TECHNICAL MANUAL

ZSP400
Digital Signal Processor Architecture

December 2001

LSI LOGIC®

Document DB14-000121-03, Fourth Edition (December 2001)
This document describes LSI Logic Corporation's ZSP400 Digital Signal Processing Architecture and will remain the official reference source for all revisions/releases of this product until rescinded by an update.

GL

To receive product literature, visit us at http://www.lsilogic.com.

For a current list of our distributors, sales offices, and design resource centers, view our web page located at

http://www.lsilogic.com/contacts/na_salesoffices.html

Preface

This book is the primary reference and Technical Manual of the LSI Logic ZSP400 Digital Signal Processor Architecture. It contains a functional description of the architecture and details the instruction set.

Audience

This document assumes that you have some familiarity with microprocessors and related support devices. The people who benefit from this book are:

- Engineers and managers who are evaluating the ZSP400 architecture for possible use in a system
- Engineers who are designing a device based on the ZSP400 architecture into a system
- Engineers who are programming a device based on the ZSP400 architecture

Organization

This document has the following chapters and appendixes:

- Chapter 1, **Introduction**, introduces the features of the ZSP400 DSP architecture and the instruction set.
- Chapter 2, **ZSP400 Architecture Overview**, briefly describes the functional blocks that make up a ZSP400 device.
- Chapter 3, **Control Registers**, describes the control registers and mode bits of a ZSP400 device.
- Chapter 4, **Pipeline Control Unit**, describes the pipeline operation, the control register file, interrupts, and instruction grouping.

- Chapter 5, **Instruction Unit**, describes the instruction control unit, which is responsible for fetching instructions from memory and forwarding them to the pipeline.
- Chapter 6, **Data Unit**, describes the data unit, which is responsible for fetching data from memory and forwarding it to the pipeline. The data unit also handles data linking.
- Chapter 7, **Execution Unit**, describes the arithmetic logic units and multiply/accumulate units.
- Chapter 8, **ZSP400 Instruction Set**, describes the ZSP400 instruction set in detail.

Related Publications

LSI402Z Digital Signal Processor User's Guide, document number DB15-000131-01

Conventions Used in This Manual

The first time a word or phrase is defined in this manual, it is *italicized.*

The word *assert* means to drive a signal true or active. The word *deassert* means to drive a signal false or inactive. Signals that are active LOW end in an "n."

Hexadecimal numbers are indicated by the prefix "0x" —for example, 0x32CF. Binary numbers are indicated by the prefix "0b" —for example, 0b0011.0010.1100.1111.

Contents

Chapter 8 ZSP400 Instruction Set

Customer Feedback

Figures

Tables

Chapter 1
Introduction

This chapter introduces the ZSP400 digital signal processing architecture. It contains the following sections:

1.1 ZSP400 Architecture Overview

The ZSP400 architecture offers software engineers, system designers, and ASIC developers a new avenue for high performance programmable DSP solutions. The ZSP400 architecture is based on a RISC architecture and utilizes a superscalar approach. External peripherals can easily interface with the ZSP400 device, enabling complex systems.

Highlights of the ZSP400 architecture include:

- RISC-based superscalar architecture
  - Execution of multiple instructions per cycle
  - Hardware scheduling

    Programmers write serial code without worrying about parallelism.
  - Interlocked pipeline

    The pipeline controls stalls; software handling of pipeline conflicts is not required.
  - Static branch prediction

    Programmers do not need to code branch delay slots.

- Load/store architecture
  - Memory operations use load and store instructions

    Data moves from memory to registers—operations do not take place directly on memory.
  - All other instructions are register to register operations

    Manipulating registers saves memory bandwidth.
  - Variety of load/store instructions optimizes memory operations

    The architecture supports both double precision (32 bit) and single precision (16 bit) transfers to memory.
  - Flexible architecture allows result forwarding

    A functional unit's result can be used by any functional unit in the next cycle without penalty.
- Data Linking
  - Data linking keeps the data cache filled for continuous data streams
  - Linking allows streaming operands to bypass loading into a general purpose register

    Three general purpose registers support linking.
  - Contents of index registers used for linking automatically updated
- Memory structure
  - Simple, contiguous data space with memory-mapped I/O
  - Data cache and Instruction cache enhance performance and lower power dissipation
  - Data and instruction cache prefetchers allow deterministic operation
  - Extended precision operands can reside anywhere in data memory without any alignment restrictions
- Register file
  - Sixteen 16-bit general-purpose registers

    Two 16-bit register pairs can be combined into a single 32-bit register.

  - Accumulator support

    Two register pairs can be used as accumulators, each with a separate 8-bit guard.
  - Minimal special-purpose registers
  - Full support for data movement from any register to any other register
  - Any instruction can specify any general purpose registers as the source

---

1.2 Instruction Set Highlights

The ZSP400 instruction set provides very powerful instructions while maximizing processor execution speed. The load/store architecture de-couples memory access functionality from the instructions. All instructions can use any of the general purpose registers as the source.

A multiply or multiply-and-accumulate operation requires an accumulator destination register. All other instructions' results can be stored in any general purpose register.

The instruction set features:

- Single word (16-bit) compact instructions
- Single cycle execution of:
  - Any two 16-bit ALU operations
  - Any 32-bit ALU operation
  - Two 16-bit X 16-bit MUL with a single 40-bit accumulation
  - One 32-bit X 32-bit MUL with 40-bit accumulation
  - Exponent detection for 16/32-bit variables
  - Squaring of 16/32-bit variables
  - Majority of the basic functions defined by ETSI for speech coding applications
- Two parallel 16-bit additions or subtractions in the MAC units support ALU intensive code

- Excellent support for compare-select operations
  - Single cycle compare select instructions that facilitates two cycle Viterbi butterfly operation
  - Single cycle 16/32-bit minimum and maximum instructions
- 16-bit complex multiplication or multiply-accumulate using two instructions
- Extensive bit manipulation instructions
  - Bit reversal support
  - Logical and Arithmetic shift support
  - True arithmetic shift left instructions for 16/32 bit variables
  - Bit set, bit clear, and bit invert instructions for all registers
- Conditional branch support with specific prediction direction
- Double word (32 bit) load and store instructions
  - Provides high register-save bandwidth for context switches
  - Optimizes prolog/epilog code
- Load/store with short immediate offset instructions
  - Simplified stack and structure accesses
- Two to four hardware loops (specific implementations)
  - Hardware loops have zero overhead once set up.
- Fast and simple context switching support
  - Excellent store capabilities
  - User visibility into all registers
- Two hardware circular buffers
  - Circular buffers can have any starting and ending address. No special alignment is required.
- Move contents of any register to any other register
- Secondary (shadow) register bank (specific implementations)
  - 8 additional registers selectable via mode bit.
- Reverse carry indexing for FFT algorithms

The ZSP400 architecture is very compiler-friendly due to its RISC instruction set and the orthogonal set of general-purpose register instructions.

1.3 Available Implementations

The ZSP400 architecture is available in three forms: a component of the LSI Logic CoreWare® library, application-specific standard products (ASSP), and a licensable core, giving designers maximum flexibility in system design.

As a component in the LSI Logic CoreWare library, ZSP400 parts enable complex system-on-a-chip designs to take advantage of a world class DSP. The embedded in-circuit emulation capabilities and standard JTAG interface allow easy debug of system solutions containing several processor cores. Furthermore, the ZSP400 superscalar architecture provides assembly level software engineers with a simple programming model. This simple programming model leads to easier software tool development, where high level language compilers can take advantage of the highly orthogonal instruction set.

For applications not requiring a core approach, the ZSP400 family is also available in application specific standard products. System designers can use a standard product part knowing that it can be integrated into a system-on-a-chip solution later without losing their valuable software investments.

Chapter 2
ZSP400 Architecture Overview

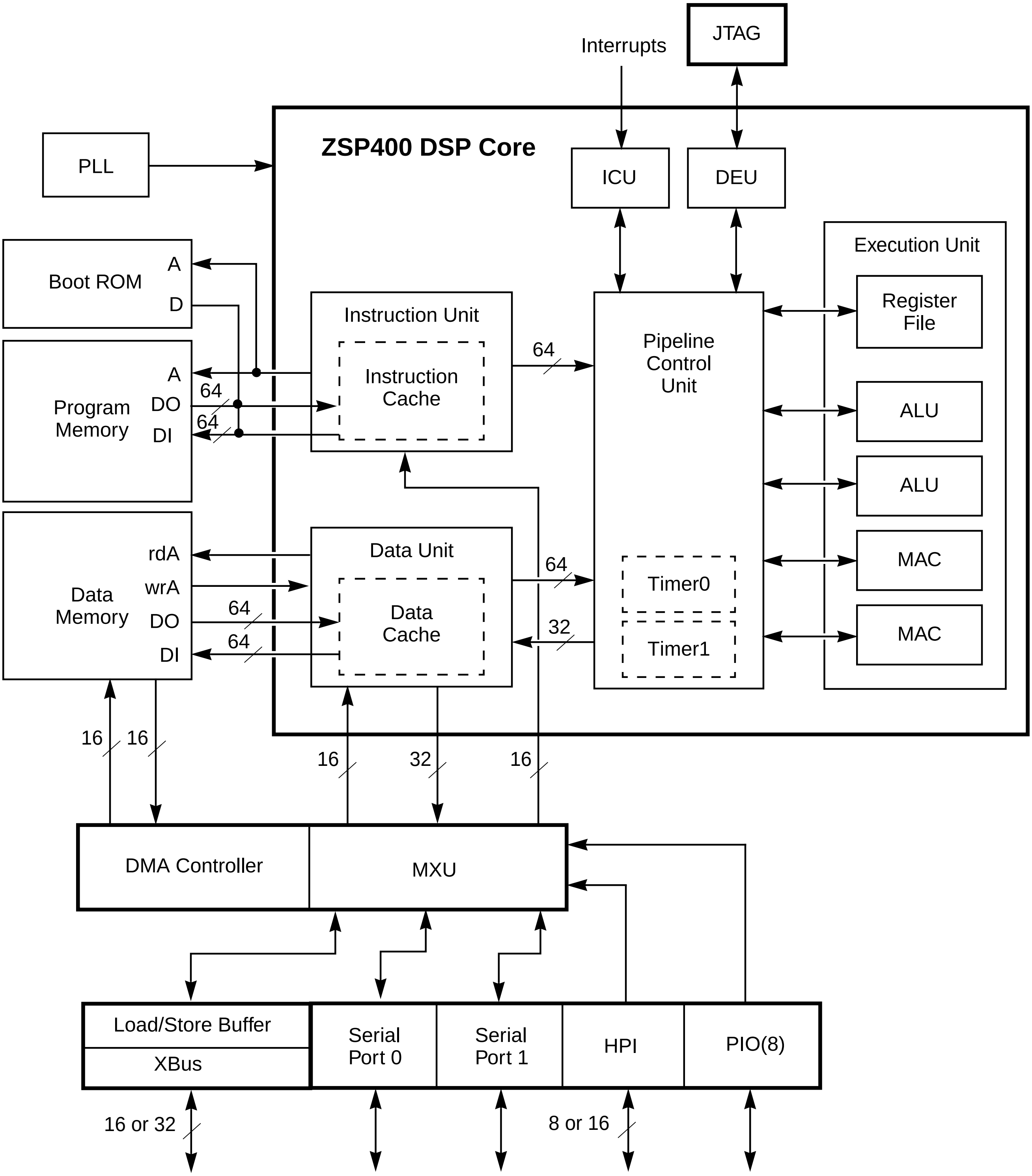
This chapter provides an overview of the ZSP400 digital signal processing architecture. It contains the following sections:

2.1 Typical ZSP400 System

Figure 2.1 is a block diagram of a typical ZSP400 system, the LSI402ZX Digital Signal Processor. The block diagram shows the ZSP400 Core, which is used in every implementation of the ZSP400 architecture, and the peripherals and memory logic that are combined with the core to implement a complete DSP device.

The JTAG Controller, PLL, Boot ROM, Program Memory, Data Memory, DMA Controller, MXU, and peripheral modules, which are not part of the ZSP400 Core, are included in Figure 2.1 to illustrate a typical system.

Figure 2.1 System Block Diagram



MXU = External Memory Interface Unit
HPI = Host Processor Interface
PIO = Programmable I/O

ICU = Interrupt Control Unit
DEU = Device Emulation Unit
XBus = External Bus

2.2 Control Register File

The ZSP400 architecture contains a set of control registers, used for mode control, status, and flag information. The ZSP400 architecture allows for 32 16-bit control registers.

2.3 Pipeline Control Unit

The pipeline control unit (PCU) receives instructions during the fetch/decode stage of the pipeline. The PCU checks for dependencies and grouping, and forwards instructions to the data unit. Only instructions that can execute in parallel are forwarded. Out-of-order execution is not allowed.

The pipeline control unit notifies the instruction unit (IU) which four instructions are needed for the next group. The data unit (DU) reads up to two 32 bit operands, and sends the operands to the execution unit (EXU). The EXU performs the necessary operation and writes the results to a general purpose register or sends the results back to the data unit to store in memory. Memory writes occur in the writeback stage of the pipeline.

The interrupt control unit (ICU) interfaces with the PCU. A nonmaskable interrupt (NMI) pin into the core allows for a separate interrupt control unit.

2.4 Instruction Unit

The instruction unit contains the instruction cache, instruction prefetcher, branch prediction logic, and an instruction dispatcher.

The instruction cache aligns instructions from main memory cache lines and reduces main memory power consumption. The prefetcher keeps the instruction cache full when running from on-chip memory and minimizes pipeline stalls. The branch predictor minimizes the need to flush the pipeline.

The instruction unit always fetches four instructions from the instruction cache. The instruction dispatcher decodes four instructions. The dispatcher issues up to four instructions to the data unit and the pipeline control unit each cycle. The data and pipeline control units read the required operands from registers or memory and execute the instructions.

2.5 Data Unit

The data unit contains the data cache, data prefetcher, and the circular buffer unit. The data unit is also responsible for data linking, a powerful concept that alleviates loads of operands from memory into general purpose registers before they can be used.

The data cache aligns operands from main memory cache lines and reduces main memory power consumption. The prefetcher keeps the data cache full when running from on-chip memory and minimizes pipeline stalls.

2.6 Execution Unit

The execution unit performs all the arithmetic and logical operations in the DSP. The execution unit contains two 16 bit arithmetic logic units (ALUs), two 16 X 16 multiply and accumulate (MAC) units, and a general purpose register file.

The ZSP400 architecture supports two identical 16 bit arithmetic logic units (ALU), which can be combined as a single 32 bit ALU. The MAC units can perform two 16-bit X 16-bit multiply operations followed by a single 40-bit accumulation or one 32-bit X 32-bit multiply followed by a 40-bit accumulation per cycle. Both MACs share one adder for the accumulate operation.

2.7 Device Emulation Unit

The device emulation unit (DEU) allows in-circuit debug and interfaces with external JTAG logic.

Chapter 3
Control Registers

This chapter discusses the control registers. It includes the following sections:

- Section 3.19, “System Mode Register (%smode),” page 3-18
- Section 3.20, “Timer Control Register (%tc),” page 3-22
- Section 3.21, “Timer 0 Register (%timer0),” page 3-23
- Section 3.22, “Timer 1 Register (%timer1),” page 3-23
- Section 3.23, “Trap Return Program Counter Register (%tpc),” page 3-24
- Section 3.24, “Viterbi Traceback Register (%vitr),” page 3-24

3.1 Introduction

The ZSP400 architecture contains a set of control registers, used for mode control, status, and flag information. The ZSP400 architecture allows for 32 16-bit control registers. Specific processors may use a subset of the 32 control registers. Unused registers are reserved; write reserved registers with zeros to guarantee compatibility with future generation devices.

Control registers are specified in assembly language by a mnemonic with a “`%`” prefix (for example, `%fmode`). All control registers are accessible using `mov` instructions.

Table 3.1 lists the control registers.

Table 3.1 ZSP400 Control Registers

Mnemonic	Reset Value	Description
%amode	0x0	Address Mode Register
%cb0_beg	Undefined	Circular Buffer 0 Begin Address Register
%cb0_end	Undefined	Circular Buffer 0 End Address Register
%cb1_beg	Undefined	Circular Buffer 1 Begin Address Register
%cb1_end	Undefined	Circular Buffer 1 End Address Register
%ded	Undefined	Device Emulation Data Register
%dei	Undefined	Device Emulation Instruction Register
%fmode	0x0	Functional Mode Register

Table 3.1 ZSP400 Control Registers (Cont.)

Mnemonic	Reset Value	Description
%guard	Undefined	Guard Bits for {r1 r0} and {r3 r2}
%hwflag	Undefined	Hardware Flag Register
%imask	0x0	Interrupt Mask Register
%ip0	0x0	Interrupt Priority Register 0
%ip1	0x0	Interrupt Priority Register 1
%ireq	0x0	Interrupt Request Register
%loop0	0x0	Loop 0 Register
%loop1	0x0	Loop 1 Register
%loop2	0x0	Loop 2 Register
%loop3	0x0	Loop 3 Register
%pc	0xF800	Program Counter
%rpc	Undefined	Return Program Counter
%smode	0x0	System Mode Register
%tc	0x0	Timer Control Register
%timer0	0x0	Timer 0 Register
%timer1	0x0	Timer 1 Register
%tpc	Undefined	Trap (Interrupt) Return Program Counter
%vitr	Undefined	Viterbi Traceback Register

Several control registers contain reserved bits. To ensure future code compatibility, do not set these reserved bits to a non-default value. This can be guaranteed by using the `bits`, `bitc`, and `biti` instructions on only the unreserved bits, or by doing a move from the control register to an operand register, a modification of the operand register that leaves the reserved bits unchanged, followed by a move from the operand register back to the control register.

3.2 Address Mode Register (%amode)

This register controls the addressing mode for operand registers r0 through r12 when using load and store with update instructions (`ldu`, `lddu`, `stu`, and `stdu`).

The reset value of this register is 0x00.

15 6	5 2	1	0
res	rev	st	ld

res **Reserved** **[15:6]**

This field is reserved.

rev **Reverse Bit Length** **[5:2]**

This field determines the bit location for reverse-carry addressing updates.

The field encoding is shown as follows.

rev Bit	Carry Bit Insertion Location
0b0000	Bit 15
0b0001	Bit 0
0b0010	Bit 1
0b0011	Bit 2
0b0100	Bit 3
0b0101	Bit 4
0b0110	Bit 5
0b0111	Bit 6
0b1000	Bit 7
0b1001	Bit 8
0b1010	Bit 9
0b1011	Bit 10
0b1100	Bit 11
0b1101	Bit 12
0b1110	Bit 13
0b1111	Bit 14

st **Store Enable** **1**

This bit enables reverse-carry addressing on stores.

When set, enables reverse-carry addressing for `stu` and `stdu` instructions.

When cleared, disables reverse-carry addressing for `stu` and `stdu` instructions.

ld **Load Enable** **0**

This bit enables reverse-carry addressing on loads.

When set, enables reverse-carry addressing for `ldu` and `lddu` instructions.

When cleared, disables reverse-carry addressing for `ldu` and `lddu` instructions.

3.3 Circular Buffer 0 Begin Address Register (%cb0_beg)

This register contains the start address for circular buffer 0. On reset, the contents of %cb0_beg are undefined.

Circular buffer 0 operations affect the following instructions when the `cb0` bit in the %smode register is set:

- `lddu  rX, r14, 2`
- `stdu  rX, r14, 2`
- `ldu   rX, r14, 1`
- `ldu   rX, r14, 2`
- `stu   rX, r14, 1`
- `stu   rX, r14, 2`

where `rX` is any operand register.

3.4 Circular Buffer 0 End Address Register (%cb0_end)

This register contains the end address +1 for circular buffer 0. The last word of the Circular Buffer is cbX_end-1. On reset, the contents of %cb0_end are undefined.

Circular buffer 0 operations affect the following instructions when the `cb0` bit in the %smode register is set:

- `lddu rX, r14, 2`
- `stdu rX, r14, 2`
- `ldu  rX, r14, 1`
- `ldu  rX, r14, 2`
- `stu  rX, r14, 1`
- `stu  rX, r14, 2`

where `rX` is any operand register.

3.5 Circular Buffer 1 Begin Address Register (%cb1_beg)

This register contains the start address for circular buffer 1. On reset, the contents of %cb1_beg are undefined.

Circular buffer 1 operations affect the following instructions when the `cb1` bit in the %smode register is set:

- `lddu rX, r15, 2`
- `stdu rX, r15, 2`
- `ldu  rX, r15, 1`
- `ldu  rX, r15, 2`
- `stu  rX, r15, 1`
- `stu  rX, r15, 2`

where `rX` is any operand register.

3.6 Circular Buffer 1 End Address Register (%cb1_end)

This register contains the end address +1 for circular buffer 1. The last word of the Circular Buffer is cbX_end-1. On reset, the contents of %cb1_end are undefined.

Circular buffer 1 operations affect the following instructions when the cb1 bit in the %smode register is set:

- `lddu  rX, r15, 2`
- `stdu  rX, r15, 2`
- `ldu   rX, r15, 1`
- `ldu   rX, r15, 2`
- `stu   rX, r15, 1`
- `stu   rX, r15, 2`

where `rX` is any operand register.

3.7 Device Emulation Data Register (%ded)

The Device Emulation Data register passes data and addresses between the processor and JTAG interface during device emulation. On reset, the contents of %ded are undefined.

3.8 Device Emulation Instruction Register (%dei)

The Device Emulation Instruction register passes instructions between the processor and JTAG interface during device emulation. On reset, the contents of %dei are undefined.

3.9 Functional Mode Register (%fmode)

The %fmode register contains the functional mode bits for the ZSP400. The functional modes determine how the results of arithmetic instructions are affected. The value of the %fmode register at reset is 0x0.

15	6	5	4	3	2	1	0
res		rez	sat	res	q15	sre	mre

res **Reserved** **[15:6]**
This field is reserved.

rez **Round Zero Enable** **5**
This bit clears the lower 16 bits of MAC results when MAC rounding is enabled (that is, when the `mre` bit of the %fmode register is set).

When the `rez` bit is set, the lower 16 bits of MAC results are set to zero. This affects the following 32-bit instructions: `mac`, `macn`, `mul`, `muln`, `mac2`, `cmacr`, `cmaci`, `cmulr`, `cmuli`, `dmac`, `dmul`, and `round.e`.

Note: The `mre` bit in the %fmode register must be set to use the `rez` bit. If the `mre` bit is clear, setting `rez` has no effect.

When cleared, the `rez` bit has no effect.

sat **Saturation Enable** **4**
This bit specifies whether saturation is enabled or disabled. When set, effected arithmetic operations saturate to MAX_POS[1] or MAX_NEG[2] on overflow.

When the `sat` bit is cleared, saturation is disabled.

The overflow check occurs after the accumulation for MAC instructions.

res **Reserved** **3**
This bit is reserved.

---

[1] MAX_POS (16 bit) = 0x7FFF; MAX_POS (32 bit) = 0x7FFF.FFFF.

[2] MAX_NEG (16 bit) = 0x8000; MAX_NEG (32 bit) = 0x8000.0000.

q15 **Fixed-Point Format q15** **2**

When set, the result of MAC and MUL instructions are shifted left one bit. For a MAC instruction, the shifted value is then accumulated. For the corner cases of 0x8000 • 0x8000 (16 bit) or 0x8000.0000 • 0x8000.0000 (32 bit), the result is saturated to 0x7FFF.FFFF.

When cleared, disable q15 format (enable integer format). No shift occurs after MAC or MUL instructions.

sre **Shift Round Enable** **1**

This feature enables rounding for the `SHRA` instruction. This rounding never causes saturation. When set, if the last bit shifted out is a 1, 0x0001 is added to the `SHRA` result.

When the `sre` bit is cleared, the processor does not round the `SHRA` result.

mre **MAC/MUL Round Enable** **0**

This feature enables rounding for most MAC/MUL instructions. Rounding may cause saturation. When set, this bit enables MAC/MUL rounding. For 16-bit MAC instructions, 0x8000 is added after the accumulation. For 32-bit MAC instructions, 0x8000.0000 is added to the 64-bit result after the accumulation and a 32-bit result is returned.

When this bit is cleared, MAC/MUL results are not rounded.

3.10 Guard Bits for {r1 r0} and {r3 r2}

The %guard register extends the precision of the operand registers by 8 bits (MSBs) for the accumulation result of MAC instructions. The %guard register can be accessed using the `mov` instruction and is not modified by shift instructions performed on either operand register pair ({r3 r2} or {r1 r0}).

The contents of this register at reset are undefined.

15 8	7 0
guard_1	guard_0

guard_1 **Guard Bits for Accumulator b** **[15:8]**
Guard bits for accumulator b, which consist of the register pair {r3 r2}.

guard_0 **Guard Bits for Accumulator a** **[7:0]**
Guard bits for accumulator a, which consist of the register pair {r1 r0}.

3.11 Hardware Flag Register (%hwflag)

The %hwflag register contains condition codes that occur as a result of various instructions or processor status. The value of this register is undefined at reset. The sticky overflow flags, sv and gsv, can only be cleared through software. The user must explicitly write a zero to these fields to clear these bits.

15	11	10	9	8	7	6	5	4	3	2	1	0
res		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er

res **Reserved** **[15:11]**
This field is reserved.

v **32-Bit Overflow** **10**
Only MAC, MUL, ADD, and shift instructions modify this bit.

When set, indicates the sign of the result of a twos complement addition is different than the sign of the operands (both operands have the same sign).

When cleared, indicates the sign of the result of a twos complement addition is the same as the sign of the operands (both operands have the same sign).

In addition, for MAC variants in q15 format, the following sequence of suboperations occurs: Multiply, add, then round if the `mre` field of the %fmode register is set, then check for overflow and update the v field of %hwflag. Saturate on overflow if the `sat` field of the %fmode register is set.

For ADD variants, the following sequence of suboperations occurs: Add, check for overflow and update the v field of %hwflag register. Saturate on overflow if the `sat` field of the %fmode register is set.

For the SHLA instruction, overflow occurs if any bit shifted through the sign bit position differs from the sign bit of the original operand.

gv **Guard Register (40-Bit) Overflow** **9**

This bit is the same as the v bit, but for 40-bit data instead of 32-bit data, and is modified only by MAC instructions.

sv **Sticky Overflow** **8**

This bit is the same as the v bit, but can only be cleared through software (by writing a zero to the bit).

gsv **Guard Register Sticky Overflow** **7**

This bit is the same as the gv bit, but can only be cleared through software (by writing to the bit).

c **Carry** **6**

This bit is set when a carry out from bit 15 (for 16-bit operations) or bit 31 (for 32-bit operations) occurs.

This bit is cleared when no carry out has occurred.

ge **Greater Than, or Equal To** **5**

This bit is set when the result is greater than, or equal to, zero. This bit is cleared when the result is less than zero.

gt **Greater Than** **4**

This bit is set when the result is greater than zero. This occurs when the ge bit is set and the `z` bit is not set.

This bit is cleared when the result is less than or equal to zero.

z **Equal To Zero** **3**

This bit is set when the result is equal to zero, or the values compared are equal.

This bit is cleared when the result is not equal to zero, or the values compared are not equal.

ir **Interrupt Pending** **2**

This bit is set when an interrupt is pending, regardless of the interrupt's masking or priority level.

The `ir` bit is cleared when no interrupt is pending.

ex **Emulation Transmit** **1**

This bit is set when the JTAG TAP controller has read the %ded or %dei register.

This bit is cleared when the %ded or %dei registers have not been read.

er **Emulation Receive** **0**

This bit is set when the JTAG TAP controller has written the %ded or %dei register.

This bit is cleared when the %ded or %dei registers have not been written.

3.12 Interrupt Mask Register (%imask)

The %imask register contains mask information for the 15 maskable interrupts supported by the ZSP400. A cleared mask bit prevents the corresponding interrupt from being serviced. All bits in the %imask register are cleared on reset. The reset value of this register is 0x0.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
gie	pgie	IM13	IM12	IM11	IM10	IM9	IM8	IM7	mt1	mt0	IM4	IM3	IM2	IM1	IM0

gie **Global Interrupt Enable** **15**

This bit is automatically cleared when an interrupt service routine is entered, and is restored from the contents of the `pgie` field by the `reti` instruction. Set this bit within an interrupt service routine to nest interrupts.

When set, this bit enables all unmasked interrupts.

When cleared, this bit disables all interrupts except `NMI` and the DEI.

pgie **Previous Global Interrupt Enable** **14**

This bit contains the original value of `gie` when executing an interrupt service routine. Execution of the `reti` instruction restores the value in `pgie` in the `gie` bit.

IM13 **Interrupt 13 Enable** **13**

When set, this bit enables interrupt 13. When cleared, this bit masks interrupt 13.

