Programming Reference Guide

xPIC Instruction Set

netX 6/10/51/52

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC141201PRG01EN | Revision 1 | English | 2015-02 | Released | Public

Table of Contents

1 Introduction

1.1 About this document

This manual describes the xPIC, its purpose, use, tasks, functions and instruction set.

1.2 List of revisions

Rev	Date	Name	Chapter	Revision
1	2015-01-16	HN	all	Created.

Table 1: List of revisions

1.3 Terms and abbreviations

Abbreviation	Explanation
ADC	Analog-to-Digital Converter
ADRU	Address unit
AHB	Advanced High-performance Bus
AHBL	Advanced High-performance Bus Light
ALU	Arithmetic Logic Unit
ARM	Advanced RISC Machines Limited (name of company)
BCD, bcd	Binary Coded Decimal
bs	bitwise smaller
bs	byte swap
CAP	CAPture
clz	count leading zeros
cond	conditional jump with the flag
const	constant
CPU	Central Processing Unit
DMAC	Direct Memory Access Controller
DPM	Dual-Port Memory
DRAM	Dynamic Random Access Memory
ENC	ENCoder
FIQ	Fast Interrupt Request
GPIO	General Purpose Input Output
HIF	Host InterFace
iem	interrupt enable mask
iev	interrupt enable value
INTRAM	INTernal SRAM
IRQ	Interrupt Request
ISR	Interrupt Service Routine
LSB	Least Significant Bit
MAS	Memory Access Size
MP	**M**icro-**P**rocessor
MPWM	Motion Pulse Width Modulation Unit
MSK	MASK or MASKED
opcode	operation code

Abbreviation	Explanation
pc	program counter
PRAM	Programmable Random Access Memory
PWM	Pulse-Width Modulation
R	Read
RAM	Random Access Memory
reg	register
REQ	REQuest
RISC	Reduced Instruction Set Computing
R/W	Read/Write
sconst	shift constant
SPI	Serial Port Interface
src	source
st	status register
stat	status
SW IRQ	SoftWare Interrupt Request
trg	target
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
USR	USer Register
VAL	VALue
VIC	Vectored Interrupt Controller
W	Write
wcond	work register condition
WDG	Watchdog
wreg	work register
ws	word swap
xpec	Flexible Protocol Execution Controller
xPIC	Flexible Peripheral Interface Controller

Table 2: Terms and abbreviations

1.4 Legal notes

1.4.1 Copyright

1.4.2 Important notes

The user manual, accompanying texts and the documentation were created for the use of the products by qualified experts, however, errors cannot be ruled out. For this reason, no guarantee can be made and neither juristic responsibility for erroneous information nor any liability can be assumed. Descriptions, accompanying texts and documentation included in the user manual do not present a guarantee nor any information about proper use as stipulated in the contract or a warranted feature. It cannot be ruled out that the user manual, the accompanying texts and the documentation do not correspond exactly to the described features, standards or other data of the delivered product. No warranty or guarantee regarding the correctness or accuracy of the information is assumed.

We reserve the right to change our products and their specification as well as related user manuals, accompanying texts and documentation at all times and without advance notice, without obligation to report the change. Changes will be included in future manuals and do not constitute any obligations. There is no entitlement to revisions of delivered documents. The manual delivered with the product applies.

Hilscher Gesellschaft für Systemautomation mbH is not liable under any circumstances for direct, indirect, incidental or follow-on damage or loss of earnings resulting from the use of the information contained in this publication.

1.4.3 Exclusion of liability

The software was produced and tested with utmost care by Hilscher Gesellschaft für Systemautomation mbH and is made available as is. No warranty can be assumed for the performance and flawlessness of the software for all usage conditions and cases and for the results produced when utilized by the user. Liability for any damages that may result from the use of the hardware or software or related documents, is limited to cases of intent or grossly negligent violation of significant contractual obligations. Indemnity claims for the violation of significant contractual obligations are limited to damages that are foreseeable and typical for this type of contract.

It is strictly prohibited to use the software in the following areas:

- for military purposes or in weapon systems;
- for the design, construction, maintenance or operation of nuclear facilities;
- in air traffic control systems, air traffic or air traffic communication systems;
- in life support systems;
- in systems in which failures in the software could lead to personal injury or injuries leading to death.

We inform you that the software was not developed for use in dangerous environments requiring fail-proof control mechanisms. Use of the software in such an environment occurs at your own risk. No liability is assumed for damages or losses due to unauthorized use.

1.4.4 Warranty

Although the hardware and software was developed with utmost care and tested intensively, Hilscher Gesellschaft für Systemautomation mbH does not guarantee its suitability for any purpose not confirmed in writing. It cannot be guaranteed that the hardware and software will meet your requirements, that the use of the software operates without interruption and that the software is free of errors. No guarantee is made regarding infringements, violations of patents, rights of ownership or the freedom from interference by third parties. No additional guarantees or assurances are made regarding marketability, freedom of defect of title, integration or usability for certain purposes unless they are required in accordance with the law and cannot be limited. Warranty claims are limited to the right to claim rectification.

1.4.5 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different counters, in particular those of Germany and the USA. The software may not be exported to countries where this is prohibited by the United States Export Administration Act and its additional provisions. You are obligated to comply with the regulations at your personal responsibility. We wish to inform you that you may require permission from state authorities to export, re-export or import the product.

1.4.6 Registered trademarks

2 Functional description

2.1 Overview

The flexible Peripheral Interface Controller xPIC is a 100 MHz CPU for real-time applications and time-critical processes. It has been designed to process fast I/O signals with a latency time of down till five clock cycles.

The xPIC is a general purpose 32-bit RISC CPU with an 8 KB local data RAM, an 8 KB local program RAM, and 2 AHBL channels for data and instructions. The xPIC is used for data exchange with sensors as well as for special communication tasks in the data link layer. The xPIC has a debug unit for hardware breakpoints/watchpoints and separate busses to peripheral components.

The xPIC can be used as a fast CPU to filter, analyze, collect, convert and process sensor data, ranging from simple IOs to complex encoders and sensors with an analog frontend. Another area of application is the control of any actuators ranging from simple digital and analog IOs, pumps, valves, or switches to the control of virtually any type of electric motor.

The xPIC can realize many kinds of control applications in the motion or process control field. The xPIC is also used for the IO-Link controller and the additional third Ethernet MAC channel. In that case the xPIC is not available for user applications. The key concept of the xPIC is the deterministic instruction execution time. As long as the local RAM is used only, one instruction is executed with each clock cycle.

netX 51/52-specific

On netX 51/52, the xPIC is used as an additional CPU which relieves the ARM CPU, e.g. by preprocessing data and taking over the IO-Link data transfer.

netX 10-specific

On netX 10, the xPIC is used as an additional CPU which relieves the ARM CPU, e.g. by preprocessing data and taking over the IO-Link data transfer.

Note: The following commands are not available on netX 10:
bcd2hex, hex2bcd, gie, gsie, sie and S7 commands.

netX 6-specific

On netX 6, the xPIC is the only CPU with all required functions.

2.1.1 Features

- 32-bit RISC CPU, 100 MHz, 3-address architecture
- 8 work registers, 5 user registers
- 8 shadow work registers for fast interrupt switch
- 2 AHBL master channel (data, instruction)
- 8 KB local program RAM, 8 KB local data RAM
- 32-bit hardware signed/unsigned multiplication (64-bit result)
- conditional execution on many instructions
- special instructions for saturation, min. and max. in signed and unsigned
- load/store architecture and loop support
- special instructions with load and arithmetical combination
- access to the entire periphery; short bus cycles
- separate busses to peripheral components
- debug unit with 2 hardware break/watchpoints
- vector interrupt controller, fast interrupt, nested interrupt support
- deterministic instruction execution time

2.1.2 Block diagram

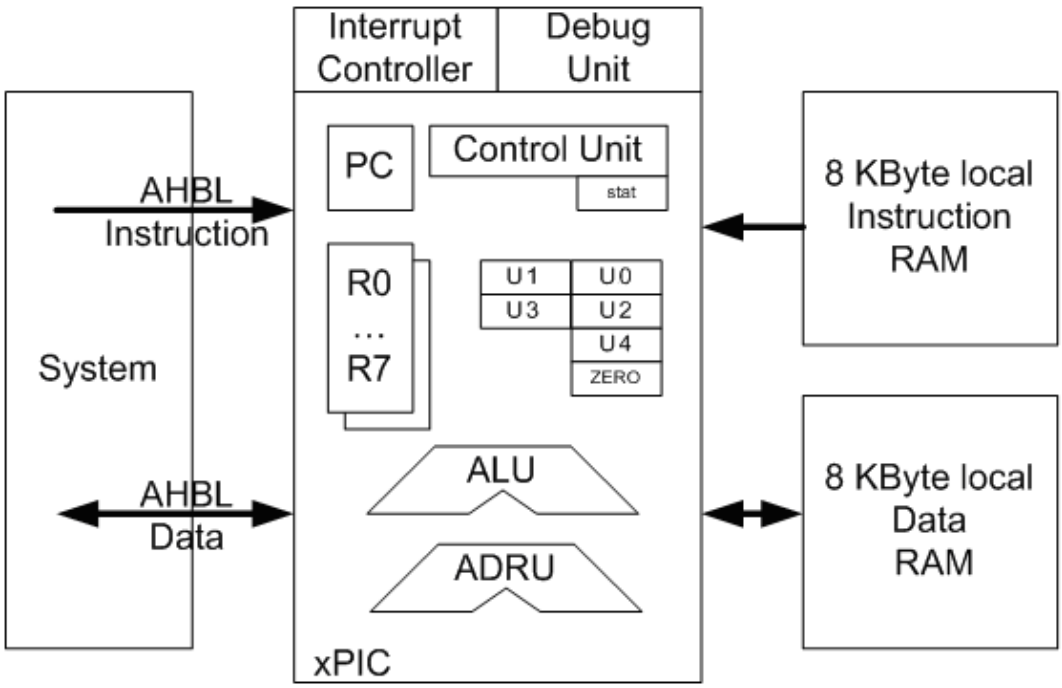


Figure 1: xPIC block diagram

2.1.3 Use cases

The xPIC can be used

- for real-time applications and time-critical processes
- for processing fast IO signals
- for data exchange with sensors
- for special communication tasks in the data link layer
- for data preprocessing and IO-Link data transfer
- for filtering, analyzing, collecting, converting and processing sensor data ranging from simple IOs to complex encoders and sensors with an analog frontend
- for controlling any actuators ranging from simple digital and analog IOs, pumps, valves or switches to virtually any type of electric motor
- for the IO-Link controller and the additional third Ethernet MAC channel (in that case, however, the xPIC is not available for user applications)

2.2 Internal registers

The xPIC has 16 internal registers as source and target.

Register	Description
r0..r7	**Work registers, 32 bit** ▪ support auto-in/decrement ▪ can be used as address offset ▪ shadow register bank for fast interrupt
u0..u4	**User registers, 32 bit** ▪ simple work registers without auto-in/decrement ▪ can only be used as address base ▪ u0/1 and u2/3 are used as one 64-bit target register by multiplication commands (write u10, u32)
pc	**Program counter, 32 bit** ▪ special CPU register which contains the memory addresses of the current/next command
st	**Processor status, 32 bit** ▪ bit[6:0] are ALU flags [gs, gu, e, v, s, c, z] ▪ these bits may be written by any commands ▪ bit[25:7] are reserved for internal processor status flags, must not be used by the program
z0	**Zero register** ▪ always zero (useful as immediate “0” value where registers are acceptable only)

Table 3: Internal registers

2.3 Overview of commands

The following tables briefly describe the different command types.

2.3.1 Standard ALU commands

Commands	Description	Page
add	add	29
nor	bitwise NOR	83
bs	bitwise smaller	44
inv	bitwise inversion	61
nimp	bitwise inverted implication	80
addus	add with unsigned saturation	33
xor	bitwise XOR	120
nand	bitwise NAND	78
and	bitwise AND	35
xnor	bitwise XNOR	102
addss	add with signed saturation	31
imp	bitwise implication	58
mov	move	75
bbe	bitwise bigger or equal	39
or	bitwise OR	85
addc	add with carry	30
sub	subtract	96
subus	subtract with unsigned saturation	100
subss	subtract with signed saturation	98
subc	subtract with carry	97
lsl	logical shift left	65
lsr	logical shift right	67
asr	arithmetic shift right	37
rol	rotate left	91
bcd2hex	binary coded decimal to hexadecimal	41
hex2bcd	hexadecimal to binary coded decimal	56
maxu	maximum unsigned	70
maxs	maximum signed	69
minu	minimum unsigned	74
mins	minimum signed	73
clz	count leading zeros	50
clo	count leading ones	48
mean	arithmetic mean	71
clmsb	count leading most significant bit	46

Table 4: Standard ALU commands

2.3.2 Load/store commands

Commands	Description	Page
load	load	64
store	store	95

Table 5: Load/store commands

2.3.3 S7 commands

Note: S7 commands are not available on netX 10.

Instruction	Description	Page
s7_a	AND	102
s7_an	AND-NOT	104
s7_o	OR	110
s7_on	OR-NOT	112
s7_x	EXCLUSIVE-OR	114
s7_xn	EXCLUSIVE-OR-NOT	116
s7_fp	EDGE-POSITIVE	108
s7_fn	EDGE-NEGATIVE	106

Table 6: S7 commands

2.3.4 Increment/decrement commands

For increment/decrement commands, the xPIC interprets source 2 as a small constant from 0 - 15.

Commands	Description	Page
inc	increment	60
dec	decrement	52

Table 7: Increment/decrement commands

2.3.5 Shift and rotate commands with small constant 0 to 31

Commands	Description	Page
lsl	logical shift left	65
lsr	logical shift right	67
asr	arithmetic shift right	37
rol	rotate left	91

Table 8: Shift and rotate commands with small constant 0 to 31

2.3.6 Multiplication commands

The source operands, source 1 and 2, must be registers for xPIC multiplication. The target register selects the type of multiplication (32-bit or 64-bit). For r0 to r7, u0 to u4 and st register, the xPIC executes a 32-bit multiplication with 32-bit target and 32-bit-conform condition flags. For target register usr01 (coded like the z0 register in other commands) and u32 (coded like the pc register in other commands) the xPIC executes a 64-bit multiplication with 64-bit target and 64-bit-conform condition flags.

Commands	Description	Page
muls	multiply signed	76
mulu	multiply unsigned	77

Table 9: Multiplication commands

2.3.7 Jump commands

Commands	Description	Page
jmp	jump	62
jmpdec	jump and decrement	63

Table 10: Jump commands

2.3.8 Special commands

Commands	Description	Page
nop	no operation	82
break	software breakpoint	43
retf	return from fast interrupt	89
reti	return from interrupt	90
reli	release interrupt	87
relreti	release and return from interrupt	88
sie	set interrupt enable flags	93
gie	get interrupt enable flags	53
gsie	get and set interrupt enable flags	54

Table 11: Special commands

2.3.9 Command-internal functions

swap

The xPIC can swap the bytes (and words) within a 32-bit or 16-bit load or store instruction.

swap	description
no swap	-
byte swap	dword access: [b1][b2][b3][b4] -> [b4][b3][b2][b1] word access: [b1][b2] -> [b2][b1]
word swap	dword access: [w1][w2] -> [w2][w1]

Table 12: swap

mas (memory access size)

The CPU supports 8-, 16- and 32-bit memory accesses. The mas code indicates the respective bit count.

mas	bit count
byte access	8
word access	16
dword access	32

Table 13: mas

sign (sign extension)

The CPU can process data with and without sign.

sign extension	description
no	No sign extension
yes	With sign extension

Table 14: sign extension

inc/dec (postincrement or predecrement)

Incrementation/decrementation takes place before or after command execution or not at all.

inc/dec	description
no change	No increment, no decrement
postincrement	The used working register is incremented after the operation is excecuted
predecrement	The used working register is decremented before the operation is excecuted.

Table 15: inc/dec

2.4 Overview of condition flags

Condition flags	Description
always	always
z	Zero of last ALU command
c	Carry of last ALU command
s	Sign of last ALU command
v	Overflow of last ALU command
e	Equal (src1 == src2)
gu	Greater Than Unsigned (src1 > src2)
gs	Greater Than Signed (src1 > src2)
geu	Greater or Equal Unsigned (src1 >= src2)
ges	Greater or Equal Signed (src1 >= src2)
r0z	Work register 0 zero (r0 == 0)
r1z	Work register 1 zero (r1 == 0)
r2z	Work register 2 zero (r2 == 0)
r3z	Work register 3 zero (r3 == 0)
r0123z	Work register 0 to 3 zero (r0 == 0 && r1 == 0 && r2 == 0 && r3 == 0)
never	Never
nz	Not Zero of last ALU command
nc	Not Carry of last ALU command
ns	Not Sign of last ALU command
nv	Not Overflow of last ALU command
ne	Not Equal (src1 != src2)
leu	Less or Equal Unsigned (src1 <= src2)
les	Less or Equal Signed (src1 <= src2)
lu	Less Than Unsigned (src1 < src2)
ls	Less Than Signed (src1 < src2)
r0nz	Work register 0 not zero (r0 != 0)
r1nz	Work register 1 not zero (r1 != 0)
r2nz	Work register 2 not zero (r2 != 0)
r3nz	Work register 3 not zero (r3 != 0)
r0123nz	Work register 0 to 3 not zero (r0 != 0 || r1 != 0 || r2 != 0 || r3 != 0)

Table 16: Overview of conditions flags

2.5 Address mapping

2.5.1 Local RAM

The xPIC has two local internal RAM segments. It can always access them via zero wait states.

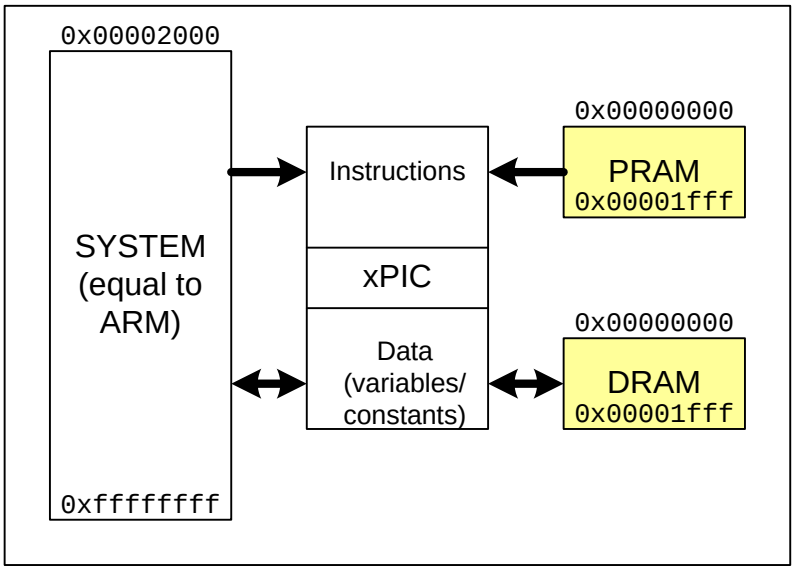


Figure 2: local xPIC RAM segments

PRAM 8 Kbyte program memory, up to 2048 instructions (1 instruction = 4 byte)

DRAM 8 Kbyte data memory

Both segments share the same address area 0x0 up to and including 0x1fff. The instructions are fetched from the PRAM. The data is accessed on the DRAM.

The local PRAM is mapped to the above-mentioned address area for the xPIC only, the same address area as the DRAM (for data access). Any other AHB master (e.g. ARM) accesses this RAM via the preload window (address: xpic_pram_start). AHB accesses via the preload window are possible only if the xPIC is in HOLD. The xPIC cannot access the PRAM within this preload window.

The local DRAM is mapped to the above-mentioned address area for the xPIC only, the same address area as the PRAM (for instruction access). Any other AHB master (e.g. ARM) accesses this RAM via the preload window (address: xpic_dram_start). The DRAM should NOT be used for data exchange between xPIC and any other AHB master during runtime in order to avoid undeterministic wait states blocking the AHB channel. The xPIC can also access the DRAM within this preload window, **but this is very slow**.

The xPIC can also access the netX registers, the INTRAM or any extension memory for instruction and data access. There are two separate AHBL channels.

2.5.2 Periphery

The xPIC has access to the netX periphery.

2.6 Pipeline and bypassing

The xPIC pipeline has two steps:

- Step 1: Instruction fetch and decoding
- Step 2: Instruction execution

Jump commands (**jmp**, **jmpdec**) and interrupt control commands (**gie**, **sie**, **gsie**) are always executed during step 1.

The xPIC never flushes the pipeline. This is very important for direct and indirect branching. Direct branching is accomplished with a jump command fetched and executed during pipeline step 1. The next instruction fetched is the jump target which makes a pipeline flush unnecessary. ALU commands and load commands with the program counter as target register are called indirect branches. The program counter is written in step 2 after the instruction execution. In step 2 the succeeding instruction has already been fetched and will be executed in the next operation.

Example

```
add r6, #4, pc            // return address
load pc, [r1]             // function call with indirect jump
store [z0 + --r7], r6     // push return address to stack before function
jmp r2nz, failed          // check function result after return from function
```

Bypassing, also known as forwarding, is used to avoid data hazards in the pipeline. For bypassing, instructions have to be executed which help transport available data and results from pipeline stages located further back to the front, i.e. to the point before the hazard can occur.

Since data cannot be bypassed back to an earlier stage, the solution is to delay the AND instruction. While a nop instruction inserts a bubble (which occupies resources without producing any useful results), the data is forwarded by one cycle thereby preventing the occurrence of the data hazard.

Example

```
add r0, r1, r2
and r7, r0, r4
```

2.6.1 Data dependencies

To avoid data hazards, the core has some bypasses which allow reading registers written in the previous step. However, not all combinations are possible. The compiler should return an error if the code contains such hazardous combinations. In this case an additional command (e.g. 'nop') must be inserted between the two commands causing the hazard.

Non-bypassed data dependencies

A nop command is necessary for the following command sequences:

- arithmetic logical command, indirect addressing

```
add  r0, r1, r2
nop
load r3, [r0]
```

- load command, indirect addressing

```
mov  r0, r1, z0
nop
load r3, [r0]
```

- arithmetic logical commands, conditional arithmetic logical command

```
sub  r0, r1, r2
nop
[nz] add r0, r1, r2
```

- multiplication with 64-bit target

```
mulu u10, r1, r2
nop                  // mulu is still computing, u1 and u0 are not yet ready
add r0, u1, u0
```

Bypassed data dependencies

A hardware bypass exists for the following data dependencies. A nop command is not necessary in these command sequences.

- arithmetic logical command, arithmetic logical command

```
add r0, r1, r2
add r7, r0, r4
```

- load command, arithmetic logical command

```
load r0, #0xff
add  r7, r0, r4
```

- arithmetic logical command, store command

```
add r0, r1, r2
store [r1 + r2], r0
```

- arithmetic logical command, ALU flag conditional jump

```
loop: sub r0, r1, r2
jmp nz, loop
```

Note: Register flags (r0z, r0nz, etc.) are never bypassed, not even for jump commands!

```
loop: sub r0, r1, r2
nop
jmp r0z, loop
```

2.6.2 Conditional jumps

The conditions (flags) are bypassed so that the conditional jump may directly follow the instruction which sets the condition flag. Exceptions are the work register flags (r0z..r0123z, r0nz..r0123nz). These flags are stable only after the next-but-one instruction.

Examples

jump after setting a flag:

```
sub r0, r1, r1
jmp z, label
```

same effect, but slower:

```
sub r0, r1, r1
nop
jmp r0z, label
```

2.6.3 Conditional execution

The flags of the conditional execution of arithmetic logical or load/store commands are not bypassed.

Examples

```
sub r0, r1, r1      // set the zero flag
nop                 // insert nop or other command before the conditional execution
[z] add, r2, r3, r4 // conditional execution
```

Note: The carry flag is bypassed for the following instruction, but not the carry flag for a conditional execution.

```
line_0: add u0, u1, u2
line_1: add r0, r1, r2
line_2: [c] addc r3, r4, r5      // condition [c] comes from line_0 but
                                 // carry operand is from line_1
```

2.7 Interrupt handling

Two interrupt types are available: Normal Interrupt Request (IRQ) and Fast Interrupt Request (FIQ). With IRQs, the xPIC has to perform a fixed set of operations to save the current processor state and to restore this state after the current interrupt service routine.

With FIQs, the xPIC uses a second register bank while the previous processor state is frozen until the interrupt service routine is finished. This reduces the overhead for saving and restoring the normal program flow.

The xPIC VIC (Vectored Interrupt Controller) controls the interrupts. 16 IRQ vectors and 1 FIQ vector can be configured. Each vector assigns a system-specific interrupt source to an interrupt service routine.

Controlling the interrupt enable flags (ief)

Note: The **gie, sie**, **gsie** commands are not available on netX 10.

Two interrupt enable flags can be read and written using the special instructions **gie**, **sie**, **gsie** in order to control the interrupt processing. Bit 0 of the 2-bit register enables the IRQs, bit 1 enables the FIQs.

2.7.1 Interrupt controller (xPIC VIC)

The VIC contains 17 interrupt vectors (1 FIQ, 15 IRQs, 1 default IRQ). 64 IRQ sources are selectable. All sources can be triggered manually for software interrupt and debug. The vector table must be stored in the memory.

VIC Initialization, Reference Implementation

```
// Include register definitions
//include regdef.ass

// Example settings

// for jump table with near jumps:
$MSK_xpic_vic_config = $MSK_xpic_vic_config_enable

// for jump table with far jumps:
$MSK_xpic_vic_config = $MSK_xpic_vic_config_enable | $MSK_xpic_vic_config_table


////////////////////////////////////////////////////////////////////////
// Configurate xPIC VIC registers

     // disable xPIC VIC
     load r0, $Addr_xpic_vic
     nop
     store [r0 + $REL_Adr_xpic_vic_config], z0

     // set FIQ handler address
     load r1, $fiq_handler
     store [r0 + $REL_Adr_xpic_vic_fiq_addr], r1
     // set IRQ handler address
     load r1, $irq_handler
     store [r0 + $REL_Adr_xpic_vic_irq_addr], r1
     // set ISR jump table address
     load r1, $isr_table
     store [r0 + $REL_Adr_xpic_vic_table_base_addr], r1

     // config FIQ vector
     load r1, #h000000XX         // enable bit (h40) | interrupt source 0-63 (bxxxxxx)
     store [r0 + $REL_Adr_xpic_vic_fiq_vect_config], r1

     // config IRQ 0 vector
     load r1, #h000000YY         // enable bit (h40) | interrupt source 0-63 (byyyyyy)
     store [r0 + $REL_Adr_xpic_vic_vect_config0], r1

     // config IRQ 1 vector
     load r1, #h000000ZZ         // enable bit (h40) | interrupt source 0-63 (bzzzzzz)
     store [r0 + $REL_Adr_xpic_vic_vect_config1], r1

     // config other IRQ vectors

     // enable xPIC VIC
     load r1, MSK_xpic_vic_config
     store [r0 + $REL_Adr_xpic_vic_config], r1
```

2.7.2 Interrupt request (IRQ) handling

In case of an IRQ, we assume the processor has 2 instructions in the pipeline: Instruction A in the execution phase and instruction B in the fetch/decode phase. The next instruction without interrupt would be C. The IRQs are blocked for one cycle after writing **pc** or **st**, so B is any instruction that does not use **pc** or **st** as target. The processor performs the following 3 steps automatically. Each step can be tracked in single-step mode:

1. Finish the execution of A. Fetch/decode B. Set next **pc** to **pc** += 4 (address of C).
2. Execute B. Save extended processor state in **st**. Latch pc (address of C) for next cycle. Set next **pc** to the address of IRQ handler (configured in VIC).
3. Fetch first instruction of IRQ handler. Continue normally.

In step 2, the **pc** value is latched for one cycle, so a read access gives the last **pc** before the execution of the IRQ jump. The extended processor status is stored in the bits[25:7] of the **st** register. There is no register bank switch for IRQs. This behavior prescribes two actions for the IRQ handler:

1. To be able to resume the interrupted program at the correct location, the program counter (**pc**) must be saved (most likely on the stack) in the very first instruction of the IRQ handler.
2. To be able to restore the processor status when the interrupted program is resumed, the status register (**st**) must be saved (most likely on the stack) before any ALU command (preferably in instruction 2 of the IRQ handler).

The last three instructions of the IRQ handler are prescribed as follows:

1. Write the saved extended register status back to **st**
2. Write the saved return address into **pc**
3. To leave the IRQ handler and resume the interrupted program, call **relreti** (or **reti** if **reli** was called within the IRQ handler)

The command **relreti** restores the processor status and allows new IRQs.

With the command **reli**, the IRQ handler allows the handling of any pending IRQ. In this case, the IRQ handler starts before the return of the previous IRQ handler. Such handling is called **nested**. It may improve the performance in terms of IRQ response times, but it includes the danger of stack overflows. To finalize a nested IRQ handler, use **reti**.

Note: By using **st** and **pc** as target registers, the last three instructions cannot be interrupted by IRQs even if **reli** was called before.

The application program defines the IRQ handler settings.

In a usual multi-interrupt system, the IRQ handler would query the VIC for the address of the ISR and execute it. The following example can be regarded as a guideline for IRQ handler implementations.

In case of an IRQ, the rough operation sequence is as follows:

IRQ handler → get ISR address from VIC → jump into ISR → service request → jump back to IRQ handler → end interrupt

IRQ handler reference implementation

```
irq_handler: store [z0 + --r7], pc       // save pc on stack (pc is latched
                                         // within first cycle of irq handler)
             store [z0 + --r7], st       // save processor status on stack
             store [z0 + --r7], r0       // save r0 on stack
             load r0, [pc + $Adr_xpic_vic_vector_addr]  // address of ISR vector
             store [z0 + --r7], r1       // save r1 on stack
             add r0, [r0], z0            // get ISR address
             jmp e, irq_exit             // abort IRQ if ISR is undefined
             add r1, #4, pc              // compute return address for ISR

             load pc, [z0 + r0]          // call ISR (jump into ISR jump table)
             store [z0 + --r7], r1       // note: this instruction is executed
                                         // before ISR is called
irq_exit:    load r1, [z0 + r7++]        // restore r1
             load r0, [z0 + r7++]        // restore r0
             load st, [z0 + r7++]        // restore processor status
             load pc, [z0 + r7++]        // jump back to main program and ...
             relreti                     // ... allow next irq*

// (*) For nested IRQs, the last instruction of the irq_handler
// is reti instead of relreti.
// In this case, each ISR has to contain the reli command, whereupon
// other IRQs are possible.
```

The VIC has an IRQ latency of one clock cycle (10 ns). This is the period between receiving an external interrupt signal and triggering the IRQ for the xPIC.

The xPIC executes 2 instructions before fetching the first instruction of the IRQ handler. The worst case execution time depends on the operation and the AHB slave being accessed during these instructions, the min. execution time is 20 ns.

The above-mentioned reference IRQ handler executes 10 instructions before the ISR is completed. Assuming that the IRQ handler code resides in the PRAM, the stack in DRAM, and assuming that it takes 40 ns to read register XPIC_VIC_VECTOR_ADDR, these 10 instructions add another latency of 130 ns.

The overall IRQ latency becomes 140 ns plus two times the worst case AHB access time.

2.7.3 Fast Interrupt Request (FIQ)

Facts about FIQ handling:

- On FIQ, the pc jumps into FIQ handler
- Highest priority, FIQ suspends IRQ
- Switches register banks
- No need to save r0..r7, pc, st on stack
- u0 ... u4 must be saved on stack, if used by FIQ handler
- FIQ cannot be nested
- Return from FIQ handler with command retf
- On retf, the register bank is switched back (including the pc!)

2.8 Non-interruptable swap operation

A fast memory swap operation which is non-interruptable in user/irq mode can be implemented by using the IRQ blocking behavior when the status register (**st**) is written. This allows very fast mutex locking implementations.

A swap operation reads a value from memory location (i.e. the address of the mutex) and stores another value at the same location. The read and write operation is non-interruptable.

