack push

push operand

Operation: sp - 1 → sp
operand → (sp)

The stack pointer, sp, is pre-decremented and the operand is pushed onto the software stack.

operand : REG <1>
##long immediate

Affects flags: **No** effect when operand is not aXl, aXh, bXl, bXh.

When operand is aXl, aXh, bXl, bXh:

Z	M	N	V	C	E	L	R
-	-	-	-	-	-	*	-

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

<1> The REG cannot be aX, bX, p, sp.

<2> The *push* instruction cannot follow the following instructions:

- *mov soperand, sp (except* for *mov ##long immediate,sp)*
- *movp (aXl), sp ; addv/subv/set/rst/chng ##long immediate, sp*.

rep Repeat Next Instruction rep

rep operand

Operation: Begins a single word instruction loop that is to be repeated operand+1 times.

The repetition range is from 1 to 65536.

The repeat mechanism is interruptible (see remark <1>) and the interrupt service routine can use another repeat (i.e. nested repeat). The nested repeat can not be interrupted.

operand:	#unsigned short immediate	<2>
	REG	<3>

Affects flags: No

Cycles:

If PSYNC_FAST, and operand is st0/st1 then:	2
Else:	1

Words:	1

<1> Interrupts are not accepted within *rep* loop in the following places :

- Between the *rep* command and the first repetition of the repeated instruction.
- If the repeated command is a one cycle command - also between the first instruction repetition and the second repetition.
- Between the last instruction repetition and the next sequential instruction.

<2> When using an unsigned short immediate operand, the number of repetitions is between 1 and 256. When transferring the #unsigned short immediate number into the repc register, it is copied into the low-order 8 bits of the repc. The high-order 8 bits are zero-extension of the low-order bits.

<3> The REG cannot be aX, bX, p.

rep Repeat Next Instruction (Cont'd) rep

<4> The following single word instructions cannot be repeated (most of these instructions break the pipeline): *brr; callr; trap; ret; reti; retd; retid; rets; rep; calla; mov operand,pc; pop pc; movp (aX),pc; mov repc,ab.*

<5> *rep* can be performed inside block-repeat (*bkrep)*.

ret Return Conditionally ret

ret [cond]

Operation:

```
If condition then
      (sp)    → pc
      sp + 1 → sp
```

If the condition is met, the program counter (pc) is pulled from the software stack, while the previous program counter is lost ; the stack pointer (sp) is post-incremented. This instruction is used to return from subroutines or interrupts.

Affects flags: No

Cycles:

If PSYNC_FAST, then:	3 in case the return is not performed
	4 in case the return is performed
Else:	2 in case the return is not performed
	3 in case the return is performed

Words: 1

<1> This instruction can also be used as return from the maskable interrupts service routines (INT0 or INT1 or INT2), to enable additional interrupts, the IE bit in st0 must be set by the user.

<2> The *ret* instruction cannot follow the following instructions:

- *mov soperand, sp (except* for *mov ##long immediate,sp)*
- *movp (aXl), sp ; addv/subv/set/rst/chng ##long immediate, sp*.

retd Delayed Return retd

retd

Operation: (sp) → temporary storage
sp + 1 → sp
Two one-cycle instructions or <1>
one two-cycles instruction following *retd* instruction, are/is executed
temporary storage → pc

Delayed return. The two single-cycle instructions, or one two-cycles instruction are/is fetched and executed <1> , before executing the return. When returned - the program counter (pc) is pulled from the software stack, while the previous program counter is lost ; the stack pointer (sp) is post-incremented. This instruction is used to obtain a delayed return from subroutines or interrupts.
The *retd* instruction and the instruction(s) which follow the *retd* (two one-cycle instructions or one two-cycles instruction) can not be interrupted.

Affects flags: No

Cycles: If PSYNC_FAST, then: 2
Else: 1

Words: 1

<1> Single cycle and the dual -cycles instructions cannot be instructions, which break the pipeline. These instructions are: I.e. *brr; callr; rep; trap; retd; retid; mov operand,pc pop pc*

<2> This instruction can also be used as return from the maskable interrupts service routines (INT0 or INT1 or INT2), to enable additional interrupts, the IE bit at st0 must be set by the user.

<3> The *retd* instruction cannot follow the following instructions:

- *mov soperand, sp (except* for *mov ##long immediate,sp)*
- *movp (aXl), sp ; addv/subv/set/rst/chng ##long immediate, sp*.

reti Return from Interrupt Conditionally reti

reti [cond [, context]]

Operation:

```
If condition  then
(sp)      → pc
sp + 1  → sp
1         → IE                         <1>
```

If the condition is met : the program counter (pc) is pulled from the software stack, and the previous program counter is lost ; the stack pointer (sp) is post-incremented and the IE bit is set - enable interrupts. <1>
This instruction is used for return from interrupts with or without interrupt context switching.

Affects flags: No

Cycles:

If PSYNC_FAST, then:	3 in case the return is not performed
	4 in case the return is performed
Else:	2 in case the return is not performed
	3 in case the return is performed

Words: 1

<1> IE is not changed when returning from NMI, BI, TRAP service routine.

<2> This instruction can be used for returning from interrupts. The TRAP/BI and NMI interrupt service routine must end with a reti or a retid instruction and it is restricted to use reti instruction while executing Trap/BI or NMI routines, rather then returning from the interrupt itself.

<3> Assembler syntax examples :
reti
reti ge
reti true , context
reti ge , context

<4> The *reti* instruction cannot follow the following instructions:

- *mov soperand, sp (except* for *mov ##long immediate,sp)*
- *movp (aXl), sp ; addv/subv/set/rst/chng ##long immediate, sp*.

retid Delayed Return from Interrupt retid

retid

Operation:

(sp) → temporary storage
sp + 1 → sp
Two one-cycle instructions or <2>
One two-cycles instruction following *retid* instruction, are/is executed
temporary storage → pc
1 → IE <1>

Delayed return from interrupt.
The IE bit is set - enable interrupts <1> and the stack pointer (sp) is post-incremented.
The two one-cycle instructions, or one two-cycle instruction are/is fetched and accomplished <2> , before executing the return.
When returned - the program counter (pc) is pulled from the software stack, whereas the previous program counter is lost.

The *retid* instruction and the instruction(s) following the *retid* (two one-cycle instructions or one two-cycle instruction) can not be interrupted.

Affects flags: No

Cycles:

If PSYNC_FAST, then:	2
Else:	1

Words: 1

<1> IE is not changed when returning from NMI, BI, TRAP service routine.

retid Delayed Return from Interrupt (Cont'd) retid

<2> Single-cycle and the two-cycle instructions; i.e. *brr; callr; rep; trap; retd; retid; mov operand,pc and pop pc*, can not break the pipe line.

<3> This instruction can be used for returning from interrupts. The TRAP/BI and NMI interrupt service routine must end with a *reti* or a *retid* instruction and it is restricted to use *reti* instruction while executing Trap/BI or NMI routines, rather then returning from the interrupt itself.

<4> The *retid* instruction cannot follow the following instructions:

- *mov soperand, sp (except* for *mov ##long immediate,sp)*
- *movp (aXl), sp ; addv/subv/set/rst/chng ##long immediate, sp*.

rets Return and Adjust Stack Pointer with a Short Immediate Offset rets

rets #unsigned short immediate

Operation: (sp) → pc
sp + 1 + #immediate → sp

The program counter is pulled from the stack. The previous program counter is lost. The Stack pointer is post-increment by one and by an 8 bit unsigned short immediate value.
This instruction is used for the return from subroutines or interrupts. It is also utilized to delete unnecessary parameters from the stack.

Affects flags: No

Cycles:

If PSYNC_FAST, then:	4
Else:	3
Words:	1

<1> This instruction can also be used for the return from interrupts (INT0, INT1 or INT2).To enable additional interrupts, the IE bit at st0 must be set by the user.

<2> The *rets* instruction cannot follow the following instructions:

- *mov soperand, sp (except* for *mov ##long immediate,sp)*
- *movp (aXl), sp ; addv/subv/set/rst/chng ##long immediate, sp*.

rnd Round Upper 20 bits of aX-Accumulator rnd

rnd aX [,cond]

Operation:	Round upper 20 bits of the aX aX = aX+0x8000

See ***moda*** instruction.

rol Rotate Accumulator Left through Carry rol

rol aX [,cond]

Operation: Rotate aX left through carry

See ***moda*** instruction.

or

rol bX [,cond]

Operation: Rotate bX left through carry

See ***mod*b** instruction.

ror Rotate Accumulator Right through Carry ror

ror aX [,cond]

Operation: Rotate aX right through carry

See ***moda*** instruction.

or

ror bX [,cond]

Operation: Rotate bX right through carry

See ***modb*** instruction.

rst Reset Bit-field rst

rst ##long immediate , operand

Operation: operand AND $\overline{\#\#long\_immediate} \rightarrow$ operand

Reset specific bit-field in a 16-bit operand according to a long immediate value. The long immediate value contains ones in the bit-field location, where the bits will be cleared.

If the operand is not part of an accumulator (aXl, aXh, aXe, bXl, bXh), then the accumulators are unaffected. If the operand is part of an accumulator, only the addressed part is affected.

The operand and the long immediate values are sign-extension suppressed.

operand: REG <1>
(rN)
direct address

Affects flags: When the operand is not st0:

Z	M	N	V	C	E	L	R
*	*	-	-	-	-	-	-

When the operand is st0 :
The specified bits are reset according to the bit-field in the long immediate value, regardless of whether the a0e bits have changed or not.

Cycles: 2

Words: 2

<1> The REG cannot be : aX, bX, p, pc.

rst Reset Bit-field (Cont'd) rst

<2> When resetting the a0e bits (*rst ##long immediate,st0*) the flags are reset according to the long immediate value. When resetting the a1e bits (*rst ##long immediate,st1*) the flags are reset according to the ALU output.

set Set Bit-field set

set ##long immediate , operand

Operation: ##long immediate OR operand → operand

Set specific bit-field in a 16-bit operand according to a long immediate value. The long immediate value contains ones in the bit-field location, where the bits will be set.

If the operand is not part of an accumulator (aXl, aXh, aXe, bXl, bXh) then the accumulators are **unaffected**. If the operand is part of an accumulator, only the addressed part is affected.

The operand and the long immediate values are sign-extension suppressed.

Operand: REG <1>
(rN)
direct address

Affects flags: When the operand is not st0:

Z	M	N	V	C	E	L	R
*	*	-	-	-	-	-	-

When the operand is st0 :
The specified bits are set according to the bit-field in the long immediate value, regardless of whether the a0e bits have changed. or not

Cycles: 2

Words: 2

<1> The REG cannot be : aX, bX, p, pc.

set Set Bit-field (Cont'd) set

<2> When setting the a0e bits (*set ##long immediate,st0*) the flags are set according to the long immediate value. When setting the a1e bits (*set ##long immediate,st1*) the flags are set according to the ALU output.

shfc Shift Accumulators according to Shift Value Register Conditionally shfc

shfc soperand , doperand [,cond]

Operation:

```
If condition is true then
    If 0 < sv ≤ 36 then
        soperand << sv → doperand

    If -36 ≤ sv < 0  then
        soperand >> |sv| → doperand

    If sv = 0  then
        soperand → doperand            <1>

    If soperand ≠ doperand  then
        soperand is un-affected.
```

soperand , doperand : ab , ab

Affects flags:

When **arithmetic shift** is performed:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

V - If **-36 ≤ sv ≤ 0** (shift right), then V is cleared.
If **0 < sv < 36** (shift left) and the operand before being shifted can be represented in (36 - sv) bits, then V is cleared; set otherwise.
If **sv = 36** (shift left) and the operand ≠ 0, then V is set; cleared otherwise.

When **logic shift** is performed:

Z	M	N	V	C	E	L	R
*	*	*	-	*	*	-	-

Note: if sv = 0, the C flag is cleared.

shfc Shift Accumulators according to Shift Value Register Conditionally (Cont'd) shfc

Cycles: 1

Words: 1

<1> In case the sv content is zero, this instruction is a conditional move between accumulators.

shfi Shift Accumulators by an Immediate Shift Value shfi

shfi soperand , doperand , #signed 6 bit immediate

Operation:

```
If  0 < #immediate ≤ 31  then
      soperand << #immediate → doperand

If  -32 ≤ #immediate < 0  then
      soperand >> #|immediate → doperand

If #immediate = 0  then
      soperand → doperand                    <1>

If soperand ≠ doperand  then
      soperand is un-affected.
```

soperand , doperand : ab , ab

Affects flags:

When arithmetic shift is performed:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

V - If -32 ≤ #immediate ≤ 0 (shift right) then V is cleared.
If 0 < #immediate ≤ 31 (shift left) and the operand before being shifted can be represented in (36 - #immediate) bits, then V is cleared; set otherwise.

