Cambridge Consultants

XAP4

Programmer's Manual

Cambridge Consultants Limited
Science Park
Milton Road
Cambridge
England
CB4 0DW

+44 (0) 1223 420024

xap@CambridgeConsultants.com

www.CambridgeConsultants.com

C7432-UM-002 v1.49
1 September 2014

Revision History

Revision	Date	Details
v0.5	28 July 2005	Conforms to C7432-S-001 v0.11.
v0.6	10 August 2005	Conforms to C7432-S-001 v1.0.
v0.7	11 August 2005	Changes after internal review.
v0.8	15 August 2005	Conforms to C7432-S-001 v1.1.
v0.9	15 August 2005	Changes after internal review.
v0.10	29 September 2005	Minor changes.
v0.11	23 November 2005	Conforms to C7432-S-001 v2.0.
v1.0	24 November 2005	Changes after internal review.
v1.1	25 November 2005	Minor changes.
v1.2	25 November 2005	Minor changes.
v1.3	29 November 2005	Minor changes.
v1.4	19 December 2005	Minor changes.
v1.5	20 December 2005	Minor changes.
v1.9	12 December 2006	Minor changes.
v1.10	30 March 2007	Conforms to C7432-S-001 v3.8.
v1.11	3 May 2007	Conforms to C7432-S-001 v3.11.
v1.12	25 May 2007	Minor changes.
v1.13	9 August 2007	Minor changes.
v1.14	16 August 2007	Conforms to C7432-S-001 v3.17.
v1.15	16 August 2007	Conforms to C7432-S-001 v3.17.
v1.16	5 October 2007	Conforms to C7432-S-001 v3.17.
v1.17	5 October 2007	Formatting.
v1.18	22 October 2007	Conforms to C7432-S-001 v3.18.
v1.19	25 October 2007	Minor updates after review.
v1.20	26 October 2007	Formatting.
v1.21	31 October 2007	Formatting.
v1.22	1 November 2007	Formatting.
v1.23	6 November 2007	Minor changes.
v1.24	18 December 2007	Conforms to C7432-S-001 v3.20.
v1.25	21 December 2007	Minor changes.

V1.26	4 March 2008	Conforms to C7432-S-001 v3.25.
v1.27	6 March 2008	Conforms to C7432-S-001 v3.26.
v1.28	12 March 2008	Conforms to C7432-S-001 v3.27.
v1.29	13 March 2008	Conforms to C7432-S-001 v3.28.
v1.30	14 March 2008	Conforms to C7432-S-001 v3.29.
v1.31	4 April 2008	Minor updates after review. Conforms to C7432-S-001 v3.30.
v1.32	7 April 2008	Fix broken links.
v1.33	14 April 2008	Minor updates.
v1.34	20 June 2008	Conforms to C7432-S-001 v3.32.
v1.35	2 July 2008	Update legal notices.
v1.36	3 July 2008	Update legal notices.
v1.37	8 August 2008	Addition of CLU instructions.
v1.38	8 August 2008	Correction of corrupted formatting.
v1.39	6 October 2008	Add missing CLU instruction pages.
v1.40	22 January 2009	Correct encodings of clu.dds and clu.dss.
v1.41	8 May 2009	Added new instructions: sub.c.i, and.1.r, or.1.r, xor.1.r, bra.m and bsr.m. Conforms to C7432-S-001 v3.36, and XAP Architecture 4.3.
v1.42	14 May 2009	Update information on exceptions caused by clu instructions. Conforms to C7432-S-001 v3.37
v1.43	27 June 2011	Formatting and content corrections
v1.44	6 December 2011	Update Programmer's model diagram to reflect correct SP size
v1.45	16 August 2013	Updated to Word 2010
v1.46	16 August 2013	Minor corrections
v1.47	27 November 2013	Corrections and clarifications
v1.48	1 September 2014	Minor additions and corrections to sections 1-6.

Legal Notices

This is the Programmer's Manual for the XAP4 processor. You may use this if you accept the following conditions. If you do not accept these conditions, you must delete or destroy this copy of the XAP4 Programmer's Manual immediately.

Copyright: This manual is © Copyright Cambridge Consultants 2005-2014. You are authorised to open, view and print any electronic copy we send you of this manual within your organisation. Printouts of this manual must be kept within your organisation. Distribution of this manual, in whole or in part, to anyone outside your organisation is prohibited without prior written permission from Cambridge Consultants Ltd.

Fit for purpose: The XAP4 processor and the xIDE software development environment must not be used for safety critical and/or life support applications without a specific written agreement from Cambridge Consultants Ltd.

Liability: Cambridge Consultants Ltd. makes no warranty of any kind with regard to the information contained in this manual, including but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Cambridge Consultants Ltd. shall not be liable for omissions or errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material. The information contained herein may be updated from time to time.

Ownership and Licence: The XAP4 processor (and associated technologies including SIF and xIDE), XAP4 documentation and the xIDE software development environment and all intellectual property rights associated with them are proprietary to and owned by Cambridge Consultants Ltd and are protected by copyright law and international copyright treaties, patent applications, letters patent and other intellectual property laws and treaties. You must not attempt to use the XAP4 processor or the xIDE developer's environment unless you have a valid licence agreement with Cambridge Consultants Ltd. You must not use the information within this manual or other items supplied to you for the purposes of designing, developing or testing a device or simulator which is, or is intended to be, wholly or partly instruction set compatible with the XAP4 processor.

Trademarks: Cambridge Consultants, the Cambridge Consultants logo and XAP are registered trademarks and XAP1, XAP2, XAP3, XAP4, XAP5, SIF and xIDE are trademarks of Cambridge Consultants Ltd. All other trademarks mentioned herein are the property of their respective owners.

Cambridge Consultants Ltd
Science Park
Milton Rd
Cambridge CB4 0DW
England

www.CambridgeConsultants.com

Project Team

With thanks to the XAP4 project team: Alistair Morfey, Karl Swepson, Derek Henderson, Peter Lloyd, Chris Roberts, Robin Summerhill, Martin Cooper, Chris Turner, Hereward Mills, James Crosby, Jerome Woodwark, Nimrod Gileadi, Rodrigo Queiro.

Table of Contents

1 Introduction

1.1 This Document

This document is the primary reference for software engineers developing programs to run on XAP4 systems. It contains all the information needed to understand the XAP4 architecture from a software perspective.

This document conforms to XAP Architecture 4.3.

Further documents describe other aspects of XAP4 systems. Refer to section 1.5, "Other documentation" for details.

1.2 Assumptions

The reader is assumed to be familiar with microprocessor architectures in general, and to have some understanding of high level languages such as C. A detailed understanding of digital electronics is not required.

1.3 About XAP4

The XAP4 is a powerful 16-bit processor optimised for low gate count and low power. It has a von Neumann architecture and can address 64kB of linear byte-addressable memory.

High Code Density

The XAP4 architecture, instruction set, compiler toolchain and application binary interface are all designed for efficient execution of programs written in C.

Complex Systems

The XAP4 implements the features necessary to support complex software systems where high reliability or high integrity is a requirement. The XAP4 provides hardware support for common real time operating system primitives, including a clear definition of user and privileged modes, atomic instructions for synchronisation constructs, and position independent code to allow dynamic linkage.

The XAP4 MMU implements memory protection systems for instruction and data, throwing exceptions if processes access code or data memory illegally.

Interrupts and Exceptions

The XAP4 supports 32 exceptions and 16 interrupts. Each has its own handler defined in a vector table.

Rich RISC-like Instruction Set

A rich set of instructions is provided, including a complete set of branches and arithmetic operations on 8, 16 and 32 bit data. Several instructions map closely onto C language constructs thereby providing very high code density and fast execution.

16-bit and 32-bit Instructions

The XAP4 instruction set contains 174 unique instructions. Most common instructions are 16 bits long with less common instructions using 32 bits. Some instructions have 16-bit and 32-bit forms, with the toolchain using the 16-bit form whenever possible.

This approach gives high code density. Programs can mix 16-bit and 32-bit instructions at will. No switching between modes is necessary and the processor executes all instructions at full speed.

Hardware Acceleration

The processor includes a multiplier, divider and barrel shifter. Instruction execution is pipelined, resulting in operating speeds of greater than 100MHz on ASIC geometries of 0.13µm and below.

Targeted for Speed or Size

XAP4a is targeted for low cost applications and requires approximately 12,000 gates. It uses a 2-stage pipeline. Other implementations of the XAP4 architecture will be available in the future which target speed over size.

Language Support

The XAP4 GNU Compiler Collection (GCC) provides an industry standard ANSI C compiler.

Development Tools

The xIDE integrated development environment provides a cross-platform development and debugging tool for XAP4 developers, using an industry-standard look and feel.

1.4 The XAP Family

The XAP4 is the fourth member of the Cambridge Consultants XAP family of embedded processors for ASICs and FPGAs. All XAP processors use the Cambridge Consultants SIF interface for debugging and the xIDE development environment for software development.

Legacy Processors

The XAP1 is targeted at extremely small or extremely low power applications. It is a 16-bit Harvard architecture processor and fits in approximately 3000 gates. The XAP1 has been used in various ASIC projects since 1994.

The XAP2 is an evolutionary development from XAP1. It is again a 16-bit Harvard architecture processor but has a larger address space and better optimisation for C language applications. It requires approximately 12,000 gates. It was developed in 1999 and is used in many high-volume products.

Current Processors

The XAP3 is a completely new development which maintains the strategy of low power, low cost and low risk, whilst providing the size and sophistication to run complex applications. It is a 32-bit von Neumann processor and fits in approximately 30,000 gates. Variants of the XAP3 architecture are available which specifically target speed or size. The XAP3 supports C and C++.

The XAP4 is a 16-bit successor to the XAP3 and shares many of its architectural features to maintain the strategy of low power, low cost and low risk. It is capable of addressing 64 kB of memory and requires approximately 12,000 gates. The XAP4 has a GCC compiler.

XAP5, like XAP4, is an advanced 16-bit processor but with a 24-bit address bus. XAP5 can address 16 MB of memory and requires approximately 18,000 gates. The XAP5 has a GCC Compiler.

1.5 Other Documentation

The following documents describe the software tools and xIDE integrated development environment:

Document	Contains
xIDE User Manual C7066-TM-001	Describes the general features of xIDE.
xIDE Python Object Model C7066-TM-003	Describes the Python object model used within xIDE. This is essential for developers wanting to use the scripting features of xIDE. This allows developers to perform automated

	testing or to model other parts of a system during simulation.
xIDE for XAP4 C7066-TM-017	Describes specific features of the XAP4 plugin for xIDE.
XAP4 Binutils Manual C7432-UM-003	Describes the XAP4 Assembler, Linker and other binary utilities.
XAP4 GCC Manual C7432-UM-004	Describes the XAP4 ANSI C compiler
XAP4a Instruction Set Quick Reference Card C7432-UM-005	A short guide to the XAP4 instruction set.

For details of the XAP4 hardware, refer to these documents:

Document	Contains
XAP4a Hardware Reference Manual C7432-UM-006	Information needed by digital designers using the XAP4a in an FPGA or ASIC
XAP4 Datasheet ASICs-SB-011	A short overview of the XAP4 architecture.
xEMU mini User Manual C7245-UM-016	Describes the xEMU mini configuration delivered as part of the XAP4, XAP5 and XAP6 processor IP.

1.6 For Further Help

The XAP4 documentation set provides answers to most questions.

If a problem remains unsolved, contact Cambridge Consultants technical support at:

xap@CambridgeConsultants.com

1.7 Glossary and Acronyms

Term	Meaning
Double-Word	A 4-byte (or 32-bit) quantity.
Word	A 2-byte (or 16-bit) quantity. The XAP4 is a 16-bit processor.
Byte	An 8-bit quantity.

Little Endian	A system in which the least significant byte of a word is stored at the lowest memory address. The XAP4 is a little endian processor.
Big Endian	A system in which the most significant byte of a word is stored at the lowest memory address.
ALU	Arithmetic and Logic Unit.
IVC	Interrupt Vector Controller.
MMU	Memory Management Unit.
NMI	Non-Maskable Interrupt.
MI	Maskable Interrupt.
IRQ	Interrupt Request.
Set	In connection with the flags, set indicates a logic 1 in the flag.
Clear	In connection with the flags, clear indicates a logic 0 in the flag.

2 XAP4 Systems

2.1 System Peripherals

The XAP4 exists as a soft IP core written entirely in Verilog. This can be targeted at either system-on-chip (ASIC) or FPGA implementations.

The xIDE toolset includes an instruction set simulator for XAP4. This can be extended to include models of other parts of the system, including memories and interrupts.

The XAP4 normally exists with other standard components in a system. These are:

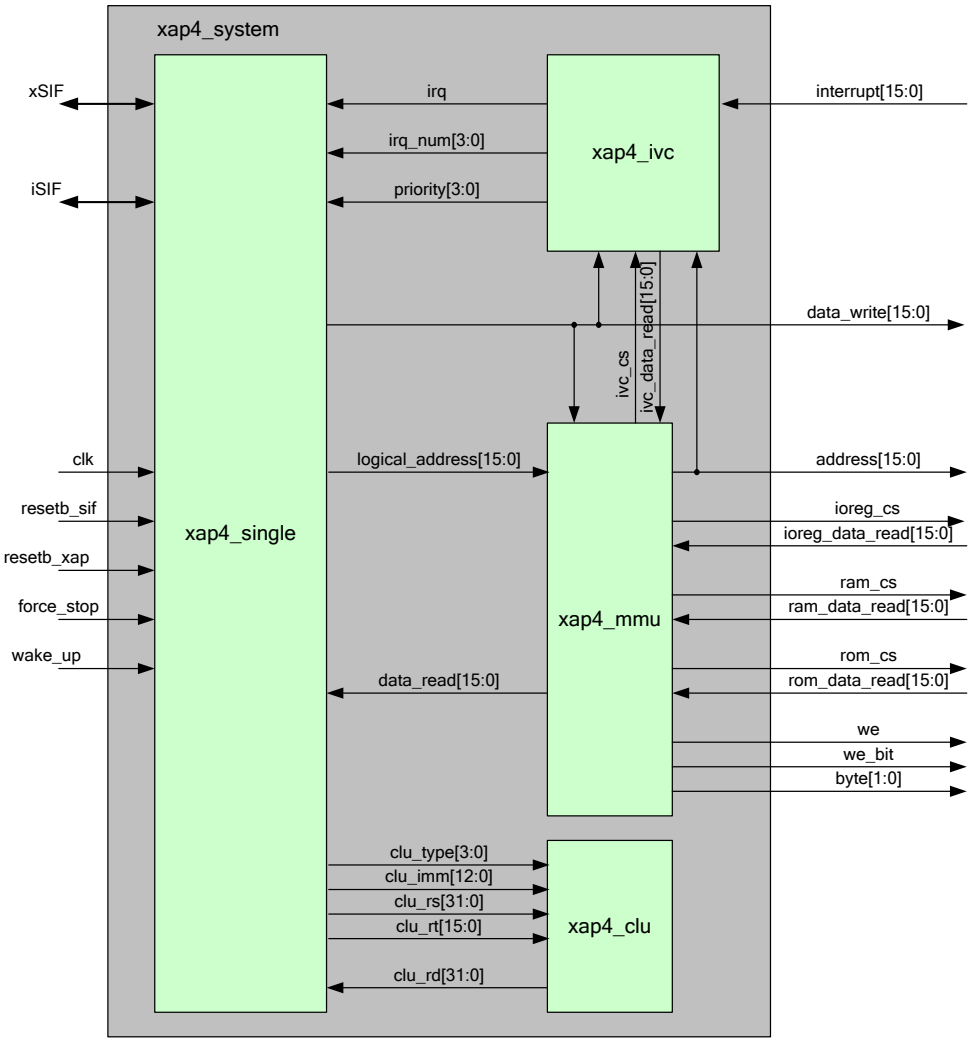
- A Memory Management Unit (MMU) to interface with memories, typically flash and RAM.
- An Interrupt Vector Controller (IVC) to prioritise interrupt sources and provide an interrupt number and priority to the XAP4.

The contents of both the MMU and the IVC can be customised for the specific memory and interrupt needs of a particular application.

Application-specific circuitry is needed for:

- System control - clock and reset generation, watchdog functions.
- Application-specific memories, peripherals and interrupt sources.

The XAP4 can support a Custom Logic Unit (CLU) as an optional component. The CLU is the external equivalent of the ALU (internal Arithmetic Logic Unit). It allows users to develop their own custom instructions that can use the general purpose registers R0-R7.



The XAP4 can address 64 kB of linear memory space, normally populated with a mixture of Flash, RAM and I/O registers. Code and data can exist in the same memory.

The address from `xap4_single` is called the Logical Address. All of the processor operation and xIDE debug tools refer to this Logical Address.

Sections 2.2 to 2.4 outline the default XAP4 System modules.

The remainder of this document concentrates solely on the XAP4 core and SIF. Further information on the system peripherals can be found in the XAP4a Hardware Reference Manual, C7432-UM-006.

2.2 Interrupt Vector Controller (IVC)

There are two Interrupt Vector Controllers available; the Basic Module and the xEMU mini module.

2.2.1 Basic Module

The IVC contains registers to configure the 16 Interrupt inputs:

- Enable or disable each one.
- Hold the status for each one.
- Clear each one after being processed.

2.2.2 xEMU mini Module

In addition to the capabilities of the Basic module, the xEMU mini module can set the priority (4 bit) for each one.

For further information see xEMU mini User Manual, C7245-UM-016.

2.3 Memory Management Unit (MMU)

There are two MMU modules available; the Basic module, and the xEMU mini module.

2.3.1 Basic Module

The Basic MMU contains circuitry for:

- Managing wait states.

2.3.2 xEMU mini Module

The xEMU mini MMU contains registers to define valid address regions for:

- User Data access.
- User Program access.
- Privileged Data access.
- Privileged Program access.

For further information see xEMU mini User Manual, C7245-UM-016.

2.4 Custom Logic Unit (CLU)

The CLU hardware decodes the instruction based on the type and the immediate passed. Together they define the custom instruction to be used. Rd, Rs and Rt are then parameters of that custom instruction.

There are fifteen CLU instruction types, each passing a different combination of source and destination registers to the CLU. The instruction types are:

Mnemonic	Operands	Type	Rd regs	Rs regs	Rt regs
non clu* instructions		0			

Mnemonic	Operands	Type	Rd regs	Rs regs	Rt regs
clu	#imm	1	0	0	0
clu.d	#imm, Rd	2	1	0	0
clu.dd	#imm, Rd	3	2	0	0
clu.s	#imm, Rs	4	0	1	0
clu.ss	#imm, Rs	5	0	2	0
clu.st	#imm, Rs, Rt	6	0	1	1
clu.sst	#imm, Rd, Rt	7	0	2	1
clu.ds	#imm, Rd, Rs	8	1	1	0
clu.dss	#imm, Rd, Rs	9	1	2	0
clu.dds	#imm, Rd, Rs	10	2	1	0
clu.ddss	#imm, Rd, Rs	11	2	2	0
clu.dst	#imm, Rd, Rs, Rt	12	1	1	1
clu.dsst	#imm, Rd, Rs, Rt	13	1	2	1
clu.ddst	#imm, Rd, Rs, Rt	14	2	1	1
clu.ddsst	#imm, Rd, Rs, Rt	15	2	2	1

CLU instructions may throw `InstructionError`. Refer to section 3.8.9, “Error Details”.

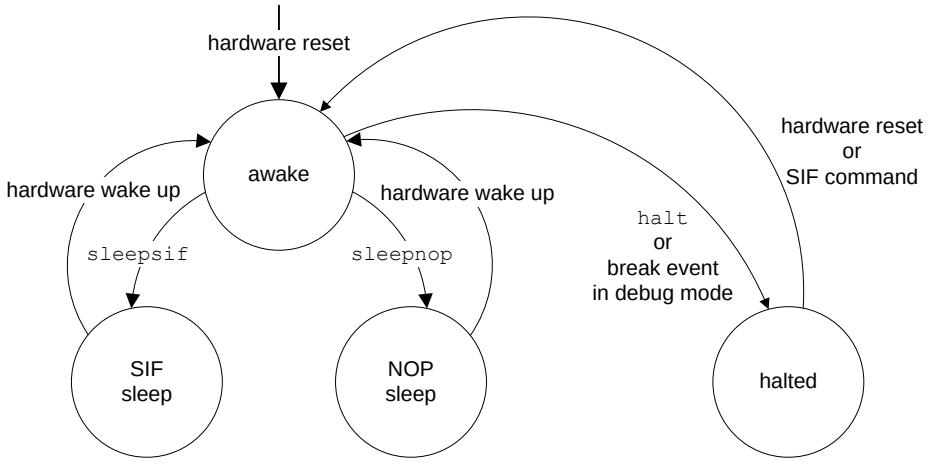
Further information about the CLU can be found in the XAP4a Hardware Reference Manual C7432-UM-006.

3 Programmer's Model

3.1 Processor Execution States

The XAP4 has four processor execution states:

Processor State	Description
Awake	The XAP4 is able to execute instructions. This is the XAP4 processor state following a reset or after a wake up from one of the sleep states.
SIF Sleep	The XAP4 is asleep but SIF cycles can still occur. This state is entered with the `sleepsif` instruction.
NOP Sleep	The XAP4 is asleep and SIF cycles are not permitted. This state is entered with the `sleepnop` instruction.
Halted	The XAP4 is stopped. This state is entered with the `halt` instruction or after a `Break` event when in debug mode. Refer to section 3.4.4 "Breakpoint Registers", for details of break events.



Returning to the awake state from any other state requires hardware activity. Power consumption is significantly reduced in the two sleep states.

3.2 Processor Operating Modes and States

The mode/state of the processor is dependant on INFO[NL], INFO[R], FLAGS[M1:0].

Mode/State	INFO[NL]	INFO[R]	FLAGS[M1:0]
User mode	0	0	3
Trusted mode	0	0	2
Supervisor mode	0	0	0
Interrupt mode	0	0	1
Recovery state	0	1	X
NMI state	1	X	X

For details of how mode changes take place, see section 3.8, "Interrupts and Exceptions"

3.2.1 Capabilities of the Processor Modes and States

The processor's mode/state affects which instructions can be used and which stack is used.

Mode/State	Instructions	Stack
User mode	User	Stack1
Trusted mode	User and Privileged	Stack1
Supervisor mode	User and Privileged	Stack0
Interrupt mode	User and Privileged	Stack0
Recovery state	User and Privileged	Stack0
NMI state	User and Privileged	Stack0

User mode

Code running in User mode cannot affect the operation of code running in the other modes. Some instructions are not permitted in User mode.

Code running in User mode cannot access all registers. It can access registers R0-R7 and the User mode stack pointer (SP1). It can also perform a limited set of operations on the FLAGS register.

User mode is suitable for untrusted application code.

Trusted mode

Code running in Trusted mode can execute all instructions and can therefore control the activity of code running in User mode.

Trusted mode is entered when a system call is made from User mode.

Trusted mode is suitable for operating system services.

Supervisor mode

Code running in Supervisor mode can execute all instructions and can therefore control the activity of code running in User mode.

Supervisor mode is entered after a Hard or Soft reset or when an error occurs in User mode.

Supervisor mode is suitable for operating system services.

Interrupt mode

Interrupt mode is entered on a maskable hardware interrupt.

Interrupt mode is suitable for writing interrupt handlers.

Recovery state

Recovery state is entered when errors occur in Supervisor or Interrupt mode.

It is intended to be used for writing short fast handlers to recover from errors.

NMI state

NMI state is entered when an NMI occurs in any other mode/state.

It is intended to be used for writing the handlers for the NMI events.

3.2.2 Typical usage of Processor Modes and States

Simple Applications

Simple applications may execute in Supervisor mode and make no use of User and Trusted mode. Interrupt mode is used for interrupt handlers. Any errors are likely to just halt the processor or be ignored.

Complex Applications

Complex applications are likely to contain a task scheduler as part of an operating system. Operating systems can choose whether to allow tasks full access to the processor and resources or whether to enforce restrictions. This is done by changing the mode from which the tasks execute.

If no restrictions are enforced the tasks will be running in Trusted mode and the services provided by the operting system will execute in Supervisor mode.

Tasks can be restricted by running them in User mode, which is not a privileged mode. Operating system services will execute in Trusted mode. In addition, MMU restrictions can allow an operating system to have complete control of User mode code such that it will be unable to affect the operation of the rest of the system.

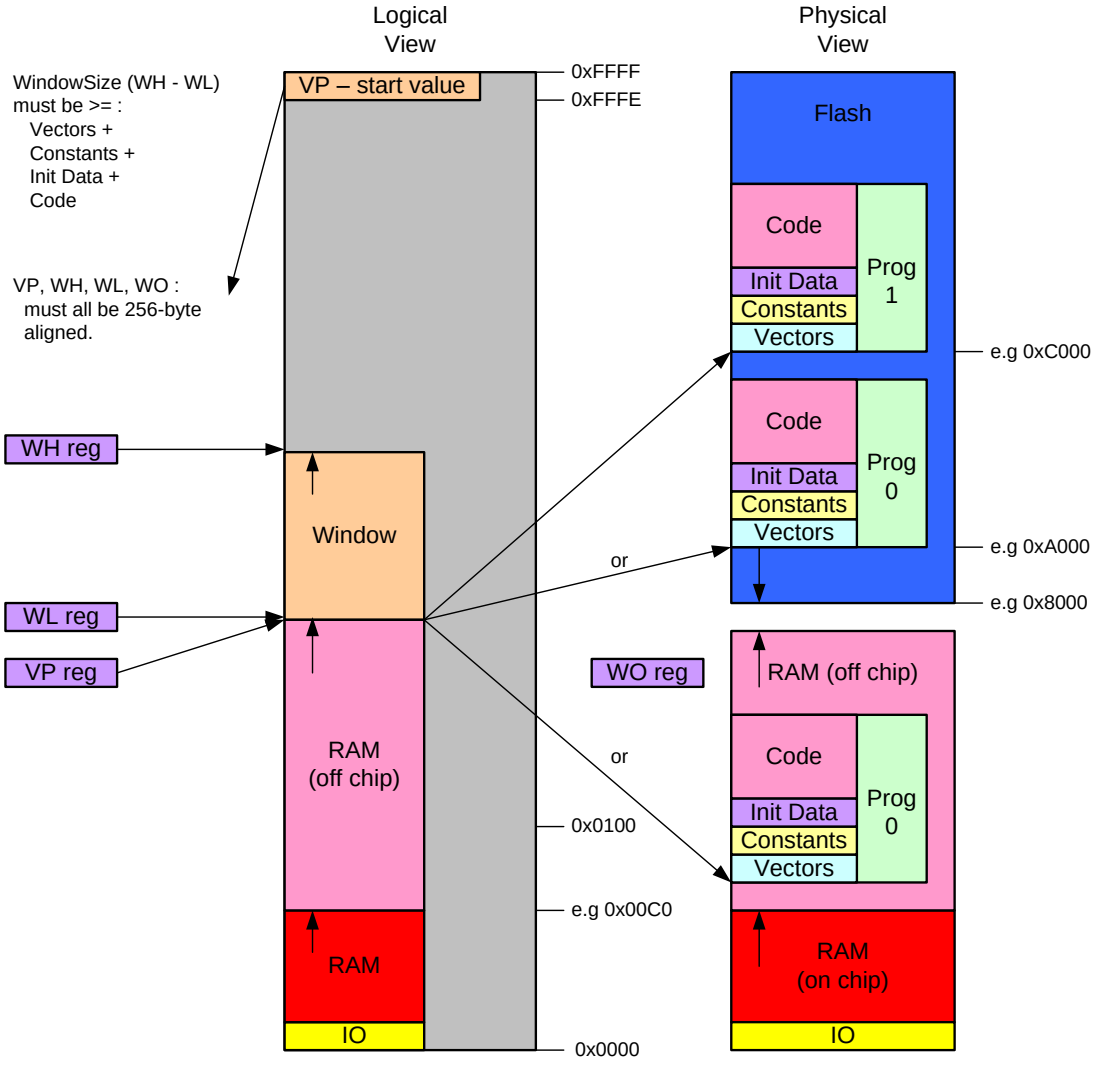
3.3 Memory Architecture

3.3.1 Program Relocation

The Vector Pointer indicates the start address for the vector table. We recommend that this is followed by constants, initial data and code (to keep the whole program as a single contiguous block). For details for how VP is stored and aligned, see section 3.4.2, "Address Registers".

The diagram below shows how the XAP4 supports position-independent code.

Section	Relative
Vectors	Zero-relative (Function table entries accessed through window)
Code	PC-relative
Constants	Zero-relative (through window)
Global variables & heap	Zero-relative
Stack	SP-relative (initial value formed zero-relative)
IO registers	Zero-relative

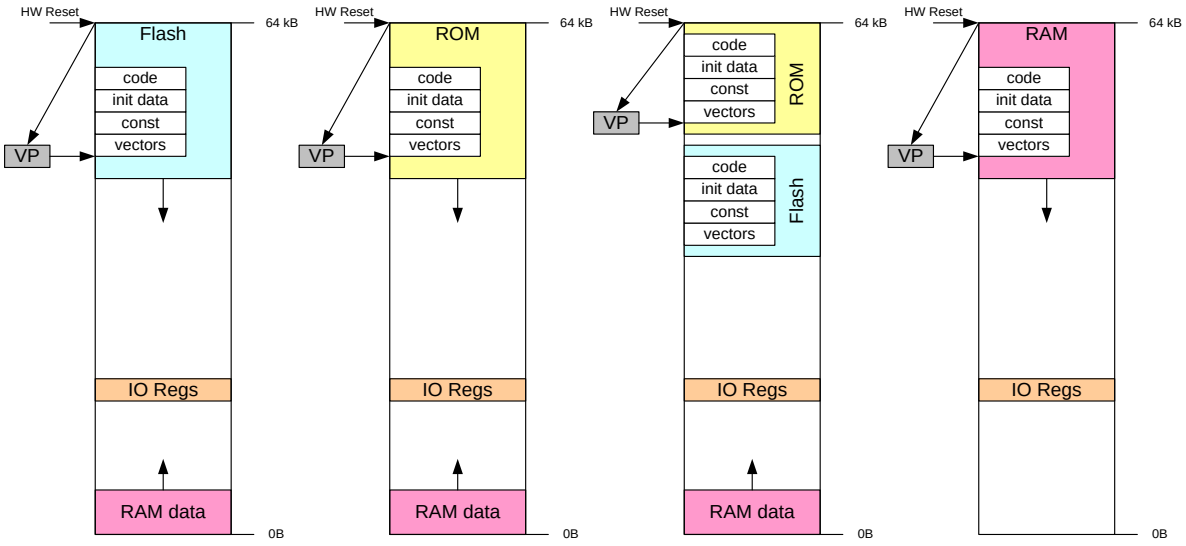


Global variables and the heap (stored in RAM) have their position defined at link-time, and are accessed with zero-relative addressing. IO registers are also accessed with zero-relative addressing, since their addresses are constant for a given hardware design. Vectors are accessed relative to VP by the hardware, and contain the absolute (zero-relative) addresses of the targets.

The figures below show the recommended memory maps for the four most common configurations.

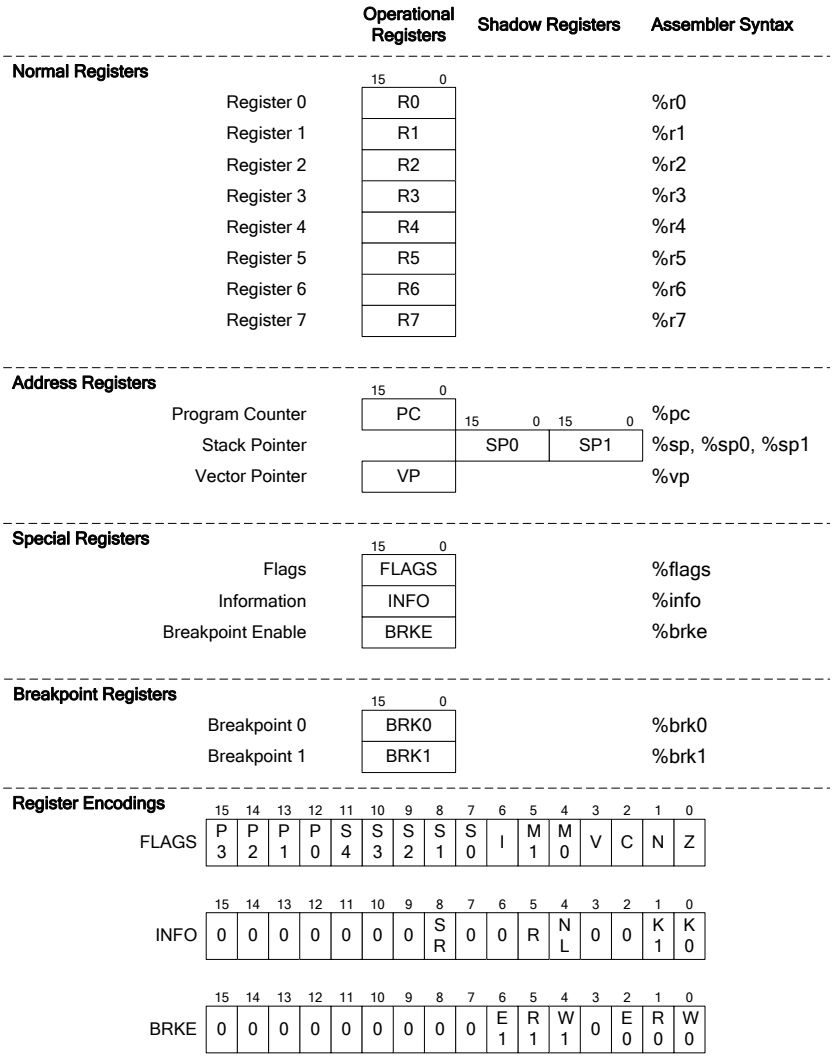
Recommended Memory Layouts

- It is best to avoid splitting memories into multiple mappings to allow simple MMU implementations.
- The location of VP is fixed so any non-volatile memory is mapped to the top of memory.



3.4 Registers

The XAP4 register set is shown below.



16-bit Normal registers

- Eight Normal registers, called R0 to R7.

16-bit Address registers

- The Program Counter, called PC.
- The Stack Pointers, called SP1 and SP0.
- The Vector Pointer, called VP.

16-bit Special registers

- Processor status register, called FLAGS.
- Read-only status register, called INFO.
- Breakpoint Enable register, called BRKE.

16-bit Breakpoint registers

- Two 16-bit Breakpoint registers, called BRK0 and BRK1.

In the descriptions of the registers which follow, hardware registers are in upper case (for example, R1) whilst the assembly name for a register is in lower case (for example, `%r1`).

3.4.1 Normal registers

The XAP4 has eight normal registers, called R0 to R7. These form the normal register operands for the majority of the instruction set.

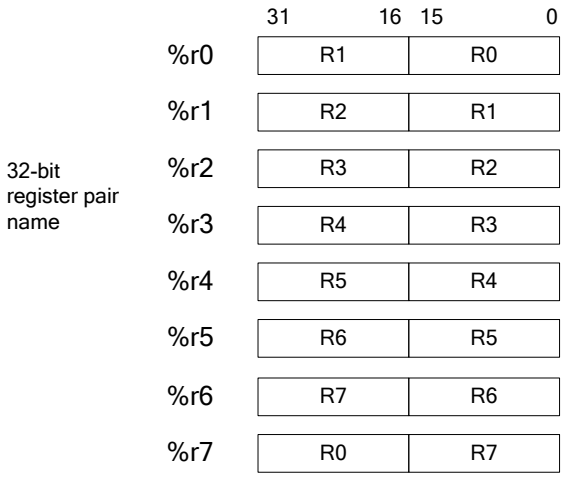
When an interrupt or exception changes the processor mode, R0 and R1 are copied to the new mode's stack. When the service routine completes (with an `rtie` instruction), the registers are restored to their previous values from the stack.

Assembly name	Hardware register name	Notes
`%r0`	R0	Used in the compiler calling convention for function arguments and return values.
`%r1`	R1	
`%r2`	R2	
`%r3`	R3	
`%r4`	R4	General purpose registers, preserved by functions.
`%r5`	R5	
`%r6`	R6	
`%r7`	R7	

32-bit register pairs

For some instructions, the XAP4 uses adjacent pairs of 16-bit registers to create a 32-bit register pair. The 32-bit register pair is referenced by the name of the lower register.

The register pairs available are:



The 32-bit register pairs are used in the `b2swap.32.*`, `div.32.*`, `divrem.32.*`, `ld.32.*`, `mov.32.*`, `mult.32.*`, `rem.32.*`, `rotatel.32.*`, `shiftl.32.*` `shiftr.32.*` ,, and `st.32.*` instructions.

3.4.2 Address Registers

The XAP4 has four 16-bit address registers – a Program Counter (PC), two Stack Pointers (SP0 and SP1) and the Vector Pointer (VP).

Address Register Summary

Assembly name	Hardware register Name	Notes
`%pc`	PC	The program counter. This is used in all processor modes and is not accessible directly, although there are a range of PC-relative instructions.
`%sp`	SP0 or SP1	The hardware register used as `%sp` depends on the processor mode.
`%sp1`	SP1	User mode and Trusted mode stack pointer. In these modes, this register is accessed as `%sp`.
`%sp0`	SP0	Supervisor mode, Interrupt mode, Recovery state and NMI state stack pointer. In these modes and states, this register is accessed as `%sp`.
`%vp`	VP	The vector pointer. This can be accessed from privileged modes as `%vp`.

Program Counter

There is a single 16-bit program counter which is used by all processor modes. The program counter normally points to the next instruction to be executed. It is implictly used by many instructions, including the PC-relative variants of the `*.i` instructions and the `rtie` instruction.

The XAP4 forces the least-significant bit of the program counter to be zero. Attempts to set the least significant bit throw an `AlignError` exception.

Stack Pointers

There are two hardware stack pointer registers – SP0 and SP1. The register used depends on the current processor mode.

SP0 is shared between Supervisor mode, Interrupt mode, Recovery state and NMI state; SP1 is shared between User mode and Trusted mode.

SP is always even, this ensures that it is word-aligned for speed. As such, SP[0] is always zero.

When `%sp` is used as an operand (either implicity or explicitly), the currently active SP register is used. The `movr2a.*` and `mova2r.*` instructions allow the privileged modes to access each of the two stack pointers explicitly.

Vector Pointer

VP points to the base of the vector table. The convention is to group a program (vectors, constants, initialisation data, code) into a contiguous block. In this case, VP points to the base of the whole program. The hard reset reads the initial value of VP from memory at address 0xFFFE For details of how this is used, see section 3.3.1, "Program Relocation".

The least significant byte VP[7:0] is always zero. VP[15:8] is the register.

VP can also be accessed from the privileged modes with the `movr2a` and `mova2r` instructions.

3.4.3 Special registers

The XAP4 has a FLAGS register containing processor state information, a read-only INFO register also containing processor state information, and registers controlling the hardware breakpoint function.

FLAGS register

The FLAGS register contains condition code bits and other bits related to the processor state.

When an interrupt or exception changes the processor mode, the FLAGS register is copied to the Stack. When the event handler completes (with an `rtie` instruction), the FLAGS are copied back from the stack.

Assembly name	Hardware register Name	Notes
`%flags`	FLAGS	The flags register. This is used in all processor modes.

Bits in the FLAGS register

Bit(s)	Flag	Name	Description
0	Z	Zero	Updated by most instructions according to the result of their operation. Refer to the individual instruction descriptions in section 7, "Instruction Set Reference" for details of the flag behaviour.
1	N	Negative	
2	C	Carry	
3	V	Overflow	
5:4	M[1:0]	Mode	Indicates the current processor operating mode: 00 Supervisor mode 01 Interrupt mode

Bit(s)	Flag	Name	Description
			10 Recovery mode 11 User mode 0 – 2 = Privileged modes. 3 = Non-privileged mode. 0 - 1 use Stack0 and SP0. 2 - 3 use Stack1 and SP1. Only applies when the Proccessor is not in Recovery or NMI state. See NL and R bits in the INFO register.
6	I	Interrupt Enable	If set, maskable interrupts are enabled. Non-maskable interrupts (NMIs) cannot be disabled. (These are interrupts numbered 0-1). The I bit is cleared when an interrupt or exception occurs.
11:7	S[4:0]	State	The S bits are used to maintain the state of multi-cycle instructions (e.g. `push`, `pop`, `pop.ret`) when an interrupt or exception occurs. Set these bits to zero when initialising the FLAGS register. It is not normally necessary for a program to manipulate these bits.
15:12	P[3:0]	Priority	These bits are set by the Context Pushes of Interrupts and Error Exceptions. The meaning of these bits depends on the current mode. See the table below.