The xPIC lacks a dedicated swap instruction, but with a simple load instruction, with **st** as target register and a subsequent store instruction, the memory accesses cannot be interrupted by an IRQ since the write to **st** blocks IRQs for one instruction cycle. The loaded value can be evaluated by using a conditional jump or executions since the read value directly controls the processor flags in the status register.

The following instruction sequence implements a mutex lock operation for a single byte size mutex. The mutex address is given in r0. The mutex lock is successful if the swap reads a value of zero. A value of 1 means that the mutex is already locked and, moreover, sets the zero flag when loaded into **st**. If the lock operation fails, the conditional jump will take place.

```
add r1, #1, z0              // flag value is equal z (processor flag zero in st)
load st, [r0]b              // read flag and lock IRQ
store [r0]b, r1             // set flag
jmp z, mutex_lock_failed    // jump away if the flag was already set
```

Notes

1. This swap operation can still be interrupted by FIQs!
2. From netX 6/51/52, the IRQ disable bits allow the non-interruptibility!

2.9 xPIC timer

The xPIC timer module contains 3 timers with 3 independent interrupt sources. Each timer has one preload register and one timer register and can be configured in 3 different modes:

- Mode “2’b00” (config register): The programmed timer stops at zero and releases an interrupt.
- Mode “2’b01”: The time restarts with the preload value after reaching zero and releases an interrupt.
- Mode “2’b10”: The programmed value is compared with the systime nanosecond and releases an interrupt after reaching the systime.

It is also possible to trigger the motion encoder unit with each timer in each mode.

A systime seconds compare function in the xPIC timer unit can also generate an interrupt.

2.10 xPIC watchdog

As an external system observer, the XPIC_WDG (xPIC watchdog) unit signals hardware or software failures. Once activated, the watchdog must be “fed” by the xPIC program to keep it “quiet”. If the watchdog is not “fed” within a defined time, the system is assumed to have failed.

A failure can be reported by generating:

- an interrupt for the xPIC
- an interrupt for the ARM

2.11 Debug unit

The debug unit starts and stops the xPIC core and enables a hardware reset. For debugging there are two hardware breakpoints which can create an interrupt to the host CPU. Software breakpoints and single-step mode are set and reset in the debug unit. The status of xPIC break and break reasons can also be polled.

To determine the break status, read register **xpic_debug_irq_raw**. To release the break/IRQ, write to register **xpic_debug_irq_raw**.

2.11.1 Reset

The **reset_xpic** bit of register **xpic_hold_pc** tidily resets the xPIC status and all internal xPIC states without affecting the xPIC registers or the xPIC pc. To reset both register banks (normal bank and FIQ bank), the **bank_control** bit requires setting. To select the bank, use the **bank_select** bit. To return the control to the xPIC, write zero into the **bank_control** bit.

2.11.2 Misaligned access

The xPIC may sometimes execute a misaligned data access, e.g. if an instruction performs an indirect memory access with 16 or 32-bit memory access size and the register value (calculated address) is not 16 or 32-bit aligned. There are three ways to report a misaligned memory access:

- the xPIC is held on misaligned access
- a xPIC IRQ is generated
- an ARM IRQ is generated

All options can be combined. Break delay is 1 instruction. Rewinding the program counter to continue running is not necessary.

2.11.3 Hardware breakpoints/watchpoints

Two hardware watchpoints are available. The debug unit breaks the xPIC if a specified memory address is accessed with a specified data value. Two different modes are available: Instruction fetch and data access. The debug unit can detect IRQ and FIQ mode. In case of data access mode the controller distinguishes between read and write and the memory access size (byte, word and dword). Watchpoints can be chained to test conditions of both watchpoints at the same time.

Timing

Type of breakpoint	Break delay (number of instructions fetched after breakpoint match)
instruction fetch	0
data access (write)	1
data access (read)	2

2.11.4 Software breakpoints

With the special instruction **break** it is possible to implement unlimited software breakpoints. The instruction **break** can be inserted anywhere in a program, but it must not be inserted directly after a load xpic_pc instruction. This instruction only halts the xPIC in the next instruction (break delay is 1). To continue the original program flow, three steps must be performed:

- Exchange the break command in the program memory with the original instruction
- Rewind the pc for two instructions: pc := pc - 8
- Release the xPIC break by writing SOFT_BREAK_IRQ (bit 2) into register **xpic_debug_irq_raw**.

2.11.5 Single-step mode

If started in single-step mode, the xPIC will break at every single instruction. Break delay is 0, i.e the pc must not be rewound before the program continues. The break is released by writing SINGLE_STEP_IRQ (bit 3) into register **xpic_debug_irq_raw**.

3 Instruction set

3.1 Using large values

Each instruction used with a constant/immediate value has two value ranges, see *Figure 3*. In range 1 (green segment), the value can be used directly. In range 2 (orange segment), the value can only be used indirectly.

Example: ***load reg, const***. The total range of values is $\boldsymbol{-2^{18} \ldots 2^{18} - 1}$. That means a register can be loaded directly with values within

0 (0x0) … 262143 (0x3ffff)

and

-262144 / 4294705152 (0xfffc0000) ... -1 / 4294967295 (0xffffffff).

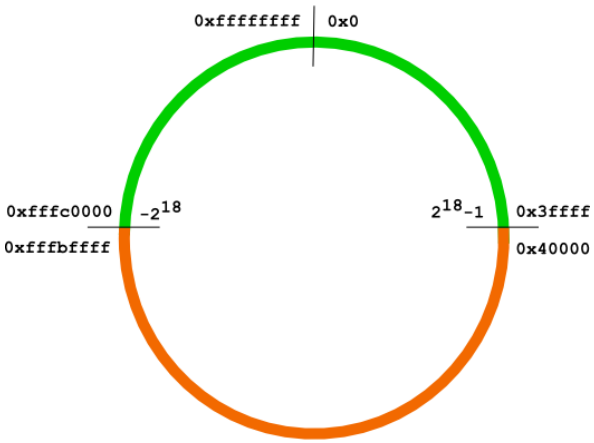


Figure 3: Value ranges 1 and 2

For all other values there are 2 indirect ways of loading a value which is not within this range:

1) using arithmetic instructions

```
load r0, #0x1ffff    // r0 := 0x0001ffff
lsl  r0, r0, #14     // r0 := 0x7fffc000
add  r0, #0x3fff, r0 // r0 := 0x7fffffff
```

2) using data segment

```
.data
Data.value:
  .long 0x7fffffff
.align 2
.text
load r0, [pc + $Data.value]d
```

3.2 Reference

3.2.1 add - add

Integer addition with target (c), source 1 (a) and source 2 (b).

Operation

c := a + b

Syntax

add reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

add reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

add reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **add** reg, reg, reg

{[cond]} **add** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **add** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	+1	+1	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Add the value of work register 0 to that of user register 1. Save the result in work register 4:

```
load r0, #5             // r0 := 5
load u1, #15            // u1 := 15
add  r4, r0, u1         // r4 := r0 + u1 = 5 + 15 = 20
```

Add the sign extended byte value of address 0x1000 to the value of work register 0. Save the result in work register 6:

```
load r1, #0x1000
nop
add r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: add u0, u1, u2
line_1: add r0, r1, r2
line_2: [z] add r3, r4, r0 // condition [z] comes from line_0 but r0 from line_1
```

3.2.2 addc - add with carry

Integer addition with target (c), source 1 (a), source 2 (b) and the carry flag.

Operation

c := a + b + carry

Syntax

addc reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

addc reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

addc reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **addc** reg, reg, reg

{[cond]} **addc** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **addc** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	+1	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Add the value of work register 0 to the constant value 0x2000 and the carry flag. Save the result in work register 4:

```
load r0, #0x80000000 // r0 := 0x80000000 *
add z0, r0, r0       // carry := (r0 + r0) >> 32 = 1
addc r4, #0x2000, r0 // r4 := 0x2000 + r0 + carry = 0x80002001
```

* see section *3.1 Using large values* on page 28 for handling large values.

Add the 64-bit value of user register 3/2 to that of user register 1/0. Save the result in user register 1/0.

```
add u0, u0, u2            // u0 := u0 + u2
addc u1, u3, u1           // u1 := u1 + u3 + carry
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: add u0, u1, u2
line_1: add r0, r1, r2
line_2: [c] addc r3, r4, r5 // condition [c] comes from line_0 but carry operand
                            // is from line_1
```

3.2.3 addss - add with signed saturation

Signed integer addition with saturation. Source 1 (a) and source 2 (b) are interpreted as signed integers. The target (c) is set to 0x80000000 if there is an underflow or to 0x7fffffff if there is an overflow in the addition of source 1 and 2.

Operation

c := ssat(signed(a) + signed(b))

```
          | -0x80000000;   (x < -0x80000000)
ssat(x) = |  0x7fffffff;   (x >  0x7fffffff)
          |  x;            else
```

Syntax

addss reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

addss reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

addss reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **addss** reg, reg, reg

{[cond]} **addss** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **addss** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

The flags are computed before saturation. Overflow flag (v) is an indicator for saturation: If v is set after the instruction, the result is saturated.

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	+1	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Add the signed value of work register 0 to the signed value of user register 1. Save the result in work register 4 after saturation:

```
load r0, #0x7ffffffd // r0 := 2147483645 (0x7ffffffd) *
load u1, #5          // u1 := 5
addss r4, r0, u1     // r4 := ssat(r0 + u1) = 2147483647 (max int)
```

* see section *3.1 Using large values* on page 28 for handling large values.

Add the sign extended and signed byte value of address 0x1000 to the signed value of work register 0. Save the result in work register 6 after saturation:

```
load r1, #0x1000
nop
addss r6, s[r1]b,r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: add u0, u1, u2
line_1: add r5, r1, r2
line_2: [z] addss r3, r4, r5  // condition [z] comes from line_0 but r5 from
                              // line_1
```

3.2.4 addus - add with unsigned saturation

Unsigned integer addition with saturation. Source 1 (a) and source 2 (b) are interpreted as unsigned. The target is set to 0xffffffff if there is an overflow in the addition of source 1 and 2.

Operation

c := usat(a + b)

usat(x) = | 0xffffffff; (x > 0xffffffff)
| x; else

Syntax

addus reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

addus reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

addus reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **addus** reg, reg, reg

{[cond]} **addus** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **addus** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

The flags are computed before saturation. The overflow flag (v) is an indicator for the saturation: If v is set after the instruction the result has been saturated.

set for	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
conditional	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	+1	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Add the unsigned value of work register 0 to the unsigned value of user register 1. Save the result in work register 4 after saturation:

```
load r0, #0xfffffffd // r0 := 4294967293 (0xfffffffd) *
load u1, #5              // u1 := 5
addus r4, r0, u1         // r4 := usat(r0 + u1) = 4294967295 (max uint)
```

* see section *3.1 Using large values* on page 28 for handling large values.

Add the unsigned dword value of address 0x1000 to the unsigned value of work register 0. Save the result in work register 6 after saturation:

```
load r1, #0x1000
nop
addus r6, [r1]d, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: add u0, u1, u2
line_1: add r5, r1, r2
line_2: [z] addus r3, r4, r5   // condition [z] comes from line_0
                               // but r5 from line_1
```

3.2.5 and - bitwise AND

Bitwise AND operation with target (c), source 1 (a) and source 2 (b).

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

Operation

c := a & b

Syntax

and reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

and reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

and reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **and** reg, reg, reg

{[cond]} **and** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **and** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7,u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Perform a bitwise AND operation with the value of work register 0 and that of user register 1. Save the result in work register 4:

```
and r4, r0, u1       // r4 := r0 & u1
```

Perform a bitwise AND operation with the sign extended byte value of address 0x1000 and the value of work register 0. Save the result in work register 6:

```
load r1, #0x1000
nop
and r6, s[r1]b,r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: and u0, u1, u2
line_1: and r5, r1, r2
line_2: [z] and r3, r4, r5 // condition [z] comes from line_0 but r5 from line_1
```

3.2.6 asr - arithmetic shift right

Arithmetic shift right of source 1 (a) with source 2 (b). The sign of source 1 is preserved for target (c).

If source 2 is a register, an additional bit mask operation can be applied to the result of the shift operation. The mask operation is either NAND, AND or OR and is defined by b[6:5]. The operand (bit mask) is constructed from b[31:7].

Operation

```
c := a
for (count := 1; count <= b[4:0]; count++)
    c := c >> 1
    c[31] := c[30]
endfor
// additional mask operation
case b[6:5]
    00: c = c
    01: c = c & ~(b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[31:8])
    10: c = c &  (b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[31:8])
    11: c = c |  (b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[31:8])
endcase
```

Syntax

asr reg, [pc + const], reg* // const = -2^{16} … 2^{16}-4 (dword aligned)

asr reg, [const], reg* // const = 0 … 2^{17}-4 (dword aligned)

asr reg, const, reg* // const = -2^{14} … 2^{14}-1

{[cond]} **asr** reg, reg, reg*

{[cond]} **asr** reg, {s}[pc + wreg{++}]{b}, reg*

{[cond]} **asr** reg, {s}[wreg{++}]{b}, reg*

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

*reg** *1) reg[4:0] = shift value; reg[6:5] = mode; reg[7] = mask[31 … 24]; reg[31:8] = mask[23:0]*

or

2.) instead of register use constant: 0 ... 31

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

The signed value of work register 0 is arithmetic shifted right by 6. Save the result in work register 4:

```
load r0, #0xfffff000      // r0 := -4096
asr r4, r0, #6            // r4 := -4096 / 2^6 = -64 (0xffffffC0)
```

The sign extended value of address 0x1000 is arithmetic shifted right by r0 (count 5). Save the result in work register 6:

```
load r1, #0x1000
load r0, #5
asr r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: asr u0, u1, u2
line_1: asr r5, r1, r2
line_2: [z] asr r3, r4, r5      // condition [z] comes from line_0
                                // but r5 from line_1
```

3.2.7 bbe - bitwise bigger or equal

Bitwise bigger or equal operation with target (c), source 1 (a) and source 2 (b).

a	b	c
0	0	1
0	1	0
1	0	1
1	1	1

Operation

c := a | ~b

Syntax

bbe reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

bbe reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

bbe reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **bbe** reg, reg, reg

{[cond]} **bbe** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **bbe** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
conditional	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Perform a bitwise bigger or equal operation with the value of work register 0 (source 1) and that of user register 1 (source 2). Save the result in work register 4:

```
bbe r4, r0, u1                // r4 := r0 | ~u1
```

Perform a bitwise bigger or equal operation with the sign extended byte value (source 1) of address 0x1000 and the value of work register 0 (source 2). Save the result in work register 6:

```
load r1, #0x1000
nop
bbe r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: bbe u0, u1, u2
line_1: bbe r5, r1, r2
line_2: [z] bbe r3, r4, r5      // condition [z] comes from line_0
                                // but r5 from line_1
```

3.2.8 bcd2hex – binary coded decimal to hexadecimal

Note: The **bcd2hex**-command is not available on netX 10.

Convert the 3 binary coded decimal digits (a 4 bit) in source 1 (a) to hexadecimal target. Source 2 (b) will be ignored. The result (c) is from 0x0 to 0x3e7. If one of the 3 source 1 (a) digits is greater than 0x9, the overflow flag (v) will be set and the result (c) will be undefined. The upper bits of source 1 (a[31:12]) will be ignored.

Operation

c := hex(bcd[a])

Syntax

bcd2hex reg, [pc + const], reg* // const = -2^{16} … 2^{16}-4 (dword aligned)

bcd2hex reg, [const], reg* // const = 0 … 2^{17}-4 (dword aligned)

bcd2hex reg, const, reg* // const = -2^{14} … 2^{14}-

{[cond]} **bcd2hex** reg, reg, reg*

{[cond]} **bcd2hex** reg, {s}[pc + wreg{++}]{b}, reg*

{[cond]} **bcd2hex** reg, {s}[wreg{++}]{b}, reg*

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

*reg** *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command) source 2 is ignored in this operation, use z0*

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	-	-	-	+1	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	+2	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Convert work register 0 from bcd to hex. Save the result in work register 4:

```
load r0, #0x123
bcd2hex r4, r0, z0   // r4 := 0x7b
load r0, #0x01000999 *
bcd2hex r4, r0, z0   // r4 := 0x3e7 (upper bits ignored)
load r0, #0x2f2
bcd2hex r4, r0, z0   // r4 := undefined (v flag set)
```

* see section *3.1 Using large values* on page 28 for handling large values.

Convert the byte value of address 0x1000. Save the result in work register 6:

```
load r1, #0x1000
nop
bcd2hex r6, [r1]b, z0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: bcd2hex u0, u1, z0
line_1: bcd2hex r4, r1, z0
line_2: [v] bcd2hex r3, r4, z0  // condition [v] comes from line_0
                                // but r4 from line_1
```

3.2.9 break - software breakpoint

Stops the xPIC for debugging (software breakpoint with interrupt for Host CPU).

Operation

-

Syntax

break

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	-	-	-	-	-
execution	-	-	-	-	-	-	-	-	-	-	-	-	-	-

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Halt the xPIC in line_1:

```
line_0: sub r0, r1, r2
line_1: break              // xPIC halts here
line_2: sub r2, r3, r4
```

For further details and software breakpoint handling, see chapter *Debug unit* on page 26.

3.2.10 bs - bitwise smaller

Bitwise smaller operation with target (c), source 1 (a) and source 2 (b).

a	b	c
0	0	0
0	1	1
1	0	0
1	1	0

Operation

c := ~a & b

Syntax

bs reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

bs reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

bs reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **bs** reg, reg, reg

{[cond]} **bs** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **bs** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Perform a bitwise smaller operation with the value of work register 0 (source 1) and that of user register 1 (source 2). Save the result in work register 4:

```
bs r4, r0, u1              // r4 := ~r0 & u1
```

Perform a bitwise smaller operation with the sign extended byte value (source 1) of address 0x1000 and the value of work register 0 (source 2). Save the result in work register 6:

```
load r1, #0x1000
nop
bs r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: bs u0, u1, u2
line_1: bs r5, r1, r2
line_2: [z] bs r3, r4, r5    // condition [z] comes from line_0
                             // but r5 from line_1
```

3.2.11 clmsb - count leading most significant bit

Count the leading most significant bit(s) of source 1 (a). Source 2 is ignored. The result (c) is from 1 to 32.

Operation

```
c := 32
for (count := 1; count < 32; count++)
    if (a[(31 - count)] != a[31])
        c := count
        break
endfor
```

Syntax

clmsb reg, [pc + const], reg* // const = -2^{16} … 2^{16}-4 (dword aligned)

clmsb reg, [const], reg* // const = 0 … 2^{17}-4 (dword aligned)

clmsb reg, const, reg* // const = -2^{14} … 2^{14}-1

{[cond]} **clmsb** reg, reg, reg*

{[cond]} **clmsb** reg, {s}[pc + wreg{++}]{b}, reg*

{[cond]} **clmsb** reg, {s}[wreg{++}]{b}, reg*

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

*reg** *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command) source 2 is ignored in this operation, use z0*

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Count the leading most significant bit(s) of work register 0. Save the result in work register 4:

```
load r0, #0xff000000 *
clmsb r4, r0, z0     // r4 := 8 (r0 has eight leading ones)
load r0, #0x00800000 *
clmsb r4, r0, z0     // r4 := 8 (r0 has eight leading zeros)
load r0, #0x7f000000 *
clmsb r4, r0, z0     // r4 := 1 (r0 has one leading zero)
```

* see section *3.1 Using large values* on page 28 for handling large values.

Count the leading most significant bit(s) of the sign extended byte value of address 0x1000. Save the result in work register 6:

```
load r1, #0x1000
nop
clmsb r6, s[r1]b, z0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: clmsb u0, u1, z0
line_1: clmsb r4, r1, z0
line_2: [z] clmsb r3, r4, z0    // condition [z] comes from line_0
                                   // but r4 from line_1
```

3.2.12 clo - count leading ones

Count the leading bits set (‘ones’) of source 1 (a). Source 2 is ignored. The result (c) is from 0 to 32.

Operation

```
c := 32
for (count := 0; count < 32; count++)
    if (a[(31 - count)] != 1)
        c := count
        break
endfor
```

Syntax

clo reg, [pc + const], reg* // const = -2^{16} … 2^{16}-4 (dword aligned)

clo reg, [const], reg* // const = 0 … 2^{17}-4 (dword aligned)

clo reg, const, reg* // const = -2^{14} … 2^{14}-1

{[cond]} **clo** reg, reg, reg*

{[cond]} **clo** reg, {s}[pc + wreg{++}]{b}, reg*

{[cond]} **clo** reg, {s}[wreg{++}]{b}, reg*

{[cond]} conditional execution with the flags (not bypassed, i.e. from the next-to-last command)

{s} sign extension

{++} post-increment or pre-decrement (--wreg)

{b} b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

wreg r0 … r7 (not bypassed, i.e. from the next-to-last command)

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command) source 2 is ignored in this operation, use z0*

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Count the leading ones of work register 0. Save the result in work register 4:

```
load r0, #0xff000000 *
clo r4, r0, z0  // r4 := 8 (r0 has eight leading ones)
load r0, #0x00800000 *
clo r4, r0, z0  // r4 := 0 (r0 has no leading ones)
load r0, #0x7f000000 *
clo r4, r0, z0  // r4 := 0 (r0 has no leading one)
```

* see section *3.1 Using large values* on page 28 for handling large values.

Count the leading ones of the sign extended byte value of address 0x1000. Save the result in work register 6:

```
load r1, #0x1000
nop
clo r6, s[r1]b, z0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: clo u0, u1, z0
line_1: clo r4, r1, z0
line_2: [z] clo r3, r4, z0          // condition [z] comes from line_0
                                    // but r4 from line_1
```

3.2.13 clz - count leading zeros

Count the leading zeros of source 1 (a). Source 2 is ignored. The result (c) is from 0 to 32.

Operation

```
c := 32
for (count := 0; count < 32; count++)
    if (a[(31 - count)] != 0)
        c := count
        break
endfor
```

Syntax

clz reg, [pc + const], reg* // const = -2^{16} … 2^{16}-4 (dword aligned)

clz reg, [const], reg* // const = 0 … 2^{17}-4 (dword aligned)

clz reg, const, reg* // const = -2^{14} … 2^{14}-1

{[cond]} **clz** reg, reg, reg*

{[cond]} **clz** reg, {s}[pc + wreg{++}]{b}, reg*

{[cond]} **clz** reg, {s}[wreg{++}]{b}, reg*

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

*reg** *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command) source 2 is ignored in this operation, use z0*

Flags

set for conditional	**z**	**c**	**s**	**v**	**e**	**gu**	**gs**	**geu**	**ges**	**r0z**	**r1z**	**r2z**	**r3z**	**r0123z**
	nz	**nc**	**ns**	**nv**	**ne**	**leu**	**les**	**lu**	**ls**	**r0nz**	**r1nz**	**r2nz**	**r3nz**	**r0123nz**
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Count the leading zeros of work register 0. Save the result in work register 4:

```
load r0, #0xff000000 *
clz r4, r0, z0  // r4 := 0 (r0 has no leading zeros)
load r0, #0x00800000 *
clz r4, r0, z0  // r4 := 8 (r0 has eight leading zeros)
load r0, #0x7f000000 *
clz r4, r0, z0  // r4 := 1 (r0 has one leading zero)
```

* see section *3.1 Using large values* on page 28 for handling large values.

Count the leading zeros of the sign extended byte value of address 0x1000. Save the result in work register 6:

```
load r1, #0x1000
nop
clz r6, s[r1]b, z0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: clz u0, u1, z0
line_1: clz r4, r1, z0
line_2: [z] clz r3, r4, z0  // condition [z] comes from line_0
                            // but r4 from line_1
```

3.2.14 dec - decrement

Decrement source 1 (a) by a constant (b) (0 to 15).

Operation

c := a – b

Syntax

dec reg, [pc + const], #0…15 // const = -2^{16} … 2^{16}-4 (dword aligned)

dec reg, [const], #0…15 // const = 0 … 2^{17}-4 (dword aligned)

dec reg, const, #0…15 // const = -2^{14} … 2^{14}-1

{[cond]} **dec** reg, reg, #0…15

{[cond]} **dec** reg, {s}[pc + wreg{++}]{b}, #0…15

{[cond]} **dec** reg, {s}[wreg{++}]{b}, #0…15

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	+1	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Decrement the value of work register 0 by two. Save the result in work register 4:

```
load r0, #256                // r0 := 256
dec r4, r0, #2               // r4 := 256 - 2 = 254
```

Decrement the sign extended byte value of address 0x1000 by one. Save the result in work register 6:

```
load r1, #0x1000
nop
dec r6, s[r1]b, #1
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: dec u0, u1, #3
line_1: dec r4, r1, #15
line_2: [z] dec r3, r4, #1    // condition [z] comes from line_0
                              // but r4 from line_1
```

3.2.15 gie - get interrupt enable flags

Note: The **gie**-command is not available on netX 10.

Read the interrupt enable flags into a work register. This command does not affect the interrupt enable flags.

The two LSBs of the target work register (c) are set to the value of the interrupt enable flags. All other bits are set to zero.

Operation

c := ief

Syntax

gie wreg

wreg r0 ... r7

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Copy the interrupt enable flags into r0:

```
gie r0              // r0 := interrupt enable flags
```

3.2.16 gsie - get and set interrupt enable flags

Note: The **gsie**-command is not available on netX 10.

Read the interrupt enable flags into a work register and set, clear or preserve the flags.

The two LSBs of the target work register (c) are set to the value of the interrupt enable flags. All other bits are set to zero.

The irq and fiq flags can be changed separately. The interrupt enable mask (iem) defines if a flag is to be affected or not. The interrupt enable value (iev) defines the new value of each flag.

Note: Defining an “iem” of zero is not allowed.

iem	fiq enable flag	irq enable flag
b01	not affected	iev[0]
b10	iev[1]	not affected
b11	iev[1]	iev[0]

Interrupt locks can be implemented efficiently with this instruction. The lock latency is one instruction cycle. This means that an interrupt can still occur directly after a locking gsie instruction, but not after the next instruction, i. e. the first instruction of the critical section (see example below).

Operation

c := ief

ief := (~iem & ief) | (iem & iev)

Syntax

gsie wreg, #iem, #iev

wreg r0 ... r7

iem interrupt enable mask

iev interrupt enable value

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Copy the interrupt enable flags into r0, disable all interrupts:

```
// Enter critical section
gsie r0, #3, #0         // r0 := interrupt enable flags
                        // irq enable flag = 0
                        // fiq enable flag = 0

// <= latest position of interrupts before critical section

load r1, [r2]           // First instruction of critical section
store [r2], z0

// Leave critical section
sie r0
```

Copy the interrupt enable flags into r0, disable IRQs, leave FIQs unaffected:

```
gsie r0, #1, #0         // r0 := interrupt enable flags
                        // irq enable flag = 0
```

3.2.17 hex2bcd – hexadecimal to binary coded decimal

Note: The **hex2bcd**-command is not available on netX 10.

Convert the lower 10 bits of source 1 (a) to binary coded decimal target. Source 2 (b) will be ignored. The result (c) is from 0x000 to 0x999. If source 1 (a) is greater 999 and smaller 1024, the overflow flag (v) will be set and result (c) will be undefined. The upper bits of source 1 (a[31:10]) will be ignored.

Operation

c := bcd(hex[a])

Syntax

hex2bcd reg, [pc + const], reg* // const = -2^{16} … 2^{16}-4 (dword aligned)

hex2bcd reg, [const], reg* // const = 0 … 2^{17}-4 (dword aligned)

hex2bcd reg, const, reg* // const = -2^{14} … 2^{14}-1

{[cond]} **hex2bcd** reg, reg, reg*

{[cond]} **hex2bcd** reg, {s}[pc + wreg{++}]{b}, reg*

{[cond]} **hex2bcd** reg, {s}[wreg{++}]{b}, reg*

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

*reg** *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command) source 2 is ignored in this operation, use z0*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	+1	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	+2	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Convert the work register 0 from bcd to hex. Save the result in work register 4:

```
load r0, #0x7b
hex2bcd r4, r0, z0   // r4 := 0x123
load r0, #0x010003e7 *
hex2bcd r4, r0, z0   // r4 := 0x999 (upper bits ignored)
load r0, #h0x3e8
hex2bcd r4, r0, z0   // r4 := undefined (v flag set)
```

* see section *3.1 Using large values* on page 28 for handling large values.

Convert the value of address 0x1000. Save the result in work register 6:

```
load r1, #0x1000
nop
hex2bcd r6, [r1]d, z0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: hex2bcd u0, u1, z0
line_1: hex2bcd r4, r1, z0
line_2: [v] hex2bcd r3, r4, z0  // condition [v] comes from line_0
                                // but r4 from line_1
```

3.2.18 imp - bitwise implication

Implication a → b with target (c), source 1 (a) and source 2 (b).

a	b	c
0	0	1
0	1	1
1	0	0
1	1	1

Operation

c := ~a | b

Syntax

imp reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

imp reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

imp reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **imp** reg, reg, reg

{[cond]} **imp** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **imp** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
conditional	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Perform a bitwise implication with the value of work register 0 (source 1) and that of user register 1 (source 2). Save the result in work register 4:

```
imp r4, r0, u1              // r4 := ~r0 | u1
```

Perform a bitwise implication with the sign extended byte value (source 1) of address 0x1000 and the value of work register 0 (source 2). Save the result in work register 6:

```
load r1, #0x1000
nop
imp r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: imp u0, u1, u2
line_1: imp r0, r1, r2
line_2: [z] imp r3, r4, r0     // condition [z] comes from line_0
                               // but r0 from line_1
```

3.2.19 inc - increment

Increment source 1 (a) by a constant (b) (0 to 15).

Operation

c := a + b

Syntax

inc reg, [pc + const], #0…15 // const = -2^{16} … 2^{16}-4 (dword aligned)

inc reg, [const], #0…15 // const = 0 … 2^{17}-4 (dword aligned)

inc reg, const, #0…15 // const = -2^{14} … 2^{14}-1

{[cond]} **inc** reg, reg, #0…15

{[cond]} **inc** reg, {s}[pc + wreg{++}]{b}, #0…15

{[cond]} **inc** reg, {s}[wreg{++}]{b}, #0…15

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	+1	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Increment the value of work register 0 by two. Save the result in work register 4:

```
load r0, #256                // r0 := 256
inc r4, r0, #2               // r4 := 256 + 2 = 258
```

Increment the sign extended byte value of address 0x1000 by one. Save the result in work register 6:

```
load r1, #0x1000
nop
inc r6, s[r1]b, #1
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: inc u0, u1, #3
line_1: inc r4, r1, #15
line_2: [z] inc r3, r4, #1    // condition [z] comes from line_0
                              // but r4 from line_1
```

3.2.20 inv - bitwise inversion

Inversion of source 1 (a). Source 2 is ignored. Save the result in target register (c)

a	b	c
0	x	1
1	x	0

Operation

c := ~a

Syntax

inv reg, [pc + const], reg* // const = -2^{16} … 2^{16}-4 (dword aligned)

inv reg, [const], reg* // const = 0 … 2^{17}-4 (dword aligned)

inv reg, const, reg* // const = -2^{14} … 2^{14}-1

{[cond]} **inv** reg, reg, reg*

{[cond]} **inv** reg, {s}[pc + wreg{++}]{b}, reg*

{[cond]} **inv** reg, {s}[wreg{++}]{b}, reg*

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

*reg** *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command) source 2 is ignored in this operation, use z0*

Flags

set for conditional	z / nz	c / nc	s / ns	v / nv	e / ne	gu / leu	gs / les	geu / lu	ges / ls	r0z / r0nz	r1z / r1nz	r2z / r2nz	r3z / r3nz	r0123z / r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Invert the value in work register 0 (source 1). Save the result in work register 4:

```
inv r4, r0, z0                // r4 := ~r0
```

Invert the sign extended byte value (source 1) of address 0x1000. Save result in work register 6:

```
load r1, #0x1000
nop
inv r6, s[r1]b, z0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: inv u0, u1, z0
line_1: inv r0, r1, z0
line_2: [z] inv r3, inv, z0   // condition [z] comes from line_0
                              // but r0 from line_1
```

3.2.21 jmp - jump

A near jump with or without condition (cond) to a specified target address (trg).