IM12	**Interrupt 12 Enable** When set, this bit enables interrupt 12. When cleared, this bit masks interrupt 12.	**12**
IM11	**Interrupt 11 Enable** When set, this bit enables interrupt 11. When cleared, this bit masks interrupt 11.	**11**
IM10	**Interrupt 10 Enable** When set, this bit enables interrupt 10. When cleared, this bit masks interrupt 10.	**10**
IM9	**Interrupt 9 Enable** When set, this bit enables interrupt 9. When cleared, this bit masks interrupt 9.	**9**
IM8	**Interrupt 8 Enable** When set, this bit enables interrupt 8. When cleared, this bit masks interrupt 8.	**8**
IM7	**Interrupt 7 Enable** When set, this bit enables interrupt 7. When cleared, this bit masks interrupt 7.	**7**
mt1	**Timer1 Interrupt (t1)** When set, this bit enables the timer1 interrupt. When cleared, this bit masks the timer1 interrupt.	**6**
mt0	**Timer0 Interrupt (t0)** When set, this bit enables the timer0 interrupt. When cleared, this bit masks the timer0 interrupt.	**5**
IM4	**Interrupt 4 Enable** When set, this bit enables interrupt 4. When cleared, this bit masks interrupt 4.	**4**
IM3	**Interrupt 3 Enable** When set, this bit enables interrupt 3. When cleared, this bit masks interrupt 3.	**3**
IM2	**Interrupt 2 Enable** When set, this bit enables interrupt 2. When cleared, this bit masks interrupt 2.	**2**
IM1	**Interrupt 1 Enable** When set, this bit enables interrupt 1. When cleared, this bit masks interrupt 1.	**1**

IM0 **Interrupt 0 Enable** **0**

When set, this bit enables interrupt 0. When cleared, this bit masks interrupt 0.

3.13 Interrupt Priority Register 0 (%ip0)

The %ip0 register contains interrupt priority level and processor execution priority level information. The user may write to any field of this register. User-defined priorities are given values of `0b00` to `0b11`, with `0b11` being the highest user-defined priority and `0b00` the lowest. This register contains `0x0` at reset.

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="2">epl</td><td colspan="2">pepl</td><td colspan="2">IP13</td><td colspan="2">IP12</td><td colspan="2">IP11</td><td colspan="2">IP10</td><td colspan="2">IP9</td><td colspan="2">IP8</td></tr>
</table>

epl **Current Execution Priority Level** **[15:14]**

This field determines if an interrupt request is serviced. If the interrupt priority level of the pending interrupt is lower than that of the current `epl`, the pending interrupt is not serviced.

If the interrupt priority level of the pending interrupt is greater than or equal to the current `epl`, then the pending interrupt is serviced and the pending interrupt's `epl` is copied into the `epl` field of %ip0.

pepl **Previous Execution Priority Level** **[13:12]**

When an interrupt is taken, the ZSP400 writes the contents of the `epl` field into this field.

IP13 **Interrupt 13 Priority Level** **[11:10]**

This field sets the priority level of interrupt 13.

IP12 **Interrupt 12 Priority Level** **[9:8]**

This field sets the priority level of interrupt 12.

IP11 **Interrupt 11 Priority Level** **[7:6]**

This field sets the priority level of interrupt 11.

IP10 **Interrupt 10 Priority Level** **[5:4]**

This field sets the priority level of interrupt 10.

IP9 **Interrupt 9 Priority Level** **[3:2]**
This field sets the priority level of interrupt 9.

IP8 **Interrupt 8 Priority Level** **[1:0]**
This field sets the priority level of interrupt 8.

3.14 Interrupt Priority Register 1 (%ip1)

The %ip1 register sets interrupt priority level and controls interrupt behavior of the timer and external interrupts. The user may write to any field of this register. User-defined priorities are given values of 0b00 to 0b11, with 0b11 being the highest user-defined priority and 0b00 the lowest. This register contains 0x0 at reset.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IP7		t1		t0		IP4		IP3		IP2		IP1		IP0	

IP7 **Interrupt 7 Priority Level** **[15:14]**
This field sets the priority level of interrupt 7.

t1 **Timer 1 (t1) Interrupt Priority Level** **[13:12]**
This field sets the priority level of the Timer 1 interrupt.

t0 **Timer 0 (t0) Interrupt Priority Level** **[11:10]**
This field sets the priority level of the Timer 0 interrupt.

IP4 **Interrupt 4 Priority Level** **[9:8]**
This field sets the priority level of interrupt 4.

IP3 **Interrupt 3 Priority Level** **[7:6]**
This field sets the priority level of interrupt 3.

IP2 **Interrupt 2 Priority Level** **[5:4]**
This field sets the priority level of interrupt 2.

IP1 **Interrupt 1 Priority Level** **[3:2]**
This field sets the priority level of interrupt 1.

IP0 **Interrupt 0 Priority Level** **[1:0]**
This field sets the priority level of interrupt 0.

3.15 Interrupt Request Register (%ireq)

The %ireq control register shows the pending interrupts for all sources. Any of these bits may be set through software to create a user trap. All bits are sticky in the sense that a pending interrupt status is only cleared when that interrupt is serviced or the bit is explicitly cleared by software. This register contains 0x0 at reset.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
nmi	dei	IR13	IR12	IR11	IR10	IR9	IR8	IR7	t1	t0	IR4	IR3	IR2	IR1	IR0

nmi **Nonmaskable Interrupt Request** **15**
When set, this bit indicates that a NMI is pending.

dei **Device Emulation Interrupt Request** **14**
When set, this bit indicates that a device emulation interrupt is pending.

IR13 **Interrupt 13 Request** **13**
When set, this bit indicates that interrupt 13 is pending.

IR12 **Interrupt 12 Request** **12**
When set, this bit indicates that interrupt 12 is pending.

IR11 **Interrupt 11 Request** **11**
When set, this bit indicates that interrupt 11 is pending.

IR10 **Interrupt 10 Request** **10**
When set, this bit indicates that interrupt 10 is pending.

IR9 **Interrupt 9 Request** **9**
When set, this bit indicates that interrupt 9 is pending.

IR8 **Interrupt 8 Request** **8**
When set, this bit indicates that interrupt 8 is pending.

IR7 **Interrupt 7 Request** **7**
When set, this bit indicates that interrupt 7 is pending.

t1 **Timer 1 Interrupt Request** **6**
When set, this bit indicates a timer 1 interrupt is pending.

t0 **Timer 0 Interrupt Request** **5**
When set, this bit indicates a timer 0 interrupt is pending.

IR4	**Interrupt 4 Request** When set, this bit indicates that interrupt 4 is pending.	**4**
IR3	**Interrupt 3 Request** When set, this bit indicates that interrupt 3 is pending	**3**
IR2	**Interrupt 2 Request** When set, this bit indicates that interrupt 2 is pending.	**2**
IR1	**Interrupt 1 Request** When set, this bit indicates that interrupt 1 is pending.	**1**
IR0	**Interrupt 0 Request** When set, this bit indicates that interrupt 0 is pending.	**0**

3.16 Loop Counter Registers (%loop0, %loop1, %loop2, %loop3)

The %loop0, %loop1, %loop2, and %loop3 registers are 16-bit counters that decrement as part of the execution of a low-overhead looping construct that uses the `agn0`, `agn1`, `agn2`, and `agn3` instructions, respectively. Use the `mov` instruction to place an initial value into these registers. The value initially loaded to these registers is always the number of iterations minus one (N – 1). This is due to the fact that the `agn` instructions test for N ≤ 0 before decrementing the count, so the `0`th loop iteration is always executed. After the loop has completed, the value in the corresponding loop register is `0`xFFFF (– 1). The value in the loop counter registers is `0`x`0` at reset.

Figure 3.1 Low-Overhead Looping Construct Code Example

```
mov %loop0, 15           /* loop count = N-1. Execute this loop 16 times */
Loop:
    lddu  r4, r14, 2     /* load r4 and r5, post increment r14 by 2 */
    lddu  r8, r15, 2     /* load r8 and r9, post increment r15 by 2 */
    mac2.a r4, r8        /* {g0 r1 r0} += (r4*r8) + (r5*r9) */
    agn0 Loop            /* test for 0, decrement count, then branch */
```

3.17 Program Counter Register (%pc)

The program counter register contains the address of the instruction currently being executed. This register is implicitly written when branches are taken. On reset, the value of this register is 0xF800.

15 0
pc

3.18 Return Program Counter Register (%rpc)

This register contains the return address from a subroutine call. When the `call` instruction is executed, the value of %rpc is updated with the value of %pc +1, which is the address of the instruction following the `call`. When the `ret` instruction is executed at the end of the routine, %rpc is copied into %pc. The %rpc register is undefined at reset.

15 0
rpc

3.19 System Mode Register (%smode)

The %smode register controls the ZSP400 system modes. System modes affect the operation of hardware, including power-saving features, circular buffers, and memory accesses. The value of this register at reset is 0x0.

15 13	12	11	10	9	8	7	6	5	4	3	2	1	0
lvl	shd	dct	fie	ict	dsb	uvt	us	lis	sis	cb0	cb1	dir	ddr

lvl **Power Level** **[15:13]**

This field specifies the power level of the ZSP400. The available power levels are normal, idle, sleep, and halt.

lvl Bits	Power Level
0b000	Normal
0b001	Idle
0b010	Sleep
0b011	Reserved
0b100	Halt
0b101	Reserved
0b110	Reserved
0b111	Reserved

The effect of the power level field on system peripherals depends on the system implementation.

shd **Enable Shadow Registers** **12**

This bit toggles between the primary operand registers and a set of shadow registers. Only operand registers r2 through r9 have shadow register analogs.

When set, this bit uses shadow registers for accesses to operand registers r2 through r9.

When cleared, this bit uses primary registers for accesses to operand registers r2 through r9.

dct **Data Cache Invalidate** **11**

Inverting this bit invalidates the contents of the data cache. The value of the `dct` bit does not indicate valid or invalid cache contents.

fie **Force Internal Execution** **10**

When set, this bit overrides the execution control of the `dir` bit in the %smode register and forces the processor to execute instructions from internal memory if internal memory and external memory are both physically present at the executing address.

When this bit is cleared, execution of instructions is controlled by the `dir` bit of the %smode register.

ict	**Instruction Cache Invalidate**	**9**

Inverting this bit invalidates the contents of the instruction cache. The value of the `ict` bit does not indicate valid or invalid cache contents.

dsb	**MXU Store Buffer Enable**	**8**

When set, this bit controls an external memory interface unit store buffer if the device includes a store buffer. The store buffer is an optional module that is not included in the core. When this bit is set, the store buffer is disabled.

When this bit is cleared, the store buffer is enabled.

uvt	**User Vector Table Starting Address**	**7**

This bit selects the interrupt vector table (IVT) address. When this bit is set, the interrupt vector table starting address is internal SRAM address 0x0000.

When this bit is cleared, the interrupt vector table starting address is 0xF800, but the ROM used (internal or external) depends upon the state of the `IBOOT` pin.

IBOOT Level	IVT Address	Memory Location
LOW	0xF800	External ROM
HIGH	0xF800	Internal ROM

us	**Uniscalar Mode Enable**	**6**

This bit toggles between uniscalar (the processor executes one instruction per cycle) and superscalar mode (the processor executes multiple instructions per cycle).

When this bit is set, the processor operates in uniscalar mode.

When this bit is cleared, the processor operates in superscalar mode.

lis	**Load Instruction Space Enable**	**5**

This bit selects the location for data reads from internal and external memory. This bit is cleared by default.

sis	**Store Instruction Space Enable**	**4**

This bit selects the location for data writes to internal and external memory.

This bit is cleared by default.

cb0 **Circular Buffer 0 Enable** **3**

When set, this bit enables r14 as Circular Buffer 0.

When cleared, this bit disables Circular Buffer 0.

cb1 **Circular Buffer 1 Enable** **2**

When set, this bit enables r15 as Circular Buffer 1.

When cleared, this bit disables Circular Buffer 1.

dir **Disable Internal Instruction RAM** **1**

This bit toggles between internal and external RAM for instruction fetches. This bit also affects the location for instructions when either the `sis` or `lis` bit is set.

When this bit is set, if the %pc register points to a memory location that is physically present in both internal and external instruction memory, the processor executes instructions from external instruction memory.

If the `lis` bit is set, loads from a memory location that is physically present in both internal and external instruction memory are loaded from external instruction memory. If the `sis` bit is set, stores to a memory location that is physically present in both internal and external instruction memory are stored to external instruction memory.

When this bit is cleared, if the PC points to a memory location that is physically present in both internal and external instruction memory, the processor executes instructions from internal instruction memory.

If the `lis` bit is set, loads from a memory location that is physically present in both internal and external instruction memory are loaded from internal instruction memory. If the `sis` bit is set, stores to a memory location that is physically present in both internal and external instruction memory are stored to internal instruction memory.

This bit toggles between internal and external RAM for data loads and stores.

Note: If the `lis` or `sis` bit is set, the `dir` bit overrides the `ddr` bit.

ddr **Disable Internal Data RAM** **0**

This bit determines where the processor loads and stores data from. The lis and sis bits override this bit if set. In this case, the ddr bit controls the load/store location.

When this bit is set, if the location of data memory accessed by a load/store instruction is physically present in both internal and external data memory, the processor accesses the data from the external data memory.

When cleared, if the location of data memory accessed by a load/store instruction is physically present in both internal and external data memory, the processor accesses the data from the internal data memory.

3.20 Timer Control Register (%tc)

The fields in the timer control register enable the two timers in the ZSP400, set the prescale value for each timer, and set the timer mode. The lower half of the %tc register sets the enable, mode, and prescale values for Timer 0, and the upper bytes set these values for Timer 1. The values of the %tc, %timer0, and %timer1 registers are 0x0 at reset.

15	14	13 8	7	6	5 0
et1	cm1	tmrdiv1	et0	cm0	tmrdiv0

et1 **Enable Timer 1** **15**
This bit controls the operation of Timer 1. When set, this bit enables Timer 1.

When cleared, this bit disables Timer 1.

cm1 **Control Mode for Timer 1** **14**
This bit controls the reloading feature of Timer 1. When this bit is set, Timer 1 counts down to 0, then reloads the initial count.

When this bit is cleared, autoreload is disabled. Timer 1 counts down to 0 and stops (single-shot mode).

tmrdiv1 **Prescale Value for Timer 1** **[13:8]**
For a value N represented by 6 bits in this field, the clock divisor is (N + 1). Therefore, if N = 2, then Timer 1 decrements every three clock cycles.

et0 **Enable Timer 0** **7**
This bit controls the operation of Timer 0. When set, this bit enables Timer 0.

When cleared, this bit disables Timer 0.

cm0 **Control Mode for Timer 0** **6**

This bit controls the reloading feature of Timer 0. When this bit is set, Timer 0 works with autoreload. Timer 0 counts down to 0, then reloads the initial count.

When this bit is cleared, autoreload is disabled. Timer 0 counts down to 0 and stops (single-shot mode).

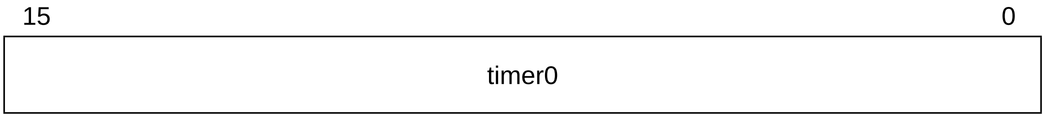
tmrdiv0 **Prescale Value for Timer 0** **[5:0]**

For a value N represented by 6 bits in this field, the clock divisor is (N + 1). Therefore, if N = 2, then Timer 0 decrements every three clock cycles.

3.21 Timer 0 Register (%timer0)

The Timer 0 register contains a 16-bit counter that decrements at a constant rate. The decrement rate for %timer0 is controlled by the device clock period and the prescale value set in the `timrdiv0` field of the %tc register. On reset, the contents of %timer0 are 0x0.

Load a value into %timer0 using register mov instructions. When the counter decrements to zero, the `t0` bit in the %ireq register is set and an interrupt request is generated. Upon reaching zero, the counter reloads or remains at zero based on the contents of the `cm0` field of the %tc register.



3.22 Timer 1 Register (%timer1)

The Timer 1 register contains a 16-bit counter that decrements at a constant rate. The decrement rate for %timer1 is controlled by the device clock period and the prescale value set in the `timrdiv1` field of the %tc register. On reset, the contents of %timer1 are 0x0.

Load a value into %timer1 using register mov instructions. When the counter decrements to zero, the `t1` bit in the %ireq register is set, and an interrupt request is generated. Upon reaching zero, the counter reloads or remains at zero based on the contents of the `cm1` field of the %tc register.

15	0
timer1	

3.23 Trap Return Program Counter Register (%tpc)

The %tpc register contains the return value from an ISR. When the processor takes an unmasked interrupt, %tpc is updated with the address of the next sequential instruction (%pc + 1). When the interrupt is serviced, the contents of %tpc are copied to %pc.

When the processor takes a DEI, the current value of %tpc is stored in the %ded register before copying the next instruction's value to %tpc.

On reset, the contents of the %tpc register are undefined.

15	0
tpc	

3.24 Viterbi Traceback Register (%vitr)

The %vitr register holds Viterbi traceback information. The oldest traceback bit is contained in bit 15, and the most recent traceback bit is in bit 0. The `vit_a` and `vit_b` instructions update the LSB of the %vitr register. The contents of the register are shifted left one bit when the LSB is updated. At reset, the contents of the %vitr register are undefined.

15	0
vitr	

Chapter 4
Pipeline Control Unit

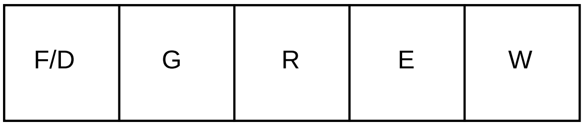
This chapter discusses the pipeline control unit. It includes the following sections:

4.1 Introduction

In a pipelined processor, instructions execute in stages. This separation allows the overlap of instructions in the pipeline. The ZSP400 architecture is a superscalar processor that employs a five-stage pipeline.

Figure 4.1 shows the ZSP400 pipeline.

Figure 4.1 ZSP400 Pipeline



Fetch/Decode Stage – The processor fetches instructions from memory and decodes them during this stage.

Group – The processor checks grouping and dependency rules and issues valid instructions to the pipeline.

Read – Operands are read from the data unit during this stage.

Execute – The appropriate execution unit (ALU or MAC) executes the instruction and writes the results to a general purpose register or sends them to the Data Unit.

Write Back – The Data Unit writes the results to memory and updates all control registers.

The pipeline control unit (PCU) takes care of instruction grouping, housekeeping functions, and arithmetic unit result bypassing. The PCU synchronizes the operation of the pipeline and handles interrupt requests from the Interrupt Control Unit.

4.2 Interlocking Pipeline

The ZSP400 architecture uses an interlocking pipeline—hardware controls the pipeline. Stalls and pipeline dependencies are not visible to the programmer. Stalls occur under the following conditions:

- Slow external memory accesses starve the pipeline of data
- The instruction prefetcher needs additional cycles to load cache lines from main memory at a program flow discontinuity or a branch mispredict
- The data prefetcher needs additional cycles to load two cache lines from main memory during setup
- The write-through cache needs an additional cycle to write the results of an extended precision operand back to main memory. (The result straddles two cache lines. The pipeline stalls one cycle to allow both cache lines to be written.)

4.3 Grouping Rules

This section discusses the ZSP400 instruction grouping rules. The Pipeline Control Unit (PCU) receives instructions from the Instruction Unit while the instructions are in the fetch/decode stage of the pipeline. The PCU checks the instructions for dependencies and groups them for execution according to the grouping rules listed in this section.

Within the constraints imposed by the grouping rules, the PCU attempts to issue instructions in groups of up to four instructions per group to maximize DSP throughput, but such dense instruction grouping is not possible with every instruction sequence without violating a grouping rule.

Programmers who write ZSP400 code in a high-level language do not need to know these rules to write functional code because the ZSP400 C-compiler optimizer knows the grouping rules and attempts to optimize the machine-level instructions to minimize pipelines stalls, and the PCU automatically applies the grouping rules to the optimized code.

However, knowledge of these grouping rules is useful for writing or debugging assembly-level code that can be densely grouped to improve DSP speed. To facilitate debug, the SDK debugger displays the grouping rule that was applied to each group of instructions.

1. Do not group invalid instructions.
2. Do not group the following instructions with any ZSP400 instruction; these instructions must be placed in a group that consists of only one instruction. Following a group that includes one of these instructions, do not group any ZSP400 instruction until two processor clock cycles after the first instruction reaches the W pipeline stage (W+2).
   - call rX
   - mov %smode, rX
   - bitc %ip*, x ; * = 0, 1;
   - mov %pc, rX
   - bitc %smode, x
   - bits %ip*, x ; * = 0, 1;
   - mov %amode, rX
   - bits %smode, x
   - biti %ip*, x ; * = 0, 1;
   - mov %imask, rX
   - bitc %amode, x
   - biti %smode, x
   - bitc %ireq, x

  - mov %ip0, rX
  - bits %amode, x
  - bitc %imask, x
  - bits %ireq, x
  - mov %ip1, rX
  - biti %amode, x
  - bits %imask, x
  - biti %ireq, x
  - mov %ireq, rX
  - bitc %imask, x
  - biti %imask, x

3. Do not group the following instructions, which use the ALU or MAC units, if there is an instruction in the R or G pipeline stage that effects the %fmode register.
  - Store instructions: all the Memory Reference instructions (Table 8.9) starting with st.
  - Unlinked Load instructions (Table 8.9) are all ld and ldx instructions plus the ldu and lddu instructions for which no link has been established.
  - ALU operations: all Arithmetic instructions (Table 8.5) and all Bitwise Logical instructions (Table 8.6).
  - MAC instructions: All MAC instructions (Table 8.4)
  - mov rY, cx
  - mov cx, rY
  - mov rX, rY
  - mov rX, IMM
4. Do not group the following instructions, which read the %guard or %vitr registers, if there is a MAC instruction (Table 8.4) in the G or R pipeline stage.
  - mov rX, cy
  - bits cx, y (cx: %guard or %vitr)
  - bitc cx, y (cx: %guard or %vitr)

  - biti cx, y (cx: %guard or %vitr)
  - bitt cx, y (cx: %guard or %vitr)

  In addition, do not group the following instruction if there is a MAC instruction (Table 8.4) in the G or R pipeline stage.

  - mov %guard, IMM

5. Do not group any MAC instruction (Table 8.4) if there is an instruction in the G or R pipeline stage that writes to the %guard or %vitr registers through one of the following instructions:
   - mov cx, rY
   - bitc cx, y
   - bits cx, y
   - biti cx, y
6. Do not group a Branch Conditional instruction (all instructions on Table 8.8 except for the agn and br) if there is a mov %hwflag, rX instruction in the G or R pipeline stage.
7. Do not group the following instructions:
   - mov rX, cy
   - bitc cx, y
   - bits cx, y
   - biti cx, y
   - bitt cx, y
   - ret
   - reti
   - mov cx, IMM
   - call rX
   - call IMM

   If one of the following instructions are in the R or E pipeline stages:

   - mov rX, cy
   - bitc cx, y
   - bits cx, y
   - biti cx, y

- bitt cx, y
- ret
- reti
- mov cx, rY
- call rX
- call IMM

Or, if one of the following instructions is in the G pipeline stage:

- mov cx, rY
- bitc cx, y
- bits cx, y
- biti cx, y

8. Do not group the following instructions if any instruction is in the G, R, or E pipeline stage:
   - mov rX, %hwflag
   - bits %hwflag, y
   - bitc %hwflag, y
   - biti %hwflag, y
   - bitt %hwflag, y
9. Do not group the following instructions if there is a mov %hwflag, rX instruction in the R pipeline stage, or any instruction in the G pipeline stage.
   - Conditional Branch instruction (all instructions on Table 8.8 except for the agn and br).
   - addc.e
   - subc.e
10. Do not group the following instructions if there is a mov cb*, rX instruction in the G, R, E, or W pipeline stages:
    - ldu
    - lddu
    - stu
    - stdu

11. Do not group the following instructions if there is a Conditional Branch instruction (all instructions on Table 8.8, page 8-20, except for the agn and br instructions) or an reti instruction in the G, R, or E pipeline stages:
    - call IMM
    - call rX
    - agn0
    - agn1
    - agn2
    - agn3
    - ret
    - reti
    - mov cx, IMM
12. Do not group the following agnx instructions if the instruction has a corresponding mov %loopx, rY instruction in the G, R, E, W, or W+1 pipeline stages. For example, do not group the agn0 instruction if the mov %loop0, rY instruction is in the G, R, E, W, or W+1 pipeline stages.
    - agn0
    - agn1
    - agn2
    - agn3
13. Do not group an ret or reti instruction if one of the following instructions is in the G, R, E, W, or W+1 pipeline stages:
    - mov rX, cy
    - bitc cx, y
    - bits cx, y
    - biti cx, y
    - bitt cx, y
    - ret
    - reti
    - mov cx, rY

  - call rX
  - call IMM

14. Do not group any instruction in a processor clock cycle in which the processor has an interrupt request pending or in a processor clock cycle in which the processor takes an interrupt.
15. Do not group an agnx instruction if there is a mov %loopx, rX instruction in the G, R, E, W, or W+1 pipeline stages, or there is a mov loopx, IMM instruction in the R or E pipeline stages.
16. Do not group an ret or reti instruction if one of the following instructions is in the G, R, E, W, or W+1 pipeline stages.
    - mov rX, cy
    - mov cx, rY
    - bitc cx, y
    - bits cx, y
    - biti cx, y
    - ret
    - reti
17. Do not group instructions in a manner that causes them to execute out of order.
18. Do not group more than one instruction if the device is in Uniscalar mode.
19. Do not group a mov rX, %pc instruction if any instruction is in the G pipeline stage.
20. Do not group the following instructions if there is a mov cx, IMM instruction in the G pipeline stage:
    - mov rX, cy
    - bitt cx, y
    - bits cx, y
    - bitc cx, y
    - biti cx, y
21. Do not group the following instructions if there is a Conditional Branch instruction (all instructions on Table 8.8 except for the agn and br) or an reti instruction in the G pipeline stage:

  - mov cx, rY
  - biti cx, y
  - bits cx, y
  - bitc cx, y
  - bitt cx, y
  - mov cx, IMM

22. Do not group more than one MAC instruction (Table 8.4) per group.

23. Do not group more than two instructions that use one ALU (Arithmetic instructions, Table 8.5, and Bitwise Logical instructions, Table 8.6) each, or more than one instruction that uses both the ALUs.

24. Do not group more than one Unlinked Load instruction per group. Unlinked Load instructions (Table 8.9) are all ld and ldx instructions plus the ldu and lddu instructions for which no link has been established.

25. Do not group more than one Store instruction per group. Store instructions are all the Memory Reference instructions (Table 8.9) that start with st.

26. Do not group a Load instruction after a Store instruction, and do not group a Store instruction after an Unlinked Load instruction. Where:
    - Load instructions are all the Memory Reference instructions (Table 8.9) that start with ld.
    - Store instructions are all the Memory Reference instructions (Table 8.9) starting with st.
    - Unlinked Load instructions (Table 8.9) are all ld and ldx instructions plus the ldu and lddu instructions for which no link has been established.

27. Do not group an agn0 instruction under the following circumstances:
    - An agn0 instruction is in the G pipeline stage.
    - A mov loop0, IMM instruction is in the G, R, or E pipeline stage.
    - A mov rX, %loop0 instruction is in the G pipeline stage
    - A bitt %loop0, y instruction is in the G pipeline stage
    - A biti %loop0, y instruction is in the G pipeline stage

- A bits %loop0, y instruction is in the G pipeline stage
- A bitc %loop0, y instruction is in the G pipeline stage

Rule 27 also applies to agn0, agn1, agn2, and ang3, For these instructions, the relevant operands are loop0, loop1, loop2, loop3.