When logic shift is performed:

Z	M	N	V	C	E	L	R
*	*	*	-	*	*	-	-

Note: if #immediate = 0, the C flag is cleared.

shfi Shift Accumulators by an Immediate Shift Value (Cont’d) shfi

Cycles:	1
Words:	1

<1> In case the immediate shift value is zero, this instruction can be used as a move instruction between the 36-bit accumulators.

shl Shift Left shl

shl aX [,cond]	See ***moda*** instruction.
Operation:	aX = aX<<1
or	
shl bX [,cond]	See ***modb*** instruction
Operation:	bX = bX<<1

shl4 Shift Left by 4 shl4

shl4 aX [,cond]	See ***moda*** instruction.
Operation:	aX = aX<<4
or	
shl4 bX [,cond]	See ***modb*** instruction
Operation:	bX = bX<<4

shr Shift Right shr

shr aX [,cond]	See ***moda*** instruction.
Operation:	aX = aX>>1
or	
shr bX [,cond]	See ***modb*** instruction
Operation:	bX = bX>>1

shr4 Shift Right by 4 shr4

shr4 aX [,cond]	See ***moda*** instruction.
Operation:	aX = aX>>4

or

shr4 bX [,cond]	See ***modb*** instruction
Operation:	bX = bX>>4

sqr Square sqr

sqr operand

Operation:	operand → y operand → x signed y * signed x → p	
operand :	(rN)	
	REG	<1>
	direct address	
Affects flags:	No	
Cycles:	1	
Words:	1	

<1> The REG cannot be aX, bX, p.

sqra Square and Accumulate Previous Product sqra

sqra operand ,aX

Operation:	aX + shifted p → aX operand → y operand → x signed y * signed x → p	<1>
operand :	(rN) REG direct address	 <2>

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1

Words: 1

<1> Shifted p register means that the previous product is sign-extended into 36 bits, then shifted as defined by the ps field, status register st1.

<2> The REG cannot be aX, bX, p.

sub Subtract sub

sub operand , aX

Operation: aX - operand → aX

operand: REG <1>
(rN)
direct address
[##direct address]
#unsigned short immediate
##long immediate
(rb+offset7)
(rb+##offset)

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1
2 when the instruction is a two-word instruction

Words: 1
2 when the operand is: *##long immediate* or *(rb+##offset) or [##direct address]*

<1> The REG cannot be bX.

subh Subtract from High Accumulator subh

subh operand , aX

Operation: aX - operand*2^16 → aX
The aXl is unaffected.

operand: REG <1>
(rN)
direct address

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1

Words: 1

<1> The REG cannot be: aX, bX, p.

subl Subtract from Low Accumulator subl

subl operand , aX

Operation: aX - operand → aX
The operand is sign-extension suppressed.

operand: REG <1>
(rN)
direct address

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	*	*	*	*	-

Cycles: 1

Words: 1

<1> The REG cannot be: aX, bX, p.

subv Subtract Long Immediate Value from a Register or a Data Memory Location subv

subv ##long immediate , operand

Operation: operand - ##long immediate → operand
The operand and the long immediate values are sign-extended .
If the operand is not a part of an accumulator (aXl, aXh, aXe, bXl, bXh) then the accumulators are unaffected. If the operand is a part of an accumulator, only the addressed part is affected.

operand: REG <1>
(rN)
direct address

Affects flags: When the operand is not **st0**: <2> <3> <4>

Z	M	N	V	C	E	L	R
*	*	-	-	*	-	-	-

Z, M and C are a result of the 16-bit operation.
M is affected by bit 15.

When the operand is **st0** :
st0 (including the flags) accepts the subtraction result, regardless of a0e bits.

Cycles: 2

Words: 2

<1> The REG cannot be : aX, bX, p, pc.
Note that aX can be used in *sub ##long immediate,aX* instruction.

<2> When subtracting a long immediate value from st0, st0 (including the flags) accepts the ALU output. When subtracting a long immediate value from st1, the flags are affected by the ALU output, as usual.

subv Subtract Long Immediate Value from a Register or a Data Memory Location (Cont'd) subv

<3> Note that when the operand is a part of an accumulator, only the addressed part is affected. For example, if the instruction *subv ##long immediate, a0l* generates a borrow, the carry flag is set, yet a0h is unchanged. On the other hand, the instruction *subl ##long immediate, a0l* (with same a0 and immediate values) changes the a0h and affects the carry flag according to the 36 bit ALU output.

<4> Note that when using *subv ##long immediate,st0* and *cmpv ##long immediate,st0* the flags are set differently.

swap Swap Ax and Bx accumulators swap

swap (See the following detailed mnemonics)

Swap between aX and bX accumulators according to the following options.

Assembler mnemonics	Operation
swap (a0,b0),(a1,b1)	a0 ↔ b0 and a1 ↔ b1
swap (a0,b1),(a1,b0)	a0 ↔ b1 and a1 ↔ b0
swap (a0,b0)	a0 ↔ b0
swap (a0,b1)	a0 ↔ b1
swap (a1,b0)	a1 ↔ b0
swap (a1,b1)	a1 ↔b1
swap (a0,b0,a1	a0 → b0 → a1
swap (a0,b1,a1)	a0 → b1 → a1
swap (a1,b0,a0)	a1 → b0 → a0
swap (a1,b1,a0)	a1 → b1 → a0
swap (b0,a0,b1)	b0 → a0 → b1
swap (b0,a1,b1)	b0 → a1 → b1
swap (b1,a0,b0)	b1 → a0 → b0
swap (b1,a1,b0)	b1 → a1 → b0

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

In case of :

swap (a0,b0),(a1,b1)
swap (a0,b1),(a1,b0)
The flags represent the data transferred into a0.

In other cases :
The flags represent the data transferred into aX.

Cycles: 1

Words: 1

<1> When the operation is x → y → z it means that:
y →z and then x → y.

trap Software Interrupt trap

trap

Operation:

sp - 1 → sp
pc → (sp)
pc → dvm
0x0002 → pc <1>
Disable interrupts (INT0 , INT1, INT2, NMI, BI) <1>,<2>

Software interrupt.
The stack pointer (sp) is pre-decrement. The program counter (pc) which points at the next instruction is pushed onto the stack and into the DVM register. A branch to address location 0x0002 is performed. The interrupts (INT0, INT1, INT2, NMI or BI) are disabled regardless of the interrupt mask bits: IE, IM0, IM1, IM2 in st0 and IM2 in st2.

Affects flags: No

Cycles: 2

Words: 1

<1> The software interrupt (TRAP) and the breakpoint interrupt (BI) share the same interrupt vector address. For further details concerning TRAP/BI, refer to paragraph 5.5.3.

<2> The *trap* instruction should not be used in a TRAP/BI service routine.

<3> To return from TRAP/BI service routine, use *reti* or *retid* instruction.

tst0 Test Bit-Field for Zeros tst0

tst0 mask , operand

Operation: If (operand AND mask) = 0x0000 then Z = 1
else Z = 0
The operand and the mask are sign-extension suppressed.

Test a specified bit field of REG (one of the registers), or a data space location (using a direct or indirect addressing mode) is all zero's. The field to be tested is specified by a mask, which contains ones in the bit field location. The mask is the content of an a-accumulator (aXl) or a long immediate operand. The test operation affects the zero flag - set it if the specified bit-field is all zeros, clear it otherwise.

mask:	aXl	<1>
	##long immediate	
operand:	REG	<1>,<2>
	(rN)	
	direct address	

Affects flags:

Z	M	N	V	C	E	L	R
*	-	-	-	-	-	-	-

Cycles: 1
2 when the mask is ##long immediate

Words: 1
2 when the mask is ##long immediate

<1> The instructions *tst0 a0l,a0l* ; *tst0 a1l,a1l* are not allowed.

<2> The REG cannot be: aX, bX, p.

tst1 Test Bit-Field for Ones tst1

tst1 mask , operand

Operation: If $(\overline{operand}$ AND mask) = 0x0000 then Z = 1
else Z = 0
The operand and the mask are sign-extension suppressed.

Test a specified bit field of REG (one of the registers), or a data space location (using a direct or indirect addressing mode) is all one's. The field to be tested is specified by a mask, which contains ones in the bit field location. The mask is the content of an a-accumulator (aXl) or a long immediate operand. The test operation affects the zero flag - set it if the specified bit-field is all one's, clear it otherwise.

mask:	aXl	<1>
	##long immediate	
operand:	REG	<1>,<2>
	(rN)	
	direct address	

Affects flags:

Z	M	N	V	C	E	L	R
*	-	-	-	-	-	-	-

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

<1> The instructions *tst1 a0l,a0l* ; *tst1 a1l,a1l* are not allowed.

<2> The REG cannot be: aX, bX, p.

tstb Test Specific Bit tstb

tstb operand , #bit number

Operation: If operand[bit number] = 1 then
Z=1

If operand[bit number] = 0 then
Z=0

operand: REG <1>
(rN)
direct address

$0 \leq$ bit number $\leq$ 15

Affects flags:

Z	M	N	V	C	E	L	R
*	-	-	-	-	-	-	-

Z flag reflects the tested bit status.

Cycles: 1

Words: 1

<1> The REG cannot be: aX, bX, p.

xor Exclusive - Or xor

xor operand , aX

Operation:

```
If operand is aX or p
    aX[35-0] XOR operand → aX[35-0]

        If operand is REG, (rN),
                unsigned short immediate, long immediate
            aX[15-0] XOR operand → aX[15-0]
            aX[35-16] → aX[35-16]
```

If the operand is one of the aX-accumulators or the p register, it is Exclusive-ORed with the destination accumulator.

If the operand is a 16-bit register or an immediate value, the operand is zero-extended to form a 36 bits operand, then Exclusive-ORed with the accumulator. Therefore, the upper bits of the accumulator are unaffected by this instruction.

operand:		
	REG	<1>
	(rN)	
	direct address	
	[##direct address]	
	#unsigned short immediate	
	##long immediate	
	(rb+offset7)	
	(rb+##offset)	

Affects flags:

Z	M	N	V	C	E	L	R
*	*	*	-	-	*	-	-

xor Exclusive - Or (Cont'd) xor

Cycles:	1 2 when the instruction is a two-word instruction
Words:	1 2 when the operand is: *##long immediate* or *(rb+##offset)* or *[##direct address]*

<1> The REG cannot be bX.

4.5 INSTRUCTION ENCODING

This section is an overview of the TeakLite encoding instruction set details. Listed are all the codes, number of cycles and words for all instructions.

Paragraph 4.5.1 includes definitions of the fields' abbreviations and their encoding.

Paragraph 4.5.2 contains detailed encoding tables of all the TeakLite instructions, including the number of cycles and the number of words.

4.5.1 Abbreviations Definition and Encoding

- The '.' letter anywhere in the code means don't care. It is translated as '0' by the Assembler.
- Opcodes in bold are coding families, which are translated to several opcodes according to the table below the code.

The coding families are

ALU - ALU opcodes
ALM - ALU and MULTIPLY opcodes
ALB - ALU and BMU opcodes

A (aX)	=	0	Accumulator a0
		1	Accumulator a1
B (bX)	=	0	Accumulator b0
		1	Accumulator b1
L	=	0	Low
		1	High
AB	=	00	b0
		01	b1
		10	a0
		11	a1

ABl =

00	b0l
01	b1l
10	a0l
11	a1l

ABLH =

000	b0l
001	b0h
010	b1l
011	b1h
100	a0l
101	a0h
110	a1l
111	a1h

dddddddd = direct address bits

vvvvvvvv = 8 bit short immediate

ooooooo = 7 bit offset (offset7) of relative and index addressing modes.

BBBB = bit number (one of 16 bits of a register)

.

nnn (rN) =

000	r0
001	r1
010	r2
011	r3
100	r4
101	r5

nnn (rN*) =

000	r0
001	r1
010	r2
011	r3
100	r4
101	r5
110	rb
111	y

rrrrr (register) =

00000	r0
00001	r1
00010	r2
00011	r3
00100	r4
00101	r5
00110	rb
00111	y
01000	st0
01001	st1
01010	st2
01011	p / ph
01100	pc
01101	sp
01110	cfgi
01111	cfgj
10000	b0h
10001	b1h
10010	b0l
10011	b1l
10100	ext0
10101	ext1
10110	ext2
10111	ext3
11000	a0
11001	a1
11010	a0l
11011	a1l
11100	a0h
11101	a1h
11110	lc
11111	sv

Modification of rN :

mm	=	00	No modification
		01	+1
		10	-1
		11	+step

Modification of rI :

ii	=	00	No modification
		01	+1
		10	-1
		11	+step

Modification of rJ :

jj	=	00	No modification
		01	+1
		10	-1
		11	+step

w (rJ)	=	0	r4
		1	r5

qq (rI)	=	00	r0
		01	r1
		10	r2
		11	r3

cccc	=	0000	true
		0001	eq
		0010	neq
		0011	gt
		0100	ge
		0101	lt
		0110	le
		0111	nn
		1000	c
		1001	v
		1010	e
		1011	l
		1100	nr
		1101	niu0
		1110	iu0
		1111	iu1

4.5.2 Instructions Encoding Table

This paragraph contains a detailed coding table of all TeakLite instructions, including the number of cycles and the number of words.

The instructions are organized groups (refer to paragraph 4.3.1, Instruction Set Summary by Instruction Group). Some of the codes are arranged in subgroups. Each subgroup is followed by a list of instructions, which use the code of this subgroup and the encoding of the XX..X field for each instruction. Note that the same instruction may appear in several subgroups, depending on the addressing mode or operand usage.