The Priority bits P[3:0] are interpreted as follows:

Mode/state	Meaning of P[3:0]
User	P[0]: B: Break Controls response to `brk` instruction or hardware breakpoints. P[1]: T: Single Step If set, each User mode instruction that is executed throws the `SingleStep` exception. P[3:2]: Reserved
Trusted	Reserved.
Supervisor Recovery	Indicates additional information about an Error Exception. This value is called ErrorPval. For details, see section 3.8.2 “Exceptions”
Interrupt NMI	Priority level of current interrupt: ■ 0 = Most important maskable interrupt (also NMIs). ■ 15 = Least important maskable interrupt.

The following events cause the `FLAGS` register to be updated:

- Hard and soft resets.
- An instruction which updates the flags.
- An instruction is executed that writes to the FLAGS register directly (such as `movr2s`, `mov.1.i`, `mov.2.i` or `mov.4.i`).
- An interrupt or exception.
- The `rtie` instruction.

If the Rd operand of an instruction is an address register then the flags are not updated.

INFO Register

This register contains read-only information about the processor state. It allows the software to read information that it cannot freely change.

Bits in the INFO register

Bit(s)	Flag	Name	Description
0	K0	Stack0 – Enable	This bit is cleared on Hard and Soft Reset. Set to 1 by a successful write to SP0 (StackPointer0) with `movr2a`. Once set to 1, it cannot be set back to 0. Context Pushes to Stack0 cannot be performed until K0 is set. If a context push is attempted before K0 is set a Soft Reset wil be generated. Context pushes are necessary for Interrupts and Exceptions. MIs and NMIs will remain in a pending state until Stack0 is initialised and exceptions will cause a Soft Reset. When K0 is 0, only `movr2a` and `mova2r` instructions can use SP0. Any other instruction using SP0 causes a Soft Reset. The instruction is not completed and the registers are not updated. SIF memory reads and writes are unaware of whether the memory region is part of Stack0, so are not affected by K0.

Bit(s)	Flag	Name	Description
1	K1	Stack1 - Enable	This bit is cleared on Hard and Soft Reset. Set to 1 by a successful write to SP1 (StackPointer1) with `movr2a` if K0 = 1. If K0 = 0, the `movr2a` will update SP1 but will leave K1 at 0. This means that K1 can only be set to 1 after K0 has been set to 1. Once set to 1, it cannot be set back to 0. When K1 is 0, only `movr2a` and `mova2r` instructions can use SP1. Any other instruction using SP1 causes an exception. SIF memory reads and writes are unaware of whether the memory region is part of the Stack1, so are not affected by K1.
3:2			FREE
4	NL	NMI-Lock	This bit is cleared on Hard and Soft Reset. It is set when an NMI occurs, preventing further events (non maskable or maskable interrupts, exceptions etc.) from interrupting the NMI service routine. When this bit is set, execeptions cause a Soft Reset. It is cleared by executing the `rtie` instruction in NMI state.
5	R	Recovery	This bit is cleared on Hard and Soft Reset. Set to 1 when an Error Exception occurs in Supervisor or Interrupt mode. When this bit is set, exceptions cause a Soft Reset. It is cleared by executing the `rtie` instruction in Recovery state.
7:6			FREE
8	SR	Soft-Reset	This bit is cleared on Hard Reset. It is set when a soft reset happens. A bug-free program should never cause a Soft Reset, so this bit should always be 0. If the programmer sees that this bit is 1, they know that there has been 1 or more Soft Resets since the XAP received a Hard Reset. This tells them that there is a software bug that needs to be investigated.
15:9			FREE

Breakpoint enable register

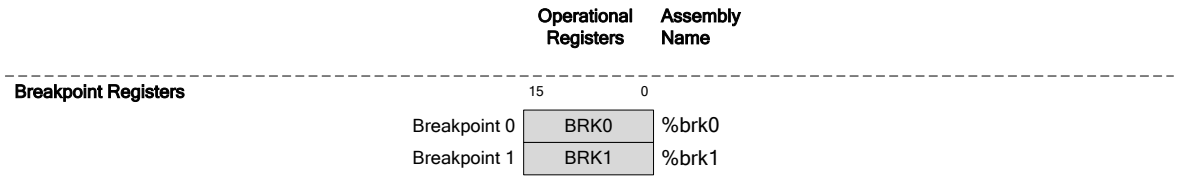
The BRKE register is used in conjunction with the breakpoint address registers. It is accessed with the `movr2s` and `movs2r` instructions and is accessible from the privileged modes only.

Bits in the BRKE register

Bit(s)	Flag	Name	Description
0 4	W0 W1	Write	These two bits enable data write breakpoints for BRK0 and BRK1 respectively.
1 5	R0 R1	Read	These two bits enable data read breakpoints for BRK0 and BRK1 respectively.
2 6	E0 E1	Execute	These two bits enable execution breakpoints for BRK0 and BRK1 respectively.
3 7:15			FREE

3.4.4 Breakpoint Registers

The XAP4 has two breakpoint address registers. They are accessed with the `movr2b` and `movb2r` instructions and are accessible from the privileged modes only.



Along with the BRKE special register described above, these can be used for both stop-mode and run-mode debugging.

For each breakpoint address register, there are read, write and execute bits in the breakpoint enable register. A break event will occur when an address stored in a breakpoint address register is read, written or executed, and the corresponding bit in BRKE is set.

Note: For execute break conditions, the break event will occur before the instruction is executed. For read/write break conditions, the break event will occur after the instruction has executed, and the PC will point to the following instruction.

3.5 Pipeline

The XAP4 is a pipelined processor. XAP4a has a two-stage pipeline. For further details, including details of instruction cycle counts, refer to the appropriate Hardware Reference Manual.

3.6 Stack Operation

The XAP4 stacks start at high addresses and grow downwards in memory. The stacks must be aligned to a word boundary, as SP1 and SP0 can only store even addresses. XAP4 GCC maintains a word aligned stack, where push operations decrease the stack pointer by a multiple of two bytes, and pop operations increase the stack pointer by a multiple of two bytes.

A dedicated 16-bit register is used as the stack pointer and points to the last used location on the stack. Separate stacks exist for Supervisor mode and Interrupt mode, and Recovery and NMI state (SP0), and for User mode and Trusted mode (SP1).

The compilers operate a fully covered stack. When accessing stack data (local automatic variables or parameters), stack offsets are always positive or zero.

See section 3.4 for more information.

3.7 Reset

In addition to conventional, 'hard resets', the XAP4 can do a 'soft reset', which is forced by the hardware if error or service events occur in Recovery or NMI state, or, if the `softreset` instruction is executed in a privileged mode. See section 3.8.2, "Exceptions".

3.7.1 Hard reset

Following a hard reset, the XAP4 state is as follows:

- The Processor State is Awake.
- The Processor mode is Supervisor.
- All registers are zero.

The XAP4 then takes the following steps:

- Loads the initial VP value from 0xFFFE to VP.
- Loads the HardReset handler address from (VP + 0x2) to PC.

With the FLAGS register being zero, interrupts are disabled, and on a switch to User mode, single stepping and breakpoints are disabled.

3.7.2 Soft reset

The soft reset mechanism exists to try and protect the XAP4 system in case of an errant exception or NMI handler.

A soft reset is triggered by:

- Exceptions when the the stack to be used for the context push is not initialised
- Memory errors during a context push
- Attempts to use the stack pointer before the stack is initialised
- The `softreset` instruction
- Service or Error events in Recovery or NMI state

Following a soft reset, the only changes in the XAP4 state are as follows:

- The FLAGS and BRKE registers are set to 0.
- The SR bit of the INFO register is set to 1; all other bits are set to 0.
- VP is reloaded from memory as for a hard reset.
- R0 contains the value of the FLAGS register at the time of the error.
- R1 contains the error code:
- R2 contains the value of the BRKE register at the time of the error.
- BRK0 contains the value of the PC register at the time of the error.
- BRK1 contains the value of the VP register at the time of the error.
- All other registers remain as they were before the soft reset.

The XAP4 then takes the following steps:

- Loads the initial VP value from 0xFFFE to VP.
- Loads the SoftReset handler address from (VP + 0x4) to PC.

R1 Error Code

The bits in R1 are used as shown below.

Bit(s)	Name	Description
4:0	Event No.	This indicates which event of the given event type caused the soft reset: Event No can have values 0 – 31. Details of Event No values for each of the 8 Event Types are given in the Event No. table below.
7:5		FREE

Bit(s)	Name	Description
10:8	Event Type	This indicates the type of the event that caused the soft reset: 0: Exception (when relevant StackPointer has been initialised. i.e K0=1 or K1=1). 1: Free. 2: Stack0 errors. 3: Stack1 errors. 7:4: Free.
11	Soft-Reset	The value of INFO[SR] before the error.
12	Stack0-Enable	The value of INFO[K0] before the error.
13	Stack1-Enable	The value of INFO[K1] before the error.
14	NMI-Lock	The value of INFO[NL] before the error.
15	Recovery	The value of INFO[R] before the error.

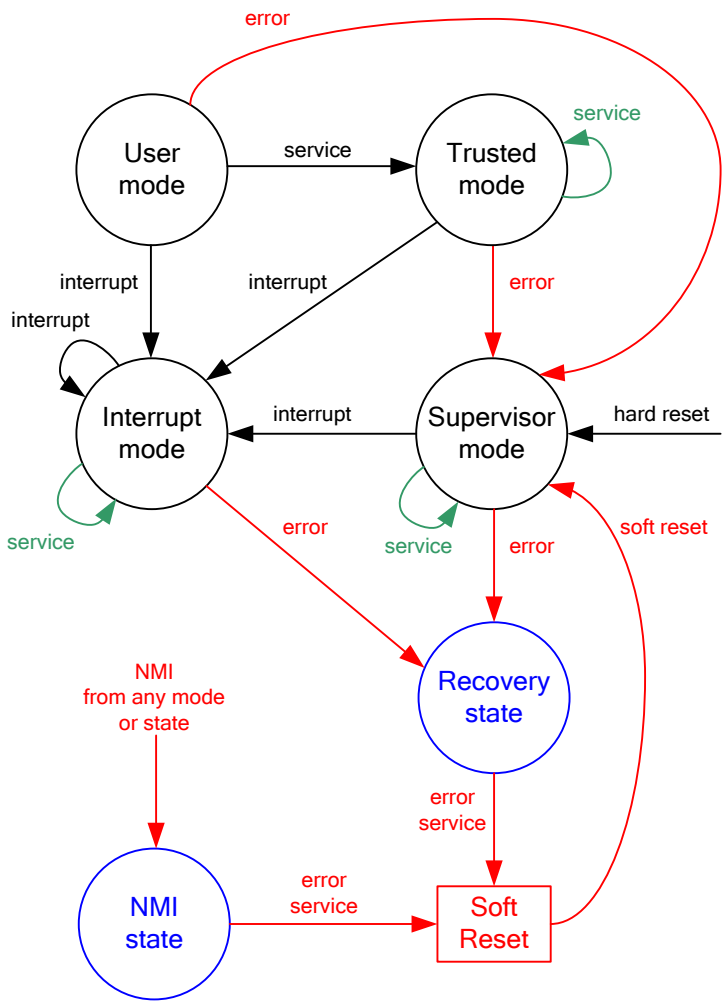
The table below shows the values used for Event Type and Event No.

Event Type	Event No	
0	Exception Number (0-31). Uses exception numbers (2, 19 - 31) . See section 3.8.3, Vector table.	
1	FREE	
2	0	Memory error during context push to Stack0 (when K0=1).
	1	Exception to Stack0 when SP0 is not initialised (K0=0).
	2	SP-relative instruction to Stack0 when SP0 is not initialised (K0=0).
	31:3	FREE
3	0	FREE
	1	Exception to Stack1 when SP1 is not initialised (K1=0).
	2	SP-relative instruction to Stack1 when SP1 is not initialised (K1=0).
	31:3	FREE
4-7	FREE	

3.8 Interrupts and Exceptions

Interrupts and exceptions are referred to collectively as events. With the exception of resets, they both involve a context switch, where the processor state is pushed to the stack of the destination mode and a handler is executed. The handler should then restore the processor state with the `rtie` instruction.

This diagram shows how different events result in mode changes:



Black = normal events

Red = error events

Green = other supported events

The mode/state of the processor is dependant on INFO[NL], INFO[R], FLAGS[M1:0].

Mode/State	INFO[NL]	INFO[R]	FLAGS[M1:0]
User mode	0	0	3

Trusted mode	0	0	2
Supervisor mode	0	0	0
Interrupt mode	0	0	1
Recovery state	0	1	X
NMI state	1	X	X

The table below summarises the mode and state transitions.

		From					
		User mode	**Trusted mode**	**Supervisor mode**	**Interrupt mode**	**Recovery state**	**NMI state**
To	**User mode**						
	Trusted mode	Service	Service				
	Supervisor mode	Hard Reset Error	Hard Reset Soft Reset Error	Hard Reset Soft Reset Service	Hard Reset Soft Reset	Hard Reset Soft Reset	Hard Reset Soft Reset
	Interrupt mode	Interrupt	Interrupt	Interrupt	Interrupt Service		
	Recovery state			Error	Error		
	NMI state	NMI	NMI	NMI	NMI	`NMI`	

In addition, the `movr2s`, `mov.2.i` and `rtie` instructions can be used to change from any privileged mode to any other mode.

3.8.1 Interrupts

Interrupts are a response to an event outside the XAP4 core, and can occur in any mode.

Interrupts 0 and 1 are non-maskable interrupts (NMIs), and cannot be disabled by software. Interrupts 2 to 15 are referred to as maskable interrupts (MIs). While an NMI is being serviced, it cannot be interrupted. Interrupts which occur during this time are remembered and are serviced after the handler returns. If an exception occurs in an NMI handler, a soft reset is triggered.

When an MI is serviced, the XAP4 moves to Interrupt mode and disables interrupts. The XAP4 receives an interrupt number (typically from an external interrupt controller) and branches to an appropriate interrupt handler. At the end of the interrupt processing, the XAP4 returns to the interrupted code with the `rtie` instruction. This restores the processor status to that prior to the interrupt.

When an NMI is serviced, the XAP4 moves to NMI state and sets the NMI-lock bit, INFO[NL], to 1.

All maskable interrupts can be nested. Nested interrupts can be achieved by re-enabling interrupts in an interrupt handler. Maskable interrupt handlers can then be interrupted by maskable interrupts of greater importance (lower priority number) and NMIs. Services can be called in a Maskable Interrupt handler, but will cause a

soft reset if in an NMI handler. Note: Priorities are only meaningful for maskable interrupts – not for exceptions or NMIs.

Interrupts are disabled by all events. The previous state is stored to FLAGS[C]: FLAGS[C] = FLAGS[I]. The handler can restore the I bit using `mov.1.r %flags[i], %flags[c]`.

3.8.2 Exceptions

Exceptions are caused by internal events, and are split into resets, services and errors. The various details depend on the type.

Resets

Resets are either hard resets or soft resets, and result in a mode change to Supervisor mode. For details, see section 3.7 "Reset".

Services

Services are used for calls to operating system functions, to run privileged code. In User mode they cause a switch to Trusted mode, but in privileged modes, no mode change is required. Services from NMI or Recovery state cause a Soft Reset.

Errors

Errors indicate a flaw in the code, or a problem with the processor. They include null pointer accesses and unknown instruction decodes. Errors from User and Trusted modes go to Supervisor mode. Errors from Supervisor and Interrupt modes stay in the same mode but add Recovery state (INFO[R]=1). Errors from NMI or Recovery state cause a Soft Reset.

The FLAGS[P] bits are used to pass additional diagnostic information about what the processor was doing when the error happened.

For all errors apart from an `InstructionError` this is set as follows:

Value of P[3:0]	Meaning of P[3:0]
0	All other cases
1	Error happened during a Context Push
2	Error happened during an `rtie` instruction

For an `InstructionError` this is set as follows:

Value of P[3:0]	Meaning of P[3:0]
0	`InstructionError` is not from CLU
1	Unknown CLU instruction
2	CLU instruction execution error

The software error handler may choose to perform different actions for different values of ErrorPval.

3.8.3 Vector Table

The Vector Table (VT) is located in memory, and the bottom is pointed to by the Vector Pointer. Each vector in the VT is 16 bits wide, and contains the absolute address of the handler.

The VT manages exceptions (including resets, services and errors) and interrupts.

The layout of the VT is shown below. The reserved entries are for future expansion.

Offset from VP	Type	Number	Vector Name	Resulting mode/state
0x00		0	Reserved	
0x02	Reset	1	HardReset	Supervisor
0x04	Reset	2	SoftReset	Supervisor
0x06	Error	3	InstructionError_S	Supervisor
0x08	Error	4	NullPointer_S	Supervisor
0x0A	Error	5	DivideByZero_S	Supervisor
0x0C	Error	6	UnknownInstruction_S	Supervisor
0x0E	Error	7	AlignError_S	Supervisor
0x10	Error	8	MMUDataError_S	Supervisor
0x12	Error	9	MMUProgError_S	Supervisor
0x14	Error	10	MMUUserDataError_S	Supervisor
0x16	Error	11	MMUUserProgError_S	Supervisor
0x18	Service	12	SysCall0_T	Supervisor
0x1A	Service	13	SysCall1_T	Supervisor
0x1C	Service	14	SysCall2_T	Supervisor
0x1E	Service	15	SysCall3_T	Supervisor
0x20	Service	16	SingleStep_T	Supervisor
0x22	Service	17	Break_T	Trusted
0x24	Error	18	PrivInstruction_S	Supervisor
0x26	Error	19	InstructionError_R	Recovery
0x28	Error	20	NullPointer_R	Recovery
0x2A	Error	21	DivideByZero_R	Recovery
0x2C	Error	22	UnknownInstruction_R	Recovery
0x2E	Error	23	AlignError_R	Recovery
0x30	Error	24	MMUDataError_R	Recovery

0x32	Error	25	MMUProgError_R	Recovery
0x34		26	Reserved	
0x36		27	Reserved	
0x38	Service	28	SysCall0_SI	No Change
0x3A	Service	29	SysCall1_SI	No Change
0x3C	Service	30	SysCall2_SI	No Change
0x3E	Service	31	SysCall3_SI	No Change
0x40 – 0x42	NMI	0-1	Int00 – Int01 (Non Maskable Interrupts)	NMI
0x44 – 0x5E	MI	2-15	Int02 – Int15 (Maskable Interrupts)	Interrupt

The response of the processor to errors depends on the processor mode or state:

Processor mode/state	**Response to error**
User	<ErrorName>_S
Trusted	<ErrorName>_S
Supervisor	<ErrorName>_R
Interrupt	<ErrorName>_R
Recovery	Soft Reset
NMI	Soft Reset

Note: PrivInstruction, MMUUserDataError and MMUUserProgError errors are only possible from User mode.

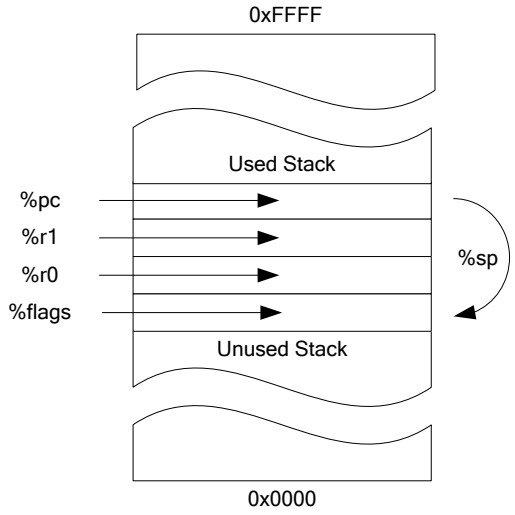
The response of the processor to services depends on the processor mode or state:

Processor mode/state	**Response to service**
User	<ServiceName>_T
Trusted	<ServiceName>_T
Supervisor	<ServiceName>_SI
Interrupt	<ServiceName>_SI
Recovery	Soft Reset
NMI	Soft Reset

Note: SingleStep, and Break services are only possible from User mode.

3.8.4 Context Push

When the XAP4 processes an interrupt or exception it does a context push. The Context Push is made to the Stack associated with the destination mode. The registers pushed onto the stack are popped back at the end of the handler (by the `rtie` instruction). Four words are pushed onto the stack:



A Soft Reset will take place if a Context Push is made before the required stack is initialised.

Error Exceptions can happen during a Context Push. The way they are handled depends on the stack being used. If the Context Push is being made to SP0 a Soft Reset will take place. If the Context Push is being made to SP1 the Error Exception will be handled in Supervisor mode in the normal way. Context Pushes to SP1 can only happen for Service Exceptions from User mode and Trusted mode.

3.8.5 Interrupt Processing

The XAP4 has a single interrupt input (`irq`), along with interrupt number (`irq_num`) and interrupt priority (`priority`) inputs. The XAP4 checks the `irq` input at the end of every instruction and during the execution of some long instructions (such as the `blk*` instructions, divides and stack accesses).

The Interrupt Vector Controller manages an interrupt priority system.

Interrupts will take priority over anything except an NMI, or a maskable interrupt of higher priority (lower priority number).

When the `irq` input is activated, the XAP4 uses the following logic to decide whether to service the interrupt:

```
if ((INFO[NL] == 0) && (INFO[K0] == 1) &&
    (Interrupt event))
{
    MI-Now = 0;
    NMI-Now = 0;
```

```
    lastFLAGS = FLAGS

    if (FLAGS[M] == User or Trusted or Supervisor)
    {
        if (InterruptNumber == 0-1)
        {
            NMI-Now = 1;
        }

        if ((InterruptNumber == 2-15) && (FLAGS[I] == 1) &&
            (INFO[R] == 0))
        {
            MI-Now = 1;
        }
    }

    if (FLAGS[M] == Interrupt)
    {
        if (InterruptNumber == 0-1)
        {
            NMI-Now = 1;
        }

        If ((InterruptNumber == 2-15) && (FLAGS[I] == 1) &&
            (INFO[R]==0) && (newPriority < currentPriority))
        {
            MI-Now = 1;
        }
    }

    if (MI-Now)
    {
        FLAGS[M]    = InterruptMode;
    }
    if (NMI-Now)
    {
       INFO[NL] = 1;
    }
    if (NMI-Now || MI-Now)
    {
        FLAGS[C]    = FLAGS[I];
        FLAGS[I]    = 0;
        FLAGS[P]    = irq_priority[3:0];  // Input signal

        Stack0 push = nextPC; // to be executed after rtie
        Stack0 push = R1;
        Stack0 push = R0;
        Stack0 push = lastFLAGS;

        EventVector = 16-bit value at (VP + 0x40 + (IrqNum*2))
        PC          = EventVector;
    }
}
else
{
    Hold Interrupt in pool and wait for
    MI-Now or NMI-Now conditions to be met;
}
```

Then the XAP4 uses the interrupt number to calculate an index into the Interrupt Vector Table. Entries in the Vector Table are absolute addresses to the start of the associated interrupt handler.

Most instructions are completed, except:

- `div.*`, `rem.*` and `divrem.*` are aborted.
- `push`, `push.i`, `pop` and `pop.ret` are stopped part way through, and their state is recorded in the S flags.

- `blk*` instructions are stopped at an intermediate memory cycle and the state is maintained in the registers or in the S flags.

If the instruction completes, the return address of the handler is the address of the next instruction, otherwise it is the address of the current instruction.

In detail, the interrupt processing is:

- The return address of the handler is pushed to the privileged stack.
- The contents of the R1, R0 and FLAGS registers are pushed (in that order) to the privileged stack.
- The processor switches to Interrupt mode if the interrupt is an MI.
- The processor switches to NMI state if interrupt is an NMI.
- The priority flags are updated with the priority of the interrupt being processed (0 for NMIs).
- The C flag is set to the value of the I flag
- The I flag is cleared, blocking maskable interrupts.
- The S flags are cleared
- The handler address is read from the VT (at address VP + 0x40 + `irq_num` * 2) and written into PC.

3.8.6 Exception Processing

Exceptions are handled differently, based on whether they are resets, services or errors.

It should be noted that services and errors are processed by different handlers depending on the mode they occur from:

- The handlers for User and Trusted mode errors are run in Supervisor mode, and have names suffixed by `_S`. Their vectors are in the range 0-19.
- The handlers for User and Trusted mode services are run in Trusted mode, and have names suffixed by `_T`.
- The handlers for Supervisor and Interrupt mode errors cause a switch to Recovery state, and have names suffixed by `_R`. Their vectors are in the range 20-27.
- The handlers for Supervisor and Interrupt mode services are run in the mode from which they occurred, and have names suffixed by `_SI`. Their vectors are in the range 28-31.

The XAP4 has the soft reset mechanism to avoid becoming stuck in a loop of faulty exception handlers. See section 3.7.2, "Soft Reset" for details.

When an exception occurs, the XAP4 uses the following logic to decide how to process it:

```
if ((INFO[NL] == 1) || (INFO[R] == 1) || (INFO[K0] == 0))
{
    if (Service Event)
    {
        Soft Reset;
    }
```

```
    if (Error Event)
    {
        Soft Reset;
    }
}

if ((INFO[K1] == 0) && (Service Event))
{
    Soft Reset;
}

if (  (INFO[NL] == 0) && (INFO[R] == 0) &&
      (
      ((Error event) && (INFO[K0] == 1)) ||
      ((Service event from User/Trusted) && (INFO[K1] == 1)) ||
            ((Service event from Supervisor/Int) && (INFO[K0] == 1))
      )
    )
{
    lastFLAGS = FLAGS;
    lastR0 = R0;

    if (Service Event)
    {
        R0  = Service Argument;

        if  (FLAGS[M] == UserMode or TrustedMode)
        {
            Stack = Stack1;
        }

        if  (FLAGS[M] == SupervisorMode or InterruptMode)
        {
            Stack = Stack0;
        }

        // FLAGS[P] unchanged
    }

    if (Error Event)
    {
        R0  = LastPC;
        Stack = Stack0;

        if (lastFLAGS[M] = UserMode or TrustedMode)
        {
            FLAGS[M] = SupervisorMode;
        }
        FLAGS[P] = ErrorPVal;
    }

    Stack push= Next Instruction or Current Instruction;
    Stack push= R1;
    Stack push= LastR0;
    Stack push= lastFLAGS;
    FLAGS[S] = 0;
    FLAGS[C] = FLAGS[I];
    FLAGS[I] = 0;
    R1 = 0;

    if ((Error Event) &&
        (lastFLAGS[M] = SupervisorMode or InterruptMode))
    {
        INFO[R] = 1;
    }

    If ((Service Event) &&
         (lastFLAGS[M] == User Mode))
    {
        FLAGS[M] = Trusted Mode;
    }
```

```
    EventVector = 16-bit value at (VP + (ExceptionNum * 2))
    PC          = EventVector;
}
```

Error Exceptions and Service Exceptions cause a Soft Reset in Recovery state and NMI state. See section 3.7, "Reset", for details of reset processing.

The processor pushes context to the stack and branches to the handler in all other cases. The stack is Stack0 when the destination is Supervisor mode, Interrupt mode, Recovery state or NMI state. The stack is Stack1 when the destination is Trusted mode. The whole of this sequence is atomic and cannot be interrupted.

- The return address of the handler is pushed to the stack.
- R1, R0, and the contents of the FLAGS register just before the exception is pushed to the Stack.
- R1 is set to 0 and R0 is set as shown in the pseudo-code above.
- The processor mode (FLAGS[M]) is set as shown in the pseudo-code above.
- Set FLAGS[C] = FLAGS[I]. Carry bit is used as temporary storage for I bit. Handler can restore I bit if first instruction is : mov.1.r %flags[i], %flags[c]
- Interrupts are disabled (FLAGS[I] = 0).
- For Errors, FLAGS[P] is updated. See section 3.8.2 "Exceptions.
- Note that some Exceptions do not change the value in FLAGS[P].
- FLAGS[S] is set to 0.
- The handler address is read from the VT (at address VP + ExceptionNum * 2) and written into PC.

3.8.7 Reset Details

These differ from other exception handlers because they don't need to return, and as such don't need to preserve processor state. For further details, see section 3.7, "Reset".

3.8.8 Service Details

SysCall_T, SysCall_SI

The `syscall.*` instructions cause the XAP4 to move into a privileged mode and may be used for User mode code to call an operating system function.

There are four `SysCall` handlers for User/Trusted mode calls, and four for Supervisor/Interrupt mode calls. The first argument to the instruction specifies the handler, and the second argument (either a register or an immediate) is used as the service argument.

`syscall.*` instructions cannot be used when in NMI state (INFO[NL] =1) or Recovery state (INFO[R]=1). Using them will cause a SoftReset.

`syscall.*` from User and Trusted modes transfers execution to Trusted mode. This means that the Stack is Stack1 before and after the syscall.

`syscall.*` from Supervisor and Interrupt modes does not change mode. This means that the Stack is Stack0 before and after the syscall.

SingleStep_T

When the T is set in the `FLAGS` register and the processor is in User Mode, this exception is triggered after every User mode instruction, but only if that instruction has not caused any other exception (e.g., `DivideByZero`). This generates an exception and transfers execution to Trusted mode. Stack1 is used before and after the Event. The argument to the service handler is the address of the instruction that was executed. This supports run-mode debugging and single stepping of User mode applications.

Multi-atom instructions (e.g. `blk*`,`push*`,`pop*`) appear as a sequence of instructions when single stepping.

Break_T

When the B is set in the `FLAGS` register and the processor is in User Mode, this exception is triggered by either a `brk` instruction or by a break condition being satisfied in the breakpoint registers. This generates an exception and transfers execution to Trusted mode. Stack1 is used before and after the Event. The service argument is the address of the instruction that caused the break event.

Break events are handled according to this logic.

```
if ((FLAGS[M] == User) && (FLAGS[P0] == 1))
{
    throw break exception
}
else
{
    if (RUN_STATE == RunToBreak)
    {
        halt;
    }
    else
    {
        nop;
    }
}
```

`RunToBreak` mode is enabled using the SIF.

When running a program in the xIDE simulator, `RunToBreak` mode is always enabled. Any break conditions halt the processor unless the processor is in User mode and the B flag is set.

3.8.9 Error Details

MMUUserDataError_S

This error is triggered by the external MMU activating the user_data_error signal in response to User mode code accessing memory that violates the current access rights (e.g., reading from memory that does not belong to the currently executing process, or writing to memory that is tagged as read-only). As the MMU is aware of the processor mode, this can only occur in User mode.

The MMU is expected to place the accessed address in a memory mapped register accessible to the error handler code.

This error can be used to implement virtual memory systems or to implement memory protection schemes between different processes.

`MMUUserDataError_S` can be caused by the following instructions:

- `blkcp.i, blkst.i, blkst.8.i`
- `blkcp.r, blkst.r, blkst.8.r`
- `bra.m, bsr.m`
- `ld.1.i, st.1.i, ldand.1.i, ldor.1.i, ldxor.1.i`
- `ld.8z.i, ld.i, ld.32.i`
- `ld.8z.r, ld.r, ld.32.r`
- `push, push.i`
- `pop, pop.ret`
- `st.8.i, st.i, st.32.i`
- `st.8.r, st.r, st.32.r`
- `swap.i`

MMUUserProgError_S

This error is triggered by the external MMU activating the user_prog_error signal in response to User mode code attempting to fetch an instruction from a location that violates the current access rights. As the MMU is aware of the processor mode, this can only occur in User mode.

This error can be used to implement virtual memory systems or to implement memory protection schemes between different processes.

PrivInstruction_S

In an application using an operating system or supervisor, the XAP4 executes most code in User mode. This mode is restricted in its access rights to certain instructions; executing any privileged instruction in User mode triggers a `PrivInstruction` error.

The following instructions cause this error because they could be used to create mode changes or to interfere with the behaviour of other modes:

- `mov.1.i` (unless operating on the Z, N, C or V flags), `mov.2.i`, `mov.4.i`
- `mov.1.r %flags[i], %flags[c]`
- `mov.1.r %flags[i], Rs`

- `mova2r, movr2a, movb2r, movr2b, movs2r` (unless operating on the flags register), `movr2s`
- `rtie`

The following instructions cause this error because they alter the behaviour of the XAP4 core.

- `halt`
- `sleepsif, sleepnop`
- `sif`
- `softreset`

The `sif` instruction is not allowed in User mode, because it can alter the timing of accesses from external devices to the registers and memory of the processor. If SIF cycles are required in User mode code, the `sif` instruction should be wrapped as a `SysCall` service handler and made available via a `syscall` instruction.

InstructionError_S, InstructionError_R

This error is triggered

- When the Register arguments (Rad, Ras, Ra, Rn, Rs) re-use the same register more than once (which would produce useless instruction behaviour).
- When a non-existent special register, address register or breakpoint register is accessed.
- When a `clu` instruction causes an error. ErrorPval is set as shown in section 3.8.2 ”Exceptions”.

This error can be caused by the following instructions:

- `blkcp.i, blkst.i, blkst.8.i`
- `blkcp.r, blkst.r, blkst.8.r`
- `mova2r, movr2a, movb2r, movr2b, movs2r, movr2s`
- `clu.*`

NullPointer_S, NullPointer_R

The XAP4 detects attempts by code to perform data accesses to memory address zero. Such accesses are normally the result of a program using an uninitialised pointer variable.

The MMU is expected to place the address that caused the error in a memory mapped register accessible to the error handler code.

`NullPointer` can be caused by the following instructions:

- `blkcp.i, blkst.i, blkst.8.i`
- `blkcp.r, blkst.r, blkst.8.r`
- `bra.m, bsr.m`
- `ld.1.i, st.1.i, ldand.1.i, ldor.1.i, ldxor.1.i`
- `ld.8z.i, ld.i, ld.32.i`
- `ld.8z.r, ld.r, ld.32.r`
- `push, push.i`
- `pop, pop.ret`

- `rtie`
- `st.8.i, st.i, st.32.i`
- `st.8.r, st.r, st.32.r`
- `swap.i`

DivideByZero_S, DivideByZero_R

All forms of the divide and remainder instructions check the value of the denominator. If it is zero, then the instruction throws a `DivideByZero` error.

The following instructions can cause this error:

- `div.*`
- `divrem.*`
- `rem.*`

UnknownInstruction_S, UnknownInstruction_R

All opcodes that do not correspond to a valid instruction trigger this error.

AlignError_S, AlignError_R

This error is caused by attempts to write to the low bits of PC, SP and VP that should be zero.

This error is also caused when either an odd function table address or an odd address is passed to `bra.m` or `bsr.m`. The return address of the handler is the target address of the instruction with the lowest bit cleared. If the exception is caused by `bra.*` `bsr.*` then the address of the instruction following the offending instruction is pushed to the stack as usual.

The following instructions can cause this exception:

- `bra.i`
- `bra.m`
- `bsr.i`
- `bsr.m`
- `pop.ret`
- `rtie`
- `movr2a`
- `mov.r`

MMUDataError_S, MMUDataError_R

This error is caused by the MMU activating the `data_error` signal, to indicate that an instruction's memory access has failed in some way. This may be because it violates access rights (e.g., reading from memory that does not belong to the currently executing process, or writing to memory that is tagged as read-only or to an address which does not exist)..

The MMU is expected to place the address that caused the error in a memory mapped register accessible to the error handler code.

This error can be used to implement memory protection schemes between different processes.

`MMUDataError` can be caused by the following instructions:

- `blkcp.i, blkst.i, blkst.8.i`
- `blkcp.r, blkst.r, blkst.8.r`
- `bra.m, bsr.m`
- `ld.1.i, st.1.i, ldand.1.i, ldor.1.i, ldxor.1.i`
- `ld.8z.i, ld.i, ld.32.i`
- `ld.8z.r, ld.r, ld.32.r`
- `pop, pop.ret`
- `push, push.i`
- `st.8.i, st.i, st.32.i`
- `st.8.r, st.r, st.32.r`
- `swap.i`

MMUProgError_S, MMUProgError_R

This error is caused by the MMU activating the `prog_error` signal, to indicate that an instruction fetch has failed in some way. This may be because it violates access rights.

This error can be used to implement memory protection schemes between different processes.

3.8.10 Returning from interrupts and exceptions

The `rtie` instruction is used to return from all Exceptions and Interrupts. It can only be executed in Privileged modes. It pops the context from the current Stack (Stack1 in Trusted mode, Stack0 in Supervisor mode, Interrupt mode, Recovery state and NMI state).

The `rtie` instruction does the following:

- Loads FLAGS from the current stack.
- Loads R0 from the current stack.
- Loads R1 from the current stack.
- Loads PC from the current stack.
- Clears the INFO[NL] bit when executed in NMI state
- Clears the INFO[R] bit when executed in Recovery state

Thus the null interrupt handler is simply the `rtie` instruction.

Refer to the diagram in section 3.8.4, "Context Push" for details of the stack usage during the context push that happens when the XAP4 processes an interrupt or exception.

3.9 Debugging

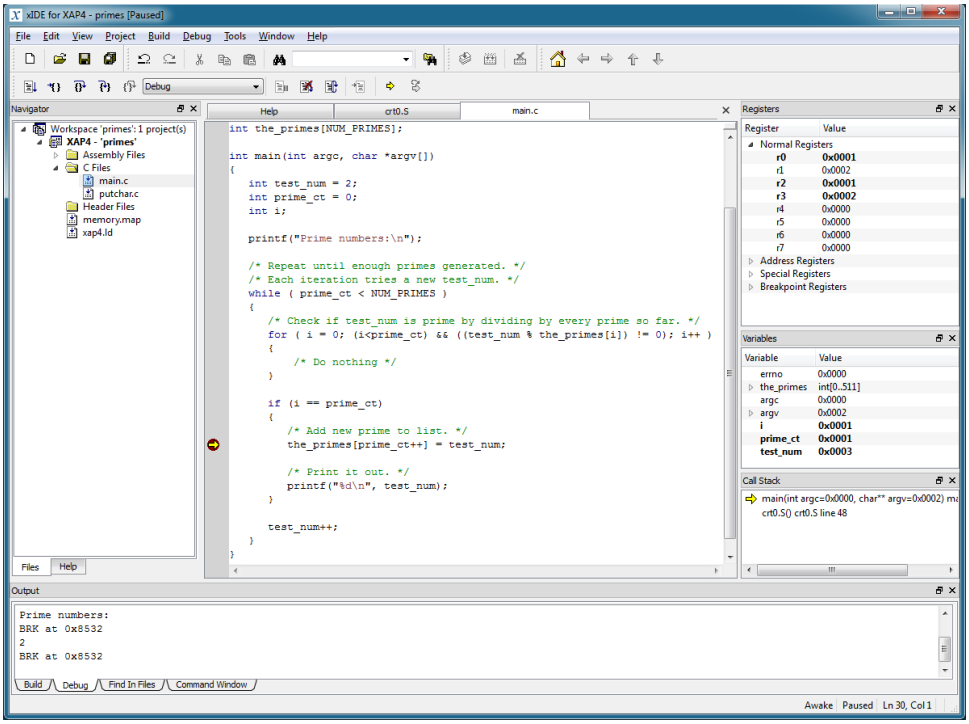
The xSIF Interface

The xSIF is a patented four-wire serial interface for external communication with ASICs. Using the xSIF, it is possible to develop and debug software on emulators such as xEMU mini, or on ASIC devices. Control and data acquisition using the xSIF

is also useful during silicon characterisation and qualification and in product testing and calibration during manufacture.

The xIDE Integrated Development Environment

Systems containing a XAP4 are developed and debugged with the xIDE integrated development environment using the SIF interface.



xIDE provides:

- An integrated development and unified debug tool interface.
- Multiprocessor support.
- Familiar interface with HTML-based on-line help.
- Extensible architecture through software plug-ins.
- Project navigator to project files and program builds.
- Built-in multiple-document text editor with syntax highlighting.
- Customisable docking windows, toolbars and GUI widgets for system-specific requirements.
- Integrated Python command line. Python macros automate frequently used tasks.
- Supports Python macros to automate and simulate target system functionality.
- Multiple document windows for source code, browsing memory, debug output, register views, peripherals and variables watch.
- Support for simultaneous debug of multi-processor designs.
- Support for stop-mode debugging.

More information on xIDE can be found in the xIDE User Manual, C7066-TM-001.

Stop-Mode Debugging

In stop-mode debugging, all code running on the XAP4 is stopped when a breakpoint is encountered or whilst single stepping. This is the normal debugging situation.

Run-Mode Debugging

The XAP4 instruction set supports run-mode debugging of User mode code. An on-chip debugger running in a privileged mode can use the `SingleStep` and `Break` exceptions to debug code running in User mode.

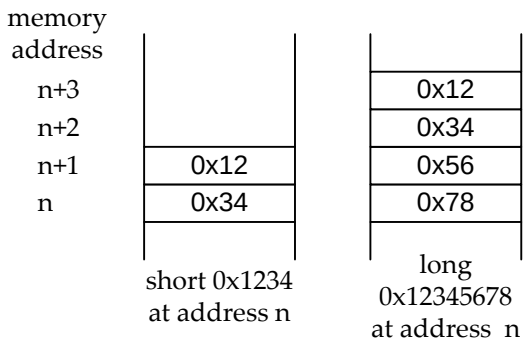
4 C Language Interface

4.1 Data Types

The XAP4 has instructions to operate on 1-bit, 8-bit, 16-bit or 32-bit data. There is no hardware support for floating point data types, but the standard C libraries provide a full IEEE754 implementation.