28. Do not group a call IMM or a call rX instruction under the following circumstances:
    - A call IMM or call rX instruction is in the G pipeline stage
    - A mov rX, %rpc instruction is in the G pipeline stage
29. Do not group a mov %fmode, IMM instruction under the following circumstances:
    - An ALU instruction (Arithmetic instructions, Table 8.5, and Bitwise Logical instructions, Table 8.6) is in the G pipeline stage
    - A MAC instruction (Table 8.4) is in the G pipeline stage
30. Do not group an instruction that depends on the result of a previous instruction in the same group, with the following exceptions:
    - The first instruction depends on the result of a Linked Load instruction. Linked Load instructions (Table 8.9) are ldu rX, rY, n and lddu rX, rY, n where a link is established and rY = {r13, r14, r15}.
    - The first instruction is a store instruction, and the second instruction is not a MAC instruction or an unlinked load instruction. Where:
        - ◊ Store instructions are all the Memory Reference instructions (Table 8.9) starting with st.
        - ◊ Unlinked Load instructions (Table 8.9) are all ld and ldx instructions plus the ldu and lddu instructions for which no link has been established.
        - ◊ MAC instructions are all instructions in Table 8.4
31. Stall the pipeline for one processor clock cycle if one of the following instructions is in R pipeline stage:
    - mov %hwflag, rY
    - bits %hwflag, y
    - bitc %hwflag, y
    - biti %hwflag, y

32. Miscellaneous grouping rules:
    - A Conditional Branch instruction must be the first instruction in its group.
    - Do not group more than one Conditional Branch instruction per group.
    - Do not group more than one call IMM or call rX instruction per group.
    - Do not group the following instructions if two or more instructions have the same address register.
        - ◊ ldu
        - ◊ lddu
        - ◊ ldxu
        - ◊ stu
        - ◊ stdu
        - ◊ stxu
    - An reti instruction must be the first instruction in its group.
    - Do not group the following instructions if there is an reti instruction in the G pipeline stage:
        - ◊ mov rX, [imask/ip0/tpc]
        - ◊ bitc [imask/ip0/tpc], y
        - ◊ bitt [imask/ip0/tpc], y
        - ◊ biti [imask/ip0/tpc], y
        - ◊ bits [imask/ip0/tpc], y

4.4 Interrupts

The interrupt controller provides support for 16 interrupt sources. Two sources are not maskable, the rest are individually and globally maskable. The nonmaskable interrupts have a fixed priority level higher than the rest, the rest have a user assignable priority level between 0 to 3. The interrupt controller uses masking, the priority and order within the %ireq register (bit 15 to 0) to determine the highest unmasked priority interrupt to service next. Once taken, it stores the priority of the selected interrupt for use in determining the next interrupt to service when nesting interrupts. Interrupts are normally serviced serially, but an interrupt routine may enable nesting by saving the interrupt state, changing the current interrupt level (if required) and enabling interrupts. Nested interrupts are supported without saving interrupt state for the nonmaskable interrupts.

There are two nonmaskable interrupts: NMI (external to the core), DEI (device emulation interrupt.) NMI has the highest priority (over all interrupts), DEI has the second highest priority (over all maskable interrupts regardless of priority). If either of these two interrupts occur while any maskable interrupt is being serviced, the existing interrupt state is saved and the new interrupt routine is dispatched. If a DEI interrupt is being serviced and an NMI occurs, the existing interrupt state is saved and the NMI routine is dispatched. If while executing a DEI or NMI routine the same interrupt occurs again (executing DEI and another DEI occurs, or executing NMI and another NMI occurs), the processor will restart the interrupt service routine without saving or changing any of the interrupt state. While servicing nonmaskable interrupts, nesting of interrupts is not possible because the interrupt priority level cannot be changed to allow maskable interrupts. To allow this nesting, the hardware will preserve the interrupt state as follows.

When an interrupt is taken, the following state (the interrupt state) changes occur:

- gie (global interrupt enable) is saved in pgie (previous global enable.) gie is then cleared. If a nonmaskable interrupt is interrupting an executing routine, pgie is saved too.
- epl (executing privilege level) is saved in pepl (previous epl.) epl is set to the new interrupt priority level unless a nonmaskable interrupt is occurring (in which epl will not change.) If a nonmaskable interrupt is interrupting an executing routine, pepl will also be saved.
- pc+1 is saved in %tpc (trap program counter.) If a nonmaskable interrupt is interrupting an executing routine, tpc will also be saved.
- Bit in %ireq corresponding to interrupt being serviced is cleared.

If a nonmaskable interrupt occurs while servicing any interrupt routine (except for NMI), the state is preserved (as noted above) and restored at the end of the routine. When the interrupt routine completes, via the ‘reti’ instruction, the following state is restored:

- epl is restored from pepl. pepl is restored to its previous value if completing a nonmaskable interrupt.
- %pc is restored from %tpc. %tpc is restored to its previous value if completing a nonmaskable interrupt.
- gie is restored from pgie. pgie is restored to its previous value if completing a nonmaskable interrupt.

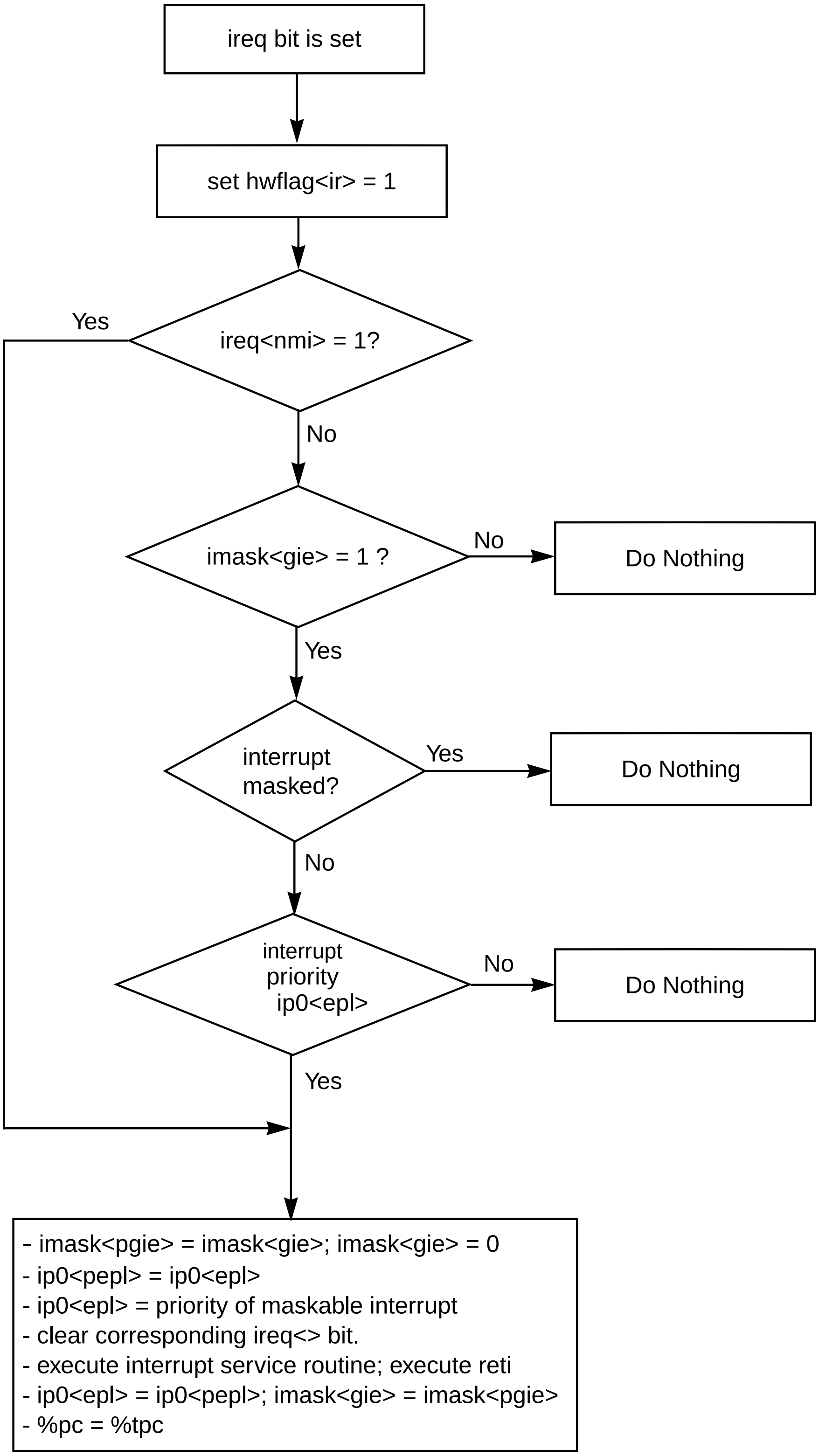
If another maskable interrupt request occurs while an interrupt routine is already executing, it is considered a pending interrupt. This interrupt will be serviced after (unless the interrupt routine intends to allow nesting of interrupt routines - see next paragraph) the current routine completes (the reti instruction is executed.) If there are pending interrupts when the reti instruction executes (and the pending interrupt is not masked or lower priority than the current priority), the next interrupt service routine will be executed without executing any more code in the main routine. The %tpc register will not be changed in this case, the execution will directly flow from one interrupt routine to another interrupt routine. Only after the interrupt routines complete, will main routine code begin execution at the %tpc address.

To nest interrupts, this state (the interrupt state) must be saved: %tpc, epl, pepl. After saving this state, epl (executing privilege level) can be changed to allow lower priority interrupts to occur if necessary. Now set gie (global interrupt enable) to allow new interrupts to occur. At the end of this routine, the interrupt state must be restored before exiting:

- Clear gie (so no interrupts corrupt this state restoration.)
- Restore tpc, pgie, and pepl.
- Issue the ‘reti’ instruction.

Once an interrupt is determined, the core will group no additional instructions and allow the pipe to empty of executing instructions. The new interrupt routine will be fetched and executed. For software enabled interrupts (writes to the interrupt request register) subsequent instructions are flushed and the interrupt routine serviced instead.

Figure 4.2 Interrupt Processing Flow



4.5 Timers

There are two built-in timers. Three control registers control the operation of the timer unit (%timer0, %timer1, and %tc). %timer0 is a 16-bit counter containing the current timer 0 value or the value loaded by the user. %timer1 is a 16-bit counter containing the current timer 1 value or the value loaded by the user. %tc is the 16 bit control register for both timers. Timer 0 utilizes the lower 8-bits of this register, while timer 1 uses the upper 8 bits. Refer to Section 3.20, “Timer Control Register (%tc),” page 3-22.

The control register specifies the mode that the timer will operate in. The user can control when the timer is enabled (et), the mode it operates in single shot or continuous mode (cm) and the prescale value for decrementing the timer (timrdiv).

The timer decrements when it is enabled and the counter is not at 0. The timer can be loaded by the user. If the timer was loaded and is enabled it will begin decrementing. It will decrement based on the prescale value. The prescale value is N+1 clocks for a prescale value of N. Every N+1 clocks the timer will then decrement. Once the timer transitions to zero, an interrupt will be generated. There is an interrupt bit in the interrupt request register for each timer (t0, t1).

Once the timer reaches 0 and has generated the interrupt, it will not count unless the timer is reloaded by the user or the timer is in continuous mode. If continuous mode is enabled, the timer will reload the last value the user loaded into the timer and it will begin decrementing again.

The timer can be disabled at any time while the timer is running. The timer value can be loaded at any time while the timer is running to shorten the timing period. If the timer was loaded with a value of zero and the timer did not contain the value 0 (there was a transition to zero) an interrupt will be generated. If the timer is enabled while the timer count is 0 no interrupt will be generated. The timer will not reload in continuous mode if the timer was loaded with zero by the user.

Chapter 5
Instruction Unit

This chapter explains the ZSP400 Instruction Unit (IU). It includes the following sections:

5.1 Introduction

The instruction unit contains the instruction cache, instruction prefetcher, branch prediction logic, and an instruction dispatcher.

The instruction cache aligns instructions from main memory and reduces main memory power consumption. The prefetcher keeps the instruction cache full when running from on-chip memory and minimizes pipeline stalls. The branch predictor minimizes the need to flush the pipeline.

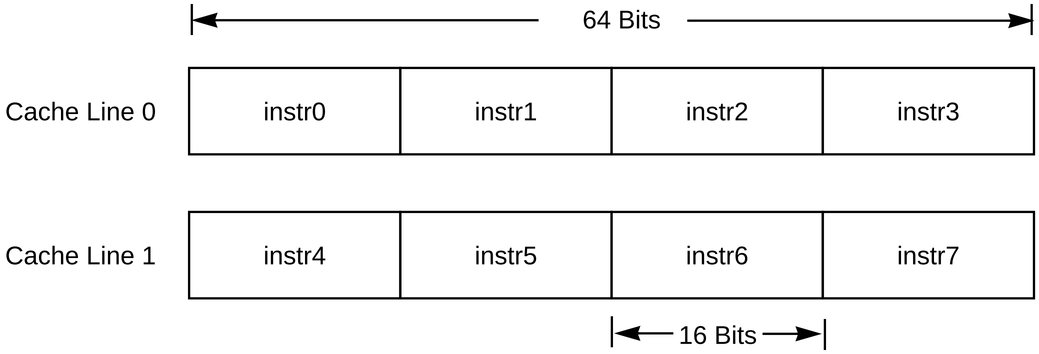
The instruction unit always fetches four instructions from the instruction cache. The instruction dispatcher decodes four instructions. The dispatcher issues up to four instructions to the data unit and the pipeline control unit each cycle. The data and pipeline control units read the required operands from registers or memory and execute the instructions.

5.2 Instruction Cache and Prefetcher

The instruction cache and prefetcher work closely together. These submodules support two primary functions: instruction alignment and main memory power reduction.

On-chip main memory is structured such that four instructions reside in a cache line. Figure 5.1 shows the layout of instructions in a cache line. A maximum of one cache line loads from main memory into the instruction cache each cycle.

Figure 5.1 Cache Line Organization



The instruction cache contains eight cache lines, or 32 instructions. It is a direct mapped cache. In a direct mapped cache, each line in main memory line maps into one specific cache line (the main memory line address modulo the number of cache lines).

5.2.1 Cache Miss Penalty

At every program flow break causing a cache miss, the processor incurs a minimum two cycle penalty.

The two cycle instruction cache miss penalty is illustrated in Figure 5.2. The "–" prefix indicates fetch packets before the branch and the "+" are those after the branch. Thus, +g1, +g2, and +g3 are the fetch packets, or groups of four instructions, after a branch. Suppose two instructions, I0 and I1, are issued from of the +g1 fetch packet. Thus, the +g2 fetch packet needs to start with the unissued instructions I2 and I3 from the last attempt together with new instructions I4 and I5.

Given the scenario of a fetch packet sequence (–g4, –g3, –g2, and –g1) running to a BRANCH where the target of the branch is not in the instruction cache, the following steps are taken.

In cycle n + 1, the target fetch packet is not found in the cache. So, the Cache Line 1 address is sent to main memory.

Cache Line 1 is returned and loaded into the instruction cache in cycle n + 2. Also in cycle n + 2, the address for Cache Line 2 is sent to main memory. Recall that two cache lines must be prefetched into the cache so that the machine can sustain a four instruction issue rate. The pipeline is stalled for both the n + 1 and n + 2 cycles. The +g1 target fetch packet is read from the cache and loaded into the pipeline at cycle n + 3. These two stall cycles represent the two cycle penalty when running from on-chip memory at a program flow discontinuity.

In cycle n + 3, the +g1 fetch packet is sent to the pipeline. Cache Line 2 is loaded into the cache and the address for Cache Line 3 is sent to the main memory.

By cycle n + 4, the +g2 fetch packet is sent to the pipeline. Cache Line 3 is saved in the instruction cache. The address for Cache Line 4 is sent to main memory.

In cycle n + 5, the +g3 fetch packet will be loaded into the pipeline. Cache Line 4 is saved in the instruction cache. The last line that the prefetcher loaded into the instruction cache was Cache Line 3. That is, the contents of Cache Line 3 are available for fetching. However, the +g3 fetch packet does not require any instructions from the previously loaded Cache Line 3. Thus, the prefetcher will halt and not request the next cache line from main memory.

The prefetcher only fetches approximately one cache line in advance. It will stop if the Instruction Unit does not use any of the instructions in the last prefetched line residing in the instruction cache. Stopping the prefetcher keeps it from over running the instruction cache and also reduces power for main memory accesses.

[Only fetch stage is stalled —restof pipe moves with an instruction fetch delay. Only a data unit delay stalls the entire pipe.]

Figure 5.2 Instruction Cache Miss Penalty

Cache Line	+g1, issue I0–I1			
1	I0	I1	I2	I3
2	I4	I5	I6	I7
3	I8	I9	I10	I11
4	I12	I13	I14	I15

+g2, issue I2			
I0	I1	I2	I3
I4	I5	I6	I7
I8	I9	I10	I11
I12	I13	I14	I15

+g3, issue I3			
I0	I1	I2	I3
I4	I5	I6	I7
I8	I9	I10	I11
I12	I13	I14	I15

Fetch Packets

Cycle	F	G	R	E	W	Instruction Sequence
n	BR	–g1	–g2	–g3	–g4	–g4
n + 1 stall	–	BR	–g1	–g2	–g3	–g3
	Target Fetch packet +g1 is not in I-cache. Need to send Cache Line 1’s address to main memory.					–g2 –g1
n + 2 stall	–	–	BR	–g1	–g2	BR TARGET
	Cache Line 1 is loaded into the I-cache. Cache Line 2’s address is sent to main memory.					TARGET: +g1
n + 3	+g1	–	–	BR	–g1	+g2
	Fetch packet +g1 is loaded into the pipeline from the I-cache. Cache Line 2 is saved in the I-Cache. The address for Cache Line 3 is sent to main memory					+g3
n + 4	+g2	+g1	–	–	BR	
	Fetch packet +g2 is loaded into the pipeline from the I-cache. Cache Line 3 is saved in the I-cache. The address for Cache Line 4 is sent to main memory.					
n + 5	+g3	+g2	+g1	–	–	
	Fetch packet +g3 is loaded into the pipeline from the I-cache. Cache Line 4 is saved in the I-cache. The +g3 fetch packet does not use any of the instructions in Cache Line 3, the last cache line saved. Thus, the prefetcher will not request the next cache line from main memory.					

Execution Direction ↓

5.2.2 Cache Line Straddling

The prefetcher is not used when executing from external memory. If the requested group of four fetch instructions are not in the instruction cache, the pipeline stalls while the external memory is accessed and instructions are saved in the cache. The prefetcher does not load instructions before they are requested by the IU. When executing from external memory, the number of pipeline stalls depend on the number of wait states to the memory.

In the event that a fetch packet of four instructions span two cache lines, reading directly from the main memory would require two separate SRAM output ports. Also, the Fetch and Decode stages are combined into one in the ZSP400 pipelines. Fetch directly accesses a register in the instruction cache instead of main memory. The prefetcher works in the background to fetch instructions from main memory.

In order to keep memory design simple by using a single read port, allow a shorter five stage pipeline, and use only a single read port instruction memory, the ZSP400 relies on the instruction cache. The instruction dispatcher will always find four prefetched instructions in the cache barring a break in the program flow, such as an unconditional branch or a branch mispredict.

In a program discontinuity where the next fetch packet (of four instructions) spans two cache lines and is not already available in the cache, both cache lines must be loaded in the instruction cache. This load incurs a three cycle start-up penalty. Thus, aligning the target of a branch to the beginning of a cache line would save one start up cycle.

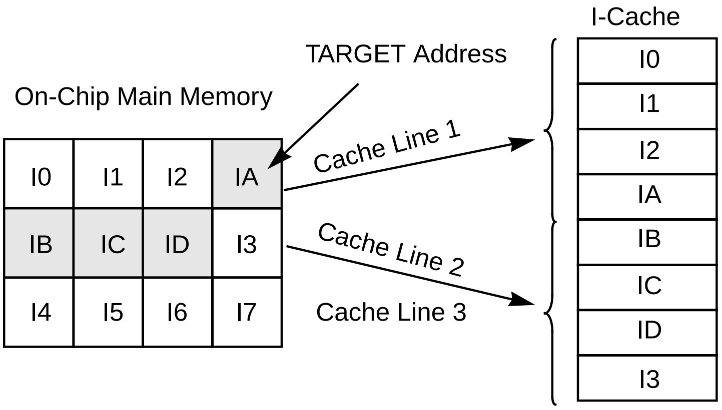
Figure 5.3 shows an example of how the prefetcher and cache scheme solve the data alignment dilemma at a program discontinuity in on-chip memory.

In cycle n + 1, the instruction prefetcher finds that the branch target fetch packet is not in the cache and issues the address for Cache Line 1 to main memory.

Cache Line 1, which contains the first part of the fetch packet, is saved in the instruction cache and the prefetcher sends the address for Cache Line 2 to main memory in cycle n + 2. Cache line 2, which contains the remainder of the target fetch packet, is loaded into the cache in cycle n + 3.

The pipeline is stalled for cycles n + 1, n + 2, and n + 3. By cycle n + 4, however, the entire fetch packet is in the instruction cache and can be used by the Fetch stage of the pipeline. Subsequent instruction fetches will incur no penalty, even if the maximum of four instructions are issued every cycle.

Figure 5.3 Cache and Prefetcher Solve Data Alignment Dilemma



IA, IB, IC, and ID are the instructions required for a fetch. This fetch packet spans two cache lines. Without the cache, two read ports are needed from main memory. With the cache and prefetcher, these four instructions are always available in the cache.

Cycle	F	G	R	E	W	Instruction Sequence
n	BR	–g1	–g2	–g3	–g4	–g4
n + 1 stall	–	BR	–g1	–g2	–g3	–g3
	Target Instruction +g1 is not in I-cache. Need to send Cache Line 1 address to main memory.					–g2 –g1
n + 2 stall	–	–	BR	–g1	–g2	BR TARGET
	Cache Line with IA instruction is loaded into the I-cache. The remainder of the fetch packet is not in the cache. Send Cache Line 2 address to main memory.					TARGET: +g1 +g2
n + 3 stall	–	–	–	BR	–g1	+g3
	Cache line with IB, IC, ID, and I3 instructions are loaded into the cache. The next cache line address is sent to main memory.					Assume that TARGET address is not in the cache.
n + 4	+g1	-	–	–	BR	
	The +g1 instructions are fetched from the instruction cache. The cache line with I4, I5, I6, and I7 are saved in the I-cache. Since Cache Line 2 has been used, the address for Cache Line 4 is sent to main memory.					Also assume all the instructions in a fetch packet are issued. +g1 = {IA, Ib, IC, ID}

Execution Direction

5.2.3 Issue Rate Slower than Prefetch Rate

The prefetcher only loads approximately one cache line ahead. In the event that the instruction dispatcher finds all the instructions in the cache, main memory (whether it is on-chip or off-chip) will not be accessed. Thus, the cache saves main memory from being accessed every cycle and reduces system level power.

Figure 5.4 shows another example of the prefetcher stopping when the issue rate does not keep up with the prefetch rate.

In cycle n + 1, the address for Cache Line 1 is sent to main memory and the pipeline is stalled.

By cycle n + 2, Cache Line 1 is loaded into the instruction cache and the address for Cache Line 2 is sent to main memory. Instructions are not available for reading from the cache, so n + 2 is a stall cycle.

In cycle n + 3, the +g1 fetch packet is sent to the pipeline. Cache Line 2 is saved in the instruction cache and the address for Cache Line 3 is sent to main memory.

During cycle n + 4, the +g2 fetch packet is sent to the pipeline. Cache Line 3 is saved in the instruction cache. The address for Cache Line 4 is sent to main memory.

In cycle n + 5, the +g3 fetch packet is loaded into the pipeline. Cache Line 4 is saved in the instruction cache. The +g3 fetch packet does not use any instructions from the previously loaded Cache Line 4. Thus, the prefetcher stops and does not request a main memory access for the next cache line.

Figure 5.4 Example of Prefetcher Staying Slightly Ahead of Instruction Consumption

Cache Line	+g1, issue I0			
1	I0	I1	I2	I3
2	I4	I5	I6	I7
3	I8	I9	I10	I11
4	I12	I13	I14	I15

+g2, issue I1			
I0	I1	I2	I3
I4	I5	I6	I7
I8	I9	I10	I11
I12	I13	I14	I15

+g3, issue I2			
I0	I1	I2	I3
I4	I5	I6	I7
I8	I9	I10	I11
I12	I13	I14	I15

+g4, issue I3			
I0	I1	I2	I3
I4	I5	I6	I7
I8	I9	I10	I11
I12	I13	I14	I15

Cycle	F	G	R	E	W	Instruction Sequence
n	BR	–g1	–g2	–g3	–g4	–g4
n + 1 stall	–	BR	–g1	–g2	–g3	–g3
	Target Fetch packet +g1 is not in I-cache. Need to send Cache Line 1 address to main memory.					–g2 –g1
n + 2 stall	–	–	BR	–g1	–g2	BR TARGET
	Cache Line 1 is loaded into the I-cache. The address for Cache Line 2 is sent to main memory.					TARGET: +g1
n + 3	+g1	–	–	BR	–g1	+g2
	The +g1 instructions are sent to the pipeline from the I-cache. Cache Line 2 is stored in the I-Cache. The address for Cache Line 3 is sent to main memory					+g3 +g4
n + 4	+g2	+g1	–	–	BR	
	The +g2 fetch packet is sent to the IU. Cache Line 3 is stored in the I-cache. Since the IU reads I4 from the last line saved in the cache, the prefetcher goes ahead and sends the address for Cache Line 4 to main memory.					Assume that the TARGET fetch packet is not in the cache
n + 5	+g3	+g2	+g1	BR	–g1	
	The +g3 instructions are sent to the IU. None of the contents of Cache Line 3, the last line that the prefetcher loaded, are used by the +g3 fetch packet. Thus, the prefetcher does not send a read request to main memory. Meanwhile, Cache Line 4 is stored in the cache.					
n + 6	+g4	+g3	+g2	+g1	BR	
	The +g4 instructions are sent to the IU. None of the contents of Cache Line 4, the last line that the prefetcher loaded, are used by the +g4 fetch packet. Thus, the prefetcher does not send a read request to main memory.					

Execution Direction ↓

To summarize, a two cycle setup penalty is incurred if the first fetch packet after a program discontinuity fits within a cache line and is not already in the instruction cache. Otherwise, a three cycle penalty is incurred since two cache lines must be retrieved from main memory. These values are only relevant for program execution from on-chip memory. Off-chip program execution does not use the prefetcher and depends on the number of wait states to external memory.

5.3 Branch Prediction

The ZSP400 architecture uses static branch prediction. In static branch prediction, the direction of conditional branches are based on the branch type and the branch direction. The prefetcher assumes the branch target and loads the pipeline accordingly. In the event that the branch assumption is incorrect, the pipeline has to be flushed and instructions from the actual target need to be loaded. In most cases, the prediction is correct and the branch incurs zero penalty.

Using static branch prediction, there is no need for branch delay slots found in other processors.

Table 5.1 shows the ZSP400 static branch prediction rules.

Table 5.1 Static Branch Prediction Rules

Instruction	Branch Direction	Notes	Prediction
br rX	either		not taken
br IMM	either		taken
bz IMM	either		taken
bnz IMM	backward		taken
bnz IMM	forward		not taken
blt IMM	backward		taken
blt IMM	forward		not taken
ble IMM	backward		taken
ble IMM	forward		not taken

Table 5.1 Static Branch Prediction Rules (Cont.)

Instruction	Branch Direction	Notes	Prediction
bgt IMM	either		taken
bge IMM	either		taken
bov IMM	either		taken
bnov IMM	backward		taken
bnov IMM	forward		not taken
bc IMM	either		taken
bnc	backward		taken
bnc	forward		not taken
agnX IMM	backward	loopX! = 0	taken
call rX	either		not taken
call IMM	either		taken
ret	either	no instructions in the pipeline will modify rpc	taken
ret	either	at least one instruction in the pipeline will modify rpc	not taken
reti	either	no instruction in the pipeline will modify tpc	taken
reti	either	at least one instruction in the pipeline will modify tpc	not taken

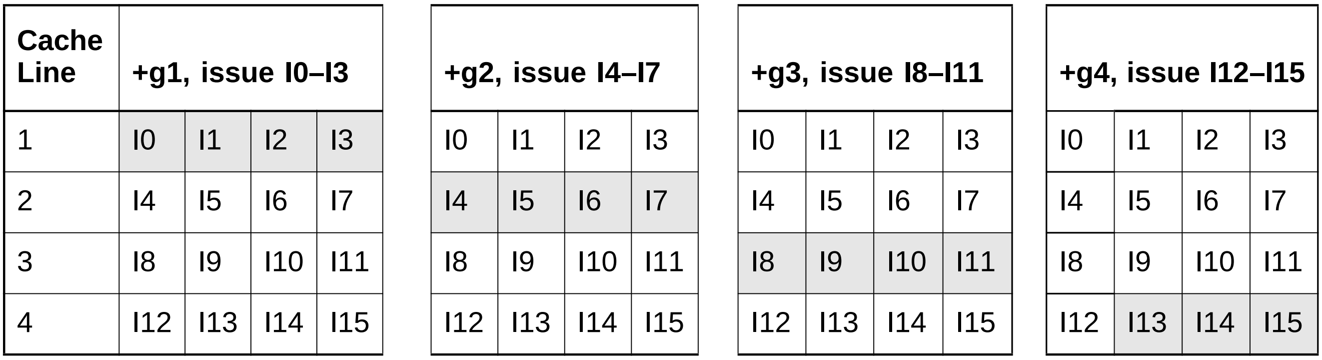
A mispredicted branch when running from on-chip memory usually incurs a five cycle penalty if the mispredicted fetch packet is aligned within one cache line. Otherwise, the misprediction penalty is six cycles because another cache line must be fetched into the instruction cache. When the branch error is discovered, the branch instruction is in a group of instructions at the execute stage. Suppose this time is called Cycle N. Figure 5.5 lists the events occurring in the subsequent clock cycles.