Operation

if (cond) pc := trg

Syntax

jmp trg

jmp cond, trg

cond conditional flag (bypassed, i.e. from the last command)

trg 1) const = -2^{25} … 2^{25} (dword aligned) for near jumps (use load pc for far jumps)

2) label (compiler handling)

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	-	-	-	-	-	-	-	-	-	-	-	-	-	-
execution	-	-	-	-	-	-	-	-	-	-	-	-	-	-

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Conditional jump:

```
line_0:  sub r3, r2, r1
line_1:  jmp nz, line_0   // jump if the zero flag was not set in the last command

line_2:  add r3, r2, r1
line_3:  jmp c, line_2    // jump if the carry flag was set in the last command
```

Conditonal jump with work register flags, work register (not-) zero flags are from the next-to-last command (no bypass):

```
line_0:  add r0, r1, r2
nop                       // push r0 out of pipeline
line_1:  jmp r0z, line_0  // jump if the work register 0 is zero
```

Jump without condition:

```
line_end:  jmp line_end
```

3.2.22 jmpdec - jump and decrement

A near jump with work register condition. Jumps to the designated target address (trg) if the condition flag is set. Additionally, work register (c) associated with the condition is decremented (irrespective of flag value). This can be used for loops. A data dependency exists between jmpdec and the previous instruction, i.e. there must always be an additional instruction between loading counter register (r0) and jmpdec r0nz. However, the decrementation of r0 in the command is bypassed, i.e. no additional instruction is necessary and the flags for jump and execution are valid for the next instruction.

Operation

if (flag) pc := trg; c := c – 1

Syntax

jmpdec wr_flag, trg

wr_flag *work register flag r0nz … r3nz, r0123nz*

trg *1) const = -2^{25} … 2^{25} (dword aligned) for near jumps (use load pc for far jumps)*

2) label (compiler handling)

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	-	-	-	-	-	-	-	-	-	+1	+1	+1	+1	+1
execution	-	-	-	-	-	-	-	-	-	+1	+1	+1	+1	+1

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Conditional jump with work register decrementation:

```
line_0:  load r0, #100
line_1:  nop                  // push r0 out of pipeline
line_2:  jmpdec r0nz, line_2  // jump if r0 is not zero and decrement r0
                              // (the new value is bypassed to the next command,
                              // i.e. jumping to the same line is possible)
```

3.2.23 load

Load source (a) to target (c). (Format: **load** target, source)

Operation

c := a

or

c := [a]

Syntax

load reg, {s}[reg + const]b // const = -2^{16} … 2^{16}-1 (byte aligned)

load reg, {s}[reg + const]w // const = -2^{17} … 2^{17}-2 (word aligned)

load reg, [reg + const]d // const = -2^{18} … 2^{18}-4 (dword aligned)

load reg, const // const = -2^{18} … 2^{18}-1

{[cond]} **load** reg, s[reg + wreg++]w_bs

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

{d_bs} *d_bs - dword access with byte swap*

d_ws - dword access with word swap

w_bs - word access with byte swap

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Load and sign extend the word (16-bit) value from address pc – 0x1000 to user register 4:

```
load u4, s[pc + #-0x1000]w *
```

Load the constant 0x55 to work register r6:

```
load r6, #0x55
```

Load the sign extended and byte swapped word value (swap before sign extension) from the calculated address (work register r0, add work register r2 with pre-decrement) to work register 6. Execute this command only if the next-to-last command sets the zero flag to valid:

```
[z] load r6, s[r0 + --r2]w_bs
```

* Currently not supported, use -4096 instead of -0x1000.

3.2.24 lsl - logical shift left

Logical shift left of source 1 (a) with source 2 (b). The trailing bits of target (c) are set to zero.

If source 2 is a register, an additional bit mask operation can be applied to the result of the shift operation. The mask operation is either NAND, AND or OR and it is defined by b[6:5]. The operand (bit mask) is constructed from b[31:7].

The carry flag is set to bit[32] of the shift operation.

Operation

c := a << b[4:0]

// additional mask operation

case b[6:5]

00: c := c

01: c := c & ~(b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[31:8])

10: c := c & (b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[31:8])

11: c := c | (b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[31:8])

endcase

Syntax

lsl reg, [pc + const], reg* // const = -2^{16} … 2^{16}-4 (dword aligned)

lsl reg, [const], reg* // const = 0 … 2^{17}-4 (dword aligned)

lsl reg, const, reg* // const = -2^{14} … 2^{14}-1

{[cond]} **lsl** reg, reg, reg*

{[cond]} **lsl** reg, {s}[pc + wreg{++}]{b}, reg*

{[cond]} **lsl** reg, {s}[wreg{++}]{b}, reg*

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

*reg** *1) reg[4:0] = shift value; reg[6:5] = mode; reg[7] = mask[31 … 24]; reg[31:8] = mask[23:0]*

or

2) instead of register use constant: 0 … 31

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	+1	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	+2	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

The value of work register 0 is logically shifted left by 6. Save the result in work register 4:

```
load r0, #0xf        // r0 := 15
lsl r4, r0, #6  // r4 := 15 << 6 = 960 (0x3c0)
```

The sign extended byte value of address 0x1000 is logically shifted left by r0 (count 5). Save the result in work register 6:

```
load r1, #0x1000
load r0, #5
lsl r6, s[r1]b, r0
```

Shift r0 left by 1. Set the LSB to 1. Save the result in r1:

```
load u0, #0x161
lsl r1, r0, u0             // r1 := (r0 << 1) | 1
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: lsl u0, u1, u2
line_1: lsl r5, r1, r2
line_2: [z] lsl r3, r4, r5 // condition [z] comes from line_0 but r5 from line_1
```

3.2.25 lsr - logical shift right

Logical shift right of source 1 (a) with source 2 (b). The leading bits of the target (c) are set to zero.

If source 2 is a register, an additional bit mask operation can be applied to the result of the shift operation. The mask operation is either NAND, AND or OR and it is defined by b[6:5]. The operand (bit mask) is constructed from b[31:7].

Operation

c := a >> b[4:0]

// additional mask operation

case b[6:5]

00: c := c

01: c := c & ~(b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[31:8])

10: c := c & (b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[31:8])

11: c := c | (b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[31:8])

endcase

Syntax

lsr reg, [pc + const], reg* // const = -2^{16} … 2^{16}-4 (dword aligned)

lsr reg, [const], reg* // const = 0 … 2^{17}-4 (dword aligned)

lsr reg, const, reg* // const = -2^{14} … 2^{14}-1

{[cond]} **lsr** reg, reg, reg*

{[cond]} **lsr** reg, {s}[pc + wreg{++}]{b}, reg*

{[cond]} **lsr** reg, {s}[wreg{++}]{b}, reg*

{[cond]} conditional execution with the flags (not bypassed, i.e. from the next-to-last command)

{s} sign extension

{++} post-increment or pre-decrement (--wreg)

{b} b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

wreg r0 … r7 (not bypassed, i.e. from the next-to-last command)

*reg** 1) reg[4:0] = shift value; reg[6:5] = mode; reg[7] = mask[31 … 24]; reg[31:8] = mask[23:0]

or

2) instead of register use constant: 0 ... 31

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

The value of work register 0 is logically shifted right by 6. Save the result in work register 4:

```
load r0, #0xff  // r0 := 0xff
```

```
lsr r4, r0, #6  // r4 := 0xff >> 6 = 3
```

The sign extended byte value of address 0x1000 is logically shifted right by r0 (count 5). Save the result in work register 6. Execute this command only if the next-to-last command sets the zero flag to valid:

```
load r1, #0x1000
load r0, #5
[z] lsr r6, s[r1]b,r0
```

Shift r0 right by 16 bits. Mask out the byte value. Save the result in r1

```
load u0, #0xff50
lsr r1, r0, u0       // r1 := (r0 >> 16) & 0xff
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: lsr u0, u1, u2
line_1: lsr r5, r1, r2
line_2: [z] lsr r3, r4, r5 // condition [z] comes from line_0 but r5 from line_1
```

3.2.26 maxs - maximum signed

Signed mathematical maximum operation. The target (c) is set to the greater signed value of both, source 1 (a) and source 2 (b).

Operation

c := (signed(a) > signed(b)) ? a : b

Syntax

maxs reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

maxs reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

maxs reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **maxs** reg, reg, reg

{[cond]} **maxs** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **maxs** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
conditional	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

The signed value of work register 0 is the limit. The signed value of user register 1 is the value. If the signed u1 is smaller than the signed r0, the result is r0, otherwise it is u1. Save the result in work register 4.

```
maxs r4, r0, u1
```

The sign extended and signed byte value of address 0x1000 is the limit. The signed value of work register 0 is the value. If the signed value is smaller than the signed limit, the result is the signed value of address 0x1000, otherwise it is r0. Save the result in work register 6:

```
load r1, #0x1000
nop
maxs r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: maxs u0, u1, u2
line_1: maxs r5, r1, r2
line_2: [z] maxs r3, r4, r5    // condition [z] comes from line_0
                               // but r5 from line_1
```

3.2.27 maxu - maximum unsigned

Unsigned mathematical maximum operation. The target (c) is set to the greater unsigned value of both, source 1 (a) and source 2 (b).

Operation

c := (unsigned(a) > unsigned(b)) ? a : b

Syntax

maxu reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

maxu reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

maxu reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **maxu** reg, reg, reg

{[cond]} **maxu** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **maxu** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

The unsigned value of work register 0 is the limit. The unsigned value of user register 1 is the value. If the unsigned u1 is smaller than the unsigned r0, the result is r0, otherwise it is u1. Save the result in work register 4.

```
maxu r4, r0, u1
```

The unsigned byte value of address 0x1000 is the limit. The unsigned value of work register 0 is the value. If the unsigned value is smaller than the unsigned limit, the result is the unsigned value of address 0x1000, otherwise it is r0. Save the result in work register 6:

```
load r1, #0x1000
nop
maxu r6, [r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: maxu u0, u1, u2
line_1: maxu r5, r1, r2
line_2: [z] maxu r3, r4, r5     // condition [z] comes from line_0
                                // but r5 from line_1
```

3.2.28 mean - arithmetic mean

Arithmetic mean of signed source 1 (a) and signed source 2 (b) rounded up or down to the nearest even of the result (c).

Operation

c := RoundToNearestEven((signed(a) + signed(b)) / 2)

Syntax

mean reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

mean reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

mean reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **mean** reg, reg, reg

{[cond]} **mean** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **mean** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	**z**	**c**	**s**	**v**	**e**	**gu**	**gs**	**geu**	**ges**	**r0z**	**r1z**	**r2z**	**r3z**	**r0123z**
	nz	**nc**	**ns**	**nv**	**ne**	**leu**	**les**	**lu**	**ls**	**r0nz**	**r1nz**	**r2nz**	**r3nz**	**r0123nz**
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Calculate the arithmetic mean of the signed value of work register 0 and the signed value of user register 1. The result is round up or down to the nearest even. Save the result in work register 4:

```
load r0, #2          // r0 := 2
load u1, #4          // u1 := 4
mean r4, r0, u1      // r4 := (2 + 4)/2 = 3
load u1, #5          // u1 := 5
mean r4, r0, u1      // r4 := 4 (3.5 rounded up to nearest even)
load u1, #6          // u1 := 6
mean r4, r0, u1      // r4 := 4
load u1, #7          // u1 := 7
mean r4, r0, u1      // r4 := 4 (4.5 rounded down to nearest even)
```

Calculate the arithmetic mean of the sign extended and signed byte value of address 0x1000 and the signed value of work register 0. Save the result in work register 6:

```
load r1, #0x1000
nop
mean r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: mean u0, u1, u2
line_1: mean r5, r1, r2
line_2: [z] mean r3, r4, r5 // condition [z] comes from line_0 but r5 from line_1
```

3.2.29 mins - minimum signed

Signed mathematical minimum operation. The target (c) is set to the smaller signed value of both, source 1 (a) and source 2 (b).

Operation

c := (signed(a) < signed(b)) ? a : b

Syntax

mins reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

mins reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

mins reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **mins** reg, reg, reg

{[cond]} **mins** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **mins** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

The signed value of work register 0 is the limit. The signed value of user register 1 is the value. If the signed u1 is greater than the signed r0, the result is r0, otherwise it is u1. Save the result in work register 4.

```
mins r4, r0, u1
```

The sign extended and signed byte value of address 0x1000 is the limit. The signed value of work register 0 is the value. If the signed value is greater than the signed limit, the result is the signed value of address 0x1000, otherwise it is r0. Save the result in work register 6:

```
load r1, #0x1000
nop
mins r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: mins u0, u1, u2
line_1: mins r5, r1, r2
line_2: [z] mins r3, r4, r5     // condition [z] comes from line_0
                                  // but r5 from line_1
```

3.2.30 minu - minimum unsigned

Unsigned mathematical minimum operation. The target (c) is set to the smaller unsigned value of both, source 1 (a) and source 2 (b).

Operation

c := (unsigned(a) < unsigned(b)) ? a : b

Syntax

minu reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

minu reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

minu reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **minu** reg, reg, reg

{[cond]} **minu** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **minu** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

The unsigned value of work register 0 is the limit. The unsigned value of user register 1 is the value. If the unsigned u1 is greater than the unsigned r0, the result is r0, otherwise it is u1. Save the result in work register 4.

```
minu r4, r0, u1
```

The unsigned byte value of address 0x1000 is the limit. The unsigned value of work register 0 is the value. If the unsigned value is greater than the unsigned limit, the result is the unsigned value of address 0x1000, otherwise it is r0. Save the result in work register 6:

```
load r1, #0x1000
nop
minu r6, [r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: minu u0, u1, u2
line_1: minu r5, r1, r2
line_2: [z] minu r3, r4, r5      // condition [z] comes from line_0
                                 // but r5 from line_1
```

3.2.31 mov - move

Copies source 1 (a) to target (c). Source 2 is ignored. This operation sets no flags except for the work register flags.

a	b	c
0	x	0
1	x	1

Operation

c := a

Syntax

mov reg, [pc + const], reg* // const = -2^{16} … 2^{16}-4 (dword aligned)

mov reg, [const], reg* // const = 0 … 2^{17}-4 (dword aligned)

mov reg, const, reg* // const = -2^{14} … 2^{14}-1

{[cond]} **mov** reg, reg, reg*

{[cond]} **mov** reg, {s}[pc + wreg{++}]{b}, reg*

{[cond]} **mov** reg, {s}[wreg{++}]{b}, reg*

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

*reg** *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command) source 2 is ignored in this operation, use z0*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Move the value in work register 0 (source 1) to work register 4:

```
mov r4, r0, z0       // r4 := r0
```

Move the sign extended byte value (source 1) from address 0x1000 to work register 6:

```
load r1, #0x1000
nop
mov r6, s[r1]b, z0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: mov u0, u1, z0
line_1: mov r0, r1, z0
line_2: [z] mov r3, r0, z0      // condition [z] comes from line_0 but r0 from line_1
```

3.2.32 muls - multiply signed

32-/64-bit signed multiplication with target (c), source 1 (a) and source 2 (b). If the target register is 32-bit, a 32-bit multiplication is performed. If the target register is 64-bit, a 64-bit multiplication is performed. In all cases, defined and mathematically correct overflow flags are set.

Operation

c := signed(a) * signed(b)

Syntax

muls reg, const, reg // const = -2^{14} … $2^{14}-1$

{[cond]} **muls** reg*, reg, reg

{[cond]} **muls** reg64, reg, reg

{[cond]} conditional execution with the flags (not bypassed, i.e. from the next-to-last command)

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

*reg** r0 … r7, u0 … u4, st

reg64 u10, u32 (64-bit registers)

Flags

set for	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
conditional	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Multiply the signed value of work register 0 with the signed value of user register 1. Save the result in work register 4 (32-bit result, set overflow flag) and user register 3/2 (64-Bit):

```
muls r4, r0, u1      // r4 := r0 * u1 (32-bit result, set overflow flag)
muls u32, r0, u1     // u32 := r0 * u1 (64-bit result)
```

Note: If a 64-bit multiplication is performed (target is u10 or u32), the result is available in the instruction after next. This data dependency is not bypassed by hardware. An additional command (e.g. nop) is required for that after the multiplication.

```
muls u10, r1, r2
nop                  // muls is still computing, u1 and u0 are not yet ready
add r0, u1, u0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: muls u0, u1, u2
line_1: muls r0, r1, r2
line_2: [v] muls r3, r4, r0    // condition [v] comes from line_0
                               // but r0 from line_1
```

3.2.33 mulu - multiply unsigned

32/64-bit unsigned multiplication with target (c), source 1 (a) and source 2 (b).). If the target register is 32-bit, a 32-bit multiplication is performed. If the target register is 64-bit, a 64-bit multiplication is performed. In all cases, defined and mathematically correct overflow flags are set.

Operation

c := unsigned(a) * unsigned(b)

Syntax

mulu reg, const, reg // const = -2^{14} … $2^{14}-1$

{[cond]} **mulu** reg*, reg, reg

{[cond]} **mulu** reg64, reg, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

*reg** *r0 … r7, u0 … u4, st*

reg64 *u10, u32 (64-bit registers)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Multiply the unsigned value of work register 0 with the unsigned value of user register 1. Save the result in work register 4 (32-bit result, set overflow flag) and user register 3/2 (64-Bit):

```
mulu r4, r0, u1      // r4 := r0 * u1 (32-bit result, set overflow flag)
mulu u32, r0, u1     // r4 := r0 * u1 (64-bit result)
```

Note: If a 64-bit multiplication is performed (target is u10 or u32), the result is available in the instruction after next. This data dependency is not bypassed by hardware. An additional command (e.g. nop) is required for that after the multiplication.

```
mulu u10, r1, r2
nop                  // mulu is still computing, u1 and u0 are not yet ready
add r0, u1, u0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: mulu u0, u1, u2
line_1: mulu r0, r1, r2
line_2: [v] mulu r3, r4, r0      // condition [v] comes from line_0
                                 // but r0 from line_1
```

3.2.34 nand - bitwise NAND

Bitwise NAND operation with target (c), source 1 (a) and source 2 (b).

a	b	c
0	0	1
0	1	1
1	0	1
1	1	0

Operation

c := ~(a & b)

Syntax

nand reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

nand reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

nand reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **nand** reg, reg, reg

{[cond]} **nand** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **nand** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Perform a bitwise NAND operation with the value of work register 0 and the value of user register 1. Save the result in work register 4:

```
nand r4, r0, u1       // r4 := ~(r0 & u1)
```

Perform a bitwise NAND operation with the sign extended byte value of address 0x1000 and the value of work register 0. Save the result in work register 6:

```
load r1, #0x1000
nop
nand r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: nand u0, u1, u2
line_1: nand r5, r1, r2
line_2: [z] nand r3, r4, r5   // condition [z] comes from line_0
                              // but r5 from line_1
```

3.2.35 nimp - bitwise inverted implication

Inverted implication a → b with target (c), source 1 (a) and source 2 (b).

a	b	c
0	0	0
0	1	0
1	0	1
1	1	0

Operation

c := a & ~b

Syntax

nimp reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

nimp reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

nimp reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **nimp** reg, reg, reg

{[cond]} **nimp** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **nimp** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Perform a bitwise inverted implication with the value of work register 0 (source 1) and the value of user register 1 (source 2). Save the result in work register 4:

```
nimp r4, r0, u1                 // r4 := r0 & ~u1
```

Perform a bitwise inverted implication with the sign extended byte value (source 1) of address 0x1000 and the value of work register 0 (source 2). Save the result in work register 6:

```
load r1, #0x1000
nop
nimp r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: nimp u0, u1, u2
line_1: nimp r0, r1, r2
line_2: [z] nimp r3, r4, r0     // condition [z] comes from line_0
                                // but r0 from line_1
```

3.2.36 nop - no operation

Wait one instruction cycle. All flags of the previous instruction that are not bypassed are stable in next instruction.

Operation

-

Syntax

nop

Flags

set for	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
conditional	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	-	-	-	-	-
execution	-	-	-	-	-	-	-	-	-	-	-	-	-	-

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Example

Conditional jump with work register decrementation:

```
line_0: sub r0, r1, r2
line_1: nop                  // push r0 out of pipeline
line_2: jmp r0nz, line_xyz   // jump if r0 is not zero
```

3.2.37 nor - bitwise NOR

Bitwise NOR operation with target (c), source 1 (a) and source 2 (b).

a	b	c
0	0	1
0	1	0
1	0	0
1	1	0

Operation

c := ~(a | b)

Syntax

nor reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

nor reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

nor reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **nor** reg, reg, reg

{[cond]} **nor** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **nor** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Perform a bitwise NOR operation with the value of work register 0 and the value of user register 1. Save the result in work register 4:

```
nor r4, r0, u1                // r4 := ~(r0 | u1)
```

Perform a bitwise NOR operation with the sign extended byte value of address 0x1000 and the value of work register 0. Save the result in work register 6:

```
load r1, #0x1000
nop
nor r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: nor u0, u1, u2
line_1: nor r5, r1, r2
line_2: [z] nor r3, r4, r5      // condition [z] comes from line_0
                                // but r5 from line_1
```

3.2.38 or - bitwise OR

Bitwise OR operation with target (c), source 1 (a) and source 2 (b).

a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

Operation

c := a | b

Syntax

or reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

or reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

or reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **or** reg, reg, reg

{[cond]} **or** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **or** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Perform a bitwise OR operation with the value of work register 0 and the value of user register 1. Save the result in work register 4:

```
or r4, r0, u1               // r4 := r0 | u1
```

Perform a bitwise OR operation with the sign extended byte value of address 0x1000 and the value of work register 0. Save the result in work register 6:

```
load r1, #0x1000
nop
or r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: or u0, u1, u2
line_1: or r5, r1, r2
line_2: [z] or r3, r4, r5   // condition [z] comes from line_0
                            // but r5 from line_1
```

3.2.39 reli - release interrupt

Release interrupt to allow nested interrupts. After reli, the xPIC can execute further interrupts.

Operation

-

Syntax

reli

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	-	-	-	-	-
execution	-	-	-	-	-	-	-	-	-	-	-	-	-	-

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

```
line_0: sub r4, r5, r6
line_1: add r0, r1, r2
line_2: reli
line_3: add r7, r0, r4
```

For further details, see chapter *Interrupt handling* on page 21.

3.2.40 relreti - release and return from interrupt

Release and return from interrupt. Use for non-nested interrupts. The registers of the main program must be restored manually. The pc and stats must be prepared before this instruction, see example.

Operation

-

Syntax

relreti

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	-	-	-	-	-
execution	-	-	-	-	-	-	-	-	-	-	-	-	-	-

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

```
line_0: load st, [z0 + --r3]d
line_1: load xpic_pc, [z0 + --r3]d
line_2: relreti
```

For further details, see chapter *Interrupt handling* on page 21.

3.2.41 retf - return from fast interrupt

Return from fast interrupt. Switches register bank and pc to main program. The flags of the main program are restored.

Operation

-

Syntax

retf

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	R	R	R	R	R	R	R	R	R	R	R	R	R	R
execution	R	R	R	R	R	R	R	R	R	R	R	R	R	R

(R) Restored

Examples

```
line_0: sub r0, r1, r2
line_1: store [z0 + r3++]d, r0
line_2: retf
```

For further details, see chapter *Interrupt handling* on page 21.

3.2.42 reti - return from interrupt

Return from interrupt. The registers of the main program must be restored manually. IRQ must be released before this command may be called (see **reli)**. The pc and stats must be prepared before this instruction, see example.

Operation

-

Syntax

reti

Flags

set for	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
conditional	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	-	-	-	-	-
execution	-	-	-	-	-	-	-	-	-	-	-	-	-	-

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

```
line_50: reli
…
line_100: load st, [z0 + --r3]d
line_101: load xpic_pc, [z0 + --r3]d
line_1022: reti
```

For further details, see chapter *Interrupt handling* on page 21.

3.2.43 rol - rotate left

Rotate left of source 1 (a) by source 2 (b).

If source 2 is a register, an additional bit mask operation can be applied to the result of the shift operation. The mask operation is either NAND, AND or OR and it is defined by b[6:5]. The operand (bit mask) is constructed from b[31:7].

Note: Right rotation (ror) can also be implemented with this command due to symmetry: ror(x) == rol(32-x)

Operation

```
c := a << b[4:0] | a >> (32 - b[4:0])
// additional mask operation
case b[6:5]
    00: c = c
    01: c = c & ~(b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[31:8])
    10: c = c &  (b[7], b[7], b[7] ,b[7], b[7], b[7], b[7], b[7], b[31:8])
    11: c = c |  (b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[7], b[31:8])
endcase
```

Syntax

rol reg, [pc + const], reg* // const = -2^{16} … 2^{16}-4 (dword aligned)

rol reg, [const], reg* // const = 0 … 2^{17}-4 (dword aligned)

rol reg, const, reg* // const = -2^{14} … 2^{14}-1

{[cond]} **rol** reg, reg, reg*

{[cond]} **rol** reg, {s}[pc + wreg{++}]{b}, reg*

{[cond]} **rol** reg, {s}[wreg{++}]{b}, reg*

{[cond]} conditional execution with the flags (not bypassed, i.e. from the next-to-last command)

{s} sign extension

{++} post-increment or pre-decrement (--wreg)

{b} b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

wreg r0 … r7 (not bypassed, i.e. from the next-to-last command)

*reg** 1) reg[4:0] = shift value; reg[6:5] = mode; reg[7] = mask[31 … 24]; reg[31:8] = mask[23:0]

or

2) instead of register use constant: 0 ... 31

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

The value of work register 0 is rotated left by 6. Save the result in work register 4:

```
load r0, #0xff00ff00 *
rol r4, r0, #6        // r4 := 0xc03fc03f
```

* see section *3.1 Using large values* on page 28 for handling large values.

The sign extended byte value of address 0x1000 is rotated left by r0 (count 5). Save the result in work register 6:

```
load r1, #0x1000
load r0, #5
rol r6, s[r1]b, r0
```

Rotate r0 left by 14 bits. Apply an AND operation with mask 0xff00ff00. Save the result in r1:

```
load u0, #0x00ff00ce *
rol r1, r0, u0                // r1 := ((r0 << 14) | (r0 >> 18)) & 0xff00ff00
```

* see section *3.1 Using large values* on page 28 for handling large values.

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: rol u0, u1, u2
line_1: rol r5, r1, r2
line_2: [z] rol r3, r4, r5     // condition [z] comes from line_0
                               // but r5 from line_1
```

3.2.44 sie - set interrupt enable flags

Note: The **sie**-command is not available on netX 10.

Write the interrupt enable flags from a work register or a constant mask/value pair.

The data source can either be a work register or a pair of constant 3-bit values.

When a work register is used as source (c), the two LSBs of the source register are written to the interrupt enable flags, all other bits are ignored.

This command does not use the bypass value of the work register if it was the target of an ALU/load operation in the previous instruction.

When a constant mask/value pair is used as source, the irq and fiq flags can be changed independently. The interrupt enable mask (iem) defines if a flag is to be affected or not. The interrupt enable value (iev) defines the new value of each flag:

iem	fiq enable flag	irq enable flag
b01	not affected	iev[0]
b10	iev[1]	not affected
b11	iev[1]	iev[0]

Note: Defining an “iem” of zero is not allowed.

Interrupts can be locked or unlocked with this instruction. The latency is one instruction cycle. This means that an interrupt can still occur directly after a locking *sie* instruction, but not after the next instruction (see example below).

Operation

work register: ief := c[1:0]

constants: ief := (~iem & ief) | (iem & iev)

Syntax

sie wreg

sie #iem, #iev

wreg r0 … r7

iem interrupt enable mask

iev interrupt enable value

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Mutex acquisition with preservative interrupt lock/unlock:

```
load u1, #1

// Enter critical section
gsie r0, #3, #0          // clear both interrupt enable flags

// <= latest position of interrupts before critical section

load u0, [r1]            // First instruction of critical section
store [r1], u1

// Leave critical section
sie r0                   // restore interrupt enable flags

or z0, u0, z0

// <= processing of pending interrupts

jmp z, mutex_fail
```

Mutex acquisition with overwriting IRQ lock/unlock and non-affected FIQs:

```
load u1, #1

// Enter critical section
sie #1, #0               // clear IRQ enable flag

// <= latest position of IRQ processing before critical section

load u0, [r1]            // First instruction of critical section
// <= possible FIQ processing here since FIQs are not locked
store [r1], u1

// Leave critical section
sie #1, #1               // set IRQ enable flag

or z0, u0, z0

// <= processing of pending IRQs

jmp z, mutex_fail
```

3.2.45 store - store

Store source register (b) to target memory address (a). (Format: **store** target, source)

Operation

[a] := b

Syntax

store [reg + const]b, reg // const = -2^{16} … 2^{16}-1 (byte aligned)

store [reg + const]w, reg // const = -2^{17} … 2^{17}-2 (word aligned)

store [reg + const]d, reg // const = -2^{18} … 2^{18}-4 (dword aligned)

{[cond]} **store** [reg + wreg++]w_bs, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

{d_bs} *d_bs - dword access with byte swap*

d_ws - dword access with word swap

w_bs - word access with byte swap

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Store the lower word (16-bit) value from user register 4 to address pc – 0x1000:

```
store [pc + #-0x1000]w, u4 *
```

Store the byte swapped lower word (16-bit) value from work register 6 to the calculated address (work register r0, add work register r2 with pre-decrement). Execute this command only if the next-to-last command sets the zero flag to valid:

```
[z] store [r0 + --r2]w_bs, r6
```

* Currently not supported, use -4096 instead of -0x1000.

3.2.46 sub - subtract

Integer subtraction, target (c) is source 1 (a) minus source 2 (b). The carry (borrow) flag is set if source 2 is greater source 1.

Operation

c := a - b

Syntax

sub reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

sub reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

sub reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **sub** reg, reg, reg

{[cond]} **sub** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **sub** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

Note: The carry flag in the subtraction is also known as borrow flag.

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	+1	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Subtract the value of user register 1 from that of work register 0. Save the result in work register 4:

```
sub r4, r0, u1                    // r4 := r0 - u1
```

Subtract the sign extended byte value of address 0x1000 from the value of work register 0. Save the result in work register 6:

```
load r1, #0x1000
nop
sub r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: sub u0, u1, u2
line_1: sub r0, r1, r2
line_2: [z] sub r3, r4, r0      // condition [z] comes from line_0
                                // but r0 from line_1
```

3.2.47 subc - subtract with carry

Integer subtraction with carry, target (c) is source 1 (a) minus source 2 (b) minus carry flag. The carry (borrow) flag is set if source 2 is greater source 1 minus carry flag.

Operation

c := a – b – carry(flag)

Syntax

subc reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

subc reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

subc reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **subc** reg, reg, reg

{[cond]} **subc** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **subc** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

Note: The carry flag in subtraction is also known as borrow flag.

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	+1	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Subtract the value of user register 1 and the carry flag from work register 0. Save the result in work register 4:

```
subc r4, r0, u1                 // r4 := r0 - u1 - carry
```

Subtract the value of work register 0 and the carry flag from the sign extended byte value of address 0x1000. Save the result in work register 6:

```
load r1, #0x1000
nop
subc r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: subc u0, u1, u2
line_1: subc r0, r1, r2
line_2: [c] subc r3, r4, r0     // condition [c] comes from line_0
                                // but the carry for operation from line_1
```

3.2.48 subss - subtract with signed saturation

Signed integer subtraction with saturation. Source 1 (a) minus source 2 (b) is interpreted as signed. The target (c) is set to 0x80000000 if there is an underflow or to 0x7fffffff if there is an overflow in the subtraction of source 1 minus source 2.

Operation

c := ssat(signed(a) - signed(b))

ssat(x) = | -0x80000000; (x < -0x80000000)
| 0x7fffffff; (x > 0x7fffffff)
| x; else

Syntax

subss reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

subss reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

subss reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **subss** reg, reg, reg

{[cond]} **subss** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **subss** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

The flags are computed before saturation. Overflow flag (v) is an indicator for saturation: If v is set after the instruction, the result is saturated.

set for	**z**	**c**	**s**	**v**	**e**	**gu**	**gs**	**geu**	**ges**	**r0z**	**r1z**	**r2z**	**r3z**	**r0123z**
conditional	**nz**	**nc**	**ns**	**nv**	**ne**	**leu**	**les**	**lu**	**ls**	**r0nz**	**r1nz**	**r2nz**	**r3nz**	**r0123nz**
jump	+1	+1	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Subtract the signed value of user register 1 from that of work register 0. Save the result in work register 4 after saturation:

```
subss r4, r0, u1                // r4 := ssat(r0 - u1)
```

Subtract the sign extended and signed byte value of address 0x1000 from the signed value of work register 0. Save the result in work register 6 after saturation:

```
load r1, #0x1000
nop
subss r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: sub u0, u1, u2
line_1: sub r5, r1, r2
line_2: [z] subss r3, r4, r5 // condition [z] comes from line_0 but r5 from line_1
```

3.2.49 subus - subtract with unsigned saturation

Unsigned integer subtraction with saturation. Source 1 (a) minus source 2 (b) is interpreted as unsigned. The target (c) is set to 0 if there is an underflow in the subtraction of source 1 minus source 2.

Operation

c := usat(a - b)

usat(x) = | 0; (x < 0)
| x; else

Syntax

subus reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

subus reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

subus reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **subus** reg, reg, reg

{[cond]} **subus** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **subus** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

The flags are computed before saturation. Overflow flag (v) is an indicator for saturation: If v is set after the instruction, the result is saturated.

set for conditional	**z**	**c**	**s**	**v**	**e**	**gu**	**gs**	**geu**	**ges**	**r0z**	**r1z**	**r2z**	**r3z**	**r0123z**
	nz	**nc**	**ns**	**nv**	**ne**	**leu**	**les**	**lu**	**ls**	**r0nz**	**r1nz**	**r2nz**	**r3nz**	**r0123nz**
jump	+1	+1	+1	+1	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Subtract the unsigned value of user register 1 from that of work register 0. Save the result in work register 4 after saturation:

```
subus r4, r0, u1              // r4 := usat(r0 - u1)
```

Subtract the unsigned dword value of address 0x1000 from the unsigned value of work register 0. Save the result in work register 6 after saturation:

```
load r1, #0x1000
nop
subus r6, [r1]d, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: sub u0, u1, u2
line_1: sub r5, r1, r2
line_2: [z] subus r3, r4, r5    // condition [z] comes from line_0
                                // but r5 from line_1
```

3.2.50 s7_a – AND (S7 command)

Note: S7 commands are not available on netX 10.

Sets the flags in S7 status word (register u4) depending on source 1 (a - byte of the operand), source 2 (b – bit position in a) and the initial S7 status word. U4 register is hardwired for this command and represents the s7 status register with the following flags:

Bit position	S7 flag
usr4[0]	/ER
usr4[1]	VKE
usr4[2]	STA
usr4[3]	OR
usr4[4]	OS
usr4[5]	OV
usr4[6]	A0
usr4[7]	A1
usr4[8]	BIE

Operation

```
BIT = a[b]              // operand bit; a = byte; b = [0..7]
IF (BIT = 1)
      /ER = 1; STA = 1;
      IF ((OR | /ER) = 0)
            VKE = 1;
ELSE
      /ER = 1;
      STA = 0;
IF (OR = 0)
            VKE = 0;
```

Syntax

s7_a z0, [pc + const], #0…7 // const = -2^{16} … 2^{16}-4 (dword aligned)

s7_a z0, [const], #0…7 // const = 0 … 2^{17}-4 (dword aligned)

s7_a z0, const, #0…7 // const = -2^{14} … 2^{14}-1

{[cond]} **s7_a** z0, reg, #0…7

{[cond]} **s7_a** z0, {s}[pc + wreg{++}]{b}, #0…7

{[cond]} **s7_a** z0, {s}[wreg{++}]{b}, #0…7

z0 select z0 for target register, the xpic moves source 1 to target

{[cond]} conditional execution with the flags (not bypassed, i.e. from the next-to-last command)

{s} sign extension

{++} post-increment or pre-decrement (--wreg)

{b} b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

wreg r0 … r7 (not bypassed, i.e. from the next-to-last command)

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Excecute an S7 “AND” command of bit 2 and 3 from the byte in r0 (using S7 status word always located in u4).