Endianness

Data and code is stored in memory in a little-endian byte order:



4.2 Data Alignment

The XAP4 supports accesses to 8, 16 and 32-bit data at any memory address. Accesses to unaligned data are performed natively and with no programmer effort.

Alignment of data objects within memory is described in detail in the XAP4 GCC Manual, C7432-UM-004.

4.2.1 Aligned data

Aligned data produces the fastest code and is the normal model for the compiler. The compiler aligns data as follows:

- Single-byte variables are allocated to byte boundaries.
- All other data is allocated to a word boundary.

Fields within structures are similarly aligned. The compiler adds padding bytes between variables and within structures as necessary to achieve this alignment.

4.2.2 Unaligned data

The compiler `__packed__` option allows fields within structures to be packed. This prevents the compiler from adding padding bytes between fields and can result in fields lying at non-word-aligned addresses.

The XAP4 supports accesses to unaligned data natively.

Unaligned data can be accessed with the `ld.*`, `st.*`, `swap.i` and `blk*` instructions.

Two memory accesses are required for each unaligned word operation. There is therefore a runtime penalty associated with unaligned data.

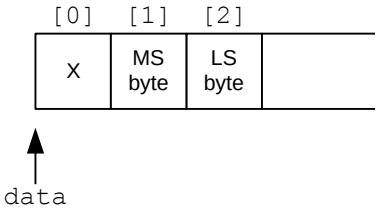
The MMU may detect accesses to unaligned data. This may provide useful debugging information, or, if speed of execution is an issue, it allows the programmer to restructure the data to remove the unaligned access. Refer to the XAP4 Hardware Reference Manual for more details.

4.2.3 Methods of accessing unaligned data

There are two methods of accessing unaligned data. The methods vary in their speed, readability and portability.

	Relative duration to access an unaligned half-word	**Readability**	**Portability**
Read the data as a byte stream	Slow	Average	Good
Use C to produce unaligned accesses	Fast	Average	Poor

The following sections illustrate both methods extracting a single little-endian 16-bit integer from an unaligned big-endian byte stream. The incoming data is in a byte buffer called `data`.



Read the data as a byte stream

The data is read as a sequence of bytes which are reassembled into the correct order. This code is entirely portable across compilers and architectures but requires two memory reads per word extracted, and also a shift and a logical OR.

```
extern uint8 *data;
short y = (*(data+1) << 8) | (*(data+2)); // Two byte reads
```

Use C to produce unaligned accesses

The first assignment generates an unaligned read from the data buffer. This code will not work on architectures that do not support unaligned memory accesses but is possible on XAP4.

```
extern uint8 *data;
short y = *(int *)(data+1);                 // Unaligned read
y = ((y&0xFF00)>>8) | ((y&0x00FF)<<8)    // Convert endianness
```

4.3 Calling Convention

The calling conventions clearly define how functions are to be called, and how the return value, if any, is passed back to the caller.

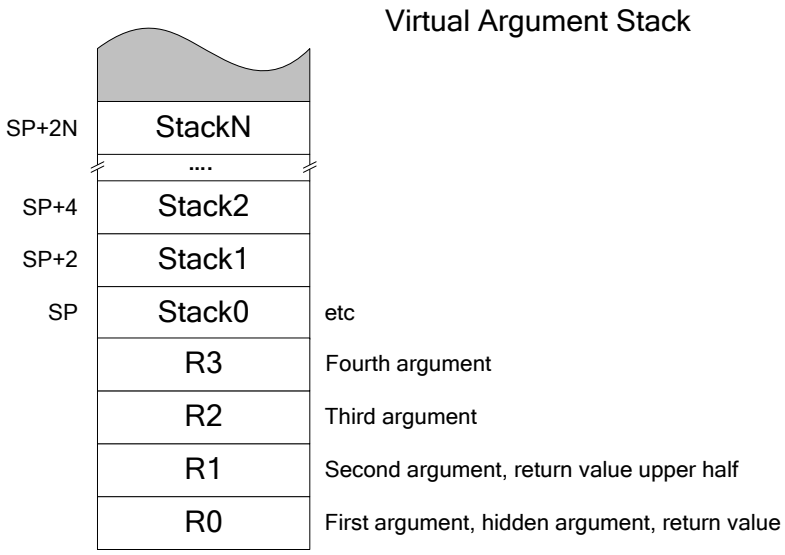
The most common direction of call is from high-level C code down to a low-level assembler routine. This section describes how a C function passes arguments into a function, and how that function must return them to the C function.

See XAP4 GCC Manual, C7432-UM-004.

Virtual Argument Stack

The XAP4 calling convention uses a combination of the four registers R0, R1, R2 and R3, and the stack to pass arguments to functions. Together, they form a virtual argument stack on which all the function arguments are pushed.

As can be seen from the diagram below, the bottom four slots of the virtual argument stack are actually held in registers. Pushing an argument into these virtual stack slots is in practice a `mov` operation rather than a `push`. This optimisation simplifies calling functions with a small number of arguments.



The rest of the virtual argument stack is held in the XAP4's physical stack in memory. While it is slower to access than registers, space on the physical stack can be dynamically allocated and freed at runtime simply by pushing and popping values on and off the stack.

Each function argument is placed onto the stack in ascending order, with the first argument bottommost and the last argument topmost.

Return Values

Scalar return values are passed in R0, and the upper half of double-word values additionally in R1. Larger multi-word return values, and aggregate return values (i.e. structures and unions), are returned via a hidden pointer passed in R0 by the caller.

5 Instruction Set Overview

5.1 Summary of Assembler Syntax

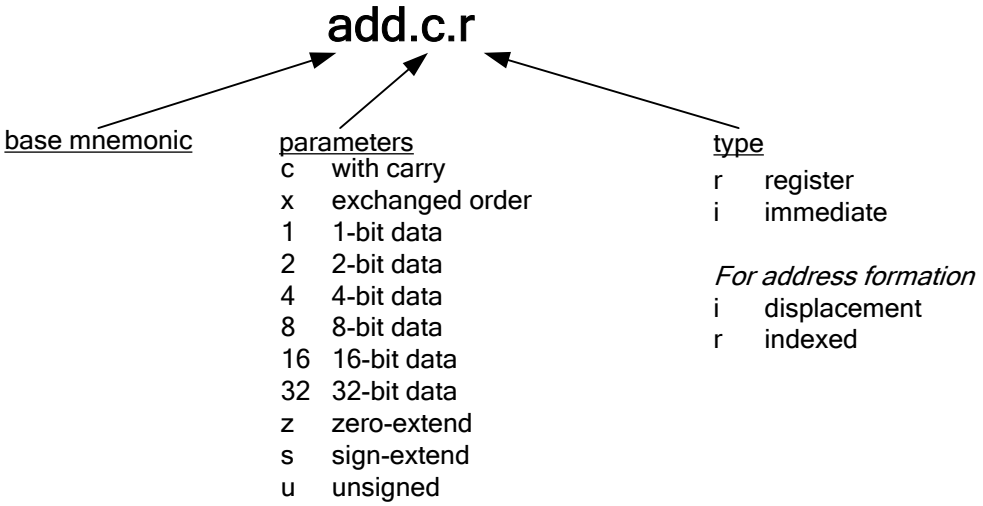
The XAP4 assembly language is case-sensitive. Instruction mnemonics, registers, number prefixes and directives must be in lower case.

Instructions are terminated by an end-of-line. Instructions cannot span more than one line. Spaces and tabs are treated as white space that delimits assembler tokens.

Full details of the assembler syntax are in the XAP4 Binutils Manual, C7432-UM-003.

5.1.1 Instruction mnemonics

XAP4 instruction mnemonics have a regular structure, consisting of a base mnemonic, optional parameters and an optional type.



If no width is explicitly stated, the instruction operates on 16-bit data.

5.1.2 Operands

Operands are separated by commas.

5.1.3 Registers

Register names are lower case and are prefixed by `%`. XAP4 register names do not pollute the C namespace. In all instances the operands denoted by Rd, Rs, Rt, Ra, Ras, Rad and Rx may take the values of the normal registers R0 – R7 (`%r0` – `%r7`). In some instructions, SP (`%sp`), PC (`%pc`) and Zero (0) may also be valid operands. The `movr2s` and `movs2r` instructions accept registers such as FLAGS (`%flags`) and BRKE (`%brke`) as operands.

5.1.4 Register Lists

Some instructions take a list of registers as an operand. A RegList can take one of three forms:

- A list of one or more separate registers – `{%r3, %r6}`
- A range of registers – `{%r3-%r5}`
- Any mixture of the two forms – `{%r3-%r4, %r6}`

In all three forms, the registers must be:

- low to high (`%r0-%r7`) for `pop` and `pop.ret`
- high to low (`%r7-%r0`) for `push`.

A register may not appear twice in the RegList.

5.1.5 Comments

C and C++ style comments are accepted:

```
/*
This is a block comment
*/
and.i %r1, %r2, #0xF00F  // This is a line comment
```

5.1.6 Number formats

Immediate values can be entered in decimal, hexadecimal, octal or binary. The assembler converts all immediates to 16 bits and then rejects any immediate value which cannot be represented in an instruction. Immediates (except those used in address formation) are prefixed with '`#`'.

Decimal

Decimal numbers require no prefix and have an optional sign. To avoid confusion with octal, the first digit must not be zero.

Hexadecimal

Hexadecimal numbers have an 0x prefix. To enter a negative number in hexadecimal, two approaches can be used:

- Explicitly define all bits – for example, for a 16-bit integer, 0xFFFE.

- Use the unary minus operator – for example, -0x2.

The characters a-f and A-F can be used in hexadecimal numbers. The XAP software tools use the upper case versions, A-F.

Octal

Numbers with a leading zero are treated as octal. Only digits 0 to 7 are valid in octal numbers.

Binary

Binary numbers have an 0b prefix. To enter a negative number in binary, two approaches can be used:

- Explicitly define all bits – for example, for a 16-bit integer, 0b1111111111111110.
- Use the unary minus operator – for example, -0b10.

Integer Sizes

On XAP4 all immediates are treated as 16-bit numbers.

Negative numbers can be specifed in hexadecimal or binary without using the unary minus operator by specifying all 16 bits in the integer.

The sign bit is taken to be the most significant bit in the integer.

Examples of valid numbers

Assembler syntax	Decimal equivalent
0x1234	4660
0xFEDC	-292
-0x1234	-4660
-0xFEDC	292
012	10
0b10	2
0b1111111111111110	-2
-0b10	-2
-0b1111111111111110	2

5.1.7 Labels

Labels are case sensitive. Labels start in column 1 and end with a colon. The `.equ` and `.set` directives can also be used (see the section on directives below). For example:

```
.equ n, 0x12345678

myvariable:
.long 0xABCD

// The start of the program
start:

// This is a different label
Start:

ld.i %r0, @(myvariable, 0)    // R0 = 0xABCD
st.i #0, @(myvariable, 0)     // myvariable = 0


mov.i %r0, (myvariable, %pc) // PC-relative
mov.i %r0, (myvariable, 0)   // Zero-relative
```

5.1.8 Expressions

The XAP4 assembler can support arithmetic expressions where you would otherwise give an immediate or an address. For details, see the original GNU Binutils documentation. The chapter on expressions is available online at the following URL:

http://sourceware.org/binutils/docs-2.17/as/Expressions.html

Some examples of valid expressions follow:

```
add.i     %r0, %r0, #(2 + 3)
ld.i      %r0, @(label + 2, %pc)
mov.i     %r0, #(19 * 2)
mov.i     %r0, (label + 4, %pc)
mov.i     %r0, #(block_end - block_start)
```

5.1.9 Directives

Directives start with a period (“.”) as the first non-blank character of the line. For example:

```
.org 0x1000
.file “Initialisation Code”
.text
```

Full details of the directives are in the XAP4 Binutils Manual, C7432-UM-003.

5.2 Instruction Encoding

XAP4 instructions are either 32 bits or 16 bits long. Some instructions have only 32-bit encodings, some have only 16-bit encodings and some have both.

For instructions available in both 32-bit and 16-bit forms, the assembler syntax is identical for both. Unless instructed otherwise, the linker selects the smallest encoding available.

An exception is `bra.i`. This instruction is commonly used to implement C switch statements. In this situation it is necessary to force the assembler to generate 16-bit or 32-bit `bra.i` instructions. There are therefore specific variants of the `bra.i` instruction:

- `bra.i.2` forces the assembler to generate a 16-bit encoding.
- `bra.i.4` forces the assembler to generate a 32-bit encoding.

Instructions of different sizes can be freely mixed in the instruction stream.

5.3 Address Formation

The XAP4 assembly language uses one form to express address formation. It is consistent between instructions that require it:

```
[prefix](offset, base address) // Displacement addressing
[prefix](index, base address) // Indexed addressing
```

The offset is always a literal and the index is always stored in a normal register. The base address depends on the instruction and addressing mode used. It can be `%r0-%r7`, `%sp`, `%pc` or `0`.

The optional one-character prefix can be either `@` or `!`, and has the following meaning:

- @: indicates that a value will be loaded from or stored to the address formed. This has the additional implication that the instruction can cause a variety of exceptions (see section `3.8.6`, Exception Processing).
- !: This indicates that the linker should create a function table entry for the given symbol, and the address of the function table entry should be used in the instruction. This is allowed on the PC-relative form of mov.i as well as bra.m and bsr.m instructions and is intended to be used for taking the address of functions.

Address formations are used with the following instructions:

- Conditional and unconditional branches
- `ld*, st*`
- `mov.i`
- `swap.i`

5.3.1 Addressing Modes

The XAP4 assembly language has two addressing modes.

- Displacement addressing uses a literal offset relative to a base address stored in normal registers, the stack pointer, the program counter or 0.

- Indexed addressing uses an index relative to a base address. The index register is always a normal register. The base address can be a normal register or the stack pointer.

Displacement addressing

Displacement addressing is indicated by a `.i` suffix to the instruction. It is also used for the `bra.m` and `bsr.m` instructions. The instruction specifies a base address and a literal offset. The memory address is calculated as the sum of the two. The base address can be stored in a normal register, SP or PC, or can be given as zero (this is zero-relative addressing). These examples all load R1 from memory address 0x1020:

```
// Declare a variable in uninitialised memory (bss)
.bss
.org 0x1020
myvariable:
.short

// Start the code section
.text

mov.i    %r2, (0x1000, 0)  // R2 = 0x1000
ld.i     %r1, @(0x20, %r2) // address = 0x20 + R2

ld.i     %r1, @(0x1020, 0) // address = 0x1020
ld.i     %r1, @(myvariable, 0) // labels can be used too

mov.r    %sp, %r2          // SP = 0x1000
ld.i     %r1, @(0x20, %sp) // address = 0x20 + SP

.org 0x1000                // PC = 0x1000
ld.i     %r1, @(0x20, %pc)
ld.i     %r1, @(myvariable, %pc) // labels can be used too
```

Example Instructions	Calculated memory address
`ld.i Rd, @(offset, Ra)`	offset[15:0]s + Ra
`ld.i Rd, @(label, 0)`	label[15:0]
`ld.i Rd, @(offset, 0)`	offset[15:0]u + 0
`ld.i Rd, @(offset, %sp)`	offset[15:0]u + SP
`ld.i Rd, @(label, %pc)`	label[15:0]
`ld.i Rd, @(offset, %pc)`	offset[15:0]s + PC[15:0]
`mov.i Rd, (label, %pc)`	label[15:0]
`mov.i Rd, (offset, %pc)`	offset[15:0]s + PC[15:0]
`mov.i Rd, (label, 0)`	label[15:0]
`mov.i Rd, (offset, 0)`	offset[15:0]u + 0
`bra.i (label, %pc)`	label[15:0] (must be even)

`bra.i (offset, %pc)`	offset[15:0]s + PC[15:0] (must be even)
`bra.i (label, 0)`	label[15:0] (must be even)
`bra.i (offset, 0)`	offset[15:0]u + 0 (must be even)
`beq (label, %pc)`	label (must be even)
`beq (offset, %pc)`	offset[15:0]s + PC[15:0] (must be even)
`bra.m @(label, 0)`	*(label[15:0]) (must be even)
`bra.m @(offset, 0)`	*(offset[15:0]u + 0) (must be even)
`bra.m @(0, Ra)`	*(0 + Ra) (must be even)

PC –relative instructions use the current value of PC as Ra.

Indexed addressing

Indexed addressing is indicated by a `.r` suffix to the instruction. The instruction specifies two registers, containing a base address and an index. The memory address is calculated as follows:

Example Instructions	Calculated memory address
`ld.8z.r Rd, @(Rx, Ra)`	1*Rx + Ra
`ld.r Rd, @(Rx, Ra)`	2*Rx + Ra
`ld.32.r Rd, @(Rx, Ra)`	4*Rx + Ra

Ra can be `%r0-%r7` or `%sp`.

6 Instruction Groups

6.1 Instruction Set Overview

The following sections provide an overview of each group of instructions.

6.1.1 Branches

Unconditional branches

The XAP4 unconditional branch address can be specified in several ways:

- `bra.i` uses either displacement addressing relative to PC, displacement addressing relative to 0, or an address contained in a register.
- `bra.m` uses an address contained in memory, the address of which is specified with displacement addressing, relative to a normal register or 0.

The same mnemonics are available when branching to a subroutine (`bsr.i and bsr.m`). These push the return address to the stack.

Conditional branches

All conditional branches are relative to the program counter. A full set of conditions based on the FLAGS register is supported.

There are also branches that are conditional on the contents of a general purpose register.

6.1.2 Load and Store

Single memory transfers

Loads and stores are available in 8, 16 and 32-bit versions and with displacement, indexed and PC-relative addressing. 8-bit data is zero-extended to 16 bits on loads.

The store instructions can also be used to write the constants -1, 0 or 1 into memory.

Swap

The `swap.i` instruction performs an atomic swap between a register and a memory location. This is a useful instruction for implementing sempahores in operating systems. Refer to section 6.2.3, "Semaphore" for an example.

6.1.3 Stack Operations

The `push`, `push.i`, `pop` and `pop.ret` instructions enable very compact code for stack operations and function entry and exit. Multiple registers can be pushed or popped in a single instruction.

These instructions can be interrupted. State is maintained in the S flags and the operation resumes when the `rtie` instruction is executed.

The stack pointer is SP. The stack grows downwards in memory. Lower numbered registers are stored at lower stack addresses. The stack pointer points to the last used location on the stack.

push

The `push` instruction pushes a selection of registers on to the stack. The flags are not updated. An additional operand adjusts the stack pointer downwards to create a stack frame for the callee function's automatic variables.

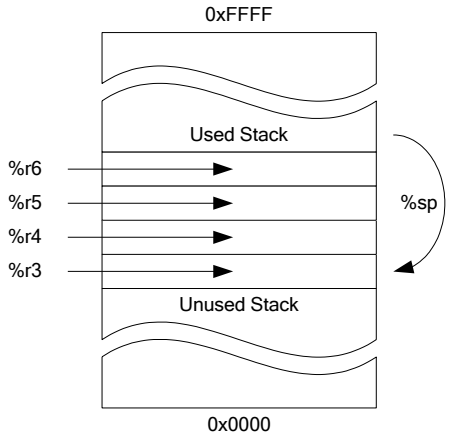
push – example 1

```
push {%r6-%r3}, #0
```

This is equivalent to the following sequence of instructions.

```
add.i     %sp, %sp, #-2     // Decrease SP for 1 register
st.i      %r6, @(0, %sp)    // Store R6 to stack memory
add.i     %sp, %sp, #-2     // Decrease SP for 1 register
st.i      %r5, @(0, %sp)    // Store R5 to stack memory
add.i     %sp, %sp, #-2     // Decrease SP for 1 register
st.i      %r4, @(0, %sp)    // Store R4 to stack memory
add.i     %sp, %sp, #-2     // Decrease SP for 1 register
st.i      %r3, @(0, %sp)    // Store R3 to stack memory
```

This is illustrated in the diagram below.



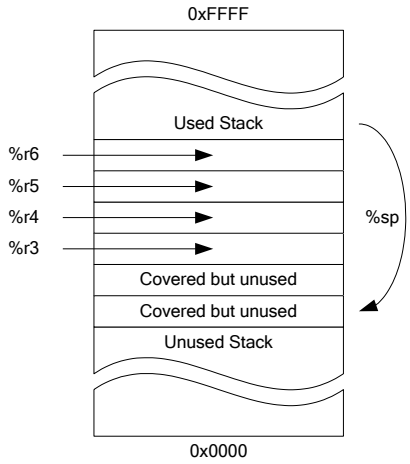
push – example 2

```
push {%r6-%r3}, #4
```

This is equivalent to the following sequence of instructions.

```
add.i     %sp, %sp, #-2     // Decrease SP for 1 register
st.i      %r6, @(0, %sp)    // Store R6 to stack memory
add.i     %sp, %sp, #-2     // Decrease SP for 1 register
st.i      %r5, @(0, %sp)    // Store R5 to stack memory
add.i     %sp, %sp, #-2     // Decrease SP for 1 register
st.i      %r4, @(0, %sp)    // Store R4 to stack memory
add.i     %sp, %sp, #-2     // Decrease SP for 1 register
st.i      %r3, @(0, %sp)    // Store R3 to stack memory
add.i     %sp, %sp, #-4     // Decrease SP by #offset[4:1]u
                            // 4 bytes of stack free for
                            //        other purposes
```

This is illustrated in the diagram below.



push.i

The `push.i` instruction pushes list of immediates. The flags are not updated. Note that it can use up to four immediates, although the range of the immediates depends on the number used:

Number of immediates	Range of each immediate
1	-32768 to 32767
2	-128 to 127
3	-8 to 7
4	-8 to 7

push.i - example 1

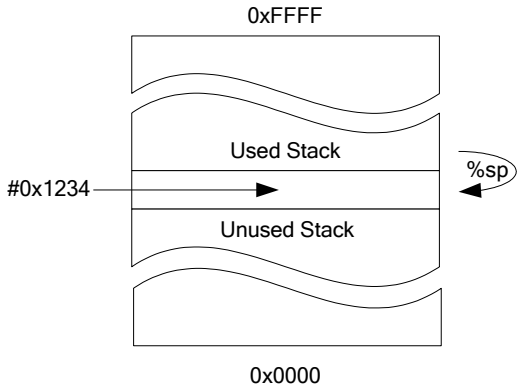
```
push.i {#0x1234}, #0
```

This is equivalent to the following sequence of instructions:

```
add.i     %sp, %sp, #-2       // Decrease SP for 1 word
st.i      #0x1234, @(0, %sp) // Store 0x1234 to stack memory
```

Note that `st.i #0x1234, @(0, %sp)` is not a valid instruction.

This is illustrated in the diagram below.



push.i - example 2

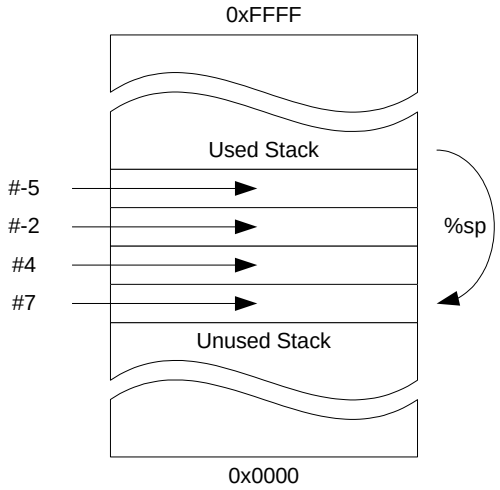
```
push.i {#-5, #-2, #4, #7}, #0
```

This is equivalent to the following sequence of instructions:

```
add.i     %sp, %sp, #-2       // Decrease SP for 1 word
st.i      #-5, @(0, %sp)      // Store -5 to stack memory
add.i     %sp, %sp, #-2       // Decrease SP for 1 word
st.i      #-2, @(0, %sp)      // Store -2 to stack memory
add.i     %sp, %sp, #-2       // Decrease SP for 1 word
st.i      #4, @(0, %sp)       // Store 4 to stack memory
add.i     %sp, %sp, #-2       // Decrease SP for 1 word
st.i      #7, @(0, %sp)       // Store 7 to stack memory
```

Note that `st.i #(value), @(0, %sp)` are not valid instructions.

This is illustrated in the diagram below.



pop

The `pop` instruction pops a selection of registers off the stack. The flags are not updated. It is not normally necessary to pop R0, R1, R2 or R3 from the stack as they are not preserved over a function call.

`pop` takes an additional operand to adjust the stack pointer upwards to remove the stack space allocated for the callee function's automatic variables.

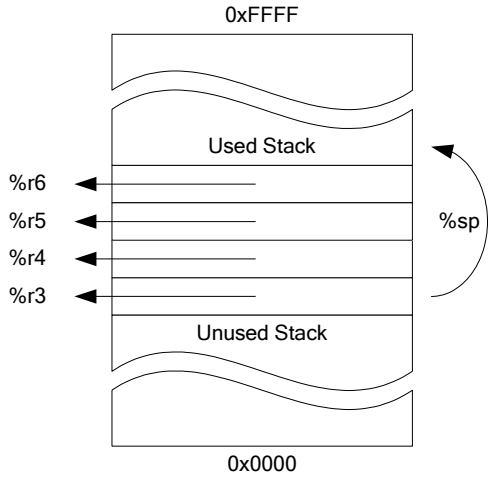
pop – example 1

```
pop {%r3-%r6}, #0
```

This is equivalent to the following sequence of instructions:

```
ld.i      %r3, @(0, %sp)    // Load R3 from stack memory
add.i     %sp, %sp, #2      // Increase SP for 1 register
ld.i      %r4, @(0, %sp)    // Load R4 from stack memory
add.i     %sp, %sp, #2      // Increase SP for 1 register
ld.i      %r5, @(0, %sp)    // Load R5 from stack memory
add.i     %sp, %sp, #2      // Increase SP for 1 register
ld.i      %r6, @(0, %sp)    // Load R6 from memory
add.i     %sp, %sp, #2      // Increase SP for 1 register
```

This is illustrated in the diagram below.



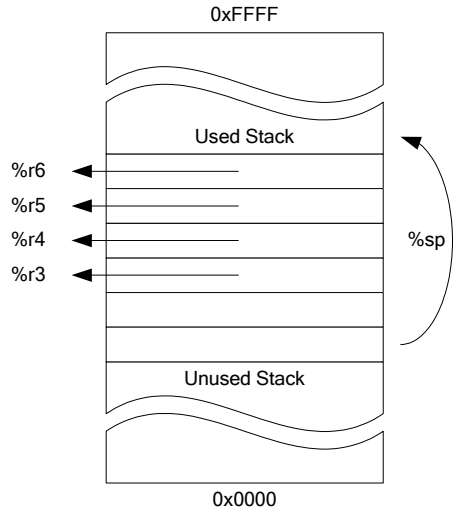
pop – example 2

```
pop {%r3-%r6}, #4
```

This is equivalent to the following sequence of instructions:

```
add.i     %sp, %sp, #4     // Increase SP by #offset[4:1]u
ld.i      %r3, @(0, %sp)   // Load R3 from stack memory
add.i     %sp, %sp, #2     // Increase SP for 1 register
ld.i      %r4, @(0, %sp)   // Load R4 from stack memory
add.i     %sp, %sp, #2     // Increase SP for 1 register
ld.i      %r5, @(0, %sp)   // Load R5 from stack memory
add.i     %sp, %sp, #2     // Increase SP for 1 register
ld.i      %r6, @(0, %sp)   // Load R6 from memory
add.i     %sp, %sp, #2     // Increase SP for 1 register
```

This is illustrated in the diagram below.



pop.ret

`pop.ret` performs the same operations as pop but additionally returns from the function by popping the return address from the stack into the program counter, and sets the flags for the value of R0, as specified by the calling convention.

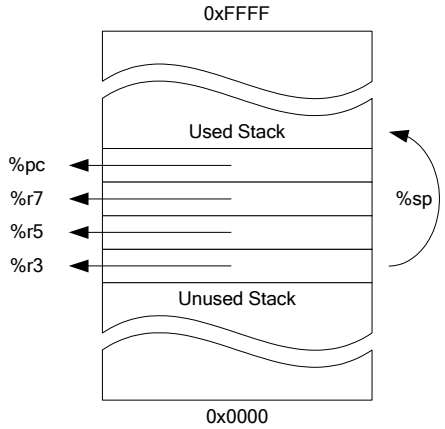
pop.ret – example 1

`pop.ret {%r3, %r5, %r7}, #0`

This is equivalent to the following sequence of instructions:

```
ld.i      %r3, @(0, %sp)     // Load R3 from stack memory
add.i     %sp, %sp, #2       // Increase SP for 1 register
ld.i      %r5, @(0, %sp)     // Load R5 from stack memory
add.i     %sp, %sp, #2       // Increase SP for 1 register
ld.i      %r7, @(0, %sp)     // Load R7 from stack memory
add.i     %sp, %sp, #2       // Increase SP for 1 register
cmp.i     %r0, #0            // Set the flags
// Retrieve return address from stack to PC and
// increase the stack pointer by 2
```

Note: in practise, the final action is equivalent to `pop.ret {}, #0`.



pop.ret example 2

`pop.ret {%r3, %r5, %r7}, #4`

This is equivalent to the following sequence of instructions:

```
add.i     %sp, %sp, #4       // Increase SP by 4 bytes
ld.i      %r3, @(0, %sp)     // Load R3 from stack memory
add.i     %sp, %sp, #2       // Increase SP for 1 register
ld.i      %r5, @(0, %sp)     // Load R5 from stack memory
add.i     %sp, %sp, #2       // Increase SP for 1 register
ld.i      %r7, @(0, %sp)     // Load R7 from stack memory
add.i     %sp, %sp, #2       // Increase SP for 1 register
cmp.i     %r0, #0            // Set the flags
// Retrieve return address from stack to PC and
// increase the stack pointer by 2
```

Note: in practise, the final action is equivalent to `pop.ret {}, #0`.

6.1.4 Move

The `mov.*` instructions provide various methods for loading one or more registers with various values.

mov.i – address or immediate

The instruction has three forms. With displacement addressing, it can specify a signed offset relative to the program counter, in the range -32kB to +32kB, or an unsigned offset relative to zero, in the range 0 to 64kB. It can also specify a 16-bit immediate. The resulting 16-bit value is loaded into a register.

mov.r – register to register

`mov.r` allows a single register to register copy. SP can also be used.

mov.32.r, mov.32s.r, move.32z.r – register pairs

`mov.32.r` allows a register pair to register pair copy.

`mov.32s.r` allows a sign-extended register to register pair copy.

`mov.32z.r` allows a zero-extended register to register pair copy.

6.1.5 ALU operations

The ALU supports a diverse set of logical (and, or, xor) and arithmetic (add, subtract, multiply, divide) operations. The general operands are a destination register, a source register, and a third operand which may be either a third register or an immediate value. For example:

- `and.i Rd, Rs, #imm`
- `and.r Rd, Rs, Rt`

Subtract

Both normal (a-b) and exchanged order (b-a) subtractions are supported, with and without carry (borrow).

Multiply instructions

XAP4 has instructions for 16x16 integer multiplies, giving either a 16-bit result or the full 32-bit result.

The `mult.sh.r` instruction produces an intermediate 32-bit result which is then shifted right a specified number of places to give a 16-bit result. This is useful for normalised fixed point multiplication.

Divide and remainder instructions

Division is performed FORTRAN-style:

- Both operands are made unsigned
- The division is performed
- The sign of the result is corrected if necessary

The div and rem instructions work together such that

`((a/b)*b) + (a%b) = a`

This is true for `div.s` with `rem.s` and `div.u` with `rem.u`.

The signed divide and remainder instructions truncate towards zero:

```
 5 / 2 =  2    5 % 2 =  1
-5 / 2 = -2   -5 % 2 = -1
 5 /-2 = -2    5 %-2 =  1
-5 /-2 =  2   -5 %-2 = -1
```

Divide by zero

If a division by zero is attempted, it has the following results:

- The result(s) of the instruction (quotient, remainder or both) are set to zero.
- The C flag is set for unsigned divides. The V flag is set for signed divides.
- in User or Trusted mode it will trigger the `DivideByZero_S` exception,
- in Supervisor or Interrupt mode it will trigger the `DivideByZero_R` exception, or
- in Recovery or NMI state will trigger a soft reset.

6.1.6 Compare operations

The compare instructions are used to set the processors's flags. These are used by the conditional branch instructions.

The 8-bit variants of the compare instructions operate on the bottom 8 bits of the operands only.

6.1.7 Shift and rotate

The XAP4 has signed (arithmetic) and unsigned (logical) shift right instructions. It also has shift left and rotate left instructions. Instructions can operate on either 16-bit registers or 32-bit register pairs.

A register can be shifted up to 15 bits left or right in a single instruction (register pairs can be shifted up to 31 bits). The last bit shifted out is copied to the carry flag. The carry flag is not modified for shifts of length zero.

The `rotatel` instruction performs a rotate left and does not circulate through the carry bit. The carry bit is set to the LSB of the result if the rotate length is non-zero.

The `b2swap.32` instruction swaps the order of the bytes in each 16-bit register of the register pair. This instruction can be followed by a `rotatel.32` to change the endianness of a 32-bit number.

```
// Change endianness of a 32-bit number in {%r1, %r0}
b2swap.32.r  %r0              // Bytes ABCD -> BADC
rotatel.32.i %r0, %r0, #16    // Bytes BADC -> DCBA
```

6.1.8 Block operations

The XAP4 has multi-cycle instructions for copying data and filling memory. These are similar to the `memcpy()` and `memset()` library functions in C.

Because the block operations can take many cycles, interrupts are processed during the execution of these instructions. All state is maintained in the registers and %flags[S]. If the block operation has not yet completed, the return address will be the block instruction. If the block operation has completed, the return address will be the next instruction to be executed. This allows the instruction to resume once the interrupt handler exits with `rtie`. The instruction is re-fetched after the return from interrupt.

The state of the registers when the instructions complete is undefined.

Block copy – blkcp.r and blkcp.i

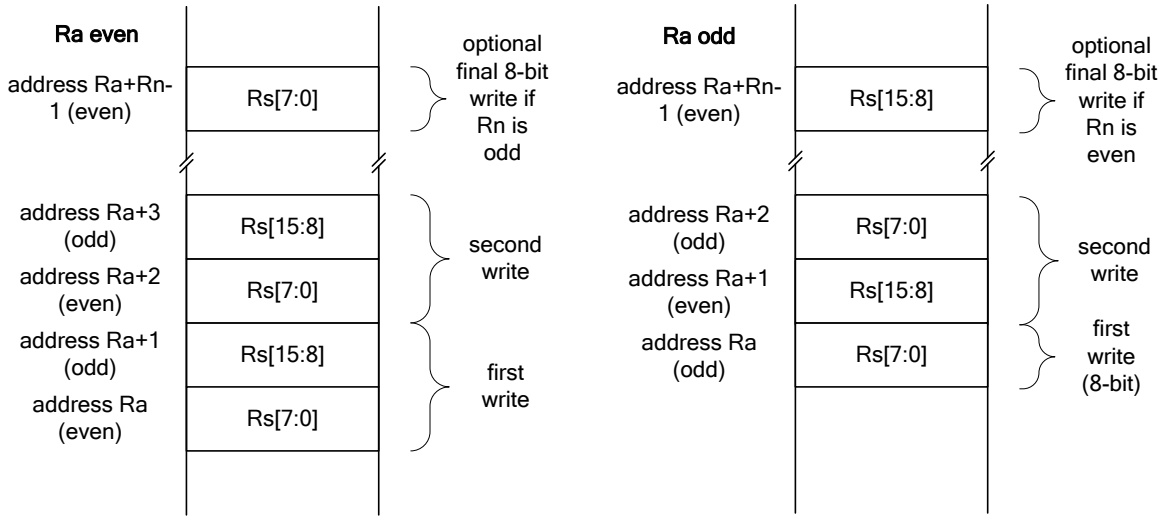
The `blkcp.r` and `blkcp.i` instructions copy a fixed number of bytes.

Copies use a mixture of 8-bit and 16-bit reads and writes:

		Rad	
		Odd	**Even**
Ras	**Odd**	An initial 8-bit operation is used, followed by 16-bit operations. An optional final 8-bit operation is used if needed.	8-bit operations are used.
	Even	8-bit operations are used.	16-bit operations are used. An optional final 8-bit operation is used if needed.

Block store – blkst.r and blkst.i

The `blkst.r` and `blkst.i` instructions use 16-bit writes where possible. An initial 8-bit write is used if the destination address is odd. A final 8-bit write is used if the final destination address is odd.



Block store – blkst.8.r and blkst.8.i

The `blkst.8.r` and `blkst.8.i` instructions copy the low byte of Rs into the high byte of Rs and then use the same write pattern as `blkst.r` and `blkst.i`. Thus, in general, `blkst.8.*` corrupts the Rs register.

6.1.9 DSP Instructions

The XAP4 has a powerful instruction for complex data manipulation. While not intended to be used by the C compiler this is a useful assembly instruction.

Instruction	Description
`mult.sh.r`	Multiplies two registers and then shifts the result right. Useful when multiplying coefficients and data, then normalising to pick out the resultant 16 bits of interest.

6.1.10 Single-bit instructions

The XAP4 provides a series of instructions for operating on single bits. Single bit memory accesses can be performed using load and store instructions (`ld.1.i` and `st.1.i`). A single bit move between a subset of the registers is possible with the `mov.1.*` instructions. Single bit logical operations (AND, OR and XOR) are also available, with the `ldand.1.i`, `ldor.1.i` and `ldxor.1.i`.

These instructions allow simple bit operations to be performed using only the carry bit in the FLAGS register. No other registers are needed.

The `bcc` and `bcs` instructions can be used to act conditionally upon the results of a bitwise expression.

6.1.11 Miscellaneous instructions

Sign extend

The `sext.r` instruction sign-extends an 8-bit value by copying the sign bit into the top 8 bits.

Interrupt-related

The `mov.1.i` and `mov.1.r` instructions can be used to access any flag in the FLAGS register, including the I flag.

`rtie` is used to return from an interrupt or exception.

These actions are not permitted in User mode.

Services

`syscall.r` and `syscall.i` call the various `SysCall` services. This allows processes to ensure that they are in a privileged mode to call privileged code, e.g. an

operating system function. The `syscall` instructions will not change mode in the privileged modes, but will run the handler in the current mode.

Sleeping and no-operation

`sleepnop` and `sleepsif` put the XAP4 into the NOP Sleep and SIF Sleep states, respectively. The hardware WAKE_UP signal is necessary to restart the processor. In the SIF Sleep state (`sleepsif`), SIF cycles are permitted. In the NOP Sleep state (`sleepnop`), SIF cycles are not permitted. Neither instruction is permitted in User mode.

`nop` is the no operation instruction.

Debug-related

`brk` implements a software breakpoint. If the processor is in User mode and the B bit (also called P0) in the FLAGS register is set, the `Break` exception is thrown. If the processor is not in User mode and is on debug mode the processor is stopped otherwise a `nop` is executed.

`halt` stops the processor. This instruction is not permitted in User mode.

`sif` allows the processor to perform a SIF cycle. This instruction is not permitted in User mode.

`print.r` causes xIDE to print a character in the debug window. This is useful for `putchar()` emulation during simulation. Hardware implementations of XAP4 treat this instruction as a `nop`.

The `ver` and `lic` instructions return information about the XAP4 core.

Reset

The `softreset` instruction causes a Soft Reset. This instruction is not permitted in User Mode.

Operations on special registers

`movr2s` and `movs2r` provide access to the special registers.

The `mov.1.i`, `mov.1.r`, `xor.1.i`, `mov.2.i`, `mov.2.r`, `mov.4.i` and `mov.4.r` instructions provide access to parts of the FLAGS register.

The `fimode` instruction can be used in any mode/state to find out the current mode/state.

Operations on breakpoint registers

`movr2b` and `movb2r` provide access to the breakpoint registers.

Operations on address registers

`mova2r` and `movr2a` provide access to the address registers.

6.2 Common Code Sequences

6.2.1 Function Prologue and Epilogue

For a function with no stack requirement, no prologue is necessary and the simplest function epilogue is:

```
   pop.ret {}, #0  // Return to caller by popping return
                   // address from the stack to PC
```

For arbitrary functions, the prologue and epilogue code varies according to:

- Which of the registers R4-R7 the function changes.
- The amount of stack needed for the function's automatic variables, if any.

6.2.2 Nested Interrupt Prologue and Epilogue

The only action required to allow nested interrupts is for the interrupt handler to re-enable interrupts. However specific code to preserve some or all of R2-R7 might be required before doing this.

```
push    {%r7-r2),  #0 // Pushing R7-R2 to the stack (optional)
mov.1.i %flags[i], %flags[c] // Re-enable interrupts

// further interrupts can now be processed safely
```

Then on exit `rtie` restores the necessary state:

```
pop   {%r2-%r7}, #0 // Popping R2-R7 from the stack (optional)
rtie                // Pop FLAGS, R0 and R1 from the stack
                    // Pop the return address to PC
```

6.2.3 Semaphore

The `swap.i` instruction can be used to implement a semaphore without needing to disable interrupts. Two assembly functions are needed – one to lock (acquire) a semaphore, and a second to release it. Alternatively, it could be written in C using inline assembly or library functions to generate the `swap` instructions.

