Cambridge Consultants

XAP6

Programmer's Manual

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

+44 (0) 1223 420024

xap@CambridgeConsultants.com

www.CambridgeConsultants.com

C7920-UM-002 v0.8
1 September 2014

Revision History

Version	Date	Details
0.2	17 August 2013	Copied from C7432-UM-002 v1.46
0.3	26 August 2013	Should conform to C7920-S-001 v3.13 Converted a lot from XAP4 to XAP6. Still more to do!
0.4	15 November 2013	Section 7 mostly updated for XAP6
0.5	15 November 2013	Section 7 complete
0.7	8 May 2014	Sections 1-6 updated
0.8	1 September 2014	Minor additions and corrections to sections 1-6.

Legal Notices

This is the Programmer's Manual for the XAP6 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 XAP6 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 XAP6 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 XAP6 processor (and associated technologies including SIF and xIDE), XAP6 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 XAP6 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 XAP6 processor.

Trademarks: Cambridge Consultants, the Cambridge Consultants logo and XAP are registered trademarks and XAP1, XAP2, XAP3, XAP4, XAP5, XAP6, 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 XAP6 project team: Alistair Morfey, Peter Lloyd, Alan Egan, Eric Wieser, Karl Swepson, Chris Roberts, Saajan Chana, 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 XAP6 systems. It contains all the information needed to understand the XAP6 architecture from a software perspective.

This document conforms to XAP Architecture 6.0.

Further documents describe other aspects of XAP6 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 XAP6

The XAP6 is a powerful 32-bit processor optimised for low gate count and low power. It has a von Neumann architecture and can address 4GByte of linear byte-addressable memory.

High Code Density

The XAP6 architecture, instruction set, compiler toolchain and application binary interface are all designed for efficient execution of programs written in C.

Complex Systems

The XAP6 implements the features necessary to support complex software systems where high reliability or high integrity is a requirement. The XAP6 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 XAP6 MMU implements memory protection systems for instruction and data, throwing exceptions if processes access code or data memory illegally.

Interrupts and Exceptions

The XAP6 supports 32 exceptions and 32 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, 32-bit and 48-bit Instructions

The XAP6 instruction set has over 140 instructions. Most common instructions are 16 bits long with less common instructions using 32 and 48 bits. Some instructions have 16-bit, 32-bit and 48-bit forms, with the toolchain using the smaller forms whenever possible.

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

Hardware Acceleration

The processor includes a fast multiplier, divider and barrel shifter. Instruction execution is pipelined, resulting in operating speeds of over 200MHz on a 90nm ASIC process.

Targeted for Speed or Size

XAP6a is targeted for low cost applications and requires approximately 30,000 gates. It uses a 3-stage pipeline.

Language Support

The XAP6 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 XAP6 developers, using an industry-standard look and feel.

1.4 The XAP Family

The XAP6 is the sixth member of the Cambridge Consultants XAP family of embedded processors for ASICs and FPGAs. All six 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 XAP4 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 16-bit von Neumann processor and fits in approximately 12k gates. 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 18k gates. The XAP5 has a GCC Compiler.

XAP6 is a 32-bit processor that maintains the same strategy as XAP4 and XAP5. XAP6 can address 4 GB of memory and requires approximately 30k gates. XAP6 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 XAP6 C7066-TM-027	Describes specific features of the XAP6 plugin for xIDE.
XAP6 Binutils Manual C7920-UM-003	Describes the XAP6 Assembler, Linker and other binary utilities.

XAP6 GCC Manual C7920-UM-004	Describes the XAP6 GCC Compiler and support library.
XAP6a Instruction Set Quick Reference Card C7920-UM-005	A short guide to the XAP6 instruction set.

For details of the XAP6 hardware, refer to these documents:

Document	Contains
XAP6a Hardware Reference Manual C7920-UM-006	Information needed by digital designers using the XAP6a in an FPGA or ASIC
XAP6 Datasheet ASICs-SB-017	Sales Brochure, providing an overview of the XAP6 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 XAP6 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 8-byte (or 64-bit) quantity.
Word	A 4-byte (or 32-bit) quantity. The XAP6 is a 32-bit processor.
Half-Word	A 2 byte (or 16-bit) quantity.
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 XAP6 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 XAP6 Systems

2.1 System Peripherals

The XAP6 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 XAP6. This can be extended to include models of other parts of the system, including memories and interrupts.

The XAP6 normally exists with other standard components in a system. These are:

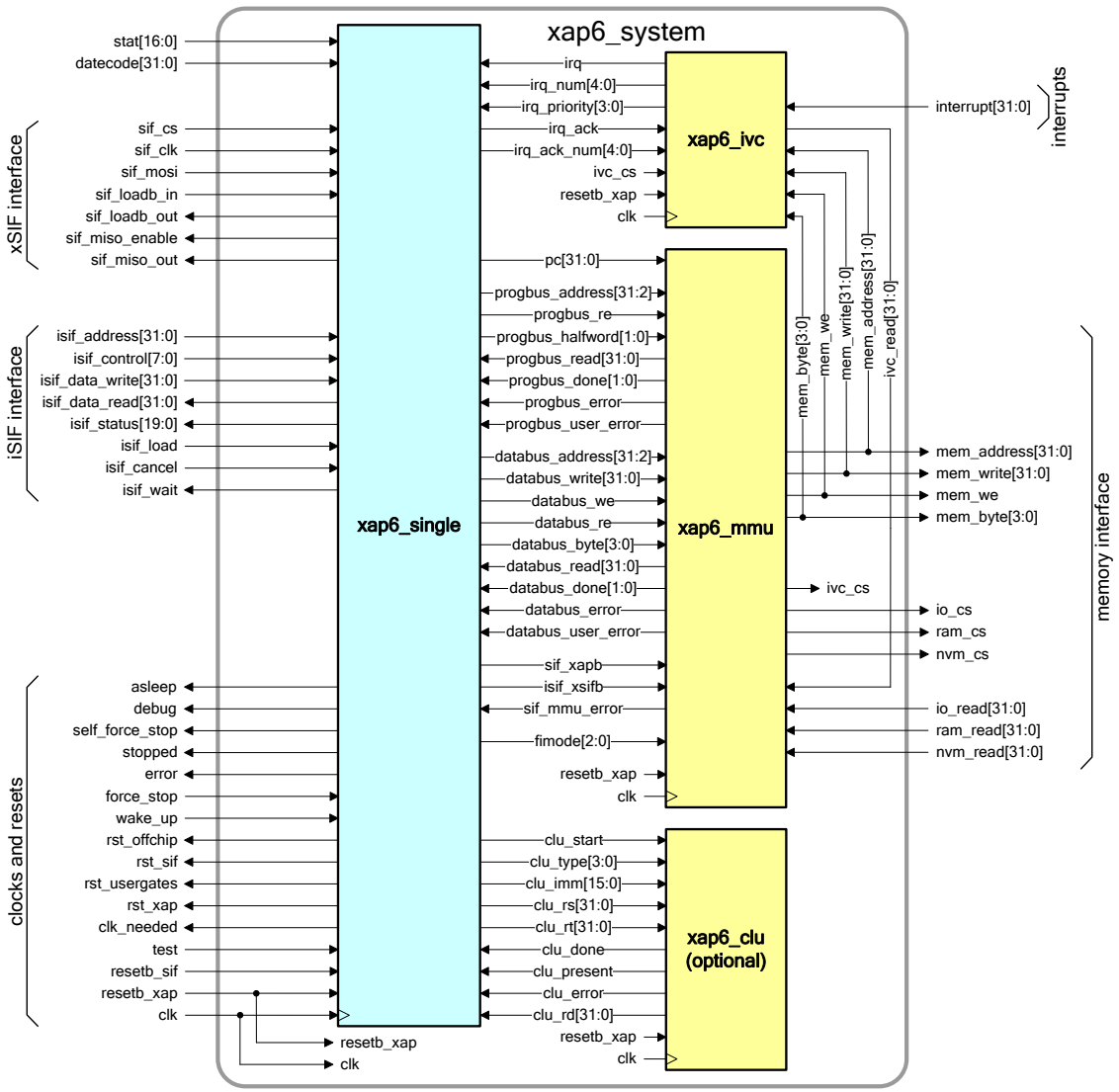
- A Memory Management Unit (MMU) to interface with memories, typically flash and RAM.
- An Interrupt Vector Controller (IVC) to prioritise interrupt sources and provide an interrupt number and priority to the XAP6.

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 XAP6 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 XAP6 can address 4GB 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 `xap6_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 XAP6 System modules.

The remainder of this document concentrates solely on the XAP6 core and SIF. Further information on the system peripherals can be found in the XAP6a Hardware Reference Manual, C7920-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 six 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.ds	#imm, Rd, Rs	3	1	1	0
clu.dst	#imm, Rd, Rs, Rt	4	1	1	1
clu.s	#imm, Rs	5	0	1	0
clu.st	#imm, Rs, Rt	6	0	1	1

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

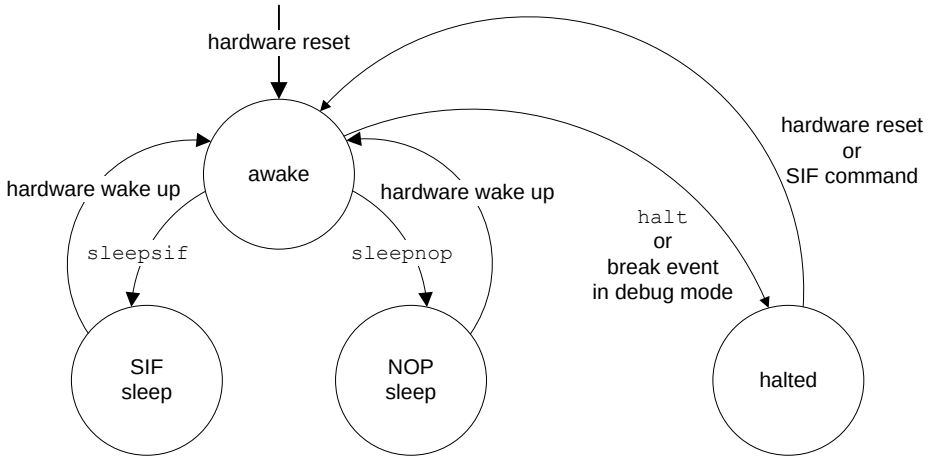
Further information about the CLU can be found in the XAP6a Hardware Reference Manual C7920-UM-006.

3 Programmer's Model

3.1 Processor Execution States

The XAP6 has four processor execution states:

Processor State	Description
Awake	The XAP6 is able to execute instructions. This is the XAP6 processor state following a reset or after a wake up from one of the sleep states.
SIF Sleep	The XAP6 is asleep but SIF cycles can still occur. This state is entered with the `sleepsif` instruction.
NOP Sleep	The XAP6 is asleep and SIF cycles are not permitted. This state is entered with the `sleepnop` instruction.
Halted	The XAP6 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 dependent 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 operating 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 XAP6 supports position-independent code and data.

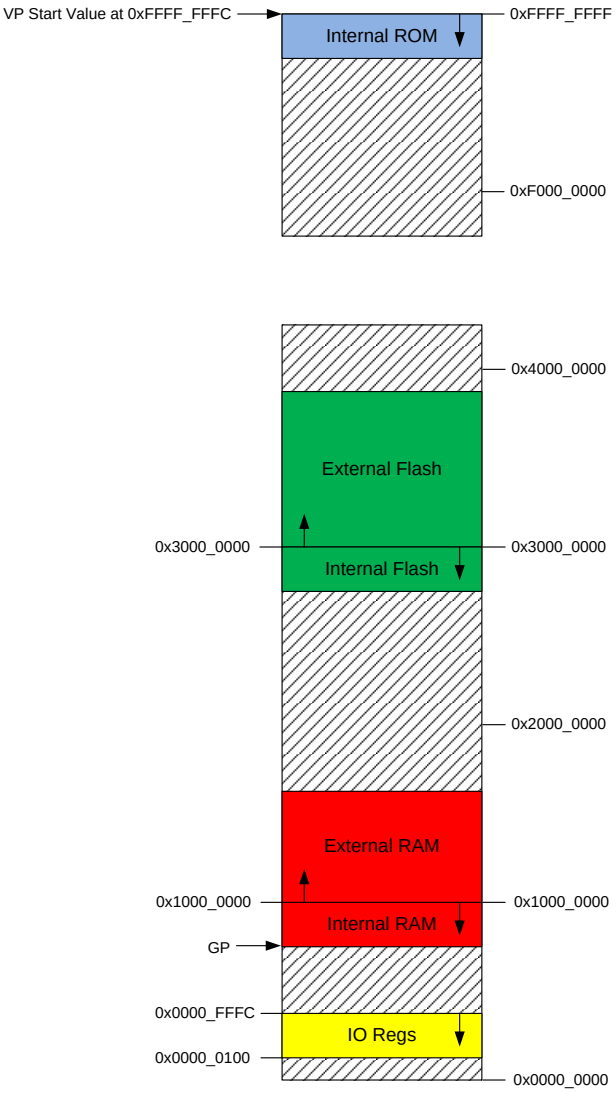
Section	Relative
Vectors	Zero-relative
Code	PC-relative
Constants	PC-relative
Global variables & heap	GP-relative
Stack	SP-relative (initial value formed zero-relative)
IO registers	Zero-relative

Global variables and the heap (stored in RAM) have their position defined at link-time, and are accessed with GP-relative addressing. IO registers are 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 figure below shows the recommended memory map.

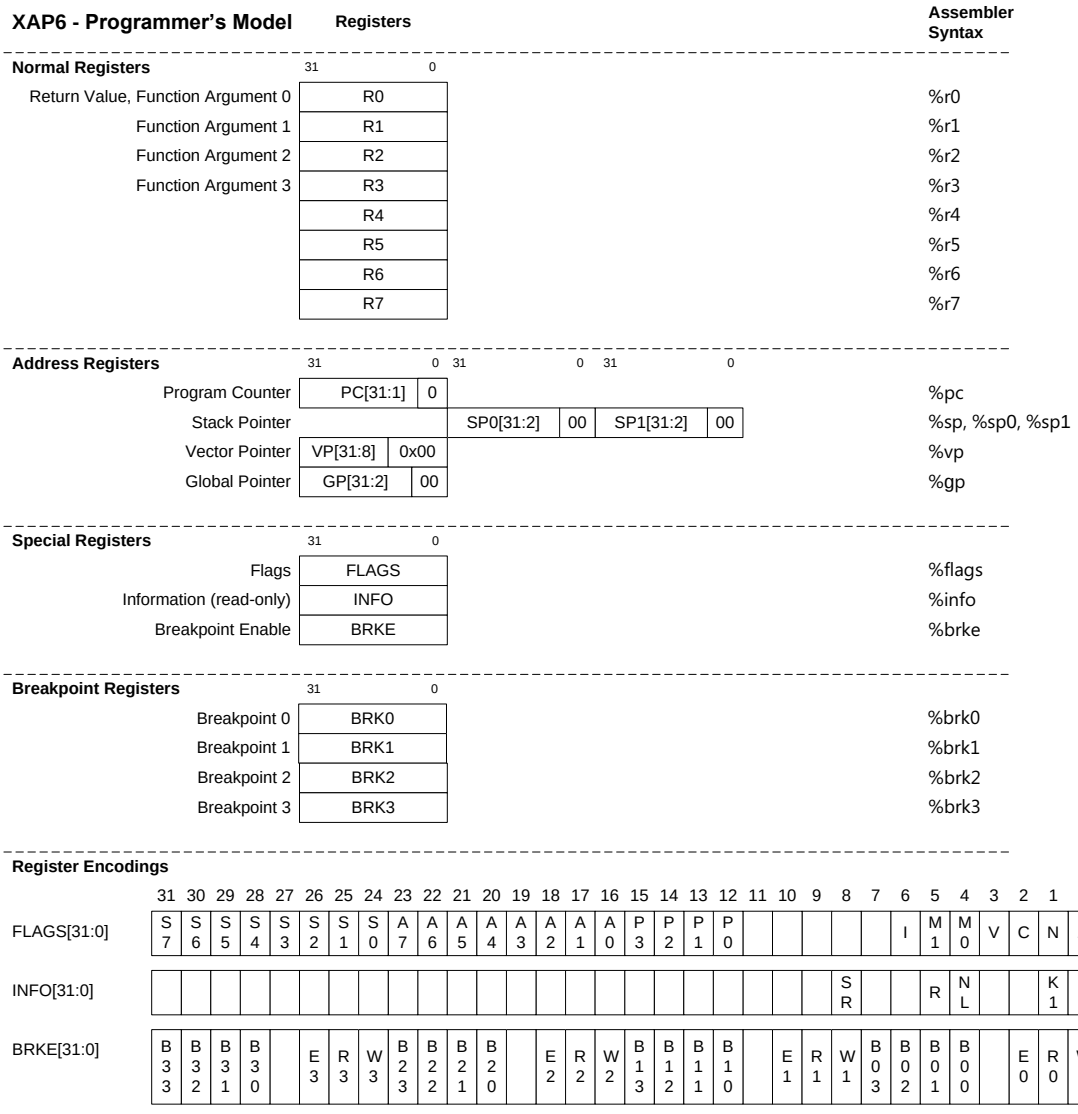
Recommended Memory Layout

- 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 XAP6 register set is shown below.



32-bit Normal registers

- Eight Normal registers, called R0 to R7.

32-bit Address registers

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

32-bit Special registers

- Processor status register, called FLAGS.
- Read-only status register, called INFO.

- Breakpoint Enable register, called BRKE.

32-bit Breakpoint registers

- Four 32-bit Breakpoint registers, called BRK0, BRK1, BRK2 and BRK3.

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 XAP6 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	

3.4.2 Address Registers

The XAP6 has five 32-bit address registers – a Program Counter (PC), two Stack Pointers (SP0 and SP1), a Global Pointer (GP) 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`.
`%gp`	GP	Should point to the base of global variables.
`%vp`	VP	The vector pointer. This can be accessed from privileged modes as `%vp`.

Program Counter

There is a single 32-bit program counter which is used by all processor modes. The program counter normally points to the next instruction to be executed. It is implicitly used by many instructions, including the PC-relative variants of the `*.i` instructions and the `rtie` instruction.

The XAP6 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 word-aligned for speed. As such, SP[1:0] is always zero.

When `%sp` is used as an operand (either implicitly 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.

Global Pointer

GP points to the base of the RAM area used for global variables.

GP is always word-aligned for speed. As such, GP[1:0] is always zero.

GP allows smaller instruction encodings to be used and also permits position independent data.

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 0xFFFF_FFFC 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[31:8] is the register.

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

3.4.3 Special registers

The XAP6 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 10 Recovery mode

Bit(s)	Flag	Name	Description
			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 Processor 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 0-3). The I bit is cleared when an interrupt or exception occurs.
11:7			FREE
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.
23:16	A[7:0]	Accumulator	Extra 8 bits above Rd[31:0]. This enables a selected single register (R0-R7) to be extended to be a 40-bit accumulator.
31:24	S[7:0]	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.

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”

Mode/state	Meaning of P[3:0]
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 will 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 3:2	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. 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, exceptions 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.
31:9			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 8 16 24	W0 W1 W2 W3	Write	These four bits enable data write breakpoints for BRK0, BRK1, BRK2 and BRK3 respectively.
1 9 17 25	R0 R1 R2 R3	Read	These four bits enable data read breakpoints for BRK0, BRK1, BRK2 and BRK3 respectively.
2 10 18 26	E0 E1 E2 E3	Execute	These four bits enable execution breakpoints for BRK0, BRK1, BRK2 and BRK3 respectively.
7:4 15:12 23:20 31:28	B0[3:0] B1[3:0] B2[3:0] B3[3:0]	Byte select	These four fields contain four byte select bits for BRK0, BRK1, BRK2 and BRK3 respectively. Read and write breakpoints are triggered when any selected byte in the address BRKn[31:2] is accessed. These bits do not affect execute breakpoints. Bn[0] represents the lowest byte address Bn[3] represents the highest byte address
3 11 19 27			FREE

3.4.4 Breakpoint Registers

The XAP6 has four 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/bits in BRKE is/are 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 XAP6 is a pipelined processor. XAP6a has a three-stage pipeline. For further details, including details of instruction cycle counts, refer to the appropriate Hardware Reference Manual.

3.6 Stack Operation

The XAP6 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 word aligned addresses. XAP6 GCC maintains a word aligned stack, where push operations decrease the stack pointer by a multiple of four bytes, and pop operations increase the stack pointer by a multiple of four bytes.

A dedicated 32-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 XAP6 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 XAP6 state is as follows:

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

The XAP6 then takes the following steps:

- Loads the initial VP value from 0xFFFFFFFC to VP.
- Loads the HardReset handler address from (VP + 0x4) 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 XAP6 system in case of an errant exception or NMI handler.

A soft reset is triggered by:

- Exceptions when the stack to be used for the context push is not initialised
- Memory errors during a context push
- Attempts to 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 XAP6 state are as follows:

- The FLAGS, BRKE and GP 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.
- BRK2 contains the value of the GP register at the time of the error.
- All other registers remain as they were before the soft reset.

The XAP6 then takes the following steps:

- Loads the initial VP value from 0xFFFFFFFC 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 Stack Pointer 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.
31:16		FREE

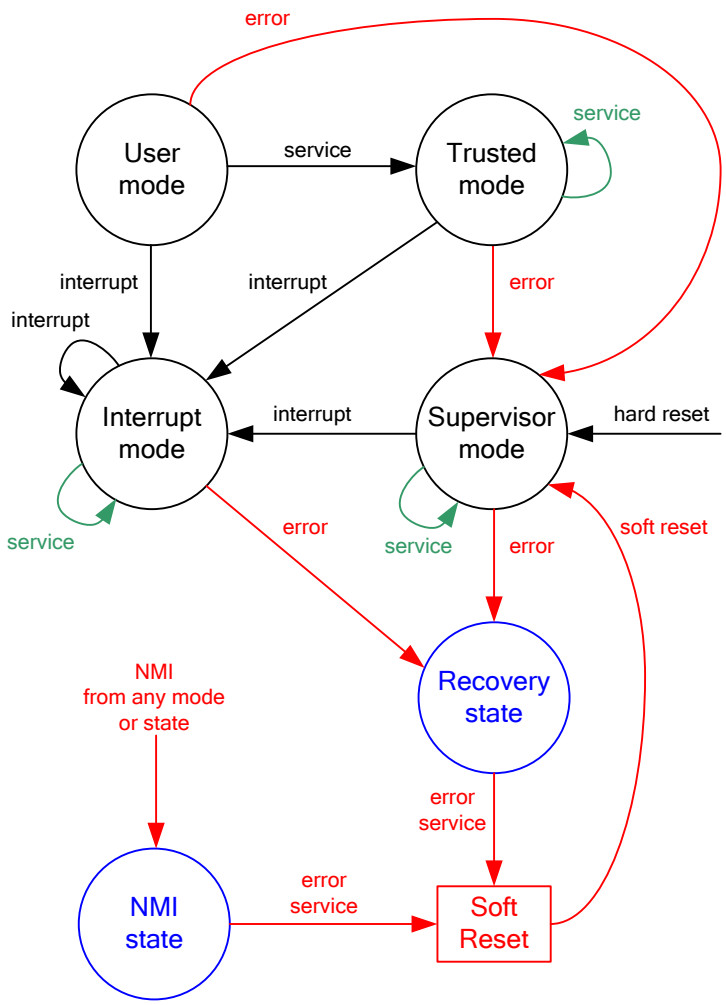
The table below shows the values used for Event Type and Event No.

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 dependent 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 XAP6 core, and can occur in any mode.

Interrupts 0 to 3 are non-maskable interrupts (NMIs), and cannot be disabled by software. Interrupts 4 to 31 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 XAP6 moves to Interrupt mode and disables interrupts. The XAP6 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 XAP6 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 XAP6 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 32 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	
0x04	Reset	1	`HardReset`	Supervisor
0x08	Reset	2	`SoftReset`	Supervisor
0x0C	Error	3	`InstructionError_S`	Supervisor
0x10	Error	4	`NullPointer_S`	Supervisor
0x14	Error	5	`DivideByZero_S`	Supervisor
0x18	Error	6	`UnknownInstruction_S`	Supervisor
0x1C	Error	7	`AlignError_S`	Supervisor
0x20	Error	8	`MMUDataError_S`	Supervisor
0x24	Error	9	`MMUProgError_S`	Supervisor
0x28	Error	10	`MMUUserDataError_S`	Supervisor
0x2C	Error	11	`MMUUserProgError_S`	Supervisor
0x30	Service	12	`SysCall0_T`	Supervisor
0x34	Service	13	`SysCall1_T`	Supervisor
0x38	Service	14	`SysCall2_T`	Supervisor
0x3C	Service	15	`SysCall3_T`	Supervisor
0x40	Service	16	`SingleStep_T`	Supervisor
0x44	Service	17	`Break_T`	Trusted
0x48	Error	18	`PrivInstruction_S`	Supervisor
0x4C	Error	19	`InstructionError_R`	Recovery
0x50	Error	20	`NullPointer_R`	Recovery
0x54	Error	21	`DivideByZero_R`	Recovery
0x58	Error	22	`UnknownInstruction_R`	Recovery
0x5C	Error	23	`AlignError_R`	Recovery
0x60	Error	24	`MMUDataError_R`	Recovery

0x64	Error	25	MMUProgError_R	Recovery
0x68		26	Reserved	
0x6C		27	Reserved	
0x70	Service	28	SysCall0_SI	No Change
0x74	Service	29	SysCall1_SI	No Change
0x78	Service	30	SysCall2_SI	No Change
0x7C	Service	31	SysCall3_SI	No Change
0x80 – 0x8C	NMI	0-3	Int00 – Int03 (Non Maskable Interrupts)	NMI
0x90 – 0xFC	MI	3-31	Int04 – Int31 (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.