In cycle n + 1, the pipeline stalls as the actual target address is sent to instruction memory. The instruction cache is loaded from memory in cycle n + 2 while the pipeline is kept in a stalled state. Cycles n + 3, n + 4, and n + 5 flush the mispredicted instructions out of the pipeline. In cycle n + 6, the first instruction of the mispredicted branch is executed.

In the event that the mispredicted fetch packet is in the instruction cache, the two stall cycles for cache loading are not needed. Thus, the processor will only encounter a three cycle pipeline flush penalty on the branch mispredict.

To summarize, a three cycle mispredict penalty is encountered if the branch target instructions are in the cache. Otherwise, the branch mispredict penalty is five cycles if the branch target is aligned to the beginning of a cache line. In the event that the branch target is not aligned at the beginning of a word line, the penalty is six cycles.

Figure 5.5 Explanation of Branch Misprediction Penalties



Cache Line	+g1, issue I0–I3			
1	I0	I1	I2	I3
2	I4	I5	I6	I7
3	I8	I9	I10	I11
4	I12	I13	I14	I15

+g2, issue I4–I7			
I0	I1	I2	I3
I4	I5	I6	I7
I8	I9	I10	I11
I12	I13	I14	I15

+g3, issue I8–I11			
I0	I1	I2	I3
I4	I5	I6	I7
I8	I9	I10	I11
I12	I13	I14	I15

+g4, issue I12–I15			
I0	I1	I2	I3
I4	I5	I6	I7
I8	I9	I10	I11
I12	I13	I14	I15

Figure 5.5 Explanation of Branch Misprediction Penalties (Cont.)

Cycle	F	G	R	E	W	Instruction Sequence
n	–g1	–g2	–g3	BNE	–g1	
	Status flags are checked for ! = 0. Finds that a mispredict occurred.					START
n + 1	–	–	–	-	BNE	–g3 Execution Direction
	Since +g1 is not in the cache, send the address for Cache Line 1 to main memory.					-g2 -g1
n + 2	–	–	–	-	-	BNE START
	Cache Line 1 is saved in the I-cache. Address for Cache Line 2 is sent to main memory.					+g1 +g2
n + 3 Flush	+g1	–	–	–	-	+g3
	The +g1 fetch packet is loaded into the pipeline. Cache Line 2 is saved in the I-cache. The address for Cache Line 3 is sent to main memory.					The BNE predicts that the branch direction is to START. However, the program flow falls through to +g1 in this example.
n + 4 Flush	+g2	+g1	–	–	–	
	The +g2 fetch packet is loaded into the pipeline. Cache Line 3 is saved in the I-cache. The address for Cache Line 4 is sent to main memory.					
n + 5 Flush	+g3	+g2	+g1	–	–	Assume that the loop has been executing through the pipeline. In cycle n, there is a mispredict because the loop is exiting into the +g1 fetch group.
	The +g3 fetch packet is loaded into the pipeline. Cache Line 4 is saved in the I-cache. The address for the next cache line is sent to main memory.					
n + 6	+g4	+g3	+g2	+g1	–	
	The first mispredicted program flow instruction reaches the E stage.					This illustration implies that all four instructions in the +g1 fetch packet are found in one cache line. Otherwise, an extra stall cycle after n + 2 is required to load the remainder of the +g1 fetch packet.

Chapter 6
Data Unit

This chapter explains the ZSP400 data unit. It contains the following sections:

- Section 6.1, "Introduction," page 6-1 on page 6-1
- Section 6.2, "Data Cache, Data Prefetcher, and Data Linking," page 6-2
- Section 6.3, "Data Linking Setup," page 6-5
- Section 6.4, "Data Unit Stores," page 6-6
- Section 6.5, "Circular Buffers," page 6-8
- Section 6.6, "Reverse Carry Addressing," page 6-10

6.1 Introduction

The data unit (DU) is a comprised of the data cache, data prefetcher, and the circular buffer unit. The DU is also responsible for data linking, a powerful concept that alleviates loads of operands from memory into general purpose registers before they can be used.

DSP applications often require streaming data. For example, in a filtering operation, two operands are read, operated upon, and the result saved in a register. This process is set in a long loop. Operands are "streamed" into the execution unit.

In a RISC machine, one cycle is needed for each of the two operands loads into general purpose registers. For DSP applications where these type of loops are often found, direct-from-memory operand reads enhance program efficiency.

As opposed to general purpose computing, DSP data is saved in an orderly fashion in memory. That is, operands are generally arranged in sequential data memory addresses. The ZSP400 architecture, while typically a load/store machine, can be made to do operand and result data streaming by using this concept of data linking. When a set of contiguous memory locations are linked, the operands are read during the R stage of the pipeline and used directly in the E stage without requiring that they first be loaded into a general purpose register.

The ZSP400 architecture does not impose any memory alignment restrictions on extended precision operands. For example, some architectures require that a double word operand be aligned such that it starts on an even address. Without this restriction, the ZSP400 is friendlier to program and eases the compiler complexity.

6.2 Data Cache, Data Prefetcher, and Data Linking

The data cache and data prefetcher perform the same functions as the instruction cache and prefetcher. The data cache is a fully associative write-through cache consisting of 17 lines. Each cache line contains four single precision words of 16 bits. For the most part, the cache and prefetcher are needed to ensure that double precision, or 32-bit, operands, can be accessed via a single read port memory in one cycle without any stalls once setup is complete. That is, if an operand should straddle two cache lines, both lines would be available in the cache when the data is needed.

Data linking is established by storing values to data linking index registers. These index registers, located in the Execution Units's general purpose register file, are r13, r14, and r15. Whenever an address is saved in a data linking index register, the DU resets the link pointer. When the data referenced by the link register is next used in a load instruction, the data prefetcher brings two cache lines into the data cache and establishes the data link.

The ZSP400 supports three data linking registers, or three discrete sets of contiguous data streams. Once the data linking setup is complete, contiguous operand accesses to any of these three sets of data incur zero cycles for a register load. In other words, these linked regions may be accessed with no load penalty if accessed in a sequentially increasing order.

Two cache lines are needed to ensure that the cache contains valid data for each data link throughout the streaming loop. Each cycle, four 16-bit operands or two 32-bit operands can be used. Hence, a maximum of 64 bits can be consumed each cycle with 32 bits from one data link and 32 bits from the other.

From main data memory, one cache line of 64 bits can be fetched each cycle. This cache line is specific to one set of operands, or one data link. From a system standpoint, the fill rate of 64 bits per cycle matches the consumption rate of 64 bits per cycle. The data prefetcher can only service one data link each cycle, though. Thus, two cache lines usually need to be prefetched in the setup sequence to ensure that the DSP does not run out of data in a loop.

However, if the first operand in a data link is aligned to the beginning of a cache line, then the second cache line fetch is not immediately needed because the next operand of 32 bits is already in the same cache line. The data prefetcher can always fill the cache faster than the machine can consume data from the link. Thus, the DSP can save one cycle in data linking setup.

Once two cache lines from each linked set are loaded in the cache, data is guaranteed to remain in the cache at the fastest operand data rate.

Only two of the three data linking pointers may be used in any given cycle. The third data linking register is a convenience which allows the programmer to switch to another set of data without resetting one of the two existing links.

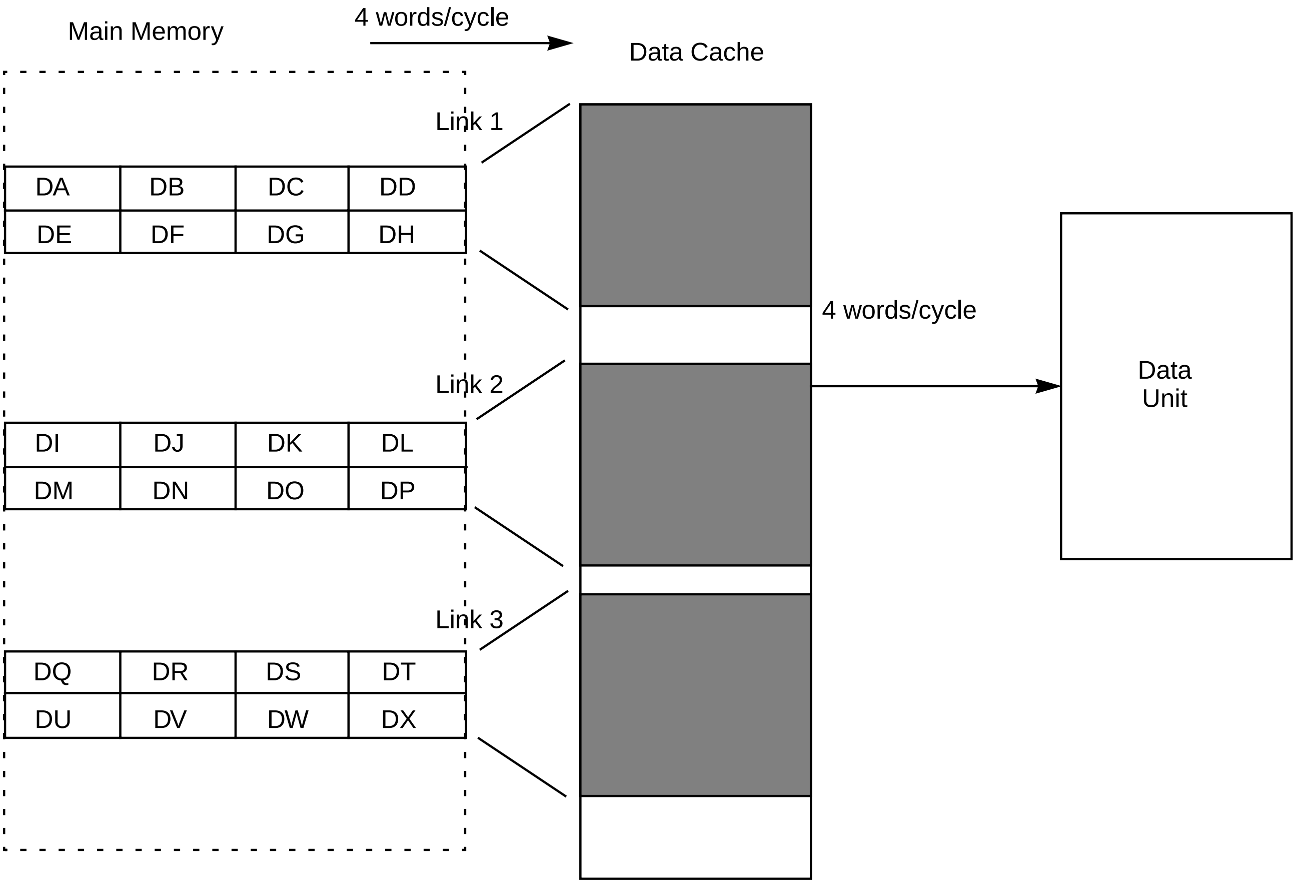
The three data links can each use three cache lines. If two or more data links are to the same address, then obviously redundant cache lines are not loaded into the data cache. The remaining three cache lines are used for operands when accessing other general purpose registers. Thus, even if a general purpose register is loaded from internal memory, the entire cache line containing that value will be loaded into the data cache.

In addition to the setup penalty required for bringing two cache lines into the data cache, the data linking setup incurs an extra penalty due to checking for circular buffer boundaries.

Since the setup of the data linking index registers incurs these cycle penalties, they should not normally be used as general purpose registers in load instructions. These registers do not incur a setup penalty if they are used in other instructions. Data linking registers will always work as general purpose registers.

Figure 6.1 illustrates the data linking concept.

Figure 6.1 Data Linking in Detail



Up to three Data Links may be established from main memory to the data cache. At setup, two cache lines from each link are read into the Data Cache. Once setup has completed, up to four 16-bit operands or two 32-bit operands may be read by the Data Unit continuously. The data prefetcher can maintain cache fullness by loading one cache line from main memory every cycle. Note that extended precision operands spanning two cache lines will be loaded into the data cache such that both halves are available in the same cycle.

Of course, the side benefits of the data cache is that it lowers system level power requirements. A cache line containing operands is not usually required every cycle from main memory. Thus, it is not necessary to do a main memory accesses when operands are not needed.

Data linking does not work when operands are in external memory space. Operands from external memory must be loaded using a dedicated cycle into a general purpose register in the E stage before they can be used in calculations. From a programming point of view, though, this difference is transparent. The programming model remains the same. The DU just forces the operands to load into a general purpose register before using them.

6.3 Data Linking Setup

Once a value in a link register is changed by a move or load, that particular link is reset. The next time this link register is used as an index for a load instruction, a linking setup sequence is required to establish the new link. This sequence involves loading two cache lines from main memory into the data cache. Figure 6.2 shows the link setup. This example assumes that data is in on-chip memory.

The instruction sequence first resets the link by moving a new value into r15. Next, three sets of load and add instructions are grouped together. The pipeline diagram shows three stalls as two cache lines are read from main memory in cycles n + 1, n + 2, and n + 3.

In cycle n + 3, the first operand for instruction i3 is read from the data cache into the operand bypass register. By cycle n + 4, the pipeline can start operating with no data stalls. The data prefetcher can always keep the three data links full since the maximum main memory bandwidth is four words per cycle and the maximum operand consumption bandwidth is also four words per cycle.

The data linking setup also has to check the circular buffer end registers when first loading the two cache lines. That is, the first cache line in the data linking setup could be at a circular buffer boundary. Thus, the second cache line needs to be fetched from the circular buffer start address.

Figure 6.2 Example of Data Linking Setup

	F	G	R	E	W
n	g4	g3	i2 i3	i1	?
	r15 data link is reset.				
n + 1 Stall	g5	g4	i4 i5	i2 i3	i1
	First load on a cleared data link. Send mem(r15) address to data memory. Increment r15.				
n + 2 Stall	g5	g4	i4 i5	i2 i3	i1
	Save Cache Line 1 into the data cache. Send mem(r15) address to data memory. Increment r15.				
n + 3 Stall	g5	g4	i4 i5	i2 i3	i1
	Load ALU bypass register with D0 from the data cache. Save Cache Line 2 into data cache. Increment r15.				
n + 4	g6	g5	i4 i5	i2 i3	i1
	Execute i2/i3. Load D1 into ALU bypass register from data cache. Increment r15.				
n + 5	g7	g6	i6 i7	i4 i5	i2 i3
	Execute i4/i5. Load D2 into ALU bypass register from data cache. Increment r15.				

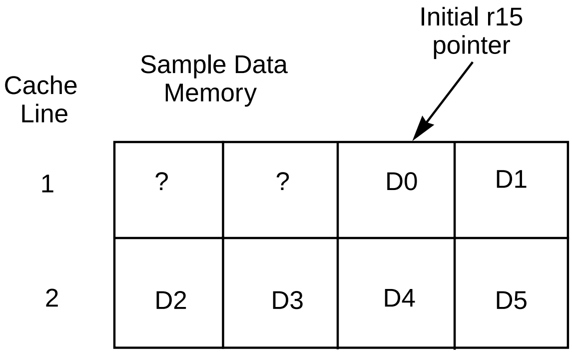
Instruction Sequence

group 1 i1: mov r15, r0 ! reset r15 link

group 2 { i2: ldu r2, r15, 1 ! 1st load
i3: add r1, r2 ! use D0

group 3 { i4: ldu r2, r15, 1 ! use D1
i5: add r1, r2

group 4 { i6: ldu r2, r15, 1 ! use D2
i7: add r1, r2



Sample Data Memory

Cache Line				
1	?	?	D0	D1
2	D2	D3	D4	D5

6.4 Data Unit Stores

The data cache is write through–when data is stored to memory in the W stage of the pipeline, the processor writes the data to both the main memory and the data cache.

In the event of an extended precision 32-bit store where the data word straddles two cache lines, the pipeline stalls one cycle when running from on-chip data memory to allow both lines to be written. Figure 6.3 illustrates this extended precision cache-line straddling store.

Figure 6.3 Double Operand Store Straddling Two Cache Lines

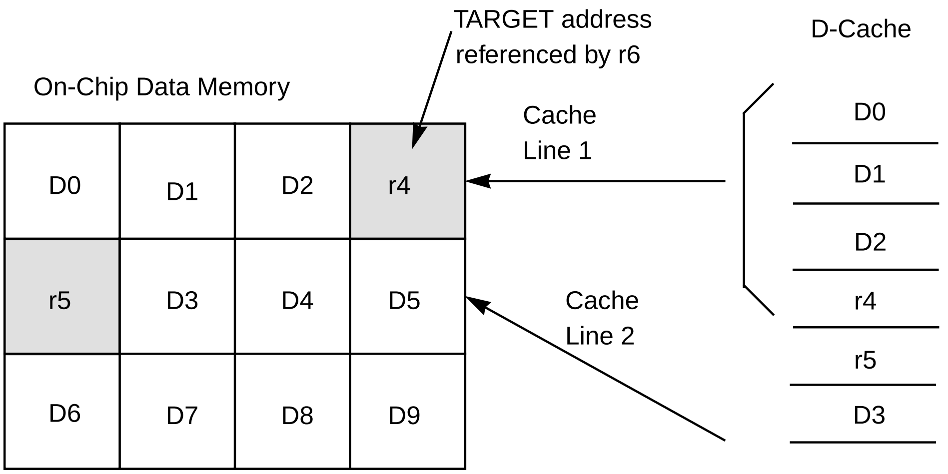
Cycle	F	G	R	E	W
n	+g3	+g2	+g1	STDU	–g1
n + 1	+g3	+g2	+g1	STDU	–
Stall	The address and data for Cache Line 1 are sent to on-chip main memory.				
n + 2	+g4	+g3	+g2	+g1	STDU
Stall	The address and data for Cache Line 2 are sent to on-chip main memory,				
n + 3	+g5	+g4	+g3	+g2	+g1

Instruction Sequence:

-g1
STDU r4, r6, 2
+g1
+g2
+g3
+g4
+g5

Execution Direction

The STDU instruction will save the r4–r5 registers into the memory location referenced by r6 and update the r6 pointers. Assume that the address in r6 straddles two cache lines.



6.5 Circular Buffers

The ZSP400 supports two circular buffers. A circular buffer is defined by a programmable starting address and an ending address. These addresses can exist anywhere in data memory and do not need to be aligned in any way. However, the buffer end address must be greater than the buffer start address. If the buffer end address is less than the buffer start address, then the circular buffer is not considered enabled. The minimum size of a circular buffer is two elements. Sizes smaller than two elements will not enable the circular buffer. The size of the buffer is defined as the difference between the end address and the start address. The last work of the circular buffer is at end-address –1.

When using an index register to automatically increment addresses for a load or store operation, the index value will automatically wrap around to the buffer starting address once it crosses the buffer end boundary. Only positive increments are supported. Negative increments (decrements) are not affected by the circular buffer operation, they decrement the required amount.

Data linking comprehends circular buffers. When a circular buffer control register is modified, the processor re-establishes all the data links.

Table 6.1 shows the functionality of circular buffer 0 loads, and Table 6.2 shows the functionality of circular buffer 0 stores. The functionality of circular buffer 1 loads and stores is the same as for circular buffer 0, except r15 replaces r14, cb1_beg replaces cb0_beg, and cb1_end replaces cb0_end.

Table 6.1 Circular Buffer 0 (cb0) Load Operations

Instruction	Current r14	Next rX	Next r(X + 1)	Next r14
lddu rX, r14, 2	< cb0_end – 2	mem[r14]	mem[r14 + 1]	r14 + 2
lddu rX, r14, 2	cb0_end – 2	mem[r14]	mem[r14 + 1]	cb0_beg
lddu rX, r14, 2	cb0_end – 1	mem[r14]	mem[cb0_beg]	cb0_beg + 1
lddu rX, r14, 2	≥ cb0_end	mem[r14]	mem[r14 + 1]	r14 + 2
ldu rX, r14, 2	< cb0_end – 2	mem[r14]	–	r14 + 2

Table 6.1 Circular Buffer 0 (cb0) Load Operations (Cont.)

Instruction	Current r14	Next rX	Next r(X + 1)	Next r14
ldu rX, r14, 2	cb0_end – 2	mem[r14]	–	cb0_beg
ldu rX, r14, 2	cb0_end – 1	mem[r14]	–	cb0_beg + 1
ldu rX, r14, 2	≥ cb0_end	mem[r14]	–	r14 + 2
ldu rX, r14, 1	< cb0_end – 1	mem[r14]	–	r14 + 1
ldu rX, r14, 1	cb0_end – 1	mem[r14]	–	cb0_beg
ldu rX, r14, 1	≥ cb0_end	mem[r14]	–	r14 + 1

Table 6.2 Circular Buffer 0 (cb0) Store Operations

Instruction	Current r14	rX ->	r(X + 1) ->	Next r14
stdu rX, r14, 2	< cb0_end – 2	mem[r14]	mem[r14 + 1]	r14 + 2
stdu rX, r14, 2	cb0_end – 2	mem[r14]	mem[r14 + 1]	cb0_beg
stdu rX, r14, 2	cb0_end – 1	mem[r14]	mem[cb0_beg]	cb0_beg + 1
stdu rX, r14, 2	≥ cb0_end	mem[r14]	mem[r14 + 1]	r14 + 2
stu rX, r14, 2	< cb0_end – 2	mem[r14]	–	r14 + 2
stu rX, r14, 2	cb0_end – 2	mem[r14]	–	cb0_beg
stu rX, r14, 2	cb0_end – 1	mem[r14]	–	cb0_beg + 1
stu rX, r14, 2	≥ cb0_end	mem[r14]	–	r14 + 2
stu rX, r14, 1	< cb0_end – 1	mem[r14]	–	r14 + 1
stu rX, r14, 1	cb0_end – 1	mem[r14]	–	cb0_beg
stu rX, r14, 1	≥ cb0_end	mem[r14]	–	r14 + 1

The circular buffer index pointer automatically wraps around if the update value on an `ldu`, `lddu`, `stu`, or `stdu` causes the pointer address to equal or exceed the circular buffer end address.

For example, if circular buffer 0 is enabled:

```
If (cb0_end > r14 ≥ cb0_beg) and (r14 + update ≥ cb0_end)
then
          r14 ← cb0_beg + (r14 + update - cb0_end).
else
          r14 ← r14 + update.
```

6.6 Reverse Carry Addressing

The ZSP400 supports an alternate mode of indexing the base address registers. This mode is called reverse-carry addressing (rca.) This mode causes the address update of ldu, lddu, stu, or stdu instructions to be modified as described below. This addressing mode only works with address base registers R0 through R12.

The idea behind reverse-carry addressing is to speed up FFT and other similar operations that require the next load or next store address to be modified in a reverse-carry fashion. Typically, these algorithms work on a buffer of 2^N words, which are aligned at a 2^N word boundary. In these instances, the reverse-carry width of N is used.

With regular addressing, an address is updated by adding 1 or 2 to the least significant bit position, and the carry out (if any) propagate to the left. But with reverse-carry addressing, an address is updated by adding a 1 to the 'N – 1' bit position, and the carry out (if any) propagates to the right.

This is best illustrated by an example: Suppose we enable reverse-carry addressing on loads-with-update with a reverse bit length of 4 (N = 4). Thus, the %amode register will be: 0000 0000 0001 0001. If our address, stored in R4, is initialized to 0x0000, and the above reverse-carry addressing is employed with the below instruction stream, then the update address will be as follows:

```
ldu r0, r4, 1       new r4 = 0000 0000 0000 1000
ldu r0, r4, 1       new r4 = 0000 0000 0000 0100
ldu r0, r4, 1       new r4 = 0000 0000 0000 1100
ldu r0, r4, 1       new r4 = 0000 0000 0000 0010
...    ...          ...    ...    ...    ...
ldu r0, r4, 1       new r4 = 0000 0000 0000 1111
ldu r0, r4, 1       new r4 = 0000 0000 0000 0000
```

Notice how a ‘1’ is added to bit position 3 and the carry is propagated to the right. When there are all 1’s in the final 4 bit positions, these bits become zero and the carry-out is discarded. This has the affect of wrapping the address around the initial value of 0x0000.

It is important to note that reverse-carry addressing will also work with offsets other than +1. The usual offsets of –2, –1, +1, and +2 all work with reverse-carry addressing. Instead of adding a 1 to bit position N – 1, for example, a –2 can be added at that position and the carry will propagate to the right. It is usually the programmer’s responsibility to align the data buffer at a 2^N word boundary, so that proper wrap-around operation is insured during reverse-carry addressing.

Chapter 7
Execution Unit

This chapter explains the ZSP400 execution unit. It contains the following sections:

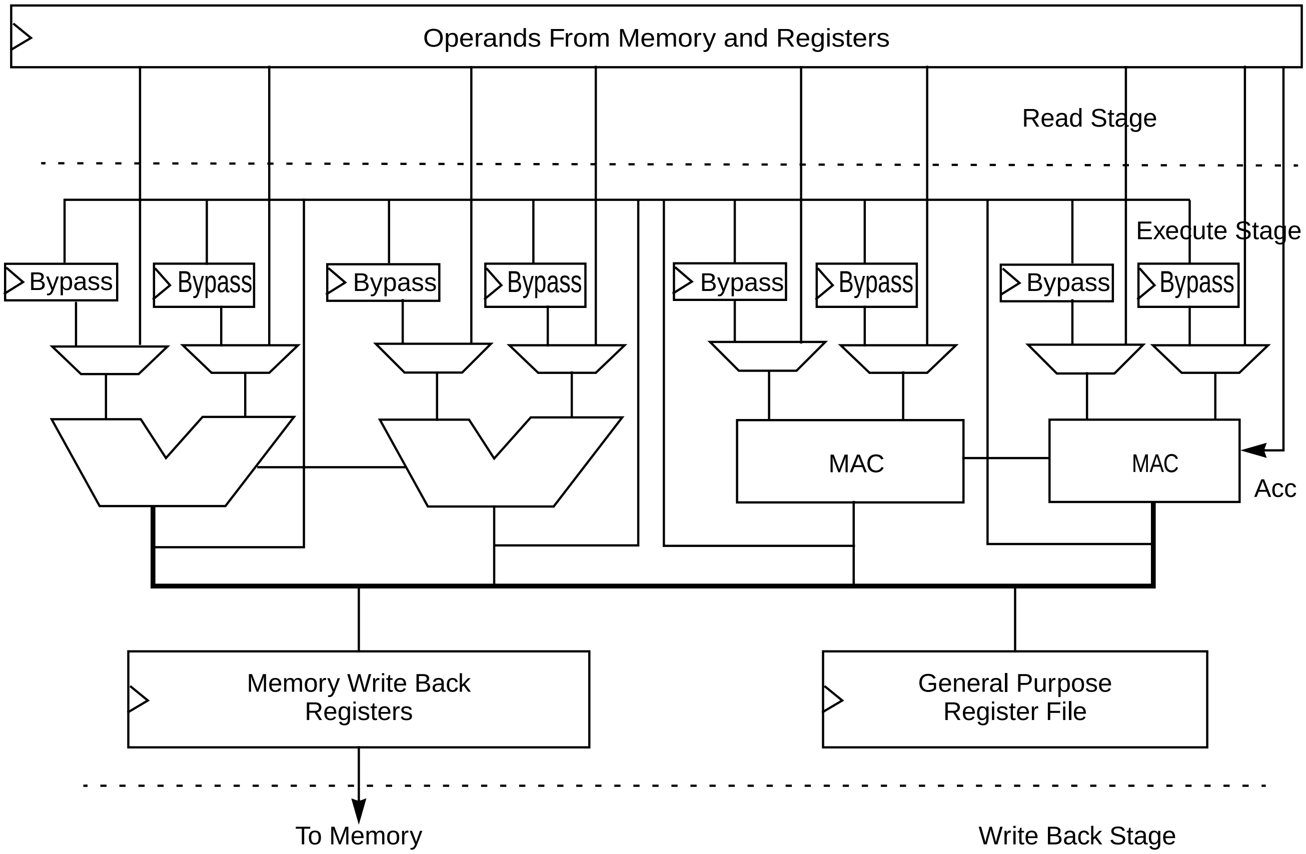
7.1 Introduction

The execution unit performs all the arithmetic and logical operations in the DSP. The execution unit contains two identical 16 bit arithmetic logic units (ALUs), two 16 X 16 multiply and accumulate (MAC) units, and a general purpose register file.