The function `simpleGetSem()` attempts to acquire a semaphore. The `swap.i` instruction swaps-in the "locked" value, and gives the current semaphore state in R0.

```
// int simpleGetSem(int* sem)
simpleGetSem:
   mov.i   %r1, #1
   swap.i  %r1, @(0, %r0)
   mov.r   %r0, %r1
   pop.ret {}, #0
```

The function `simpleReleaseSem()` uses the `swap.i` instruction to set the semaphore state to "unlocked". The value retrieved from the semaphore should be

1 but this simple code does not check that the semaphore was locked on entry; nor that it was locked by the current process.

```
// void simpleReleaseSem(int* sem)
simpleReleaseSem:
   mov.i   %r1, #0
   swap.i  %r1, @(0, %r0)
   pop.ret {}, #0
```

These functions can be used from C as follows:

```
extern int  simpleGetSem    (int* sem);
extern void simpleReleaseSem(int* sem);

int mySemaphore;

{
      if ( 0 == osXSimpleGetSem(&mySemaphore) )
      {
          // We have the semaphore
          ...
          osXSimpleReleaseSem(&mySemaphore);
      }
}
```

6.2.4 Operating system task creation

Operating systems need to be able to manage many tasks running on the same processor and using the same registers. They need to be able to create new tasks and to save and restore the context of tasks. On a XAP4 the code managing the tasks will be running in a privileged mode. The tasks could be running in either Trusted or User mode. The code will need to initialise all registers available to the task before switching to the task.

The SP register is shadowed and should be initialised by setting SP1. User mode registers R0, R1 and FLAGS can be initialised by pushing the desired values to the privileged stack and then performing an `rtie` instruction, which will pop the values to the registers and enter User mode (if FLAGS[M] is set correctly).

```
   // Initialise FLAGS, R0, R1 and SP for User mode task
   // Processor is currently in Supervisor mode
   mov.i     %r0, (usertask,%pc) // Address of user task code
   push      {%r0}, #0           // Store as return address
   push.i    {#0x1111}, #0       // Store initial R1 as saved R1
   push.i    {#0x2222}, #0       // Store initial R0 as saved R0
   push.i    {#0x0070}, #0       // Store initial FLAGS as saved FLAGS
                                 // FLAGS: User Mode, interrupts enabled

   mov.i     %r0, (stacktop, 0)  // Top of task stack

   add.i     %r0, %r0, #-2       // Space for user task return address
   mov.i     %r2, (killtask,%pc) // To kill the user task if it ends
   st.i      %r2, @(0, %r0)      // Store on Stack0
```

```
movr2a    %sp1, %r0           // Set user SP (Stack Pointer)

// Return to User Mode
rtie // This will pop the stored FLAGS, R0, R1 & return address from
     // Stack0
```

6.3 Instructions Grouped by Function

6.3.1 Branches

Function	Mnemonic	Description	Flags
Unconditional Branch	bra.i bra.i.2 bra.i.4	Branch	----
	bra.m	Branch, via memory, displacement	----
Unconditional Branch Subroutine	bsr.i	Branch to subroutine	----
	bsr.m	Branch to subroutine, via memory, displacement	----
Conditional Branch	bcc	Branch if carry clear	----
	bcs	Branch if carry set	----
	beq	Branch if equal	----
	bez.r	Branch if register zero	ZNCV
	bge.s	Branch if greater than or equal, signed	----
	bge.u	Branch if greater than or equal, unsigned	----
	bgt.s	Branch if greater than, signed	----
	bgt.u	Branch if greater than, unsigned	----
	ble.s	Branch if less than or equal, signed	----
	ble.u	Branch if less than or equal, unsigned	----
	blt.s	Branch if less than, signed	----
	blt.u	Branch if less than, unsigned	----
	bmi	Branch if minus	----
	bne	Branch if not equal	----
	bnz.r	Branch if register not zero	ZNCV
	bpl	Branch if plus	----
	bvc	Branch if overflow clear	----
	bvs	Branch if overflow set	----

6.3.2 Load and Store

Function	Mnemonic	Description	Flags
Load (displacement)	ld.8z.i	Load, 8-bit, zero-extend, displacement	ZN--
	ld.i	Load, displacement	ZN--
	ld.32.i	Load, 32-bit, displacement	ZN--
Load (indexed)	ld.8z.r	Load, 8-bit, zero-extend, indexed	ZN--
	ld.r	Load, indexed	ZN--
	ld.32.r	Load, 32-bit, indexed	ZN--
Store (displacement)	st.8.i	Store, 8-bit, displacement	----
	st.i	Store, displacement	----
	st.32.i	Store, 32-bit, displacement	----
Store (indexed)	st.8.r	Store, 8-bit, indexed	----
	st.r	Store, indexed	----
	st.32.r	Store, 32-bit, indexed	----
Swap	swap.i	Swap register with memory	ZN--

6.3.3 Push and Pop

Function	Mnemonic	Description	Flags
Push	push	Push to stack	----
	push.i	Push immediates to stack	----
Pop	pop	Pop from stack	----
	pop.ret	Pop from stack and return	ZNCV

6.3.4 Move

Function	Mnemonic	Description	Flags
Move	mov.i	Move, immediate	ZN--
	mov.r	Move, register	ZN--
	mov.32.r	Move, register pair	ZN--
	mov.32s.r	Move, register to register pair, sign-extended	ZN--
	mov.32z.r	Move, register to register pair, zero-extended	ZN--

6.3.5 ALU operations

Function	Mnemonic	Description	Flags
Add	add.i	Add, immediate	ZNCV

	add.r	Add, register	ZNCV
	add.c.i	Add with carry, immediate	ZNCV
	add.c.r	Add with carry, register	ZNCV
Subtract	sub.r	Subtract, register	ZNCV
	sub.c.i	Subtract, with carry, immediate	ZNCV
	sub.c.r	Subtract, with carry, register	ZNCV
	sub.x.i	Subtract, exchanged order,immediate	ZNCV
	sub.xc.i	Subtract, with carry, exchanged order, immediate	ZNCV
Logical	and.i	And, immediate	ZN--
	and.r	And, register	ZN--
	or.i	Or, immediate	ZN--
	or.r	Or, register	ZN--
	xor.i	Exclusive-or, immediate	ZN--
	xor.r	Exclusive-or, register	ZN--
Multiply	mult.i	Multiply, immediate	ZN--
	mult.r	Multiply, register	ZN--
	mult.32s.i	Multiply, 32-bit signed, immediate	ZN--
	mult.32s.r	Multiply, 32-bit signed, register	ZN--
	mult.32u.i	Multiply, 32-bit unsigned, immediate	ZN--
	mult.32u.r	Multiply, 32-bit unsigned, register	ZN--
	mult.sh.r	Multiply, signed shift right, register	ZN-V
Divide and Remainder (signed)	div.s.i	Divide, signed, immediate	ZN-V
	div.s.r	Divide, signed, register	ZN-V
	div.32s.i	Divide, 32-bit signed, immediate	ZN-V
	div.32s.r	Divide, 32-bit signed, register	ZN-V
	divrem.s.i	Divide and remainder, signed, immediate	---V
	divrem.s.r	Divide and remainder, signed, register	---V
	divrem.32s.i	Divide and remainder, 32-bit signed, immediate	---V
	divrem.32s.r	Divide and remainder, 32-bit signed, register	---V
	rem.s.i	Remainder, signed, immediate	ZN-V
	rem.s.r	Remainder, signed, register	ZN-V
	rem.32s.i	Remainder, 32-bit signed, immediate	ZN-V
	rem.32s.r	Remainder, 32-bit signed, register	ZN-V

Divide and Remainder (unsigned)	div.u.i	Divide, unsigned, immediate	ZNC-
	div.u.r	Divide, unsigned, register	ZNC-
	div.32u.i	Divide, 32-bit unsigned, immediate	ZNC-
	div.32u.r	Divide, 32-bit unsigned, register	ZNC-
	divrem.u.i	Divide and remainder, unsigned, immediate	--C-
	divrem.u.r	Divide and remainder, signed, register	--C-
	divrem.32u.i	Divide and remainder, 32-bit unsigned, immediate	--C-
	divrem.32u.r	Divide and remainder, 32-bit unsigned, register	--C-
	rem.u.i	Remainder, unsigned, immediate	ZNC-
	rem.u.r	Remainder, unsigned, register	ZNC-
	rem.32u.i	Remainder, 32-bit unsigned, immediate	ZNC-
	rem.32u.r	Remainder, 32-bit unsigned, register	ZNC-

6.3.6 Compare operations

	16-bit	8-bit
Immediate	cmp.i	cmp.8.i
Register	cmp.r	cmp.8.r
Immediate with carry	cmp.c.i	cmp.8c.i
Register with carry	cmp.c.r	cmp.8c.r
Immediate, exchanged order	cmp.x.i	cmp.8x.i
Immediate with carry, exchanged order	cmp.xc.i	cmp.8xc.i

All cmp instructions affect the ZNCV flags.

6.3.7 Shift and rotate

Function	Mnemonic	Description	Flags
Shift left	shiftl.i	Shift left, immediate	ZNC-
	shiftl.r	Shift left, register	ZNC-
	shiftl.32.i	Shift left, 32-bit, immediate	ZNC-
	shiftl.32.r	Shift left, 32-bit, register	ZNC-
Shift right	shiftr.s.i	Shift right, signed, immediate	ZNC-
	shiftr.s.r	Shift right, signed, register	ZNC-
	shiftr.32s.i	Shift right, 32-bit signed, immediate	ZNC-
	shiftr.32s.r	Shift right, 32-bit signed, register	ZNC-

	shiftr.u.i	Shift right, unsigned, immediate	ZNC-
	shiftr.u.r	Shift right, unsigned, register	ZNC-
	shiftr.32u.i	Shift right, 32-bit unsigned, immediate	ZNC-
	shiftr.32u.r	Shift right, 32-bit unsigned, register	ZNC-
Rotate	rotatel.i	Rotate left, immediate	ZNC-
	rotatel.r	Rotate left, register	ZNC-
	rotatel.32.i	Rotate left, 32-bit, immediate	ZNC-
	rotatel.32.r	Rotate left, 32-bit, register	ZNC-
Byte swap	b2swap.32.r	Bytes swap, 32-bit, register	ZN--

6.3.8 Block copy and store

Function	Mnemonic	Description	Flags
Block copy	blkcp.i	Block copy, immediate	----
	blkcp.r	Block copy, register	----
Block store	blkst.8.i	Block store, 8-bit, immediate	----
	blkst.8.r	Block store, 8-bit, register	----
	blkst.i	Block store, immediate	----
	blkst.r	Block store, register	----

6.3.9 Single-bit instructions

Function	Mnemonic	Description	Flags
Read memory	ld.1.i	Load, single-bit, displacement (zero-relative)	--C-
	ldand.1.i	Load with AND, single-bit, displacement (zero-relative)	--C-
	ldor.1.i	Load with OR, single-bit, displacement (zero-relative)	--C-
	ldxor.1.i	Load with XOR (exclusive-or), single-bit, displacement (zero-relative)	--C-
Write memory	st.1.i	Store, single-bit, displacement (zero-relative)	----

6.3.10 CLU Instructions

Function	Mnemonic	Description	Flags
Customisable Logic Instructions	clu	CLU Instruction Type 1	----
	clu.d	CLU Instruction Type 2	ZN--
	clu.dd	CLU Instruction Type 3	ZN--

	clu.s	CLU Instruction Type 4	----
	clu.ss	CLU Instruction Type 5	----
	clu.st	CLU Instruction Type 6	----
	clu.sst	CLU Instruction Type 7	----
	clu.ds	CLU Instruction Type 8	ZN--
	clu.dss	CLU Instruction Type 9	ZN--
	clu.dds	CLU Instruction Type 10	ZN--
	clu.ddss	CLU Instruction Type 11	ZN--
	clu.dst	CLU Instruction Type 12	ZN--
	clu.dsst	CLU Instruction Type 13	ZN--
	clu.ddst	CLU Instruction Type 14	ZN--
	clu.ddsst	CLU Instruction Type 15	ZN--

6.3.11 Miscellaneous instructions

Function	Mnemonic	Description	Flags
Sign extend	sext.r	Sign-extend, register	ZN--
System instructions	brk	Break	----
	halt	Halt	----
	fimode	Flags and info mode	----
	nop	No operation	----
	sif	SIF	----
	sleepnop	Sleep	----
	sleepsif	Sleep and allow SIF	----
	print.r	Print, register	----
	lic	Read XAP4 licence number	----
	rtie	Return from interrupt/exception	ZNCV
	softreset	Soft Reset	----
	syscall.i	System call, immediate	----
	syscall.r	System call, register	----
	ver	Read XAP4 version number	----
FLAGS register	mov.1.i	Move, single-bit, immediate	--C-
	mov.1.r	Move, single-bit, register	--C-
	mov.2.i	Move, 2-bit, immediate	----

	mov.2.r	Move, 2-bit, register	----
	mov.4.i	Move, 4-bit, immediate	----
	mov.4.r	Move, 4-bit, register	----
	and.1.r	AND, single-bit, register	--C-
	or.1.r	OR, single-bit, register	--C-
	xor.1.i	XOR (exclusive-or), single-bit, immediate	--C-
	xor.1.r	XOR (exclusive-or), single-bit, register	--C-
Address registers	mova2r	Move address register to register	ZN--
	movr2a	Move register to address register	----
Breakpoint registers	movb2r	Move breakpoint register to register	ZN--
	movr2b	Move register to breakpoint register	----
Special registers	movs2r	Move special register to register	ZN--
	movr2s	Move register to special register	----

6.4 Alphabetical List of All Instructions

The XAP4 instruction set is listed alphabetically below. The "Sizes" column indicates whether the instruction can be encoded in 16 bits, 32 bits or both.

Note: for some instructions, not all possible operand combinations are shown. This table lists those intended for normal use. Others can be found in section 7.

Mnemonic	Operands	Operation	Sizes
add.c.i	Rd, Rs, #immediate	Add with carry, immediate	16, 32
add.c.r	Rd, Rs, Rt	Add with carry, register	16
add.i	Rd, Rs, #immediate	Add, immediate	16, 32
add.r	Rd, Rs, Rt	Add, register	16
and.1.r	%flags[c], %flags[c], Rs	AND, single-bit, register	16
and.i	Rd, Rs, #immediate	AND, immediate	16, 32
and.r	Rd, Rs, Rt	AND, register	16
b2swap.32.r	Rd	Byte swap, 32-bit, register	16
bcc	(label, %pc)	Branch if carry clear	16, 32
bcs	(label, %pc)	Branch if carry set	16, 32
beq	(label, %pc)	Branch if equal	16, 32
bez.r	Rs, (label, %pc)	Branch if register zero	16, 32
bge.s	(label, %pc)	Branch if greater than or equal, signed	16, 32
bge.u	(label, %pc)	Branch if greater than or equal, unsigned	32
bgt.s	(label, %pc)	Branch if greater than, signed	16, 32
bgt.u	(label, %pc)	Branch if greater than, unsigned	16, 32
ble.s	(label, %pc)	Branch if less than or equal, signed	16, 32
ble.u	(label, %pc)	Branch if less than or equal, unsigned	16, 32
blkcp.i	@(0, Rad), @(0, Ras), #num	Block copy, immediate	32
blkcp.r	@(0, Rad), @(0, Ras), Rn	Block copy, register	32
blkst.8.i	Rs, @(0, Ra), #num #0, @(0, Ra), #num #0xFF, @(0, Ra), #num	Block store, 8-bit, immediate	32

Mnemonic	Operands	Operation	Sizes
blkst.8.r	Rs, @(0, Ra), Rn #0, @(0, Ra), Rn #0xFF, @(0, Ra), Rn	Block store, 8-bit, register	32
blkst.i	Rs, @(0, Ra), #num	Block store, immediate	32
blkst.r	Rs, @(0, Ra), Rn	Block store, register	32
blt.s	(label, %pc)	Branch if less than, signed	16, 32
blt.u	(label, %pc)	Branch if less than, unsigned	32
bmi	(label, %pc)	Branch if minus	32
bne	(label, %pc)	Branch if not equal	16, 32
bnz.r	Rs, (label, %pc)	Branch if register not zero	16, 32
bpl	(label, %pc)	Branch if plus	32
bra.i bra.i.2 bra.i.4	(label, %pc)	Branch	16, 32
	(label, 0)		
	(0, Ra)		
bra.m	@(label, 0)	Branch, via memory, displacement	16, 32
	@(0, Ra)		
brk		Break	16
bsr.i	(label, %pc)	Branch to subroutine	32
	(label, 0)		
	(0, Ra)		
bsr.m	@(label, 0)	Branch to subroutine, via memory, displacement	16, 32
	@(0, Ra)		
bvc	(label, %pc)	Branch if overflow clear	32
bvs	(label, %pc)	Branch if overflow set	32
clu	#immediate	CLU Instruction Type 1	32
clu.d	#immediate, Rd	CLU Instruction Type 2	32
clu.dd	#immediate, Rd	CLU Instruction Type 3	32
clu.dds	#immediate, Rd, Rs	CLU Instruction Type 10	32
clu.ddss	#immediate, Rd, Rs	CLU Instruction Type 11	32
clu.ddsst	#immediate, Rd, Rs, Rt	CLU Instruction Type 15	32
clu.ddst	#immediate, Rd, Rs,	CLU Instruction Type 14	32

Mnemonic	Operands	Operation	Sizes
	Rt		
clu.ds	#immediate, Rd, Rs	CLU Instruction Type 8	32
clu.dss	#immediate, Rd, Rs	CLU Instruction Type 9	32
clu.dsst	#immediate, Rd, Rs, Rt	CLU Instruction Type 13	32
clu.dst	#immediate, Rd, Rs, Rt	CLU Instruction Type 12	32
clu.s	#immediate, Rs	CLU Instruction Type 4	32
clu.ss	#immediate, Rs	CLU Instruction Type 5	32
clu.sst	#immediate, Rs, Rt	CLU Instruction Type 7	32
clu.st	#immediate, Rs, Rt	CLU Instruction Type 6	32
cmp.8.i	Rs, #immediate	Compare, 8-bit, immediate	16, 32
cmp.8.r	Rs, Rt	Compare, 8-bit, register	16
cmp.8c.i	Rs, #immediate	Compare, 8-bit with carry, immediate	32
cmp.8c.r	Rs, Rt	Compare, 8-bit with carry, register	16
cmp.8x.i	Rs, #immediate	Compare, 8-bit, exchanged order, immediate	32
cmp.8xc.i	Rs, #immediate	Compare, 8-bit with carry, exchanged order, immediate	32
cmp.c.i	Rs, #immediate	Compare, with carry, immediate	32
cmp.c.r	Rs, Rt	Compare, with carry, register	16
cmp.i	Rs, #immediate	Compare, immediate	16, 32
cmp.r	Rs, Rt	Compare, register	16
cmp.x.i	Rs, #immediate	Compare, exchanged order, immediate	32
cmp.xc.i	Rs, #immediate	Compare, with carry, exchanged order, immediate	32
div.32s.i	Rd, Rs, #immediate	Divide, 32-bit signed, immediate	32
div.32s.r	Rd, Rs, Rt	Divide, 32-bit signed, register	32
div.32u.i	Rd, Rs, #immediate	Divide, 32-bit unsigned, immediate	32
div.32u.r	Rd, Rs, Rt	Divide, 32-bit unsigned, register	32
div.s.i	Rd, Rs, #immediate	Divide, signed, immediate	32
div.s.r	Rd, Rs, Rt	Divide, signed, register	32

Mnemonic	Operands	Operation	Sizes
div.u.i	Rd, Rs, #immediate	Divide, unsigned, immediate	32
div.u.r	Rd, Rs, Rt	Divide, unsigned, register	32
divrem.32s.i	Rd, Rs, #immediate	Divide and remainder, 32-bit signed, immediate	32
divrem.32s.r	Rd, Rs, Rt	Divide and remainder, 32-bit signed, register	32
divrem.32u.i	Rd, Rs, #immediate	Divide and remainder, 32-bit unsigned, immediate	32
divrem.32u.r	Rd, Rs, Rt	Divide and remainder, 32-bit unsigned, register	32
divrem.s.i	Rd, Rs, #immediate	Divide and remainder, signed, immediate	32
divrem.s.r	Rd, Rs, Rt	Divide and remainder, signed, register	32
divrem.u.i	Rd, Rs, #immediate	Divide and remainder, unsigned, immediate	32
divrem.u.r	Rd, Rs, Rt	Divide and remainder, unsigned, register	32
fimode	Rd	Flags and info mode	16
halt		Halt	16
ld.1.i	%flags[c], @(label[bit], 0) @(offset[bit], 0)	Load, single-bit, displacement (zero-relative)	32
ld.8z.i	Rd, @(offset, Ra)	Load, 8-bit, zero-extend, displacement	16, 32
	Rd, @(label, 0) Rd, @(offset, 0)		
	Rd, @(offset, %sp)		
	Rd, @(label, %pc) Rd, @(offset, %pc)		
ld.8z.r	Rd, @(Rx, Ra) Rd, @(Rx, %sp)	Load, 8-bit, zero-extend, indexed	16, 32
ld.32.i	Rd, @(offset, Ra)	Load, 32-bit, displacement	16, 32
	Rd, @(label, 0) Rd, @(offset, 0)		
	Rd, @(offset, %sp)		
	Rd, @(label, %pc) Rd, @(offset, %pc)		
ld.32.r	Rd, @(Rx, Ra) Rd, @(Rx, %sp)	Load, 32-bit, indexed	16, 32

Mnemonic	Operands	Operation	Sizes
ld.i	Rd, @(offset, Ra)	Load, displacement	16, 32
	Rd, @(label, 0) Rd, @(offset, 0)		
	Rd, @(offset, %sp)		
	Rd, @(label, %pc) Rd, @(offset, %pc)		
ld.r	Rd, @(Rx, Ra) Rd, @(Rx, %sp)	Load, indexed	16, 32
ldand.1.i	%flags[c], %flags[c], @(label[bit], 0) @(offset[bit], 0)	Load with AND, single-bit, displacement (zero-relative)	32
ldor.1.i	%flags[c], %flags[c], @(label[bit], 0) @(offset[bit], 0)	Load with OR, single-bit, displacement (zero-relative)	32
ldxor.1.i	%flags[c], %flags[c], @(label[bit], 0) @(offset[bit], 0)	Load with XOR (exclusive-or), single-bit, displacement (zero-relative)	32
lic	Rd	Read XAP4 licence number	32
mov.1.i	%flags[F], #immediate	Move, single-bit, immediate	16
mov.1.r	%flags[c], Rs	Move, single-bit, register	16, 32
	%flags[i], Rs		
	Rd, %flags[F]		
	%flags[i], %flags[c]		
	%flags[c], %flags[i]		
mov.2.i	%flags[m], #immediate	Move, 2-bit, immediate	16
mov.2.r	Rd, %flags[m]	Move, 2-bit, register	32
mov.4.i	%flags[p], #immediate	Move, 4-bit, immediate	16
mov.4.r	Rd, %flags[p]	Move, 4-bit, register	32
mov.32.r	Rd, Rs	Move, register pair	16
mov.32s.r	Rd, Rs	Move, register to register pair, sign-extended	16
mov.32z.r	Rd, Rs	Move, register to register pair, zero-extended	16
mov.i	Rd, #immediate	Move, displacement or immediate	16, 32
	Rd, (label, 0) Rd, (offset, 0)		
	Rd, (label, %pc)		

Mnemonic	Operands	Operation	Sizes
	@(offset[bit], %pc)		
mov.r	Rd, Rs	Move, register	16, 32
mova2r	Rd, As	Move address register to register	32
movb2r	Rd, Bs	Move breakpoint register to register	32
movr2a	Ad, Rs	Move register to address register	32
movr2b	Bd, Rs	Move register to breakpoint register	32
movr2s	Sd, Rs	Move register to special register	32
movs2r	Rd, Ss	Move special register to register	32
mult.32s.i	Rd, Rs, #immediate	Multiply, 32-bit signed, immediate	32
mult.32s.r	Rd, Rs, Rt	Multiply, 32-bit signed, register	32
mult.32u.i	Rd, Rs, #immediate	Multiply, 32-bit unsigned, immediate	32
mult.32u.r	Rd, Rs, Rt	Multiply, 32-bit unsigned, register	32
mult.i	Rd, Rs, #immediate	Multiply, immediate	32
mult.r	Rd, Rs, Rt	Multiply, register	32
mult.sh.r	Rd, Rs, Rt, #immediate	Multiply, signed shift right, register	32
nop		No operation	16
or.1.r	%flags[c], %flags[c], Rs	OR, single-bit, register	16
or.i	Rd, Rs, #immediate	OR, immediate	16, 32
or.r	Rd, Rs, Rt	OR, register	16
pop	RegList, #offset	Pop from stack	16, 32
pop.ret	RegList, #offset	Pop from stack and return	16, 32
print.r	Rs	Print, register	32
push	RegList, #offset	Push to stack	16, 32
push.i	{#i0}, #offset {#i0, #i1}, #offset {#i0, #i1, #i2}, #offset {#i0, #i1, #i2, #i3}, #offset	Push immediates to stack	16, 32
rem.32s.i	Rd, Rs, #immediate	Remainder, 32-bit signed, immediate	32
rem.32s.r	Rd, Rs, Rt	Remainder, 32-bit signed, register	32

Mnemonic	Operands	Operation	Sizes
rem.32u.i	Rd, Rs, #immediate	Remainder, 32-bit unsigned, immediate	32
rem.32u.r	Rd, Rs, Rt	Remainder, 32-bit unsigned, register	32
rem.s.i	Rd, Rs, #immediate	Remainder, signed, immediate	32
rem.s.r	Rd, Rs, Rt	Remainder, signed, register	32
rem.u.i	Rd, Rs, #immediate	Remainder, unsigned, immediate	32
rem.u.r	Rd, Rs, Rt	Remainder, unsigned, register	32
rotatel.32.i	Rd, Rs, #immediate	Rotate left, 32-bit, immediate	16, 32
rotatel.32.r	Rd, Rs, Rt	Rotate left, 32-bit, register	32
rotatel.i	Rd, Rs, #immediate	Rotate left, immediate	16, 32
rotatel.r	Rd, Rs, Rt	Rotate left, register	32
rtie		Return from interrupt/exception	16
sext.r	Rd	Sign extend, register	16
shiftl.32.i	Rd, Rs, #immediate	Shift left, 32-bit, immediate	32
shiftl.32.r	Rd, Rs, Rt	Shift left, 32-bit, register	32
shiftl.i	Rd, Rs, #immediate	Shift left, immediate	16
shiftl.r	Rd, Rs, Rt	Shift left, register	16
shiftr.32s.i	Rd, Rs, #immediate	Shift right, 32-bit signed, immediate	32
shiftr.32s.r	Rd, Rs, Rt	Shift right, 32-bit signed, register	32
shiftr.32u.i	Rd, Rs, #immediate	Shift right, 32-bit unsigned, immediate	32
shiftr.32u.r	Rd, Rs, Rt	Shift right, 32-bit unsigned, register	32
shiftr.s.i	Rd, Rs, #immediate	Shift right, signed, immediate	16
shiftr.s.r	Rd, Rs, Rt	Shift right, signed, register	16
shiftr.u.i	Rd, Rs, #immediate	Shift right, unsigned, immediate	16
shiftr.u.r	Rd, Rs, Rt	Shift right, unsigned, register	16
sif		SIF	16, 32
sleepnop		Sleep	16
sleepsif		Sleep and allow SIF	16
softreset		Soft reset	16
st.1.i	#imm[0], @(label[bit], 0)	Store, single-bit, displacement (zero-	32

Mnemonic	Operands	Operation	Sizes
	`@(offset[bit], 0)`	relative)	

	`%flags[c],`
	`@(label[bit], 0)`
	`@(offset[bit], 0)`		

`st.8.i`	`data, @(offset, Ra)`	Store, 8-bit, displacement	16, 32
	`data, @(label, 0)` `data, @(offset, 0)`		
	`data, @(offset, %sp)`		
	`data, @(label, %pc)` `data, @(offset, %pc)`		
`st.8.r`	`data, @(Rx, Ra)` `data, @(Rx, %sp)`	Store, 8-bit, indexed	16
`st.32.i`	`data, @(offset, Ra)`	Store, 32-bit, displacement	16, 32
	`data, @(label, 0)` `data, @(offset, 0)`		
	`data, @(offset, %sp)`		
	`data, @(label, %pc)` `data, @(offset, %pc)`		
`st.32.r`	`data, @(Rx, Ra)` `data, @(Rx, %sp)`	Store, 32-bit, indexed	16
`st.i`	`data, @(offset, Ra)`	Store, displacement	16, 32
	`data, @(label, 0)` `data, @(offset, 0)`		
	`data, @(offset, %sp)`		
	`data, @(label, %pc)` `data, @(offset, %pc)`		
`st.r`	`data, @(Rx, Ra)` `data, @(Rx, %sp)`	Store, indexed	16
`sub.c.i`	`Rd, Rs, #0`	Subtract, with carry, immediate	16
`sub.c.r`	`Rd, Rs, Rt`	Subtract, with carry, register	16
`sub.r`	`Rd, Rs, Rt`	Subtract, register	16
`sub.x.i`	`Rd, Rs, #immediate`	Subtract, exchanged order, immediate	16, 32
`sub.xc.i`	`Rd, Rs, #immediate`	Subtract, exchanged order, with carry, immediate	16, 32
`swap.i`	`Rd, @(0, Ra)`	Swap register with memory	32
`syscall.i`	`num, #immediate`	System call, immediate	32
`syscall.r`	`num, Rs`	System call, register	32

Mnemonic	Operands	Operation	Sizes
ver	Rd	Read XAP4 version number	32
xor.1.i	%flags[c], %flags[c], #1	XOR (exclusive-or), single-bit, immediate	16
xor.1.r	%flags[c], %flags[c], Rs	XOR (exclusive-or), single-bit, register	16
xor.i	Rd, Rs, #immediate	XOR (exclusive-or), immediate	16, 32
xor.r	Rd, Rs, Rt	XOR (exclusive-or), register	16

6.5 Privileged Instructions

The following instructions are not permitted in User mode:

- `mov.1.i` (unless operating on the Z,N,C or V flags), `mov.2.i`, `mov.4.i`
- `mov.1.r %flags[i], %flags[c]`
- `movr2s`, `movs2r` (unless operating on the flags register), `movr2b`, `movb2r`, `movr2a`, `mova2r`
- `sleepsif`, `sleepnop`, `softreset`, `halt`, `sif`
- `rtie`

These instructions either provide access to the FLAGS register (thereby allowing them to change the processor mode), or otherwise are capable of affecting the operation of the XAP4 core.

If a privileged instruction is executed in User mode, the XAP4 throws a `PrivInstruction_S` exception.

6.6 Aliased Instructions

The assembler translates a small number of instructions into other, equivalent instuctions. These are:

Original instruction	Translated into
blt.u label	bcs label
bge.u label	bcc label
bra.i.2	bra.i (16-bit encoding)
bra.i.4	bra.i (32-bit encoding)

The instructions are disassembled into the translated instructions by xIDE.

6.7 Register Naming in Instructions

Register operands are named as follows:

Register symbol	Description
Rd	Destination register.
Rs	Primary source register.
Rt	Secondary source register.
Ra	Register containing a base address. This is interpreted as a byte address.
Rad	Register containing a base destination address for use with `blkcp.*` This is interpreted as a byte address.
Ras	Register containing a base source address for use with `blkcp.*` This is interpreted as a byte address.
Rn	Register containing a number. This is interpreted as the number of bytes to be copied by a `blk` instruction.
Rx	Register containing an index. This is interpreted according to the width of the instruction. Rx is used in the indexed addressing mode versions of the load and store instructions.
Sd	Destination special register.
Ss	Source special register.
Ad	Destination address register.
As	Source address register.
Bd	Destination breakpoint register.
Bs	Source breakpoint register.

6.8 Register Specification Fields

3-bit register fields

Registers are generally specified with a 3-bit field. The mapping from the 3-bit field to the 16-bit register set is as follows:

3-bit R field	register
000	%r0
001	%r1

010	%r2
011	%r3
100	%r4
101	%r5
110	%r6
111	%r7

Address register fields

The mapping from a 3-bit address register field to the address registers is:

Ax	register
000	`%sp0`
001	`%sp1`
010	`%vp`
011-111	reserved

Special register fields

The mapping from a 3-bit special register field to the special registers is:

Sx	register
000	`%flags`
001	`%info`
010	`%brke`
011 - 111	reserved

Refer to section `3.4.3`, "Special registers", for details of the bits within the `FLAGS`, `INFO` and `BRKE` registers.

Breakpoint register fields

The mapping from a 3-bit breakpoint register field to the breakpoint register is:

Bx	register
000	`%brk0`
001	`%brk1`
010 - 111	reserved

6.9 Immediates

Many instructions take an immediate operand which is encoded in the instruction. For a subset of the permitted immediate values, the instruction encoding rules allow 16-bit immediates to be compressed in the instruction encoding. The rules governing immediate encoding are:

- Zero or sign extension.
- implicit bits.

Zero or sign extension

In the 16-bit instruction encoding, the immediate field is typically 4 or 8 bits wide. In some of these instructions, the XAP4 extends the immediate to 16 bits before use. This is either a zero-extension or a sign-extension, depending on the instruction.

Sign-extension of an immediate is indicated by an "s" suffix in the immediate field in the instruction descriptions. Zero-extension is indicated by a "u" or "z" suffix.

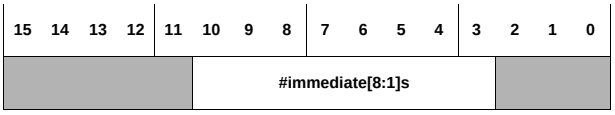
If the assembler cannot represent the immediate in the shortened form, it selects the equivalent 32-bit encoding of the instruction.

Implicit bits

In some cases, not all bits of the operand are encoded in the instruction.

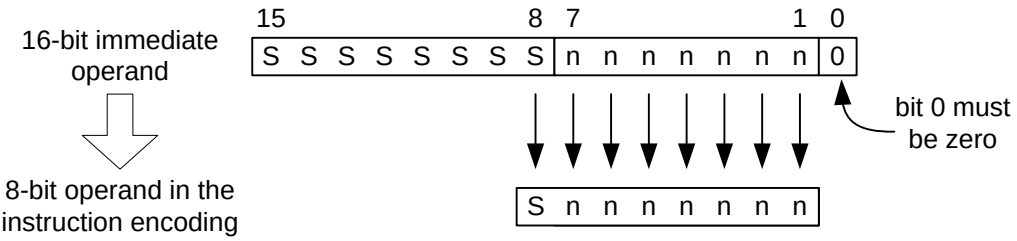
Example

An example of an instruction with sign-extension and implicit bits is a 16-bit encoding of the `add.i` instruction:



This syntax indicates that only bits 8 to 1 of the immediate are encoded in the instruction. Bit 0 is an implicit 0.

The "s" suffix indicates that bits 15 to 9 are sign-extended from bit 8 before use in the ALU.



This encoding can therefore represent immediates in the range -256, -254, …, +252, +254. Immediates outside this range require an alternative encoding of the instruction.

6.10 Register List Fields

Two-bit RegList fields are used in the 16-bit encodings of the `push` and `pop.ret` instructions when an offset is used, and are interpreted as follows:

2-bit RegList	register(s)	
	`push`	`pop.*`
00	`{%r4}`	`{%r4}`
01	`{%r5, %r4}`	`{%r4, %r5}`
10	`{%r6-%r4}`	`{%r4-%r6}`
11	`{%r7-%r4}`	`{%r4-%r7}`

7 Instruction Set Reference

The following pages describe each instruction in detail. Instructions are ordered alphabetically.

add.c.i Add with carry, immediate

Instruction `add.c.i Rd, Rs, #immediate`

Description Add immediate[15:0]s and the carry bit to a register

Flags

Z	Set if the result is zero and the Z flag was already set; cleared otherwise
N	Set if the result is negative; cleared if the result is positive
C	Set if the result of the unsigned operation is not correct; cleared otherwise
V	Set if the result of the signed operation is not correct; cleared otherwise

Operation `Rd = Rs + #immediate[15:0]s + C`

Usage Notes `add.c.i` can be used for signed or unsigned, integer or fixed-point arithmetic.

The Z flag behaviour is useful for 32-bit arithmetic.

Examples `add.c.i %r1, %r2, #10`

32-bit Encoding

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
#immediate[15:0]s				Rd	Rs	1 0 1 0 0 1	0 1 1 1

16-bit Encoding - 1

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 1 1 1	1 0 Rd	Rs	0 0 1 1

The following rule applies:

- The immediate is 0x0.

16-bit Encoding - 2

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 1 1 1	1 0 Rd	Rs	1 0 1 1

The following rule applies:

- The immediate is 0xFFFF.

16-bit Encoding - 3

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
Rd	1 1 1 0 1 0	#imm[3:0]u	0 1 1

The following rules apply:

- Rs is the same as Rd.
- This encoding can represent immediates in the range 0 to 15.

add.c.r — Add with carry, register

Instruction

```
add.c.r Rd, Rs, Rt
```

Description

Add two registers and the carry bit

Flags

Z	Set if the result is zero and the Z flag was already set; cleared otherwise
N	Set if the result is negative; cleared if the result is positive
C	Set if the result of the unsigned operation is not correct; cleared otherwise
V	Set if the result of the signed operation is not correct; cleared otherwise

Operation

```
Rd = Rs + Rt + C
```

Usage Notes

`add.c.r` can be used for signed or unsigned, integer or fixed-point arithmetic.

The Z flag behaviour is useful for 32-bit arithmetic.

Examples

```
add.c.r %r1, %r2, %r3
```

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Rs				Rt				1	1	0	1	1	1	0

add.i — Add, immediate

Instruction

```
add.i Rd, Rs, #immediate
add.i Rd, %sp, #immediate
add.i %sp, %sp, #immediate
```

Description

Add a register and an immediate[15:0]s

Flags

Z	Set if the result is zero; cleared if the result is non-zero
N	Set if the result is negative; cleared if the result is positive
C	Set if the result of the unsigned operation is not correct; cleared otherwise
V	Set if the result of the signed operation is not correct; cleared otherwise

Operation

```
Rd = Rs + #immediate[15:0]s
Rd = SP + #immediate[15:0]s
SP = SP + #immediate[15:0]s
```

Usage Notes

`add.i` can be used for signed or unsigned, integer or fixed-point arithmetic.

For small immediates, 16-bit encodings of this instruction are available.

SP is valid in place of the operand Rs.

SP is valid in place of the operand Rd, but only if SP is used for the source as well. If this is the case the flags are not updated.

Examples

```
add.i %r1, %r2, #0x1234
add.i %r1, %r2, #0xFEDC
```

32-bit Encoding - 1

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
#immediate[15:0]s				Rd	Rs	1 0 0 0 0 1	0 1 1 1

32-bit Encoding - 2

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
#immediate[15:0]s				Rd	0 0 0 0 1	1 0 1 0	1 1 1 1

The following rule applies:

- Rs is SP.

32-bit Encoding - 3

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

#immediate[15:1]s	0	0	0	0	0	1	0	0	1	1	0	1	0	1	1	1	1

The following rules apply:

- Rs is SP.
- Rd is SP.
- This encoding does not update the flags.

16-bit Encoding - 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	Rd		Rs				0	0	1	1

The following rule applies:

- The immediate is 0x1.

16-bit Encoding - 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	Rd		Rs				1	0	1	1

The following rule applies:

- The immediate is 0xFFFF.

16-bit Encoding - 3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs			1	1	1	0	0	1	#immediate[3:0]u			0	1	1	

The following rule applies:

- Rd is R2
- This encoding can represent immediates in the range 0, 1,…, 14, 15.

16-bit Encoding - 4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd			1	1	1	1	1	0	#immediate[3:0]u			0	1	1	

The following rule applies:

- Rs is Rd
- The immediate value is (#immediate[3:0]u – 16) and this gives a range of -1, -2, …, -15, -16.

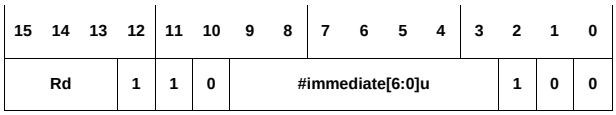
16-bit Encoding - 5

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd			0	1	1	#immediate[6:0]u						1	0	0	

The following rules apply:

- Rs is Rd.
- This encoding can represent immediates in the range 0, 1, …, 126, 127.

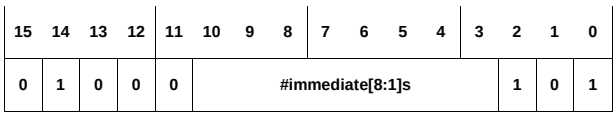
16-bit Encoding - 6



The following rules apply:

- Rs is SP.
- This encoding can represent immediates in the range 0, 1, … 126, 127.