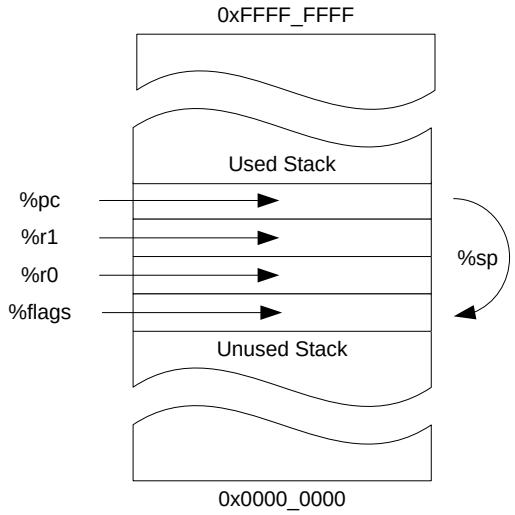
The response of the processor to services depends on the processor mode or state:

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 XAP6 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 XAP6 has a single interrupt input (`irq`), along with interrupt number (`irq_num`) and interrupt priority (`priority`) inputs. The XAP6 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 XAP6 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 == 0-3)
        {
            NMI-Now = 1;
        }

        if ((InterruptNumber == 4-31) && (FLAGS[I] == 1) &&
            (INFO[R] == 0))
        {
            MI-Now = 1;
        }
    }

    if (FLAGS[M] == Interrupt)
    {
        if (InterruptNumber == 0-3)
        {
            NMI-Now = 1;
        }

        If ((InterruptNumber == 4-31) && (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 = 32-bit value at (VP + 0x80 + (IrqNum*4))
        PC          = EventVector;
    }
}
else
{
    Hold Interrupt in pool and wait for
    MI-Now or NMI-Now conditions to be met;
}
```

Then the XAP6 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 + 0x80 + `irq_num` * 4) 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 XAP6 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 XAP6 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 = 32-bit value at (VP + (ExceptionNum * 4))
    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 Services 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 * 4) 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 XAP6 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 bit 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:

- `ld.8z.i, ld.16z.i, ld.i`
- `ld.8z.r, ld.16z.r, ld.r`
- `st.8.i, st.16.i, st.i`
- `st.8.r, st.16.r, st.r`
- `swap.i`
- `blkcp.i, blkst.8.i`
- `blkcp.r, blkst.8.r`
- `push, push.i`
- `pop, pop.ret`
- `bra.m`
- `bsr.i, bsr.m`

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 XAP6 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.* %flags[i], *`
- `mov.2.* %flags[m], *`
- `mov.4.* %flags[p], *`
- `movr2s`

- `movs2r %info, *`
- `movs2r %brke, *`
- `movr2b, movb2r`
- `movr2a, mova2r`
- `rtie`

The following instructions cause this error because they alter the behaviour of the XAP6 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, Rdr, Rdq) 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”.
- When the field specified in `mov.f.r` includes invalid bit.
- When the ranges specified in the 32-bit forms of `and.i` are invalid. Note that this should not happen if the assembler is functioning correctly.

This error can be caused by the following instructions:

- `blkcp.i, blkst.8.i`
- `blkcp.r, blkst.8.r`
- `mova2r, movr2a, movb2r, movr2b, movs2r, movr2s`
- `clu.*`
- `mov.f.r`
- `divrem.s.r, divrem.u.r`

NullPointer_S, NullPointer_R

The XAP6 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` for data accesses can be caused by the following instructions:

- `ld.8z.i, ld.16z.i, ld.i`
- `ld.8z.r, ld.16z.r, ld.r`

- st.8.i, st.16.i, st.i
- st.8.r, st.16.r, st.r
- swap.i
- blkcp.i, blkst.8.i
- blkcp.r, blkst.8.r
- push, push.i
- pop, pop.ret
- bra.m,
- bsr.i, bsr.m
- rtie

The XAP6 will also generate a `NullPointer` if it detects an attempt to branch to address zero.

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, GP and VP that should be zero.

This error is also caused when either a non-word aligned function table address or a non-word-aligned 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:

- `ld.8z.i, ld.16z.i, ld.i`
- `ld.8z.r, ld.16z.r, ld.r`
- `st.8.i, st.16.i, st.i`
- `st.8.r, st.16.r, st.r`
- `swap.i`
- `blkcp.i, blkst.8.i`
- `blkcp.r, blkst.8.r`
- `push, push.i`
- `pop, pop.ret`
- `bra.m,`
- `bsr.i, bsr.m`
- `syscall.*`
- `rtie`

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 XAP6 processes an interrupt or exception.

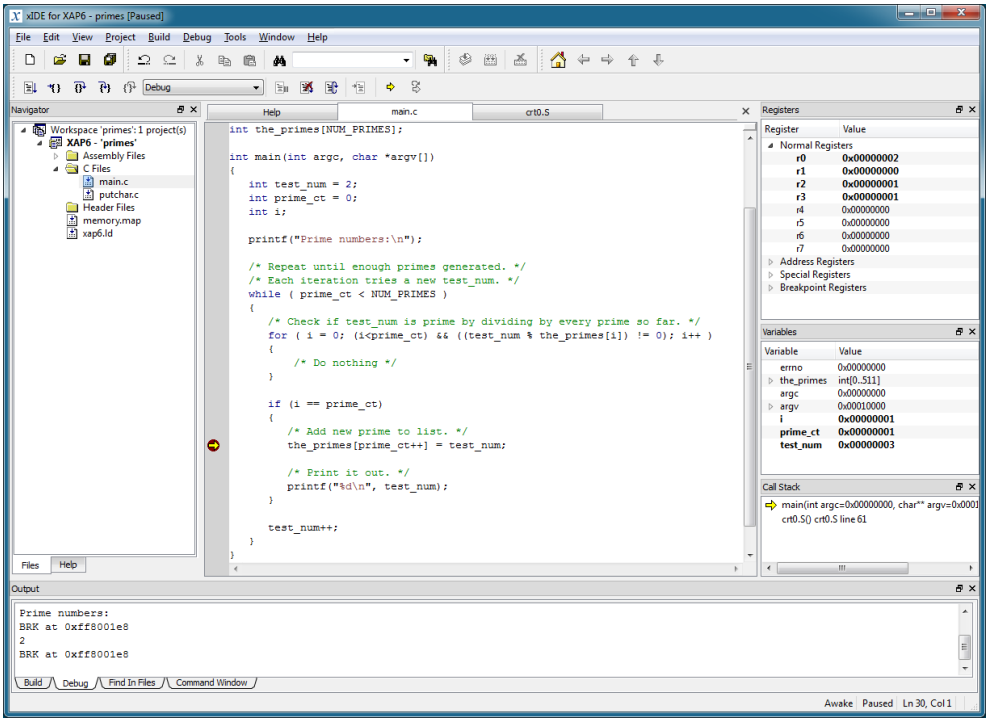
3.9 Debugging

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 XAP6 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 XAP6 is stopped when a breakpoint is encountered or whilst single stepping. This is the normal debugging situation.

Run-Mode Debugging

The XAP6 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.

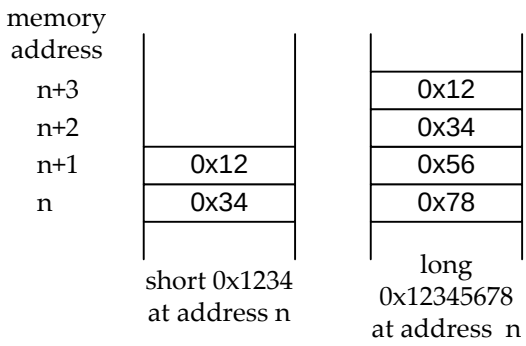
4 C Language Interface

4.1 Data Types

The XAP6 has instructions to operate on 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 XAP6 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 XAP6 GCC Manual, C7920-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 XAP6 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.

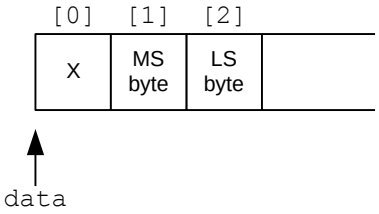
The MMU may detect accesses to unaligned data. This may provide useful debugging information, or, if speed of execution is an issue, it allows the programmer to restructure the data to remove the unaligned access. Refer to the XAP6 Hardware Reference Manual for more details.

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 XAP6.

```
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.

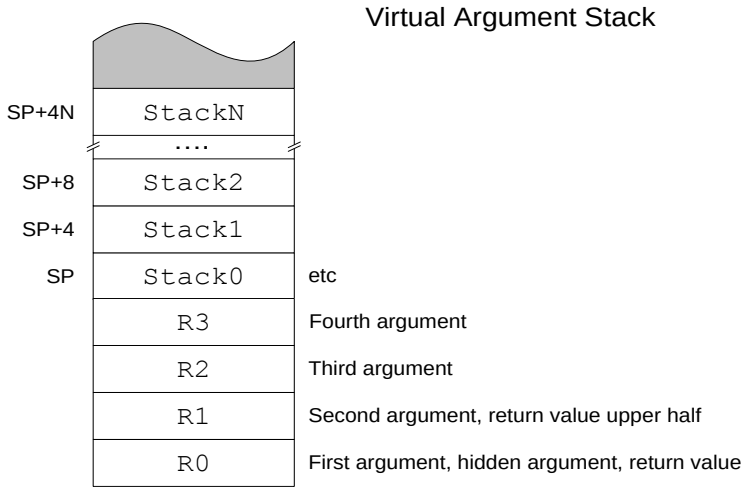
The most common direction of call is from high-level C code down to a low-level assembler routine. This section describes how a C function passes arguments into a function, and how that function must return them to the C function.

See XAP6 GCC Manual, C7920-UM-004.

Virtual Argument Stack

The XAP6 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 XAP6'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

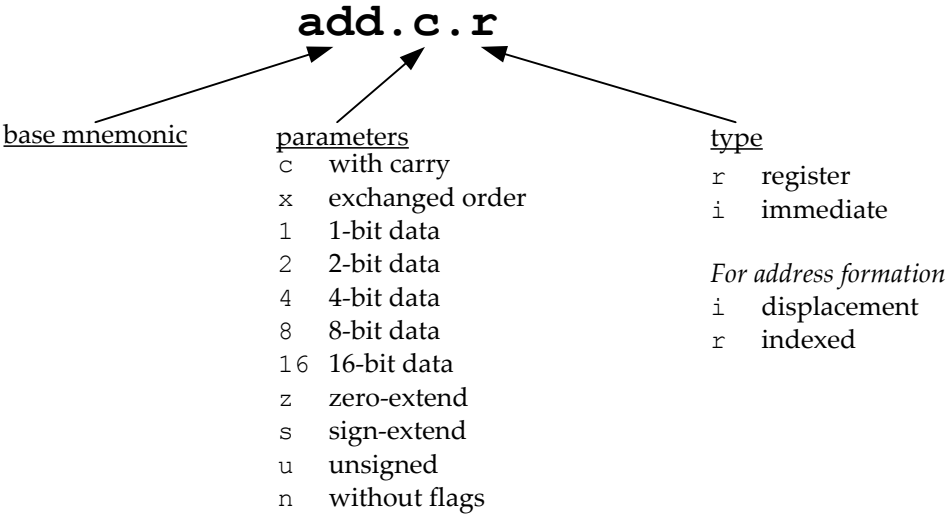
The XAP6 assembly language is case-sensitive. Instruction mnemonics, registers, number prefixes and directives must be in lower case.

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 XAP6 Binutils Manual, C7920-UM-003.

5.1.1 Instruction mnemonics

XAP6 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 32-bit data.

5.1.2 Operands

Operands are separated by commas.

5.1.3 Registers

Register names are lower case and are prefixed by `%`. XAP6 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`), GP (%gp), 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 a 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 a 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 XAP6 all immediates are treated as 32-bit numbers.

Negative numbers can be specified in hexadecimal or binary without using the unary minus operator by specifying all 32 bits in the integer.

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

Examples of valid numbers

Assembly syntax for 16-bit integers	Assembly syntax for 32-bit integers	Decimal equivalent
0x1234	0x1234	4660
0xFEDC	0xFFFFFEDC	-292
-0x1234	-0x1234	-4660
-0xFEDC	-0xFFFFFEDC	292
012	012	10
0b10	0b10	2
0b1111111111111110	0b11111111111111111111111111111110	-2
-0b10	-0b10	-2
-0b1111111111111110	-0b11111111111111111111111111111110	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 XAP6 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-2.23/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 XAP6 Binutils Manual, C7920-UM-003.

5.2 Instruction Encoding

XAP6 instructions are 48, 32 or 16 bits long. Some instructions have only one encoding size, others have multiple.

For instructions available in multiple sizes, 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 `bra.i` instructions of a known size. 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.
- `bra.i.6` forces the assembler to generate a 48-bit encoding.

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

5.3 Address Formation

The XAP6 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`, `%gp`, `%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 zero-relative and PC-relative forms 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`
- `blk*`

5.3.1 Addressing Modes

The XAP6 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 global 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, GP 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[31:0]s + Ra
`ld.i Rd, @(offset, %pc)`	offset[31:0]s + PC[31:0]
`ld.i Rd, @(offset, %sp)`	offset[31:0]u + SP
`ld.i Rd, @(offset, %gp)`	offset[31:0]u + GP
`ld.i Rd, @(offset, 0)`	offset[31:0]u + 0
`ld.i Rd, @(label, %pc)`	label[31:0]
`ld.i Rd, @(label, %gp)`	label[31:0]
`ld.i Rd, @(label, 0)`	label[31:0]
`mov.i Rd, (offset, %pc)`	offset[31:0]s + PC[31:0]
`mov.i Rd, (offset, %gp)`	offset[31:0]u + GP[31:0]

Example Instructions	Calculated memory address
`mov.i Rd, (offset, 0)`	offset[31:0]s + 0
`mov.i Rd, (label, %pc)`	label[31:0]
`mov.i Rd, (label, %gp)`	label[31:0]
`mov.i Rd, (label, 0)`	label[31:0]
`bra.i (label, %pc)`	label[31:0] (must be even)
`bra.i (offset, %pc)`	offset[31:0]s + PC[31:0] (must be even)
`bra.i (label, 0)`	label[31:0] (must be even)
`bra.i (offset, 0)`	offset[31:0]u + 0 (must be even)
`beq (label, %pc)`	label (must be even)
`beq (offset, %pc)`	offset[31:0]s + PC[31:0] (must be even)
`bra.m @(offset, %pc)`	*(offset[31:1]u + PC[31:0]) address must be word aligned, contents must be even
`bra.m @(offset, 0)`	*(offset[31:2]u + 0) address will be word aligned, contents must be event
`bra.m @(0, Ra)`	*(0 + Ra) address must be word aligned, contents 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)`	Rx + Ra
`ld.16z.r Rd, @(Rx, Ra)`	Rx + Ra
`ld.r Rd, @(Rx, Ra)`	Rx + Ra

Ra can be `%r0-%r7, %gp 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 XAP6 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, GP-relative and PC-relative addressing. 8-bit and 16-bit data is zero-extended to 32 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 semaphores 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.

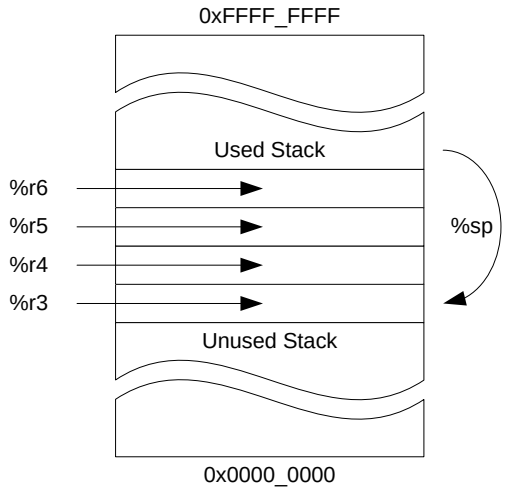
push – example 1

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

This is equivalent to the following sequence of instructions.

```
add.i     %sp, %sp, #-4     // Decrease SP for 1 register
st.i      %r6, @(0, %sp)    // Store R6 to stack memory
add.i     %sp, %sp, #-4     // Decrease SP for 1 register
st.i      %r5, @(0, %sp)    // Store R5 to stack memory
add.i     %sp, %sp, #-4     // Decrease SP for 1 register
st.i      %r4, @(0, %sp)    // Store R4 to stack memory
add.i     %sp, %sp, #-4     // Decrease SP for 1 register
st.i      %r3, @(0, %sp)    // Store R3 to stack memory
```

This is illustrated in the diagram below.



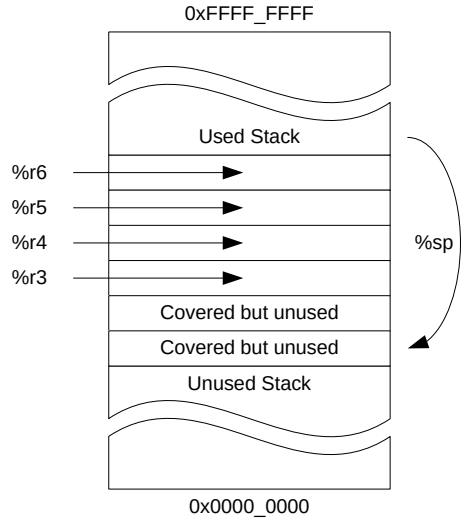
push – example 2

```
push {%r6-%r3}, #8
```

This is equivalent to the following sequence of instructions.

```
add.i     %sp, %sp, #-4     // Decrease SP for 1 register
st.i      %r6, @(0, %sp)    // Store R6 to stack memory
add.i     %sp, %sp, #-4     // Decrease SP for 1 register
st.i      %r5, @(0, %sp)    // Store R5 to stack memory
add.i     %sp, %sp, #-4     // Decrease SP for 1 register
st.i      %r4, @(0, %sp)    // Store R4 to stack memory
add.i     %sp, %sp, #-4     // Decrease SP for 1 register
st.i      %r3, @(0, %sp)    // Store R3 to stack memory
add.i     %sp, %sp, #-8     // Decrease SP by #offset[7:2]u
                            // 8 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	Any 32-bit integer
2	-32768 to 32767
3	-128 to 127
4	-128 to 127

push.i - example 1

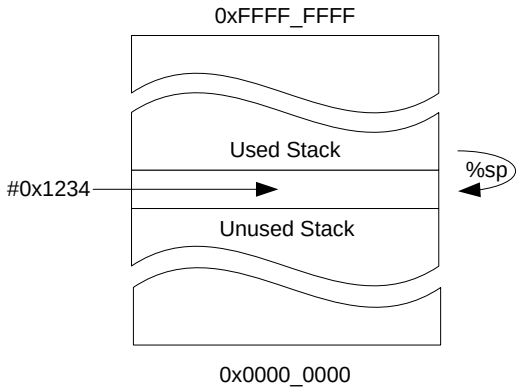
```
push.i {#0x1234}, #0
```

This is equivalent to the following sequence of instructions:

```
add.i     %sp, %sp, #-4       // 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

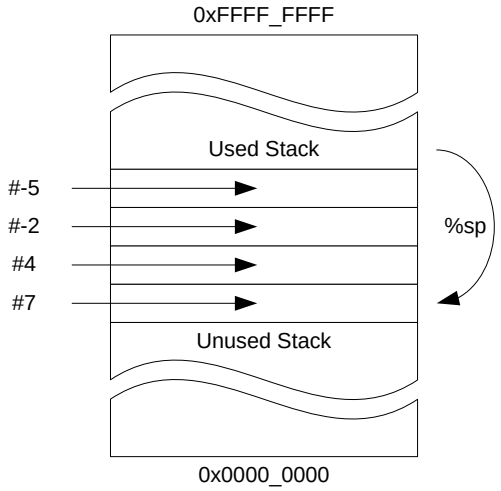
```
push.i {#-5, #-2, #4, #7}, #0
```

This is equivalent to the following sequence of instructions:

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

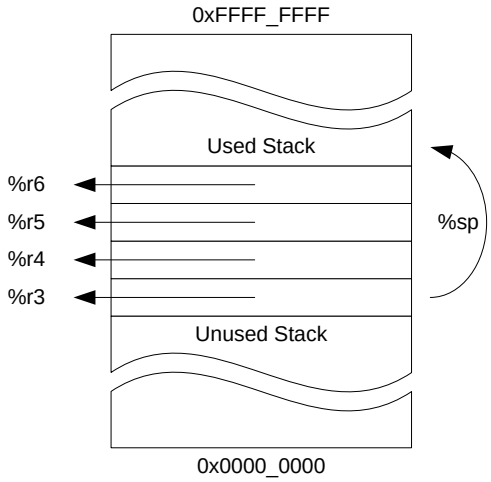
pop – example 1

```
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, #4      // Increase SP for 1 register
ld.i      %r4, @(0, %sp)    // Load R4 from stack memory
add.i     %sp, %sp, #4      // Increase SP for 1 register
ld.i      %r5, @(0, %sp)    // Load R5 from stack memory
add.i     %sp, %sp, #4      // Increase SP for 1 register
ld.i      %r6, @(0, %sp)    // Load R6 from memory
add.i     %sp, %sp, #4      // Increase SP for 1 register
```

This is illustrated in the diagram below.



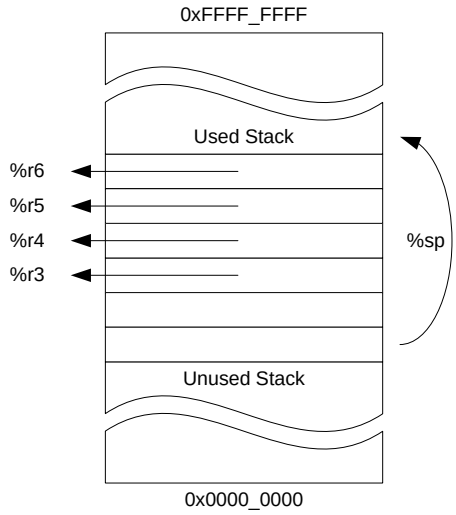
pop – example 2

```
pop {%r3-%r6}, #8
```

This is equivalent to the following sequence of instructions:

```
add.i     %sp, %sp, #8      // Increase SP by #offset[7:2]u
ld.i      %r3, @(0, %sp)    // Load R3 from stack memory
add.i     %sp, %sp, #4      // Increase SP for 1 register
ld.i      %r4, @(0, %sp)    // Load R4 from stack memory
add.i     %sp, %sp, #4      // Increase SP for 1 register
ld.i      %r5, @(0, %sp)    // Load R5 from stack memory
add.i     %sp, %sp, #4      // Increase SP for 1 register
ld.i      %r6, @(0, %sp)    // Load R6 from memory
add.i     %sp, %sp, #4      // 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.

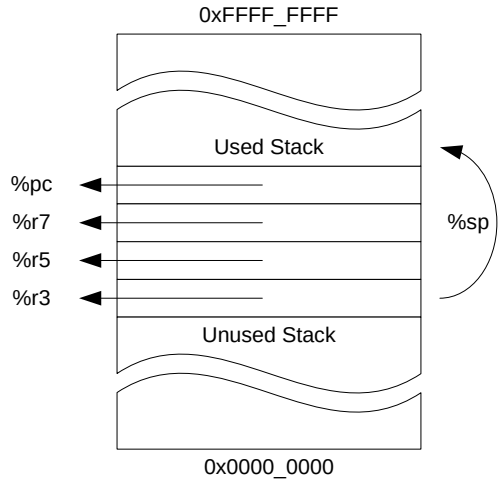
pop.ret – example 1

`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, #4       // Increase SP for 1 register
ld.i      %r5, @(0, %sp)     // Load R5 from stack memory
add.i     %sp, %sp, #4       // Increase SP for 1 register
ld.i      %r7, @(0, %sp)     // Load R7 from stack memory
add.i     %sp, %sp, #4       // 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 4
```

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



pop.ret example 2

`pop.ret {%r3, %r5, %r7}, #8`

This is equivalent to the following sequence of instructions:

```
add.i     %sp, %sp, #8       // Increase SP by 8 bytes
ld.i      %r3, @(0, %sp)     // Load R3 from stack memory
add.i     %sp, %sp, #4       // Increase SP for 1 register
ld.i      %r5, @(0, %sp)     // Load R5 from stack memory
add.i     %sp, %sp, #4       // Increase SP for 1 register
ld.i      %r7, @(0, %sp)     // Load R7 from stack memory
add.i     %sp, %sp, #4       // 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 4
```

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

6.1.4 Move

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

mov.i – address or immediate

The instruction has multiple forms. With displacement addressing, it can specify a signed offset relative to the program counter or the global pointer, or an unsigned offset relative to zero. It can also specify an immediate. The resulting 32-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.f.r – register to register

`mov.f.r` extracts a bitfield from a register, or inserts a bitfield into a register.

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

XAP6 has instructions for 16x16 and 32x32 integer multiplies giving a 32-bit result.

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 processor'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.

The 16-bit variants of the compare instructions operate on the bottom 16 bits of the operands only.

6.1.7 Shift and rotate

The XAP6 has signed (arithmetic) and unsigned (logical) shift right instructions. It also has shift left and rotate left instructions. Instructions operate on 32-bit registers.

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

The `shiftr.c.i` and `shiftl.c.i` instructions perform single bit shifts that go through the carry flag.

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.r` instruction swaps the order of the half-word in a 32-bit register. This instruction can be followed by a `rotatel` to change the endianness of a 32-bit number.

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

6.1.8 Block operations

The XAP6 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 can use a mixture of 8-bit, 16-bit and 32-bit reads and writes.

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 higher three byte of Rs to permit 16/32-bit writes to be used where possible.

6.1.9 DSP Instructions

The XAP6 has a number of powerful instructions for complex data manipulation. While not intended to be used by the C compiler these are useful assembly instructions.

Instruction	Description
`flip.*`	Two instructions useful for endianness conversion and selecting bit slices from a 32-bit input structure.
`abs.r`	Calculates the zero-relative value of a signed number
`msbit.r`	Finds the most significant set bit in the word. This can be used to simplify the process of normalisation in floating point arithmetic.

6.1.10 Miscellaneous instructions

Sign Extend

The `sext.8.r` instruction sign-extends an 8-bit value by copying the sign bit into the top 24 bits.

The `sext.16.r` instruction sign-extends an 16-bit value by copying the sign bit into the top 16 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 FLAGS[I].

`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 XAP6 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 XAP6 treat this instruction as a `nop`.

The `ver` and `lic` instructions return information about the XAP6 core.

The `fill` and `flush` instructions control the processor instruction pipeline.

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*`, `mov.2.*`, `mov.4.*` and `mov.8.*` 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.r %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 XAP6 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, #-4       // 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 bra.i.6	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.16z.i	Load, 16-bit, zero-extend, displacement	ZN--
	ld.i	Load, displacement	ZN--
Load (indexed)	ld.8z.r	Load, 8-bit, zero-extend, indexed	ZN--
	ld.16z.r	Load, 16-bit, zero-extend, indexed	ZN--
	ld.r	Load, indexed	ZN--
Store (displacement)	st.8.i	Store, 8-bit, displacement	----
	st.16.i	Store, 16-bit, displacement	----
	st.i	Store, displacement	----
Store (indexed)	st.8.r	Store, 8-bit, indexed	----
	st.18.r	Store, 16-bit, indexed	----
	st.r	Store, 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, displacement or immediate	ZN--
	mov.r	Move, register	ZN--
	mov.f.r	Move, field, register	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
	add.n.i	Add without flags, immediate	ZN--
	add.n.r	Add without flags, register	ZN--
Subtract	sub.r	Subtract, register	ZNCV
	sub.c.r	Subtract, with carry, register	ZNCV
	sub.x.i	Subtract, exchange, immediate	ZNCV
	sub.xc.i	Subtract, exchange, with carry, 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	XOR (exclusive-or), immediate	ZN--
	xor.r	XOR (exclusive-or), register	ZN--
Multiply	mult.16s.i	Multiply, 16-bit signed, immediate	ZN--
	mult.16s.r	Multiply, 16-bit signed, register	ZN--
	mult.16u.i	Multiply, 16-bit unsigned, immediate	ZN--
	mult.16u.r	Multiply, 16-bit unsigned, register	ZN--
	mult.i	Multiply, immediate	ZN--
	mult.r	Multiply, register	ZN--
Divide and Remainder (signed)	div.s.r	Divide, signed, register	ZN-V
	divrem.s.r	Divide and remainder, signed, register	---V
	rem.s.r	Remainder, signed, register	ZN-V
Divide and Remainder (unsigned)	div.u.r	Divide, unsigned, register	ZNC-
	divrem.u.r	Divide and remainder, signed, register	--C-
	rem.u.r	Remainder, unsigned, register	ZNC-

6.3.6 Compare operations

	32-bit	16-bit	8-bit
Immediate	cmp.i	cmp.16.i	cmp.8.i
Register	cmp.r	cmp.16.r	cmp.8.r
With carry, immediate	cmp.c.i	cmp.16c.i	cmp.8c.i
With carry, register	cmp.c.r	cmp.16c.r	cmp.8c.r
Exchange, immediate	cmp.x.i	cmp.16x.i	cmp.8x.i

	32-bit	16-bit	8-bit
With carry, exchange, immediate	`cmp.xc.i`	`cmp.16xc.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.c.i`	Shift left, with carry	`ZNC-`
Shift right	`shiftr.s.i`	Shift right, signed, immediate	`ZNC-`
	`shiftr.s.r`	Shift right, signed, register	`ZNC-`
	`shiftr.u.i`	Shift right, unsigned, immediate	`ZNC-`
	`shiftr.u.r`	Shift right, unsigned, register	`ZNC-`
	`shiftr.c.i`	Shift right, with carry	`ZNC-`
Rotate	`rotatel.i`	Rotate left, immediate	`ZNC-`
	`rotatel.r`	Rotate left, register	`ZNC-`
Byte swap	`b2swap.r`	Byte swap, 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	`----`

6.3.9 DSP Instructions

Function	Mnemonic	Description	Flags
Flip	`flip.r`	Flip bits	`ZN--`
	`flip.8.r`	Flip byte bits	`ZN--`
	`flip.16.r`	Flip word bits	`ZN--`
Absolute	`abs.r`	Absolute, register	`ZN--`
Find most significant bit	`msbit.r`	Most significant bit, register	`ZN--`

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.ds	CLU Instruction, type 3	ZN--
	clu.dst	CLU Instruction, type 4	ZN--
	clu.s	CLU Instruction, type 5	----
	clu.st	CLU Instruction, type 6	----

6.3.11 Miscellaneous instructions

Function	Mnemonic	Description	Flags
Sign extend	sext.8.r	Sign-extend, 8-bit, register	ZN--
	sext.16.r	Sign-extend, 16-bit, 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	----
	fill	Fill prefetch buffer	----
	flush	Flush prefetch buffer	----
	print.r	Print, register	----
	lic	Read licence number	----
	rtie	Return from interrupt/exception	ZNCV
	softreset	Soft Reset	----
	syscall.i	System call, immediate	----
	syscall.r	System call, register	----
	ver	Read 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	----

Function	Mnemonic	Description	Flags
	mov.4.i	Move, 4-bit, immediate	----
	mov.4.r	Move, 4-bit, register	----
	mov.8.i	Move, 8-bit, immediate	----
	mov.8.r	Move, 8-bit, register	----
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 XAP6 instruction set is listed alphabetically below. The "Sizes" column indicates whether the instruction can be encoded in 16 bits, 32 bits and 48 bits.