```
s7_a z0, r0, #2              // bit #2 of r0
s7_a z0, r0, #3              // bit #3 of r0
```

Execute an S7 “AND” command of the first bit in byte value from address 0x1000 (using S7 status word always located in u4:

```
load r1, #0x1000
nop
s7_a z0, [r1]b, #0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: inc u0, u1, #3
line_1: inc r4, r1, #15
line_2: [z] s7_a z0, r4, #1    // condition [z] comes from line_0
                               // but r4 from line_1
```

Note: This command always uses register u4 as S7 status word. Register u4 is not bypassed. Bits 0 to 9 of this register will be interpreted and affected.

```
load u4, #13
nop                          // insert nop or independent command
s7_a z0, r0, #0
s7_a z0, r0, #1              // nop not necessary
s7_a z0, r0, #2
s7_a z0, r0, #3
nop                          // insert nop or independent command
store [r5],u4
```

3.2.51 s7_an – AND-NOT (S7 command)

Note: S7 commands are not available on netX 10.

Sets the flags in S7 status word (register u4) depending on source 1 (a - byte of the operand), source 2 (b – bit position in a) and the initial S7 status word. U4 register is hardwired for this command and represents the s7 status register with the following flags:

Bit position	S7 flag
usr4[0]	/ER
usr4[1]	VKE
usr4[2]	STA
usr4[3]	OR
usr4[4]	OS
usr4[5]	OV
usr4[6]	A0
usr4[7]	A1
usr4[8]	BIE

Operation

```
BIT = a[b]              // operand bit; a = byte; b = [0..7]
IF (BIT = 0)
      /ER = 1; STA = 0;
      IF ((OR | /ER) = 0)
            VKE = 1;
ELSE
      /ER = 1;
      STA = 1;
IF (OR = 0)
            VKE = 0;
```

Syntax

s7_an z0, [pc + const], #0…7 // const = -2^{16}… 2^{16}-4 (dword aligned)

s7_an z0, [const], #0…7 // const = 0 … 2^{17}-4 (dword aligned)

s7_an z0, const, #0…7 // const = -2^{14} … 2^{14}-1

{[cond]} **s7_an** z0, reg, #0…7

{[cond]} **s7_an** z0, {s}[pc + wreg{++}]{b}, #0…7

{[cond]} **s7_an** z0, {s}[wreg{++}]{b}, #0…7

z0 select z0 for target register, the xpic moves source 1 to target

{[cond]} conditional execution with the flags (not bypassed, i.e. from the next-to-last command)

{s} sign extension

{++} post-increment or pre-decrement (--wreg)

{b} b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

wreg r0 … r7 (not bypassed, i.e. from the next-to-last command)

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Excecute an S7 “AND-NOT” command of bit 2 and 3 from the byte in r0 (using S7 status word always located in u4).

```
s7_an z0, r0, #2              // bit #2 of r0
s7_an z0, r0, #3              // bit #3 of r0
```

Execute an S7 “AND-NOT” command of the first bit in byte value from address 0x1000 (using S7 status word always located in u4:

```
load r1, #0x1000
nop
s7_an z0, [r1]b, #0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: inc u0, u1, #3
line_1: inc r4, r1, #15
line_2: [z] s7_an z0, r4, #1    // condition [z] comes from line_0
                                // but r4 from line_1
```

Note: This command always uses register u4 as S7 status word. Register u4 is not bypassed. Bits 0 to 9 of this register will be interpreted and affected.

```
load u4, #13
nop                           // insert nop or independent command
s7_an z0, r0, #0
s7_an z0, r0, #1              // nop not necessary
s7_an z0, r0, #2
s7_an z0, r0, #3
nop                           // insert nop or independent command
store [r5],u4
```

3.2.52 s7_fn – EDGE-NEGATIVE (S7 command)

Note: S7 commands are not available on netX 10.

Sets the flags in S7 status word (register u4) depending on source 1 (a - byte of the operand), source 2 (b – bit position in a) and the initial S7 status word. Store the result bit in register (c). U4 register is hardwired for this command and represents the s7 status register with the following flags:

Bit position	S7 flag
usr4[0]	/ER
usr4[1]	VKE
usr4[2]	STA
usr4[3]	OR
usr4[4]	OS
usr4[5]	OV
usr4[6]	A0
usr4[7]	A1
usr4[8]	BIE

Operation

```
BIT = a[b]              // operand bit; a = byte; b = [0..7]
OR = 0; STA = 0; /ER = 1;
IF (BIT = 1)
    IF (VKE = 0)
        VKE = VKE xor 1;
        a[b] = 0;
    ELSE
        VKE = VKE xor 1;
        STA = 1;
ELSE
    OR = 0;    /ER = 1;
IF (VKE = 0)
        STA = 0;
    ELSE
        STA = 1;
        a[b] = 1;
```

Syntax

s7_fn reg, [pc + const], #0…7 // const = -2^{16} … 2^{16}-4 (dword aligned)

s7_fn reg, [const], #0…7 // const = 0 … 2^{17}-4 (dword aligned)

s7_fn reg, const, #0…7 // const = -2^{14} … 2^{14}-1

{[cond]} **s7_fn** reg, reg, #0…7

{[cond]} **s7_fn** reg, {s}[pc + wreg{++}]{b}, #0…7

{[cond]} **s7_fn** reg, {s}[wreg{++}]{b}, #0…7

{[cond]} conditional execution with the flags (not bypassed, i.e. from the next-to-last command)

{s} sign extension

{++} post-increment or pre-decrement (--wreg)

{b} b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

wreg r0 … r7 (not bypassed, i.e. from the next-to-last command)

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Excecute an S7 “EDGE-NEGATIVE” command of bit 2 and 3 from the byte in r0 (using S7 status word always located in u4). Store the result in r1 / r2:

```
s7_fn r1, r0, #2              // bit #2 of r0
s7_fn r2, r0, #3              // bit #3 of r0
```

Execute an S7 “EDGE-NEGATIVE” command of the first bit in byte value from address 0x1000 (using S7 status word always located in u4). Store the result in r6:

```
load r1, #0x1000
nop
s7_fn r6, [r1]b, #0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: inc u0, u1, #3
line_1: inc r4, r1, #15
line_2: [z] s7_fn r0, r4, #1    // condition [z] comes from line_0
                                // but r4 from line_1
```

Note: This command always uses register u4 as S7 status word. Register u4 is not bypassed. Bits 0 to 9 of this register will be interpreted and affected.

```
load u4, #13
nop                           // insert nop or independent command
s7_fn r4, r0, #0
s7_fn r5, r0, #1              // nop not necessary
s7_fn r6, r0, #2
s7_fn r7, r0, #3
nop                           // insert nop or independent command
store [r5],u4
```

3.2.53 s7_fp – EDGE-POSITIVE (S7 command)

Note: S7 commands are not available on netX 10.

Sets the flags in S7 status word (register u4) depending on source 1 (a - byte of the operand), source 2 (b – bit position in a) and the initial S7 status word. Store the result bit in register (c). U4 register is hardwired for this command and represents the s7 status register with the following flags:

Bit position	S7 flag
usr4[0]	/ER
usr4[1]	VKE
usr4[2]	STA
usr4[3]	OR
usr4[4]	OS
usr4[5]	OV
usr4[6]	A0
usr4[7]	A1
usr4[8]	BIE

Operation

```
BIT = a[b]              // operand bit; a = byte; b = [0..7]
IF (BIT = 1)
      OR = 0; STA = 0; /ER = 1;
      IF (VKE = 0)
            a[b] = 0;
      ELSE
            VKE = 0;
ELSE
      OR = 0;     /ER = 1;
IF (VKE = 0)
            STA = 0;
      ELSE
            STA = 1;
            a[b] = 1;
```

Syntax

s7_fp reg, [pc + const], #0…7 // const = -2^{16} … 2^{16}-4 (dword aligned)

s7_fp reg, [const], #0…7 // const = 0 … 2^{17}-4 (dword aligned)

s7_fp reg, const, #0…7 // const = -2^{14} … 2^{14}-1

{[cond]} **s7_fp** reg, reg, #0…7

{[cond]} **s7_fp** reg, {s}[pc + wreg{++}]{b}, #0…7

{[cond]} **s7_fp** reg, {s}[wreg{++}]{b}, #0…7

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

wreg r0 … r7 (not bypassed, i.e. from the next-to-last command)

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Excecute an S7 “EDGE-POSITIVE” command of bit 2 and 3 from the byte in r0 (using S7 status word is always located in u4). Store the result in r1 / r2:

```
S7_fp r1, r0, #2              // bit #2 of r0
S7_fp r2, r0, #3              // bit #3 of r0
```

Execute an S7 “EDGE-POSITIVE” command of the first bit in byte value from address 0x1000 (using S7 status word is always located in u4). Store the result in r6:

```
load r1, #0x1000
nop
S7_fp r6, [r1]b, #0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: inc u0, u1, #3
line_1: inc r4, r1, #15
line_2: [z] S7_fp r0, r4, #1    // condition [z] comes from line_0
                                // but r4 from line_1
```

Note: This command always uses register u4 as S7 status word. Register u4 is not bypassed. Bits 0 to 9 of this register will be interpreted and affected.

```
load u4, #13
nop                           // insert nop or independent command
S7_fp r4, r0, #0
S7_fp r5, r0, #1              // nop not necessary
S7_fp r6, r0, #2
S7_fp r7, r0, #3
nop                           // insert nop or independent command
store [r5],u4
```

3.2.54 s7_o – OR (S7 command)

Note: S7 commands are not available on netX 10.

Sets the flags in S7 status word (register u4) depending on source 1 (a - byte of the operand), source 2 (b – bit position in a) and the initial S7 status word. U4 register is hardwired for this command and represents the s7 status register with the following flags:

Bit position	S7 flag
usr4[0]	/ER
usr4[1]	VKE
usr4[2]	STA
usr4[3]	OR
usr4[4]	OS
usr4[5]	OV
usr4[6]	A0
usr4[7]	A1
usr4[8]	BIE

Operation

```
BIT = a[b]              // operand bit; a = byte; b = [0..7]
IF (BIT = 1)
    /ER = 1; STA = 1;
    IF (OR) = 0)
        VKE = 1;
ELSE
    /ER = 1;
    STA = 0;
IF (OR | /ER = 0)
        VKE = 0;
```

Syntax

s7_o z0, [pc + const], #0…7 // const = -2^{16} … 2^{16}-4 (dword aligned)

s7_o z0, [const], #0…7 // const = 0 … 2^{17}-4 (dword aligned)

s7_o z0, const, #0…7 // const = -2^{14} … 2^{14}-1

{[cond]} **s7_o** z0, reg, #0…7

{[cond]} **s7_o** z0, {s}[pc + wreg{++}]{b}, #0…7

{[cond]} **s7_o** z0, {s}[wreg{++}]{b}, #0…7

z0 select z0 for target register, the xpic moves source 1 to target

{[cond]} conditional execution with the flags (not bypassed, i.e. from the next-to-last command)

{s} sign extension

{++} post-increment or pre-decrement (--wreg)

{b} b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

wreg r0 … r7 (not bypassed, i.e. from the next-to-last command)

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Excecute an S7 “OR” command of bit 2 and 3 from the byte in r0 (using S7 status word always located in u4).

```
s7_o z0, r0, #2              // bit #2 of r0
s7_o z0, r0, #3              // bit #3 of r0
```

Execute an S7 “OR” command of the first bit in byte value from address 0x1000 (using S7 status word always located in u4):

```
load r1, #0x1000
nop
s7_o z0, [r1]b, #0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: inc u0, u1, #3
line_1: inc r4, r1, #15
line_2: [z] s7_o z0, r4, #1    // condition [z] comes from line_0
                               // but r4 from line_1
```

Note: This command always uses register u4 as S7 status word. Register u4 is not bypassed. Bits 0 to 9 of this register will be interpreted and affected.

```
load u4, #13
nop                          // insert nop or independent command
s7_o z0, r0, #0
s7_o z0, r0, #1              // nop not necessary
s7_o z0, r0, #2
s7_o z0, r0, #3
nop                          // insert nop or independent command
store [r5],u4
```

3.2.55 s7_on – OR-NOT (S7 command)

Note: S7 commands are not available on netX 10.

Sets the flags in S7 status word (register u4) depending on source 1 (a - byte of the operand), source 2 (b – bit position in a) and the initial S7 status word. U4 register is hardwired for this command and represents the s7 status register with the following flags:

Bit position	S7 flag
usr4[0]	/ER
usr4[1]	VKE
usr4[2]	STA
usr4[3]	OR
usr4[4]	OS
usr4[5]	OV
usr4[6]	A0
usr4[7]	A1
usr4[8]	BIE

Operation

```
BIT = a[b]              // operand bit; a = byte; b = [0..7]
OR = 0;
IF (BIT = 1)
      /ER = 1; STA = 1;
      IF ((OR | /ER) = 0)
            VKE = 0;
ELSE
      /ER = 1; STA = 0; VKE = 1;
```

Syntax

s7_on z0, [pc + const], #0…7 // const = - 2^{16} … 2^{16}-4 (dword aligned)

s7_on z0, [const], #0…7 // const = 0 … 2^{17}-4 (dword aligned)

s7_on z0, const, #0…7 // const = -2^{14} … 2^{14}-1

{[cond]} **s7_on** z0, reg, #0…7

{[cond]} **s7_on** z0, {s}[pc + wreg{++}]{b}, #0…7

{[cond]} **s7_on** z0, {s}[wreg{++}]{b}, #0…7

z0 select z0 for target register, the xpic moves source 1 to target

{[cond]} conditional execution with the flags (not bypassed, i.e. from the next-to-last command)

{s} sign extension

{++} post-increment or pre-decrement (--wreg)

{b} b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

wreg r0 … r7 (not bypassed, i.e. from the next-to-last command)

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Excecute an S7 “OR NOT” command of bit 2 and 3 from the byte in r0 (using S7 status word always located in u4).

```
s7_on z0, r0, #2              // bit #2 of r0
s7_on z0, r0, #3              // bit #3 of r0
```

Execute an S7 “OR NOT” command of the first bit in byte value from address 0x1000 (using S7 status word always located in u4):

```
load r1, #0x1000
nop
s7_on z0, [r1]b, #0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: inc u0, u1, #3
line_1: inc r4, r1, #15
line_2: [z] s7_on z0, r4, #1    // condition [z] comes from line_0
                                // but r4 from line_1
```

Note: This command always uses register u4 as S7 status word. Register u4 is not bypassed. Bits 0 to 9 of this register will be interpreted and affected.

```
load u4, #13
nop                           // insert nop or independent command
s7_on z0, r0, #0
s7_on z0, r0, #1               // nop not necessary
s7_on z0, r0, #2
s7_on z0, r0, #3
nop                           // insert nop or independent command
store [r5],u4
```

3.2.56 s7_x – EXCLUSIVE-OR (S7 command)

Note: S7 commands are not available on netX 10.

Sets the flags in S7 status word (register u4) depending on source 1 (a - byte of the operand), source 2 (b – bit position in a) and the initial S7 status word. U4 register is hardwired for this command and represents the s7 status register with the following flags:

Bit position	S7 flag
usr4[0]	/ER
usr4[1]	VKE
usr4[2]	STA
usr4[3]	OR
usr4[4]	OS
usr4[5]	OV
usr4[6]	A0
usr4[7]	A1
usr4[8]	BIE

Operation

```
BIT = a[b]              // operand bit; a = byte; b = [0..7]
IF (BIT = 1)
      /ER = 1; STA = 1;
      IF ((OR | /ER) = 0)
            VKE = 1;
      ELSE
            VKE = VKE xor 1;
ELSE
      /ER = 1; STA = 0;
IF (OR | /ER = 0)
            VKE = 0;
```

Syntax

s7_x z0, [pc + const], #0…7 // const = -2^{16} … 2^{16}-4 (dword aligned)

s7_x z0, [const], #0…7 // const = 0 … 2^{17}-4 (dword aligned)

s7_x z0, const, #0…7 // const = -2^{14} … 2^{14}-1

{[cond]} **s7_x** z0, reg, #0…7

{[cond]} **s7_x** z0, {s}[pc + wreg{++}]{b}, #0…7

{[cond]} **s7_x** z0, {s}[wreg{++}]{b}, #0…7

z0 select z0 for target register, the xpic moves source 1 to target

{[cond]} conditional execution with the flags (not bypassed, i.e. from the next-to-last command)

{s} sign extension

{++} post-increment or pre-decrement (--wreg)

{b} b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

wreg r0 … r7 (not bypassed, i.e. from the next-to-last command)

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Excecute an S7 “EXCLUSIVE-OR” command of bit 2 and 3 from the byte in r0 (using S7 status word always located in u4).

```
s7_x z0, r0, #2              // bit #2 of r0
s7_x z0, r0, #3              // bit #3 of r0
```

Execute an S7 “EXCLUSIVE-OR” command of the first bit in byte value from address 0x1000 (using S7 status word always located in u4):

```
load r1, #0x1000
nop
s7_x z0, [r1]b, #0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: inc u0, u1, #3
line_1: inc r4, r1, #15
line_2: [z] s7_x z0, r4, #1    // condition [z] comes from line_0
                               // but r4 from line_1
```

Note: This command always uses register u4 as S7 status word. Register u4 is not bypassed. Bits 0 to 9 of this register will be interpreted and affected.

```
load u4, #13
nop                          // insert nop or independent command
s7_x z0, r0, #0
s7_x z0, r0, #1              // nop not necessary
s7_x z0, r0, #2
s7_x z0, r0, #3
nop                          // insert nop or independent command
store [r5],u4
```

3.2.57 s7_xn – EXCLUSIVE-OR-NOT (S7 command)

Note: S7 commands are not available on netX 10.

Sets the flags in S7 status word (register u4) depending on source 1 (a - byte of the operand), source 2 (b – bit position in a) and the initial S7 status word. U4 register is hardwired for this command and represents the s7 status register with the following flags:

Bit position	S7 flag
usr4[0]	/ER
usr4[1]	VKE
usr4[2]	STA
usr4[3]	OR
usr4[4]	OS
usr4[5]	OV
usr4[6]	A0
usr4[7]	A1
usr4[8]	BIE

Operation

```
BIT = a[b]              // operand bit; a = byte; b = [0..7]
IF (BIT = 1)
      /ER = 1; STA = 1;
      IF ((OR | /ER) = 0)
            VKE = 0;
ELSE
      /ER = 1; STA = 0;
IF (OR | /ER = 1)
            VKE = VKE xor 1;
```

Syntax

s7_ **xn** z0, [pc + const], #0…7 // const = -2^{16} … 2^{16}-4 (dword aligned)

s7_ **xn** z0, [const], #0…7 // const = 0 … 2^{17}-4 (dword aligned)

s7_ **xn** z0, const, #0…7 // const = -2^{14} … 2^{14}-1

{[cond]} **s7**_ **xn** z0, reg, #0…7

{[cond]} **s7**_ **xn** z0, {s}[pc + wreg{++}]{b}, #0…7

{[cond]} **s7**_ **xn** z0, {s}[wreg{++}]{b}, #0…7

z0 select z0 for target register, the xpic moves source 1 to target

{[cond]} conditional execution with the flags (not bypassed, i.e. from the next-to-last command)

{s} sign extension

{++} post-increment or pre-decrement (--wreg)

{b} b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access

reg r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)

wreg r0 … r7 (not bypassed, i.e. from the next-to-last command)

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2
execution	-	-	-	-	-	-	-	-	-	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Excecute an S7 “EXCLUSIVE-OR-NOT” command of bit 2 and 3 from the byte in r0 (using S7 status word always located in u4).

```
s7_xn z0, r0, #2              // bit #2 of r0
s7_xn z0, r0, #3              // bit #3 of r0
```

Execute an S7 “EXCLUSIVE-OR-NOT” command of the first bit in byte value from address 0x1000 (using S7 status word always located in u4):

```
load r1, #0x1000
nop
s7_xn z0, [r1]b, #0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: inc u0, u1, #3
line_1: inc r4, r1, #15
line_2: [z] s7_xn z0, r4, #1   // condition [z] comes from line_0
                               // but r4 from line_1
```

Note: This command always uses register u4 as S7 status word. Register u4 is not bypassed. Bits 0 to 9 of this register will be interpreted and affected.

```
load u4, #13
nop                           // insert nop or independent command
s7_xn z0, r0, #0
s7_xn z0, r0, #1              // nop not necessary
s7_xn z0, r0, #2
s7_xn z0, r0, #3
nop                           // insert nop or independent command
store [r5],u4
```

3.2.58 xnor - bitwise XNOR

Bitwise XNOR operation with target (c), source 1 (a) and source 2 (b).

a	b	c
0	0	1
0	1	0
1	0	0
1	1	1

Operation

c := ~((a | b) & (~a | ~b))

Syntax

xnor reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

xnor reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

xnor reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **xnor** reg, reg, reg

{[cond]} **xnor** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **xnor** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z	c	s	v	e	gu	gs	geu	ges	r0z	r1z	r2z	r3z	r0123z
	nz	nc	ns	nv	ne	leu	les	lu	ls	r0nz	r1nz	r2nz	r3nz	r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set and valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Perform a bitwise XNOR operation with the value of work register 0 and the value of user register 1. Save the result in work register 4:

```
xnor r4, r0, u1
```

Perform a bitwise XNOR operation with the sign extended byte value of address 0x1000 and the value of work register 0. Save the result in work register 6:

```
load r1, #0x1000
nop
xnor r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: xnor u0, u1, u2
line_1: xnor r5, r1, r2
line_2: [z] xnor r3, r4, r5      // condition [z] comes from line_0
                                 // but r5 from line_1
```

3.2.59 xor - bitwise XOR

Bitwise XOR operation with target (c), source 1 (a) and source 2 (b).

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

Operation

c := (a | b) & (~a | ~b)

Syntax

xor reg, [pc + const], reg // const = -2^{16} … 2^{16}-4 (dword aligned)

xor reg, [const], reg // const = 0 … 2^{17}-4 (dword aligned)

xor reg, const, reg // const = -2^{14} … 2^{14}-1

{[cond]} **xor** reg, reg, reg

{[cond]} **xor** reg, {s}[pc + wreg{++}]{b}, reg

{[cond]} **xor** reg, {s}[wreg{++}]{b}, reg

{[cond]} *conditional execution with the flags (not bypassed, i.e. from the next-to-last command)*

{s} *sign extension*

{++} *post-increment or pre-decrement (--wreg)*

{b} *b, w, d – byte (8-bit), word (16-bit) or dword (32-bit) access*

reg *r0 … r7, u0 … u4, pc, st, z0 (bypassed, i.e. from last command)*

wreg *r0 … r7 (not bypassed, i.e. from the next-to-last command)*

Flags

set for conditional	z nz	c nc	s ns	v nv	e ne	gu leu	gs les	geu lu	ges ls	r0z r0nz	r1z r1nz	r2z r2nz	r3z r3nz	r0123z r0123nz
jump	+1	-	+1	-	+1	+1	+1	+1	+1	+2	+2	+2	+2	+2
execution	+2	-	+2	-	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

(+1) set a nd valid for next command; (+2) set and valid in the command after next; (-) unaffected; (u) undefined

Examples

Perform a bitwise XOR operation with the value of work register 0 and the value of user register 1. Save the result in work register 4:

```
xor r4, r0, u1              // r4 := r0 ^ u1
```

Perform a bitwise XOR operation with the sign extended byte value of address 0x1000 and the value of work register 0. Save the result in work register 6:

```
load r1, #0x1000
nop
xor r6, s[r1]b, r0
```

Note: None of the flags (conditions) for the conditional execution is bypassed. The flag (condition) from the next-to-last command is valid for the conditional execution:

```
line_0: xor u0, u1, u2
line_1: xor r5, r1, r2
line_2: [z] xor r3, r4, r5    // condition [z] comes from line_0
                              // but r5 from line_1
```

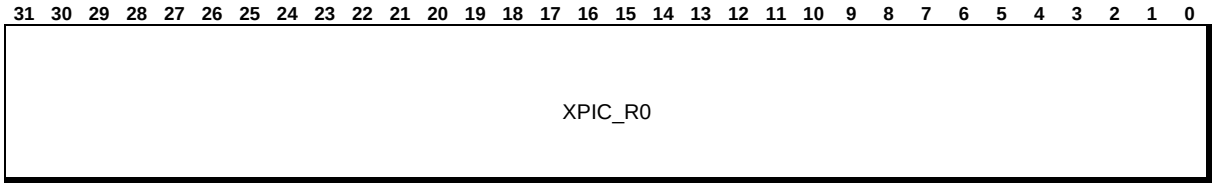
4 Software interface

4.1 xPIC register list

Register name	Brief description
XPIC_R0	xPIC work register for indirect addressing
XPIC_R1	xPIC work register for indirect addressing
XPIC_R2	xPIC work register for indirect addressing
XPIC_R3	xPIC work register for indirect addressing
XPIC_R4	xPIC work register for indirect addressing
XPIC_R5	xPIC work register for indirect addressing
XPIC_R6	xPIC work register for indirect addressing
XPIC_R7	xPIC work register for indirect addressing
XPIC_USR0	xPIC user register additional work register
XPIC_USR1	xPIC user register additional work register
XPIC_USR2	xPIC user register additional work register
XPIC_USR3	xPIC user register additional work register
XPIC_USR4	xPIC user register additional work register
XPIC_PC	xPIC program counter
XPIC_STAT	Processor status register
XPIC_ZERO	Zero register
XPIC_IRQ_SATUS	Read access shows xpic irq status and xpic irq enable bits

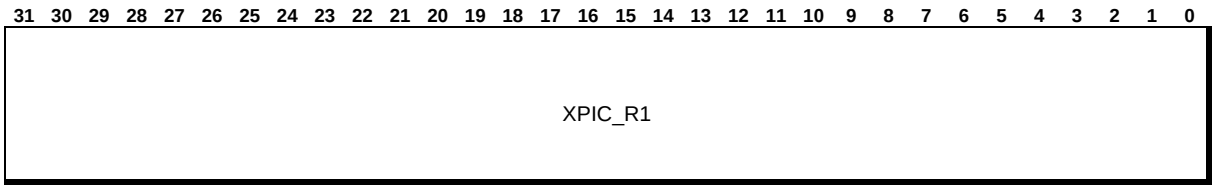
Table 17: xPIC register list

XPIC_R0 – xPIC work register for indirect addressing



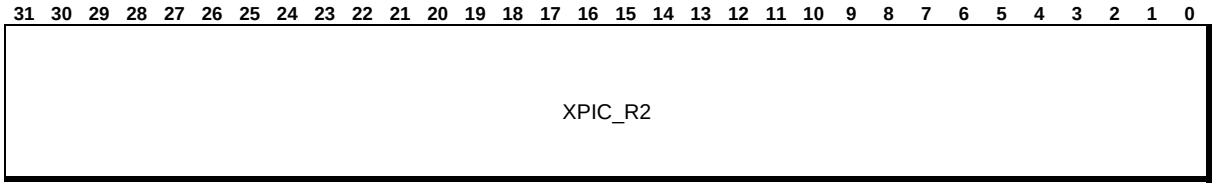
Bits	Name	Description	R/W	Default
31:0	r0	Work register 0	R/W	0x0

XPIC_R1 – xPIC work register for indirect addressing



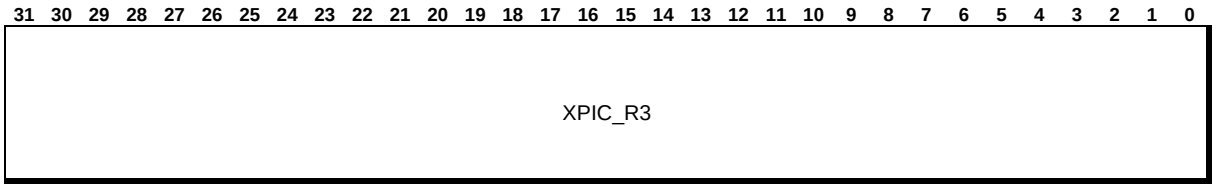
Bits	Name	Description	R/W	Default
31:0	r1	Work register 1	R/W	0x0

XPIC_R2 – xPIC work register for indirect addressing



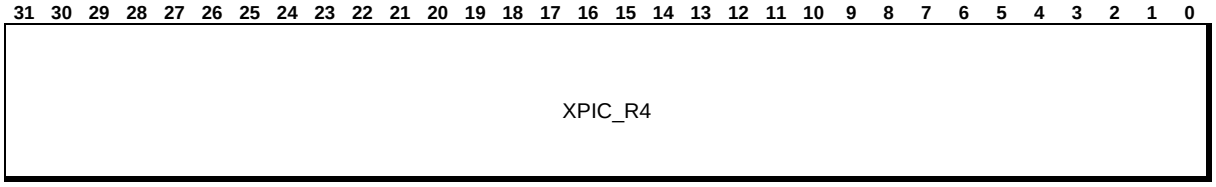
Bits	Name	Description	R/W	Default
31:0	r2	Work register 2	R/W	0x0

XPIC_R3 – xPIC work register for indirect addressing



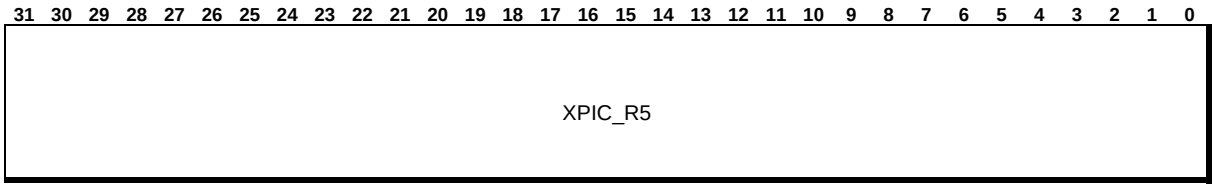
Bits	Name	Description	R/W	Default
31:0	r3	Work register 3	R/W	0x0

XPIC_R4 – xPIC work register for indirect addressing



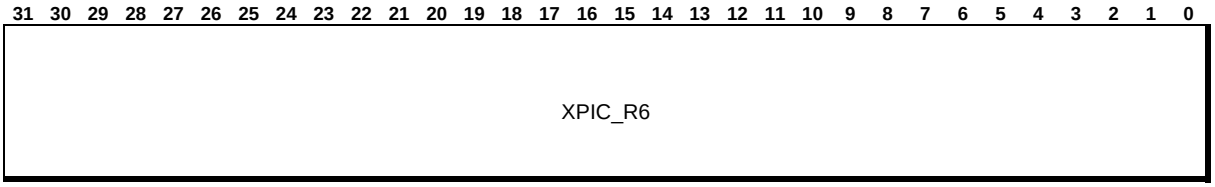
Bits	Name	Description	R/W	Default
31:0	r4	Work register 4	R/W	0x0

XPIC_R5 – xPIC work register for indirect addressing



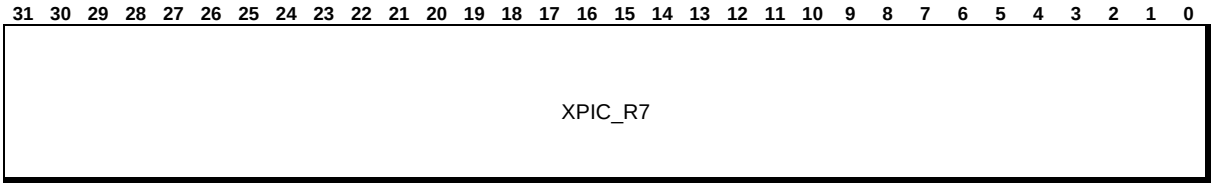
Bits	Name	Description	R/W	Default
31:0	r5	Work register 5	R/W	0x0

XPIC_R6 – xPIC work register for indirect addressing



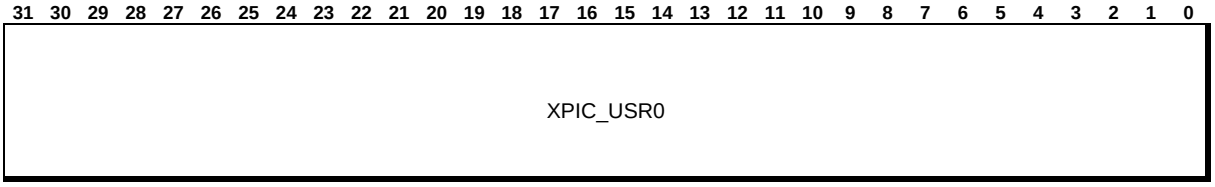
Bits	Name	Description	R/W	Default
31:0	r6	Work register 6	R/W	0x0

XPIC_R7 – xPIC work register for indirect addressing



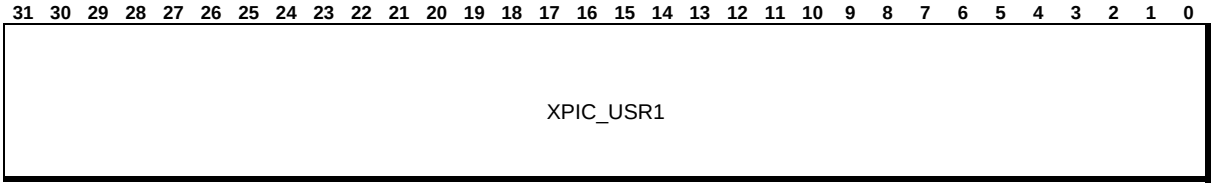
Bits	Name	Description	R/W	Default
31:0	r7	Work register 7	R/W	0x0

XPIC_USR0 – xPIC user register additional work register



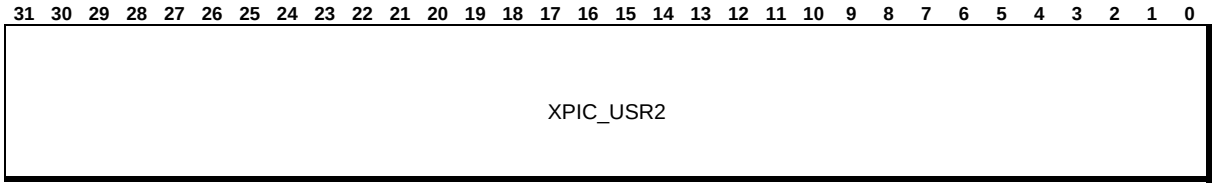
Bits	Name	Description	R/W	Default
31:0	u0	User register 0	R/W	0x0

XPIC_USR1 – xPIC user register additional work register



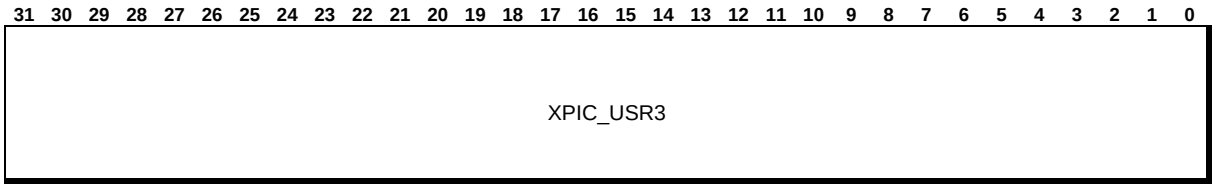
Bits	Name	Description	R/W	Default
31:0	u1	User register 1	R/W	0x0

XPIC_USR2 – xPIC user register additional work register



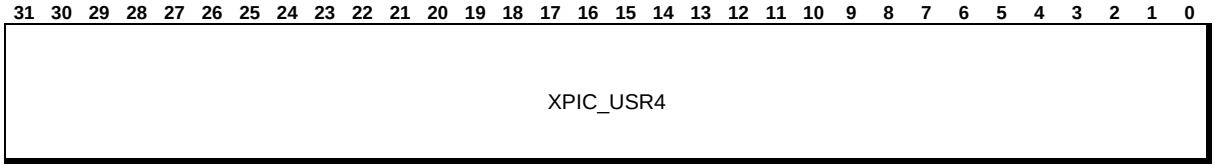
Bits	Name	Description	R/W	Default
31:0	u2	User register 2	R/W	0x0

XPIC_USR3 – xPIC user register additional work register



Bits	Name	Description	R/W	Default
31:0	u3	User register 3	R/W	0x0

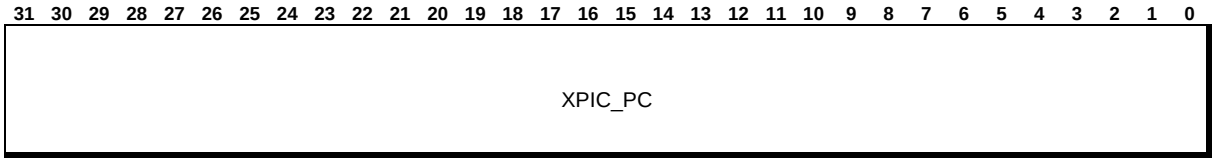
XPIC_USR4 – xPIC user register additional work register



Bits	Name	Description	R/W	Default
31:0	u4	User register 4	R/W	0x0

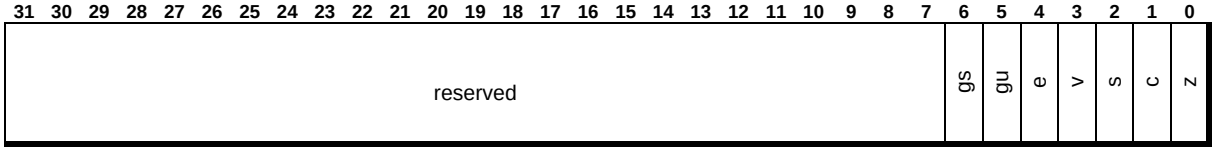
XPIC_PC – xPIC program counter

Shared in xPIC 64_BIT_MUL_TARGET mode with u32 (w mode)



Bits	Name	Description	R/W	Default
31:0	pc	Program counter	R/W	0xfffffffc

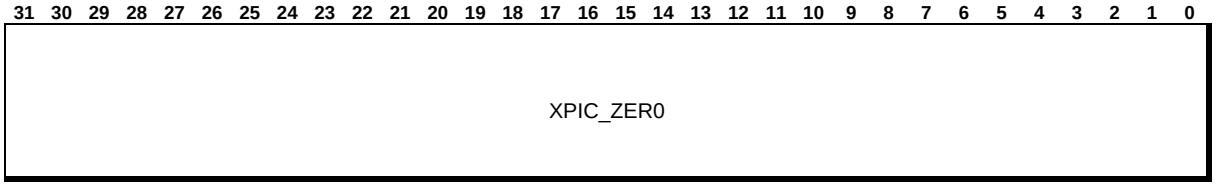
XPIC_STAT – processor status register



Bits	Name	Description	R/W	Default
31:7	reserved	Reserved	R/W	0x0
6	gs	Greater signed	R/W	0x0
5	gu	Greater unsigned	R/W	0x0
4	e	Equal	R/W	0x0
3	v	Overflow	R/W	0x0
2	s	Sign	R/W	0x0
1	c	Carry	R/W	0x0
0	z	Zero	R/W	0x0

XPIC_ZERO – zero register

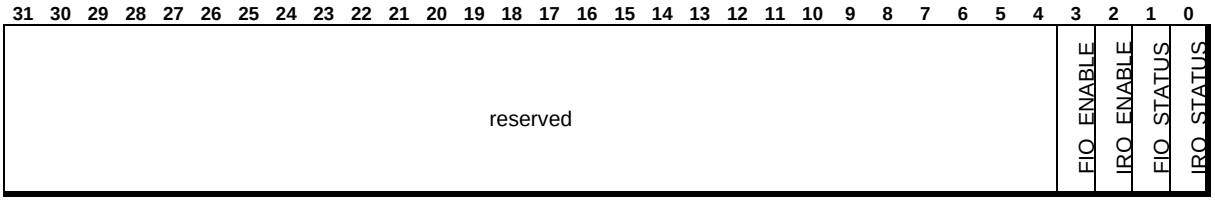
Shared in xPIC 64_BIT_MUL_TARGET mode with u10 (w mode)



Bits	Name	Description	R/W	Default
31:0	z0	Always zero	R/W	0x0

XPIC_IRQ_STATUS

Note: Register XPIC_IRQ_STATUS is not available on netX 10.



Bits	Name	Description	R/W	Default
31:4	-	reserved	R	0x0
3	FIQ_ENABLE	FIQ enable bit	R	0x0
2	IRQ_ENABLE	IRQ enable bit	R	0x0
1	FIQ_STATUS	FIQ status	R	0x0
0	IRQ_STATUS	IRQ status	R	0x0

4.2 XPIC_DEBUG register list

Register name	Brief description
XPIC_HOLD_PC	
XPIC_BREAK0_ADDR	
XPIC_BREAK0_ADDR_MASK	
XPIC_BREAK0_DATA	
XPIC_BREAK0_DATA_MASK	
XPIC_BREAK0_CONTR	
XPIC_BREAK0_CONTR_MASK	
XPIC_BREAK1_ADDR	
XPIC_BREAK1_ADDR_MASK	
XPIC_BREAK1_DATA	
XPIC_BREAK1_DATA_MASK	
XPIC_BREAK1_CONTR	
XPIC_BREAK1_CONTR_MASK	
XPIC_BREAK_LAST_PC	
XPIC_BREAK_STATUS	Read access shows why xPIC is in HOLD / BREAK
XPIC_BREAK_IRQ_RAW	xPIC_DEBUG raw IRQ register
XPIC_BREAK_IRQ_MASKED	xPIC_DEBUG masked IRQ register: for other CPU (ARM)
XPIC_BREAK_IRQ_MSK_SET	xPIC_DEBUG interrupt mask enable: for other CPU (ARM)
XPIC_BREAK_IRQ_MSK_RESET	xPIC_DEBUG interrupt mask disable: for other CPU (ARM)
XPIC_BREAK_OWN_IRQ_MASKED	xPIC_DEBUG own masked IRQ register: for xPIC
XPIC_BREAK_OWN_IRQ_MSK_SET	xPIC_DEBUG own interrupt mask enable: for xPIC
XPIC_BREAK_OWN_IRQ_MSK_RESET	xPIC_DEBUG own interrupt mask disable: for XPIC
XPIC_BREAK_RETURN_FIQ_PC	xPIC_DEBUG information FIQ return PC value
XPIC_BREAK_RETURN_IRQ_PC	xPIC_DEBUG information last IRQ return PC value

Table 18: XPIC_DEBUG register list

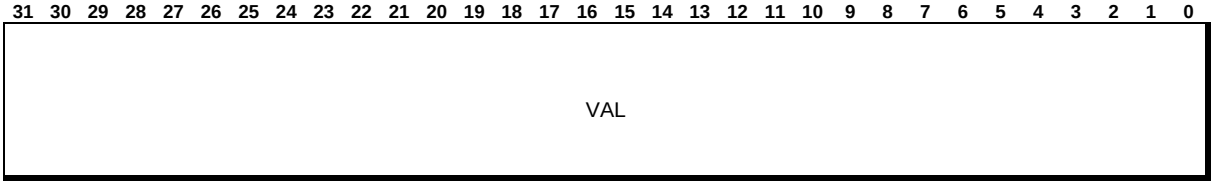
XPIC_HOLD_PC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved																								RESET_XPIC	BANK_CONTROL	BANK_SELECT	MISALIGNMENT_HOLD	DISABLE_INT	MONITOR_MODE	SINGLE_STEP	HOLD

Bits	Name	Description	R/W	Default
31:8	-	reserved	R	0x0
7	RESET_XPIC	REQUEST reset all internal states and the pipeline EXCEPT: the internal register (r0-r7, u0-4), bank0 and bank1, reset these registers manually EXCEPT: xPIC hard_breaker/debug registers 1 - xPIC reset request	R/W	0x0
6	BANK_CONTROL	control over the register bank selection WARNING: Reset this BIT to 0 BEFORE start xPIC (clear hold bits)	R/W	0x0

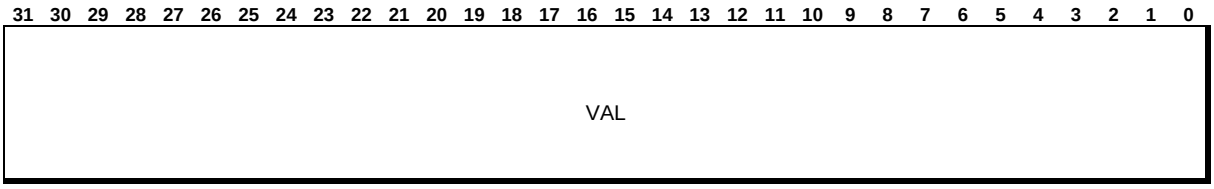
Bits	Name	Description	R/W	Default
5	BANK_SELECT	select register bank 0 or 1 if bank_control=1 (r0-r7; st)	R/W	0x0
4	MISALIGNMENT_HOLD	hold xPIC on misalignment	R/W	0x0
3	DISABLE_INT	disable interrupts	R/W	0x0
2	MONITOR_MODE	do not hold xPIC on break0 and break1, only monitor	R/W	0x0
1	SINGLE_STEP	single_step mode	R/W	0x0
0	HOLD	0: start xPIC, 1: hold xPIC	R/W	0x1

XPIC_BREAK0_ADDR



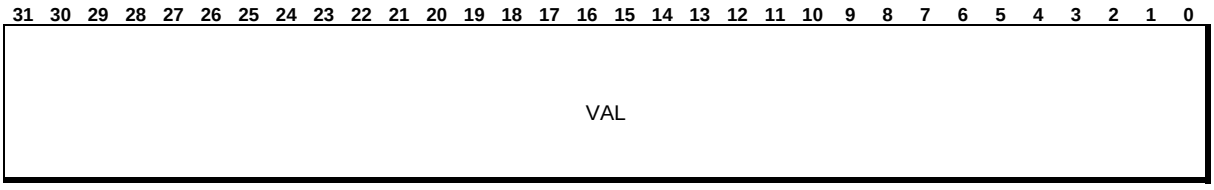
Bits	Name	Description	R/W	Default
31:0	VAL	Breakpoint 0 address value	R/W	0x0

XPIC_BREAK0_ADDR_MASK



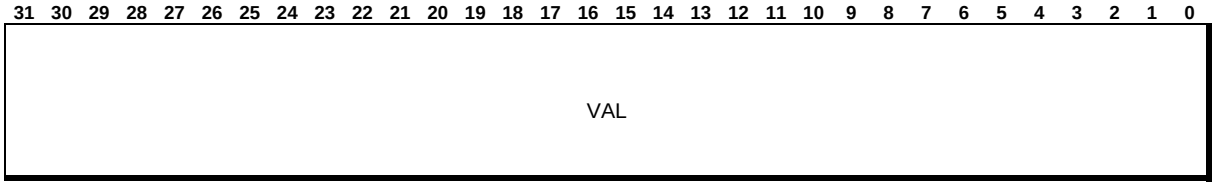
Bits	Name	Description	R/W	Default
31:0	VAL	Breakpoint 0 address mask	R/W	0x0

XPIC_BREAK0_DATA



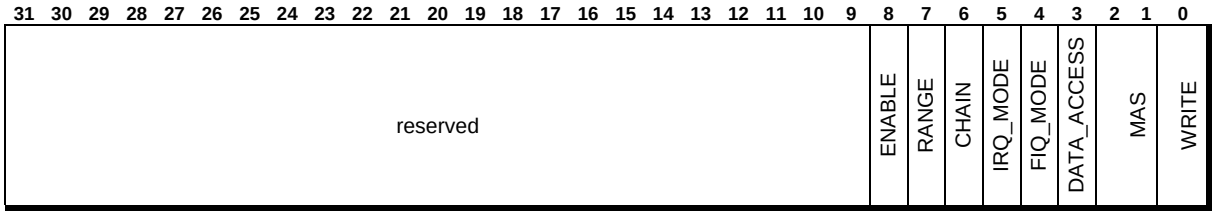
Bits	Name	Description	R/W	Default
31:0	VAL	Breakpoint 0 data value (for data access only)	R/W	0x0

XPIC_BREAK0_DATA_MASK



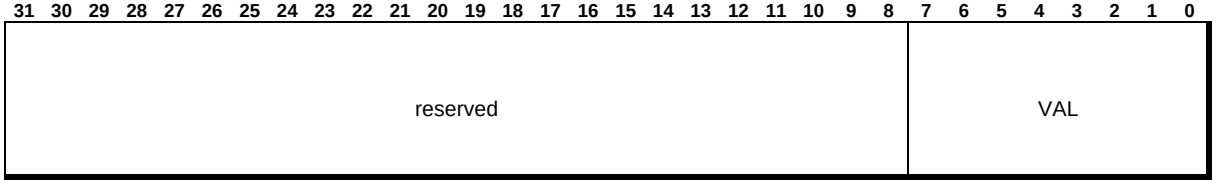
Bits	Name	Description	R/W	Default
31:0	VAL	Breakpoint 0 data mask (for data access only)	R/W	0x0

XPIC_BREAK0_CONTR



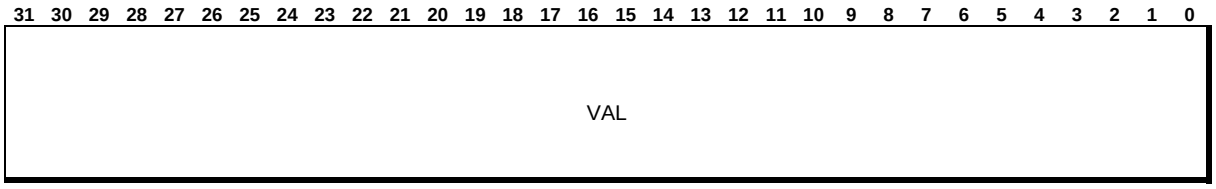
Bits	Name	Description	R/W	Default
31:9	-	reserved	R	0x0
8	ENABLE	Breakpoint 0	R/W	0x0
7	RANGE	Breakpoint 0 input from breakpoint 1	R/W	0x0
6	CHAIN	Breakpoint 0 input from breakpoint 1	R/W	0x0
5	IRQ_MODE	Breakpoint 0 xPIC in IRQ mode	R/W	0x0
4	FIQ_MODE	Breakpoint 0 xPIC in FIQ mode	R/W	0x0
3	DATA_ACCESS	Breakpoint 0 (1: data access, 0: instruction fetch)	R/W	0x0
2:1	MAS	Breakpoint 0 memory access size (00: byte, 01: word, 10: dword, 11: reserved)	R/W	0x0
0	WRITE	Breakpoint 0 write/read access	R/W	0x0

XPIC_BREAK0_CONTR_MASK



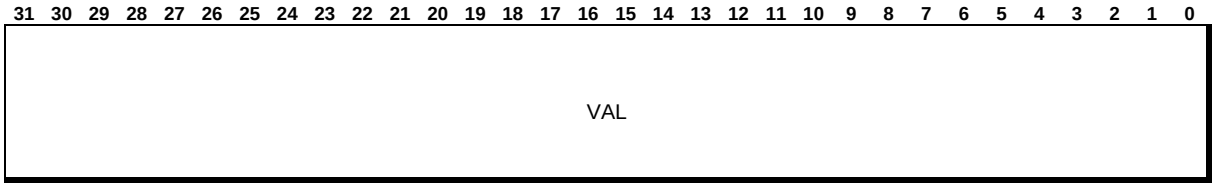
Bits	Name	Description	R/W	Default
31:8	-	reserved	R	0x0
7:0	VAL	Breakpoint 0 control mask	R/W	0x0

XPIC_BREAK1_ADDR



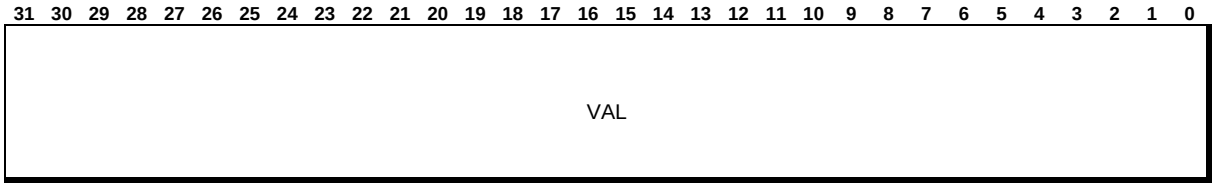
Bits	Name	Description	R/W	Default
31:0	VAL	Breakpoint 1 address value	R/W	0x0

XPIC_BREAK1_ADDR_MASK



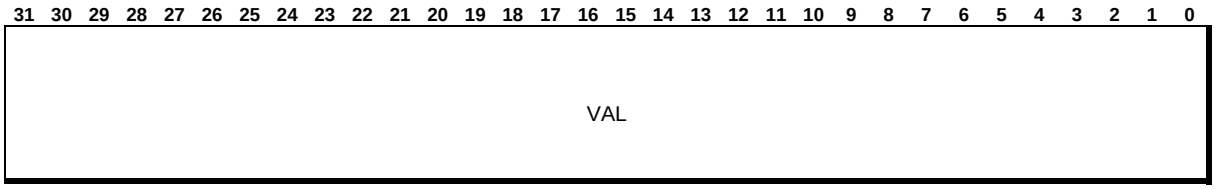
Bits	Name	Description	R/W	Default
31:0	VAL	Breakpoint 1 address mask	R/W	0x0

XPIC_BREAK1_DATA



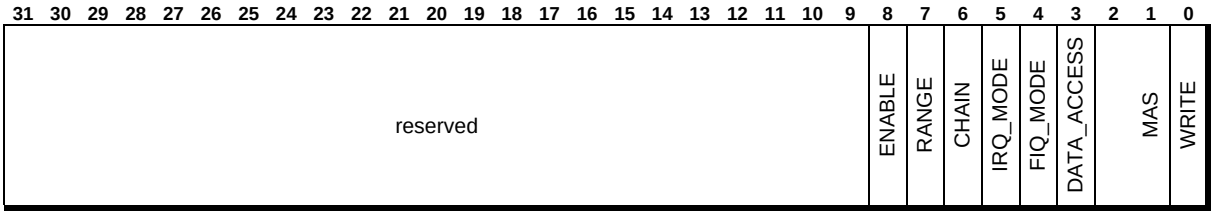
Bits	Name	Description	R/W	Default
31:0	VAL	Breakpoint 1 data value (for data access only)	R/W	0x0

XPIC_BREAK1_DATA_MASK



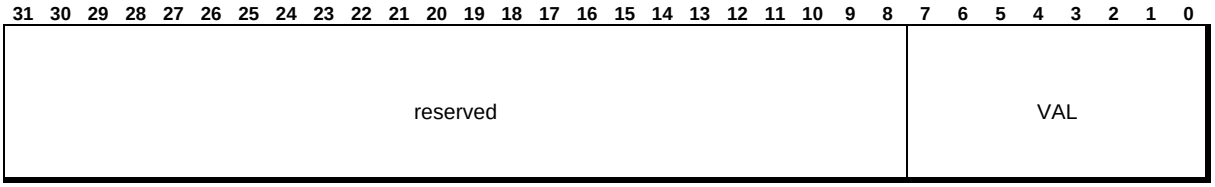
Bits	Name	Description	R/W	Default
31:0	VAL	Breakpoint 1 data mask (for data access only)	R/W	0x0

XPIC_BREAK1_CONTR



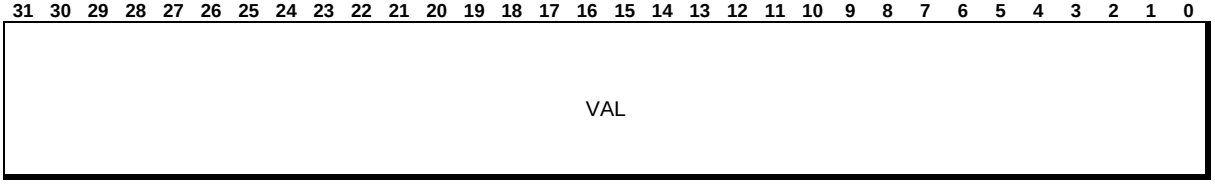
Bits	Name	Description	R/W	Default
31:9	-	reserved	R	0x0
8	ENABLE	Breakpoint 1	R/W	0x0
7	RANGE	reserved	R/W	0x0
6	CHAIN	reserved	R/W	0x0
5	IRQ_MODE	Breakpoint 1 xPIC in IRQ mode	R/W	0x0
4	FIQ_MODE	Breakpoint 1 xPIC in FIQ mode	R/W	0x0
3	DATA_ACCESS	Breakpoint 1 (1: data access, 0: instruction fetch)	R/W	0x0
2:1	MAS	Breakpoint 1 memory access size (00: byte, 01: word, 10: dword, 11: reserved)	R/W	0x0
0	WRITE	Breakpoint 1 write/read access	R/W	0x0

XPIC_BREAK1_CONTR_MASK



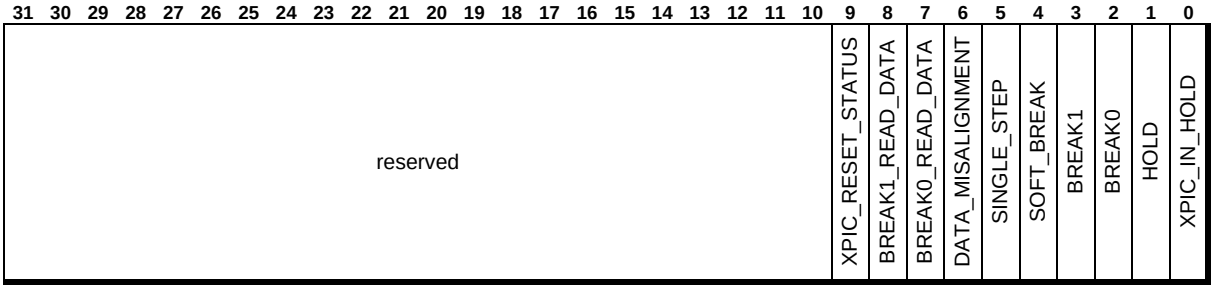
Bits	Name	Description	R/W	Default
31:8	-	reserved	R	0x0
7:0	VAL	Breakpoint 1 control mask	R/W	0x0

XPIC_BREAK_LAST_PC



Bits	Name	Description	R/W	Default
31:0	VAL	last PC	R	0x0

XPIC_BREAK_STATUS – read access shows why xPIC is in HOLD/BREAK

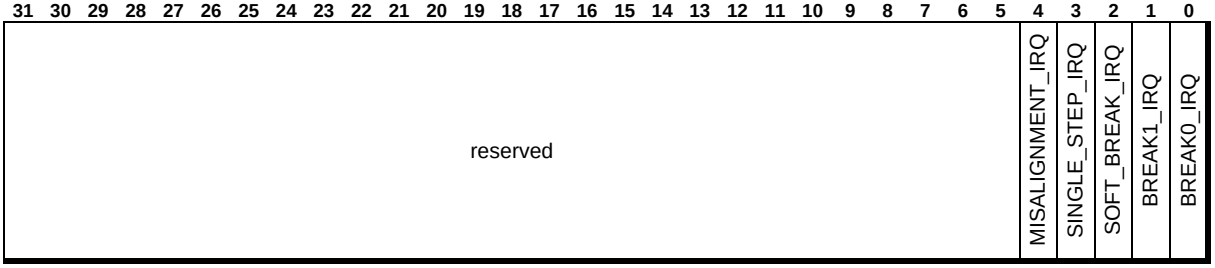


Bits	Name	Description	R/W	Default
31:10	-	reserved	R	0x0
9	XPIC_RESET_STATUS	1 = xPIC ist in reset	R	0x0
8	BREAK1_READ_DATA	Breakpoint 1 last load access	R	0x0
7	BREAK0_READ_DATA	Breakpoint 0 last load access	R	0x0
6	DATA_MISALIGNMENT	Data misalignment is active	R	0x0
5	SINGLE_STEP	single-step break is active	R	0x0
4	SOFT_BREAK	Software break is active	R	0x0
3	BREAK1	Breakpoint 1 is active	R	0x0
2	BREAK0	Breakpoint 0 is active	R	0x0
1	HOLD	global HOLD BIT status 0: start xPIC, 1: hold xPIC	R	0x0
0	XPIC_IN_HOLD	xPIC is in break or hold	R	0x0

XPIC_BREAK_IRQ_RAW – xPIC_DEBUG raw IRQ register

Read access shows status of unmasked IRQs.

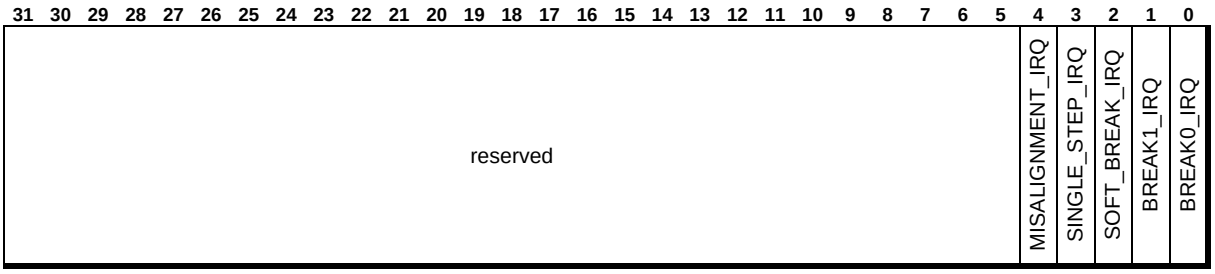
Write access with ’1’ resets the appropriate IRQ and clears the HOLD reason.



Bits	Name	Description	R/W	Default
31:5	-	reserved	R	0x0
4	MISALIGNMENT_IRQ	Data misalignment error interrupt	R/W	0x0
3	SINGLE_STEP_IRQ	single-step breakpoint interrupt	R/W	0x0
2	SOFT_BREAK_IRQ	Software breakpoint interrupt	R/W	0x0
1	BREAK1_IRQ	Breakpoint 1 interrupt	R/W	0x0
0	BREAK0_IRQ	Breakpoint 0 interrupt	R/W	0x0

XPIC_BREAK_IRQ_MASKED – xPIC_DEBUG masked IRQ register: for other CPU (ARM)

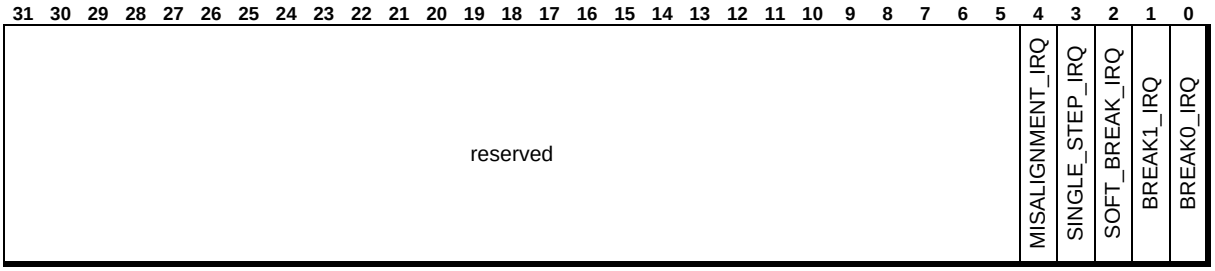
Shows status of masked IRQs (as connected to ARM)



Bits	Name	Description	R/W	Default
31:5	-	reserved	R	0x0
4	MISALIGNMENT_IRQ	Data misalignment error interrupt	R	0x0