16-bit Encoding - 7



The following rules apply:

- Rs is SP.
- Rd is SP.
- This encoding can represent immediates in the range -256, -254, …, 252, 254.
- This encoding does not update the flags.

add.r — Add, register

Instruction `add.r Rd, Rs, Rt`

Description Add two registers

Flags

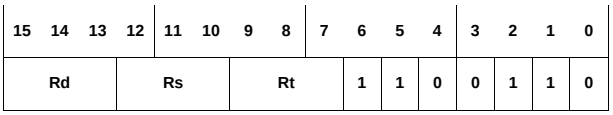
Z	Set if the result is zero; cleared if the result is non-zero
N	Set if the result is negative; cleared if the result is positive
C	Set if the result of the unsigned operation is not correct; cleared otherwise
V	Set if the result of the signed operation is not correct; cleared otherwise

Operation `Rd = Rs + Rt`

Usage Notes `add.r` can be used for signed or unsigned, integer or fixed-point arithmetic.

Examples `add.r %r1, %r2, %r3`

16-bit Encoding



and.i AND, immediate

Instruction `and.i Rd, Rs, #immediate`

Description Bitwise AND of register with an immediate[15:0]u value

Flags

Z	Set if the result is zero; cleared if the result is non-zero
N	Set if the result is negative; cleared if the result is positive
C	Unchanged
V	Unchanged

Operation `Rd = Rs & #immediate[15:0]u`

Usage Notes A number of commonly used combinations of registers and immediates are available in four 16-bit encodings of this instruction.

Examples

```
and.i %r2, %r1, #0x1234
and.i %r2, %r1, #0xFEDC
```

32-bit Encoding

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
#immediate[15:0]																Rd				Rs				0	0	0	0	0	1	0	1	1	1

16-bit Encoding - 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	Rd			Rs			0	0	1	1

The following rule applies:

- The immediate is 0x1.

16-bit Encoding - 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	Rd			Rs			1	0	1	1

The following rule applies:

- The immediate is 0xFF.

16-bit Encoding - 3

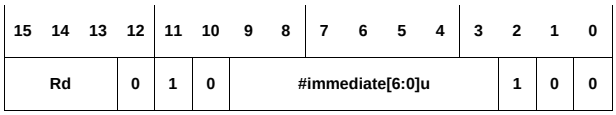
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd			1	1	1	1	0	1	#immediate[3:0]u			0	1	1	

The following rules apply:

- Rd is the same as Rs.
- The immediate value is (#immediate[3:0]u – 16) and this gives a

range of -1, -2, …, -15, -16.

16-bit Encoding - 4



The following rules apply:

- Rd is the same as Rs.
- This encoding can represent immediates in the range 0 to 127.

and.1.r AND, single-bit, register

Instruction `and.1.r %flags[c], %flags[c], Rs`

Description Bitwise AND of carry-flag with Rs[0]

Flags

Z	Unchanged
N	Unchanged
C	Set to the result of a bitwise AND of original value and Rs[0]
V	Unchanged

Operation `FLAGS[C] = FLAGS[C] & Rs[0]`

Usage Notes Used to alter the value of the carry flag.

Examples `and.1.r %flags[c], %flags[c], %r1`

16-bit Encoding

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

and.r — AND, register

Instruction `and.r Rd, Rs, Rt`

Description Bitwise AND of two registers

Flags

Z	Set if the result is zero; cleared if the result is non-zero
N	Set if the result is negative; cleared if the result is positive
C	Unchanged
V	Unchanged

Operation `Rd = Rs & Rt`

Usage Notes -

Examples `and.r %r1, %r2, %r3`

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Rs				Rt				0	1	1	0	1	1	0

b2swap.32.r — Byte swap, 32-bit, register

Instruction

```
b2swap.32.r Rd
```

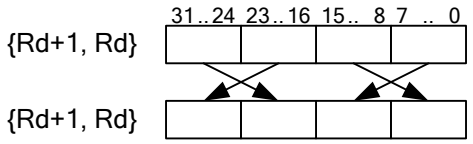
Description

Swap the order of the bytes in each 16-bit register of a register pair

Flags

Z	Set if the result is zero; cleared if the result is non-zero
N	Set if the result is negative; cleared if the result is positive
C	Unchanged
V	Unchanged

Operation



Usage Notes

This instruction can be combined with `rotatel.32.i` to efficiently change the endianness of a 32-bit number in a register pair.

```
b2swap.32.r %r0                 // Bytes ABCD -> BADC
rotatel.32.i %r0, %r0, #16      // Bytes BADC -> DCBA
```

Examples

```
b2swap.32.r %r0
```

16-bit Encoding

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

bcc — Branch if carry clear

Instruction

```
bcc (label, %pc)
bcc (offset, %pc)
```

Description

Branch if carry clear

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( C == 0 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

The `bge.u` (branch if greater than or equal, unsigned) instruction is translated into `bcc` by the assembler.

Examples

```
bcc (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0				0 0 1 0	1 0 0 1	0 0 1 0	1 1 1 1

16-bit Encoding

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 0 1 1	1 1 #offset[7:1]s		1 0 1

The following rule applies:

- This encoding can represent offsets in the range -128, -126, …, +124, +126.

bcs — Branch if carry set

Instruction

```
bcs (label, %pc)
bcs (offset, %pc)
```

Description

Branch if carry set

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( C == 1 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

The `blt.u` (branch if less than, unsigned) instruction is translated into `bcs` by the assembler.

Examples

```
bcs (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0				0 0 0 0	1 0 0 1	0 0 1 0	1 1 1 1

16-bit Encoding

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 0 1 1	1 0	#offset[7:1]s	1 0 1

The following rule applies:

- This encoding can represent offsets in the range -128, -126, …, +124, +126.

beq — Branch if equal

Instruction

```
beq (label, %pc)
beq (offset, %pc)
```

Description

Branch if equal

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( Z == 1 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
beq (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0																0	0	0	0	0	1	0	1	0	0	1	0	1	1	1	1

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	#offset[8:1]s								1	0	1

The following rule applies:

- This encoding can represent offsets in the range -256, -254, …, +252, +254.

bez.r — Branch if register zero

Instruction

```
bez.r Rs, (label, %pc)
bez.r Rs, (offset, %pc)
```

Description

Branch if Rs is zero

Flags

Z	Set if Rs is zero; cleared otherwise
N	Set if Rs[15] is set; cleared otherwise
C	Cleared
V	Cleared

Operation

```
if ( Rs == 0 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

`bez.r` is equivalent to these two instructions:

```
cmp.r  Rs, #0
beq    (label, %pc)
```

Examples

```
bez.r %r0, (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0																1	0	0	Rs			0	0	1	0	1	0	1	1	1	1

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Rs			0	#offset[6:1]s					1	0	1	

The following rule applies:

- This encoding can represent offsets in the range -64, -62, …, +60, +62.

bge.s Branch if greater than or equal, signed

Instruction

```
bge.s (label, %pc)
bge.s (offset, %pc)
```

Description

Branch if greater than or equal, signed

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( N == V )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
bge.s (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0				1 0 1 0	0 1 0 1	0 0 1 0	1 1 1 1

16-bit Encoding

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 0 1 0	1 #offset[8:1]s		1 0 1

The following rule applies:

- This encoding can represent offsets in the range -256, -254, …, +252, +254.

bge.u Branch if greater than or equal, unsigned

Instruction

```
bge.u (label, %pc)
bge.u (offset, %pc)
```

Description

Branch if greater than

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( C == 0 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

This instruction is translated into `bcc` by the assember. xIDE prints this instruction as `bcc (label, %pc)`.

Examples

```
bge.u (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0																0	0	1	0	1	0	0	1	0	0	1	0	1	1	1	1

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	#offset[7:1]s						1	0	1	

The following rule applies:

- This encoding can represent offsets in the range -128, -126, …, +124, +126.

bgt.s — Branch if greater than, signed

Instruction

```
bgt.s (label, %pc)
bgt.s (offset, %pc)
```

Description

Branch if greater than, signed

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( (N == V) && (Z == 0) )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
bgt.s (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0																0	1	1	0	0	1	0	1	0	0	1	0	1	1	1	1

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	#offset[8:1]s								1	0	1

The following rule applies:

- This encoding can represent offsets in the range -256, -254, …, +252, +254.

bgt.u — Branch if greater than, unsigned

Instruction

```
bgt.u (label, %pc)
bgt.u (offset, %pc)
```

Description

Branch if greater than, unsigned

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( (C == 0) && (Z == 0) )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

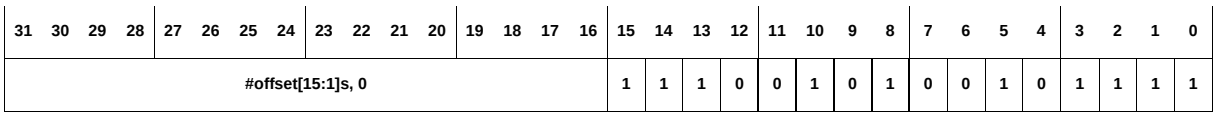
The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

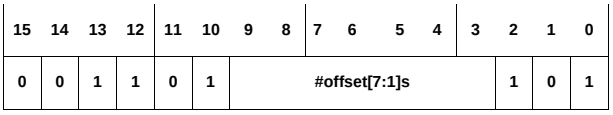
```
bgt.u (label, %pc)
```

32-bit Encoding



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
#offset[15:1]s, 0				1 1 1 0	0 1 0 1	0 0 1 0	1 1 1 1

16-bit Encoding



15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 0 1 1	0 1	#offset[7:1]s	1 0 1

The following rule applies:

- This encoding can represent offsets in the range -128, -126, …, +124, +126.

ble.s Branch if less than or equal, signed

Instruction

```
ble.s (label, %pc)
ble.s (offset, %pc)
```

Description

Branch if less than or equal, signed

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( (N != V) && (Z == 1) )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
ble.s (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0				0 1 0 0	0 1 0 1	0 0 1 0	1 1 1 1

16-bit Encoding

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 0 0 1	0 #offset[8:1]s		1 0 1

The following rule applies:

- This encoding can represent offsets in the range -256, -254, …, +252, +254.

ble.u Branch if less than or equal, unsigned

Instruction

```
ble.u (label, %pc)
ble.u (offset, %pc)
```

Description

Branch if less than or equal, unsigned

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( (C == 1) || (Z == 1) )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
ble.u (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0				1 1 0 0	0 1 0 1	0 0 1 0	1 1 1 1

16-bit Encoding

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 0 1 1	0 0 #offset[7:1]s		1 0 1

The following rule applies:

- This encoding can represent offsets in the range -128, -126, …, +124, +126.

blkcp.i — Block copy, immediate

Instruction

```
blkcp.i @(0, Rad), @(0, Ras), #num
```

Description

Copy num bytes from the address in Ras to the address in Rad

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
while (#num > 0)
{
    #num--
    *Rad = *Ras
    Rad++
    Ras++
}
```

Usage Notes

#num bytes of data are copied from the address specified in Ras to the address specified in Rad. The source and destination addresses increment during the copy. If the source area and destination area overlap, part of the source data may be overwritten.

The instruction can be interrupted before the copy is complete. When the interrupt handler exits, the copy resumes.

This instruction is useful for implementing `memcpy()`-like functions.

Refer to section 6.1.8, "Block operations", for more details of this instruction.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
blkcp.i @(0, %r1), @(0, %r0), #0x8
```

32-bit Encoding

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
Rad				Ras				1	1	0	n[4:0]u					1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1

n[4:0] is coded as follows:

- #num = 1 to 31 : n[4:0] = #num.
- #num = 32 : n[4:0] = 0.

blkcp.r — Block copy, register

Instruction

```
blkcp.r @(0, Rad), @(0, Ras), Rn
```

Description

Copy Rn bytes from Ras to Rad

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
while (Rn > 0)
{
    Rn--
    *Rad = *Ras
    Rad++
    Ras++
}
```

Usage Notes

Rn bytes of data are copied from the address specified in Ras to the address specified in Rad. The source and destination addresses increment during the copy. If the source area and destination area overlap, part of the source data may be overwritten.

The instruction can be interrupted before the copy is complete. When the interrupt handler exits, the copy resumes.

This instruction is useful for implementing `memcpy()`-like functions.

Refer to section 6.1.8, "Block operations", for more details of this instruction.

If Rn is zero, the instruction copies no data.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
blkcp.r @(0, %r1), @(0, %r0), %r3
```

32-bit Encoding

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
Rad				Ras				Rn				0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

blkst.8.i Block store, 8-bit, immediate

Instruction

```
blkst.8.i Source, @(0, Ra), #num
```

Description

Stores the low 8 bits of Source to a block of memory

Source is one of the following:

Normal Register %r0 … %r7

Immediate #0 or #0xFF

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
while (#num > 0)
{
    #num--
    *Ra = Source[7:0]
    Ra++
}
```

Usage Notes

The low 8 bits of Source are stored to a block of memory starting at an address specified in Ra and length specified by #num.

The destination address increments during the store.

The instruction can be interrupted before the store is complete. When the interrupt handler exits, the store resumes.

This instruction is useful for implementing `memset()`-like functions.

Refer to section 6.1.8, "Block operations", for more details of this instruction.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
blkst.8.i %r1, @(0, %r2), #0x8
blkst.8.i #0, @(0, %r2), #0x8
blkst.8.i #0xFF, @(0, %r2), #0x8
```

32-bit Encoding

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
OPD2	Ra	OPB2	1 0	n[4:0]u	1 1 1 1	1 0 1 1	1 1 1 1	1 1 1 1

n[4:0] is coded as follows:

- #num = 1 to 31 : n[4:0] = #num.
- #num = 32 : n[4:0] = 0.

Source can be a register (Rs) or an immediate.

OPB2	OPD2	Source
0	Rs	Rs = OPD2
2	0	#0
	1	#-1

blkst.8.r — Block store, 8-bit, register

Instruction

```
blkst.8.r Source, @(0, Ra), Rn
```

Description

Stores the low 8 bits of Source to a block of memory

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
while (Rn > 0)
{
    Rn--
    *Ra = Source[7:0]
    Ra++
}
```

Usage Notes

The low 8 bits of Rs are stored to a block of memory starting at an address specified in Ra and length specified in Rn.

The destination address increments during the store.

The instruction can be interrupted before the store is complete. When the interrupt handler exits, the store resumes.

This instruction is useful for implementing `memset()`-like functions.

Refer to section 6.1.8, "Block operations", for more details of this instruction.

If Rn is zero, the instruction stores no data.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
blkst.8.r %r1, @(0, %r2), %r3
blkst.8.r #0, @(0, %r2), %r3
blkst.8.r #0xFF, @(0, %r2), %r3
```

32-bit Encoding

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
OPD2				Ra				Rn				OPA2				0	1	1	1	1	1	1	1			1	1	1	1	1	1	1	1

Source can be a register (Rs) or an immediate.

OPA2	OPD2	Source
0	Rs	Rs = OPD2

2	0	#0
	1	#-1

blkst.i — Block store, immediate

Instruction

```
blkst.r Rs, @(0, Ra), #num
```

Description

Stores a register to a block of memory

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
while (#num > 0)
{
    #num--
    *Ra = Rs
    Ra++
}
```

Usage Notes

Rs is stored to a block of memory starting at an address specified in Ra and length specified by #num.

The destination address increments during the store.

The instruction can be interrupted before the store is complete. When the interrupt handler exits, the store resumes.

This instruction is useful for implementing `memset()`-like functions.

Refer to section 6.1.8, "Block operations", for more details of this instruction.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
blkst.i %r1, @(0, %r2), #0x8
```

32-bit Encoding

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
Rs				Ra				0	0	1	1	0	n[4:0]u			1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1

n[4:0] is coded as follows:

- #num = 1 to 31 : n[4:0] = #num.
- #num = 32 : n[4:0] = 0.

blkst.r — Block store, register

Instruction

```
blkst.r Rs, @(0, Ra), Rn
```

Description

Stores a register to a block of memory

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
while (Rn > 0)
{
    Rn--
    *Ra = Rs
    Ra++
}
```

Usage Notes

Rs is stored to a block of memory starting at an address specified in Ra and length specified in Rn.

The destination address increments during the store.

The instruction can be interrupted before the store is complete. When the interrupt handler exits, the store resumes.

This instruction is useful for implementing `memset()`-like functions.

Refer to section 6.1.8, "Block operations", for more details of this instruction.

If Rn is zero, the instruction stores no data.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
blkst.r %r1, @(0, %r2), %r3
```

32-bit Encoding

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
Rs				Ra				Rn				0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

blt.s — Branch if less than, signed

Instruction

```
blt.s (label, %pc)
blt.s (offset, %pc)
```

Description

Branch if less than, signed

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( N != V )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
blt.s (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0				1 0 0 0	0 1 0 1	0 0 1 0	1 1 1 1

16-bit Encoding

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 0 1 0	0 #offset[8:1]s		1 0 1

The following rule applies:

- This encoding can represent offsets in the range -256, -254, …, +252, +254.

blt.u Branch if less than, unsigned

Instruction

```
blt.u (offset, %pc)
blt.u (offset, %pc)
```

Description

Branch if less than

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( C == 1 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

This instruction is translated into `bcs` by the assember. xIDE prints this instruction as `bcs (label,%pc)`.

Examples

```
blt.u (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0																0	0	0	0	1	0	0	1	0	0	1	0	1	1	1	1

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	#offset[7:1]s							1	0	1

The following rule applies:

- This encoding can represent offsets in the range -128, -126, …, +124, +126.

bmi — Branch if minus

Instruction

```
bmi (label, %pc)
bmi (offset, %pc)
```

Description

Branch if minus

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( N == 1 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

Examples

```
bmi (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0																1	0	0	0	1	0	0	1	0	0	1	0	1	1	1	1

bne — Branch if not equal

Instruction

```
bne (label, %pc)
bne (offset, %pc)
```

Description

Branch if not equal

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( Z == 0 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
bne (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0																0	0	1	0	0	1	0	1	0	0	1	0	1	1	1	1

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	#offset[8:1]s								1	0	1

The following rule applies:

- This encoding can represent offsets in the range -256, -254, …, +252, +254.

conbnz.r — Branch if register not zero

Instruction

```
bnz.r Rs, (label, %pc)
bnz.r Rs, (offset, %pc)
```

Description

Branch if Rs is not zero

Flags

Z	Set if Rs is zero; cleared otherwise
N	Set if Rs[15] is set; cleared otherwise
C	Cleared
V	Cleared

Operation

```
if ( Rs != 0 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

For small offsets, a 16-bit encoding of this instruction is available.

`bnz.r` is equivalent to these two instructions:

```
cmp.r  Rs, #0
bne    (label, %pc)
```

Examples

```
bnz.r %r0, (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0																1	0	1	Rs			0	0	1	0	1	0	1	1	1	1

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Rs			1	#offset[6:1]s					1	0	1	

The following rule applies:

- This encoding can represent offsets in the range -64, -62, …, +60, +62.

bpl — Branch if plus

Instruction

```
bpl (label, %pc)
bpl (offset, %pc)
```

Description

Branch if plus

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( N == 0 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

Examples

```
bpl (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0																1	0	1	0	1	0	0	1	0	0	1	0	1	1	1	1

bra.i, bra.i.2, bra.i.4 — Branch

Instruction

```
bra.i   (label, %pc)
bra.i.2 (label, %pc)
bra.i.4 (label, %pc)
bra.i   (offset, %pc)
bra.i.2 (offset, %pc)
bra.i.4 (offset, %pc)

bra.i   (label, 0)
bra.i.4 (label, 0)
bra.i   (offset, 0)
bra.i.4 (offset, 0)

bra.i (0, Ra)
```

Description

Unconditional branch to the register specified.

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
PC = offset[15:0]s + PC

PC = offset[15:0]u + 0

PC = Ra
```

Usage Notes

The assembler inserts the offset from either the current PC or 0 to the label into the instruction.

This instruction can branch to any address in the 64kB of memory.

As all instructions start on a 16-bit boundary, using align errors will cause the `AlignError` exception.

In the PC-relative form, a 16-bit encoding is available for small offsets. However, if the `bra.i.4` mnemonic is used, then the 32-bit encoding will be used regardless. This is intended for use in jump tables.

Examples

```
bra.i   (label, %pc)
bra.i.2 (label, %pc)
bra.i.4 (label, %pc)

bra.i   (label, 0)
bra.i.4 (label, 0)

bra.i (0, %r6)
```

32-bit Encoding – 1

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
#offset[15:1]u, 0																0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1

This encodes the zero-relative form.

32-bit Encoding – 2

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
#offset[15:1]s, 0																0	1	0	0	0	0	1	0	0	0	1	0	1	1	1	1

This encodes the PC-relative form.

16-bit Encoding – 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
#offset[9:7]s			1	1	0	#offset[6:1]s						0	1	0	0

This encodes the PC-relative form.
The following rules apply:

- This encoding can represent offsets in the range -512, -510, …, +508, +510.

16-bit Encoding – 2

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

This encodes the Ra-relative form.

bra.m Branch, via memory, displacement

Instruction

```
bra.m @(label, 0)
bra.m @(offset, 0)
bra.m @(0, Ra)
```

Description

Unconditional branch to via memory with displacement addressing

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
newPC = *(offset[15:1]u + 0)
if newPC[0] == 1:
    throw AlignError
else:
    PC = newPC

if Ra[0] == 1:
    throw AlignError
else:
    newPC = *(0 + Ra)
    if newPC[0] == 1:
        throw AlignError
    else:
        PC = newPC
```

Usage Notes

The table entry address is specified with displacement addressing. The processor reads the 16-bit function pointer at the address and then branches to that function pointer.

If the table entry address is odd, or the address in the table is odd, an `AlignError` exception is thrown.

If the instruction accesses memory address zero, the `NullPointer` exception is thrown. If the instruction violates User mode access rules implemented in the MMU, the `MMUUserDataError` exception is thrown. In addition, the MMU can trigger the `MMUDataError` exception in any mode.

Examples

```
bra.m @(label, 0)

bra.m @(0, %r2)
```

32-bit Encoding

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
#offset[15:1]															0	0	0	0	0	1	1	0	1	0	0	1	0	1	1	1	1

This encodes the zero-relative form.

16-bit Encoding

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

This encodes the register-relative form.

brk — Break

Instruction `brk`

Description Throw a `Break` exception or halt the processor

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if (FLAGS[M] == User && FLAGS[B] == 1) then
       throw Break exception
else if debug mode enabled then
       halt
else
       nop
endif
```

Usage Notes

The `brk` instruction is used for debugging.

If the processor is in User mode and the B bit of the FLAGS register (P[0]) is set, the `Break` exception is thrown. This allows an on-chip debugger or operating system to deal with the break.

Otherwise, if the processor is in debug mode the processor is stopped.

When debugging with the XAP4 simulator, the processor is always in debug mode. On the XAP4 emulator and real XAP4s, debug mode is controlled with the SIF interface. The processor can be restarted using the SIF interface or a reset.

The B flag can be written with the `movr2s`, `mov.1.i` and `mov.4.i` instructions. This is not permitted in User mode.

Examples `brk`

16-bit Encoding

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

bsr.i — Branch to subroutine

Instruction

```
bsr.i (label, %pc)
bsr.i (offset, %pc)

bsr.i (label, 0)
bsr.i (offset, 0)

bsr.i (0, Ra)
```

Description

Unconditional branch to subroutine.

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
Push next instruction address to stack
PC = offset[15:0]s + PC

Push next instruction address to stack
PC = offset[15:0]u + 0

Push next instruction address to stack
PC  = Ra
```

Usage Notes

The `bsr.i` instruction changes the program counter and saves the address of the instruction after the branch onto the stack.

The 16-bit address is specified with displacement addressing, and can address the whole of memory.

Examples

```
bsr.i (label, %pc)

bsr.i (label, 0)

bsr.i (0, %r1)
```

32-bit Encoding - 1

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
#offset[15:1]s, 0				0 0 1 0	0 0 1 0	0 0 1 0	1 1 1 1

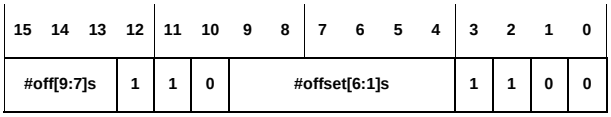
This encodes the zero-relative form.

32-bit Encoding – 2

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
#offset[15:1]u, 0				0 1 1 0	0 0 1 0	0 0 1 0	1 1 1 1

This encodes the PC-relative form.

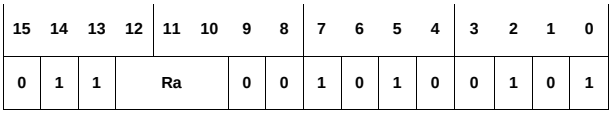
16-bit Encoding – 1



This encodes the PC-relative form.
The following rules apply:

- This encoding can represent offsets in the range -512, -510, …, +508, +510.

16-bit Encoding – 2



This encodes the Ra-relative form.

bsr.m Branch to subroutine, via memory, displacement

Instruction

```
bsr.m @(label, 0)
bsr.m @(offset, 0)

bsr.m @(0, Ra)
```

Description

Unconditional branch to subroutine via memory with displacement addressing

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
newPC = *(offset[15:1]u + 0)
if newPC[0] == 1:
    throw AlignError
else:
    Push next instruction to stack
    PC = newPC

Push next instruction to stack
if Ra[0] == 1:
    throw AlignError
else:
    newPC = *(0 + Ra)
    if newPC[0] == 1:
        throw AlignError
    else:
        Push next instruction to stack
        PC = newPC
```

Usage Notes

The `bsr.m` instruction changes the program counter and saves the address of the instruction after the branch onto the stack.

The table entry address is supplied to `bsr.m` as an argument. The processor reads the 16-bit function pointer at the address argument and then branches to that function pointer.

If the table entry address is odd, or the address in the table is odd,an `AlignError` exception is thrown.

If the instruction accesses memory address zero, the `NullPointer` exception is thrown. If the instruction violates User mode access rules implemented in the MMU, the `MMUUserDataError` exception is thrown. In addition, the MMU can trigger the `MMUDataError` exception in any mode.

Examples

```
bsr.m @(label, 0)
```

```
bsr.m @(0, %r2)
```

32-bit Encoding

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
#offset[15:0]u																0	0	0	1	0	1	1	0	1	0	0	1	0	1	1	1	1

This encodes the zero-relative form.

16-bit Encoding

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

This encodes the register-relative form.

bvc — Branch if overflow clear

Instruction

```
bvc (label, %pc)
bvc (offset, %pc)
```

Description

Branch if overflow clear

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( V == 0 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

Examples

```
bvc (label, %pc)
```

32-bit Encoding

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
#offset[15:1]s, 0																0	1	1	0	1	0	0	1	0	0	1	0	1	1	1	1

bvs — Branch if overflow set

Instruction

```
bvs (label, %pc)
bvs (offset, %pc)
```

Description

Branch if overflow set

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( V == 1 )
   PC = offset[15:0]s + PC
else
   PC = next instruction
```

Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -32kB to +32kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

Examples

```
bvs (label, %pc)
```

32-bit Encoding

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
#offset[15:1,]s, 0																0	1	0	0	1	0	0	1	0	0	1	0	1	1	1	1

clu — CLU Instruction Type 1

Instruction

`clu #imm`

Description

User customisable instruction with the following arguments:
- #immediate

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

User defined

Usage Notes

Sets clu_type[3:0] = 1.
The immediate is passed to the CLU.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

`clu #129`

32-bit Encoding

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
0	1	1	#immediate[12:0]u													1	1	1	0	1	1	0	0	1	0	1	1	0	1	1	1

clu.d — CLU Instruction Type 2

Instruction

```
clu.d #imm, Rd
```

Description

User customisable instruction with the following arguments:
- #immediate
- 16-bit destination register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

User defined

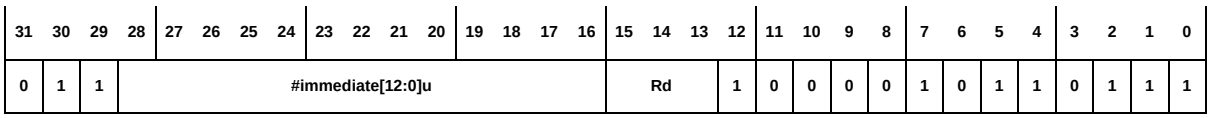
Usage Notes

Sets clu_type[3:0] = 2.
The immediate is passed to the CLU.
The result is put in Rd.
The Z and N flags are updated based on the value in Rd.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.d #0x1234, %r4
```

32-bit Encoding



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
0	1	1	#immediate[12:0]u													Rd				1	0	0	0	0	1	0	1	1	0	1	1	1

clu.dd CLU Instruction Type 3

Instruction

```
clu.dd #imm, Rd
```

Description

User customisable instruction with the following arguments:
- #immediate
- 32-bit destination register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

User defined

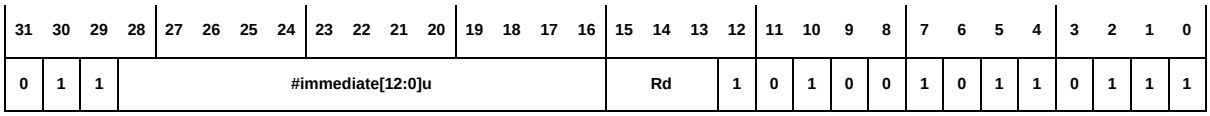
Usage Notes

Sets clu_type[3:0] = 3.
The immediate is passed to the CLU.
The result is put in {R(d+1), Rd}.
The Z and N flags are updated based on the value in {R(d+1), Rd}.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.dd #0xFE2, %r1
```

32-bit Encoding



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
0	1	1	#immediate[12:0]u													Rd			1	0	1	0	0	1	0	1	1	0	1	1	1

clu.dds — CLU Instruction Type 10

Instruction

```
clu.dds #imm, Rd, Rs
```

Description

User customisable instruction with the following arguments:
- #immediate
- 32-bit destination register
- 16-bit source register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

User defined

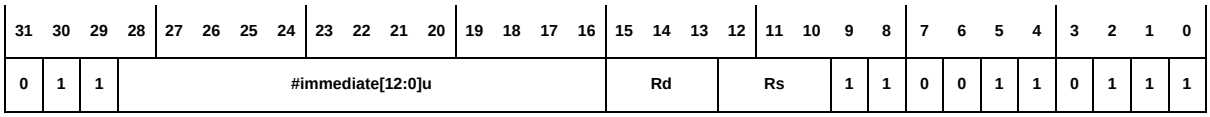
Usage Notes

Sets clu_type[3:0] = 10.
The immediate is passed to the CLU.
Rs is passed to the CLU.
The result is put in {R(d+1), Rd}.
The Z and N flags are updated based on the value in {R(d+1), Rd}.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.dds #60, %r0, %r7
```

32-bit Encoding



clu.ddss — CLU Instruction Type 11

Instruction

```
clu.ddss #imm, Rd, Rs
```

Description

User customisable instruction with the following arguments:
- #immediate
- 32-bit destination register
- 32-bit source register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

User defined

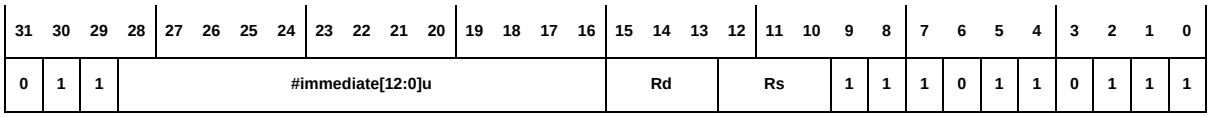
Usage Notes

Sets clu_type[3:0] = 11.
The immediate is passed to the CLU.
{R(s+1), Rs} is passed to the CLU.
The result is put in {R(d+1), Rd}.
The Z and N flags are updated based on the value in {R(d+1), Rd}.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.ddss #0x72E, %r0, %r2
```

32-bit Encoding



clu.ddsst — CLU Instruction Type 15

Instruction

```
clu.ddsst #imm, Rd, Rs, Rt
```

Description

User customisable instruction with the following arguments:
- #immediate
- 32-bit destination register
- 32-bit source register
- 16-bit secondary source register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

User defined

Usage Notes

Sets clu_type[3:0] = 15.
The immediate is passed to the CLU.
{R(s+1), Rs} is passed to the CLU.
Rt is passed to the CLU.
The result is put in {R(d+1), Rd}.
The Z and N flags are updated based on the value in {R(d+1), Rd}.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.ddsst #0xF, %r0, %r3, %r6
```

32-bit Encoding

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
1	1	1	#immediate[12:0]u													Rd				Rs				Rt				0	1	1	0	1	1	1

clu.ddst — CLU Instruction Type 14

Instruction

```
clu.ddst #imm, Rd, Rs, Rt
```

Description

User customisable instruction with the following arguments:
- #immediate
- 32-bit destination register
- 16-bit source register
- 16-bit secondary source register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

User defined

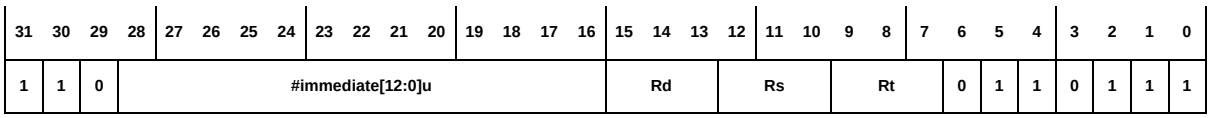
Usage Notes

Sets clu_type[3:0] = 14.
The immediate is passed to the CLU.
Rs is passed to the CLU.
Rt is passed to the CLU.
The result is put in {R(d+1), Rd}.
The Z and N flags are updated based on the value in {R(d+1), Rd}.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.ddst #23, %r2, %r0, %r6
```

32-bit Encoding



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
1	1	0	#immediate[12:0]u													Rd				Rs				Rt				0	1	1	0	1	1	1

clu.ds — CLU Instruction Type 8

Instruction

```
clu.ds #imm, Rd, Rs
```

Description

User customisable instruction with the following arguments:
- #immediate
- 16-bit destination register
- 16-bit source register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

User defined

Usage Notes

Sets clu_type[3:0] = 8.
The immediate is passed to the CLU.
Rs is passed to the CLU.
The result is put in Rd.
The Z and N flags are updated based on the value in Rd.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.ds #918, %r2, %r7
```

32-bit Encoding

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
0	1	1	#immediate[12:0]u													Rd				Rs				1	0	0	0	1	1	0	1	1	1

clu.dss — CLU Instruction Type 9

Instruction

```
clu.dss #imm, Rd, Rs
```

Description

User customisable instruction with the following arguments:
- #immediate
- 16-bit destination register
- 32-bit source register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

User defined

Usage Notes

Sets clu_type[3:0] = 9.
The immediate is passed to the CLU.
{R(s+1), Rs} is passed to the CLU.
The result is put in Rd.
The Z and N flags are updated based on the value in Rd.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.dss #3, %r0, %r1
```

32-bit Encoding

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
0	1	1	#immediate[12:0]u													Rd				Rs				1	0	1	0	1	1	0	1	1	1

clu.dsst — CLU Instruction Type 13

Instruction

```
clu.dsst #imm, Rd, Rs, Rt
```

Description

User customisable instruction with the following arguments:
- #immediate
- 16-bit destination register
- 32-bit source register
- 16-bit secondary source register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

User defined

Usage Notes

Sets clu_type[3:0] = 13.
The immediate is passed to the CLU.
{R(s+1), Rs} is passed to the CLU.
Rt is passed to the CLU.
The result is put in Rd.
The Z and N flags are updated based on the value in Rd.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.dsst #0x89, %r7, %r4, %r2
```

32-bit Encoding

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
1	0	1	#immediate[12:0]u													Rd				Rs				Rt				0	1	1	0	1	1	1

clu.dst — CLU Instruction Type 12

Instruction

```
clu.dst #imm Rd, Rs, Rt
```

Description

User customisable instruction with the following arguments:
- #immediate
- 16-bit destination register
- 16-bit source register
- 16-bit secondary source register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

User defined

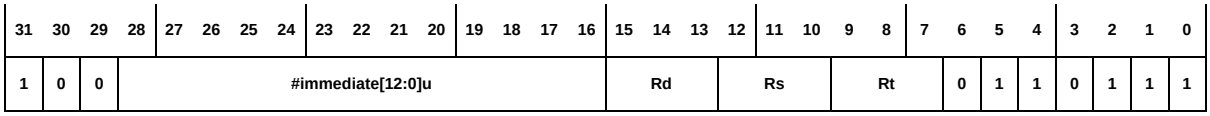
Usage Notes

Sets clu_type[3:0] = 12.
The immediate is passed to the CLU.
Rs is passed to the CLU.
Rt is passed to the CLU.
The result is put in Rd.
The Z and N flags are updated based on the value in Rd.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.dst #93, %r1, %r2, %r3
```

32-bit Encoding



clu.s — CLU Instruction Type 4

Instruction

```
clu.s #imm, Rs
```

Description

User customisable instruction with the following arguments:
- #immediate
- 16-bit source register

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

User defined

Usage Notes

Sets clu_type[3:0] = 4.
The immediate is passed to the CLU.
Rs is passed to the CLU.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.s #0, %r5
```

32-bit Encoding

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
0	1	1	#immediate[12:0]u													Rs			1	1	0	0	0	1	0	1	1	0	1	1	1

clu.ss — CLU Instruction Type 5

Instruction

```
clu.ss #imm, Rs
```

Description

User customisable instruction with the following arguments:
- #immediate
- 32-bit source register

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

User defined

Usage Notes

Sets clu_type[3:0] = 5.
The immediate is passed to the CLU.
{R(s+1), Rs} is passed to the CLU.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.ss #7, %r2
```

32-bit Encoding

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
0	1	1	#immediate[12:0]u													Rs			1	1	1	0	0	1	0	1	1	0	1	1	1

clu.sst — CLU Instruction Type 7

Instruction

```
clu.sst #imm, Rs, Rt
```

Description

User customisable instruction with the following arguments:
- #immediate
- 32-bit source register
- 16-bit secondary source register

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

User defined

Usage Notes

Sets clu_type[3:0] = 7.
The immediate is passed to the CLU.
{R(s+1), Rs} is passed to the CLU.
Rt is passed to the CLU.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.sst #0xFA, %r1, %r6
```

32-bit Encoding

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
0	1	1	#immediate[12:0]u													Rt				Rs				0	1	1	0	1	1	0	1	1	1

clu.st — CLU Instruction Type 6

Instruction

```
clu.st #imm Rs, Rt
```

Description

User customisable instruction with the following arguments:
- #immediate
- 16-bit source register
- 16-bit secondary source register

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

User defined

Usage Notes

Sets clu_type[3:0] = 6.
The immediate is passed to the CLU.
Rs is passed to the CLU.
Rt is passed to the CLU.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details"

Examples