Table 4-4 Instructions Encoding

Opcode	Code	Cycles	Words
ALM direct	101XXXXAdddddddd	1	1
ALM (rN)	100XXXXA100mmnnn	1	1
ALM register	100XXXXA101rrrrr	1	1
XXXX = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111	or and xor add tst0_a (mask in aXl) tst1_a (mask in aXl) cmp sub msu addh addl subh subl sqr sqra cmpu		
ALU #short immediate	1100XXXAvvvvvvvv	1	1
ALU ##long immediate	1000XXXA110.....	2	2
ALU (rb+#offset7),aX	0100XXXA0ooooooo	1	1
ALU (rb+##offset),aX	1101010A11011XXX	2	2

Table 4-4 Instructions Encoding (Cont'd)

Opcode	Code	Cycles	Words
ALU [##direct add.],aX	1101010A11111XXX	2	2
XXX = 000 or 001 and 010 xor 011 add 110 cmp 111 sub			
Moda	011A0111ffffcccc	1	1
ffff = 0000 shr 0001 shr4 0010 shl 0011 shl4 0100 ror 0101 rol 0110 clr 0111 reserved 1000 not 1001 neg 1010 rnd 1011 pacr 1100 clrr 1101 inc 1110 dec 1111 copy			
Norm	1001010A11.mmnnn	2	1
Divs	0000111Adddddddd	1	1
ALB (rN)	1000XXX0111mmnnn	2	2
ALB register	1000XXX1111rrrrr	2	2
ALB direct	1110XXX1dddddddd	2	2
XXX = 000 set 001 rst 010 chng 011 addv 100 tst0_l (mask in ##long immediate) 101 tst1_l (mask in ##long immediate) 110 cmpv 111 subv			

Table 4-4 Instructions Encoding (Cont'd)

Opcode	Code	Cycles	Words
maxd	100000fA011mm...	1	1
f = 0 ge 1 gt			
max	100001fA011mm...	1	1
f = 0 ge 1 gt			
min	10001.fA011mm...	1	1
f = 0 le 1 lt			
lim	0100100111XX....	1	1
XX = 00 lim a0 01 lim a0,a1 10 lim a1,a0 11 lim a1			
MUL y, (rN)	1000AXXX001mmnnnn	1	1
MUL y, register	1000AXXX010rrrrr	1	1
MUL (rJ), (rI)	1101AXXX0jjiiwqq	1	1
MUL (rN), ##long imm.	1000AXXX000mmnnnn	2	2
XXX = 000 mpy 001 mpysu 010 mac 011 macus 100 maa 101 macuu 110 macsu 111 maasu			
MUL y, direct address	1110AXX0dddddddd	1	1
mpyi	00001000vvvvvvvv	1	1

Table 4-4 Instructions Encoding (Cont'd)

Opcode	Code	Cycles	Words
msu (rN), ##long imm.	1001000A11.mmnnn	2	2
msu (rJ), (rI)	1101000A1jjiiwqq	1	1
tstb (rN)	1001bbbb001mmnnn	1	1
tstb register	1001bbbb000rrrrr	1	1
tstb direct address	1111bbbbdddddddd	1	1
shfc	1101ab101AB0cccc	1	1
ab is the source (it's code is as of AB)			
shfi	1001ab1AB1vvvvvv	1	1
ab is the source (it's code is as of AB) vvvvvv = 6 bit immediate			
modb	011B1111.fffcccc	1	1
fff = 000 shr 001 shr4 010 shl 011 shl4 100 ror 101 rol 110 clr 111 reserved			
exp (rN), aX	1001100A01.mmnnn	1	1
exp register , aX	1001000A010rrrrr	1	1
exp bX, aX	1001000A011....B	1	1
exp (rN), sv	1001110.01.mmnnn	1	1
exp register , sv	1001010.010rrrrr	1	1
exp bx, sv	1001010.011....B	1	1
mov register, register	010110RRRRRrrrrr	1 *	1
RRRRR is the destination (it's code is as of rrrrr)			
mov ab, AB	1101ab101AB10...	1	1
ab is the source (it's code is as of AB)			

Table 4-4 Instructions Encoding (Cont'd)

Opcode	Code	Cycles	Words
mov ABl, dvm	1101ab1010.11...	1	1
mov ABl, x	1101AB1011.11...	1	1
mov register, bX	0101111011brrrrr	1	1
mov register , mixp	0101111010.rrrrr	1	1
mov register , (rN)	000110rrrrrmmnnn	1	1
mov mixp, register	01000111110rrrrr	1	1
mov repc, AB	1101010.1AB10.00	1	1
mov dvm, AB	1101010.1AB10.01	1	1
mov icr, AB	1101010.1AB10.10	1	1
mov x, AB	1101010.1AB10.11	1	1
mov (rN), register	000111rrrrrmmnnn	1 *	1
mov (rN), bX	1001100B11.mmnnn	1	1
mov (sp), register	01000111111rrrrr	1 *	1
mov rN*, direct	0010nnn0dddddddd	1	1
mov ABLH, direct	0011ABL0dddddddd	1	1
mov direct, rN*	011nnn00dddddddd	1	1
mov direct, AB	011AB001dddddddd	1	1
mov direct, ABLH	011ABL10dddddddd	1	1
mov direct, aXHeu	011A0101dddddddd	1	1
mov direct, sv	01101101dddddddd	1	1
mov sv, direct	01111101dddddddd	1	1
mov [##direct add.],aX	1101010A101110..	2	2
mov aXl,[##direct add]	1101010A101111..	2	2
mov ##long immediate, register	0101111.000rrrrr	2	2
mov ##long imm., bX	0101111B001.....	2	2

Table 4-4 Instructions Encoding (Cont’d)

Opcode	Code	Cycles	Words
mov #short, aXl	001A0001vvvvvvvv	1	1
mov #short, aXh	001A0101vvvvvvvv	1	1
mov #short, rN*	001nnn11vvvvvvvv	1	1
mov #short, ext0-3	001X1X01vvvvvvvv	1	1
XX = 00 ext0 01 ext1 10 ext2 11 ext3			
mov #short, sv	00000101vvvvvvvv	1	1
mov register, icr	0100111111.rrrrr	1	1
mov #immediate, icr	0100111110.vvvvv	1	1
vvvvv = 5 bit immediate			
mov (rb+#offset7), aX	1101100A1ooooooo	1	1
mov aXl, (rb+#offset7)	1101110A1ooooooo	1	1
mov (rb+##offset), aX	1101010A100110..	2	2
mov aXl, (rb+##offset)	1101010A100111..	2	2
movp (aX), register	0000000001Arrrrr	3 *	1
movp (rN), (rI)	0000011iiqqmmnnn	3	1
movs register , AB	000000010ABrrrrr	1	1
movs (rN), AB	000000011ABmmnnn	1	1
movs direct, AB	011AB011dddddddd	1	1
movsi rN*, AB	0100nnn01ABvvvvv	1	1
vvvvv = 5 bit immediate			
movr register , aX	1001110A110rrrrr	1	1
movr (rN), aX	1001110A111mmnnn	1	1
movr (rN),abXh	100010AB011NN1MM	1	1
NN = 00 r0 01 r4 10 r2 11 r5	MM = 00 +1 01 -1 10 +0 11 +s		
push register	01011110010rrrrr	1	1
push ##long immediate	0101111101......	2	2

Table 4-4 Instructions Encoding (Cont'd)

Opcode	Code	Cycles	Words
pop register	01011110011rrrrr	1 *	1
swap swap-options	0100100110..swap	1	1
swap-options = 0000	a0 ↔ b0		
0001	a0 ↔ b1		
0010	a1 ↔ b0		
0011	a1 ↔ b1		
0100	a0 ↔ b0 and a1 ↔ b1		
0101	a0 ↔ b1 and a1 ↔ b0		
0110	a0 → b0 → a1		
0111	a0 → b1 → a1		
1000	a1 → b0 → a0		
1001	a1 → b1 → a0		
1010	b0 → a0 → b1		
1011	b0 → a1 →b1		
1100	b1 → a0 → b0		
1101	b1 → a1 → b0		
banke	010010111...bank	1	1
bank = r0, r1, r4, cfgi For example, bank = 0101 means bank switch of r1 and cfgi.			
movd (rI), (rJ)	010111111jjiiwqq	4	1
rep register	00001101...rrrrr	1 *	1
rep #short	00001100vvvvvvvv	1	1
bkrep #short	01011100vvvvvvvv	2	2
bkrep register	01011101...rrrrr	2	2
break	1101001111......	1	1
br	0100000110..cccc	2/3 *	2
brr	01010ooooooocccc	2 *	1
call	0100000111..cccc	2/3 *	2
callr	00010ooooooocccc	2 *	1
calla	1101010A1..0....	3	1
ret	0100010110..cccc	2/3 *	1

Table 4-4 Instructions Encoding (Cont'd)

Opcode	Code	Cycles	Words
retd	1101011110	1 *	1
reti	0100010111.fcccc	2/3 *	1
f = 0 Don't do context switching f = 1 Do context switching			
retid	1101011111.0....	1 *	1
cntx	1101001110.f....	1	1
f = 0 (s) store shadows of context switching f = 1 (r) restore shadows of context switching			
rets	00001001vvvvvvvv	3 *	1
nop	00000000000.....	1	1
modr	000000001.fmmnnn	1	1

f = 0 Don't disable modulo
f = 1 Disable modulo			

eint	0100001110......	1	1
dint	0100001111......	1	1
trap	00000000001.....	2	1
load page	00000100vvvvvvvv	1	1
load modX	0000x01vvvvvvvvv	1	1

x = 0 load modi
x = 1 load modj
vvvvvvvvv = 9 bit immediate			

load stepX	11011x111vvvvvvv	1	1

x = 0 load stepi
x = 1 load stepj
vvvvvvv = 7 bit immediate			

load ps	010011011.....vv	1	1
vv = 2 bit immediate			

***** Refer to **4.4 INSTRUCTION SET DETAILS** for PSYNC_FAST mode cycle count.

4.6 INSTRUCTION EXECUTION

4.6.1 Pipeline Method

The program controller implements a four level pipeline architecture. In the operation of. the pipeline concurrent fetch; decode, operand fetch and execution occur. Consequently, up to four different instructions are being processed, each at a different stage of pipeline. During a given cycle The pipeline is of, so called 'interlocking' type.

The following chart shows the pipeline operation for sequential single cycle instructions:

	cycle1	cycle2	cycle3	cycle4	cycle5
Fetch	← n →	← n+1 →	← n+2 →	← n+3 →	
Decode		← n →	← n+1 →	← n+2 →	← n+3 →
Op. Fetch			← n →	← n+1 →	← n+2 →
Execute				← n →	← n+1 →

Instruction n is executed in cycle 4. During this cycle instruction n+3 is pre-fetched, instruction n+2 is decoded and the operand required at instruction n+1 is fetched.

For multiple cycles instruction the pipeline structure may differ, yet those details are beyond the scope of this document.

5. CORE INTERFACE

5.1 INTRODUCTION

This chapter describes the TeakLite core interface. It provides information as regards different power save modes (SLOW and STOP). Generally described are the *clock* block, and the *reset* mode, the interrupts handling, latency and their priority. All possible memory interface modes are described: asynchronous, synchronous data and synchronous program (regular and fast) interface modes. Given is also a description of the core pins along with a detailed information concerning the built in mechanisms, available for the generation of a TeakLite based application specific IC. Some remarks related to the core support for the On Chip Emulation Module, OCEM, are also included.

Note that the signals and the pin names, used in this chapter comply with the design methodology core pin names (refer to Appendix 5.B, Signal Names Conversion Table).

A common term used in this chapter is *machine cycle*. A machine cycle is one clock cycle, which may be stretched to more than one in case of wait states. It is stretched until the end of the wait interval.

Note that all interrupt and reset diagrams illustrate the core's signals behavior at asynchronous program memory interface.

5.2 POWER SAVE MODES

Power save modes are the core operation states where the current consumption is minimized during application run. There are two power save modes: Slow mode and Stop mode.

It should be noted that all power save modes are activated via the clock block. All power save modes are only affecting the core through multiplication of the core's clock output. Moreover, depending on a specific an application, other power save modes can be defined, where additional power consumption sources besides the core (e.g. I/O, memories, peripherals) can be "shut off" or reduced.

5.2.1 SLOW Mode of Operation

SLOW Mode is a state where the core main phases are slowed down through clock division logic. A typical way to implement the SLOW mode is to define a memory mapped I/O register where the division factor is specified through software programming. This memory mapped I/O register controls the clock division circuitry.

5.2.2 STOP Mode of Operation

The TeakLite core can enter a STOP mode state, where it consumes power due to the leakage within CMOS devices. This is done by keeping the clock input constantly low.
Owing it to the TeakLite static design, when returning from STOP mode, the RAM contents and all the registers remain unaffected. Recovery from STOP mode is by utilizing off-core logic, which reactivates the clock. The core will continue the regular instruction flow, from the point it has been stopped. TeakLite Recovery can also be accomplished by interrupts activation or the reset pins. When recovering by way of an interrupt (once it is enabled), the core will begin the execution from the corresponding interrupt handler address. When recovering through a reset, the RAM contents and all registers, which are not defined as being affected by the reset, will remain unaffected. The execution will resume from address zero.