The two ALUs can be combined to form a single 32 bit ALU. The MAC units can perform two 16-bit X 16-bit multiply operations followed by a single 40-bit accumulation or one 32-bit X 32-bit multiply followed by a 40-bit accumulation per cycle. Both MACs share one adder for the accumulate operation.

Figure 7.1 describes the EXU data path.

Figure 7.1 Execution Unit Datapath



7.2 Arithmetic Logic Units (ALU)

The ZSP400 has two identical 16 bit arithmetic logic units (ALU), which can be combined as a single 32 bit ALU. ALU functionality includes addition, subtraction, left and right shift, all basic logic operations, negation, absolute value calculation, rounding and normalization. The ALU also implements bit manipulation instructions.

ALU operations affect the following hwflag register bits:

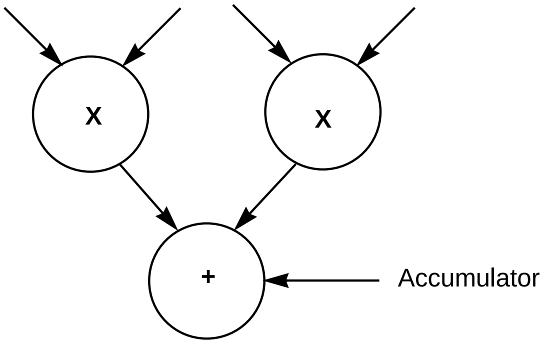
- carry
- zero
- overflow
- gt (greater than)
- ge (greater or equal to zero)

All ALU operations are single cycle operations. An ALU result bypass mechanism allows the results from the ALUs to be used in subsequent instructions in the next cycle by any functional unit without requiring that they be written back to a register first.

7.3 Multiply Accumulate Units (MAC)

The ZSP400 can perform two 16-bit x 16-bit multiply operations followed by a single 40-bit accumulation or one 32-bit x 32-bit multiply followed by a 40-bit accumulation per cycle. Both MACs share one adder for the accumulate. Figure 7.2 shows the dual MAC approach.

Figure 7.2 Dual MAC



MAC hardware performs two instruction Viterbi butterfly operations.

The parallel add and subtract (padd and psub) instructions allow Integer intensive code to use the MAC accumulator as two 16-bit adder/subtractors. Mode bit settings do not affect the padd and psub instructions, nor do they set any flags.

Normal MAC operations affect the ge (greater than or equal to zero) and overflow flags.

The MAC result bypass mechanism allows the results from the MACs to be used in subsequent instructions in the next cycle by any functional unit without requiring that they be written into the operand register file first.

Saturation and rounding are supported depending on the instruction and the operating mode settings.

7.4 General Purpose Register File

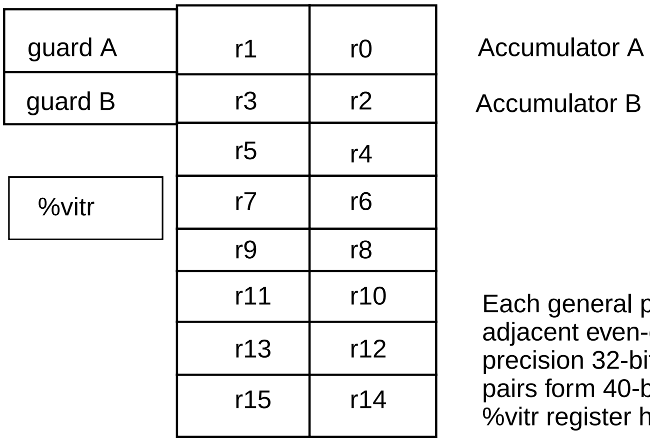
The baseline ZSP400 general purpose register file contains sixteen 16-bit registers labelled r0 to r15. Each of these registers may be used as the input or output of any functional unit. Three registers in this set, r13–15, are used to establish data linking.

To form extended precision 32-bit registers, use two adjacent even-odd pairs. The even-numbered register specifies the extended precision register in an extended precision instruction.

The r0–r1 pair along with an eight bit guard in the control register file form the 40-bit A accumulator. Likewise, the r2–r3 register pair and another eight bit control register guard comprise the 40-bit B accumulator.

In addition to these registers, Figure 7.3 shows the organization of the base general purpose register file.

Figure 7.3 General Purpose Register File



Each general purpose register is 16 bits. An adjacent even-odd pair form an extended precision 32-bit register. The r0–1 and r2–3 pairs form 40-bit Accumulators A and B. The %vitr register holds the viterbi trace back bits.

The ZSP400 provides a flexible stack and structure access by allowing any general purpose register to be used as a stack pointer.

7.5 Shadow Registers

General Purpose registers r2–r9 have a set of shadow registers. These shadow registers are exchanged for the primary registers when the shadow bit in the %smode register is set.

Both register sets (primary and shadow) preserve their values when they are exchanged, so the registers can be used to preserve processor states. For examples, interrupt state, special subroutine state, or control routine state.

Chapter 8
ZSP400 Instruction Set

This chapter provides detailed information on the ZSP400 instruction set for the ZSP400 family of processors, and contains the following sections:

8.1 Functional and Execution Unit Usage

Table 8.1 shows the functional unit (ALU or MAC) used by each instruction, and the stage in which the instruction executes.

A ▲ indicates that the instruction uses a particular unit and a ֍ indicates that an ALU is used only if the load instruction does not complete in the R stage.

Table 8.1 Instruction Functional Unit Usage and Execution Stage

Instruction		One ALU	Both ALU	MAC	Execution Stage
mov	rX, rY	▲			E
mov	cX, rY	▲			E
mov	rX, cX	▲			E
mov	*%pc*, cY				G
mov	rX, IMM	▲			E
movl	rX, IMM	▲			E
movh	rX, IMM	▲			E
movl	cX, IMM				R
movh	cX, IMM				R
mac.a	rX, rY			▲	E
mac.b	rX, rY			▲	E
macn.a	rX, rY			▲	E
macn.b	rX, rY			▲	E
mul.a	rX, rY			▲	E
mul.b	rX, rY			▲	E
muln.a	rX, rY			▲	E
muln.b	rX, rY			▲	E
mac2.a	rX, rY			▲	E
mac2.b	rX, rY			▲	E
cmacr.a	rX, rY			▲	E

(Sheet 1 of 6)

Table 8.1 Instruction Functional Unit Usage and Execution Stage (Cont.)

Instruction		One ALU	Both ALU	MAC	Execution Stage
cmacr.b	rX, rY			▲	E
cmaci.a	rX, rY			▲	E
cmaci.b	rX, rY			▲	E
cmulr.a	rX, rY			▲	E
cmulr.b	rX, rY			▲	E
cmuli.a	rX, rY			▲	E
cmali.b	rX, rY			▲	E
dmac.a	rX, rY			▲	E
dmac.b	rX, rY			▲	E
dmul.a	rX, rY			▲	E
dmul.b	rX, rY			▲	E
imul.a	rX, rY			▲	E
imul.b	rX, rY			▲	E
padd.a	rX, rY			▲	E
padd.b	rX, rY			▲	E
psub.a	rX, rY			▲	E
psub.b	rX, rY			▲	E
norm	rX, rY	▲			E
norm.e	rX, rY		▲		E
add	rX, rY	▲			E
add.e	rX, rY		▲		E
add	rX, IMM	▲			E
addc.e	rX, rY		▲		E
(Sheet 2 of 6)					

Table 8.1 Instruction Functional Unit Usage and Execution Stage (Cont.)

Instruction		One ALU	Both ALU	MAC	Execution Stage
sub	rX, rY	▲			E
sub.e	rX, rY		▲		E
subc.e	rX, rY		▲		E
cmp	rX, rY	▲			E
cmp.e	rX, rY		▲		E
cmp	rX, IMM	▲			E
abs	rX, rY	▲			E
abs.e	rX, rY		▲		E
shla	rX, rY	▲			E
shla.e	rX, rY		▲		E
shla	rX, IMM	▲			E
shla.e	rX, IMM		▲		E
shra	rX, rY	▲			E
shra.e	rX, rY		▲		E
shra	rX, IMM	▲			E
shra.e	rX, IMM		▲		E
min	rX, rY	▲			E
min.e	rX, rY		▲		E
max	rX, rY	▲			E
max.e	rX, rY		▲		E
round.e	rX, rY		▲		E
vit_a	rX, rY			▲	E
vit_b	rX, rY			▲	E

(Sheet 3 of 6)

Table 8.1 Instruction Functional Unit Usage and Execution Stage (Cont.)

Instruction		One ALU	Both ALU	MAC	Execution Stage
and	rX, rY	▲			E
and.e	rX, rY		▲		E
or	rX, rY	▲			E
or.e	rX, rY		▲		E
xor	rX, rY	▲			E
xor.e	rX, rY		▲		E
neg	rX, rY	▲			E
neg.e	rX, rY		▲		E
not	rX, rY	▲			E
not.e	rX, rY		▲		E
bitc	rX, IMM	▲			E
bitc	cX, IMM	▲			E
bits	rX, IMM	▲			E
bits	cX, IMM	▲			E
biti	rX, IMM	▲			E
biti	cX, IMM	▲			E
bitt	rX, IMM	▲			E
bitt	cX, IMM	▲			E
revb	rX, IMM	▲			E
shll	rX, rY	▲			E
shll.e	rX, rY		▲		E
shll	rX, IMM	▲			E
shll.e	rX, IMM		▲		E

(Sheet 4 of 6)

Table 8.1 Instruction Functional Unit Usage and Execution Stage (Cont.)

Instruction		One ALU	Both ALU	MAC	Execution Stage
shrl	rX, rY	▲			E
shrl.e	rX, rY		▲		E
shrl	rX, IMM	▲			E
shrl.e	rX, IMM		▲		E
br	LABEL				E
bz	LABEL				E
bnz	LABEL				E
blt	LABEL				E
ble	LABEL				E
bgt	LABEL				E
bge	LABEL				E
bov	LABEL				E
bnov	LABEL				E
bc	LABEL				E
bnc	LABEL				E
agn0	LABEL				G
agn1	LABEL				G
call	rX				E
call	LABEL				G
ld	rX, rY[, n]	▲			E
ldu	rX, rY, n	❃			R/E
lddu	rX, rY	❃			R/E
ldx	rX, rY	▲			E
(Sheet 5 of 6)					

Table 8.1 Instruction Functional Unit Usage and Execution Stage (Cont.)

Instruction		One ALU	Both ALU	MAC	Execution Stage
ldxu	rX, rY	▲			E
st	rX, rY[, n]	▲			W
stu	rX, rY, n	▲			W
stdu	rX, rY	▲			W
stx	rX, rY	▲			W
stxu	rX, rY	▲			W
nop					
(Sheet 6 of 6)					

8.2 Control Register–Instruction Interaction

This section presents the set of baseline instructions supported by architecture-compliant ZSP400 processors. The architecture supports some instructions natively, while others are synthetic or pseudo operations. The assembler replaces synthetic instructions with one or more native instructions. Synthetic instructions enhance code readability and improve programmer productivity.

The ZSP400 instruction set supports the following classes of instructions:

- Move Instructions
- MAC Instructions
- Arithmetic Instructions
- Bitwise Logical Instructions
- Bit Manipulation Instructions
- Branch Instructions
- Memory Reference Instructions
- NOP Instruction
- Synthetic Instructions

Table 8.2 summarizes the notation used to describe the instruction set. For detailed descriptions of each instruction, refer to Section 8.4, "ZSP400 Instruction Set," on page 8-36.

Table 8.2 Notational Conventions

Notation	Description
cX, cY	Any valid control register
rX, rY	Any valid operand register: r0 through r15
rX.e, rY.e	Any valid operand register pair specifier. An even numbered register r0 through r14.
IMM32U	32-bit unsigned immediate value: 0 ≤ IMM32 ≤ 4294967296
IMM16U	16-bit unsigned immediate value: 0 ≤ IMM16U ≤ 65535
IMM8U	8-bit unsigned immediate value: 0 ≤ IMM8U ≤ 255
IMM5U	5-bit unsigned immediate value: 0 ≤ IMM5U ≤ 32
IMM4S	4-bit signed immediate value: –8 ≤ IMM4S ≤ 7
IMM4U	4-bit unsigned immediate value: 0 ≤ IMM4U ≤ 15
LABEL	Label references
{LABEL}	Address of a label
[value]	Optional parameter in the instruction
{r(X + 1) rX}	Pair of consecutive operand registers with rX being an even numbered register. Example: {r(X+1) rX} may be {r1 r0} or {r3 r2}, not {r2 r1}.
g0	Contents of guard[7:0]
g1	Contents of guard[15:8]
.a	MUL Operations write to the register pair {r1 r0}. MAC operations that accumulate to registers {g0 r1 r0}.
.b	MUL Operations write to the register pair {r3 r2}. MAC operations that accumulate to registers {g1 r3 r2}.
.e	Extended precision (32 bit) ALU operation
rX[n]	Bit n of register rX
rX[m:n]	A set of bits (bit m to bit n inclusive) of register rX
(Sheet 1 of 2)	

Table 8.2 Notational Conventions (Cont.)

Notation	Description
hwf	The hardware flag (hwflag) control register.
mem[rX]	Contents of memory location addressed by the contents of rX.
mem[X]	Contents of memory location with address X.
x	Don’t care condition
✓	The corresponding hwflag register bit is modified based on the result of the instruction.
●	The corresponding hwflag register bit is cleared.
▲	The corresponding mode bit has an effect on the instruction.
rX += rY	The contents of rX and rY are added and the result is stored in rX.
rX –= rY	The contents of rY are subtracted from rX and the result is stored in rX.
rX &= rY	The logical AND of the contents of rX and rY is performed and the result is stored in rX.
rX l= rY	The logical OR of the contents of rX and rY is performed and the result is stored in rX.
rX ^= rY	The logical exclusive OR (XOR) of the contents of rX and rY is performed and the result is stored in rX.
rX =~ rY	The logical complement of the contents of rY are stored in rX.
(Sheet 2 of 2)	

8.2.1 Move Instructions

Table 8.3 shows the ZSP400 move instructions.

Table 8.3 Move Instructions

fmode Register Bits				Instruction			hwflag Register Bits					
sat	q15	sre	mre	Name	Syntax	Description	v, sv	gv, gsv	c	ge	gt	z
				MOV	mov rX, rY	rX = rY						
				MOV	mov cX, rY	cX = rY						
				MOV	mov rX, cY	rX = cY						
				MOV	mov rX, IMM4S	rX = IMM4S						
				MOVL	movl rX, IMM8U	rX[7:0] = IMM8U						
				MOVH	movh rX, IMM8U	rX[15:8] = IMM8U						
				MOVL	movl cX, IMM8U	cX[7:0] = IMM8U, cX = {%fmode, %loop0, %loop1, %loop2, %loop3, %guard}						
				MOVH	movh cX, IMM8U	cX[15:8] = IMM8U, cX = {%fmode, %loop0, %loop1, %loop2, %loop3, %guard}						

8.2.2 MAC Instructions

Table 8.4 shows the ZSP400 MAC instructions.

Table 8.4 MAC Instructions

fmode Register Bits				Instruction			hwflag Register Bits					
sat	**q15**	**rez**	**mre**	**Name**	**Syntax**	**Description**	**v, sv**	**gv, gsv**	**c**	**ge**	**gt**	**z**
▲	▲	▲	▲	MAC.A	mac.a rX, rY	{g0 r1 r0} += rX * rY	✓	✓	✓	✓		
▲	▲	▲	▲	MAC.B	mac.b rX, rY	{g1 r3 r2} += rX * rY	✓	✓	✓	✓		
▲	▲	▲	▲	MACN.A	macn.a rX, rY	{g0 r1 r0} –= rX * rY	✓	✓	✓	✓		
▲	▲	▲	▲	MACN.B	macn.b rX, rY	{g1 r3 r2} –= rX * rY	✓	✓	✓	✓		
▲	▲	▲	▲	MUL.A	mul.a rX, rY	{r1 r0} = rX * rY	✓		●	✓		
▲	▲	▲	▲	MUL.B	mul.b rX, rY	{r3 r2} = rX * rY	✓		●	✓		
▲	▲	▲	▲	MULN.A	muln.a rX, rY	{r1 r0} = –rX * rY	✓		●	✓		
▲	▲	▲	▲	MULN.B	muln.b rX, rY	{r3 r2} = –rX * rY	✓		●	✓		
▲	▲	▲	▲	MAC2.A	mac2.a rX.e, rY.e	{g0 r1 r0} += rX * r(Y) + r(X + 1) * r(Y + 1)	✓	✓	●	✓		
▲	▲	▲	▲	MAC2.B	mac2.b rX.e, rY.e	{g1 r3 r2} += rX * r(Y) + r(X + 1) * r(Y + 1)	✓	✓	●	✓		
▲	▲	▲	▲	CMACR.A	cmacr.a rX.e, rY.e	{g0 r1 r0} += r(X + 1) * r(Y + 1) – rX * rY	✓	✓	●	✓		
▲	▲	▲	▲	CMACR.B	cmacr.b rX.e, rY.e	{g1 r3 r2} += r(X + 1) * r(Y + 1) – rX * rY	✓	✓	●	✓		
(Sheet 1 of 2)												

Table 8.4 MAC Instructions (Cont.)

fmode Register Bits				Instruction			hwflag Register Bits					
sat	**q15**	**rez**	**mre**	**Name**	**Syntax**	**Description**	**v, sv**	**gv, gsv**	**c**	**ge**	**gt**	**z**
▲	▲	▲	▲	CMACI.A	cmaci.a rX.e, rY.e	{g0 r1 r0} += rX * r(Y + 1) + r(X + 1) * rY	✓	✓	●	✓		
▲	▲	▲	▲	CMACI.B	cmaci.b rX.e, rY.e	{g1 r3 r2} += rX * r(Y + 1) + r(X + 1) * rY	✓	✓	●	✓		
▲	▲	▲	▲	CMULR.A	cmulr.a rX.e, rY.e	{r1 r0} = r(X + 1) * r(Y + 1) – rX * rY	✓		●	✓		
▲	▲	▲	▲	CMULR.B	cmulr.b rX.e, rY.e	{r3 r2} = r(X + 1) * r(Y + 1) – rX * rY	✓		●	✓		
▲	▲	▲	▲	CMULI.A	cmuli.a rX.e, rY.e	{r1 r0} = rX * r(Y + 1) + r(X + 1) * rY	✓		●	✓		
▲	▲	▲	▲	CMULI.B	cmuli.b rX.e, rY.e	{r3 r2} = rX * r(Y + 1) + r(X + 1) * rY	✓		●	✓		
▲	▲		▲	DMAC.A	dmac.a rX.e, rY.e	{g0 r1 r0} += {r(X + 1) rX} * {r(Y + 1) rY}	✓	✓	✓	✓		
▲	▲		▲	DMAC.B	dmac.b rX.e, rY.e	{g1 r3 r2} += {r(X + 1) rX} * {r(Y + 1) rY}	✓	✓	✓	✓		
▲	▲		▲	DMUL.A	dmul.a rX.e, rY.e	{r1 r0} = {r(X + 1) rX} * {r(Y + 1) rY}	✓		●	✓		
▲	▲		▲	DMUL.B	dmul.b rX.e, rY.e	{r3 r2} = {r(X + 1) rX} * {r(Y + 1) rY}	✓		●	✓		
				IMUL.A	imul.a rX, rY	{r1 r0} = rX * rY	✓		●	✓		
				IMUL.B	imul.b rX, rY	{r3 r2} = rX * rY	✓		●	✓		
(Sheet 2 of 2)												

8.2.3 Arithmetic Instructions

Table Table 8.5 shows the ZSP400 Arithmetic instructions.

Table 8.5 Arithmetic Instructions

fmode Register Bits			Instruction			hwflag Register Bits					
sat	**sre**	**rez**	**Name**	**Syntax**	**Description**	**v, sv**	**gv, gsv**	**c**	**ge**	**gt**	**z**
			IMUL.A	imul.a rX, rY	{r1 r0} = rX * rY	✓		●	✓		
			IMUL.B	imul.b rX, rY	{r3 r2} = rX * rY	✓		●	✓		
			PADD.A	padd.a rX.e, rY.e	r0 = rX + rY; r1 = r(X + 1) + r(Y + 1)						
			PADD.B	padd.b rX.e, rY.e	r2 = rX + rY; r3 = r(X + 1) + r(Y + 1)						
			PSUB.A	psub.a rX.e, rY.e	r0 = rX – rY; r1 = r(X + 1) – r(Y + 1)						
			PSUB.B	psub.b rX.e, rY.e	r2 = rX – rY; r3 = r(X + 1) – r(Y + 1)						
			NORM	norm rX, rY	If rY == 0 then rX = 0 else if rY == –1 then rX = 15 else if rY >= 0 then rX = 14 - bit position of leading 1 in rY else rX = 14 - bit position of leading 0 in rY	✓		✓	✓	✓	✓
			NORM.E	norm.e rX.e, rY.e	If rY.e == 0 then rX = 0 else if rY.e == -1 then rX = 31 else if rY.e >= 0 then rX = 30 - bit position of leading 1 in rY.e else rX = 30 - bit position of leading 0 in rY.e	✓		✓	✓	✓	✓
▲			ADD	add rX, rY	rX += rY	✓		✓	✓	✓	✓
▲			ADD.E	add.e rX.e, rY.e	{r(X + 1) rX} += {r(Y + 1) rY}	✓		✓	✓	✓	✓

(Sheet 1 of 4)

Table 8.5 Arithmetic Instructions (Cont.)

fmode Register Bits			Instruction			hwflag Register Bits					
sat	sre	rez	Name	Syntax	Description	v, sv	gv, gsv	c	ge	gt	z
▲			ADD	add rX, IMM4S	rX = rX + IMM4S	✓		✓	✓	✓	✓
▲			ADDC.E	addc.e rX.e, rY.e	{r(X + 1) rX} += {r(Y + 1) rY} + carry	✓		✓	✓	✓	✓
▲			SUB	sub rX, rY	rX –= rY	✓		✓	✓	✓	✓
▲			SUB.E	sub.e rX.e, rY.e	{r(X + 1) rX} –= {r(Y + 1) rY}	✓		✓	✓	✓	✓
▲			SUBC.E	subc.e rX.e, rY.e	{r(X + 1) rX} –= {r(Y + 1) rY} - logical inverse of carry	✓		✓	✓	✓	✓
			NEG	neg rX, rY	rX = –rY	✓		✓	✓	✓	✓
			NEG.E	neg.e rX.e, rY.e	{r(X + 1) rX} = –{r(Y + 1) rY}	✓		✓	✓	✓	✓
			CMP	cmp rX, rY	If rX rY: hwf<ge> = 1 If rX > rY: hwf<gt> = 1 other flags set by the result of (rX – rY)	✓		✓	✓	✓	✓
			CMP.E	cmp.e rX.e, rY.e	If {r(X + 1) rX} ≥ {r(Y + 1) rY}: hwf<ge> = 1 If {r(X + 1) rX} > {r(Y + 1) rY}: hwf<gt> = 1 other flags set by the result of: ({r(X + 1) rX} – {r(Y + 1) rY})	✓		✓	✓	✓	✓
			CMP	cmp rX, IMM4S	If rX ≥ IMM4S: hwf<ge> = 1; If rX > IMM4S: hwf<gt> = 1, other flags set by the result of (rX – IMM4S)	✓		✓	✓	✓	✓

(Sheet 2 of 4)

Table 8.5 Arithmetic Instructions (Cont.)

fmode Register Bits			Instruction			hwflag Register Bits					
sat	sre	rez	Name	Syntax	Description	v, sv	gv, gsv	c	ge	gt	z
			CMP.E	cmp.e rX.e, IMM4S	If {r(X + 1) rX} ≥ IMM4S: hwf<ge> = 1 If {r(X + 1) rX} > IMM4S: hwf<gt> = 1 other flags set by the result of: ({r(X + 1) rX} – sign extended (IMM4S))	✓		✓	✓	✓	✓
			ABS	abs rX, rY	rX = |rY|	✓		✓	✓	✓	✓
			ABS.E	abs.e rX.e, rY.e	{r(X + 1) rX} = |{r(Y + 1) rY}|	✓		✓	✓	✓	✓
▲			SHLA	shla rX, rY	rX = rX << rY[3:0]	✓		✓	✓	✓	✓
▲			SHLA.E	shla.e rX.e, rY.e	{r(X + 1) rX} = {r(X + 1) rX} << rY[4:0]	✓		✓	✓	✓	✓
▲			SHLA	shla rX, IMM4U	rX = rX << IMM4U	✓		✓	✓	✓	✓
▲			SHLA.E	shla.e rX.e, IMM4U	{r(X + 1) rX} = {r(X + 1) rX} << IMM4U	✓		✓	✓	✓	✓
	▲		SHRA	shra rX, rY	rX = rX >> rY[3:0]	✓		✓	✓	✓	✓
	▲		SHRA.E	shra.e rX.e, rY.e	{r(X + 1) rX} = {r(X + 1) rX} >> rY[4:0]	✓		✓	✓	✓	✓
	▲		SHRA	shra rX, IMM5U	rX = rX >> IMM5U	✓		✓	✓	✓	✓
	▲		SHRA.E	shra.e rX.e, IMM5U	{r(X + 1) rX} = {r(X + 1) rX} >> IMM5U	✓		✓	✓	✓	✓
			MIN	min rX, rY	rX = min (rX, rY) if rX ≤ rY hwflag<c> = 1; other flags are set on the result of (rX – rY)	✓		✓	✓	✓	✓

(Sheet 3 of 4)

Table 8.5 Arithmetic Instructions (Cont.)

fmode Register Bits			Instruction			hwflag Register Bits					
sat	**sre**	**rez**	**Name**	**Syntax**	**Description**	**v, sv**	**gv, gsv**	**c**	**ge**	**gt**	**z**
			MIN.E	min.e rX.e, rY.e	{r(X + 1) rX} = min ({r(X + 1) rX, {r(Y + 1) rY}) if {r(X +1)rX} ≤ {r(Y + 1)rY} hwf<c> = 1; other flags are set on the result of {r(X + 1)rX} – {r(Y + 1)rY}	✓		✓	✓	✓	✓
			MAX	max rX, rY	rX = max (rX, rY) if rX rY, hwf<c> = 1; other flags are set by the result of (rX – rY)	✓		✓	✓	✓	✓
			MAX.E	max.e rX.e, rY.e	{r(X + 1) rX} = max ({r(X + 1) rX, {r(Y + 1) rY}) if {r(X + 1) rX} {r(Y + 1)rY} hwf<c> = 1; other flags are set on the result of {r(X + 1)rX} – {r(Y + 1)rY}	✓		✓	✓	✓	✓
		▲	ROUND.E	round.e rX.e, rY.e	{r(X + 1) rX} = {r(Y + 1) rY} + 0x0000 8000	✓		✓	✓	✓	✓

			VIT_A	vit_a rX.e, rY.e	r0 = min {(rX + rY), (r(X + 1) + r(Y + 1))}
					if ((rX + rY) < (r(X + 1) + r(Y + 1)))
					vitr = vitr << 1 | 0x0001
					else
					vitr = vitr << 1
						✓		✓	✓		

			VIT_B	vit_b rX.e, rY.e	r1 = min {(rX + r(Y + 1)), (r(X + 1) + rY)}
					if ((rX + r(Y + 1)) < (r(X + 1) + rY))
					vitr = vitr << 1 | 0x0001
					else
					vitr = vitr << 1
						✓		✓	✓		

(Sheet 4 of 4)											

8.2.4 Bitwise Logical Instructions

Table 8.6 shows the ZSP400 bitwise logical instructions.

Table 8.6 Bitwise Logical Instructions

fmode Register Bits				Instruction			hwflag Register Bits					
sat	q15	sre	mre	Name	Syntax	Description	v, sv	gv, gsv	c	ge	gt	z
				AND	and rX, rY	rX &= rY	✓		✓	✓	✓	✓
				AND.E	and.e rX.e, rY.e	{r(X + 1)rX} &= {r(Y + 1)rY}	✓		✓	✓	✓	✓
				OR	or rX, rY	rX |= rY	✓		✓	✓	✓	✓
				OR.E	or.e rX.e, rY.e	{r(X + 1)rX} |= {r(Y + 1)rY}	✓		✓	✓	✓	✓
				XOR	xor rX, rY	rX ^= rY	✓		✓	✓	✓	✓
				XOR.E	xor.e rX.e, rY.e	{r(X + 1)rX} ^= {r(Y + 1)rY}	✓		✓	✓	✓	✓
				NOT	not rX, rY	rX =~ rY	✓		✓	✓	✓	✓
				NOT.E	not.e rX.e, rY.e	{r(X + 1)rX} =~ {r(Y + 1)rY}	✓		✓	✓	✓	✓

8.2.5 Bit Manipulation Instructions

Table 8.7 shows the ZSP400 bit manipulation instructions.