```
clu.st #374, %r2, %r1
```

32-bit Encoding

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
0	1	1	#immediate[12:0]u													Rt				Rs				0	1	0	0	1	1	0	1	1	1

cmp.8.i Compare, 8-bit, immediate

Instruction `cmp.8.i Rs, #immediate`

Description An 8-bit compare of the register with an immediate

Flags

- **Z** Set if `Rs[7:0] == #immediate[7:0]`; cleared otherwise
- **N** Set if the result of the subtraction is negative; cleared if the result is positive
- **C** Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise
- **V** Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise

Operation `Rs[7:0] - #immediate[7:0]`

Usage Notes This is an 8-bit compare of an 8-bit immediate with the lower 8 bits of a register. The high 8 bits of the immediate and register are ignored.

Examples `cmp.8.i %r1, #0x12`

32-bit Encoding

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
0	0	0	0	0	0	0	0	#immediate[7:0]								0	0	0	Rs			0	0	0	0	1	0	1	1	1	1

16-bit Encoding - 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs			1	0	0	#immediate[6:0]u						1	0	0	

The following rule applies:

- This encoding can represent immediates in the range 0, 1, , 126, 127.

16-bit Encoding - 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs			1	1	1	0	1	1	#imm[3:0]u			0	1	1	

The following rule applies:

- The immediate value is (#immediate[3:0]u – 16) and this gives a range of -1, -2, …, -15, -16.

cmp.8.r — Compare, 8-bit, register

Instruction

`cmp.8.r Rs, Rt`

Description

Compare the low 8 bits in two registers

Flags

Z	Set if `Rs[7:0] == Rt[7:0]`; cleared otherwise
N	Set if the result of the subtraction is negative; cleared if the result is positive
C	Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise

Operation

`Rs[7:0] - Rt[7:0]`

Usage Notes

This is an 8-bit compare of the lower 8 bits from two registers. The high 8 bits of the two registers are ignored.

Examples

`cmp.8.r %r1, %r2`

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	Rs					Rt				1	0	0	0	1	0	1

cmp.8c.i Compare, 8-bit with carry, immediate

Instruction `cmp.8c.i Rs, #immediate`

Description An 8-bit compare of the register with an immediate and the carry flag

Flags

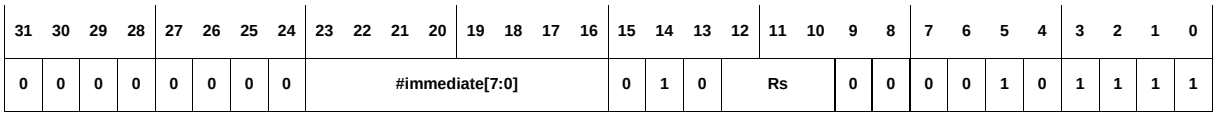
- **Z** Set if `Rs[7:0] = #immediate[7:0]` and the Z flag was already set; cleared otherwise
- **N** Set if the result of the subtraction is negative; cleared if the result is positive
- **C** Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise
- **V** Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise

Operation `Rs[7:0] - #immediate[7:0] - C`

Usage Notes This is an 8-bit compare of an 8-bit immediate with the lower 8 bits of a register and the carry flag. The high 8 bits of the immediate and register are ignored.

Examples `cmp.8c.i %r1, #0x12`

32-bit Encoding



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
0	0	0	0	0	0	0	0	#immediate[7:0]								0	1	0	Rs			0	0	0	0	1	0	1	1	1	1

cmp.8c.r Compare, 8-bit with carry, register

Instruction `cmp.8c.r Rs, Rt`

Description An 8-bit compare of two registers and the carry bit

Flags

Z	Set if `Rs[7:0] == Rt[7:0]` and the Z flag was already set; cleared otherwise
N	Set if the result of the subtraction is negative; cleared if the result is positive
C	Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise

Operation `Rs[7:0] - Rt[7:0] - C`

Usage Notes This is an 8-bit compare of the lower 8 bits from two registers and the carry bit. The high 8 bits of the two registers are ignored.

Examples `cmp.8c.r %r1, %r2`

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	Rs				Rt		1	0	0	1	1	0	1

cmp.8x.i Compare, 8-bit, exchange, immediate

Instruction `cmp.8x.i Rs, #immediate`

Description An 8-bit compare of the register with an immediate and with the operand order reversed

Flags

Z	Set if `Rs[7:0] == #immediate[7:0]`; cleared otherwise
N	Set if the result of the subtraction is negative; cleared if the result is positive
C	Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise

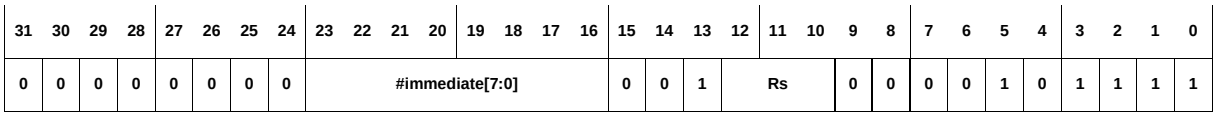
Operation `#immediate[7:0] - Rs[7:0]`

Usage Notes This is an 8-bit compare of the lower 8 bits of a register with an 8-bit immediate. The high 8 bits of the register and immediate are ignored.

This is the same as `cmp.8.i` but with the operand order reversed.

Examples `cmp.8x.i %r1, #0x12`

32-bit Encoding



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
0	0	0	0	0	0	0	0	#immediate[7:0]								0	0	1	Rs			0	0	0	0	1	0	1	1	1	1

cmp.8xc.i Compare, 8-bit carry, exchange, immediate

Instruction

`cmp.8xc.i Rs, #immediate`

Description

An 8-bit compare of a register with an immediate and the carry bit, and with the operand order reversed

Flags

Z Set if `Rs[7:0] == #immediate[7:0]` and the Z flag was already set; cleared otherwise

N Set if the result of the subtraction is negative; cleared if the result is positive

C Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise

V Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise

Operation

`#immediate[7:0] - Rs[7:0] - C`

Usage Notes

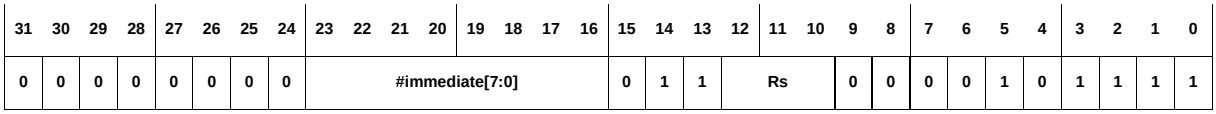
This is an 8-bit compare of an 8-bit immediate with the lower 8 bits of a register and the carry flag. The high 8 bits of the immediate and register are ignored.

This is the same as `cmp.8c.i` but with the operand order reversed.

Examples

`cmp.8xc.i %r1, #0x12`

32-bit Encoding



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
0	0	0	0	0	0	0	0	#immediate[7:0]								0	1	1	Rs			0	0	0	0	1	0	1	1	1	1

cmp.c.i Compare, with carry, immediate

Instruction `cmp.c.i Rs, #immediate`

Description A 16-bit compare of the register with an immediate and the carry flag

Flags

- **Z** Set if `Rs == #immediate[15:0]` and the Z flag was already set; cleared otherwise
- **N** Set if the result of the subtraction is negative; cleared if the result is positive
- **C** Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise
- **V** Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise

Operation `Rs - #immediate[15:0] - C`

Usage Notes This is a 16-bit compare of an immediate with a register and the carry bit.

The Z flag behaviour is useful for 32-bit arithmetic.

A 32-bit compare of register pair %r1 with an immediate (for example, 0x12345678) can be performed in two instructions:

```
cmp.i   %r1, 0x5678
cmp.c.i %r2, 0x1234
```

Examples

```
cmp.c.i %r1, #0x1234
cmp.c.i %r1, #0xFEDC
```

32-bit Encoding

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
#immediate[15:0]s				1	1	0	Rs		0	0	0	0	1	0	1	1	1	1

cmp.c.r — Compare, with carry, register

Instruction

`cmp.c.r Rs, Rt`

Description

A 16-bit compare of two registers and the carry bit

Flags

Z	Set if `Rs == Rt` and the Z flag was already set; cleared otherwise
N	Set if the result of the subtraction is negative; cleared if the result is positive
C	Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise

Operation

`Rs - Rt - C`

Usage Notes

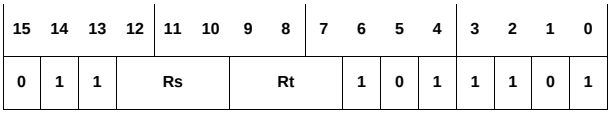
This is a 16-bit compare of two registers and the carry bit.

The Z flag behaviour is useful for 32-bit arithmetic.

Examples

`cmp.c.r %r1, %r2`

16-bit Encoding



cmp.i — Compare, immediate

Instruction `cmp.i Rs, #immediate`

Description A 16-bit compare of the register with an immediate

Flags

- **Z** Set if `Rs == #immediate[15:0]s`; cleared otherwise
- **N** Set if the result of the subtraction is negative; cleared if the result is positive
- **C** Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise
- **V** Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise

Operation `Rs - #immediate[15:0]s`

Usage Notes This is a 16-bit compare of a register with an immediate.

A number of commonly used combinations of registers and immediates are available in two 16-bit encodings of this instruction.

Examples

```
cmp.i %r1, #0x1234
cmp.i %r1, #0xFEDC
```

32-bit Encoding

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
#immediate[15:0]s				1	0	0	Rs		0	0	1	0	1	1	1	1

16-bit Encoding - 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs			1	0	1	#immediate[6:0]u						1	0	0	

The following rule applies:

- This encoding can represent immediates in the range 0, 1, …, 126, 127.

16-bit Encoding - 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs			1	1	1	1	1	1	#immediate[3:0]u			0	1	1	

The following rule applies:

- The immediate value is (#immediate[3:0]u – 16) and this gives a range of -1, -2, …, -15, -16.

cmp.r — Compare, register

Instruction `cmp.r Rs, Rt`

Description Compares two registers

Flags

Z	Set if `Rs == Rt`; cleared otherwise
N	Set if the result of the subtraction is negative; cleared if the result is positive
C	Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise

Operation `Rs - Rt`

Usage Notes This is a 16-bit compare of two registers.

Examples

```
cmp.r %r1, %r2
```

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	Rs					Rt				1	0	1	
0	1	1										0	1	0	1

cmp.x.i Compare, exchange, immediate

Instruction `cmp.x.i Rs, #immediate`

Description A 16-bit compare of the register with an immediate and with the operand order reversed

Flags

Z	Set if `Rs == #immediate[15:0]s`; cleared otherwise
N	Set if the result of the subtraction is negative; cleared if the result is positive
C	Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise

Operation `#immediate[15:0]s - Rs`

Usage Notes This is a 16-bit compare of an immediate with a register.

This is the same as `cmp.i` but with the operand order reversed.

Examples

```
cmp.x.i %r1, #0x1234
cmp.x.i %r1, #0xFEDC
```

32-bit Encoding

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
#immediate[15:0]s				1	0	1	Rs		0	0	0	0	1	0	1	1	1	1

cmp.xc.i Compare, with carry, exchange, immediate

Instruction

`cmp.xc.i Rs, #immediate`

Description

A 16-bit compare of a register with an immediate and the carry bit, and with the operand order reversed

Flags

Z Set if `Rs == #immediate[15:0]s` and the Z flag was already set; cleared otherwise

N Set if the result of the subtraction is negative; cleared if the result is positive

C Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise

V Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise

Operation

`#immediate[15:0]s - Rs - C`

Usage Notes

This is a 16-bit compare of an immediate with a register and the carry bit.

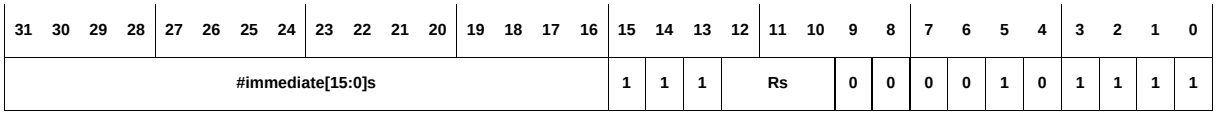
This is the same as `cmp.c.i` but with the operand order reversed.

The Z flag behaviour is useful for 32-bit arithmetic.

Examples

```
cmp.xc.i %r1, #0x1234
cmp.xc.i %r1, #0xFEDC
```

32-bit Encoding



div.32s.i Divide, 32-bit signed, immediate

Instruction `div.32s.i Rd, Rs, #immediate`

Description Signed 32-bit / 16-bit integer divide to give a 16-bit quotient

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Set if an overflow occurs or if an attempt is made to divide by zero; cleared otherwise

Operation `Rd = {R(s+1), Rs} / #immediate[15:0]s`

Usage Notes Rs is a register pair.

If the quotient cannot be represented in 16 bits, the overflow flag is set and the quotient is forced to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `div.32s.i %r4, %r1, #20`

32-bit Encoding

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
#immediate[15:0]s				Rd	Rs 0 1	0 0 0 1	1 1 1 1

div.32s.r — Divide, 32-bit signed, register

Instruction `div.32s.r Rd, Rs, Rt`

Description Signed 32-bit / 16-bit integer divide to give a 16-bit quotient

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Set if an overflow occurs or if an attempt is made to divide by zero; cleared otherwise

Operation `Rd = {R(s+1), Rs} / Rt`

Usage Notes Rs is a register pair.

If the quotient cannot be represented in 16 bits, the overflow flag is set and the quotient is forced to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `div.32s.r %r4, %r1, %r3`

32-bit Encoding

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
Rd	Rs	Rt	0 0 1	0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1

div.32u.i Divide, 32-bit unsigned, immediate

Instruction `div.32u.i Rd, Rs, #immediate`

Description Signed 32-bit / 16-bit integer divide to give a 16-bit quotient

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if bit 31 of the result is 1; cleared otherwise
C	Set if an overflow occurs or if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation `Rd = {R(s+1), Rs} / #immediate[15:0]u`

Usage Notes Rs is a register pair.

If the quotient cannot be represented in 16 bits, the carry flag is set and the quotient is forced to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `div.32u.i %r4, %r1, #10`

32-bit Encoding

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
#immediate[15:0]u				Rd	Rs 0 1	1 0 0 1	1 1 1 1

div.32u.r Divide, 32-bit unsigned, register

Instruction `div.32u.r Rd, Rs, Rt`

Description Signed 32-bit / 16-bit integer divide to give a 16-bit quotient

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if bit 31 of the result is 1; cleared otherwise
C	Set if an overflow occurs or if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation `Rd = {R(s+1), Rs} / Rt`

Usage Notes Rs is a register pair.

If the quotient cannot be represented in 16 bits, the carry flag is set and the quotient is forced to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `div.32u.r %r4, %r1, %r3`

32-bit Encoding

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
Rd	Rs	Rt	0 0 1	1 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1

div.s.i — Divide, signed, immediate

Instruction `div.s.i Rd, Rs, #immediate`

Description Signed 16-bit /16-bit integer divide to give a 16-bit quotient

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Set if an overflow occurs or if an attempt is made to divide by zero; cleared otherwise

Operation `Rd = Rs / #immediate[15:0]s`

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

The division -32768 / -1 gives a result of 0 and sets the overflow flag.

Examples `div.s.i %r1, %r2, #10`

32-bit Encoding

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
#immediate[15:0]s				Rd	Rs	0	0	0	0	0	1	1	1	1	1

div.s.r — Divide, signed, register

Instruction `div.s.r Rd, Rs, Rt`

Description Signed 16-bit / 16-bit integer divide to give a 16-bit quotient

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Set if an overflow occurs or if an attempt is made to divide by zero; cleared otherwise

Operation `Rd = Rs / Rt`

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

The division -32768 / -1 gives a result of 0 and sets the overflow flag.

Examples

```
div.s.r %r1, %r2, %r3
```

32-bit Encoding

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
Rd				Rs				Rt				0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

div.u.i — Divide, unsigned, immediate

Instruction `div.u.i Rd, Rs, #immediate`

Description Unsigned 16-bit / 16-bit integer divide to give a 16-bit quotient

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if bit 15 of the result is 1; cleared otherwise
C	Set if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation `Rd = Rs / #immediate[15:0]u`

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `div.u.i %r1, %r2, #0x10`

32-bit Encoding

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
#immediate[15:0]u				Rd	Rs	0	0	1	0	0	1	1	1	1	1

div.u.r — Divide, unsigned, register

Instruction `div.u.r Rd, Rs, Rt`

Description Unsigned 16-bit /16-bit integer divide to give a 16-bit quotient

Flags

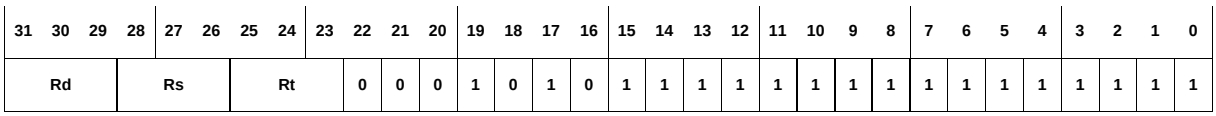
Z	Set if the result is zero; cleared otherwise
N	Set if bit 15 of the result is 1; cleared otherwise
C	Set if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation `Rd = Rs / Rt`

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `div.u.r %r1, %r2, %r3`

32-bit Encoding



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
Rd				Rs				Rt				0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

divrem.32s.i Divide&remainder, 32-bit signed, immediate

Instruction

```
divrem32.s.i Rd, Rs, #immediate
```

Description

Signed 32-bit / 16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Set if an attempt is made to divide by zero; cleared otherwise

Operation

```
Rd     = {R(s+1), Rs} / #immediate[15:0]s
R(d+1) = {R(s+1), Rs} % #immediate[15:0]s
```

Usage Notes

Rd and Rs are register pairs.

If the quotient cannot be represented in 16 bits, the overflow flag is set and the quotient and remainder are forced to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples

```
divrem32.s.i %r4, %r1, #10
```

32-bit Encoding

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
#immediate[15:0]s				Rd	Rs	0	1	0	0	1	0	0	1	1	1

divrem.32s.r Divide&remainder, 32-bit signed, register

Instruction `divrem.32s.r Rd, Rs, Rt`

Description Signed 32-bit /16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Set if an attempt is made to divide by zero; cleared otherwise

Operation

```
Rd     = {R(s+1), Rs} / Rt
R(d+1) = {R(s+1), Rs} % Rt
```

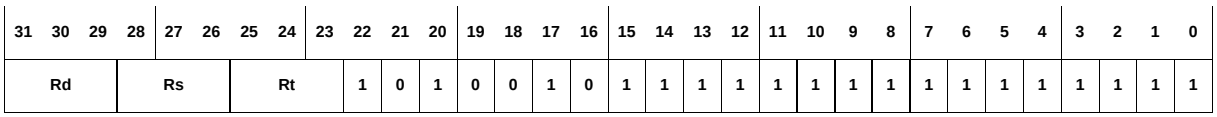
Usage Notes Rd and Rs are register pairs.

If the quotient cannot be represented in 16 bits, the overflow flag is set and the quotient and remainder are forced to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `divrem.32s.r %r4, %r1, %r3`

32-bit Encoding



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
Rd				Rs				Rt				1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1

divrem.32u.i Divide&remainder, 32-bit unsigned, immediate

Instruction

```
divrem.32u.i Rd, Rs, #immediate
```

Description

Unsigned 32-bit /16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

Flags

Z	Unchanged
N	Unchanged
C	Set if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation

```
Rd     = {R(s+1), Rs} / #immediate[15:0]u
R(d+1) = {R(s+1), Rs} % #immediate[15:0]u
```

Usage Notes

Rd and Rs are register pairs.

If the quotient cannot be represented in 16 bits, the carry flag is set and the quotient and remainder are forced to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples

```
divrem.32u.i %r4, %r1, #immediate
```

32-bit Encoding

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
#immediate[15:0]u				Rd	Rs	0	1	1	0	1	0	0	1	1	1

divrem.32u.r Divide&remainder, 32-bit unsigned, register

Instruction `divrem.32u.r Rd, Rs, Rt`

Description Unsigned 32-bit /16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

Flags

Z	Unchanged
N	Unchanged
C	Set if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation

```
Rd     = {R(s+1), Rs} / Rt
R(d+1) = {R(s+1), Rs} % Rt
```

Usage Notes Rd and Rs are register pairs.

If the quotient cannot be represented in 16 bits, the carry flag is set and the quotient and remainder are forced to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `divrem.32u.r %r4, %r1, %r3`

32-bit Encoding

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
Rd	Rs	Rt	1 0 1	1 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1

divrem.s.i Divide&remainder, signed, immediate

Instruction `divrem.s.i Rd, Rs, #immediate`

Description Unsigned 16-bit / 16-bit integer divide to give a 16-bit integer quotient and a 16-bit remainder

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Set if an overflow occurs or an attempt is made to divide by zero; cleared otherwise

Operation

```
Rd     = Rs / #immediate[15:0]s
R(d+1) = Rs % #immediate[15:0]s
```

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

The division -32768 / -1 gives results of 0 and sets the overflow flag.

Examples `divrem.s.i %r1, %r2, #10`

32-bit Encoding

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
#immediate[15:0]s				Rd	Rs	0	0	0	0	1	0	0	1	1	1

divrem.s.r Divide&remainder, signed, register

Instruction `divrem.s.r Rd, Rs, Rt`

Description Unsigned 16-bit / 16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Set if an overflow occurs or an attempt is made to divide by zero; cleared otherwise

Operation

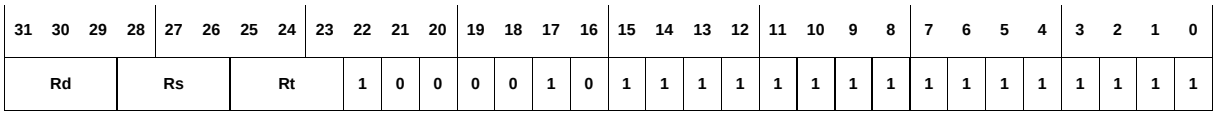
```
Rd     = Rs / Rt
R(d+1) = Rs % Rt
```

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

The division -32768 / -1 gives results of 0 and sets the overflow flag.

Examples `divrem.s.r %r1, %r2, %r3`

32-bit Encoding



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
Rd				Rs				Rt				1	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1

divrem.u.i Divide&remainder, unsigned, immediate

Instruction `divrem.u.i Rd, Rs, #immediate`

Description Unsigned 16-bit / 16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

Flags

Z	Unchanged
N	Unchanged
C	Set if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

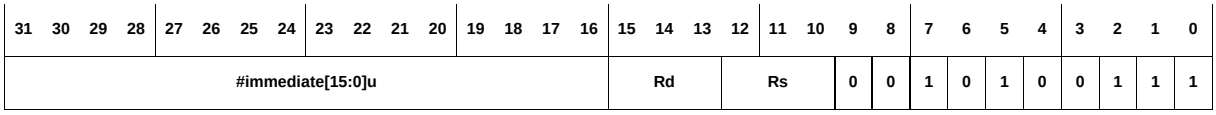
Operation

```
Rd     = Rs / #immediate[15:0]u
R(d+1) = Rs % #immediate[15:0]u
```

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `divrem.u.i %r1, %r2, #10`

32-bit Encoding



divrem.u.r Divide&remainder, unsigned, register

Instruction `divrem.u.r Rd, Rs, Rt`

Description Unsigned 16-bit / 16-bit integer divide to give a 16-bit quotient and a 16-bit remainder

Flags

Z	Unchanged
N	Unchanged
C	Set if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation

```
Rd     = Rs / Rt
R(d+1) = Rs % Rt
```

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `divrem.u.r %r1, %r2, %r3`

32-bit Encoding

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
Rd				Rs				Rt				1	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

fimode — Flags and info mode

Instruction `fimode Rd`

Description Read a number representing the processor mode/state

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

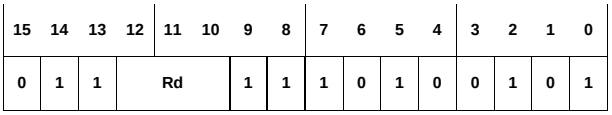
Operation `Rd = flags and info mode`

Usage Notes The value written to Rd is:

- 3 = User mode
- 2 = Trusted mode
- 0 = Supervisor mode
- 1 = Interrupt mode
- 4 = Recovery state
- 5 = NMI state

Examples `fimode %r1`

16-bit Encoding



halt — Halt

Instruction `halt`

Description Stop the processor

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if (FLAGS[M] != UserMode) then
   halt processor
else
   throw PrivInstruction_S exception
endif
```

Usage Notes

A `PrivInstruction_S` exception is thrown if this instruction is executed in User mode.

The processor is restarted using the SIF interface or a reset.

Examples `halt`

16-bit Encoding

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

ld.1.i Load, single-bit, displacement

Instruction

```
ld.1.i %flags[c], @(label[bit], 0)
ld.1.i %flags[c], @(offset[bit], 0)
```

Description

Load specified bit from memory to C flag.

Flags

Z	-
N	-
C	Set to the result of the operation.
V	-

Operation

```
C = ((* (int8*) offset[15:0]u ) >> bit) & 1)
```

Usage Notes

This can only use zero-relative addressing.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
ld.1.i %flags[c], @(label[6], 0)
```

32-bit Encoding

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
#offset[15:0]u				bit[2:0] 0	0 0 1 1	0 0 1 0	0 1 1 1

ld.8z.i Load, 8-bit, zero-extend, displacement

Instruction

```
ld.8z.i Rd, @Address
```

Description

Load zero extended 8-bit value from 16-bit address with displacement addressing

`Address` is one of the following:

```
(offset, Ra)
(label, %pc)
(offset, %pc)
(offset, %sp)
(label, 0)
(offset, 0)
```

Flags

Z	Set if the result is zero; cleared otherwise
N	Cleared to zero
C	Unchanged
V	Unchanged

Operation

```
Rd = (uint16) (* (int8*) Address )
```

Usage Notes

Ra and offset are interpreted as byte addresses.

The memory value is zero-extended to 16 bits when written to Rd.

An 8-bit memory read is performed.

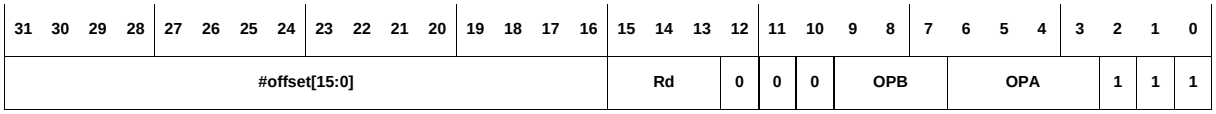
Refer to section 3.8.9, "Error Details", for details of possible exceptions.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
ld.8z.i %r2, @(28, %r7)
ld.8z.i %r1, @(label, %pc)
ld.8z.i %r5, @(23, %pc)
ld.8z.i %r3, @(0, %sp)
ld.8z.i %r6, @(label, 0)
ld.8z.i %r7, @(0x8100, 0)
```

32-bit Encoding - 1

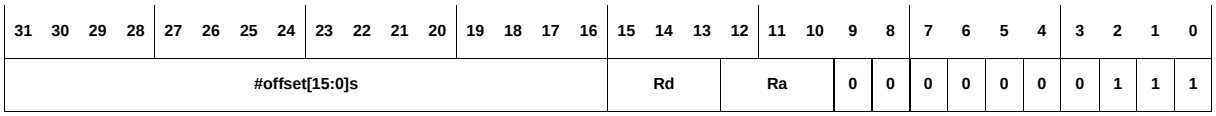


This encodes the form `ld.8z.i Rd, @(offset, BaseAddress)`.

OPB	OPA	BaseAddress	Offset Type
7	1	PC	Signed

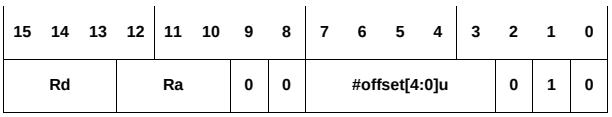
3	1	SP	Unsigned
3	0	0	Unsigned

32-bit Encoding - 2



This encodes the form `ld.8z.i Rd, @(offset, Ra)`.

16-bit Encoding - 1

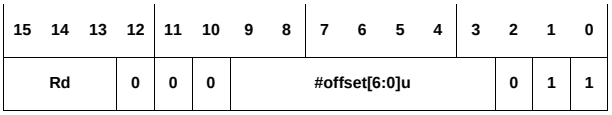


This encodes the form `ld.8z.i Rd, @(offset, Ra)`.

The following rule applies:

- This encoding can represent offsets in the range 0 to 31.

16-bit Encoding - 2



This encodes the form `ld.8z.i Rd, @(offset, %sp)`.

The following rule applies:

- This encoding can represent offsets in the range 0 to 127.

ld.8z.r Load, 8-bit, zero-extend, indexed

Instruction

`ld.8z.r Rd, @Address`

Description

Load zero extended 8-bit value from 16-bit address with indexed addressing

`Address` is one of the following:

`(Rx, Ra)`

`(Rx, %sp)`

Flags

Z	Set if the result is zero; cleared otherwise
N	Cleared to zero
C	Unchanged
V	Unchanged

Operation

```
Rd = (uint16) (* (int8*) ( Rx + Ra ) )
Rd = (uint16) (* (int8*) ( Rx + SP ) )
```

Usage Notes

Ra is interpreted as byte addresses. Rx is interpreted as a byte offset from Ra or SP.

The memory value is zero-extended to 16 bits when written to Rd.

An 8-bit memory read is performed.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
ld.8z.r %r1, @(%r6, %r7)
ld.8z.r %r1, @(%r6, %sp)
```

32-bit Encoding

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
Rd			0	0	0	Rx		1	0	1		1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

This encodes the form `ld.8z.r Rd, @(Rx, %sp)`.

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Ra				Rx			0	0	0	0	1	1	0

This encodes the form `ld.8z.r Rd, @(Rx, Ra)`.

ld.32.i Load, 32-bit, displacement

Instruction

```
ld.32.i Rd, @Address
```

Description

Load 32-bit value from 16-bit address with displacement addressing

`Address` is one of the following:

```
(offset, Ra)
(label, %pc)
(offset, %pc)
(offset, %sp)
(label, 0)
(offset, 0)
```

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
{R(d+1), Rd} = *(int32*)(Address)
```

Usage Notes

Ra and offset are interpreted as byte addresses.

Rd is a 32-bit register pair

A 32-bit memory read is performed.

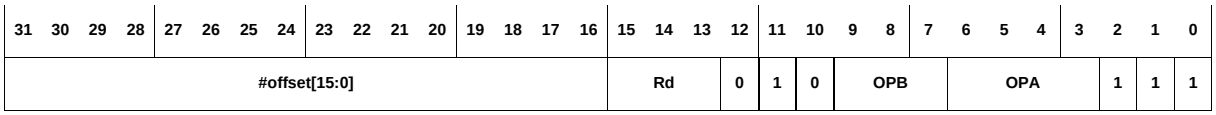
Refer to section 3.8.9, "Error Details", for details of possible exceptions.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
ld.32.i %r2, @(28, %r7)
ld.32.i %r1, @(label, %pc)
ld.32.i %r5, @(23, %pc)
ld.32.i %r3, @(0, %sp)
ld.32.i %r6, @(label, 0)
ld.32.i %r7, @(0x8100, 0)
```

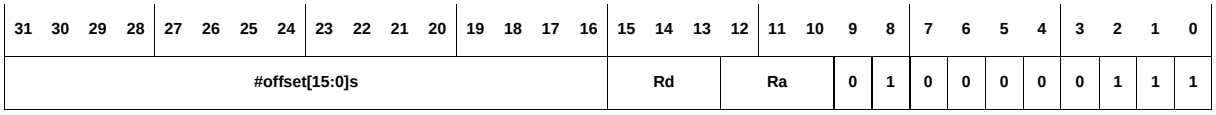
32-bit Encoding - 1



This encodes the form `ld.32.i Rd, @(offset, BaseAddress)`.

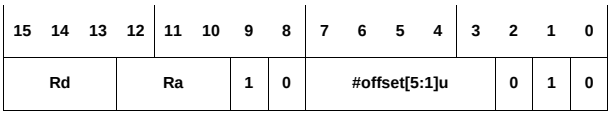
OPB	OPA	BaseAddress	Offset Type
7	1	PC	Signed
3	1	SP	Unsigned
3	0	0	Unsigned

32-bit Encoding - 2



This encodes the form `ld.32.i Rd, @(offset, Ra)`.

16-bit Encoding - 1

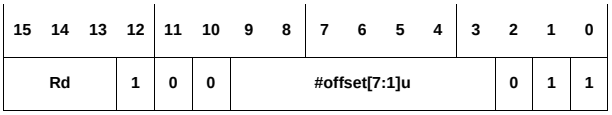


This encodes the form `ld.32.i Rd, @(offset, Ra)`.

The following rules apply:

- This encoding can represent offsets in the range 0 to 62.
- Rd is in the range R0-R6.

16-bit Encoding - 2



This encodes the form `ld.32.i Rd, @(offset, %sp)`.

The following rule applies:

- This encoding can represent offsets in the range 0 to 254.
- Rd is in the range R0-R6.

ld.32.r Load, 32-bit, indexed

Instruction `ld.32.r Rd, @Address`

Description Load 32-bit value from 16-bit address with indexed addressing

`Address` is one of the following:

```
(Rx, Ra)
(Rx, %sp)
```

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
{R(d+1), Rd} = *(int32*)(Address)
```

Usage Notes Rd is a register pair. Ra is interpreted as a byte address. Rx is interpreted as a 4-byte offset from Ra or SP.

A 32-bit memory read is performed.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
ld.32.r %r2, @(%r0, %r7)
ld.32.r %r1, @(%r3, %sp)
```

32-bit Encoding

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
Rd			0	1	0	Rx			1	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

This encodes the form `ld.32.r Rd, @(Rx, %sp)`.

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Ra				Rx				0	1	0	0	1	1	0

ld.i — Load, displacement

Instruction

```
ld.i Rd, @Address
```

Description

Load 16-bit value from 16-bit address with displacement addressing

`Address` is one of the following:

```
(offset, Ra)
(label, %pc)
(offset, %pc)
(offset, %sp)
(label, 0)
(offset, 0)
```

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
Rd = * (int16*)(Address)
```

Usage Notes

Ra and offset are interpreted as byte addresses.

A 16-bit memory read is performed.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
ld.i %r2, @(28, %r7)
ld.i %r1, @(label, %pc)
ld.i %r5, @(23, %pc)
ld.i %r3, @(0, %sp)
ld.i %r6, @(label, 0)
ld.i %r7, @(0x8100, 0)
```

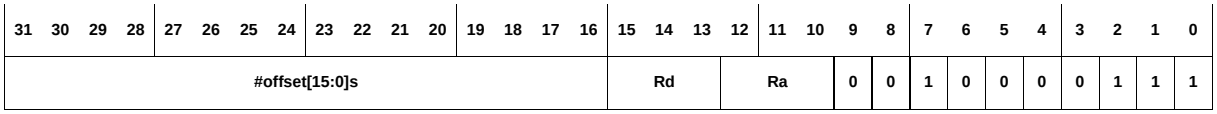
32-bit Encoding - 1

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
#offset[15:0]				Rd	0 0 1 OPB	OPA	1 1 1

This encodes the form `ld.i Rd, @(offset, BaseAddress)`.

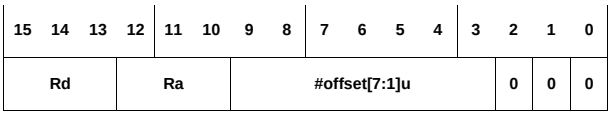
OPB	OPA	BaseAddress	Offset Type
7	1	PC	Signed
3	1	SP	Unsigned
3	0	0	Unsigned

32-bit Encoding - 2



This encodes the form `ld.i Rd, @(offset, Ra)`.

16-bit Encoding - 1

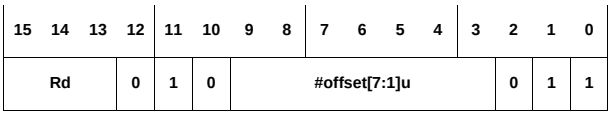


This encodes the form `ld.i Rd, @(offset, Ra)`.

The following rule applies:

- This encoding can represent offsets in the range 0, 2, …, 252, 254.

16-bit Encoding - 2



This encodes the form `ld.i Rd, @(offset, %sp)`.

The following rule applies:

- This encoding can represent offsets in the range 0, 2, …, 252, 254.

ld.r — Load, indexed

Instruction

```
ld.r Rd, @Address
```

Description

Load 16-bit value from 16-bit address with indexed addressing

`Address` is one of the following:

```
(Rx, Ra)
(Rx, %sp)
```

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
Rd = * (int16*)(Address)
```

Usage Notes

Ra is interpreted as a byte address. Rx is interpreted as a 2-byte offset from Ra or SP.

A 16-bit memory read is performed.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
ld.r %r2, @(%r0, %r7)
ld.r %r1, @(%r3, %sp)
```

32-bit Encoding

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
Rd			0	0	1	Rx		1	0	1		1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

This encodes the form `ld.r Rd, @(Rx, %sp)`.

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Ra				Rx				0	0	1	0	1	1	0

ldand.1.i Load with AND, single-bit, displacement

Instruction

```
ldand.1.i %flags[c], %flags[c], @(label[bit], 0)
ldand.1.i %flags[c], %flags[c], @(offset[bit], 0)
```

Description

AND of the C flag with the specified bit in memory.

Flags

Z	-
N	-
C	Set to the result of the operation.
V	-

Operation

```
C = C & ((* (int8*) offset[15:0]u ) >> bit) & 1)
```

Usage Notes

This can only use zero-relative addressing.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
ldand.1.i %flags[c], %flags[c], @(label[6], 0)
```

32-bit Encoding

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
#offset[15:0]u																bit[2:0]			0	0	1	1	1	0	0	1	0	0	1	1	1

ldor.1.i Load with OR, single-bit, displacement

Instruction

```
ldor.1.i %flags[c], %flags[c], @(label[bit], 0)
ldor.1.i %flags[c], %flags[c], @(offset[bit], 0)
```

Description

OR of the C flag with the specified bit in memory.

Flags

Z	-
N	-
C	Set to the result of the operation.
V	-

Operation

```
C = C | ((* (int8*) offset[15:0]u ) >> bit) & 1)
```

Usage Notes

This can only use zero-relative addressing.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
ldor.1.i %flags[c], %flags[c], @(label[6], 0)
```

32-bit Encoding

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
#offset[15:0]u				bit[2:0] 0	1 0 1 1	0 0 1 0	0 1 1 1

ldxor.1.i Load with XOR, single-bit, displacement

Instruction

```
ldxor.1.i %flags[c], %flags[c], @(label[bit], 0)
ldxor.1.i %flags[c], %flags[c], @(offset[bit], 0)
```

Description

XOR of the C flag with the specified bit in memory.

Flags

Z	-
N	-
C	Set to the result of the operation.
V	-

Operation

```
C = C ^ ((* (int8*) offset[15:0]u ) >> bit) & 1)
```

Usage Notes

This can only use zero-relative addressing.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
ldxor.1.i %flags[c], %flags[c], @(label[6], 0)
```

32-bit Encoding

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
#offset[15:0]u																bit[2:0]			0	1	1	1	1	0	0	1	0	0	1	1	1

lic — Read XAP4 licence number

Instruction `lic Rd`

Description Read the XAP4's licence number

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation `{R(d+1), Rd} = XAP4 licence number`

Usage Notes Each XAP4 delivery contains a different licence number.

Examples `lic %r1`

32-bit Encoding

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
Rd			1	0	1	0	1	1	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

mov.1.i — Move, single-bit, immediate

Instruction `mov.1.i %flags[F], #immediate[0]`

Description Set the specified flag to an immediate value.

Flags

Z	Unchanged unless destination is Z flag
N	Unchanged unless destination is N flag
C	Unchanged unless destination is C flag
V	Unchanged unless destination is V flag

Operation `FLAGS[F] = #immediate[0]`

Usage Notes `F can be any one of: z, n, c, v, m0, m1, i, s0, s1, s2, s3, s4, p0, p1, p2 or p3`. Also, `b` and `t` can be used as synonyms for `p0` and `p1`, respectively. Specifying `%flags[i]` allows enabling and disabling interrupts. Operations on flags other than `c` are only possible in the privileged modes.

Refer to flag bit positions in section 3.4.3, "FLAGS register".

This instruction throws a `PrivInstruction_S` exception if an attempt is made to operate a flag other than `z`, `n`, `c` or `v` in User mode.

Examples

```
mov.1.i %flags[i], #1   // Enable interrupts
```

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	1	#i	F[3:0]			1	0	1	

The following rules apply:

- Bit 7 is #immediate[0].
- F[3:0] represents any bit in the FLAGS register.

mov.1.r — Move, single-bit, register

Instruction

```
mov.1.r %flags[c], Rs
mov.1.r %flags[i], Rs
mov.1.r Rd, %flags[F]
mov.1.r %flags[i], %flags[c]
mov.1.r %flags[c], %flags[i]
```

Description

Move between the specified flag and a register (bit 0).

Flags

Z -

N -

C Set to the LSB of Rs in the first form. Unaffected in the second, third and fourth forms. Set to the value of the interrupt flag in the fifth form.

V -

Operation

FLAGS[c] = Rs[0]
Rd[0] = FLAGS[F], Rd[15:1] = 0

Usage Notes

This instruction can only write to c or i, but can read from any flag in the FLAGS register (F can be any one of: `z, n, c, v, m0, m1, i, s0, s1, s2, s3, s4, p0, p1, p2 or p3`. Also, `b` and `t` can be used as synonyms for `p0` and `p1`, respectively). To write to other flags, use `mov.*.i or movr2s.`

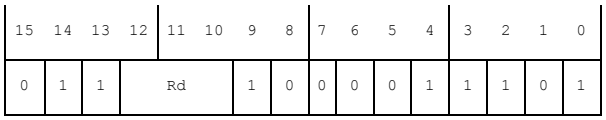
`mov.1.r %flags[i], *` are privileged instructions, the other encodings are not.

Refer to flag bit positions in section 3.4.3, "FLAGS register".

Examples

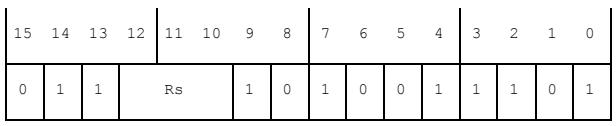
```
mov.1.r %flags[c], %r1
mov.1.r %r1, %flags[i]
mov.1.r %r1, %flags[m1]
```

16-bit Encoding - 1



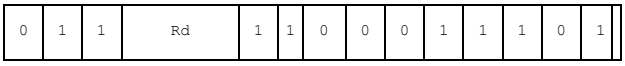
This encodes the form `mov.1.r Rd, %flags[c]`.

16-bit Encoding - 2



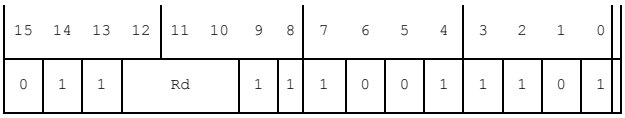
This encodes the form `mov.1.r %flags[c], Rs`.