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, 48
abs.r	Rd, Rs	Absolute, register	32
add.c.i	Rd, Rs, #imm	Add with carry, immediate	48,32,16
add.c.r	Rd, Rs, Rt	Add with carry, register	16
add.i	Rd, Rs, #imm Rd, %sp, #imm %sp, %sp, #imm	Add, immediate	48,32,16
add.n.i	Rd, Rs, #imm	Add without flags, immediate	48,32
add.n.r	Rd, Rs, Rt	Add without flags, register	32
add.r	Rd, Rs, Rt	Add, register	16
and.i	Rd, Rs, #imm	AND, immediate	48,32,16
and.r	Rd, Rs, Rt	AND, register	16
b2swap.r	Rd, Rs	Byte swap, register	16
bcc	(label, %pc) (offset, %pc)	Branch if carry clear	32,16
bcs	(label, %pc) (offset, %pc)	Branch if carry set	32,16
beq	(label, %pc) (offset, %pc)	Branch if equal	32,16
bez.r	Rs, (label, %pc) Rs, (offset, %pc)	Branch if register zero	32,16
bge.s	(label, %pc) (offset, %pc)	Branch if greater than or equal, signed	32,16
bge.u	(label, %pc) (offset, %pc)	Branch if greater than or equal, unsigned	32,16
bgt.s	(label, %pc) (offset, %pc)	Branch if greater than, signed	32,16
bgt.u	(label, %pc) (offset, %pc)	Branch if greater than, unsigned	32,16

Mnemonic	Operands	Operation	Sizes
ble.s	(label, %pc) (offset, %pc)	Branch if less than or equal, signed	32,16
ble.u	(label, %pc) (offset, %pc)	Branch if less than or equal, unsigned	32,16
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 #imm, @(0, Ra), #num	Block store, 8-bit, immediate	32
blkst.8.r	Rs, @(0, Ra), Rn #imm, @(0, Ra), Rn	Block store, 8-bit, register	32
blt.s	(label, %pc) (offset, %pc)	Branch if less than, signed	32,16
blt.u	(label, %pc) (offset, %pc)	Branch if less than, unsigned	32,16
bmi	(label, %pc) (offset, %pc)	Branch if minus	32
bne	(label, %pc) (offset, %pc)	Branch if not equal	32,16
bnz.r	Rs, (label, %pc) Rs, (offset, %pc)	Branch if register not zero	32,16
bpl	(label, %pc) (offset, %pc)	Branch if plus	32
bra.i	(label, %pc) (offset, %pc) (label, 0) (offset, 0) (0, Ra)	Branch	48,32,16
bra.i.2	(label, %pc)	Branch	16
bra.i.4	(label, %pc) (label, 0)	Branch	32
bra.i.6	(label, %pc) (label, 0)	Branch	48
bra.m	@(offset, %pc) @(offset, 0) @!(offset, %pc) @!(offset, 0) @(0, Ra)	Branch, via memory, displacement	48,32,16
brk		Break	16
bsr.i	(label, %pc) (offset, %pc) (label, 0) (offset, 0)	Branch to subroutine	48,32,16

Mnemonic	Operands	Operation	Sizes
	(0, Ra)		
bsr.m	@(offset, %pc) @(offset, 0) @!(offset, %pc) @!(offset, 0) @(0, Ra)	Branch to subroutine, via memory, displacement	48,32,16
bvc	(label, %pc) (offset, %pc)	Branch if overflow clear	32
bvs	(label, %pc) (offset, %pc)	Branch if overflow set	32
clu	#imm	CLU Instruction, type 1	32
clu.d	#imm, Rd	CLU Instruction, type 2	32
clu.ds	#imm, Rd, Rs	CLU Instruction, type 3	32
clu.dst	#imm, Rd, Rs, Rt	CLU Instruction, type 4	32
clu.s	#imm, Rs	CLU Instruction, type 5	32
clu.st	#imm, Rs, Rt	CLU Instruction, type 6	32
cmp.16.i	Rs, #imm	Compare, 16-bit, immediate	32,16
cmp.16.r	Rs, Rt	Compare, 16-bit, register	16
cmp.16c.i	Rs, #imm	Compare, 16-bit with carry, immediate	32
cmp.16c.r	Rs, Rt	Compare, 16-bit with carry, register	16
cmp.16x.i	Rs, #imm	Compare, 16-bit, exchange, immediate	32
cmp.16xc.i	Rs, #imm	Compare, 16-bit carry, exchange, immediate	32
cmp.8.i	Rs, #imm	Compare, 8-bit, immediate	32,16
cmp.8.r	Rs, Rt	Compare, 8-bit, register	16
cmp.8c.i	Rs, #imm	Compare, 8-bit with carry, immediate	32
cmp.8c.r	Rs, Rt	Compare, 8-bit with carry, register	16
cmp.8x.i	Rs, #imm	Compare, 8-bit, exchange, immediate	32
cmp.8xc.i	Rs, #imm	Compare, 8-bit carry, exchange, immediate	32
cmp.c.i	Rs, #imm	Compare, with carry, immediate	48,32
cmp.c.r	Rs, Rt	Compare, with carry, register	16

Mnemonic	Operands	Operation	Sizes
cmp.i	Rs, #imm	Compare, immediate	48,32,16
cmp.r	Rs, Rt	Compare, register	16
cmp.x.i	Rs, #imm	Compare, exchange, immediate	48,32
cmp.xc.i	Rs, #imm	Compare, with carry, exchange, immediate	48,32
div.s.r	Rd, Rs, Rt	Divide, 32-bit signed, register	32
div.u.r	Rd, Rs, Rt	Divide, unsigned, register	32
divrem.s.r	Rdr, Rdq, Rs, Rt	Divide and remainder, signed, register	32
divrem.u.r	Rdr, Rdq, Rs, Rt	Divide and remainder, unsigned, register	32
fill		Fill prefetch buffer	16
fimode	Rd	Flags and info mode	16
flip.16.r	Rd, Rs	Flip word bits	32
flip.8.r	Rd, Rs	Flip byte bits	32
flip.r	Rd, Rs	Flip bits	32
flush		Flush prefetch buffer	16
halt		Halt	16
ld.16z.i	Rd, @(offset, Ra) Rd, @(offset, %pc) Rd, @(offset, %sp) Rd, @(offset, %gp) Rd, @(offset, 0) Rd, @(label, %pc) Rd, @(label, %gp) Rd, @(label, 0)	Load, 16-bit, zero-extend, displacement	48,32,16
ld.16z.r	Rd, @(Rx, Ra) Rd, @(Rx, %sp)	Load, 16-bit, zero-extend, indexed	32,16
ld.8z.i	Rd, @(offset, Ra) Rd, @(offset, %pc) Rd, @(offset, %sp) Rd, @(offset, %gp) Rd, @(offset, 0) Rd, @(label, %pc) Rd, @(label, %gp) Rd, @(label, 0)	Load, 8-bit, zero-extend, displacement	48,32,16
ld.8z.r	Rd, @(Rx, Ra) Rd, @(Rx, %sp)	Load, 8-bit, zero-extend, indexed	32,16

Mnemonic	Operands	Operation	Sizes
ld.i	Rd, @(offset, Ra) Rd, @(offset, %pc) Rd, @(offset, %sp) Rd, @(offset, %gp) Rd, @(offset, 0) Rd, @(label, %pc) Rd, @(label, %gp) Rd, @(label, 0)	Load, displacement	48,32,16
ld.r	Rd, @(Rx, Ra) Rd, @(Rx, %sp)	Load, Indexed	32,16
lic	Rd	Read licence number	32
mov.1.i	%flags[i], #imm	Move, single-bit, immediate	16
mov.1.r	Rd, %flags[ZNCV] Rd, %flags[i] %flags[ZNCV], Rs %flags[i], Rs %flags[i], %flags[c] %flags[c], %flags[i]	Move, single-bit, register	32,16
mov.2.i	%flags[m], #imm	Move, 2-bit, immediate	16
mov.2.r	Rd, %flags[m] %flags[m], Rs	Move, 2-bit, register	16
mov.4.i	%flags[p], #imm	Move, 4-bit, immediate	32
mov.4.r	Rd, %flags[p] %flags[p], Rs	Move, 4-bit, register	32
mov.8.i	%flags[a], #imm	Move, 8-bit, immediate	32
mov.8.r	Rd, %flags[a] %flags[a], Rs	Move, 8-bit, register	32
mov.f.r	Rd[msb:lsb], Rs Rd, Rs[msb:lsb]	Move, field, register	32,16
mov.i	Rd, (offset, %pc) Rd, (offset, %gp) Rd, (imm, 0) Rd, #imm Rd, (label, %pc) Rd, (label, %gp) Rd, (label, 0) Rd, !(label, %pc) Rd, !(label, 0)	Move, displacement or immediate	48,32,16
mov.r	Rd, Rs %sp, Rs Rd, %sp	Move, register	32,16
mova2r	Rd, As	Move address register to register	32
movb2r	Rd, Bs	Move breakpoint register to register	32

Mnemonic	Operands	Operation	Sizes
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
msbit.r	Rd, Rs	Most significant bit, register	32
mult.16s.i	Rd, Rs, #imm	Multiply, 16-bit signed, immediate	32
mult.16s.r	Rd, Rs, Rt	Multiply, 16-bit signed, register	32
mult.16u.i	Rd, Rs, #imm	Multiply, 16-bit unsigned, immediate	32
mult.16u.r	Rd, Rs, Rt	Multiply, 16-bit unsigned, register	32
mult.i	Rd, Rs, #imm	Multiply, immediate	48,32
mult.r	Rd, Rs, Rt	Multiply, register	32
nop		No operation	16
or.i	Rd, Rs, #imm	OR, immediate	48,32,16
or.r	Rd, Rs, Rt	OR, register	16
pop	RegList, #offset	Pop from stack	32,16
pop.ret	RegList, #offse	Pop from stack and return	32,16
print.r	Rs	Print, register	32
push	RegList, #offset	Push to stack	32,16
push.i	{#i0}, #0 {#i0, #i1}, #0 {#i0, #i1, #i2}, #0 {#i0, #i1, #i2, #i3}, #0	Push immediates to stack	48,32,16
rem.s.r	Rd, Rs, Rt	Remainder, signed, register	32
rem.u.r	Rd, Rs, Rt	Remainder, unsigned, register	32
rotatel.i	Rd, Rs, #imm	Rotate left, immediate	32,16
rotatel.r	Rd, Rs, Rt	Rotate left, register	32
rtie		Return from interrupt/exception	16
sext.16.r	Rd, Rs	Sign extend, 16-bit, register	16
sext.8.r	Rd, Rs	Sign extend, 8-bit, register	16
shiftl.c.i	Rd, Rs, #1	Shift left, with carry	16
shiftl.i	Rd, Rs, #imm	Shift left, immediate	32,16

Mnemonic	Operands	Operation	Sizes
shiftl.r	Rd, Rs, Rt	Shift left, register	16
shiftr.c.i	Rd, Rs, #1	Shift right, with carry	16
shiftr.s.i	Rd, Rs, #imm	Shift right, signed, immediate	32,16
shiftr.s.r	Rd, Rs, Rt	Shift right, signed, register	16
shiftr.u.i	Rd, Rs, #imm	Shift right, unsigned, immediate	32,16
shiftr.u.r	Rd, Rs, Rt	Shift right, unsigned, register	16
sif		SIF	16
sleepnop		Sleep	16
sleepsif		Sleep and allow SIF	16
softreset		Soft Reset	16
st.16.i	Rm, @(offset, Ra) Rm, @(offset, %pc) Rm, @(offset, %sp) Rm, @(offset, %gp) Rm, @(offset, 0) Rm, @(label, %pc) Rm, @(label, %gp) Rm, @(label, 0)	Store, 16-bit, displacement	48,32,16
st.16.r	Rm, @(Rx, Ra) Rm, @(Rx, %sp)	Store, 16-bit, indexed	32,16
st.8.i	Rm, @(offset, Ra) Rm, @(offset, %pc) Rm, @(offset, %sp) Rm, @(offset, %gp) Rm, @(offset, 0) Rm, @(label, %pc) Rm, @(label, %gp) Rm, @(label, 0)	Store, 8-bit, displacement	48,32,16
st.8.r	Rm, @(Rx, Ra) Rm, @(Rx, %sp)	Store, 8-bit, indexed	32,16
st.i	Rm, @(offset, Ra) Rm, @(offset, %pc) Rm, @(offset, %sp) Rm, @(offset, %gp) Rm, @(offset, 0) Rm, @(label, %pc) Rm, @(label, %gp) Rm, @(label, 0)	Store, displacement	48,32,16
st.r	Rm, @(Rx, Ra) Rm, @(Rx, %sp)	Store, indexed	32,16
sub.c.r	Rd, Rs, Rt	Subtract, with carry, register	16

Mnemonic	Operands	Operation	Sizes
sub.r	Rd, Rs, Rt	Subtract, register	16
sub.x.i	Rd, Rs, #imm	Subtract, exchange, immediate	48,32,16
sub.xc.i	Rd, Rs, #imm	Subtract, exchange, with carry, immediate	48,32,16
swap.i	Rd, @(0, Ra)	Swap register with memory	32
syscall.i	num, #imm	System call, immediate	48,32
syscall.r	num, Rs	System call, register	32
ver	Rd	Read version number	32
xor.i	Rd, Rs, #imm	XOR (exclusive-or), immediate	48,32,16
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.* %flags[i], *`
- `mov.2.* %flags[m], *`
- `mov.4.* %flags[p], *`
- `movr2s`
- `movs2r %info, *`
- `movs2r %brke, *`
- `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 XAP6 core.

If a privileged instruction is executed in User mode, the XAP6 throws a `PrivInstruction_S` exception.

6.6 Aliased Instructions

The assembler translates a small number of instructions into other, equivalent instructions. 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)
`bra.i.6`	`bra.i` (48-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.
Rdr	Destination register for remainder in `divrem.*`.
Rdq	Destination register for quotient in `divrem.*`.
Rs	Primary source register.
Rt	Secondary source register.
Rm	Source register for stores to memory.
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. 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	`%gp`
100-111	reserved

Special register fields

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

Sx	register
000	`%flags`

Sx	register
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	%brk2
011	%brk3
100 - 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 32-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 just a few bits wide. In some of these instructions, the XAP6 extends the immediate to 32 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.

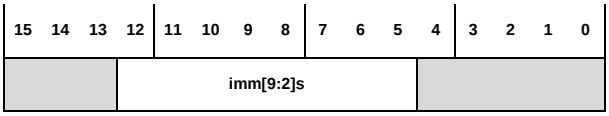
If the assembler cannot represent the immediate in the shortened form, it selects the equivalent 32-bit or 48-bit encoding of the instruction.

Implicit bits

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

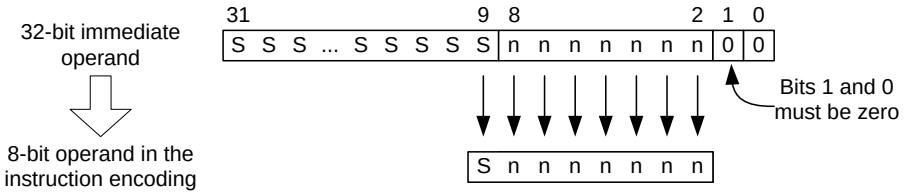
Example

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 9 to 2 of the immediate are encoded in the instruction. Bits 0 and 1 are an implicit 0.

The "s" suffix indicates that bits 31 to 10 are sign-extended from bit 9 before use in the ALU.



This encoding can therefore represent immediates in the range -512, -508, …, +504, +508. Immediates outside this range require an alternative encoding of the instruction.

7 Instruction Set Reference

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

abs.r — Absolute, register

Instruction

```
abs.r Rd, Rs
```

Description

Put the absolute value of Rs into Rd

Flags

Z	Set if the result is zero; cleared otherwise
N	Cleared
C	Unchanged
V	Set if the input is 0x8000 0000 (result = 0x8000 0000), cleared otherwise

Operation

```
if (Rs ≥ 0)
    Rd = Rs;
else
    Rd = -Rs;
```

Usage Notes

This instruction calculates the absolute value of a signed 32-bit number.

Examples

```
abs.r %r4, %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	0	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0	0	0	Rs				Rd				0	0	0	0	1	1	1

add.c.i — Add with carry, immediate

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

Description Add immediate[31:0]s and the carry flag 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 + #imm[31:0]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 64-bit arithmetic.

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

48-bit Encoding

47 ... 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
imm[31:0]s	Rd	Rs 1 0	1 1 0 1	1 1 1 1

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
imm[15:0]s				Rd	Rs 1 0	1 1 0 1	0 1 1 1

16-bit Encodings

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

This encoding is only valid when Rs = Rd

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

This encodes the instruction `add.c.i Rd, Rs, #0`

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

This encodes the instruction `add.c.i Rd, Rs, #1`

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

This encodes the instruction `add.c.i Rd, Rs, #0xFFFFFFFF`

add.c.r — Add with carry, register

Instruction

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

Description

Add two registers and the carry flag

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

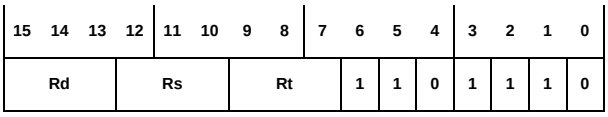
`add.c.r` can be used for signed or unsigned, integer or fixed-point arithmetic.

The Z flag behaviour is useful for 64-bit arithmetic.

Examples

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

16-bit Encoding



add.i — Add, immediate

Instruction

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

Description

Add a register and an immediate

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[31:0]s
Rd = SP + #immediate[31:0]s
SP = SP + #immediate[31:0]s
```

Usage Notes

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

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. In this case the flags are unchanged.

Examples

```
add.i %r1, %r2, #0x12345678
add.i %r1, %r2, #0xFEDC
```

48-bit Encodings

47 ... 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
imm[31:0]s	Rd	Rs 1 0	0 1 0 1	1 1 1 1

This encodes the instruction `add.i Rd, Rs, #imm`

47 ... 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
imm[31:0]s	Rd 0	1 1 1 0	0 1 1 1	1 1 1 1

This encodes the instruction `add.i Rd, %sp, #imm`

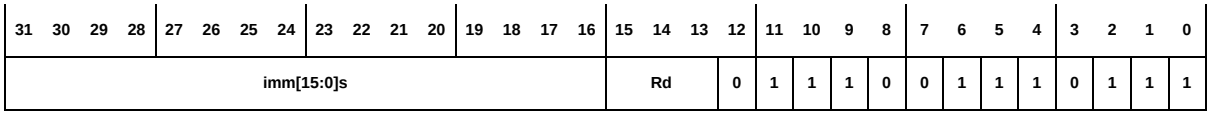
47 ... 18	17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
imm[31:2]s	0 0	1 0 0 1	0 0 1 0	0 1 1 1	1 1 1 1

This encodes the instruction `add.i %sp, %sp, #imm`

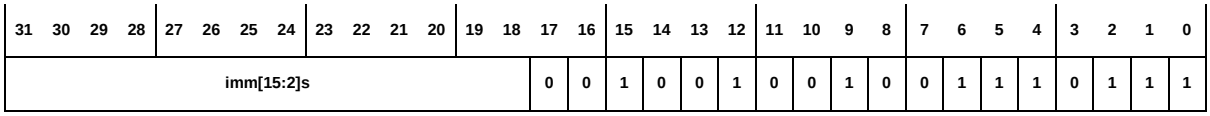
32-bit Encodings

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
imm[15:0]s				Rd	Rs 1 0	0 1 0 1	0 1 1 1

This encodes the instruction `add.i Rd, Rs, #imm`

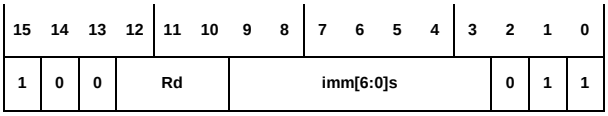


This encodes the instruction `add.i Rd, %sp, #imm`



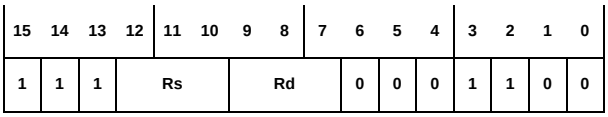
This encodes the instruction `add.i %sp, %sp, #imm`

16-bit Encodings

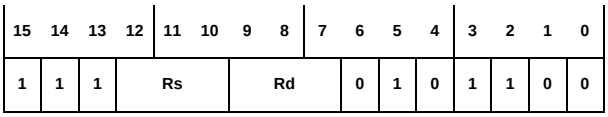


This encodes the instruction `add.i Rd, Rs, #imm`

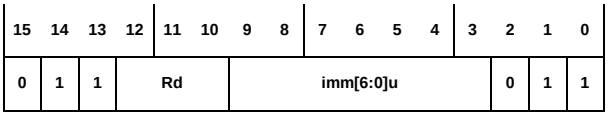
This encoding is only valid when Rs = Rd



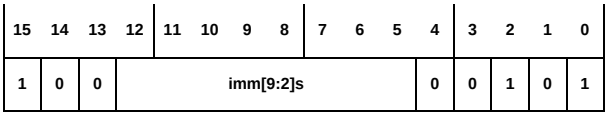
This encodes the instruction `add.i Rd, Rs, #1`



This encodes the instruction `add.i Rd, Rs, #0xFFFFFFFF`



This encodes the instruction `add.i Rd, %sp, #imm`



This encodes the instruction `add.i %sp, %sp, #imm`

add.n.i — Add without flags, immediate

Instruction

```
add.n.i Rd, Rs, #imm
```

Description

Add a register and an immediate . This instruction does not alter %flags[c] and %flags[v].

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[31:0]s
```

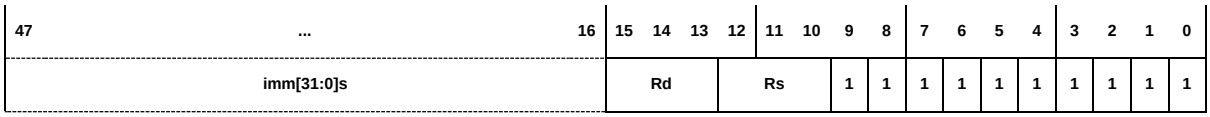
Usage Notes

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

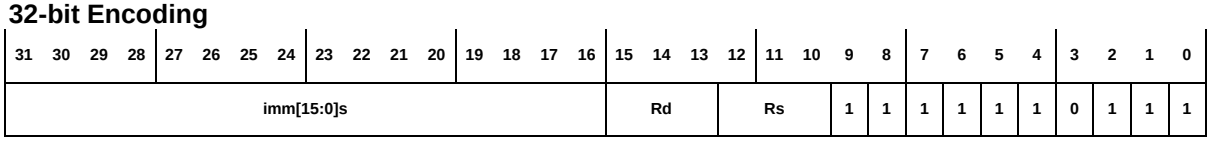
Examples

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

48-bit Encoding



32-bit Encoding



add.n.r — Add without flags, register

Instruction `add.n.r Rd, Rs, Rt`

Description Add two registers. This instruction does not alter %flags[c] and %flags[v].

Flags

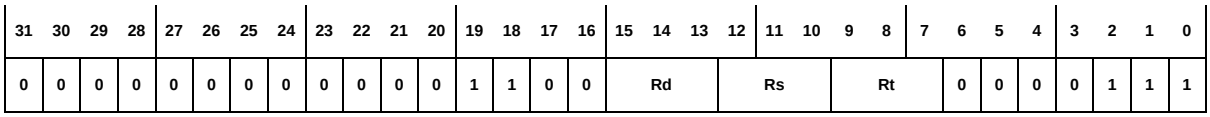
Z	Set if the result is zero; cleared if the result is non-zero
N	Set if the result is negative; cleared if the result is positive
C	Unchanged
V	Unchanged

Operation `Rd = Rs + Rt`

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

Examples `add.n.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
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	Rd				Rs				Rt				0	0	0	0	1	1	1

add.r — Add, register

Instruction `add.r Rd, Rs, Rt`

Description Add two registers

Flags

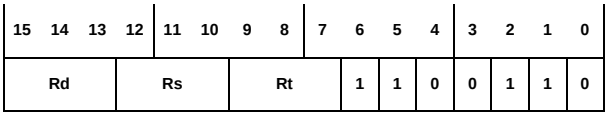
Z	Set if the result is zero; cleared if the result is non-zero
N	Set if the result is negative; cleared if the result is positive
C	Set if the result of the unsigned operation is not correct; cleared otherwise
V	Set if the result of the signed operation is not correct; cleared otherwise

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, #imm`

Description Bitwise AND of register with an immediate 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[31:0]u`

Usage Notes The bit sequence versions of this instruction will raise an InstructionError exception if the arguments are out of range.

Examples

```
and.i %r2, %r1, #0x12345678
and.i %r2, %r1, #0xFEDC
```

48-bit Encoding

47 ... 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
imm[31:0]u	Rd	Rs	0 0 0 1	0 1 1 1

32-bit Encodings

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	size[4:0]u	0 1 0	lsb[4:0]u	0 1 1	Rs	Rd	0 1 0 0 1 1 1

This encoding is used to generate immediates with a sequence of set bits.

msb = lsb + size -1, #imm[msb:lsb] = 1s, other bits 0

(size = 0, lsb = 0 used to encode #imm[31:0] = 0xFFFF FFFF.
If msb > 31, then an InstructionError exception will be raised)

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	size[4:0]u	0 1 1	lsb[4:0]u	0 1 1	Rs	Rd	0 1 0 0 1 1 1

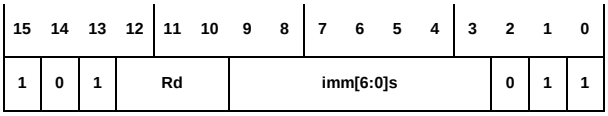
This encoding is used to generate immediates with a sequence of clear bits.

msb = lsb + size -1, #imm[msb:lsb] = 0s, other bits 1

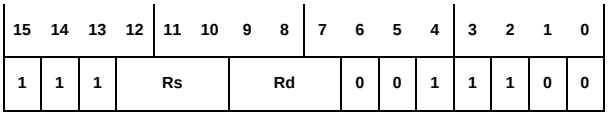
(size = 0, lsb = 0 used to encode #imm[31:0] = 0x0000 0000.
If msb > 31, then an InstructionError exception will be raised)

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
imm[15:0]u				Rd	Rs	0 0 0 1	0 1 1 1

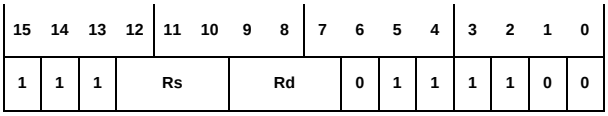
16-bit Encodings



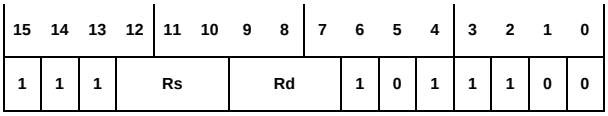
This encoding is only valid when Rs = Rd



This encodes the instruction `and.i Rd, Rs, #1`



This encodes the instruction `and.i Rd, Rs, #0xFF`



This encodes the instruction `and.i Rd, Rs, #0xFFFF`

and.r — AND, register

Instruction `and.r Rd, Rs, Rt`

Description Bitwise AND of two registers

Flags

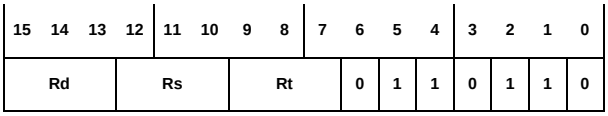
Z	Set if the result is zero; cleared if the result is non-zero
N	Set if the result is negative; cleared if the result is positive
C	Unchanged
V	Unchanged

Operation `Rd = Rs & Rt`

Usage Notes -

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

16-bit Encoding



b2swap.r — Byte swap, register

Instruction

```
b2swap.r Rd, Rs
```

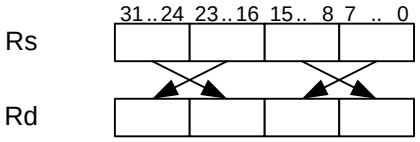
Description

Swap the order of the bytes in a register

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.i` to efficiently change the endianness of a 32-bit number.

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

This instruction can also be used to efficiently change the endianness of a 64-bit number.

Examples

```
b2swap.r %r0, %r0
```

16-bit Encoding

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

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[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

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, 16]s																0	1	0	0	0	1	0	0	1	0	1	0	0	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	0	offset[8:1]s								1	1	0	1

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[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

The `blt.u` (branch if less than, unsigned) instruction is translated into `bcc` 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, 16]s																0	1	0	0	0	0	0	0	1	0	1	0	0	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	0	offset[8:1]s								0	1	0	1

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[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

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, 16]s																0	1	0	0	0	0	0	0	0	0	1	0	0	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	offset[8:1]s								0	1	0	1

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[31] is set; cleared otherwise
C	Cleared
V	Cleared

Operation

```
if ( Rs == 0 )
   PC = offset[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

`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, 16]s				0	0	1	Rs		1	0	0	0	1	0	0	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	Rs		offset[6:1]s				0	1	0	0

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[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

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, 16]s																0	1	0	1	0	1	0	0	0	0	1	0	0	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	0	offset[8:1]s								1	1	0	1

bge.u Branch if greater than or equal, unsigned

Instruction

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

Description

Branch if greater than or equal

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( C == 0 )
   PC = offset[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

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

Examples

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

Encodings

Encodings are shown for the 'bcc' instruction

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[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

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, 16]s																0	1	0	0	1	1	0	0	0	0	1	0	0	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	offset[8:1]s								1	1	0	1

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[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

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, 16]s																0	1	0	1	1	1	0	0	0	0	1	0	0	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	offset[8:1]s								1	1	0	1

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[16:1]s + PC
else
   PC = next instruction
```

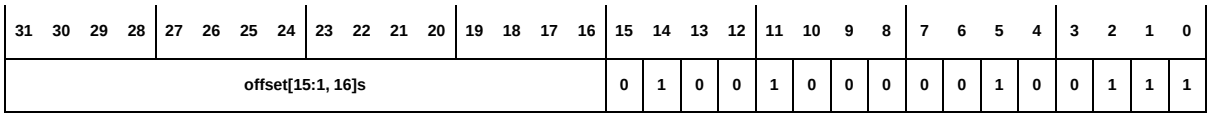
Usage Notes

The assembler inserts the offset from the current PC to the label into the instruction. The offset can take values in the range -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

Examples

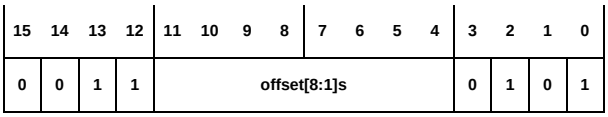
```
ble.s (label, %pc)
```

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, 16]s				0 1 0 0	1 0 0 0	0 0 1 0	0 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	offset[8:1]s		0 1 0 1

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[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

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, 16]s																0	1	0	1	1	0	0	0	0	0	1	0	0	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	offset[8:1]s								0	1	0	1

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 returns, 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 Encodings

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	1	0	1	num[7:0]u								0	0	0	Ras				Rad				0	0	0	0	1	1	1

This encoding can represent num in the range 1 to 256.
num = 256 is encoded with the value 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 returns, the copy resumes.

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

If Rn is zero, the instruction copies no data.

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

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

Instruction

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

Description

Stores the low 8 bits of Rs or #imm to a block of memory.

The source is one of the following:

Normal Register	%r0 - %r7. All bytes equal Rs[7:0].
Immediate	#0 or #0xFF. All bytes equal #imm.

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 the source are stored to a block of memory starting at an address specified in Ra and length specified by #num.

If the source is a register, bits[31:8] are modified to be copies of bits[7:0]

The destination address increments during the store.

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

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

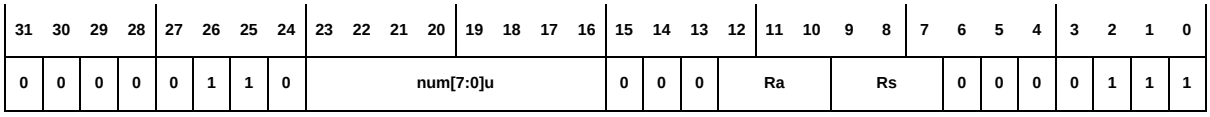
Refer to section 6.1.8, "Block operations", for more details of this instruction.

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
```

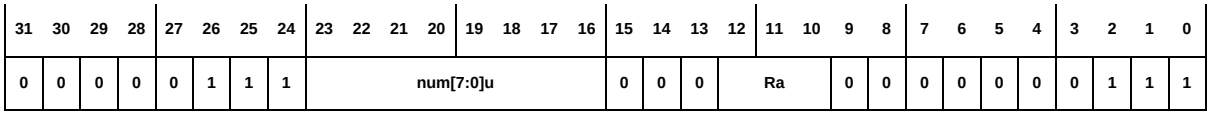
32-bit Encodings



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	1	1	0	num[7:0]u								0	0	0	Ra				Rs				0	0	0	0	1	1	1

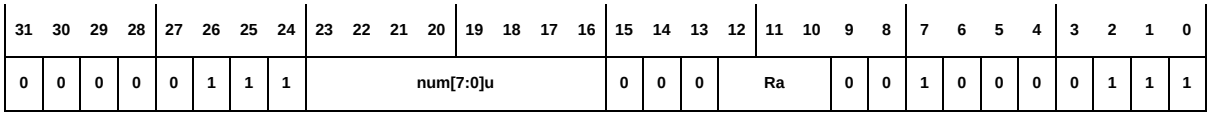
This encodes the instruction `blkst.8.i Rs, @(0, Ra), #num`

This encoding can represent num in the range 1 to 256.
num = 256 is encoded with the value 0.