A typical implementation of a STOP mode is through a dedicated memory mapped I/O port that, when it is written, enters into STOP mode (i.e. causes the core main clocks to be "shut-off"). Recovery from STOP mode can be either through activation of an interrupt, reset or a continue signal. If the recovery is by an interrupt, four *nop* instructions should be inserted following the instruction that causes the STOP mode activation. This will ensure that when recovering from STOP mode no meaningful operations / instructions are performed before the interrupt service routine.

5.3 CLOCK

The TeakLite is driven by an off-core clock generator. It generates a single core clock input pin, LCLK. The user can slow or stop the processor operation by slowing or disabling the clock. The TeakLite can operate with clock speeds ranging from 0 to maximum core operating frequency allowed by its static design. All the TeakLite registers are activated by the rising edge of the clock. In addition, the TeakLite employs clock gating in order to reduce power consumption.
For AC specification regarding the clock, refer to Appendix A.

5.3.1 Wait State Generation through Clock

The TeakLite does not receive a special wait state indication. Whenever an additional wait state cycle is required, such an indication is issued to the core clock generation unit. In such cases, the clock unit does not assert the core clock signal LCLKP (in is kept low throughout the entire cycle). This implementation leads to notable power consumption saving since the core is kept completely inactive during wait state cycles. Not that the actual implementation of the wait state logic within the clock block is beyond the scope of this document.

5.4 RESET

Reset is an external event that can be issued whenever setting the core to a predetermined state is desired. Reset is activated by driving the core's reset pin (LRSTP) into 'high' state. The LRSTP signal has to be synchronized to core clock (LCLKP) before it is applied to the core pin.
When the LRSTP is applied to the core, the core terminates any execution and enters a reset state. During the reset state various registers and status bits are initialized, other registers as well as the RAM content are unaffected. Deactivation of the LRSTP will cause the execution to start from location 0x0000.

The following registers bits are cleared during reset:

- ST0 bits 0¸11, ST1 bits 10¸11, ST2 bits 0¸9
- ICR bits 0 - 7
- PC bits 0 - 15

For further details regarding the reset effect on the status registers and the ICR register, refer to paragraph 3.7.2.2, Status Register Field Definition and to paragraph 3.7.3, Internal Configuration Register.

The LRSTP signal must be activated for at least 6 clock cycles. The fetch from address 0x0000 is executed one cycle after the LRSTP input is deactivated.

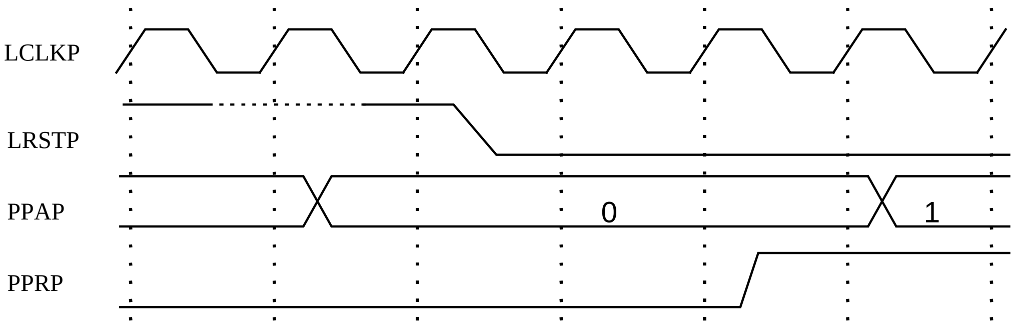


Figure 5-1 Reset

During reset state (LRSTP input is '1') the core drives the program address bus (PPAP) with ‘0’. The first instruction fetch is accomplished 1.5 cycles after the deactivation of LRSTP. At that time the program read enable (PPRP) is activated.

5.5 INTERRUPTS

The TeakLite core features three maskable interrupts (INT0, INT1 and INT2) and one non maskable interrupt (NMI). It also has one software interrupt (TRAP) and a hardware breakpoint (BI) interrupt (both sharing the same interrupt vector).

The core uses one stack level to store the program counter. When one of the interrupts NMI, INTO, INT1 or INT2 is accepted it has an option for automatic context switching mechanism to save/restore critical parameters For further details refer to paragraph 3.7.3, Interrupt Context Switching Mechanism. The hardware interrupts are of high level-sensitivity and have to be synchronized with the LCLKP before they are applied to the TeakLite core.

Table 5-1 Interrupt Vectors and Priorities

Memory Location*	**Interrupt Name and Function**	**Priority**
0x0000	**RESET**	1 highest
0x0002	**TRAP/BI** Software interrupt / breakpoint interrupt	2
0x0004	**NMI** Non maskable interrupt	3
0x0006	**INT0** External user interrupt 0	4
0x000E	**INT1** External user interrupt 1	5
0x0016	**INT2** External user interrupt 2	6 lowest

* Start address for the interrupt/reset routine.

5.5.1 NMI - Non Maskable Interrupt

The NMI is an active high non-maskable interrupt. During execution of the NMI service routine, an additional NMI or any of the maskable interrupts are not serviced. A NMI is accepted while the core is in an interruptible state (as described in paragraph 5.5.4). When the core accepts a NMI the following actions take place:

SP - 1 → SP
PC → (SP)
0x0004 → PC (interrupt starting address)
PIACKN - Interrupt acknowledge signal is activated for a single cycle once the control is being transferred to the service routine.

The NMI service routine must end with a *reti or retid* instruction.

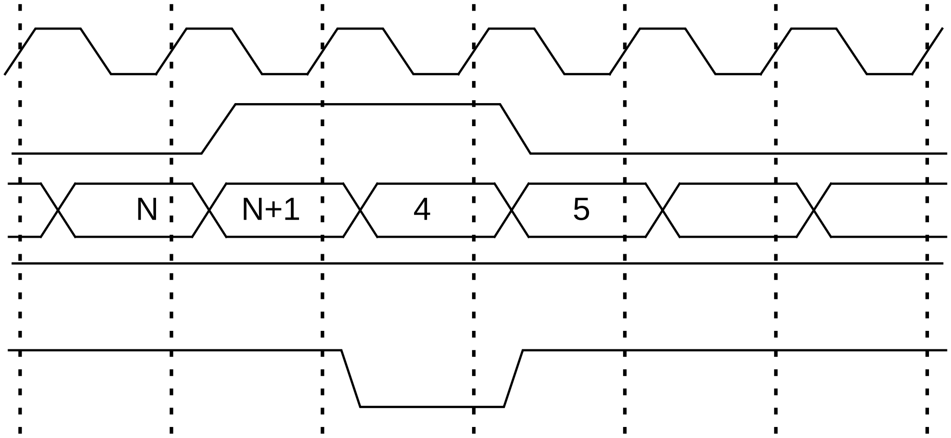


Figure 5-2 MMI Request

Activation of the LNMIP input (NMI request pin) prevents the execution of the N+1 instruction. The core starts fetching from address 4 and activates the interrupt acknowledge output (PIACKN). PIACKN can be used to reset the LNMIP input.

5.5.2 INT0, INT1 and INT2 - Maskable Interrupts

INT0, INT1 and INT2 are active high maskable interrupts.

A maskable interrupt is accepted while the core is in an interruptible state (as described in 5.5.4), the interrupt enable bit (IE) within ST0 register is set, and the corresponding interrupt mask bit (IMx x=0,1,2) is set.

When an interrupt is accepted the core performs the following:

IE status bit is cleared

SP - 1 → SP

PC → (SP)

0x0006 or 0x000E or 0x0016 → PC (INT0, INT1 or INT2 interrupt starting address)

PIACKN - Interrupt acknowledge signal is activated for a single cycle once the control is being transferred to the service routine.

IMx is unaffected.

When the interrupt is acknowledged, the IE bit within the ST0 register is reset, disabling other maskable interrupts from being serviced. Additional pending interrupts are serviced once the user re-enables (sets) the IE bit by the service routine.

A return from an interrupt service routine is accomplished by the user who is using the instruction *reti, retid, ret, rets* or *retd*. The instructions *reti* and *retid* set the IE flag allowing pending interrupts to be serviced. When using the *ret* or *retd* instructions, the IE bit must be set in order to enable additional interrupts.

Interrupt *priority* is utilized to arbitrate simultaneous interrupts. INT0 has the higher priority and INT2 has the lowest. Nesting is supported if enabled by the service routine. The priority between INT0/INT1/INT2 is significant only if more than one interrupt is received at the same time, or when the IE is disabled for some time and more than one of INT0, INT1 and INT2 were received. In these cases, the interrupt will be acknowledged according to the interrupt priorities. For example: in case the processor is handling the INT1 service routine and INT0 is received, the IP0 bit will be set and the processor will enter the INT0 service routine if the status bits, IE and IM0, enable this interrupt.

IP0, IP1 or IP2 bits within ST2 register are set once the corresponding interrupt LINT0P, LINT1P or LINT2P is activated. These bits can be used in applications which use interrupt polling, while disabling (via the IE bit) the automatic respond to the interrupts.

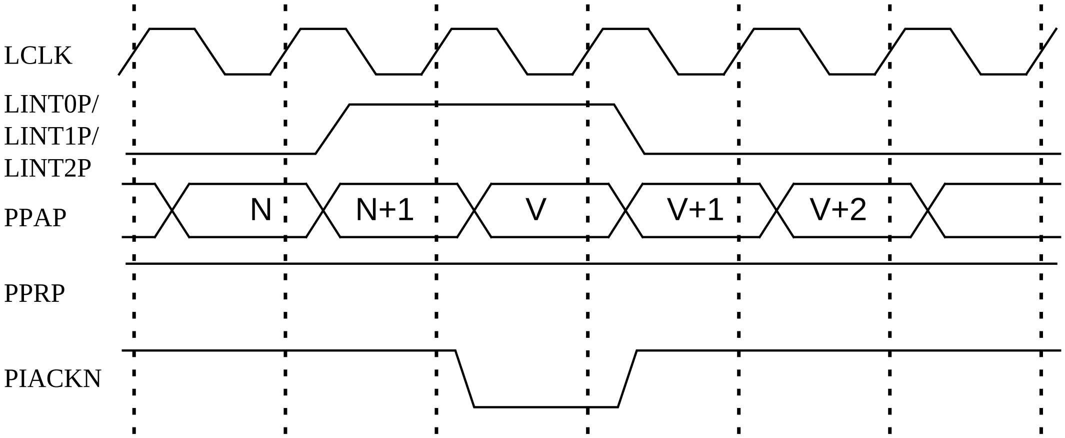


Figure 5-3 INT0/INT1/IN2 Request

Activation of the LINT0P, LINT1P or LINT2P inputs (the core maskable interrupt request pins) prevents instruction N+1 from being executed. The core starts fetching from address 0x0006 for INT0, 0x000E for INT1 and 0x0016 for INT2 and activates the interrupt acknowledge output (PIACKN). PIACKN can be used to reset the activated input.

5.5.3 TRAP/BI

TRAP is a software interrupt, BI is a hardware breakpoint interrupt, and both share the same interrupt vector. During execution of the TRAP/BI service routine other interrupts (i.e. NMI, INT0, INT1, INT2 and BI) requests are disabled.

When a software interrupt, TRAP, is executed the following is performed:
SP - 1 → SP
PC → DVM
PC → (SP)
0x0002 → PC (interrupt starting address)
PIACKN - Interrupt acknowledge signal is activated for a single cycle once the control is being transferred to the service routine.
PTRAPAP - TRAP/BI active. Activated throughout the TRAP service routine.
PSFTP - Software trap indication.

BI is accepted while the core is in an interruptible state (see paragraph 5.5.4). When the core accepts a BI, the following actions are taken:
SP - 1 → SP
PC → DVM
PC -1 → (SP)
0x0002 ->PC (interrupt starting address)
PIACKN - Interrupt acknowledge signal is activated for a single cycle once the control is being transferred to the service routine.
PTRAPAP - TRAP/BI active. Activated throughout the TRAP service routine.

The TRAP/BI service routine *must end* with a *reti* or *retid* instruction.
The *trap* instruction should not be used in a TAPB/BI service routine.

The PSFTP output is used to differentiate between a TRAP and a BI, activated by the instruction (software TRAP) or by the hardware breakpoint interrupt pin. It is used by the TeakLite debugger for emulation actions.

The On Chip Emulation Module (OCEM) uses the BI interrupt to provide emulation capability within the core. When BI is used as hardware interrupt, note the following differences between BI and other interrupts:

- The latency is two instruction cycles (compared to a single cycle in other interrupts).
- BI has the highest priority
- BI has no masking option.
- BI has no option for automatic context switching.