Table 8.7 Bit Manipulation Instructions

fmode Register Bits				Instruction			hwflag Register Bits					
sat	q15	sre	mre	Name	Syntax	Description	v, sv	gv, gsv	c	ge	gt	z
				BITC	bitc rX, IMM4U	rX &= ~(1 << IMM4U)						✓
				BITC	bitc cX, IMM4U	cX &= ~(1 << IMM4U), cX = {%fmode, %tc, %imask, %ip0, %ip1, %guard, %hwflag, %ireq, %vitr, %smode, %amode}						
				BITS	bits rX, IMM4U	rX |= (1 << IMM4U)						✓
				BITS	bits cX, IMM4U	cX |= (1 << IMM4U), cX ={%fmode, %tc, %imask, %ip0, %ip1, %guard, %hwflag, %ireq, %vitr, %smode, %amode}						
				BITI	biti rX, IMM4U	rX ^= (1 << IMM4U)						✓
				BITI	biti cX, IMM4U	cX ^= (1 << IMM4U), cX ={%fmode, %tc, %imask, %ip0, %ip1, %guard, %hwflag, %ireq, %vitr, %smode, %amode}						
				BITT	bitt rX, IMM4U	Update hwf<z> depending on whether rX[IMM4U] is zero or one.						✓
				BITT	bitt cX, IMM4U	Update hwf<z> depending on whether cX[IMM4U] is zero or one. cX = {%fmode, %tc, %imask, %ip0, %ip1, %guard, %hwflag, %ireq, %vitr, %smode, %amode}						✓

(Sheet 1 of 2)

Table 8.7 Bit Manipulation Instructions (Cont.)

fmode Register Bits				Instruction			hwflag Register Bits					
sat	q15	sre	mre	Name	Syntax	Description	v, sv	gv, gsv	c	ge	gt	z
				REVB	revb rX, IMM4U	Reverses order of rX[IMM4U:0]. If IMM4U <15, rX[15:IMM4U] = 0	✓		✓	✓	✓	✓
				SHLL	shll rX, rY	rX = rX << rY[3:0]	✓		✓	✓	✓	✓
				SHLL.E	shll.e rX.e, rY.e	{r(X + 1) rX} = {r(X + 1) rX} << rY[4:0]	✓		✓	✓	✓	✓
				SHLL	shll rX, IMM5U	rX = rX << IMM5U	✓		✓	✓	✓	✓
				SHLL.E	shll.e rX.e, IMM5U	{r(X + 1) rX} = {r(X + 1) rX} << IMM5U	✓		✓	✓	✓	✓
				SHRL	shrl rX, rY	rX = rX >> rY[3:0]	✓		✓	✓	✓	✓
				SHRL.E	shrl.e rX.e, rY.e	{r(X + 1) rX} = {r(X + 1) rX} >>rY; [4:0]	✓		✓	✓	✓	✓
				SHRL	shrl rX, IMM5U	rX = rX >> IMM5U	✓		✓	✓	✓	✓
				SHRL.E	shrl.e rX.e, IMM5U	{r(X + 1) rX} = {r(X + 1) rX} >> IMM5U	✓		✓	✓	✓	✓
(Sheet 2 of 2)												

8.2.6 Branch Instructions

Unconditional branch instructions can span a 12-bit displacement, corresponding to a range of –2048 to +2047 words. An out-of-range error is emitted by the linker (SDLD) if this is violated. Conditional branch instructions can span an 8-bit displacement (–128 to +127 words). The agn0, agn1, agn2, agn3 instructions span an 8-bit negative displacement (–256 to –1).

Note: For a branch operation, an immediate value can be used in the place of a LABEL.

Table 8.8 shows the ZSP400 branch instructions.

Table 8.8 Branch Instructions

fmode Register Bits				Instruction			hwflag Register Bits					
sat	**q15**	**sre**	**mre**	**Name**	**Syntax**	**Description**	**v, sv**	**gv, gsv**	**c**	**ge**	**gt**	**z**
				BR	br LABEL	pc = {LABEL}						
				BZ	bz LABEL	if hwf<z>{ pc = {LABEL} } else { pc += 1; }						
				BNZ	bnz LABEL	if !hwf<z>{ pc = {LABEL} } else { pc += 1; }						
				BLT	blt LABEL	if !hwf<ge> { pc = {LABEL} } else { pc += 1; }						
				BLE	ble LABEL	if !hwf<gt>{ pc = {LABEL} } else { pc += 1; }						
				BGT	bgt LABEL	if hwf<gt>{ pc = {LABEL} } else { pc += 1; }						
				BGE	bge LABEL	if hwf<ge>{ pc = {LABEL} } else { pc += 1; }						

(Sheet 1 of 3)

Table 8.8 Branch Instructions (Cont.)

fmode Register Bits				Instruction			hwflag Register Bits					
sat	q15	sre	mre	Name	Syntax	Description	v, sv	gv, gsv	c	ge	gt	z
				BOV	bov LABEL	if hwf<v>{ pc = {LABEL} } else { pc += 1; }						
				BNOV	bnov LABEL	if !hwf<v>{ pc = {LABEL} } else { pc += 1; }						
				BC	bc LABEL	if hwf<c>{ pc = {LABEL} } else { pc += 1; }						
				BNC	bnc LABEL	if !hwf<c>{ pc = {LABEL} } else { pc += 1; }						
				AGN0	agn0 LABEL	if (! loop0) { pc += 1; loop0 –= 1; } else { pc = {LABEL} ; loop0 –= 1 }						
				AGN1	agn1 LABEL	if (! loop1) { pc += 1; loop1 –= 1; } else { pc = {LABEL}; loop1 –= 1 }						

(Sheet 2 of 3)

Table 8.8 Branch Instructions (Cont.)

fmode Register Bits				Instruction			hwflag Register Bits					
sat	q15	sre	mre	Name	Syntax	Description	v, sv	gv, gsv	c	ge	gt	z
				AGN2	agn2 LABEL	if (! loop2) { pc += 1; loop2 –= 1; } else { pc = {LABEL}; loop2 –= 1 }						
				AGN3	agn3 LABEL	if (! loop3) { pc += 1; loop3 –= 1; } else { pc = {LABEL}; loop3 –= 1 }						
				CALL	call rX	rpc = pc + 1; pc = rX;						
				CALL	call LABEL	pc - {LABEL} must be representable as a 13-bit signed even number. rpc = pc +1 pc = {LABEL}						
				RET	ret	pc = rpc						
				RETI	reti	pc = tpc; imask<gie> = imask<pgie>; ip0<epl> = ip0<pepl>						
(Sheet 3 of 3)												

8.2.7 Memory Reference Instructions

Table 8.9 shows the ZSP400 memory reference instructions.

Table 8.9 Memory Reference Instructions

smode Register Bits						Instruction			hwflag Register Bits					
lis	**sis**	**cb0**	**cb1**	**dir**	**ddr**	**Name**	**Syntax**	**Description**	**v, sv**	**gv, gsv**	**c**	**ge**	**gt**	**z**
▲				▲	▲	LD	ld rX, rY [, n]	–4 ≤ n ≤ 3 rX ← mem[rY + n]						
▲		▲	▲	▲	▲	LDU	ldu rX, rY, n	n = {1, 2} rX ← mem[rY] rY = rY + n						
▲				▲	▲	LDU	ldu rX, rY, n	n = {–2, –1} rX ← mem[rY] rY = rY + n						
▲		▲	▲	▲	▲	LDDU	lddu rX, rY.e, 2	if n = 2 {rX ← mem[rY] r(X + 1) ← mem[rY + 1]} else {rX ← mem[rY – 1] r(X + 1) ← mem[rY]} rY = rY + n						
▲				▲	▲	LDDU	lddu rX, rY.e, –2	if n = 2 {rX ← mem[rY] r(X + 1) ← mem[rY + 1]} else {rX ← mem[rY – 1] r(X + 1) ← mem[rY]} rY = rY + n						
▲				▲	▲	LDX	ldx rX, rY.e	rX ← mem[rY + r(Y + 1)]						
▲				▲	▲	LDXU	ldxu rX, rY.e	rX ← mem[rY + r(Y + 1)]; rY += r(Y + 1)						
	▲			▲	▲	ST	st rX, rY [, n]	–4 ≤ n ≤ 3 mem[rY + n] = rX						

(Sheet 1 of 2)

Table 8.9 Memory Reference Instructions (Cont.)

smode Register Bits						Instruction			hwflag Register Bits					
lis	**sis**	**cb0**	**cb1**	**dir**	**ddr**	**Name**	**Syntax**	**Description**	**v, sv**	**gv, gsv**	**c**	**ge**	**gt**	**z**
	▲	▲	▲	▲	▲	STU	stu rX, rY, n	n = {1, 2} mem[rY] ← rX rY = rY + n						
	▲			▲	▲	STU	stu rX, rY, n	n = {–2, –1} mem[rY] ← rX rY = rY + n						
	▲	▲	▲	▲	▲	STDU	stdu rX.e, rY, 2	mem[rY] ← rX mem[rY + 1] ← r(X + 1) rY = rY + 2						
	▲			▲	▲	STDU	stdu rX.e, rY, –2	mem[rY] ← r(X + 1) mem[rY – 1] ← rX rY = rY – 2						
	▲			▲	▲	STX	stx rX, rY.e	mem[rY + r(Y + 1)] = rX						
	▲			▲	▲	STXU	stxu rX, rY.e	mem[rY + r(Y + 1)] = rX rY = rY + r(Y + 1)						
(Sheet 2 of 2)														

8.2.8 NOP Instruction

Table 8.10 shows the ZSP400 NOP (no operation) instructions.

Table 8.10 NOP Instruction

fmode Register Bits				Instruction			hwflag Register Bits					
sat	**q15**	**sre**	**mre**	**Name**	**Syntax**	**Description**	**v, sv**	**gv, gsv**	**c**	**ge**	**gt**	**z**
				NOP	nop	No operation						

8.2.9 Synthetic Instructions

Table 8.11 shows the ZSP400 synthetic instructions. The description column also describes the assembler replacements for each synthetic instruction.

Table 8.11 Synthetic Instructions

fmode Register Bits				**Instruction**			**hwflag Register Bits**					
sat	**q15**	**sre**	**mre**	**Name**	**Syntax**	**Description**	**v, sv**	**gv, gsv**	**c**	**ge**	**gt**	**z**
				LDA	lda rX, LABEL	Replaced by: movl rX, {LABEL}[7:0] and movh rX, {LABEL}[15:8]						
				MOV	mov rX, IMM	If –8 ≤ IMM ≤ 7, then replaced by: mov rX, IMM4S else replaced by: movl rX, IMM[7:0] and movh rX, IMM[15:8]						
				MOV	mov cX, IMM16U	cX = {%fmode, %loop0, %loop1, %loop2, %loop3, %guard} replaced by: movl cX, IMM16U[7:0] and movh cX, IMM16U[15:8]						
				MOVLH	movlh rX, IMM32U	Replaced by: movl rX, IMM32U[23:16] movl r(X – 1), IMM32U[7:0] movh rX, IMM32U[31:24] movh r(X – 1), IMM32U[15:0]						
				BR	br rX	Replaced by “mov *pc, rX*”						
				HALT	halt	Replaced by bits smode, 15						
				SLEEP	sleep	Replaced by bits smode, 14						
				IDLE	idle	Replaced by bits smode, 13						

8.3 Instruction Coding

This section describes the instruction set coding for the ZSP400 architecture. The ZSP400 machine code is an example of the orthogonal nature of the instruction set architecture. Fetching from the Instruction Cache and preliminary decoding are accomplished in a single F/D pipeline stage.

All processors conforming to the ZSP400 architecture must be able to execute the machine code listed in this document.

8.3.1 Instruction Opcode

Table 8.12 summarizes the instruction set Opcodes.

Table 8.12 Instruction Set Opcode Summary

Instruction	15–8	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
branch IMM	0x	0	0	0	0	immediate											
call IMM	1x	0	0	0	1	immediate											
movl rX, IMM	2x	0	0	1	0	rX				immediate							
movh rX, IMM	3x	0	0	1	1	rX				immediate							
bc IMM	4x	0	1	0	0	**condition**				immediate							
mac inst	5x	0	1	0	1	**op0**				rX				rY			
store inst	6x	0	1	1	0	**op1**				rX				rY			
load inst	7x	0	1	1	1	**op1**				rX				rY			
short alu inst	8x	1	0	0	0	**op2**				rX				rY			
extended alu inst	9x	1	0	0	1	**op3**				rX.e			0	rY.e			0
reserved	9x	1	0	0	1	x	x	x	x	x	x	x	1	x	x	x	x
reserved	9x	1	0	0	1	x	x	x	x	x	x	x	x	x	x	x	1
short IMM inst	Ax	1	0	1	0	**op4**				rX				immediate			
extended alu inst	Bx	1	0	1	1	0	**op5**			rX.e			0	immediate / rY			
reserved	Bx	1	0	1	1	0	x	x	x	x	x	x	1	x	x	x	x
mov rX, cY	Bx	1	0	1	1	1	0	0	cY					rX			
mov cX, rY	Bx	1	0	1	1	1	0	1	cX					rY			
mov rX, rY	Bx	1	0	1	1	1	1	0	0	rX				rY			
reserved	Bd	1	0	1	1	1	1	0	1	x	x	x	x	x	x	x	x

Table 8.12 Instruction Set Opcode Summary (Cont.)

Instruction	**15–8**	**b15**	**b14**	**b13**	**b12**	**b11**	**b10**	**b9**	**b8**	**b7**	**b6**	**b5**	**b4**	**b3**	**b2**	**b1**	**b0**
reserved	Be	1	0	1	1	1	1	1	0	x	x	x	x	x	x	x	x
reserved	Bf	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
ret	Bf	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	1
reti	Bf	1	0	1	1	1	1	1	1	0	0	0	0	0	0	1	0
reserved	Bf	1	0	1	1	1	1	1	1	0	0	0	0	0	0	1	1
reserved	Bf	1	0	1	1	1	1	1	1	0	0	0	0	0	1	x	x
reserved	Bf	1	0	1	1	1	1	1	1	0	0	0	0	1	x	x	x
reserved	Bf	1	0	1	1	1	1	1	1	0	0	0	1	x	x	x	x
reserved	Bf	1	0	1	1	1	1	1	1	0	0	1	x	x	x	x	x
reserved	Bf	1	0	1	1	1	1	1	1	0	1	x	x	x	x	x	x
reserved	Bf	1	0	1	1	1	1	1	1	1	x	x	x	x	x	x	x
misc inst	Cx	1	1	0	0	**op6**				rX				immediate			
movl cX	Dx	1	1	0	1	0	cX			immediate							
movh cX	Dx	1	1	0	1	1	cX			immediate							
mac inst	Ex	1	1	1	0	**op7**				rX				rY			
reserved	Fx	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x

Table 8.13 lists the Condition field for the bc IMM instructions.

Table 8.13 Condition Field

Instruction	b11	b10	b9
zero	0	0	0
not zero	0	0	0
greater than or equal to zero	0	0	1
less than zero	0	0	1
greater than zero	0	1	0
less than or equal to zero	0	1	0
overflow	0	1	1
not overflow	0	1	1
carry	1	0	0
not carry	1	0	0
reserved	1	0	1
reserved	1	0	1
loop2 is zero	1	1	0
loop3 is zero	1	1	0
loop0 is zero	1	1	1
loop1 is zero	1	1	1

Table 8.14Table lists the op0 field for the MAC instructions.

Table 8.14 op0 Field

Instruction	b11	b10	b9	b8
mac.a	0	0	0	0
macn.a	0	0	0	1
mul.a	0	0	1	0
muln.a	0	0	1	1
mac2.a	0	1	0	0
cmacr.a	0	1	0	1
dmac.a	0	1	1	0
cmaci.a	0	1	1	1
mac.b	1	0	0	0
macn.b	1	0	0	1
mul.b	1	0	1	0
muln.b	1	0	1	1
mac2.b	1	1	0	0
cmacr.b	1	1	0	1
dmac.b	1	1	1	0
cmaci.b	1	1	1	1

Table Table 8.15 lists the op1 field for the Load and Store instructions.

Table 8.15 op1 Field

Instruction	b11	b10	b9	b8
ld/st	0	signed offset		
lddu/stdu rX, rY, 2	1	0	0	0
ldu/stu rX, rY, 1	1	0	0	1
ldu/stu rX, rY, 2	1	0	1	0
lddu/stdu rX, rY, –2	1	0	1	1
ldx/stx	1	1	0	0
ldxu/stxu	1	1	0	1
ldu/stu rX, rY, –2	1	1	1	0
ldu/stu rX, rY, –1	1	1	1	1

Table Table 8.16 lists the op2 field for the Short ALU instructions.

Table 8.16 op2 Field

Instruction	b11	b10	b9	b8
add	0	0	0	0
cmp	0	0	0	1
shll	0	0	1	0
shrl	0	0	1	1
shla	0	1	0	0
shra	0	1	0	1
sub	0	1	1	0
norm	0	1	1	1
and	1	0	0	0
or	1	0	0	1
xor	1	0	1	0
not	1	0	1	1
abs	1	1	0	0
min	1	1	0	1
max	1	1	1	0
neg	1	1	1	1

Table Table 8.17 lists the op3 field for the Extended ALU instructions.

Table 8.17 op3 Field

Instruction	b11	b10	b9	b8
add.e	0	0	0	0
cmp.e	0	0	0	1
shll.e	0	0	1	0
shrl.e	0	0	1	1
shla.e	0	1	0	0
shra.e	0	1	0	1
sub.e	0	1	1	0
norm.e	0	1	1	1
and.e	1	0	0	0
or.e	1	0	0	1
xor.e	1	0	1	0
not.e	1	0	1	1
abs.e	1	1	0	0
min.e	1	1	0	1
max.e	1	1	1	0
neg.e	1	1	1	1

Table 8.18Table lists the op4 field for the Short IMM instructions.

Table 8.18 op4 Field

Instruction	b11	b10	b9	b8
addsi	0	0	0	0
cmp	0	0	0	1
shll	0	0	1	0
shrl	0	0	1	1
shla	0	1	0	0
shra	0	1	0	1
mov	0	1	1	0
call rX	0	1	1	1
bitc rX	1	0	0	0
bits rX	1	0	0	1
biti rX	1	0	1	0
bitt rX	1	0	1	1
bitc cX	1	1	0	0
bits cX	1	1	0	1
biti cX	1	1	1	0
bitt cX	1	1	1	1

Table Table 8.19 lists the op5 field for the Extended ALU instructions.

Table 8.19 op5 Field

Instruction	b10	b9	b8
round.e rX, rY	0	0	0
cmp.e rX, IMM	0	0	1
shll.e rX, IMM	0	1	0
shrl.e rX, IMM	0	1	1
shla.e rX, IMM	1	0	0
shra.e rX, IMM	1	0	1
addc.e rX, rY	1	1	0
subc.e rX, rY	1	1	1

Table 8.20 lists the op6 field for the Miscellaneous instructions.

Table 8.20 op6 Field

Instruction	b11	b10	b9	b8
revb	0	0	0	0
reserved	0	0	0	1
reserved	0	0	1	X
reserved	0	1	X	X
reserved	1	0	X	X
reserved	1	1	0	X
reserved	1	1	1	0
nop	1	1	1	1

Table 8.21 lists the op7 field for the MAC instructions.

Table 8.21 op7 Field

Instruction	b11	b10	b9	b8
vit_a	0	0	0	0
vit_b	0	0	0	1
padd.a	0	0	1	0
psub.a	0	0	1	1
imul.a	0	1	0	0
cmulr.a	0	1	0	1
dmul.a	0	1	1	0
cmuli.a	0	1	1	1
reserved	1	0	0	0
reserved	1	0	0	1
padd.b	1	0	1	0
psub.b	1	0	1	1
imul.b	1	1	0	0
cmulr.b	1	1	0	1
dmul.b	1	1	1	0
cmuli.b	1	1	1	1

8.4 ZSP400 Instruction Set

The remainder of this chapter describes each ZSP400 instruction in detail. Each instruction description includes:

- Instruction Syntax
- Description
- Examples

All ZSP400 instructions are single-word (16-bit) in length and execute in a single cycle.

ABS Absolute Value

Assembly Syntax abs rX, rY

Description rX = |rY|

The absolute value of the contents of register rY is computed and placed in register rX. In the corner case where the contents of rY = 0x8000 the absolute value is calculated to be 0x7FFF.

Example abs r9, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r4	0x8299
							x	x		x	x	x	r9	0x0421
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r4	0x8299
		0	x	x	x	0	1	1	0	x	x	x	r9	0x7d67

ABS.E Absolute Value (Extended Precision)

Assembly Syntax abs.e rX.e, rY.e

Description {r(X + 1) rX} = |{r(Y + 1) rY}|
The absolute value of the contents of register pair {rY + 1 rY} is computed and placed in register pair {rX + 1 rX}. In the corner case where the contents of rY = 0x8000 0000 the absolute value is calculated to be 0x7FFF FFFF.

Example abs.e r6, r0

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved							rez	sat	res	q15	sre	mre	{r7 r6}	0x0421	0x8821
								x	x		x	x	x	{r1 r0}	0x8000	0x0000
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er			
			x	x	x	x	x	x	x	x	x	x	x			

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r7 r6}	0x7fff	0xffff
			1	x	1	x	0	1	1	0	x	x	x	{r1 r0}	0x8000	0x0000

ADD Add Immediate

Assembly Syntax add rX, IMM4S

Description rX = rX + IMM4S

Example add r5, 5

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	r5	0x4512
								x	0		x	x	x		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r5	0x4517
			0	x	x	x	0	1	1	0	x	x	x		

Add Registers

Assembly Syntax add rX, rY

Description rX += rY

Example add r3, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r3	0x8c23
							x	0		x	x	x	r4	0x4f34
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r3	0xdb57
		0	x	x	x	0	0	0	0	x	x	x	r4	0x4f34

ADD.E Add Registers (Extended Precision)

Assembly Syntax add.e rX.e, rY.e

Description {r(X + 1) rX} += {r(Y + 1) rY}

Example add.e r2, r4

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved							rez	sat	res	q15	sre	mre	{r3 r2}	0x8f34	0xc342
								x	1		x	x	x	{r5 r4}	0x8e0a	0x8c23
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er			
			x	x	x	x	x	x	x	x	x	x	x			

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r3 r2}	0x8000	0x0000
			1	x	1	x	1	0	0	0	x	x	x	{r5 r4}	0x8e0a	0x8c23

ADDC.E Add with Carry (Extended Precision)

Assembly Syntax addc.e rX.e, rY.e

Description {r(X + 1) rX} += {r(Y + 1) rY} + carry

Example addc.e r8, r14

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved						rez	sat	res	q15	sre	mre	{r9 r8}	0x0785c	0xcffe
							x	1		x	x	x	{r15 r14}	0x0000	0x0000

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	1	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r9 r8}	0x785c	0xcfff
		0	x	x	x	0	1	1	0	x	x	x	{r15 r14}	0x0000	0x0000

AGN0 Again0

Assembly Syntax agn0 LABEL

Description

```
if (! loop0) {
pc += 1;
loop0 –= 1;
}
else {
pc = {LABEL}
; loop0 –= 1
}
```

Example agn0 jmp

Architectural state before the instruction is executed:

	15 11 10 9 8 7 6	5	4	3	2	1	0
fmode	reserved	rez	sat	res	q15	sre	mre
		x	x		x	x	x

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
loop0	0x0010
pc	0x000a
{jmp}	0x0006

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
loop0	0x000f
pc	0x0006
{jmp}	0x0006

AGN1 Again1

Assembly Syntax agn1 LABEL

Description

```
if (! loop1) {
pc += 1;
loop1 –= 1;
}
else {
pc = {LABEL};
loop1 –= 1
}
```

Example agn1 jmp

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	x		x	x	x
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register	
loop1	0x0000
pc	0x000a
{jmp}	0x0006

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register	
loop1	0xffff
pc	0x000b
{jmp}	0x0006

AGN2 Again2

Assembly Syntax agn2 LABEL

Description

```
if (! loop2) {
pc += 1;
loop2 –= 1;
}
else {
pc = {LABEL};
loop2 –= 1
}
```

Example agn2 jmp

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	x		x	x	x
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register	
loop2	0x0000
pc	0x000a
{jmp}	0x0006

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register	
loop2	0xffff
pc	0x000b
{jmp}	0x0006

AGN3 Again3

Assembly Syntax agn3 LABEL

Description

```
if (! loop3) {
    pc += 1;
    loop3 –= 1;
}
else {
    pc = {LABEL};
    loop3 –= 1
}
```

Example agn3 jmp

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
loop3	0x0000
pc	0x000a
{jmp}	0x0006

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
loop3	0xffff
pc	0x000b
{jmp}	0x0006

AND

Logical AND

Assembly Syntax and rX, rY

Description rX &= rY

Example and r11, r2

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r11	0x8f34
							x	x		x	x	x	r2	0x70cb
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r11	0x0000
		0	x	x	x	0	1	0	1	x	x	x	r2	0x70cb

AND.E Logical AND (Extended Precision)

Assembly Syntax and.e rX.e, rY.e

Description {r(X + 1)rX} &= {r(Y + 1)rY}

Example and.e r0, r2

Architectural state after the instruction is executed:

	15 11 10 9 8 7 6	5	4	3	2	1	0
fmode	reserved	rez	sat	res	q15	sre	mre
		x	x		x	x	x

Register		
{r1 r0}	0x3f34	0xd2a1
{r3 r2}	0x4343	0x7734

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	1	1	0	x	x	x

Register		
{r1 r0}	0x0300	0x5220
{r3 r2}	0x4343	0x7734

BC Branch on Carry

Assembly Syntax bc LABEL

Description

```
if hwf<c>{
pc = {LABEL} }
else {
pc += 1;
}
```

Example bc jmp

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	pc	0x000a
							x	x		x	x	x		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	1	x	x	x	x	x	x	{jmp}	0x0006

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	pc	0x0006
		x	x	x	x	1	x	x	x	x	x	x		
													{jmp}	0x0006

BGE Branch on Greater Than or Equal To

Assembly Syntax bge LABEL

Description

```
if hwf<ge>{
pc = {LABEL} }
else {
pc += 1;
}
```

Example bge jmp

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	0	x	x	x	x	x

Register	
pc	0x000a
{jmp}	0x0006

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	0	x	x	x	x	x

Register	
pc	0x000b
{jmp}	0x0006

BGT Branch on Greater Than

Assembly Syntax bgt LABEL

Description

```
if hwf<gt>{
pc = {LABEL} }
else {
pc += 1;
}
```

Example bgt jmp

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	x		x	x	x
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	1	x	x	x	x

Register	
pc	0x000a
{jmp}	0x0006

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	1	x	x	x	x

Register	
pc	0x0006
{jmp}	0x0006

BITC Bit Clear Control Register

Assembly Syntax bitc cX, IMM4U

Description cX &= ~(1 << IMM4U), cX = {%fmode, %tc, %imask, %ip0, %ip1, %guard, %hwflag, %ireq, %vitr, %smode, %amode}

Example bitc %fmode, 2

Architectural state before the instruction is executed:

15 11 10 9 8 7 6 5 4 3 2 1 0 Register

fmode	reserved	rez	sat	res	q15	sre	mre	fmode	0x0004
		x	x		x	x	x		

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

15 11 10 9 8 7 6 5 4 3 2 1 0 Register

fmode	reserved	sat	res	q15	sre	mre	fmode	0x0000
		x		0	x	x		

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Bit Clear Operand Register

Assembly Syntax bitc rX, IMM4U

Description rX &= ~(1 << IMM4U)

Example bitc r10, 5

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r10	0x3432
							x	x		x	x	x		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r10	0x3412
		x	x	x	x	x	x	x	0	x	x	x		

BITI Bit Invert Control Register

Assembly Syntax biti cX, IMM4U

Description cX ^= (1 << IMM4U), cX ={%fmode, %tc, %imask, %ip0, %ip1, %guard, %hwflag, %ireq, %vitr, %smode, %amode}

Example biti %fmode, 2

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	fmode	0x0000
								x	x		1	x	x		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved								sat	res	q15	sre	mre	fmode	0x0004
									x		0	x	x		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

BITI Bit Invert Operand Register

Assembly Syntax biti rX, IMM4U

Description rX ^= (1 << IMM4U)

Example biti r5, 0

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	r5	0xf812
								x	x		x	x	x		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r5	0xf813
			x	x	x	x	x	x	x	0	x	x	x		

Assembly Syntax bits cX, IMM4U

Description cX |= (1 << IMM4U), cX ={%fmode, %tc, %imask, %ip0, %ip1, %guard, %hwflag, %ireq, %vitr, %smode, %amode}

Example bits %smode, 4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	smode	0x0029
							x	x		x	x	x		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	smode	0x0039
		x	x	x	x	x	x	x	x	x	x	x		

BITS Bit Set Operand Register

Assembly Syntax bits rX, IMM4U

Description rX |= (1 << IMM4U)

Example bits r0, 14

Architectural state before the instruction is executed:

<table>
<tr><td></td><td>15 11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>Register</td><td></td></tr>
<tr><td>fmode</td><td colspan="6">reserved</td><td>rez</td><td>sat</td><td>res</td><td>q15</td><td>sre</td><td>mre</td><td>r0</td><td>0x3412</td></tr>
<tr><td></td><td colspan="6"></td><td>x</td><td>x</td><td></td><td>x</td><td>x</td><td>x</td><td></td><td></td></tr>
<tr><td>hwflag</td><td>reserved</td><td>v</td><td>gv</td><td>sv</td><td>gsv</td><td>c</td><td>ge</td><td>gt</td><td>z</td><td>ir</td><td>ex</td><td>er</td><td></td><td></td></tr>
<tr><td></td><td></td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td></tr>
</table>

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r0	0x7412
		x	x	x	x	x	x	x	0	x	x	x		

BITT Bit Test Control Register

Assembly Syntax bitt cX, IMM4U

Description Update hwf<z> depending on whether cX[IMM4U] is zero or one.
cX = {%fmode, %tc, %imask, %ip0, %ip1, %guard, %hwflag, %ireq, %vitr, %smode, %amode}

Example bitt %fmode, 2

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	fmode	0x0000
								x	x		0	x	x		

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved								sat	res	q15	sre	mre	fmode	0x0000
									x		0	x	x		

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	1	x	x	x

BITT Bit Test Operand Register

Assembly Syntax bitt rX, IMM4U

Description Update hwf<z> depending on whether rX[IMM4U] is zero or one.