This encodes the instruction `blkst.8.i #0, @(0, Ra), #num`

This encoding can represent num in the range 1 to 256.
num = 256 is encoded with the value 0.



This encodes the instruction `blkst.8.i #0xFF, @(0, Ra), #num`

This encoding can represent num in the range 1 to 256.
num = 256 is encoded with the value 0.

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

Instruction

```
blkst.8.r Rs, @(0, Ra), Rn
blkst.8.r #imm, @(0, Ra), Rn
```

Description

Stores the low 8 bits of Rs or #imm to a block of memory.

The source is one of the following:

Normal Register	%r0 - %r7. All bytes equal Rs[7:0].
Immediate	#0 or #0xFF. All bytes equal #imm.

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 the source are stored to a block of memory starting at an address specified in Ra and length specified by Rn.

If the source is a register, bits[31:8] are modified to be copies of bits[7:0]

The destination address increments during the store.

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

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

If Rn is zero, the instruction stores no data.

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

32-bit Encodings

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	0	0	0	0	1	0	1	1	Rm				Ra				Rn				0	0	0	0	1	1	1

This encodes the instruction `blkst.8.r Rs, @(0, Ra), Rn`

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

This encodes the instruction `blkst.8.r #0, @(0, Ra), Rn`

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

This encodes the instruction `blkst.8.r #0xFF, @(0, Ra), Rn`

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[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

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, 16]s																0	1	0	1	0	0	0	0	0	0	1	0	0	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	0	offset[8:1]s								0	1	0	1

blt.u Branch if less than, unsigned

Instruction

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

Description

Branch if less than, unsigned

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if ( C == 1 )
   PC = offset[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

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

Examples

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

Encodings

Encodings are shown for the 'bcc' instruction

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[16:1]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 -64kB to +64kB. 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, 16]s																0	1	0	1	0	0	0	0	1	0	1	0	0	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[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

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, 16]s																0	1	0	0	0	1	0	0	0	0	1	0	0	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	offset[8:1]s								1	1	0	1

bnz.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[31] is set; cleared otherwise
C	Cleared
V	Cleared

Operation

```
if ( Rs != 0 )
   PC = offset[16:1]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 -64kB to +64kB. As all instructions start on a 16-bit boundary the least significant bit of the offset is always zero.

`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, 16]s																0	0	1	Rs			1	0	1	0	1	0	0	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	Rs				offset[6:1]s					1	1	0	0

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[16:1]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 -64kB to +64kB. 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, 16]s																0	1	0	1	0	1	0	0	1	0	1	0	0	1	1	1

bra.i, bra.i.2, bra.i.4, bra.i.6 Branch

Instruction

```
bra.i (label, %pc)
bra.i (offset, %pc)
bra.i (label, 0)
bra.i (offset, 0)
bra.i (0, Ra)
bra.i.2 (label, %pc)
bra.i.4 (label, %pc)
bra.i.6 (label, %pc)
bra.i.4 (label, 0)
bra.i.6 (label, 0)
```

Description

Unconditional branch to the address specified

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

For the PC-relative form:

```
PC = offset[31:1]s + PC
```

For the Zero-relative form:

```
PC = offset[31:1]u + 0
```

For the Ra-relative form:

```
PC = Ra
```

Usage Notes

This instruction can branch to any address in memory.

For the PC-relative form, the assembler inserts the offset from the current PC to the label into the instruction.

For the Zero-relative form, the assembler inserts the offset from 0 to the label into the instruction.

For the Ra-relative form, the PC is set to the address contained in the register. The value in Ra should always be even. If not, the processor will generate an AlignError exception. Refer to section 3.8.9, "Error details", for more details.

Normally, the assembler selects the smallest valid encoding of the instruction. The fixed length instructions (`bra.i.2, bra.i.4, bra.i.6`) are useful for jump tables as they force the assembler to generate

2-byte (16-bit), 4-byte (32-bit) or 6-byte (48-bit) instructions.

Examples

```
bra.i   (label, %pc)
bra.i.2 (label, %pc)
bra.i.4 (label, %pc)
bra.i.6 (label, %pc)

bra.i   (label, 0)
bra.i.4 (label, 0)
bra.i.6 (label, 0)

bra.i (0, %r6)
```

48-bit Encodings

47 ... 17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
offset[31:1]u	0	0	0	0	0	0	0	1	1	0	1	1	1	1	1	1	1

This encodes the instruction `bra.i (offset, 0)`

47 ... 17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
offset[31:1]s	0	0	0	1	0	0	0	1	1	0	1	1	1	1	1	1	1

This encodes the instruction `bra.i (offset, %pc)`

32-bit Encodings

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, 16]u				0	0	0	0	0	0	1	1	0	1	1	1	0	1	1	1

This encodes the instruction `bra.i (offset, 0)`

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	offset[24:16]s							1	1	0	1	1	1	1	

This encodes the instruction `bra.i (offset, %pc)`

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
offset[15:1, 16]s				0	1	0	0	0	0	1	1	0	1	1	1	0	1	1	1

This encodes the instruction `bra.i (offset, %pc)`

16-bit Encodings

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

This encodes the instruction `bra.i (offset, %pc)`

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

This encodes the instruction `bra.i (0, Ra)`

bra.m Branch, via memory, displacement

Instruction

```
bra.m @(offset, %pc)
bra.m @(offset, 0)
bra.m @!(offset, %pc)
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

For the PC-relative form:

```
addr = offset[31:1]u + PC
```

For the Zero-relative form:

```
addr = offset[31:2]u + 0
```

For the Ra-relative form:

```
addr = 0 + Ra
```

For all forms:

```
newPC = *(addr)
PC = newPC[31:1]
if addr[1:0] != 0 || newPC[0] == 1:
    throw AlignError
```

Usage Notes

Branches may be done via a function table in memory. This is useful for software patches. The function table entries are 32-bit Zero-relative addresses (function pointers). The function pointers must be a multiple of 2.

The function table may be located anywhere in memory, but each table address should be located at an address which is a multiple of 4.

The table entry address is specified with displacement addressing. The processor reads the 32-bit function pointer at the address and then branches to that function pointer. This instruction is like a ld.i followed by a bra.i, except that it is a single atom.

If the table entry address is not a multiple of 4, or the address in the table is odd, an `AlignError` exception is thrown. PC is updated in both cases with bits [31:1] of the value read from memory.

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.

The `bra.m @!(label, %pc)` and `bra.m @!(label, 0)` forms encode identical instructions to the forms without the !. However, the ! is an instruction to the assembler to create and manage the function table entry in memory as well. For more details see the Binutils Manual, C7920-UM-003.

Examples

```
bra.m @(label, %pc)
bra.m @(label, 0)
bra.m @!(label, 0)
bra.m @(0, %r2)
```

48-bit Encodings

47 ... 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
offset[31:2]u	0	0	0	1	0	0	0	0	1	1	0	1	1	1	1	1	1	1

This encodes the instruction `bra.m @(offset, 0)`

47 ... 17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
offset[31:1]s	0	0	1	1	0	0	0	1	1	0	1	1	1	1	1	1	1

This encodes the instruction `bra.m @(offset, %pc)`

32-bit Encodings

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[15:2, 17, 16]u				1	0	0	0	0	0	1	1	0	1	1	1	0	1	1	1

This encodes the instruction `bra.m @(offset, 0)`

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, 16]s				1	1	0	0	0	0	1	1	0	1	1	1	0	1	1	1

This encodes the instruction `bra.m @(offset, %pc)`

16-bit Encoding

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

This encodes the instruction `bra.m @(0, Ra)`

brk — Break

Instruction

`brk`

Description

Breakpoint. Halt the processor or throw a `Break` exception

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if (FLAGS[M] == User && FLAGS[B] == 1) then
        throw Break exception
else if debugging then
        halt
else
        nop
endif
```

Usage Notes

The `brk` instruction is used to insert a breakpoint 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 breakpoint.

Otherwise, if the processor is in debug mode, the processor is stopped. The processor can be restarted using the SIF interface or a reset.

When using the XAP6 simulator, the processor is always in debug mode. On the XAP6 emulator and real XAP6s, debug mode is controlled with the SIF interface. The processor is in debug mode if RUN_STATE is 'run to break'. xIDE will always put the processor in debug mode.

The B flag can be written with the `movr2s`, `mov.4.i` and `mov.4.r` 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
1	0	1	0	0	0	0	1	0	0	0	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

For the PC-relative form:

```
Push next instruction address to stack
PC = offset[31:1]s + PC
```

For the Zero-relative form:

```
Push next instruction address to stack
PC = offset[31:1]u + 0
```

For the Ra-relative form:

```
Push next instruction address to stack
PC  = 0 + Ra
```

Usage Notes

The `bsr.i` instruction changes the program counter and saves the address of the instruction after the branch onto the stack.

This instruction can branch to any address in memory.

For the PC-relative form, the assembler inserts the offset from the current PC to the label into the instruction.

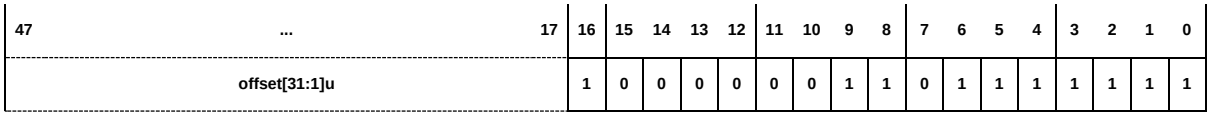
For the Zero-relative form, the assembler inserts the offset from 0 to the label into the instruction.

For the Ra-relative form, the PC is set to the address contained in the register. The value in Ra should always be even. If not, the processor will generate an AlignError exception. Refer to section 3.8.9, "Error details", for more details.

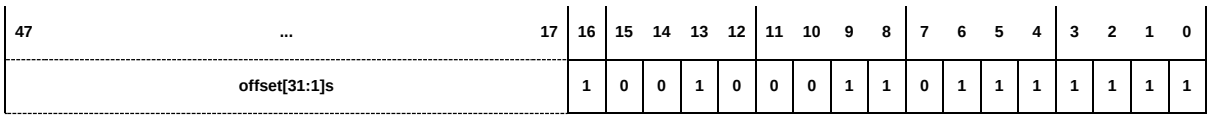
Examples

```
bsr.i (label, %pc)
bsr.i (label, 0)
bsr.i (0, %r1)
```

48-bit Encodings

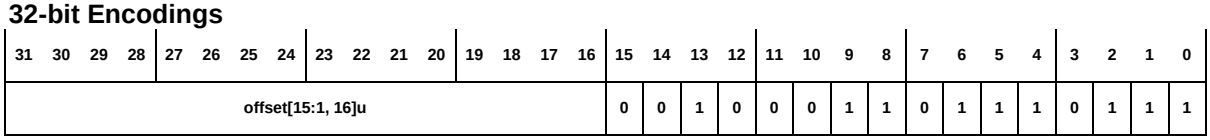


This encodes the instruction `bsr.i (offset, 0)`

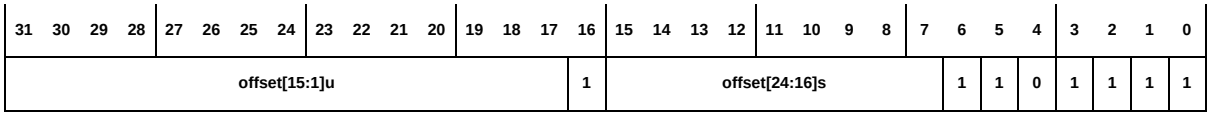


This encodes the instruction `bsr.i (offset, %pc)`

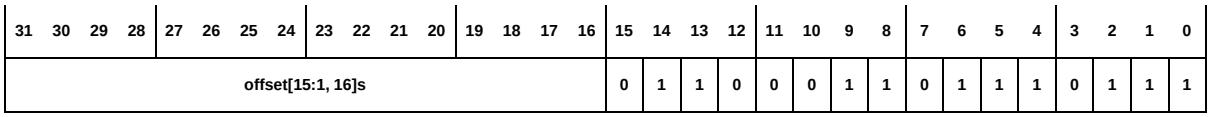
32-bit Encodings



This encodes the instruction `bsr.i (offset, 0)`

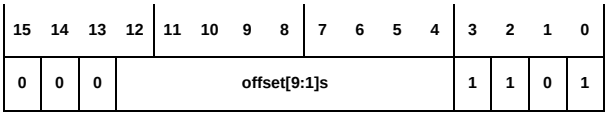


This encodes the instruction `bsr.i (offset, %pc)`

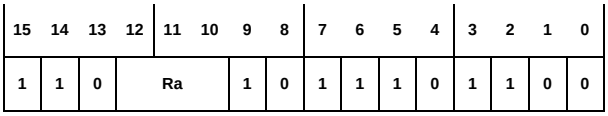


This encodes the instruction `bsr.i (offset, %pc)`

16-bit Encodings



This encodes the instruction `bsr.i (offset, %pc)`



This encodes the instruction `bsr.i (0, Ra)`

bsr.m Branch to subroutine, via memory, displacement

Instruction

```
bsr.m @(offset, %pc)
bsr.m @(offset, 0)
bsr.m @!(offset, %pc)
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

For the PC-relative form:

```
addr = offset[31:1]u + PC
```

For the Zero-relative form:

```
addr = offset[31:2]u + 0
```

For the Ra-relative form:

```
addr = 0 + Ra
```

For all forms:

```
Push next instruction address to stack
newPC = *(addr)
PC = newPC[31:1]
if addr[1:0] != 0 || newPC[0] == 1:
    throw AlignError
```

Usage Notes

The `bsr.m` instruction changes the program counter and saves the address of the instruction after the branch onto the stack.

Branches may be done via a function table in memory. This is useful for software patches. The function table entries are 32-bit Zero-relative addresses (function pointers). The function pointers must be a multiple of 2.

The function table may be located anywhere in memory, but each table address should be located at an address which is a multiple of 4.

The table entry address is specified with displacement addressing. The processor reads the 32-bit function pointer at the address and then branches to that function pointer. This instruction is like a ld.i followed by a bra.i, except that it is a single atom.

If the table entry address is not a multiple of 4, or the address in the table is odd, an `AlignError` exception is thrown. PC is updated in both cases with bits [31:1] of the value read from memory.

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.

The `bsr.m @!(label, %pc)` and `bsr.m @!(label, 0)` forms encode identical instructions to the forms without the !. However, the ! is an instruction to the assembler to create and manage the function table entry in memory as well. For more details see the Binutils Manual, C7920-UM-003.

Examples

```
bsr.m @(label, %pc)
bsr.m @(label, 0)
bsr.m @!(label, 0)
bsr.m @(0, %r2)
```

48-bit Encodings

47 ... 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
offset[31:2]u	0	1	0	1	0	0	0	0	1	1	0	1	1	1	1	1	1	1

This encodes the instruction `bsr.m @(offset, 0)`

47 ... 17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
offset[31:1]s	1	0	1	1	0	0	0	1	1	0	1	1	1	1	1	1	1

This encodes the instruction `bsr.m @(offset, %pc)`

32-bit Encodings

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[15:2, 17, 16]u				1	0	1	0	0	0	1	1	0	1	1	1	0	1	1	1

This encodes the instruction `bsr.m @(offset, 0)`

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, 16]s				1	1	1	0	0	0	1	1	0	1	1	1	0	1	1	1

This encodes the instruction `bsr.m @(offset, %pc)`

16-bit Encoding

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

This encodes the instruction `bsr.m @(0, Ra)`

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[16:1]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 -64kB to +64kB. 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, 16]s																0	1	0	0	1	1	0	0	1	0	1	0	0	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[16:1]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 -64kB to +64kB. 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, 16]s																0	1	0	0	1	0	0	0	1	0	1	0	0	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". ErrorPval (%flags[p]) can be used to identify the type of exception.

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
imm[15:0]																0	1	0	0	1	1	0	1	0	0	1	0	0	1	1	1

clu.d CLU Instruction, type 2

Instruction `clu.d #imm, Rd`

Description User customisable instruction with the following arguments:
- #immediate
- destination register

Flags

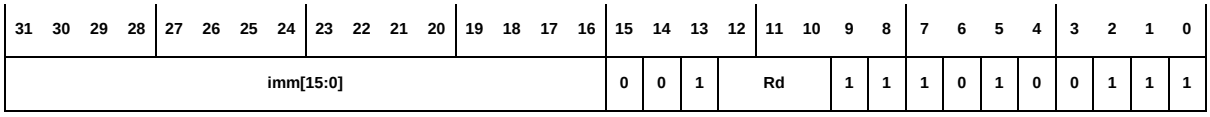
Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

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". ErrorPval (%flags[p]) can be used to identify the type of exception.

Examples `clu.d #0x1234, %r4`

32-bit Encoding



clu.ds CLU Instruction, type 3

Instruction

```
clu.ds #imm, Rd, Rs
```

Description

User customisable instruction with the following arguments:
- #immediate
- destination register
- 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

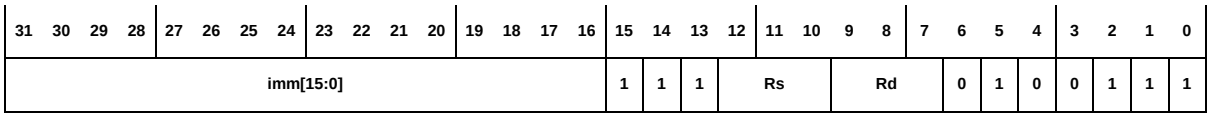
Usage Notes

Sets clu_type[3:0] = 3.
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". ErrorPval (%flags[p]) can be used to identify the type of exception.

Examples

```
clu.ds #0x1234, %r4, %r7
```

32-bit Encoding



clu.dst CLU Instruction, type 4

Instruction

```
clu.dst #imm, Rd, Rs, Rt
```

Description

User customisable instruction with the following arguments:
- #immediate
- destination register
- source register
- 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

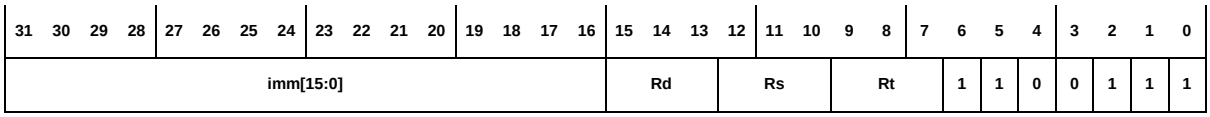
Usage Notes

Sets clu_type[3:0] = 4.
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". ErrorPval (%flags[p]) can be used to identify the type of exception.

Examples

```
clu.dst #0x1234, %r4
```

32-bit Encoding



clu.s CLU Instruction, type 5

Instruction `clu.s #imm, Rs`

Description User customisable instruction with the following arguments:
- #immediate
- 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.
Rs is passed to the CLU.
This may throw an `InstructionError`. Refer to section 3.8.9, "Error details". ErrorPval (%flags[p]) can be used to identify the type of exception.

Examples `clu.s #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
imm[15:0]				0	0	1	Rs		1	1	0	0	1	0	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
- source register
- secondary source register

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

User defined

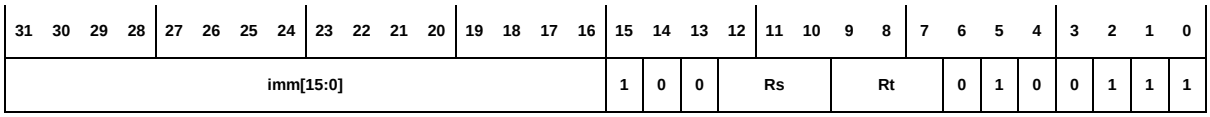
Usage Notes

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". ErrorPval (%flags[p]) can be used to identify the type of exception.

Examples

```
clu.st #0x1234, %r4, %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
imm[15:0]				1	0	0	Rs	Rt	0 1 0	0 1 1 1

cmp.16.i Compare, 16-bit, immediate

Instruction `cmp.16.i Rs, #imm`

Description A 16-bit compare of the register with an immediate

Flags

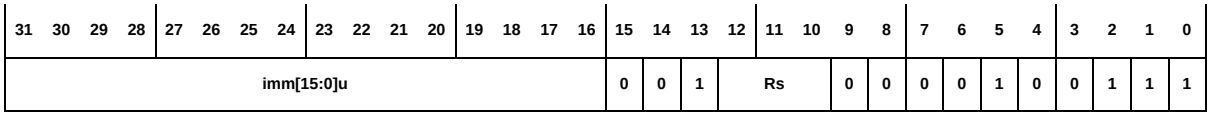
Z	Set if `Rs[15:0] == #immediate[15:0];` cleared otherwise
N	Set if the result of the subtraction is negative; cleared if the result is positive
C	Set if the result of the unsigned subtraction cannot be represented in 16 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise

Operation `Rs[15:0] - #immediate[15:0]`

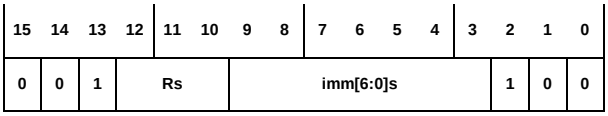
Usage Notes This is a 16-bit compare of a 16-bit immediate with the lower 16 bits of a register. The high 24 bits of the immediate and register are ignored.

Examples `cmp.16.i %r1, #0x12`

32-bit Encoding



16-bit Encoding



cmp.16.r — Compare, 16-bit, register

Instruction

```
cmp.16.r Rs, Rt
```

Description

Compare the low 16 bits in two registers

Flags

Z	Set if `Rs[15:0] == Rt[15:0]`; 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 16 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise

Operation

```
Rs[15:0] - Rt[15:0]
```

Usage Notes

This is a 16-bit compare of the lower 16 bits from two registers. The high 24 bits of the two registers are ignored.

Examples

```
cmp.16.r %r1, %r2
```

16-bit Encoding

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

cmp.16c.i Compare, 16-bit with carry, immediate

Instruction

`cmp.16c.i Rs, #imm`

Description

A 16-bit compare of the register with an immediate and the carry flag

Flags

Z Set if `Rs[15:0] == #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[15:0] - #immediate[15:0] - C`

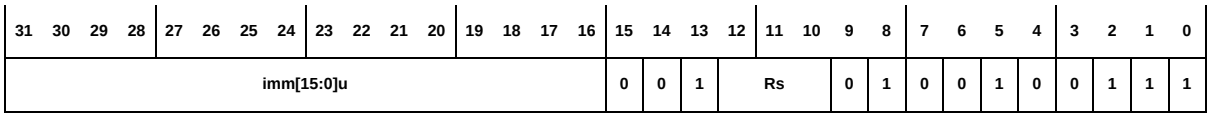
Usage Notes

This is a 16-bit compare of a 16-bit immediate with the lower 16 bits of a register. The high 24 bits of the immediate and register are ignored.

Examples

`cmp.16c.i %r1, #0x12`

32-bit Encoding



cmp.16c.r Compare, 16-bit with carry, register

Instruction

```
cmp.16c.r Rs, Rt
```

Description

A 16-bit compare of two registers and the carry flag

Flags

Z Set if `Rs[15:0] == Rt[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[15:0] - Rt[15:0] - C
```

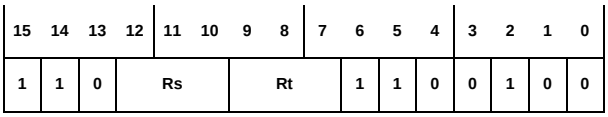
Usage Notes

This is a 16-bit compare of the lower 16 bits from two registers and the carry flag. The high 24 bits of the two registers are ignored.

Examples

```
cmp.16c.r %r1, %r2
```

16-bit Encoding



cmp.16x.i Compare, 16-bit, exchange, immediate

Instruction `cmp.16x.i Rs, #imm`

Description A 16-bit compare of the register with an immediate and with the operand order reversed

Flags

Z	Set if `Rs[15:0] == #immediate[15:0];` 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 16 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 16 bits; cleared otherwise

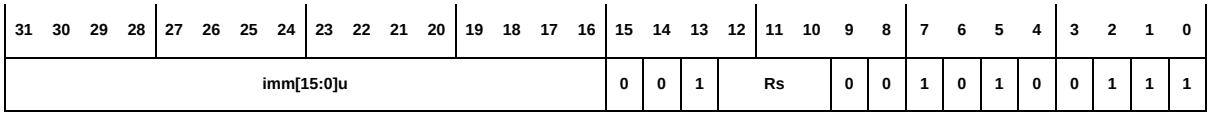
Operation `#immediate[15:0] - Rs[15:0]`

Usage Notes This is a 16-bit compare of a 16-bit immediate with the lower 16 bits of a register. The high 24 bits of the immediate and register are ignored.

This is the same as `cmp.16.i` but with the operand order reversed.

Examples `cmp.16x.i %r1, #0x12`

32-bit Encoding



cmp.16xc.i Compare, 16-bit carry, exchange, immediate

Instruction

`cmp.16xc.i Rs, #imm`

Description

A 16-bit compare of the register with an immediate and the carry flag, and with the operand order reversed

Flags

Z Set if `Rs[15:0] == #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

`#immediate[15:0] - Rs[15:0] - C`

Usage Notes

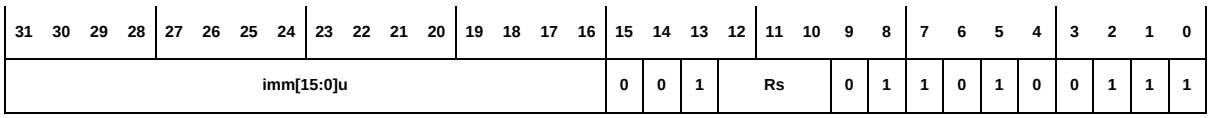
This is a 16-bit compare of a 16-bit immediate with the lower 16 bits of a register. The high 24 bits of the immediate and register are ignored.

This is the same as `cmp.16c.i` but with the operand order reversed.