16-bit Encoding - 3



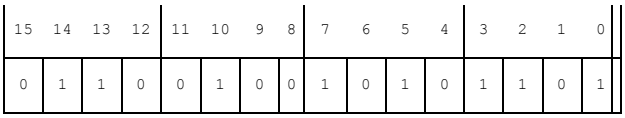
This encodes the form `mov.1.r Rd, %flags[i]`.

16-bit Encoding - 4



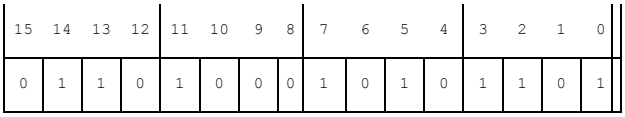
This encodes the form `mov.1.r %flags[i], Rs`. This is a privileged instruction.

16-bit Encoding - 5



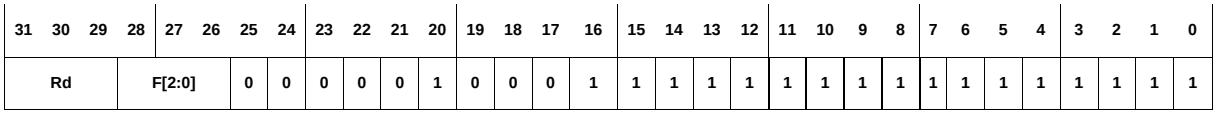
This encodes the form `mov.1.r %flags[c], %flags[i]`.

16-bit Encoding - 6



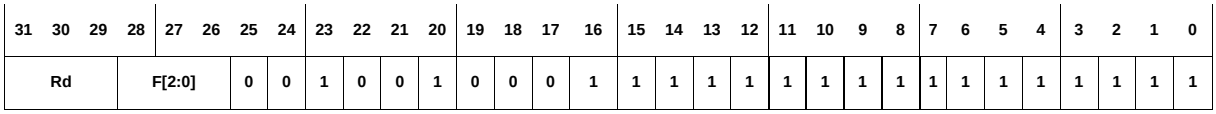
This encodes the form `mov.1.r %flags[i], %flags[c]`. This is a privileged instruction.

32-bit Encoding - 1



This encodes the form `mov.1.r Rd, %flags[F]`, where F is the index of any flag in the lower byte of the FLAGS register (i.e. 0-7).

32-bit Encoding - 2



This encodes the form `mov.1.r Rd, %flags[F]`, where F is the index of any flag in the higher byte of the FLAGS register (i.e. 8-15).

mov.2.i — Move, 2-bit, immediate

Instruction `mov.2.i %flags[m], #immediate`

Description Set the mode flags to an immediate value.

Flags

Z	-
N	-
C	-
V	-

Operation

```
FLAGS[M] = #immediate[1:0]
```

Usage Notes Refer to section 3.4.3, "FLAGS register" for the encodings of the four processor modes.

This instruction throws a `PrivInstruction_S` exception if `excuted` in User mode.

Examples

```
mov.2.i %flags[m], #0  // Enter Supervisor Mode
mov.2.i %flags[m], #3  // Enter User Mode
```

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	#i[1:0]		0	0	1	0	1	0	1	1	0	1

Bits [11:10] specify `#immediate`.

mov.2.r — Move, 2-bit, register

Instruction `mov.2.r Rd, %flags[m]`

Description Store the current mode in a register.

Flags

Z	Set if Rd is 0, cleared otherwise
N	Cleared
C	-
V	-

Operation `Rd[1:0] = FLAGS[M] ; Rd[15:2] = 0`

Usage Notes Refer to section 3.4.3, "FLAGS register" for the encodings of the four processor modes.

Examples `mov.2.r %r1, %flags[m]`

32-bit Encoding

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
Rd			0	0	0	0	1	1	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

mov.4.i — Move, 4-bit, immediate

Instruction

`mov.4.i %flags[p], #immediate`

Description

Set the priority flags to an immediate value.

Flags

Z	-
N	-
C	-
V	-

Operation

```
FLAGS[P] = #immediate[3:0]
```

Usage Notes

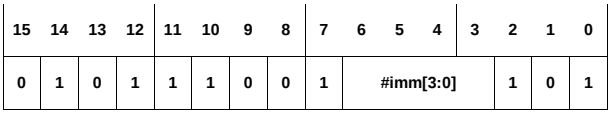
Refer to section 3.4.3, "FLAGS register" for the meanings of the priority flags.

This instruction throws a `PrivInstruction_S` exception if `excuted` in User mode.

Examples

```
mov.4.i %flags[p], #0
mov.4.i %flags[p], #13
```

16-bit Encoding



Bits [6:3] specify `#immediate[3:0]`.

mov.4.r — Move, 4-bit, register

Instruction `mov.4.r Rd, %flags[p]`

Description Store the current priority flags in a register.

Flags

Z	Set if Rd is 0, cleared otherwise
N	Cleared
C	-
V	-

Operation `Rd[3:0] = FLAGS[P] ; Rd[15:4] = 0`

Usage Notes Refer to section 3.4.3, "FLAGS register" for the meanings of the priority flags.

Examples `mov.4.r %r1, %flags[p]`

32-bit Encoding

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
Rd			0	0	1	0	1	1	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

mov.32.r — Move, register pair

Instruction `mov.32.r Rd, Rs`

Description Register pair-to-register pair move

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `{R(d+1), Rd} = {R(s+1), Rs}`

Usage Notes Rd and Rs must specify one of the 32-bit register pairs (`%r0` to `%r7`).

When the destination (Rd) is SP (stack pointer), the Z and N flags are not updated.

Examples `mov.32.r %r1, %r3`

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	Rs					Rd				1	1	0	1

mov.32s.r — Move, register pair, sign extended

Instruction

`mov.32s.r Rd, Rs`

Description

Register-to-register pair move, sign extended.

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
Rd = Rs
R(d+1)[15:0] = Rs[15]
```

Usage Notes

Rd and Rs must specify one of the 32-bit register pairs (`%r0` to `%r7`).

Examples

```
mov.32s.r %r1, %r3
```

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	Rs				Rd		0	1	1	1	1	0	1

mov.32z.r — Move, register pair, zero extended

Instruction

`mov.32z.r Rd, Rs`

Description

Register-to-register pair move, zero extended.

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
Rd = Rs
R(d+1) = 0
```

Usage Notes

Rd and Rs must specify one of the 32-bit register pairs (`%r0` to `%r7`).

Examples

```
mov.32z.r %r1, %r3
```

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	Rs				Rd		0	1	1	0	1	0	1

mov.i — Move, displacement or immediate

Instruction

```
mov.i Rd, (label, %pc)
mov.i Rd, (offset, %pc)

mov.i Rd, (label, 0)
mov.i Rd, (offset, 0)

mov.i Rd, #immediate
```

Description

Move 16-bit address or immediate to register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
Rd = (void*)(offset[15:0]s + PC)

Rd = (void*)(offset[15:0]u + 0)

Rd = #immediate[15:0]
```

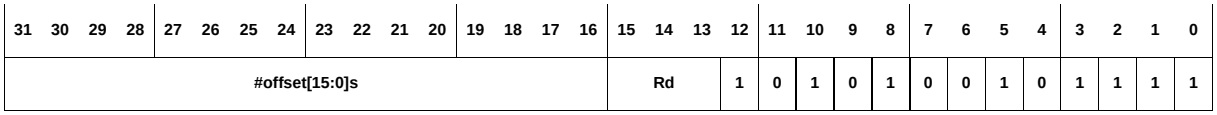
Usage Notes

For small immediates and offsets, 16-bit encodings of this instruction are available.

Examples

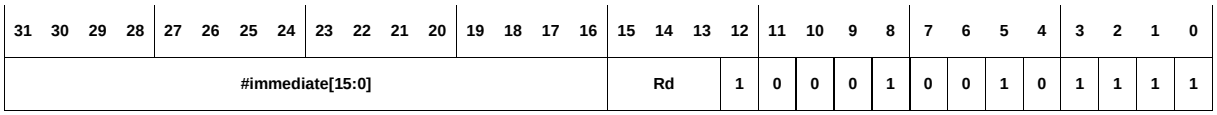
```
mov.i %r1, (label, %pc)

mov.i %r1, (label, 0)

mov.i %r1, #0xFE12
```

32-bit Encoding - 1



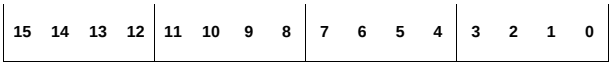
This encodes the PC-relative form.

32-bit Encoding - 2



This encodes the immediate or zero-relative form.
In the zero-relative address forms, offset[15:0]u in used in place of #immediate[15:0].

16-bit Encoding - 1



Rd	0	0	#immediate[7:0]u	1	0	0

In the zero-relative address forms, offset[7:0]u in used in place of #immediate[7:0]u.

The following rule applies:

- This encoding can represent offsets or immediate values in the range 0 to 255.

16-bit Encoding - 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				1	1	1	0	0	#immediate[3:0]u			0	1	1	

The following rules apply:

- Rs is the same as Rd.
- The offset or immediate value is (#immediate[3:0]u – 16) and this gives a range of -1, -2, …, -15, -16.

mov.r — Move, register

Instruction `mov.r Rd, Rs`

Description Register-to-register move

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `Rd = Rs`

Usage Notes SP is valid for the operands Rd and Rs.

When the destination (Rd) is SP (stack pointer), the Z and N flags are not updated.

Examples `mov.r %r1, %r2`

16-bit Encoding -1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	Rs					Rd				1	1	1	0	1	0	1

32-bit Encoding - 1

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
0	0	1	Rs					0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

The following rule applies:

Rd is SP.

32-bit Encoding - 2

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
Rd				0	0	1	0	1	1	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

The following rule applies:

Rs is SP.

mova2r — Move address register to register

Instruction `mova2r Rd, As`

Description Move an address register to a normal register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `Rd = As`

Usage Notes As is an address register. See section 6.8, "Address register fields" for valid operands.

This instruction throws a `PrivInstruction` exception if executed in User mode.

Examples `mova2r %r1, %sp1`

32-bit Encoding

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
Rd				As				1	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

See section 6.8, "Address register fields" for the encoding of As.

movb2r — Move breakpoint register to register

Instruction `movb2r Rd, Bs`

Description Move a breakpoint register to a normal register

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `Rd = Bs`

Usage Notes Bs is a breakpoint register. See section 6.8, "Breakpoint register fields" for valid operands.

A `PrivInstruction_S` exception is thrown if this instruction is executed in User mode.

To access the BRKE register, use the `movs2r` instruction.

Examples `movb2r %r1, %brk0`

32-bit Encoding

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
Rd				Bs				0	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

See section 6.8, "Breakpoint register fields" for the encoding of Bs.

movr2a — Move register to address register

Instruction `movr2a Ad, Rs`

Description Move a normal register to an address register

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation `Ad = Rs`

Usage Notes

Ad is an address register. See section 6.8, "Address register fields" for valid operands.

This instruction throws a `PrivInstruction` exception if executed in User mode.

This instruction throws an `AlignError` exception if an attempt is made to write non-zero values to the low bits of PC, VP and SP that should be zero.

If no exceptions are thrown this instruction can set the K0 and K1 bits in the INFO register: writes to SP0 cause K0 to be set; If K0 is set, writes to SP1 cause K1 to be set. See section 3.6, "Stack Operation", and section 3.4.3, "Special registers", for more information.

Examples `movr2a %sp1, %r1`

32-bit Encoding

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
Ad				Rs																											
						1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

See section 6.8, "Address register fields" for the encoding of Ad.

movr2b — Move register to breakpoint register

Instruction `movr2b Bd, Rs`

Description Move a normal register to a breakpoint register

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation `Bd = Rs`

Usage Notes Bd is a breakpoint register. See section 6.8, "Breakpoint register fields" for valid operands.

A `PrivInstruction_S` exception is thrown if this instruction is executed in User mode.

To access the BRKE register, use the `movr2s` instruction.

Examples `movr2b %brk0, %r1`

32-bit Encoding

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
Bd				Rs				0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

See section 6.8, "Breakpoint register fields" for the encoding of Bd.

movr2s — Move register to special register

Instruction `movr2s Sd, Rs`

Description Move a normal register to a special register

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation `Sd = Rs`

Usage Notes

Sd is one of the 16-bit special registers. See section 6.8, "Special register fields" for valid operands.

The `movs2r` and `movr2s` instructions allow privileged mode access to the special registers.

The `movr2s` instruction can be used to change from any privileged mode to any mode, although it may be easier to use the `mov.2.i` instruction.

A `PrivInstruction_S` exception is thrown if this instruction is executed in User mode.

If this instruction is used to write to the read-only INFO register, it has no effect.

Writing to the S bits of the FLAGS register has no effect.

Examples `movr2s %flags, %r1`

32-bit Encoding

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
Sd				Rs				0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

See section 6.8, "Special register fields" for the encoding of Sd.

movs2r — Move special register to register

Instruction `movs2r Rd, Ss`

Description Move a special register to a normal register

Flags

Z	Set if the result is zero, cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `Rd = Ss`

Usage Notes Ss is one of the 16-bit special registers. See section 6.8, "Special register fields" for valid operands.

The `movs2r` and `movr2s` instructions allow privileged mode access to the Special Registers.

A `PrivInstruction_S` exception is thrown if this instruction is executed in User mode, unless the instruction is executing on the Flags register.

Examples `movs2r %r1, %brke`

32-bit Encoding

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
Rd				Ss				0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

See section 6.8, "Special register fields" for the encoding of Ss.

mult.32s.i — Multiply, 32-bit signed, immediate

Instruction `mult.32s.i Rd, Rs, #immediate`

Description 16-bit by 16-bit signed integer multiply to give a 32-bit result

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `{R(d+1), Rd} = Rs * #immediate[15:0]s`

Usage Notes This is a signed 16x16 multiply, giving a 32-bit product in register pair Rd.

Both operands are treated as 16-bit signed variables and represent numbers in the range -32768 to +32767.

Examples `mult.32s.i %r1, %r2, #0x1234`

32-bit Encoding

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
#immediate[15:0]s																Rd				Rs				1	0	0	0	1	0	0	1	1	1

mult.32s.r — Multiply, 32-bit signed, register

Instruction

```
mult.32s.r Rd, Rs, Rt
```

Description

16-bit by 16-bit signed integer multiply to give a 32-bit result

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
{R(d+1), Rd} = Rs * Rt
```

Usage Notes

This is a signed 16x16 multiply, giving a 32-bit product in register pair Rd.

Both operands are treated as 16-bit signed variables and represent numbers in the range -32768 to +32767.

Examples

```
mult.32s.r %r1, %r2, %r3
```

32-bit Encoding

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
Rd				Rs				Rt				1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

mult.32u.i Multiply, 32-bit unsigned, immediate

Instruction `mult.32u.i Rd, Rs, #immediate`

Description 16-bit by 16-bit unsigned integer multiply to give a 32-bit result

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if bit 31 of Rd is 1; cleared otherwise
C	Unchanged
V	Unchanged

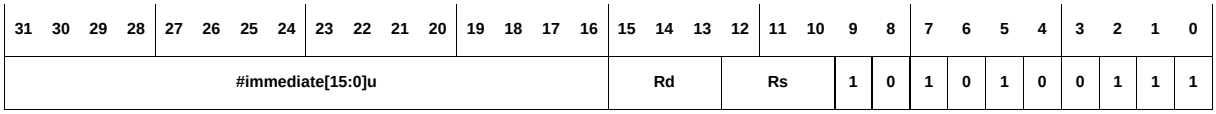
Operation `{R(d+1), Rd} = Rs * #immediate[15:0]u`

Usage Notes This is an unsigned 16x16 multiply, giving a 32-bit product in register pair Rd.

Both operands are treated as 16-bit unsigned variables and represent numbers in the range 0 to 65535.

Examples `mult.32u.i %r1, %r2, #0x1234`

32-bit Encoding



mult.32u.r — Multiply, 32-bit unsigned, register

Instruction

```
mult.32u.r Rd, Rs, Rt
```

Description

16-bit by 16-bit unsigned integer multiply to give a 32-bit result

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if bit 31 of Rd is 1; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
{R(d+1), Rd} = Rs * Rt
```

Usage Notes

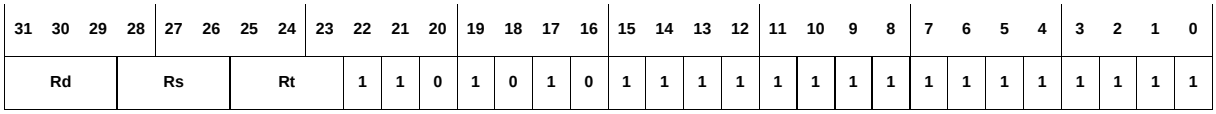
This is an unsigned 16x16 multiply, giving a 32-bit product in register pair Rd.

Both operands are treated as 16-bit unsigned variables and represent numbers in the range 0 to 65535.

Examples

```
mult.32u.r %r1, %r2, %r3
```

32-bit Encoding



mult.i — Multiply, immediate

Instruction `mult.i Rd, Rs, #immediate`

Description 16-bit by 16-bit integer multiply to give a 16-bit result

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `Rd = Rs * #immediate[15:0]s`

Usage Notes This is a 16x16 multiply, giving the low 16 bits of the product in Rd.

Examples

```
mult.i %r1, %r2, #0x1234
mult.i %r1, %r2, #0xFEDC
```

32-bit Encoding

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
#immediate[15:0]s																Rd				Rs				0	1	1	0	0	1	0	1	1	1

mult.r — Multiply, register

Instruction `mult.r Rd, Rs, Rt`

Description 16-bit by 16-bit integer multiply to give low 16 bits of the result

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `Rd = Rs * Rt`

Usage Notes This is a 16x16 multiply, giving the low 16 bits of the product in Rd.

Examples `mult.r %r1, %r2, %r3`

32-bit Encoding

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
Rd	Rs	Rt	1 0 1	0 0 0 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1

mult.sh.r — Multiply, signed shift right, register

Instruction

```
mult.sh.r Rd, Rs, Rt, #immediate
```

Description

16-bit by 16-bit signed integer multiply to give a 32-bit result which is then signed-shifted right by the immediate

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Set if the result is incorrect; cleared otherwise

Operation

```
Rd = (Rs * Rt) >> #immediate[4:0]u
```

Usage Notes

The 16x16 signed multiply of Rs and Rt produces a 32-bit product in a temporary register. The 32-bit product is signed-shifted right by the immediate operand. This shift, like `shiftr.32s.i`, preserves the sign of the product by re-inserting the sign bit into the vacated most-significant bit.

Finally, the low 16 bits are written to Rd.

This instruction is useful when operating on fixed-point numbers, where a normalising shift is needed after a multiplication.

The V flag is set if the upper 16-bits of the temporary register are not equal to the upper bit of the destination register:

```
V = Rtemp[31:16] != Rd[15]
```

Examples

```
mov.i  %r1, #0x3733     // +3.45 in signed 4.12 format
mov.i  %r2, #0xEC52     // -1.23 in signed 4.12 format
mult.sh.r %r3, %r1, %r2, #(16-4)
// Here, %r3 is 0xBC1B, or -4.24 in 4.12 format
```

32-bit Encoding

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
Rd				Rs				Rt				0	0	#immediate[4:0]u		1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1

nop — No operation

Instruction		`nop`
Description		No operation
Flags	**Z**	Unchanged
	N	Unchanged
	C	Unchanged
	V	Unchanged
Operation		-
Usage Notes		-
Examples		`nop`

16-bit Encoding

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

or.1.r OR, single-bit, register

Instruction `or.1.r %flags[c], %flags[c], Rs`

Description Bitwise OR of carry-flag with Rs[0]

Flags

Z	Unchanged
N	Unchanged
C	Set to the result of a bitwise OR of original value and Rs[0]
V	Unchanged

Operation `FLAGS[C] = FLAGS[C] | Rs[0]`

Usage Notes Used to alter the value of the carry flag.

Examples `or.1.r %flags[c], %flags[c], %r1`

16-bit Encoding

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

or.i OR, immediate

Instruction `or.i Rd, Rs, #immediate`

Description Bitwise OR of Rs with an immediate[15:0]u value

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

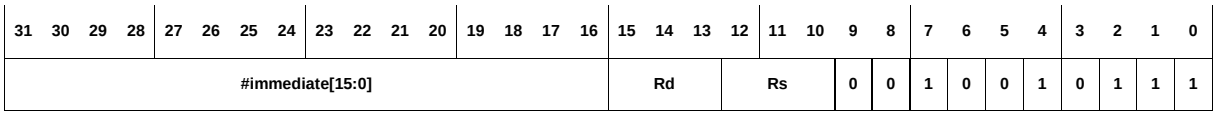
Operation `Rd = Rs | #immediate[15:0]u`

Usage Notes –

Examples

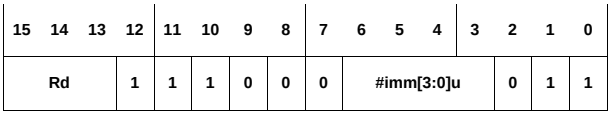
```
or.i %r1, %r2, #0x1234
or.i %r1, %r2, #0xFEDC
```

32-bit Encoding



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
#immediate[15:0]																Rd				Rs				0	0	1	0	0	1	0	1	1	1

16-bit Encoding



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd			1	1	1	0	0	0	#imm[3:0]u			0	1	1	

The following rules apply:

- Rs is the same as Rd.
- This encoding can represent immediates in the range 0, 1, …, 14, 15.

or.r — OR, register

Instruction `or.r Rd, Rs, Rt`

Description Bitwise OR of two registers

Flags

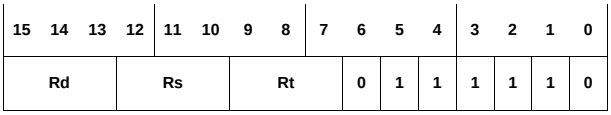
Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `Rd = Rs | Rt`

Usage Notes -

Examples `or.r %r1, %r2, %r3`

16-bit Encoding



pop — Pop from stack

Instruction `pop RegList, #offset`

Description Pop registers from stack

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation `Pop registers in RegList from stack.`

Usage Notes

This instruction can be used to close a stack frame. In addition to popping registers off the stack, the stack pointer can be increased by a further amount to remove the callee's local variables.

The RegList operand specifies which registers are popped and can contain registers in the range R0 to R7. Refer to section 6.10, "Register Lists" for details of RegList specifications. Refer to section 6.3.3, "Push and Pop" for further details of the push and pop instructions.

The offset can take values in the range 0, 2, …, 252, 254.

The operation sequence is:

- Registers `%r0` to `%r7` are loaded from the stack, as specified in RegList. Lower numbered registers are loaded first and from lower stack addresses. Higher numbered registers are loaded last and from higher memory addresses.
- The stack pointer is increased by `#offset`+2n, where n is the number of normal selected registers in RegList.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
pop   {%r3-%r7}, #0
pop   {%r1, %r4-%r6}, #4
```

32-bit Encoding

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
regmask[7:0]		offset[7:1,]u	0	1 0 0 0	1 1 0 1	1 0 1 0	1 1 1 1

16-bit Encoding

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

0	1	0	1	0	regmask[7:0]	1	0	1

The following note applies:

- The offset must be 0.

pop.ret — Pop from stack and return

Instruction `pop.ret RegList, #offset`

Description Pop registers from stack and then return

Flags

Z	Updated
N	Updated
C	Updated
V	Updated

Operation `Pop registers in RegList from stack and return.`

Usage Notes

This instruction is identical to `pop` but additionally returns from a subroutine by popping the return address to PC and setting the flags for R0.

In addition to popping registers off the stack, the stack pointer can be increased by a further amount to remove the callee's local variables.

The RegList operand specifies which registers are popped and can contain registers in the range R0 to R7. Refer to section 6.10, "Register Lists" for details of RegList specifications. Refer to section 6.3.3, "Push and Pop" for further details of the push and pop instructions.

The offset can take values in the range 0, 2, …, 252, 254.

The operation sequence is:

- Registers `%r0` to `%r7` are loaded from the stack, as specified in RegList. Lower numbered registers are loaded first and from lower stack addresses. Higher numbered registers are loaded last and from higher memory addresses.
- The stack pointer is increased by `#offset`+2n+2, where n is the number of normal selected registers in RegList.
- As required by the calling convention, the flags are updated as with a `cmp.i %r0, #0` instruction.
- The return address of the subroutine is popped from the stack into PC.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

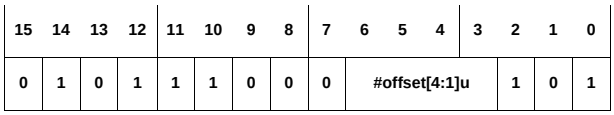
Examples

```
pop.ret {%r3-%r7}, #0
pop.ret {%r1, %r4-%r6}, #4
```

32-bit Encoding

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
regmask[7:0]		offset[7:1]u	0	1 0 1 0	1 1 0 1	1 0 1 0	1 1 1 1

16-bit Encoding - 1

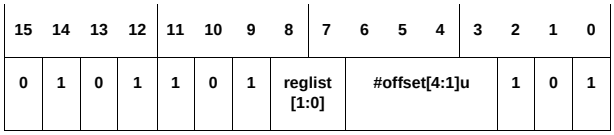


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	0	0	#offset[4:1]u				1	0	1

The following notes apply:

- The RegList is empty.
- The offset is in the range 0, 2, …, 28, 30.

16-bit Encoding - 2



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	reglist [1:0]	#offset[4:1]u					1	0	1

The following notes apply:

- The encoding of registers in the 16-bit form is more complicated than in the 32-bit. Refer to section 6.10, "Register Lists" for details of RegList specifications.
- The offset is in the range 0, 2, …, 28, 30.

print.r — Print, register

Instruction `print.r Rs`

Description Print a register (used in conjunction with a debugger)

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
Print Register
```

Usage Notes The xIDE simulator prints the value in Rs[7:0], interpreted as a character. This can be used for generating `putchar()` output in the debugger.

Hardware implementations of XAP4 treat this instruction as a `nop`.

Examples

```
print.r %r1
```

32-bit Encoding

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
1	0	0	Rs					0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

push — Push to stack

Instruction `push RegList, #offset`

Description Push registers onto the stack

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation `Push registers in RegList to stack.`

Usage Notes

In addition to pushing registers to the stack, the stack pointer can be decreased by a further amount to create a stack frame for the callee's local variables.

The RegList operand specifies which registers are to be loaded and can contain registers in the range R0 to R7. Refer to section 6.10, "Register Lists" for details of RegList specifications. Refer to section 6.3.3, "Push and Pop" for further details of the push and pop instructions.

The offset can take values in the range 0, 2, …, 252, 254.

The operation sequence is:

- The stack pointer is decreased by `#offset`+2n, where n is the number of normal selected registers in RegList.
- Store the selected registers to memory, as specified in RegList. Higher numbered registers are stored next and to higher memory addresses. Lower numbered registers are stored last and to lower stack addresses.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
push {%r7-%r3}, #0
push {%r6-%r4, %r1}, #4
```

32-bit Encoding

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
regmask[7:0]		offset[7:1]u	0	1 1 0 0	1 1 0 1	1 0 1 0	1 1 1 1

16-bit Encoding – 1

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

0	1	0	0	1	regmask[7:0]	1	0	1

The following note applies:

- The offset must be 0.

16-bit Encoding - 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	reglist [1:0]	#offset[4:1]u				1	0	1	

The following notes apply:

- The encoding of registers in the 16-bit form is more complicated than in the 32-bit. Refer to section 6.10, "Register Lists" for details of RegList specifications.
- The offset is in the range 0, 2, …, 28, 30.

push.i — Push immediate to stack

Instruction

```
push.i {#i0 }, #0
push.i {#i0, #i1}, #0
push.i {#i0, #i1, #i2}, #0
push.i {#i0, #i1, #i2, #i3}, #0
```

Description

Push immediates onto the stack

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
Push immediates in ImmList to stack
(i0 first to i3 last)
```

Usage Notes

This is similar to the `push` instruction, although immediates are used instead of registers.

A maximum of 4 immediates can be supplied. The only valid offset is 0. Also, the range of the immediates depends on the number used:

Number of immediates	Range of each immediate
1	-32768 to 32767
2	-128 to 127
3	-8 to 7
4	-8 to 7

Refer to section 6.3.3, "Push and Pop" for further details of the `push.i` instruction.

The operation sequence is:

- The stack pointer is decreased by 2n, where n is the number of immediates given.
- Store the specified immediates. Earlier immediates are stored first and to higher stack addresses. Later immediates are stored last and to lower stack addresses.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
push.i {#0x1234}, #0

push.i {#12, #3, #9, #0}, #0
```

32-bit Encoding

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

#immlist[15:0]	ImmNum	0	1	1	0	1	1	0	1	0	1	1	1	1

The `immlist` field is split into separate immediates in the following way, depending on `ImmNum`:

ImmNum	Format of #immlist[15:0]
0	i0[15:0]
1	i1[7:0]s, i0[7:0]s
2	0000, i2[3:0]s, i1[3:0]s, i0[3:0]s
3	i3[3:0]s, i2[3:0]s, i1[3:0]s, i0[3:0]s

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	immlist[4:0]					1	0	1

This is used for lists of 1-bit immediates.

immlist[4:0] is encoded as follows:

immlist[4:0]	Immediates
0, 0, 0, 1, i0	`{#i0}`
0, 0, 1, i1, i0	`{#i0, #i1}`
0, 1, i2, i1, i0	`{#i0, #i1, #i2}`
1, i3, i2, i1, i0	`{#i0, #i1, #i2, #i3}`

rem.32s.i Remainder, 32-bit signed, immediate

Instruction `rem.32s.i Rd, Rs, #immediate`

Description 32-bit by 16-bit signed divide to give a 16-bit remainder

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Set if an overflow occurs or an attempt is made to divide by zero; cleared otherwise

Operation `Rd = {R(s+1), Rs} % #immediate[15:0]s`

Usage Notes

Rs is a 32-bit register pair.

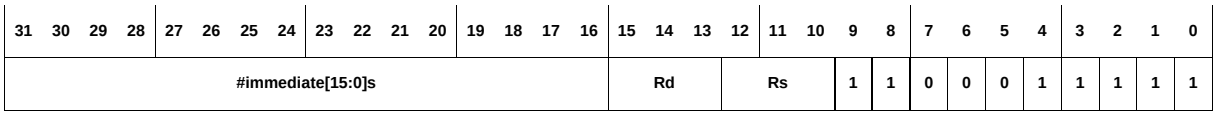
If the (discarded) quotient cannot be represented in 16 bits, the overflow flag is set and the remainder is set to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

The remainder -32768 % -1 gives a result of 0 and sets the overflow flag.

Examples `rem.32s.i %r4, %r1, #10`

32-bit Encoding



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
#immediate[15:0]s				Rd	Rs	1 1 0 0 0 1	1 1 1 1

rem.32s.r Remainder, 32-bit signed, register

Instruction `rem.32s.r Rd, Rs, Rt`

Description 32-bit by 16-bit signed divide to give a 16-bit remainder

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Set if an overflow occurs or an attempt is made to divide by zero; cleared otherwise

Operation `Rd = {R(s+1), Rs} % Rt`

Usage Notes Rs is a 32-bit register pair.

If the (discarded) quotient cannot be represented in 16 bits, the overflow flag is set and the remainder is set to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `rem.32s.r %r4, %r1, %r3`

32-bit Encoding

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
Rd				Rs				Rt				0	1	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1

rem.32u.i Remainder, 32-bit unsigned, immediate

Instruction `rem.32u.i Rd, Rs, #immediate`

Description 32-bit by 16-bit signed divide to give a 16-bit remainder

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if bit 31 of the result is 1; cleared otherwise
C	Set if an overflow occurs or if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation `Rd = {R(s+1), Rs} % #immediate[15:0]u`

Usage Notes Rs is a 32-bit register pair.

If the (discarded) quotient cannot be represented in 16 bits, the carry flag is set and the remainder is set to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `rem.32u.i %r3, %r1, #10`

32-bit Encoding

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
#immediate[15:0]u				Rd	Rs	1 1 1 0 0 1	1 1 1 1

rem.32u.r Remainder, 32-bit unsigned, register

Instruction `rem.32u.r Rd, Rs, Rt`

Description 32-bit by 16-bit signed divide to give a 16-bit remainder

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if bit 31 of the result is 1; cleared otherwise
C	Set if an overflow occurs or if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation `Rd = {R(s+1), Rs} % Rt`

Usage Notes Rs is a 32-bit register pair.

If the (discarded) quotient cannot be represented in 16 bits, the carry flag is set and the remainder is set to zero.

See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `rem.32u.r %r3, %r1, %r2`

32-bit Encoding

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
Rd				Rs				Rt				0	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

rem.s.i — Remainder, signed, immediate

Instruction `rem.s.r Rd, Rs, Rt`

Description 16-bit by 16-bit signed divide to give a 16-bit remainder

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Set if an overflow occurs or an attempt is made to divide by zero; cleared otherwise

Operation `Rd = Rs % #immediate[15:0]s`

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `rem.s.i %r1, %r2, #10`

32-bit Encoding

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
#immediate[15:0]				Rd	Rs 1 0	0 0 0 1	1 1 1 1

rem.s.r — Remainder, signed, register

Instruction `rem.s.r Rd, Rs, Rt`

Description 16-bit by 16-bit signed divide to give a 16-bit remainder

Flags

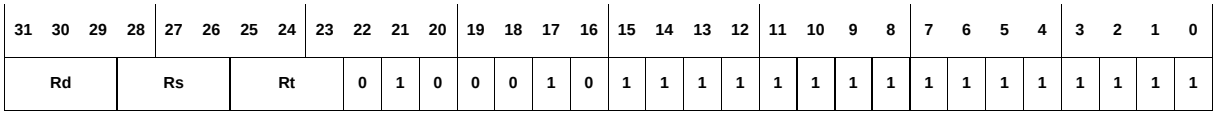
Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Set if an overflow occurs or an attempt is made to divide by zero; cleared otherwise

Operation `Rd = Rs % Rt`

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `rem.s.r %r1, %r2, %r3`

32-bit Encoding



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
Rd	Rs	Rt	0 1 0	0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1

rem.u.i Remainder, unsigned, immediate

Instruction `rem.u.i Rd, Rs, #immediate`

Description 16-bit by 16-bit unsigned divide to give a 16-bit remainder

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if bit 15 of the result is 1; cleared otherwise
C	Set if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation `Rd = Rs % #immediate[15:0]u`

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `rem.u.i %r1, %r2, #10`

32-bit Encoding

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
#immediate[15:0]u				Rd	Rs	1	0	1	0	0	1	1	1	1	1

rem.u.r — Remainder, unsigned, register

Instruction `rem.u.r Rd, Rs, Rt`

Description 16-bit by 16-bit unsigned divide to give a 16-bit remainder

Flags

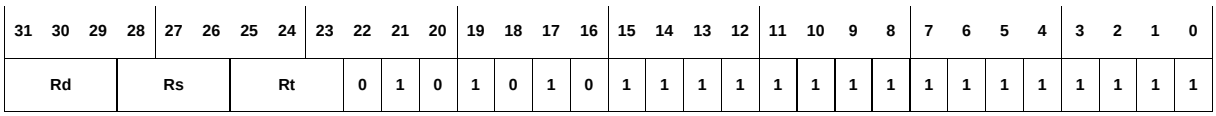
Z	Set if the result is zero; cleared otherwise
N	Set if bit 15 of the result is 1; cleared otherwise
C	Set if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation `Rd = Rs % Rt`

Usage Notes See section 6.1.5, "Divide by zero" for the effects of dividing by zero.

Examples `rem.u.r %r1, %r2, %r3`

32-bit Encoding



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
Rd				Rs				Rt				0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1

rotatel.32.i — Rotate left, 32-bit, immediate

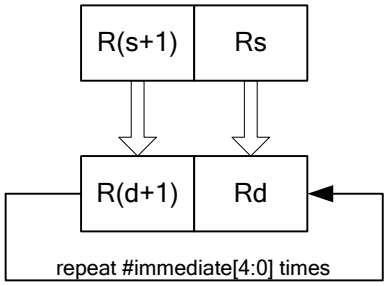
Instruction `rotatel.32.i Rd, Rs, #immediate`

Description 32-bit rotate left

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 31 of Rd
C	The C bit is the final value of bit 0 of Rd
V	Unchanged

Operation



Usage Notes

The rotation is a 32-bit rotation and does not rotate through the carry bit.

Rd and Rs must specify one of the 32-bit register pairs (`%r0` to `%r7`).

The immediate value must be between 0 and 31.

The C flag is not modified if the rotate count is zero.

Examples `rotatel.32.i %r1, %r2, #10`

32-bit Encoding

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
Rd				Rs				0	1	1	0	1	#immediate[4:0]			1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1

16-bit Encoding

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

The following rules apply:

- Rs is Rd
- The immediate value is 16

rotatel.32.r — Rotate left, 32-bit, register

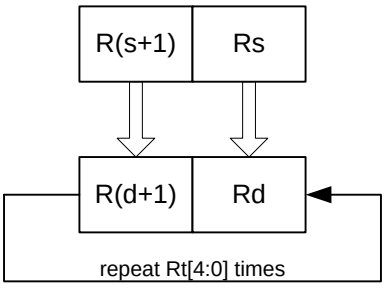
Instruction `rotatel.32.r Rd, Rs, Rt`

Description Rotate left

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 31 of Rd
C	The C bit is the final value of bit 0 of Rd
V	Unchanged

Operation



Usage Notes

The rotation is a 32-bit rotation and does not rotate through the carry bit.

Rd and Rs must specify one of the 32-bit register pairs (%r0 to %r7).

The C flag is not modified if the rotate count is zero.

Examples `rotatel.32.r %r1, %r2, %r3`

32-bit Encoding

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
Rd	Rs	Rt	1 1 1	1 0 0 1	1 1 1 1	1 1 1 1	1 1 1 1

rotatel.i — Rotate left, immediate

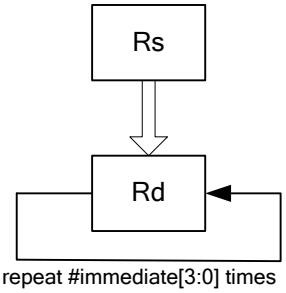
Instruction `rotatel.i Rd, Rs, #immediate`

Description Rotate left

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 15 of Rd
C	The C bit is the final value of bit 0 of Rd
V	Unchanged

Operation



Usage Notes The rotation is a 16-bit rotation and does not rotate through the carry bit.

The immediate value must be between 0 and 15.

The C flag is not modified if the rotate count is zero.

Examples `rotatel.i %r1, %r2, #10`

32-bit Encoding

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
Rd	Rs	1 0	0 0 1 0	#immediate[3:0]	1 1 1 1	1 0 1 1	1 1 1 1	1 1 1 1

16-bit Encoding

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

The following rules apply:

- Rs is Rd
- The immediate value is 8

rotatel.r — Rotate left, register

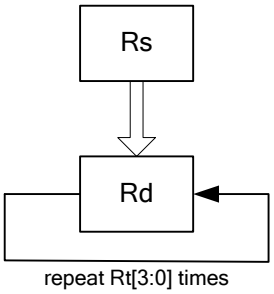
Instruction `rotatel.r Rd, Rs, Rt`

Description Rotate left

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 15 of Rd
C	The C bit is the final value of bit 0 of Rd
V	Unchanged

Operation



Usage Notes The rotation is a 16-bit rotation and does not rotate through the carry bit.

The C flag is not modified if the rotate count is zero.

Examples `rotatel.r %r1, %r2, %r3`

32-bit Encoding

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
Rd	Rs	Rt	1 0 1	1 0 0 1	1 1 1 1	1 1 1 1	1 1 1 1

rtie — Return from interrupt/exception

Instruction `rtie`

Description Return from interrupt or exception

Flags

Z	Restored from stack
N	Restored from stack
C	Restored from stack
V	Restored from stack

Operation

Clear `INFO[NL]` when executed in NMI state
Clear `INFO[R]` when executed in Recovery state
Current Stack popped to `FLAGS`
Current Stack popped to `R0`
Current Stack popped to `R1`
Current Stack popped to `PC`

The stack used is Stack0 (`%sp0`) for Supervisor mode, Interrupt mode, Recovery state and NMI state.

The stack used is Stack1 (`%sp1`) for Trusted mode.

This instruction throws a `PrivInstruction_S` exception if executed in User mode.

Usage Notes The null interrupt handler is simply the `rtie` instruction, restoring the flags and returning to the interrupted code.

Examples

```
rtie
```

16-bit Encoding

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

sext.r — Sign extend, register

Instruction `sext.r Rd`

Description Sign extend from 8 bits to 16 bits

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `Rd[15:8] = Rd[7]`

Usage Notes This instruction sign-extends an 8-bit value by copying the sign bit into the top 8 bits.