Example bitt r5, 0

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	r5	0x0001
								x	x		x	x	x		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r5	0x0001
			x	x	x	x	x	x	x	0	x	x	x		

BLE Branch on Less Than or Equal To

Assembly Syntax ble LABEL

Description

```
if !hwf<gt>{
pc = {LABEL} }
else {
pc += 1;
}
```

Example ble jmp

Architectural state before the instruction is executed:

	15 11 10 9 8 7 6	5	4	3	2	1	0
fmode	reserved	rez	sat	res	q15	sre	mre
		x	x		x	x	x

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	1	x	x	x	x

Register	
pc	0x000a
{jmp}	0x0006

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	1	x	x	x	x

Register	
pc	0x000b
{jmp}	0x0006

BLT Branch on Less Than

Assembly Syntax blt LABEL

Description

```
if !hwf<ge> {
pc = {LABEL} }
else {
pc += 1;
}
```

Example blt jmp

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	pc	0x000a
							x	x		x	x	x		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	0	x	x	x	x	x	{jmp}	0x0006

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	pc	0x0006
		x	x	x	x	x	0	x	x	x	x	x		
													{jmp}	0x0006

BNC Branch on No Carry

Assembly Syntax bnc LABEL

Description

```
if !hwf<c>{
pc = {LABEL} }
else {
pc += 1;
}
```

Example bnc jmp

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	pc	0x000a
							x	x		x	x	x		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	1	x	x	x	x	x	x	{jmp}	0x0006

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	pc	0x000b
		x	x	x	x	1	x	x	x	x	x	x		
													{jmp}	0x0006

BNZ Branch on Not Zero

Assembly Syntax bnz LABEL

Description

```
if !hwf<z>{
pc = {LABEL} }
else {
pc += 1;
}
```

Example bnz jmp

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	1	x	x	x

Register	
pc	0x000a
{jmp}	0x0006

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	1	x	x	x

Register	
pc	0x000b
{jmp}	0x0006

BNOV Branch on No Overflow

Assembly Syntax bnov LABEL

Description

```
if !hwf<v>{
pc = {LABEL} }
else {
pc += 1;
}
```

Example bnov jmp

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	pc	0x000a
								x	x		x	x	x		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			0	x	x	x	x	x	x	x	x	x	x	{jmp}	0x0006

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	pc	0x0006
			0	x	x	x	x	x	x	x	x	x	x		
														{jmp}	0x0006

BOV Branch on Overflow

Assembly Syntax bov LABEL

Description

```
if hwf<v>{
pc = {LABEL} }
else {
pc += 1;
}
```

Example bov jmp

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		1	x	x	x	x	x	x	x	x	x	x

Register	
pc	0x000a
{jmp}	0x0006

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		1	x	x	x	x	x	x	x	x	x	x

Register	
pc	0x0006
{jmp}	0x0006

BR Unconditional Branch

Assembly Syntax br LABEL

Description pc = {LABEL}

Example br jmp

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	pc	0x000a
								x	x		x	x	x		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x	{jmp}	0x0006

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	pc	0x0006
			x	x	x	x	x	x	x	x	x	x	x		
														{jmp}	0x0006

BR Unconditional Branch on Register Value (Synthetic Instruction)

Assembly Syntax	br rX
Description	Replaced by: mov pc, rX

BZ Branch on Zero

Assembly Syntax bz LABEL

Description

```
if hwf<z>{
pc = {LABEL} }
else {
pc += 1;
}
```

Example bz jmp

Architectural state before the instruction is executed:

	15 11 10 9 8 7 6	5	4	3	2	1	0
fmode	reserved	rez	sat	res	q15	sre	mre
		x	x		x	x	x

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	1	x	x	x

Register	
pc	0x000a
{jmp}	0x0006

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	1	x	x	x

Register	
pc	0x0006
{jmp}	0x0006

Assembly Syntax call LABEL

Description pc - {LABEL} must be representable as a 13-bit signed even number.
rpc = pc +1
pc = {LABEL}

Example call jmp

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
rpc	0x0f34
pc	0x500
{LABEL}	0x0088

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
rpc	0x501
pc	0x0088
{LABEL}	0x0088

CALL Call

Assembly Syntax call rX

Description rpc = pc + 1;
pc = rX;

Example call r10

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
r10	0x3013
pc	0x0030
rpc	0x0010

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
r10	0x3013
pc	0x3013
rpc	0x0031

CMACR.A Complex MAC Real to Accumulator A

Assembly Syntax cmacr.a rX.e, rY.e

Description {g0 r1 r0} += r(X + 1) * r(Y + 1) – rX * rY

Example cmacr.a r4, r12

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											1	1		1	0	1
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	0	1	1	1	1	1	1	1

Register		
{r1 r0}	0xfddb	0x8c64
{r5 r4}	0x7777	0x7777
{r13 r12}	0x8120	0x3214

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						1	0	1	x	0	0	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	1	1	1	1	1	1	1	1

Register		
{r1 r0}	8000	0000
{r5 r4}	0x7777	0x7777
{r13 r12}	0x8120	0x3214

Assembly Syntax cmacr.b rX.e, rY.e

Description {g1 r3 r2} += r(X + 1) * r(Y + 1) – rX * rY

Example cmacr.b r8, r10

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											0	0		1	0	1
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0xfddb	0x8c64
{r9 r8}	0x83ff	0x5231
{r11 r10}	0x73ff	0x73ff

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						1	0	1	x	0	1	x	x	x	x	x
guard	guard_1								guard_0							
	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x4300	0x58c8
{r9 r8}	0x83ff	0x5231
{r11 r10}	0x73ff	0x73ff

CMACI.A Complex MAC Imaginary to Accumulator A

Assembly Syntax cmaci.a rX.e, rY.e

Description {g0 r1 r0} += rX * r(Y + 1) + r(X + 1) * rY

Example cmaci.a r4, r12

Architectural state before the instruction is executed:

	15 14 13 12 11 10 9 8 7 6	5	4	3	2	1	0
fmode	reserved	rez	sat	res	q15	sre	mre
		0	1		1	0	1

	15 14 13 12 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	1	1	1	1	1	1	1	1

Register		
{r1 r0}	0xfddb	0x8c64
{r5 r4}	0x63ff	0x63ff
{r13 r12}	0x63ff	0x63ff

Architectural state after the instruction is executed:

	15 14 13 12 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		1	0	1	x	0	1	x	x	x	x	x

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0

Register		
{r1 r0}	0x7fff	0xffff
{r5 r4}	0x63ff	0x63ff
{r13 r12}	0x63ff	0x63ff

CMACI.B Complex MAC Imaginary to Accumulator B

Assembly Syntax cmaci.b rX.e, rY.e

Description {g1 r3 r2} += rX * r(Y + 1) + r(X + 1) * rY

Example cmaci.b r8, r10

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											0	0		1	0	1
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0xfddb	0x8c64
{r9 r8}	0x8012	0x7777
{r11 r10}	0x3214	0x8312

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						1	0	1	x	0	0	x	x	x	x	x
guard	guard_1								guard_0							
	0	0	0	0	0	0	0	1	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0xa975	0xa184
{r9 r8}	0x8012	0x7777
{r11 r10}	0x3214	0x8312

CMP Compare Immediate

Assembly Syntax cmp rX, IMM4S

Description If rX ≥ IMM4S: hwf<ge> = 1;
If rX > IMM4S: hwf<gt> = 1,
other flags set by the result of (rX – IMM4S)

Example cmp r9, –8

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
r9	0x7fff

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	1	1	0	x	x	x

Register	
r9	0x7fff

CMP

Compare Register to Register

Assembly Syntax cmp rX, rY

Description

If rX ≥ rY: hwf<ge> = 1
If rX > rY: hwf<gt> = 1
other flags set by the result of (rX – rY)

Example cmp r8, r3

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	x		x	x	x
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register	
r8	0xffff
r3	0xffff

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			0	x	x	x	1	1	0	1	x	x	x

Register	
r8	0xffff
r3	0xffff

CMP.E Compare (Extended Precision)

Assembly Syntax cmp.e rX.e, rY.e

Description If {r(X + 1) rX} ≥ {r(Y + 1) rY}: hwf<ge> = 1
If {r(X + 1) rX} > {r(Y + 1) rY}: hwf<gt> = 1
other flags set by the result of: ({r(X + 1) rX} – {r(Y + 1) rY})

Example cmp.e r12, r0

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r13 r12}	0x8f34	0xd2a1
{r1 r0}	0x4343	0x7734

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	1	0	0	0	x	x	x

Register		
{r13 r12}	0x8f34	0xd2a1
{r1 r0}	0x4343	0x7734

CMP.E Compare Immediate (Extended Precision)

Assembly Syntax cmp.e rX.e, IMM4S

Description
If {r(X + 1) rX} ≥ IMM4S: hwf<ge> = 1
If {r(X + 1) rX} > IMM4S: hwf<gt> = 1
other flags set by the result of: ({r(X + 1) rX} – IMM4S)

Example cmp.e r12, –1

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved						rez	sat	res	q15	sre	mre	{r13 r12}	0x8f34	0xd2a1
							x	x		x	x	x			
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er			
		x	x	x	x	x	x	x	x	x	x	x			

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r13 r12}	0x8f34	0xd2a1
		0	x	x	x	0	0	0	0	x	x	x			

CMULR.A Complex Multiplication Real to Accumulator A

Assembly Syntax cmulr.a rX.e, rY.e

Description {r1 r0} = r(X + 1) * r(Y + 1) – rX * rY

Example cmulr.a r4, r12

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								1	1		1	0	1
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0xfddb	0x8c64
{r5 r4}	0x7777	0x7777
{r13 r12}	0x8120	0x3214

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			1	x	1	x	1	0	x	x	x	x	x

Register		
{r1 r0}	8000	0000
{r5 r4}	0x7777	0x7777
{r13 r12}	0x8120	0x3214

CMULR.B Complex Multiplication Real to Accumulator B

Assembly Syntax cmulr.b rX.e, rY.e

Description {r3 r2} = r(X + 1) * r(Y + 1) – rX * rY

Example cmulr.b r8, r10

Architectural state before the instruction is executed:

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

fmode	reserved	rez	sat	res	q15	sre	mre
		x	0		1	0	0

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0xfddb	0x8c64
{r9 r8}	0x0001	0xffff
{r11 r10}	0x73ff	0x0800

Architectural state after the instruction is executed:

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

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	1	x	x	x	x	x

Register		
{r3 r2}	0x0000	0x7bff
{r9 r8}	0x0001	0xffff
{r11 r10}	0x73ff	0x0800

CMULI.A Complex Multiplication Imaginary to Accumulator A

Assembly Syntax cmuli.a rX.e, rY.e

Description {r1 r0} = rX * r(Y + 1) + r(X + 1) * rY

Example cmuli.a r4, r12

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								0	1		1	0	1
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0xfddb	0x8c64
{r5 r4}	0x63ff	0x63ff
{r13 r12}	0x63ff	0x63ff

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			1	0	1	x	1	1	x	x	x	x	x

Register		
{r1 r0}	0x7fff	0xffff
{r5 r4}	0x63ff	0x63ff
{r13 r12}	0x63ff	0x63ff

CMULI.B Complex Multiplication Imaginary to Accumulator B

Assembly Syntax cmuli.b rX.e, rY.e

Description {r3 r2} = rX * r(Y + 1) + r(X + 1) * rY

Example cmuli.b r8, r10

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							0	0		1	0	1
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0xfddb	0x8c64
{r9 r8}	0xffff	0x0001
{r11 r10}	0x3214	0x8312

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	1	x	x	x	x	x

Register		
{r3 r2}	0x0000	0xaf02
{r9 r8}	0xffff	0x0001
{r11 r10}	0x3214	0x8312

DMAC.A Double MAC to Accumulator A

Assembly Syntax dmac.a rX.e, rY.e

Description {g0 r1 r0} += {r(X + 1) rX} * {r(Y + 1) rY}

Example dmac.a r10, r8

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											0	0		1	0	1
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0

Register		
{r1 r0}	0x0000	0x0000
{r9 r8}	0x8000	0x0000
{r11 r10}	0x0000	0x0001

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						0	0	x	x	0	0	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	1	1	1	1	1	1	1	1

Register		
{r1 r0}	0xffff	0xffff
{r9 r8}	0x8000	0x0000
{r11 r10}	0x0000	0x0001

DMAC.B Double MAC to Accumulator B

Assembly Syntax dmac.b rX.e, rY.e

Description {g1 r3 r2} += {r(X + 1) rX} * {r(Y + 1) rY}

Example dmac.b r10, r8

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											x	0		1	0	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x0000	0x0000
{r9 r8}	0x8000	0x0000
{r11 r10}	0x0000	0x0001

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						0	0	x	x	0	0	x	x	x	x	x
guard	guard_1								guard_0							
	1	1	1	1	1	1	1	0	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0xffff	0xffff
{r9 r8}	0x8000	0x0000
{r11 r10}	0x0000	0x0001

DMUL.A Multiplication (Extended Precision) to Accumulator A

Assembly Syntax dmul.a rX.e, rY.e

Description {r1 r0} = {r(X + 1) rX} * {r(Y + 1) rY}

Example dmul.a r10, r8

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											x	0		1	0	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0x0000	0x0000
{r9 r8}	0x8000	0x0000
{r11 r10}	0x8000	0x0000

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						0	x	x	x	0	1	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0x7fff	0xffff
{r9 r8}	0x8000	0x0000
{r11 r10}	0x8000	0x0000

DMUL.B Multiplication (Extended Precision) to Accumulator B

Assembly Syntax dmul.b rX.e, rY.e

Description {r3 r2} = {r(X + 1) rX} * {r(Y + 1) rY}

Example dmul.b r10, r8

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											x	0		1	0	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x0000	0x0000
{r9 r8}	0x8000	0x0000
{r11 r10}	0x8000	0x0000

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						0	x	x	x	0	1	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x7fff	0xffff
{r9 r8}	0x8000	0x0000
{r11 r10}	0x8000	0x0000

HALT Halt (Synthetic Instruction)

Assembly Syntax halt

Description Replaced by bits %smode, 15

IDLE Idle (Synthetic Instruction)

Assembly Syntax idle

Description Replaced by bits %smode, 13

IMUL.A Integer Multiply to Accumulator A

Assembly Syntax imul.a rX, rY

Description {r1 r0} = rX * rY

Example imul.a r3, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0xf678	0xc521
r3	0x8000	
r4	0x8020	

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	1	x	x	x	x	x

Register		
{r1 r0}	0x3ff0	0x0000
r3	0x8000	
r4	0x8020	

IMUL.B Integer Multiply to Accumulator B

Assembly Syntax imul.b rX, rY

Description {r3 r2} = rX * rY

Example imul.b r1, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0xf678	0xc521
r1	0x8000	
r4	0x8020	

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	1	x	x	x	x	x

Register		
{r3 r2}	0x3ff0	0x0000
r1	0x8000	
r4	0x8020	

LD Load

Assembly Syntax ld rX, rY [, n]

Description –4 ≤ n ≤ 3
rX ← mem[rY + n]

Example ld r3, r15, 2

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
r15	0x3401
r3	0x5673

Data Memory	
0x3403	0x4500

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
r15	0x3401
r3	0x4500

Data Memory	
0x3403	0x4500

LDA Load (Synthetic Instruction)

Assembly Syntax lda rX, LABEL

Description Replaced by:
movl rX, {LABEL}[7:0] and
movh rX, {LABEL}[15:8]

LDU Load with Update

Assembly Syntax ldu rX, rY, n

Description n = {–2, –1, 1, 2}
rX ← mem[rY]; rY = rY + n;

Example ldu r3, r15, 2

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	x		x	x	x
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register	
r15	0x3401
r3	0x5673

Data Memory	
0x3401	0x4500

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register	
r15	0x3403
r3	0x4500

Data Memory	
0x3401	0x4500

Load Double with Update

Assembly Syntax lddu rX, rY.e, n where n = {2,–2}

Description

```
if n = 2 {rX ← mem[rY]
r(X + 1) ← mem[rY + 1]}
else {rX ← mem[rY – 1]
r(X + 1) ← mem[rY]}
rY = rY + n
```

Example lddu r0, r15, 2

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
r15	0x3400	
{r1 r0}	0x8976	0x5682

Data Memory	
0x3400	0x4500
0x3401	0xff56

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
r15	0x3402	
{r1 r0}	0xff56	0x4500

Data Memory	
0x3400	0x4500
0x3401	0xff56

LDX Load with Register Based Offset

Assembly Syntax ldx rX, rY.e

Description rX ← mem[rY + r(Y + 1)]

Example ldx r4, r14

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r15 r14}	0x0001	0x3400
r4	0xc567	

Data Memory	
0x3401	0xff56

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r15 r14}	0x0001	0x3400
r4	0xff56	

Data Memory	
0x3401	0xff56

LDXU Load with Register Based Offset and Update

Assembly Syntax ldxu rX, rY.e

Description rX ← mem[rY + r(Y + 1)]; rY += r(Y + 1)

Example ldxu r4, r14

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r15 r14}	0x0001	0x3400
r4	0xc567	

Data Memory	
0x3401	0xff56

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r15 r14}	0x0001	0x3401
r4	0xff56	

Data Memory	
0x3401	0xff56

MAC2.A Dual MAC to Accumulator A

Assembly Syntax mac2.a rX.e, rY.e

Description {g0 r1 r0} += rX * r(Y) + r(X + 1) * r(Y + 1)

Example mac2.a r2, r4

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											x	0		0	0	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0

Register		
{r1 r0}	0x7ffd	0xf750
{r5 r4}	0xee3a	0x04b0
{r3 r2}	0xf060	0x0050

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						1	0	1	x	0	0	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0

Register		
{r1 r0}	0x8115	0x2410
{r5 r4}	0xee3a	0x04b0
{r3 r2}	0xf060	0x0050

MAC2.B Dual MAC to Accumulator B

Assembly Syntax mac2.b rX.e, rY.e

Description {g1 r3 r2} += rX * r(Y) + r(X + 1) * r(Y + 1)

Example mac2.b r0, r4

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											0	1		1	0	1
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x7ffd	0xf750
{r5 r4}	0xee3a	0x04b0
{r1 r0}	0xf060	0x0050

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						1	0	1	x	0	1	x	x	x	x	x
guard	guard_1								guard_0							
	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x7fff	0xffff
{r5 r4}	0xee3a	0x04b0
{r1 r0}	0xf060	0x0050

MAC.A Multiply Accumulate to Accumulator A

Assembly Syntax mac.a rX, rY

Description {g0 r1 r0} += rX * rY

Example mac.a r6, r8

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											0	0		1	0	1
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0

Register		
{r1 r0}	0x0000	0x0000
r6	0x6b85	
r8	0x2b85	

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						0	0	x	x	0	1	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0

Register		
{r1 r0}	0x248e	0xe632
r6	0x6b85	
r8	0x2b85	

MAC.B Multiply Accumulate to Accumulator B

Assembly Syntax mac.b rX, rY

Description {g1 r3 r2} += rX * rY

Example mac.b r6, r8

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											0	0		1	0	1
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x0000	0x0000
r6	0x8685	
r8	0x2b85	

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						0	0	x	x	0	0	x	x	x	x	x
guard	guard_1								guard_0							
	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0xd6b2	0xf432
r6	0x8685	
r8	0x2b85	

MACN.A Multiply Accumulate with Negation to Accumulator A

Assembly Syntax macn.a rX, rY

Description {g0 r1 r0} –= rX * rY

Example macn.a r4, r6

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											1	1		1	0	1
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	1	1	1	1	1	1	1	1

Register		
{r1 r0}	0x8003	0x7c6e
r6	0x0fa0	
r4	0x0320	

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						1	0	1	x	1	0	x	x	x	x	x
guard	guard_1								guard_0							
	x	x	x	x	x	x	x	x	1	1	1	1	1	1	1	1

Register		
{r1 r0}	0x8000	0x0000
r6	0x0fa0	
r4	0x0320	

MACN.B Multiply Accumulate with Negation to Accumulator B

Assembly Syntax macn.b rX, rY

Description {g1 r3 r2} –= rX * rY

Example macn.b r4, r6

Architectural state before the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved										rez	sat	res	q15	sre	mre
											x	0		0	0	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						x	x	x	x	x	x	x	x	x	x	x
guard	guard_1								guard_0							
	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x8003	0x7c6e
r6	0x0fa0	
r4	0x0320	

Architectural state after the instruction is executed:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved					v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
						1	0	1	x	1	1	x	x	x	x	x
guard	guard_1								guard_0							
	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x7fd2	0xa86e
r6	0x0fa0	
r4	0x0320	

MAX Maximum

Assembly Syntax max rX, rY

Description rX = max (rX, rY)
if rX ≥ rY, hwf<c> = 1;
other flags are set by the result of (rX – rY)

Example max r2, r12

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r2	0x8f34
							x	x		x	x	x	r12	0x7734
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0x7734
		0	x	x	x	0	1	1	0	x	x	x	r12	0x7734

MAX.E Maximum (Extended Precision)

Assembly Syntax max.e rX.e, rY.e

Description {r(X + 1) rX} = max ({r(X + 1) rX, {r(Y + 1) rY})
if {r(X + 1) rX} ≥ {r(Y + 1)rY} hwf<c> = 1;
other flags are set on the result of {r(X + 1)rX} – {r(Y + 1)rY}

Example max.e r10, r0

Architectural state before the instruction is executed:

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

fmode	reserved	rez	sat	res	q15	sre	mre
		x	x		x	x	x

Register		
{r1 r0}	0x3fff	0x7fff
{r11 r10}	0x7fff	0x7fff

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

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

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	1	1	1	0	x	x	x

Register		
{r1 r0}	0x3fff	0x7fff
{r11 r10}	0x7fff	0x7fff

MIN Minimum

Assembly Syntax min rX, rY

Description rX = min (rX, rY)
if rX ≤ rY hwflag<c> = 1;
other flags are set on the result of (rX – rY)

Example min r2, r12

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r2	0x8f34
							x	x		x	x	x	r12	0x7734

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0x8f34
		0	x	x	x	1	0	0	0	x	x	x	r12	0x7734

MIN.E Minimum (Extended Precision)

Assembly Syntax min.e rX.e, rY.e

Description {r(X + 1) rX} = min ({r(X + 1) rX, {r(Y + 1) rY})
if {r(X +1)rX} ≤ {r(Y + 1)rY} hwf<c> = 1;
other flags are set on the result of {r(X + 1)rX} – {r(Y + 1)rY}

Example min.e r10, r0

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	x		x	x	x
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0x3fff	0x7fff
{r11 r10}	0x7fff	0x7fff

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			0	x	x	x	0	1	1	0	x	x	x

Register		
{r1 r0}	0x3fff	0x7fff
{r11 r10}	0x3fff	0x7fff

MOV Move to PC

Assembly Syntax mov %pc, cY

Description cY = tpc/rpc

Example mov %pc, %rpc

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	pc	0xf154
								x	x		x	x	x	rpc	0xd4a5
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	pc	0xd4a5
			x	x	x	x	x	x	x	x	x	x	x	rpc	0xd4a5

MOV Move Operand Register to Control Register

Assembly Syntax mov cX, rY

Description cX = rY

Example mov %loop0, r3

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	loop0	0xaf56
							x	x		x	x	x	r3	0xf00d
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	loop0	0xf00d
		x	x	x	x	x	x	x	x	x	x	x	r3	0xf00d

MOV Move Control Register to Operand Register

Assembly Syntax mov rX, cY

Description rX = cY

Example mov r3, %fmode

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	fmode	0xaf56
								x	x		x	x	x	r3	0xf00d

hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	fmode	0xaf56
			x	x	x	x	x	x	x	x	x	x	x	r3	0xaf56

MOV Move Operand Register to Operand Register

Assembly Syntax mov rX, rY

Description rX = rY

Example mov r3, r4

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	r4	0xaf56
								x	x		x	x	x	r3	0xf00d
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r4	0xaf56
			x	x	x	x	x	x	x	x	x	x	x	r3	0xaf56

MOV Move Immediate to Operand Register

Assembly Syntax mov rX, IMM4S

Description rX = IMM4S

Example mov r4, –8

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r4	0xaf56
							x	x		x	x	x		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r4	0xfff8
		x	x	x	x	x	x	x	x	x	x	x		

MOV Move Immediate to Operand Register (Synthetic Instruction)

Assembly Syntax mov rX, IMM

Description If –8 ≤ IMM ≤ 7, then
replaced by:
  mov rX, IMM4S
else replaced by:
  movl rX, IMM[7:0] and
  movh rX, IMM[15:8]

MOV Move Immediate to Control Register (Synthetic Instruction)

Assembly Syntax mov cX, IMM16U

Description cX = {%fmode, %loop0, %loop1, %loop2, %loop3, %guard}
replaced by:
movl cX, IMM16U[7:0] and
movh cX, IMM16U[15:8]

MOVH Move Immediate to Higher Byte of Control Register

Assembly Syntax movh cX, IMM8U

Description cX[15:8] = IMM8U, cX = {%fmode, %loop0, %loop1, %loop2, %loop3, %guard}

Example movh %guard, 0

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	guard	0xaf56
								x	x		x	x	x		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	guard	0x0056
			x	x	x	x	x	x	x	x	x	x	x		

MOVH Move Immediate to Higher Byte of Operand Register

Assembly Syntax movh rX, IMM8U

Description rX[15:8] = IMM8U

Example movh r4, 0

Architectural state before the instruction is executed:

	15 11 10 9 8 7 6	5	4	3	2	1	0	Register	
fmode	reserved	rez	sat	res	q15	sre	mre	r4	0xaf56
		x	x		x	x	x		

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r4	0x0056
		x	x	x	x	x	x	x	x	x	x	x		

MOVL — Move Immediate to Lower Byte of Control Register

Assembly Syntax movl cX, IMM8U

Description cX[7:0] = IMM8U, cX = {%fmode, %loop0, %loop1, %loop2, %loop3, %guard}

Example movl %guard, 255

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	guard	0xaf56
							x	x		x	x	x		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	guard	0xafff
		x	x	x	x	x	x	x	x	x	x	x		

MOVL Move Immediate to Low Byte of Operand Register

Assembly Syntax movl rX, IMM8U

Description rX[7:0] = IMM8U

Example movl r4, 255

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	r4	0xaf56
								x	x		x	x	x		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r4	0xafff
			x	x	x	x	x	x	x	x	x	x	x		

MOVLH Move Long Immediate to Operand Register (Synthetic Instruction)

Assembly Syntax movlh rX, IMM32U

Description Replaced by:

```
movl rX, IMM32U[23:16]
movl r(X – 1), IMM32U[7:0]
movh rX, IMM32U[31:24]
movh r(X – 1), IMM32U[15:0]
```

MUL.A Multiply to Accumulator A

Assembly Syntax mul.a rX, rY

Description {r1 r0} = rX * rY

Example mul.a r3, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	0		0	x	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0xf678	0xc521
r3	0x8000	
r4	0x8020	

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	1	x	x	x	x	x

Register		
{r1 r0}	0x3ff0	0x0000
r3	0x8000	
r4	0x8020	

MUL.B Multiply to Accumulator B

Assembly Syntax mul.b rX, rY

Description {r3 r2} = rX * rY

Example mul.b r1, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	0		0	x	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0xf678	0xc521
r1	0x8000	
r4	0x8020	

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	1	x	x	x	x	x

Register		
{r3 r2}	0x3ff0	0x0000
r1	0x8000	
r4	0x8020	

MULN.A Multiply with Negation to Accumulator A

Assembly Syntax muln.a rX, rY

Description {r1 r0} = –rX * rY

Example muln.a r3, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							0	0		0	x	1
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0xf678	0xc521
r3	0x03ff	
r4	0x03ff	

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	0	x	x	x	x	x

Register		
{r1 r0}	0xfff0	0x87ff
r3	0x03ff	
r4	0x03ff	

MULN.B Multiply with Negation to Accumulator B

Assembly Syntax muln.b rX, rY

Description {r3 r2} = –rX * rY

Example muln.b r1, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	0		0	x	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0xf678	0xc521
r1	0x03ff	
r4	0x03ff	

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	0	x	x	x	x	x

Register		
{r3 r2}	0xfff0	0x07ff
r1	0x03ff	
r4	0x03ff	

NEG Negate

Assembly Syntax neg rX, rY

Description rX = –rY

Example neg r3, r4

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	r3	0x8000
								x	x		x	x	x	r4	0x8000
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r3	0x7fff
			1	x	0	x	0	1	1	0	x	x	x	r4	0x8000

Note: Negation of 0 does not set the carry bit in hwflag.