Examples

`cmp.16xc.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
imm[15:0]u				0	0	1		Rs	0	1	1	0	1	0	0	1	1	1

cmp.8.i Compare, 8-bit, immediate

Instruction `cmp.8.i Rs, #imm`

Description An 8-bit compare of the register with an immediate

Flags

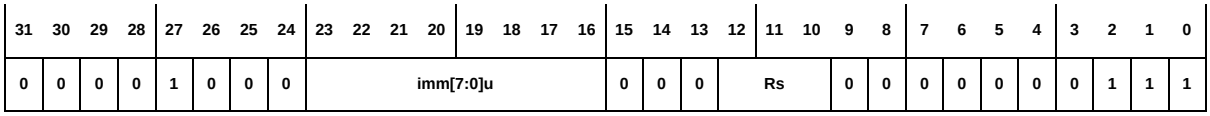
Z	Set if `Rs[7:0] == #immediate[7:0];` cleared otherwise
N	Set if the result of the subtraction is negative; cleared if the result is positive
C	Set if the result of the unsigned subtraction cannot be represented in 8 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 8 bits; cleared otherwise

Operation `Rs[7:0] - #immediate[7:0]`

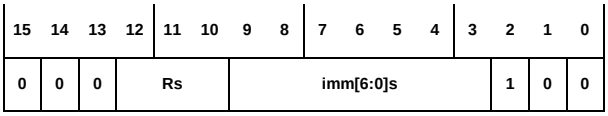
Usage Notes This is an 8-bit compare of an 8-bit immediate with the lower 8 bits of a register. The high 24 bits of the immediate and register are ignored.

Examples `cmp.8.i %r1, #0x12`

32-bit Encoding



16-bit Encoding



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 24 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
1	1	0	Rs				Rt		0	0	0	0	1	0	0

cmp.8c.i Compare, 8-bit with carry, immediate

Instruction

```
cmp.8c.i Rs, #imm
```

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 24 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	1	0	1	0	imm[7:0]u								0	0	0	Rs			0	0	0	0	0	0	0	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 flag

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
```

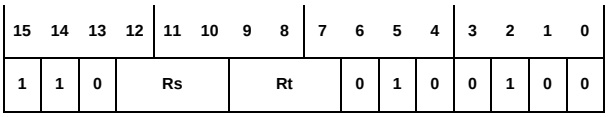
Usage Notes

This is an 8-bit compare of the lower 8 bits from two registers and the carry flag. The high 24 bits of the two registers are ignored.

Examples

```
cmp.8c.r %r1, %r2
```

16-bit Encoding



cmp.8x.i Compare, 8-bit, exchange, immediate

Instruction `cmp.8x.i Rs, #imm`

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

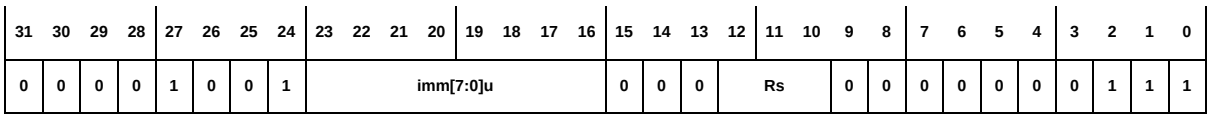
Operation `#immediate[7:0] - Rs[7:0]`

Usage Notes This is an 8-bit compare of the lower 8 bits of a register with an 8-bit immediate. The high 24 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	1	0	0	1	imm[7:0]u								0	0	0	Rs			0	0	0	0	0	0	0	1	1	1

cmp.8xc.i Compare, 8-bit carry, exchange, immediate

Instruction

`cmp.8xc.i Rs, #imm`

Description

An 8-bit compare of a register with an immediate and the carry flag, 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

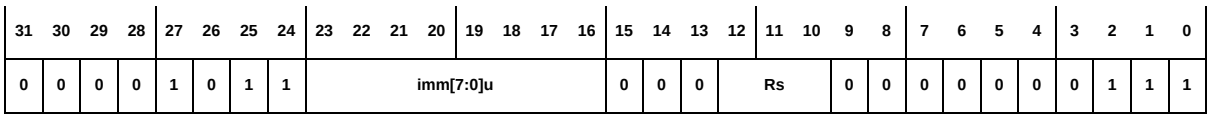
This is an 8-bit compare of an 8-bit immediate with the lower 8 bits of a register and the carry flag. The high 24 bits of the immediate and register are ignored.

This is the same as `cmp.8c.i` but with the operand order reversed.

Examples

```
cmp.8xc.i %r1, #0x12
```

32-bit Encoding



cmp.c.i — Compare, with carry, immediate

Instruction

```
cmp.c.i Rs, #imm
```

Description

A 32-bit compare of the register with an immediate and the carry flag

Flags

Z Set if `Rs == #immediate[31:0]` 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 32 bits; cleared otherwise

V Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise

Operation

```
Rs - #immediate[31:0] - C
```

Usage Notes

The Z flag behaviour is useful for 64-bit arithmetic.

A 64-bit comparison of two registers with an immediate (for example, 0x0000123400005678) 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
```

48-bit Encoding

47 ... 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[31:0]s	0	1	0	Rs			1	0	1	1	1	1	1	1	1	1

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
imm[15:0]s																0	1	0	Rs			1	0	1	1	1	1	0	1	1	1

cmp.c.r — Compare, with carry, register

Instruction

```
cmp.c.r Rs, Rt
```

Description

A 32-bit compare of two registers and the carry flag

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 32 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise

Operation

```
Rs - Rt - C
```

Usage Notes

The Z flag behaviour is useful for 64-bit arithmetic.

Examples

```
cmp.c.r %r1, %r2
```

16-bit Encoding

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

cmp.i — Compare, immediate

Instruction `cmp.i Rs, #imm`

Description A 32-bit compare of the register with an immediate

Flags

Z	Set if `Rs == #immediate[31:0]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 32 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise

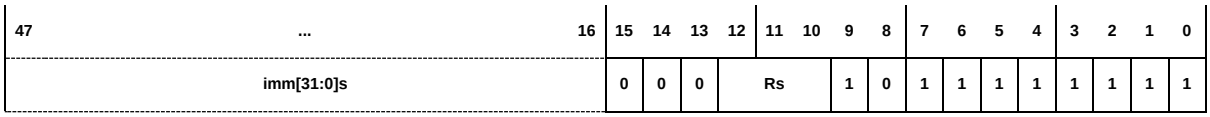
Operation `Rs - #immediate[31:0]s`

Usage Notes -

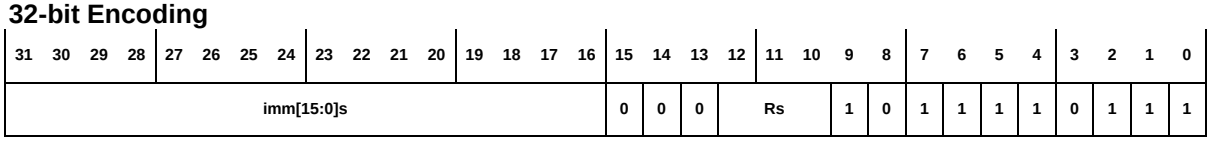
Examples

```
cmp.i %r1, #0x1234
cmp.i %r1, #0xFEDC
```

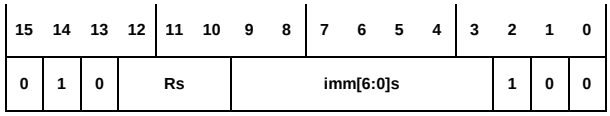
48-bit Encoding



32-bit Encoding



16-bit Encoding



cmp.r — Compare, register

Instruction

`cmp.r Rs, Rt`

Description

A 32-bit compare of 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 32 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise

Operation

`Rs - Rt`

Usage Notes

-

Examples

```
cmp.r %r1, %r2
```

16-bit Encoding

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

cmp.x.i — Compare, exchange, immediate

Instruction

```
cmp.x.i Rs, #imm
```

Description

A 32-bit compare of the register with an immediate and with the operand order reversed

Flags

Z	Set if `Rs == #immediate[31:0]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 32 bits; cleared otherwise
V	Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise

Operation

```
#immediate[31:0]s - Rs
```

Usage Notes

This is the same as `cmp.i` but with the operand order reversed.

Examples

```
cmp.x.i %r1, #0x1234
cmp.x.i %r1, #0xFEDC
```

48-bit Encoding

47 ... 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[31:0]s	0	0	1	Rs			1	0	1	1	1	1	1	1	1	1

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
imm[15:0]s				0	0	1	Rs			1	0	1	1	1	1	0	1	1	1

cmp.xc.i Compare, with carry, exchange, immediate

Instruction

`cmp.xc.i Rs, #imm`

Description

A 32-bit compare of a register with an immediate and the carry flag, and with the operand order reversed

Flags

Z Set if `Rs == #immediate[31:0]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 32 bits; cleared otherwise

V Set if the result of the signed subtraction cannot be represented in 32 bits; cleared otherwise

Operation

`#immediate[31:0]s - Rs - C`

Usage Notes

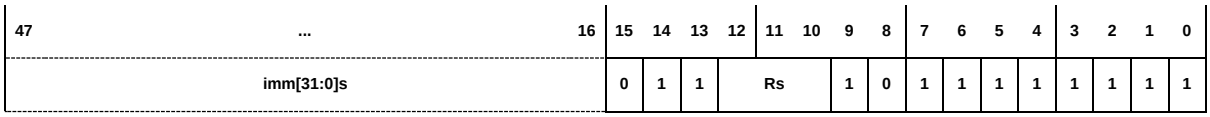
This is the same as `cmp.c.i` but with the operand order reversed.

The Z flag behaviour is useful for 64-bit arithmetic.

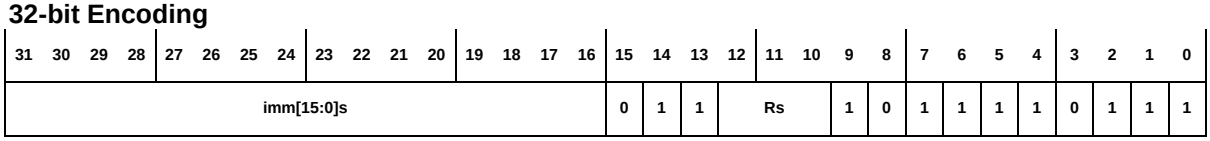
Examples

```
cmp.xc.i %r1, #0x1234
cmp.xc.i %r1, #0xFEDC
```

48-bit Encoding



32-bit Encoding



div.s.r — Divide, 32-bit signed, register

Instruction `div.s.r Rd, Rs, Rt`

Description Signed 32-bit integer divide to give a 32-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 If the quotient cannot be represented in 32 bits, the overflow flag is set and the quotient is forced to zero.

See section 0, "Divide by zero" for the effects of dividing by zero.

Examples `div.s.r %r4, %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
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	Rd				Rs				Rt				0	0	0	0	1	1	1	1

div.u.r — Divide, unsigned, register

Instruction `div.u.r Rd, Rs, Rt`

Description Unsigned 32-bit integer divide to give a 32-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 = Rs / Rt`

Usage Notes If the quotient cannot be represented in 32 bits, the overflow flag is set and the quotient is forced to zero.

See section 0, "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
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	Rd				Rs				Rt				0	0	0	0	1	1	1

divrem.s.r Divide and remainder, signed, register

Instruction `divrem.s.r Rdr, Rdq, Rs, Rt`

Description Signed 32- integer divide to give a 32-bit quotient and a 32-bit remainder

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Set if an overflow occurs or if an attempt is made to divide by zero; cleared otherwise

Operation

```
Rdq     = Rs / Rt
Rdr     = Rs % Rt
```

Usage Notes

If the quotient cannot be represented in 32 bits, the overflow flag is set and the quotient and remainder are forced to zero.

Rs and Rdq are forced to be the same register.

Rdr and Rdq must not be the same register. If they are, then an InstructionError exception is raised. Refer to section 3.8.9, "Error details", for more details.

See section 0, "Divide by zero" for the effects of dividing by zero.

Examples `divrem.s.r %r4, %r1, %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
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	Rdr				Rrq				Rt				0	0	0	0	1	1	1

This encoding is only valid when Rs = Rdq

divrem.u.r Divide and remainder, unsigned, register

Instruction

```
divrem.u.r Rdr, Rdq, Rs, Rt
```

Description

Unsigned 32- integer divide to give a 32-bit quotient and a 32-bit remainder

Flags

Z	Unchanged
N	Unchanged
C	Set if an overflow occurs or if an attempt is made to divide by zero; cleared otherwise
V	Unchanged

Operation

```
Rdq     = Rs / Rt
Rdr     = Rs % Rt
```

Usage Notes

If the quotient cannot be represented in 32 bits, the overflow flag is set and the quotient and remainder are forced to zero.

Rs and Rdq are forced to be the same register.

Rdr and Rdq must not be the same register. If they are, then an InstructionError exception is raised. Refer to section 3.8.9, "Error details", for more details.

See section 0, "Divide by zero" for the effects of dividing by zero.

Examples

```
divrem.u.r %r4, %r1, %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
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	Rdr				Rrq				Rt				0	0	0	0	1	1	1

This encoding is only valid when Rs = Rdq

fill — Fill prefetch buffer

Instruction `fill`

Description This instruction fills the processor's prefetch buffer.

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

Usage Notes This instruction pauses execution and allows the following instructions to be fetched into the processor's prefetch buffer.

The instruction completes when the buffer is full, or aborts if an interrupt occurs.

There are no functional effects of this instruction, but it will affect timing of subsequent instructions.

Examples `fill`

16-bit Encoding

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

fimode — Flags and info mode

Instruction `fimode Rd`

Description Read a number representing the processor mode/state

Flags

Z	Set if Rd is set to 0 (Supervisor mode), cleared otherwise
N	Cleared
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

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

flip.16.r — Flip word bits

Instruction

```
flip.16.r Rd, Rs
```

Description

Reverses the order of bits in the half-words of the source register

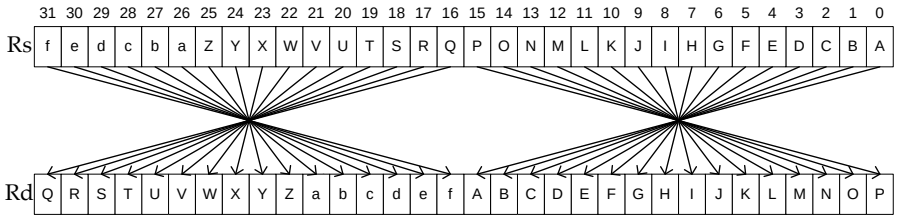
Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
Rd[0] = Rs[15]          Rd[16] = Rs[31]
Rd[1] = Rs[14]          Rd[17] = Rs[30]
...                     ...
Rd[14] = Rs[1]          Rd[30] = Rs[17]
Rd[15] = Rs[0]          Rd[31] = Rs[16]
```

Usage Notes



Examples

```
flip.16.r %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
0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	1	0	0	0	Rs				Rd		0	0	0	0	1	1	1

flip.8.r — Flip byte bits

Instruction

```
flip.8.r Rd, Rs
```

Description

Reverses the order of bits in the bytes of the source register

Flags

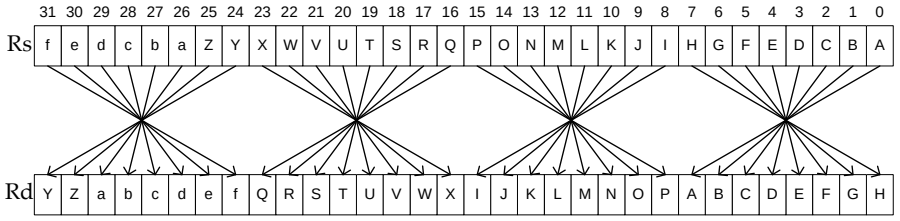
Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
Rd[0] = Rs[7]                 Rd[16] = Rs[23]
Rd[1] = Rs[6]                 Rd[17] = Rs[22]
...                           ...
Rd[6] = Rs[1]                 Rd[22] = Rs[17]
Rd[7] = Rs[0]                 Rd[23] = Rs[23]

Rd[8] = Rs[15]                Rd[24] = Rs[31]
Rd[9] = Rs[14]                Rd[25] = Rs[30]
...                           ...
Rd[14] = Rs[9]                Rd[30] = Rs[25]
Rd[15] = Rs[8]                Rd[31] = Rs[24]
```

Usage Notes



Examples

```
flip.8.r %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
0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	Rs				Rd				0	0	0	0	1	1	1

flip.r — Flip bits

Instruction

```
flip.r Rd, Rs
```

Description

Reverses the order of bits in the source register

Flags

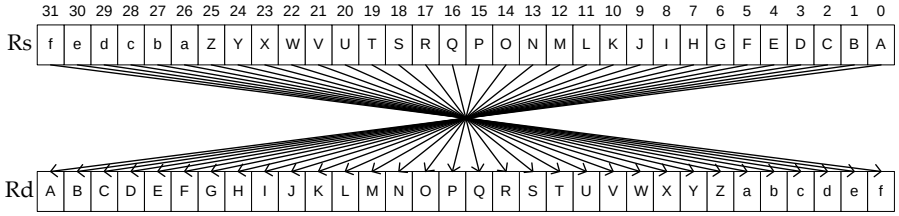
Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
Rd[0] = Rs[31]
Rd[1] = Rs[30]
...
Rd[30] = Rs[1]
Rd[31] = Rs[0]
```

Usage Notes

Bit reversals are found in communications systems and in signal processing.



Examples

```
flip.r %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
0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	0	0	Rs				Rd		0	0	0	0	1	1	1

flush — Flush prefetch buffer

Instruction `flush`

Description This instruction empties the processor's prefetch buffer.

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

Usage Notes This instruction empties the processor's prefetch buffer.

This may be necessary when self-modifying code is being used.

This instruction will affect timing of subsequent instructions.

Examples `Flush`

16-bit Encoding

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

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 exception
endif
```

Usage Notes

A `PrivInstruction` error 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
1	0	1	0	0	0	0	1	0	0	0	1	0	1	0	1

ld.16z.i Load, 16-bit, zero-extend, displacement

Instruction

```
ld.16z.i Rd, @(offset, Ra)
ld.16z.i Rd, @(offset, %pc)
ld.16z.i Rd, @(offset, %sp)
ld.16z.i Rd, @(offset, %gp)
ld.16z.i Rd, @(offset, 0)
ld.16z.i Rd, @(label, %pc)
ld.16z.i Rd, @(label, %gp)
ld.16z.i Rd, @(label, 0)
```

Description

Load zero-extended 16-bit value from memory into `Rd` with displacement addressing.

Flags

Z	Set if the result is zero; cleared otherwise
N	Cleared
C	Unchanged
V	Unchanged

Operation

For the Ra-relative form:

```
addr = offset[31:0]s + Ra
Rd = (uint32) *(int16*)(addr)
```

For the PC-relative form:

```
addr = offset[31:0]s + PC
Rd = (uint32) *(int16*)(addr)
```

For the SP-relative form:

```
addr = offset[31:0]u + SP
Rd = (uint32) *(int16*)(addr)
```

For the GP-relative form:

```
addr = offset[31:0]u + GP
Rd = (uint32) *(int16*)(addr)
```

For the Zero-relative form:

```
addr = offset[31:0]u + 0
Rd = (uint32) *(int16*)(addr)
```

Usage Notes

Ra and offset are interpreted as byte addresses.

A 16-bit memory read is performed.

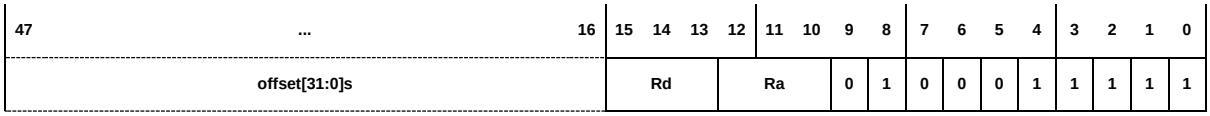
Refer to section 3.8.9, "Error details", for details of possible exceptions.

Examples

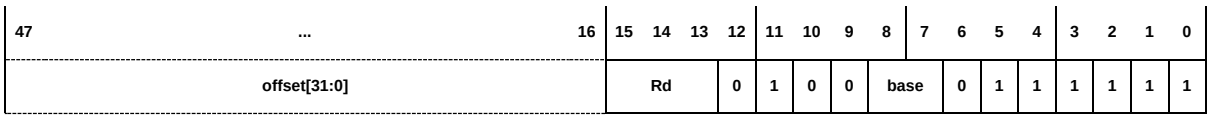
```
ld.16z.i %r2, @(28, %r7)
ld.16z.i %r1, @(label, %pc)
ld.16z.i %r5, @(23, %pc)
```

```
ld.16z.i %r3, @(0, %sp)
ld.16z.i %r3, @(4, %gp)
ld.16z.i %r6, @(label, 0)
ld.16z.i %r7, @(0x8100, 0)
```

48-bit Encodings



This encodes the instruction `ld.16z.i Rd, @(offset, Ra)`

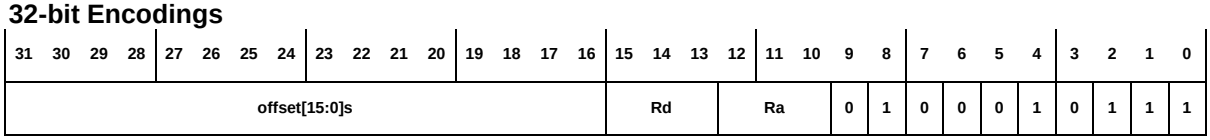


This encodes the instruction `ld.16z.i Rd, @(offset, base)`

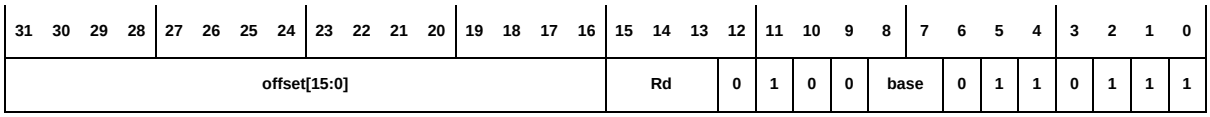
Base is encoded as follows:

encoding	base	Offset Type
0	0	Unsigned
1	%sp	Unsigned
2	%pc	Signed
3	%gp	Unsigned

32-bit Encodings



This encodes the instruction `ld.16z.i Rd, @(offset, Ra)`

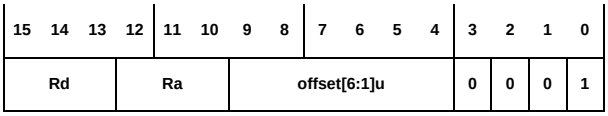


This encodes the instruction `ld.16z.i Rd, @(offset, base)`

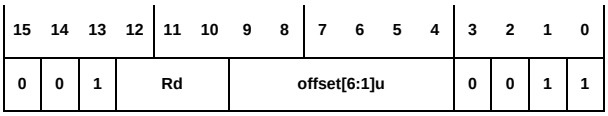
Base is encoded as follows:

encoding	base	Offset Type
0	0	Unsigned
1	%sp	Unsigned
2	%pc	Signed
3	%gp	Unsigned

16-bit Encodings



This encodes the instruction `ld.16z.i Rd, @(offset, Ra)`



This encodes the instruction `ld.16z.i Rd, @(offset, %sp)`

ld.16z.r Load, 16-bit, zero-extend, indexed

Instruction

```
ld.16z.r Rd, @(Rx, Ra)
ld.16z.r Rd, @(Rx, %sp)
```

Description

Load zero extended 16-bit value from memory into `Rd` with indexed addressing.

Flags

Z	Set if the result is zero; cleared otherwise
N	Cleared
C	Unchanged
V	Unchanged

Operation

For the Ra-relative form:

```
addr = Rx + Ra
Rd = (uint32) *(int16*)(addr)
```

For the SP-relative form:

```
addr = Rx + SP
Rd = (uint32) *(int16*)(addr)
```

Usage Notes

Ra and Rx are interpreted as byte addresses.

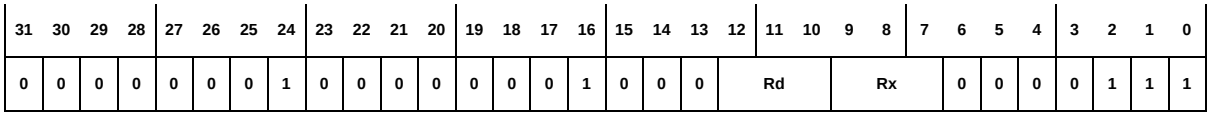
A 16-bit memory read is performed.

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

Examples

```
ld.16z.r %r2, @(%r0, %r7)
ld.16z.r %r1, @(%r3, %sp)
```

32-bit Encoding



This encodes the instruction `ld.16z.r Rd, @(Rx, %sp)`

16-bit Encoding

This encodes the instruction `ld.16z.r Rd, @(Rx, Ra)`

ld.8z.i Load, 8-bit, zero-extend, displacement

Instruction

```
ld.8z.i Rd, @(offset, Ra)
ld.8z.i Rd, @(offset, %pc)
ld.8z.i Rd, @(offset, %sp)
ld.8z.i Rd, @(offset, %gp)
ld.8z.i Rd, @(offset, 0)
ld.8z.i Rd, @(label, %pc)
ld.8z.i Rd, @(label, %gp)
ld.8z.i Rd, @(label, 0)
```

Description

Load zero-extended 8-bit value from memory into `Rd` with displacement addressing.

Flags

Z	Set if the result is zero; cleared otherwise
N	Cleared
C	Unchanged
V	Unchanged

Operation

For the Ra-relative form:

```
addr = offset[31:0]s + Ra
Rd = (uint32) *(int8*)(addr)
```

For the PC-relative form:

```
addr = offset[31:0]s + PC
Rd = (uint32) *(int8*)(addr)
```

For the SP-relative form:

```
addr = offset[31:0]u + SP
Rd = (uint32) *(int8*)(addr)
```

For the GP-relative form:

```
addr = offset[31:0]u + GP
Rd = (uint32) *(int8*)(addr)
```

For the Zero-relative form:

```
addr = offset[31:0]u + 0
Rd = (uint32) *(int8*)(addr)
```

Usage Notes

Ra and offset are interpreted as byte addresses.

An 8-bit memory read is performed.

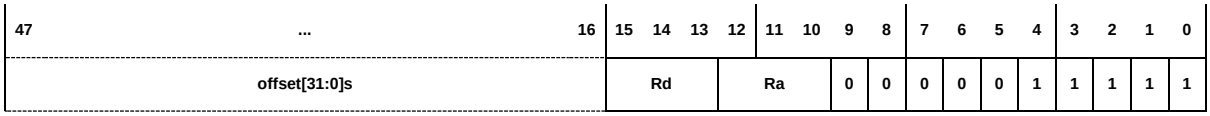
Refer to section 3.8.9, "Error details", for details of possible exceptions.

Examples

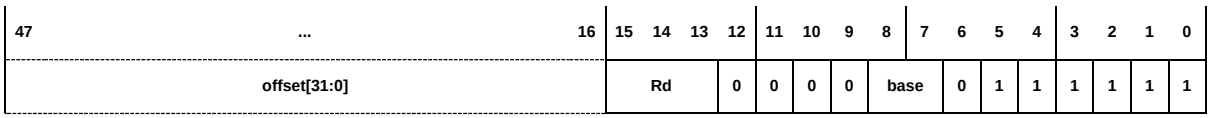
```
ld.8z.i %r2, @(28, %r7)
ld.8z.i %r1, @(label, %pc)
ld.8z.i %r5, @(23, %pc)
```

```
ld.8z.i %r3, @(0, %sp)
ld.8z.i %r3, @(4, %gp)
ld.8z.i %r6, @(label, 0)
ld.8z.i %r7, @(0x8100, 0)
```

48-bit Encodings



This encodes the instruction `ld.8z.i Rd, @(offset, Ra)`

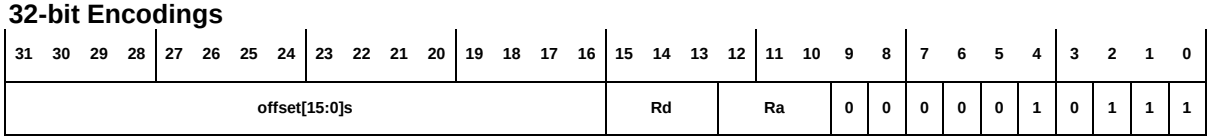


This encodes the instruction `ld.8z.i Rd, @(offset, base)`

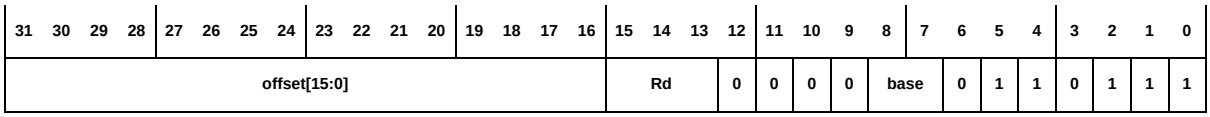
Base is encoded as follows:

encoding	base	Offset Type
0	0	Unsigned
1	%sp	Unsigned
2	%pc	Signed
3	%gp	Unsigned

32-bit Encodings



This encodes the instruction `ld.8z.i Rd, @(offset, Ra)`

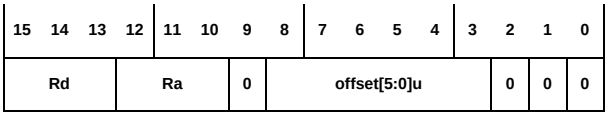


This encodes the instruction `ld.8z.i Rd, @(offset, base)`

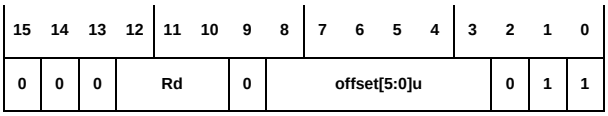
Base is encoded as follows:

encoding	base	Offset Type
0	0	Unsigned
1	%sp	Unsigned
2	%pc	Signed
3	%gp	Unsigned

16-bit Encodings



This encodes the instruction `ld.8z.i Rd, @(offset, Ra)`



This encodes the instruction `ld.8z.i Rd, @(offset, %sp)`

ld.8z.r — Load, 8-bit, zero-extend, indexed

Instruction

```
ld.8z.r Rd, @(Rx, Ra)
ld.8z.r Rd, @(Rx, %sp)
```

Description

Load zero extended 8-bit value from memory into `Rd` with indexed addressing.