3	SINGLE_STEP_IRQ	single-step breakpoint interrupt	R	0x0
2	SOFT_BREAK_IRQ	Software breakpoint interrupt	R	0x0
1	BREAK1_IRQ	Breakpoint 1 interrupt	R	0x0
0	BREAK0_IRQ	Breakpoint 0 interrupt	R	0x0

XPIC_BREAK_IRQ_MSK_SET – xPIC_DEBUG interrupt mask enable: for other CPU (ARM)

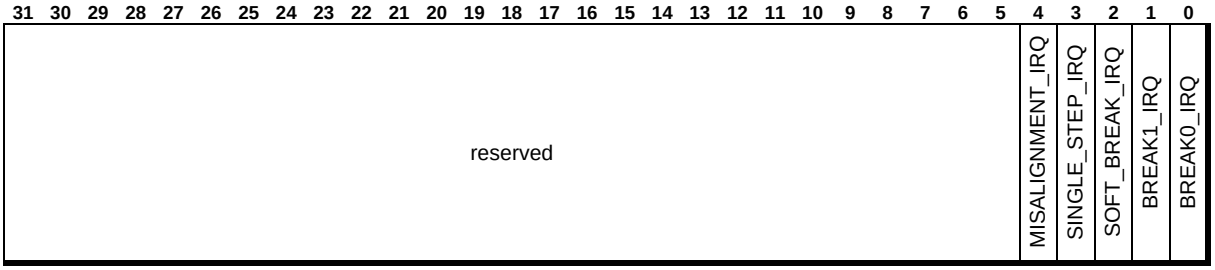
Write access with ’1’ sets interrupt mask bit (enables IRQ for corresponding interrupt source).
Write access with ’0’ does not influence this bit. Read access shows actual interrupt mask.



Bits	Name	Description	R/W	Default
31:5	-	reserved	R	0x0
4	MISALIGNMENT_IRQ	Data misalignment error interrupt	R/W	0x0
3	SINGLE_STEP_IRQ	single-step breakpoint interrupt	R/W	0x0
2	SOFT_BREAK_IRQ	Software breakpoint interrupt	R/W	0x0
1	BREAK1_IRQ	Breakpoint 1 interrupt	R/W	0x0
0	BREAK0_IRQ	Breakpoint 0 interrupt	R/W	0x0

XPIC_BREAK_IRQ_MSK_RESET – xPIC_DEBUG interrupt mask disable: for other CPU (ARM)

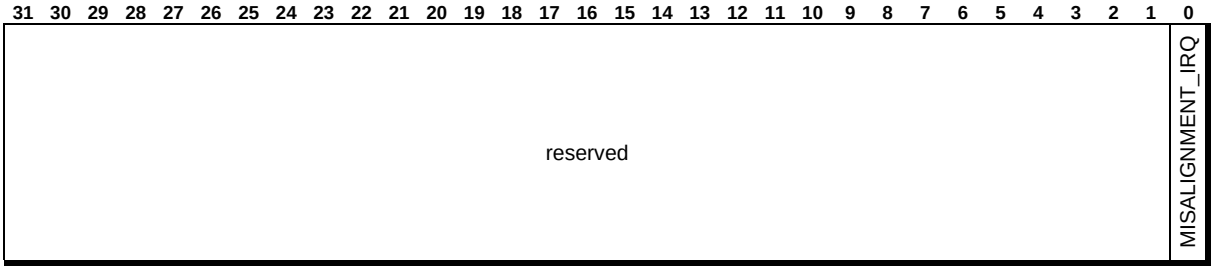
Write access with ’1’ resets interrupt mask bit (disables IRQ for corresponding interrupt source).
Write access with ’0’ does not influence this bit. Read access shows actual interrupt mask.



Bits	Name	Description	R/W	Default
31:5	-	reserved	R	0x0
4	MISALIGNMENT_IRQ	Data misalignment error interrupt	R/W	0x0
3	SINGLE_STEP_IRQ	Single-step breakpoint interrupt	R/W	0x0
2	SOFT_BREAK_IRQ	Software breakpoint interrupt	R/W	0x0
1	BREAK1_IRQ	Breakpoint 1 interrupt	R/W	0x0
0	BREAK0_IRQ	Breakpoint 0 interrupt	R/W	0x0

XPIC_BREAK_OWN_IRQ_MASKED – xPIC_DEBUG own masked IRQ register: for xPIC

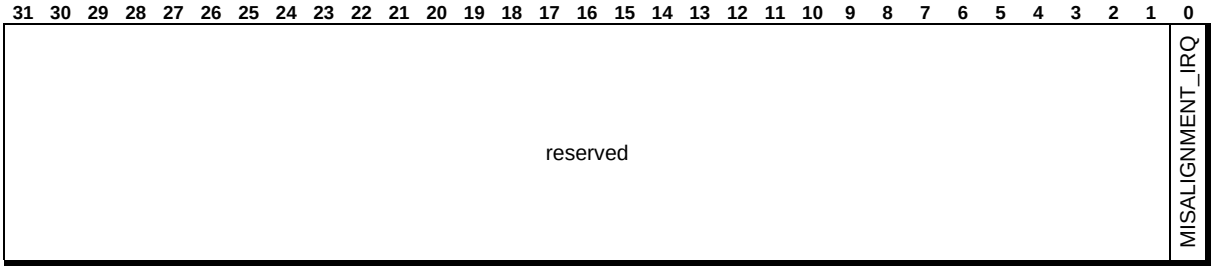
Shows status of masked IRQs (as connected to xPIC)



Bits	Name	Description	R/W	Default
31:1	-	reserved	R	0x0
0	MISALIGNMENT_IRQ	Data misalignment error interrupt	R	0x0

XPIC_BREAK_OWN_IRQ_MSK_SET – xPIC_DEBUG own interrupt mask enable: for xPIC

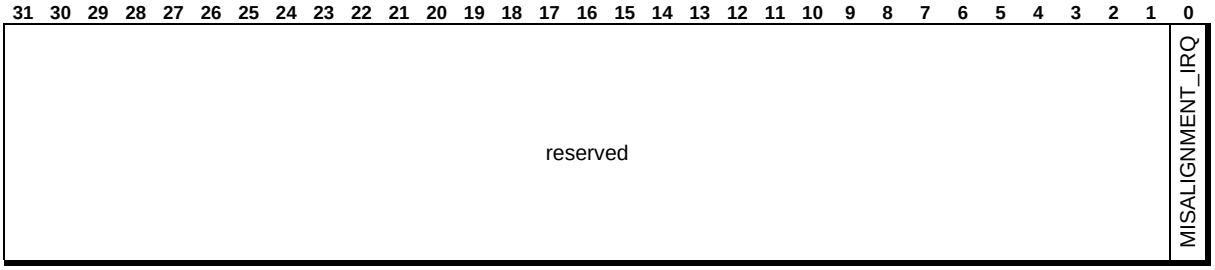
Write access with ’1’ sets interrupt mask bit (enables IRQ for corresponding interrupt source).
Write access with ’0’ does not influence this bit. Read access shows actual interrupt mask.



Bits	Name	Description	R/W	Default
31:1	-	reserved	R	0x0
0	MISALIGNMENT_IRQ	Data misalignment error interrupt	R/W	0x0

XPIC_BREAK_OWN_IRQ_MSK_RESET – xPIC_DEBUG own interrupt mask disable: for xPIC

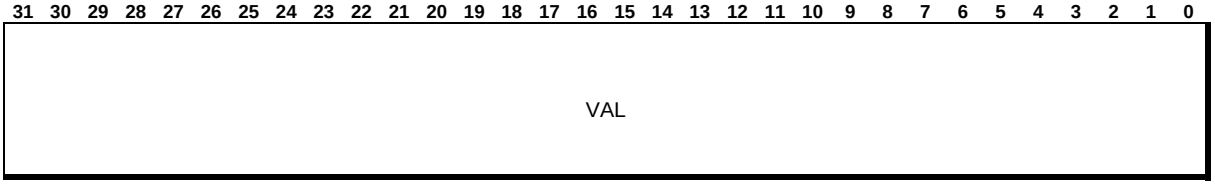
Write access with ’1’ resets interrupt mask bit (disables IRQ for corresponding interrupt source). Write access with ’0’ does not influence this bit. Read access shows actual interrupt mask.



Bits	Name	Description	R/W	Default
31:1	-	reserved	R	0x0
0	MISALIGNMENT_IRQ	Data misalignment error interrupt	R/W	0x0

XPIC_BREAK_RETURN_FIQ_PC – xPIC_DEBUG information FIQ return PC value

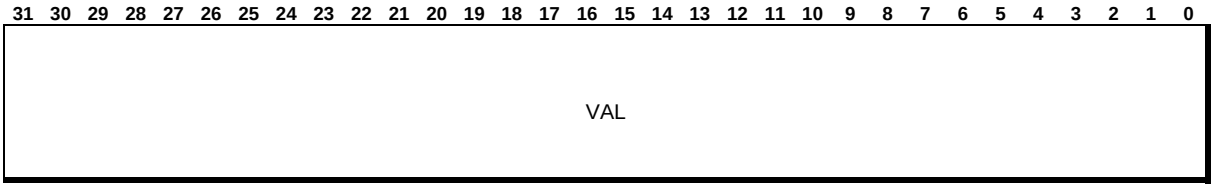
valid if xPIC is in FIQ



Bits	Name	Description	R/W	Default
31:0	VAL	xPIC FIQ return value	R	0x0

XPIC_BREAK_RETURN_IRQ_PC – xPIC_DEBUG information last IRQ return PC value

valid if xPIC is in IRQ



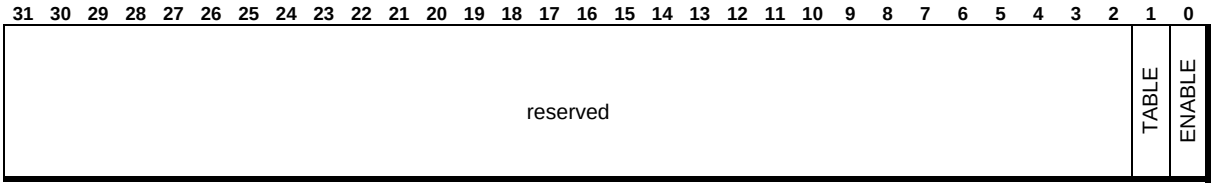
Bits	Name	Description	R/W	Default
31:0	VAL	xPIC last IRQ return value	R	0x0

4.3 XPIC_VIC (vectored interrupt controller) register list

Register name	Brief description
XPIC_VIC_CONFIG	XPIC VIC configuration register
XPIC_VIC_RAW_INTR0	XPIC VIC raw0 interrupt status register
XPIC_VIC_RAW_INTR1	XPIC VIC raw1 interrupt status register
XPIC_VIC_SOFTINT0_SET	XPIC VIC software0 interrupt set register
XPIC_VIC_SOFTINT1_SET	XPIC VIC software1 interrupt set register
XPIC_VIC_SOFTINT0_RESET	XPIC VIC software0 interrupt reset register
XPIC_VIC_SOFTINT1_RESET	XPIC VIC software1 interrupt reset register
XPIC_VIC_FIQ_ADDR	XPIC VIC FIQ vector address 0 register
XPIC_VIC_IRQ_ADDR	XPIC VIC normal IRQ address register
XPIC_VIC_VECTOR_ADDR	XPIC VIC IRQ vector address
XPIC_VIC_TABLE_BASE_ADDR	XPIC VIC IRQ TABLE ADDRESS BASE POINTER
XPIC_VIC_FIQ_VECT_CONFIG	
XPIC_VIC_VECT_CONFIG0	highest priority
XPIC_VIC_VECT_CONFIG1	
XPIC_VIC_VECT_CONFIG2	
XPIC_VIC_VECT_CONFIG3	
XPIC_VIC_VECT_CONFIG4	
XPIC_VIC_VECT_CONFIG5	
XPIC_VIC_VECT_CONFIG6	
XPIC_VIC_VECT_CONFIG7	
XPIC_VIC_VECT_CONFIG8	
XPIC_VIC_VECT_CONFIG9	
XPIC_VIC_VECT_CONFIG10	
XPIC_VIC_VECT_CONFIG11	
XPIC_VIC_VECT_CONFIG12	
XPIC_VIC_VECT_CONFIG13	
XPIC_VIC_VECT_CONFIG14	
XPIC_VIC_VECT_CONFIG15	XPIC default interrupt vector, all interrupt sources (wired-OR)
XPIC_VIC_DEFAULT0	XPIC default interrupt vector select0
XPIC_VIC_DEFAULT1	XPIC default interrupt vector select1

Table 19: XPIC_VIC register list

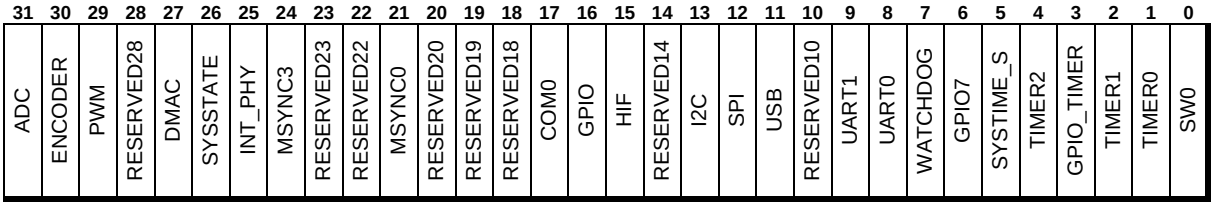
XPIC_VIC_CONFIG – XPIC VIC configuration register



Bits	Name	Description	R/W	Default
31:2	-	reserved	R	0x0
1	TABLE	use far or near table 0 = Base Pointer Addr for IRQ Jmp Table + (n*4) DWORD Table 1 = Base Pointer Addr for IRQ Jmp Table + (n*16) 4 DWORD Table n = IRQ vector number	R/W	0x0
0	ENABLE	global enable of xPIC VIC (0: disable/ 1: enable)	R/W	0x0

XPIC_VIC_RAW_INTR0 – xPIC VIC raw0 interrupt status register

valid for netX 10:

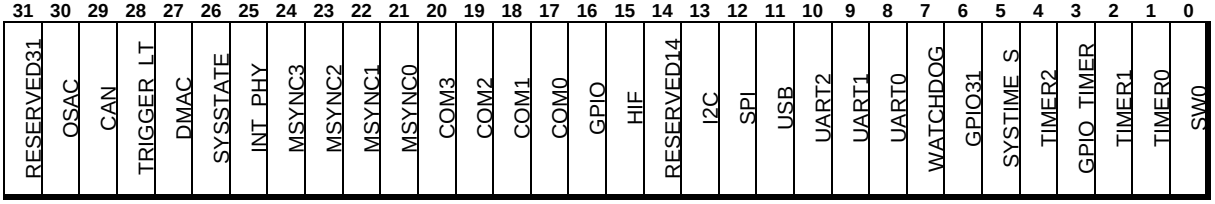


Bits	Name	Description	R/W	Default
31	ADC	ADC0 or ADC1	R	0x0
30	ENCODER	Any encoder IRQ	R	0x0
29	PWM	Any PWM IRQ	R	0x0
28	RESERVED28	reserved for netX compatibility (trigger_lu)	R	0x0
27	DMAC	DMA controller	R	0x0
26	SYSSTATE	License error or extmem_timeout	R	0x0
25	INT_PHY	Interrupt from internal Phy	R	0x0
24	MSYNC3	reserved for SW IRQ from ARM to xPIC	R	0x0
23	RESERVED23	reserved for netX compatibility (msync2)	R	0x0
22	RESERVED22	reserved for netX compatibility (msync1)	R	0x0
21	MSYNC0	Motion synchronization channel 0 (= |xpec0_irq[15:12])	R	0x0
20	RESERVED20	reserved (com3)	R	0x0
19	RESERVED19	reserved for netX compatibility (com2)	R	0x0
18	RESERVED18	reserved for netX compatibility (com1)	R	0x0
17	COM0	Communication channel 0 (= |xpec0_irq[11:0])	R	0x0
16	GPIO	other external Interrupts from GPIO 0-6 / IOLINK	R	0x0
15	HIF	HIF/DPM interrupt	R	0x0
14	RESERVED14	reserved for netX compatibility (lcd)	R	0x0
13	I2C	I2C	R	0x0
12	SPI	combined SPI0, SPI1 interrupt	R	0x0
11	USB	USB interrupt	R	0x0
10	RESERVED10	reserved for netX compatibility (uart2)	R	0x0
9	UART1	UART 1	R	0x0
8	UART0	UART 0 -> Diagnostic channel, Windows CE required	R	0x0
7	WATCHDOG	Watchdog IRQ from xPIC_WDG module	R	0x0
6	GPIO7	external interrupt 7, Windows CE required (NMI)	R	0x0
5	SYSTIME_S	Systime 1day IRQ from xPIC_TIMER module	R	0x0
4	TIMER2	xPIC timer2 from xPIC_TIMER module	R	0x0
3	GPIO_TIMER	GPIO timer0 or timer1 (sep. gpio_irq registers for ARM(intlogic) and xPIC(intlogic_motion))	R	0x0
2	TIMER1	xPIC timer1 from xPIC_TIMER module	R	0x0
1	TIMER0	xPIC timer0 from xPIC_TIMER module real time operating system timer, Windows CE required	R	0x0
0	SW0	reserved for software interrupt	R	0x0

Table 20: XPIC_VIC_RAW_INTR0 – xPIC VIC raw0 interrupt status register (valid for netX 10)

XPIC_VIC_RAW_INTR0 – xPIC VIC raw0 interrupt status register

valid for netX 6/51/52:

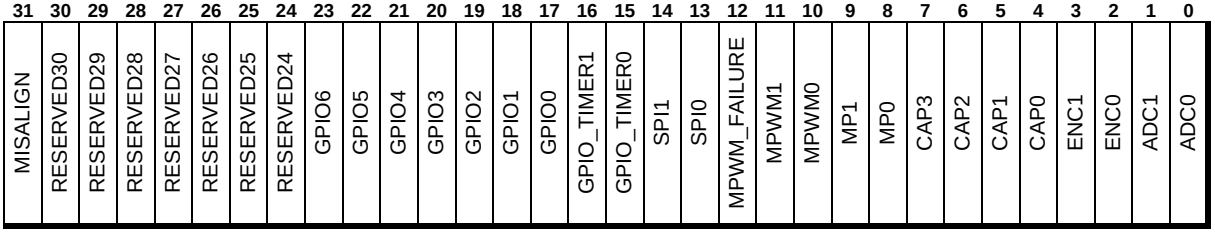


Bits	Name	Description	R/W	Default
31	RESERVED31	reserved for netX compatibility for ADC0 or ADC1	R	0x0
30	OSAC	OSAC nfifo or scheduler	R	0x0
29	CAN	CAN IRQ	R	0x0
28	TRIGGER_LT	trigger_lt	R	0x0
27	DMAC	DMA controller	R	0x0
26	SYSSTATE	License error or extmem_timeout	R	0x0
25	INT_PHY	Interrupt from internal Phy	R	0x0
24	MSYNC3	reserved for SW IRQ from ARM to xPIC	R	0x0
23	MSYNC2	reserved for netX compatibility (msync2)	R	0x0
22	MSYNC1	Motion synchronization channel 1 (= |xpec0_irq[15:12])	R	0x0
21	MSYNC0	Motion synchronization channel 0 (= |xpec0_irq[15:12])	R	0x0
20	COM3	reserved (com3)	R	0x0
19	COM2	reserved for netX compatibility (com2)	R	0x0
18	COM1	Communication channel 1 (= |xpec0_irq[11:0])	R	0x0
17	COM0	Communication channel 0 (= |xpec0_irq[11:0])	R	0x0
16	GPIO	Other external Interrupts from GPIO 0-30 / IOLINK	R	0x0
15	HIF	combined HIF interrupt: DPM, Handshake-Cells (HANDSHAKE_CTRL) and HIF PIOs (HIF_IO_CTRL)	R	0x0
14	RESERVED14	reserved for netX compatibility (LCD)	R	0x0
13	I2C	combined I2C0, I2C1 interrupt	R	0x0
12	SPI	combined SPI0, SPI1 interrupt	R	0x0
11	USB	USB interrupt	R	0x0
10	UART2	UART 2	R	0x0
9	UART1	UART 1	R	0x0
8	UART0	UART 0 -> Diagnostic channel, Windows CE required	R	0x0
7	WATCHDOG	Watchdog IRQ from XPIC_WDG module	R	0x0
6	GPIO31	external interrupt 31, Windows CE required (NMI)	R	0x0
5	SYSTIME_S	systime_s/systime_uc_s IRQ from ARM_TIMER module	R	0x0
4	TIMER2	xPIC Timer2 from XPIC_TIMER Module	R	0x0
3	GPIO_TIMER	GPIO Timer0-4 (sep. gpio_irq registers for ARM(intlogic) and xPIC(intlogic_motion))	R	0x0
2	TIMER1	xPIC Timer1 from XPIC_TIMER Module	R	0x0
1	TIMER0	xPIC Timer0 from XPIC_TIMER Module Real time operating system timer, Windows CE required	R	0x0
0	SW0	Reserved for Software Interrupt	R	0x0

Table 21: XPIC_VIC_RAW_INTR0 – xPIC VIC raw0 interrupt status register (valid for netX 6/51/52)

XPIC_VIC_RAW_INTR1 – xPIC VIC raw1 interrupt status register

valid for netX 10:

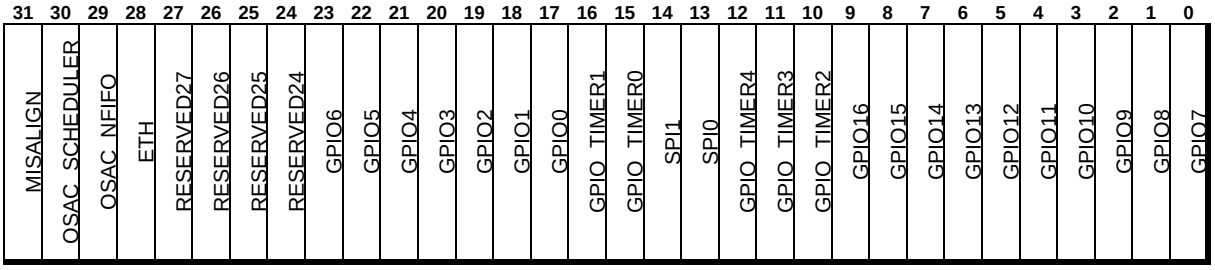


Bits	Name	Description	R/W	Default
31	MISALIGN	xPIC data misalignment	R	0x0
30	RESERVED30	reserved	R	0x0
29	RESERVED29	reserved	R	0x0
28	RESERVED28	reserved	R	0x0
27	RESERVED27	reserved	R	0x0
26	RESERVED26	reserved	R	0x0
25	RESERVED25	reserved	R	0x0
24	RESERVED24	reserved	R	0x0
23	GPIO6	gpio6	R	0x0
22	GPIO5	gpio5	R	0x0
21	GPIO4	gpio4	R	0x0
20	GPIO3	gpio3	R	0x0
19	GPIO2	gpio2	R	0x0
18	GPIO1	gpio1	R	0x0
17	GPIO0	gpio0	R	0x0
16	GPIO_TIMER1	gpio_timer1	R	0x0
15	GPIO_TIMER0	gpio_timer0	R	0x0
14	SPI1	spi1	R	0x0
13	SPI0	spi0	R	0x0
12	MPWM_FAILURE	mpwm_failure	R	0x0
11	MPWM1	mpwm1	R	0x0
10	MPWM0	mpwm0	R	0x0
9	MP1	Encoder mp1	R	0x0
8	MP0	Encoder mp0	R	0x0
7	CAP3	Encoder capture unit 3	R	0x0
6	CAP2	Encoder capture unit 2	R	0x0
5	CAP1	Encoder capture unit 1	R	0x0
4	CAP0	Encoder capture unit 0	R	0x0
3	ENC1	Encoder1 (ovfl, edge)	R	0x0
2	ENC0	Encoder0 (ovfl, edge)	R	0x0
1	ADC1	ADC1	R	0x0
0	ADC0	ADC0	R	0x0

Table 22: XPIC_VIC_RAW_INTR1 – xPIC VIC raw1 interrupt status register (valid for netX 10)

XPIC_VIC_RAW_INTR1 – xPIC VIC raw1 interrupt status register

valid for netX 6/51/52:



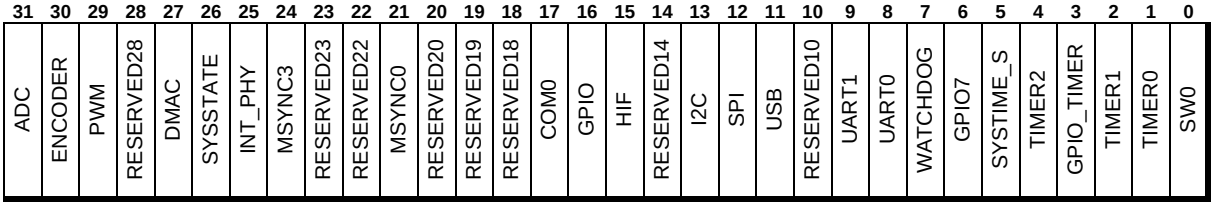
mm

Bits	Name	Description	R/W	Default
31	MISALIGN	xPIC data misalignment	R	0x0
30	OSAC_SCHEDULER	osac_scheduler	R	0x0
29	OSAC_NFIFO	osac_nfifo	R	0x0
28	ETH	ETH module	R	0x0
27	RESERVED27	reserved	R	0x0
26	RESERVED26	reserved	R	0x0
25	RESERVED25	reserved	R	0x0
24	RESERVED24	reserved	R	0x0
23	GPIO6	gpio6	R	0x0
22	GPIO5	gpio5	R	0x0
21	GPIO4	gpio4	R	0x0
20	GPIO3	gpio3	R	0x0
19	GPIO2	gpio2	R	0x0
18	GPIO1	gpio1	R	0x0
17	GPIO0	gpio0	R	0x0
16	GPIO_TIMER1	gpio_timer1	R	0x0
15	GPIO_TIMER0	gpio_timer0	R	0x0
14	SPI1	spi1	R	0x0
13	SPI0	spi0	R	0x0
12	GPIO_TIMER4	gpio_timer4	R	0x0
11	GPIO_TIMER3	gpio_timer3	R	0x0
10	GPIO_TIMER2	gpio_timer2	R	0x0
9	GPIO16	gpio16	R	0x0
8	GPIO15	gpio15	R	0x0