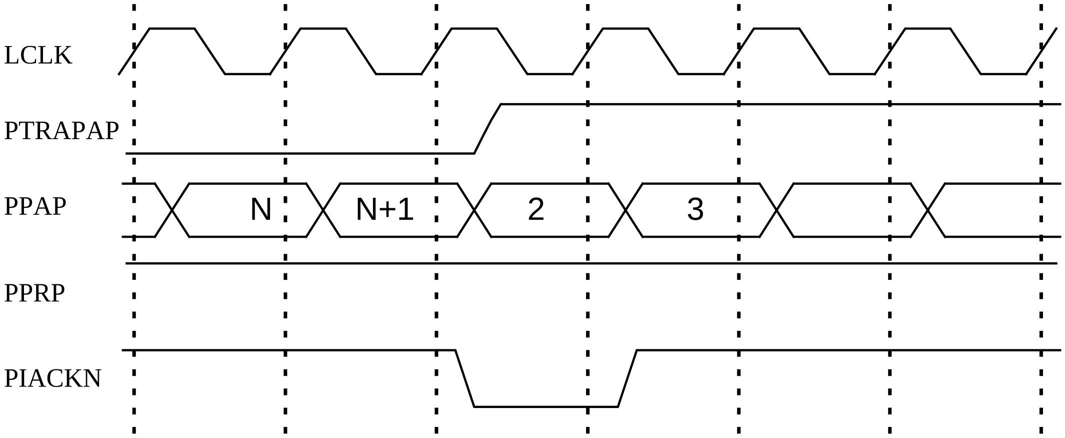


Figure 5-4 *Trap* Instruction Execution

Instruction N+1 is a *trap* instruction, therefore instruction N+1 will not be executed. The PIACKN output (interrupt acknowledge) is activated when the core transfers the control to the service routine. The PTRAPAP output becomes active once the TRAP/BI vector is generated. It is deactivated once a *reti* or a *retid* instruction is executed. The N+1 instruction will be automatically executed after returning from the service routine.

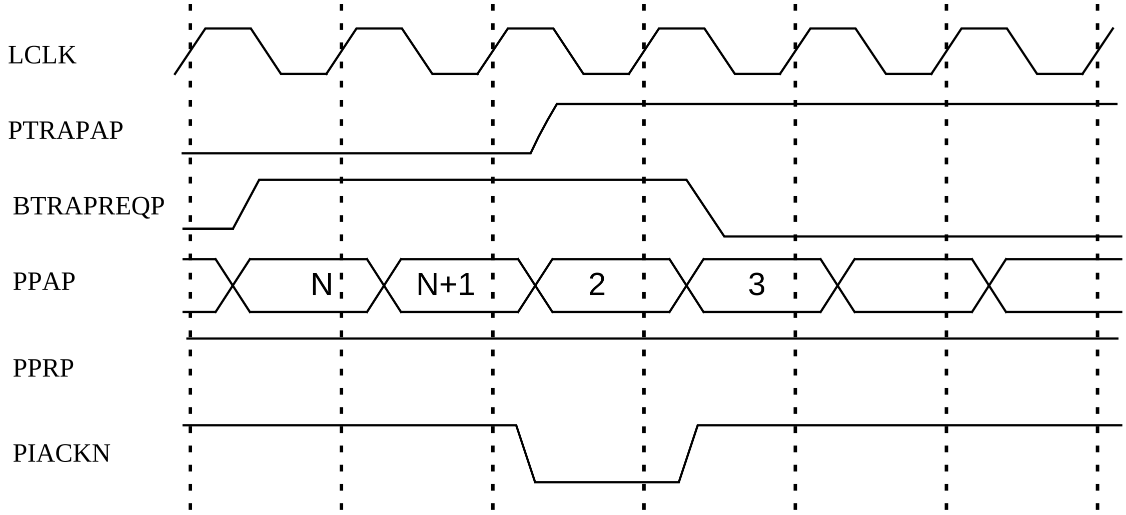


Figure 5-5 BI Request

Activation of the BTRAPREQP input (BI request pin) causes the fetched instructions N+1 and N+2 to not be executed. The core starts fetching from address 2 after a latency of two instruction executions. The output PIACKN (interrupt acknowledge) is activated when the core transfers the control to the service routine. This output can be used to reset the BTRAPREQP input. The PTRAPAP output signal becomes active once the TRAP vector is generated. It is deactivated once a *reti* or a *retid* instruction is executed. The N+1 instruction will be automatically executed after returning from the service routine.

5.5.4 Interrupt Latency

The NMI, INT0, INT1, INT2 interrupts latency is *one* machine cycle, assuming that the processor is in an interruptible state. The BI latency is *two* machine cycles, assuming that the processor is in an interruptible state. During non-interruptible states the core will not service the interrupt, but continue to execute instructions. The interrupt will be accepted once the core exits this state.

The non-interruptible states are as follows:

- During reset (LRSTP signal is active)
- The first three instruction fetches after deactivation of LRSTP input (except for BI).
- During boot procedure (NMI only). Refer to paragraph 5.9, Automatic Boot Procedure.
- Execution of multi-cycle instruction.
- During wait-states.
- STOP mode where no clocks are provided to the core.
- During a nested repeat loop execution.

For specific instructions where interrupts are delayed after their execution, refer to Appendix 5B, More on Interrupt Latency.

5.6 CORE PIN LIST

The following table lists each pin, its functionality, and the off core entity with which it is connected.
Note that all core pins are unidirectional, therefore there is no need for tri-stated core output pins.

The naming convention is as follows: A prefix of one character designates the unit which accounts for that pin, the pin name and a postfix with either P for positive and N for negative polarity.

The prefixes are:

- Core inputs: B for the bus interface unit, L for clock, reset and interrupt related pins and G for all other pins.

- Core outputs: P for PCU, D for DAAU, C for CBU, I for IDU (instruction decoder, a PCU sub section), and O for OFU (operand fetch unit, central data bus implementation) .

The pins in the following table are grouped according to their function. The core pins names that are used with the design methodology pins names, refer to Appendix 5.A in the Signal Names Conversion Table.

Table 5-2 Core Input pins:

Pin name	Size	Source	Description
BBOOTP	1	BIU	BOOT reset indication
BEXTPP	1	BIU	External program indication
BTRAPREQP	1	OCEM	BI interrupt request
LCLKP	1	CLOCK	main clock
LINT0P	1		interrupt 0
LINT1P	1		interrupt 1
LINT2P	1		interrupt 2
LNMIP	1		non maskable interrupt
LRSTP	1	CLOCK	main reset
GXINP	16	X memory	Xspace input data bus
GYINP	16	Y memory	Yspace input data bus
GZINP	16	Z memory	Zspace input data bus

Table 5-2 Core Input pins (Cont'd):

Pin name	Size	Source	Description
GEXTINP	16	External registers	External registers input data bus
PRPAGE	4	BIU	Program page bits
GPIP	16	Program memory	Instruction input bus
GYRAMCONFP	6		Y memory configuration pins
GXRAMCONFP	6		X memory configuration pins
BIUSER0P	1		User input 0
BIUSER1P	1		User input 1
GREFWBAP	1		Causes DZAP to reflect any value latched on the X/Y write buffer address register

Table 5-3 Core Output pins:

Pin name	Size	Source	Description
DXAP	16	X memory	Xspace address
DYAP	16	Y memory	Yspace address
DZAP	16	Zspace	Zspace address
DRXRMP	1	X memory	X memory read strobe
DRYRMP	1	Y memory	Y memory read strobe
DRZRMP	1	Zspace	Zspace read strobe
DWXRMP	1	X memory	X memory write strobe
DWYRMP	1	Y memory	Y memory write strobe
DWZRMP	1	Zspace	Zspace write strobe
OXOUTP	16	X memory	X output data bus
OYOUTP	16	Y memory	Y output data bus
OZOUTP	16	Zspace	Z output data bus
OEXTOP	16	External registers	External registers data bus
ODTVMP	1	OCEM	Data value match indication
DENOMEMP	1		Indication for no data memory access during the following cycle (free access slot)
PENOMEMP	1		Indication for no program memory access during the following cycle (free access slot)
IXSRCP	2	External registers	External registers source bus

Pin name	Size	Source	Description
IXDSTP	2	External registers	External registers destination bus

Table 5-3 Core Output pins (Cont'd)

Pin name	Size	Source	Description
IXRDP	1	External registers	External registers read strobe
IXWRP	1	External registers	External registers write strobe
PPAP	16	Program memory	Program memory address bus
PPOP	16	Program memory	Program data output bus (core to memory)
PPRP	1	Program memory	Program memory read strobe
PPWP	1	Program memory	Program memory write strobe
PEXTIP	1	BIU	MOVP instruction indication
PIACKN	1	BIU	Interrupt acknowledge
PDUMMYP	1	OCEM	Dummy fetch indication
PTRAPAP	1	OCEM	BI/TRAP acceptance indication
PSFTP	1	OCEM	SW interrupt acceptance indication
PSTATUSP	4	OCEM	Machine status for trace buffer logic
CUSER0P	1		user output 0
CUSER1P	1		user output 1
PBKENP	1	OCEM	Block repeat loop indication (appears on the last address of the loop while more loops are left to be executed)
DWWBP	1	OCEM	Indication of write to write buffer
DWBFULXP	1		Write buffer full, containing data for Xspace
DWBFULYP	1		Write buffer full, containing data for Yspace

5.7 MEMORY INTERFACE

The core memory interface includes separate interfaces for program memory and data memory. Separate buses and controls are provided for each memory type. The transactions can be of read type or write type for both the data and the program memories. Slow memory transactions are also supported, using a wait-state mechanism.
All address/data busses are unidirectional static busses, i.e. dedicated busses for read and write, the selection between different sources of a bus is made using muxes, without any tri-state buffers.

5.7.1 Synchronous and Asynchronous Memory Interfaces

For both program and data memories (except the Z space), synchronous and asynchronous memory models are supported. The program and data memories do not have to use the same type of memory, that can be defined by PSYNC and DSYNC configurable switches (for program and data memories, respectively). The Z data space interface is always asynchronous. However, user can choose registered Z core output data bus interface, or non-registered (by switch ZSYNC), as will be illustrated below.
When work in synchronous program memory interface, the user can choose PSYNC_FAST that allows operating with better program address and controls valid time. However, using of this mode imposes an extra cycles to some of instructions, as specified in section 4.4.

5.7.1.1 Asynchronous Memories

When using the Asynchronous Interface, all core outputs are registered, hence they are stable immediately after the rising edge of core’s clock (LCLKP). These include the address buses, the program page bus, the data out buses and the read and write controls.

When this interface is used, power dissipation of the core is kept to a minimum since the address values are always stable. An address will be changed only if memory access is performed in a given cycle.

Following are typical read and write transactions for the Asynchronous Interface:

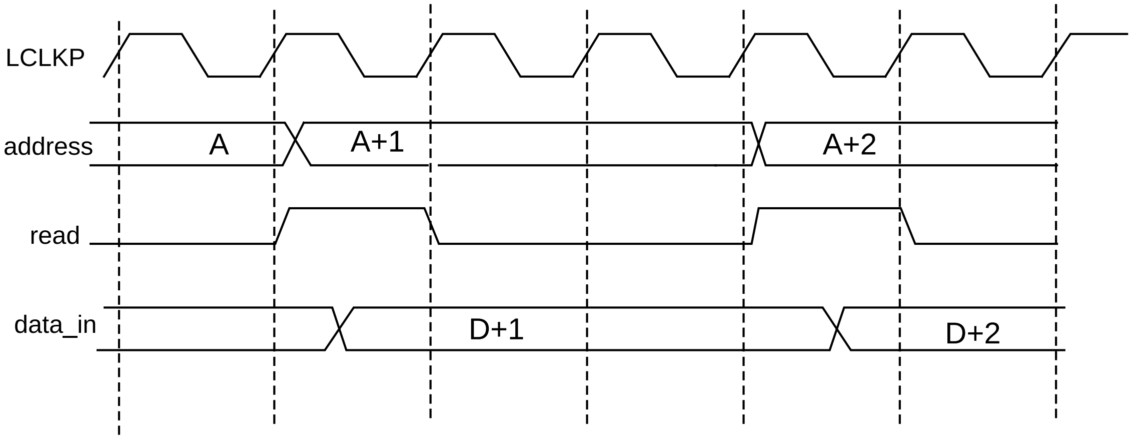


Figure 5-6 Read Transaction With Asynchronous Memory

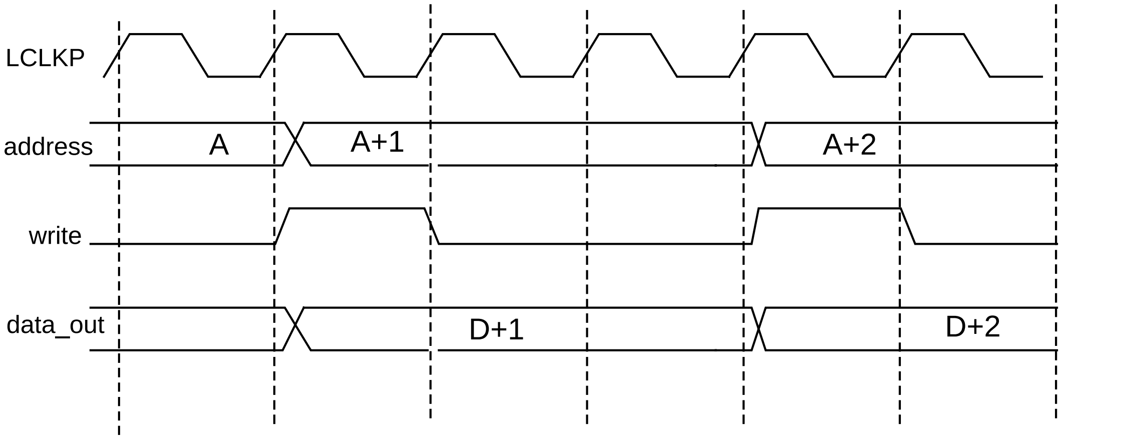


Figure 5-7 Write Transaction With Asynchronous Memory

5.7.1.2 Synchronous Memories

There are two methods for using synchronous memories with TeakLite core:

1. Using the same memory interface as for asynchronous memory, all core outputs are registered, and the memory uses skewed clock, so that the memory's inputs have a suitable setup time before the rising edge of its clock.