Flags

Z	Set if the result is zero; cleared otherwise
N	Cleared
C	Unchanged
V	Unchanged

Operation

For the Ra-relative form:

```
addr = Rx + Ra
Rd = (uint32) *(int8*)(addr)
```

For the SP-relative form:

```
addr = Rx + SP
Rd = (uint32) *(int8*)(addr)
```

Usage Notes

Ra and Rx are interpreted as byte addresses.

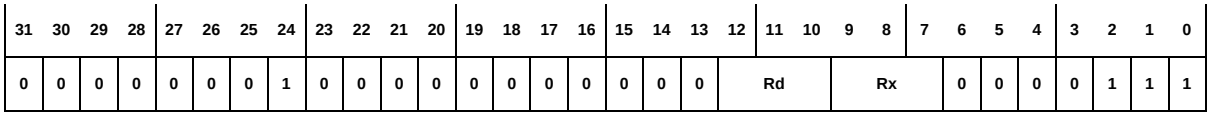
An 8-bit memory read is performed.

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

Examples

```
ld.8z.r %r2, @(%r0, %r7)
ld.8z.r %r1, @(%r3, %sp)
```

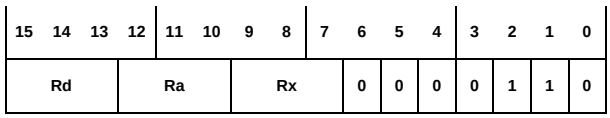
32-bit Encoding



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
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	Rd				Rx				0	0	0	0	1	1	1

This encodes the instruction `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 instruction `ld.8z.r Rd, @(Rx, Ra)`

ld.i — Load, displacement

Instruction

```
ld.i Rd, @(offset, Ra)
ld.i Rd, @(offset, %pc)
ld.i Rd, @(offset, %sp)
ld.i Rd, @(offset, %gp)
ld.i Rd, @(offset, 0)
ld.i Rd, @(label, %pc)
ld.i Rd, @(label, %gp)
ld.i Rd, @(label, 0)
```

Description

Load 32-bit value from memory into `Rd` with displacement addressing.

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

For the Ra-relative form:

```
addr = offset[31:0]s + Ra
Rd = *(int32*)(addr)
```

For the PC-relative form:

```
addr = offset[31:0]s + PC
Rd = *(int32*)(addr)
```

For the SP-relative form:

```
addr = offset[31:0]u + SP
Rd = *(int32*)(addr)
```

For the GP-relative form:

```
addr = offset[31:0]u + GP
Rd = *(int32*)(addr)
```

For the Zero-relative form:

```
addr = offset[31:0]u + 0
Rd = *(int32*)(addr)
```

Usage Notes

Ra and offset are interpreted as byte addresses.

A 32-bit memory read is performed.

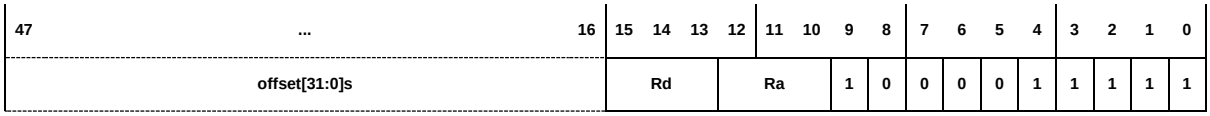
Refer to section 3.8.9, "Error details", for details of possible exceptions.

Examples

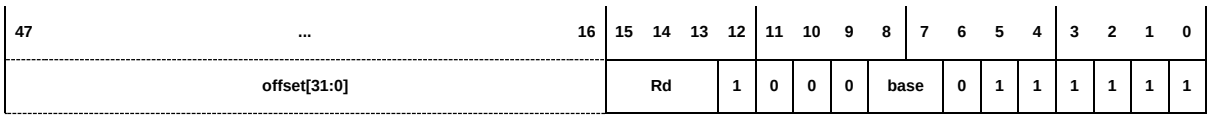
```
ld.i %r2, @(28, %r7)
ld.i %r1, @(label, %pc)
ld.i %r5, @(23, %pc)
```

```
ld.i %r3, @(0, %sp)
ld.i %r3, @(4, %gp)
ld.i %r6, @(label, 0)
ld.i %r7, @(0x8100, 0)
```

48-bit Encodings



This encodes the instruction `ld.i Rd, @(offset, Ra)`

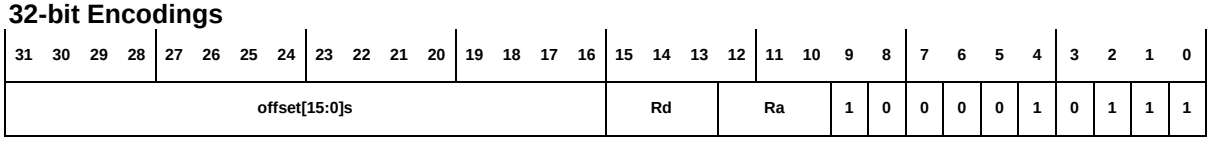


This encodes the instruction `ld.i Rd, @(offset, base)`

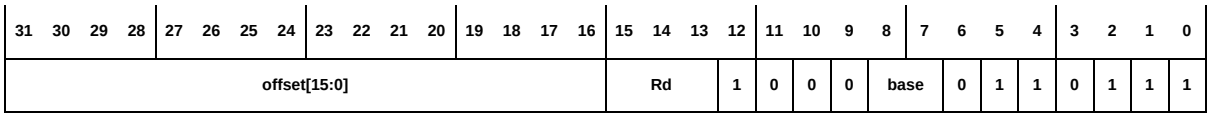
Base is encoded as follows:

encoding	base	Offset Type
0	0	Unsigned
1	%sp	Unsigned
2	%pc	Signed
3	%gp	Unsigned

32-bit Encodings



This encodes the instruction `ld.i Rd, @(offset, Ra)`

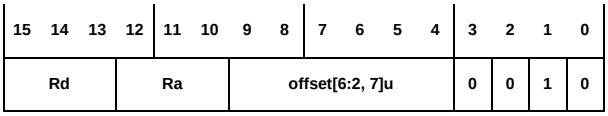


This encodes the instruction `ld.i Rd, @(offset, base)`

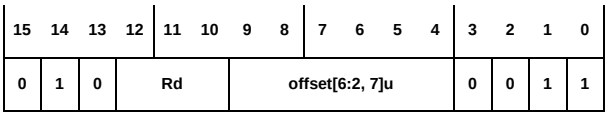
Base is encoded as follows:

encoding	base	Offset Type
0	0	Unsigned
1	%sp	Unsigned
2	%pc	Signed
3	%gp	Unsigned

16-bit Encodings



This encodes the instruction `ld.i Rd, @(offset, Ra)`



This encodes the instruction `ld.i Rd, @(offset, %sp)`

ld.r — Load, Indexed

Instruction

```
ld.r Rd, @(Rx, %sp)
ld.r Rd, @(Rx, Ra)
```

Description

Load 32-bit value from memory into `Rd` with indexed addressing.

Flags

Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

For the Ra-relative form:

```
addr = Rx + Ra
Rd = *(int32*)(addr)
```

For the SP-relative form:

```
addr = Rx + SP
Rd = *(int32*)(addr)
```

Usage Notes

Ra and Rx are interpreted as byte addresses.

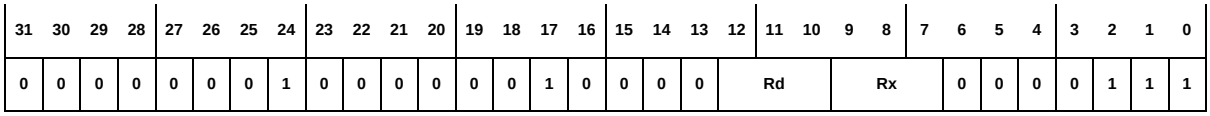
A 32-bit memory read is performed.

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

Examples

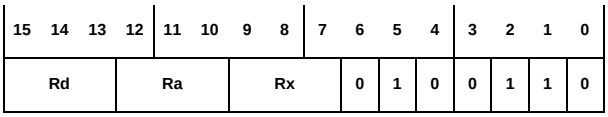
```
ld.r %r2, @(%r0, %r7)
ld.r %r1, @(%r3, %sp)
```

32-bit Encoding



This encodes the instruction `ld.r Rd, @(Rx, %sp)`

16-bit Encoding



This encodes the instruction `ld.r Rd, @(Rx, Ra)`

lic — Read licence number

Instruction `lic Rd`

Description Read the XAP6's licence number

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation `Rd = XAP6 licence number`

Usage Notes Each XAP6 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
0	0	0	0	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	Rd		0	0	0	0	0	1	1	1

mov.1.i — Move, single-bit, immediate

Instruction

```
mov.1.i %flags[i], #imm
```

Description

Set the interrupt flag to an immediate value

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
FLAGS[I] = #imm[0]
```

Usage Notes

This instruction can be used to enable and disable interrupts.

This instruction throws a `PrivInstruction` exception if it is executed in User mode

Refer to section 3.4.3, "FLAGS register" for the meaning of the interrupt flag.

Examples

```
mov.1.i %flags[i], #1   // Enable interrupts
mov.1.i %flags[i], #0   // Disable interrupts
```

16-bit Encoding

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

'im' encodes the immediate value

mov.1.r — Move, single-bit, register

Instruction

```
mov.1.r Rd, %flags[ZNCV]
mov.1.r Rd, %flags[i]
mov.1.r %flags[ZNCV], Rs
mov.1.r %flags[i], Rs
mov.1.r %flags[i], %flags[c]
mov.1.r %flags[c], %flags[i]
```

Description

Move between two flags, or between the specified flag and bit 0 of a register.

Flags

Z

Unchanged, except
1) when the destination is Rd, where the Z flag is set if the result is zero; cleared otherwise
2) when the destination is %flags[z], where this is set to the value of Rs[0]

N

Unchanged, except
1) when the destination is Rd, where the N flags is cleared
2) when the destination is %flags[n], where this is set to the value of Rs[0]

C

Unchanged, except when the destination is %flags[c], where the C flag is set to the value of Rs[0] or to the value of the I (interrupt) flag

V

Unchanged, except when the destination is %flags[v], where the V flag is set to the value of Rs[0]

Operation

Forms 1 and 2: `Rd[0] = FLAGS[F], Rd[31:1] = 0` F = z,n,c,v,i

Forms 3 and 4: `FLAGS[F] = Rs[0]` F = z,n,c,v,i

Form 5: `FLAGS[i] = FLAGS[c]`

Form 6: `FLAGS[c] = FLAGS[i]`

Usage Notes

This instruction can copy from the Z, N, C, V bits of the Flags register into bit 0 of r0-r7

This instruction can copy from bit 0 of r0-r7 into the Z,N,C,V, or I bits of the Flags register

This instruction can also copy between the C flag and the I flag

The two forms of the instruction which write to the I flag are privileged instructions, and will throw a `PrivInstruction` exception if executed in User mode

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 %flags[c], %flags[i]
```

32-bit Encodings

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	1	0	0	1	0	0	0	0	0	0	0	0	0	bit[1:0]		Rd		0	0	0		0	1	1	1

This encodes the instruction `mov.1.r Rd, %flags[bit]`

Values of bit are as follows:

Bit field value	Flag Modified
0	Z
1	N
2	C
3	V

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	1	0	0	1	0	0	0	0	1	0	0	0		Rs		0	bit[1:0]	0	0	0		0	1	1	1

This encodes the instruction `mov.1.r %flags[bit], Rs`

Values of bit are as above.

16-bit Encodings

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

This encodes the instruction `mov.1.r Rd, %flags[i]`

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

This encodes the instruction `mov.1.r %flags[i], Rs`

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

This encodes the instruction `mov.1.r %flags[i], %flags[c]`

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

This encodes the instruction `mov.1.r %flags[c], %flags[i]`

mov.2.i — Move, 2-bit, immediate

Instruction

```
mov.2.i %flags[m], #imm
```

Description

Set the mode flags to an immediate value.

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
FLAGS[M] = #imm[1:0]
```

Usage Notes

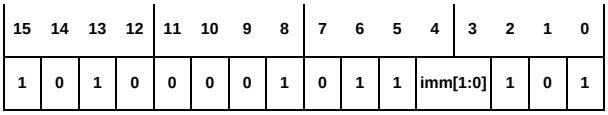
Refer to section section 3.4.3, "FLAGS register"for the encodings of the processor modes.

This instruction throws a `PrivInstruction` exception if executed 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
1	0	1	0	0	0	0	1	0	1	1	imm[1:0]		1	0	1

mov.2.r — Move, 2-bit, register

Instruction

```
mov.2.r Rd, %flags[m]
mov.2.r %flags[m], Rs
```

Description

Store the current mode in a register or set the mode flags to a value contained in a register

Flags

Z When the destination is Rd, Set if the result is zero; cleared otherwise
When the destination is %flags[m], Unchanged

N When the destination is Rd, Cleared
When the destination is %flags[m], Unchanged

C Unchanged

V Unchanged

Operation

```
Rd[1:0] = FLAGS[M] ; Rd[31:2] = 0
FLAGS[M] = Rd[1:0]
```

Usage Notes

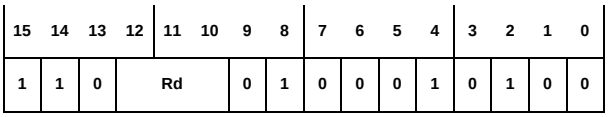
Refer to section 3.4.3, "FLAGS register" for the encodings of the processor modes.

The form of this instruction which writes to the M flags is a privileged instruction and will throw a `PrivInstruction` exception if executed in User mode.

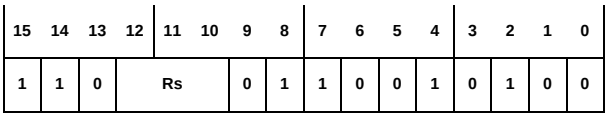
Examples

```
mov.2.r %r1, %flags[m]
mov.2.r %flags[m], %r1
```

16-bit Encodings



This encodes the instruction `mov.2.r Rd, %flags[m]`



This encodes the instruction `mov.2.r %flags[m], Rs`

mov.4.i — Move, 4-bit, immediate

Instruction `mov.4.i %flags[p], #imm`

Description Set the priority flags to an immediate value.

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation `FLAGS[P] = #imm[3:0]`

Usage Notes Refer to section 3.4.3, "FLAGS register" for the meaning of the priority flags.

This instruction throws a `PrivInstruction` exception if executed in User mode.

Examples

```
mov.4.i %flags[p], #0
mov.4.i %flags[p], #13
```

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	1	1	1	0	0	0	0	imm[3:0]u				0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1

mov.4.r — Move, 4-bit, register

Instruction

```
mov.4.r Rd, %flags[p]
mov.4.r %flags[p], Rs
```

Description

Store the current priority flags in a register, or set the priority flags to a value contained in a register

Flags

Z When the destination is Rd, Set if the result is zero; cleared otherwise
When the destination is %flags[p], Unchanged

N When the destination is Rd, Cleared
When the destination is %flags[p], Unchanged

C Unchanged

V Unchanged

Operation

```
Rd[3:0] = FLAGS[M] ; Rd[31:4] = 0
FLAGS[P] = Rd[3:0]
```

Usage Notes

Refer to section 3.4.3, "FLAGS register" for the encodings of the priorities.

The form of this instruction which writes to the P flags is a privileged instruction and will throw a `PrivInstruction` exception if executed in User mode.

Examples

```
mov.4.r %r1, %flags[p]
mov.4.r %flags[p], %r1
```

32-bit Encodings

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	1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	Rd		0	0	0	0	0	1	1	1

This encodes the instruction `mov.4.r Rd, %flags[p]`

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	1	1	0	0	0	0	0	0	1	0	0	0	0	Rs			0	0	0	0	0	0	0	1	1	1

This encodes the instruction `mov.4.r %flags[p], Rs`

mov.8.i — Move, 8-bit, immediate

Instruction `mov.8.i %flags[a], #imm`

Description Set the accumulator flags to an immediate value.

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
FLAGS[A] = #imm[7:0]
```

Usage Notes Refer to section 3.4.3, "FLAGS register" for the encodings of the accumulator flags.

This instruction throws a `PrivInstruction` exception if `executed` in User mode.

Examples

```
mov.4.i %flags[a], #0
mov.4.i %flags[a], #13
```

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	1	1	1	imm[7:0]u								0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	1

mov.8.r — Move, 8-bit, register

Instruction

```
mov.8.r Rd, %flags[a]
mov.8.r %flags[a], Rs
```

Description

Store the current accumulator flags in a register, or set the accumulator flags to a value contained in a register

Flags

Z When the destination is Rd, Set if the result is zero; cleared otherwise
When the destination is %flags[a], Unchanged

N When the destination is Rd, Cleared
When the destination is %flags[a], Unchanged

C Unchanged

V Unchanged

Operation

```
Rd[7:0] = FLAGS[A] ; Rd[31:8] = 0
FLAGS[A] = Rd[7:0]
```

Usage Notes

Refer to section 3.4.3, "FLAGS register" for the encodings of the accumulator flags.

Examples

```
mov.4.r %r1, %flags[a]
mov.4.r %flags[a], %r1
```

32-bit Encodings

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	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	Rd		0	0	0	0	0	1	1	1

This encodes the instruction `mov.8.r Rd, %flags[a]`

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	1	1	0	0	0	0	0	0	1	1	0	0	0	Rs			0	0	0	0	0	0	0	1	1	1

This encodes the instruction `mov.8.r %flags[a], Rs`

mov.f.r — Move, field, register

Instruction

```
mov.f.r Rd[msb:lsb], Rs
mov.f.r Rd, Rs[msb:lsb]
```

Description

Extracts a bitfield from a register, or inserts a bitfield into a register

Flags

Z	Set if Rd is zero; cleared otherwise
N	Set to have the value of Rd[31]
C	Unchanged
V	Unchanged

Operation

Rd[(msb-lsb):0] = Rs[msb:lsb], Rd[higher bits] = 0

Rd[msb:lsb] = Rs[(msb-lsb):0], Rd[other bits] = unchanged

Usage Notes

This instruction will raise an InstructionError exception if its arguments are out of range (msb > 31 or lsb > msb). The assembler will catch most out-of-range values as well.

Examples

```
mov.f.r %r0[4], %r1
mov.f.r %r0[12:4], %r1
mov.f.r %r0, %r1[4]
mov.f.r %r0, %r1[12:4]
```

32-bit Encodings

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	size[4:0]u					0	0	1	lsb[4:0]u					0	1	1	Rs				Rd	0	1	0	0	1	1	1	

This encodes the instruction `mov.f.r Rd[msb:lsb], Rs`
size = msb – lsb + 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	0	size[4:0]u					0	0	0	lsb[4:0]u					0	1	1	Rs				Rd	0	1	0	0	1	1	1	

This encodes the instruction `mov.f.r Rd, Rs[msb:lsb]`
size = msb – lsb + 1

16-bit Encoding

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

This encodes the instruction `mov.f.r Rd, Rs[msb:lsb]`

This encoding is only valid when Rs = Rd

The encoding of `field` is given by the following table:

Field	Size	MSB	LSB
0	2	5	4
1	1	1	1
2	1	2	2
3	1	3	3
4	1	4	4
5	1	8	8
6	1	16	16
7	1	24	24
8	4	7	4
9	4	11	8
10	4	15	12
11	4	19	16
12	4	23	20
13	4	27	24
14	8	15	8
15	8	23	16

mov.i — Move, displacement or immediate

Instruction

```
mov.i Rd, (offset, %pc)
mov.i Rd, (offset, %gp)
mov.i Rd, (imm, 0)
mov.i Rd, #imm

mov.i Rd, (label, %pc)
mov.i Rd, (label, %gp)
mov.i Rd, (label, 0)
mov.i Rd, !(label, %pc)
mov.i Rd, !(label, 0)
```

Description

Move 32-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[31:0]s + PC)
Rd = (void*)(offset[31:0]u + GP)
Rd = (void*)(offset[31:0]s + 0)
Rd = #immediate[31:0]
```

Usage Notes

When the Zero-relative or PC-relative forms are used, an `!` can precede the address formation. If it is used, then a function table entry is created for the label specified in the instruction and when executed the instruction will produce the address of the function table entry. It is intended that the `!` be used for code addresses, e.g. functions.

Examples

```
mov.i %r1, (label, %pc)
mov.i %r1, (label, %gp)
mov.i %r1, !(label, %pc)
mov.i %r1, (label, 0)
mov.i %r1, #0xFE12
```

48-bit Encodings

47 ... 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
offset[31:0]s	Rd 0	0 1 1 0	0 1 1 1	1 1 1 1

This encodes the instruction `mov.i Rd, (offset, %pc)`

47 ... 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
offset[31:0]u	Rd			0	1	0	1	0	0	1	1	1	1	1	1	1

This encodes the instruction `mov.i Rd, (offset, %gp)`

47 ... 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[31:0]s	Rd			0	0	0	1	0	0	1	1	1	1	1	1	1

This encodes the instruction `mov.i Rd, #imm`

32-bit Encodings

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[15:1,22]s				Rd				offset[21:16]s				0	1	0	1	1	1	1	

This encodes the instruction `mov.i Rd, (offset, %pc)`

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
offset[15:0]s				Rd			0	0	1	1	0	0	1	1	1	0	1	1	1

This encodes the instruction `mov.i Rd, (offset, %pc)`

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
offset[15:2, 23, 22]u				Rd				offset[21:16]u				1	0	0	1	1	1	1	

This encodes the instruction `mov.i Rd, (offset, %gp)`

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				Rd			0	1	0	1	0	0	1	1	1	0	1	1	1

This encodes the instruction `mov.i Rd, (offset, %gp)`

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
imm[15:0]s				Rd				imm[21:16]s				0	0	0	1	1	1	1	

This encodes the instruction `mov.i Rd, #imm`

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
imm[15:0]s				Rd			0	0	0	1	0	0	1	1	1	0	1	1	1

This encodes the instruction `mov.i Rd, #imm`

16-bit Encodings

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

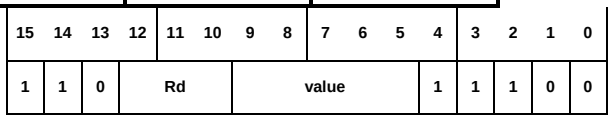
This encodes the instruction `mov.i Rd, #imm`

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

This encodes the instruction `mov.i Rd, #imm`

This instruction encodes immediates that are powers of 2, as follows:

Value in instruction	Immediate	Value in instruction	Immediate	Value in instruction	Immediate
1	#0x100	**2**	#0x1 0000	**3**	#0x100 0000
5	#0x200	**6**	#0x2 0000	**7**	#0x200 0000
9	#0x400	**10**	#0x4 0000	**11**	#0x400 0000
13	#0x800	**14**	#0x8 0000	**15**	#0x800 0000
17	#0x1000	**18**	#0x10 0000	**19**	#0x1000 0000
21	#0x2000	**22**	#0x20 0000	**23**	#0x2000 0000
25	#0x4000	**26**	#0x40 0000	**27**	#0x4000 0000
29	#0x8000	**30**	#0x80 0000	**31**	#0x8000 0000



This encodes the instruction `mov.i Rd, #imm`

This instruction encodes immediates that are powers of 2 minus 1, as follows:

Value in instruction	Immediate	Value in instruction	Immediate	Value in instruction	Immediate
1	#0xFFFF FFFF	**2**	#0xFFFF	**3**	#0xFF FFFF
5	#0x1FF	**6**	#0x1 FFFF	**7**	#0x1FF FFFF
9	#0x3FF	**10**	#0x3 FFFF	**11**	#0x3FF FFFF
13	#0x7FF	**14**	#0x7 FFFF	**15**	#0x7FF FFFF
17	#0xFFF	**18**	#0xF FFFF	**19**	#0xFFF FFFF
21	#0x1FFF	**22**	#0x1F FFFF	**23**	#0x1FFF FFFF
25	#0x3FFF	**26**	#0x3F FFFF	**27**	#0x3FFF FFFF
29	#0x7FFF	**30**	#0x7F FFFF	**31**	#0x7FFF FFFF

mov.r — Move, register

Instruction

```
mov.r Rd, Rs
mov.r %sp, Rs
mov.r Rd, %sp
```

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 Move a value from one register to another, or to and from the Stack Pointer

When the destination is the Stack Pointer, Flags are not updated

Examples

```
mov.r %r1, %r2
mov.r %sp, %r2
mov.r %r1, %sp
```

32-bit Encodings

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	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	Rd		0	0	0	0	0	1	1	1

This encodes the instruction `mov.r Rd, %sp`

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	1	1	0	0	0	0	0	0	0	0	0	0	0	Rs			0	0	0	0	0	0	0	1	1	1

This encodes the instruction `mov.r %sp, Rs`

16-bit Encoding

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

This encodes the instruction `mov.r Rd, Rs`

mova2r — Move address register to register

Instruction `mova2r Rd, As`

Description Read from an address 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
0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0	0	0	As				Rd				0	0	0	0	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 Read from a breakpoint 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.

This instruction throws a `PrivInstruction` exception if 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
0	0	0	0	0	0	0	1	0	0	0	1	1	1	0	1	0	0	0	Bs				Rd				0	0	0	0	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 Write 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, GP 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
0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1	0	0	0	Rs					Ad				0	0	0	0	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 Write 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.

This instruction throws a `PrivInstruction` exception if 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
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	Rs				Bd		0	1	0	0	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 Write to a special register

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation `Sd = Rs`

Usage Notes

Sd is one of the 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.

This instruction throws a `PrivInstruction` exception if executed in User mode.

If this instruction is used to write to the read-only INFO register, it has no effect.

Attempts to change %flags[s] will have 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
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Rs				Sd		0	1	0	0	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 Read from a special 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 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.

Reading the Flags register is permitted in any mode.

This instruction throws a `PrivInstruction` exception if executed in User mode, unless Ss is 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
0	0	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	Ss				Rd		0	0	0	0	1	1	1

See section 6.8, "Special register fields" for the encoding of Ss.

msbit.r — Most significant bit, register

Instruction `msbit.r Rd, Rs`

Description Set Rd to the position of the highest 1 in Rs

Flags

Z	Set if the result is zero, cleared otherwise
N	Cleared
C	Unchanged
V	Unchanged

Operation `Rd = (1 + highest bit to contain a 1 in Rs)`

Usage Notes The instruction calculates the minimum number of right shifts required to make Rs = 0.

It should be used on positive or unsigned numbers.

The values of Rd for example values of Rs are shown below:

Rs	Rd
0	0
1	1
2	2
4	3
7	3
8	4
0x80	8
0x400	11
0x800	12
0x7FFF	15
0x8000	16
0xFFFF	16
0xFFFFFFFF	32

Examples `msbit.r %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
0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	0	0	0	Rs				Rd				0	0	0	0	1	1	1

mult.16s.i — Multiply, 16-bit signed, immediate

Instruction `mult.16s.i Rd, Rs, #imm`

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

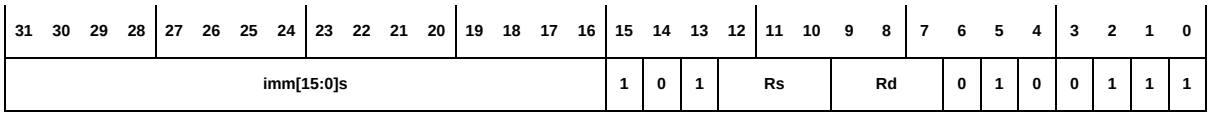
Operation `Rd = Rs[15:0] * #imm[15:0]s`

Usage Notes This is a signed 16x16 multiply, giving a 32-bit product.

Both operands are treated as 16-bit signed variables and represent numbers in the range -32768 to +32767.

Examples `mult.16s.i %r1, %r2, #0x1234`

32-bit Encoding



mult.16s.r — Multiply, 16-bit signed, register

Instruction `mult.16s.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 `Rd = Rs[15:0] * Rs[15:0]`

Usage Notes This is a signed 16x16 multiply, giving a 32-bit product.

Both operands are treated as 16-bit signed variables and represent numbers in the range -32768 to +32767.

Examples `mult.16s.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
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Rd				Rs				Rt				0	0	0	0	1	1	1

mult.16u.i Multiply, 16-bit unsigned, immediate

Instruction `mult.16u.i Rd, Rs, #imm`

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 the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

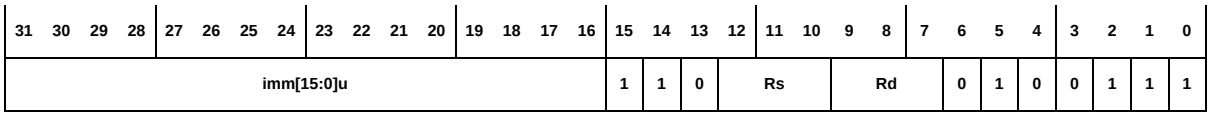
Operation `Rd = Rs[15:0] * #imm[15:0]u`

Usage Notes This is an unsigned 16x16 multiply, giving a 32-bit product.

Both operands are treated as 16-bit unsigned variables and represent numbers in the range 0 to 65535.

Examples `mult.16u.i %r1, %r2, #0x1234`

32-bit Encoding



mult.16u.r — Multiply, 16-bit unsigned, register

Instruction

```
mult.16u.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 the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation

```
Rd = Rs[15:0] * Rs[15:0]
```

Usage Notes

This is an unsigned 16x16 multiply, giving a 32-bit product.