7	GPIO14	gpio14	R	0x0
6	GPIO13	gpio13	R	0x0
5	GPIO12	gpio12	R	0x0
4	GPIO11	gpio11	R	0x0
3	GPIO10	gpio10	R	0x0
2	GPIO9	gpio9	R	0x0
1	GPIO8	gpio8	R	0x0
0	GPIO7	gpio7	R	0x0

Table 23: XPIC_VIC_RAW_INTR1 – xPIC VIC raw1 interrupt status register (valid for netX 6/51/52)

XPIC_VIC_SOFTINT0_SET – xPIC VIC software0 interrupt set register

read status write (1) set

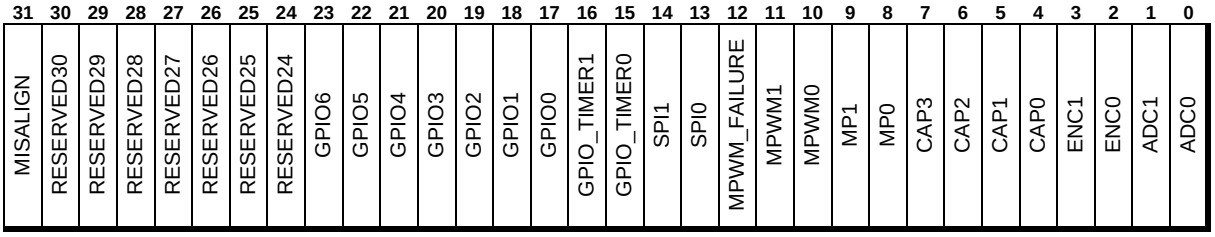


Bits	Name	Description	R/W	Default
31	ADC	ADC0 or ADC1	R/W	0x0
30	ENCODER	any encoder IRQ	R/W	0x0
29	PWM	any PWM IRQ	R/W	0x0
28	RESERVED28	reserved for netX compatibility (trigger_lu)	R/W	0x0
27	DMAC	DMA controller	R/W	0x0
26	SYSSTATE	License error or extmem_timeout	R/W	0x0
25	INT_PHY	Interrupt from internal Phy	R/W	0x0
24	MSYNC3	reserved for SW IRQ from ARM to xPIC	R/W	0x0
23	RESERVED23	reserved for netX compatibility (msync2)	R/W	0x0
22	RESERVED22	reserved for netX compatibility (msync1)	R/W	0x0
21	MSYNC0	Motion synchronization channel 0 (= |xpec0_irq[15:12])	R/W	0x0
20	RESERVED20	reserved (com3)	R/W	0x0
19	RESERVED19	reserved for netX compatibility (com2)	R/W	0x0
18	RESERVED18	reserved for netX compatibility (com1)	R/W	0x0
17	COM0	Communication channel 0 (= |xpec0_irq[11:0])	R/W	0x0
16	GPIO	other external Interrupts from GPIO 0-6 / IOLINK	R/W	0x0
15	HIF	HIF/DPM interrupt	R/W	0x0
14	RESERVED14	reserved for netX compatibility (lcd)	R/W	0x0
13	I2C	I2C	R/W	0x0
12	SPI	combined SPI0, SPI1 interrupt	R/W	0x0
11	USB	USB interrupt	R/W	0x0
10	RESERVED10	reserved for netX compatibility (uart2)	R/W	0x0
9	UART1	UART 1	R/W	0x0
8	UART0	UART 0 -> Diagnostic channel, Windows CE required	R/W	0x0
7	WATCHDOG	Watchdog IRQ from XPIC_WDG module	R/W	0x0
6	GPIO7	external interrupt 7, Windows CE required (NMI)	R/W	0x0
5	SYSTIME_S	Systime 1day IRQ from XPIC_TIMER module	R/W	0x0
4	TIMER2	xPIC timer2 from XPIC_TIMER module	R/W	0x0
3	GPIO_TIMER	GPIO timer0 or timer1 (sep. gpio_irq registers for ARM(intlogic) and xPIC(intlogic_motion))	R/W	0x0
2	TIMER1	xPIC timer1 from XPIC_TIMER module	R/W	0x0
1	TIMER0	xPIC timer0 from XPIC_TIMER module Real time operating system timer, Windows CE required	R/W	0x0
0	SW0	reserved for software interrupt	R/W	0x0

Table 24: XPIC_VIC_SOFTINT0_SET – xPIC VIC software0 interrupt set register

XPIC_VIC_SOFTINT1_SET – xPIC VIC software1 interrupt set register

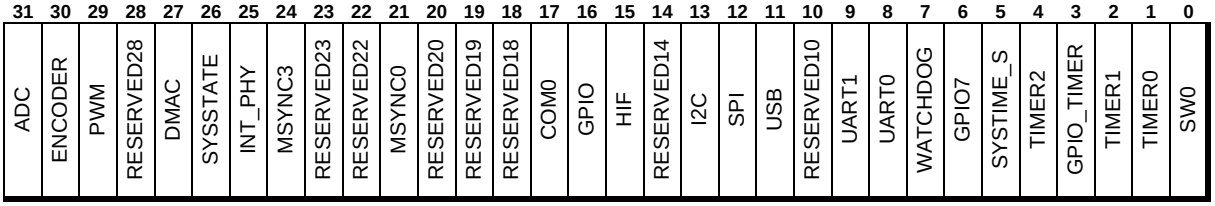
read status write (1) set



Bits	Name	Description	R/W	Default
31	MISALIGN	xPIC data misalignment	R/W	0x0
30	RESERVED30	reserved	R/W	0x0
29	RESERVED29	reserved	R/W	0x0
28	RESERVED28	reserved	R/W	0x0
27	RESERVED27	reserved	R/W	0x0
26	RESERVED26	reserved	R/W	0x0
25	RESERVED25	reserved	R/W	0x0
24	RESERVED24	reserved	R/W	0x0
23	GPIO6	gpio6	R/W	0x0
22	GPIO5	gpio5	R/W	0x0
21	GPIO4	gpio4	R/W	0x0
20	GPIO3	gpio3	R/W	0x0
19	GPIO2	gpio2	R/W	0x0
18	GPIO1	gpio1	R/W	0x0
17	GPIO0	gpio0	R/W	0x0
16	GPIO_TIMER1	gpio_timer1	R/W	0x0
15	GPIO_TIMER0	gpio_timer0	R/W	0x0
14	SPI1	spi1	R/W	0x0
13	SPI0	spi0	R/W	0x0
12	MPWM_FAILURE	mpwm_failure	R/W	0x0
11	MPWM1	mpwm1	R/W	0x0
10	MPWM0	mpwm0	R/W	0x0
9	MP1	Encoder mp1	R/W	0x0
8	MP0	Encoder mp0	R/W	0x0
7	CAP3	Encoder capture unit 3	R/W	0x0
6	CAP2	Encoder capture unit 2	R/W	0x0
5	CAP1	Encoder capture unit 1	R/W	0x0
4	CAP0	Encoder capture unit 0	R/W	0x0
3	ENC1	Encoder1 (ovfl, edge)	R/W	0x0
2	ENC0	Encoder0 (ovfl, edge)	R/W	0x0
1	ADC1	ADC1	R/W	0x0
0	ADC0	ADC0	R/W	0x0

XPIC_VIC_SOFTINT0_RESET – xPIC VIC software0 interrupt reset register

read status write (1) reset

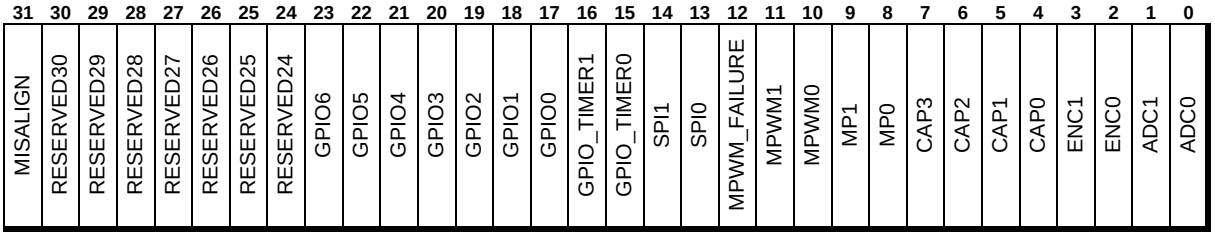


Bits	Name	Description	R/W	Default
31	ADC	ADC0 or ADC1	R/W	0x0
30	ENCODER	any encoder IRQ	R/W	0x0
29	PWM	any PWM IRQ	R/W	0x0
28	RESERVED28	reserved for netX compatibility (trigger_lu)	R/W	0x0
27	DMAC	DMA controller	R/W	0x0
26	SYSSTATE	License error or extmem_timeout	R/W	0x0
25	INT_PHY	Interrupt from internal Phy	R/W	0x0
24	MSYNC3	reserved for SW IRQ from ARM to xPIC	R/W	0x0
23	RESERVED23	reserved for netX compatibility (msync2)	R/W	0x0
22	RESERVED22	reserved for netX compatibility (msync1)	R/W	0x0
21	MSYNC0	Motion synchronization channel 0 (= |xpec0_irq[15:12])	R/W	0x0
20	RESERVED20	reserved (com3)	R/W	0x0
19	RESERVED19	reserved for netX compatibility (com2)	R/W	0x0
18	RESERVED18	reserved for netX compatibility (com1)	R/W	0x0
17	COM0	Communication channel 0 (= |xpec0_irq[11:0])	R/W	0x0
16	GPIO	other external interrupts from GPIO 0-6 / IOLINK	R/W	0x0
15	HIF	HIF/DPM interrupt	R/W	0x0
14	RESERVED14	reserved for netX compatibility (lcd)	R/W	0x0
13	I2C	I2C	R/W	0x0
12	SPI	combined SPI0, SPI1 interrupt	R/W	0x0
11	USB	USB interrupt	R/W	0x0
10	RESERVED10	reserved for netX compatibility (uart2)	R/W	0x0
9	UART1	UART 1	R/W	0x0
8	UART0	UART 0 -> Diagnostic channel, Windows CE required	R/W	0x0
7	WATCHDOG	Watchdog IRQ from xPIC_WDG module	R/W	0x0
6	GPIO7	external interrupt 7, Windows CE required (NMI)	R/W	0x0
5	SYSTIME_S	Systime 1day IRQ from xPIC_TIMER module	R/W	0x0
4	TIMER2	xPIC timer2 from xPIC_TIMER module	R/W	0x0
3	GPIO_TIMER	GPIO timer0 or timer1 (sep. gpio_irq registers for ARM(intlogic) and xPIC(intlogic_motion))	R/W	0x0
2	TIMER1	xPIC timer1 from xPIC_TIMER module	R/W	0x0
1	TIMER0	xPIC timer0 from xPIC_TIMER module Real time operating system timer, Windows CE required	R/W	0x0
0	SW0	reserved for software interrupt	R/W	0x0

Table 25: XPIC_VIC_SOFTINT0_RESET – xPIC VIC software0 interrupt reset register

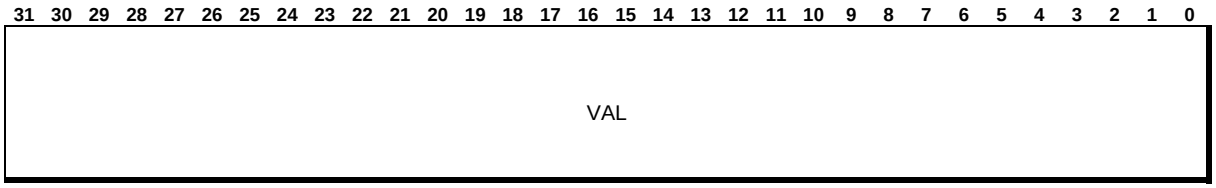
XPIC_VIC_SOFTINT1_RESET – xPIC VIC software1 interrupt reset register

read status write (1) reset



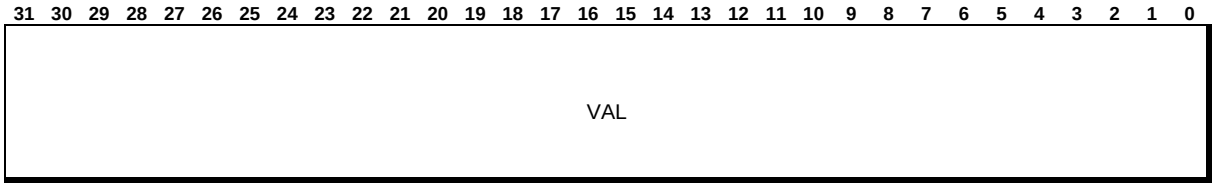
Bits	Name	Description	R/W	Default
31	MISALIGN	xPIC data misalignment	R/W	0x0
30	RESERVED30	reserved	R/W	0x0
29	RESERVED29	reserved	R/W	0x0
28	RESERVED28	reserved	R/W	0x0
27	RESERVED27	reserved	R/W	0x0
26	RESERVED26	reserved	R/W	0x0
25	RESERVED25	reserved	R/W	0x0
24	RESERVED24	reserved	R/W	0x0
23	GPIO6	gpio6	R/W	0x0
22	GPIO5	gpio5	R/W	0x0
21	GPIO4	gpio4	R/W	0x0
20	GPIO3	gpio3	R/W	0x0
19	GPIO2	gpio2	R/W	0x0
18	GPIO1	gpio1	R/W	0x0
17	GPIO0	gpio0	R/W	0x0
16	GPIO_TIMER1	gpio_timer1	R/W	0x0
15	GPIO_TIMER0	gpio_timer0	R/W	0x0
14	SPI1	spi1	R/W	0x0
13	SPI0	spi0	R/W	0x0
12	MPWM_FAILURE	mpwm_failure	R/W	0x0
11	MPWM1	mpwm1	R/W	0x0
10	MPWM0	mpwm0	R/W	0x0
9	MP1	Encoder mp1	R/W	0x0
8	MP0	Encoder mp0	R/W	0x0
7	CAP3	Encoder capture unit 3	R/W	0x0
6	CAP2	Encoder capture unit 2	R/W	0x0
5	CAP1	Encoder capture unit 1	R/W	0x0
4	CAP0	Encoder capture unit 0	R/W	0x0
3	ENC1	Encoder1 (ovfl, edge)	R/W	0x0
2	ENC0	Encoder0 (ovfl, edge)	R/W	0x0
1	ADC1	ADC1	R/W	0x0
0	ADC0	ADC0	R/W	0x0

XPIC_VIC_FIQ_ADDR – xPIC VIC FIQ vector address 0 register



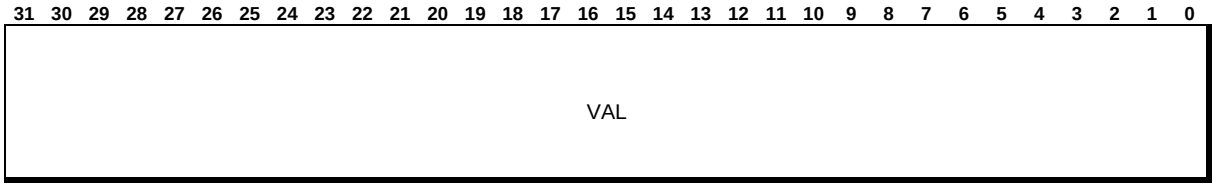
Bits	Name	Description	R/W	Default
31:0	VAL	FIQ handler address	R/W	0x0

XPIC_VIC_IRQ_ADDR – xPIC VIC normal IRQ address register



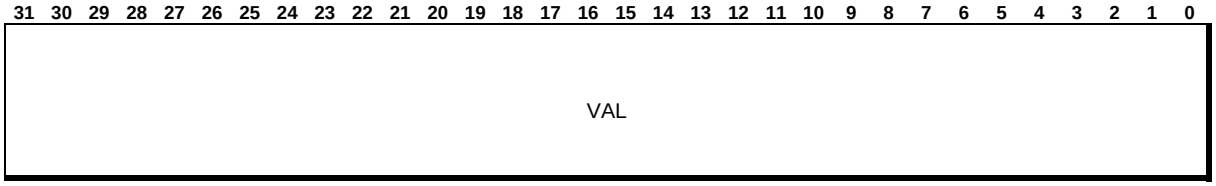
Bits	Name	Description	R/W	Default
31:0	VAL	IRQ handler address	R/W	0x0

XPIC_VIC_VECTOR_ADDR – xPIC VIC IRQ vector address



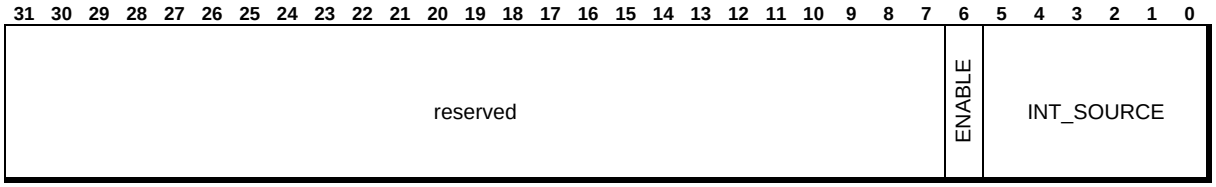
Bits	Name	Description	R/W	Default
31:0	VAL	IRQ vector address read access get actual highest priority IRQ read access get adr_xpic_vic_table_base_addr + IRQ Number * (4/16)	R	0x0

XPIC_VIC_TABLE_BASE_ADDR – xPIC VIC IRQ TABLE ADDRESS BASE POINTER



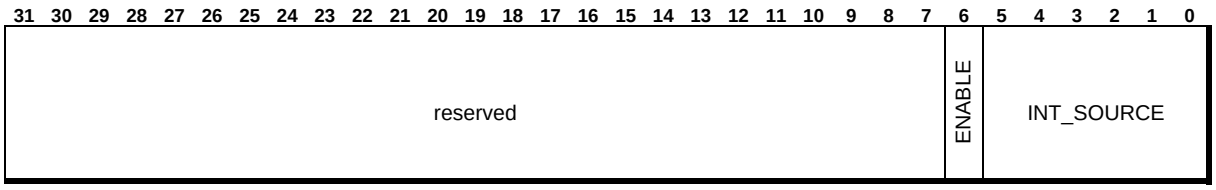
Bits	Name	Description	R/W	Default
31:0	VAL	IRQ table base address the Base Pointer Addr for IRQ Jmp Table	R/W	0x0

XPIC_VIC_FIQ_VECT_CONFIG



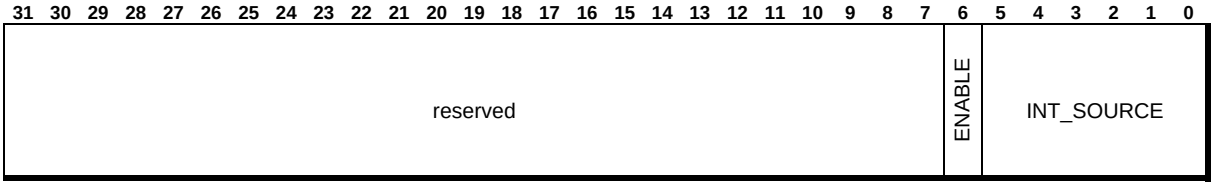
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG0 – highest priority



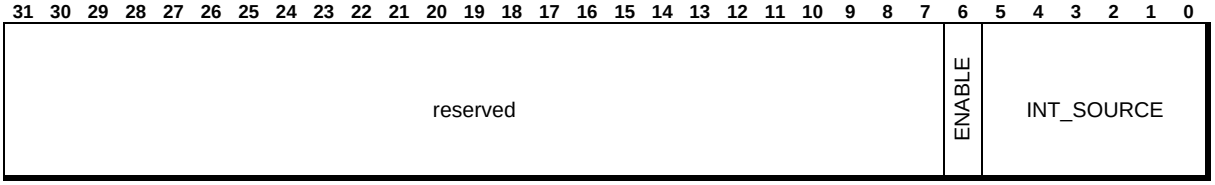
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG1



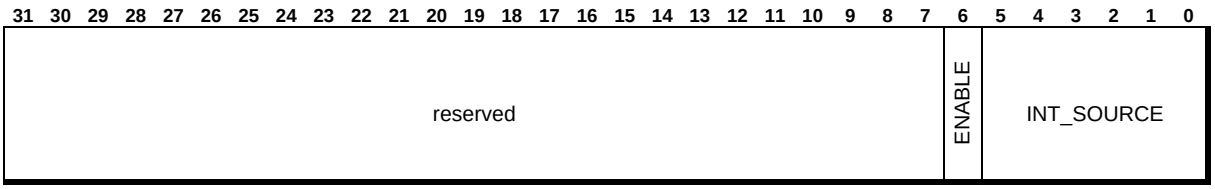
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG2



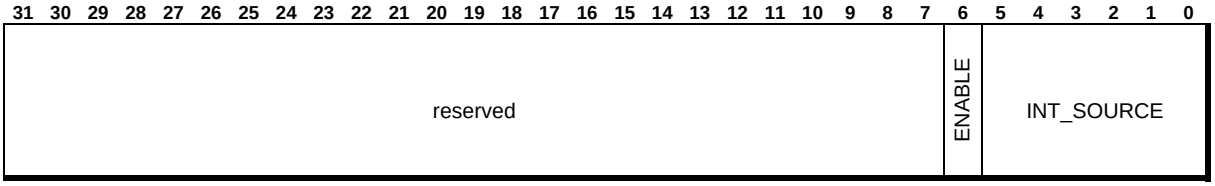
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG3



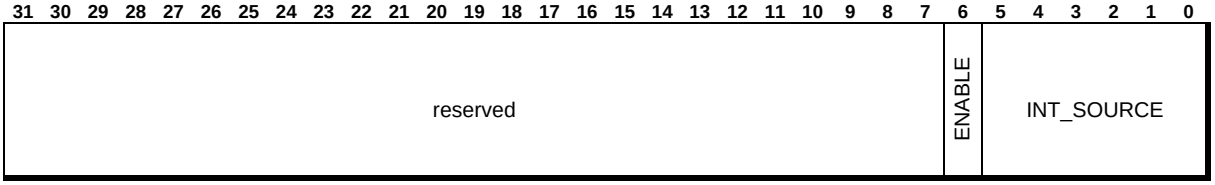
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG4



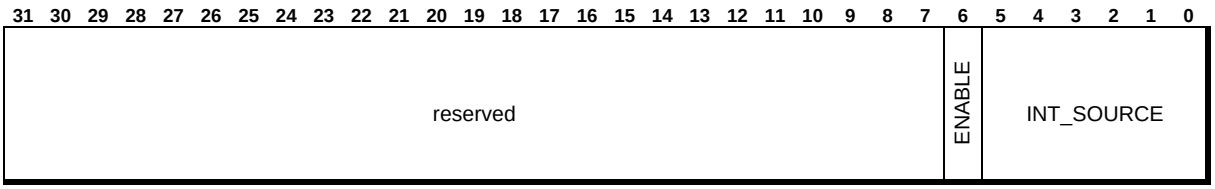
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG5



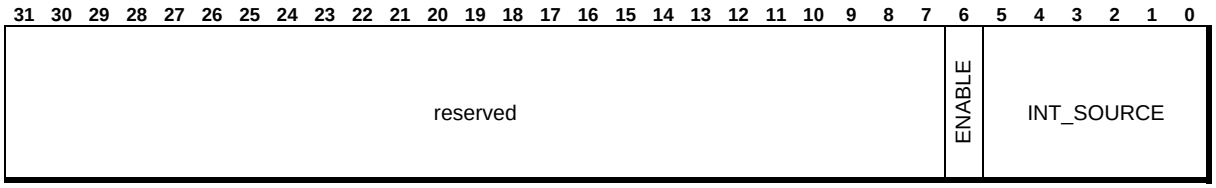
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG6



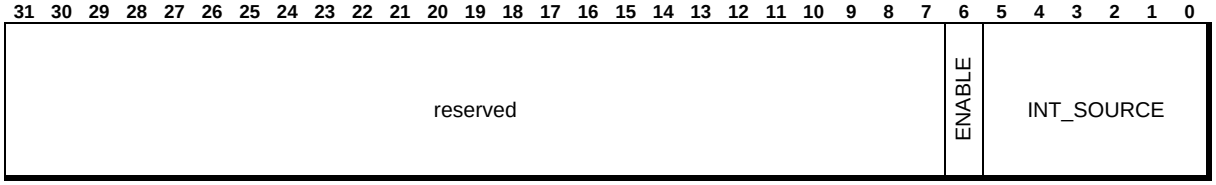
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG7



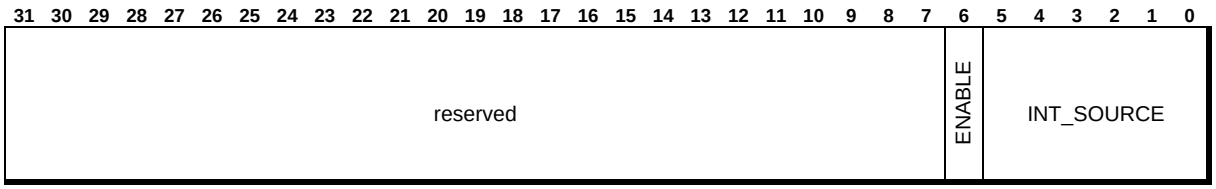
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG8



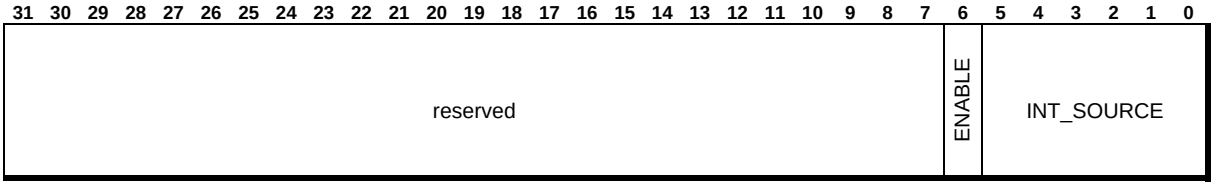
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG9



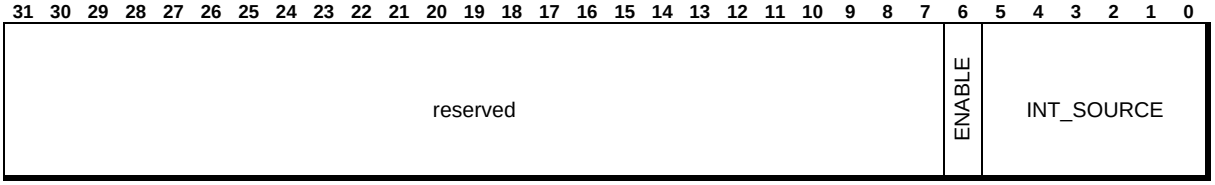
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG10



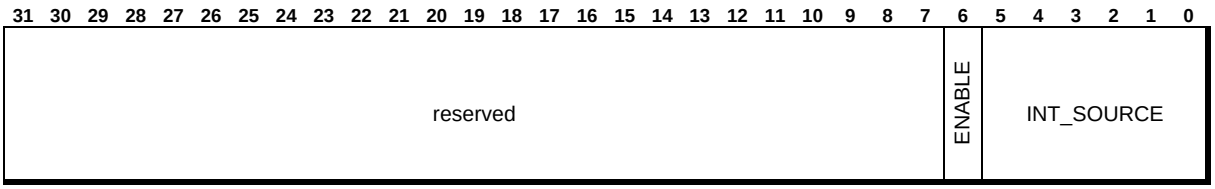
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG11



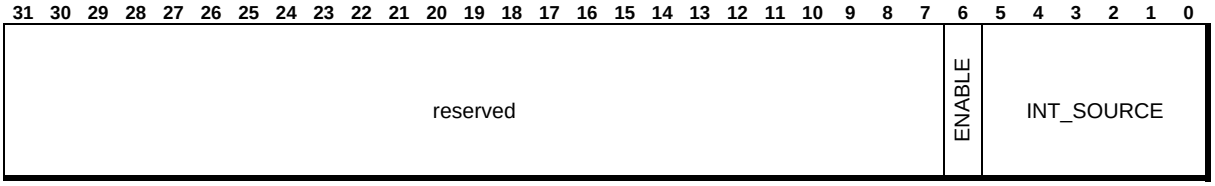
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG12



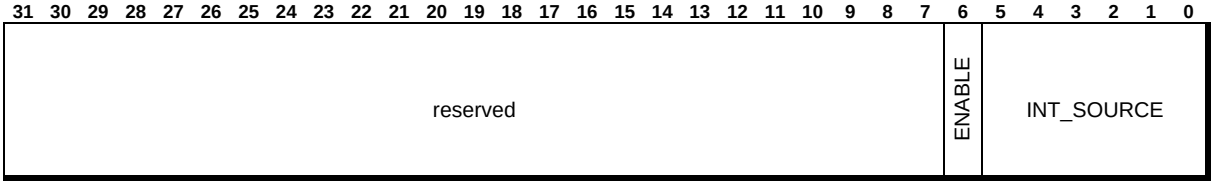
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

XPIC_VIC_VECT_CONFIG13



Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

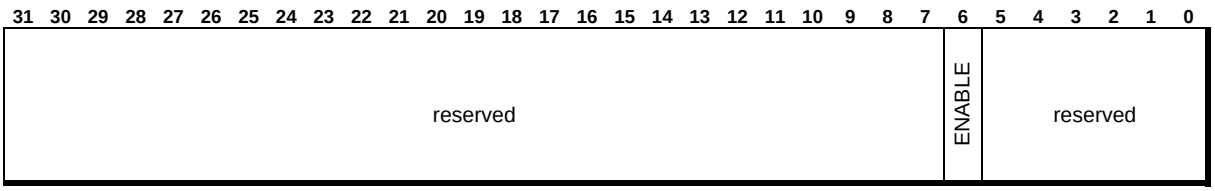
XPIC_VIC_VECT_CONFIG14



Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	INT_SOURCE	interrupt source 0 - 63	R/W	0x0

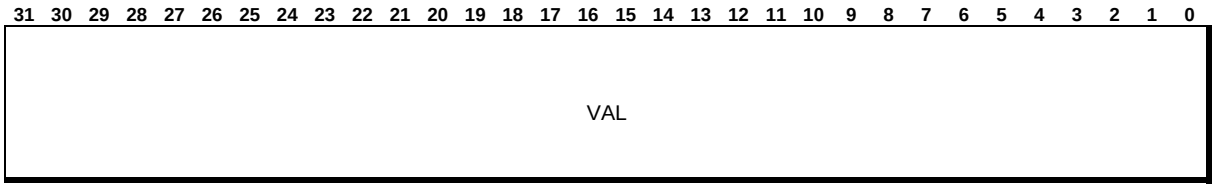
XPIC_VIC_VECT_CONFIG15 – xPIC default interrupt vector, all interrupt sources (wired-OR)

select with default interrupt vector register lowest priority



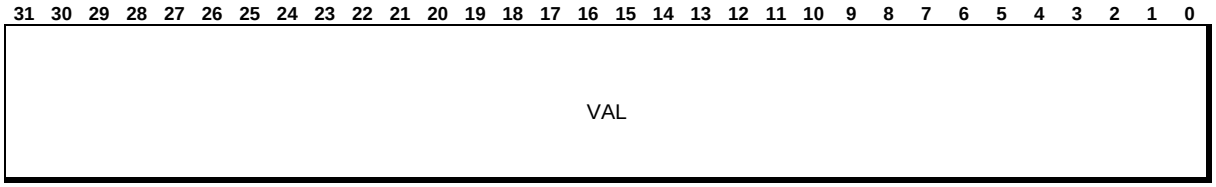
Bits	Name	Description	R/W	Default
31:7	-	reserved	R	0x0
6	ENABLE	vector interrupt enable	R/W	0x0
5:0	-	reserved	R	0x0

XPIC_VIC_DEFAULT0 – xPIC default interrupt vector select0



Bits	Name	Description	R/W	Default
31:0	VAL	select int0 - int31 (wired-OR); 1: selected, 0: not selected	R/W	0x0

XPIC_VIC_DEFAULT1 – xPIC default interrupt vector select1



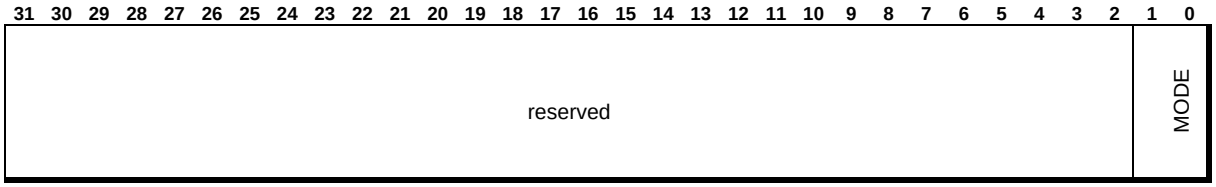
Bits	Name	Description	R/W	Default
31:0	VAL	select int32 - int63 (wired-OR); 1: selected, 0: not selected	R/W	0x0

4.4 XPIC_TIMER register list

Register name	Brief description
XPIC_TIMER_CONFIG_TIMER0	xPIC TIMER config register0
XPIC_TIMER_CONFIG_TIMER1	xPIC TIMER config register1
XPIC_TIMER_CONFIG_TIMER2	xPIC TIMER config register2
XPIC_TIMER_PRELOAD_TIMER0	xPIC TIMER timer 0 preload
XPIC_TIMER_PRELOAD_TIMER1	xPIC TIMER timer 1 preload
XPIC_TIMER_PRELOAD_TIMER2	xPIC TIMER timer 2 preload
XPIC_TIMER_TIMER0	xPIC TIMER timer 0
XPIC_TIMER_TIMER1	xPIC TIMER timer 1
XPIC_TIMER_TIMER2	xPIC TIMER timer 2
XPIC_TIMER_IRQ_RAW	xPIC_TIMER raw IRQ register
XPIC_TIMER_IRQ_MASKED	xPIC_TIMER masked IRQ register
XPIC_TIMER_IRQ_MSK_SET	xPIC_TIMER interrupt mask enable
XPIC_TIMER_IRQ_MSK_RESET	xPIC_TIMER interrupt mask disable
XPIC_TIMER_SYSTIME_S	xPIC_TIMER upper SYSTIME register
XPIC_TIMER_SYSTIME_NS	xPIC_TIMER lower SYSTIME register
XPIC_TIMER_COMPARE_SYSTIME_S_VALUE	xPIC_TIMER SYSTIME sec compare register