Instead of skewed clock, the falling edge of the clock can be used, so that the memory has half a cycle to operate.

The read transaction with synchronous memory using a skewed clock is illustrated on Fig. 5.8.

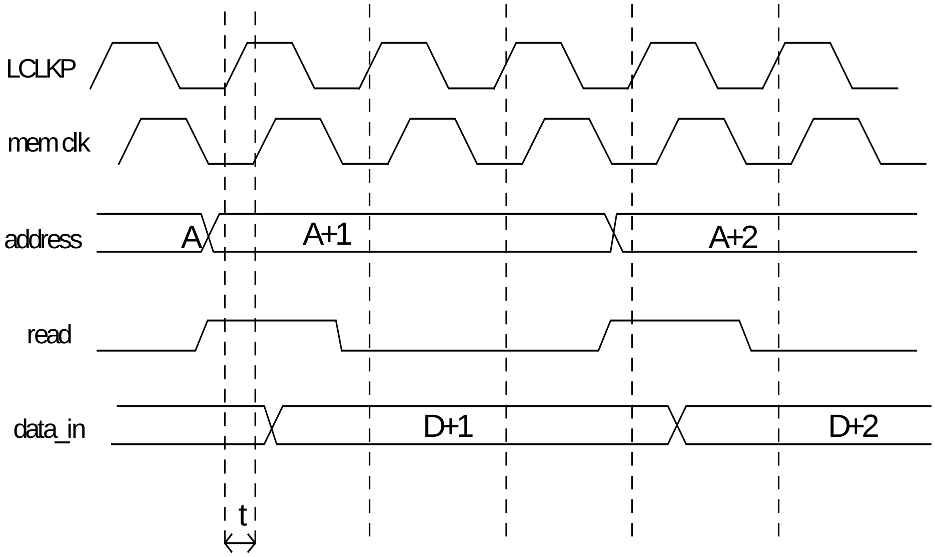


Figure 5-8 Synchronous Read Transaction With Skewed Clock

2. Using a synchronous memory interface, whenever a memory access occurs, the core outputs are non-registered. Hence, they are valid before the rising edge of the clock, allowing a setup time for the memories. These include the address buses, the program page bus, the data-out buses and the read and write controls.

Following are read and write transactions using this interface:

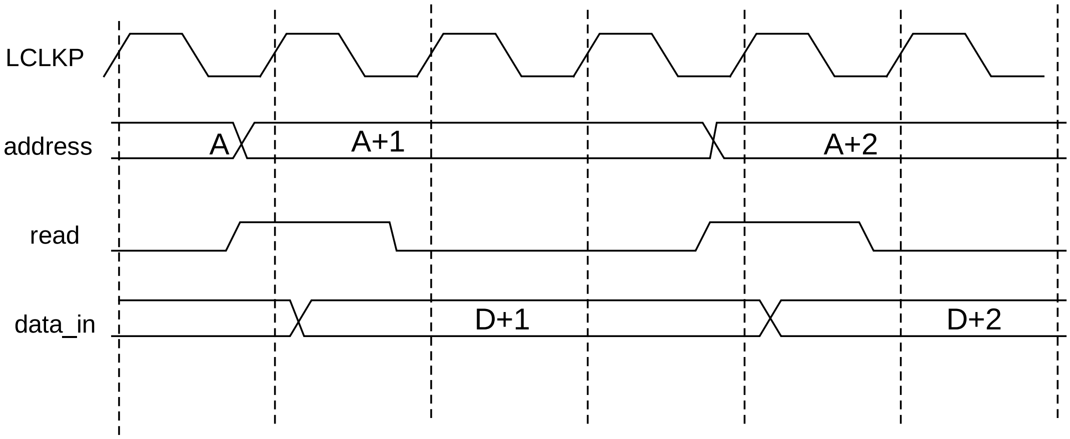


Figure 5-9 Synchronous Read Transaction (Non-Registered Outputs)

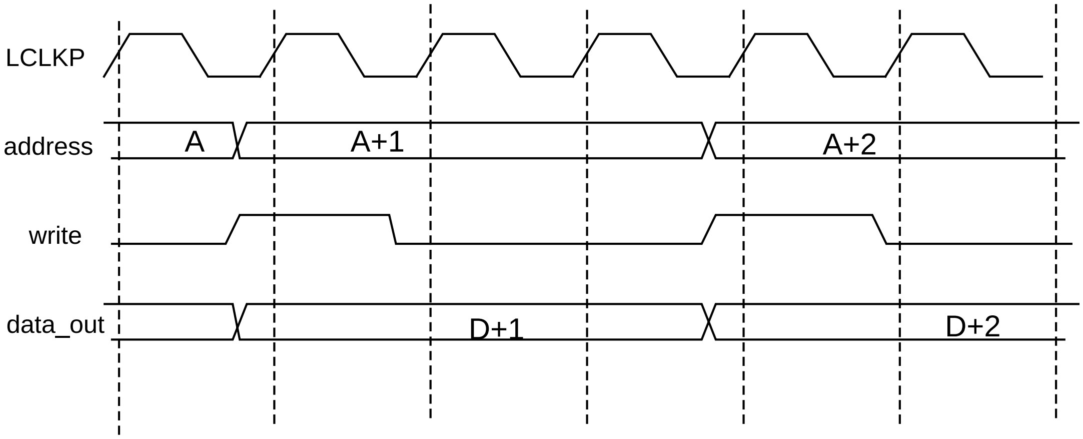


Figure 5-10 Synchronous Write Transaction (Non-Registered Outputs)

Note: When synchronous interface is used, address buses and read and write controls should have appropriate setup time before clock rising edge in order to be sampled by the memory.

5.7.2 Memory Interface With Slow Memory Devices

The core supports transactions with slow memory devices (either program or data) using a clock stop mechanism.

During wait states, the TeakLite main clock, LCLKP, is disabled by the off-core logic. Thus, the memory transactions are stretched without consuming any power inside the core; all address/data and read/write strobes are stable during that time.

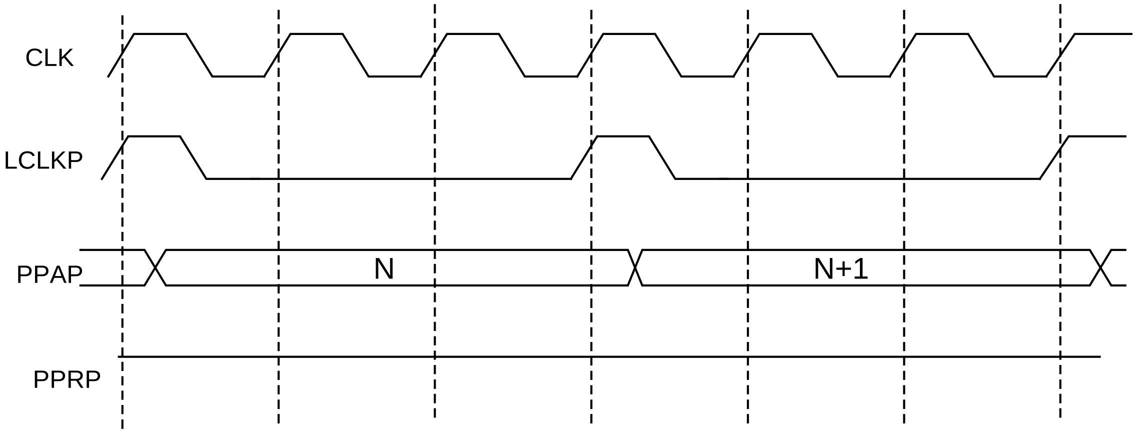


Figure 5-11 Core Operation Using Wait States

5.7.3 Program memory

The TeakLite supports up to 64K of program space.
The core is using at 16-bit program address bus - PPAP, instructions are fetched by the core using a 16-bit bus - GPIP. These busses may have multiple sources such as on-chip ROM/RAM or off chip ROM/RAM. The user's glue logic has to arbitrate among the potential sources.
Program memory write transactions are supported using a 16-bit bus - PPOP, data written into program space is passed on these busses.
The TeakLite gives partial support up to 16 program pages for correct performing of hardware loops. However, the generation of program paging should be done in off-core logic to provide 4-bit program page input bus to the core.

5.7.3.1 Program Paging Support

Although the program paging is implemented off-core, the changing of the program pages should be performed as following:

Disable all interrupts
Inst1
Inst2

When Inst1 is change the program page, and have to be single word single cycle instruction. Inst2 must be one of the following: *CALL/BR/CALLA/RET/RETI/RETS*.

5.7.4 Data Memory

5.7.4.1 Xspace and Yspace data memory interface

The following interface signal set comprises the memory interface, two such sets exists, one for X, the other for Y:

- Memory address - DXAP/DYAP, 16-bit address busses, issued directly off a register, those busses are valid throughout the entire memory access cycle.
- Memory read strobe - DRXRMP/DRYRMP, issued directly off a register, those signals are valid throughout the entire memory access cycle.
- Memory write strobe - DWXRMP/DWYRMP, issued directly off a register, those signals are valid throughout the entire memory access cycle.
- Core to memory data busses - OXOUTP/OYOUTP, 16 bit data busses, issued directly off registers, those busses are valid throughout the entire memory access cycle.
- Memory to core data busses - GXINP/GYINP, 16 bit data busses, issued by the memory in a required time setup with relation to the clock rising edge of the next cycle (with respect to the cycle in which address and read strobe were asserted).

All figures below illustrate transactions with asynchronous memory.

All read and write transactions are performed in the same way for X and Y memories.

The following figure illustrates a read transaction from an X memory address:

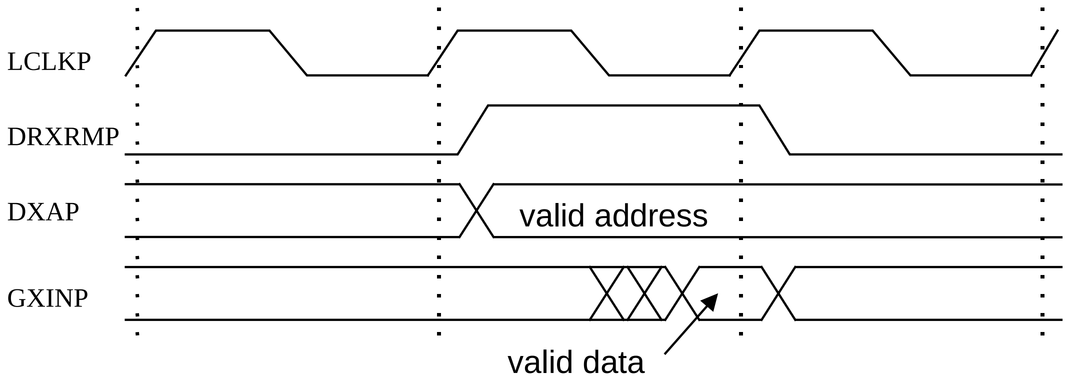


Figure 5.12 Xspace Memory Read

Address and read strobes are issued at the beginning of the read cycle, memory is expected to produce the data in sufficient setup time with respect to the next clock rising edge.

This memory read sequence has the following advantages:

- Both synchronous and asynchronous memory elements can be used for memory read purposes (allows for the attachment of asynchronous ROM or Flash modules).
- Power dissipation is kept to a minimum, address values are always stable. Address will change only if a memory access is performed in the given cycle.