Examples `sext %r1`

16-bit Encoding

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

shiftl.32.i — Shift left, 32-bit, immediate

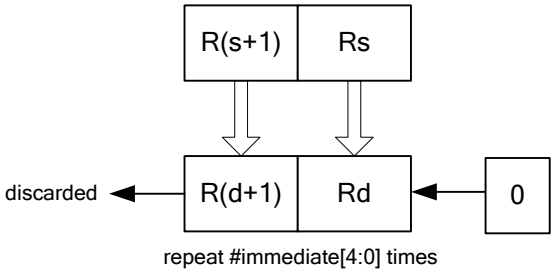
Instruction `shiftl.32.i Rd, Rs, #immediate`

Description 32-bit left shift

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 31 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes

Rd and Rs must specify one of the 32-bit register pairs (%r0 to %r7).

The immediate can take values in the range 0 to 31.

The vacated least-significant bits are filled with zeros.

The C flag is not modified if the shift count is zero.

Examples `shiftl.32.i %r1, %r3, #3`

32-bit Encoding

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
Rd				Rs				0	1	0	0	1	#immediate[4:0]			1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1

shiftl.32.r — Shift left, 32-bit, register

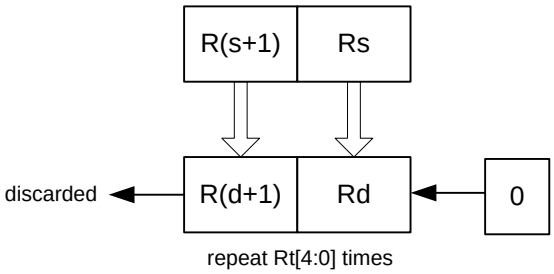
Instruction `shiftl.32.r Rd, Rs, Rt`

Description 32-bit left shift

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 31 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes Rd and Rs must specify one of the 32-bit register pairs (%r0 to %r7).

The vacated least-significant bits are filled with zeros.

The C flag is not modified if the shift count is zero.

Examples `shiftl.32.r %r1, %r3, %r5`

32-bit Encoding

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
Rd	Rs	Rt	1 1 1	0 0 0 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1

shiftl.i — Shift left, immediate

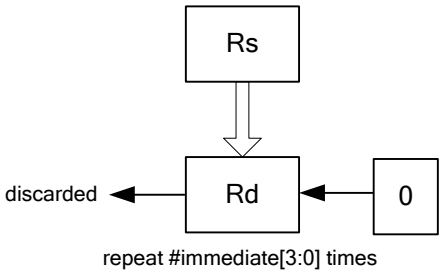
Instruction `shiftl.i Rd, Rs, #immediate`

Description Shift left

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 15 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes

Rd and Rs must be in the range %r0 to %r7.

The immediate can take values in the range 0 to 15.

The vacated least-significant bits are filled with zeros.

The C flag is not modified if the shift count is zero.

Examples `shiftl.i %r1, %r2, #10`

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	Rs			Rd		#immediate[3:0]				1	0	1	

shiftl.r — Shift left, register

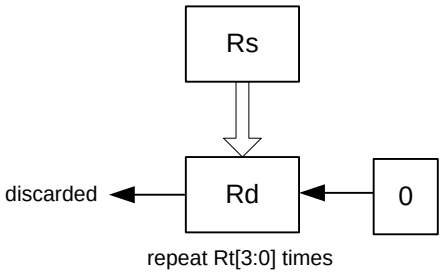
Instruction `shiftl.r Rd, Rs, Rt`

Description Shift left

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 15 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes Rd, Rs and Rt must be in the range %r0 to %r7.

The vacated least-significant bits are filled with zeros.

The C flag is not modified if the shift count is zero.

Examples `shiftl.r %r1, %r2, %r3`

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Rs				Rt				1	0	0	1	1	1	0

shiftr.32s.i — Shift right, 32-bit signed, immediate

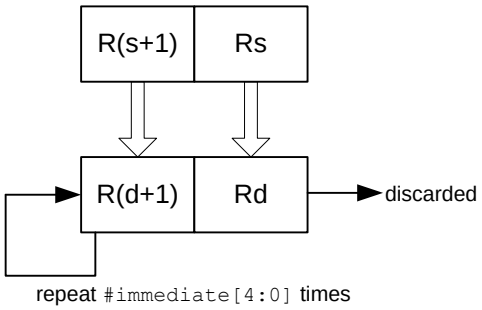
Instruction `shiftr.32s.i Rd, Rs, #immediate`

Description 32-bit signed right shift

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 31 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes

Rd and Rs must specify one of the 32-bit register pairs (%r0 to %r7).

The immediate can take values between 0 and 31.

The vacated most-significant bits are filled with the sign bit.

The C flag is not modified if the shift count is zero.

Examples `shiftr.32s.i %r1, %r3, #3`

32-bit Encoding

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
Rd				Rs				0	0	0	0	1	#immediate[4:0]			1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1

shiftr.32s.r — Shift right, 32-bit signed, register

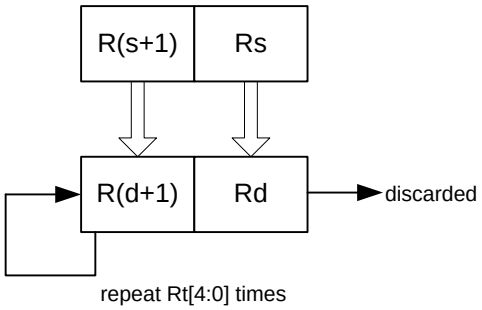
Instruction `shiftr.32s.r Rd, Rs, Rt`

Description 32-bit signed right shift

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 31 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes Rd and Rs must specify one of the 32-bit register pairs (%r0 to %r7).

The vacated most-significant bits are filled with the sign bit.

The C flag is not modified if the shift count is zero.

Examples `shiftr.32s.r %r1, $r3, %r5`

32-bit Encoding

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
Rd				Rs				Rt				1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

shiftr.32u.i Shift right, 32-bit unsigned, immediate

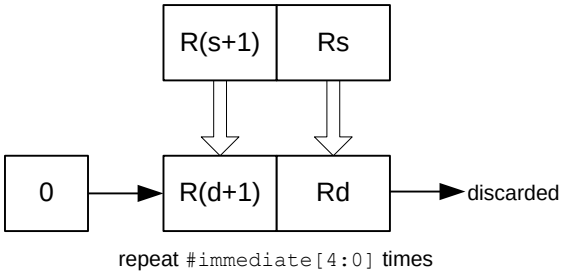
Instruction `shiftr.32u.i Rd, Rs, #immediate`

Description 32-bit signed right shift

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 31 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes

Rd and Rs must specify one of the 32-bit register pairs (%r0 to %r7).

The immediate can take values between 0 and 31.

The vacated most-significant bits are filled with zeros.

The C flag is not modified if the shift count is zero.

Examples `shiftr.32u.i %r1, %r3, #3`

32-bit Encoding

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
Rd	Rs	0 0 1 0 1	#immediate[4:0]	1 1 1 1	1 0 1 1	1 1 1 1	1 1 1 1

shiftr.32u.r Shift right, 32-bit unsigned, register

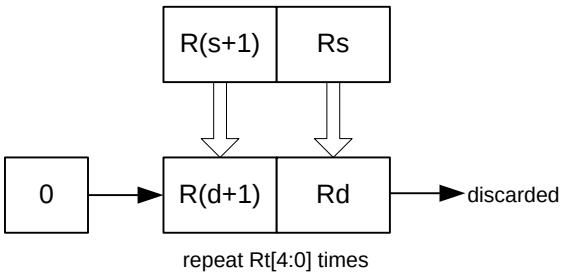
Instruction `shiftr.32u.r Rd, Rs, Rt`

Description 32-bit unsigned right shift

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 31 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes Rd and Rs must specify one of the 32-bit register pairs (%r0 to %r7).

The vacated most-significant bits are filled with zeros.

The C flag is not modified if the shift count is zero.

Examples `shiftr.32u.r %r1, %r3, %r5`

32-bit Encoding

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
Rd				Rs				Rt				1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

shiftr.s.i — Shift right, signed, immediate

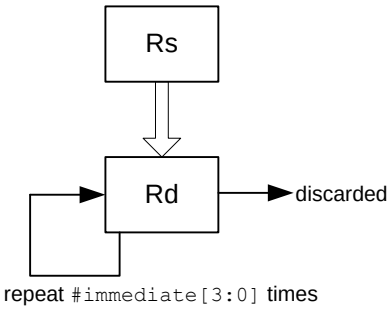
Instruction `shiftr.s.i Rd, Rs, #immediate`

Description 16-bit signed right shift

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 15 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes The immediate can take values between 0 and 15.

The vacated most-significant bits are filled with the sign bit.

The C flag is not modified if the shift count is zero.

Examples `shiftr.s.i %r1, %r3, #3`

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	Rs				Rd				#imm[3:0]u		1	0	1

shiftr.s.r — Shift right, signed, register

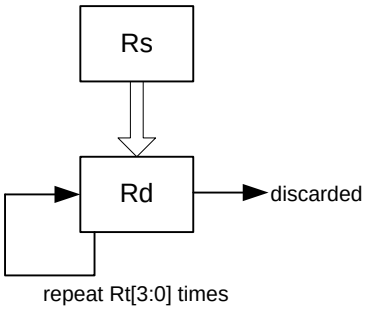
Instruction `shiftr.s.r Rd, Rs, Rt`

Description 16-bit signed right shift

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 15 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes The vacated most-significant bits are filled with the sign bit.

The C flag is not modified if the shift count is zero.

Examples `shiftr.s.r %r1, %r3, %r5`

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Rs				Rt				1	0	1	0	1	1	0

shiftr.u.i — Shift right, unsigned, immediate

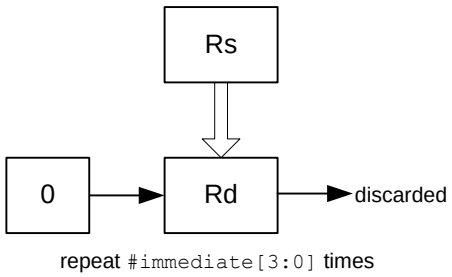
Instruction `shiftr.u.i Rd, Rs, #immediate`

Description 16-bit unsigned shift right

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 15 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes The immediate value must be between 0 and 15.

The vacated most-significant bits are filled with zeros.

The C flag is not modified if the shift count is zero.

Examples

```
shiftr.u.i %r1, %r2, #3
```

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	Rs				Rd				#imm [3:0]u		1	0	1

shiftr.u.r — Shift right, unsigned, register

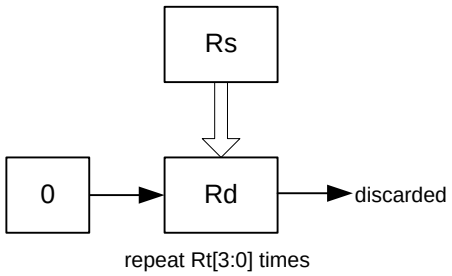
Instruction `shift.u.r Rd, Rs, Rt`

Description 16-bit unsigned right shift

Flags

Z	Set if the result is zero; cleared otherwise
N	The N bit is the final value of bit 15 of Rd
C	The C bit contains the last bit shifted out
V	Unchanged

Operation



Usage Notes The vacated most-significant bits are filled with zeroes.

The C flag is not modified if the shift count is zero.

Examples `shiftr.u.r %r1, %r2, %r3`

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Rs				Rt				1	0	1	1	1	1	0

sif SIF

Instruction `sif`

Description Perform SIF cycle

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if (FLAGS[M] != UserMode) then
   allow SIF access
else
   throw PrivInstruction_S exception
endif
```

Usage Notes The `sif` instruction allows the XAP4 to perform a SIF cycle.

This instruction throws a `PrivInstruction_S` exception if executed in User mode.

Examples

```
sif
```

32-bit Encoding

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
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

16-bit Encoding

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

sleepnop — Sleep

Instruction `sleepnop`

Description Put the XAP4 into NOP Sleep state

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if (FLAGS[M] == UserMode) then
   throw PrivInstruction_S exception
else
   put XAP4 into NOP Sleep state
endif
```

Usage Notes

Put the XAP4 into the NOP Sleep state. SIF cycles are not allowed in the NOP Sleep state. xIDE is unable to gain access to XAP4 in this state.

A `PrivInstruction_S` exception is thrown if this instruction is executed in User mode.

Examples `sleepnop`

16-bit Encoding

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

sleepsif — Sleep and allow SIF

Instruction `sleepsif`

Description Put the XAP4 into SIF Sleep state

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if (FLAGS[M] == UserMode) then
   throw PrivInstruction_S exception
else
   put XAP4 into SIF Sleep mode
endif
```

Usage Notes

Put the XAP4 into the SIF Sleep state. SIF cycles are allowed in the SIF Sleep state.

A `PrivInstruction_S` exception is thrown if this instruction is executed in User mode.

Examples `sleepsif`

16-bit Encoding

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

softreset — Soft Reset

Instruction `softreset`

Description Trigger a Soft Reset

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if (FLAGS[M] != UserMode) then
   throw SoftReset
else
   throw PrivInstruction_S error
endif
```

Usage Notes

A `PrivInstruction_S` error is thrown if this instruction is executed in User mode.

See section 3.7.2, "Soft Reset" for details. Note that, for the error code in R3, it will appear as though the Soft Reset were caused by the `SoftReset` exception.

Examples `softreset`

16-bit Encoding

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

st.1.i Store, single-bit, displacement

Instruction

```
st.1.i %flags[c], @(label[bit], 0)
st.1.i %flags[c], @(offset[bit], 0)

st.1.i #0,        @(label[bit], 0)
st.1.i #0,        @(offset[bit], 0)

st.1.i #1,        @(label[bit], 0)
st.1.i #1,        @(offset[bit], 0)
```

Description

Store 1-bit value to specified bit in memory.

Flags

Z	-
N	-
C	-
V	-

Operation

-

Usage Notes

`This instruction sets, clears or copies the FLAGS[C] bit into a single bit of memory. The destination address must support bit operations. Typically this will be the case for I/O registers but not for general RAM.`

This can only use zero-relative addressing.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
st.1.i %flags[c], @(label[6], 0)

st.1.i #0, @(label[6], 0)

st.1.i #1, @(label[6], 0)
```

32-bit Encoding - 1

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
#offset[15:0]u				bit[2:0] 1	1 0 1 1	0 0 1 0	0 1 1 1

This encodes the form `st.1.i %flags[c], @(offset[bit], 0)`.

32-bit Encoding - 2

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
#offset[15:0]u				bit[2:0] 1	0 #b 1 1	0 0 1 0	0 1 1 1

This encodes the form `st.1.i #b, @(offset[bit], 0)`.

st.8.i Store, 8-bit, displacement

Instruction

```
st.8.i Source, @Address
```

Description

Store 8-bit value to 16-bit address with displacement addressing

`Source` is one of the following:

```
Normal Register - %r0 … %r7
Immediate - # -1, #0 or #1.
```

`Address` is one of the following:

```
(offset, Ra)
(label, %pc)
(offset, %pc)
(offset, %sp)
(label, 0)
(offset, 0)
```

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
* (int8*)(Address) = Source
```

Usage Notes

Ra and offset are interpreted as byte addresses.

An 8-bit memory write is performed.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
st.8.i %r1, @(28, %r7)
st.8.i #1, @(label, %pc)
st.8.i %r3, @(23, %pc)
st.8.i #0, @(0, %sp)
st.8.i %r6, @(label, 0)
st.8.i #-1, @(0x8100, 0)
```

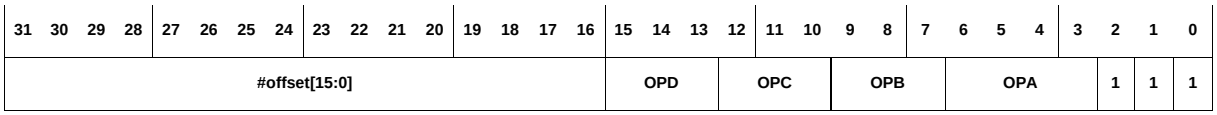
Immediate Codes

Small immediates are represented with the following codes:

ImmCode	Value
0	#0
3	#-1
4	#1

These are used in the following 32-bit encodings, in the OPD field.

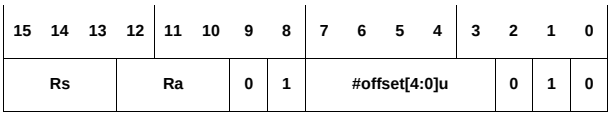
32-bit Encoding



This encodes the form `st.8.i Source, @(offset, BaseAddress).` `Source` can be a register (Rs) or an immediate (coded by OPD).

OPD	OPC	OPB	OPA	BaseAddress	Offset Type
Rs	Ra	4	0	R0 – R7	Signed
ImmCode	Ra	7	0	R0 – R7	Signed
Rs	4	7	1	PC	Signed
ImmCode	7	7	1	PC	Signed
Rs	4	3	1	SP	Unsigned
ImmCode	7	3	1	SP	Unsigned
Rs	4	3	0	0	Unsigned
ImmCode	7	3	0	0	Unsigned

16-bit Encoding - 1

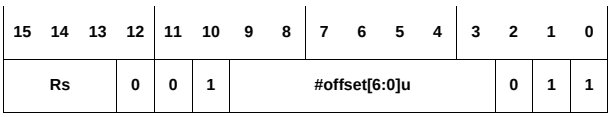


This encodes the form `st.8.i Rs, @(offset, Ra).`

The following rule applies:

- This encoding can represent offsets in the range 0 to 31.

16-bit Encoding - 2

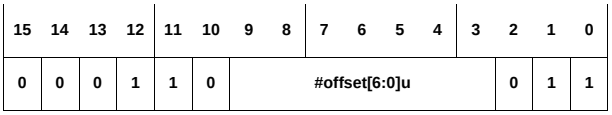


This encodes the form `st.8.i Rs, @(offset, %sp).`

The following rule applies:

- This encoding can represent offsets in the range 0 to 127.

16-bit Encoding - 3



This encodes the form `st.8.i #0, @(offset, %sp).`

The following rule applies:

- This encoding can represent offsets in the range 0 to 127.

16-bit Encoding - 4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	Ra		#offset[3:0]u				0	1	1	

This encodes the form `st.8.i #0, @(offset, Ra)`.

The following rule applies:

- This encoding can represent offsets in the range 0 to 15.

st.8.r Store, 8-bit, indexed

Instruction

```
st.8.r Source, @Address
```

Description

Store 8-bit `Source` to `Address` with indexed addressing.

`Source` is one of the following:

```
Normal Register - %r0 … %r7
Immediate - #-1, #0 or #1
```

`Address` is one of the following:

```
(Rx, Ra)
(Rx, %sp)
```

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
*(int8*)(Address) = Source
```

Usage Notes

Ra is interpreted as a 16-bit byte address. Rx is interpreted as a byte offset from Ra or SP.

An 8-bit memory write is performed.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
st.8.r %r1, @(%r2, %r7)
st.8.r #1, @(%r3, %sp)
st.8.r #0, @(%r1, %r0)
st.8.r #-1, @(%r0, %r3)
```

Immediate Codes

Small immediates are represented with the following codes:

ImmCode	Value
0	#0
3	#-1
4	#1

These are used in the following 32-bit encodings, in the OPD2 field.

32-bit Encoding – 1

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
ImmCode				Ra				Rx				0	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

This encodes the form `st.8.r #immediate, @(Rx, Ra).`

32-bit Encoding - 2

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
OPD2				OPC2				Rx				1	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

This encodes the form `st.8.r Source, @(Rx, %sp).`

`Source` can be a register (Rs) or an immediate (coded by OPD2).

OPD2	OPC2
Rs	4
ImmCode	7

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs				Ra				Rx				0	0	0	1	1	1	0

This encodes the form `st.8.r Rs, @(Rx, Ra).`

st.32.i Store, 32-bit, displacement

Instruction

```
st.32.i Source, @Address
```

Description

Store 32-bit value to 16-bit address with displacement addressing

`Source` is one of the following:

```
Normal Register - %r0 … %r7
Immediate - #0, or #1
```

`Address` is one of the following:

```
(offset, Ra)
(label, %pc)
(offset, %pc)
(offset, %sp)
(label, 0)
(offset, 0)
```

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
* (int32*)(Address) = Source
```

Usage Notes

Ra and offset are interpreted as byte addresses.

Rs is a 32-bit register pair.

A 32-bit memory write is performed.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

For small offsets, a 16-bit encoding of this instruction is available.

Examples

```
st.32.i %r1, @(28, %r7)
st.32.i #1, @(label, %pc)
st.32.i %r3, @(23, %pc)
st.32.i #0, @(0, %sp)
st.32.i %r6, @(label, 0)
st.32.i #0, @(0x8100, 0)
```

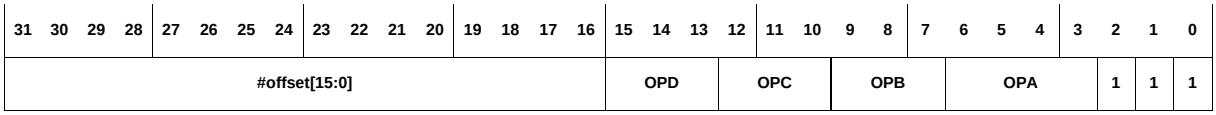
Immediate Codes

Small immediates are represented with the following codes:

ImmCode	Value
2	#0
6	#1

These are used in the following 32-bit encodings, in the OPD field.

32-bit Encoding

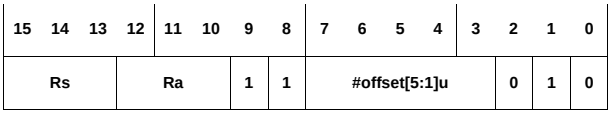


This encodes the form `st.32.i Source, @(offset, BaseAddress)`.

`Source` can be a register (Rs) or an immediate (coded by OPD).

OPD	OPC	OPB	OPA	BaseAddress	Offset Type
Rs	Ra	6	0	R0 – R7	Signed
ImmCode	Ra	7	0	R0 – R7	Signed
Rs	6	7	1	PC	Signed
ImmCode	7	7	1	PC	Signed
Rs	6	3	1	SP	Unsigned
ImmCode	7	3	1	SP	Unsigned
Rs	6	3	0	0	Unsigned
ImmCode	7	3	0	0	Unsigned

16-bit Encoding - 1

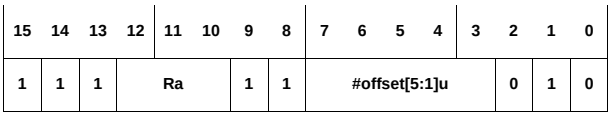


This encodes the form `st.32.i Rs, @(offset, Ra)`.

The following rules apply:

- This encoding can represent offsets in the range 0, 2 …, 60, 62.
- Rs is in the range R0-R6.

16-bit Encoding - 2

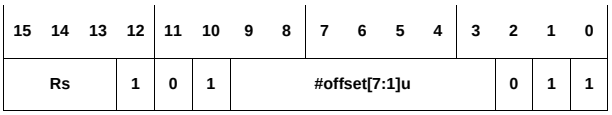


This encodes the form `st.32.i #0, @(offset, Ra)`.

The following rule applies:

- This encoding can represent offsets in the range 0, 2 …, 60, 62.

16-bit Encoding - 3

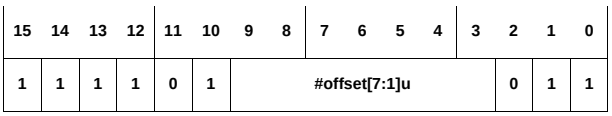


This encodes the form `st.32.i Rs, @(offset, %sp)`.

The following rule applies:

- This encoding can represent offsets in the range 0, 2, …, 252, 254.
- Rs is in the range R0-R6.

16-bit Encoding - 4



This encodes the form `st.32.i #0, @(offset, %sp)`.

The following rule applies:

- This encoding can represent offsets in the range 0, 2, …, 252, 254.

st.32.r Store, 8-bit, indexed

Instruction

`st.32.r Source, @Address`

Description

Store 32-bit value to 16-bit address with indexed addressing

`Source` is one of the following:

```
Normal Register - %r0 … %r7
Immediate - #0 or #1
```

`Address` is one of the following:

```
(Rx, Ra)
(Rx, %sp)
```

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
*(int32*)(Address) = Source
```

Usage Notes

Ra is interpreted as a 16-bit byte address. Rx is interpreted as a 4-byte offset from Ra or SP

Rs is a 32-bit register pair.

A 32-bit memory write is performed.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
st.32.r %r1, @(%r2, %r7)
st.32.r #1, @(%r3, %sp)
st.32.r #0, @(%r1, %r0)
st.32.r #-1, @(%r0, %r3)
```

Immediate Codes

Small immediates are represented with the following encodings:

ImmCode	Value
2	#0
6	#1

These are used in the following 32-bit encodings, in the OPD2 field.

32-bit Encoding - 1

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
ImmCode				Ra				Rx				0	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

This encodes the form `st.32.r immediate, @(Rx, Ra).`

32-bit Encoding - 2

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
OPD2				OPC2				Rx				1	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

This encodes the form `st.32.r Source, @(Rx, %sp)`

`Source` can be a register (Rs) or an immediate (coded in OPD2).

OPD2	OPC2
Rs	6
ImmCode	7

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs				Ra				Rx				0	1	0	1	1	1	1	0

This encodes the form `st.32.r Rs, @(Rx, Ra).`

st.i — Store, displacement

Instruction

```
st.i Source, @Address
```

Description

Store 16-bit value to 16-bit address with displacement addressing

`Source` is one of the following:

```
Normal Register - %r0 … %r7
Immediate - #-1,  #0 or #1
```

`Address` is one of the following:

```
(offset, Ra)
(label, %pc)
(offset, %pc)
(offset, %sp)
(label, 0)
(offset, 0)
```

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
*(int16*)(Address) = Source
```

Usage Notes

Ra and offset are interpreted as byte addresses.

A 16-bit memory write is performed.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

For small offsets, 16-bit encodings of this instruction are available.

Examples

```
st.i %r1, @(28, %r7)
st.i #1, @(label, %pc)
st.i %r3, @(23, %pc)
st.i #0, @(0, %sp)
st.i %r6, @(label, 0)
st.i #-1, @(0x8100, 0)
```

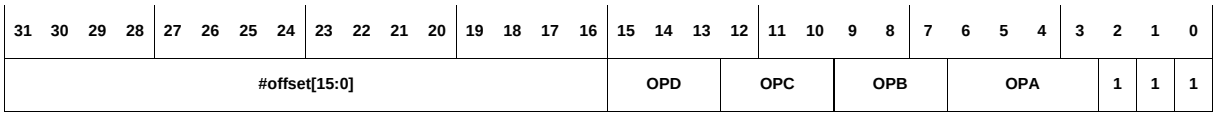
Immediate Codes

Small immediates are represented with the following codes:

These are used in the following 32-bit encodings, in the OPD field.

ImmCode	Value
1	#0
5	#1
7	#-1

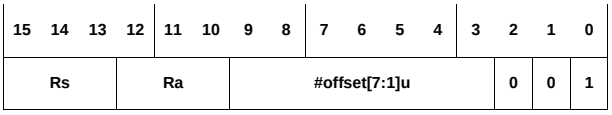
32-bit Encoding - 1



This encodes the form `st.i Source, @(offset, BaseAddress)`. `Source` can be a register (Rs) or an immediate (coded by OPD).

OPD	OPC	OPB	OPA	BaseAddress	Offset Type
Rs	Ra	5	0	R0 – R7	Signed
ImmCode	Ra	7	0	R0 – R7	Signed
Rs	5	7	1	PC	Signed
ImmCode	7	7	1	PC	Signed
Rs	5	3	1	SP	Unsigned
ImmCode	7	3	1	SP	Unsigned
Rs	5	3	0	0	Unsigned
ImmCode	7	3	0	0	Unsigned

16-bit Encoding - 1

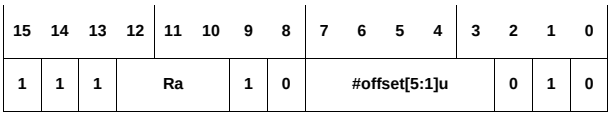


This encodes the form `st.i Rs, @(offset, Ra)`.

The following rule applies:

- This encoding can represent offsets in the range 0, 2, …, 252, 254.

16-bit Encoding - 2

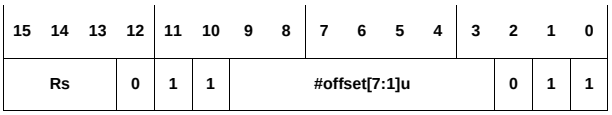


This encodes the form `st.i #0, @(offset, Ra)`.

The following rule applies:

- This encoding can represent offsets in the range 0, 2, …, 60, 62.

16-bit Encoding - 3

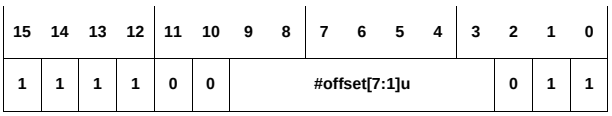


This encodes the form `st.i Rs, @(offset, %sp)`.

The following rule applies:

- This encoding can represent offsets in the range0, 2, …, 252, 254.

16-bit Encoding - 4



This encodes the form `st.i #0, @(offset, %sp)`.

The following rule applies:

- This encoding can represent offsets in the range 0, 2, …, 252, 254.

st.r Store, indexed

Instruction

```
st.r Source, @Address
```

Description

Store 16-bit value to 16-bit address with indexed addressing

`Source` is one of the following:

```
Normal Register - %r0 … %r7
Immediate - #-1, #0 or #1
```

`Address` is one of the following:

```
(Rx, Ra)
(Rx, %sp)
```

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
*(int16*)(Address) = Source
```

Usage Notes

Ra is interpreted as a 16-bit byte address. Rx is interpreted as a 2-byte offset from Ra or SP.

A 16-bit memory write is performed.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples

```
st.r %r1, @(%r2, %r7)
st.r #1, @(%r3, %sp)
st.r #0, @(%r1, %r0)
```

Immediate Codes

Small immediates are represented with the following codes:

These are used in the following 32-bit encodings, in the OPD2 field.

ImmCode	Value
1	#0
5	#1
7	#-1

32-bit Encoding - 1

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
ImmCode				Ra				Rx				0	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

This encodes the form `st.r immediate, @(Rx, Ra)`.

32-bit Encoding - 2

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
OPD2				OPC2				Rx				1	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

This encodes the form `st.r Source, @(Rx, %sp)`

`Source` can be a register (Rs) or an immediate .

OPD2	OPC2
Rs	5
ImmCode	7

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs				Ra				Rx				0	0	1	1	1	1	0

This encodes the form `st.r Rs, @(Rx, Ra)`.

sub.c.i — Subtract, with carry, immediate

Instruction

```
sub.c.i Rd, Rs, #0
```

Description

Subtract with carry

Flags

Z	Set if the result is zero and the Z flag was already set; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Set if the result of the unsigned operation is incorrect; cleared otherwise
V	Set if the result of the signed operation is incorrect; cleared otherwise

Operation

```
Rd = Rs - C
```

Usage Notes

`sub.c.i` can be used for signed or unsigned, integer or fixed-point arithmetic.

The Z flag behaviour is useful for 32-bit arithmetic.

Examples

```
sub.c.i %r1, %r2, #0
```

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	Rd				Rs		0	0	1	1

sub.c.r — Subtract, with carry, register

Instruction `sub.c.r Rd, Rs, Rt`

Description Subtract with carry

Flags

Z	Set if the result is zero and the Z flag was already set; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Set if the result of the unsigned operation is incorrect; cleared otherwise
V	Set if the result of the signed operation is incorrect; cleared otherwise

Operation `Rd = Rs - Rt - C`

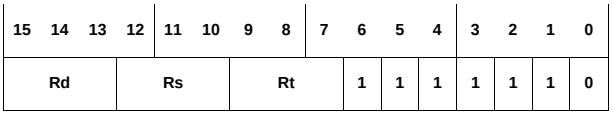
Usage Notes `sub.c.r` can be used for signed or unsigned, integer or fixed-point arithmetic.

The Z flag behaviour is useful for 32-bit arithmetic.

Examples

```
sub.c.r %r1, %r2, %r3
```

16-bit Encoding



sub.r — Subtract, register

Instruction `sub.r Rd, Rs, Rt`

Description Subtract

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Set if the result of the unsigned operation is incorrect; cleared otherwise
V	Set if the result of the signed operation is incorrect; cleared otherwise

Operation `Rd = Rs - Rt`

Usage Notes `sub.r` can be used for signed or unsigned, integer or fixed-point arithmetic.

Examples `sub.r %r1, %r2, %r3`

16-bit Encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				Rs				Rt				1	1	1	0	1	1	0

sub.x.i — Subtract, exchange, immediate

Instruction `sub.x.i Rd, Rs, #immediate`

Description Exchanged order subtract immediate[15:0]s

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Set if the result of the unsigned operation is incorrect; cleared otherwise
V	Set if the result of the signed operation is incorrect; cleared otherwise

Operation `Rd = #immediate[15:0]s - Rs`

Usage Notes `sub.x.i` can be used for signed or unsigned, integer or fixed-point arithmetic.

Examples

```
sub.x.i %r1, %r2, #0x1234
sub.x.i %r1, %r2, #0xFEDC
```

32-bit Encoding

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
#immediate[15:0]s																Rd				Rs				1	1	0	0	0	1	0	1	1	1

16-bit Encoding - 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	Rd			Rs			0	0	1	1

The following rule applies:

1. The immediate is 0x0.

sub.xc.i Subtract, exchange, with carry, immediate

Instruction `sub.x.i Rd, Rs, #immediate`

Description Exchanged order subtract immediate[15:0]s

Flags

Z	Set if the result is zero and the Z flag was already set; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Set if the result of the unsigned operation is incorrect; cleared otherwise
V	Set if the result of the signed operation is incorrect; cleared otherwise

Operation `Rd = #immediate[15:0]s - Rs - C`

Usage Notes `sub.xc.i` can be used for signed or unsigned, integer or fixed-point arithmetic.

The Z flag behaviour is useful for 32-bit arithmetic.

Examples

```
sub.xc.i %r1, %r2, #0x1234
sub.xc.i %r1, %r2, #0xFEDC
```

32-bit Encoding

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
#immediate[15:0]s				Rd	Rs	1 1 1 0 0 1	0 1 1 1

16-bit Encoding

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 0 0 1	1 0 Rd	Rs	1 0 1 1

The following rule applies:

2. The immediate is 0x0.

swap.i — Swap register with memory

Instruction `swap.i Rd, @(0, Ra)`

Description Swap register with memory

Flags

Z	Set if the new Rd value is zero; cleared otherwise
N	Set if the new Rd value is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `*Ra <-> Rd`

Usage Notes

The `swap.i` instruction performs an atomic swap between a register and a memory location.

Ra is interpreted as a byte address.

The memory operations are 16 bits wide.

Refer to section 3.8.9, "Error Details", for details of possible exceptions.

Examples `swap.i %r1, (0, %r3)`

32-bit Encoding

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
Rd				Ra																											
						0	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

syscall.i — System call, immediate

Instruction

```
syscall.i num, #immediate
```

Description

Enter privileged mode from any mode

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if (INFO[NL] != 0 || INFO[R] != 0) then
    soft reset
endif
if (FLAGS[M] == User Mode) then
    switch to Trusted Mode
    throw SysCall<num>_T service
endif

if (FLAGS[M] == Trusted Mode) then
    throw SysCall<num>_T service
else
    throw SysCall<num>_SI service
endif
```

In the Syscall exception handler:

```
R0 = #imm[15:0]
R1 = 0
```

Usage Notes

Allows User mode to call Privileged mode functions. When used in User mode, the mode is changed to Trusted mode. When used in Trusted mode, the mode is unchanged. These cases use Stack1.

Supervisor and Interrupt modes may also use syscall.i, but it will be more efficient to make a direct function call. When used in these modes, the mode is unchanged. These cases use Stack0.

Syscall.i should not be used when INFO[NL]=1 or INFO[R]=1. Such attempts will generate a Soft Reset.

The number can take values in the range 0 to 3. The immediate can take values in the range 0 to 65535.

Examples

```
syscall.i 0, #0x0123
syscall.i 3, #0xFEDC
```

32-bit Encoding

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
#immediate[15:0]				0 num[1:0] 0	0 0 0 1	0 0 1 0	1 1 1 1

syscall.r — System call, register

Instruction

```
syscall.r num, Rs
```

Description

Enter privileged mode from any mode

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if (INFO[NL] != 0 || INFO[R] != 0) then
    soft reset
endif
if (FLAGS[M] == User Mode) then
    switch to Trusted Mode
    throw SysCall<num>_T service
endif

if (FLAGS[M] == Trusted Mode) then
    throw SysCall<num>_T service
else
    throw SysCall<num>_SI service
endif
```

In the Syscall exception handler:

```
R0 = Rs
R1 = 0
```

Usage Notes

Allows User mode to call Privileged mode functions. When used in User mode, the mode is changed to Trusted mode. When used in Trusted mode, the mode is unchanged. These cases use Stack1.

Supervisor and Interrupt modes may also use syscall.r, but it will be more efficient to make a direct function call. When used in these modes, the mode is unchanged. These cases use Stack0.

Syscall.r should not be used when INFO[NL]=1 or INFO[R]=1. Such attempts will generate a Soft Reset.

The number can take values in the range 0 to 3.

Examples

```
syscall.r 0, %r1
syscall.r 3, %r6
```

32-bit Encoding

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
0	num[1:0]			Rs				1	0	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

ver — Read XAP4 version number

Instruction `ver Rd`

Description Read the XAP4's version number

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation `{R(d+1), Rd} = XAP4 version number`

Usage Notes This instruction reads the XAP4 version number.

The format of this number is:

Bits	Meaning
31:24	Reserved. 0 for this processor.
23:20	XAP Architecture major number. 4 for this processor.
19:16	XAP Architecture minor number. 2 for this processor.
15:12	Hardware Type. 0: XAP4a 1: XAP4b
11:8	Reserved. 0 for this processor.
7:0	Hardware Edition within the XAP Architecture.

Examples

```
ver %r1
```

32-bit Encoding

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
Rd			1	0	0	0	1	1	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

xor.1.i XOR (exclusive-or), single-bit, immediate

Instruction `xor.1.i %flags[c], %flags[c], #1`

Description Toggle the C flag.

Flags

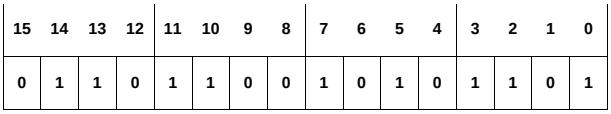
Z	-
N	-
C	Set to the inverse of the original value.
V	-

Operation `FLAGS[C] = FLAGS[C] ^ 1`

Usage Notes –

Examples `xor.1.i %flags[c], %flags[c], #1`

16-bit Encoding



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

xor.1.r XOR(exclusive-or), single-bit, register

Instruction `or.1.r %flags[c], %flags[c], Rs`

Description Bitwise XOR of carry-flag with Rs[0]

Flags

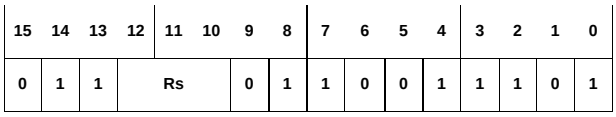
Z	Unchanged
N	Unchanged
C	Set to the result of a bitwise XOR of original value and Rs[0]
V	Unchanged

Operation `FLAGS[C] = FLAGS[C] ^ Rs[0]`

Usage Notes Used to alter the value of the carry flag.

Examples `xor.1.r %flags[c], %flags[c], %r1`

16-bit Encoding



xor.i XOR (exclusive-or), immediate

Instruction `xor.i Rd, Rs, #immediate`

Description Exclusive-OR with an immediate[15:0]u value

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `Rd = Rs ^ #immediate[15:0]u`

Usage Notes –

Examples

```
xor.i %r1, %r2, #0x1234
xor.i %r1, %r2, #0xFEDC
```

32-bit Encoding

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
#immediate[15:0]																Rd				Rs				0	1	0	0	0	1	0	1	1	1

16-bit Encoding - 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	Rd			Rs			0	0	1	1

The following rule applies:

3. The immediate is 0x1.

16-bit Encoding - 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	Rd			Rs			1	0	1	1

The following rule applies:

4. The immediate is 0xFFFF.

xor.r — XOR (exclusive-or), register

Instruction `xor.r Rd, Rs, Rt`

Description Bitwise XOR (exclusive-or) of two registers

Flags

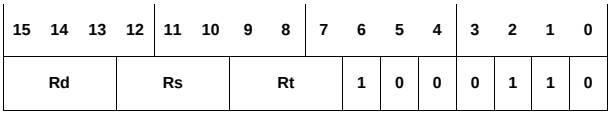
Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `Rd = Rs ^ Rt`

Usage Notes -

Examples `xor.r %r1, %r2, %r3`

16-bit Encoding



Index