Both operands are treated as 16-bit unsigned variables and represent numbers in the range 0 to 65535.

Examples

```
mult.16u.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
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	Rd				Rs				Rt				0	0	0	0	1	1	1

mult.i — Multiply, immediate

Instruction `mult.i Rd, Rs, #imm`

Description 32-bit by 32-bit integer multiply to give low 32-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 * #imm[15:0]s`

Usage Notes This is a 32x32 multiply, giving the low 32 bits of the product in Rd.

Examples

```
mult.i %r1, %r2, #0x12345678
mult.i %r1, %r2, #0xFEDCBA98
```

48-bit Encoding

47 ... 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
imm[31:0]s	Rd	Rs	0 1 1 1 0 1	1 1 1 1

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
imm[15:0]s				Rd	Rs	0 1 1 1 0 1	0 1 1 1

mult.r — Multiply, register

Instruction `mult.r Rd, Rs, Rt`

Description 32-bit by 32-bit integer multiply to give low 32-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 32x32 multiply, giving the low 32 bits of the product in Rd.

Examples

```
mult.r %r1, %r2, %r3
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
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	Rd				Rs				Rt				0	0	0	0	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
1	0	1	0	0	0	0	1	0	0	1	1	0	1	0	1

or.i — OR, immediate

Instruction `or.i Rd, Rs, #imm`

Description Bitwise Or of Rs with an immediate 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 | #imm[32:0]u`

Usage Notes –

Examples

```
or.i %r1, %r2, #0x12345678
or.i %r1, %r2, #0xFEDCBA98
```

48-bit Encoding

47 ... 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
imm[31:0]u	Rd	Rs	0 0 1 1	0 1 1 1 1

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
imm[15:0]u				Rd	Rs	0 0 1 1	0 1 0 1 1

16-bit Encodings

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

This encoding is only valid when Rs = Rd

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

This encodes the instruction `or.i Rd, Rs, #1`

or.r — OR, register

Instruction `or.r Rd, Rs, Rt`

Description Bitwise Or of two registers

Flags

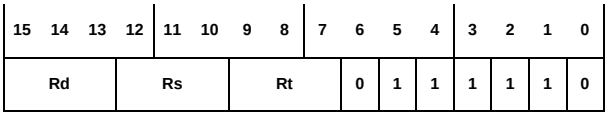
Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

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 5.1.4, "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, and section 6.1.3, "Stack Operations" for examples of pushes and pops.

The offset can take values in the range 0, 4, …, 248, 252.

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`+4n, 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[7:2]u		0	0	0	1	0	0	0 0 0 1	0 0 1 0	0 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	regmask[7:0]		1	0	1	

This encodes the instruction `pop RegList, #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 5.1.4, "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, and section 6.1.3, "Stack Operations" for examples of pushes and pops.

The offset can take values in the range 0, 4, …, 248, 252.

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`+4n+4, 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.

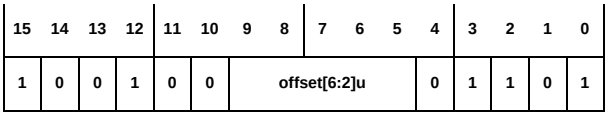
Examples

```
pop.ret {%r3-%r7}, #0
pop.ret {%r1, %r4-%r6}, #4
```

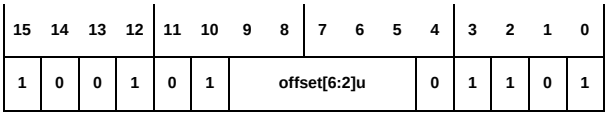
32-bit Encoding

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[7:2]u		0	0	0	1	0	0	0	1	0	1	0	0	1	0	0	1	1	1

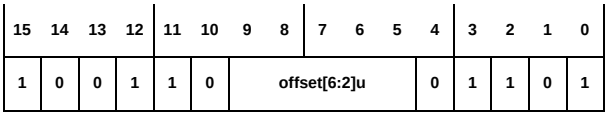
16-bit Encodings



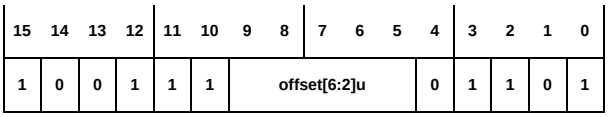
This encodes the instruction `pop.ret {%r4}, #offset`



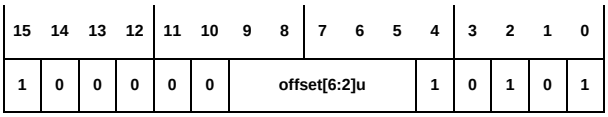
This encodes the instruction `pop.ret {%r4-%r5}, #offset`



This encodes the instruction `pop.ret {%r4-%r6}, #offset`



This encodes the instruction `pop.ret {%r4-%r7}, #offset`



This encodes the instruction `pop.ret {}, #offset`

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 XAP6 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
0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	Rs			0	0	0	0	0	0	0	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

This instruction can be used to open a stack frame. 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 popped and can contain registers in the range R0 to R7. Refer to section 5.1.4, "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, and section 6.1.3, "Stack Operations" for examples of pushes and pops.

The offset can take values in the range 0, 4, …, 248, 252.

The operation sequence is:

- The stack pointer is decreased by `#offset+4n`, 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[7:2]u		0 0	0 1 0 0	1 0 0 1	0 0 1 0	0 1 1 1

16-bit Encodings

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

This encodes the instruction `push RegList, #0`

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

This encodes the instruction `push {%r4}, #offset`

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

This encodes the instruction `push {%r5-%r4}, #offset`

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

This encodes the instruction `push {%r6-%r4}, #offset`

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

This encodes the instruction `push {%r7-%r4}, #offset`

push.i — Push immediates 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 to the stack

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

Push immediates to stack, i0 first to i3 (if present) last.

```
SP = SP - 4
*(int32*) SP = #i0[31:0]
```

```
SP = SP - 4
*(int32*) SP = #i0[15:0]u
SP = SP - 4
*(int32*) SP = #i1[15:0]u
```

```
SP = SP - 4
*(int32*) SP = #i0[7:0]u
SP = SP - 4
*(int32*) SP = #i1[7:0]u
SP = SP - 4
*(int32*) SP = #i2[7:0]u
[ SP = SP - 4
*(int32*) SP = #i3[7:0]u ]
```

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	Any 32-bit integer
2	-32768 to 32767
3	-128 to 127
4	-128 to 127

Refer to section 6.3.3, "Push and Pop" for further details of the `push.i` instruction, and section 6.1.3, "Stack Operations" for further details of stack

operations.

The operation sequence is:

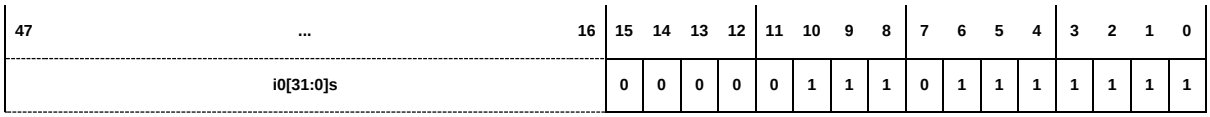
- The stack pointer is decreased by 4n, where n is the number of immediates given.
- Store the specified immediates. Earlier immediates are stored first and to higher stack addresses. Later immediates are stored last and to lower stack addresses.

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

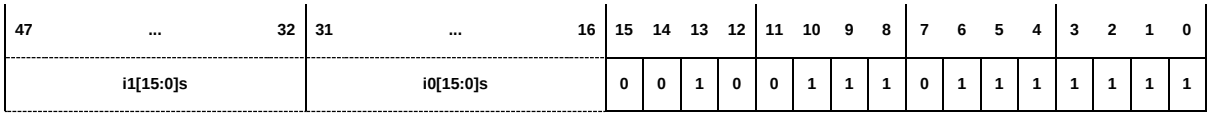
Examples

```
push.i {#0x12345678}, #0
push.i {#12, #3, #9, #0}, #0
```

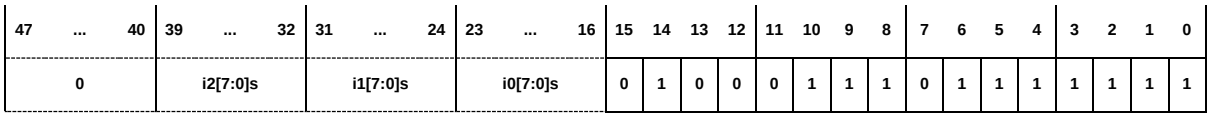
48-bit Encodings



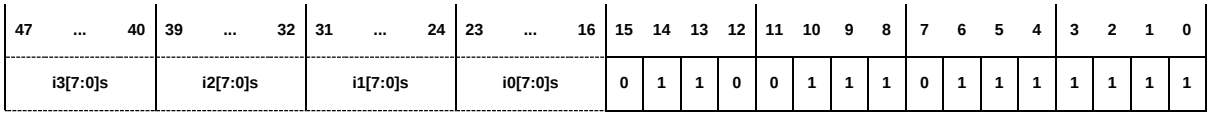
This encodes the instruction `push.i {#i0}, #0`



This encodes the instruction `push.i {#i0, #i1}, #0`

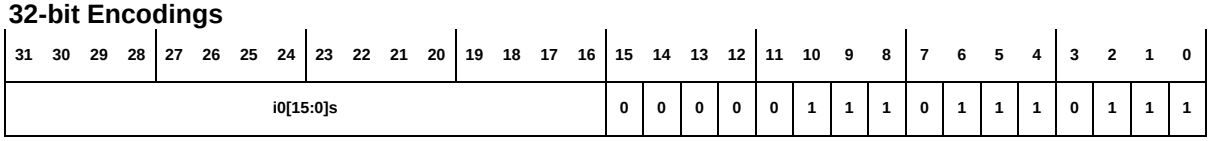


This encodes the instruction `push.i {#i0, #i1, #i2}, #0`

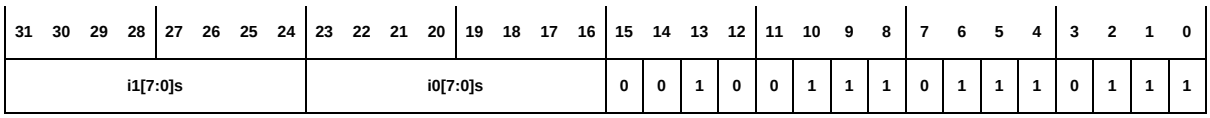


This encodes the instruction `push.i {#i0, #i1, #i2, #i3}, #0`

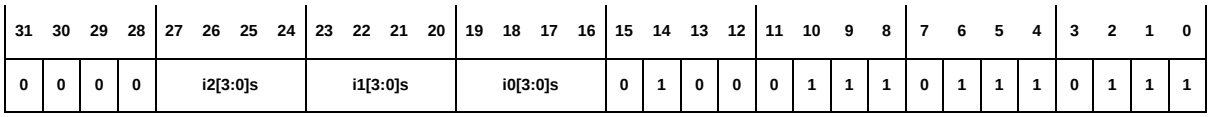
32-bit Encodings



This encodes the instruction `push.i {#i0}, #0`



This encodes the instruction `push.i {#i0, #i1}, #0`



This encodes the instruction `push.i {#i0, #i1, #i2}, #0`

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
i3[3:0]s	i2[3:0]s	i1[3:0]s	i0[3:0]s	0	1	1	0	0	1	1	1	0	1	1	1	0	1	1	1

This encodes the instruction `push.i {#i0, #i1, #i2, #i3}, #0`

16-bit Encodings

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	0	1	i0	1	0	1

This encodes the instruction `push.i {#i0}, #0`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	0	1	i1	i0	1	0	1

This encodes the instruction `push.i {#i0, #i1}, #0`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	0	1	i2	i1	i0	1	0	1

This encodes the instruction `push.i {#i0, #i1, #i2}, #0`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	1	i3	i2	i1	i0	1	0	1

This encodes the instruction `push.i {#i0, #i1, #i2, #i3}, #0`

rem.s.r — Remainder, signed, register

Instruction `rem.s.r Rd, Rs, Rt`

Description 32-bit by 32-bit signed divide to give a 32-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

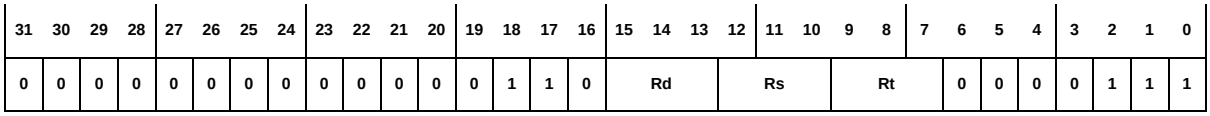
Operation `Rd = Rs % Rt`

Usage Notes If the (discarded) quotient cannot be represented in 32 bits, the overflow flag is set and the remainder is set to zero.

See section 0, "Divide by zero" for the effects of dividing by zero.

Examples `rem.s.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
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	Rd				Rs				Rt				0	0	0	0	1	1	1

rem.u.r — Remainder, unsigned, register

Instruction

```
rem.u.r Rd, Rs, Rt
```

Description

32-bit by 32-bit signed divide to give a 32-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 = Rs % Rt
```

Usage Notes

If the (discarded) quotient cannot be represented in 32 bits, the carry flag is set and the remainder is set to zero.

See section 0, "Divide by zero" for the effects of dividing by zero.

Examples

```
rem.u.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
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	Rd				Rs				Rt				0	0	0	0	1	1	1

rotatel.i — Rotate left, immediate

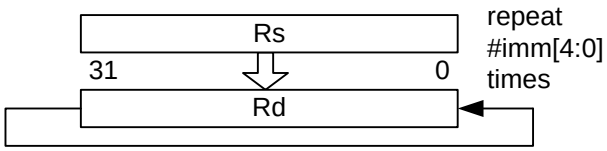
Instruction `rotatel.i Rd, Rs, #imm`

Description 32-bit rotate left

Flags

Z	Set if the result is zero; cleared otherwise
N	The N flag is the final value of bit 31 of Rd
C	The C flag 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.

The immediate value must be between 0 and 31.

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
0	0	0	0	0	1	0	0	0	1	1	imm[4:0]u					0	0	0	Rs				Rd				0	0	0	0	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	Rd			1	1	imm[4:0]u				1	0	1	

This encoding is only valid when Rs = Rd

rotatel.r — Rotate left, register

Instruction

```
rotatel.r Rd, Rs, Rt
```

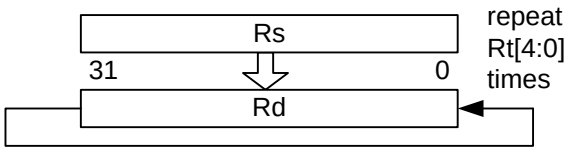
Description

Rotate left

Flags

Z	Set if the result is zero; cleared otherwise
N	The N flag is the final value of bit 31 of Rd
C	The C flag 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.

Rt[4:0] specifies the number of bits to rotate. Rt[31:5] is ignored.

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
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	Rd				Rs				Rt				0	0	0	0	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` exception if executed in User mode.

Usage Notes

The null event 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
1	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1

sext.16.r — Sign extend, 16-bit, register

Instruction `sext.16.r Rd, Rs`

Description Sign extend from 16 bits to 32 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[31:16] = Rd[15]`

Usage Notes This instruction sign-extends a 16-bit value by copying the sign bit into the top 16 bits.

Examples `sext.16.r %r1, %r2`

16-bit Encoding

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

sext.8.r — Sign extend, 8-bit, register

Instruction `sext.8.r Rd, Rs`

Description Sign extend from 8 bits to 32 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[31:8] = Rd[7]`

Usage Notes This instruction sign-extends an 8-bit value by copying the sign bit into the top 24 bits.

Examples `sext.8.r %r1, %r2`

16-bit Encoding

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

shiftl.c.i — Shift left, with carry

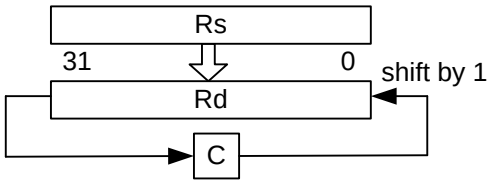
Instruction `shiftl.c.i Rd, Rs, #1`

Description Shift left with carry by 1

Flags

Z	Set if the result is zero; cleared otherwise
N	The N flag is the final value of bit 31 of Rd
C	The N flag is the initial value of bit 31 of Rs
V	Unchanged

Operation



Rd[0] is set to the value of the C flag. The C flag is set to the value of Rs[31]. All other bits in Rs are shifted left by 1.

Usage Notes

Examples `shiftl.c.i %r1, %r2, #1`

16-bit Encoding

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

shiftl.i — Shift left, immediate

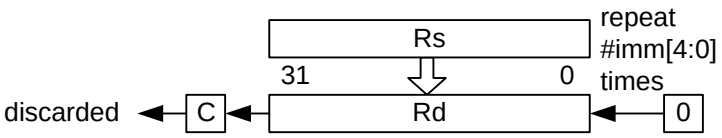
Instruction `shiftl.i Rd, Rs, #imm`

Description Shift left

Flags

Z	Set if the result is zero; cleared otherwise
N	The N flag is the final value of bit 31 of Rd
C	The C flag contains the last bit shifted out
V	Unchanged

Operation



Usage Notes

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.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
0	0	0	0	0	1	0	0	0	1	0	imm[4:0]u					0	0	0	Rs				Rd				0	0	0	0	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	Rd			1	0	imm[4:0]u				1	0	1	

This encoding is only valid when Rs = Rd

shiftl.r — Shift left, register

Instruction

```
shiftl.r Rd, Rs, Rt
```

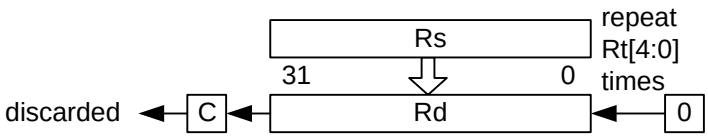
Description

Shift left

Flags

Z	Set if the result is zero; cleared otherwise
N	The N flag is the final value of bit 31 of Rd
C	The C flag contains the last bit shifted out
V	Unchanged

Operation



Usage Notes

Rt[4:0] specifies the number of bits to shift. Rt[31:5] is ignored.

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.c.i — Shift right, with carry

Instruction

```
shiftr.c.i Rd, Rs, #1
```

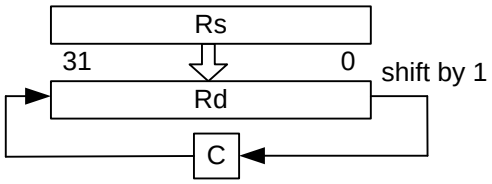
Description

Shift right with carry by 1

Flags

Z	Set if the result is zero; cleared otherwise
N	The N flag is the final value of bit 31 of Rd
C	The N flag is the initial value of bit 0 of Rs
V	Unchanged

Operation



Rd[31] is set to the value of the C flag. The C flag is set to the value of Rs[0]. All other bits in Rs are shifted right by 1.

Usage Notes

Examples

```
shiftr.c.i %r1, %r2, #1
```

16-bit Encoding

shiftr.s.i — Shift right, signed, immediate

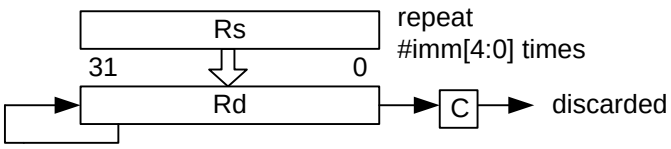
Instruction `shiftr.s.i Rd, Rs, #imm`

Description Signed right shift

Flags

Z	Set if the result is zero; cleared otherwise
N	The N flag is the final value of bit 31 of Rd
C	The C flag contains the last bit shifted out
V	Unchanged

Operation



Usage Notes 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.s.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
0	0	0	0	0	1	0	0	0	0	0	imm[4:0]u					0	0	0	Rs				Rd				0	0	0	0	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	Rd			0	0	imm[4:0]u				1	0	1	

This encoding is only valid when Rs = Rd

shiftr.s.r — Shift right, signed, register

Instruction

```
shiftr.s.r Rd, Rs, Rt
```

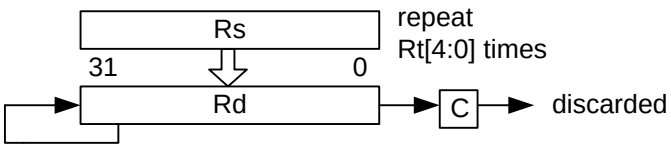
Description

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

Rt[4:0] specifies the number of bits to shift. Rt[31:5] is ignored.

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, #imm
```

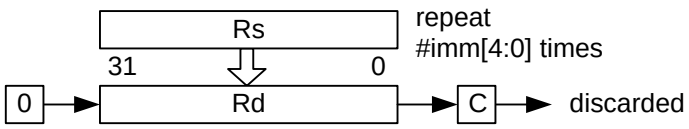
Description

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

The immediate value must be 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.u.i %r1, %r2, #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
0	0	0	0	0	1	0	0	0	0	1	imm[4:0]u					0	0	0	Rs				Rd				0	0	0	0	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	Rd				0	1	imm[4:0]u				1	0	1

This encoding is only valid when Rs = Rd

shiftr.u.r — Shift right, unsigned, register

Instruction

```
shiftr.u.r Rd, Rs, Rt
```

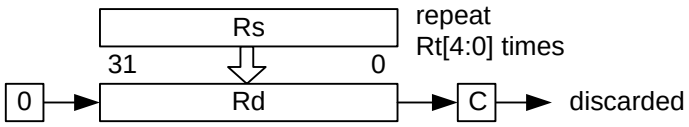
Description

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

Rt[4:0] specifies the number of bits to shift. Rt[31:5] is ignored.

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
   throw PrivInstruction exception
else
   allow SIF access
endif
```

Usage Notes The `sif` instruction allows the XAP6 to perform a SIF cycle.

This instruction throws a `PrivInstruction` exception if executed in User mode.

Examples

```
sif
```

16-bit Encoding

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

sleepnop — Sleep

Instruction

`sleepnop`

Description

Put the XAP6 into NOP Sleep state

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if (FLAGS[M] == UserMode) then
   throw PrivInstruction exception
else
   put XAP6 into NOP Sleep state
endif
```

Usage Notes

Put the XAP6 into the NOP Sleep state. SIF cycles are not allowed in the NOP Sleep state. xIDE is unable to gain access to XAP6 in this state.

This instruction throws a `PrivInstruction` exception if 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
1	0	1	0	0	0	0	1	0	0	1	0	0	1	0	1

sleepsif — Sleep and allow SIF

Instruction `sleepsif`

Description Put the XAP6 into SIF Sleep state

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
if (FLAGS[M] == UserMode) then
   throw PrivInstruction exception
else
   put XAP6 into SIF Sleep mode
endif
```

Usage Notes Put the XAP6 into the SIF Sleep state. SIF cycles are allowed in the SIF Sleep state.

This instruction throws a `PrivInstruction` exception if 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
1	0	1	0	0	0	0	1	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 PrivInstruction error
else
   throw SoftReset
endif
```

Usage Notes This instruction throws a `PrivInstruction` exception if 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
1	0	1	0	0	0	0	1	0	0	0	1	1	1	0	1

st.16.i Store, 16-bit, displacement

Instruction

```
st.16.i Rm, @(offset, Ra)
st.16.i Rm, @(offset, %pc)
st.16.i Rm, @(offset, %sp)
st.16.i Rm, @(offset, %gp)
st.16.i Rm, @(offset, 0)
st.16.i Rm, @(label, %pc)
st.16.i Rm, @(label, %gp)
st.16.i Rm, @(label, 0)
```

Description

Store a 16-bit value to memory with displacement addressing.

The source, Rm, may be one of the following:

- Normal Register - %r0 to %r7
- Immediate #0, #1, #0xFFFF

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
*(int16*)(address) = source[15:0]
```

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.

Examples

```
st.16.i %r1, @(28, %r7)
st.16.i #1, @(label, %pc)
st.16.i #1, @(label, %gp)
st.16.i %r3, @(23, %pc)
st.16.i #0, @(0, %sp)
st.16.i %r6, @(label, 0)
st.16.i #-1, @(0x8100, 0)
```

Encoding Data

In the encodings, `base` is encoded as follows:

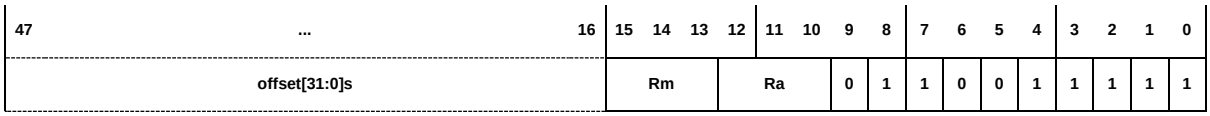
base value in instruction	base	Offset Type
0	0	Unsigned

1	%sp	Unsigned
2	%pc	Signed
3	%gp	Unsigned

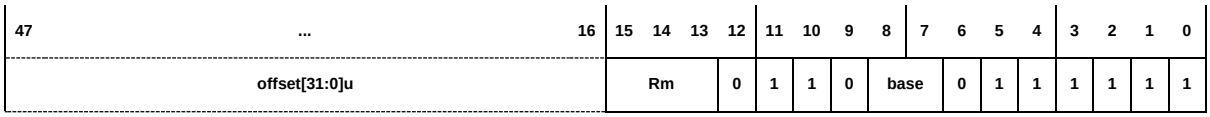
For the instructions where Rm is an immediate, the immediate to be stored is encoded as follows:

value A	value B	Immediate
0	0	0
0	1	1
1	0	0xFFFF

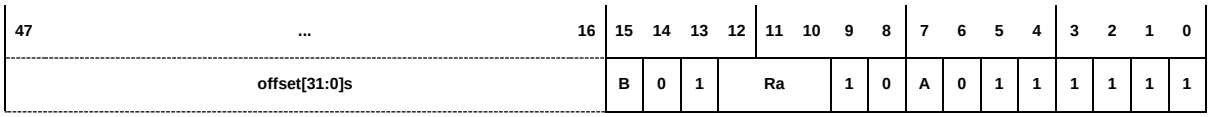
48-bit Encodings



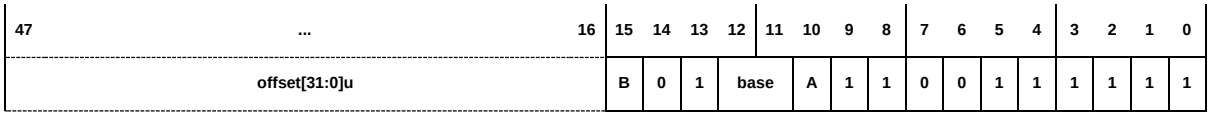
This encodes the instruction `st.16.i Rm, @(offset, Ra)`



This encodes the instruction `st.16.i Rm, @(offset, base)`

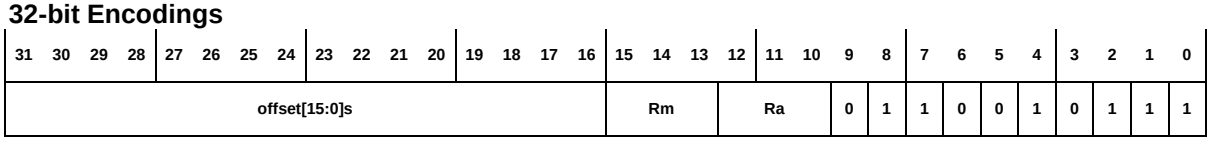


This encodes the instruction `st.16.i #imm, @(offset, Ra)`

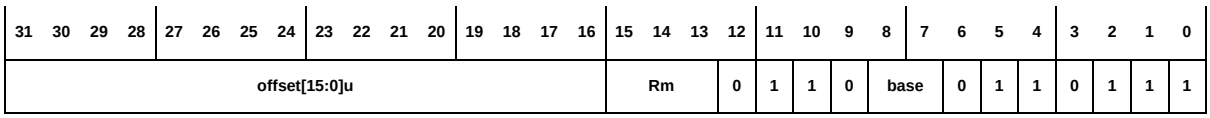


This encodes the instruction `st.16.i #imm, @(offset, base)`

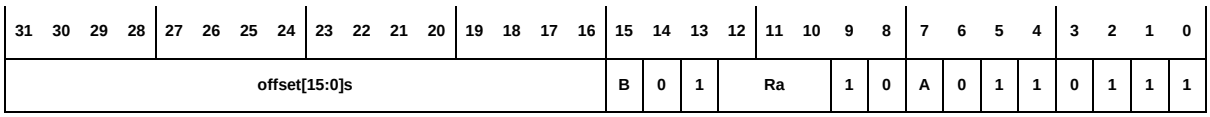
32-bit Encodings



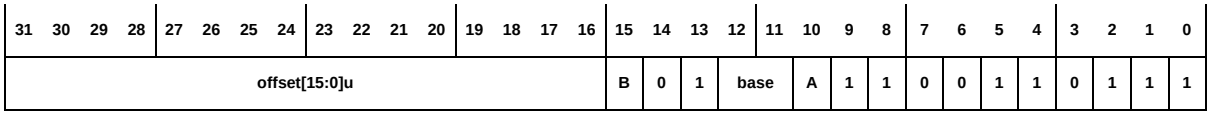
This encodes the instruction `st.16.i Rm, @(offset, Ra)`



This encodes the instruction `st.16.i Rm, @(offset, base)`

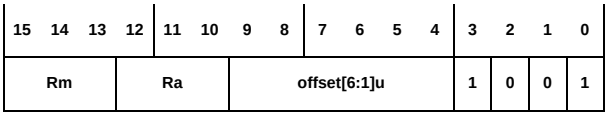


This encodes the instruction `st.16.i #imm, @(offset, Ra)`

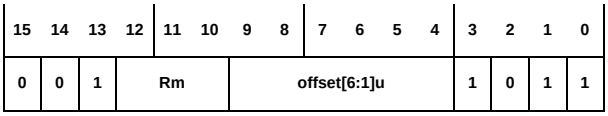


This encodes the instruction `st.16.i #imm, @(offset, base)`

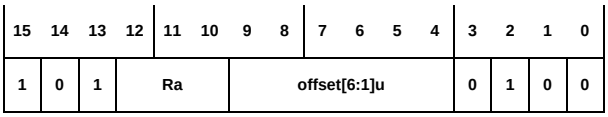
16-bit Encodings



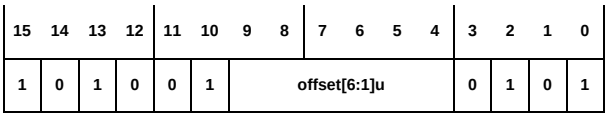
This encodes the instruction `st.16.i Rm, @(offset, Ra)`



This encodes the instruction `st.16.i Rm, @(offset, %sp)`



This encodes the instruction `st.16.i #0, @(offset, Ra)`



This encodes the instruction `st.16.i #0, @(offset, %sp)`

st.16.r Store, 16-bit, indexed

Instruction

```
st.16.r Rm, @(Rx, Ra)
st.16.r Rm, @(Rx, %sp)
```

Description

Store a 16-bit value to memory with displacement addressing.