The following figure illustrates a single operand write to an Y memory address**:**

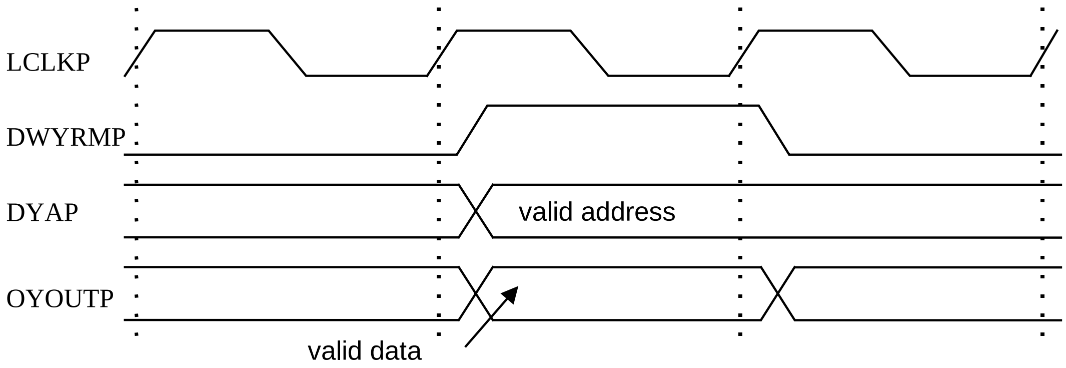


Figure 5.13 Yspace Memory Write

The write strobe, memory address and data are all issued following the clock rising edge of the write cycle, all the above signals will be stable until the next clock rising edge.

5.7.4.2 Zspace data memory interface

The following interface signal set comprises the Zspace interface:

- Address - DZAP (same as for Xspace), 16 bit address bus, issued direct off a register, valid throughout the entire Z access cycle.
- Read strobe - DRZRMP, issued directly off a register, valid throughout the entire Z access cycle.
- Write strobe - DWZRMP, issued directly off a register, valid throughout the entire Z access cycle.
- Core to Z data bus - GZOUTP, 16-bit data bus, issued directly off a register, **issued in the cycle following the write strobe assertion**.
- Z to Core data bus - GZINP, 16 bit data bus, issued by Zspace in a required setup time with relation to the clock rising edge of the next cycle (with respect to the cycle in which address and read strobe were asserted).

The following figure illustrates a single operand read from Zspace address :

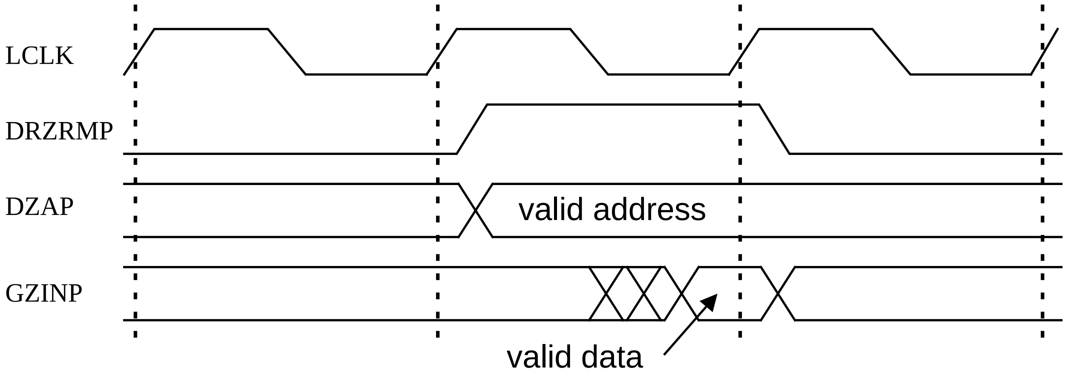


Figure 5.14 Zspace Memory Read

This transaction is identical to the X/Y read transaction.

The following figure illustrates a Zspace write cycle :

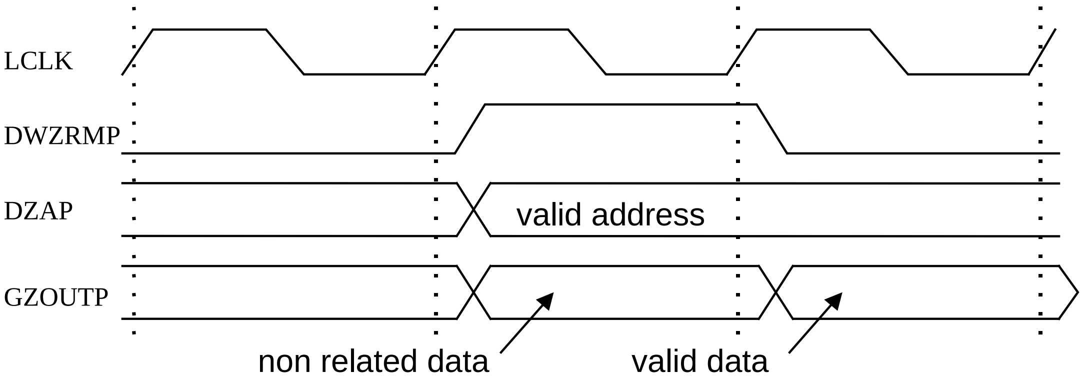


Figure 5.15 Zspace Memory Write

Contrary to X/Y write transactions, the write data, issued by the Core, is presented in the cycle, immediately following the write strobe and the relating address value.

The following figure illustrates the problem caused by a Z space write followed by read:

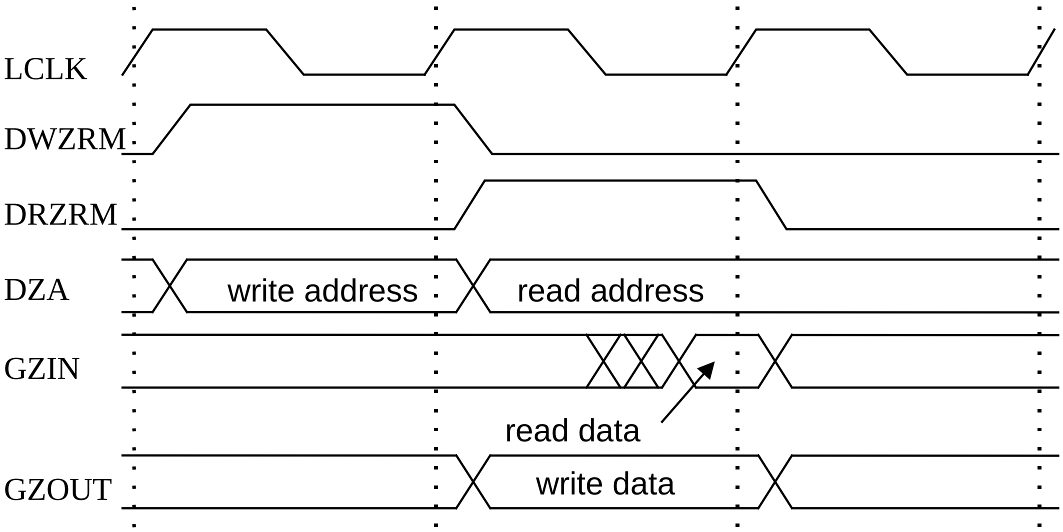


Figure 5.16 Zspace Memory Read after Write

As it is illustrated on Figure 5.16, the Z space data-out bus for the write transaction will be valid in the same cycle as the data-in bus for the read transaction (which was initiated after the write). The actual write control to Z space memory (that is generated by off-core logic) is activated simultaneously with Z space read control. This problem can be solved by insertion of a wait state as shown in Figure 5.17 and Figure 5.18.

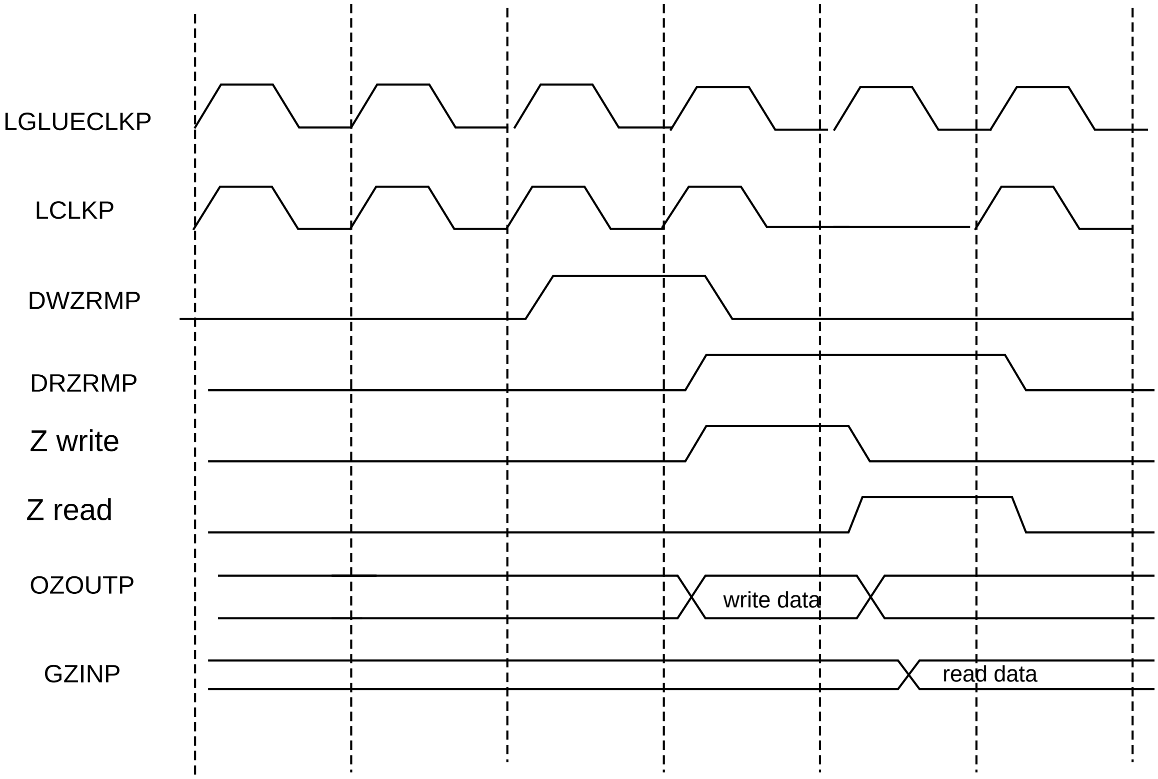


Figure 5.17 Z space read after write, registered OZOUTP with one Wait State

When the registered OZOUTP interface is used, the core generated Z space memory write strobe and address are delayed for one cycle by the off core logic. In this case, the actual write control (Z space memory write on Fig.5.17) and core generated read strobe are activated at the same time. This situation can be recognized by the off core logic, that will halt the TeakLite clock, LCLKP, for one cycle (TeakLite clock can be halted for more than one cycle if multiple wait state is needed). During the Wait State, all TeakLite outputs are unchanged. When the write transaction is completed, the read transaction can be performed by the off core logic, when actual read control (Z space memory read on Fig.5.17) is activated one cycle after the write transaction.

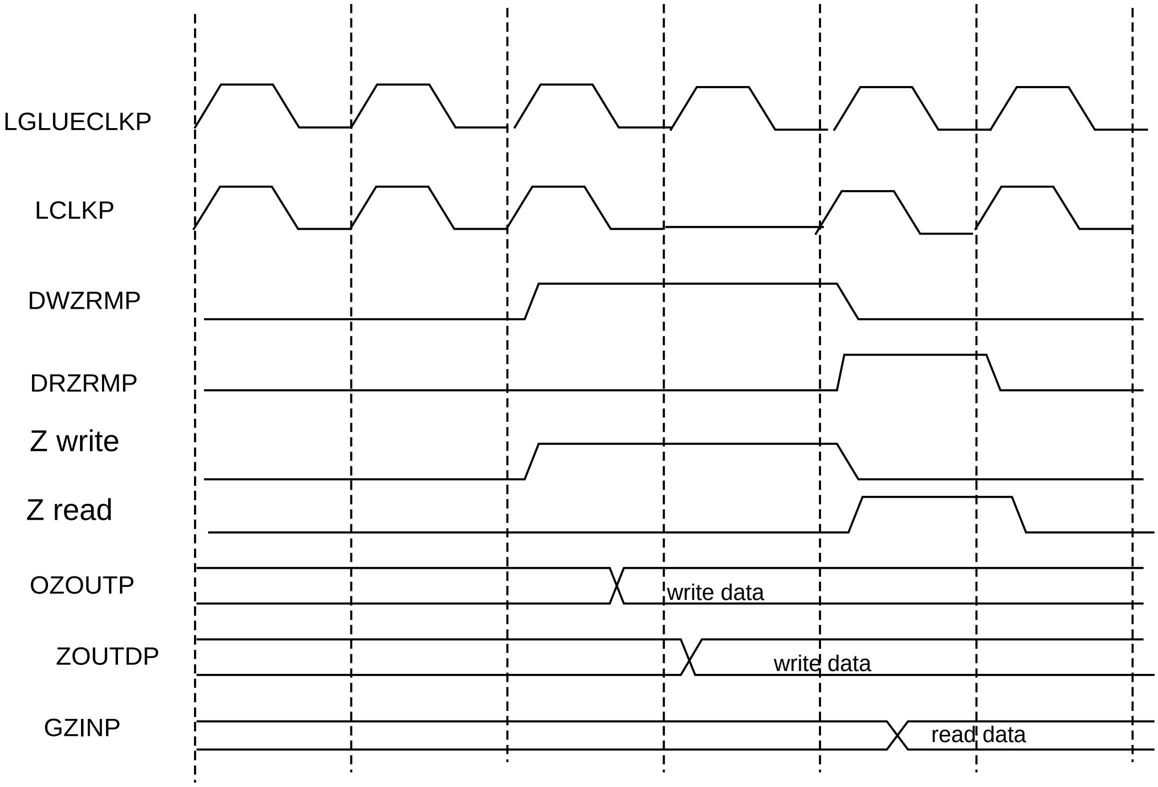


Figure 5.18 Z space read after write, non-registered OZOUTP with one Wait State

When the non-registered OZOUTP bus is used, there is no need in delayed write strobe and address bus. The off core logic can insert a wait state immediately at activation of the TeakLite write strobe. At next system clock rising edge (when the core's clock is halted), the OZOUTP can be sampled by the off core logic (ZOUTDP bus on Fig.5.18), and the registered data will be transferred to the memory. The TeakLite read strobe is generated in the next TeakLite cycle, when the write transaction is completed.