NEG.E Negate (Extended Precision)

Assembly Syntax neg.e rX.e, rY.e

Description {r(X + 1) rX} = –{r(Y + 1) rY}

Example neg.e r0, r6

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved						rez	sat	res	q15	sre	mre	{r1 r0}	0x0000	0x0000
							x	x		x	x	x	{r7 r6}	0x0000	0x0001
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er			
		x	x	x	x	x	x	x	x	x	x	x			

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r1 r0}	0xffff	0xffff
		0	x	x	x	0	0	0	0	x	x	x	{r7 r6}	0x0000	0x0001

Note: Negation of 0 does not set the carry bit in hwflag.

NOP No Operation

Assembly Syntax nop

Description No operation

Example nop

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

NORM Normalize

Assembly Syntax norm rX, rY

Description
If rY == 0 then rX = 0
else if rY == –1 then rX = 15
else if rY >= 0 then rX = 14 - bit position of leading 1 in rY
else rX = 14 - bit position of leading 0 in rY

Example norm r4, r2

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r2	0x0001
							x	x		x	x	x	r4	0x8000
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0x0001
		0	x	x	x	0	1	1	0	x	x	x	r4	0x000e

NORM.E Normalize (Extended Precision)

Assembly Syntax norm.e rX.e, rY.e

Description

If rY.e == 0 then rX = 0
else if rY.e == -1 then rX = 31
else if rY.e >= 0 then rX = 30 - bit position of leading 1 in rY.e
else rX = 30 - bit position of leading 0 in rY.e

Example norm.e r6, r4

Architectural state before the instruction is executed:

	15 11 10 9 8 7 6	5	4	3	2	1	0	Register		
fmode	reserved	rez	sat	res	q15	sre	mre	r6	0xf567	
		x	x		x	x	x	{r5 r4}	0x0000	0xffff

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r6	0x000f	
		0	x	x	x	0	1	1	0	x	x	x	{r5 r4}	0x0000	0xffff

NOT Logical Not

Assembly Syntax not rX, rY

Description rX =~ rY

Example not r4, r2

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	r2	0x0001
								x	x		x	x	x	r4	0x8000

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0x0001
			0	x	x	x	0	0	0	0	x	x	x	r4	0xfffe

NOT.E Logical Not (Extended Precision)

Assembly Syntax not.e rX.e, rY.e

Description {r(X + 1)rX} =~ {r(Y + 1)rY}

Example not.e r6, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x

Register		
{r7 r6}	0xf567	0x0984
{r5 r4}	0x0000	0xffff

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	0	0	0	x	x	x

Register		
{r7 r6}	0xffff	0x0000
{r5 r4}	0x0000	0xffff

OR Logical OR

Assembly Syntax or rX, rY

Description rX |= rY

Example or r4, r2

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r2	0x0001
							x	x		x	x	x	r4	0x8000

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0x0001
		0	x	x	x	0	1	1	0	x	x	x	r4	0x8001

OR.E Logical OR (Extended Precision)

Assembly Syntax or.e rX.e, rY.e

Description {r(X + 1)rX} |= {r(Y + 1)rY}

Example or.e r6, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r7 r6}	0xf567	0x0984
{r5 r4}	0x0000	0xffff

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	0	0	0	x	x	x

Register		
{r7 r6}	0xf567	0xffff
{r5 r4}	0x0000	0xffff

PADD.A Parallel Add Registers to Accumulator A

Assembly Syntax padd.a rX.e, rY.e

Description r0 = rX + rY; r1 = r(X + 1) + r(Y + 1)

Example padd.a r2, r4

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	x		x	x	x
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0x8f34	0xc342
{r3 r2}	0x8e0a	0x8c23
{r5 r4}	0x00f0	0x31c0

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0x8efa	0xbde3
{r3 r2}	0x8e0a	0x8c23
{r5 r4}	0x00f0	0x31c0

PADD.B Parallel Add Registers to Accumulator B

Assembly Syntax padd.b rX.e, rY.e

Description r2 = rX + rY; r3 = r(X + 1) + r(Y + 1)

Example padd.b r6, r8

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x8efa	0xc342
{r7 r6}	0x7fff	0x8000
{r9 r8}	0x0001	0xffff

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x8000	0x7fff
{r7 r6}	0x7fff	0x8000
{r9 r8}	0x0001	0xffff

PSUB.A Parallel Subtract Registers to Accumulator A

Assembly Syntax psub.a rX.e, rY.e

Description r0 = rX – rY; r1 = r(X + 1) – r(Y + 1)

Example psub.a r2, r4

Architectural state before the instruction is executed:

	15 11 10 9 8 7 6	5	4	3	2	1	0
fmode	reserved	rez	sat	res	q15	sre	mre
		x	x		x	x	x

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0x8f34	0xc342
{r3 r2}	0x8e0a	0x8c23
{r5 r4}	0x00f0	0x01c0

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r1 r0}	0x8d1a	0xda54
{r3 r2}	0x8e0a	0x8c23
{r5 r4}	0x00f0	0x01c0

PSUB.B Parallel Subtract Registers to Accumulator B

Assembly Syntax psub.b rX.e, rY.e

Description r2 = rX – rY; r3 = r(X + 1) – r(Y + 1)

Example psub.b r6, r8

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x8efa	0xc342
{r7 r6}	0x7fff	0x8000
{r9 r8}	0xffff	0x0002

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x8000	0x7ffe
{r7 r6}	0x7fff	0x8000
{r9 r8}	0xffff	0x0002

RET Return

Assembly Syntax ret

Description pc = rpc
Return: used as the last statement of a subroutine initiated by a call.

Example ret

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

RETI Interrupt Return

Assembly Syntax reti

Description pc = tpc; imask<gie> = imask<pgie>; ip0<epl> = ip0<pepl>
Return from interrupt: used as the last statement in an interrupt service routine.

Example reti

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	x		x	x	x
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

REVB Reverse Bit

Assembly Syntax revb rX, IMM4U

Description Reverses order of rX[IMM4U:0]. If IMM4U <15, rX[15:IMM4U] = 0

Example revb r2, 15

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	r2	0xc001
								x	x		x	x	x		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0x8003
			0	x	x	x	0	0	0	0	x	x	x		

ROUND.E Round (Extended Precision)

Assembly Syntax round.e rX.e, rY.e

Description {r(X + 1) rX} = {r(Y + 1) rY} + 0x0000 8000

Example round.e r6, r4

Architectural state before the instruction is executed:

	15 … 6	5	4	3	2	1	0
fmode	reserved	rez	sat	res	q15	sre	mre
		x	x		x	x	x

Register		
{r7 r6}	0xf567	0x0984
{r5 r4}	0x0000	0xffff

	15 … 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15 … 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	1	1	0	x	x	x

Register		
{r7 r6}	0x0001	0x7fff
{r5 r4}	0x0000	0xffff

SHLA Shift Left Arithmetic Immediate

Assembly Syntax shla rX, IMM4U

Description rX = rX << IMM4U
This is a true arithmetic left shift. If the *fmode* saturation bit is set, a shift which would result in a number less than MAX_NEG or greater than MAX_POS will set *hwflag*<v,sv>, and produce MAX_NEG or MAX_POS respectively. If the *fmode* saturation bit is clear, then a shift which would result in a number less than MAX_NEG or greater than MAX_POS will set *hwflag*<v,sv>, while producing the same result as a SHLL.

Example shla r2, 3

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r2	0x0001
							x	0		x	x	x		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0x0008
		0	x	x	x	0	1	1	0	x	x	x		

SHLA Shift Left Arithmetic Register

Assembly Syntax shla rX, rY

Description rX = rX << rY[3:0]
This is a true arithmetic left shift. If the fmode saturation bit is set, a shift which would result in a number less than MAX_NEG or greater than MAX_POS will set hwflag<v,sv>, and produce MAX_NEG or MAX_POS respectively. If the fmode saturation bit is clear, then a shift which would result in a number less than MAX_NEG or greater than MAX_POS will set hwflag<v,sv>, while producing the same result as a SHLL.

Example 1 shla r2, r4

Architectural state before the instruction is executed:

<table>
<tr><td></td><td>15 11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>Register</td><td></td></tr>
<tr><td>fmode</td><td colspan="6">reserved</td><td>rez</td><td>sat</td><td>res</td><td>q15</td><td>sre</td><td>mre</td><td>r2</td><td>0xfffc</td></tr>
<tr><td></td><td colspan="6"></td><td>x</td><td>1</td><td></td><td>x</td><td>x</td><td>x</td><td>r4</td><td>0x000f</td></tr>
<tr><td>hwflag</td><td>reserved</td><td>v</td><td>gv</td><td>sv</td><td>gsv</td><td>c</td><td>ge</td><td>gt</td><td>z</td><td>ir</td><td>ex</td><td>er</td><td></td><td></td></tr>
<tr><td></td><td></td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td></tr>
</table>

Architectural state after the instruction is executed:

<table>
<tr><td></td><td>15 11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>Register</td><td></td></tr>
<tr><td>hwflag</td><td>reserved</td><td>v</td><td>gv</td><td>sv</td><td>gsv</td><td>c</td><td>ge</td><td>gt</td><td>z</td><td>ir</td><td>ex</td><td>er</td><td>r2</td><td>0x8000</td></tr>
<tr><td></td><td></td><td>1</td><td>x</td><td>1</td><td>x</td><td>0</td><td>0</td><td>0</td><td>0</td><td>x</td><td>x</td><td>x</td><td>r4</td><td>0x000f</td></tr>
</table>

Example 2 shla r2, r4

Architectural state before the instruction is executed:

<table>
<tr><td></td><td>15</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>Register</td><td></td></tr>
<tr><td>fmode</td><td colspan="7">reserved</td><td>rez</td><td>sat</td><td>res</td><td>q15</td><td>sre</td><td>mre</td><td>r2</td><td>0xfffc</td></tr>
<tr><td></td><td colspan="7"></td><td>x</td><td>0</td><td></td><td>x</td><td>x</td><td>x</td><td>r4</td><td>0x000f</td></tr>
<tr><td>hwflag</td><td colspan="2">reserved</td><td>v</td><td>gv</td><td>sv</td><td>gsv</td><td>c</td><td>ge</td><td>gt</td><td>z</td><td>ir</td><td>ex</td><td>er</td><td></td><td></td></tr>
<tr><td></td><td colspan="2"></td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td></tr>
</table>

Architectural state after the instruction is executed:

<table>
<tr><td></td><td>15</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>Register</td><td></td></tr>
<tr><td>hwflag</td><td colspan="2">reserved</td><td>v</td><td>gv</td><td>sv</td><td>gsv</td><td>c</td><td>ge</td><td>gt</td><td>z</td><td>ir</td><td>ex</td><td>er</td><td>r2</td><td>0x0000</td></tr>
<tr><td></td><td colspan="2"></td><td>1</td><td>x</td><td>1</td><td>x</td><td>0</td><td>1</td><td>0</td><td>1</td><td>x</td><td>x</td><td>x</td><td>r4</td><td>0x000f</td></tr>
</table>

SHLA.E Shift Left Arithmetic Immediate (Extended Precision)

Assembly Syntax shla.e rX.e, IMM4U

Description {r(X + 1) rX} = {r(X + 1) rX} << IMM4U
This is a true arithmetic left shift. If the *fmode* saturation bit is set, a shift which would result in a number less than MAX_NEG or greater than MAX_POS will set hwflag<v,sv>, and produce MAX_NEG or MAX_POS respectively. If the fmode saturation bit is clear, then a shift which would result in a number less than MAX_NEG or greater than MAX_POS will set hwflag<v,sv>, while producing the same result as a SHLL.E

Example shla.e r2, 2

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved							rez	sat	res	q15	sre	mre	{r3 r2}	0x3fff	0xffff
								x	1		x	x	x			

hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r3 r2}	0x7fff	0xffff
			1	x	1	x	0	1	1	0	x	x	x			

SHLA.E Shift Left Arithmetic Register (Extended Precision)

Assembly Syntax shla.e rX.e, rY.e

Description {r(X + 1) rX} = {r(X + 1) rX} << rY[4:0]
This is a true arithmetic left shift. If the fmode saturation bit is set, a shift which would result in a number less than MAX_NEG or greater than MAX_POS will set hwflag<v,sv>, and produce MAX_NEG or MAX_POS respectively. If the fmode saturation bit is clear, then a shift which would result in a number less than MAX_NEG or greater than MAX_POS will set hwflag<v,sv>, while producing the same result as a SHLL.E

Example shla.e r2, r4

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved							rez	sat	res	q15	sre	mre	{r3 r2}	0x3fff	ffff
								x	1		x	0	x	r4	0x0002	

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r3 r2}	0x7fff	0xffff
			1	x	1	x	0	1	1	0	x	x	x	r4	0x0002	

Shift Left Logical Immediate

Assembly Syntax shll rX, IMM5U

Description rX = rX << IMM5U

Example shll r2, 3

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r2	0x0001
							x	x		x	x	x		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0x0008
		0	x	x	x	0	1	1	0	x	x	x		

SHLL

Shift Left Logical Register

Assembly Syntax shll rX, rY

Description rX = rX << rY[3:0]

Example shll r2, r4

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	r2	0xfffc
								x	x		x	x	x	r4	0x000f
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
			x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0x0000
			0	x	x	x	0	1	0	1	x	x	x	r4	0x000f

SHLL.E

Shift Left Logical Immediate (Extended Precision)

Assembly Syntax shll.e rX.e, IMM5U

Description {r(X + 1) rX} = {r(X + 1) rX} << IMM5U

Example shll.e r2, 2

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved						rez	sat	res	q15	sre	mre	{r3 r2}	0x3fff	0xffff
							x	x		x	x	x			
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er			
		x	x	x	x	x	x	x	x	x	x	x			

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r3 r2}	0xffff	0xfffc
		0	x	x	x	0	0	0	0	x	x	x			

SHLL.E

Shift Left Logical Register (Extended Precision)

Assembly Syntax shll.e rX.e, rY.e

Description {r(X + 1) rX} = {r(X + 1) rX} << rY[4:0]

Example shll.e r2, r4

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	x		x	x	x

Register		
{r3 r2}	0x3fff	0xffff
r4	0x0002	

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			0	x	x	x	0	0	0	0	x	x	x

Register		
{r3 r2}	0xffff	0xfffc
r4	0x0002	

SHRA Shift Right Arithmetic Immediate

Assembly Syntax shra rX, IMM5U

Description rX = rX >> IMM5U

Example shra r2, 1

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r2	0x8001
							x	x		x	1	x		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0xc001
		0	x	x	x	0	0	0	0	x	x	x		

Shift Right Arithmetic Register

Assembly Syntax shra rX, rY

Description rX = rX >> rY[3:0]

Example shra r2, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r2	0x8001
							x	x		x	0	x	r4	0x0001
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0xc000
		0	x	x	x	0	0	0	0	x	x	x	r4	0x0001

SHRA.E Shift Right Arithmetic Immediate (Extended Precision)

Assembly Syntax shra.e rX.e, IMM5U

Description {r(X + 1) rX} = {r(X + 1) rX} >> IMM5U

Example shra.e r2, 2

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved						rez	sat	res	q15	sre	mre	{r3 r2}	0x3fff	0xffff
							x	x		x	1	x			
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er			
		x	x	x	x	x	x	x	x	x	x	x			

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r3 r2}	0x1000	0x0000
		0	x	x	x	0	1	1	0	x	x	x			

SHRA.E

Shift Right Arithmetic Register (Extended Precision)

Assembly Syntax shra.e rX.e, rY.e

Description {r(X + 1) rX} = {r(X + 1) rX} >> rY[4:0]

Example shra.e r2, r4

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved						rez	sat	res	q15	sre	mre	{r3 r2}	0x3fff	0xffff
							x	x		x	0	x	r4	0x0002	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er			
		x	x	x	x	x	x	x	x	x	x	x			

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r3 r2}	0x0fff	0xffff
		0	x	x	x	0	1	1	0	x	x	x	r4	0x0002	

SHRL Shift Right Logical Immediate

Assembly Syntax shrl rX, IMM5U

Description rX = rX >> IMM5U

Example shrl r2, 3

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved						rez	sat	res	q15	sre	mre	r2	0x8001
							x	x		x	x	x		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er		
		x	x	x	x	x	x	x	x	x	x	x		

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0x1000
		0	x	x	x	0	1	1	0	x	x	x		

SHRL Shift Right Logical Register

Assembly Syntax shrl rX, rY

Description rX = rX >> rY[3:0]

Example shrl r2, r4

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
fmode	reserved							rez	sat	res	q15	sre	mre	r2	0x8001
								x	x		x	0	x	r4	0x0003

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register	
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	r2	0x1000
			0	x	x	x	0	1	1	0	x	x	x	r4	0x0003

SHRL.E Shift Right Logical Immediate (Extended Precision)

Assembly Syntax shrl.e rX.e, IMM5U

Description {r(X + 1) rX} = {r(X + 1) rX} >> IMM5U

Example shrl.e r2, 2

Architectural state before the instruction is executed:

	15 11 10 9 8 7 6	5	4	3	2	1	0
fmode	reserved	rez	sat	res	q15	sre	mre
		x	x		x	x	x

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r3 r2}	0x3fff	0xffff

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		0	x	x	x	0	1	1	0	x	x	x

Register		
{r3 r2}	0x0fff	0xffff

SHRL.E Shift Right Logical Register (Extended Precision)

Assembly Syntax shrl.e rX.e, rY.e

Description {r(X + 1) rX} = {r(X + 1) rX} >>rY; [4:0]

Example shrl.e r2, r4

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved							rez	sat	res	q15	sre	mre	{r3 r2}	0x3fff	0xffff
								x	x		x	x	x	r4	0x0002	

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r3 r2}	0x0fff	0xffff
			0	x	x	x	0	1	1	0	x	x	x	r4	0x0002	

SLEEP Sleep (Synthetic Instruction)

Assembly Syntax sleep

Description Replaced by bits %smode, 14

ST Store

Assembly Syntax st rX, rY [, n]

Description –4 ≤ n ≤ 3
mem[rY + n] = rX

Example st r3, r15, 2

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
r15	0x3401
r3	0x5673

Data Memory	
0x3403	0x4500

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
r15	0x3401
r3	0x5673

Data Memory	
0x3403	0x5673

STU Store with Update

Assembly Syntax stu rX, rY, n

Description n = {–2, –1, 1, 2}
mem[rY] = rX; rY = rY + n;

Example stu r3, r15, 2

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
r15	0x3401
r3	0x5673

Data Memory	
0x3401	0x4500

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register	
r15	0x3403
r3	0x5673

Data Memory	
0x3401	0x5673

STDU Store Double with Update

Assembly Syntax stdu rX.e, rY, n where n = {2, –2}

Description

```
if n = 2 {mem[rY] = rX
mem[rY + 1] = r(X + 1) }
else {mem[rY – 1] = rX
mem[rY] = r(X + 1) }
rY = rY + n
```

Example stdu r0, r15, –2

Architectural state before the instruction is executed:

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

	reserved	rez	sat	res	q15	sre	mre
fmode		x	x		x	x	x

	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
hwflag		x	x	x	x	x	x	x	x	x	x	x

Register		
r15	0x3400	
{r1 r0}	0x8976	0x5682

Data Memory	
0x33ff	0xff56
0x3400	0x4500

Architectural state after the instruction is executed:

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

	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
hwflag		x	x	x	x	x	x	x	x	x	x	x

Register		
r15	0x33fe	
{r1 r0}	0x8976	0x5682

Data Memory	
0x33ff	0x5682
0x3400	0x8976

STX Store with Register Based Offset

Assembly Syntax stx rX, rY.e

Description mem[rY + r(Y + 1)] = rX

Example stx r4, r14

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x

Register		
{r15 r14}	0x0001	0x3400
r4	0xc567	

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Data Memory	
0x3401	0xff56

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r15 r14}	0x0001	0x3400
r4	0xc567	

Data Memory	
0x3401	0xc567

STXU Store with Register Based Offset and Update

Assembly Syntax stxu rX, rY.e

Description mem[rY + r(Y + 1)] = rX
rY = rY + r(Y + 1)

Example stxu r4, r14

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r15 r14}	0x0001	0x3400
r4	0xc567	

Data Memory	
0x3401	0xff56

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
{r15 r14}	0x0001	0x3401
r4	0xc567	

Data Memory	
0x3401	0xc567

SUB Subtract

Assembly Syntax sub rX, rY

Description rX –= rY

Example sub r13, r14

Architectural state before the instruction is executed:

<table>
<tr><td></td><td>15 11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>Register</td><td></td></tr>
<tr><td>fmode</td><td colspan="6">reserved</td><td>rez</td><td>sat</td><td>res</td><td>q15</td><td>sre</td><td>mre</td><td>r13</td><td>0x8f34</td></tr>
<tr><td></td><td colspan="6"></td><td>x</td><td>0</td><td></td><td>x</td><td>x</td><td>x</td><td>r14</td><td>0x7734</td></tr>
<tr><td>hwflag</td><td>reserved</td><td>v</td><td>gv</td><td>sv</td><td>gsv</td><td>c</td><td>ge</td><td>gt</td><td>z</td><td>ir</td><td>ex</td><td>er</td><td></td><td></td></tr>
<tr><td></td><td></td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td></tr>
</table>

Architectural state after the instruction is executed:

<table>
<tr><td></td><td>15 11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>Register</td><td></td></tr>
<tr><td>hwflag</td><td>reserved</td><td>v</td><td>gv</td><td>sv</td><td>gsv</td><td>c</td><td>ge</td><td>gt</td><td>z</td><td>ir</td><td>ex</td><td>er</td><td>r13</td><td>0x1800</td></tr>
<tr><td></td><td></td><td>1</td><td>x</td><td>1</td><td>x</td><td>1</td><td>1</td><td>1</td><td>0</td><td>x</td><td>x</td><td>x</td><td>r14</td><td>0x7734</td></tr>
</table>

Subtract (Extended Precision)

Assembly Syntax sub.e rX.e, rY.e

Description {r(X + 1) rX} –= {r(Y + 1) rY}

Example sub.e r2, r4

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	1		x	x	x

Register		
{r3 r2}	0x0000	0x0004
{r5 r4}	0xffff	0xffff

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			0	x	x	x	0	1	1	0	x	x	x

Register		
{r3 r2}	0x0000	0x0005
{r5 r4}	0xffff	0xffff

SUBC.E Subtract with Carry (Extended Precision)

Assembly Syntax subc.e rX.e, rY.e

Description {r(X + 1) rX} –= {r(Y + 1) rY} - logical inverse of carry

Example subc.e r8, r10

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved						rez	sat	res	q15	sre	mre	{r9 r8}	0x4bf1	0x5b6d
							x	1		x	x	x	{r11 r10}	0x0000	0x0001
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er			
		x	x	x	x	1	x	x	x	x	x	x			

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r9 r8}	0x4bf1	0x5b6c
		0	x	x	x	1	1	1	0	x	x	x	{r11 r10}	0x0000	0x0001

VIT_A Viterbi Instruction for Point A

Assembly Syntax vit_a rX.e, rY.e

Description

```
r0 = min {(rX + rY), (r(X + 1) + r(Y + 1))}
if ((rX + rY) < (r(X + 1) + r(Y + 1)))
vitr = vitr << 1 | 0x0001
else
vitr = vitr << 1
```

Example vit_a r4, r6

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	x		x	x	x
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Register		
r0	xxxx	
{r5 r4}	0x1123	0x0030
{r7 r6}	0x000a	0x0008
vitr	0x0000	

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			0	x	x	x	1	1	x	x	x	x	x

Register		
r0	0x0038	
{r5 r4}	0x1123	0x0030
{r7 r6}	0x000a	0x0008
vitr	0x0001	

VIT_B Viterbi Instruction for Point B

Assembly Syntax vit_b rX.e, rY.e

Description

```
r1 = min {(rX + r(Y + 1)), (r(X + 1) + rY)}
if ((rX + r(Y + 1)) < (r(X + 1) + rY))
vitr = vitr << 1 | 0x0001
else
vitr = vitr << 1
```

Example vit_b r4, r6

Architectural state before the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved						rez	sat	res	q15	sre	mre
							x	x		x	x	x
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Register		
r1	xxxx	
{r5 r4}	8000	0xff30
{r7 r6}	0x000a	0xff00
vitr	0x0001	

Architectural state after the instruction is executed:

	15 11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		1	x	1	x	0	1	x	x	x	x	x

Register		
r1	0x7f00	
{r5 r4}	0xf000	0xff30
{r7 r6}	0x000a	0xff00
vitr	0x0002	

XOR Exclusive OR

Assembly Syntax xor rX, rY

Description rX ^= rY

Example xor r11, r2

Architectural state before the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
fmode	reserved							rez	sat	res	q15	sre	mre
								x	x		x	x	x

Register	
r11	0x70f2
r2	0x0901

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
			0	x	x	x	0	1	1	0	x	x	x

Register	
r11	0x79f3
r2	0x0901

XOR.E Exclusive OR (Extended Precision)

Assembly Syntax xor.e rX.e, rY.e

Description {r(X + 1)rX} ^= {r(Y + 1)rY}

Example xor.e r0, r2

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
fmode	reserved							rez	sat	res	q15	sre	mre	{r1 r0}	0x8000	0x8c23
								x	x		x	x	x	{r3 r2}	0x8f34	0x8300

hwflag	reserved	v	gv	sv	gsv	c	ge	gt	z	ir	ex	er
		x	x	x	x	x	x	x	x	x	x	x

Architectural state after the instruction is executed:

	15	11	10	9	8	7	6	5	4	3	2	1	0	Register		
hwflag	reserved		v	gv	sv	gsv	c	ge	gt	z	ir	ex	er	{r1 r0}	0x0f34	0x0f23
			0	x	x	x	0	1	1	0	x	x	x	{r3 r2}	0x8f34	0x8300

Customer Feedback

We would appreciate your feedback on this document. Please copy the following page, add your comments, and fax it to us at the number shown.

If appropriate, please also fax copies of any marked-up pages from this document.

Important: Please include your name, phone number, fax number, and company address so that we may contact you directly for clarification or additional information.

Thank you for your help in improving the quality of our documents.

Reader’s Comments

Fax your comments to:	LSI Logic Corporation Technical Publications M/S E-198 Fax: 408.433.4333

Please tell us how you rate this document: *ZSP400 Digital Signal Processor Architecture Technical Manual.* Place a check mark in the appropriate blank for each category.

	Excellent	**Good**	**Average**	**Fair**	**Poor**
Completeness of information	____	____	____	____	____
Clarity of information	____	____	____	____	____
Ease of finding information	____	____	____	____	____
Technical content	____	____	____	____	____
Usefulness of examples and illustrations	____	____	____	____	____
Overall manual	____	____	____	____	____

What could we do to improve this document?

If you found errors in this document, please specify the error and page number. If appropriate, please fax a marked-up copy of the page(s).

Please complete the information below so that we may contact you directly for clarification or additional information.

Name ____________________ Date __________
Telephone ________________ Fax ________________
Title ______________________________
Department ________________ Mail Stop __________
Company Name ______________________________
Street ______________________________
City, State, Zip ______________________________

You can find a current list of our U.S. distributors, international distributors, and sales offices and design resource centers on our web site at

http://www.lsilogic.com/contacts/na_salesoffices.html