The source, Rm, may be one of the following:

- Normal Register - %r0 to %r7
- Immediate #0, #1, #0xFFFF

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
*(int16*)(address) = source[15:0]
```

Usage Notes

Rx is interpreted as a byte offset from Ra or SP

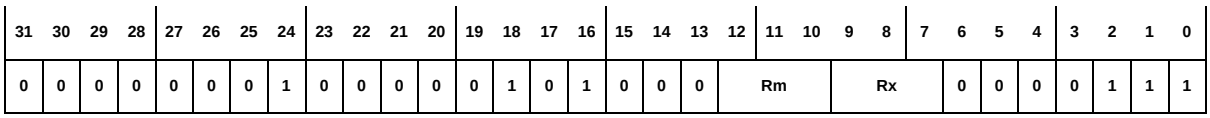
A 16-bit memory write is performed.

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

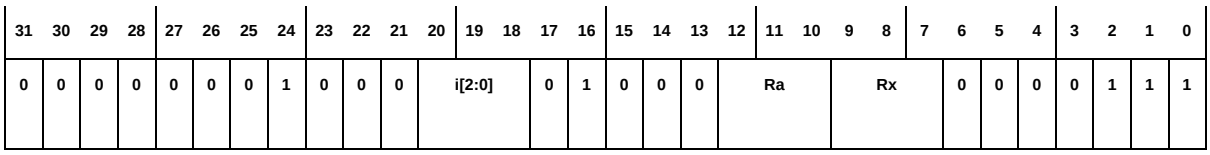
Examples

```
st.16.r %r1, @(%r2, %r7)
st.16.r #1, @(%r3, %sp)
```

32-bit Encodings



This encodes the instruction `st.16.r Rm, @(Rx, %sp)`



This encodes the instruction `st.16.r #imm, @(Rx, Ra)`

The immediate to be stored is encoded as follows:

i[2:0]	Immediate
2	0
3	1
4	0xFFFF

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	1	0	0	0	0	0	i[1:0]		0	1	0	0	0	0	0	0	Rx		0	0	0	0	0	1	1	1

This encodes the instruction `st.16.r #imm, @(Rx, %sp)`

The immediate to be stored is encoded as follows:

i[1:0]	Immediate
0	0
1	1
2	0xFFFF

16-bit Encoding

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

This encodes the instruction `st.16.r Rm, @(Rx, Ra)`

st.8.i Store, 8-bit, displacement

Instruction

```
st.8.i Rm, @(offset, Ra)
st.8.i Rm, @(offset, %pc)
st.8.i Rm, @(offset, %sp)
st.8.i Rm, @(offset, %gp)
st.8.i Rm, @(offset, 0)
st.8.i Rm, @(label, %pc)
st.8.i Rm, @(label, %gp)
st.8.i Rm, @(label, 0)
```

Description

Store an 8-bit value to memory with displacement addressing.

The source, Rm, may be one of the following:

- Normal Register - %r0 to %r7
- Immediate #0, #1, #0xFF

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
*(int8*)(address) = source[7:0]
```

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.

Examples

```
st.8.i %r1, @(28, %r7)
st.8.i #1, @(label, %pc)
st.8.i #1, @(label, %gp)
st.8.i %r3, @(23, %pc)
st.8.i #0, @(0, %sp)
st.8.i %r6, @(label, 0)
st.8.i #-1, @(0x8100, 0)
```

Encoding Data

In the encodings, `base` is encoded as follows:

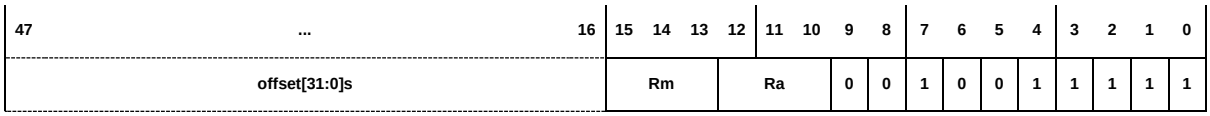
base value in instruction	base	Offset Type
0	0	Unsigned

1	%sp	Unsigned
2	%pc	Signed
3	%gp	Unsigned

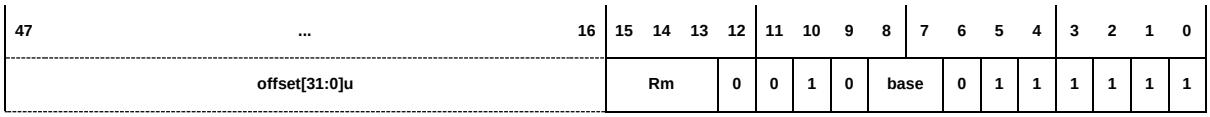
For the instructions where Rm is an immediate, the immediate to be stored is encoded as follows:

value A	value B	Immediate
0	0	0
0	1	1
1	0	0xFF

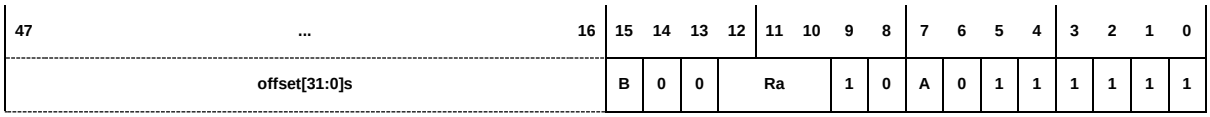
48-bit Encodings



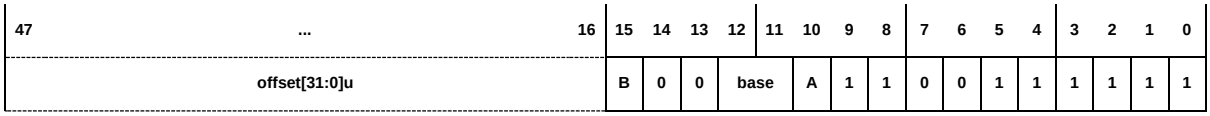
This encodes the instruction `st.8.i Rm, @(offset, Ra)`



This encodes the instruction `st.8.i Rm, @(offset, base)`

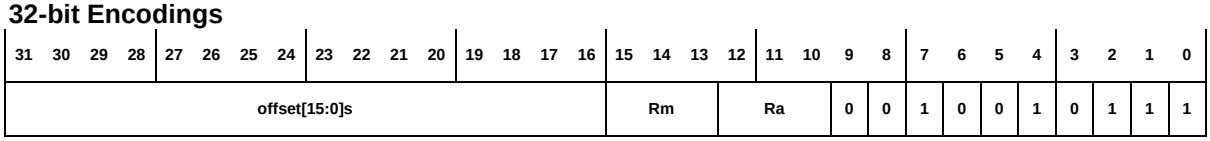


This encodes the instruction `st.8.i #imm, @(offset, Ra)`

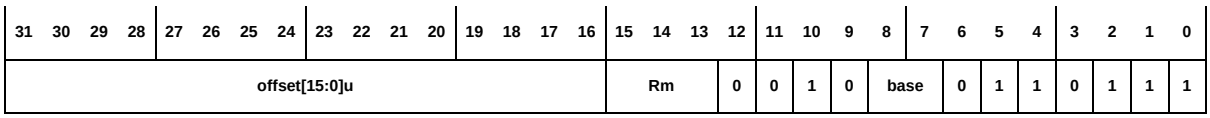


This encodes the instruction `st.8.i #imm, @(offset, base)`

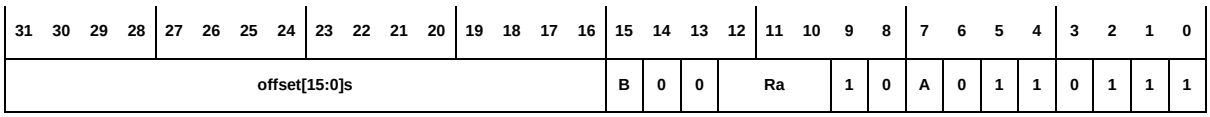
32-bit Encodings



This encodes the instruction `st.8.i Rm, @(offset, Ra)`



This encodes the instruction `st.8.i Rm, @(offset, base)`



This encodes the instruction `st.8.i #imm, @(offset, Ra)`

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																B	0	0	base					A	1	1	0	0	1	1	0	1	1	1

This encodes the instruction `st.8.i #imm, @(offset, base)`

16-bit Encodings

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

This encodes the instruction `st.8.i Rm, @(offset, Ra)`

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

This encodes the instruction `st.8.i Rm, @(offset, %sp)`

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

This encodes the instruction `st.8.i #0, @(offset, Ra)`

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

This encodes the instruction `st.8.i #0, @(offset, %sp)`

st.8.r Store, 8-bit, indexed

Instruction

```
st.8.r Rm, @(Rx, Ra)
st.8.r Rm, @(Rx, %sp)
```

Description

Store an 8-bit value to memory with displacement addressing.

The source, Rm, may be one of the following:

- Normal Register - %r0 to %r7
- Immediate #0, #1, #0xFF

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
*(int8*)(address) = source[7:0]
```

Usage Notes

Rx is interpreted as a byte offset from Ra or SP

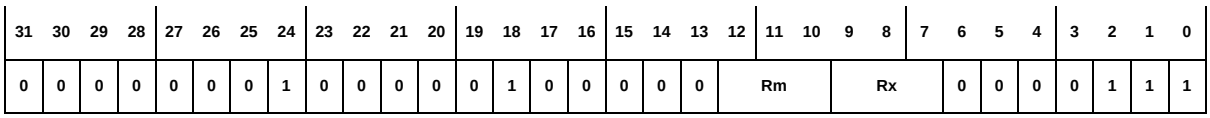
An 8-bit memory write is performed.

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

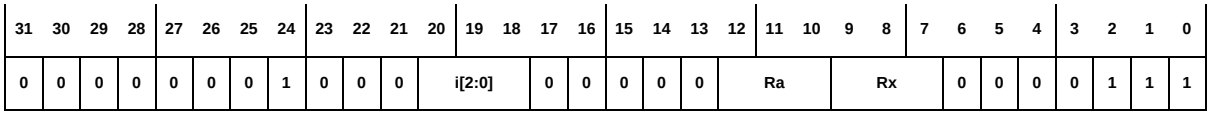
Examples

```
st.8.r %r1, @(%r2, %r7)
st.8.r #1, @(%r3, %sp)
```

32-bit Encodings



This encodes the instruction `st.8.r Rm, @(Rx, %sp)`



This encodes the instruction `st.8.r #imm, @(Rx, Ra)`

The immediate to be stored is encoded as follows:

i[2:0]	Immediate
2	0
3	1
4	0xFF

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	1	0	0	0	0	0	i[1:0]		0	0	0	0	0	0	0	0	Rx		0	0	0	0	0	1	1	1

This encodes the instruction `st.8.r #imm, @(Rx, %sp)`

The immediate to be stored is encoded as follows:

i[1:0]	Immediate
0	0
1	1
2	0xFF

16-bit Encoding

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

This encodes the instruction `st.8.r Rm, @(Rx, Ra)`

st.i — Store, displacement

Instruction

```
st.i Rm, @(offset, Ra)
st.i Rm, @(offset, %pc)
st.i Rm, @(offset, %sp)
st.i Rm, @(offset, %gp)
st.i Rm, @(offset, 0)
st.i Rm, @(label, %pc)
st.i Rm, @(label, %gp)
st.i Rm, @(label, 0)
```

Description

Store a 16-bit value to memory with displacement addressing.

The source, Rm, may be one of the following:

- Normal Register - %r0 to %r7
- Immediate #0, #1, #0xFFFF FFFF

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
*(int32*)(address) = source[31:0]
```

Usage Notes

Ra and offset are interpreted as byte addresses.

A 32-bit memory write is performed.

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

Examples

```
st.i %r1, @(28, %r7)
st.i #1, @(label, %pc)
st.i #1, @(label, %gp)
st.i %r3, @(23, %pc)
st.i #0, @(0, %sp)
st.i %r6, @(label, 0)
st.i #-1, @(0x8100, 0)
```

Encoding Data

In the encodings, `base` is encoded as follows:

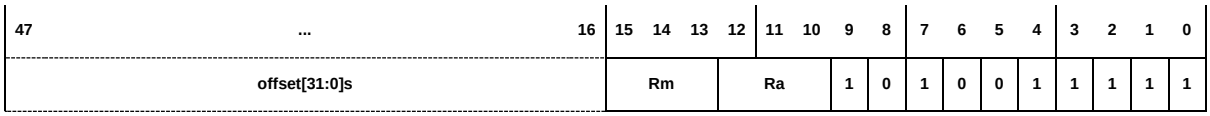
base value in instruction	base	Offset Type
0	0	Unsigned

1	%sp	Unsigned
2	%pc	Signed
3	%gp	Unsigned

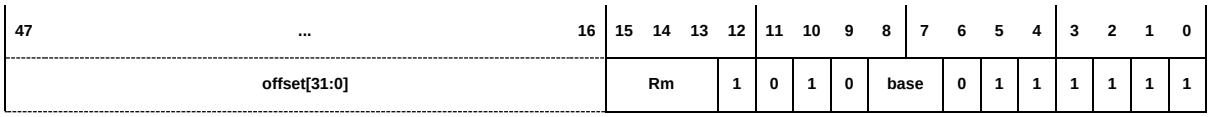
For the instructions where Rm is an immediate, the immediate to be stored is encoded as follows:

value A	value B	Immediate
0	0	0
0	1	1
1	0	0xFFFF FFFF

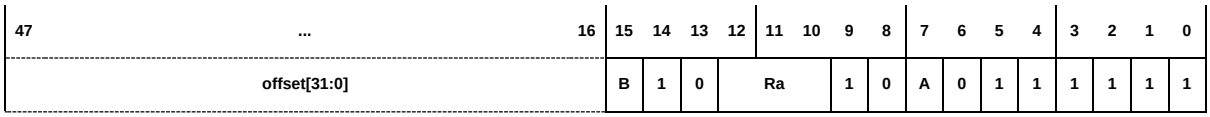
48-bit Encodings



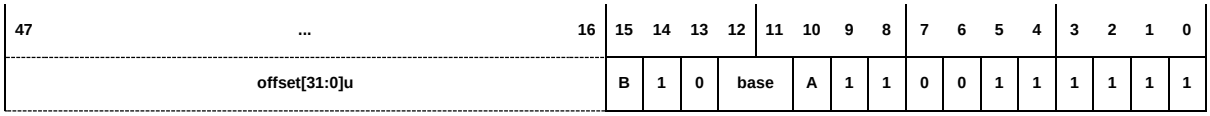
This encodes the instruction `st.i Rm, @(offset, Ra)`



This encodes the instruction `st.i Rm, @(offset, base)`

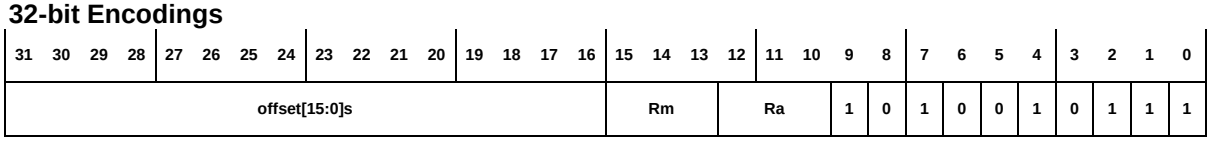


This encodes the instruction `st.i #imm, @(offset, Ra)`

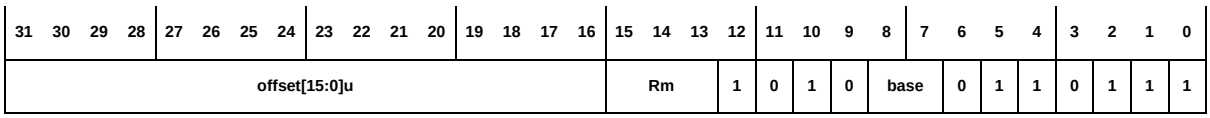


This encodes the instruction `st.i #imm, @(offset, base)`

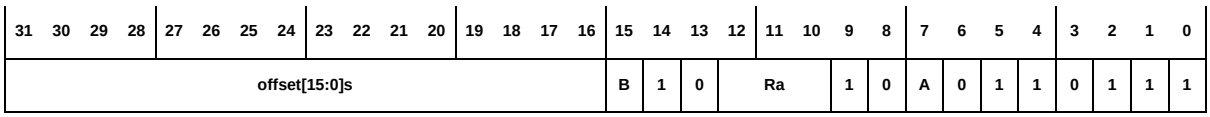
32-bit Encodings



This encodes the instruction `st.i Rm, @(offset, Ra)`



This encodes the instruction `st.i Rm, @(offset, base)`



This encodes the instruction `st.i #imm, @(offset, Ra)`

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																B	1	0	base					A	1	1	0	0	1	1	0	1	1	1

This encodes the instruction `st.i #imm, @(offset, base)`

16-bit Encodings

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rm				Ra				offset[6:2, 7]u				1	0	1	0

This encodes the instruction `st.i Rm, @(offset, Ra)`

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

This encodes the instruction `st.i Rm, @(offset, %sp)`

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

This encodes the instruction `st.i #0, @(offset, Ra)`

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

This encodes the instruction `st.i #0, @(offset, %sp)`

st.r — Store, indexed

Instruction

```
st.r Rm, @(Rx, Ra)
st.r Rm, @(Rx, %sp)
```

Description

Store a 32-bit value to memory with displacement addressing.

The source, Rm, may be one of the following:

- Normal Register - %r0 to %r7
- Immediate #0, #1, #0xFFFF FFFF

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation

```
*(int32*)(address) = source[31:0]
```

Usage Notes

Rx is interpreted as a byte offset from Ra or SP

A 32-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)
```

32-bit Encodings

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	1	0	0	0	0	0	1	1	0	0	0	0	Rm				Rx				0	0	0	0	1	1	1

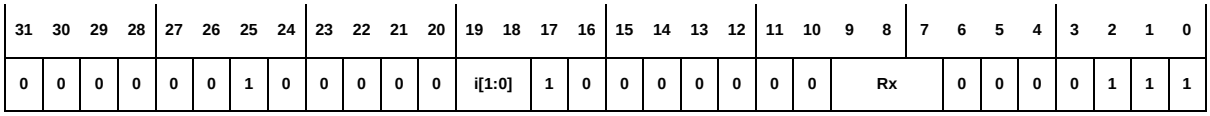
This encodes the instruction `st.r Rm, @(Rx, %sp)`

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	1	0	0	0	i[2:0]			1	0	0	0	0	Ra				Rx				0	0	0	0	1	1	1

This encodes the instruction `st.r #imm, @(Rx, Ra)`

The immediate to be stored is encoded as follows:

i[2:0]	Immediate
2	0
3	1
4	0xFFFF FFFF

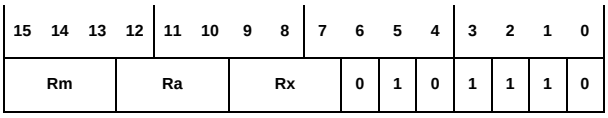


This encodes the instruction `st.r #imm, @(Rx, %sp)`

The immediate to be stored is encoded as follows:

i[1:0]	Immediate
0	0
1	1
2	0xFFFF FFFF

16-bit Encoding



This encodes the instruction `st.r Rm, @(Rx, Ra)`

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

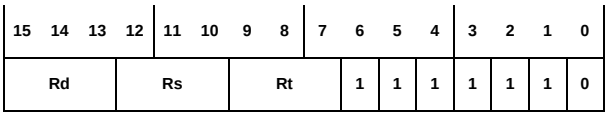
`sub.c.r` can be used for signed or unsigned, integer or fixed-point arithmetic.

The Z flag behaviour is useful for 64-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

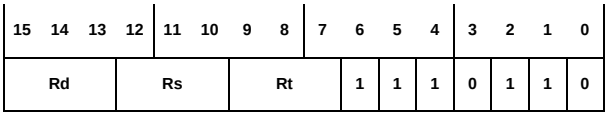
Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Set if the result of the unsigned operation is incorrect; cleared otherwise
V	Set if the result of the signed operation is incorrect; cleared otherwise

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



sub.x.i — Subtract, exchange, immediate

Instruction `sub.x.i Rd, Rs, #imm`

Description Exchanged order subtract immediate[31:0]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

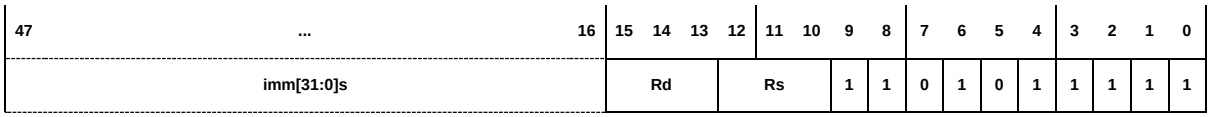
Operation `Rd = #immediate[31:0]s - Rs`

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

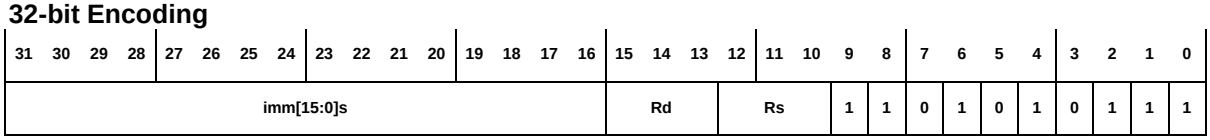
Examples

```
sub.x.i %r1, %r2, #0x12345678
sub.x.i %r1, %r2, #0xFEDC
```

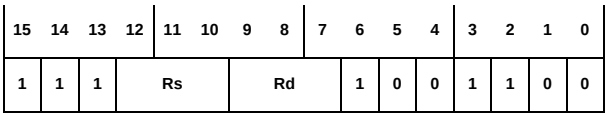
48-bit Encoding



32-bit Encoding



16-bit Encoding



This encodes the instruction `sub.x.i Rd, Rs, #0`

sub.xc.i Subtract, exchange, with carry, immediate

Instruction `sub.xc.i Rd, Rs, #imm`

Description Exchanged order subtract immediate[31:0]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[31:0]s - Rs - C`

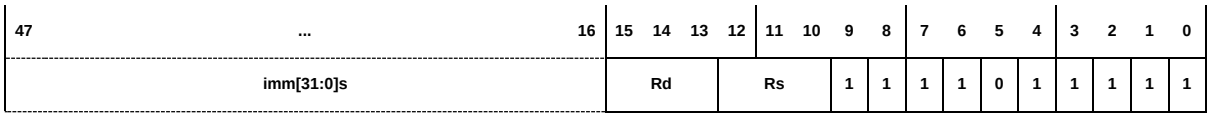
Usage Notes `sub.xc.i` can be used for signed or unsigned, integer or fixed-point arithmetic.

The Z flag behaviour is useful for 64-bit arithmetic.

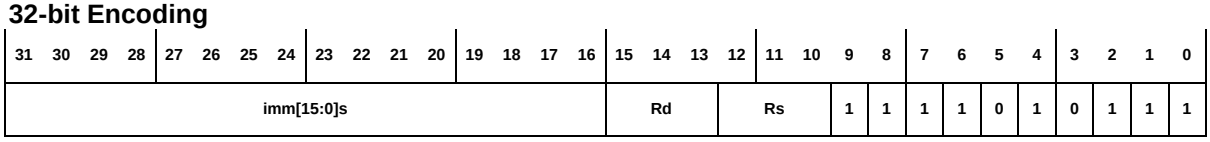
Examples

```
sub.xc.i %r1, %r2, #0x12345678
sub.xc.i %r1, %r2, #0xFEDC
```

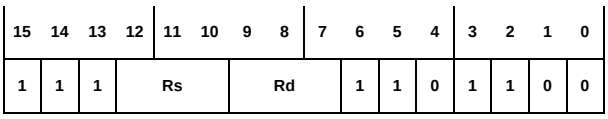
48-bit Encoding



32-bit Encoding



16-bit Encoding



This encodes the instruction `sub.xc.i Rd, Rs, #0`

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 32 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
0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	1	0	0	0	Ra					Rd				0	0	0	0	1	1	1

syscall.i — System call, immediate

Instruction

```
syscall.i num, #imm
```

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[31:0]
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 any 32 bit value.

Examples

```
syscall.i 0, #0x0123
syscall.i 3, #0xFEDC
```

48-bit Encoding

47 ... 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[31:0]s	0	num		1	0	0	1	0	0	1	1	1	1	1	1	1

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
imm[15:0]s																0	num			1	0	0	1	0	0	1	1	1	0	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	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	Rs			0	num		0	0	0	0	1	1	1

ver — Read version number

Instruction `ver Rd`

Description Read the XAP6's version number

Flags

Z	Unchanged
N	Unchanged
C	Unchanged
V	Unchanged

Operation `Rd = XAP6 version number`

Usage Notes This instruction reads the XAP6 version number.

The format of this number is:

Bits	Meaning
31:24	Reserved. 0 for this processor.
23:20	XAP Architecture major number. 6 for this processor.
19:16	XAP Architecture minor number. 0 for this processor.
15:12	Hardware Type. 0 for this processor.
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
0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	Rd		0	0	0	0	0	1	1	1

xor.i XOR (exclusive-or), immediate

Instruction `xor.i Rd, Rs, #imm`

Description Exclusive-OR with an immediate 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[31:0]u`

Usage Notes –

Examples

```
xor.i %r1, %r2, #0x12345678
xor.i %r1, %r2, #0xFEDC
```

48-bit Encoding

47 ... 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
imm[31:0]u	Rd	Rs	0 1 0 1	1 1 1 1

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
imm[15:0]u				Rd	Rs	0 1 0 1	0 1 1 1

16-bit Encodings

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

This encoding is only valid when Rs = Rd

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

This encodes the instruction `xor.i Rd, Rs, #1`

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

This encodes the instruction `xor.i Rd, Rs, #0xFFFFFFFF`

xor.r — XOR (exclusive-or), register

Instruction `xor.r Rd, Rs, Rt`

Description Bitwise XOR (exclusive-or) of two registers

Flags

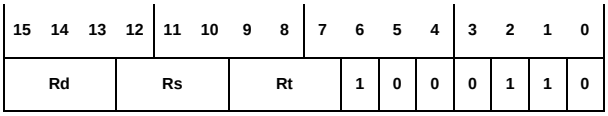
Z	Set if the result is zero; cleared otherwise
N	Set if the result is negative; cleared otherwise
C	Unchanged
V	Unchanged

Operation `Rd = Rs ^ Rt`

Usage Notes -

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

16-bit Encoding



Index