5.8 USER DEFINED REGISTERS

The TeakLite supports four user-defined registers, enabling future expansion of the core residing in off core glue logic. The user-defined registers are part of the core register list, meaning that they can be accessed by most of the TeakLite instructions.

The core issues a dedicated set of read and write strobes for the external register interface (IXRDP for reading, IXWRP for writing). The addressed external register is decoded via a dedicated set of source and destination outputs issued by the core (IXSRCP for source. IXDSTP for destination). An external register transaction is qualified by read or write strobes, the source and destination indications will reflect the previously selected external register until a new external register transaction takes place. The following table presents the external bus source and destination decoding.

Table 5.4 User Defined Register Coding

IXSRCP/IXDSTP	**User defined register**
00	ext0
01	ext1
10	ext2
11	ext3

Two dedicated buses are used for support of user defined registers: core to external register data bus, OEXTOP, for writing into the external registers, and GEXTINP for reading from external registers.

Since there may be only four external registers and since it is assumed that the external register mechanism will be used for accelerating timing critical tasks, contrary to the data busses issued by the core to X, Y and Z, the core to external registers data bus is unregistered. It is assumed that the external registers will actually serve as the data registering stage. OEXTOP is issued by the core within the OF stage with a limited setup time, this bus must be immediately fed into the external registers, combinatorial logic must not be placed between OEXTOP and the actual external registers.

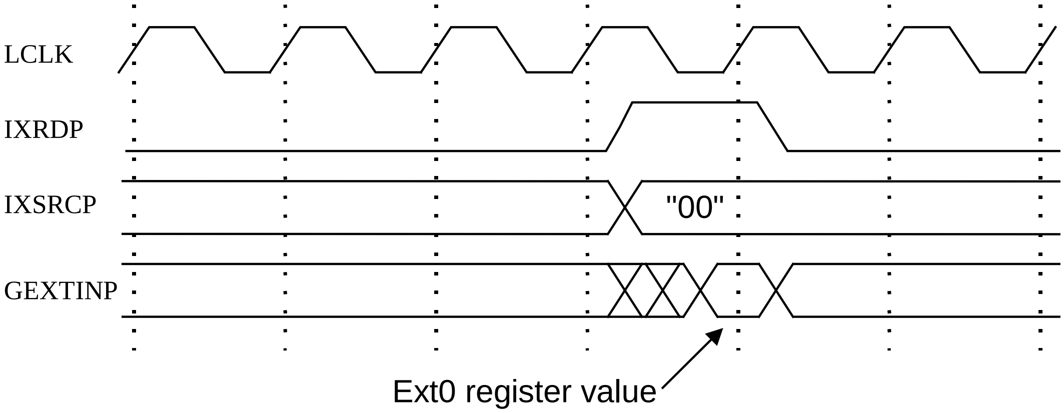


Figure 5.19 User Defined Register Read

A user-defined register read transaction is identical to a data memory read, the read strobe and external register source decoding bus are issued in the OF stage. Data is sampled in the relevant external register at the next clock rising edge.

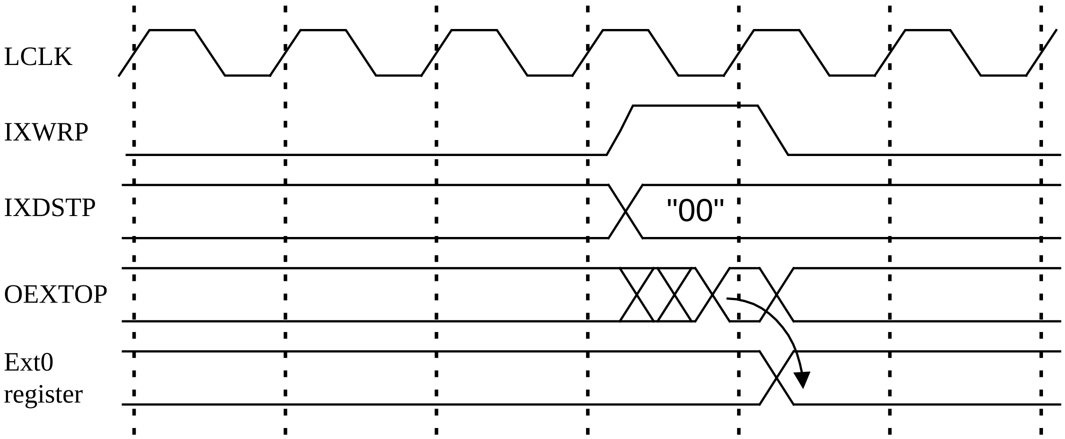


Figure 5.20 User-Defined Register Write

User-defined write transaction. The write strobe and external register destination decoding bus are issued in the beginning OF stage. The data bus, OEXTOP, is issued by the core well within the OF stage, this bus will be registered by one of the external registers before the next clock rising edge.

5.9 AUTOMATIC BOOT PROCEDURE

A boot procedure is a mechanism that allows automatic code down loading from a memory device located in the data space (e.g. EPROM, RAM) to a RAM device located on the program space.

The core supports the boot mechanism by automatic insertion of the instruction *brr #-3* as the first fetched instruction after reset. It is accomplished only if the core pin BBOOTP is valid during de-activation of LRSTP (the core reset pin). The insertion of the *brr #-3* instruction cause the next instruction to be fetched from address 0xFFFE. A branch instruction to the boot routine can be inserted at addresses 0xFFFE and 0xFFFF.

Notes:

1. The boot routine must end with a jump to the beginning of the main routine to start the main program.
2. The boot routine can use the *movd* instruction for code downloading from a data memory (slow EPROM or ROM) to a program RAM within the program space.

5.10 ON-CHIP EMULATION SUPPORT

The TeakLite core can be combined with an On-Chip Emulation Module (OCEM). The OCEM functions are hardware emulation and program- flow tracing.

Hardware emulation allows the setting of break points in accordance with a pre-defined condition such as a program address match, single stepping, a data address match etc
Program flow tracing consists of recording into a buffer (during run time), those instruction addresses that cause non-continuity in the program flow (e.g. *br, ret*). Those addresses are kept in a FIFO within the OCEM block, and are used afterwards to reconstruct the complete program flow graph.
The OCEM module uses the core BI mechanism. The pin BTRAPREQP is used as an indication for an OCEM break point.

The program-flow trace buffer control within the OCEM module uses dedicated core status outputs as an indication for special instructions performance (i.e. instructions that cause non-sequential fetches). These indications are decoded within a 4-bit bus (PSTATUSP) as described in Table 5-6 below.

Table 5-5 Status Pins

PSTATUSP	Explanation
0000	Null (No pipeline break)
0001	Reserved
0010	Block repeat loop[1]
0011	Taken Branch[2].
0100	Computed Branch / Return [3].
0101	Relative Branch[4].
0110	Move to PC[5].
0111	*movp* instruction.
1XXX	Reserved.

[1] Activated once moving from the last to the first address, an entire *bkrep* instruction.

[2] Activated three machine cycles after one of the following instructions has been fetched: *br, brr, call* or *callr* and the condition is met.

[3] Activated three machine cycles after one of the following instructions has been fetched: *calla, ret, reti* or *rets.*

[4] Activated two machine cycles after one of the following instructions has been fetched: *brr* or *callr*.

[5] Activated three machine cycles after one of the following instructions has been fetched: *retd; retid; mov source,pc* or *movp (aXl), pc.*

5.11 DMA SUPPORT

The TeakLite has a built-in support for DMA transactions. The same mechanisms are applied to the program and data memory interfaces. Since the core does not tri-state its memory interface signals, the DMA memory access logic must be multiplexed with the core memory access signals.

When implementing a DMA engine for the X or Y data memory, care must be taken to maintain coherency with the write buffer contents. Two options are available to accomplish the above (There are two options for achieving the above):

Limit shared DMA-core data memory regions to one way traffic (a certain data memory address can only be read by one of the two entities and written by the other entity).

In this manner, the write buffer data will never contradict any DMA write transactions.
b. DMA engine should initiate a transaction only when the "no memory transaction" indication (DENOMEMP for data memory, PENOPMEMP for program memory) was active in the previous cycle. It is guaranteed by design that the assertion of this signal by the core is performed only after the write buffer has been emptied.
c. DMA engine may also monitor the write buffer full indications (DWBFULXP, DWBFULYP) and act accordingly.
Two types of DMA modes are supported:

5.11.1 Cycle Stealing DMA Transactions

In this mode, the DMA "steals" cycles in which the core does not address the memory. A dedicated signal (DENODMEMP for X, Y, Z data memory and PENOPMEMP for the program memory) is issued by the core to indicate that the memory interface will not be accessed by the core during the next cycle. In the cycle, immediately following this signal's assertion, the DMA will take charge of the memory interface and perform a memory access.

Since cycle stealing does not interfere with the core's activity, this mode should be used whenever the required DMA data transfer rate is low.

5.11.2 Burst mode DMA transactions

In this mode, the DMA gains control over the memory interface by generating wait states towards the core. Once the core is "halted", the DMA engine takes over the memory interface for as long as required. Since the memory control logic is multiplexed between the core and the DMA engine (as opposed to a tri-state solution, for example), the state of the core during the DMA "take over" is not relevant (the core may have been accessing memory when it was halted by the DMA).

5.12 PROGRAM PROTECTION MECHANISM

A program protection mechanism is available to protect the user’s program from being read without authorization. The user program when placed on-chip can be read through the usage of a *movp* instruction. Therefore, it can move data from the on-chip ROM/RAM into the data memory where it can be viewed.
Off-core glue logic is necessary to activate the input BEXTPP, which indicates that the current instruction is fetched from external program memory. Once the fetched instruction is *movp,* the core activates the output PEXTIP. In this case the off-core glue logic has to disable the read from the internal ROM/RAM, preventing the protected memory from being read.

5.A SIGNAL NAMES CONVERSION TABLE

The following table can be used for the conversion of core signal names which appeared in previous chapters into pin names that comply with design methodology, data base name, used in chapter 5.

Table 5A Signal Names Conversion Table

Chapters 1 - 4 Signal Name	**Data Base Name**
PAB	PPAP
PDB	GPIP – for core to memory, PPOP - for memory to core
XAB	DXAP
XDB	OXOUTP – for core to memory, GXINP - for memory to core
YAB	DYAP
YDB	OYOUTP – for core to memory, GYINP - for memory to core
ZAB	DZAP
ZDB	OZOUTP – for core to memory, GZINP - for memory to core
RESET	LRSTP
NMI	LNMIP
INT0	LINT0P
INT1	LINT1P
INT2	LINT2P
IUSER0	BIUSER0P
IUSER1	BIUSER1P
OUSER0	CUSERO0P
OUSER1	CUSERO1P

5.B MORE ON INTERRUPT LATENCY

This appendix includes additional details regarding interrupt latency in addition to the details described in Chapter 5 in Section 5.5.4, Interrupt Latency.

If the processor begins to handle one of the maskable interrupts (INT0, INT1 or INT2), an NMI will be accepted only after the execution of the instruction at the maskable interrupt vector.

Note that the delay listed in the following table is only the additional increment to be added to the normal interrupt latency figures.

Table 5B Interrupt Latency after Specific Instructions

Current Instruction	**Interrupt delay after the inst.**	**Interrupt**
br, call, ret or reti when the condition is not met; *mov soperand*[1]*, sp ; movp (aX1). sp*; *set/rst/chng/addv/subv ## long immediate, sp*; Last repetition of instruction during a repeat loop; First instruction executed after returning from a TRAP/BI routine.	One cycle	0,1,2,NMI

mov soperand[1]*, st0; movp (aX1), st0;*
set/rst/chng/addv/subv ## long immediate, st0;
pop st0;
	Two cycles	0,1,2

mov soperand[1]*, st2; movp (aX1), st2;*
set/rst/chng/addv/subv ## long immediate, st2;
pop st2
	Two cycles	2

## long immediate, st0	One cycle	0,1,2
## long immediate, st2	One cycle	2
retd,retid,mov soperand[1]*, pc; movp (aX1), pc;pop pc;*	Two cycles	0,1,2,NMI,BI
mov ## long immediate, pc	One cycle	0,1,2,NMI,BI
rep	Two cycles	0,1,2,NMI,BI

[1] *soperand* represents every source operand except for a *## long immediate.*