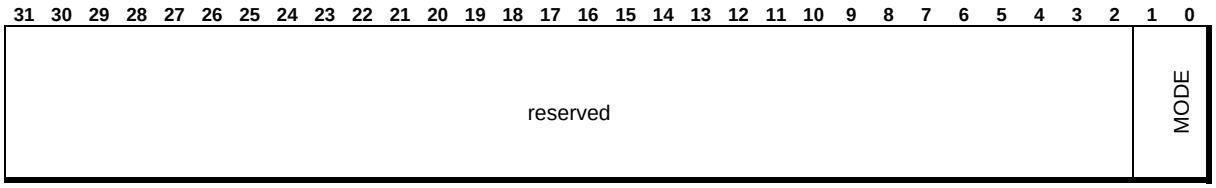
Table 26: XPIC_TIMER register list

XPIC_TIMER_CONFIG_TIMER0 – xPIC TIMER config register0



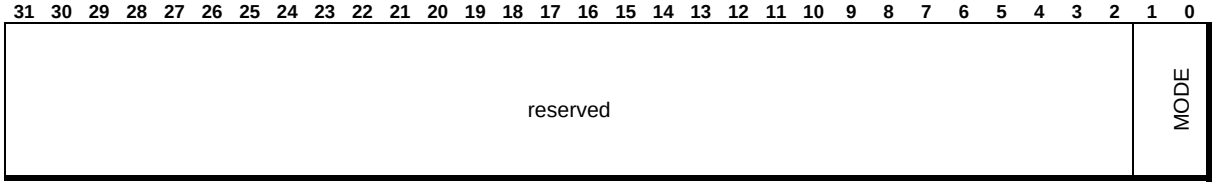
Bits	Name	Description	R/W	Default
31:2	-	reserved	R	0x0
1:0	MODE	Timer0 2’b00 : Timer stops at 0 2’b01 : Timer is preloaded with value from preload register at 0 2’b10 : Timer (value) compare with systime (once) 2’b11 : reserved	R/W	0x0

XPIC_TIMER_CONFIG_TIMER1 – xPIC TIMER config register1



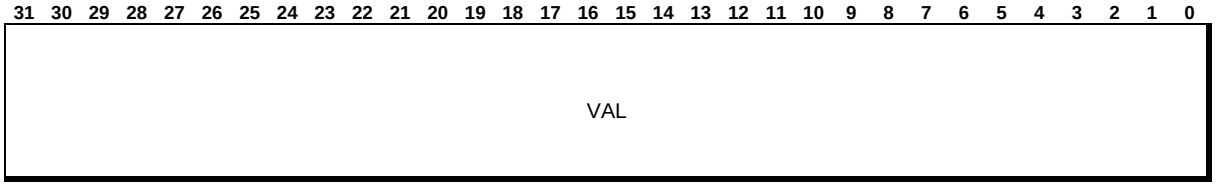
Bits	Name	Description	R/W	Default
31:2	-	reserved	R	0x0
1:0	MODE	Timer1 2’b00 : Timer stops at 0 2’b01 : Timer is preloaded with value from preload register at 0 2’b10 : Timer (value) compare with systime (once) 2’b11 : reserved	R/W	0x0

XPIC_TIMER_CONFIG_TIMER2 – xPIC TIMER Config register2



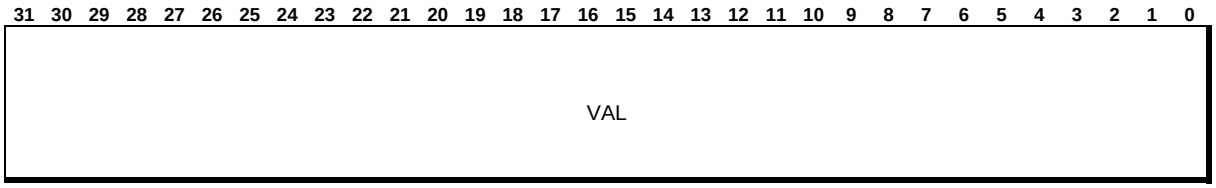
Bits	Name	Description	R/W	Default
31:2	-	reserved	R	0x0
1:0	MODE	Timer2 2’b00 : Timer stops at 0 2’b01 : Timer is preloaded with value from preload register at 0 2’b10 : Timer (value) compare with systime (once) 2’b11 : reserved	R/W	0x0

XPIC_TIMER_PRELOAD_TIMER0 – xPIC TIMER timer 0 preload



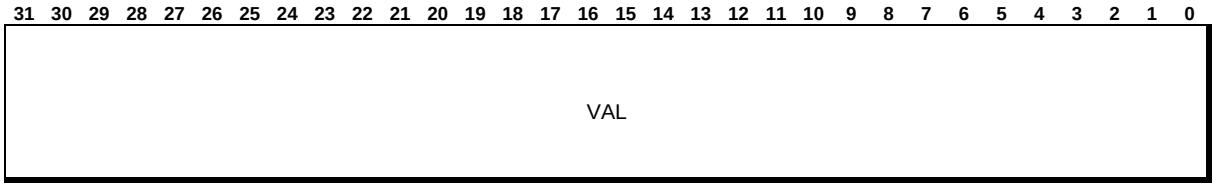
Bits	Name	Description	R/W	Default
31:0	VAL	preload value	R/W	0x0

XPIC_TIMER_PRELOAD_TIMER1 – xPIC TIMER timer 1 preload



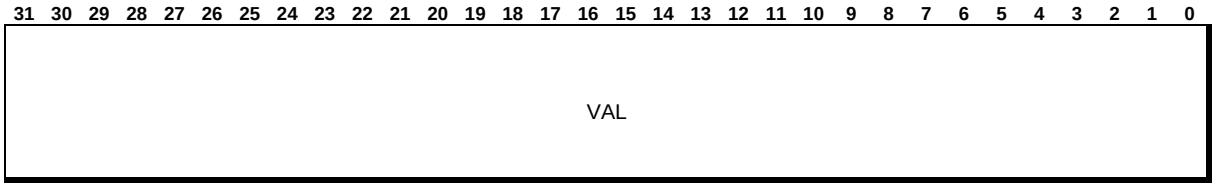
Bits	Name	Description	R/W	Default
31:0	VAL	preload value	R/W	0x0

XPIC_TIMER_PRELOAD_TIMER2 – xPIC TIMER Timer 2 preload



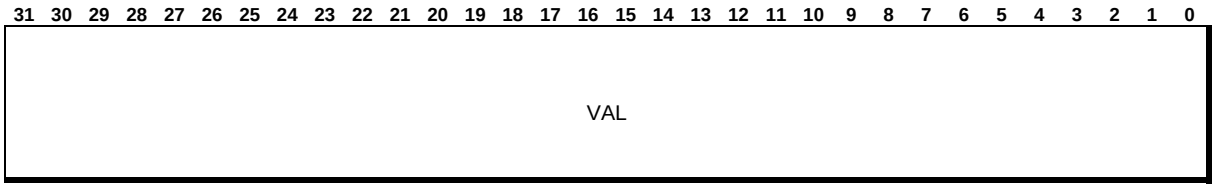
Bits	Name	Description	R/W	Default
31:0	VAL	preload value	R/W	0x0

XPIC_TIMER_TIMER0 – xPIC TIMER timer 0



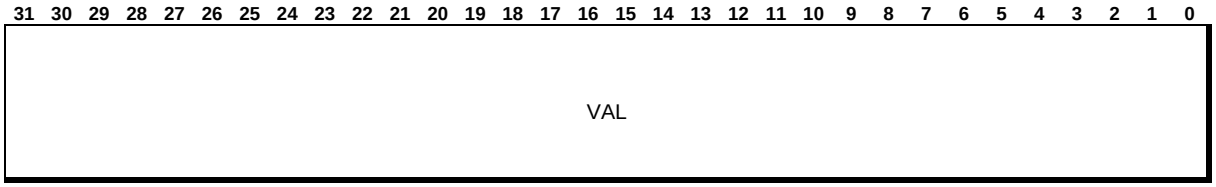
Bits	Name	Description	R/W	Default
31:0	VAL	actual value of timer / systime compare value	R/W	0x0

XPIC_TIMER_TIMER1 – xPIC TIMER timer 1



Bits	Name	Description	R/W	Default
31:0	VAL	actual value of timer / systime compare value	R/W	0x0

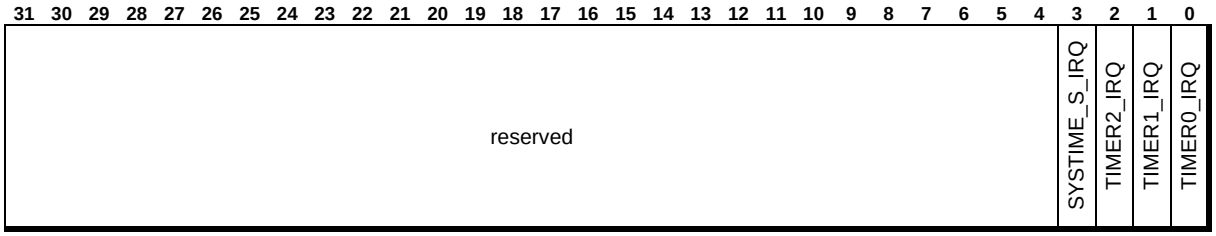
XPIC_TIMER_TIMER2 – xPIC TIMER timer 2



Bits	Name	Description	R/W	Default
31:0	VAL	actual value of timer / systime compare value	R/W	0x0

XPIC_TIMER_IRQ_RAW – xPIC_TIMER raw IRQ register

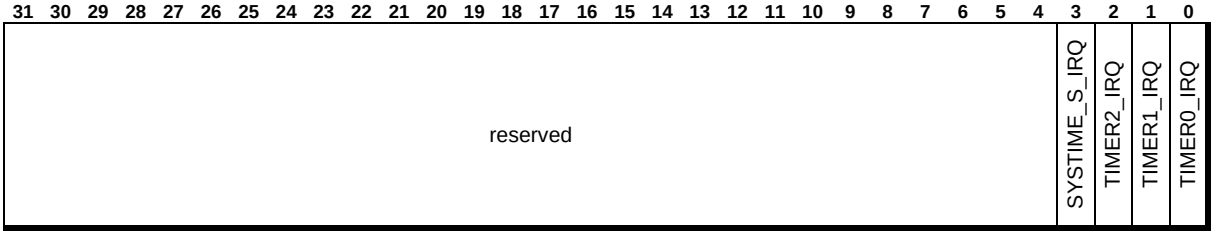
Read access shows status of unmasked IRQs Write access with ’1’ resets the appropriate IRQ



Bits	Name	Description	R/W	Default
31:4	-	reserved	R	0x0
3	SYSTIME_S_IRQ	Systime_s interrupt	R/W	0x0
2	TIMER2_IRQ	Timer 2 interrupt	R/W	0x0
1	TIMER1_IRQ	Timer 1 interrupt	R/W	0x0
0	TIMER0_IRQ	Timer 0 interrupt	R/W	0x0

XPIC_TIMER_IRQ_MASKED – xPIC_TIMER masked IRQ register

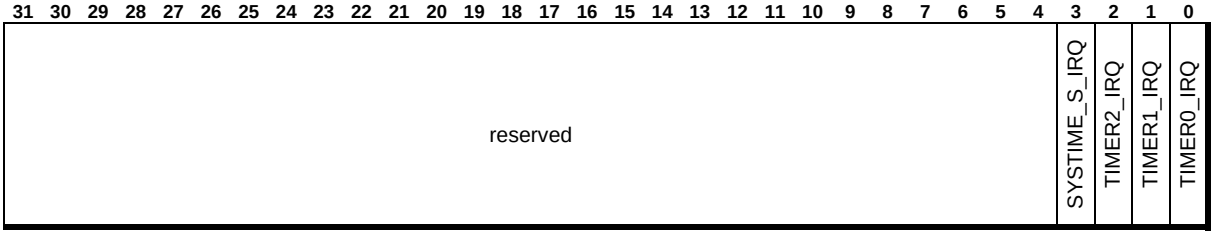
Shows status of masked IRQs (as connected to ARM/xPIC)



Bits	Name	Description	R/W	Default
31:4	-	reserved	R	0x0
3	SYSTIME_S_IRQ	Systime_s interrupt	R	0x0
2	TIMER2_IRQ	Timer 2 interrupt	R	0x0
1	TIMER1_IRQ	Timer 1 interrupt	R	0x0
0	TIMER0_IRQ	Timer 0 interrupt	R	0x0

XPIC_TIMER_IRQ_MSK_SET – xPIC_TIMER interrupt mask enable

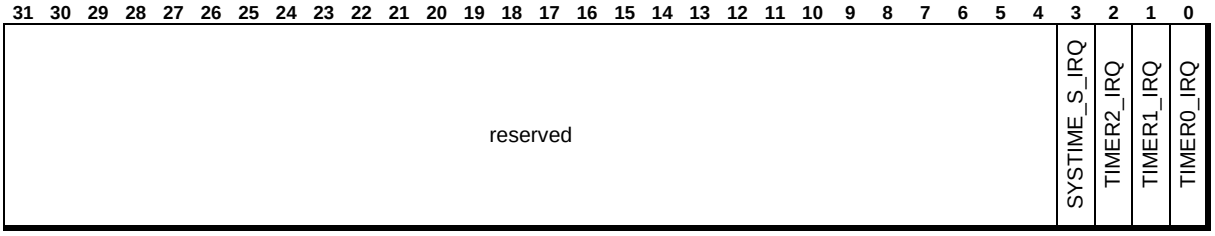
Write access with ’1’ sets interrupt mask bit (enables IR for corresponding interrupt source). Write access with ’0’ does not influence this bit. Read access shows actual interrupt mask.



Bits	Name	Description	R/W	Default
31:4	-	reserved	R	0x0
3	SYSTIME_S_IRQ	Systime_s interrupt	R/W	0x0
2	TIMER2_IRQ	Timer 2 interrupt	R/W	0x0
1	TIMER1_IRQ	Timer 1 interrupt	R/W	0x0
0	TIMER0_IRQ	Timer 0 interrupt	R/W	0x0

XPIC_TIMER_IRQ_MSK_RESET – xPIC_TIMER interrupt mask disable

Write access with ’1’ resets interrupt mask bit (disables IRQ for corresponding interrupt source).
Write access with ’0’ does not influence this bit. Read access shows actual interrupt mask.



Bits	Name	Description	R/W	Default
31:4	-	reserved	R	0x0
3	SYSTIME_S_IRQ	Systime_s interrupt	R/W	0x0
2	TIMER2_IRQ	Timer 2 interrupt	R/W	0x0
1	TIMER1_IRQ	Timer 1 Interrupt	R/W	0x0
0	TIMER0_IRQ	Timer 0 interrupt	R/W	0x0

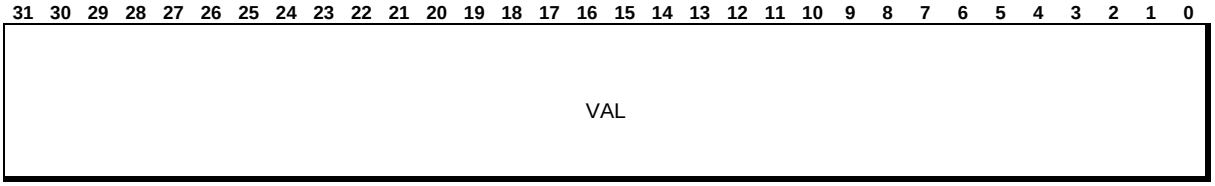
XPIC_TIMER_SYSTIME_S – xPIC_TIMER upper SYSTIME register

To allow consistent values of systime_s and systime_ns, lower bits of systime are latched to systime_ns, when systime_s is read.

This register should be dedicated to accesses via xPIC.

The ARM software should access systime via arm_timer_systime_s.

The host software should access systime via DPM at systime_s.



Bits	Name	Description	R/W	Default
31:0	VAL	Systime high: Sample systime_ns at read access to systime_s. Value is incremented, if systime_ns reaches systime_border.	R	0x0

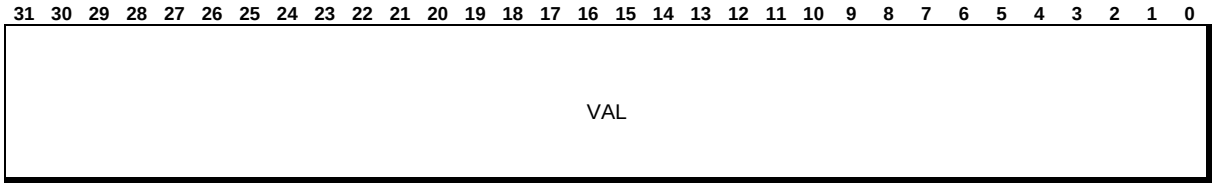
XPIC_TIMER_SYSTIME_NS – xPIC_TIMER lower SYSTIME register

To allow consistent values of systime_s and systime_ns, lower bits of systime are latched to systime_ns, when systime_s is read. If no systime_s is read before (e.g. at 2nd read access of systime_ns), the actual value of systime_ns is read.

This register should be dedicated to accesses via xPIC.

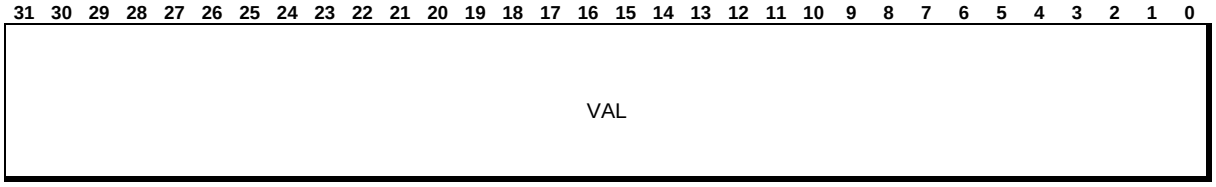
The ARM software should access systime via arm_timer_systime_ns.

The host software should access systime via DPM at systime_ns.



Bits	Name	Description	R/W	Default
31:0	VAL	Systime low: Sample systime_ns at read access to systime_s. Without sample read systime_s, read the actual value of systime_ns.	R	0x0

XPIC_TIMER_COMPARE_SYSTIME_S_VALUE – xPIC_TIMER SYSTIME sec compare register



Bits	Name	Description	R/W	Default
31:0	VAL	Compare value with systime_s (seconds): Systime_s_compare_irq is set, if systime_s matches.	R/W	0x0

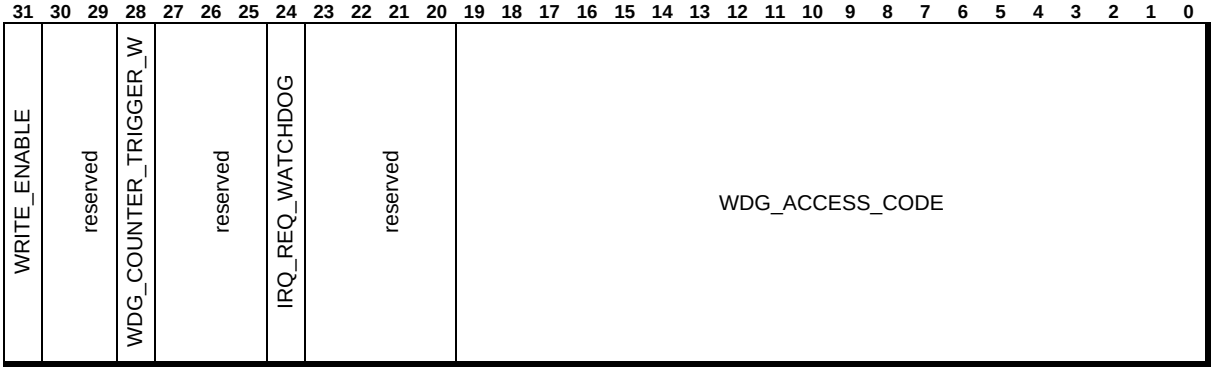
4.5 XPIC_WDG (watchdog) register list

Register name	Brief description
XPIC_WDG_TRIG	netX xPIC watchdog trigger register
XPIC_WDG_COUNTER	netX xPIC watchdog register
XPIC_WDG_XPIC_IRQ_TIMEOUT	netX xPIC watchdog xPIC interrupt timout register
XPIC_WDG_ARM_IRQ_TIMEOUT	netX xPIC Watchdog ARM interrupt timout register
XPIC_WDG_IRQ_RAW	Read access shows status of unmasked IRQs
XPIC_WDG_IRQ_MASKED	xpic_wdg masked IRQ register
XPIC_WDG_IRQ_MSK_SET	xpic_wdg interrupt mask enable
XPIC_WDG_IRQ_MSK_RESET	xpic_wdg interrupt mask disable

Table 27: XPIC_WDG (watchdog) register list

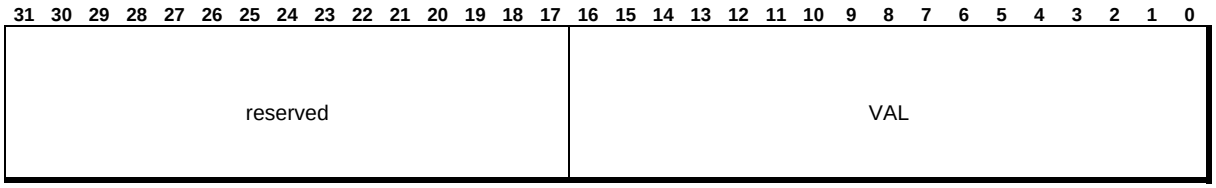
XPIC_WDG_TRIG – netX xPIC watchdog trigger register

The watchdog access code is generated by a pseudo random generator.



Bits	Name	Description	R/W	Default
31	WRITE_ENABLE	Write enable bit for timeout register: As long as this bit is not set, all write accesses to the timeout register will be ignored.	R/W	0x0
30:29	-	reserved	R	0x0
28	WDG_COUNTER_TRIGGER_W	Watchdog trigger bit: Bit must be set to trigger the watchdog counter. When read, this bit is always ’0’	R/W	0x0
27:25	-	reserved	R	0x0
24	IRQ_REQ_WATCHDOG	xPIC IRQ request of watchdog, writing 1 deletes IRQ to xPIC	R/W	0x0
23:20	-	reserved	R	0x0
19:0	WDG_ACCESS_CODE	Watchdog access code for triggering. A read access gives the next 16-bit code for triggering. A write access with correct access code will trigger the watchdog counter.	R/W	0x0

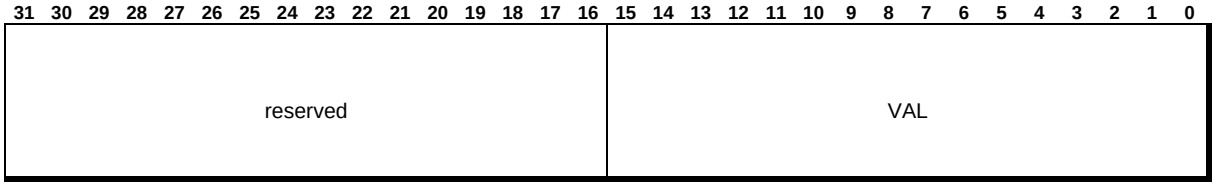
XPIC_WDG_COUNTER – netX xPIC watchdog register



Bits	Name	Description	R/W	Default
31:17	-	reserved	R	0x0
16:0	VAL	Actual watchdog counter value: Bit 16 shows: 1: Watchdog is counting down from xpic_irq_timeout to 0 for xPIC-IRQ 0: Watchdog is counting down from arm_irq_timeout to 0 for ARM-IRQ	R	0x0

XPIC_WDG_XPIC_IRQ_TIMEOUT – netX xPIC watchdog xPIC interrupt timout register

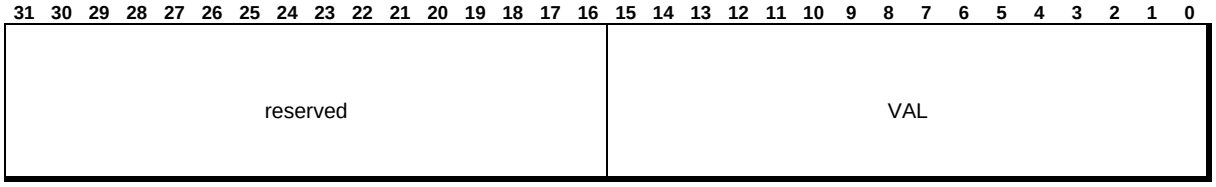
xpic_irq_timeout or arm_irq_timeout must be nonzero to enable watchdog



Bits	Name	Description	R/W	Default
31:16	-	reserved	R	0x0
15:0	VAL	Watchdog interrupt timeout	R/W	0x0

XPIC_WDG_ARM_IRQ_TIMEOUT – netX xPIC watchdog ARM interrupt timout register

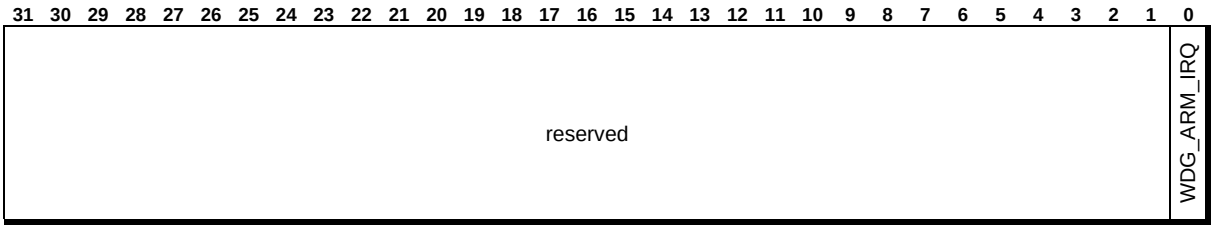
xpic_irq_timeout or arm_irq_timeout must be nonzero to enable watchdog



Bits	Name	Description	R/W	Default
31:16	-	reserved	R	0x0
15:0	VAL	Watchdog ARM interrupt timeout	R/W	0x0

XPIC_WDG_IRQ_RAW – read access shows status of unmasked IRQs

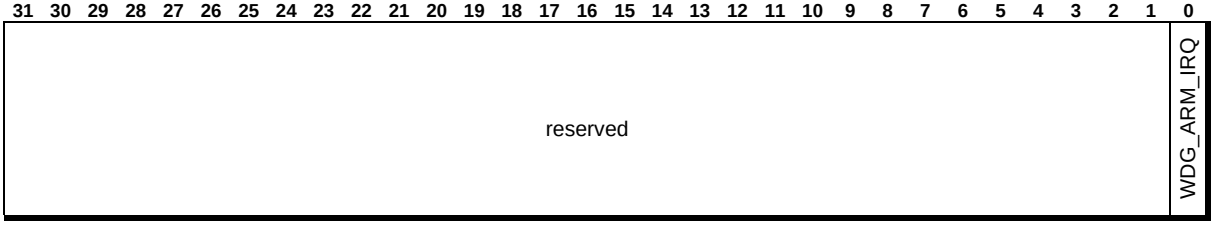
Reset by writing 1 to appropriate bit.



Bits	Name	Description	R/W	Default
31:1	-	reserved	R	0x0
0	WDG_ARM_IRQ	Interrupt from xPIC watchdog to ARM	R/W	0x0

XPIC_WDG_IRQ_MASKED – xpic_wdg masked IRQ register

Shows status of masked IRQs (as connected to ARM/xPIC).

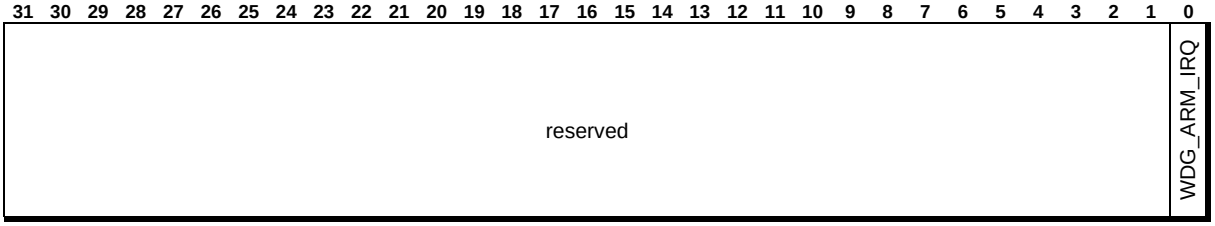


Bits	Name	Description	R/W	Default
31:1	-	reserved	R	0x0
0	WDG_ARM_IRQ	Interrupt from xPIC watchdog to ARM	R	0x0

XPIC_WDG_IRQ_MSK_SET – xpic_wdg interrupt mask enable

Write access with ’1’ sets interrupt mask bit (enables IRQ for corresponding interrupt source).
Write access with ’0’ does not influence this bit. Read access shows actual interrupt mask.

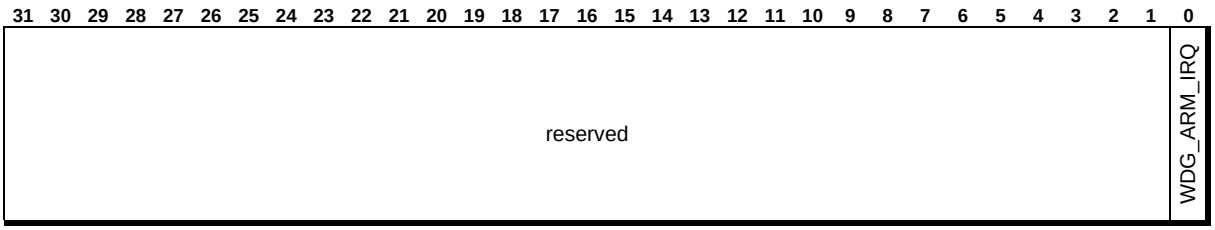
Note: Before activating the interrupt mask, delete old pending interrupts by writing the same value to adr_xpic_wdg_irq_raw.



Bits	Name	Description	R/W	Default
31:1	-	reserved	R	0x0
0	WDG_ARM_IRQ	Interrupt from xPIC watchdog to ARM	R/W	0x0

XPIC_WDG_IRQ_MSK_RESET – xpic_wdg interrupt mask disable

Write access with ’1’ resets interrupt mask bit (disables IRQ for corresponding interrupt source).
Write access with ’0’ does not influence this bit. Read access shows actual interrupt mask.



Bits	Name	Description	R/W	Default
31:1	-	reserved	R	0x0
0	WDG_ARM_IRQ	Interrupt from xPIC watchdog to ARM	R/W	0x0

5 Appendix

5.1 List of tables

5.2 List of figures

5.3 Contacts

Headquarters

Germany
Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

Support
Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

China
Hilscher Systemautomation (Shanghai) Co. Ltd.
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn

Support
Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

France
Hilscher France S.a.r.l.
69500 Bron
Phone: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr

Support
Phone: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

India
Hilscher India Pvt. Ltd.
Pune, Delhi, Mumbai
Phone: +91 8888 750 777
E-Mail: info@hilscher.in

Italy
Hilscher Italia S.r.l.
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it

Support
Phone: +39 02 25007068
E-Mail: it.support@hilscher.com

Japan
Hilscher Japan KK
Tokyo, 160-0022
Phone: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp

Support
Phone: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

Korea
Hilscher Korea Inc.
Seongnam, Gyeonggi, 463-400
Phone: +82 (0) 31-789-3715
E-Mail: info@hilscher.kr

Switzerland
Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch

Support
Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

USA
Hilscher North America, Inc.
Lisle, IL 60532
Phone: +1 630-505-5301
E-Mail: info@hilscher.us

Support
Phone: +1 630-505-5301
E-Mail: us.support@hilscher.com